UNIVERSITY OF SOUTHAMPTON

DEPARTMENT OF ELECTRONICS AND COMPUTER SCIENCE

High-level Floating-Point Synthesis

Zaher A. Baidas

July, 2000

A thesis submitted for the title of
Doctor of Philosophy.

UNIVERSITY OF SOUTHAMPTON

High Level Floating-Point Synthesis

by

Zaher Abdulkarim Baidas

A thesis submitted for the degree of
Doctor of Philosophy.

Department of Electronics and Computer Science,
University of Southampton

July, 2000

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE
DEPARTMENT OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

High Level Floating-Point Synthesis

by Zaher Abdulkarim Baidas

MOODS (Multiple Objective Optimisation in Data and control path Synthesis) is a high-level synthesis system which provides the ability to synthesise a system level behavioural description into a structural representation. The thesis represents an enhancement to the original MOODS system to allow the designer to manipulate floating-point and complex variables on an equal footing with all other data types; the additional complexities arising from floating-point manipulation are completely hidden from the user.

Originally, the data processed by MOODS was fixed (occasionally variable) width integers, and the functional units available were relatively unsophisticated (adders, subtractors, multipliers, multiplexers and so on). The floating-point synthesis system described here provides a library of high-level floating-point functions (trigonometric, transcendental, and complex) to support the synthesis of behavioural designs incorporating floating-point operations.

The floating-point library components themselves are implemented using a number of *base techniques*, namely table lookup, the CORDIC algorithm, and iterative series. Decisions about the mapping of base techniques onto functional units are left to a *floating-point optimiser*, which makes individual binding choices based on global knowledge of the overall design, allowing the internal sub-structures of these units to be shared which results in a dramatic decrease in the overall hardware resources required to implement the design.

Finally, an exemplar is designed and analysed in detail: a cubic equation solver synthesised using the floating-point capability integrated within the MOODS environment.

Contents

List of Figures

Acknowledgements

I would like to express my profound thanks to a number of people around me who helped make this project reality.

First I would like to thank my supervisor, Professor Andrew Brown. His constant and consistent guidance, advice, encouragement, and confidence were essential for completion of this thesis, and are highly appreciated

I would also like to thank Dr. Alan Williams for his invaluable help and great patience and diligence in answering my endless requests.

Thanks to all other members of the Electronics Systems Design Group at the University of Southampton, in particular I am grateful to Dr. Mark Zwolinski for his ideas and information and for giving me the chance to join the University as an MSc student at the first place.

Finally, I would like to say a big thanks to my family. They have given their unconditional support, knowing that doing so contributed greatly to my absence in my postgraduate studies, during which we could have been geographically closer.

Chapter 1

Introduction

A floating-point number representation can simultaneously provide a large range of values and a high degree of precision. However, their manipulation is considerably more complicated than the corresponding fixed point operations. As a result, a portion of modern microprocessors is often dedicated to hardware for floating-point computation.

In the past, silicon area constraints have limited the opportunity of synthesising floating-point arithmetic units. Advances in integrated circuit fabrication technology have resulted in both smaller feature size and increased die area, which has provided a larger transistor budget. It is now therefore possible to implement floating-point systems on an ASIC or even programmable logic devices. However, the complexity of floating-point units is still a major limitation in realising cost effective, low volume systems. To overcome this limitation, advances in current CAD tools are needed, to make it possible to sensibly implement floating-point systems.

Behavioural synthesis works on a description that specifies the relationship between system inputs and outputs by describing abstract data structures and functions to manipulate them. The physical structure is not described, as the emphasis is on what the design does and not how it does it. In addition, the data flow manipulation aspects for a synthesis system are not generally concerned with the data *type*; the limitations of integer arithmetic are imposed simply by the lack of functional units for more complicated data types.

The MOODS (Multiple Objective Optimisation in Data and control path Synthesis) [1, 2, 3, 4, 5] is a behavioural synthesis system which transforms a VHDL (Very High Speed IC Hardware Description Language) [6] description into a structural netlist. It implements global optimisation of a design data flow and control graph by the repeated application of small, reversible (behaviour preserving) transformations. The system is designed to

support overall optimisation with respect to widely differing objectives: currently these are total area and maximum delay. The manipulation of these objectives form the basis for exploration of the design space, which is defined as the n-dimensional space that contains all possible implementations of a specific design. The exploration is steered by a simulated annealing algorithm that allows the diverse penalty functions from the various optimisation criteria to be compared.

This thesis describes an enhancement to the basic MOODS synthesis system to support the processing of designs containing floating-point (and complex) arithmetic. In particular, the development of a floating-point module library and a floating-point optimiser capable of making strategic decisions about the high level binding of each floating-point operation in a way that meets the user's pre-defined goal.

The thesis is divided into seven chapters. Chapter 2 provides a general introduction to behavioural synthesis and describes the basic MOODS synthesis system together with more detailed examination of the core synthesis sub-tasks. This is followed in chapter 3 by a discussion of some related work and commercial systems.

The design and implementation of the floating-point library is described in chapter 4, along with several additional improvements to make the floating-point library integration more flexible.

Chapter 5 provides an in-depth look at the floating-point optimisation challenges and the way they were handled.

Chapter 6 highlights the development of a general purpose FPGA prototyping board and details the design and synthesis of an exemplar: a cubic equation solver, utilises the floating-point system discussed in the previous chapters.

Finally, chapter 7 concludes by suggesting a number of enhancements to the present system providing areas for further research.

A number of appendices are also included providing additional information on various aspects of the work. In particular, Appendix A outlines the main features of the IEEE 754 floating-point standard. Appendix B contains a detailed discussion of the CORDIC

algorithm. Appendix C provides further details of the floating-point library design and implementation. Appendix D gives implementation details of the software, and Appendix E gives details of the hardware used to support the demonstrator. Finally, Appendix F contains a pre-print of a paper submitted to IEEE-CAD.

Chapter 2

MOODS and behavioural synthesis

Digital designs can be distinguished by the level of abstraction required to describe them in three main domains [7, 8]: *Algorithmic* or *behavioural level* views the system as a set of variables and functions to manipulate them. *Register transfer level*, where the system is described as a set of registers and a set of transfer functions specifying the flow of data between these registers [9]. *Logic level* describes the system as a network of logic gates and flip-flops with logic equations specifying the behaviour.

Behavioural or high-level synthesis tools [7, 8, 10, 11, 12] bridge the gap between an abstract behavioural specification of a digital system and a register transfer level structure that realises the given behaviour. It provides an environment that allows the designer to experiment with a wide range of structural alternatives.

Starting with a behavioural description of a design and a set of user specified objectives, behavioural synthesis builds a datapath by allocating hardware elements (functional units, storage units and interconnects) and provides a controller to specify a set of operations to be performed during every control step. It frees the designer from the difficulties of selecting a good implementation, as it does not include design decisions such as timing and parallelism.

2.1 VHDL for behavioural synthesis

VHDL [6] is a language for describing digital systems. It arose from the program funded by the US Department of Defense in the late 1970s and early 1980s. In 1986, VHDL was proposed as an IEEE standard, and it was adopted as the IEEE 1076 standard in December 1987. The language is being used for documentation, verification and synthesis of large

digital designs. This is actually one of the key features of VHDL, since the same VHDL code can theoretically achieve all three of these goals.

The description of a digital system using VHDL is achieved with a set of *design units*. Each element in this hierarchy consists (usually) of a pair of design units: an *entity* and an *architecture*. The entity describes the IO ports of the element, and the architecture describes the internal structure and/or the functionality (thus it is possible for an entity to correspond to multiple architectures). This partitioning allows the design of an overall system to be distributed amongst a number of designers; once the entity definitions are established and agreed, the architecture designs can be carried out independently.

Within an architecture, VHDL allows three types of statement to describe the internals:

1. *Component instantiation* allows the use of any entity/architecture pair as a component in the design architecture. Each instantiation has two parts: the *name* and the *port map*. The component name defines the unit to be used, while the port map defines the way the signals in the design connect the component IO ports.

2. *Signal assignment* is used to describe the dataflow through the system. It is divided into two groups: 1) simple signal assignment (`x <= a xor b;`), which simply assigns to the target signal the value of the source expression, and 2) conditional signal assignment (`x <= a xor b when c = '1' else not (a xor b);`), which assigns to the target the value of the first expression when the condition is true or the second if the condition is false.

3. *Processes* provide a method to describe activities that must occur in a sequential order. A process has three main parts: 1) a sensitivity list, 2) declaration part, and 3) statement part. The sensitivity list defines the signals to which the process is sensitive. Any event occurring on one of these signals causes the process to execute once. If the sensitivity list is absent, the process will run forever, unless the user explicitly pauses the execution with a (`wait`) statement. The declaration part of the process allows the declaration of types, variables, functions, and procedures, which are local to the process. Finally, the statement part of the process contains a set of sequential statements executed every time the process is activated.

VHDL was initially designed as a simulation language. This leads to a number of problems when integrating VHDL in a synthesis environment and results in imposing some limitations on the language features. Moreover, the language synthesisable subset interpretation varies according to the level of abstraction at which the synthesis takes place. As far as *behavioural* synthesis is concerned, the set of restrictions applied to semantic interpretation of VHDL [13, 14] are summarised in the following:

- Processes do not execute in zero time, but take a number of clock cycles. Thus there is no implicit assumption about the execution time of a process. In the simulation model, the process executes in zero time unless the user explicitly defines a delay using a `wait` statement.

- Time expressions (`wait for X sec`) are converted into control steps. Therefore, delay specifications (pausing process execution) can only be implemented as multiples of the clock period. A delay of any period can be specified using the same wait statement in the simulation model.

- Processes cannot be used to specify combinational logic. In contrast, a process can be used to *model* combinational logic in a simulation and/or RTL environment.

- Structural definitions such as component instantiation and *generate* statements are not allowed.

- Recursion within procedures is excluded, due to the difficulties created.

- *Assert* statements are for verification during simulation. They are ignored during synthesis.

- Statements within a process are executed in a sequential manner governed by an implicit clock signal.

- Sensitivity lists, such as (`wait on input`) will not activate on asynchronous edges.

In the VHDL simulation model, a delay occurs within a process when a *wait* statement appears. Sequential blocks between wait statements execute in zero delay. However, when the design is synthesised, these blocks may take a number of clock cycles to execute, dictated by the data dependency between operations and the synthesis objectives. It is

common in a simulation model to employ such processes to describe combinational logic blocks that get activated when a transition occurs on any of the inputs and executes in zero delay (or a delay specified by a wait statement). When synthesised, these blocks will not be mapped to a combinational unit, instead, the system will generate a multi-cycled sequential block with a number of internal registers.

Sensitivity lists are another issue that introduce major differences between a simulation and a synthesis environment. *Wait on* and *wait until* statements originally detect asynchronous edges of the monitored signals. However, in a behavioural synthesis environment, signal edges will be synchronised to the system clock, and transitions will only be effective at clock edges, which might introduce timing mismatches between the behavioural model and the synthesised structural model.

2.2 Behavioural synthesis

There are several advantages to high-level synthesis over conventional RTL synthesis systems [9, 10, 15]. First, moving automation to a higher level assures a much shorter design cycle[1]. Second, it allows comparing several designs in a reasonable amount of time. Finally, an automated process may out-perform a human engineer in meeting most design objectives.

The main tasks involved in a behavioural synthesis process are illustrated in Figure 2.1, which shows the flow of data in a generic high-level synthesis system. A behavioural description forms a starting point for a high-level synthesis system. The behavioural description is then compiled into an *internal representation*. This stage may include a compiler-like optimisation phase [16, 17] such as loop unrolling, common sub-expression elimination, dead code elimination and inline expansion of procedures.

[1] The increase in productivity of behavioural design versus RTL design is typically quoted as a factor of five [18].

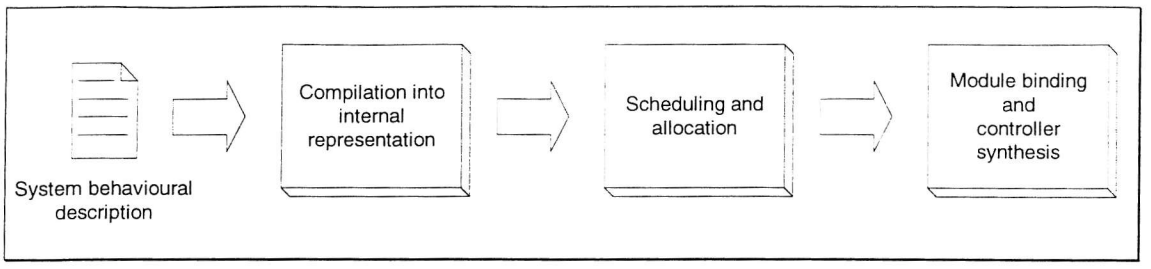


Figure 2.1 A generic high-level synthesis system

The next two steps form the basis of translating behaviour into structure: *scheduling* and *allocation*. Scheduling assigns operations to control steps (a control step is usually a single clock cycle). Allocation involves assigning operations and variables to functional units, storage hardware and communication paths.

The final step in this process consists of module *binding* and *controller synthesis*. In module binding, the abstract datapath units are mapped to specific hardware implementation provided by a technology dependent module library, while controller synthesis provides the control circuitry responsible for generating the datapath control signals.

2.3 The design space

High level synthesis allows the designer to investigate a range of implementations for a particular input description, representing different trade-offs between a set of pre-defined objectives. Each of these implementations forms a single point in what is called the *design space* [4, 7, 19, 20], which is the n-dimensional space describing all possible implementations of a single behavioural description, in terms of n design aspects. Figure 2.2 shows a two-dimensional design space represented by area and delay (processing time). The design space is divided into two regions, containing designs that are either *achievable* or *unachievable*. The two regions are separated by the *optimal design curve*, which consists of a set of discrete points representing the most efficient implementations. For a particular design, only a portion of the achievable region may be obtained as indicated by the *actual achievable region* in Figure 2.2. This limitation in the design space is due to a number of factors such as optimisation algorithms and design space modelling methods [21].

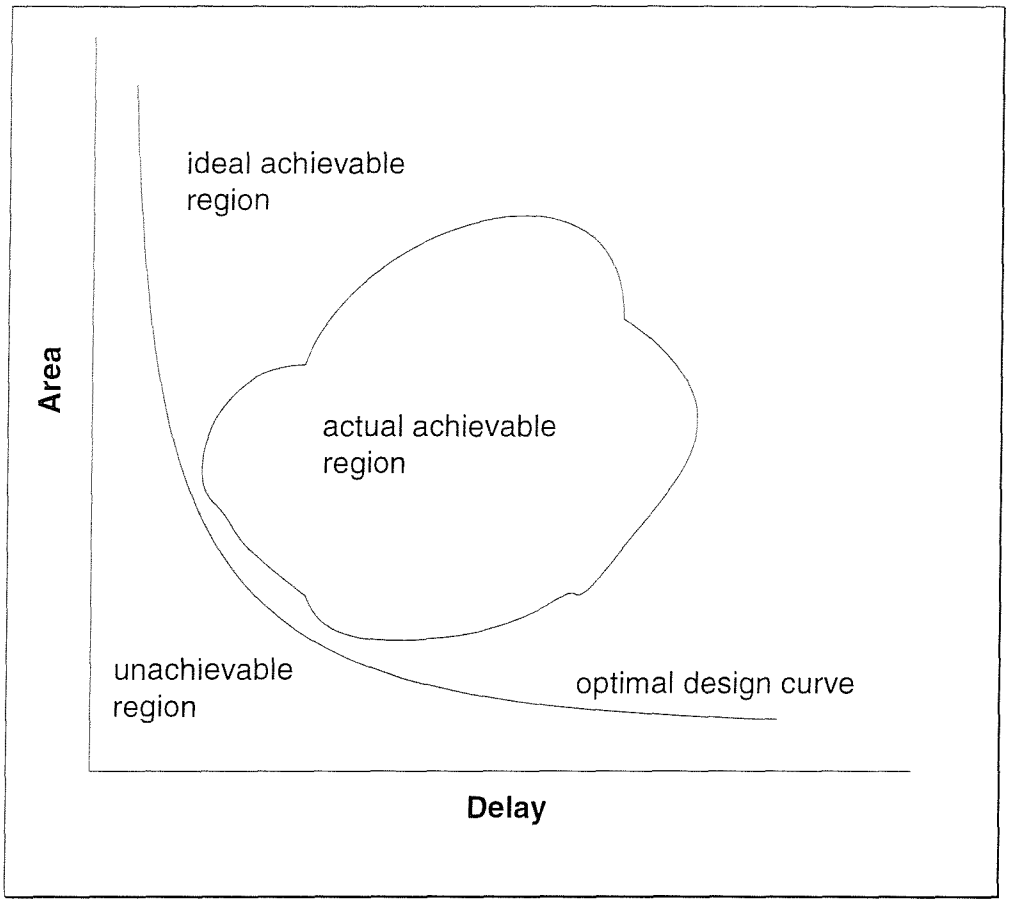


Figure 2.2 Area versus delay design space

2.4 Internal representation

The first step in high-level synthesis is to capture the behaviour of the design in the form of an internal representation. This is essentially a one-to-one translation of the behavioural description into a graph-based representation containing both the data flow and the control flow of the design.

For simple designs, the *data flow graph* (DFG) [11] can be employed to describe the system. The representation consists of a set of nodes, each node representing an operation in the original behavioural description. Data dependency between two nodes is represented by an arc connecting them. Figure 2.3 shows a sample VHDL input with the associated data flow graph. Three nodes are generated representing one addition and two subtraction operations. Node 3's dependency on nodes 1 and 2 is simply indicated by two arcs, the first arc labelled C indicates node 3 dependency on node 1 through the internal variable C, and the arc labelled D represents node 3's dependency on node 2.

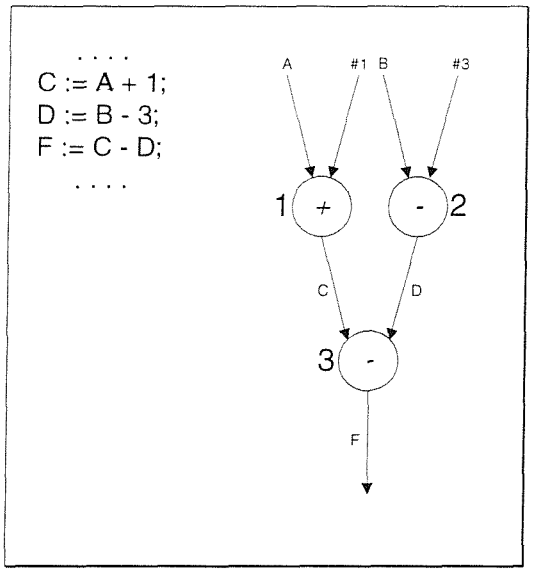


Figure 2.3 Data flow graph representation

The DFG is not sufficient for representing systems in which the execution sequence is based on external conditions (`if-else` and `case` blocks). The reason is that DFG is based on data dependency, while a method of representing the control flow as well as the data dependency is absolutely essential in such systems.

To represent the control and the data flow, some systems choose to combine the control and datapath graphs into one structure, such as the *Control Dataflow Graph* (CDFG) [22]. Other systems maintain separate graphs for data flow and control, with binding indicating the relationship between elements in both graphs. An example of the latter is the *Extended Timed Petri-Net* (ETPN) [14] representation.

To illustrate these representations, a simple example is introduced in Figure 2.4 showing a fragment of VHDL code. The graph representation of the code using CDFG and ETPN is shown in Figure 2.5 and Figure 2.6 respectively.

```
    . . . .
c :=  (a + b) / 2;
sel := sign(c) xor sign(b);
IF sel = '1' then
     r := (a + c) / 2;
ELSE
     r := (b + c) / 2;
END IF;
        . . . .
```

Figure 2.4 A sample VHDL example

The CDFG describes the control flow of the system as a directed graph. Each node in this graph is actually a separate DFG representing a block of assignments or a conditional statement. The CDFG in Figure 2.5 comprises three DFGs. The first one represents the two sequential assignments, the second two graphs representing the two conditional assignments.

ETPN represents the datapath as a directed graph [14] with nodes and conditional arcs. The nodes capture both the operators and the variables, while the arcs represent the connections between nodes. These connections are only available if the arc associated control signal (S_n) is activated. The control part of the design is described by the passage of *tokens* through a Petri-net, with vertices representing control states. The state transaction is controlled by conditions (C_i) generated in the datapath. When a control state receives a token, it activates the associated datapath conditional arc through its (S_n) signal

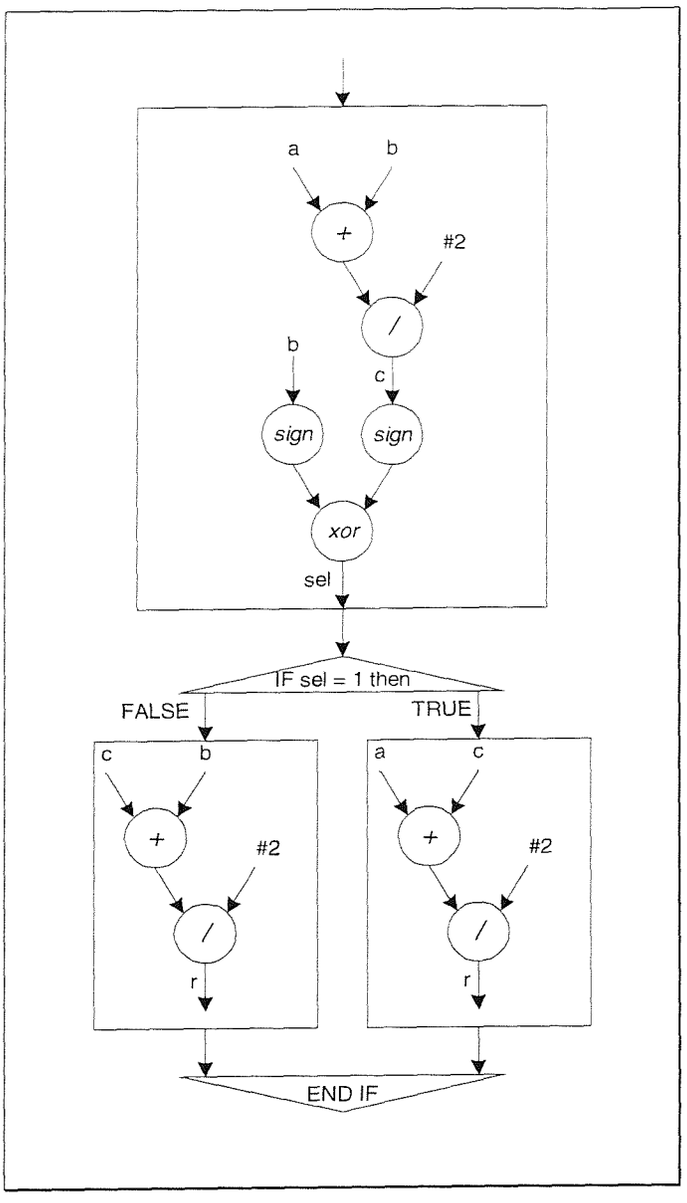


Figure 2.5 Control dataflow graph

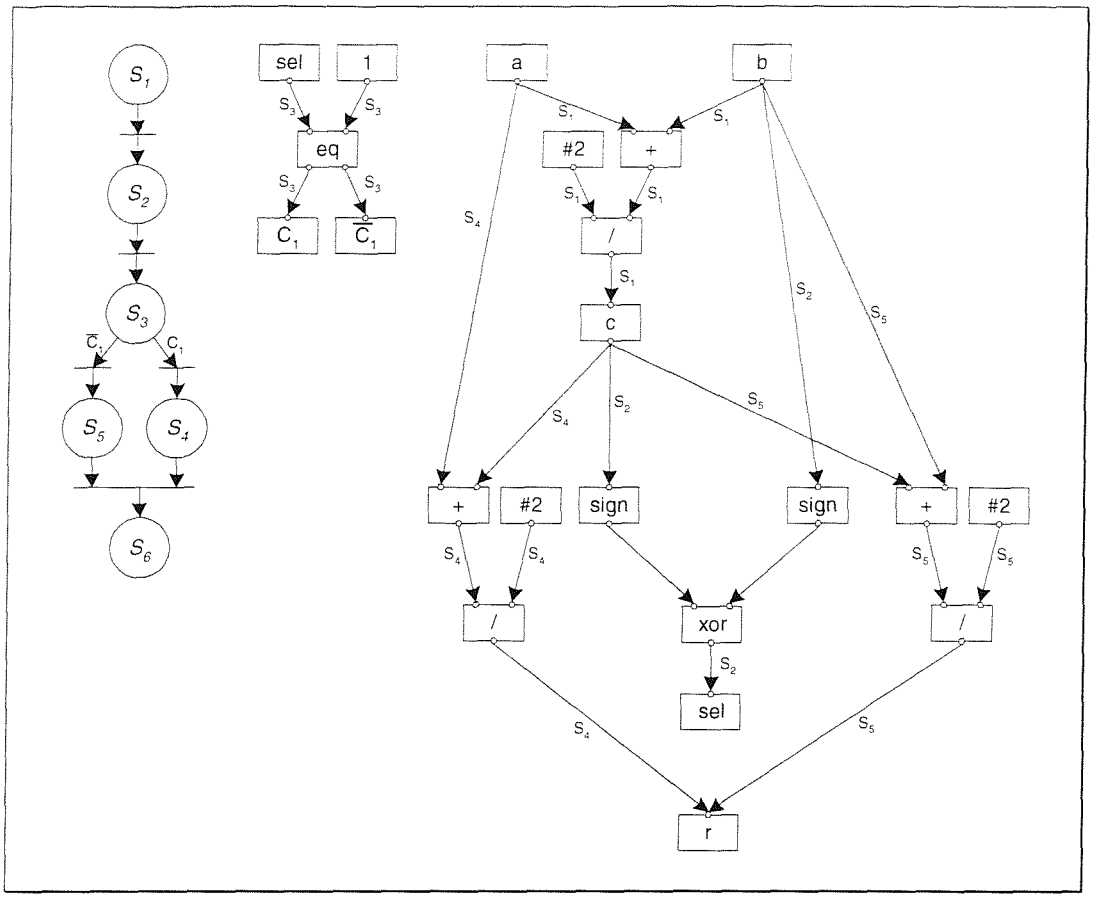


Figure 2.6 Extended timed Petri-net

2.5 Scheduling and allocation

Scheduling and *allocation* form the basis of transferring behaviour into structure [10, 15]. These two tasks are closely interconnected and dependent on each other. For example, high performance (speed optimised) designs require allocating more components in each control step, to allow the exploitation of parallel execution of operations. On the other hand, the most area-efficient designs use a minimum number of slow components, which results in a large number of control steps. This dependency gives rise to a major problem: any decision taken by one of the two tasks might reduce the number of possible implementations, hence, reduce the actual achievable region in the design space.

The simplest approach to this problem is to set some resource limit before scheduling; this is usually achieved by imposing a limit on the number of functional units available to implement the design (e.g. one multiplier and two adders). An improved version of this approach allows the process to iterate by re-synthesising with a modified resource limit. In a similar way, the resource limit is imposed, and then scheduling is performed. The result

is then evaluated against the user objectives. According to the evaluation result, the resource limit may be altered and the scheduling is performed again for a possible improved implementation.

Another approach to this problem is to perform allocation before scheduling, trying to produce an area minimised design within the timing constraints given. For example, some systems [23] perform complete datapath synthesis including hardware component mapping. Both global and local optimisations are employed at this stage to minimise the area cost. Once the datapath is implemented, controller synthesis is then performed, optimising the number of states according to the constrained imposed by allocation and the timing constraints given.

The approach employed by the MOODS synthesis system, is to combine scheduling and allocation together as a *general* optimisation problem and introduces an optimisation technique to minimise it.

The techniques that perform scheduling can be classified into two types [24]: *constructive* and *transformational*. Constructive scheduling creates a schedule from scratch by adding operations one at a time until all operations are scheduled. Transformational scheduling, on the other hand, starts with an initial schedule, generally maximally serial or maximally parallel, and attempts to improve it by applying a number of local transformations.

Simple constructive scheduling is possible by scheduling operations 'as soon as possible' (ASAP) or 'as late as possible' (ALAP) [25]. ASAP schedules operations in the earliest time step allowed by data dependency, while ALAP assigns operations to the latest possible time step. Figure 2.7 illustrates the meaning of ASAP and ALAP. The main disadvantage of both techniques is that all operations are treated equally, with no priority given to the more critical ones. When resource constraints are imposed, operations that are less critical can be scheduled first on a limited resource (e.g. single multiplier). This might block critical operations scheduling and result in an overall performance degradation.

List scheduling [25] solves this problem by taking more controlled approach in selecting the operation to be scheduled. At each control step, operations available to be scheduled are kept in a list ordered by some *priority function*; each operation in the list is then scheduled in turn as long as the required resource is available, other wise, it will postponed

to the next scheduling step. Figure 2.8 represents list scheduling of a simple control graph: operation 2 has a higher priority that operation 1, and is therefore scheduled before it, providing an optimal solution in this case.

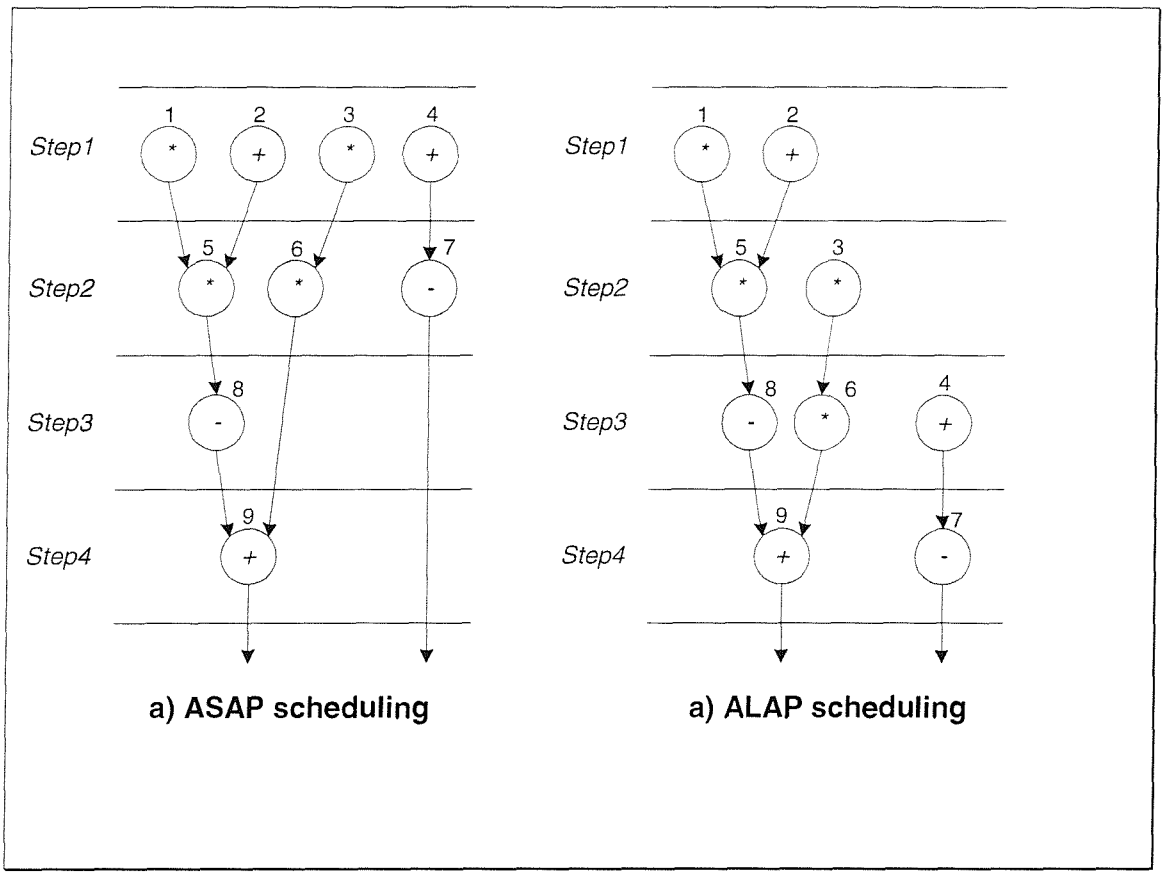


Figure 2.7 ASAP and ALAP scheduling

In contrast with the above algorithms, the *force directed scheduling* [26] attempts to create an optimal schedule based on a more global view. The algorithm attempt to minimise the number of resources required to implement the design within a given time constraint, by distributing sharable operations as evenly as possible between the control steps.

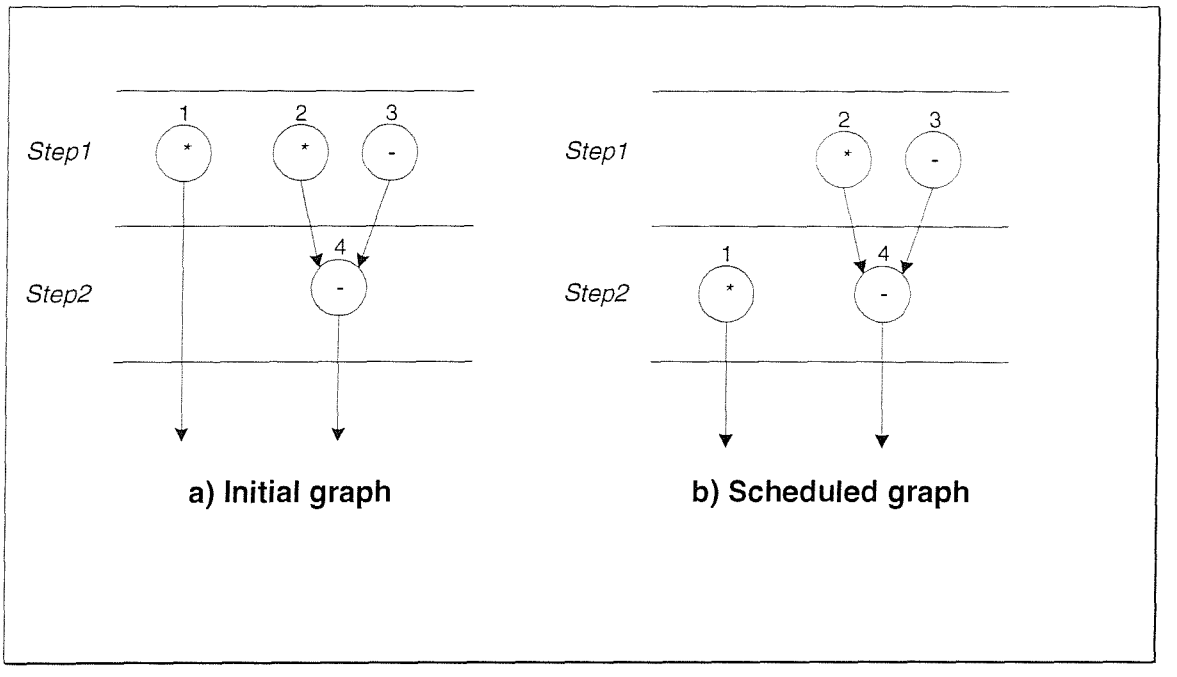


Figure 2.8 List scheduling

In contrast to constructive scheduling, transformational scheduling is based on an iterative process that applies a set of local transformations to the design initial schedule, moving the design towards the point that meets the user pre-defined objectives in the design space. Early transformational scheduling schemes employed exhaustive search to perform scheduling. The approach tests all possible combinations of transformations and chooses the best result. The method guarantees reaching an optimal solution, since all possible designs are tested. However, it is very expensive in terms of computing time and may not be considered as a viable solution for large designs.

Another approach to scheduling by transformations is to handle scheduling as an optimisation problem, and employ an optimisation algorithm that exploits different transformations to achieve the desired result [7]. At this stage, a heuristic approach may be employed to minimise the problem by selecting and applying transformations according to a pre-defined regime guided by an analysis of the design.

In a similar manner to scheduling, resource allocation can be achieved using different approaches. Allocation involves binding operators to functional units, binding variables to storage units, and providing interconnect between registers and functional units. Algorithms that implement allocation can be divided into two classes [10]: *iterative/constructive* and *global*.

The iterative/constructive algorithms perform allocation by iteratively assigning operations, one at a time. These algorithms are distinguished by the method employed to select both the element to be assigned and the unit to which it will be assigned. The selection methods can be simply implemented to select elements in a fixed order; usually the same order appears in the data flow graph. A more sophisticated approach relies on a global selection, which tries to make the most suitable selection based on some metric: for example, selecting an element that has the least effect on the total system area cost.

Global allocation techniques, on the other hand, deal with the datapath as a whole, and try to allocate all its elements at once. A number of techniques may be used for global allocation. A possible technique is to use a graph-based clique-partitioning algorithm [27], which attempts to build up a graph representing datapath elements by nodes, with arcs joining nodes that can share the same hardware. The problem is then reduced to finding a maximal partitioning of fully interconnected nodes. Since each partition will represent elements that can share the same hardware without conflict, the solution will represent the minimum hardware cost.

Alternatively, branch-and-bound techniques [28] can be employed to perform global allocation. The algorithm performs an exhaustive search by trying all possible allocations of the datapath elements. The approach is very powerful since it checks every possible solution and provides an efficient allocation for small designs. However, the exponential increase in processing time makes it very expensive as the number of elements to be allocated grows. The latter problem can be tackled by imposing bounding heuristics to limit the number of solutions tried, for example, aborting any search that results in a cost increase higher than a certain limit.

2.6 MOODS synthesis system

The vehicle used to carry out this synthesis research is called MOODS [1, 2, 3] (Multiple Objective Optimisation of Data and control path Synthesis). The MOODS synthesis system has been developed to compile a behavioural description of a digital circuit into a structural description (VHDL or Verilog structural netlist), which utilises third-party tools to implement the design. Figure 2.9 is the original MOODS system data flow showing the major building blocks. It consists of four different tasks:

1. The VHDL behavioural description passes the source level optimiser [16, 17]. This performs a source level optimisation on the VHDL source code, to reduce the area/delay cost of the final hardware. Compiler-like transformations are applied at this stage, such as algebraic simplification, dead code removal and inline expansion of procedures.

2. The optimiser output is then compiler to an *intermediate code* (ICODE) using a VHDL language compiler. The ICODE represents the behaviour of the design at the register-transfer level.

3. This stage is the actual synthesis process. It takes as input the ICODE file and a set of user objectives, such as the design total area and maximum delay, and performs scheduling, allocation and module binding and outputs a VHDL structural netlist suitable for the target logic synthesis tool.

4. The final stage in this data flow is the low-level logic synthesis and technology mapping, which utilises third-party tools, such as Cadence Synergy [29], LeonardoSpectrum [30], and Xilinx Foundation [31], to transfer the structural netlist into a physical circuit on an ASIC or a programmable logic device.

A detailed description of the MOODS synthesis system may be found in the literature [1, 2, 4, 5, 19, 32]. Outlined in the following sections are three major aspects of the synthesis system which have a particular bearing on the discussion of the floating point subsystem:

- The initial compilation into ICODE and the internal representation.
- Module expansion.
- Global optimisation.

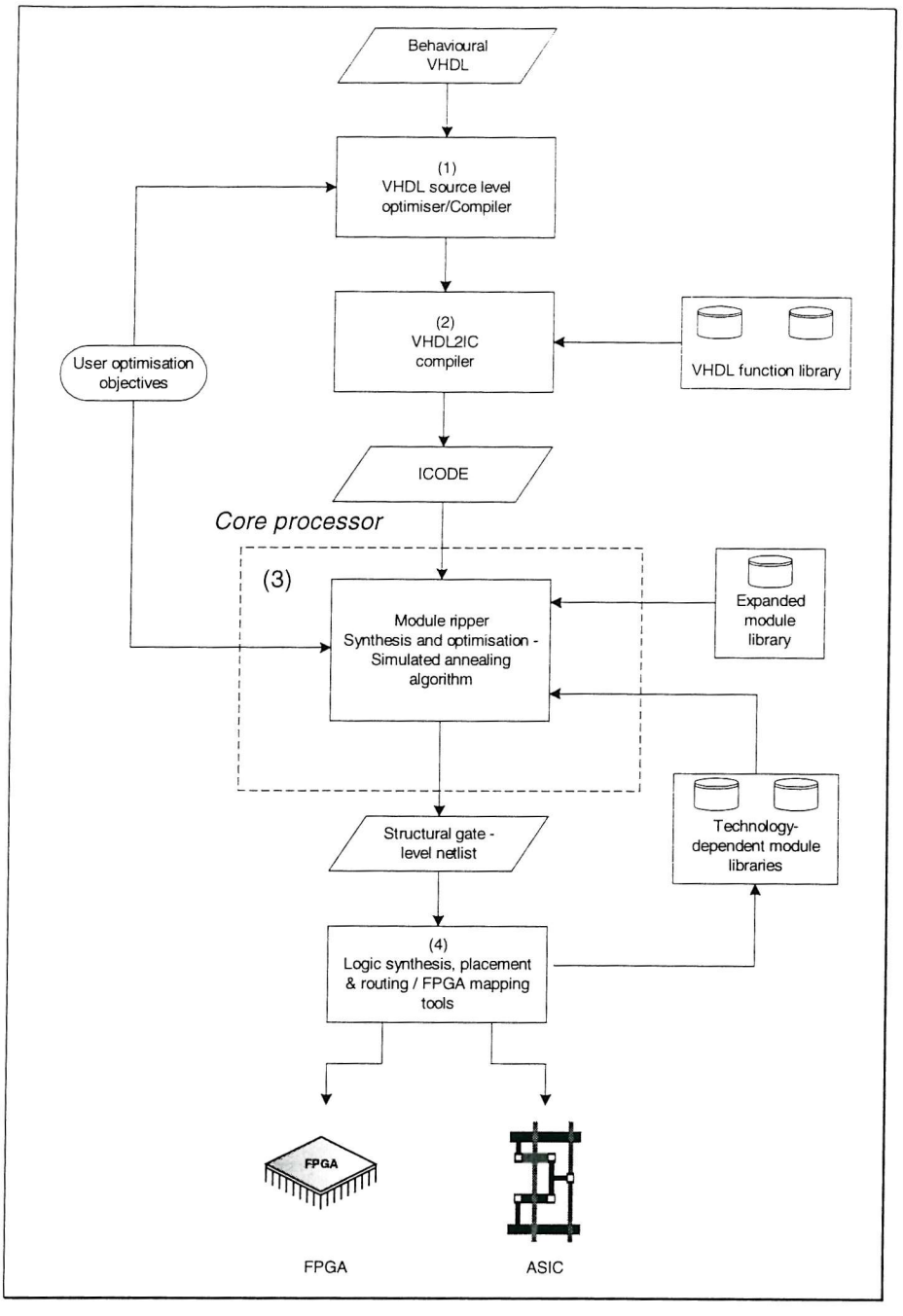


Figure 2.9 Original MOODS system data flow

2.6.1 ICODE and internal representation

The MOODS synthesis system does not directly read the input behavioural description. It reads an ICODE file. The logic behind this is to have MOODS as a general purpose synthesis system that can handle different input languages simply by changing the ICODE compiler at the front end. The VHDL2IC compiler (Figure 2.9(2)) translates the VHDL description into an ICODE representation. The ICODE is in some way similar to an assembly language, with additional control flow information. A simple example showing a

fragment of VHDL code with its equivalent ICODE is shown in Figure 2.10. It outlines the key features of the ICODE language:

- An ICODE instruction has the general form:
  `OPERATION <inputs> , <outputs>  <activation list>`

- Each ICODE instruction is executed once it has been *activated*. (Excluding the first instruction, which is activated on the system reset.) Upon conclusion of an instruction, all instructions in its *activation list* are activated. If the activation list is missing, the next instruction is activated by default. For example, instruction *i2* activates both *i3* and *i4*. While the absence of an activation list in *i6* results in an automatic activation of *i7*.

- Complex expressions are split down into a number of simple ICODE instructions, with temporary variables identified in the figure as numeric literals. In the figure, the VHDL assignment to the variable *m* is represented by five ICODE instructions (*i2* to *i6*).

- VHDL functions and procedures are implemented as a separate *module*, with a dedicated instruction MODULEAP to transfer the control to them. Instruction *i9*, for example, halts the main execution and passes the control to the *sqrt* module. The module output is returned in the variable *m* before the main execution continues.

- Conditional branches are implemented as an IF instruction with two activation lists. One for the true condition (ACTT) and the other for the false (ACTF). In Figure 2.10, the VHDL conditional statement (IF sel = 1 THEN ... ELSE ... END IF) is implemented as two instructions *i10* and *i11*, with instruction *i12* being activated if the condition is true, and *i14* being activated if the condition is false.

A complete definition of the ICODE is provided in Appendix D.

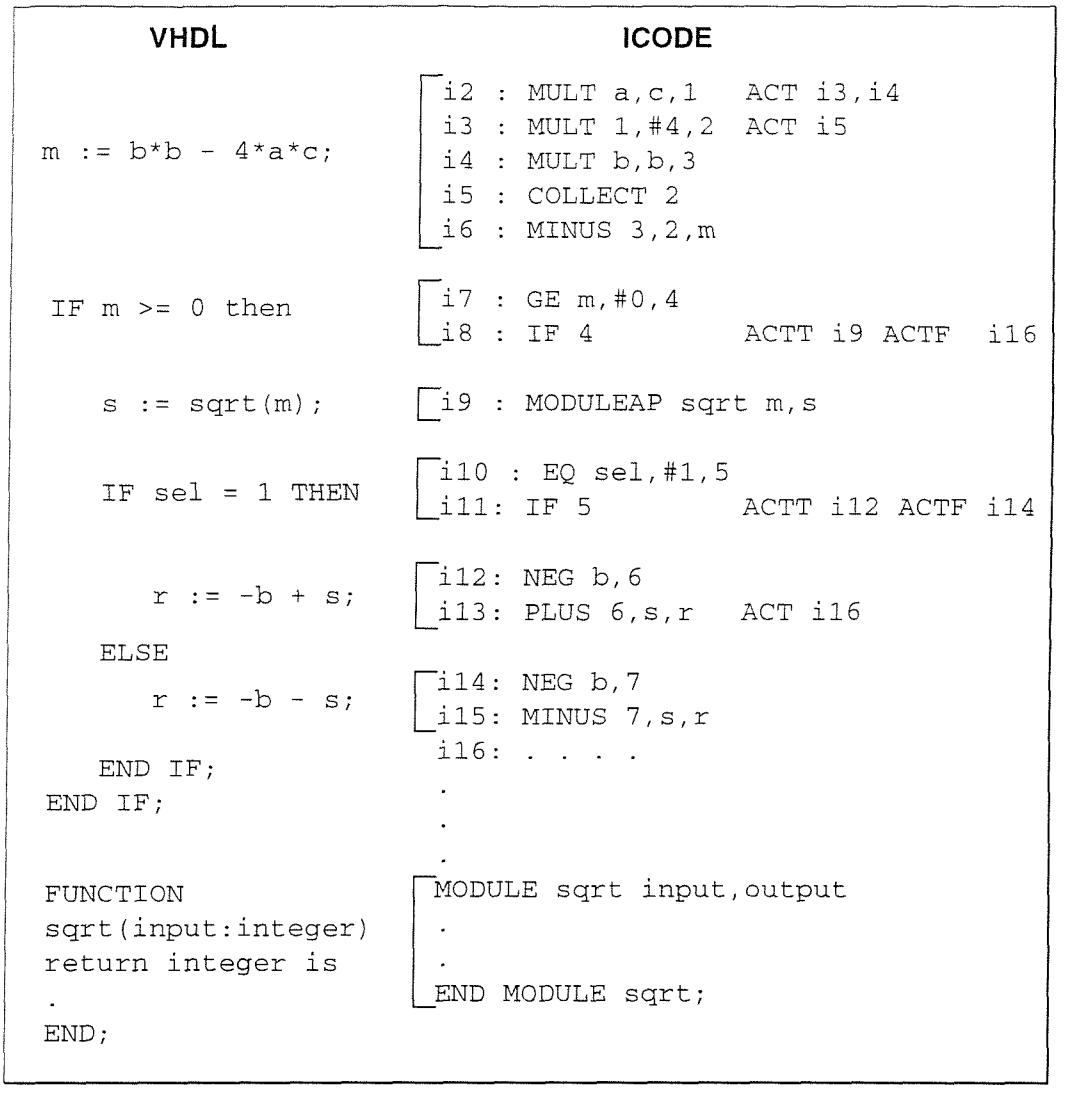


Figure 2.10 VHDL and the equivalent ICODE example

In the core processor input stage, the design, in the form of an ICODE file, is transformed into a control and datapath graph [1, 19]. Figure 2.11 shows the initial control and datapath graphs for the ICODE listed in Figure 2.10.

The control graph defines the execution order of the ICODE instructions. Each node in the graph defines a control state. Input and output arcs define a conditional control flow, governed by signals generated on the datapath. For example, the datapath signal *s41* decides on the transition from state S_8 to state S_9 or S_{16}. Each control node has an instruction list, defining the instruction to be executed when this node is activated. A set of acyclic subgraphs divide these instructions into groups of *dependent instructions*. Each group has a unique *group number*. Instructions with different group numbers may be executed concurrently. Instructions with the same group number are dependent on each other and must be executed sequentially within the same control state.

The MOODS control graph is of six types of node (refer to Figure 2.11):

1. *General node* (for example S_6): has one input and one output, and can contain any ICODE instructions except COLLECT, MODULEAP, or conditionals.

2. *Fork node* (for example S_2): the same as general node except that it has two or more unconditional outputs. This node defines the starting point of a set of parallel execution threads, where all the successors executed independently.

3. *Conditional node* (for example S_8): has one input and two or more outputs. The output conditions are controlled by a signal from the datapath. This node is generated form an ICODE conditional instruction such as an IF or CASE statement.

4. *Dot node* (for example S_{16}): has two or more inputs, any of which can activate the node. This node is a counterpart to the conditional node; it represents the reconvergence of mutually exclusive control threads.

5. *Call node* (for example S_9): the call node results from a module call instruction. When this node is activated, it activates the execution of the required sub module. When the sub module exits, control is returned to the submodule successor.

6. *Collect node* (for example S_5): results from an ICODE collect instruction. The node will not activate its descendant node until a fixed number of activations (indicated by its argument) is received, thereby synchronising a set of parallel execution threads. The node is a complement to the fork node, where the concurrent branches are joined into a single node.

The MOODS datapath graph represents the functionality of the ICODE instructions with a set of functional units, storage units, and interconnects. The flow of data though this graph is governed by control signals generated by the appropriate control state in the control graph.

The initial datapath graph is created as a one-to-one mapping of ICODE operations and variables, with each ICODE variable represented as a storage unit (register), each ICODE arithmetic or logical operation represented as a separate functional unit, and each assignment operation represented as a set of registers, interconnects and control signals.

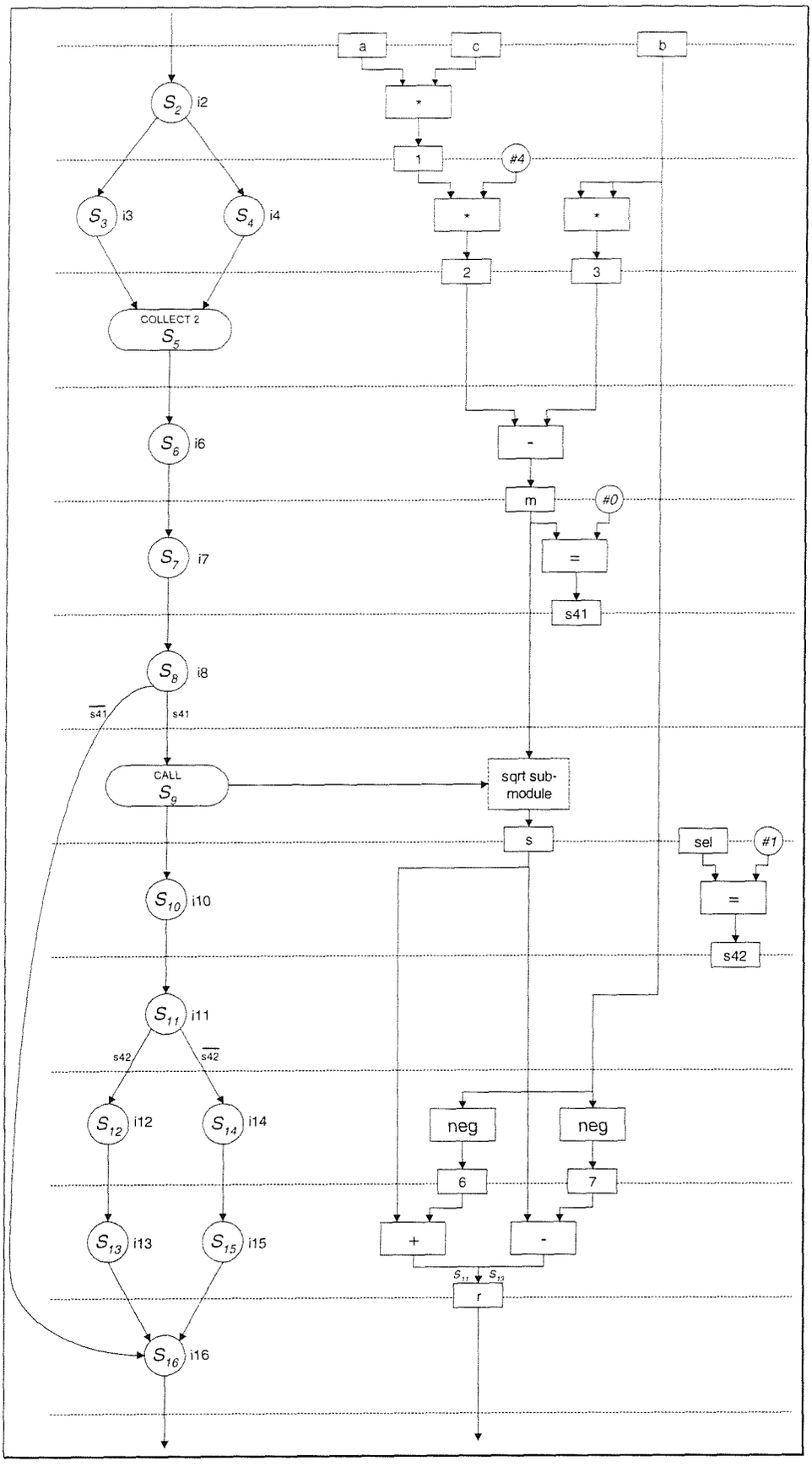


Figure 2.11 Control and datapath graphs

It is worth mentioning that the initial control and datapath graph represents a valid structural implementation of the design. However, it is almost certainly a highly inefficient implementation in terms of the total execution time and the large area cost. The optimisation phase of MOODS now moves this implementation in the design space towards the point that meets (if possible) the cost objective specified by the user

2.6.2 Transformations

MOODS employs an iterative optimisation strategy to perform synthesis. Iterative optimisation is achieved by dividing the synthesis task into a number of local transformations that are applied to different parts of the design using a dedicated optimisation algorithm. This allows simultaneous consideration of synthesis sub-tasks by performing scheduling, allocation and module binding simultaneously.

At present, MOODS has a set of fourteen different transformations. These transformations are *complete*, as a transformation applied to a valid design will result is a valid design. The availability of inverse transformations allows a previous design decision to be reversed at any stage during optimisation, which provides a solution for the problem encountered with premature binding decisions which may result in a design that is not optimal.

Transformation selection and application consists of four distinct steps, as illustrated in Figure 2.12:

1. *Data selection* involves selecting a transformation and the portion of the design to which it should be applied. The selection varies according to the optimisation algorithm involved and is performed randomly in the *simulated annealing* algorithm (see section 2.6.4).

2. *Testing* involves checking the validity of the transformation and ensuring that it will not modify the design behaviour.

3. *Estimation* predicts the effect of the transformation on the system performance without actually altering the design.

4. *Execution*, applies the transformation to the design.

MOODS transformations are divided into two groups: *scheduling transformations* which apply mainly to the control graph, and *allocation and binding transformations* which modify the design datapath. Scheduling transformations are listed in Table 2.1, while allocation and binding transformations are listed in Table 2.2.

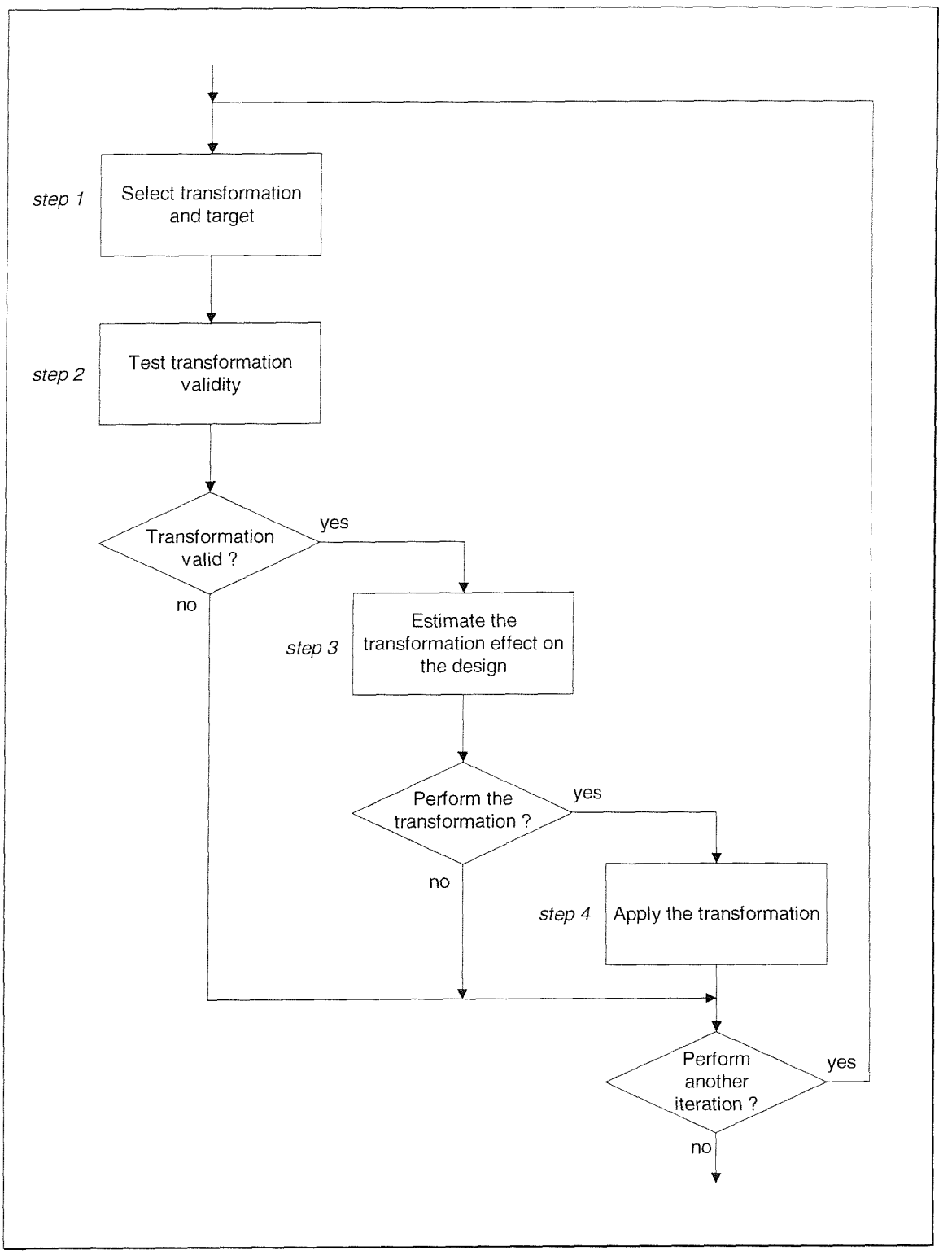


Figure 2.12 Transformation application steps

Transformation name	Effect
sequential merge	Combines two sequential control nodes (i.e. nodes executed sequentially) to form a single control node implementing multiple instructions.
parallel merge	Combines several concurrently executing nodes into one control node.
merge fork and successor	Combines a fork node with one of its successors, with the successor instructions becoming conditional instructions executed in the fork node control state.
group instructions on register	Tries to bypass datapath registers that have a single input and a single output net (i.e. a register implementing a variable accessed by one read and one write instruction) and moves the instruction group that contains the write instruction into the read instruction control node.
ungroup node into groups	Moves an instruction group into its own separate control node.
ungroup node into time slices	Divides instructions within a control node into new control nodes, such that no control state has an execution time greater than a specified period.
clock set / multi-cycling	A global optimisation transformation that employs *ungroup node into time slices* transformation to meet a clock period constraint set by the user.

Table 2.1 Scheduling transformations

Transformation name	Effect
combine functional units	Responsible for joining two functional units into one, time-shared between several operations. For example, combining an add and a subtract unit into a single add/subtract ALU.
share registers	Shares a single register between ICODE variables with non-overlapping lifetimes, or variables that occurs in mutually exclusive conditional branches (i.e. do not execute concurrently).
uncombine instructions from units	Takes a functional unit implementing a number of ICODE instructions and moves one of those instructions into a new functional unit added to the datapath.
uncombine units fully	Utilises the *uncombine instructions from units* transformation to completely remove a combined functional unit from the datapath.
unshare variable from register	Removes one of a set of shared register variables into a new register.
unshare register fully	Utilises the *unshare variable from register* transformation to completely unshare a register into separate registers, one for each variable.
alternative implementation	The only binding transformation. It provides an alternative low level module to implement a certain datapath functional unit. For example, replacing a ripple carry adder with a carry lookahead adder to enhance the speed of vice versa to reduce the total area cost.

Table 2.2 Allocating and binding transformations

2.6.3 The cost function

MOODS employs fourteen different transformations to manipulate the design data structure (see Table 2.1 and Table 2.2), by chaining, merging or separating nodes in the control and datapath graphs. A measure of the efficiency of applying these transformations is provided by means of a "cost function" that represents the state of the design in an n-dimensional design space as a single number, essentially the weighted sum of the costs in each dimension.

The MOODS cost function allows the user to specify objectives for a number of design parameters such as area and delay. These are the dimensions of design space. Each of these objectives is defined as a target value and a priority level, with one being the highest priority.

During optimisation, the effect of a transformation is predicted by evaluating its effect on the system "energy". For a single objective, the change in energy is determined by:

$$\Delta E = \frac{C_{estimate} - C_{previous}}{C_{initial}}$$

Where $C_{estimate}$ is the estimated cost after applying the transformation, $C_{previous}$ is the current implementation cost, and $C_{initial}$ is the cost of the initial implementation, with negative average energy change ($\Delta E < 0$) indicating a general improvement in terms of the target objective.

2.6.4 Simulated annealing optimisation

Design optimisation is performed using a simulated annealing algorithm [33, 34, 35, 36] to minimise the multiple-input cost function by selecting and applying different transformations. The term simulated annealing comes from a physical perspective: annealing is originally a physical process where a substance is cooled down from the liquid phase to the solid phase in a controlled, usually slow, manner. If the cooling is done carefully enough, the energy state of the solid at the end of the cooling is at its minimum.

Simulated annealing algorithm is a global optimisation method that distinguishes between different local optima. Starting from an initial point, the algorithm performs a random transformation and the cost function is evaluated: any downhill step is accepted and the process repeats from the new point. An uphill step *may* be accepted, enabling the process to escape from local minima. This uphill decision is made by the Metropolis [37] algorithm. As the optimisation process proceeds, the length of the step declines and the algorithm iterates towards a global optimum.

By way of an example, let us consider the one-dimensional configuration space represented in Figure 2.13. The design is initially represented by point *A*. An optimisation algorithm accepting only transformations that results in an improvement will hit the local minima (point *B*). Simulated annealing will accept degradation and hence allows the configuration to jump out of the local minima into the global minima (point *C*).

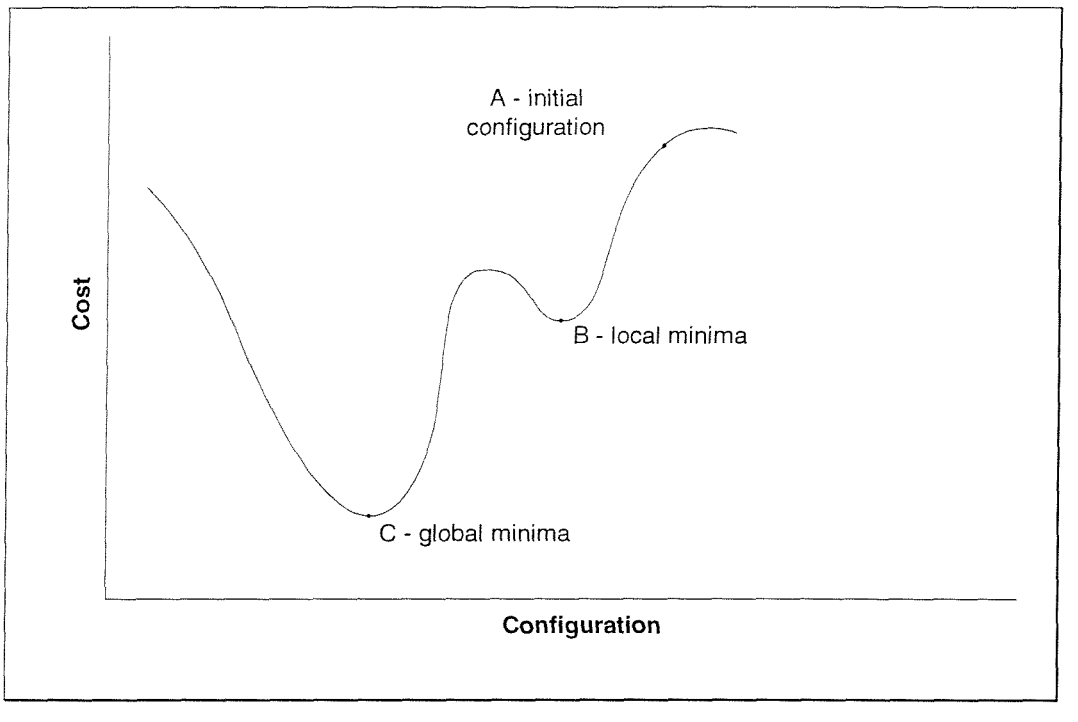


Figure 2.13 A one-dimensional configuration space

In MOODS the method selects a random transformation and evaluates the change in the cost function ΔE. If the transformation leads to an improvement ($\Delta E < 0$), it will be automatically accepted. Degradation might be accepted with a probability given by

$$P = \exp\left(\frac{-\Delta E}{T}\right) \quad ; \quad \Delta E > 0$$

Figure 2.14 describes the procedure as implemented in MOODS

```
For (temp = Tstart; temp >= Tend; temp = temp * Tstep)
{
      for (I = 0; I < Istep; I++)
      {
            t = select_transformation ();
            delt_E = estimate_cost_var (t);
            if (delt_E < 0 || rand() < exp(-delt_E/temp))
                  Execute_transformation(t);
      }
}
```

Figure 2.14 The simulated annealing algorithm

The sequence of temperatures during optimisation, and the number of transformations examined per temperature, defines the *annealing schedule*. The annealing schedule in MOODS is determined by four parameters:

1. The initial temperature T_{start}.

2. The final temperature T_{end}.

3. The number of iterations per temperature I_{step}.

4. The reduction made to the temperature in the end of each step T_{step}.

T_{start} is difficult to determine. However it should be high enough to allow the design to escape local minimas. For the end temperature T_{end}, a safe option is to always set it to zero, since at zero temperature, only improving transformations will be applied to the design.

The optimisation algorithm performance at $T_{end} = 0$, is a good aid to decide the I_{step}. If a noticeable amount of improvement is achieved at this point, then the design is not optimal and the number of iterations should be increased. On the other hand, if few improvements in the design are achieved, then this is a good indication that sufficient iterations have been performed during the optimisation phase.

Finally, temperature reduction should be small enough so that the reduction in temperature is slow enough to avoid trapping the design in a local minima because the temperature is low and hence the probability of accepting degrading transformations, that cause the design to escape this minima, is too low.

In addition to the simulation annealing algorithm, tailored heuristic optimisation is also provided to perform design optimisation. It is based on the same set of transformations, however, transformations are applied in a pre-defined order based on an analysis of the performance of each transformation on a number of designs [19].

MOODS heuristic approach only accepts improving transformation, thus there is a possibility that the algorithm delivers a local minimum. However, tests suggest that the algorithm produces results comparable to the simulated annealing.

Note that the tailored heuristic optimisation within MOODS performs only area/delay optimisation, while the simulated annealing is capable of performing a multi-dimentional optimisation between many objectives.

2.6.5 Hierarchical module expansion

Originally, MOODS considered functional units as pure combinational logic blocks. Hierarchical module expansion [19, 38, 39] provided a means of implementing multi-cycle technology-independent functional units, which get expanded in the internal design structure during synthesis. This enables inter-module optimisation at the sub-module level, allowing greater opportunities for functional unit sharing. Each expanded module is defined with separate sub-control and sub-datapath graphs, which replace the desired datapath functional unit and its activating control states.

An example of the expansion process is given in Figure 2.15. Before expansion, the addition is implemented using a combinational 64-bit adder that executes in a single control state (S_1). The functional unit is then replaced by an expanded module composed of an 16-bit adder that performs the addition operation over four control states. The original control node was replaced by the expanded module sub-control graph (S_1 to S_4), and the 64-bit combinational adder was replaced by a 16-bit adder and the required interconnect.

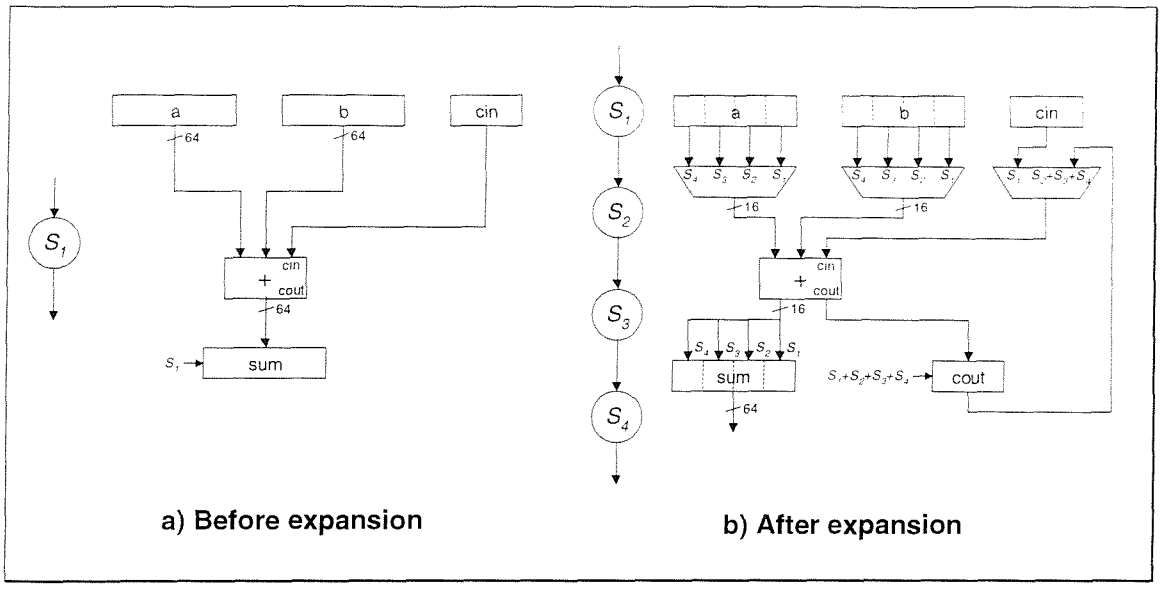


Figure 2.15 Expansion process

2.6.6 Floating-point enhancement

The core of this thesis describes an enhancement to the original MOODS synthesis system to allow synthesising designs incorporating floating-point variables and operations. These enhancements are identified in Figure 2.16, which reproduces the original system block diagram (Figure 2.9) with the newly added features.

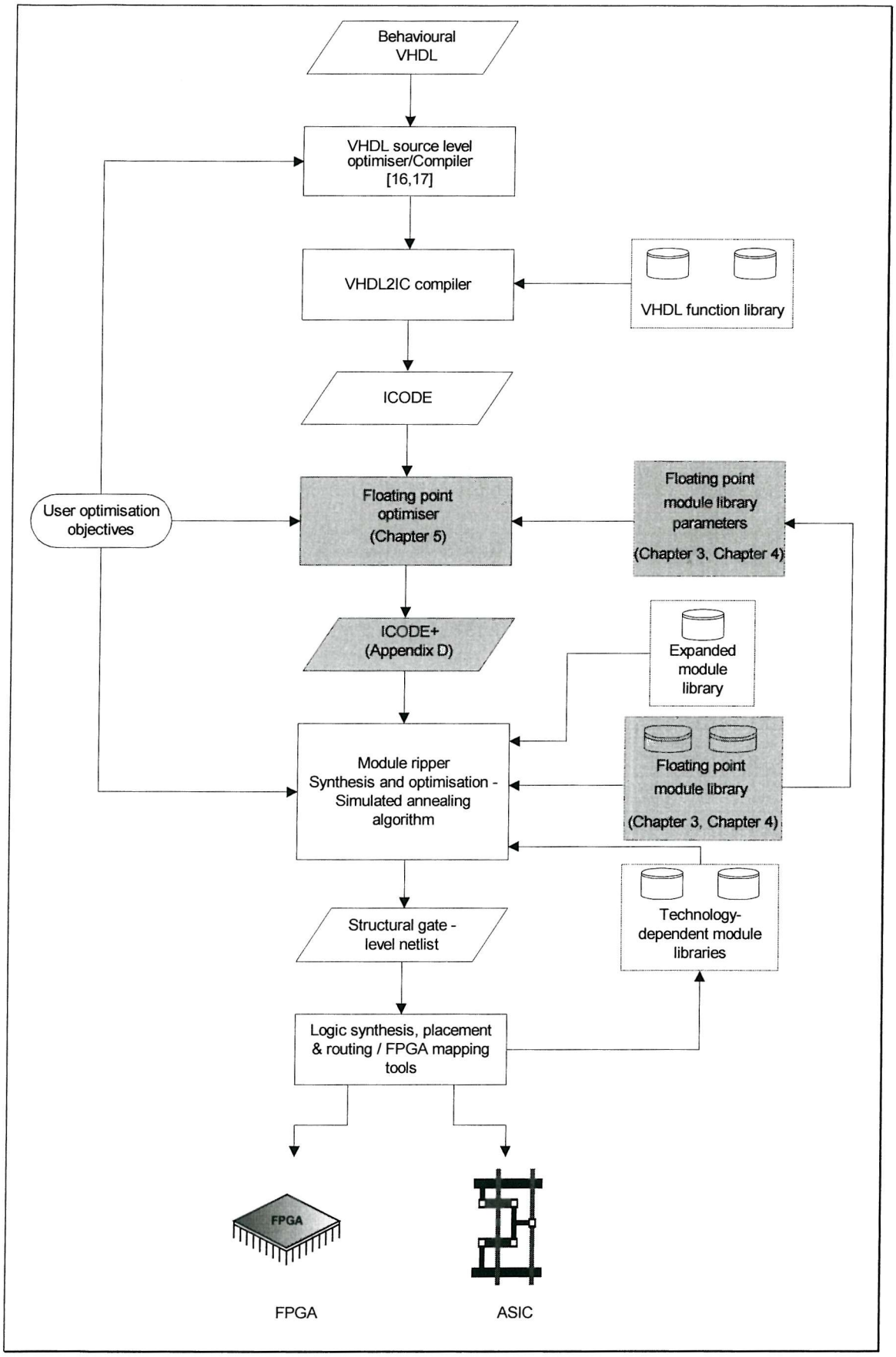


Figure 2.16 MOODS synthesis system with the floating-point enhancement

Chapter 3

Background and related work

This chapter presents background material describing influential research in the development of the *floating-point library* and *floating-point synthesis*. It is split into four main sections: section 3.1 describes the real number representation. Section 3.2 introduces some fixed-point functional units of a particular interest, while section 3.3 examines some research in the development of floating-point functional units. Attempts to implement floating-point arithmetic on programmable logic are introduced in section 3.4. Finally, section 3.5 describes a number of systems that automate the floating-point systems design process.

3.1 Real number representation

There is a fundamental difference between integer and real data types. In integer calculations, algorithms have discrete results, and ostensibly produce identical outputs on different machines. Real calculations do not always produce identical results due to the internal representation and the calculation accuracy. Early inconsistencies gave rise to a common real number representation with a clear definition of the way systems should handle real calculations, as well as the reaction of the system to exceptional situations (e.g. division by zero, overflow) [40].

The IEEE floating-point number representation [41, 42, 43] provides a solution to this problem. It provides four different representations of floating-point numbers. The standard gained a great popularity and most system manufacturers produce chips to support it. A detailed description of the standard is given in Appendix A.

This research adapts the IEEE single-precision floating-point word, which is 32-bits wide and arranged in the format shown in Figure 3.1. The floating-point word is divided into three fields: a single-bit *sign*, an 8-bit *biased exponent* and a 23-bit *fraction*.

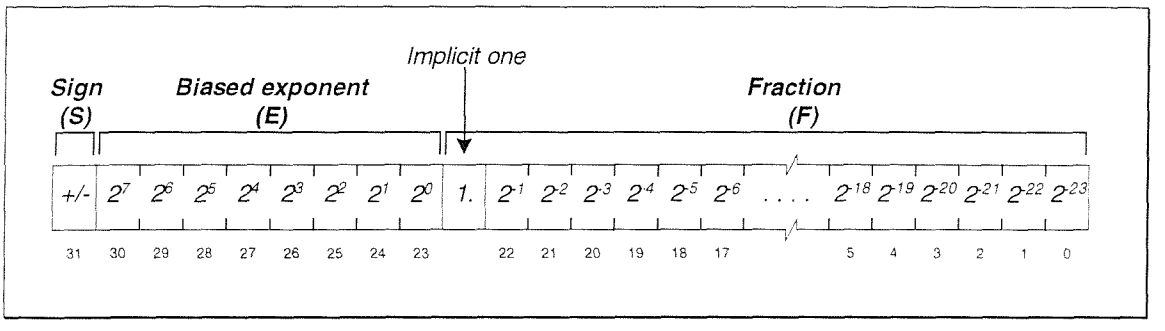


Figure 3.1 IEEE single-precision floating-point format

The *sign* bit (S) indicates the sign of the floating-point number, a negative value has a sign of 1; non-negative values have a sign of 0. The *biased exponent* is an unsigned integer field representing a multiplicative value of some power of two. The *bias* has a value of 127. If, for instance, the biased exponent has a value of x, then the actual exponent would be $x-127$. The *fraction* is a 23-bit field containing the 23 least significant bits of the number *mantissa*. The weight of the fraction most significant bit is 2^{-1}; the fraction least significant bit has a weight of 2^{-23}. The leading 1 in the mantissa field (bit 24) is implicit and does not appear in the fraction field. A 32-bit real number, y_1, is generated from $y_1 = (-1)^S \times 1.F \times 2^{E-127}$.

One of the most notable features of the IEEE standard is that it allows computation to continue if it faces an exceptional condition, such as dividing by zero. This is achieved by introducing special bit patterns that do not represent ordinary numbers. The standard defines five such bit patterns: zero, denormalised numbers, +/- infinity, and Not a number. These are described in Appendix A.

The IEEE floating-point format is not the only way to represent real numbers with finite precision. Various replacements have been proposed [40], although none have achieved the popularity of the IEEE floating-point format.

A particular number system that has been the subject of considerable interest is the *logarithmic number system* [44, 45, 46]. In this system, a real number is represented using the form $(-1)^S \times r^e$, with S being the sign bit and e is an exponent of the radix r. Figure 3.2

shows a general format of a logarithmic number. The exponent e is represented in a fixed-point number with n-bits for the integer part (i), m-bits for the fraction part (f), and 1-bit for the exponent sign (S). A real number, y_2, is generated from $y_2 = (-1)^S \times r^{i+f \cdot 2^{-m}}$, where r typically equals 2.

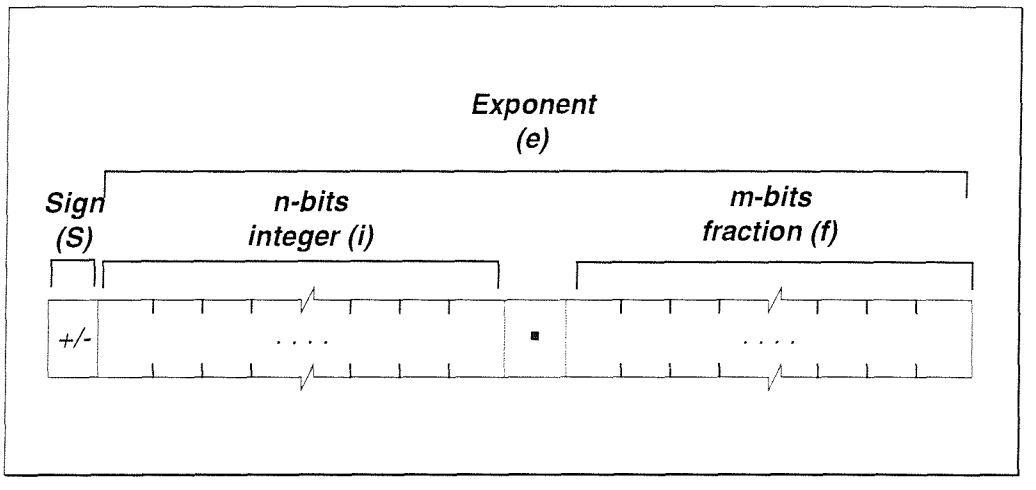


Figure 3.2 Logarithmic number format

The logarithmic number representation provides a very fast and easy basis for arithmetic operations that involve exponent manipulation, such as multiplication and division. However, addition and subtraction are slower in logarithmic number systems when compared to floating-point number systems, and also involve a sizeable lookup table. It is observed that the most frequent arithmetic operations are addition and subtraction[1] making logarithmic numbers less successful when compared to floating-point numbers. Recent work in [47] delivered a logarithmic arithmetic unit that performs addition and subtraction in a comparable speed to floating point units. However, the area cost of such implementation is still a disadvantage when a minimum area cost is the main objective.

3.2 Fixed point functional units

Multiplication and division are the basic operations underpinning most arithmetic processes. The way multiplication and division are performed have a major effect on the overall system performance. Purely combinational multipliers and dividers are not viable designs, they consistently give the largest area. This section describes multiplication and

[1] Addition and subtraction typically account for more that one half the total arithmetic operations in a typical scientific calculation [48].

division algorithms that allow a trade-off between system performance (delay) and hardware cost (area).

The section begins with a multiplication algorithm based on the modified Booth algorithm. Then, an algorithm for rapid binary division is outlined.

3.2.1 Modified Booth multiplier

The Booth multiplier was originally introduced as a uniform multiplication process, which is independent of the sign of the input operands [49]. A modification to this method allowed the reduction of the number of additions required to perform the multiplication operation at the cost of some extra control logic [50, 51].

In the serial-parallel form of the multiplication operation, the multiplicand is added to the partial product every time a one is detected in the multiplier. For a single cycle multiplication, this requires a number of add operation equals the multiplier width. The modified Booth multiplication reduces the required number of add operations by half[2], by regrouping the multiplier bits into groups of three bits (the multiplier should be first appended with zero by the lsb to form the first 3-bit group, and if necessary, zeros by the msb to form the last 3-bit group) that control the value to be added to the partial product. Modified Booth encoding is illustrated in Figure 3.3, and the value to be added in each iteration based on the multiplier bits.

Note that adding the multiplicand twice is simply achieved by shifting the multiplicand left and adding the result. Subtract twice is also performed by adding the two's complement of the latter. The example in Figure 3.4 illustrates the algorithm principle where two 5-bit (00101×01010) numbers are multiplied using this method. The multiplier is divided into three groups: the first group (100) indicates subtract twice operation, the second group (101) indicates a subtract one, and finally (001) indicates add once operation. Note that each of the three terms is sign extended up to the most significant bit of the final product.

---

[2] For an odd multiplier width, the number of adders required are n+1/2, where n is the multiplicand width.

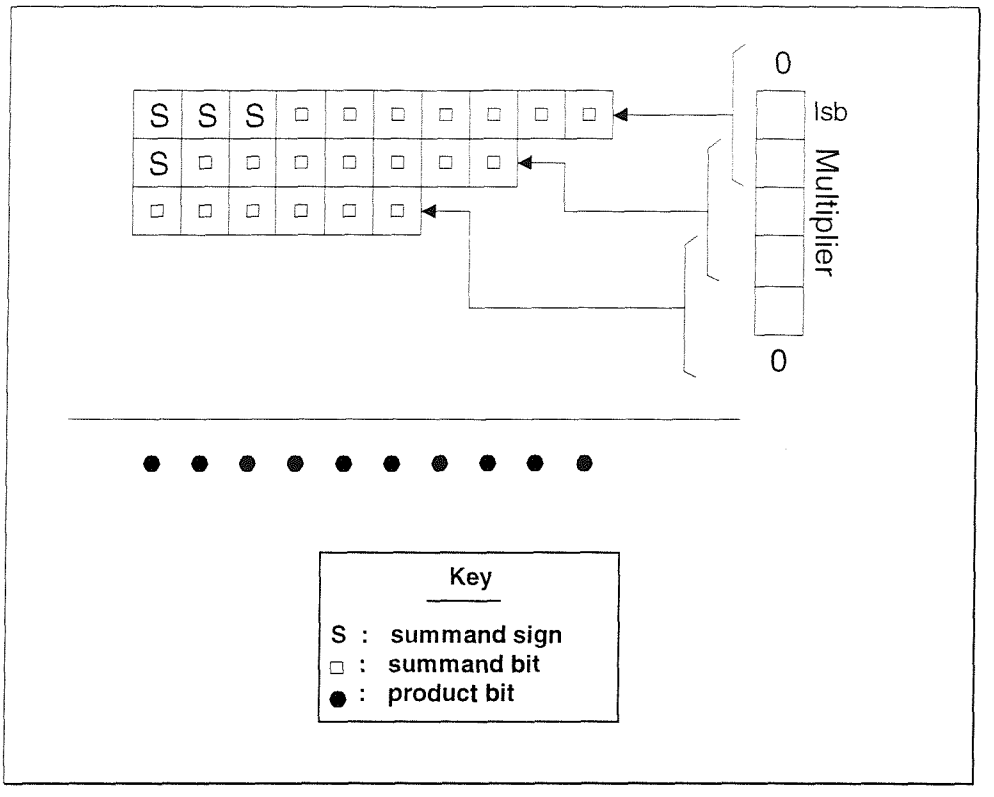


Figure 3.3 Modified Booth multiplier

Multiplier Bits	Selection	Summary
000	+0	No change to partial product.
001	+M	Add the multiplicand to the partial product.
010	+M	Add the multiplicand to the partial product.
011	+2M	Add the multiplicand twice to the partial product.
100	-2M	Subtract the multiplicand twice from the partial product.
101	-M	Subtract the multiplicand from the partial product.
110	-M	Subtract the multiplicand from the partial product.
111	-0	No change to partial product.

Table 3.1 Partial product selection

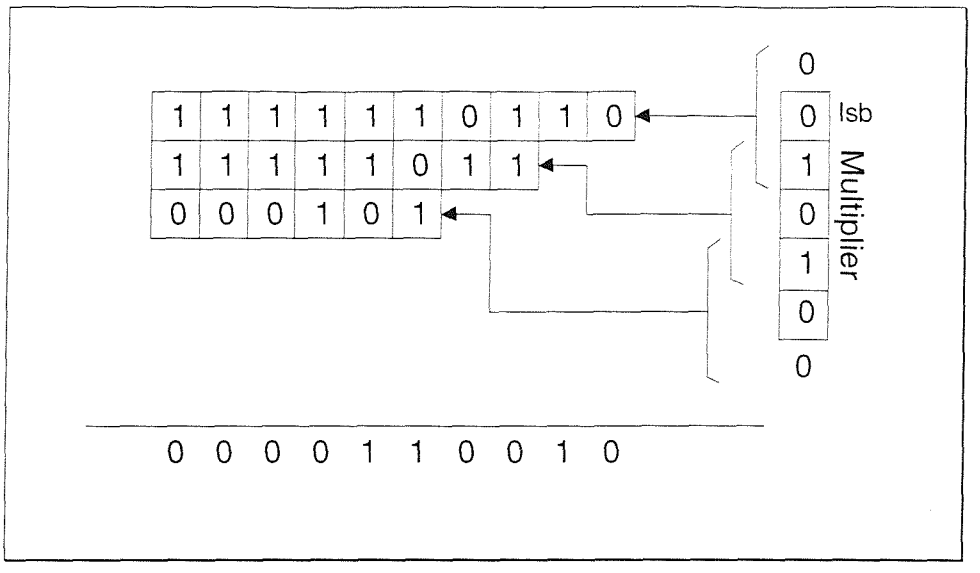


Figure 3.4 Modified Booth multiplication example

3.2.2 Rapid division algorithm

The rapid binary division algorithm or Wilson-Ledley division method [52, 53] provides a simple approach to dividing unsigned normalised fractions. The approach is based on the decomposition of a binary number into groups of strings of one of four types as illustrated in Figure 3.5. The string types are: all zeros, all ones, all zeros except one bit, and all ones except one bit.

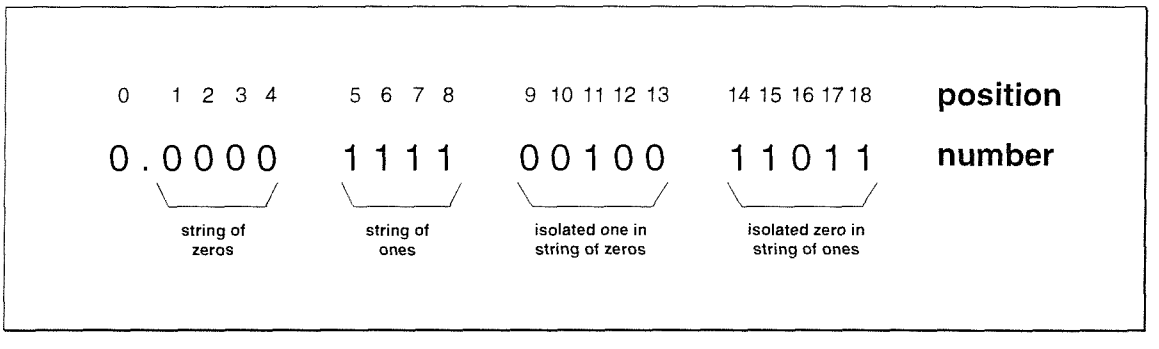


Figure 3.5 A decomposition of a number into four types of strings

The algorithm relies on a number of observations that benefit from the binary number decomposition illustrated above:

1. A string of ones from a to b positions contribute to the magnitude of the number by $(2^{-a+1} - 2^{-b})$.

2. An isolated one at position a in a string of zeros contributes by (2^{-a}) to the magnitude.

3. A string of ones from a to b positions with an isolated zero at c position contributes the value $(2^{-a+1} - 2^{-b} - 2^{-c})$ to the magnitude.

Based on these observations, the algorithm tries to detect similar strings that may occur in the division result and generates them at once. The procedure[3] is summarised in Figure 3.6; the procedure ends when i equals the result length. Before applying the procedure, three main conditions should be satisfied.

1. The denominator D should be positive and normalised.

2. The numerator N should be positive, with $N < D$.

3. N is either normalised or with a single zero to the right of the binary point.

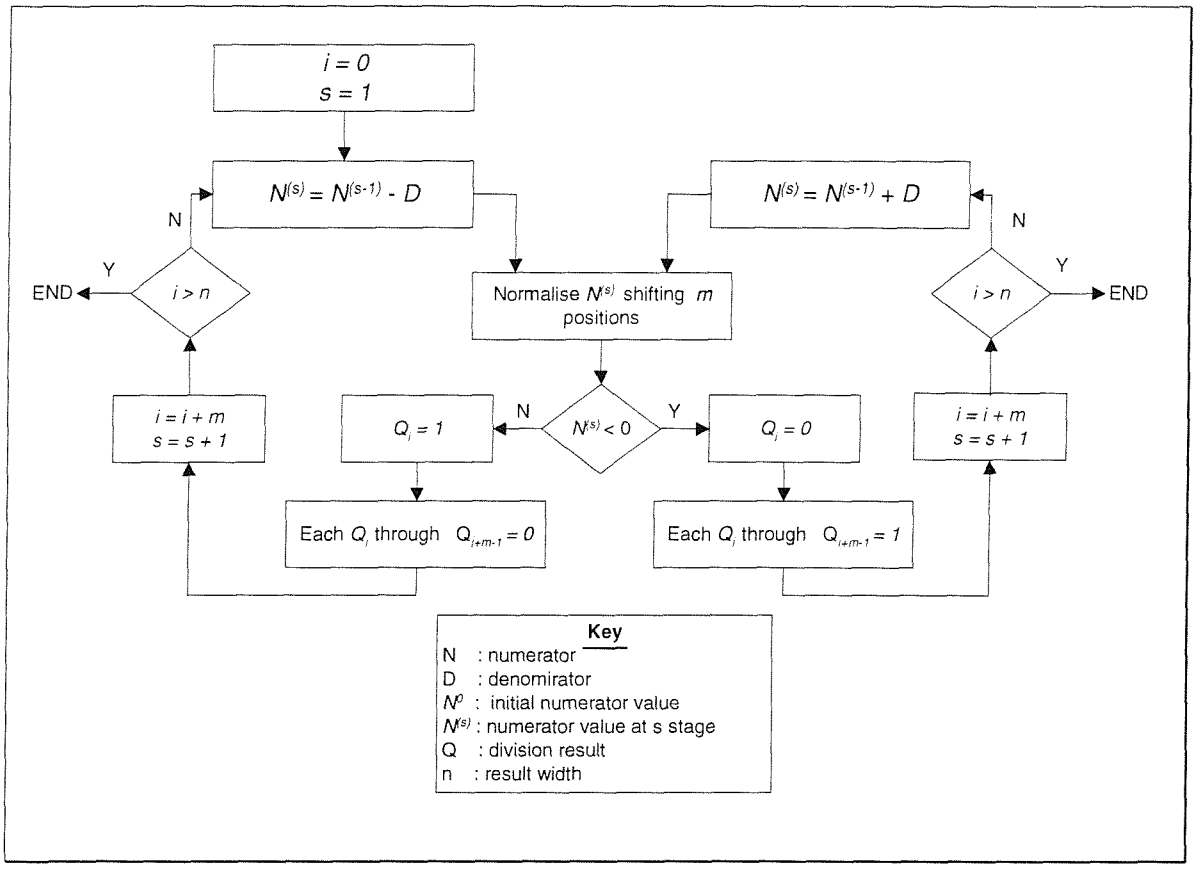


Figure 3.6 Rapid division algorithm flowchart

[3] For more details on the algorithm and its relation to the binary decomposition see [53]

3.3 Developing floating-point functional units

Research carried out in the development of floating-point functional units can be divided into two areas: research dedicated to developing floating-point arithmetic units (adder, subtractor, multiplier, divider) mainly for hardware implementation, and the development of algorithms for elementary function evaluation at both the hardware and software levels.

An example of the first area work carried out by Oberman [48, 54] to investigate different methods of implementing high-performance floating-point arithmetic units, and proposed techniques to improve the performance of these units, mainly to speed up future microprocessors. One of the techniques introduced allows a full-precision floating-point addition operation to execute with an average delay of 2.25 clock cycles. This was achieved by exploiting the distribution of operands over redundant datapath hardware and employing pipelining and fast rounding methods. However, the significant hardware cost makes these techniques unsuitable for low cost designs, or designs targeting programmable logic devices.

The CORDIC algorithm (Co-ordinate Rotation DIgital Computer) is one example of an efficient algorithms to evaluate elementary functions. The algorithm was introduced in 1959 by Volder [55] as a method to rotate a vector by an arbitrary angle, or to determine the angle and the magnitude of a vector. Besides vector transformation, the algorithm computed *sine*, *cosine* and *inverse tangent* functions. Walter [56] generalised Volders algorithm to support a wide range of hyperbolic, logarithmic and exponential functions. A recent modification to the algorithm [57] enables the computation of *inverse sine* and *inverse cosine* functions. The CORDIC algorithm exhibits linear convergence, which implies that generating an n-bit result requires n iteration. Moreover, the algorithm is simple to implement and requires minimal hardware. Details of the CORDIC algorithm may be found in Appendix B.

ATA (Add – Table lookup – Add) is another method for evaluating elementary functions [58]. The method evaluates these functions using a truncated Taylor series and a large table (around one megabit for a single instruction). The method involves evaluating a Taylor series approximation by parallel add/subtract, parallel table lookup, and followed by a multi-operand addition. The proposed hardware implementation is very fast. However, the table lookup size required to generate a single elementary function is

868352 bits, and the total amount of table size required to calculate seven elementary functions is about 14.2 Mbit. This large table size introduces a problem, in terms of internal storage area, if the algorithm is to be realised as a single chip design.

A software library for elementary function calculation using the IEEE floating-point standard was proposed in [59]. The library combined a table lookup method with minimax approximation polynomials [60, 61] to develop high-performance software models with maximum accuracy. The proposed algorithms, along with similar software-based algorithms are often discarded in the hardware domain due to the large area overhead they impose.

3.4 Floating-point arithmetic on FPGA

There have been several studies to investigate the possibility of implementing floating-point operations on programmable logic devices. Programmable logic devices impose limitation on the number of functional units, storage devices, and interconnect. This lead designers to avoid implementing floating-point operations on programmable logic devices, simply because these operations typically require a large area to be practical on these devices.

A recent study [62] offered evidence that floating-point implementations on FPGAs should be considered. It introduced a single precision floating-point adder and multiplier realised on a Xilinx 4020E FPGA. The author argued that a single precision floating-point unit implemented on FPGA would give a reasonable performance improvement for floating-point applications over the currently available microprocessor. Moreover, he suggested that if programmable logic device density and speed continue to increase, platforms based on programmable devices might offer a significant speedup to pure floating-point applications.

Similar work [63] proposed two single precision floating-point square root implementations on FPGAs. The author surveyed different methods of implementing a square root functional unit, and decided on an iterative method based on a single 24-bit adder/subtractor functional unit. A high performance implementation of the same algorithm was also highlighted. The second implementation exploited parallelism using a

fully pipelined implementation at the cost of extra hardware (almost five times the cost of the first serial implementation).

An FPGA prototyping board using an Altera Flex 81188 FPGA was the target for single precision floating-point addition and multiplication units in [64]. The design was used to simulate the interaction of galaxies in what is called a gravitational N-body model. A point of particular interest in this work is the extra limitation introduced by FPGA devices on a prototyping board. The chip pins in this case are pre-assigned, which imposes additional constraints on the placement and routing tools and results in less efficient utilisation of the FPGA resources.

A different approach to implementing floating-point operations was presented in [65]. The work minimised the floating-point implementation cost by introducing smaller floating-point formats. These formats are shown in Figure 3.7. The 16-bit format has a 9-bit fraction field and 6-bit biased exponent, with a bias of 31. The 18-bit format has a 10-bit fraction field and a 7-bit biased exponent, with a bias of 63. The approach gives a major reduction to the total cost of the implementation, but results in a reduction of both the dynamic range and the representation accuracy, which might be suitable for a specific implementation, but is not considered a general-purpose approach to floating-point calculation.

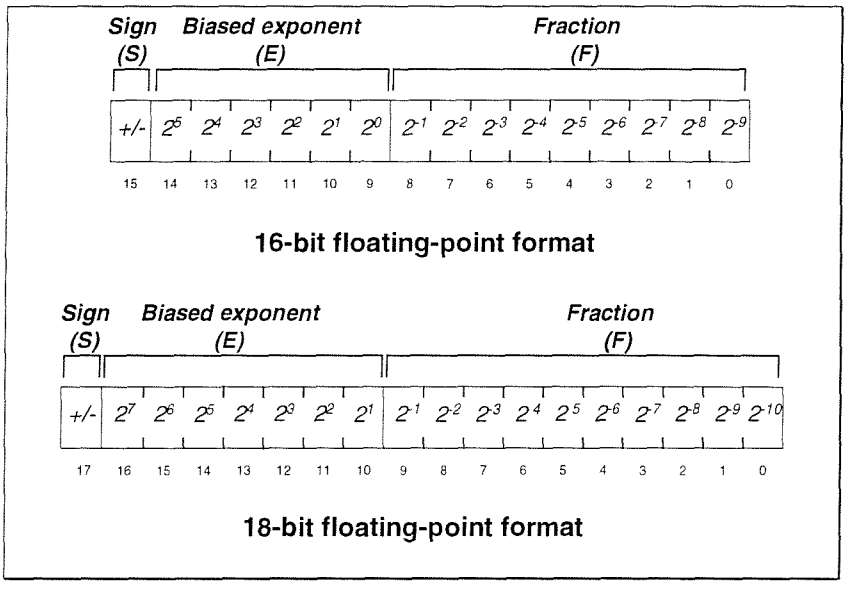


Figure 3.7 Short floating-point formats

The work purposed in [66] introduced an FPGA-based floating-point data path as a building block in a geometric processor dedicated to co-ordinate transformations in a graphics system. The data path performs 32-bit floating-point addition, subtraction, multiplication, division, and comparison operations. The design exploited the similarity in the floating-point operations to reduce the total area cost. This is achieved by partitioning the data path into four main units illustrated in Figure 3.8: an exponent manipulator; a fraction manipulator; a fraction arithmetic unit; and a control unit. A simple adder/subtractor unit is employed in the fraction arithmetic unit, which implies a serial-parallel or "pencil and paper' implementation of the fixed-point multiplication and division operations. The proposed data path provided a single flag to indicate overflow, and ignored all other exceptional situations such as (NAN) to minimise the cost of hardware resources.

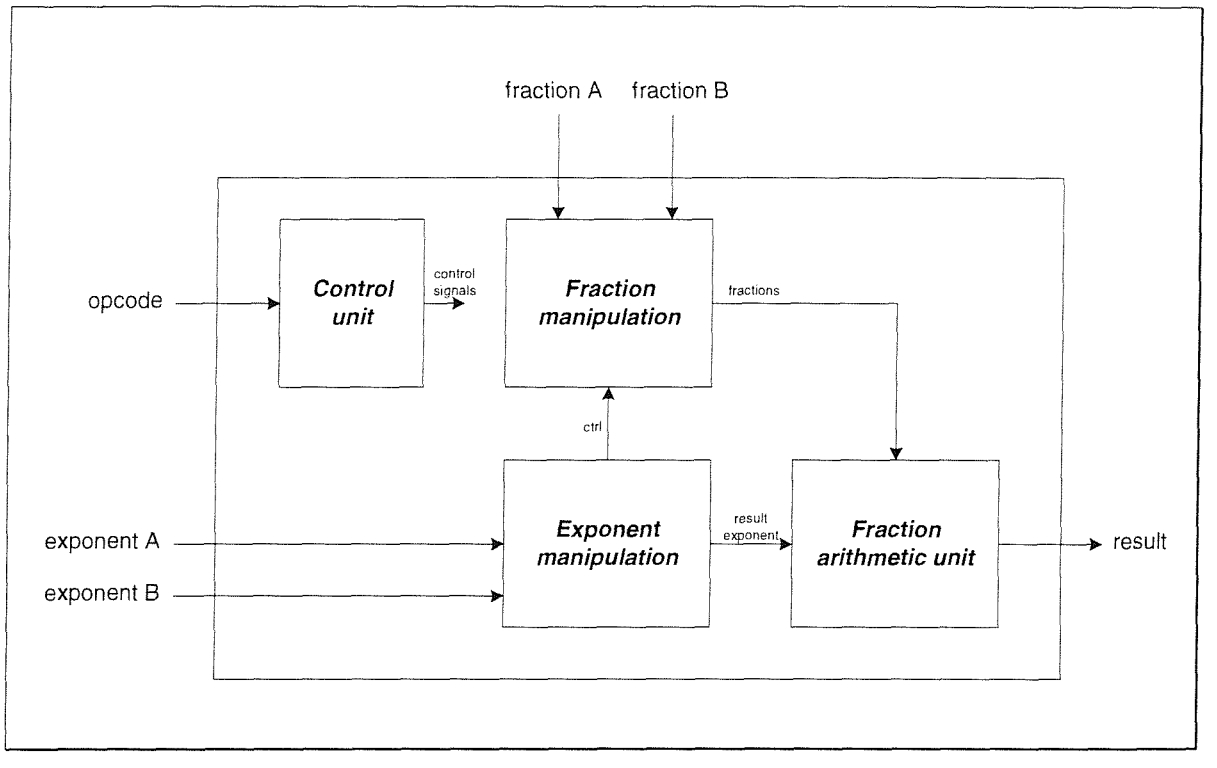


Figure 3.8 FPGA-based data path block diagram

In contrast with the above, the work in [67] introduced a self timed single precision floating-point processor based on a combination of ASIC and FPGA. Instead of a global synchronising clock signal, the system adapted a handshaking protocol, where design units are locally synchronised using handshaking signal (strobe and acknowledge). The processor performs three floating-point operations: addition, subtraction, and division.

Addition was implemented in the ASIC, while the floating-point multiplier and divider targeted the FPGA. The independence of the addition hardware and the multiplication and division hardware allowed parallel scheduling of the instructions, as well as out of order execution. The processor adapted the simplest form to implement floating-point operations, such as employing serial-parallel fixed-point multiplication and division algorithms, in order to reduce the total hardware cost and increase the probability of successful processor functionality simply by reducing the complexity of the design.

3.5 Automatic floating-point implementation

An early attempt to automate floating-point implementation appeared in [68]. The work highlighted a design concept for digital signal processing applications using floating-point primitives, which was integrated within a synthesis environment called the ASA Silicon Compiler, by means of a template library. The author introduced a 32-bit floating-point adder to demonstrate the concept of the DSP template library. The adder was integrated as a generic primitive in the template library. Unfortunately, details of primitive implementation and integration within the silicon compiler environment were not presented.

The remaining part of this section introduces two groups of floating-point implementation tools: tools that allow the generation of floating-point units that can be integrated within a system (module generators), and high-level block-diagram tools.

It is worth mentioning that floating-point cores designed for rapid insertion into an ASIC environment are available at a commercial level in the form of cell-level designs, as well as behavioural VHDL or Verilog models for synthesis. However, this work achieves its goals ultimately by sharing the internals of the floating-point units; third party 'black box' are not considered further.

3.5.1 Module generators

A format conversion module generator is introduced in [69]. The module generator allows the automatic design of VLSI modules that perform floating-point to fixed-point conversion and vice versa. The module generator accepts any standard cell library and design rules. The output of the generator consists of the physical layout view, the netlist

file and all the information required to generate a SPICE file. It also provides the physical characteristics of the generated module, such as input and output location, and area utilisation.

The module generator is considered general purpose, as it is not limited by the representation of the fixed-point and the floating-point numbers. Based on a set of parameters specified by the user, the module generator decides on the appropriate structure. For example, for a floating-point point number, the user defines the number of bits of the fraction, the number of bits of the exponent and the exponent bias. While a fixed-point number is defined by the size in bits, the point position, and the number representation which can be sign-magnitude, one's complement, or two's complement.

The module generator did not provide options to integrate the generated module within a design environment. The designer has to deal with the module as a black box that performs the conversion and provide an interface for it. A better approach would be to provide the generated module at the register transfer level using a hardware description language. In that case, the design could target an RTL-synthesis tool. This reduces the effort required to verify the functionality of the whole system that exploits the generated module since the whole system can be simulated at the RTL-level rather than at post-layout level. The suggested approach might also result in a total area cost reduction as functional units within the module might be shared with other operations when the module is idle.

Many floating-point arithmetic units are available in the form of *macrocells.* A macrocell is defined as a medium to very high complexity block with given functionality, known interconnect interfaces and different interconnect level called *views* (e.g. behavioural, RTL, layout, etc.). GenOptim [70, 71] is one example of a tool created to design portable macrocells generators. It is a CAD tool that supports the implementation of architectural representation in different layout environments and different target technologies. It provides the designer with a set of high-level C function to describe the netlist, the layout, the test vectors, and the behavioural description of a parameterised module. GenOptim then provides an implementation of this module based on what is called a *virtual library*, which is a set of parameterised high-level operations (e.g. n-bit adder, n-bit multiplier).

The generator created by GenOptim can then be used to implement a technology-dependent macrocell. The process involves defining a GenOptim virtual library in terms of the target technology cell library, and providing a set of parameters that defines the parameterised datapath units width (number of bits). The generator takes these inputs and automatically creates a set of outputs: a netlist describing the hierarchical interconnects between cells; a layout providing the placement of these cells; test vectors; and a VHDL behavioural description for simulation purposes.

GenOptim has been used to implement a set of portable floating-point arithmetic unit generators based on the IEEE floating-point standard. Four generators were introduced to provide floating-point addition, floating-point multiplication, floating-point division, and floating-point square root operations. These generators had a parameterised fraction and exponent field to allow implementing any of the standards formats (single precision, double precision, extended single precision, and extended double precision).

Another system, similar in structure to GenOptim, is the CXgen function library [72]. CXgen also provides the designer with a C library that can be used to describe and implement portable parameterised generators. The author presented a floating-point adder generator called GAF implemented using the CXgen environment. Starting from a set of parameters, GAF generates a floating-point adder described via a layout view, a netlist view, and a behavioural view. GAF also supports testability via a set of test vectors based on a structural analysis of the generated adder to ensure that the circuit is fully functional.

3.5.2 Block diagram tools

Digital systems can be represented as a network of transfer functions, data storage, I/O ports, and control functions. Such systems may be represented by *block diagrams* consisting of blocks representing functions linked by lines representing the communications paths. An example of such block diagram is represented in Figure 3.9. Each block in the diagram represents a function that can either be simple (e.g. fixed-point adder) or complicated (e.g. floating-point multiplier). These blocks are connected with directed arcs defining the data flow through the network.

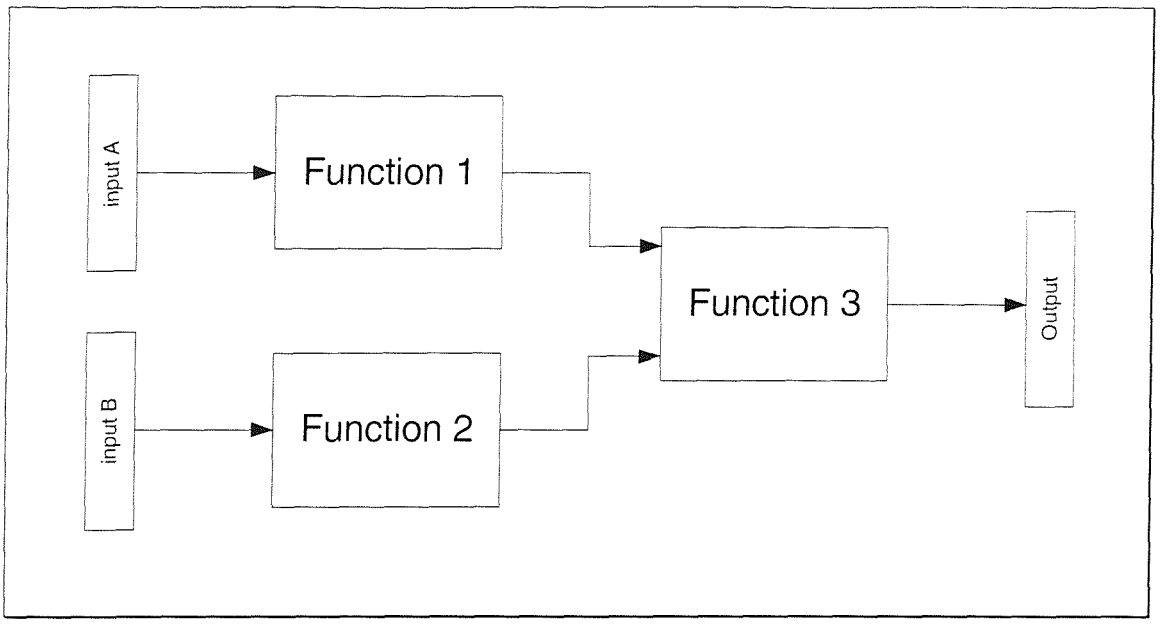


Figure 3.9 A design represented as a block diagram

Block diagrams form the input to a family of CAD tools known *as block diagram oriented systems*. The complete design flow of these systems is represented in Figure 3.10. The system allows the user to create a diagrammatic representation of the design using components provided by a block library. The design is then captured as a behavioural description or a register transfer level description. This step involves either a behavioural synthesis or an RTL synthesis depending on the nature of the design representation generated in the previous step. Finally, the structural representation of the design passes to a placement and routing tool to be realised as a physical implementation.

Block diagram oriented tools also provide the ability to add new building blocks to the block library, which increases the system productivity and allows designing reusable blocks. A number of these tools integrate floating point synthesis by providing a number of floating point building blocks that can be instantiated within the system block diagram.

COSSAP design environment [73] is one example of block-diagram oriented systems. It captures the systems representation in the form of a synthesisable HDL code (VHDL or Verilog HDL) using COSSAP HDL code generator. The system provides HDL code at both the behavioural and RTL levels, and provides two different implementation roots by integrating a behavioural synthesis tool [74] and an RTL synthesis tool [75] within the system.

COSSAP provides a powerful and efficient environment for digital signal processing applications. However, floating-point manipulation within the system is limited by the block library component, which currently support single precision floating-point addition and multiplication only.

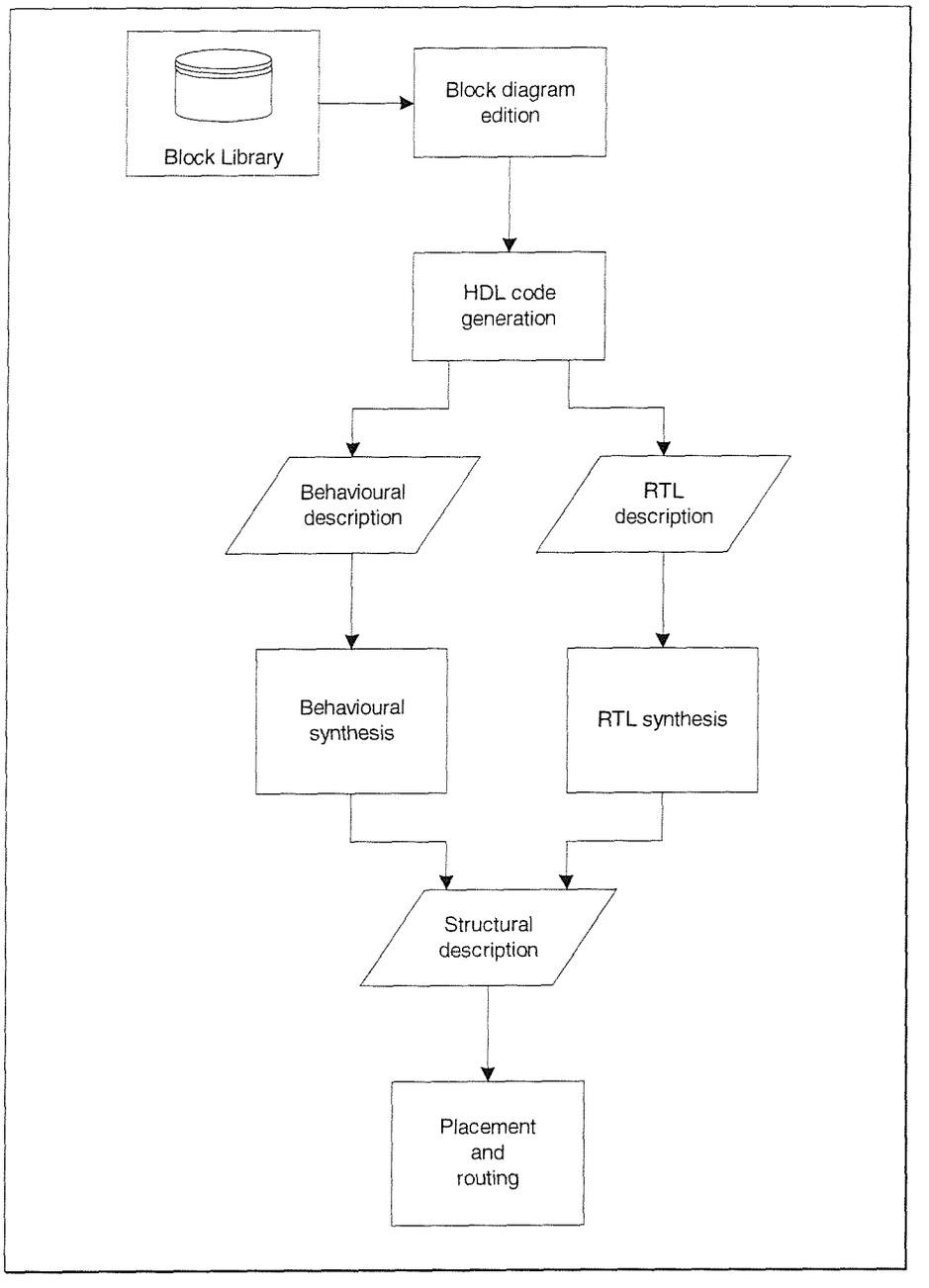


Figure 3.10 Block diagram oriented tools data flow

SPW [76] is another CAD tool that supports digital design using block diagrams. In a similar manner to COSSAP, the system automatically captures the design as a HDL behavioural description. The code is then synthesised, using an integrated behavioural synthesis tool [77], into a structural implementation. SPW appears to have more support

for floating-point manipulation in comparison to COSSAP. For example, a dedicated floating-point communication library is provided as an add-on to the system [78,79], which allows the design and implementation of digital designs incorporation floating-point building blocks, dedicated for digital communication and wireless applications.

Block diagram oriented tools provide a fast and convenient design environment for digital design where series of operations are applied continuously to a data stream. However, many applications require a significant amount of control logic based on external and internal variables. Expressing this dependency may be difficult and even impossible using block diagrams, while it can be easily achieved using conditional constructs provided by programming languages (case, if-else constructs). Moreover, when using these tools, it is the designer responsibility to perform the high level binding of the high-level floating-point operation to building blocks. This manual binding decision may result in blocking a number of possible implementations, hence, reducing the possibility of the structural implementation meeting the target objectives.

Chapter 4

Floating-point library design

The floating-point library forms the core of the floating-point synthesis system. The aim of this chapter is to highlight the various floating-point modules that form the basis for the floating-point synthesis library. Functional unit structure is introduced and different methods for evaluation of these functions are considered.

The text is divided into four main sections: section 4.1 describes the function evaluation process with an analysis of the different building blocks that composes the functional unit; section 4.2 examines the issue of the *status register* as a mean of "exception notification" to handle invalid operations; section 4.3 provides a brief description of each component in the library; finally, section 4.4 covers various issues that concerns the library implementation and integration within the MOODS synthesis system.

4.1 Function evaluation

The general structure of these functional units is represented in Figure 4.1. Each functional unit consists of three main building blocks:

1. Range reduction.

2. Function evaluation.

3. Post evaluation rounding and normalisation.

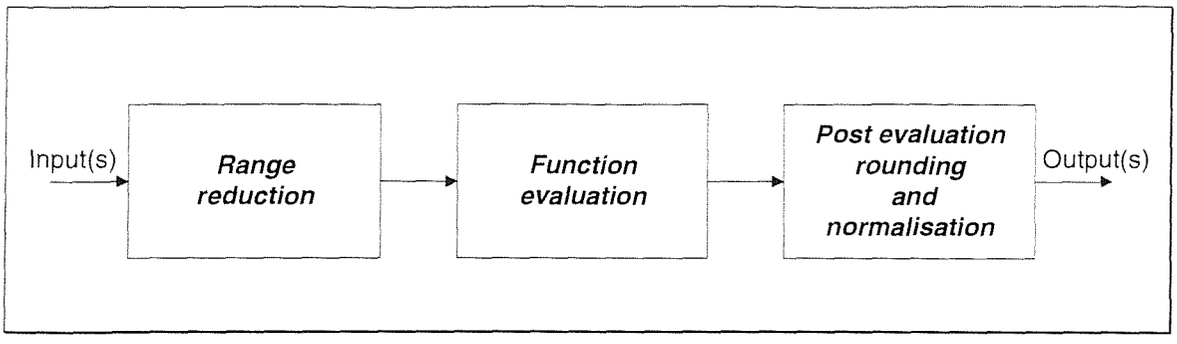


Figure 4.1 Functional unit building blocks

Three different *base techniques* are used to implement the function evaluation block:

1. Table lookup.

2. Iterative series.

3. The CORDIC algorithm.

These techniques generate modules with significantly different physical properties such as the total area cost and the total delay. This variation in the physical properties makes it possible to provide a wide range of implementations for a single floating-point design, which increases the probability to provide a single implementation that meets the user objectives. The floating-point library provides at least two different evaluation cores using two of the three base techniques listed above for each implemented function.

4.1.1 Range reduction

The large dynamic range provided by a floating-point representation introduces a problem when designing systems to handle floating-point arithmetic. Some evaluation methods, such as iterative series, converge over a wide range of input arguments. However, achieving certain accuracy over that range might require taking many terms into account, hence, increasing the evaluation time dramatically. Moreover, the time taken to achieve a given accuracy is data dependent. Other methods, such as the CORDIC [56, 61] algorithm has a limited domain of convergence. Having a suitable technique to reduce the range of the input operand(s) is therefore essential.

Periodic and symmetric functions have obvious reduction, others might require shifting and scaling. By way of an example, let us consider the natural logarithm function ($y = ln(x)$), where $x = F \times 2^E$. The function is defined for $x > 0$. Range reduction can simply be achieved by the pre-scaling identity:

$$\ln\left(F \times 2^E\right) = \ln(F) + E \times \ln 2$$

The output of the range reduction unit is generally a set of fixed-point variables and a set of control signals. The output variables form the input to the following function evaluation units, while the control signals govern the data manipulation of the unit. This dependency maybe illustrated with the aid of the example in Figure 4.2 which evaluates $y = \sin x$ for arbitrary x. The output of the sine function range reduction block is a fixed-point number D and two control signals. One to decide on generating either sine or cosine in the function evaluation block and the other controls the final sign of the output operand.

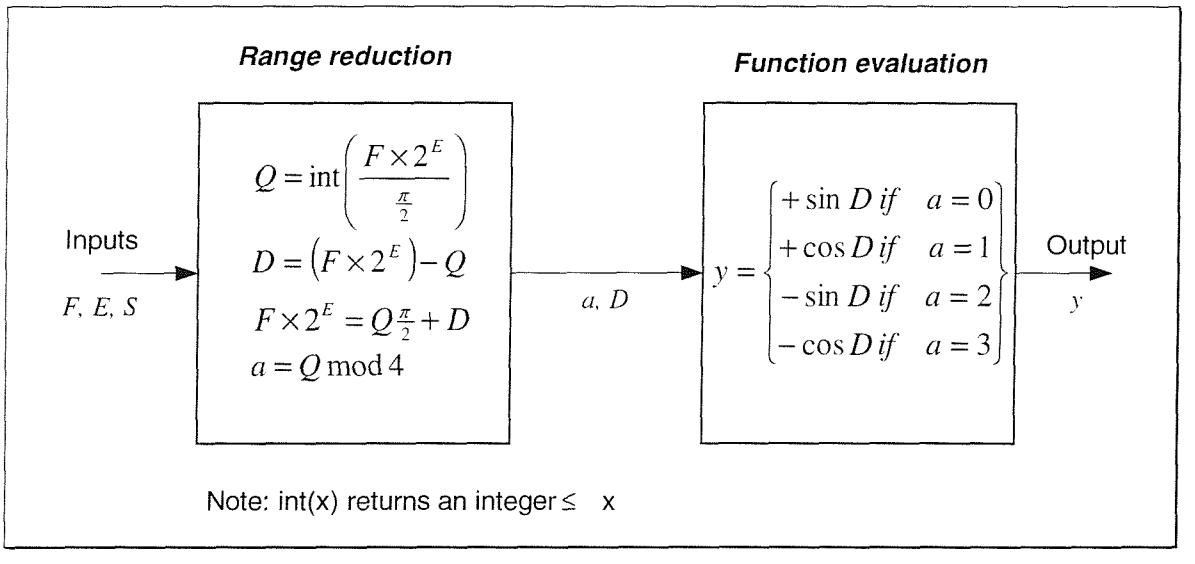


Figure 4.2 Range reduction example

4.1.2 Table lookup

Lookup tables are frequently and trivially used to evaluate mathematical functions. This scheme has often been rejected in practical cases, because of the large table sizes required for acceptable accuracy. However, combining range reduction techniques with a dedicated interpolation procedure gives rise to a large reduction in table size, often to the point that it may be reduced to an on-chip set of static registers rather than an external ROM.

Linear table lookup

For a single numerically given point x, the value of an arbitrary function $f(x)$ at this point can be evaluated [80] using the procedure described in Figure 4.3.

DEFINITIONS:

λ_i :interpolation coefficient

x_i :table break point

μ_i :weighting value

y_i :$f(x_i)$

n :interpolation order

PROCEDURE:

$$\lambda_i = \frac{1}{(x_i - x_0)\ldots(x_i - x_{i-1})(x_i - x_{i+1})\ldots(x_i - x_n)}$$

$$\mu_i = \frac{\lambda_i}{x - x_i}$$

$$f(x) = \frac{\sum_{i=0}^{n} \mu_i y_i}{\sum_{i=0}^{n} \mu_i}$$

Figure 4.3 Interpolation procedure

For linear interpolation (n = 1) and a ***linearly*** distributed table (equally spaced break points), the procedure can be simplified to the form shown in Figure 4.4, where a function $f(x)$ is defined by a set of values $(y_o - y_n)$ stored in a table. For a quadratic interpolation, the general procedure outlined in Figure 4.3 applies. However, the computation problem can be simplified for the cubic interpolation procedure [80] as illustrated in Figure 4.5 and Figure 4.6, where a function $f(x)$ is interpolated using four linearly distributed break points (x_0, x_1, x_2, x_3). From Figure 4.5, it is clear that cubic interpolation result equals to the sum of the linear interpolation (L) over the central interval (x_1, x_2) and a numerical value Z. By introducing the relative distances between the input argument and the two internal break points p,q. It can be proved that Z has the value:

$$Z = \frac{pq}{2} \times (L - L_1)$$ [1]

where L_1 is the result of the linear interpolation over the interval (x_0, x_3).

Cubic interpolation can therefore be generated in a simple way from two linear interpolations as illustrated in Figure 4.6.

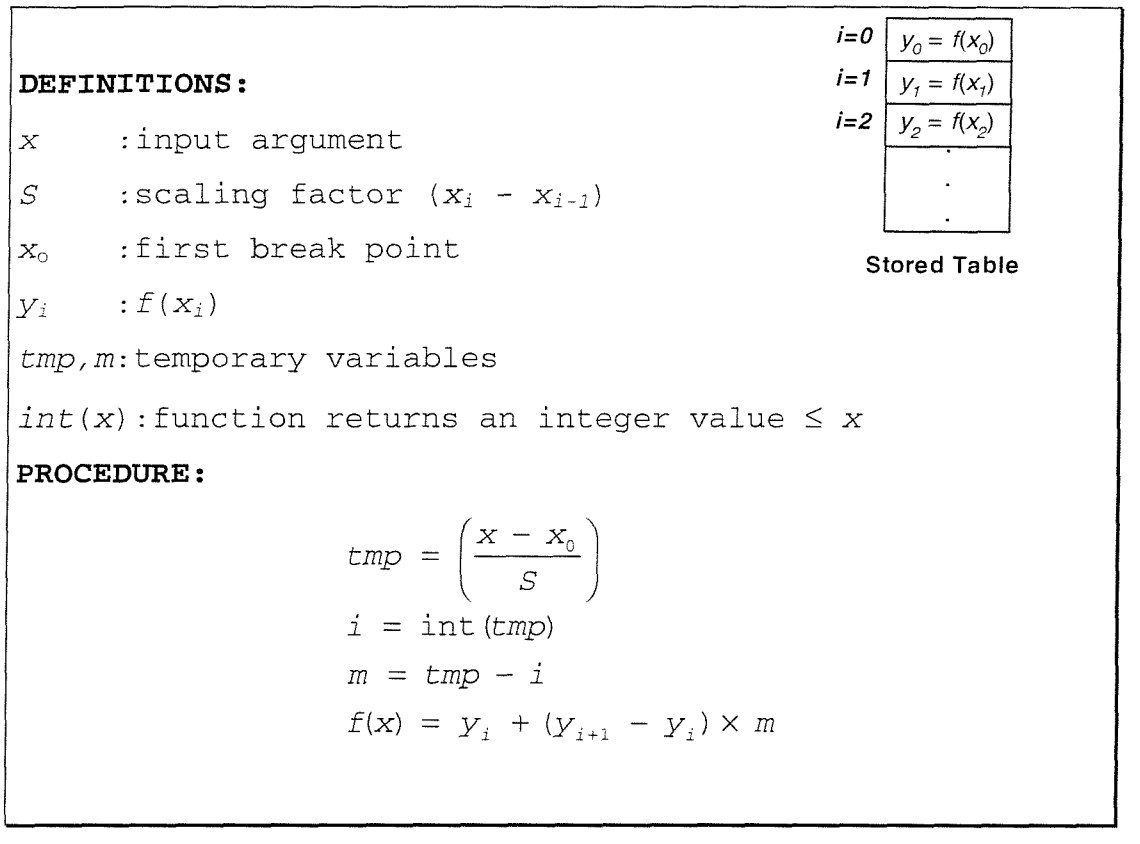


Figure 4.4 Linear interpolation procedure

[1] A proof of this equation can be obtained by consulting [80].

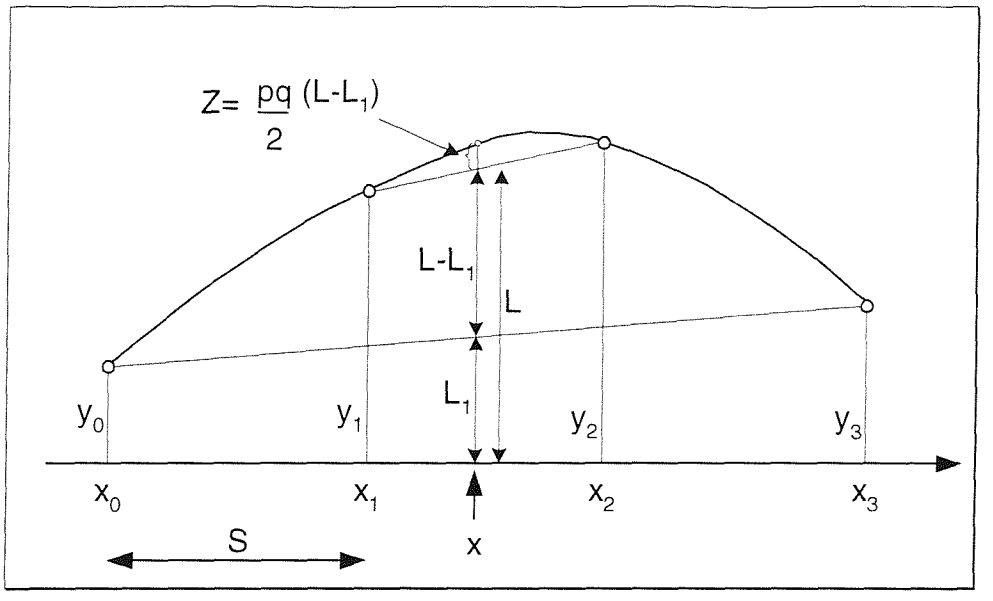


Figure 4.5 Cubic interpolation

DEFINITIONS:

x :input argument

S :scaling factor (x_i - x_{i-1})

x_0-x_3 : break points

y_i :$f(x_i)$

L :result of the linear interpolation over the interval(x_1,x_2)

L_1 :result of the linear interpolation over the interval(x_0,x_3)

p,q :relative distances between the input and the two internal break points

PROCEDURE:

$$p = \frac{x - x_1}{S} \quad , \quad q = \frac{x_2 - x}{S}$$

$$f(x) = L + \frac{pq}{2}[L - L_1]$$

Figure 4.6 Cubic interpolation procedure

For a given accuracy, a major reduction in the table size may be achieved by using higher order of interpolation. This is illustrated in Table 4.1which represents the cost as a number of table entries required to evaluate the sine function over the range $0 \leq x \leq \pi/2$, using different degrees of interpolation and for different accuracy. The results suggest that better

Interpolation degree	Number of table entries			
	Accuracy 0.1%	Accuracy 0.01%	Accuracy 0.001%	Accuracy 0.0001%
Linear	26	101	202	805
Quadratic	10	26	51	101
Cubic	8	15	28	53

Table 4.1 Number of table entries for different interpolation degrees

results can be achieved by replacing the linear interpolation procedure with a quadratic or cubic or even higher order interpolation, but the additional cost of the interpolation engine usually outweigh this advantage. The problem is quantified in Table 4.2, where the total interpolation engine cost in terms of on-chip area and total delay is provided for the sine function generator for different target accuracy and in two distinct cases:

1. An infinite off-chip ROM is available to store the table.

2. Table is stored as a set of on-chip static registers.

Each configuration is given a reference code. When applicable, the total area cost includes the cost of implementing the internal table as a set of static registers. From the table, it is clear that the linear interpolation engine provides the fastest function generation and is the best implementation when an external ROM is available. However, a cubic interpolation engine has the advantage of smaller storage area especially at high accuracy targets at the cost of extra delay (≈2.25 times the linear interpolation engine delay). The extra delay cost reduces the performance of the evaluation unit to the level that can be achieved with less expensive algorithms (such as CORDIC), which contrasts with the main objective of implementing functions using table lookup, which is minimum delay. The quadratic interpolation engine on the other hand always provides the worst area and delay figures and therefore is considered as impractical solution for all configurations. A note of

particular interest is that if the whole table may be implemented as an external ROM, accuracy variation will have absolutely no effect on the total area and delay cost of the design. That is because the interpolation procedure remains the same while accuracy in this case only affects the table size.

The results are summarised in Figure 4.7 to Figure 4.9. Figure 4.7 shows a comparison of the three interpolation engines in terms of the table size for different target accuracy. Figure 4.8 and Figure 4.9 compares the total area and delay cost the three engines for various accuracies and with or without the external ROM.

External ROM	Degree	Accuracy							
		0.0001%		0.001%		0.01%		0.1%	
		Area μm^2	delay cycles	area μm^2	Delay Cycles	area μm^2	delay cycles	area μm^2	delay cycles
∞	Linear	150000	26	150000	26	150000	26	150000	26
	Ref	A1		A2		A3		A4	
	Quad	400000	74	400000	74	400000	74	400000	74
	Ref	B1		B2		B3		B4	
	Cubic	260000	57	260000	57	260000	57	260000	57
	Ref	C1		C2		C3		C4	
0	Linear	440000	20	300000	20	200000	20	163000	20
	Ref	D1		D2		D3		D4	
	Quad	450000	65	430000	65	413000	65	405000	65
	Ref	E1		E2		E3		E4	
	Cubic	291000	45	274000	45	267500	45	264000	45
	Ref	F1		F2		F3		F4	

Table 4.2 Interpolation area and delay figures for various configurations

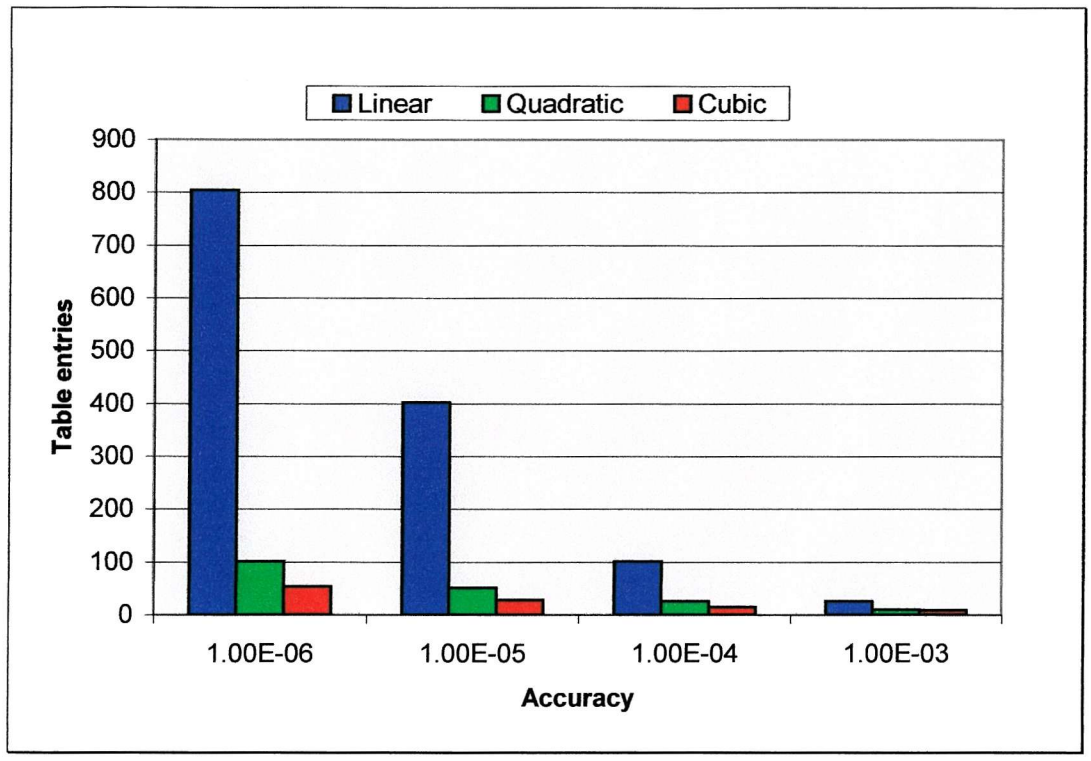


Figure 4.7 Table entries variation with different interpolation degrees

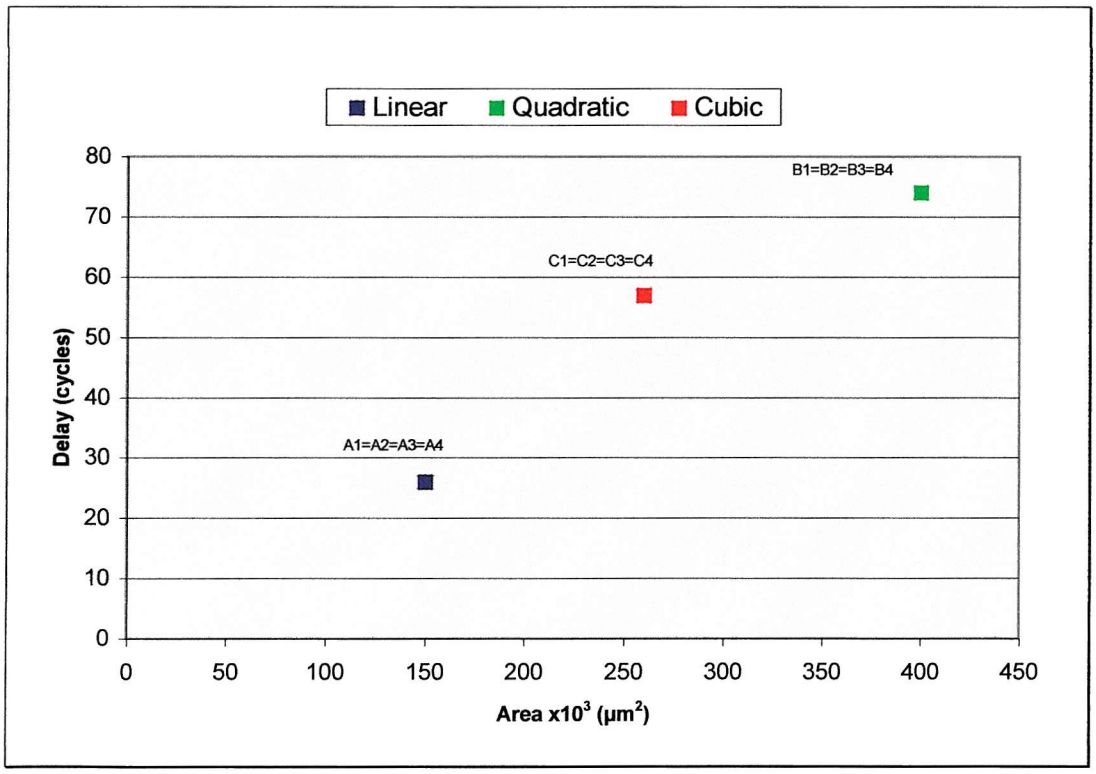


Figure 4.8 Area/delay costs for different interpolation and infinite external ROM

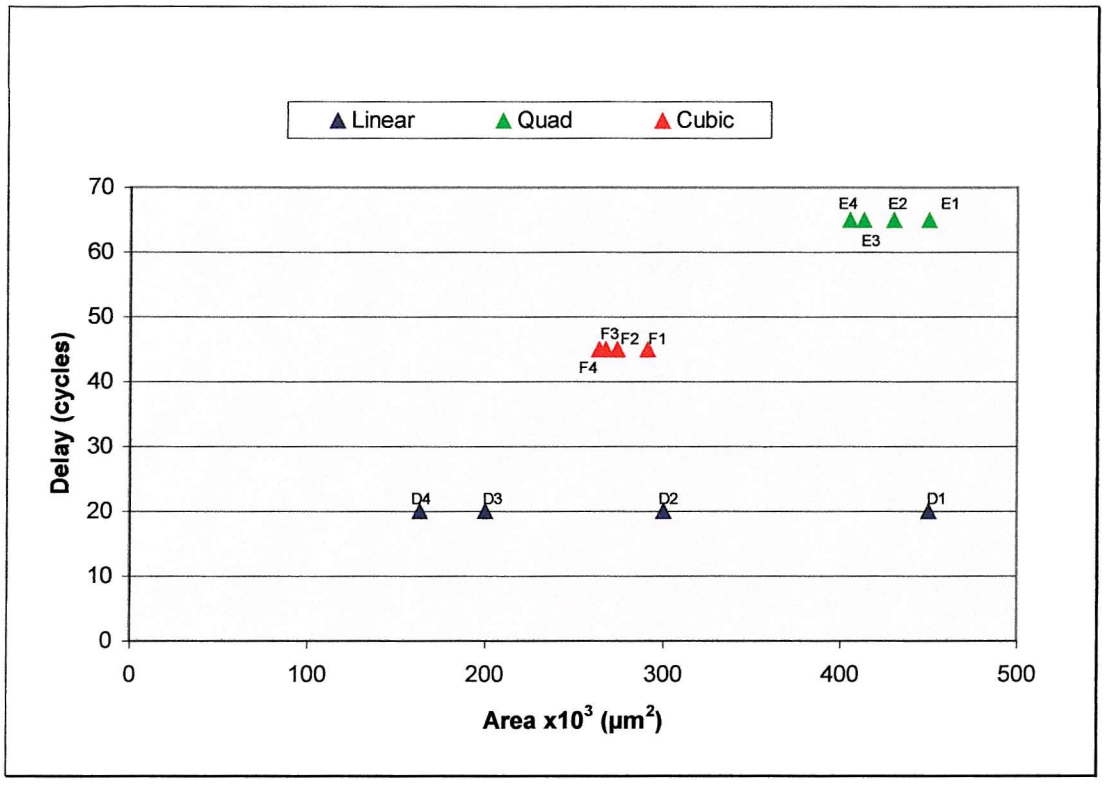


Figure 4.9 Area/delay costs for different interpolation without external ROM

Non-linear table lookup

The table size can be further reduced with negligible degradation in the function evaluation unit performance by observing the linearity of the function over the evaluation interval [81]. This allows partitioning the table into multiple sub-tables, each handling a separate interval of the function. This approach allows modifying the scaling factor of each sub-table depending on the linearity of each partition. Thus, a region where the function is linear can be tabulated with fewer break points than a region where the function is non-linear and still achieve the same accuracy.

To illustrate the advantage of table partitioning, let us consider the inverse sine function (arcsin(x)) in the interval $0 \leq x \leq 1$. Achieving an accuracy of $1e^{-6}$ requires a scaling factor of 2^{-20}, which requires a table size of 1048576 entries. However, this scaling factor is only required as $x \rightarrow 1$: partitioning the table into multiple sub-tables reduces the table size to 2796 entries and still achieves the same accuracy as shown in Figure 4.10.

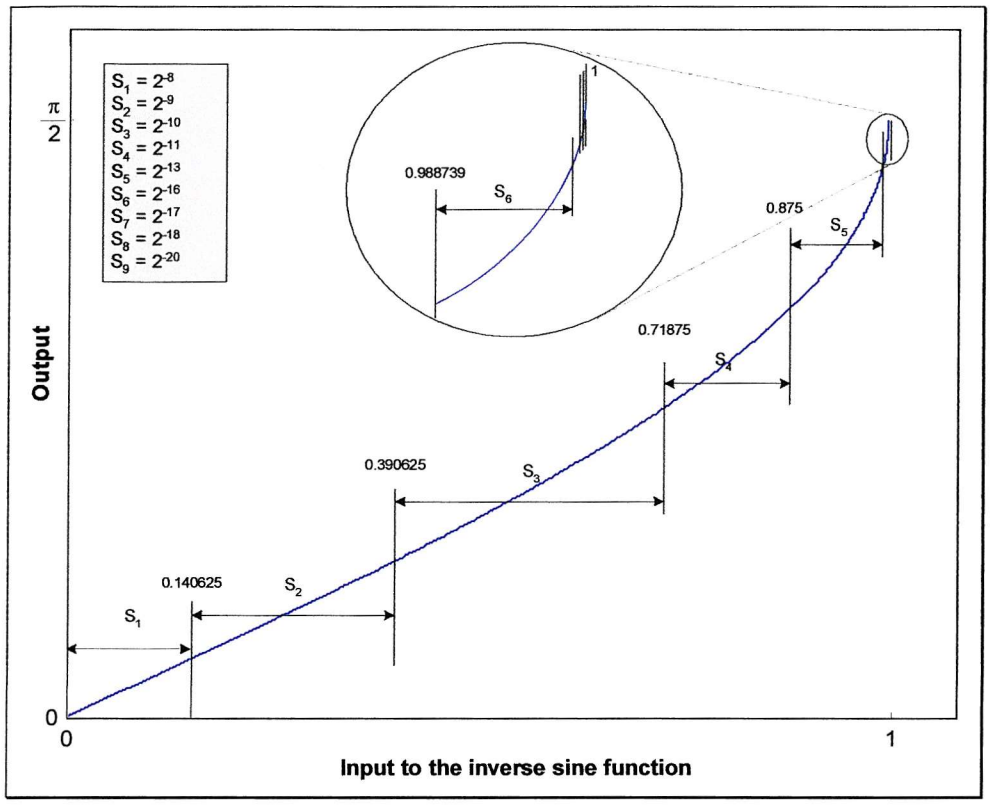


Figure 4.10 Partitioning the inverse sine function into sub-tables

Applying the table partitioning method requires a minor modification to the linear interpolation procedure represented in Figure 4.4 in order to provide a means of identifying the required sub-table. The modified procedure is listed in Figure 4.11.

Note that if the function can be divided into a number of equal intervals, each handled by a separate sub-table, then the comparison operation in Figure 4.11 may be replaced by a single operation:

$$i = \text{int}(\frac{input}{R})$$ [2]

where R is the range covered by each sub-table. Having R as a power of 2 simplifies the division operation into a fast shift operation.

Finally, the scaling factor on all previous interpolation procedures is adjusted to be some power of 2, in order to replace the division in the scaling factor operations (when possible) by a fast shift operation.

---

[2] int() is a function that returns an integer value ≤ the input argument

DEFINITIONS:

T_i :first break point in sub-table i

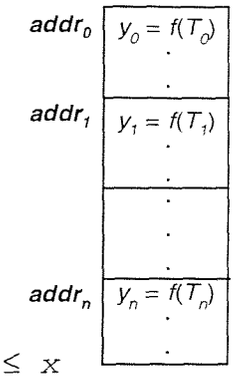
S_i :scaling factor for sub-table i

$Addr_i$:base address of sub-table i

y_i : $f(x_i)$

tmp, m, j :temporary variables

$int(x)$:function returns an integer value $\leq x$



PROCEDURE:

```
if (x <= T0) i = 0;
else if (x <= T1) i = 1;
.
.
else i = n;
```

$$tmp = \left(\frac{x - Ti}{S_i} \right)$$

$$j = \text{int}(tmp)$$

$$i = j + addr_i$$

$$m = tmp - j$$

$$f(x) = y_i + (y_{i+1} - y_i) \times m$$

Figure 4.11 Linear interpolation multiple sub-tables procedure

4.1.3 The CORDIC algorithm

The CORDIC (*Co-Ordinate Rotation DIgital Computer*) algorithm [55, 56, 61, 82] was introduced as the basis for a navigational computer. Its principal advantages are that it requires no multipliers, and can generate two function results simultaneously.

It is an iterative process, applied to a set of input variables (x, y, z) for n iterations, to generate a result accurate to n digits. Each iteration involves a shift, an add and an add constant operation. Each iteration is a rotation of a vector by a defined angle in one of three co-ordinate systems parameterised by m. The basic iteration of the CORDIC algorithm is summarised in Figure 4.12.

DEFINITIONS:

x_0, y_0, z_0 : input operand.

m : =1 for circular, =0 for linear, =-1 for hyperbolic co-ordinate system.

α_i : an angle value stored in a table.

d_n : defines the rotation direction.

PROCEDURE:

```
for (i=0;i<n;i++)
{
```

$$x_{i+1} = x_i - d_i y_i 2^{-i};$$

$$y_{i+1} = d_i x_i 2^{-i} + y_i;$$

$$z_{i+1} = z_i - d_i \alpha_i;$$

```
}
```

Figure 4.12 The CORDIC algorithm

The capabilities of the algorithm are summarised in Figure 4.13, where the input and output values are identified for the three different co-ordinate systems and for two distinct cases: 1) force z to zero, 2) force y to zero. The accuracy of the CORDIC algorithm is largely dependent on the number of iterations [83, 84]. For a large number of iterations, the algorithm delivers a high accuracy as illustrated in Figure 4.14, where the sine function is generated in the range $[0,\pi/2]$ using CORDIC for 25 iterations.

Due to the iterative nature of the CORDIC algorithm, reducing the required accuracy has absolutely no effect on the total area cost[3]. On the other hand, the total delay required to evaluate the function decreases linearly as the target accuracy reduces. This is illustrated in Figure 4.15, where the absolute error is monitored for different numbers of iterations in the same sine function generator.

[3] Unless accuracy reduction is achieved by reducing the datapath size, which might result in increasing the accumulation error and hence increasing the required number of iterations for the required accuracy.

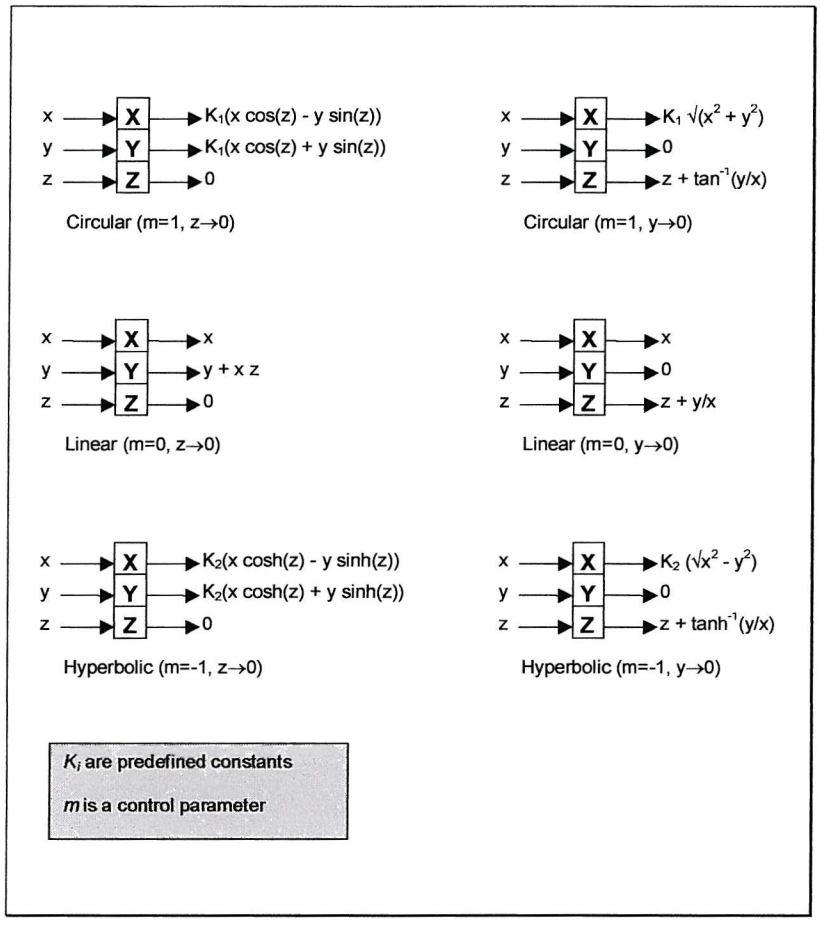


Figure 4.13 Output functions for CORDIC

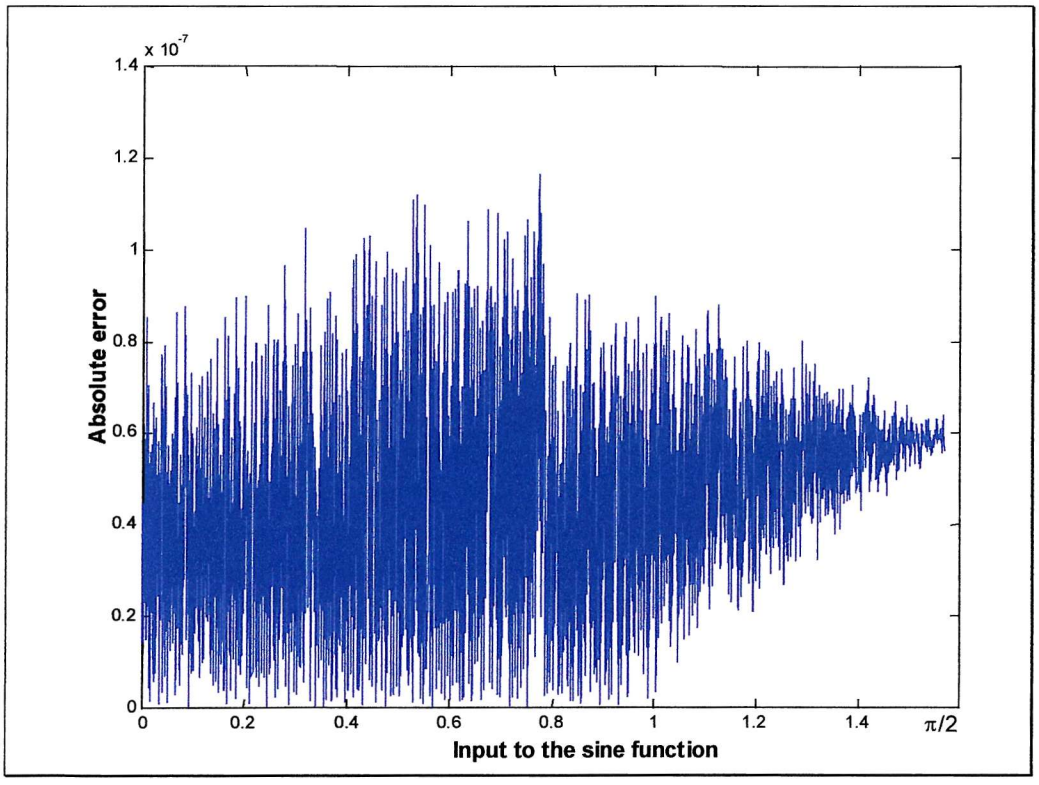


Figure 4.14 Absolute error in the CORDIC sine generator for 25 iterations

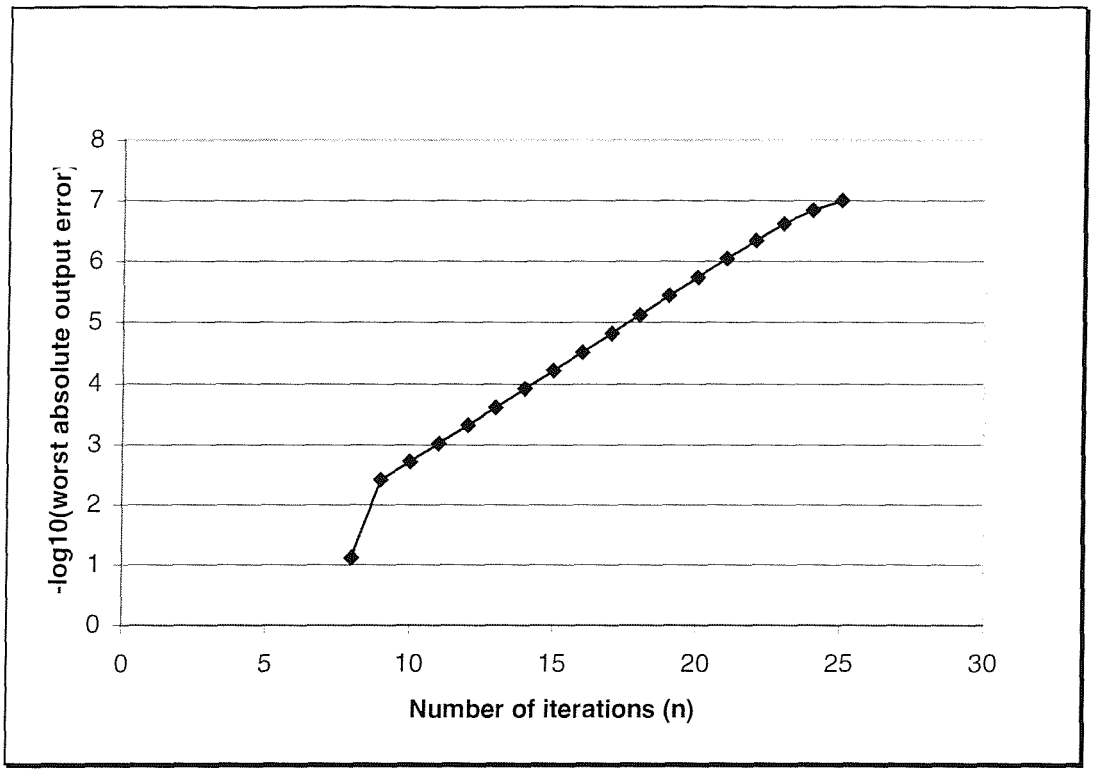


Figure 4.15 CORDIC error variation with the number of iterations

4.1.4 Iterative series

In this method, the value of the function $f(x)$ is provided by an iterative process that calculates a polynomial approximation to the target function. The value of the input operand x is inserted into some formula and after a number of operations the value $f(x)$ is obtained.

A common numerical approximation is the *Taylor series* [60, 85], which is based on the *Taylor theorem* [85, 86]. The algorithm is represented in Figure 4.16. Using this method, the following approximations (amongst others) may be obtained:

$$\sin(x) \cong x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + (-1)^{n-1} \frac{x^{2n-1}}{(2n-1)!}$$

$$\cos(x) \cong 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots + (-1)^n \frac{x^{2n}}{(2n)!}$$

$$\exp(x) \cong 1 + \frac{x}{1!} + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!}$$

DEFINITIONS:

$f(x)$:a function with n+1 derivatives in [a,b].

$f(x)^{(n)}$:the nth derivative of $f(x)$.

x, x_0 :variables in the interval [a,b].

ξ :a value between x, x_0.

$P_n(x)$:approximating polynomial.

$R_{n+1}(x)$:Remainder.

THEOREM:

$$f(x) = P_n(x) + R_{n+1}(x)$$

$$P_n(x) = f(x_0) + \frac{(x - x_0)}{1!} f'(x_0) + \ldots + \frac{(x - x_0)^n}{n!} f^{(n)}(x_0)$$

$$R_{n+1}(x) = \frac{(x - x_0)^{n+1}}{(n+1)!} f^{(n+1)}(\xi)$$

Figure 4.16 Taylor theorem

Another polynomial approximation method is called the *minimax polynomial approximation* [60, 61], which provides an approximation $P(x)$ of a function $f(x)$ that minimises the worst-case error. The minimax approximation can be summarised by the two theorems represented in Figure 4.17[4]. The first theorem says that a continuous function $f(x)$ can be approximated as accurately as desired by a polynomial. The second theorem implies that if a minimax approximation of the n degree is provided to the function $f(x)$, then the largest approximation error is reached at least n+2 times and that the error alternates.

[4] A proof of both theorems can be obtained by consulting [60].

DEFINITIONS:

$||P-f||_\infty$:maximum distance between the approximation and the actual function.

d :variable with a value of ±1.

THEOREM1:

For any $\delta>0$, a polynomial P exists such that :

$$||P-f||_\infty < \delta$$

THEOREM2:

P is the minimax approximation of degree n for $f(x)$ in the interval $[a,b]$ if and only if there are at least n+2 values $a \le x_0 \le x_1 \le \ldots \le x_{n+1} \le b$ such that:

$$P(x_j)-f(x_j) = d(-1)^j ||P-f||_\infty$$

Figure 4.17 Minimax approximation base theorems

Finding a minimax approximation of a function is not a straightforward process. However, numerical analysis tools such as Maple [87] automatically compute the minimax approximation of a function over a provided interval, and provides the corresponding approximation error.

In general, the minimax approximation provides a more accurate solution compared to a Taylor expansion for a polynomial of similar degree. This is illustrated in Figure 4.18, where the exponential function is approximated using both methods for similar approximation degrees. The error over the approximation range is provided in Figure 4.19 and Figure 4.20. Note the wide variation in ordinate scales.

The example shows that minimax approximation provides better results compared to Taylor's expansion. However, the minimax approximation provides unique polynomials for each different degree that requires pre-computing. This gives the Taylor expansion an edge when a variable precision unit is implemented (see Chapter 7), since it is not possible to pre-compute the minimax approximations for every possible precision.

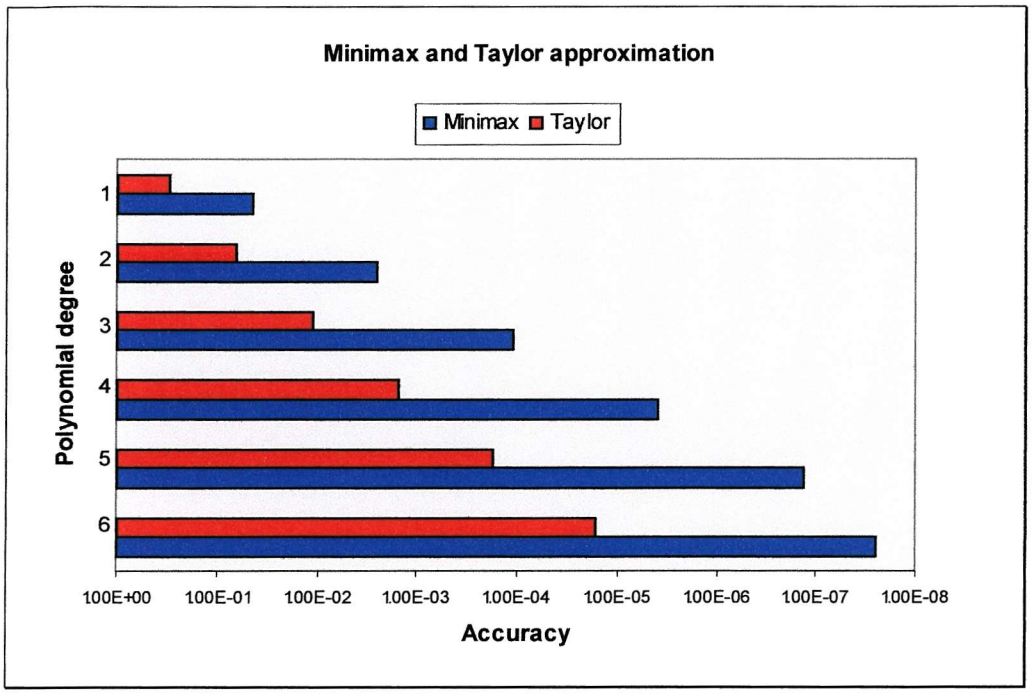


Figure 4.18 Comparison between minimax and Taylor accuracy for different interpolation degrees

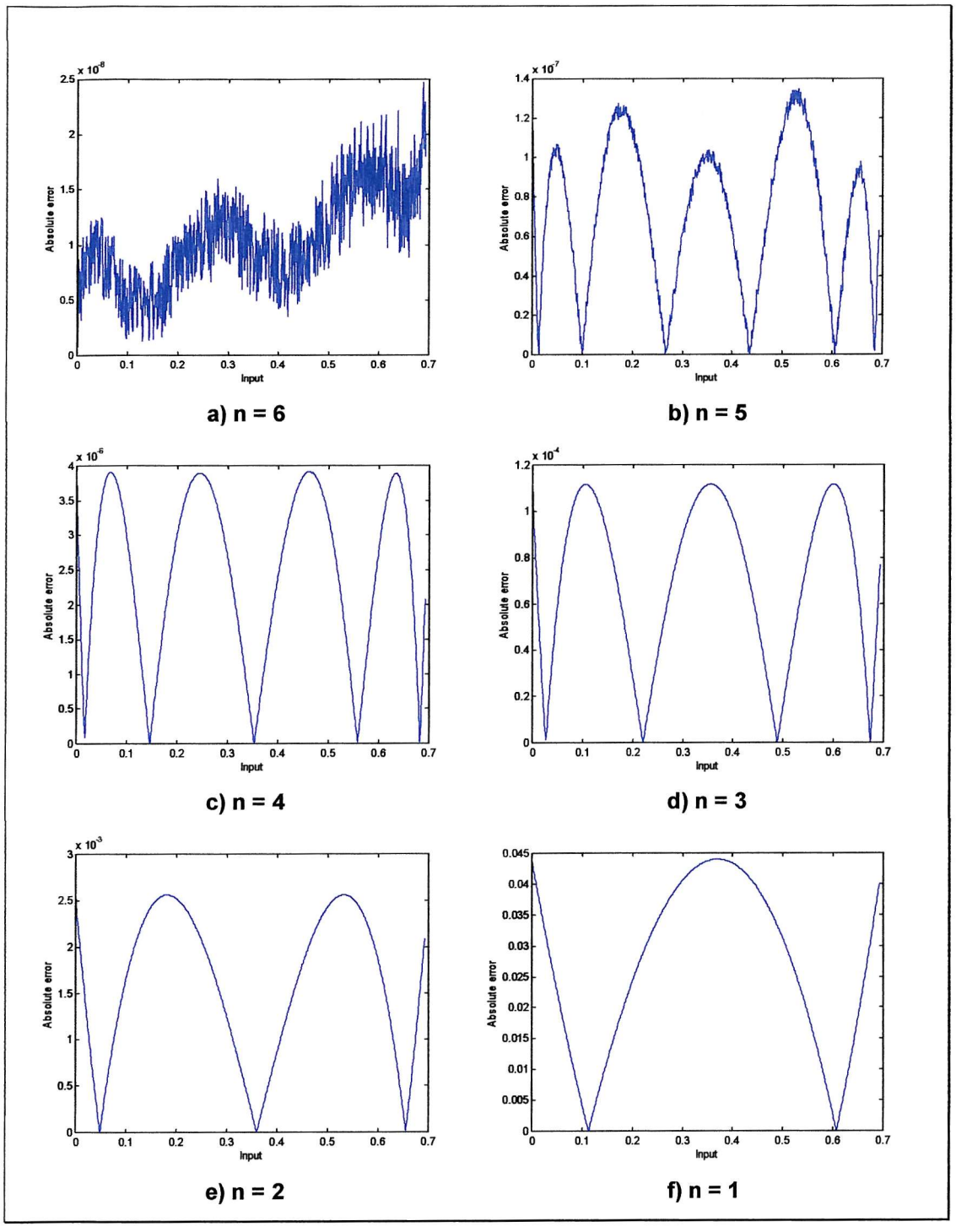


Figure 4.19 Absolute error in the minimax approximation for the exponential function different approximation degrees

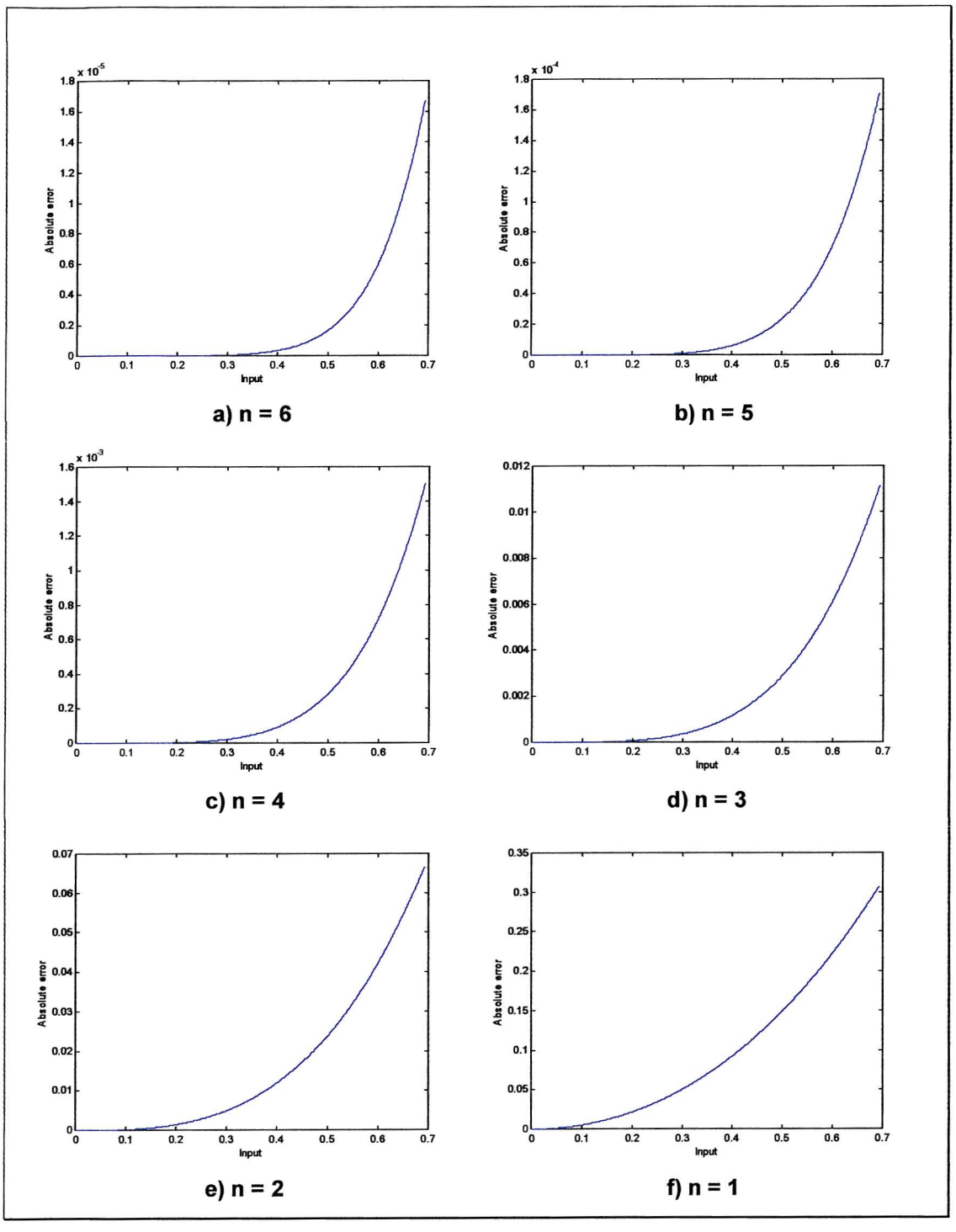


Figure 4.20 Absolute error in the Taylor expansion for the exponential function for different approximation degrees

4.1.5 Post evaluation

At this stage, the final output is adjusted to comply with the IEEE 754 floating-point standard. This involves:

1. Inverting any range reduction effect.

2. Normalising the fraction by shifting and adjusting the exponent field.

3. Rounding the fraction by conditionally adding one to the least significant bit.

4. Supporting any special action to indicate unusual events (e.g. overflow).

Inverting range reduction effects can be simply demonstrated by an example: the inverse tangent function is generated for input operands with a magnitude greater than one using the conversion:

$$\arctan(x) = \frac{\pi}{2} - \arctan\left(\frac{1}{x}\right)$$

The function generator creates the inverse tangent of $(1/x)$ and the final subtraction is performed in the post evaluation stage.

The implicit one in the floating-point representation requires normalising the fraction field, which is simply achieved by shifting the fraction and adjusting the exponent to have the fraction within the range $1 \leq f < 2$.

Rounding is required since the result in most situations cannot be represented exactly in the destination format (23-bit fraction field). In this case, the unit executes in *round to the nearest* mode, which is the default rounding mode in the IEEE standard [88]. Other rounding modes are discussed in Appendix A. In this mode, the result is rounded to the closest representation that fits in the destination format. If a result is exactly half way between two representations, it is rounded to the representation that has a zero least significant bit. Figure 4.21 illustrates three examples of rounding to the nearest, the first result X1 is to the nearest representation *a*, while X2 is rounded to *b*. X3 represents a

special case since it lies half way between c and d, therefore it is rounded to the representation that has a least significant bit of zero (d).

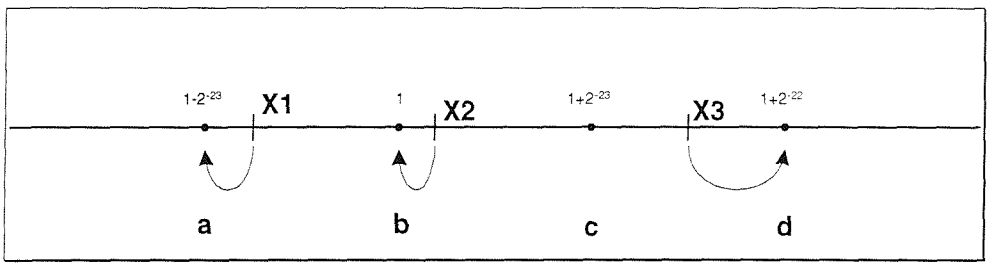


Figure 4.21 Round to the nearest example

Finally, some unusual event may occur during the operation execution that should be handled in the post evaluation stage. A good example for such situation is *overflow*. If the final result of an operation has a magnitude greater than or equal to 2^{128}, the value cannot be represented in the target format and the operation overflows. The post evaluation stage reacts to such situation by outputting a correctly signed infinity symbol and setting the overflow flag. Further details on the post evaluation stage of different floating-point units are available in Appendix C.

4.2 The status register

Each floating-point functional unit has a set of status flags indicating the "goodness" of the output value. Writing to a status flag is analogous to throwing an exception. Each functional unit in the floating-point library can generate six status flags. These are:

1. ***Invalid operation flag***: is set high when an input operand is invalid for the target operation (for example ln(-1)).

2. ***Overflow flag***: indicates that the final result has a magnitude greater than or equal to 2^{128}. The result in such a situation is a correctly signed infinity.

3. ***Underflow flag***: indicates that the final result has a magnitude greater than zero, but cannot be represented by the target format The result will be a correctly signed zero.

4. ***Inexact***: the flag is high if the final result of an operation does not equal to the infinitely precise result. This occurs in one of two situations: either the final result is rounded, or the final result is an approximation of the actual result.

5. ***Not A number flag***: the flag is high if the operation produces NAN as the final result.

6. ***Zero division flag***: the flag is high if the divisor in a division operation is zero.

Type detection blocks, integrated within the floating-point units, to detect these exceptions and output the corresponding flag register are discussed in Appendix C.

Handling exceptions written to the status register is the responsibility of the designer. Two options are available:

- A single status register per floating-point operation: The user can enable this option by providing a variable as an output argument within the floating-point function call (for example sin (input, output, monitor);), in that case, any exception will be signalled by writing the internal status register value to the provided variable. It is then the responsibility of the designer to provide an exception handling process that checks the monitor variable state and provides an appropriate reaction (similar to the C++ *try* and *catch* block).

- A global status register: if it was the designer's decision to ignore the status flag during floating-point calculation, a global port is automatically created as an output port and is shared among the floating-point operations within the process. In this case, handling the design exception should be performed externally (by interrogating (and, if necessary, resetting) the register with an independent process).

Note that raising a flag within the status register does not always indicate a hazardous situation. This is illustrated in the example in Figure 4.22 where arctan(+∞) evaluates to $\pi/2$ and the final result after the multiplication by 2 is correct. However, the divide by zero operation signals a zero division flag.

$$\arccos(x) = 2\arctan\sqrt{\frac{1-x}{1+x}}$$

$$\arccos(-1) = 2\arctan\sqrt{\frac{2}{0}}$$

$$= 2\arctan(+\infty)$$

$$= \pi$$

Figure 4.22 Raising a status flag example

4.3 Supported functions

The floating-point modules currently supported are listed in Table 4.3. A subset of the floating-point modules has the capability of handling complex operands. The complex subset has been chosen to match the IEEE math_real and math_complex VHDL standard [89]. However, the system provides the capability of adding new floating-point and complex modules as high-level functions, which are easily integrated within the floating-point design flow. More details on implementing new floating-point and complex functions may be found in Appendix D. In the following section an introduction to the real floating-point component is provided, which is followed by an explanation of the conversion functions provided, and finally the extension to complex operators is introduced.

Function	Real			Complex		
	Table	CORDIC	Series	Table	CORDIC	Series
addition	*			*		
subtraction	*			*		
multiplication	*			*		
division	*			*		
ln(z)	Y	Y	N	Y	Y	N
$\log_{10}(z)$	Y	Y	N	Y	Y	N
$\log_2(z)$	Y	Y	N	Y	Y	N
$\log_n(z)$	Y	Y	N	Y	Y	N
sin(z)	Y	Y	Y	Y	Y	Y
cos(z)	Y	Y	Y	Y	Y	Y
tan(z)	Y	Y	Y	N	N	N
arcsin(z)	N	Y	Y	N	N	N
arccos(z)	N	Y	Y	N	N	N
arctan(z)	Y	Y	Y	N	N	N
sinh(z)	Y	Y	N	Y	Y	Y
cosh(z)	Y	Y	N	Y	Y	Y
tanh(z)	Y	Y	N	N	N	N
arcsinh(z)	Y	Y	N	N	N	N
arccosh(z)	Y	Y	N	N	N	N
arctanh(z)	Y	Y	N	N	N	N
e^z	Y	Y	N	Y	Y	Y
n^z	Y	Y	N	Y	Y	Y
sqrt(z)	Y	Y	N	Y	Y	Y
conj(z)	**			*		
real(z)	**			*		
imag(z)	**			*		
magn(z)	**			*		
arg(z)	**			*		
complex_to_polar(z)	N/A			Y	Y	Y
polar_to_complex(z)	N/A			Y	Y	Y
to_float( )	*			N/A		
To_complex( )	N/A			*		

* These operations are implemented with separate functional unit unrelated to the three main techniques.

** These return trivial results.

Table 4.3 Floating-point function library

4.3.1 Algebraic operations

This group of floating-point operations performs floating-point addition, subtraction, multiplication, and division of two real operands represented in the IEEE single precision floating-point standard.

Floating-point addition and subtraction

This model performs floating-point addition and subtraction of two floating-point numbers. The inputs to the model are two floating-point numbers *a* and *b*, and a flag to indicate one of the two operations *add* or *subtract*. The outputs of the model are the results of the operation and the status flags.

Defining a floating point number as $F \times 2^E$, the floating-point addition/subtraction operation comprises the following individual operations [48, 54]:

1. *Exponent subtraction*: Perform subtraction of the exponents to form the absolute difference $|E_a - E_b| = d$.
2. *Alignment:* Right shift the fraction (F) of the smaller operand by d bits. The larger exponent is denoted E_f.
3. *Fraction addition*: Perform addition or subtraction according to the effective operation, which is a function of the opcode (add/sub) and the sign of the operands.
4. *Conversion*: Convert the fraction result, when negative to a sign magnitude representation.
5. *Leading*-one *detection*: Determine the amount of left shifts needed in the case of subtraction yielding cancellation. For addition, determine whether or not 1-bit shift right is required.
6. *Normalisation*: Normalise the fraction and update E_f.
7. *Rounding*: Round the final result by conditionally adding 1 to the lsb as required by the IEEE standard. If the rounding causes overflow, perform a 1-bit shift right and increment E_f.

Note that the sign of the exponent difference in step 1 determines which of the two operands is larger. By swapping the operands such that the smaller operand is always subtracted from the larger operand, the conversion in step 4 is eliminated in all cases except for equal exponents. In the case of equal exponents, it is possible to get a negative result in step 3. Only in this event a conversion step is required, but since there is no need for an initial alignment shift in such case, the result subtraction will be exact and there will be no rounding [90].

Note that additional functionality is added to deal with different forms of a floating-point numbers as required by the IEEE standard. The following table outlines these special cases and shows the status flag register in each case.

Case	Result	Status flag register					
		Invalid	Inexact	NAN	OVF	EUN	ZD
(+∞) + (-∞)	Quiet NAN	1	1	0	0	0	0
(-∞) + (+∞)	Quiet NAN	1	1	0	0	0	0
(+∞) - (+∞)	Quiet NAN	1	1	0	0	0	0
(-∞) - (-∞)	Quiet NAN	1	1	0	0	0	0
(+∞) + (+∞)	+∞	0	0	0	0	0	0
(-∞) + (-∞)	-∞	0	0	0	0	0	0
(+∞) - (-∞)	+∞	0	0	0	0	0	0
(-∞) - (+∞)	-∞	0	0	0	0	0	0
Signalling NAN operand	Quiet NAN	1	0	1	0	0	0
Quiet NAN operand	Quiet NAN	0	0	1	0	0	0
Exponent overflow	+/- ∞	0	1	0	1	0	0
Exponent underflow	+/- 0	0	1	0	0	1	0
Result ≠ Infinite precise result	Result	0	1	0	0	0	0
Final result is zero	+/- 0	0	0	0	0	0	0

Table 4.4 Special cases in floating-point addition

Floating-point multiplication

This model performs multiplication of two floating-point numbers provided as input operands. The outputs of the model are the results of the operation and the status flags.

There are five major operations associated with floating-point multiplication [88, 91]:

1. *Initial stage*: Check for zero operands and set the product sign.
2. *Fraction multiplication*: Fixed-point multiplication is performed on the fractions.
3. *Exponent addition*: The two exponents are added. The exponent bias shall be subtracted from result to get the final exponent E_f.
4. *Normalisation*: Normalise the fraction and update E_f.
5. *Rounding*: Round the final result by conditionally adding 1 to the lsb as required by the IEEE standard. If the rounding causes overflow, perform a 1 bit shift right and increment E_f.

The steps of fraction multiplication and exponent addition can be executed simultaneously. However, these two parallel steps must be properly synchronised before the normalisation step is initiated.

The multiplier requires additional functionality to support different forms of a floating-point number, as required by the IEEE standard. Those are listed in the following table.

Case	Result	Flag register					
		Invalid	Inexact	NAN	OVF	EUN	ZD
(+0) x (-∞)	Quiet NAN	1	1	0	0	0	0
(+0) x (+∞)	Quiet NAN	1	1	0	0	0	0
(-0) x (+∞)	Quiet NAN	1	1	0	0	0	0
(-0) x (-∞)	Quiet NAN	1	1	0	0	0	0
Signalling NAN operand	Quiet NAN	1	0	1	0	0	0
Quiet NAN operand	Quiet NAN	0	0	1	0	0	0
Exponent overflow	+/- ∞	0	1	0	1	0	0
Exponent underflow	+/- 0	0	1	0	0	1	0
Result ≠ Infinite precise result	Result	0	1	0	0	0	0
Final result is zero	+/- 0	0	0	0	0	0	0

Table 4.5 Special cases in floating-point multiplication

Floating-point division

There are five major operations associated with floating-point division [48, 54, 92, 93, 94]:

1. *Initial stage*: Check for zero operands and set the product sign.

2. *Dividend alignment*: This is an overflow prevention operation, ensuring that the dividend fraction is smaller than the divisor fraction.

3. *Fraction Division*: Fixed-point division is performed on the fractions.

4. *Exponent subtraction*: The two exponents are subtracted. The exponent bias shall be added to the result to get the final exponent E_f.

5. *Rounding*: Round the final result by conditionally adding 1 to the lsb as required by the IEEE standard. If the rounding causes overflow, perform a 1 bit shift right and increment E_f.

The alignment stage always results in a normalised quotient, so there is no need for a normalisation stage.

The divider requires additional functionality to support different forms of a floating-point number, as required by the IEEE standard. These are listed in the following table.

Case	Result	Flag register					
		Invalid	Inexact	NAN	OVF	EUN	ZD
$(\infty) \div (\infty)$	Quiet NAN	1	1	0	0	0	0
$(0) \div (0)$	Quiet NAN	1	1	0	0	0	0
Signalling NAN operand	Quiet NAN	1	0	1	0	0	0
Quiet NAN operand	Quiet NAN	0	0	1	0	0	0
Exponent overflow	+/- ∞	0	1	0	1	0	0
Exponent underflow	+/- 0	0	1	0	0	1	0
Result ≠ Infinite precise result	Result	0	1	0	0	0	0
Divisor is zero	+/- ∞	0	0	0	0	0	0

Table 4.6 Special cases in floating-point division

4.3.2 Logarithmic and exponential functions

Four main logarithmic functions are provided. The natural logarithm, base 2 logarithm, base 10 logarithm, and base x logarithm. Each model has a single input, which is floating-point operand (except for base x logarithm, where the base is also provided as an input), and two outputs: the floating-point result and the status flag register. The models are based on generating the natural logarithm function. While the remaining models are generated using the following conversions:

$$\log_2 x = \log_2 e \times \ln x$$

$$\log_{10} x = \log_{10} e \times \ln x$$

$$\log_{base} x = \ln x \times \frac{1}{\ln base}$$

The exponential function along with the power of z function are also provided in the floating-point library. Both models are based on the exponential function (exp), with the power of z function generated using the following conversion [95]:

$$x^z = \exp(z \ln x)$$

Since z is can be any real number, this module can be used to generate the square root and the cubic root functions.

The square root is also provided in the floating-point library. The unit has an additional output port that is set to one when a negative input operand is encountered. In such case, the unit evaluates SQRT (|x|) and the sign bit of the input is simply propagated to the flag that indicates a complex result. When this flag is asserted high, it indicates an output of the form: $\text{Result} = j\sqrt{|X|}$

4.3.3 Trigonometric functions

This group of functions consists of the sine, cosine and tangent functions, along with their inverses. The input angle in all these functions is defined in radians. The modules are based on generating the sine function after a range reduction process and then applying simple conversion procedures to implement both the cosine and tangent functions.

The inverse trigonometric functions on the other hand are supported using two modules. The first one generates the inverse sine or inverse cosine of an input argument in the range [-1,1]. The second module implements the inverse tangent function.

4.3.4 Hyperbolic functions

Range reduction for these functions is very expensive in terms of hardware and delay. Therefore, these functions are built upon the elementary functions discussed before as shown in the equations in Figure 4.23 [56, 96]:

$$\sinh x = \frac{\exp(x) - \exp(-x)}{2} \qquad \cosh x = \frac{\exp(x) + \exp(-x)}{2}$$

$$\tanh x = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

$$\sinh^{-1} x = \ln\left(x + \sqrt{x^2 + 1}\right) \qquad \cosh^{-1} x = \ln\left(x + \sqrt{x^2 - 1}\right)$$

$$\tanh^{-1} x = \frac{\ln\left(\frac{1+x}{1-x}\right)}{2}$$

Figure 4.23 Hyperbolic function evaluation equations

4.3.5 Type conversion functions

The VHDL math_real and math_complex [89] provides three data types to represent the floating-point number. A type for real numbers called REAL and two complex data types COMPLEX and COMPLEX_POLAR. The standard is currently provided as a simulation modelling library with no synthesis in mind. This introduces a problem when we try to provide modules to manipulate floating-point variables for synthesis purposes. To tackle this problem, three new data types are introduced to denote floating point and complex variables:

- FLOAT: Represents a 32-bit floating-point number in the IEEE single precision format, and is used to represent real numbers.
- CMPLX: Consists of two 32-bit floating-point numbers in the IEEE single precision format and is used to represent complex variables in the form x+jy.
- CMPLX_POLAR: Consists of two 32-bit floating-point numbers in the IEEE single precision format and is used to represent complex variables in the form $Re^{j\theta}$.

Note that since the real and imaginary parts in the two complex types are represented as two floating-point numbers, the same rules that handle the status register flags in the float core apply here.

A set of type conversion functions is also provided to convert between complex type and from complex to real and vice versa:

- **CONJ (Z)**: The function returns the conjugate of a complex and complex polar variable. If the input argument is a real number, an overloaded function with a real input is used, and the same input is propagated to the output

- **REAL (Z)**: The function returns the real part of a complex variable. For a real input the output is the same as the input variable.

- **IMAG (Z)**: The function returns the complex part of a complex variable. For a real input argument the function outputs zero.

- **MAGN (Z)**: The function returns the magnitude of a complex polar variable. For a real input the output is the same as the input variable.

- **ARG (Z)**: The function returns the angle of a complex polar variable. For a real input argument the output equals zero.

- **COMPLEX_TO_POLAR (Z)**: The function converts a complex input argument to a complex polar variable.

- **POLAR_TO_COMPLEX (Z)**: The function converts a complex polar input argument to a complex variable.

Two additional type changing functions (*to_float( )*, *to_complex(,)*) are also provided to support translation from a VHDL type *real* and *integer* to the IEEE single precision representation of *float* and *complex*.

4.3.5 Complex units

The type conversion functions illustrated earlier, along with the floating-point library components are used to implement the complex functional units within the synthesis library. These units are based on a hierarchical decomposition of floating-point functional units that manipulate the real and the imaginary parts of the two complex types (complx and complx_polar). By way of an example, let us consider evaluating the sine function of a complex variable based on the following equation:

$$\sin(x+jy) = \sin(x)\times\cosh(y) + j\cos(x)\times\sinh(y)$$

The functional unit block diagram is shown in Figure 4.24. The complex variable is split into its two floating-point components (real and imaginary) and passes through a number of floating-point functional units to generate the final result.

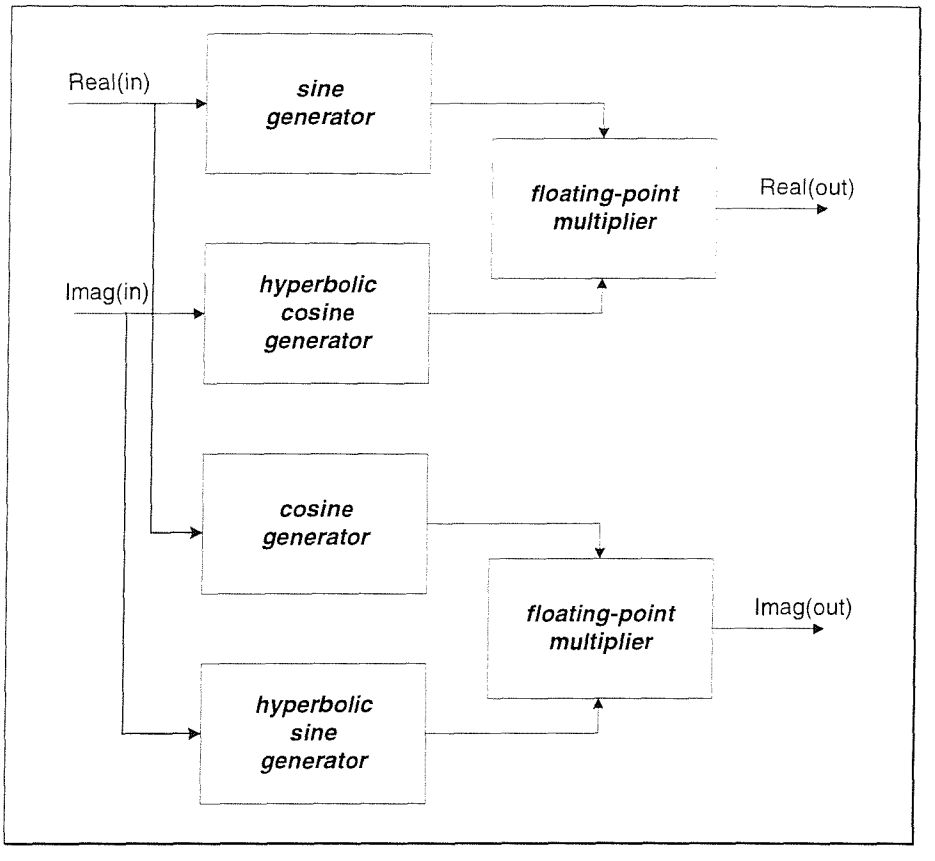


Figure 4.24 Complex sine function generator building blocks

For the polar type, the sine function generator is based on the complex sine function generator as illustrated in Figure 4.25. The polar variable is initially converted into the equivalent complex representation using the complex_to_polar function. A complex sine function generator follows this and the output result is then transferred back into the polar representation using polar_to_complex functional unit.

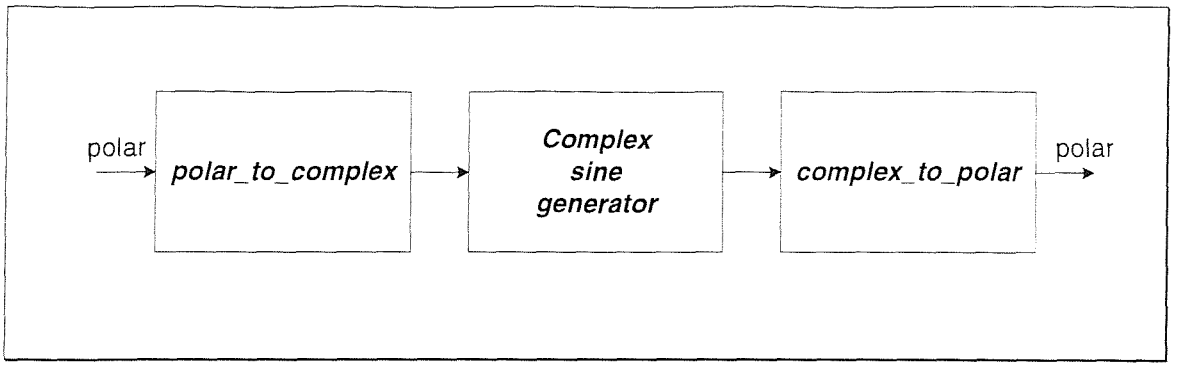


Figure 4.25 Polar sine function generator building blocks

The rest of the complex components are implemented in a similar manner to the sine function based on the set of equations listed in Figure 4.26.

$$(x1+jy1)\times(x2+jy2)=(x1x2-y2y2)+j(x1y2+x2y2)$$

$$\frac{x1+jy1}{x2+jy2}=\frac{x1x2+y1y2}{x2^2+y2^2}+j\frac{x1y2-x2y1}{x2^2+y2^2}$$

$$\exp(x+jy)=\exp(x)\times\cos(y)+j\exp(x)\times\sin(y)$$

$$\cos(x+jy)=\cos(x)\times\cosh(y)-j\sin(x)\times\sinh(y)$$

$$\sinh(x+jy)=\sinh(x)\times\cos(y)+j\cosh(x)\times\sin(y)$$

$$\cosh(x+jy)=\cosh(x)\times\cos(y)+j\sinh(x)\times\sin(y)$$

$$(r1e^{j\theta1})\times(r2e^{j\theta2})=r1r2e^{j(\theta1+\theta2)}$$

$$\frac{r1e^{j\theta1}}{r2e^{j\theta2}}=\frac{r1}{r2}e^{j(\theta1-\theta2)}$$

$$\sqrt{(re^{j\theta})}=\sqrt{r}e^{j\frac{\theta}{2}}$$

$$\ln(re^{j\theta})=\ln(r)+j(\theta)$$

$$\log_n(re^{j\theta})=\frac{\ln(r)+j(\theta)}{\ln(n)}$$

Figure 4.26 Complex function evaluation equations

4.4 Function implementation

The floating-point library is integrated into the MOODS synthesis system via the expanded module capability. This section describes two major steps in the development of

the floating-point library. Those are the hierarchical unit expansion and expanded module implementation. Further implementation details can be found in Appendix D.

4.4.1 Hierarchical unit expansion

Many floating-point and complex functional units in the library are provided as a hierarchical structure of common building blocks. This approach allows the final synthesis stage to share the common building blocks of different arithmetic units, which results in a significant reduction of the total area cost. In addition, partitioning the arithmetic units into a number of building blocks allows effective pipelining. This results in a reduction of the total delay and increases the throughput of the whole system. As an example, consider the pseudo-code of Figure 4.27.

In Figure 4.27b, the *sine* function is expanded into two sub-blocks, the range reduction stage (*sin_cos_pre()*), and the function evaluation stage (*sin_cos_main()*). A large number of sub-blocks are common to more than one floating point unit. They communicate with each other by means of (automatically generated) temporary buffers, which are initialised by the system to allow the sub-blocks to know which floating-point unit they are actually representing. For example, *buf1* in Figure 4.27b will be initialised to tell *sin_cos_pre()* it is representing a *sin()*, and *sin_cos_pre()* may write the range reduction details into *buf1* to be picked up by *sin_cos_main()*. The complex type conversion function *polar_to_complex()* is expanded into further building blocks (*sine*, *cosine*, two floating-point multipliers and two type converters) as shown in Figure 4.27c. The *sine* and *cosine* functions are then further expanded (Figure 4.27d). This approach makes it easy for the optimisation algorithm to exploit functional unit duplication. The expansion process involves a series of modification to the original ICODE file that represents the design. Details on the expansion process, along with the modifications performed on the input ICODE file to generate the ICODE+ after expansion are available in Appendix D.

Note that RE() and IM() in Figure 4.27c are similar to PL/1 pseudo functions: if they appear on the right hand side of an assignment, the return a *value*, if they appear on the left hand side, they provide *access*.

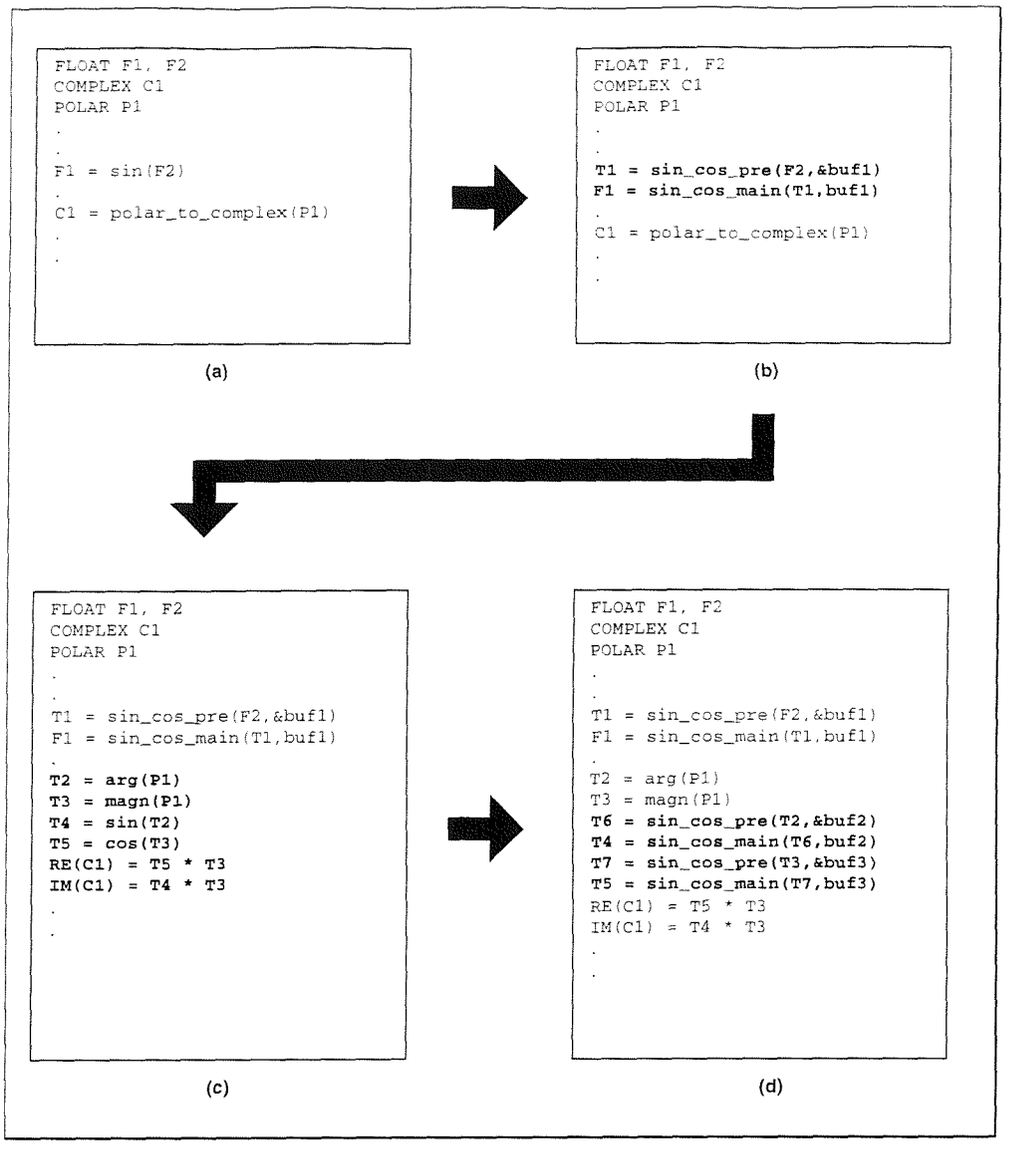


Figure 4.27 Hierarchical unit expansion example

4.4.2 Expanded module formation

The floating-point library building blocks are all implemented as expanded modules which are inline expanded within the MOODS control and datapath graphs during the design synthesis process. Developing an expanded module is a straightforward process. However, certain points of particular interest are described here to ensure the integrity of the generated expanded module.

Figure 4.28 illustrates the expanded module creation data flow. At the highest level, the expanded module is described as a VHDL entity with a single process. At this stage, simulating the VHDL behavioural description is recommended to ensure a correct module operation. An important point here is to remember that the expanded module will act as a

datapath functional unit. This implies that the input ports must be stable during the module execution. As an example consider an instruction such as (c := c + a;) in a behavioural description. Providing a single variable as an input and output port to the same multi-cycle expanded module may result in an incorrect execution as it is not guaranteed that the output port (which is also an input port) will remain stable and will not be updated during the module execution. To solve this problem, an initialising stage within the expanded module is implemented, loading the input variables into internal registers local to the expanded module body before any further manipulation.

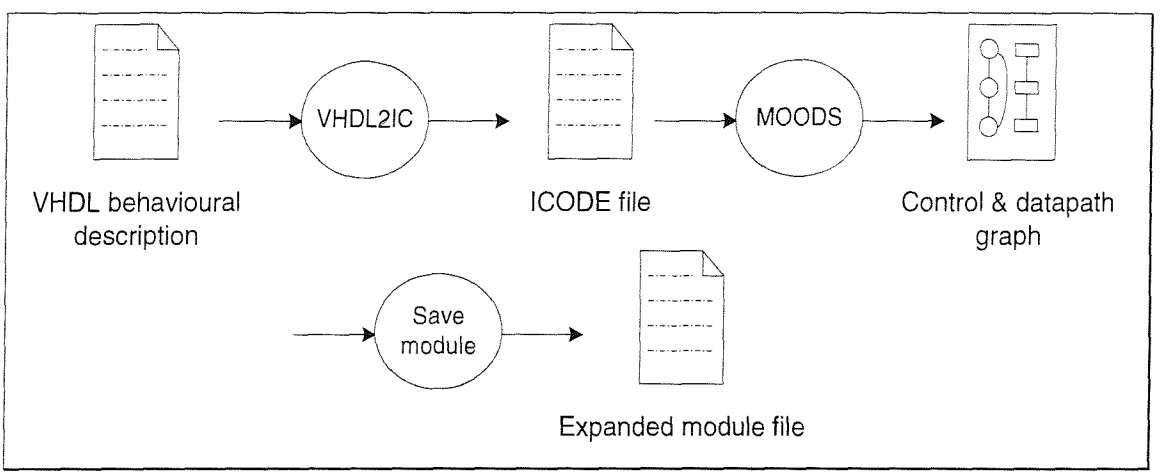


Figure 4.28 Expanded module formation

Once implemented, the VHDL behavioural description is transformed using theVHDL2IC pre-processor into an ICODE file. At this stage, a minor manual modification to the ICODE file is required before moving on in the generation process. The necessity of this manual altering of the ICODE arises from the nature of a VHDL process as an indefinitely repeating loop, which implies that there will always be an activation from the last control state to the first control state to ensure continuous execution. This activation command has to be eliminated manually from the ICODE file to match the nature of the expanded module, which has unique, non-excitable start and end control states. This manual manipulation to the ICODE file can be eliminated provided that the user follows certain guidelines. This is illustrated by the example in Figure 4.29. Figure 4.29(a) shows a simple VHDL process with its equivalent ICODE. Note that two ICODE instructions (3,4) provides a feedback to the first control state. To ensure the integrity of the generated expanded module. The design can be simply modified by assigning the output result to a temporary register in all branches and then assigning the value of this register to the output port at the last instruction (control state), as illustrated in Figure 4.29(b). This ensures that

the process will have only a single activation from the last control state to the first one, which can be deleted automatically by MOODS during the expanded module generation process.

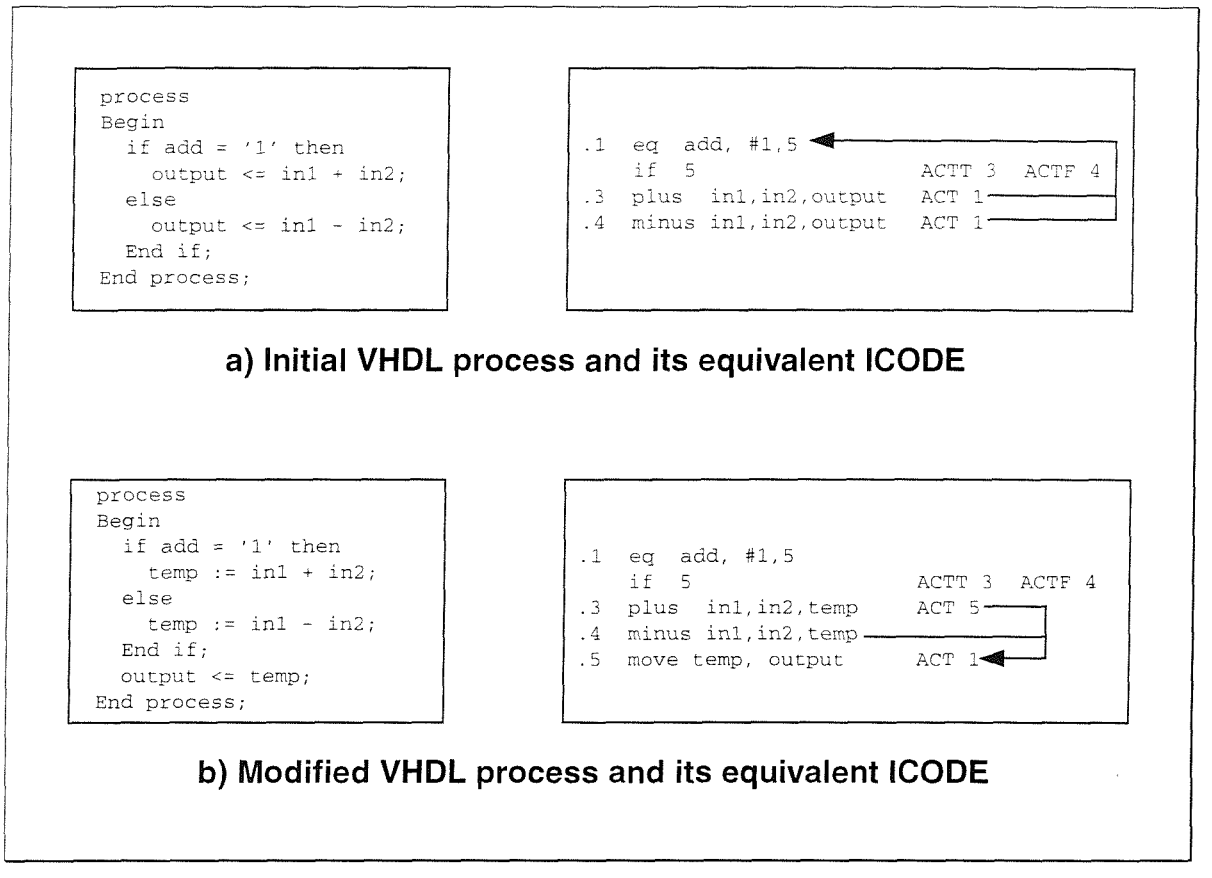


Figure 4.29 Expanded module development example

Once complete, the ICODE file is loaded into the MOODS synthesis system and is transformed into an initial control and datapath graph. Finally, the design is saved as an expanded module file and added to the MOODS floating-point library.

Chapter 5

Floating-point optimisation

The floating-point optimiser operates on floating-point and complex operations within the design, binding each floating-point operation to a suitable base technique component from the floating-point module library.

During optimisation, the high level binding decision of each floating-point unit (i.e. table lookup, iterative series, or CORDIC) takes into account a number of issues such as the type and number of floating-point operations required and the availability and the capacity of any off-chip ROM available to the system.

This chapter details the floating-point optimisation unit. The algorithm evolved from the need to map each floating-point operation to a suitable high level module in a way that enables the main synthesis system to develop designs that meet the user's pre-defined objectives.

The remainder of this chapter is divided into four sections. First, section 5.1 describes the physical interactions that arise from the nature of the high-level floating-point library components and their effects on the optimisation process. Section 5.2 introduces accuracy as a new design space parameter, and describes the way the system handles this issue. Section 5.3 describes the optimisation algorithm and details the results of an extensive analysis of its effectiveness on a number of benchmark designs. Finally, further experimental evaluation of the algorithm is provided in section 5.4.

5.1 Function implementation interactions

The attributes of each function implementation considered in isolation are easy to compare: to generate *sin(x)* with a table requires the table itself (which may be internal, or external, requiring an interface), plus an interpolation engine. To generate it with a series

requires a cumulative adder plus a term generator, which may require a table, but no interpolation engine. All these elements have easily quantifiable area and speed costs. However, when a number of functions are required, new interactions become important. Those interactions are listed in Table 5.1.

1. There is an overhead to interfacing an ASIC/FPGA to an external ROM, but it is fixed and independent of the number of external function tables.
2. Once an iterative series generator has been instantiated, the cost of switching between different functions is relatively small.
3. Some function tables are subsets of others.
4. Once a complex function is implemented, the equivalent real function is virtually free in most cases.
5. Some functions are built as a hierarchical composition of other functional units. If these units are already available, the total cost is reduced.
6. Once a CORDIC unit has been instantiated, the cost of other units based on CORDIC will be reduced.
7. The pre-processing stage of some function generators contains the fixed-point operators (multiplier, divider) required in the function generator block. This reduces the total area cost by sharing these operators within the two blocks.
8. An optimal distribution of the external ROM amongst the floating-point units has a great effect on the total system cost.
9. Providing the exact required accuracy for every functional unit could increase the total area cost.
10. When a floating-point function generator is shared between a large number of functional units, the multiplexing cost could affect the optimised decision in choosing between off-chip and on-chip implementations.

Table 5.1 Function implementation interactions

The cost of interfacing a design to an external ROM is divided into two sources:

1. I/O port cost: includes the cost of the address bus port, the data bus port and the control signal.

2. Control hardware: to control the process of reading data from the external ROM. This involves setting the address and the control signal and then latching the output data into an internal register.

Figure 5.1 shows a block diagram of an external ROM interfacing unit shared between a number of functional units. Using this method, a number of functions using the same external ROM will hardly have any effect on the total system cost when compared to the cost of implementing a single functional unit using an external ROM. The only overhead when the external ROM is shared is the cost of multiplexing the data bus and the address bus between the functional units.

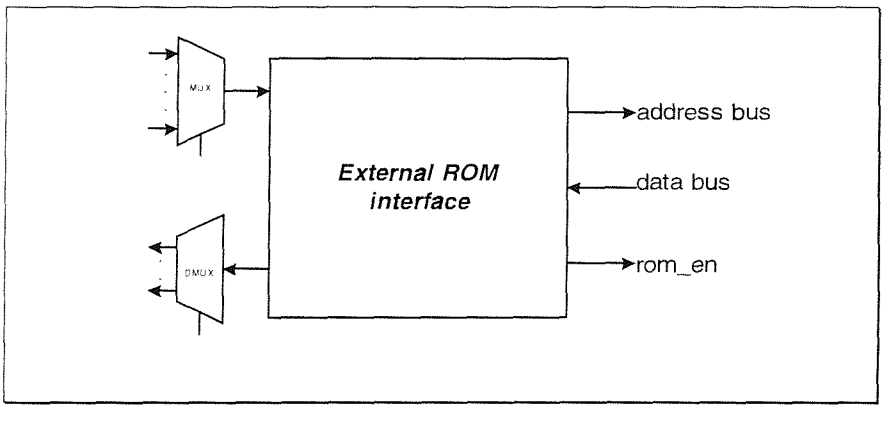


Figure 5.1 Sharing an external ROM interfacing unit

The same discussion above applies to a number of functional units implemented using an iterative series based method. The iterative series engine is an iterative process that performs multiply and add operations on a single input operand for a controlled number of loops. Sharing this unit is achieved by multiplexing four ports: the input operand, the multiply constants, the control variable that decides the number of iterations and the output results. This sharing is visualised in the block diagram in Figure 5.2.

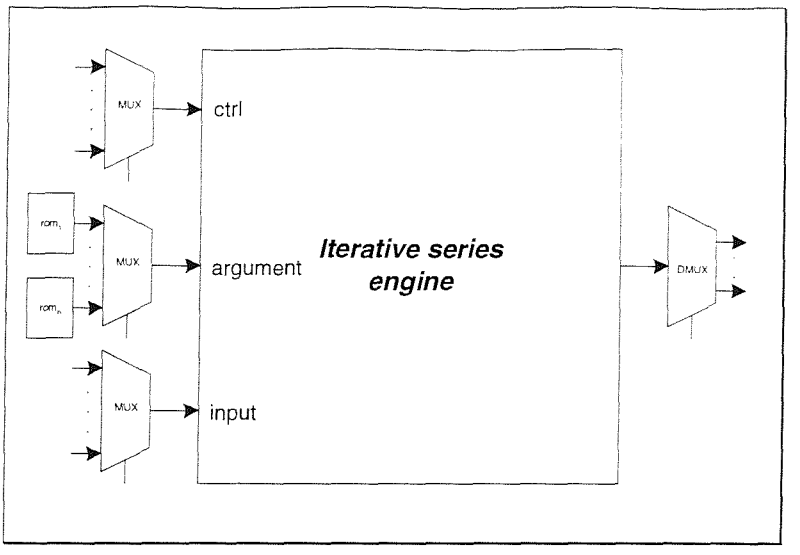


Figure 5.2 Sharing iterative series engine

Table lookup based methods can exploit algebraic identities of certain functions to reduce the total storage area required to store the table. For example: $\cos(x) = \sin(\frac{\pi}{2} - x)$. This allows implementing both functions using a single table that stores the sine function values and the subtraction unit is provided as a pre-processing stage in the case of the cosine function.

Complex variables are represented using two floating-point variables, one to represent the real part and the other to represent the imaginary part. This implies that any real number can be represented using a complex representation with an imaginary part equal to zero. Building blocks used to implement complex functional units can be used to generate the equivalent real function by setting the imaginary part of the input operand to zero.

Some functions in the floating-point library are implemented as a hierarchical decomposition of other floating-point building blocks. The hyperbolic sine is one example, which is based on the exponential function. If an exponential function generator building block is already instantiated in the design, the unit can be used to generate the hyperbolic sine, which results in a major reduction in the total cost of generating the latter function.

The total area cost required to implement a functional unit based on the CORDIC algorithm is dominated by the variable width shift operation and the table of constants. Once a decision is made to implement a functional unit using CORDIC, the cost of instantiating other CORDIC unit is reduced due to the possibility of sharing the shifter and the table of constants.

The Range reduction units in some functional unit generators require fixed-point multiplication and/or fixed-point division operators. Since instantiating these units is a definite requirement for implementing the appropriate functional unit, the same fixed-point operators can be used by the function generator block, which results in a reduction of the total area cost.

The limited capacity of the external ROM available is a major constraint imposed during the optimisation phase. A random assignment of floating-point units to the external ROM, and the nature of the interpolation engine (linear or non-linear table lookup), limits the number of operators that can exploit the external ROM, which results in a great degradation of the design performance, especially when a minimum area cost is required.

A single floating-point functional unit with different target accuracies could be present in different parts of the design datapath. Implementing the required accuracy of each individual unit eliminates the possibility of sharing these units at the highest level of hierarchy. Assigning the highest required accuracy to all similar functional units within the design allows maximal sharing of these units before flattening the design hierarchy.

The multiplexer cost required to share a large number of functional units affects the optimisation decision when comparing the off-chip and on-chip table-lookup based units. The difference in the multiplexing cost in both cases could exceed the area saved by implementing the lookup table as an external ROM.

Diverse interactions such as these require a dedicated optimisation algorithm to perform the high-level module binding. This algorithm is discussed in section 5.3. Further analysis, highlighting the effects of these interactions is provided in section 5.4.

5.2 Numerical interaction

The introduction of a floating-point capability to a synthesis environment gives rise to a new gross design parameter, that is the *accuracy* of the floating-point building blocks that comprise the mathematical expressions within the design. Accuracy cannot be treated on an equal footing with the other dimensions of the design space because the effects of changing the accuracy of a functional unit cannot be localised in most cases, and a change in the accuracy of any module will threaten all operations predicated upon it. Errors

propagate and interact nonlinearly. Furthermore, the form of this interaction is largely data dependent, it is not difficult to construct a process where a change in a component *accuracy* ultimately affects the *behaviour*.

The floating-point processes within the system support user specification of floating point accuracy at two levels: it is possible to assert an overall accuracy on a design, (each individual floating point operation in the design will deliver this accuracy) and it is possible to override this and assign individual accuracies to each floating point operation. Within each hierarchical operator, a differential error propagation model [97, 98] is employed to calculate the necessary accuracies of each of the building blocks. These calculations result in a single figure of merit assigned to each building block indicating its contribution to the total error in the parent operator. These figures are provided as a set of parameters within the file that represents the hierarchical operator. Given the required accuracy of the parent operator, the accuracy of each sub-component is calculated and assigned. When building blocks are shared between operators later by the system, the accuracy of each shared block is promoted to the value of the most accurate, with units based on CORDIC and iterative series being an exception, as they get assigned the exact required accuracy in order to reduce the total delay cost.

In the remaining part of this section, error propagation and the effect of varying accuracy on overall system performance will be discussed.

5.2.1 Error propagation

The differential error propagation model [97, 98, 99, 100] is often used to study the error propagation of a floating-point expression. Any arithmetic expression may be characterised by a *computational graph* composed of directed edges running from input operand nodes to operation nodes, and from operation nodes to the final result node.

Figure 5.3 shows a computational graph of the simple arithmetic expression: $y = 1.0 - \sin(x)$. Each directed edge from a node ξ_i to a node ξ_k is assigned a weight P_{ki}. P_{ki} is an *error propagation factor* reflecting the amount of amplification or damping that occurs on the error of ξ_i ($\rho\xi_i$) while generating ξ_k. Formally, P_{ki} is given by:

$$P_{ki} = \frac{\rho\xi_k}{\rho\xi_i} = \frac{\partial\xi_i}{\partial\xi_k} \times \frac{\xi_k}{\xi_i}$$

The final error in the output result ($\rho\xi_m$) is given by:

$$\rho\xi_m = \sum_{i=1}^{n} \rho\xi_i P_{mi}$$

Applying the previous formula to the mathematical expression given in the example in Figure 5.3 gives a total error in the output result of the form:

$$\rho\xi_4 = \frac{1}{1-\sin(x)}\rho\xi_1 - \frac{x\cos(x)}{1-\sin(x)}\rho\xi_2 - \frac{\sin(x)}{1-\sin(x)}\rho\xi_3$$

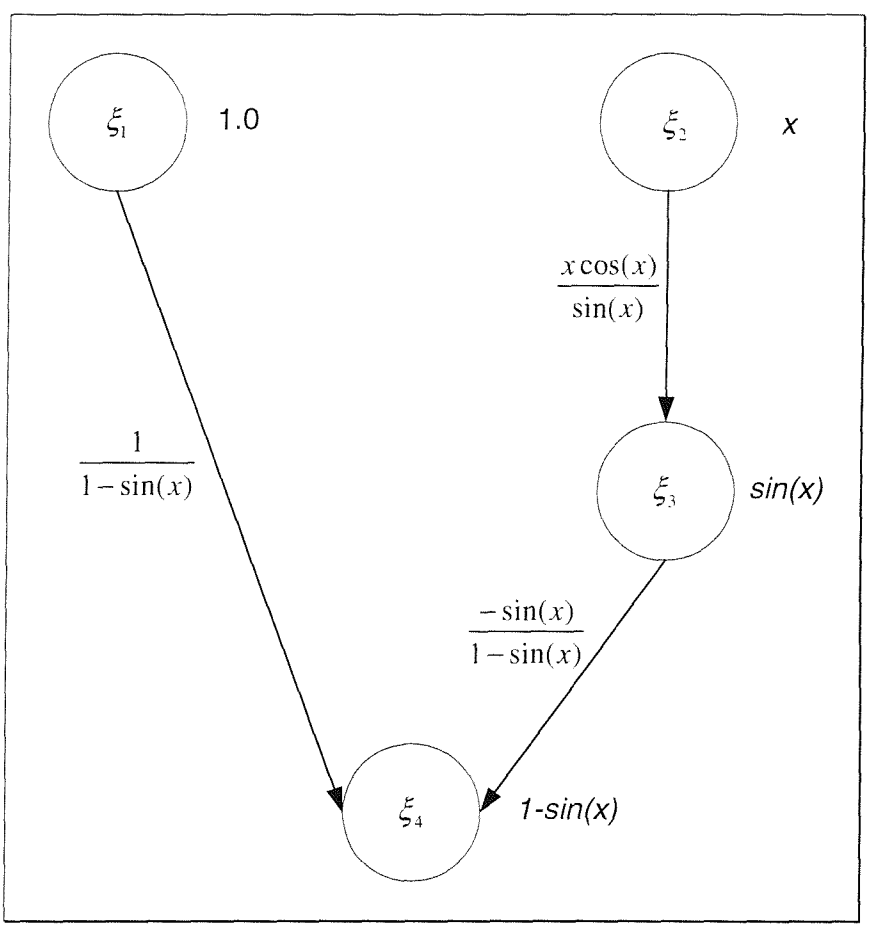


Figure 5.3 Computational graph example

From the final expression of the accumulated error, it is clear that the effect of local operation error on the final accumulated error is largely dependent on the input operand(s)

value. For software mathematical packages [87] an exhaustive approach that involves evaluating the accumulated error expression for every set of input operands is usually employed in order to define the appropriate accuracy of each operation. This is not however a possible solution for a synthesis tool since the hardware is actually implemented for every possible input operand. The approach taken to tackle this problem is to exploit the differential error propagation model to identify the major error sources in an expression and assign the accuracy of the building block in a way that minimises the total error to within the required accuracy (if possible). For example, in Figure 5.3 the error in the sine operation is magnified as x gets near $\pi/2$. Therefore, the sine function should be evaluated to the highest possible accuracy permitted by the module library.

By way of an example, Figure 5.4 shows the error propagation calculation of a simple arithmetic expression ($c = a + b$). For $a = 3$ and $b = 4$, the final result is $c = 7$. Assuming an absolute error of 0.1 in a and 0.2 in b, the absolute error in the final result is $\Delta c = 0.3$, resulting in a relative error of $\rho c = 0.0428$, which is identical of the result of the error propagation model.

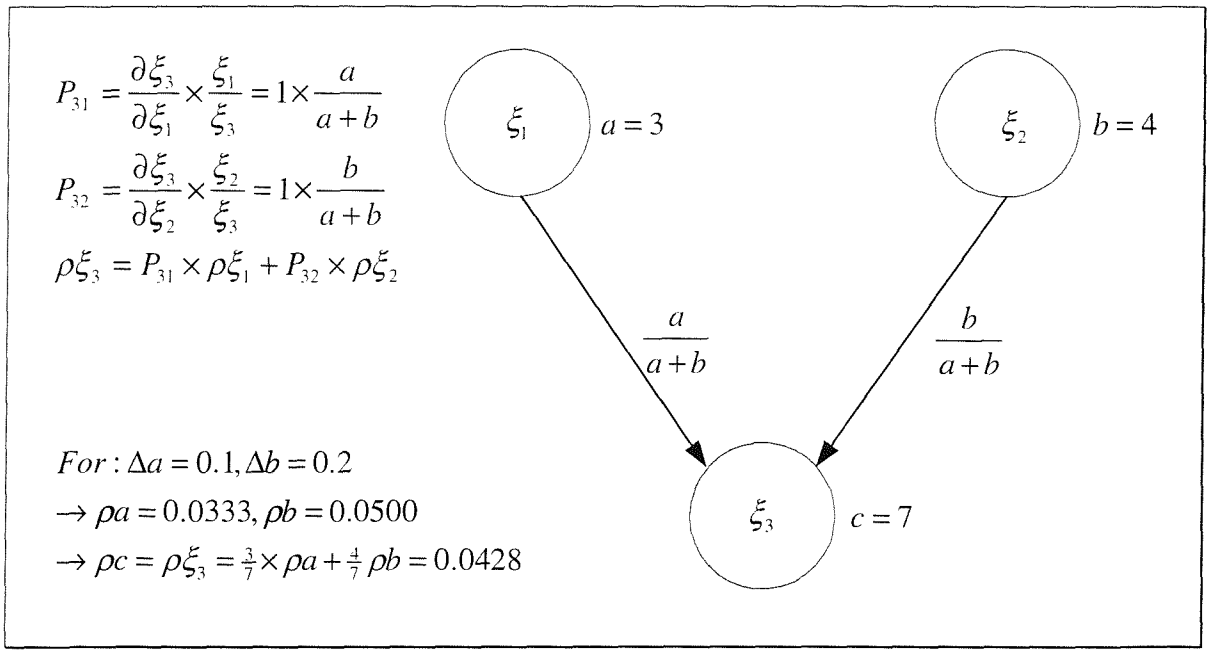


Figure 5.4 Error propagation model example

5.2.2 Accuracy variation effect

The accuracy variation impact on the system parameters is largely dependent on the function evaluation engines invoked within the design. The results presented in this section demonstrate the effect accuracy variation has on the final hardware cost on three behavioural benchmarks incorporating floating-point manipulation.

The original VHDL behavioural description contains six floating-point functions: sine, inverse sine, square root, natural logarithm, exponential, and inverse tangent function. TestA is implemented using a function generator based on an internal table lookup interpolation engine. TestB is a design utilising units based on iterative methods (CORDIC and minimax approximation). Finally, TestC employs a linear interpolation engine based on an external ROM to generate the functions.

Figure 5.5 shows the three benchmarks located in a two-dimensional design space for different target accuracies. Trajectory parameters are given in Table 5.2, where each implementation is given a reference code.

Design	Accuracy = 1e-6		Accuracy = 1e-5		Accuracy=1e-4	
	Area (μm^2)	Delay (cycles)	Area (μm^2)	Delay (cycles)	Area (μm^2)	Delay (cycles)
	Reference		Reference		Reference	
TestA	2.39e6	214	1.14e6	214	837323	214
	A1		A2		A3	
TestB	836199	721	824800	645	817900	572
	B1		B2		B3	
TestC	679513	250	679513	250	679513	250
	C1		C2		C3	

Table 5.2 Area and delay figures for various configurations

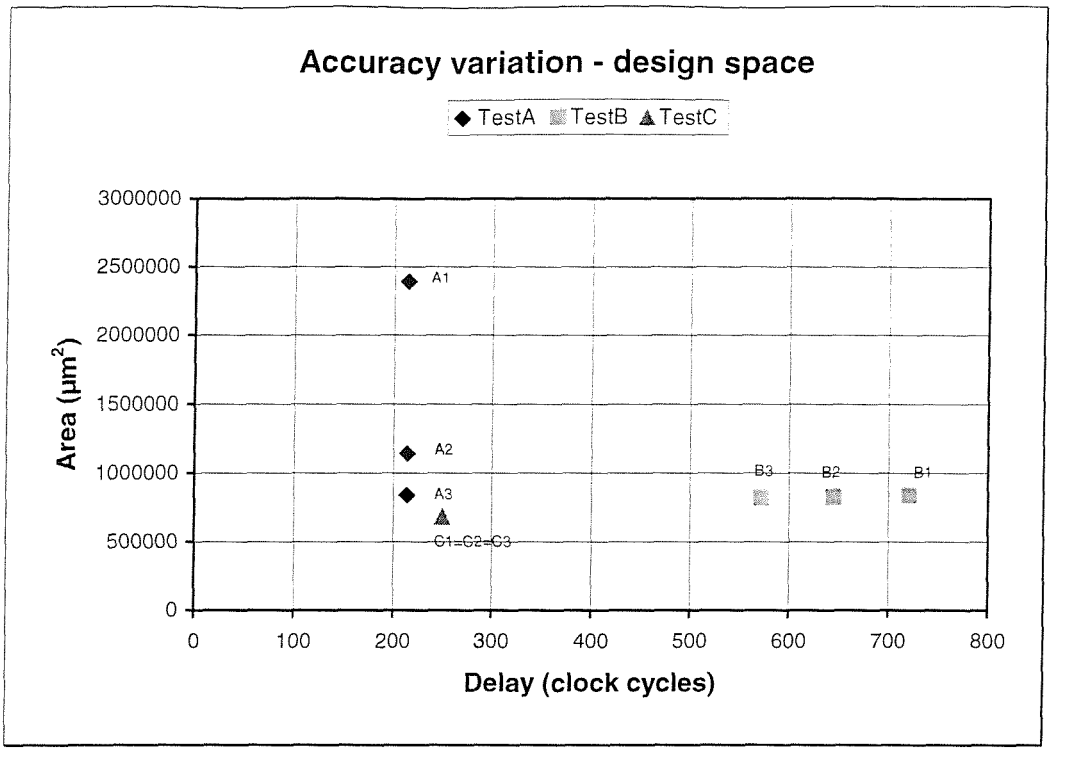


Figure 5.5 Design space for the three different benchmarks

From these results, some points of particular note:

- Major reduction of the total area cost occurs when the target accuracy is reduced on designs based on an internal table lookup interpolation (A1, A2, A3). As the accuracy reduces, table sizes for each function generator decrease. A reduction in the table size results in a smaller area required to store these tables as a static register. On the other hand, the interpolation procedure does not change with accuracy variation and therefore the total delay does not change.

- Reducing accuracy in designs based on CORDIC and iterative series methods reduces the number of iterations required to generate the output result, which result in a shorter execution time (B1, B2, B3). However, the hardware required to implement the units does not change apart from the loop control variables, which explains the negligible effect of the accuracy variation on the total area cost.

- Designs based on an external ROM maintain the same location in the design space. Accuracy variation in that case affects the function generator table size, which is stored externally. Thus the internal design hardware and the execution time are not affected by the accuracy variation.

Finally, it is worth mentioning that in the previous test, all the units in a single design were chosen to be of the same nature in order to highlight the individual effects when the accuracy changes. This is not always the case: a design in general will have different function generators and the accuracy variation will result in a change in both the overall area and delay.

5.3 Optimisation algorithm

The floating-point optimiser operates on the floating-point and complex functions within the design, binding each operation to a suitable base technique from the floating-point module library.

The algorithm relies on a number of pre-calculated metrics to guide the binding decision:

- *On-chip area* is assigned to each function generator in the library, presenting the area cost of the unit as a stand-alone design.
- Each function generator has an associated *off-chip area* figure defining the external ROM size required to implement the unit. Note that the *off-chip area* figure is only related to designs requiring a stored table and has the value of zero for other modules.
- *Delay factor* is defined for each function generator indicating the execution time of the floating-point module.
- *Sharability factor* is provided for each floating-point function generator qualifying the increase in area cost when the module is shared between a number of compatible functions.

In addition to these four metrics, the algorithm also requires extra information from the floating-point module library to identify the fixed-point sub-components, in each floating-point module, that have significant effects on the total module area and/or delay cost, such as a fixed point-multiplier. A set of graphs representing the four main metrics of the inverse tangent function generator for a target accuracy of 10^{-6} is shown in Figure 5.6.

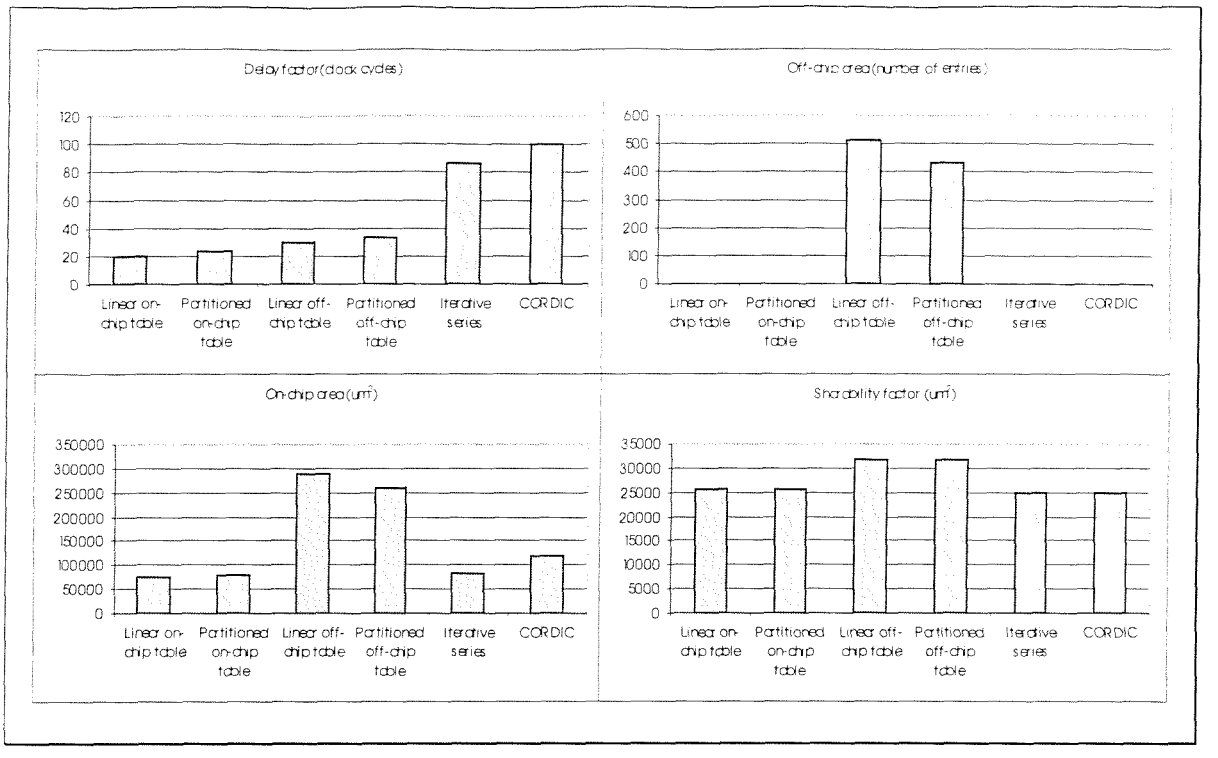


Figure 5.6 The inverse tangent function parameters for a target accuracy = 10^{-6}

The floating-point optimiser relies on two routines to perform the module binding operation:

1. *On-chip optimisation*: The main optimisation routine, responsible for assigning floating-point and complex functional units to on-chip based modules (on-chip table lookup, CORDIC, iterative series).

2. *External ROM utilisation*: A supporting routine invoked by the *on-chip optimisation* routine. It takes a number of floating-point operations and provides a possible mapping, which utilises the external ROM most efficiently.

The flowchart of the optimisation algorithm is shown in Figure 5.7. It is an iterative algorithm comprising six main steps:

1. Initially, all floating-point modules are mapped onto an on-chip table lookup based technique, implemented on an infinite, virtual, internal, on-chip ROM. The result in this step is the fastest possible implementation of the design based on the available floating-point module library. If this meets the user area constraints, and fits the physical system, the base technique mapping is complete and successful.

2. At this stage, the system starts trading speed against area trying to deliver the user requirements. A floating-point unit is selected at this stage as a target for the optimiser. Functional unit selected as a target for optimisation is mainly based on the total area cost of the functional unit (selecting the biggest unit for area optimisation) and the number of instances involved in the design.

3. Select an alternative implementation for that unit. The base mapping techniques are selected in the following order: 1)linear on-chip table, 2)partitioned on-chip table, 3) linear off-chip table, 4)partitioned off-chip table, 5)iterative series based unit, 6) CORDIC algorithm based unit. When a function is to be implemented as an off-chip table lookup, the external ROM utilisation routine is invoked to deliver a suitable implementation which utilises the ROM most efficiently. The external ROM mapping decision is based on an initial exhaustive search of all possible combinations of table lookup mappings to see which utilises the ROM most efficiently. Note that this does not lead to a combinatorial explosion, since a table is necessary for each floating point module *type*, not *instance*, and in practise, sub-table isomorphism within the floating-point module library components means that the largest number of off-chip tables ever considered cannot be larger than six.

4. The effect on the overall area of the mapping change is estimated. If the area is not reduced, goto step (5). Otherwise, the new mapping is accepted, and if the overall user requirements are satisfied, the algorithm terminates successfully.

5. If all the floating-point functional units are mapped onto the cheapest possible base technique (in terms of area cost), and the user requirements are not met, then the algorithm terminates in failure. Otherwise, return to step (2).

6. Once the previous iterative process terminates, and the user constraints are met, a final delay based optimisation pass is performed, trying to improve the overall system performance *without* violating the user constraint. For example, moving a functional unit mapping from iterative series to on-chip table if the difference between the target area cost and the actual area cost is greater than or equal to the area cost difference between the two base techniques.

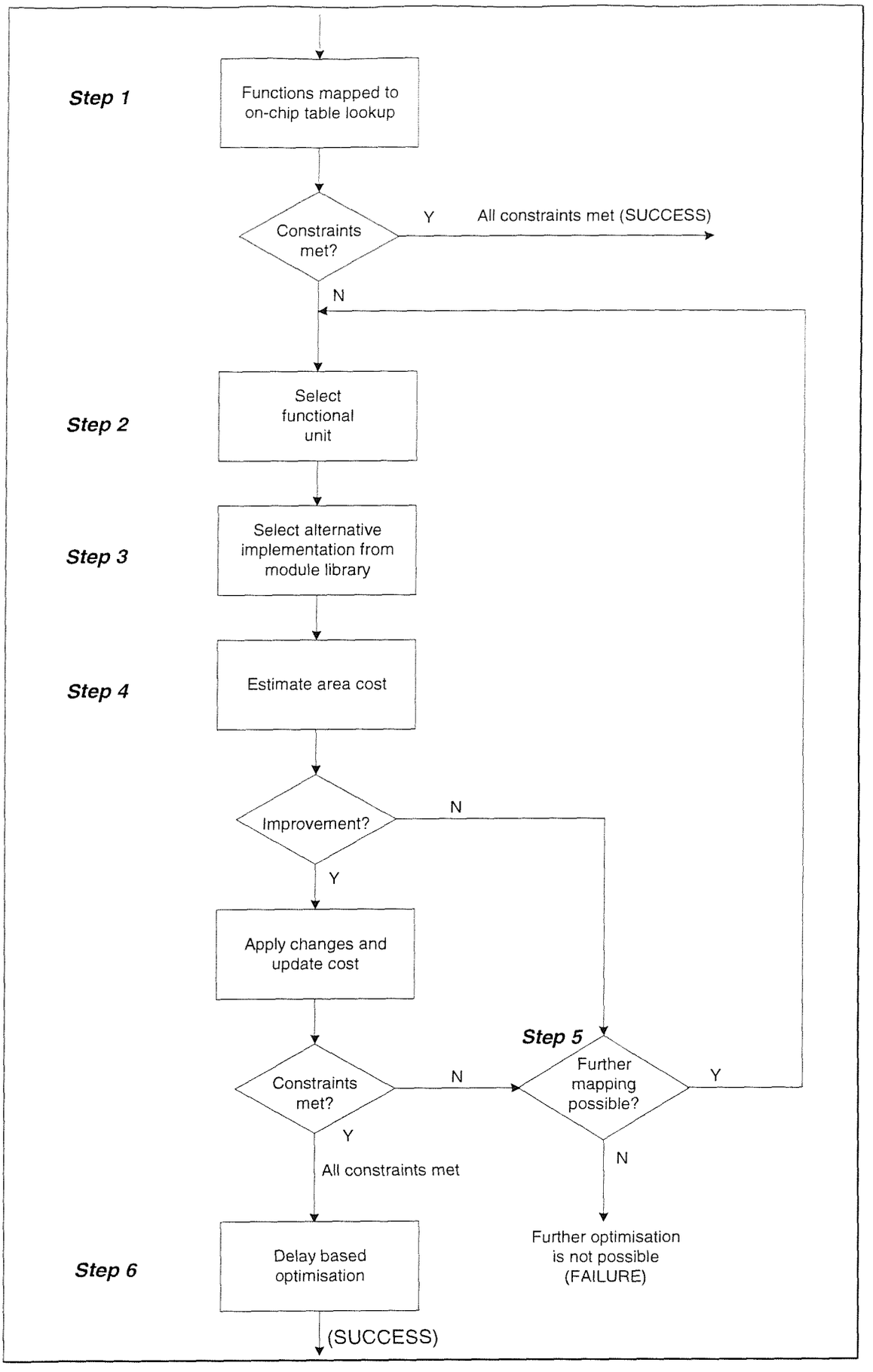


Figure 5.7 Optimisation algorithm flowchart

It is important to mention at this stage that the area cost estimation in step (4), and the area cost estimation of the initial input design are both performed using a separate *area estimator*. The main purpose of this area estimator is to predict the total area cost of the synthesised design once optimised by the MOODS system. The routine divides the design area into two parts:

1. *Fixed point cost* based on the storage units cost and the fixed point operator cost.

2. *Floating point cost* based on the floating point operators within the design.

The fixed point cost is calculated once while estimating the area cost of the initial design. Storage units cost is based on a direct accumulation of the these units cost (internal registers, internal ROMs, ...). For fixed point operators, a single pass is performed to detect the nature and the width of these operators within the design. During this initial pass, all operators of the same nature (adder, subtractor, ..) are grouped together, and the accumulated area cost of these groups is calculated.

For floating-point operators, maximum sharing of these units is expected (which is always the case as long as an initial optimisation phase is performed during the MOODS optimisation phase prior to flattening the design hierarchy). The cost of each floating point operator is then calculated as the sum of the single floating-point operator area cost and the multiplexing cost required to share this operator, which is based on the number of functional units within the design.

Although the area estimator does not take into account the effect that parallelism and registers sharing have on the design area. The nature of the floating point designs, in which area cost is dominated by the floating point functional units within the design, allows the estimator to provide a close estimation to area cost of the MOODS structural output with an accuracy close to 90%.

The design of this heuristic is derived from observations of base technique interactions. Some points of particular interest are:

- Functions based on table lookup implemented on off-chip ROM share a single ROM controller and a single I/O port.

- Expanding the hierarchical (real and complex) functions before the optimisation phase permits substructure sharing. If both the complex and real instances of a function are required, this delivers significant cost reductions.

- Mapping a function onto a CORDIC base technique makes subsequent mappings to that implementation more likely.

- Two or more functions having the same table (for example *sin()* and *cos()*) have only one physical table.

- The cost of an iterative series generator can be significantly changed by the prior availability of its primitive sub-units (multiplier, divider). Equally, the selection of this base technique reduces the cost of other operations by providing these units.

To demonstrate the effect of the floating-point optimisation algorithm, two behavioural descriptions incorporating floating-point manipulation have been chosen for analysis. The first design, labelled **bench1** is composed of nine floating-point operations: addition, multiplication, division, sine, inverse sine, natural logarithm, exponential, inverse tangent, and square root. The second design, **bench2**, contains all the operations available in **bench1,** but differs in the number of times each operation is invoked. It has: a single addition, a single multiplication, a single division, a single sine, two inverse sines, three exponentials, four inverse tangents, five natural logarithm and six square root operations.

Throughout the remaining portion of this section, performance figures are taken directly from the MOODS synthesis system using a Xilinx based module library. In this library, area and delay figures are obtained by an analysis of the floor planning results, obtained from the Xilinx Alliance development system, of the MOODS synthesis system output. Each design has been synthesised using a variety of optimisation configurations featuring different target area cost and various external ROM sizes. Note that the accuracy criterion is set to 1e-6 for all designs to eliminate the accuracy variation effect discussed earlier in this chapter.

Table 5.3 and Table 5.4 summarise the optimisation results of both benchmarks providing a range of area and delay figures. Each design is optimised several times providing twelve different implementations. Each configuration is given a unique reference code (A1, A2, A3 ...). The results also provide a breakdown of the total cost in terms of area occupied by

functional units, storage units, interconnect, and control units. Functional unit distribution among the three base techniques is also provided for each configuration.

The results are summarised by a set of graphs in Figure 5.8 to Figure 5.16. Figure 5.8 and Figure 5.9 show a section of the area/delay design space for **bench1** and **bench2** respectively. Figure 5.10 to Figure 5.15 show the functional unit distribution between the three base techniques for all configurations. Finally, Figure 5.16 provides a comparison of area breakdown of two configurations of **bench1** and **bench2** of particular interest.

Ref	Target area (μm2)	Available External ROM (Kbyte)	Estimated area (μm2)	Utilised ROM (Kbyte)	MOODS area (μm2)	MOODS delay (ns)	Delay (cycles)	Function units cost (μm2)	Storage Cost (μm2)	Muxing cost (μm2)	Control cost (μm2)	Off-chip table based units	On-chip table based units	CORDIC based units	Iterative series based units
A1	0	0	1.226E+06	0	1.117E+06	1.261E+05	865	2.123E+05	5.271E+05	3.214E+05	3.600E+04	0	0	2	4
B1	0	3.4	1.208E+06	2.98	1.101E+06	1.099E+05	753	2.134E+05	5.083E+05	3.424E+05	3.720E+04	2	0	2	2
C1	0	6.8	1.174E+06	6.05	1.110E+06	9.167E+04	627	2.156E+05	4.828E+05	3.730E+05	3.820E+04	4	0	1	1
D1	0	∞	1.163E+06	19.68	1.061E+06	5.828E+04	411	1.132E+05	4.447E+05	4.133E+05	4.010E+04	6	0	0	0
E1	2E+6	0	1.957E+06	0	1.831E+06	9.160E+04	627	9.005E+05	5.188E+05	3.755E+05	3.630E+04	0	4	2	0
F1	2E+6	3.4	1.974E+06	2.75	1.840E+06	8.223E+04	586	9.011E+05	4.883E+05	4.127E+05	3.780E+04	1	4	1	0
G1	2E+6	6.8	1.974E+06	2.75	1.840E+06	8.223E+04	586	9.011E+05	4.883E+05	4.127E+05	3.780E+04	1	4	1	0
H1	2E+6	∞	1.941E+06	12.32	1.822E+06	5.658E+04	399	8.760E+05	4.274E+05	4.803E+05	3.870E+04	2	4	0	0
I1	∞	0	3.115E+06	0	2.956E+06	5.431E+04	383	2.130E+06	4.336E+05	3.544E+05	3.810E+04	0	6	0	0
J1	∞	3.4	3.115E+06	0	2.956E+06	5.431E+04	383	2.130E+06	4.336E+05	3.544E+05	3.810E+04	0	6	0	0
K1	∞	6.8	3.115E+06	0	2.956E+06	5.431E+04	383	2.130E+06	4.336E+05	3.544E+05	3.810E+04	0	6	0	0
L1	∞	∞	3.115E+06	0	2.956E+06	5.431E+04	383	2.130E+06	4.336E+05	3.544E+05	3.810E+04	0	6	0	0

Table 5.3 Area and delay figures for various optimisation configurations of design bench1

Ref	Target area (μm2)	Available external ROM (Kbyte)	Estimated area (μm2)	Utilised ROM (Kbyte)	MOODS area (μm2)	MOODS Delay (ns)	Delay (cycles)	Function units cost (μm2)	Storage Cost (μm2)	Muxing cost (μm2)	Control cost (μm2)	Off-chip table based units	On-chip table based units	CORDIC based units	Iterative series based units
A2	0	0	2.028E+06	0	2.106E+06	3.512E+05	2365	2.069E+05	8.926E+05	9.121E+05	9.480E+04	0	0	8	13
B2	0	3.4	2.236E+06	2.98	2.233E+06	2.814E+05	1894	2.083E+05	8.950E+05	1.030E+06	9.960E+04	8	0	8	5
C2	0	6.8	2.174E+06	5.73	2.218E+06	2.715E+05	1827	2.079E+05	8.795E+05	1.030E+06	1.004E+05	9	0	8	4
D2	0	∞	2.172E+06	19.68	2.168E+06	1.458E+05	1023	9.762E+04	8.353E+05	1.130E+06	1.050E+05	21	0	0	0
E2	2.5E+6	0	2.408E+06	0	2.411E+06	2.566E+05	1722	5.156E+05	8.626E+05	9.388E+05	9.380E+04	0	11	7	3
F2	2.5E+6	3.4	2.344E+06	2.98	2.571E+06	2.088E+05	1406	5.169E+05	8.939E+05	1.060E+06	1.004E+05	7	11	3	0
G2	2.5E+6	6.8	2.475E+06	5.73	2.481E+06	1.946E+05	1311	4.791E+05	8.605E+05	1.040E+06	1.010E+05	8	11	2	0
H2	2.5E+6	∞	2.463E+06	15.3	2.377E+06	1.398E+05	981	3.769E+05	8.072E+05	1.090E+06	1.027E+05	10	11	0	0
I2	∞	0	3.955E+06	0	4.053E+06	1.278E+05	897	2.130E+06	7.956E+05	1.030E+06	9.730E+04	0	21	0	0
J2	∞	3.4	3.955E+06	0	4.053E+06	1.278E+05	897	2.130E+06	7.956E+05	1.030E+06	9.730E+04	0	21	0	0
K2	∞	6.8	3.955E+06	0	4.053E+06	1.278E+05	897	2.130E+06	7.956E+05	1.030E+06	9.730E+04	0	21	0	0
L2	∞	∞	3.955E+06	0	4.053E+06	1.278E+05	897	2.130E+06	7.956E+05	1.030E+06	9.730E+04	0	21	0	0

Table 5.4 Area and delay figures for various optimisation configurations of design bench2

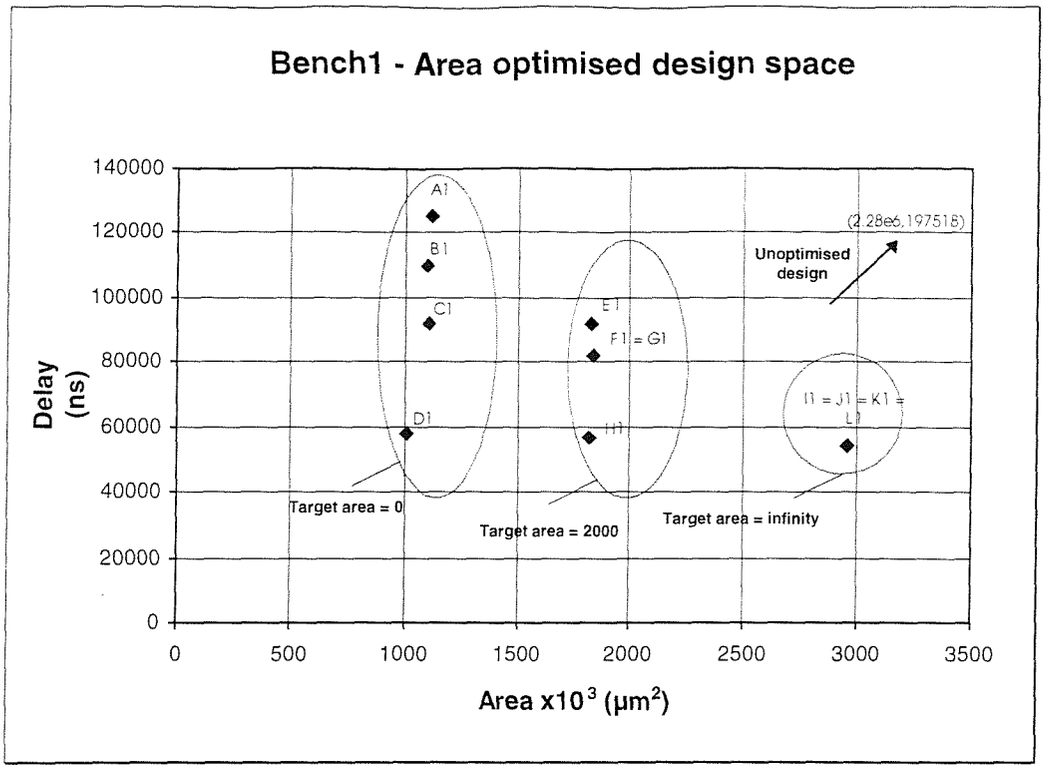


Figure 5.8 Bench1 design space

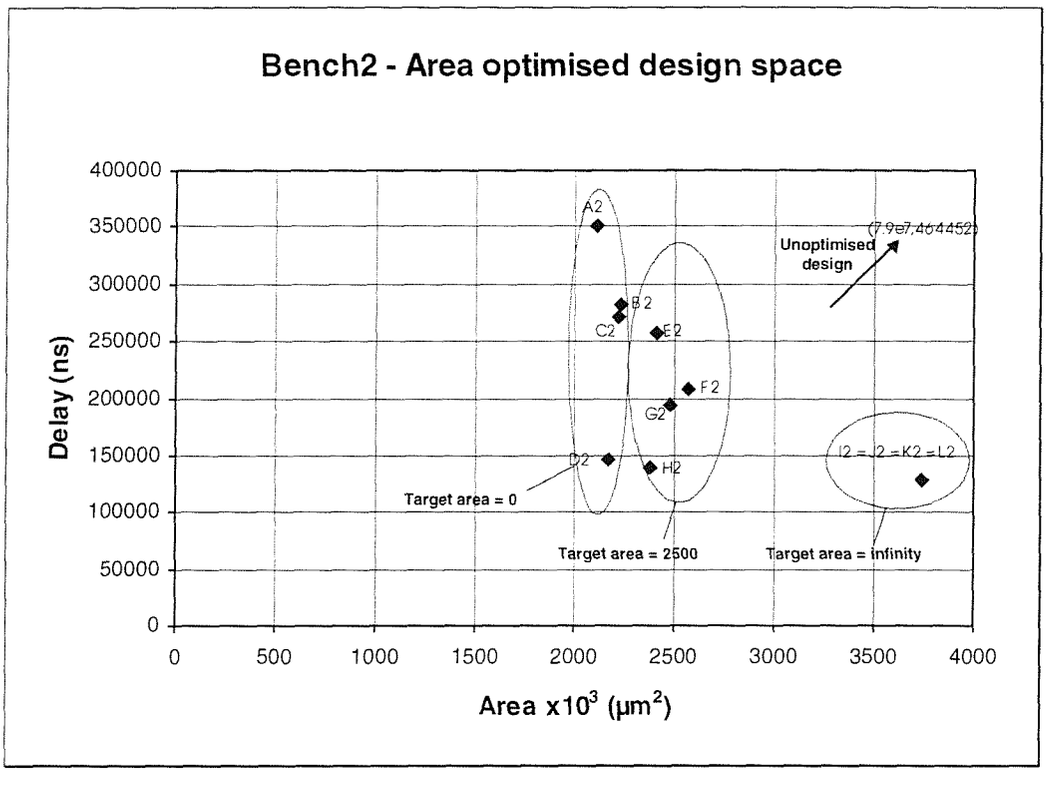


Figure 5.9 Bench2 design space

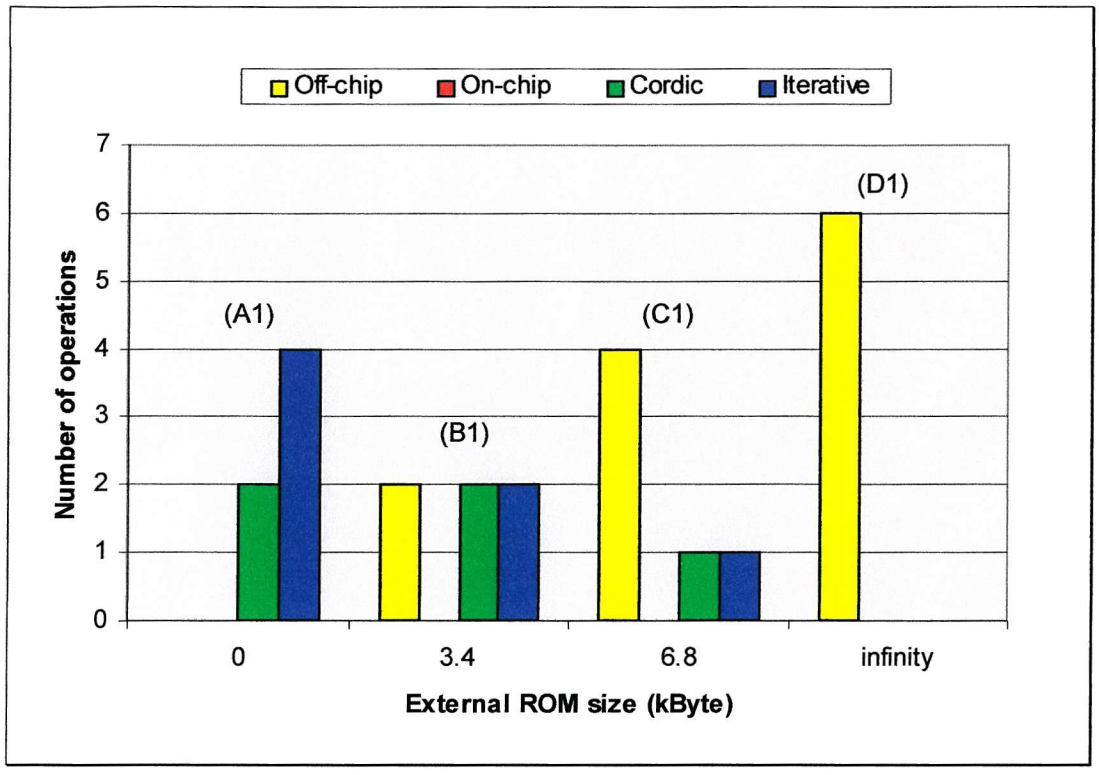


Figure 5.10 Distribution of functional units between the three base techniques for bench1 for target area = 0 μm^2 as a function of external ROM size

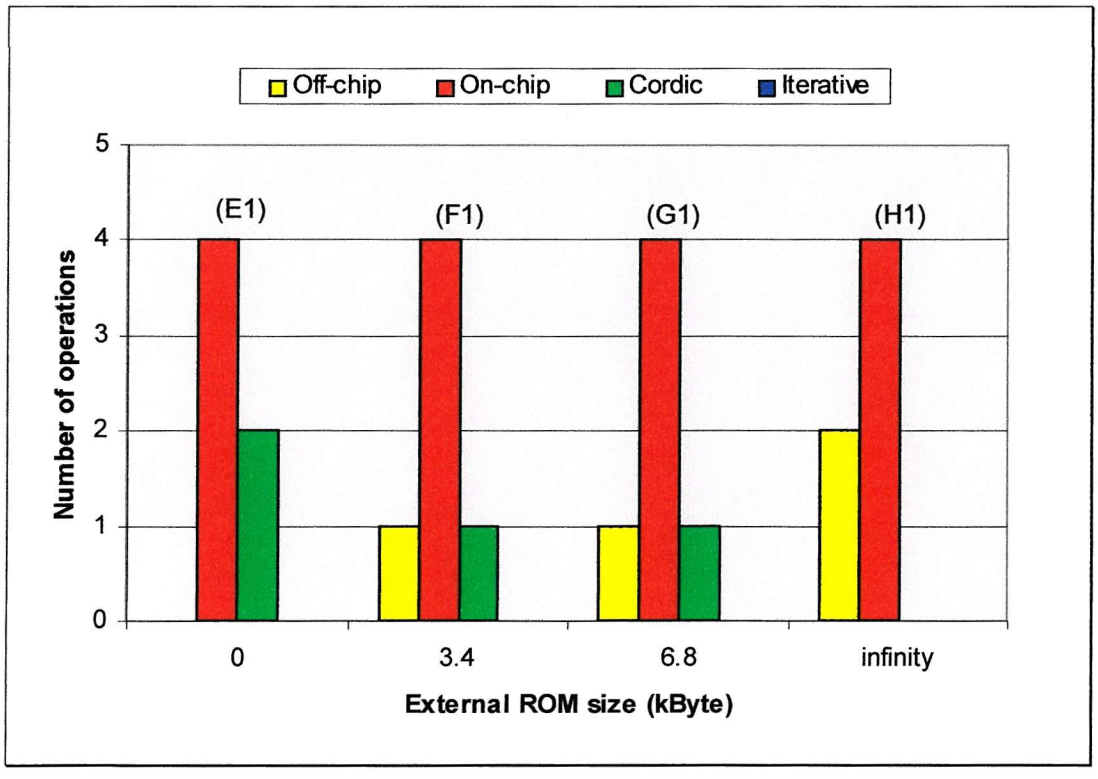


Figure 5.11 Distribution of functional units between the three base techniques for bench1 for target area = 2e6 μm^2 as a function of external ROM size

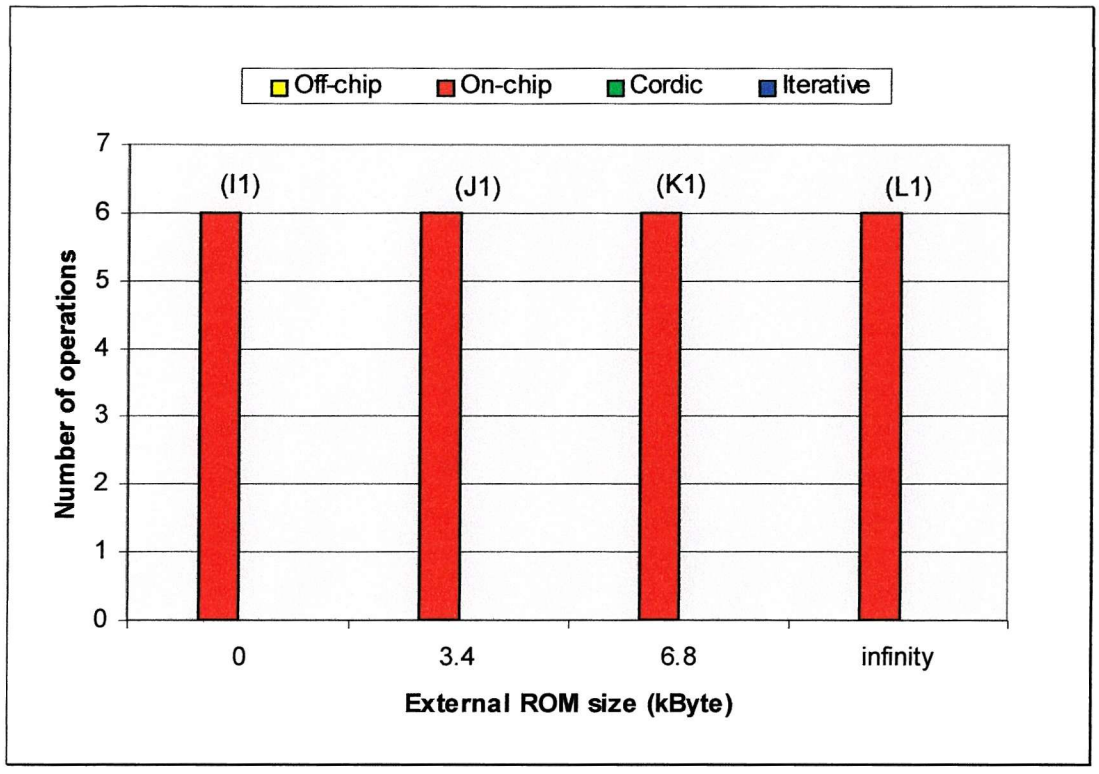


Figure 5.12 Distribution of functional units between the three base techniques for bench1 for target area = infinity μm^2 as a function of external ROM size

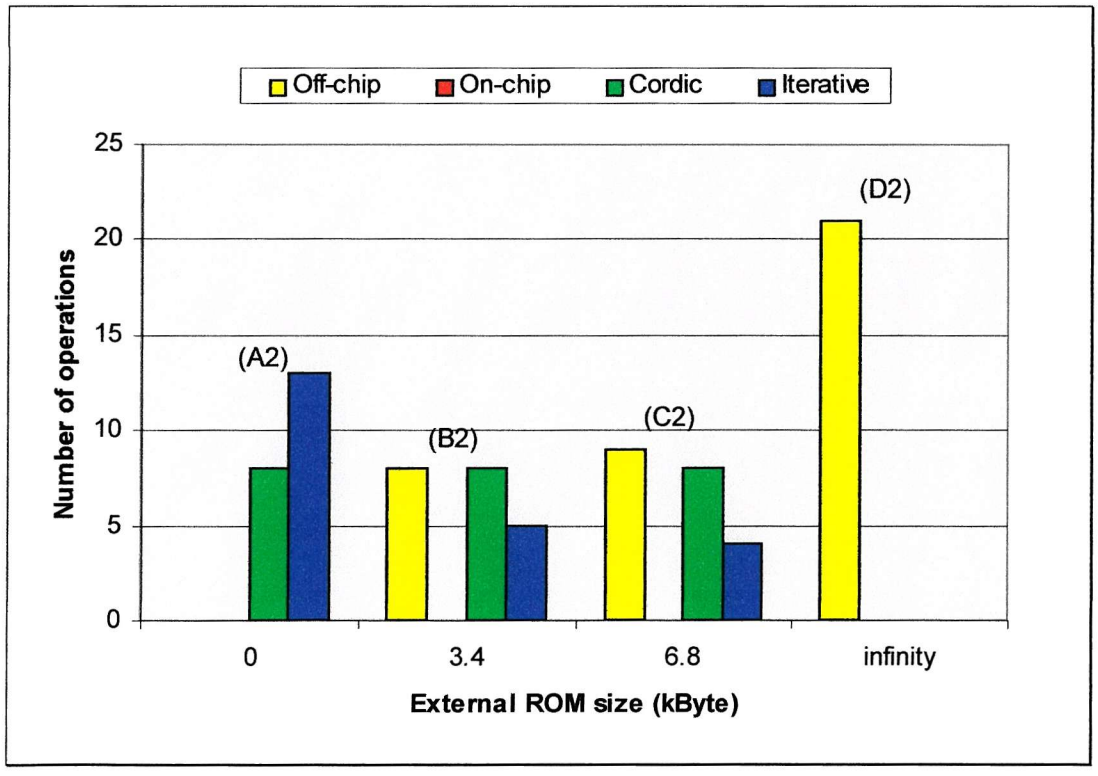


Figure 5.13 Distribution of functional units between the three base techniques for bench2 for target area = 0 μm^2 as a function of external ROM size

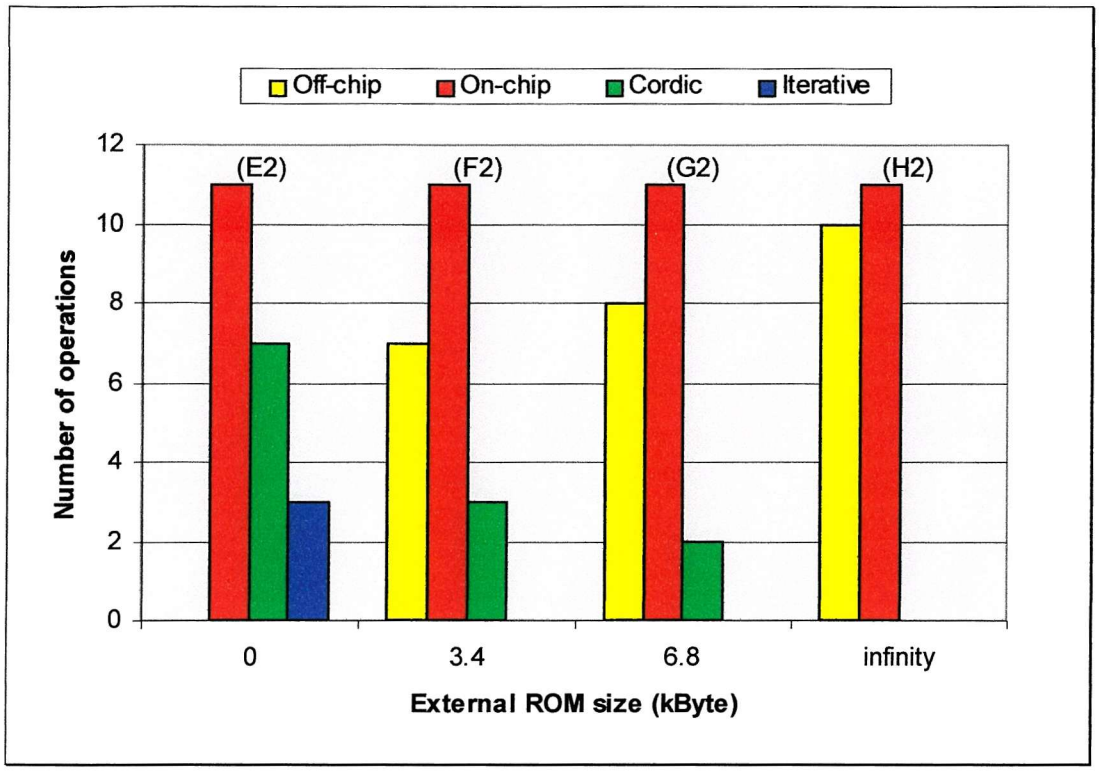


Figure 5.14 Distribution of functional units between the three base techniques for bench2 for target area = 2.5e6 μm^2 as a function of external ROM size

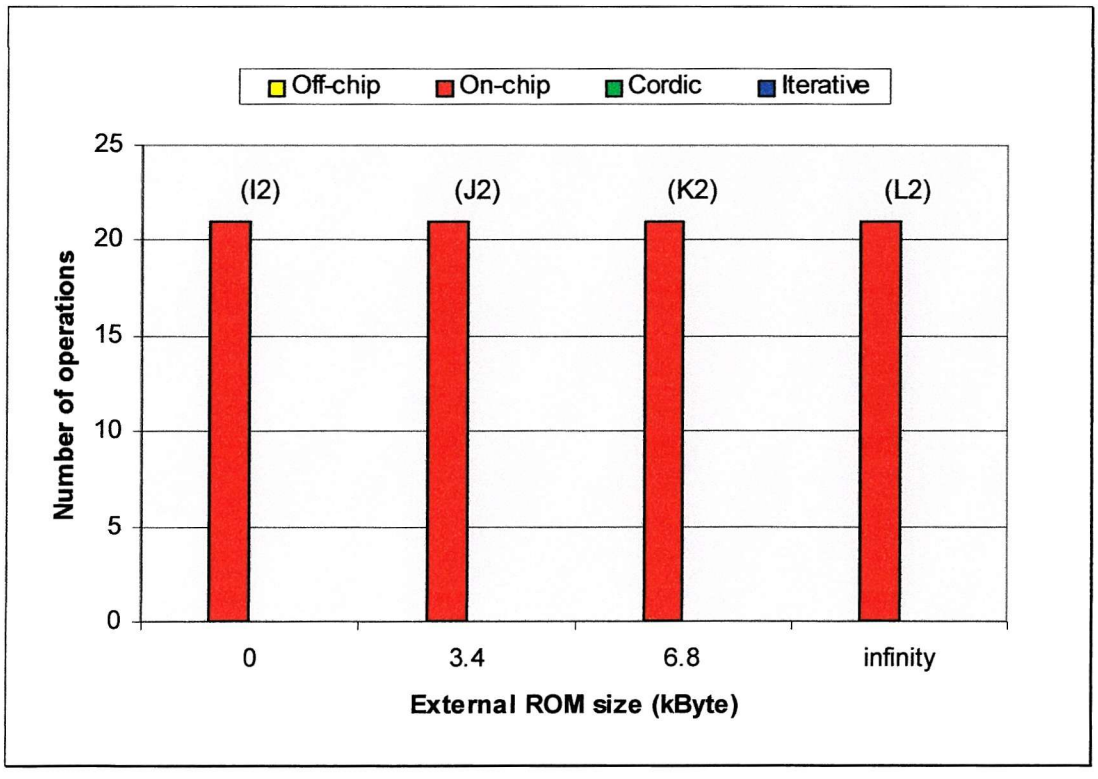


Figure 5.15 Distribution of functional units between the three base techniques for bench2 for target area = infinity μm^2 as a function of external ROM size

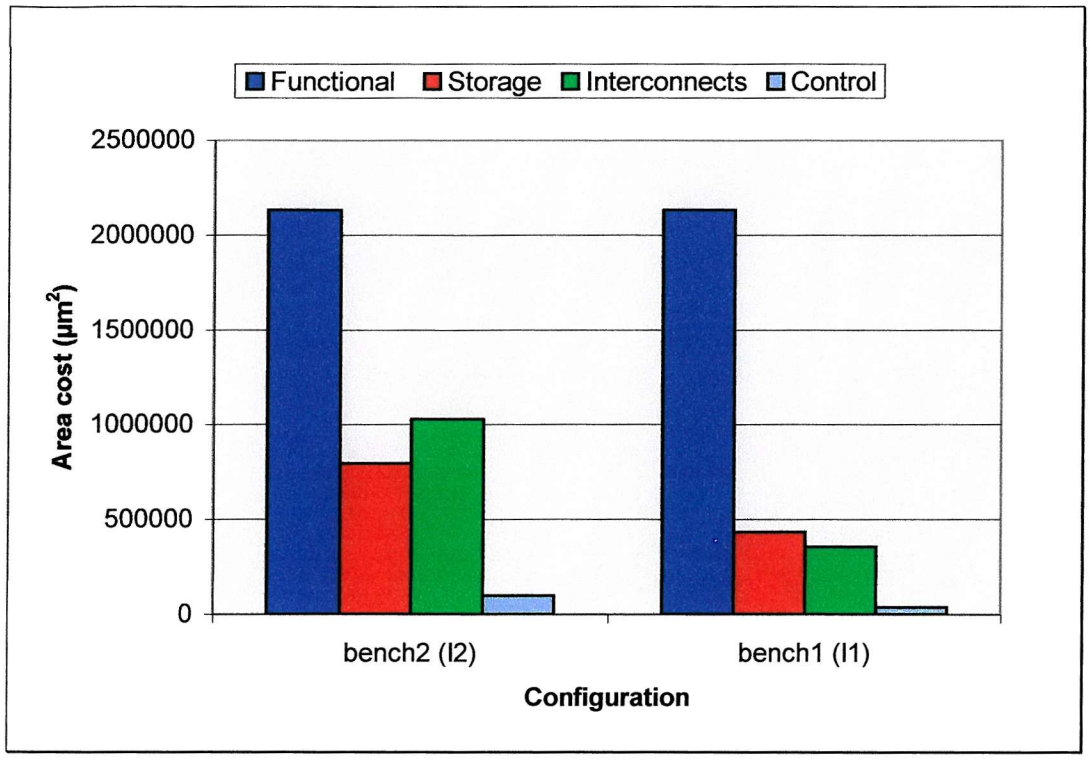


Figure 5.16 Area breakdown of the two designs based on similar base techniques (on-chip based implementation)

Comparing the floating-point optimiser estimated area cost to the final area cost of all optimised designs illustrates the ability of the floating-point optimiser to provide a very good *estimation* of the design characteristics. In all cases, the floating-point optimiser managed to *predict* the reduction the MOODS synthesis system optimisation phase will achieve with a good degree of accuracy (90%) without any feed back from the main optimisation phase.

The design spaces in Figure 5.8 and Figure 5.9 show the dominant effect of the target area cost on the achievable implementation of each design. Setting the initial target area cost fixes the optimal design space curve, with the variation in the external ROM size resulting in the design moving along that curve. Increasing the target area cost of the design shifts the curve away from the design space origin, providing considerably enhanced design performance.

Figure 5.10 and Figure 5.13 provide the distribution of functional units between the three main base techniques for **bench1** and **bench2** respectively when a minimum area cost is required. The most obvious feature is that the floating-point optimiser will always provide

an implementation based on both CORDIC and iterative series if a reasonable size external ROM is not available (A1, A2). This is expected, since both techniques always provide the most area efficient implementation. Increasing the available external ROM results in functional units moving gradually to an off-chip based implementation (B1, B2, C1, C2). When the total external ROM size is sufficient, the optimiser binds all possible floating-point units to an off-chip based implementation trying to reduce the total delay cost. This is illustrated in the same figures by (D1, D2). It is also important to notice that none of the floating-point units are bound to an on-chip table lookup based module, as they tend to introduce a noticeable increase in area cost and are not suitable when a minimum area cost is required.

If a minimum area is not required, the system will try to enhance the performance of the floating-point units in the design. This is illustrated in Figure 5.11 and Figure 5.14, where a target area cost of 2×10^6 μm^2 and 2.5×10^6 μm^2 are specified for **bench1** and **bench2** respectively[1]. In both figures, the majority of floating-point functional units were based on an on-chip table lookup module. The external ROM is only used to enhance the performance of floating-point functional units based on CORDIC or iterative series.

An interesting feature of the floating-point optimiser is illustrated in Figure 5.12 and Figure 5.15. Here the target area cost is sufficient to implement all functional units as an on-chip table lookup unit, providing a high performance design with a minimum delay. Varying the external ROM size has no effect on the module binding decision since the target area cost has already been met.

Finally, Figure 5.16 provides a comparison of the area breakdown of the two designs when implemented using similar on-chip based techniques. Note that both designs are similar in the floating-point functional units invoked, and differ only in the number of instances of each unit. The extra area cost in **bench2** is mainly caused by the interconnects required to share the floating-point functional units among compatible units, and the control required for this sharing. Functional unit costs hardly change between the two units, which illustrates the efficiency of unit sharing, as the only increase in cost when the number of

[1] The target area cost is increased in bench2 to compensate for the increase in area due to the number of internal registers required to pass data between the floating-point operators in addition to the multiplexing cost.

floating point operators within the design increases would be the input and output port multiplexing and a moderate increase in the control logic. It is also worth mentioning that this approach is even more efficient when the design targets an ASIC, since multiplexors in an ASIC are far less expensive compared to programmable logic devices, as they are based generally on pass transistors.

5.4 Experimental evaluation

The results presented in this section demonstrate the floating-point optimisation algorithm performance when applied to several designs. Designs are chosen to demonstrate and isolate the interactions listed in Table 5.1. The designs are grouped into nine different sets.

The first set of designs demonstrate the increase in area cost when a number of iterative series generators have been instantiated. Five designs are chosen: a sine (C1), an exponential (C2), a natural logarithm (C3), a combined sine and exponential (C4), and finally the three function generators in a single design.

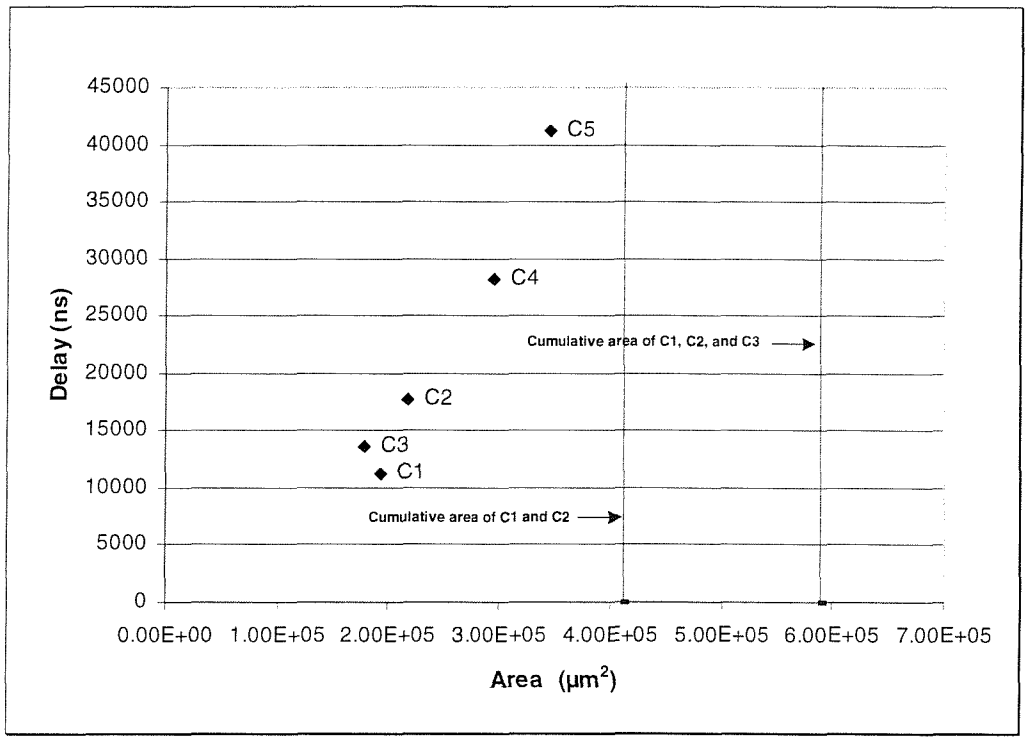


Figure 5.17 Design space for the first set of designs

It is clear from Figure 5.17 that the cost of switching between the three function generators is relatively small once an iterative series engine is implemented. An area cost reduction of

28% when two iterative series engines are shared and a 42% reduction in the total area cost is achieved when the three function generators are combined.

To demonstrate the effect of lookup table sharing, three designs are considered: a sine function generator based on an on-chip linear table lookup unit (C6), a cosine generator based on the same technique (C7), and a single design that combines the two generators (C8). The design space in Figure 5.18 shows the final area and delay cost of the three designs once optimised. Note that a major reduction in the area cost is achieved in C8 when compared to the accumulated area cost of (C6) and (C7). Over the range $[0,\pi/2]$, it is possible for these two units to share the same table lookup, which reduces the area cost required to store the internal table by 50%.

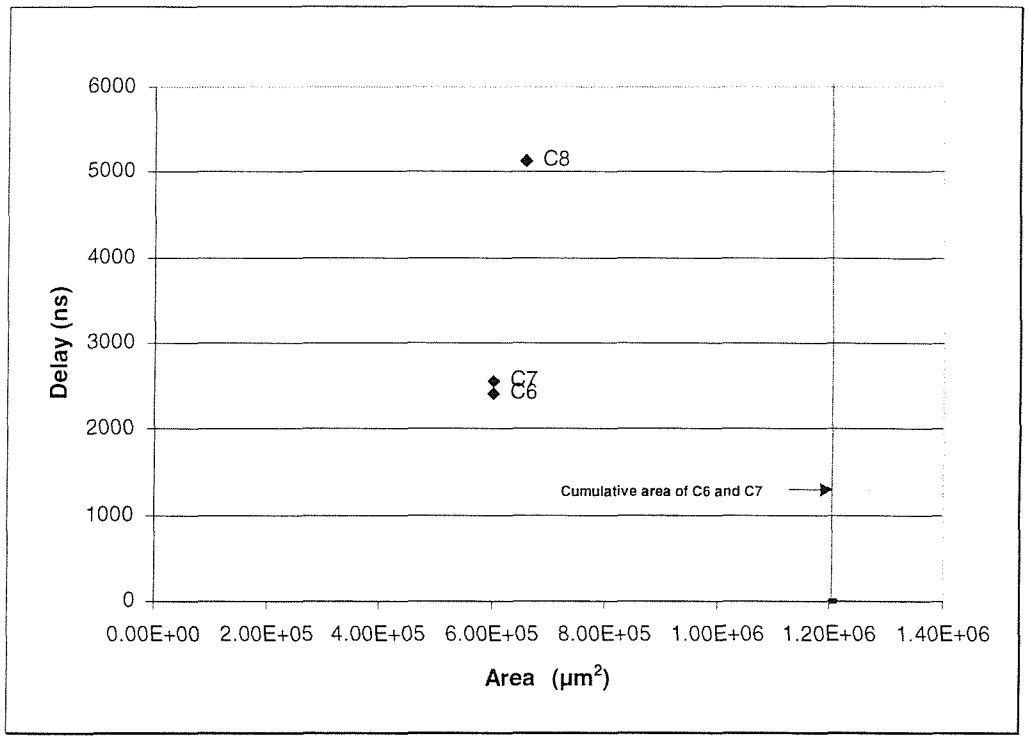


Figure 5.18 Design space for the second set of designs

The third set of designs represented in Figure 5.19 demonstrate the effect on total area cost when a complex and real function of the same nature are combined in a single design. The figure represents three designs: a real square root function generator based on an on-chip linear lookup table (C9), the corresponding complex polar function generator (C10), and a design that combines both units (C11). It is clear that when the complex function is implemented, the equivalent real function is almost free (in terms of area cost). Since the real square root building block is maximally shared between the two operators and the

moderate increase in area cost in (C11) when compared to (C10) is due to the sharing cost in terms of multiplexing and control logic.

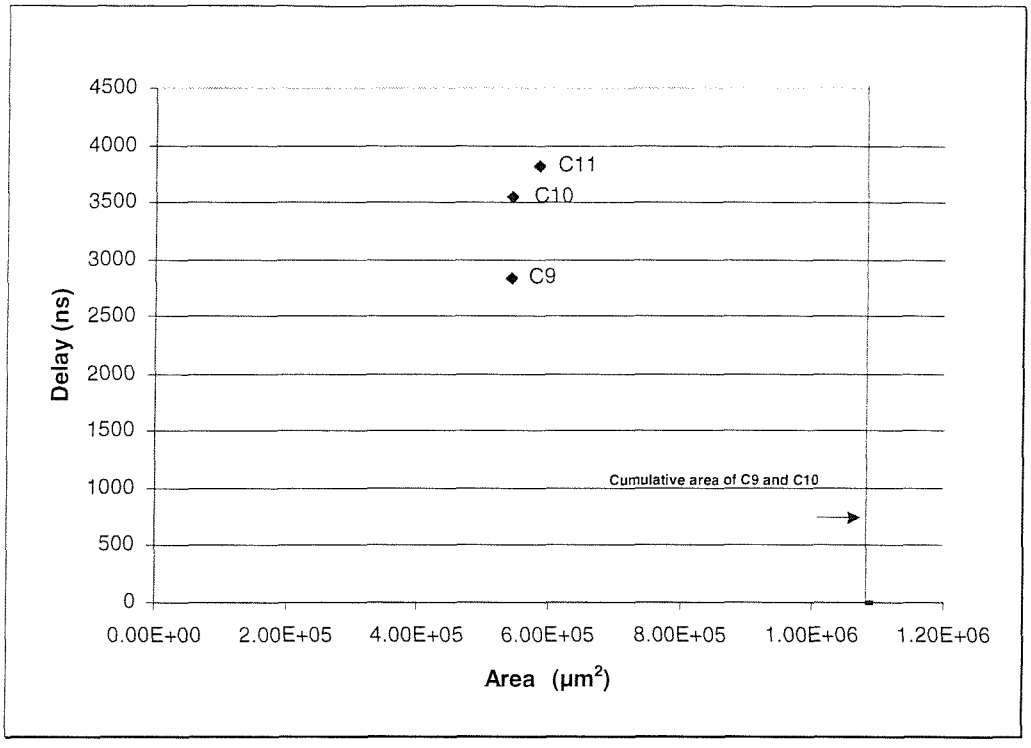


Figure 5.19 Design space for the third set of designs

Hierarchical functional units within the behavioural design are expanded to their sub-components before the floating-point optimisation phase. This allows a maximal sharing of similar units. This is illustrated in Figure 5.20. The hyperbolic sine (C13) is based on two exponential units. Which allows a reduction in area cost of 20% when both functional units exist within the same design (C14), when compared to the accumulated area cost of the exponential function (C12) and the hyperbolic sine (C13).

The CORDIC algorithm is exploited in this work to provide a cheap implementation (in terms of area cost) for a number of functional units, with the functional unit area mainly dominated by the variable width shift operation and the table of constants that store the rotation angle. When a number of CORDIC based function generators exist within a design, further reduction in the area cost is possible due to the possible sharing of the two units mentioned above. Figure 5.21 represents the design space of three designs: a cosine function generator based on CORDIC (C15), an inverse tangent function generator based on CORDIC (C16), and a design that contains both units (C17). Sharing the building

blocks in (C17) results in an area reduction of 31% when compared to the area cost of the two separate function generators.

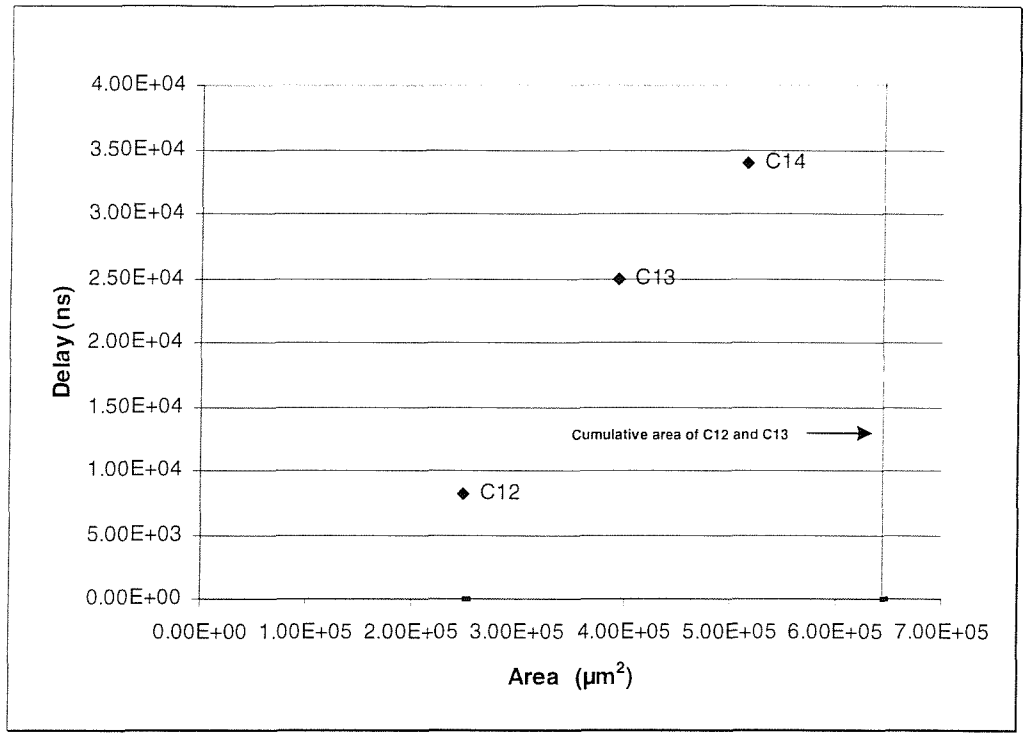


Figure 5.20 Design space for the fourth set of designs

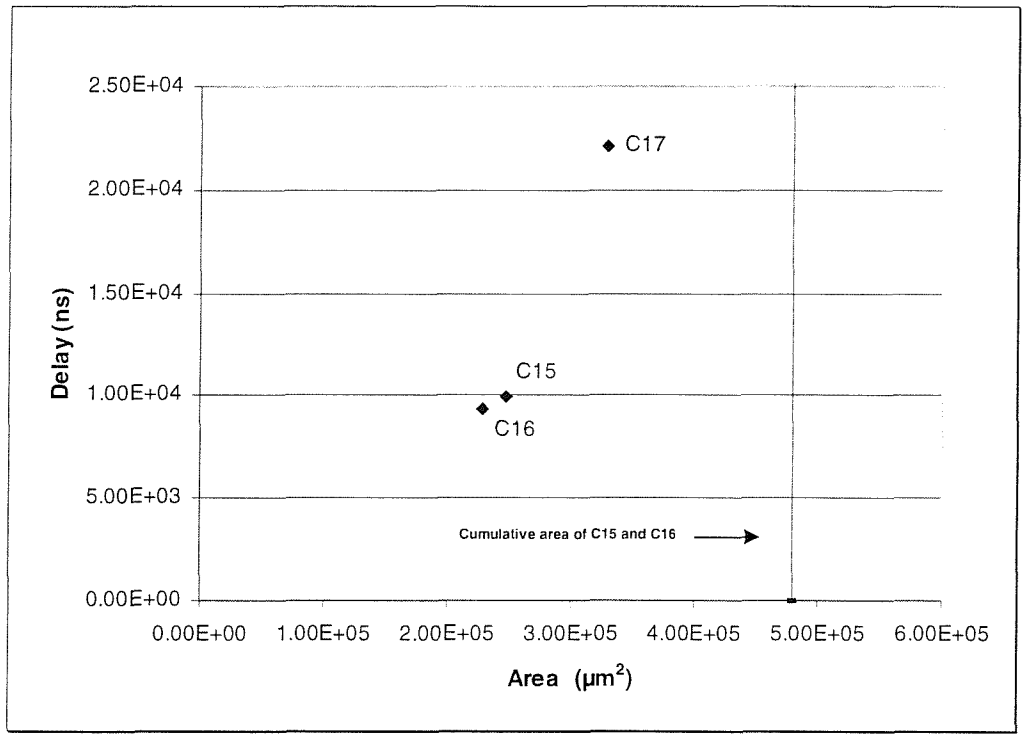


Figure 5.21 Design space for the fifth set of designs

Figure 5.22 represents the design space of three designs: sine function pre-processing stage (C18), iterative series based sine generator (C19), and a design that performs a full sine function generation based on iterative series (C20). It is clear in this example that the inline expansion of the two blocks before optimising allows datapath operator sharing at the sub-component level, which results in a 36% reduction in the total area cost in this case.

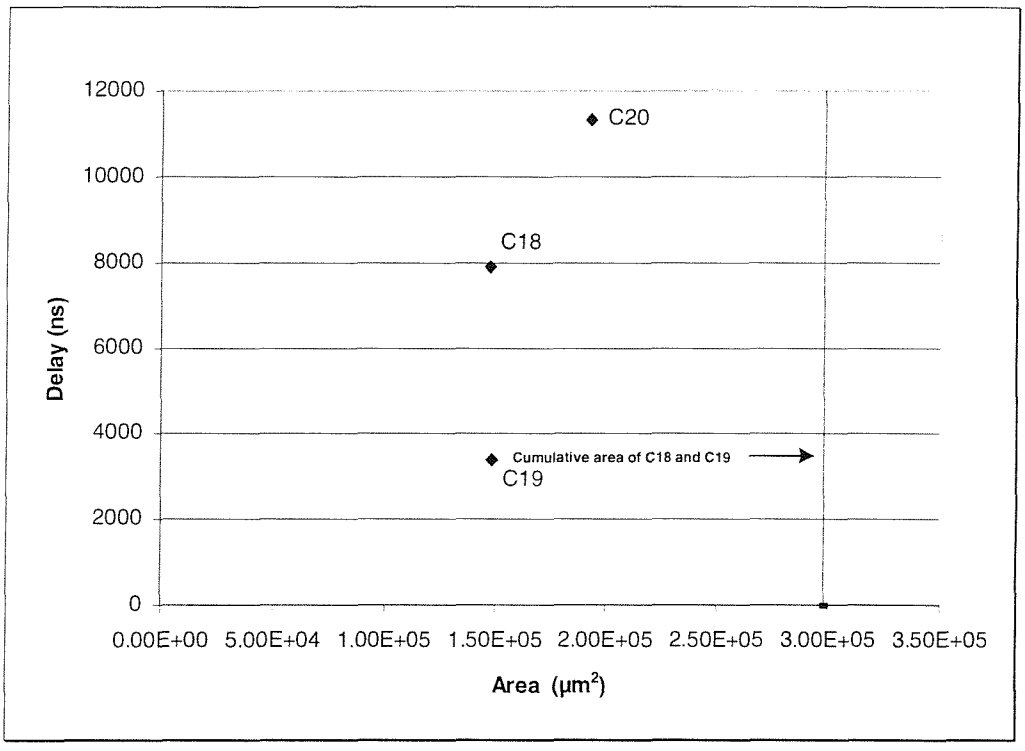


Figure 5.22 Design space for the sixth set of designs

The limited capacity of the external ROM available to implement a behavioural design requires a careful distribution of this ROM between the floating-point functional units, especially when a minimum area cost is requested. This is illustrated in example Figure 5.23. The design composes five floating point functional units: sine, inverse sine, exponential, natural logarithm, and square root. The floating point optimiser decision is to implement all but the inverse sine function utilising the external ROM (0.36 Kbyte in this example). The resulting design is illustrated by (C21). Assuming a similar design with the inverse sine function implemented using the external ROM (C22), the remaining four functional units will be mapped to CORDIC and the iterative series based technique. The random utilisation of the external ROM in the second example produces a design that is 32% slower and 2% bigger when compared to the floating point optimised output.

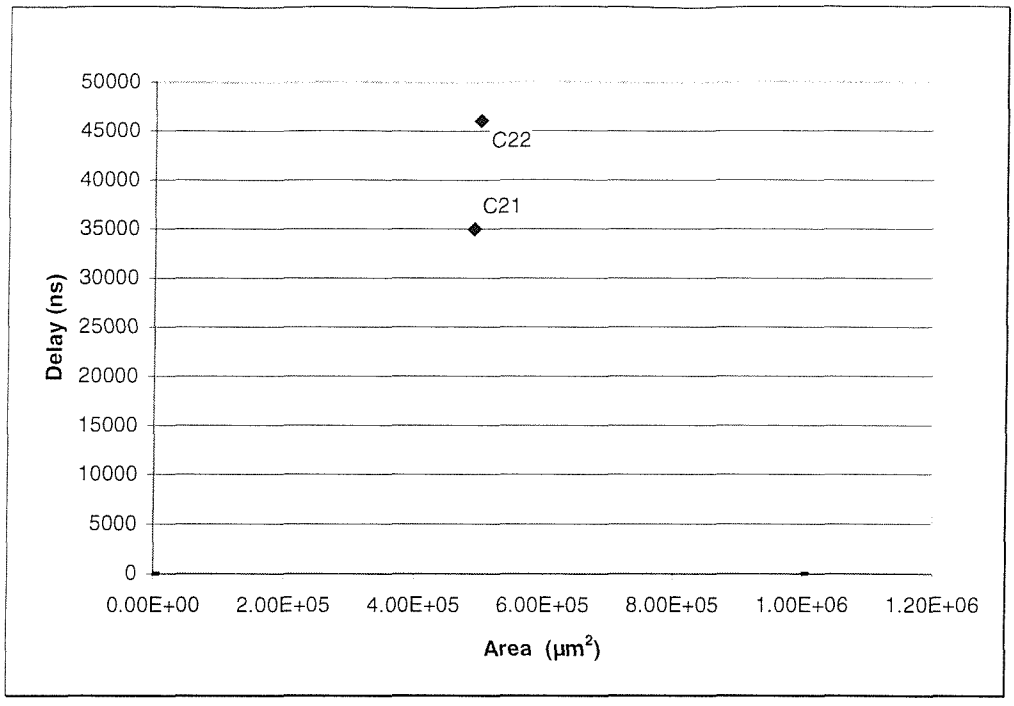


Figure 5.23 Design space for the seventh set of designs

Another important issue during the floating point optimisation phase is the final floating point functional unit accuracy selection. It is possible for a design to comprise similar floating point operators with different target accuracy. Two cases arise here based on the function generator assigned to the functional unit:

1. If the accuracy variation increases the area cost of the design without affecting the total system delay, all compatible floating point operators are assigned the highest accuracy.

2. If the accuracy variation results in delay variation, each functional unit is assigned its exact target accuracy.

This is illustrated in the example in Figure 5.24, which represents the area and delay cost of four different designs. The first two designs (C23, C24) consist of two sine generators implemented as on-chip table lookup. In (C24), the target accuracy in one of the function generators is reduced manually from 1e-6 to 1e-5. Note that the accuracy reduction had hardly any effect on the total delay which the total area cost increased. Therefore, the floating point optimiser always goes for the first choice. On the other hand, when both designs are implemented as iterative series based function generators (C25, C26), the accuracy variation reduces the total delay without affecting the total area cost.

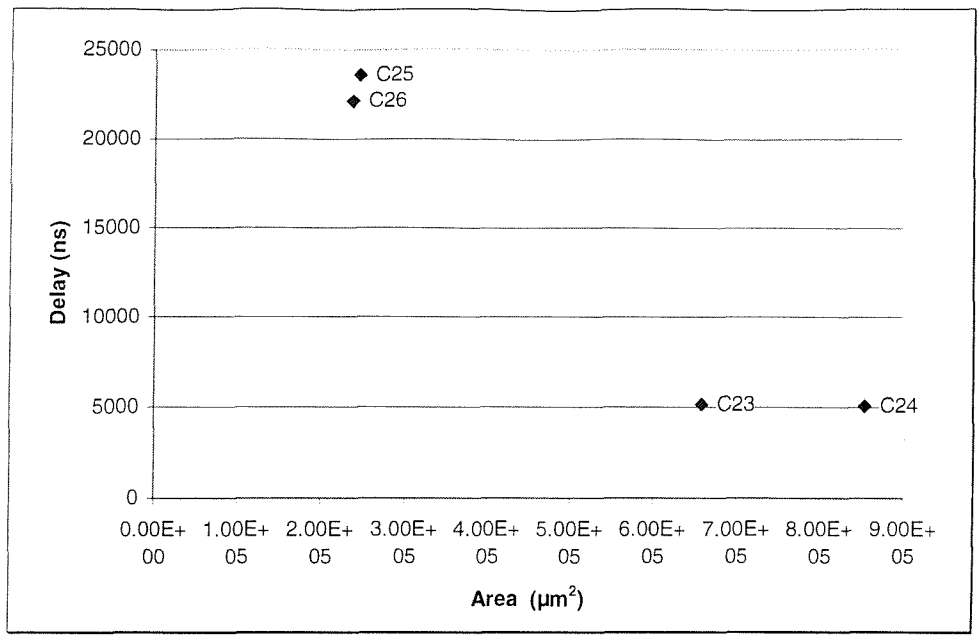


Figure 5.24 Design space for the eighth set of designs

The final set of examples demonstrates the importance of considering the area cost of floating-point operators sharing during the optimisation phase. It represents a design with ten square root operators with a target accuracy of 1e-4. The floating-point optimiser assigns the square root to an on-chip partitioned table lookup base implementation when a minimum area is requested. The reason is that the difference in the sharing cost between the off-chip (C27) and on-chip (C28) table lookup implementation once shared between ten operators exceeds the total area cost of the on-chip table as illustrated in Figure 5.25.

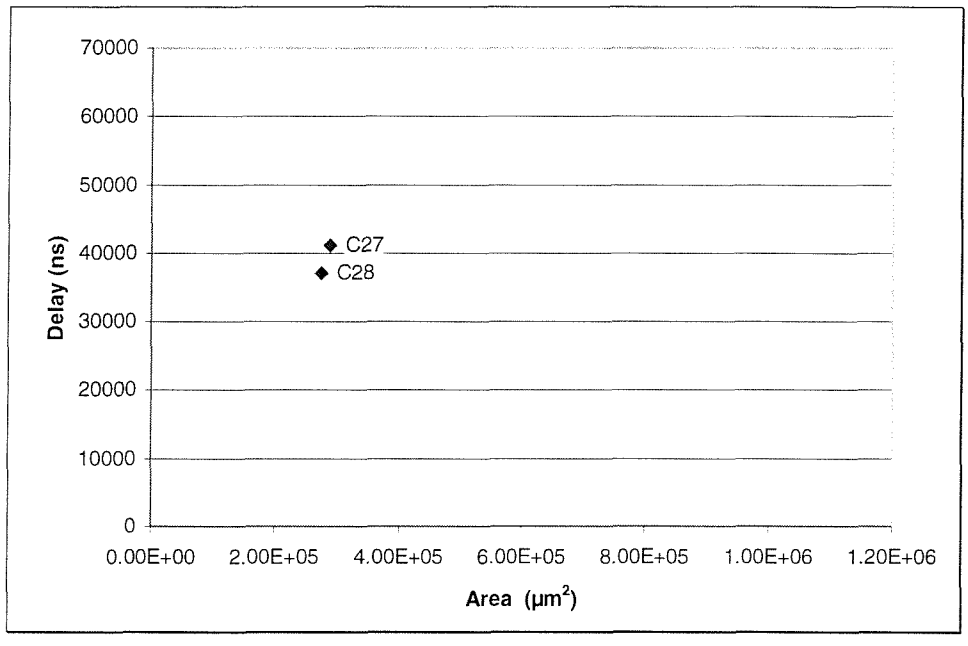


Figure 5.25 Design space for the ninth set of designs

Chapter 6

Practical synthesis using FPGAs

This chapter describes the design and implementation of a practical demonstrator, from specification to hardware. An exemplar is chosen that uses the floating-point capabilities to solve a practical problem: a cubic algebraic equation.

The chapter is divided into four sections: section 6.1 describes the FPGA hardware prototyping board. Section 6.2 discusses the floating-point cubic equation solver design and presents an exploration of the design space. Section 6.3 discusses the main problems encountered during the development cycle. Finally, section 6.4 presents comparisons with the floating-point performance of a number of microprocessors. Further details related to these topics may be found in Appendix E.

6.1 FPGA prototyping board

One of the biggest advantages of implementing digital designs on FPGAs is the possibility of fast prototyping. When behavioural synthesis tools are involved, the turn around time from an algorithmic level to an FPGA floor plan becomes extremely short. However, the last step (the physical implementation) requires a physical system to support it.

The FPGA test board is designed with the following objectives in mind. It should be:

- A flexible design, as it should be possible to reconfigure the FPGA board to almost arbitrary digital designs.
- Capable of interfacing to a PC.
- Possible to connect more than one board together to handle large designs.
- Possible to connect additional hardware to the design.

In order to accommodate these objectives, the architecture in Figure 6.1 is implemented.

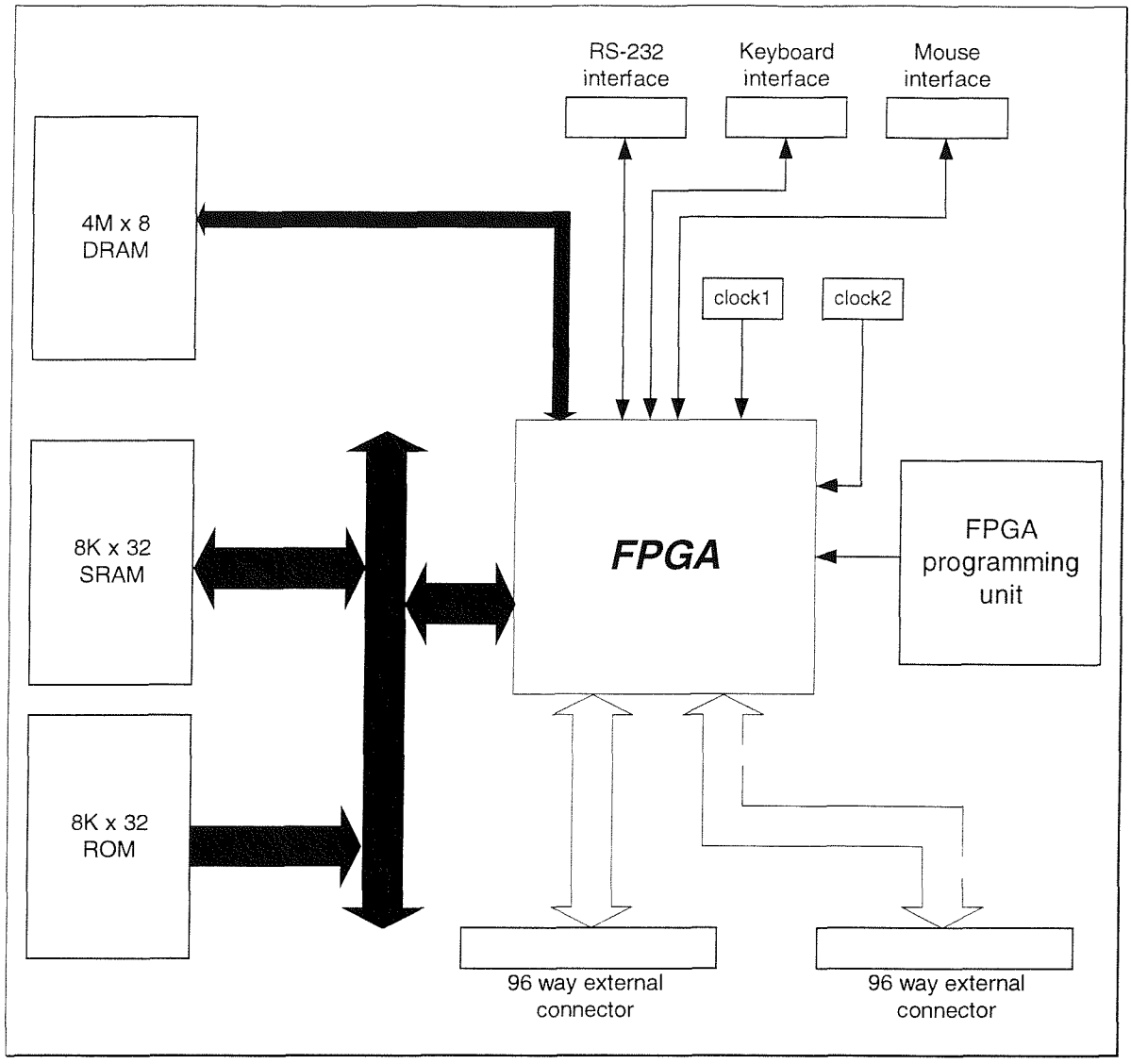


Figure 6.1 FPGA board block diagram

The FPGA board is compatible with three SRAM based Xilinx FPGAs: the XC4085XLPGA559, XC40125VXPGA559, and the XC40250XVPGA559 [101]. These devices vary in capacity as illustrated in Table 6.1. Three memory banks are provided: 8Kx32bit static RAM and 8K x 32bit ROM sharing the same address and data busses, and a 4M x 8bit dynamic RAM with a separate data and address bus.

In order to provide a simple way to interface the board to a personal computer, a RS232 serial port interface is provided. Two separate PS2 connectors are provided to allow a keyboard and a mouse input to the board.

Device	CLBs	Flip-flops	Typical gate range
XC4085XLPGA559	3136	7168	55000-180000
XC40125VXPGA559	4624	10336	80000-265000
XC40250XVPGA559	8464	18400	160000-500000

Table 6.1 FPGA devices characteristics

Two options are provided to allow programming the onboard Xilinx FPGA. A serial programming mode is supported via a separate connector that can be attached to a Xilinx programming cable [101], and a parallel programming mode is provided using an onboard EPROM. A set of dip switches is provided to switch between these two modes.

The FPGA board provides an environment where it is possible to implement a wide range of digital architectures on a single board. However, if it is required to connect two or more boards together or connect the design to a number of external units, two sets of 96 way connectors are provided to support 192 bit parallel connection to the external world.

Two external clock signals are provided to drive the FPGA. Each internal flip-flop can be triggered by any of these clocks on either the rising or the falling edge. The XC4000XV devices can run at a maximum synchronous system clock of 100 MHz. Each device in this family is available in three speed grades (-09, -08, and -07), with a maximum clock frequency of 76MHz, 87MHz, and 100MHz respectively.

Figure 6.2 shows a photograph of the final hardware unit, identifying the main components, and their position on the test board. Further details regarding different aspects of the board such as I/O port assignment and programming details are provided in Appendix E.

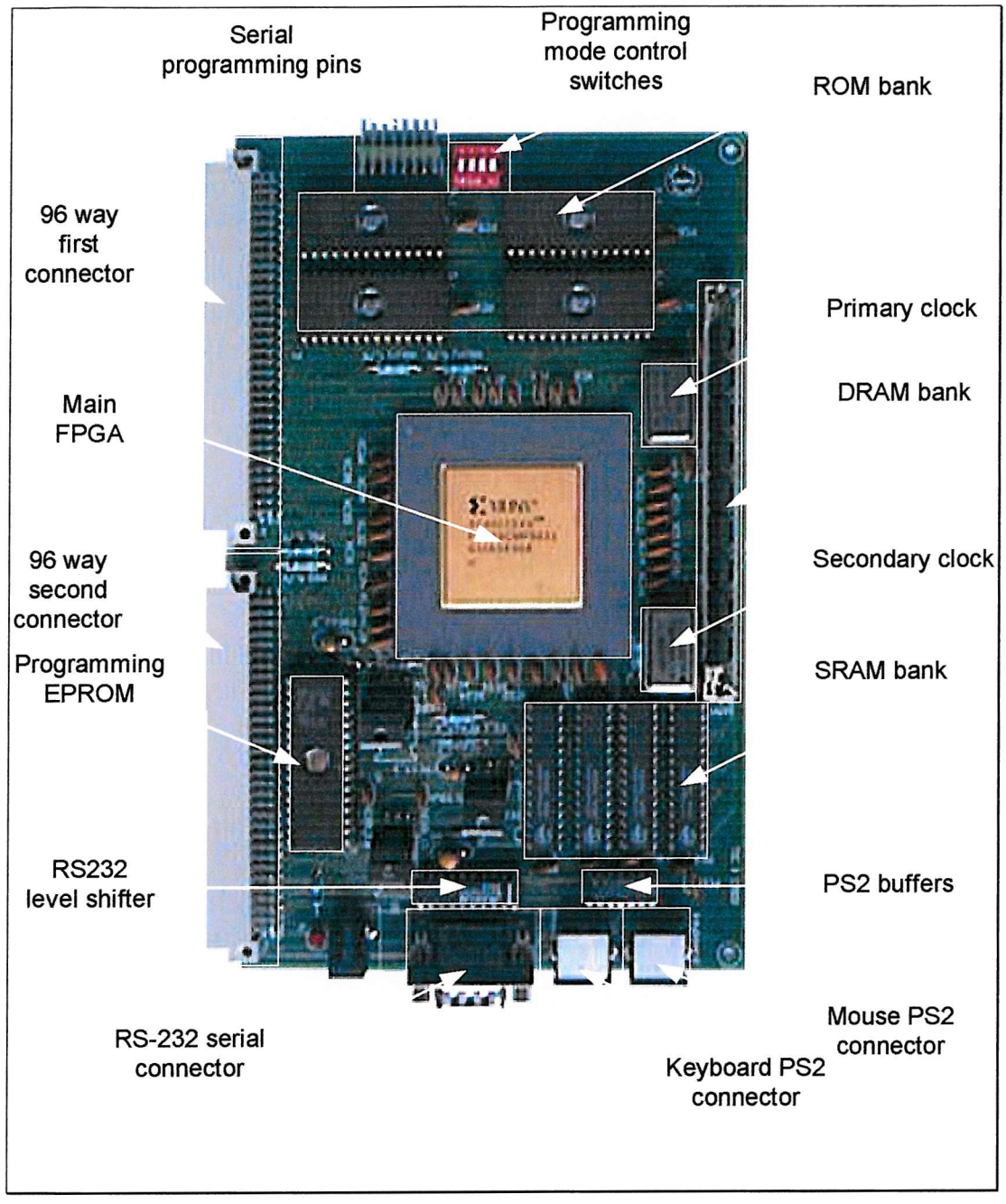


Figure 6.2 FPGA board photograph

6.2 Algebraic cubic equation solver

This section describes the detailed design and implementation of the exemplar, a cubic equation solver capable of handling real coefficients and delivering complex roots. The system reads three input variables from a keyboard unit representing the three parameters of a cubic equation and displays the input variables along with the three roots of the cubic equation on a VGA screen using the built in VGA display adapter (we assume the coefficient of x^3 is normalised to unity).

A block diagram of the system is shown in Figure 6.3. The keyboard interface unit reads the three parameters and converts them to the IEEE single-precision floating-point format. The three input parameters are also passed to the output stage to be displayed on the VGA screen. The core unit performs a number of floating-point calculations to generate the three roots. The three roots are then passed to the output stage to be displayed on the VGA screen.

An *initialise* key is provided using one of the unused numeric keypad keys in the keyboard. Pressing the *initialise* key at any stage will result in resetting the system and the output stage and the system goes into an initial state waiting for a new set of input parameters.

The design is divided into three units: the input stage which includes the keyboard interface and the format conversion unit; the output stage that drives the VGA display adapter; and the core unit which performs the floating-point calculations. These units will be discussed in detail in the following sections.

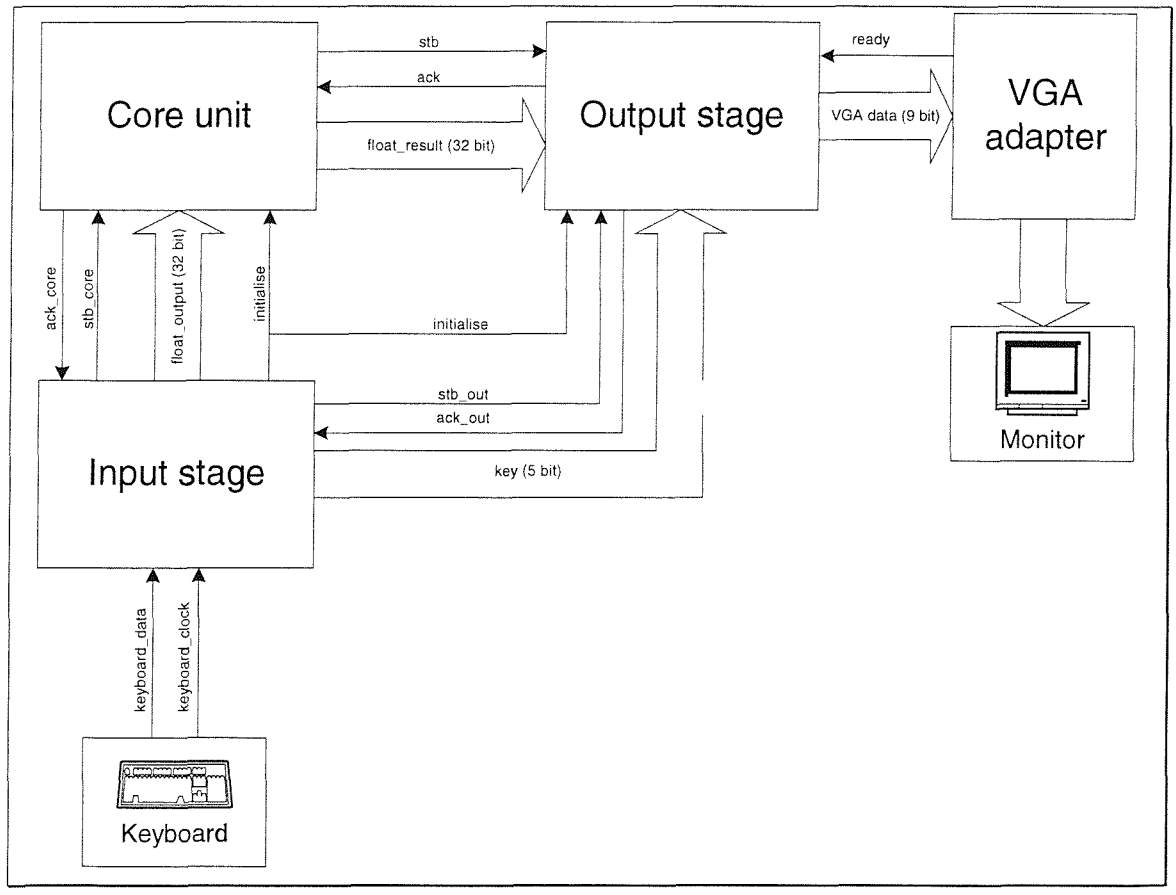


Figure 6.3 Cubic equation solver block diagram

6.2.1 Input stage

The input stage of the design performs two main operations:

1. Read the keyboard input data and decode it to numerical values.
2. Convert each numerical parameter from a decimal format to a single-precision floating-point format.

Full Details are given in Appendix E.

6.2.2 Output stage

The final section to be considered is the output stage, which displays the input parameters and the output result on a VGA display driven by a VGA adapter. An example of the displayed result is shown in Figure 6.4. A simple technique is adopted to reduce the complexity of the format conversion unit[102]. Details are available in Appendix E.

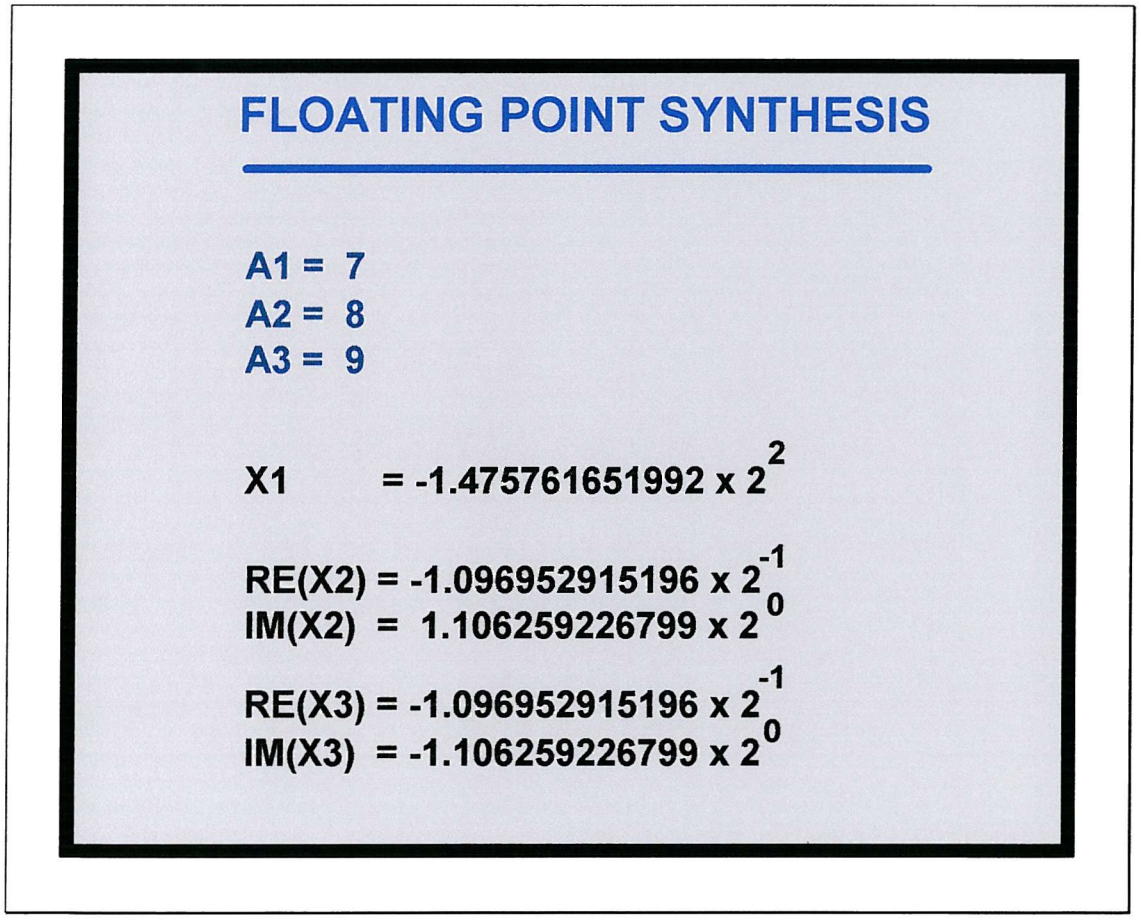


Figure 6.4 Cubic equation solver display

6.2.3 Core unit

The core unit is the most complex part of the whole system. It receives three floating-point variables from the input stage and performs a number of floating-point operations to generate the three roots of the cubic equation. The functionality of the core unit can be

DEFINITIONS:

$$x^3 + a_1 x^2 + a_2 x + a_3 = 0$$

SOLUTION:

$$Q = \frac{3a_2 - a_1^2}{9}; R = \frac{9a_1a_2 - 27a_3 - 2a_1^3}{54};$$

$$D = Q^3 + R^2;$$

if (D = 0)

$$S = \sqrt[3]{R};$$

$$x_1 = 2S - \tfrac{1}{3}a_1;$$

$$x_2 = -S - \tfrac{1}{3}a_1;$$

$$x_3 = x_2;$$

else if (D > 0)

$$S = \sqrt[3]{R + \sqrt{D}};$$

$$T = \sqrt[3]{R - \sqrt{D}};$$

$$x_1 = S + T - \tfrac{1}{3}a_1$$

$$x_2 = -\tfrac{1}{2}(S+T) - \tfrac{1}{3}a_1 + \tfrac{1}{2}i\sqrt{3}(S-T)$$

$$x_3 = -\tfrac{1}{2}(S+T) - \tfrac{1}{3}a_1 - \tfrac{1}{2}i\sqrt{3}(S-T)$$

else

$$\cos\theta = R/\sqrt{-Q^3}$$

$$x_1 = 2\sqrt{-Q}\cos(\tfrac{1}{3}\theta) - \tfrac{1}{3}a_1$$

$$x_2 = 2\sqrt{-Q}\cos(\tfrac{1}{3}\theta + \tfrac{2\pi}{3}) - \tfrac{1}{3}a_1$$

$$x_3 = 2\sqrt{-Q}\cos(\tfrac{1}{3}\theta + \tfrac{4\pi}{3}) - \tfrac{1}{3}a_1$$

Figure 6.5 Cubic equation solution

described behaviourally by the set of arithmetic operations required to solve a cubic equation. This is illustrated in Figure 6.5.

The translation to VHDL of Figure 6.5 is direct, and is shown in Figure 6.6. The full design listing (including the IO subsystems) can be found in Appendix E.

```
package CoreConst is
  constant con1 : real := 0.866025404; -- sqrt(3)/2
  constant con2 : real := 2.094395102; -- 2Pi/3
  constant con3 : real := 4.188790204; -- 4Pi/3
end;
use work.CoreConst.all;
entity core is
port (
      input     : in float;
      stb_in    : in bit;
      ack_in    : out bit;
      new_entry : in bit;
      stb_out   : out bit;
      ack_out   : in bit;
      data_out  : out float
      );
end;
architecture behave of core is
begin
process
variable a1,a2,a3,S,T : float;
variable R,Q,R_sq,Q_cu,D, sqrt_D : float;
variable X1 : float;
variable Temp1,Temp2,theta3 : float;
variable X2,X3 : cmplx;
begin
  get_input_data;
  Q := ((TO_FLOAT(3.0)*a2)-(a1*a1))/TO_FLOAT(9.0);
  R := ((TO_FLOAT(9.0)*a1*a2)-(TO_FLOAT(27.0)*a3)-(TO_FLOAT(2.0)*a1*a1 *a1))/TO_FLOAT(54.0);
  R_sq := R * R;
  Q_cu := Q * Q * Q;
  D := R_sq + Q_cu;
  if (D = TO_FLOAT(0.0)) then
    S := CBRT(R);
    Temp1 := a1/TO_FLOAT(3.0);
    X1 := TO_FLOAT(2.0)*S-Temp1;
    X2 := TO_COMPLEX(-S-Temp1,TO_FLOAT(0.0));
    X3 := X2;
  elsif (D > TO_FLOAT(0.0)) then
    sqrt_D := SQRT(D);
    S := CBRT(R+sqrt_D);
    T := CBRT(R-sqrt_D);
    Temp1 := S+T;
    Temp2 := a1/to_float(3.0);
    X1 := Temp1-Temp2;
    X2 := TO_COMPLEX((-Temp1/TO_FLOAT(2.0))-Temp2,(S-T)*TO_FLOAT(con1));
    X3 := CONJ(X2);
  else
    theta3 := ACOS(R/SQRT(-Q_cu))/TO_FLOAT(3.0);
    Temp1 := a1/TO_FLOAT(3.0);
    Temp2 := TO_FLOAT(2.0)*SQRT(-Q);
    X1 := Temp2*COS(theta3)-Temp1;
    X2 := TO_COMPLEX(Temp2*COS(theta3+TO_FLOAT(con2))-Temp1,TO_FLOAT(0.0));
    X3 := TO_COMPLEX(Temp2*COS(theta3+TO_FLOAT(con3))-Temp1,TO_FLOAT(0.0));
  end if;
  send_output_result;
end process;
end;
```

Figure 6.6 Design1 VHDL behavioural description

Figure 6.7 shows the design space for this system. A1 represents the original unoptimised design, B1 represents design optimised for area (target area = 0) with 27.7 Kbyte available external ROM, C1 represents area optimised design without an external ROM, and D1 is a delay optimised design (target area = ∞).

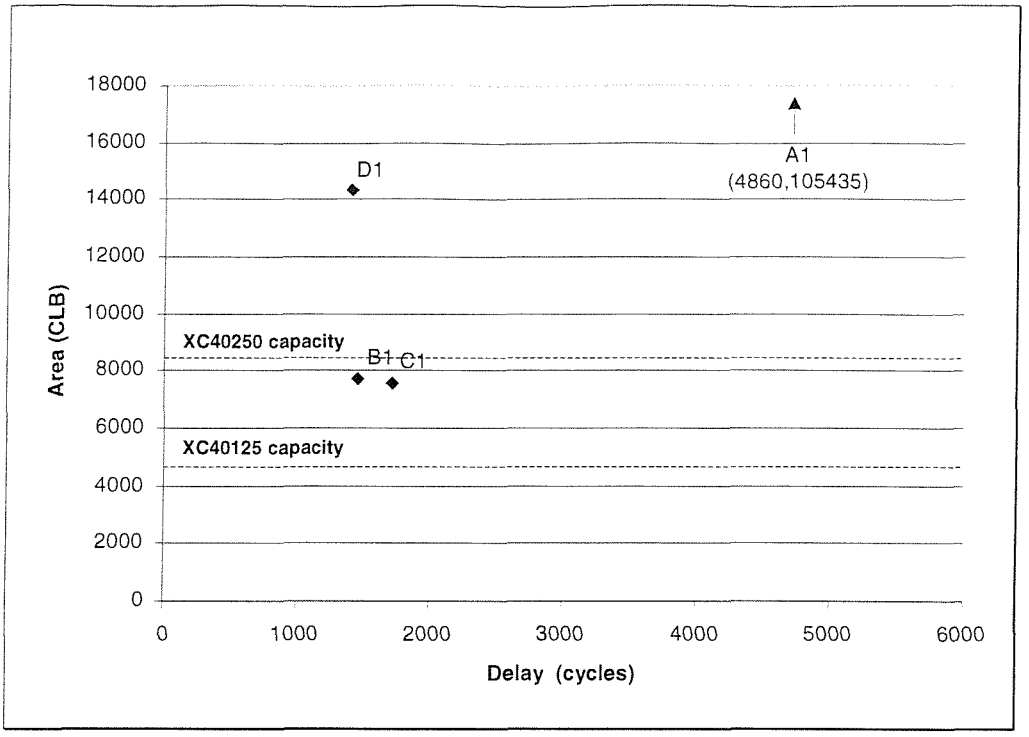


Figure 6.7 Design space for the original design

In principle, it could be expected that designs B1 and C1 would map successfully to the Xilinx XC40250XV system. However, the third party RTL synthesis tools [29,30] constantly failed to deliver a successful implementation.

To overcome this, an alternative approach was adopted; the design was manually partitioned into two blocks (arithmetic processor and controller).

Figure 6.8 is a block diagram showing the internal architecture of the partitioned core unit. The design splits into two main processors in a master-slave combination. The controller is responsible for controlling the data transfer through the system, and also provides the data and control signals required to decide the required operation to be performed in the arithmetic processor. This unit acts as a floating-point arithmetic unit that performs one of eight floating-point operations on a set of input variables passed by the controller according to the value of a *control vector*. The control vector values and the related floating-point operation are summarised in Table 6.2.

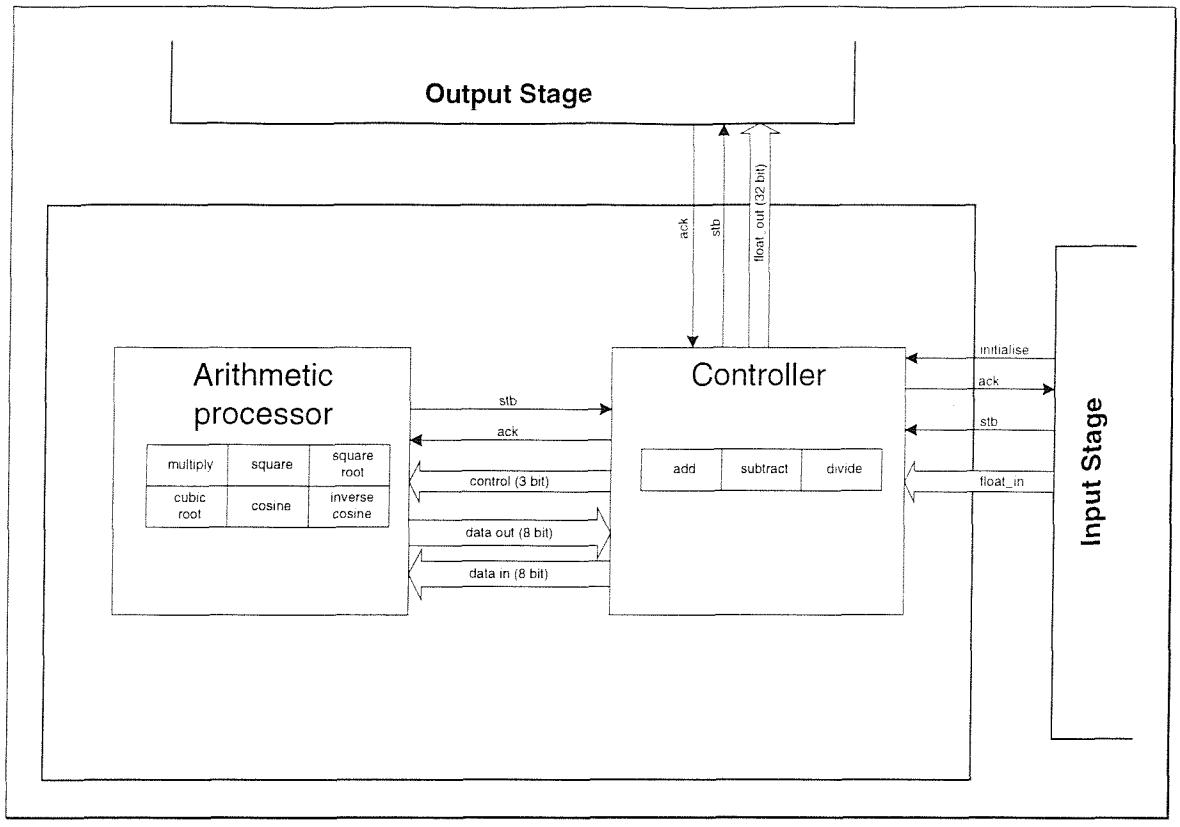


Figure 6.8 Partitioned core unit block diagram

The distribution of floating-point operations between the two units in Figure 6.8 is largely arbitrary; the chosen partitioning has the merit of keeping the unit sizes approximately equal.

Control vector	Operation	Summary
000	Multiply2	Read two input variables and output their product.
001	Square	Read a single variable and output the square.
010	Multiply3	Read three input variables and output their product.
011	Multiply4	Read four input variables and output their product.
100	Square root	Read a single variable and output the square root.
101	Cubic root	Read a single variable and output the cubic root.
110	Cosine	Read a single variable and output the cosine.
111	Inverse cosine	Read a single variable and output the inverse cosine.

Table 6.2 Arithmetic processor operations

Figure 6.9 shows the design space trajectories for the two designs. Table 6.3 shows the details of the eight points in Figure 6.9.

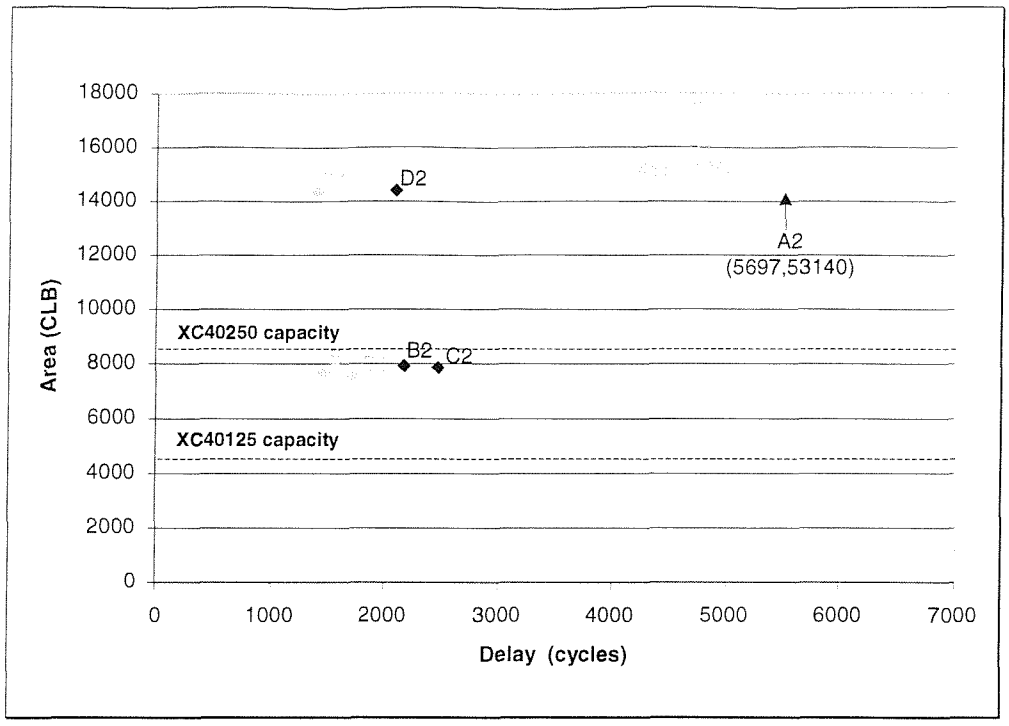


Figure 6.9 Core unit design space

Design		Target area (μm_2)	Available external ROM (Kbyte)	Total area (CLBs)	Total delay (cycles)
Original design	A1	N/A	N/A	105435	4860
	B1	0	27.7	7697	1457
	C1	0	0	7548	1719
	D1	∞	Not used	14321	1403
Partitioned design	A2	N/A	N/A	53140	5697
	B2	0	27.7	7907	2168
	C2	0	0	7849	2465
	D2	∞	Not used	14400	2087

Table 6.3 Parameters for the design space of the original and partitioned designs

The variation in the area cost of the original and partitioned design is largely dependent on sharing the functional units at both the floating-point building blocks level and the sub-component level. Different versions of the original design always deliver the most area efficient implementation for a certain set of constraints. A relatively small increase in the area cost of the partitioned design occurs due to the replication of some fixed-point building blocks within its two units.

Table 6.4 represents the partitioned design at different levels during the design synthesis flow[1]. The floating-point optimiser realises the floating-point functions within the design datapath in terms of floating-point primitives. The number of these primitives within a design is represented by the floating-point primitives (building blocks) column. The physical floating-point primitives column represent the number of unique floating-point primitives within a design. The MOODS synthesis system realises the initial design datapath in terms of virtual functional units, which are mapped during the MOODS optimisation phase onto a number of physical functional units. Finally, the third party tools map the MOODS datapath output into a number of CLBs (virtual CLBs), which gets optimised by third party tools to deliver the final implementation (physical CLBs).

Design	Floating point functions	Virtual floating point primitives	Physical floating point primitives	Virtual functional units	Physical functional units	Virtual CLBs	Physical CLBs (datapath)*	Physical CLBs (controller)
	1	2	3	4	5	6	7	8
A2	38	43	11	3497	450	9709	5710	2197
C2				3472	438	9665	5517	2332
D2				3465	439	16142	12259	2141

Table 6.4 Parameters of the design space of the partitioned design

The third party tool gain (i.e. column 6 to column 7 + column 8)is not much when compared to the MOODS synthesis system improvement (i.e. column 4 to column 5). The gain is mainly achieved by flattening the MOODS output hierarchy and optimising the combinational logic among these blocks. This suggests that integrating a logic optimisation algorithm within MOODS will eliminate the need for a third party synthesis tool and allow MOODS to target the Xilinx placement and routing step directly.

More details regarding the operation of the core unit may be found in the source code listing in Appendix E.

---

[1] Similar details could not be produced for the original design and the unoptimised design (A2) due to limitations imposed by the stability of the third party synthesis tools.

6.3 Synthesis issues

This section represents a number of issues related to the design and synthesis of the floating-point cubic equation solver. The section is divided into five units:

1. Area reduction techniques that can be used to reduce the total area cost of the design.
2. Techniques to meet timing specifications of certain units.
3. Synchronisation and communication between the design components and the modifications required to the structural output generated by MOODS.
4. Physical implementation issues.
5. The final implementation.

6.3.1 Area reduction

The FPGA targeted in this project imposed a significant limitation on the total design area. In order to meet the target cost some degree of compromise between the total design area and performance had to be made.

The main technique to reduce the total area cost is controlling the expansion process of the design expanded modules within the synthesis design flow. MOODS allows the user to expand the internal modules at any stage of the optimisation phase. It also provides user control over the level of expansion to be performed. The results presented in Table 6.5 describe two structural representations of arithmetic processor (optimised for area with external ROM) in the cubic equation solver optimised using two different techniques:

1. The design hierarchy was ***flattened*** completely ***before*** the optimisation phase.
2. The expansion process was controlled to allow maximal sharing of hierarchical units during expansion.

of 48.7% in the total storage cost, a reduction of 57.8% of the total number of functional units, and a reduction of 37.4% of the total interconnect cost.

To summarise, when a design with a minimum area cost is required, the following empirical optimisation sequence is found to be best:

1. Perform an initial optimisation to allow sharing the floating-point functional units.
2. Expand at one level ***only*** to ensure that fixed-point components within the floating-point units are ***not*** expanded.
3. Perform a second optimisation phase to allow sharing the fixed-point units.
4. Completely flatten the design hierarchy by expanding any remaining modules.
5. Perform a final optimisation having minimum area cost as the highest priority.

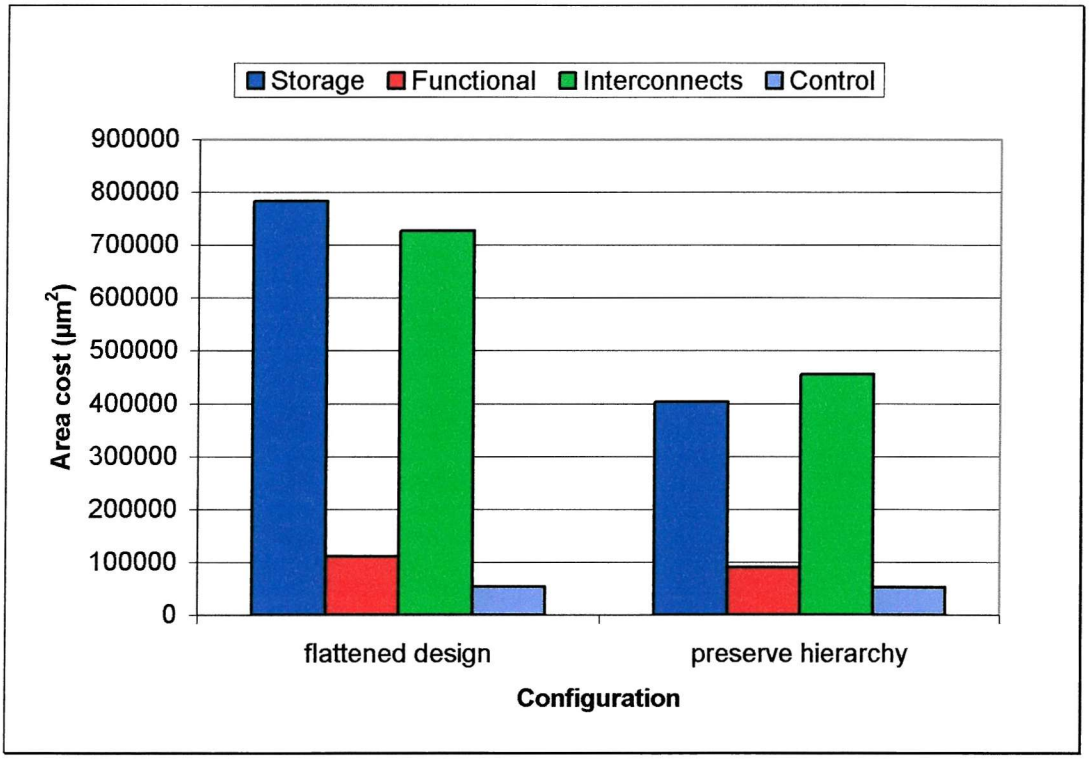


Figure 6.11 Area breakdown of both designs

Finally, the use of subprograms at the VHDL behavioural description level is recommended. Besides increasing the readability and maintainability of the design, VHDL subprograms play a role in reducing the total area cost of the design. Combining repeated portions of code in a single segment results in a reasonable area reduction mainly due to

Method	Total area (μm^2)	Total delay		Storage			Functional			Interconnects			Control	
		(ns)	Cycles	units	bits	area (μm^2)	units	bits	area (μm^2)	units	bits	area (μm^2)	units	area (μm^2)
Flatten	1674519	19596	138	821	7813	783595	638	8851	110855	150	3969	726570	825	53499
Preserve hierarchy	1000512	38482	271	388	4005	402798	269	3671	89685	103	2389	455130	522	52899

Table 6.5 Result of the two different techniques to optimise unit2

Based on the design space in Figure 6.10, it is clear that both optimisation techniques provide a significant enhancement to the design performance when compared to the initial design, with the first technique resulting in an area reduction of 83.9% and a delay reduction of 34.3%, while the second reduces the total area by 72% and the total delay by 66.5%. The second method provides the smallest design at the cost of some system performance degradation when compared to the first. This is due to the initial optimisation performed prior to any expansion allowing 100% sharing of the floating-point functional units. This early binding decision reduces the possibility of successfully applying delay optimisation transformations to the design resulting in less efficient delay optimisation.

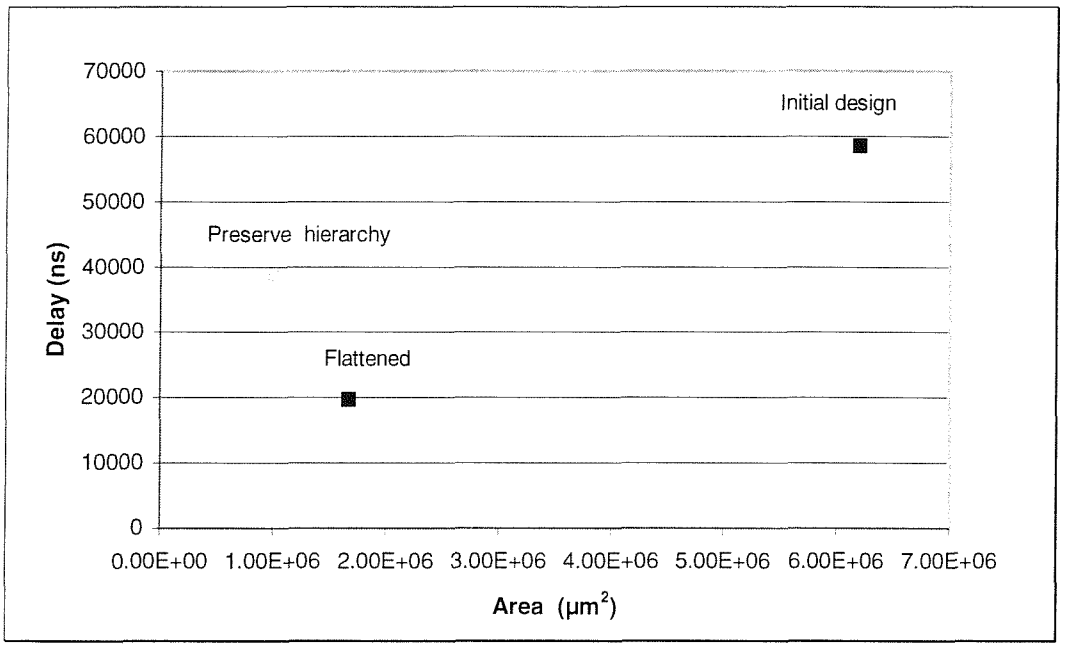


Figure 6.10 Alternative optimisation strategies

Examining the area breakdown of both structural representations in Figure 6.11 shows that forcing the optimisation algorithm to share the floating-point units allows a reduction

of 48.7% in the total storage cost, a reduction of 57.8% of the total number of functional units, and a reduction of 37.4% of the total interconnect cost.

To summarise, when a design with a minimum area cost is required, the following empirical optimisation sequence is found to be best:

1. Perform an initial optimisation to allow sharing the floating-point functional units.
2. Expand at one level ***only*** to ensure that fixed-point components within the floating-point units are ***not*** expanded.
3. Perform a second optimisation phase to allow sharing the fixed-point units.
4. Completely flatten the design hierarchy by expanding any remaining modules.
5. Perform a final optimisation having minimum area cost as the highest priority.

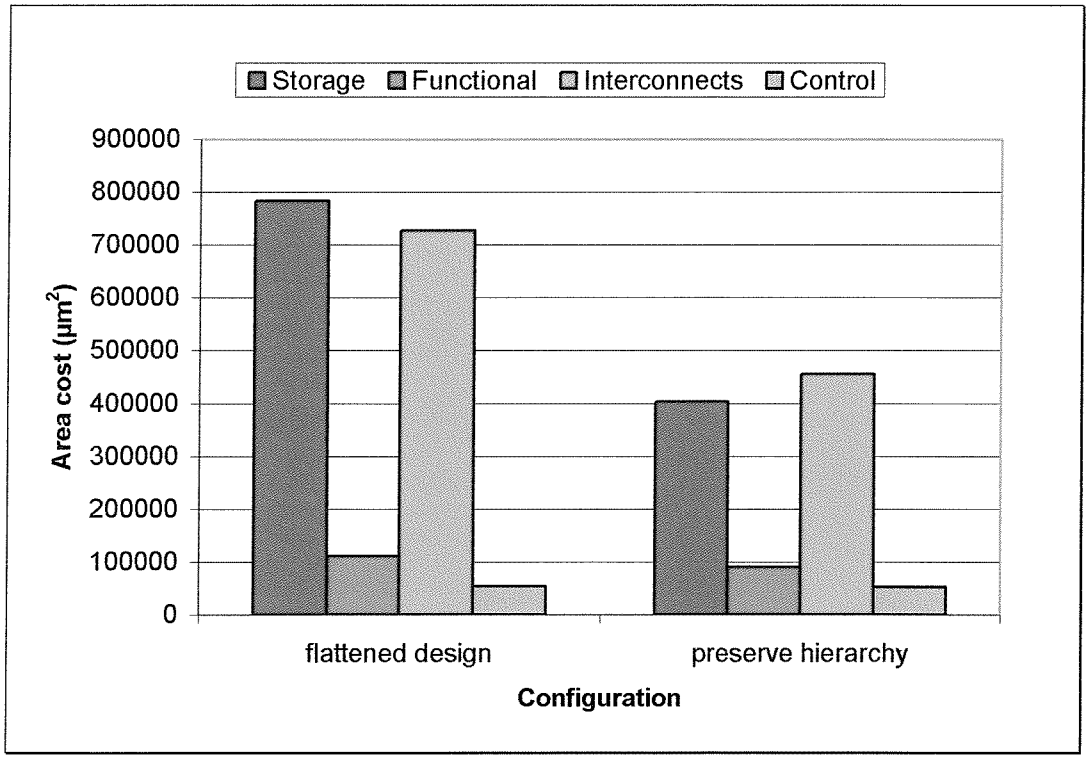


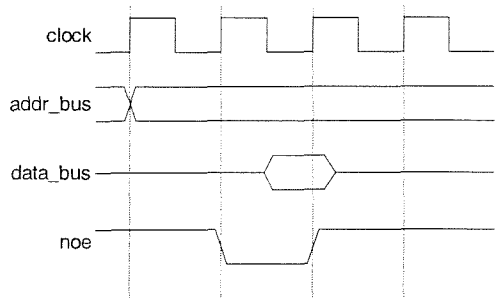
Figure 6.11 Area breakdown of both designs

Finally, the use of subprograms at the VHDL behavioural description level is recommended. Besides increasing the readability and maintainability of the design, VHDL subprograms play a role in reducing the total area cost of the design. Combining repeated portions of code in a single segment results in a reasonable area reduction mainly due to

be written. Three external ports are also required: *data_bus* and *addr_bus*, which connect to the external ROM busses and *noe* which controls the external ROM bus. Meeting the timing specification identified in Figure 6.13b requires executing instruction *i1* in the first clock cycle, then *i2* in the second clock cycle, and finally both *i3* and *i4* in the last clock cycle. This can be simply achieved by manual scheduling prior to saving the module or by inserting a *protect* command between these instructions.

```
      PROGRAM      ROM_read   addr,data,addr_bus,data_bus,noe

      INPORT       addr       [0:13]
      INPORT       data       [0:27]
      OUTPORT      data_bus   [0:27]
      OUTPORT      addr_bus   [0:13]
      OUTPORT      noe        [1:1]


i1:   MOVE         addr, addr_bus

i2:   MOVE         #0, noe

i3:   MOVE         data_bus, data
i4:   MOVE         #1, noe

      ENDMODULE    ROM_read
```

a)ROM_read macro port ICODE



b) Timing specifications

Figure 6.13 Macro port example

6.3.3 Synchronisation and communication

Data transfer between different units of the cubic equation solver is achieved by a handshaking protocol based on two handshake signals: *stb* and *ack*. Both signals are high when the system initialises. The master unit output data and asserts the *stb* signal low. The slave unit detects the change in the *stb* line, reads the data and changes the state of *ack* from high to low. The master unit then detects the change in the *ack* signal and asserts *stb* high. Finally, *ack* is asserted high as a consequence of the *stb* signal being high. This handshaking process is represented by the waveforms in Figure 6.14.

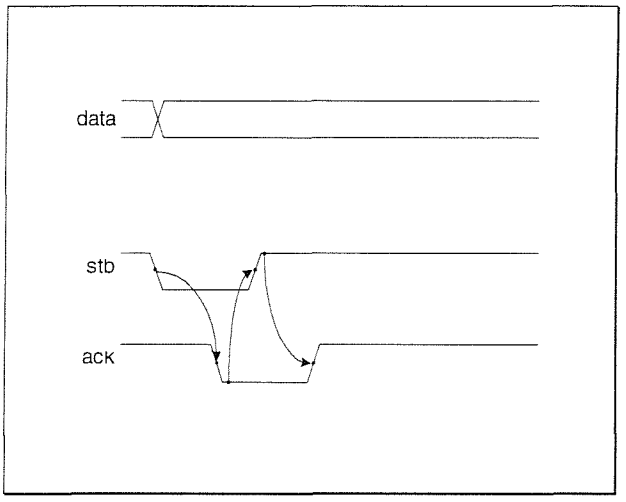


Figure 6.14 Handshaking signal waveform

Implementing this protocol in a VHDL behavioural description core requires a method to detect signal transitions. VHDL provides two statements for this purpose: *wait on signal* and *wait until condition*. *Wait on* terminates only when a transition occurs on the monitored signal, and *wait until* terminates when the condition changes from false to true.

A major problem arises from using these wait statements to synchronise two units. For example, if a *stb* signal goes low and the slave unit has not yet reached its monitoring state, the system will halt with the slave detecting a zero on the strobe line and the master waiting for a transition on the acknowledge line. The problem can be solved by providing the wait statement within a conditional block as represented in Figure 6.15c. The conditional block will ensure that the execution will continue if the transition on the handshaking signal has already occurred.

Another problem appears when two different clocks drive two units of the same system. External signal synchronisation in this case becomes a major issue. The problem can be represented with the aid of Figure 6.16 which shows the relationship of the flip-flop timing parameters: setup and hold times in this figure are denoted by t_s and t_h respectively.

```
wait on control;
```

a)wait on example

```
wait until control = '0';
```

b)wait until example

```
if control = '1' then
      wait until control = '0';
End if;
```

c) wait within a conditional block

Figure 6.15 Synchronisation within VHDL

The decision window is the interval when the flip-flop samples its inputs and decides on a change of output. If the input changes within this decision window, the flip-flop may go into a third *metastable* state half way between zero and one. The length of time it can remain in this state is theoretically unbounded [103].

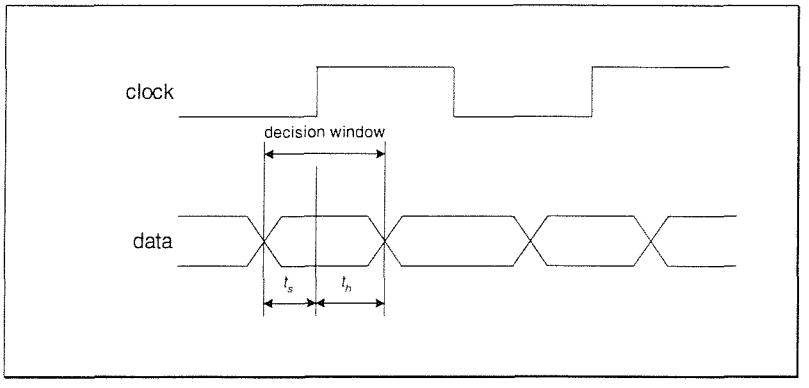


Figure 6.16 Flip-flop timing parameters

To reduce the probability of entering a metastable state, the synchroniser shown in Figure 6.17 is used. The input to the first flip-flop may violate the setup and hold time constraint

and drives the flip-flop into metastability for an arbitrary time. As long as the clock period is greater than this time, the flip-flop output becomes stable and the second flip-flop provides a synchronous copy of the initial input at its output.

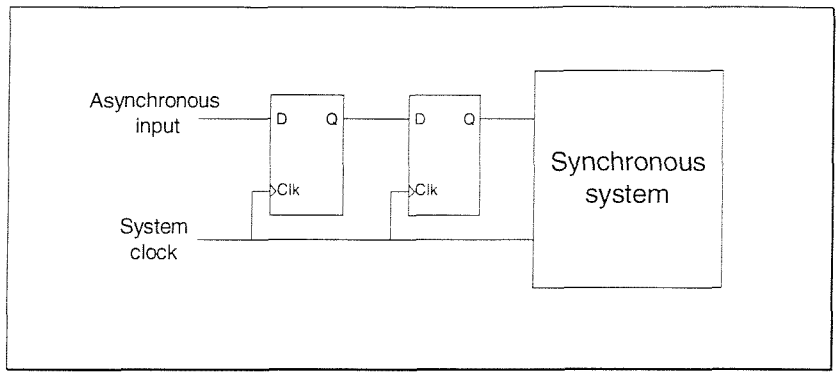


Figure 6.17 Synchroniser schematic

6.3.4 Physical implementation issues

Once the MOODS structural representation of the cubic equation solver has been simulated and verified, the system can be built. At the final stage, a major problem based on the multiplexor cost appeared. The MOODS synthesis system provides two possible implementations of the multiplexor, illustrated in Figure 6.18: a normal multiplexor with ***unencoded*** select input; and a multiplexor based on a set of tri-state buffers.

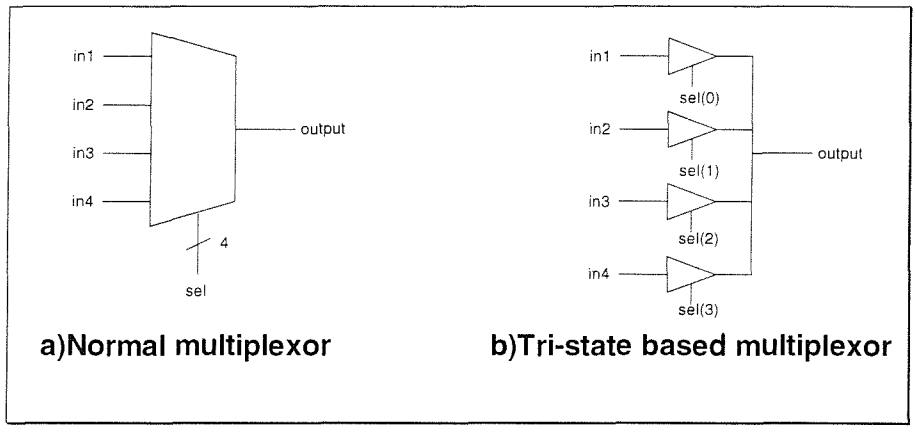


Figure 6.18 MOODS multiplexors models

Switching between these two models has a major effect in the total area cost: see Table 6.6, which represents the two parts of the cubic equation solver (FPGA1 consists of the controller, the input stage and the output stage, FPGA2 is the arithmetic processor) targeting both ASIC and FPGA, and for different combinations of multiplexors. When

targeting an ASIC, switching between the two models had a relatively small effect on the total system cost, with area varying by 16.5% for the first part of the design, and 9.9% for the second part. The increases in area cost arise from the extra cost of implementing multiplexors based on tri-state buffers, which is more expensive than the general approach based on pass transistors.

Design	Implementation	Tri-state buffers used to implement multiplexors	Total area CLBs/Gates
FPGA1	ASIC	13740	44910
		0	37461
	FPGA	13740	1514
		0	6342
		10302	4833
FPGA2	ASIC	10503	40850
		0	35826
	FPGA	10503	2419
		0	5497
		7219	4670

Table 6.6 Comparison of area cost based on multiplexors modification

When targeting Xilinx FPGAs, the area variation when switching between the two multiplexor models increases noticeably, with an increase of the total area cost of 76% in the FPGA1 and 56% in the FPGA2 when implementing multiplexors using the normal model rather than the tri-state based model. This is expected, since the limited number of multiplexors in the FPGA block forces the tool to implement multiplexors using combinational logic blocks, resulting in a great inefficiency and area cost inflation. Balancing the number of multiplexors based on each model is essential for a successful implementation.

6.3.5 Final implementation

The floating-point cubic equation solver project introduces the MOODS synthesis system floating-point capabilities. It is also as a test vehicle to establish MOODS reliability in implementing large behavioural designs (100 000+ gates).

Along the way, a number of problems were discovered in MOODS. These are not detailed here, as they have been reported and remedied.

Unfortunately, the commercial Xilinx software that drives the FPGA mapper has consistently failed to successfully target the XC40125XV FPGA, even at moderate levels of utilisation. The problems encountered in the tool have supposedly been fixed; Xilinx has withdrawn software support for one of its own products, which has placed us in a difficult position. Eventually, a XC40250VX FPGA became available. The device has twice the capacity of the XC40125XV. However, a new range of problems related to the commercial tools appeared while trying to target this FPGA, and nothing could be done to fix these. Work rounds for these problems were far more problematic than they should have been.

With a single FPGA available, the obvious solution was to implement the original design, which represents the whole algorithm in a single building block and delivers the most area efficient implementation. However, the RTL synthesis tool consistently failed in delivering a successful implementation of the design. This made the partitioned design the only sensible way forward.

Moving to the placement and routing stage, a number of problems were encountered at this stage, with the same design processing time varying between two and ten days, which dominates the design cycle time when compared to the run times of the MOODS synthesis system and the RTL synthesis tool as illustrated in Table 6.7. Methods to speed up the process such as guiding the placement and routing with a previously routed design did not function correctly.

Design flow tool	Original design (hours)	Partitioned design (hours)
MOODS	10	1.5
RTL synthesis	Failed	7
Placement and routing	Failed	48-240

Table 6.7 Run time for tools used in the design flow

These problems made it necessary to modify the design again to further reduce the FPGA load. The output stage was moved to the FPGA that includes the VGA display driver. Two

interface units (interface1 and interface2) were introduced to control passing data between the two FPGAs. VHDL behavioural description of both interface units are listen in Appendix E. A block diagram showing the final version of the cubic equation solver is represented in Figure 6.19. Figure 6.20 represents the area utilisation figures of the FPGA.

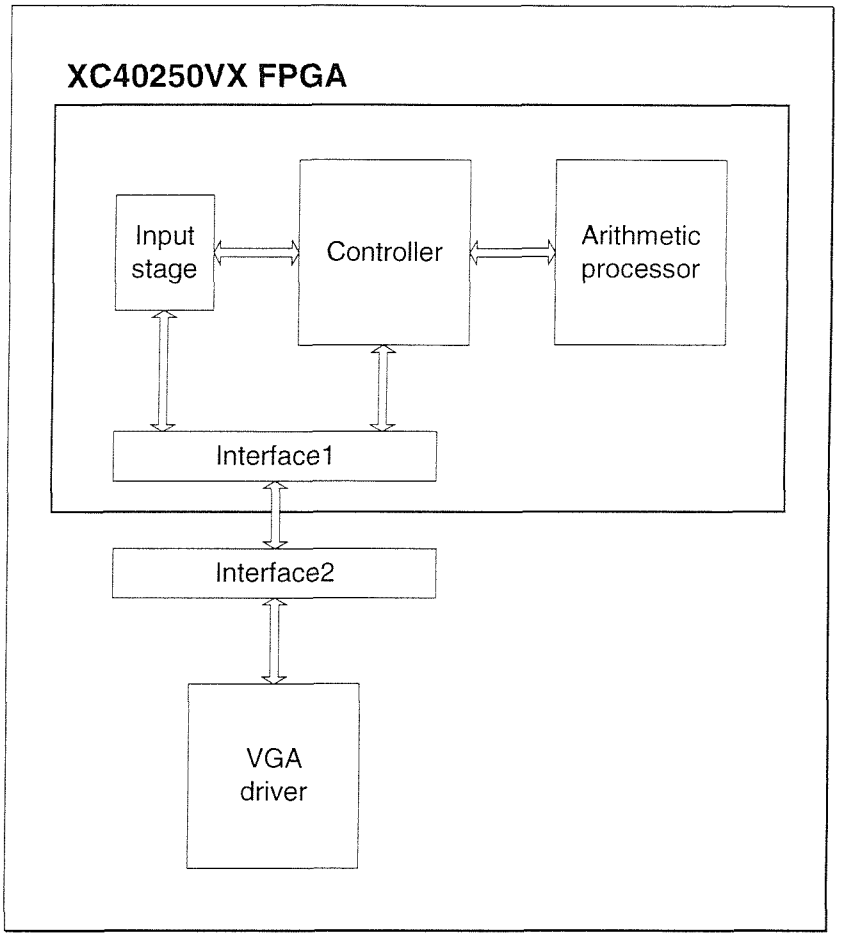


Figure 6.19 Final implementation block diagram

```
Total latches                          0 out of 16928
Total Flipflops                        6884 out of 16928
4 input combinational blocks           14764 out of 16928
3 input combinational blocks           5456 out of 8464
```

Figure 6.20 FPGA utilisation figures

6.4 Comparison with microprocessors

In recent years, advances in VLSI technology have lead to a dramatic increase in the floating-point performance of microprocessors, with the performance of the floating-point units of current computer increasing by a factor of 70 [104] in the last 10 years.

A floating-point arithmetic unit is implemented for comparison purposes. The unit performs one of seven floating-point operations (addition, subtraction, multiplication, division, square root, sine, and cosine). Targeting the AMS 0.35μ CMOS technology, the total area cost of the design is 35000 gates. Comparing the area cost of this unit to the size of the floating-point unit in a Pentium III processor (around 1.8 million transistors)[2], indicates the possibility of great performance enhancement of the synthesis system floating-point units, especially with the rapid increase in programmable logic device capacity.

For each of the seven floating point operations, a C program was constructed to estimate the total number of clock cycles required to execute this operation on five different microprocessors. The synthesised design performance was realised from the simulation results of the synthesised structural VHDL .The comparison results are illustrated in Table 6.8 and Figure 6.21.

Unit	Platform	Add (cycles)	Sub (cycles)	Mult (cycles)	Div (cycles)	Sqrt (cycles)	Sine (cycles)	Cosine (cycles)
Synthesis System	N/A	17	20	19	79	20	45	45
80486DX2	DOS 6.22	37	37	37	94	320	772	790
AMD K6-2	Windows NT 4.0	24	24	24	24	24	24	24
Pentium	Windows 95	15	15	15	15	15	15	15
Pentium II	Windows NT 4.0	9	9	9	9	9	9	9
Pentium III	Windows NT 4.0	7	7	7	7	7	7	7

Table 6.8 Benchmark results of floating-point performance of different microprocessors and the MOODS synthesis system

[2] Area estimation is based on a die shot of the processor, and assuming the equal transistor density over the chip.

A second test is carried out to compare the floating-point performance of the cubic equation solver to the AMD K6-2 processor and Intel Pentium II processor. The algorithm was written in C and the number of clock cycles required to generate the final result is averaged over a number of input samples for both microprocessors. The same set of input samples is used to simulate the structural VHDL of the synthesised design to estimate its performance. The floating-point calculation time of the FPGA based unit varies between 1158-1668 clock cycles, compared to 800-840 clock cycles for the K6-2, and 661 clock cycles for the Pentium II processors. The results are represented in Figure 6.22.

Although the majority of the modern microprocessors outperformed the synthesised ALU in executing a single floating-point operation, these devices in general are fetch-execute architectures that have limitation on the number of instructions to be executed at a time. The result they produce for a specific design, in general, is far from optimal, because they are static in nature and designed for the general case. On the other hand, a synthesis tool has the capability to deliver a near optimal solution for a specific problem. Executing as many instructions as possible in parallel to increase the design's throughput to the limit that could exceed the peak performance of these processors. In addition, if synthesis tools could achieve comparable performance to microprocessors for a pure floating-point application, it is a good indication that these tools can be considered the main target of applications that require a few floating-point operations intermixed with fixed-point calculations.

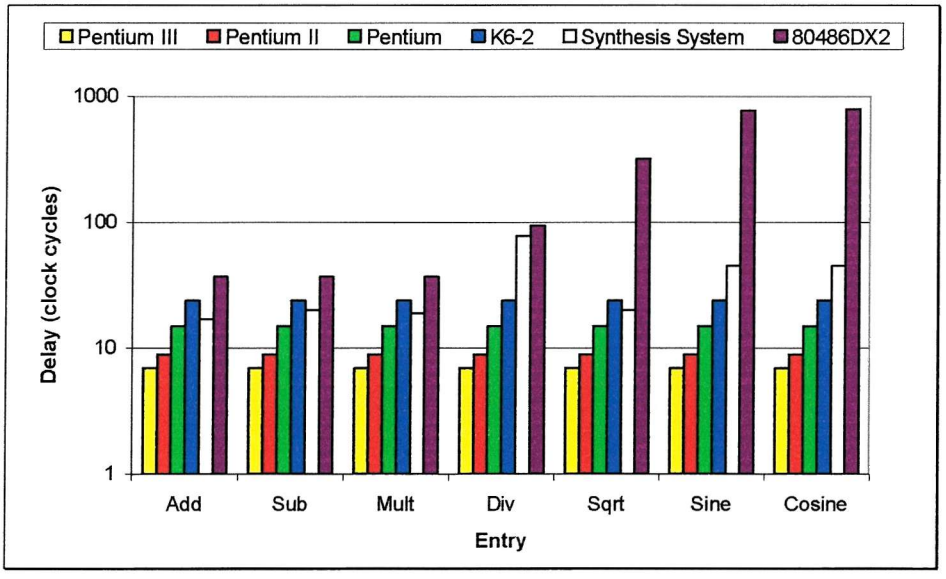


Figure 6.21 The floating-point performance of different microprocessors compared to the MOODS synthesis system

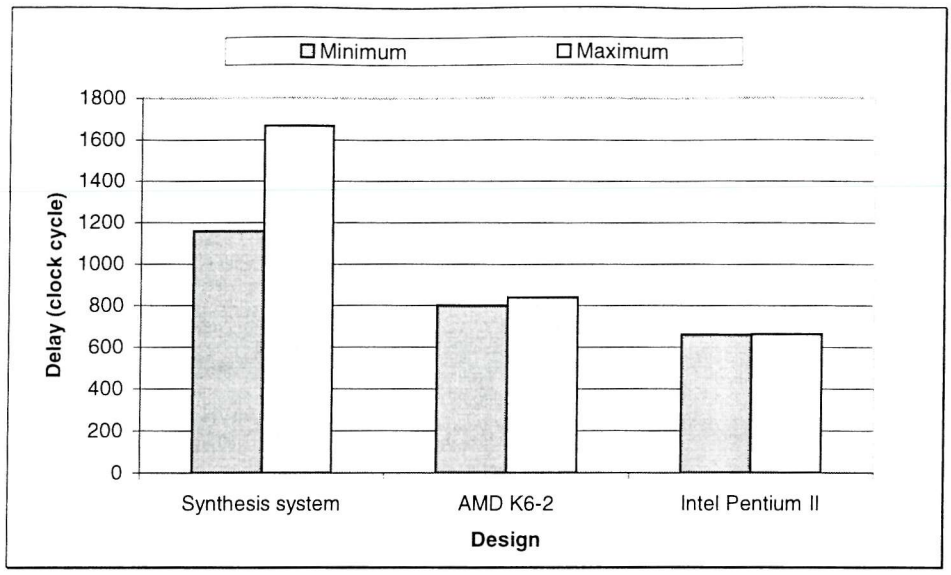


Figure 6.22 The cubic equation solver floating-point performance compared to modern microprocessors

Chapter 7

Conclusions and further work

The work described in this thesis extended the scope of the MOODS synthesis system to include floating point (both real and complex) data type manipulation. The floating point module library presented in Chapter 4 provided a wide range of function evaluators with significantly different physical properties, increasing the probability of constructing a design that meets the user pre-defined objectives.

Binding floating point functional units to suitable floating-point modules from the module library is carried out by a dedicated floating point optimiser. The optimiser is based on a heuristic algorithm derived from observations of floating point module interactions, and relies on a number of pre-calculated metrics that summarise the physical properties of each module.

The above enhancements are exploited to design and implement a physical demonstrator, an algebraic cubic equation solver with complex root capability, intended as a demonstration of the floating-point capabilities. It also demonstrates the capabilities of MOODS as a useful tool for handling relatively large circuits (>100 000 gates).

The work reported in this thesis opens the door to a number of research opportunities in the field of behavioural synthesis in general and floating-point synthesis in particular. The experience developed as a behavioural synthesis designer also suggests a number of enhancements to the behavioural synthesis tool to increase the productivity of the system. These enhancements and research suggestions are summarised in this chapter.

7.1 Source level optimisation from a floating-point perspective

Hardware engineers using behavioural synthesis tools tend to write code with more regard to clarity than optimal implementation. Using features such as constants and temporary variables enables the code to be easily understood and modified. However, this common approach adds a considerable overhead to the synthesised unit. To solve this problem, a floating-point source-level optimiser is required. It is a tool to apply a set of code-improvement transformations that target the floating-point expressions within the behavioural code.

A problem closely related to the floating-point source level optimisation is that floating-point numbers are an approximated representation of real numbers. Extra caution should be taken when exploiting algebraic identities to target floating-point arithmetic blocks [105, 106]. Consider for example: a reduction in the total cost can be achieved by replacing a more expensive operator with a cheaper one as in $x \div 4 \equiv x \times 0.25$. This suggests that $x \div 20$ should be replaced by $x \times 0.05$. However, the two equivalencies do not have the same semantics in floating-point arithmetic because 0.05 cannot be represented exactly in a floating-point representation, which introduces an extra source of error in the final result.

The example suggests that the source level optimiser should be extremely cautious when applying algebraic identities to real numbers. Rather than completely eliminating algebraic identity based transformations, an analysis of the entire expression is required every time a transformation is applied to ensure that the identity holds and that the error introduced will not affect the system accuracy.

7.2 Variable precision floating-point library

Another opening for further work is the issue of variable precision floating-point representations and variable precision floating-point library. As discussed in Chapter 5, error propagation through a floating-point expression is highly dependent on the arithmetic operations involved and the precision of the intermediate results. It is sometimes impossible to provide the required target accuracy for an arithmetic expression using

single precision floating-point operations. A trivial solution to this problem would be to provide a higher precision (for example double precision) for all floating-point operations involved. However, this solution would in general, add an unnecessary overhead to the total system cost. An ideal solution would be to calculate the required accuracy for each floating-point operation involved and to provide an appropriate floating-point data precision. To achieve that, two main enhancements are required:

1. It is not possible to provide a specific expanded module for each target precision, thus, an enhancement to the expanded module sub-system is required to provide some form of parameterised expanded module, allowing the various loop executions and variable widths to be specified during the synthesis runtime in terms of generic parameters. VHDL provides the capability of implementing parameterised units using generics [6]. For example, Taylor expansion can be employed to achieve every possible target precision by varying the number of terms involved, as well as the width of the intermediate calculations.

2. The accuracy of an arithmetic expression varies according to the input data type. To be able to identify the exact precision of each operation, the system will require the user to provide a test pattern consisting of a number of input samples. The test pattern should then be applied to the behavioural code in conjunction with a simulation environment and a detailed error analysis is performed for each input set. This way, the system can identify the worst-case error within the test pattern and adjust the precision of each floating-point operation to achieve the target accuracy.

7.3 Component library

The floating-point manipulation units provide the ability to integrate new floating-point functional units within the synthesis environment. It is possible to implement reusable floating-point algorithms that incorporate floating-point data manipulation using the hierarchical unit expansion capability. These two features provide a means of increasing the productivity of the synthesis tool by adding new high level components to the system.

There is a wide range of floating-point functional units that can be added to the module library. It is also possible to add new building blocks to generate functional units already available in the library. Table 7.1 suggests a number of these units [107].

Function	Description
ACOT (X)	Inverse cotangent of X.
ACOTH(X)	Inverse hyperbolic cotangent of X.
ACSC(X)	Inverse cosecant of X.
ACSCH(X)	Inverse hyperbolic cosecant of X.
Error Function	$erf\ x = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$
Gamma function	$\Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dt$

Table 7.1 Suggested floating-point library components

Hierarchical module expansion can be exploited to increase the scope of the complex functional units within the library. Further enhancements can be achieved by integrating a number of floating-point algorithms such as FFT processor cores or MPEG encoder/decoder. This can be taken further: pre-defined blocks of almost arbitrary complexity can be envisaged.

7.4 Function inversion block

A "function inversion block" is a functional unit that take as input a *value* and a *function*, and produce as output the **inverse** function value – see Figure 7.1

Providing this building block would enhance the functionality of the floating-point library by providing the ability to generate inverse functions that are not implemented within the current library. This block can also be used to generate the inverse of a mathematical expression that combines a number of floating-point functional units.

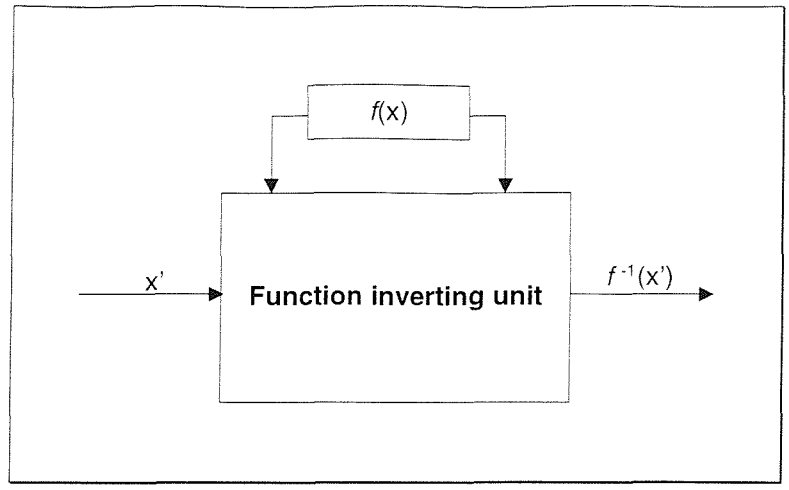


Figure 7.1 Function inversion block

A number of constraints must be applied to ensure successful implementation of the inverse function:

1. Continuous and monotonic input function.

2. The domain of the inverse function is restricted to match the range of the input function.

Given these constraints, two approaches are possible:

- Construct a generic function inversion block to numerically find the root of the equation: $f(x) - x' = 0$. Root finding methods [108] such as the bisection method, Newton method, Falsi method or secant method will form the core of the generic unit. The performance of such methods is largely dependent on the quality of the initial estimate of the solution. This requires some analysis during the synthesis process to identify the nature of the input function and provide a suitable initial solution, or even dividing the inverse function domain into a number of intervals each has its own initial solution.

- Alternatively, algebraic methods could be used to construct a formula for the inverse of the input function [109]. The inverse function can then be implemented as a hierarchical block during the initial compilation stage. The method is illustrated by the example in Figure 7.2.

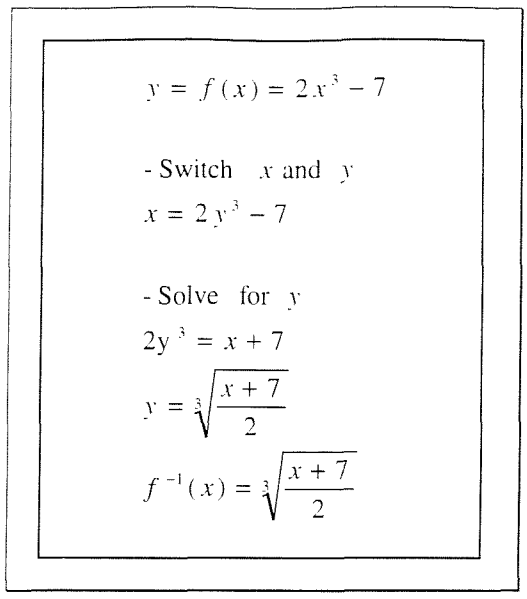


Figure 7.2 Constructing the inverse function algebraically

7.5 Multi-operand floating-point units

Multi-operand floating-point units would provide a significant enhancement to the floating-point synthesis capability. The method has already been used in modern microprocessors [110, 111] to speed up graphics manipulations which involve extensive use of floating-point calculations. The advantage of such building blocks is illustrated in the example in Figure 7.3, which represents a rotation of a point (x,y) by an angle θ. The number of building blocks involved in the expression evaluation is reduced from eight units to four. The number of temporary registers required to save intermediate results is also reduced from six to two[1]. This is also accompanied by a reduction in the number of times the intermediate results are normalised and rounded during execution.

[1] Although temporary registers are often shared during the synthesis process to reduce the total cost. There is always an increased cost in the form of multiplexors and control logic required sharing these registers.

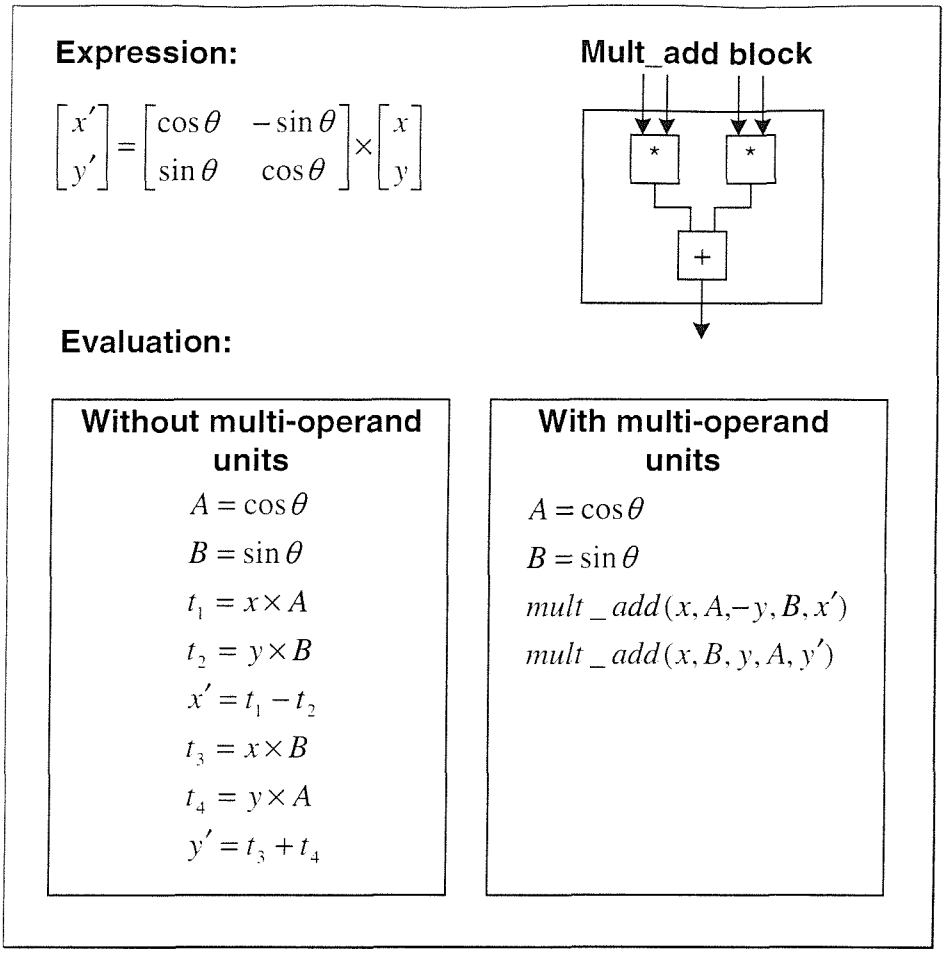


Figure 7.3 Multi-operand floating-point unit example

It is important to note that in order for the system to fully exploit such multi-operand units, it should have the ability to re-arrange the floating-point expressions within a design in a way that allows mapping to these units. For example, detecting the two multiplication and single addition combination in the previous example and map it to the single mult-add block.

Appendix A

IEEE standard for binary floating-point arithmetic

The IEEE floating-point standard [41] is the most widely used representation for floating-point numbers. This appendix provides an introduction to this standard with an emphasis on the single precision representation.

The general representation of a floating-point number in this standard is shown in Figure A.1. It represents a number of the form: $(-1)^s \times 1.F \times 2^{E-bias}$. The representation is divided into three fields:

- ***Sign (s)***: A sign bit field indicating the sign of the floating-point number. $s = 1$ represents a negative number, while $s = 0$ for positive numbers.
- ***Biased exponent***: An unsigned integer field representing the sum of the exponent and a constant (bias). The bias is introduced to make the field range non-negative (i.e. zero in this case represents the most negative value).
- ***Fraction***: an unsigned field containing the significant bits to the right of the binary point. Note that the fraction field does not include the leading digit in the significand, as it is assumed to be always one and is implied in the format.

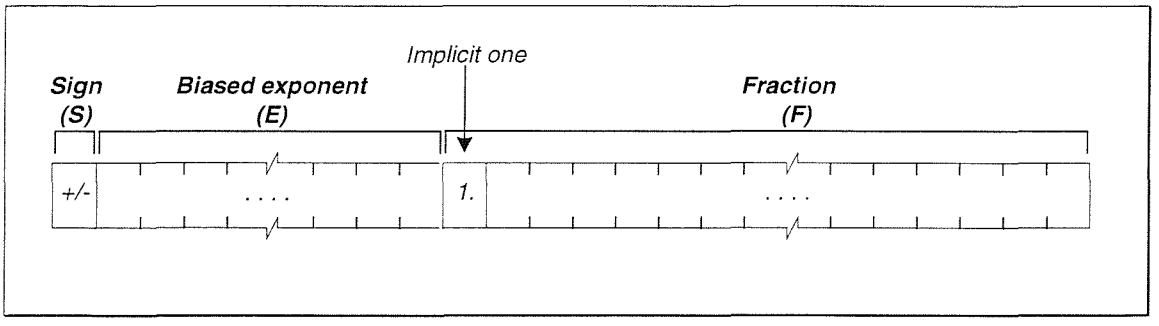


Figure A.1 Floating-point number representation

According to the width of the biased exponent and the fraction, the standard defines four different formats: single; single extended; double; and double extended. These are summarised in Table A.1.

Format	Biased exponent width	Bias value	Fraction field width	Total width
Single	8-bit	+127	23-bit	32-bit
Single extended	≥11-bit	Unspecified	≥31-bit	43-bit
Double	11-bit	+1023	52-bit	64-bit
Double extended	≥15-bit	Unspecified	≥63-bit	79-bit

Table A.1 Floating-point format parameters

A.1 Single-precisio n format evaluation

A single precision floating-point number has the general form:

$$(-1)^{s} \times 1.F \times 2^{E-bias}$$

For example numbers +4.75, -0.125 are represented as:

$$+4.75 = +100.11_2 \times 2^0$$
$$= +1.0011_2 \times 2^2$$
$$s = 0$$
$$E = 2_{10} + 127_{10}$$
$$= 129_{10}$$
$$= 10000001_2$$
$$F = 00110000000000000000000$$

Final bit pattern = 01000000100110000000000000000000

$$-0.125 = -0.001_2 \times 2^0$$
$$= -1.000_2 \times 2^{-3}$$

$$s = 1$$
$$E = -3_{10} + 127_{10}$$
$$= 124_{10}$$
$$= 01111100_2$$
$$F = 00000000000000000000000$$

Final bit pattern = 10111110000000000000000000000000

The general form represented in the previous examples is what the standard defines as *Normalised Numbers*: a representation of a number with a magnitude greater than or equal to 2^{-126} and less than 2^{128}. In addition to normalised numbers, certain bit patterns in the standard have a specific representation, as shown in Figure A.2:

- ***Zero***: The value represented by an all zero exponent field and an all zero fraction field. Zero can have either a negative or positive sign.

- ***Denormalised number***: A denormalised number indicates a quantity with magnitude less that 2^{-126}, but greater that zero. It is represented by a zero exponent field and a non-zero fraction field.

- ***Infinity***: Infinity is interpreted in the affine sense, that is,minus infinity is smaller than any finite number and plus infinity is greater than any finite number. Infinity is represented by an all zero fraction field and an all one exponent field.

- ***Not A Number (NAN)***: "not a number" is defined as a pattern indicating an invalid operation. Two types of NAN are provided: Signalling NAN and Quiet NAN. Signalling NAN is represented by an all one exponent field with the fraction field most significant bit set to one. A quiet NAN is represented by an all one exponent field, a zero in the fraction most significant bit and at least one one in any of the fraction least significant bits.

This is summarised in Table A.2.

Sign bit (s)	Exponent (E)	Fraction (F)	Value
0/1	0	0	(+0,-0)
0/1	0	F	$(-1)^s x(0.F)x2^0$
0/1	0 < E < 255	F	$(-1)^s x(1.F)x2^{(E-127)}$
0/1	255	0	$+\infty,-\infty$
0/1	255	F(22)=1	Signalling NAN
0/1	255	F(22)=1 F(21:0)≠ 0	Quiet NAN

Table A.2 Reserved bit patterns

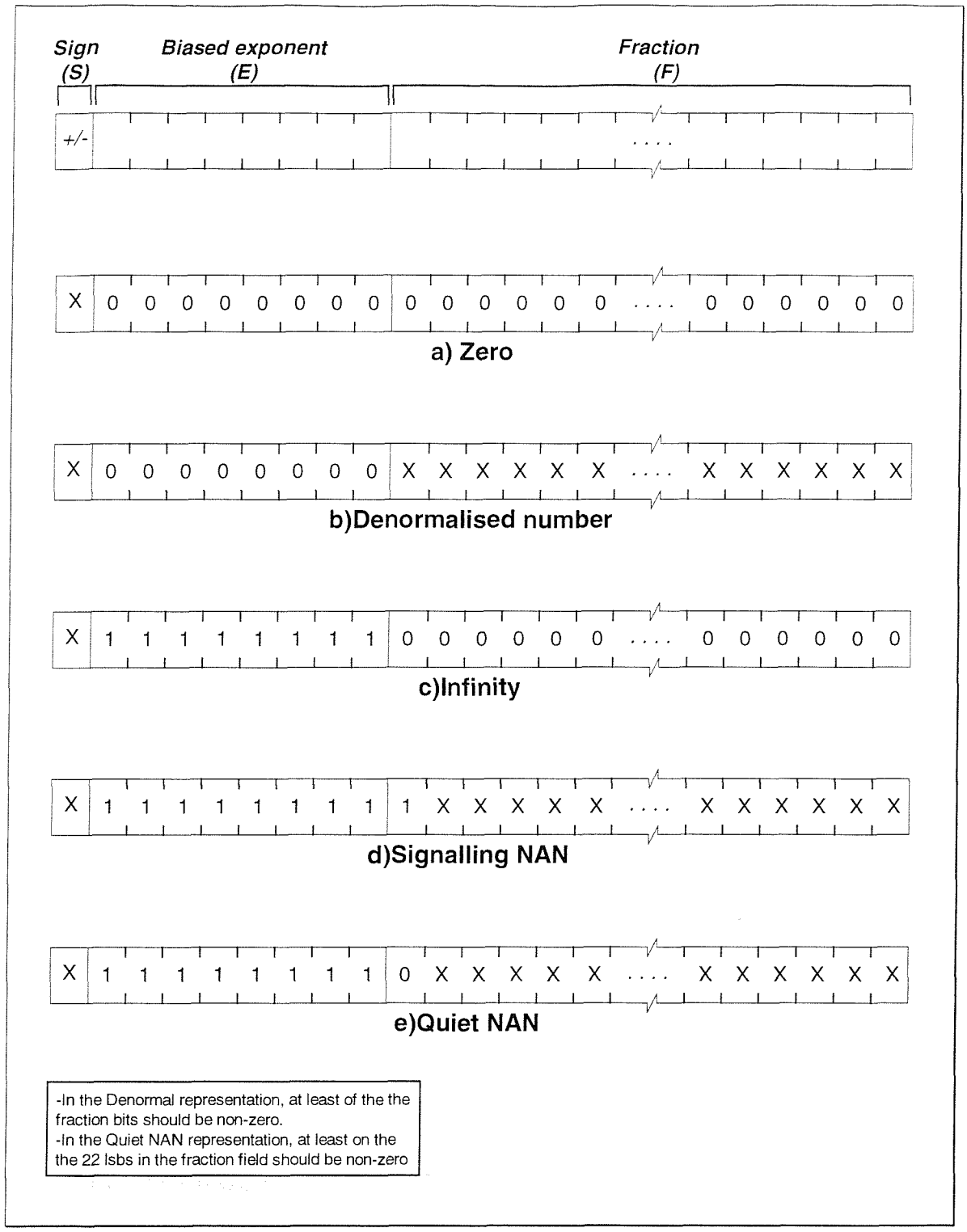


Figure A.2 Floating-point number bit patterns

A.2 Operations with NAN

"Not A number" does not represent a numerical value, instead it is a symbolic representation of an invalid result. NAN is provided in two forms: Quiet NAN and Signalling NAN.

A Quiet NAN indicates an invalid output result (e.g. $+\infty + -\infty$). If a Quiet NAN appears as an input operand to an operation, the final result will also be a Quiet NAN.

Signalling NAN is never produced as an output result from a floating-point operation. It is provided as an indication for specific situations such as uninitialised variables. If a Signalling NAN appears as an input operand, the output result would be a Quiet NAN.

Invalid floating-point operations that produce NAN as the final result are listed in Table A.3

Operation	Input operand	Final result
Addition	$(+\infty) + (-\infty)$	Quiet NAN
Addition	$(-\infty) + (+\infty)$	Quiet NAN
Subtraction	$(+\infty) - (+\infty)$	Quiet NAN
Subtraction	$(-\infty) - (-\infty)$	Quiet NAN
Multiplication	$(+0) * (+\infty)$	Quiet NAN
Multiplication	$(+0) * (-\infty)$	Quiet NAN
Multiplication	$(-0) * (+\infty)$	Quiet NAN
Multiplication	$(-0) * (-\infty)$	Quiet NAN
Addition	signalling NAN	Quiet NAN
Subtraction	signalling NAN	Quiet NAN
Multiplication	signalling NAN	Quiet NAN
Division	signalling NAN	Quiet NAN
Division	$\pm\infty/\pm\infty$	Quiet NAN
Division	0/0	Quiet NAN

Table A.3 Floating-point invalid operations

A.3 Status flags

Five status flags are required to monitor the execution of floating-point operations. Setting one of these flags indicates an exceptional situation detected while executing the operation. The following is a summary of the status flags indications:

- **Invalid operation flag**: The invalid operation flag is set high if an input operand is invalid for the operation. The result in that case would be a Quiet NAN. The invalid operation flag is signalled in all the situations listed in Table A.3.

- **Zero Division Flag**: The zero division flag is high if the divisor is zero in a floating-point division operation. If the dividend does not equal to zero then the final result would be a correctly signed infinity, otherwise the operation is invalid and the output is a Quiet NAN.

- **Underflow Flag**: If a floating-point operation produces a result of a magnitude too small to be represented as a single-precision floating-point number, the operation underflows and the underflow flag is set. It is an indication that the output result has a magnitude greater than zero, but cannot be represented as a floating point number. The output in this case is a correctly signed zero.

- **Overflow Flag**: The overflow flag is set high if an operation on finite input operands produces an output result too large to fit in the single precision format. Overflow occurs if the output result has a magnitude greater than or equal to 2^{128}. The output in this case is a correctly signed infinity.

- **Inexact Flag**: The inexact flag is high if the output of a floating-point operation does not equal the infinitely precise result. On other words, it is an indication that the final result has been ***rounded*** or ***approximated***. Inexact flag is also high if an underflow or overflow occurs.

A.4 Comparison operations

Floating-point comparison operations are exact, and never overflow or underflow. The implementation is required to support four relational operations: *less than*; *equal*; *greater than*; and *unordered*. The last operation is the result of comparing any floating-point

representation to a NAN. Every NAN should compare unordered to any other floating-point representation including another NAN.

A comparison operation can be delivered in one of two ways:

1. As a single unit that performs all the four comparison operations and provides a conditional vector identifying all the four possible relationships mentioned above.

2. A true or false block representing one of the four relationships or a combination of them (e.g greater than or equal).

In addition to the comparison true-false response, an invalid flag should be raised whenever a NAN is provided as an input to any of the comparison operations that does not involve *unordered.*

A.5 Rounding

Rounding is the process by which the result is approximated to a representation that fits in the destination formats. The IEEE standard specifies four rounding modes [43,112]: round to the nearest; round towards +infinity; round towards –infinity; and round towards zero.

Round to the nearest is the IEEE standard default rounding mode. In this mode, the result is rounded to the closest representation that fits in the destination format. If the result is exactly half way between two representations, it is rounded to the representation that has a least significant bit of zero. Figure A.3 illustrates three examples of rounding to the nearest. The first result X1 is to the nearest representation *a*, while X2 is rounded to *b*. X3 represents a special case since it lies half way between *c* and *d*, therefore it is rounded to the representation that has a least significant bit of zero (*d*).

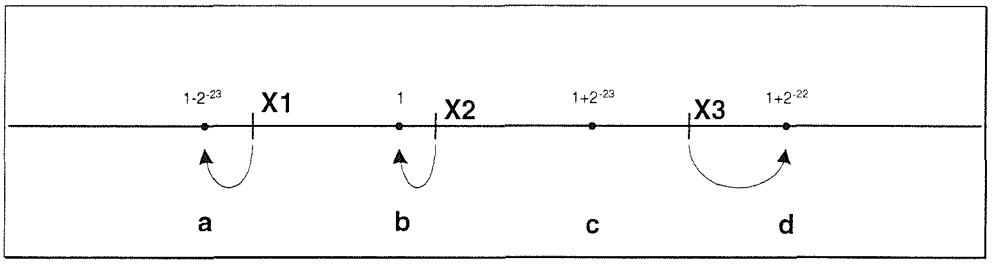


Figure A.3 "Rounding to the nearest" examples

The second IEEE rounding mode is *round towards +infinity*. In this mode, the result is rounded to the closest IEEE format representation that is greater than or equal to the output result. This is illustrated in Figure A.4. X5 cannot be represented exactly in floating-point format and is rounded to the next larger floating-point representation (*f*). The same occurs on X5 where it is rounded to *g*, the result represented by X6 fits in the target format and therefore no rounding takes place.

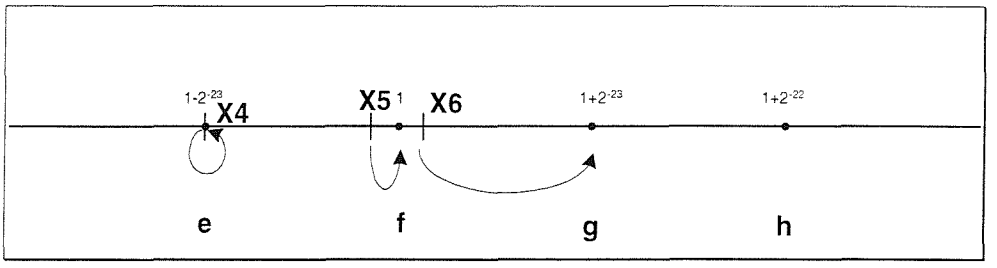


Figure A.4 "Rounding toward +infinity" example

Round towards –infinity is the third IEEE rounding mode. In contrast to the previous rounding mode, it rounds the final result to the closest floating-point representation that is less than or equal to the output result. X7 and X8 in Figure A.5 illustrate this rounding mode.

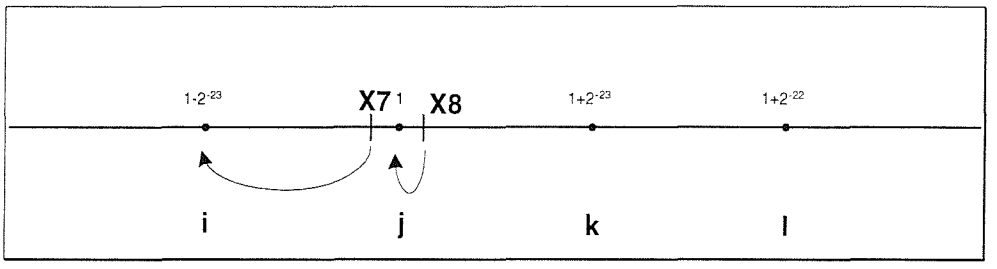


Figure A.5 "Rounding toward -infinity" example

In the final rounding mode, *round toward zero*, the result is rounded to the closest floating-point representation whose ***magnitude*** is less than or equals the output result. This mode is represented in the example in Figure A.6, where X9 is rounded to -1.0×2^{-23} and X10 is rounded to 0.

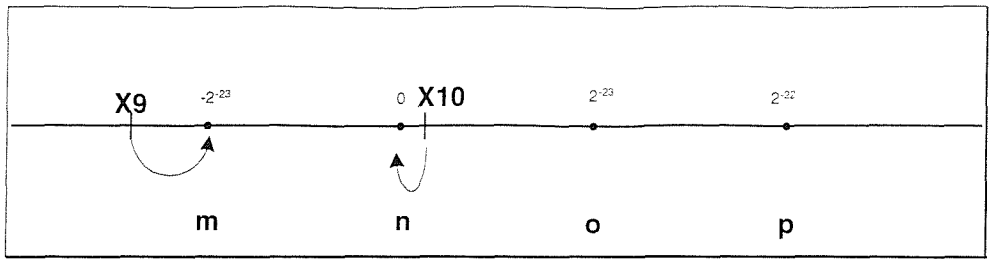


Figure A.6 "Rounding towards zero" example

The standard provides further details on a 32-bit integer format that accompanies the floating-point number representation, along with the required type conversion operations. It also discusses traps and trap handler issues, which are user defined subroutines that track a certain status flag and replaces the output result of an operation that raises that flag. It also discusses the ability of providing user control over these traps, which gives the right to enable and disable these traps. These issues are not represented in this Appendix as they are not related to this work. Further details can be found in [41].

Appendix B

The CORDIC algorithm

The CORDIC algorithm (COordinate Rotation DIgital Computer) was first introduced by Volder [55] as a computing technique to perform vector rotation. It allows computing trigonometric functions, as well as multiplying and dividing numbers using only shift and add operations. In 1971, Walter [56] provided a general form of the original algorithm to provide a means of computing a wide range of elementary functions, including hyperbolic and logarithmic functions. A slight modification of Walter's version allowed computing the inverse sine and inverse cosine functions [57].

This appendix provides a description of the CORDIC algorithm. It is organised in three sections: section B.1 outlines the main properties of the original CORDIC algorithm; section B.2 describes the enhanced version of the algorithm represented in [56]; and section B.3 shows the modifications required to include both inverse sine and inverse cosine in the set of CORDIC generated functions.

B.1 The original CORDIC algorithm

The original algorithm [55] introduced CORDIC as a special purpose computing machine that can be used to rotate a vector by an arbitrary angle or determine the angle and the magnitude of the vector. In other words, the CORDIC machine can be used to solve one of the two sets of equations:

$$y' = K(y\cos\lambda + x\sin\lambda)$$
$$x' = K(x\cos\lambda - y\sin\lambda)$$

or

$$R = \sqrt{x^2 + y^2}$$
$$\theta = \tan^{-1}\frac{y}{x}$$

In order to control the functionality of the CORDIC unit (i.e. solving one of the two sets of previous equations), CORDIC defines two modes of operation:

1. *Rotation mode*: in this mode, the original co-ordinates of the vector (x,y) together with an angle of rotation (λ) are provided, and the co-ordinates of the vector after rotation by the given angle (x',y') are calculated.

2. *Vectoring mode*: in this mode, the co-ordinates of the vector are given (x,y), and the magnitude (R) and the angle (θ) of that vector are computed.

Having two modes of operation with different functionality might suggest two computing units. This is not the case here, since the computing unit is implemented to perform *rotation* and a special feedback is provided to perform the *vectoring* mode. In the latter, the same unit is used to rotate the vector until the angle equals zero, which implies that the sum of the rotations performed in the negative of the original angle, and the value of the new x co-ordinate equal the original magnitude.

In the original CORDIC algorithm, the operation starts with a unique first rotation by an angle of $\pm\pi/2$. The new co-ordinates after the rotation are:

$$x_2 = \pm y_1 = R_1 \cos(\theta_1 \pm \frac{\pi}{2})$$
$$y_2 = \pm x_1 = R_1 \sin(\theta_1 \pm \frac{\pi}{2})$$

The remaining steps are a series of rotations by an angle α_i, where:

$$\alpha_i = \tan^{-1} 2^{-(i-2)}, i > 1$$

The general expression for the new vector co-ordinates after each step i is given by[1]:

$$y_{i+1} = \sqrt{1+2^{-2(i-2)}} R_1 \sin(\theta_i \pm \alpha_i) = y_i \pm 2^{-(i-2)} x_i$$
$$x_{i+1} = \sqrt{1+2^{-2(i-2)}} R_1 \cos(\theta_i \pm \alpha_i) = x_i \pm 2^{-(i-2)} y_i$$

[1] A proof of this can be found in [55].

By introducing a new variable d_i to control the rotation direction, the general expression becomes:

$$y_{i+1} = \sqrt{1+2^{-2(i-2)}} R_1 \sin(\theta_i + d_i\alpha_i) = y_i + d_i 2^{-(i-2)} x_i$$
$$x_{i+1} = \sqrt{1+2^{-2(i-2)}} R_1 \cos(\theta_i + d_i\alpha_i) = x_i + d_i 2^{-(i-2)} y_i$$

where

$$d_i = +1 \text{ or } -1$$

After performing n rotations the final vector co-ordinates will be:

$$y_{n+1} = (\sqrt{1+2^{-0}} + \sqrt{1+2^{-1}} + \ldots + \sqrt{1+2^{-2(n-2)}}) R_1 \sin(\theta_1 + d_1\alpha_1 + d_2\alpha_2 + \ldots + d_n\alpha_n)$$
$$x_{n+1} = (\sqrt{1+2^{-0}} + \sqrt{1+2^{-1}} + \ldots + \sqrt{1+2^{-2(n-2)}}) R_1 \cos(\theta_1 + d_1\alpha_1 + d_2\alpha_2 + \ldots + d_n\alpha_n)$$

Note that the increase in the magnitude is the constant K for a certain number of iteration. Substituting K gives the general form of the final co-ordinates:

$$y_{n+1} = K \sin(\theta_1 + \lambda)$$
$$x_{n+1} = K \cos(\theta_1 + \lambda)$$

where

$$\lambda = d_1\alpha_1 + d_2\alpha_2 + \ldots + d_n\alpha_n$$

From the previous definition of the vectoring mode, the following condition applies:

$$-\theta = d_1\alpha_1 + d_2\alpha_2 + \ldots + d_n\alpha_n$$

As mentioned earlier, controlling the rotation direction is achieved by d_n which takes a value +1 or –1. To determine the value of d_n a new variable z_n is introduced to accumulate the angle variation:

$$z_{n+1} = z_n + d_n\alpha_n$$

For the rotation mode, the sign of z_n decides the d_n value with $d_n = +1$ for $z_n \geq 0$. otherwise -1. For vectoring mode, the sign of y_n controls the d_n value with $d_n = -1$ for $y_n \geq 0$, otherwise -1.

B.2 The enhanced CORDIC algorithm

The algorithm is based on a linear, circular, and hyperbolic co-ordinate system parameterised by a constant m [56, 61] as shown in Figure B.1, where a vector P_i with a magnitude R_i and angle A_i is defined using the three co-ordinate systems, where:

$$R_i = \sqrt{x_i^2 + m y_i^2}$$

$$A_i = m^{-1} \tan^{-1}\left[m^{\frac{1}{2}} \frac{y_i}{x_i} \right]$$

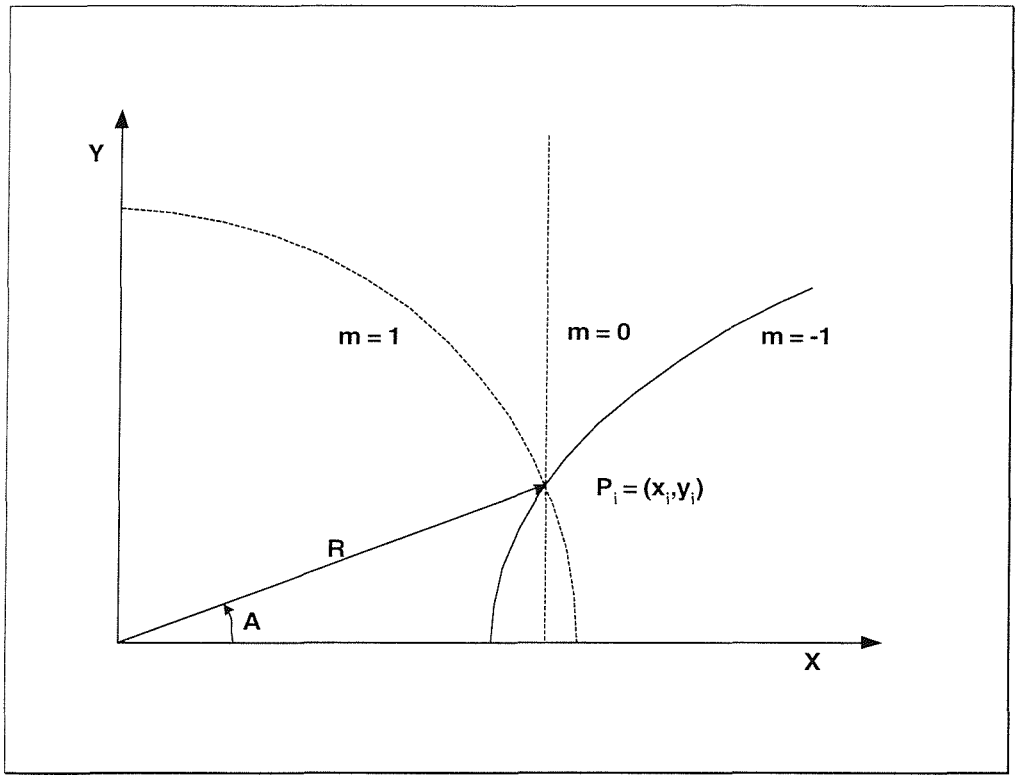


Figure B.1 A vector in three co-ordinate systems

From the previous two equations, it is clear that $m = 1$ for a circular system; $m = 0$ for a linear system; and $m = -1$ for a hyperbolic co-ordinate system.

A new vector P_{i+1} may be obtained from P_i by:

$$x_{i+1} = x_i + m y_i \delta_i$$
$$y_{i+1} = y_i - x_i \delta_i$$

The magnitude and the angle of the new vector are given by:

$$A_{i+1} = A_i - \alpha_i$$
$$R_{i+1} = R_i \times K_i$$

where

$$\alpha_i = m^{\frac{-1}{2}} \tan^{-1}(m^{\frac{1}{2}} \delta_i)$$
$$K_i = \sqrt{1 + m\delta_i^2}$$

The previous set of equations suggest that the angle and the magnitude of the original vector are modified by quantities which are independent of the x and y co-ordinates. By applying the previous transformation for n iterations we get:

$$A_n = A_0 - \alpha$$
$$R_n = R_0 \times K$$

where

$$\alpha = \sum_{i=0}^{n-1} \alpha_i$$
$$K = \prod_{i=0}^{n-1} K_i$$

This implies that the total change in the angle is an accumulation of the intermediate changes, while the total change in the magnitude is the product of the incremental changes.

The angle factor α_i, and the magnitude factor K_i are provided in Table B.1 for the three different co-ordinate systems.

Co-ordinate system	Angle factor α_i	Magnitude factor K_i
Circular	$\tan^{-1}\delta_i$	$(1+\delta_i^2)^{1/2}$
Linear	δ_i	1
Hyperbolic	$\tanh^{-1}\delta_i$	$(1-\delta_i^2)^{1/2}$

Table B.1 Angle and magnitude factors

By introducing a new variable z to accumulate the angle variation:

$$z_{i+1} = z_i + \alpha_i$$

we end up with three difference equations for (x, y, z), and solving them for n iterations gives:

$$x_n = K[x_0 \cos(\alpha m^{\frac{1}{2}}) + y_0 \sin(\alpha m^{\frac{1}{2}})]$$
$$y_n = K[y_0 \cos(\alpha m^{\frac{1}{2}}) - x_0 \sin(\alpha m^{\frac{1}{2}})]$$
$$z_n = z_0 + \alpha$$

Using the final set of equations, a wide range of elementary functions may be generated. Table B.2 and Table B.3 represent the output value after n iterations and for two different modes:

1. The angle A is forced to zero, which means that $y_n = 0$ (vectoring mode).

2. The accumulation of the angle variation is forced to zero, which means that $z_n = 0$ (rotation mode).

Co-ordinate system	Final Values
Circular	$x_n \to K(x_0 \cos z_0 - y_0 \sin z_0)$ $y_n \to K(y_0 \cos z_0 + x_0 \sin z_0)$ $z_n \to 0$
Linear	$x_n \to x_0$ $y_n \to y_0 + x_0 z_0$ $z_n \to 0$
Hyperbolic	$x_n \to K'(x_0 \cosh z_0 + y_0 \sinh z_0)$ $y_n \to K'(y_0 \cosh z_0 + x_0 \sinh z_0)$ $z_n \to 0$

Table B.2 CORDIC result for the rotation mode

Co-ordinate system	Final Values
Circular	$x_n \to K\sqrt{x_0^2 + y_0^2}$ $y_n \to 0$ $z_n \to z_0 - \tan^{-1}(\frac{y_0}{x_0})$
Linear	$x_n \to x_0$ $y_n \to 0$ $z_n \to z_0 - \frac{y_0}{x_0}$
Hyperbolic	$x_n \to K\sqrt{x_0^2 - y_0^2}$ $y_n \to 0$ $z_n \to z_0 - \tanh^{-1}(\frac{y_0}{x_0})$

Table B.3 CORDIC result for the vectoring mode

In addition to the functions listed in the previous tables, the following functions may also be generated:

$$\tan z = \frac{\sin z}{\cos z}$$

$$\tanh z = \frac{\sinh z}{\cosh z}$$

$$\ln z = 2\tanh^{-1}\left(\frac{y}{x}\right), x = z+1, y = z-1$$

$$\sqrt{z} = \sqrt{x^2 - y^2}, x = z+\tfrac{1}{4}, y = z-\tfrac{1}{4}$$

In order to be able to force the angle A to zero by a set of rotations α_i, the direction of the rotation is defined in each step so that:

$$|A_{i+1}| = ||A_i| - \alpha_i|$$

This implies that the remaining rotations in each step must be at least within α_{n-1} of zero, which defines the main convergence criterion:

$$\alpha_i - \sum_{j=i+1}^{n-1} \alpha_j < \alpha_{n-1}$$

This introduces a limitation on the domain of convergence of this algorithm:

$$\max|A_0| = \alpha_{n-1} + \sum_{j=0}^{n-1} \alpha_j$$

Another problem appears in the hyperbolic mode, as the convergence criterion is not satisfied. However, if the steps (4, 13, 40, 121,..., f, 3f+1,...) then the criterion is satisfied [56].

For a practical implementation of the algorithm, δ_i is assigned the value 2^{-i} which results in the final form of the algorithm:

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix}$$
$$z_{n+1} = z_n - d_n \alpha_n$$

Where d_n = sign (z_n) for the rotation mode and –sign (y_n) for the vectoring mode. These rotations can be performed by a series of shift (multiply by 2^{-n}) and add operations with the values of the rotation angles (α_n) pre-calculated and stored in a small table.

B.3 Computation of inverse sine and inverse cosine using CORDIC

This section shows how the method can be used to calculate the inverse sine and inverse cosine functions. Firstly, a simple algorithm is introduced, along with its main disadvantage. Then a final version of the algorithm that tackles this drawback [57] is outlined.

Assuming that we want to compute $z = \cos^{-1}(t)$, we perform a rotation of the angle z starting at the point (1,0) Using CORDIC this can be achieved by:

$$z_0 = 0$$
$$x_0 = 1$$
$$y_0 = 0$$
$$d_n = 1 \quad \text{if } z_n \le z \text{ else} -1$$
$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix}$$
$$z_{n+1} = z_n + d_n \tan^{-1} 2^{-n}$$
$$\lim_{n \to +\infty} z_n = \cos^{-1}(t)$$

The main problem faced here is that the value z is unknown, which implies that we cannot perform the test above to control the rotation direction. However, the test can be replaced with the following equivalent test[2]

[2] See [57] for a proof of this replacement.

$$d_n = sign(y_n) \quad \text{if } x_n \geq K_n t \text{ else} - sign(y_n)$$

where

$$K_n = \prod_{i=0}^{n-1} \sqrt{1+2^{-2i}}$$

The new test solves the problem encountered in the previous algorithm. However, a major drawback arises from the fact that at each step $t_n = K_n t$ is required. To compute t_n the relation $t_{n+1} = t_n (1+2^{-2n})^{1/2}$ may be used. But this would require a true multiplication at each step. To overcome *this* problem, *two* rotations of $d_n \tan^{-1}2^{-n}$ must be performed in each rotation, which reduces the computation to $t_{n+1} = t_n (1+2^{-2n})$, thereby reducing the true multiplication to an add and a shift operation. Performing this modification we obtain the following algorithm to compute the inverse cosine:

$$z_0 = 0$$
$$x_0 = 1$$
$$y_0 = 0$$
$$d_n = sign(y_n) \quad \text{if } x_n \geq t_n \text{ else} - sign(y_n)$$
$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix}^2 \begin{pmatrix} x_n \\ y_n \end{pmatrix}$$
$$z_{n+1} = z_n + 2d_n \tan^{-1} 2^{-n}$$
$$t_{n+1} = t_n + t_n 2^{-n}$$

In a similar manner the algorithm to compute the inverse sine is:

$$z_0 = 0$$
$$x_0 = 1$$
$$y_0 = 0$$
$$d_n = sign(x_n) \quad \text{if } y_n \leq t_n \text{ else} - sign(x_n)$$
$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix}^2 \begin{pmatrix} x_n \\ y_n \end{pmatrix}$$
$$z_{n+1} = z_n + 2d_n \tan^{-1} 2^{-n}$$
$$t_{n+1} = t_n + t_n 2^{-n}$$

The domain of convergence of the CORDIC algorithm is defined by the accumulated sum of the elementary rotations performed over the required number of iterations. This implies

that the double rotation performed in this algorithm to reduce the multiplication cost doubles the size of the algorithm convergence domain.

Appendix C

Elementary functions details

This appendix provides internal details of the floating-point library elementary functions discussed in Chapter 4. In each section, a detailed description of the range reduction unit is provided, as well as a description of the function generators provided to implement the function. Function generator accuracy estimates based on simulation results of uniformly distributed samples over the required input range are also provided.

C.1 Sine and cosine functions

The sine and cosine functions are combined into one building block, generating either the sine or the cosine of the input operand according to the value of control input. The input to the function generator is in radians.

C.1.1 Pre-processing stage

The pre-processing stage performs two tasks:

1. Input operand type detection.

2. Reduces the range of the input operand to the range of the function generation block $[0,\pi/2]$.

A block diagram of the pre-processing stage is provided in Figure C.1. Input type detection is the first stage in the pre-processing step. It performs a series of tests to identify certain cases represented in Table C.1. If any of these cases are detected, the appropriate value is assigned to the output and the *done* flag is raised to indicate that there is no need for further processing in the following function generation block. The type detection unit also assigns the appropriate value to the *flag register*.

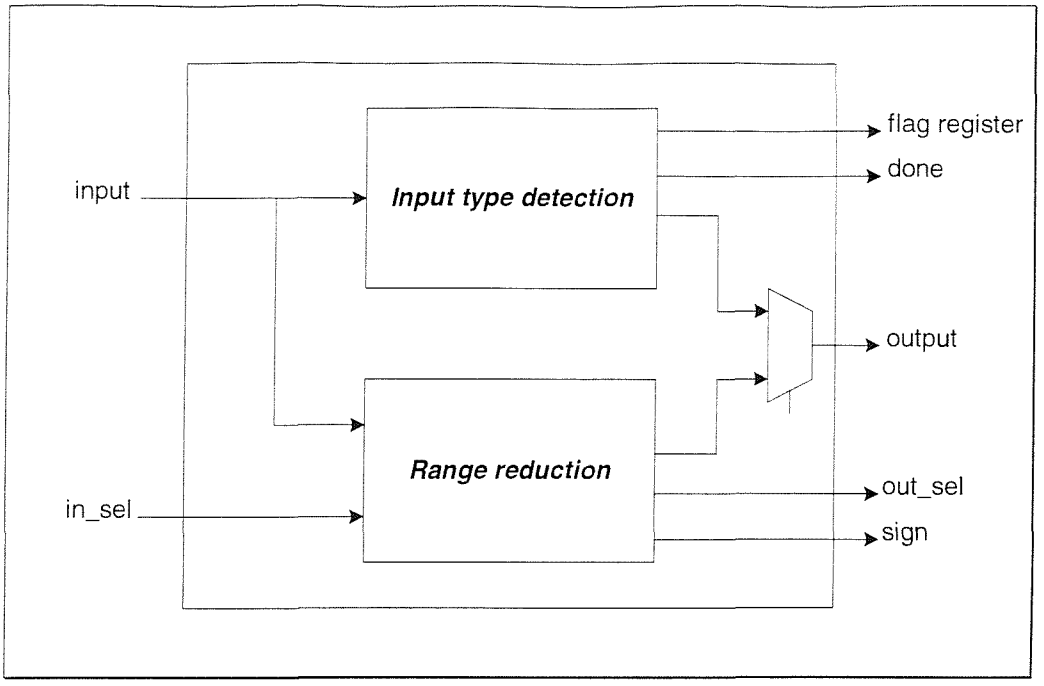


Figure C.1 Sine/cosine pre-processing stage

If the input operand passes the type detection stage, a range reduction is performed on the input to scale it within the range $|x| \in [0,\pi/2]$[1]. This is achieved using the following equation:

$$\sin(Q\tfrac{\pi}{2}+D)=\begin{cases}+\sin D\ if & Q \bmod 4=0\\ +\cos D\ if & Q \bmod 4=1\\ -\sin D\ if & Q \bmod 4=2\\ -\cos D\ if & Q \bmod 4=3\end{cases}$$

The application of the range reduction procedure takes place in a number of steps illustrated in the flow graph of Figure C.2:

1. The input is divided by $\pi/2$ (multiplied by $2/\pi$) and the output result is stored in a temporary variable.

2. The fractional part of the previous step result is then multiplied by $(\pi/2)$ and the result is provided as the output operand.

[1] If the input is already within this range, the range reduction procedure is bypassed.

3. The input control variable *in_sel* combined with the integer part of the division result in step one and the input operand sign are used to identify the final result sign and the operation to be performed in the following stage (generating either the sine or the cosine function).

Operation	Input	Output	Flag register					
			Inexact	Invalid	NAN	OVF	EUN	ZD
sine	+∞	Quiet NAN	0	1	1	0	0	0
sine	-∞	Quiet NAN	0	1	1	0	0	0
sine	Sig. NAN	Quiet NAN	0	1	1	0	0	0
sine	Quiet NAN	Quiet NAN	0	0	1	0	0	0
sine	zero	0	0	0	0	0	0	0
cosine	+∞	Quiet NAN	0	1	1	0	0	0
cosine	-∞	Quiet NAN	0	1	1	0	0	0
cosine	Sig. NAN	Quiet NAN	0	1	1	0	0	0
cosine	Quiet NAN	Quiet NAN	0	0	1	0	0	0
cosine	zero	1	0	0	0	0	0	0

Table C.1 Special input cases in the sine/cosine function

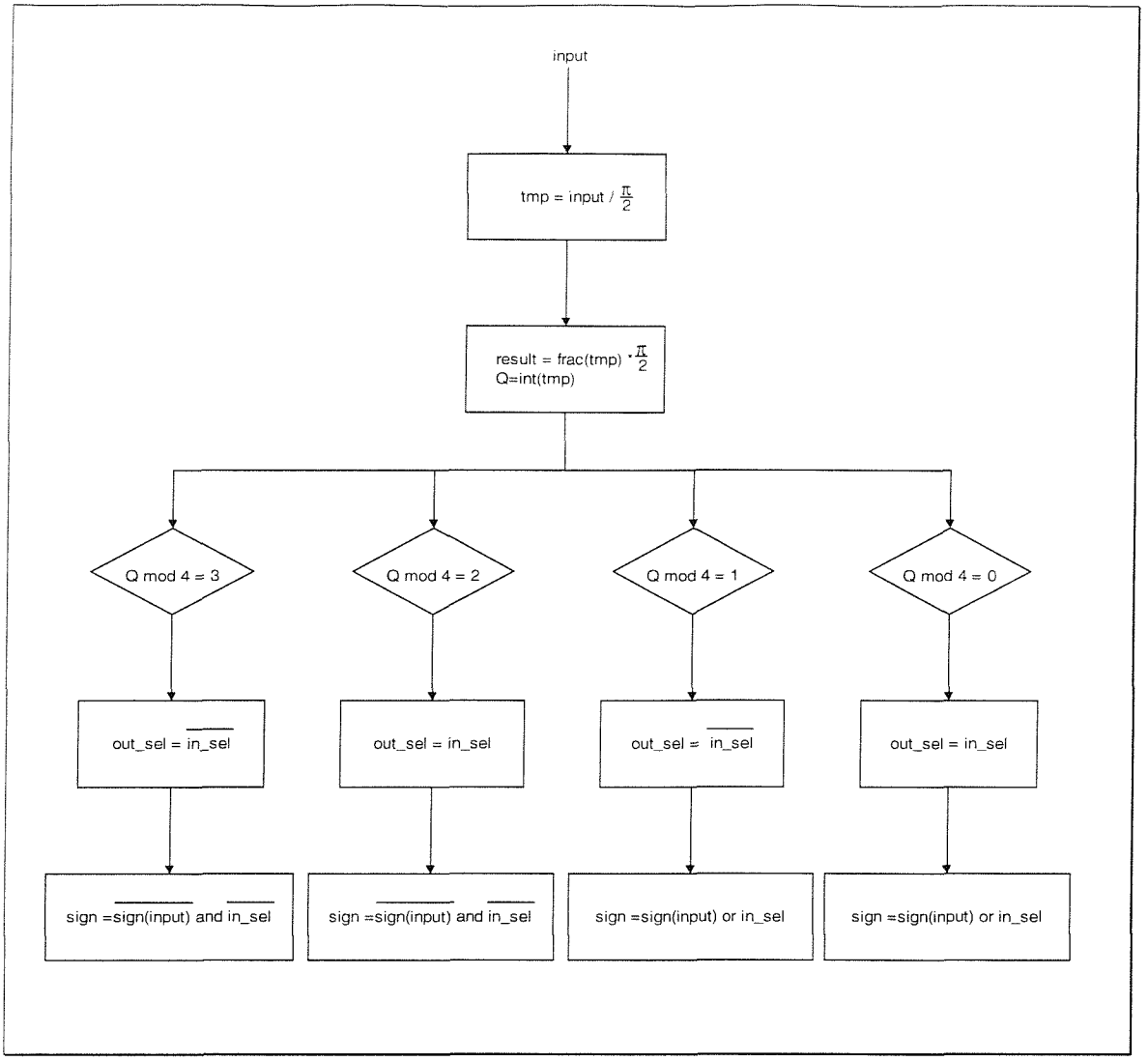


Figure C.2 Sine/cosine range reduction flow chart[2]

C.1.2 Function generation unit

The first set of function generators is based on a single lookup table with linear interpolation. The absolute error over the required range varies as the table size and hence the difference between two adjacent break points (slope) changes. The figures in Table C.2 represent the error variation as the table size changes. These results are summarised in Figure C.3, where the error is shown for different table sizes.

[2] *int(x)* returns the nearest integer less that or equal to *x* (nearest zero). *frac(x)* returns the value *x-int(x)*.

Name	Slope	Table entries	Maximum error
sin_cos_7_lsi	2^{-11}	3217	3.9539e-8
sin_cos_6_lsi	2^{-9}	805	4.8599e-7
sin_cos_5_lsi	2^{-7}	202	7.6292e-6
sin_cos_4_lsi	2^{-6}	101	3.0905e-5
sin_cos_3_lsi	2^{-4}	26	4.8783e-4
sin_cos_2_lsi	2^{-2}	7	7.7000e-3

Table C.2 Maximum error in the sine/cosine generator using single table and linear interpolation

A reduction in the table size is achieved by partitioning the lookup table into a number of sub-tables. The table is partitioned so that the maximum error generated in each sub-table is less than a limit that guarantees the target accuracy. This is illustrated in Table C.3 and Figure C.4, where the error is represented for different combinations of partitioned table. Figure C.5 shows the sub-tables distribution for the four units listed in Table C.3.

Name	Sub-table range	Sub-table slope	Table entries	Maximum error
sin_cos_7_lmi	0-0.19635	2^{-9}	403	1.0052e-7
	0.19635-0.98175	2^{-10}	805	
	0.98175-π/2	2^{-11}	1207	
sin_cos_6_lmi	0-0.490875	2^{-8}	126	8.8646e-7
	0.490875-π/2	2^{-9}	553	
sin_cos_5_lmi	0-0.294525	2^{-6}	19	8.6986e-6
	0.294525-π/2	2^{-7}	164	
sin_cos_4_lmi	0-0.883575	2^{-5}	29	9.2394e-5
	0.883575-π/2	2^{-6}	44	

Table C.3 Maximum error in the sine/cosine generator using partitioned table and linear interpolation

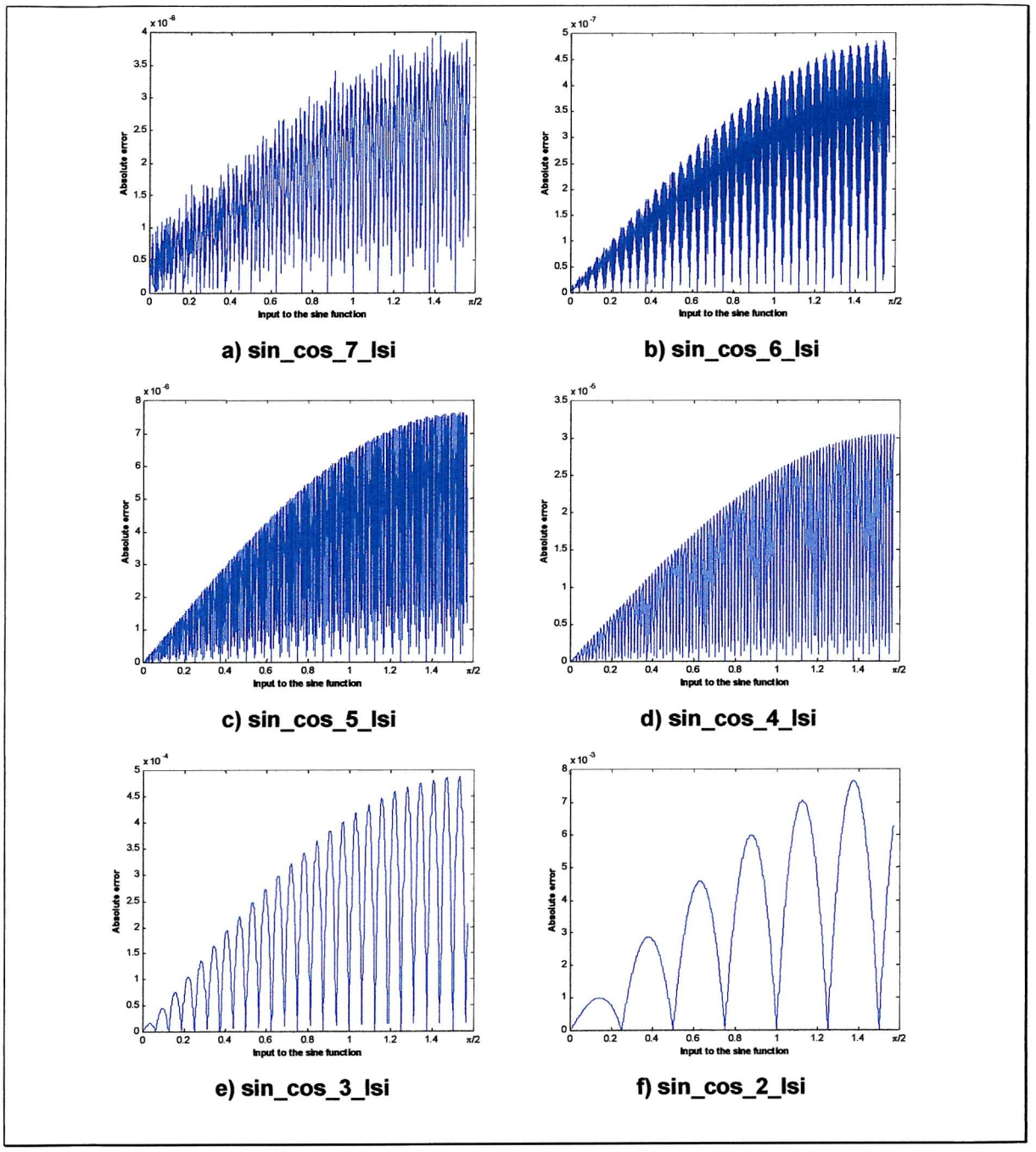


Figure C.3 Error in the sine/cosine generator using linear interpolation engine with a single-table and for different table sizes

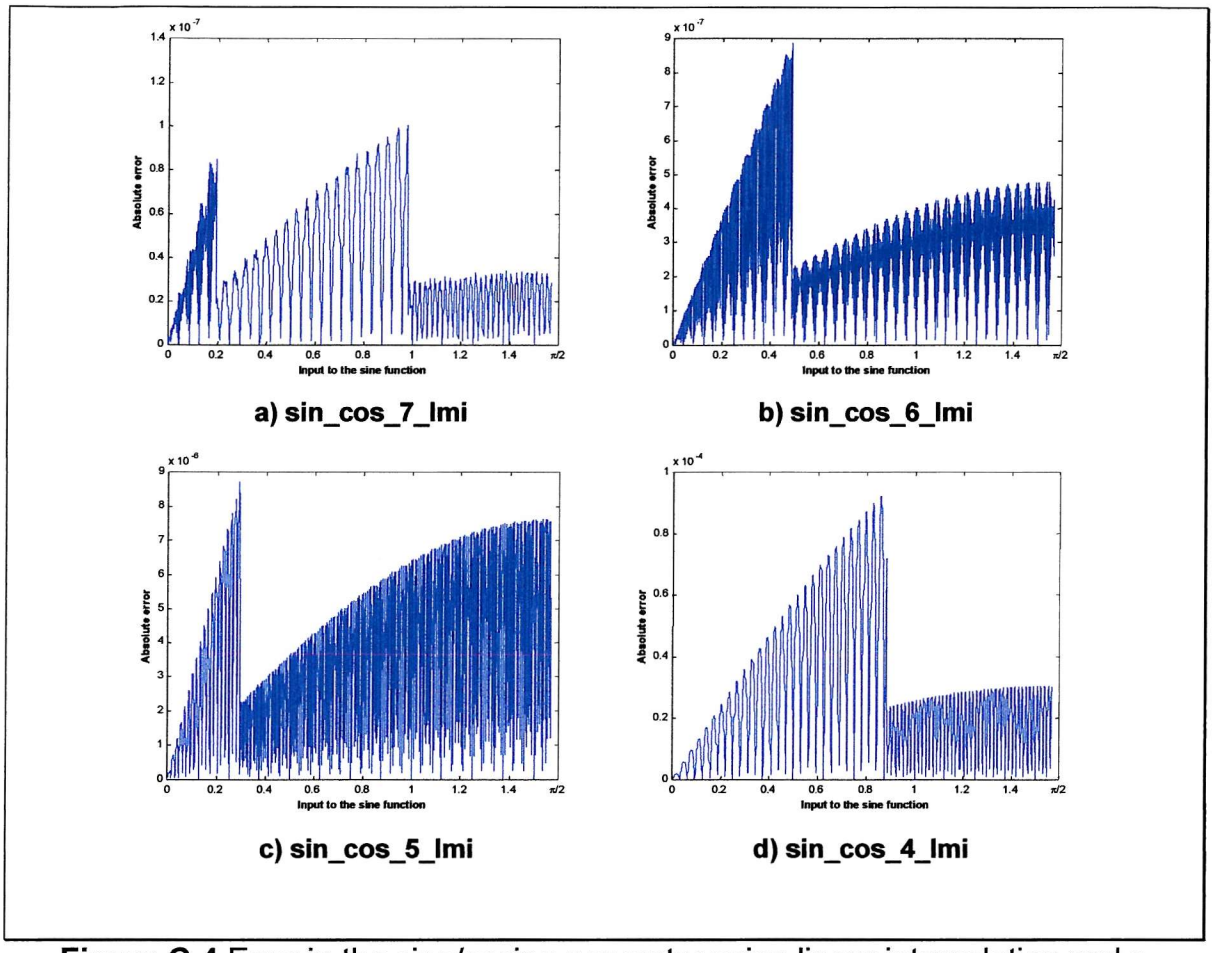


Figure C.4 Error in the sine/cosine generator using linear interpolation and a partitioned table for different table sizes

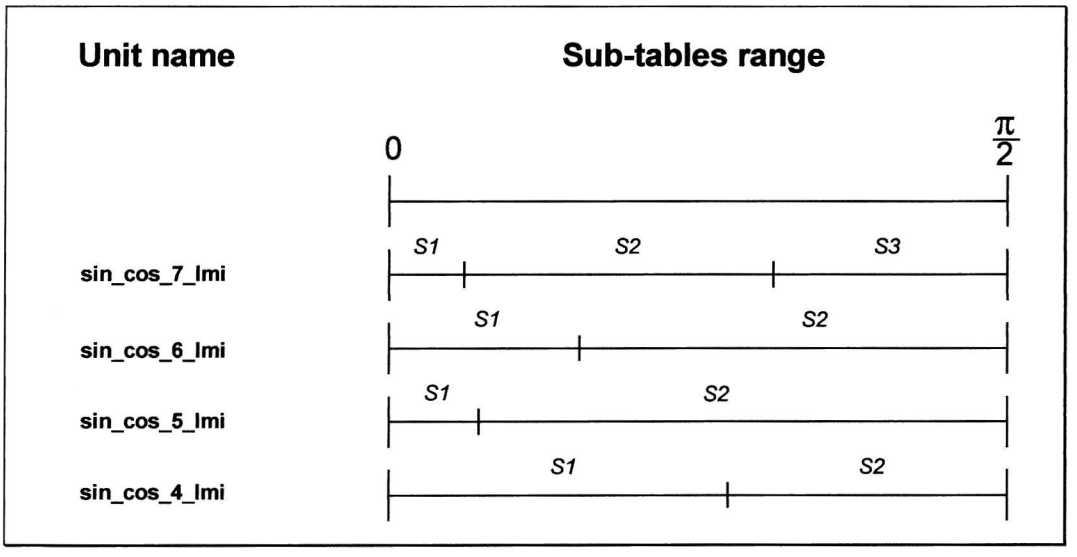


Figure C.5 Sub-tables range in the sine/cosine generator using linear interpolation and partitioned table

Note that for the table lookup based implementation, an equivalent unit that replaces the internal table with an external ROM interface is provided to allow implementing the table using an external ROM.

An iterative series method based on the minimax approximation of the sine/cosine function is also available to generate these functions. As expected, the error in the function approximation is highly dependent on the approximating function degree. The maximum approximation error for different approximation degrees is illustrated in Table C.4 and the same results are summarised in Figure C.6.

Name	Approximation degree	Maximum error
sin_cos_7_ser	7	9.1500e-8
sin_cos_6_ser	6	4.7340e-7
sin_cos_5_ser	5	7.1280e-6
sin_cos_4_ser	4	1.0400e-4

Table C.4 Maximum error in the sine/cosine generator using minimax approximation

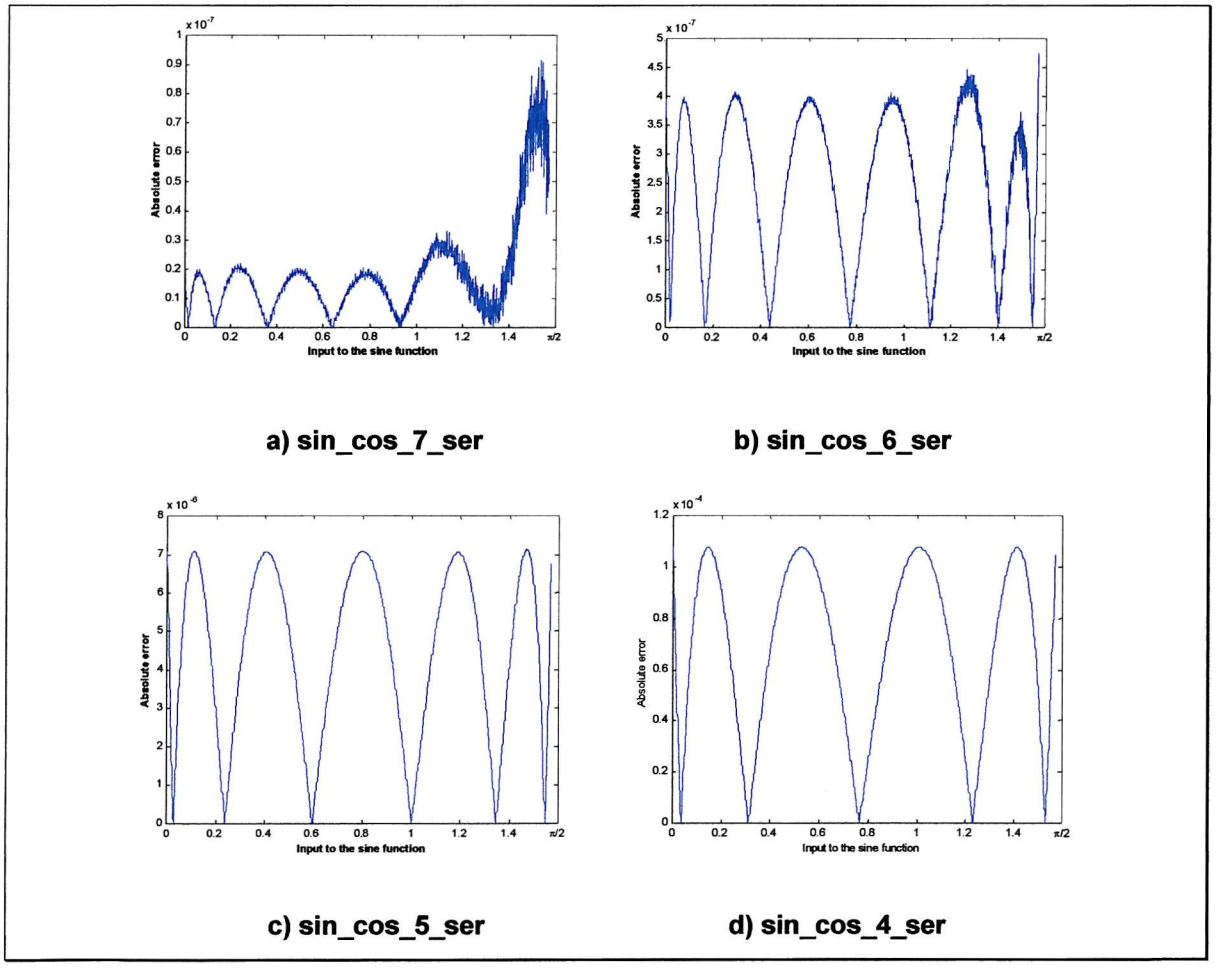


Figure C.6 Error in the sine/cosine minimax engine for different approximation degrees

Finally, a CORDIC based engine is provided to generate this function. The unit uses the CORDIC algorithm in the circular mode (m = 1) and with the input operand initialised as ($x = 1/K$, $y = 0$, z = input operand). The accuracy of these function generators varies according to the number of CORDIC iterations. This is shown in Table C.5 and Figure C.7 showing the maximum approximation errors for different number of iterations.

Name	Number of iterations	Maximum error
sin_cos_7_COR	25	1.1913e-7
sin_cos_6_COR	22	5.1109e-7
sin_cos_5_COR	18	7.5161e-6
sin_cos_4_COR	15	6.0760e-5

Table C.5 Maximum error in the sine/cosine generator using CORDIC algorithm

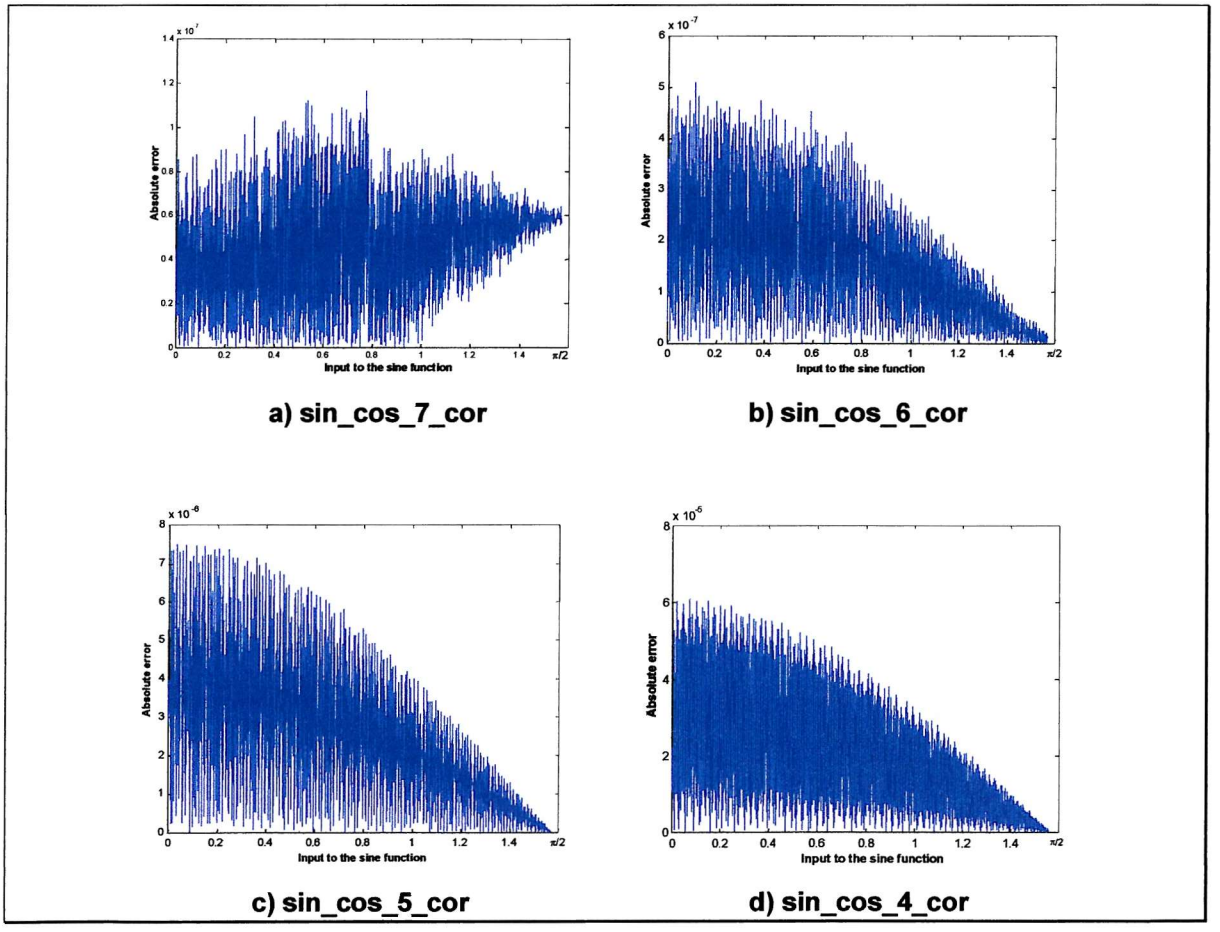


Figure C.7 Error in the sine/cosine CORDIC unit for different number of iterations

C.2 Inverse sine and inverse cosine functions

The inverse sine and inverse cosine functions are implemented using a single building block, and a control input is provided to select between the two functions. Due to the periodic nature of both the sine and cosine function, their inverses cannot be formed unless the domain is restricted. This restricts the input to the range [-1,1], which eliminates the need for a range reduction block.

A block diagram representing the building blocks of the unit is shown in Figure C.8. Input type detection performs a series of tests to detect certain cases in which the output is predefined. These cases are represented in Table C.6. If any of these cases is detected, the corresponding output value is assigned and the function generator is bypassed.

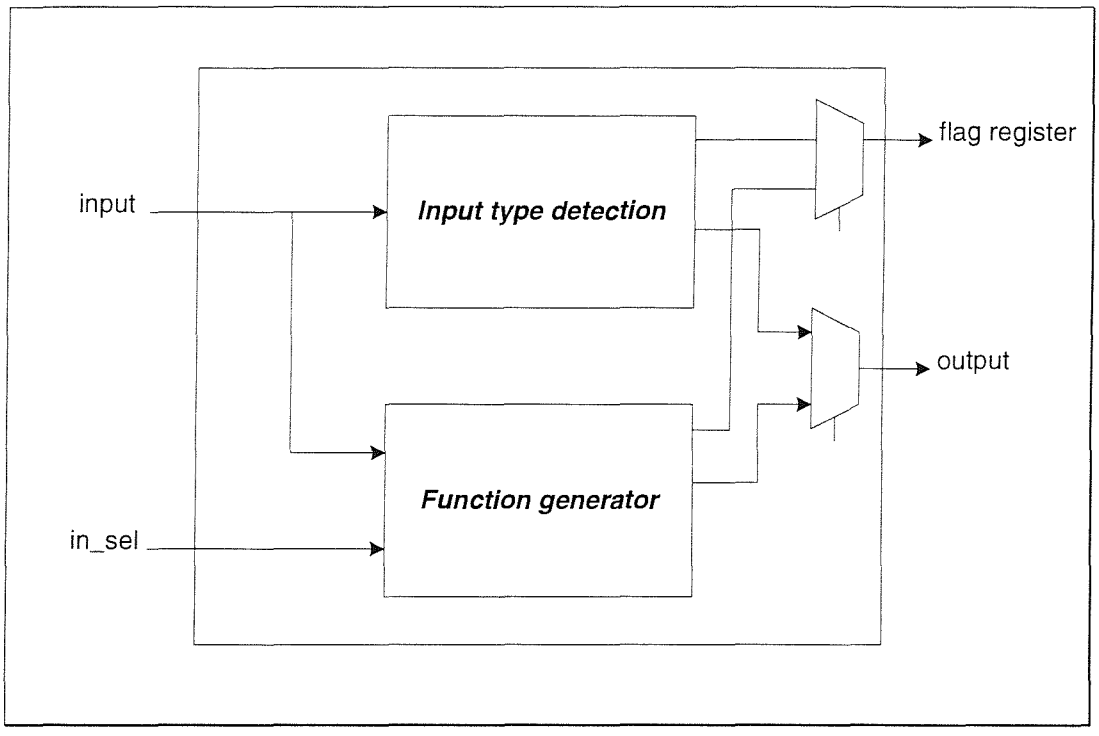


Figure C.8 inverse sine/inverse cosine generation unit

For the general case, the inverse sine function is generated in the range [0,1] and the final output is provided using the simple relationship:

$$\arcsin(\pm x) = \pm \arcsin(|x|)$$

$$\arccos(\pm x) = \frac{\pi}{2} - [\pm \arcsin(|x|)]$$

Operation	Input	Output	Flag register					
			Inexact	Invalid	NAN	OVF	EUN	ZD
Inverse sine	+∞	Quiet NAN	0	1	1	0	0	0
Inverse sine	-∞	Quiet NAN	0	1	1	0	0	0
Inverse sine	Sig. NAN	Quiet NAN	0	1	1	0	0	0
Inverse sine	Quiet NAN	Quiet NAN	0	0	1	0	0	0
Inverse sine	zero	0	0	0	0	0	0	0
Inverse sine	>|1|	Quiet NAN	0	1	1	0	0	0
inverse cosine	+∞	Quiet NAN	0	1	1	0	0	0
inverse cosine	-∞	Quiet NAN	0	1	1	0	0	0
inverse cosine	Sig. NAN	Quiet NAN	0	1	1	0	0	0
inverse cosine	Quiet NAN	Quiet NAN	0	0	1	0	0	0
inverse cosine	zero	$\pi/2$	0	0	0	0	0	0
inverse cosine	>|1|	Quiet NAN	0	1	1	0	0	0

Table C.6 special input cases in the inverse sine/inverse cosine function

The function generation unit is implemented using either a partitioned lookup table or a CORDIC base procedure[3]. For a table-based method, Table C.7 and Figure C.9 represent the maximum error encountered for different table sizes.

[3] Due to the nature of the inverse sine function as $|x| \rightarrow 1$, neither a single slope table lookup nor a polynomial approximation are not a viable solution for this function.

Name	Sub-table range	Sub-table slope	Table entries	Maximum error
asin_acos_7_lmi	0-0.234375	2^{-10}	240	3.0335e-7
	0.234375-0.53125	2^{-11}	608	
	0.53125-0.78125	2^{-12}	1024	
	0.78125-0.890625	2^{-13}	896	
	0.890625-0.96875	2^{-14}	1280	
	0.96875-0.986022	2^{-15}	566	
	0.986022-0.994353	2^{-16}	546	
	0.994353-0.999236	2^{-18}	1281	
	0.999236-0.999694	2^{-19}	241	
	0.999694-1	2^{-21}	642	
asin_acos_6_lmi	0-0.140625	2^{-8}	36	5.4699e-7
	0.140625-0.390625	2^{-9}	128	
	0.390625-0.71875	2^{-10}	336	
	0.71875-0.875	2^{-11}	320	
	0.875-0.988739	2^{-13}	932	
	0.988739-0.99884	2^{-16}	662	
	0.99884-0.999542	2^{-17}	93	
	0.999542-0.999786	2^{-18}	64	
	0.999786-1	2^{-20}	225	
asin_acos_5_lmi	0-0.28125	2^{-7}	36	5.3550e-6
	0.28125-0.640625	2^{-8}	92	
	0.640625-0.828125	2^{-9}	96	
	0.828125-0.9375	2^{-10}	112	
	0.9375-0.993958	2^{-12}	232	
	0.993958-0.996245	2^{-13}	19	
	0.996245-0.999358	2^{-15}	103	
	0.999358-0.999755	2^{-16}	27	
	0.999755-1	2^{-18}	65	
asin_acos_4_lmi	0-0.265625	2^{-5}	9	4.5791e-5
	0.265625-0.53125	2^{-6}	17	
	0.53125-0.78125	2^{-7}	32	
	0.78125-0.921875	2^{-8}	36	
	0.921875-0.994872	2^{-11}	150	
	0.994872-0.999541	2^{-13}	39	
	0.999541-1	2^{-16}	31	

Table C.7 Maximum error in the inverse sine/inverse cosine generator using partitioned table and linear interpolation

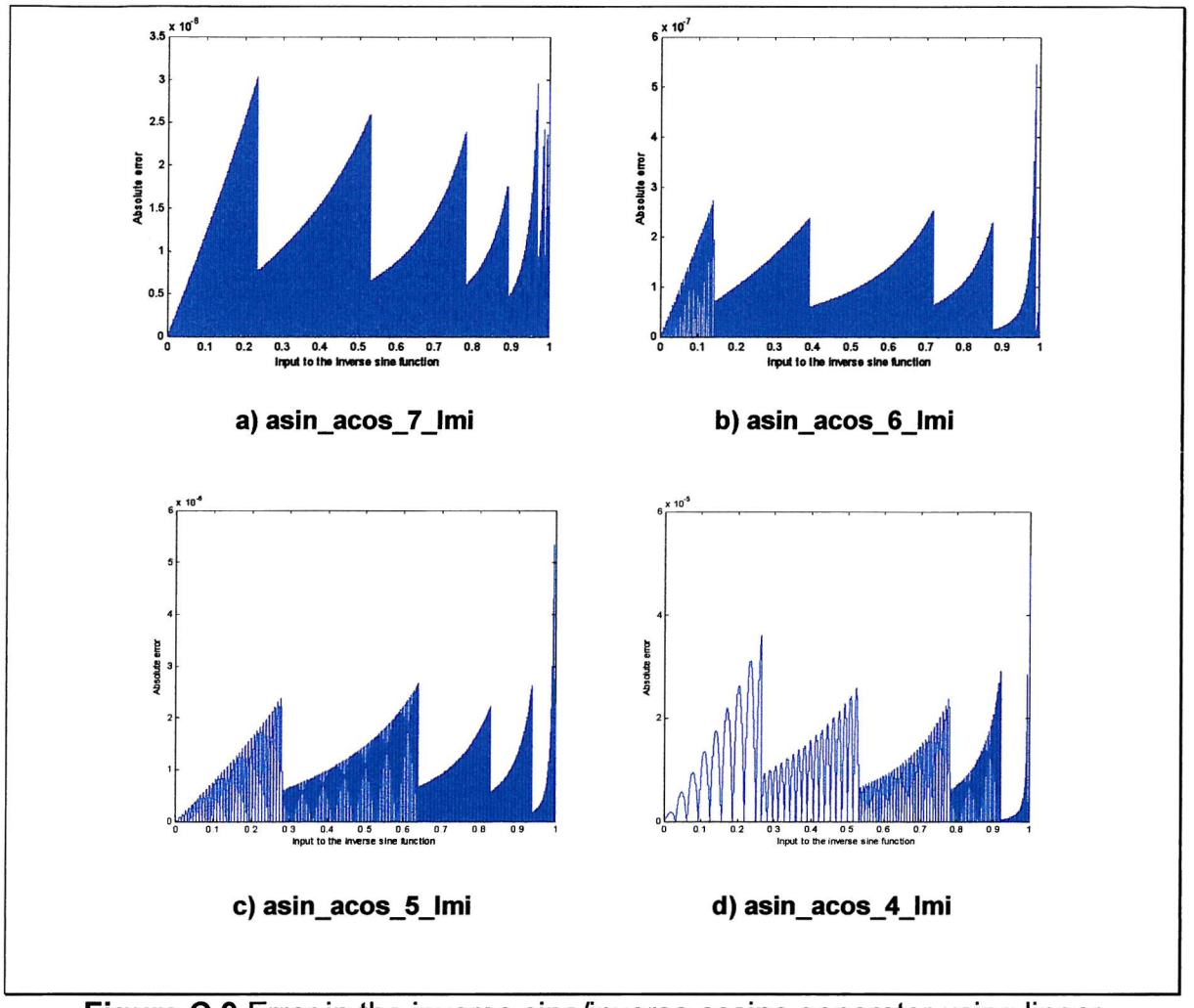


Figure C.9 Error in the inverse sine/inverse cosine generator using linear interpolation engine with a partitioned table lookup

A minor modification to the CORDIC algorithm (see Appendix B for details) provides an iterative procedure to implement the inverse sine and inverse cosine functions. Table C.8 and Figure C.10 represent the accuracy of this method for different number of iterations.

Name	Number of iterations	Maximum error
asin_acos_7_COR	26	1.1268e-7
asin_acos_6_COR	22	9.3970e-7
asin_acos_5_COR	19	7.6005e-6
asin_acos_4_COR	16	6.0995e-5
asin_acos_3_COR	13	4.7333e-4
asin_acos_2_COR	11	1.8907e-3

Table C.8 Maximum error in the inverse sine/inverse cosine generator using CORDIC algorithm

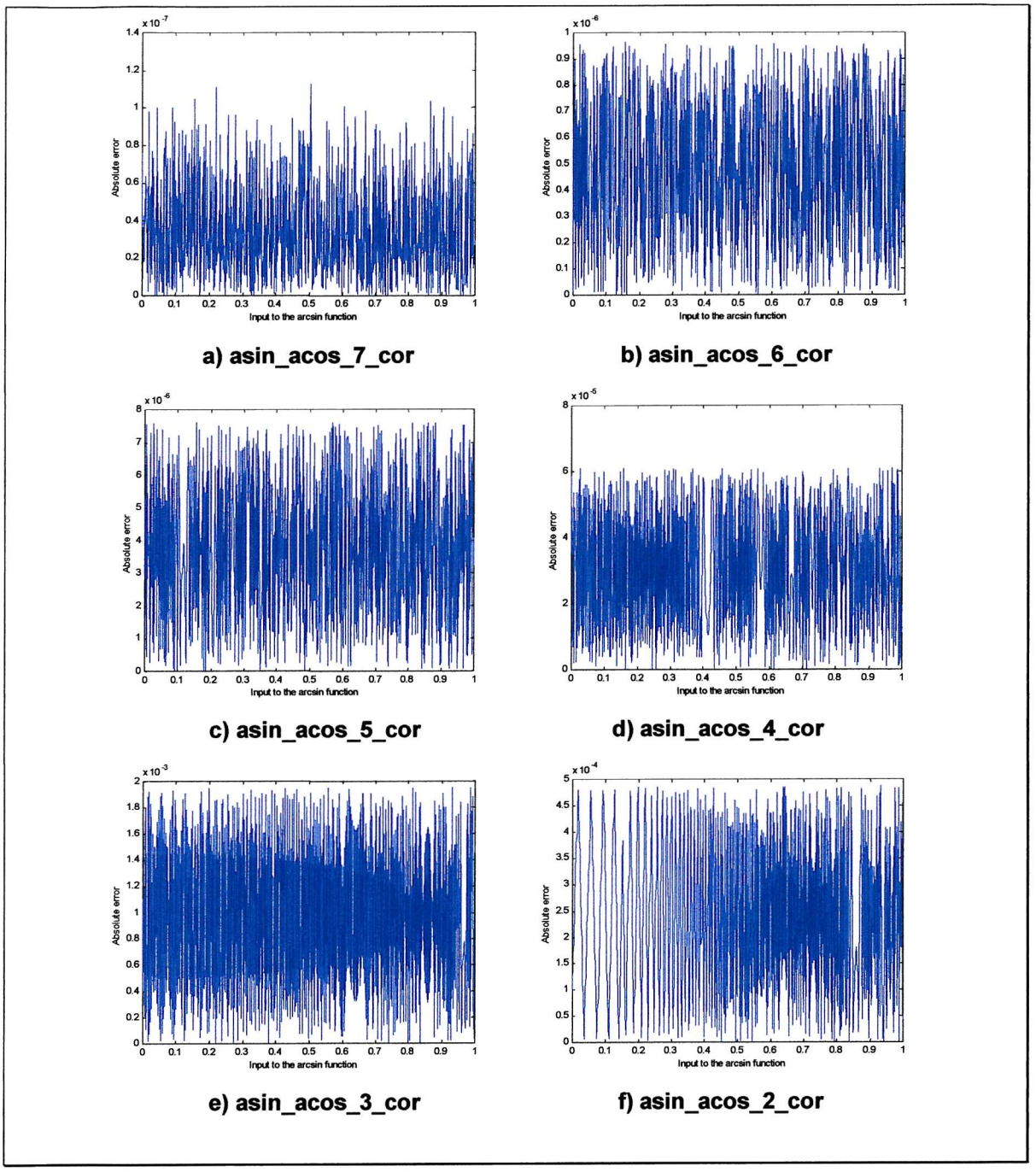


Figure C.10 Error in the asin/acos generator based on the CORDIC engine for different number of iterations

C.3 Inverse tangent function

The function generator of the inverse tangent function consists of two main building blocks:

1. A pre-processing stage that performs range reduction and input type detection.

2. The main function generation unit, which calculates the inverse tangent of an input argument within the range [0,1].

In addition to those two units, a final adjustment stage is required to undo the modification performed by the range reduction stage.

In the pre-processing stage, input type detection is performed to identify any of the input values listed in Table C.9 and output the appropriate result. If none of the listed values are detected, the execution continues to the range reduction unit adjusts the input argument to within the range [0,1], and provides the necessary control signals to govern the data flow in the following stage. A flow chart describing the range reduction procedure is given in Figure C.11. At this stage, the input is divided into two groups:

1. If input is in the range $|x| < 1$, then the function is calculated directly.

2. If $|x| \geq 1$, range reduction is required:

$$\arctan\left(\frac{1}{x}\right) = \frac{\pi}{2} - \arctan(x)$$

Input	Output	Flag register					
		Inexact	Invalid	NAN	OVF	EUN	ZD
+∞	π/2	0	0	0	0	0	0
-∞	-π/2	0	0	0	0	0	0
Sig. NAN	Quiet NAN	0	1	1	0	0	0
Quiet NAN	Quiet NAN	0	0	1	0	0	0
0	0	0	0	0	0	0	0
$\|input\|>2^{24}$	±π/2	1	0	0	0	0	0
$\|input\|<0.007$	±input	1	0	0	0	0	0

Table C.9 Special input cases in the inverse tangent function

The function generation unit is implemented using the three methods described in Chapter 4: table lookup, iterative series, and the CORDIC algorithm.

Table lookup based units are provided using both a single slope table and a partitioned table. For the first set, error variation as the table size changes is represented in Table C.10 and Figure C.12. Similar figures for the partitioned table based units are provided in Table C.11 and Figure C.13.

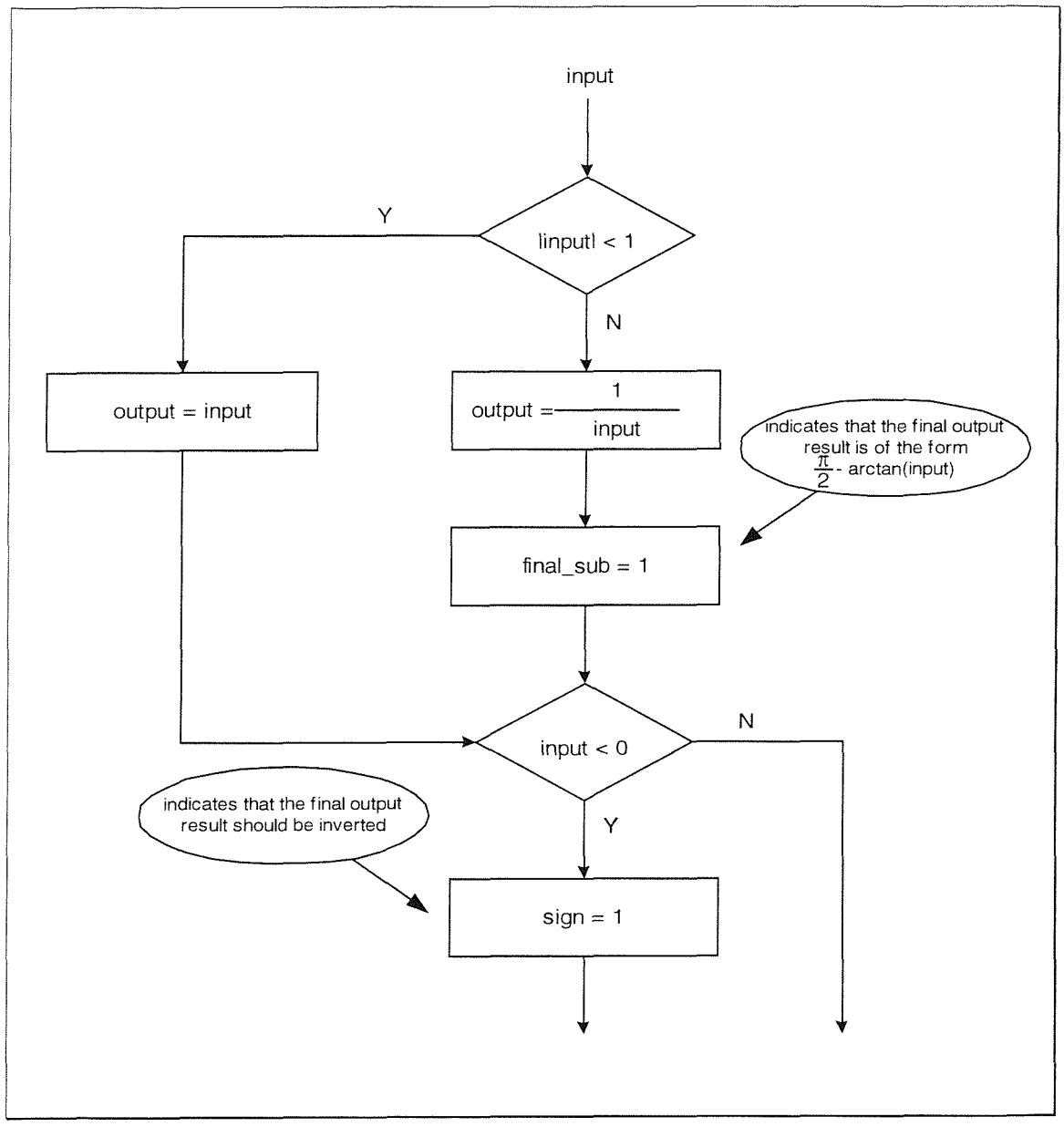


Figure C.11 Inverse tangent range reduction flow chart

Name	Slope	Table entries	Maximum error
atan_main_7_lsi	2^{-10}	1024	8.8135e-8
atan_main_6_lsi	2^{-9}	512	3.1999e-7
atan_main_5_lsi	2^{-7}	128	4.9649e-6
atan_main_4_lsi	2^{-5}	32	7.9282e-5
atan_main_3_lsi	2^{-4}	16	3.1684e-4
atan_main_2_lsi	2^{-2}	4	5.000e-3

Table C.10 Maximum error in the inverse tangent generator using a single table and linear interpolation

Name	Sub-table range	Sub-table slope	Table entries	Maximum error
atan_main_7_lmi	0-0.0625	2^{-9}	32	8.8135e-8
	0.0625-1	2^{-10}	960	
atan_main_6_lmi	0-0.3125	2^{-8}	80	9.6207e-7
	0.3125-1	2^{-9}	352	
atan_main_5_lmi	0-0.125	2^{-6}	8	6.9654e-6
	0.125-1	2^{-7}	112	
atan_main_4_lmi	0-0.125	2^{-4}	2	9.0130e-5
	0.125-1	2^{-5}	28	

Table C.11 Maximum error in the inverse tangent generator using a partitioned table and linear interpolation

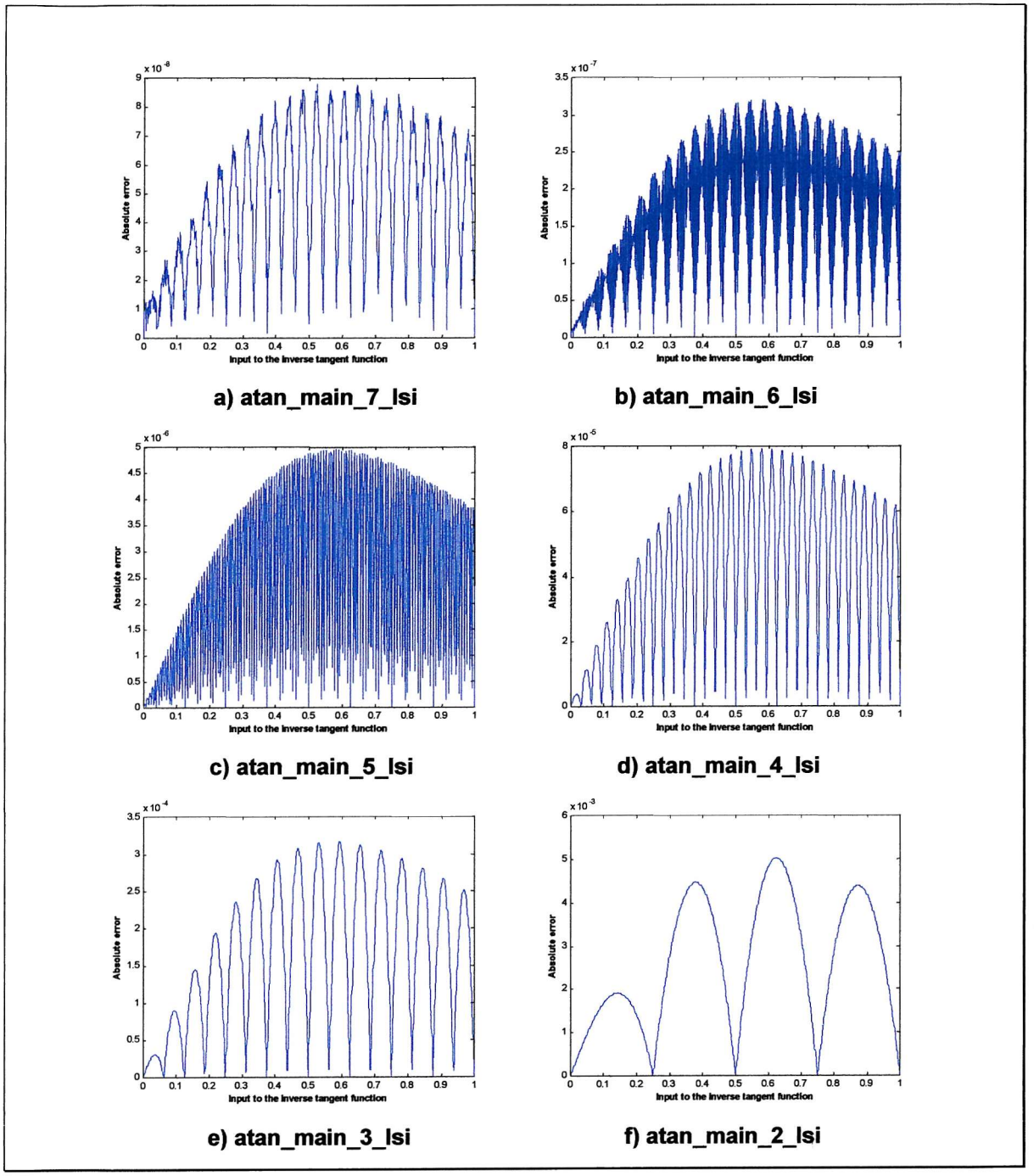


Figure C.12 Error in the inverse tangent generator using a single table and linear interpolation for different table sizes

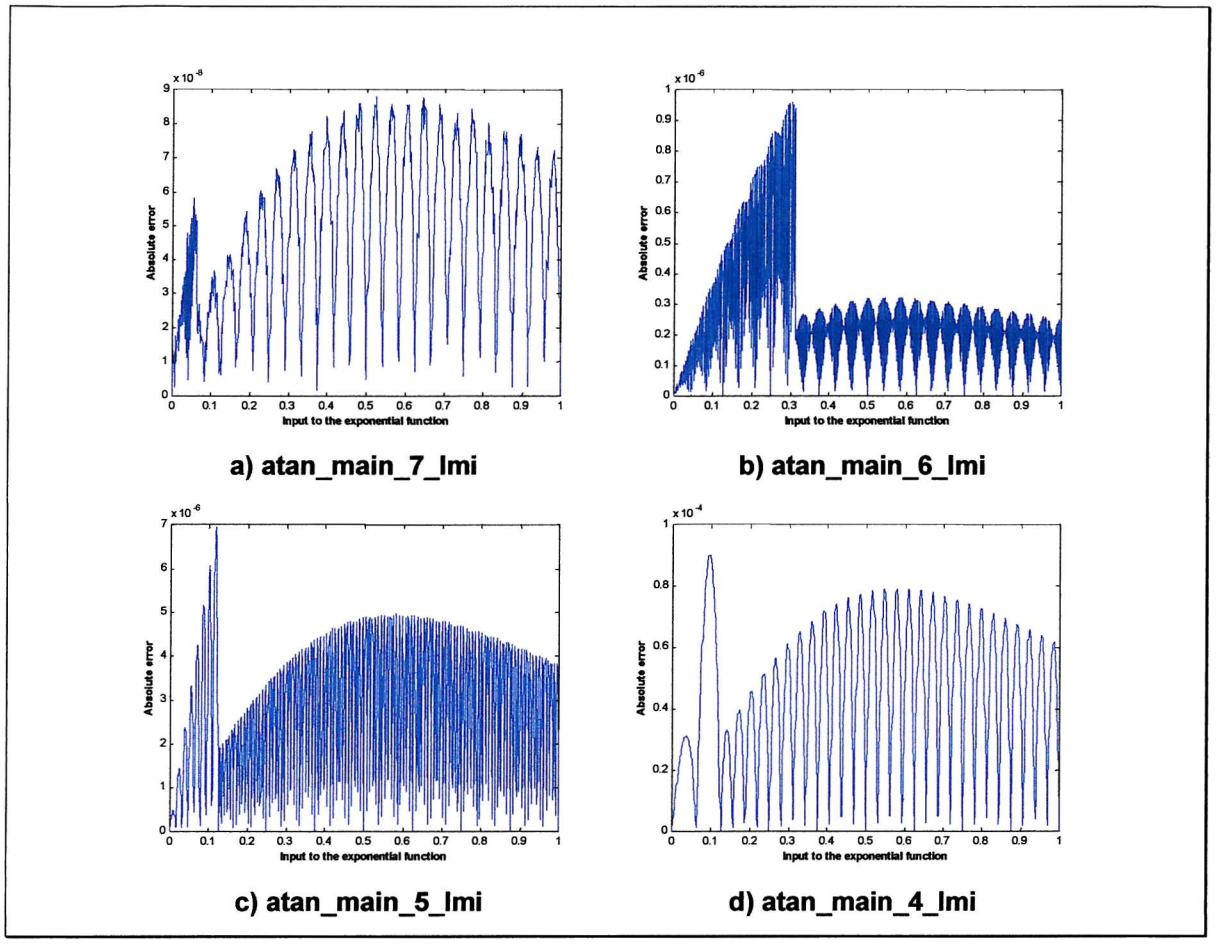


Figure C.13 Error in the inverse tangent generator using a partitioned table and linear interpolation for different table sizes

An iterative series method based on the minimax approximation is also used to generate the inverse tangent function. The maximum approximation error for different approximation degrees is illustrated in Table C.12 and Figure C.14.

Name	Approximation degree	Maximum error
atan_main_7_ser	7	7.3643e-8
atan_main_6_ser	6	4.2296e-7
atan_main_5_ser	5	6.4056e-6
atan_main_4_ser	4	2.0947e-5

Table C.12 Maximum error in the inverse tangent generator using the minimax approximation

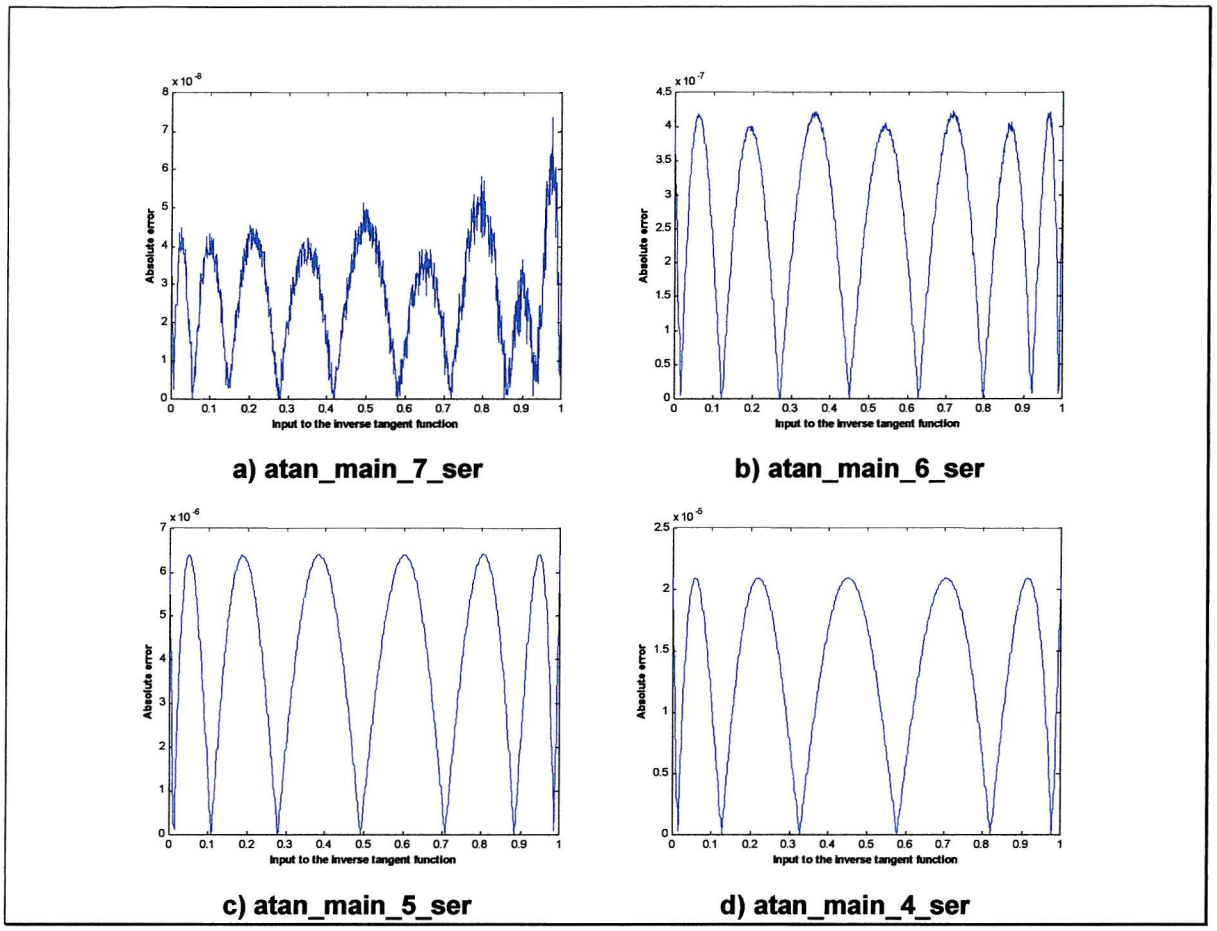


Figure C.14 Error in the inverse tangent generator using the minimax approximation for different approximation degrees

Finally, a CORDIC based engine is provided to generate this function. The units uses the CORDIC algorithm in the circular mode (m = 1) and with the input operands initialised as (x = 1, y = input operand, z =0). The accuracy of this function generator is dependent on the number of CORDIC iterations. This is shown in the results in Table C.13 and Figure C.15.

Name	Number of iterations	Maximum error
atan_main_7_COR	25	9.9845e-8
atan_main_6_COR	22	5.0590e-7
atan_main_5_COR	18	7.6204e-6
atan_main_4_COR	15	6.0870e-5

Table C.13 Maximum error in the inverse tangent generator using the CORDIC algorithm

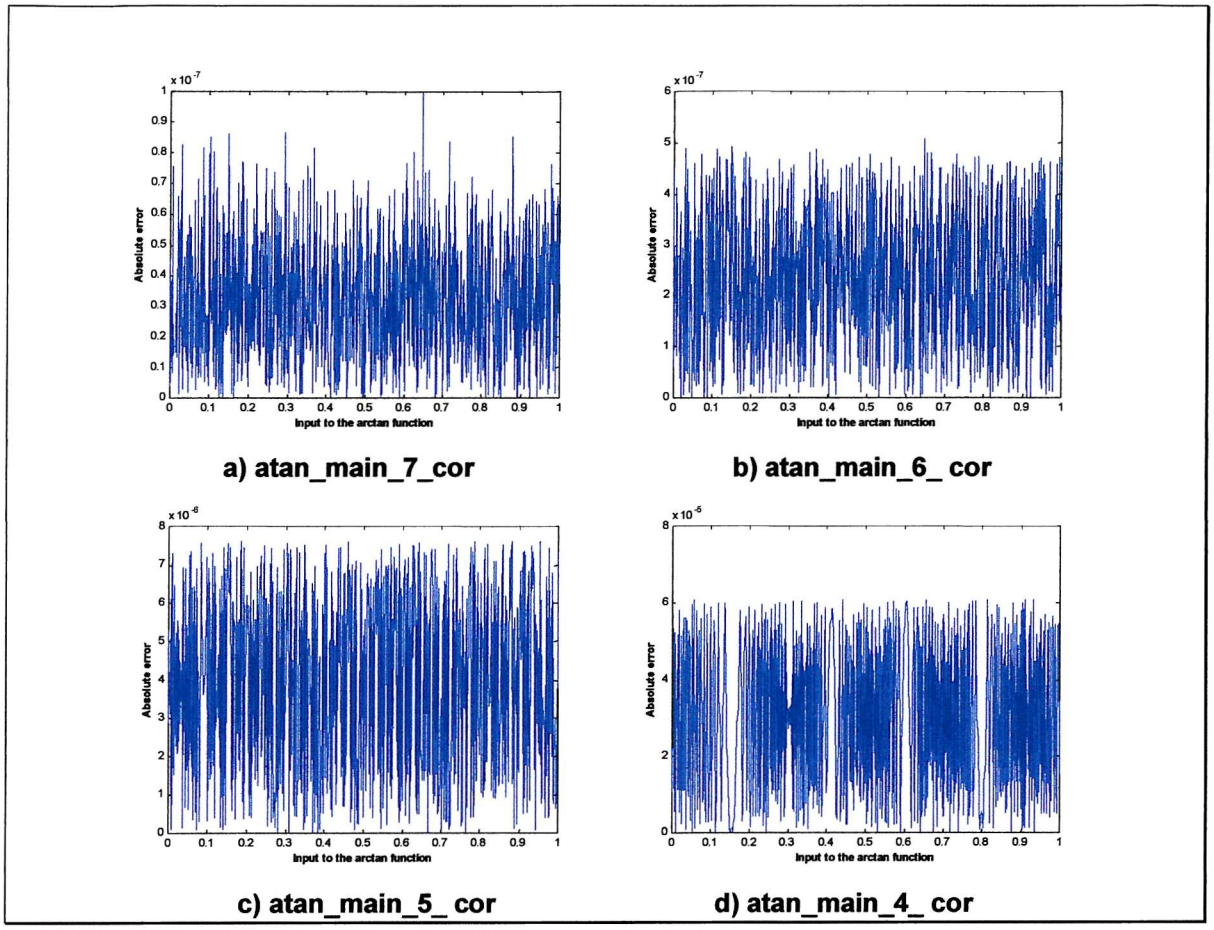


Figure C.15 Error in the inverse tangent generator using the CORDIC algorithm for different number of iterations

C.4 Logarithmic functions

The natural logarithm function is combined with the base 2 logarithm and the base 10 logarithm in a single unit[4]. The unit consists of two main components:

1. A unit that generates the natural logarithm of the input.

2. A post-processing unit that adjusts the output of the previous stage and generates the final result according to the required function.

[4] Logarithm of an arbitrary base is implemented a hierarchical using the natural logarithm unit and a floating-point divider ($\log_{base} x = \ln x / \ln base$)

The functional unit is based on one of the mathematical properties of the logarithm function:

$$\begin{aligned}\ln(1.F \times 2^E) &= \ln(1.F) + \ln(2^E) \\ &= \ln(1.F) + E \times \ln(2)\end{aligned}$$

This implies that the natural logarithm of a floating-point number can be generated using a function generator in the range [1,2].

A block diagram of the first unit is represented in Figure C.16. It consists of a type detection block, employed to detect certain situations and act according to a pre-defined regime, and the main function generator, which performs the natural logarithm calculation of the input *fraction* field.

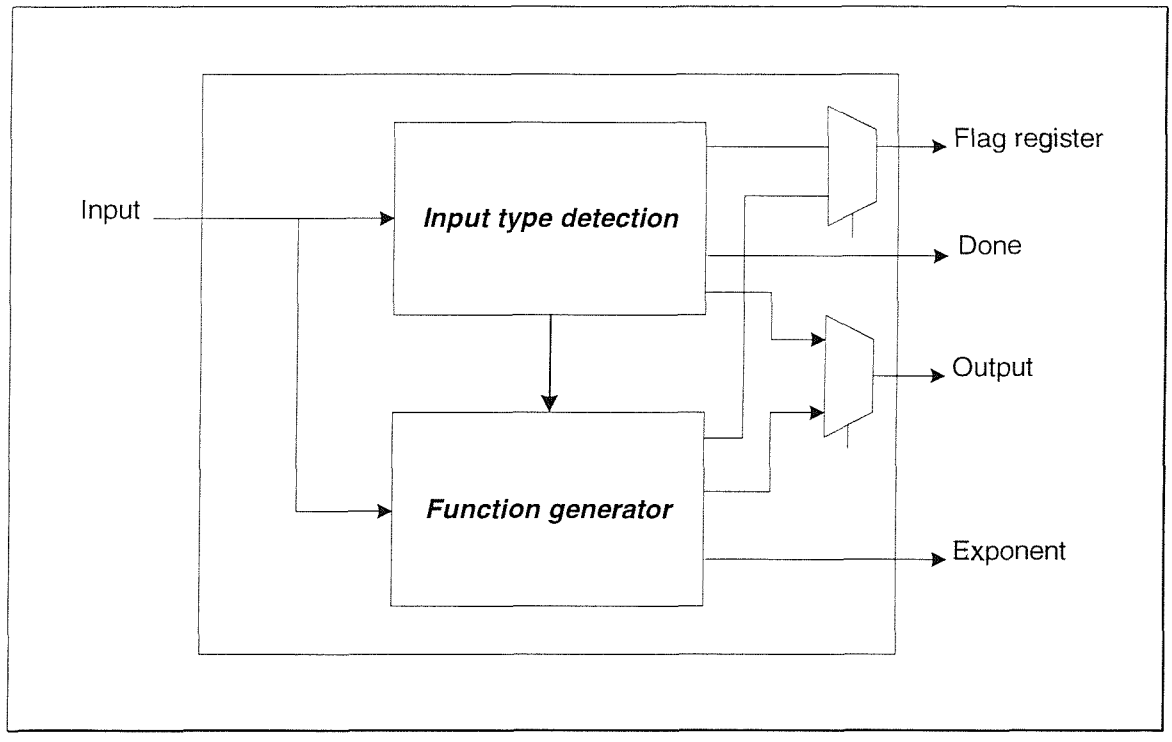


Figure C.16 Initial unit in the logarithm generator unit

Table C.9 represents the input values the type detection block detects along with the output value and the flag register content in each case. If none of these cases are detected, the execution moves to the function generation block.

For the function generation block, three sets of function generators are provided. The first is based on a single slope table lookup; the second uses a partitioned lookup table; and

finally an iterative process based on a polynomial approximation of the function is also provided.

The single slope table lookup implementation provides the fastest solution at the cost of relatively large table compared to the partitioned table. Table C.15 provides a comparison both methods for similar target accuracy, the results are summarised in Figure C.17 and Figure C.18.

Input	Output	Flag register					
		Inexact	Invalid	NAN	OVF	EUN	ZD
+∞	Quiet NAN	0	1	1	0	0	0
-∞	Quiet NAN	0	1	1	0	0	0
Sig. NAN	Quiet NAN	0	1	1	0	0	0
Quiet NAN	Quiet NAN	0	0	1	0	0	0
0	-∞	0	0	0	0	0	0
<0	Quiet NAN	0	1	1	0	0	0

Table C.14 special input cases in the logarithm function

Method	Name	Table entries	Maximum error
Single table	Ln_pre_7_lsi	1024	1.1853e-7
	Ln_pre_6_lsi	512	4.8382e-7
	Ln_pre_5_lsi	128	7.5707e6
	Ln_pre_4_lsi	64	3.0032e-5
	Ln_pre_3_lsi	32	1.1826e-4
	Ln_pre_2_lsi	16	4.594e-4
Partitioned table	Ln_pre_7_lmi	Same as LN_pre_7_lsi	
	Ln_pre_6_lmi	368	9.0378e-7
	Ln_pre_5_lmi	112	9.8758e-6
	Ln_pre_4_lmi	36	9.3764e-5

Table C.15 Maximum error in the logarithm generator using a single and partitioned table

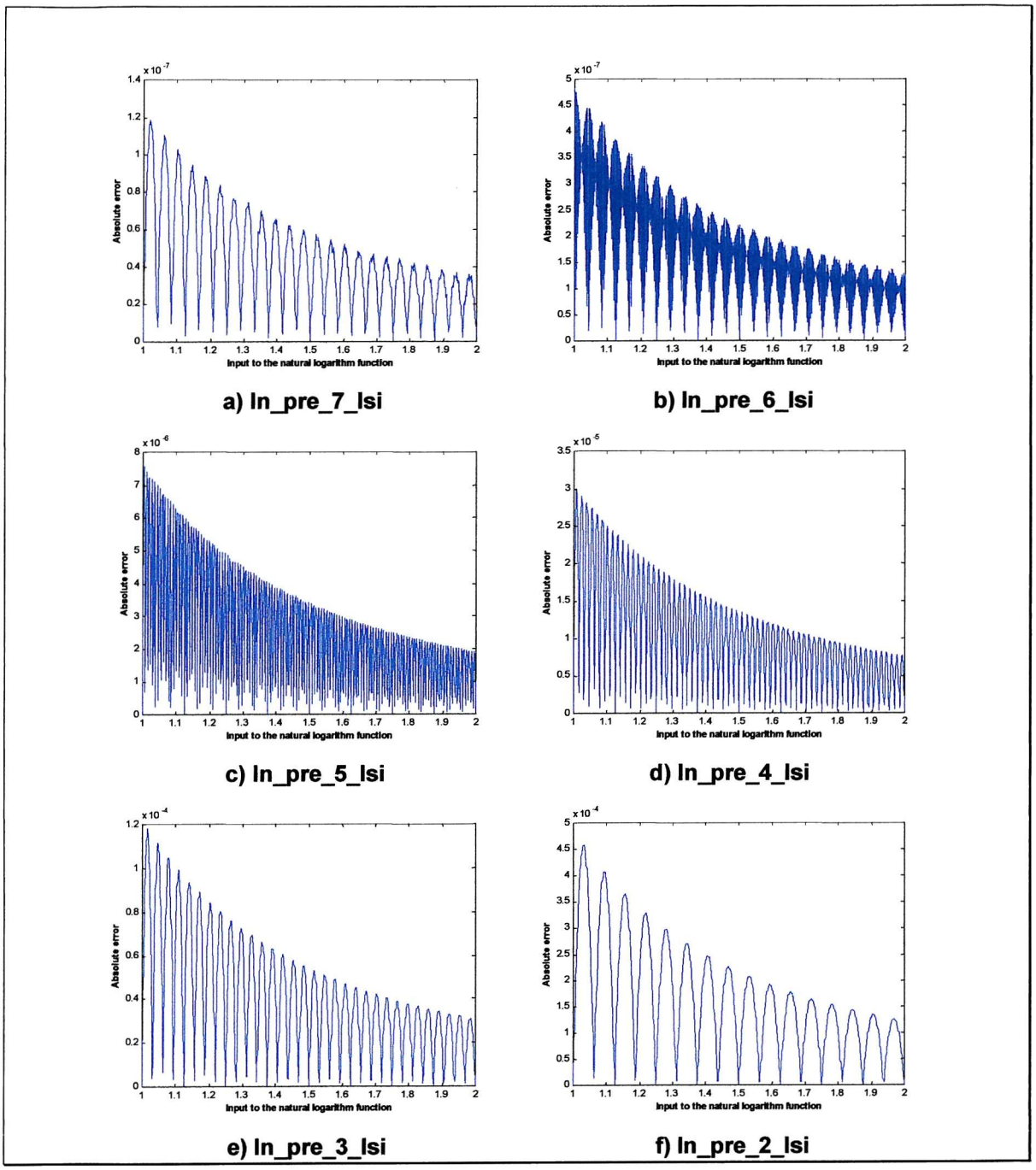


Figure C.17 Error in the natural logarithm generator using a single table and linear interpolation for different table sizes

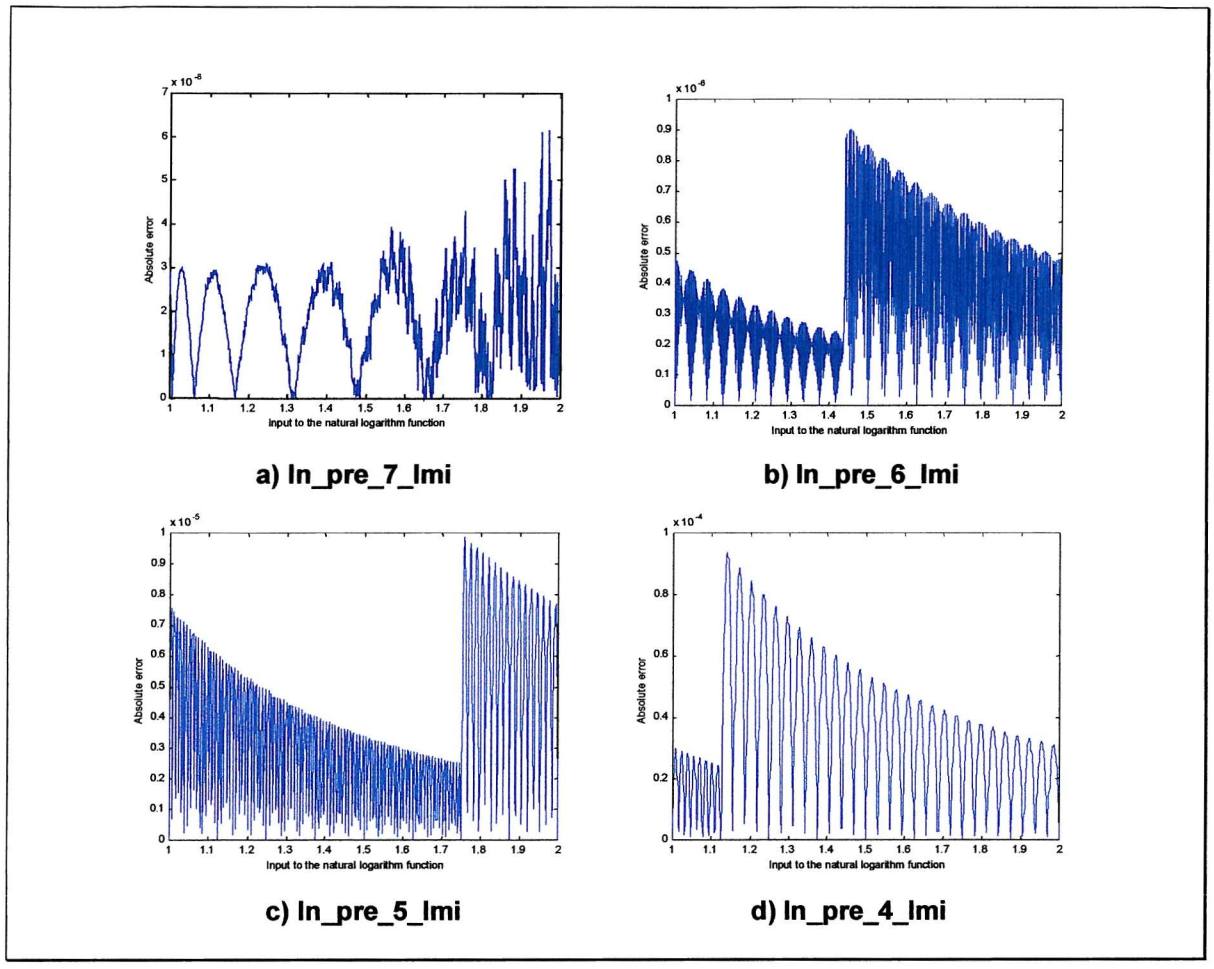


Figure C.18 Error in the natural logarithm generator using a partitioned table and linear interpolation for different table sizes

For the third set of function generators, the minimax approximation procedure provides a cheap solution in terms of area at the cost of extra delay. The unit delay is highly dependent on the target accuracy. As the required accuracy increases, the approximating polynomial degree increases and so does the number of iterations. The results in Table C.16 and Figure C.19 represent four function generators based on the minimax approximation for different accuracy target.

Name	Approximation degree	Maximum error
Ln_pre_7_ser	7	6.1669e-8
Ln_pre_6_ser	6	4.3195e-7
Ln_pre_5_ser	5	8.9136e-6
Ln_pre_4_ser	4	6.0755e-5

Table C.16 Maximum error in the logarithm generator using minimax approximation

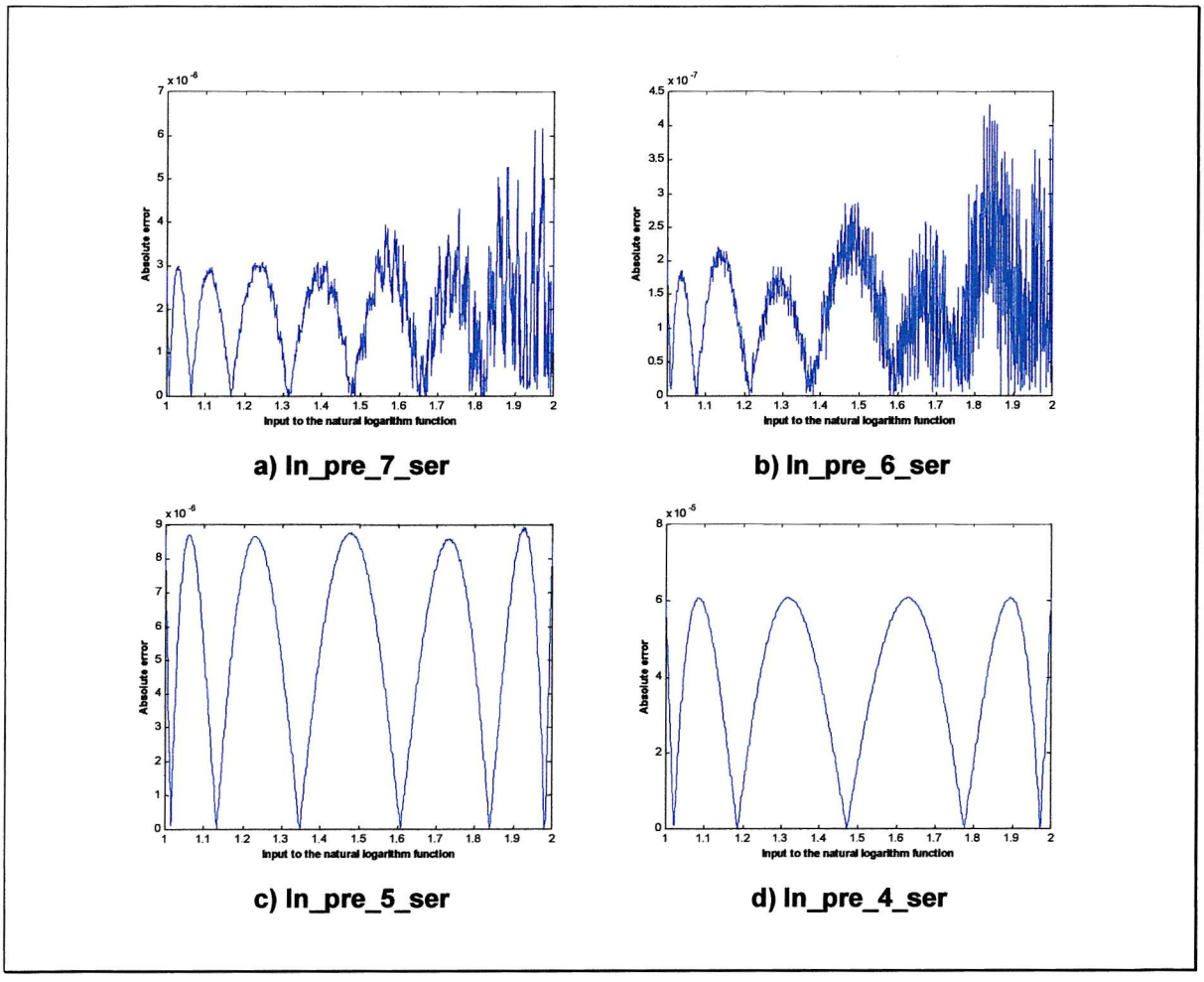


Figure C.19 Error in the natural logarithm generator using the minimax approximation and for different approximation degrees

The post-processing stage has four inputs: the output result of the previous stage; the main input exponent; a control flag (done); and the flag register. If the done flag is set, the input and the flag register are bypassed to the final output and no further processing is performed. In normal situations (done = 0), the data flow in this unit is represented in Figure C.20. It consists of three operations:

1. Multiplying the exponent by (ln2).

2. Adding the result of the previous step to the input to generate the final result.

3. This stage is required only in the case of the base 2 logarithm or base 10 logarithm, where the result is multiplied by an adjusting factor.

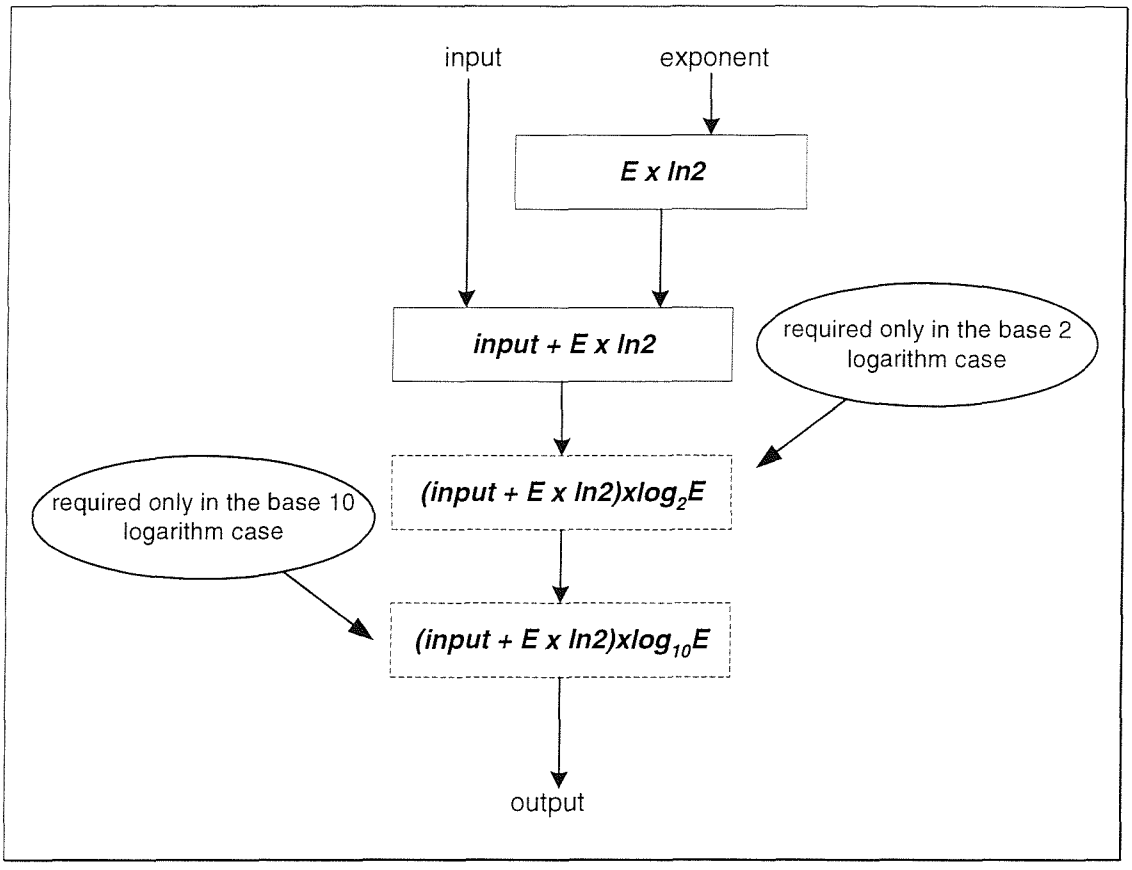


Figure C.20 Data flow in the logarithm post-processing stage

C.5 Exponential function

The exponential function generator consists of a pre-processing stage and a function generation core. The pre-processing stage performs two tasks:

1. Input operand type detection.

2. Reduces the range of the input operand to within the range of the function generation block [0,ln2].

A block diagram of the pre-processing stage is provided in Figure C.21. Input type detection is the first stage in the pre-processing step. It performs a series of tests to identify

certain cases represented in Table C.17. If any of these cases detected, the proper value is assigned to the output and the *done* flag is raised to indicate that there is no need for further processing in the following function generation block. The type detection unit also assigns the appropriate value to the *flag register*.

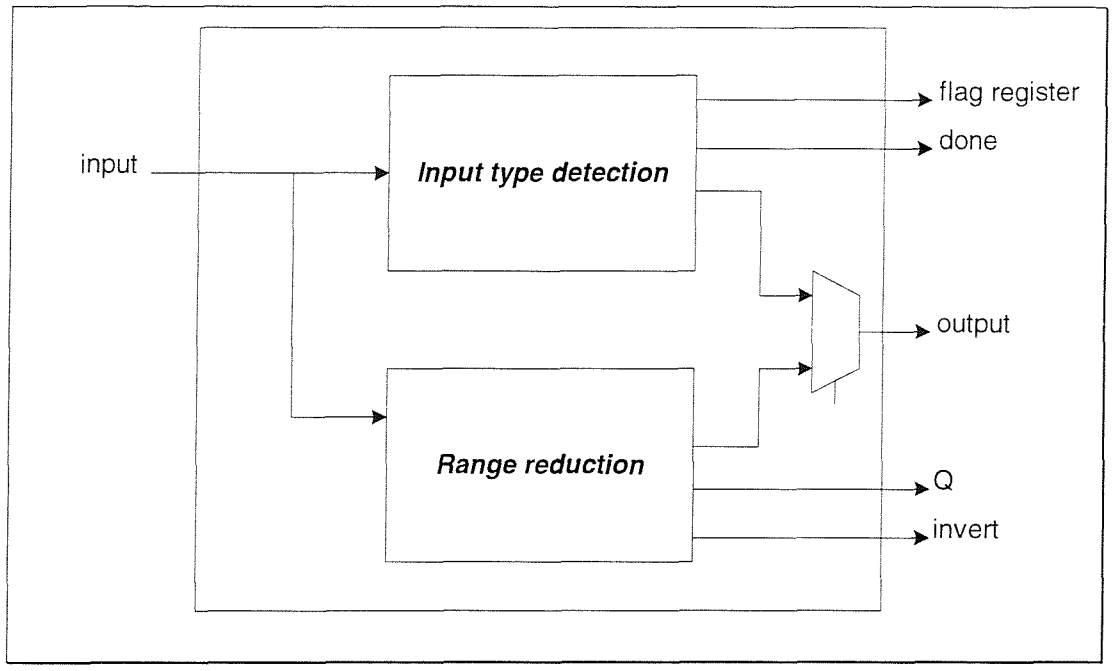


Figure C.21 Exponential pre-processing stage

If the input operand passes the type detection stage, a range reduction is performed on the input to scale it within the range $|x| \in [0,\ln 2]$[5]:

$$F \times 2^E = Q \times \ln 2 + REM \times \ln 2$$
$$\frac{F \times 2^E}{\ln 2} = Q + REM$$
$$\exp(F \times 2^E) = 2^Q \times \exp(REM \times \ln 2)$$

The procedure takes place in four steps:

1. The input is divided by ln2 (multiplied by 1/ln2) and the output result is stored in a temporary variable.

2. The fractional part of the previous step is then multiplied by (ln2) and the result is provided as the output.

[5] If the input is already within this range, the range reduction procedure is bypassed.

3. The integer part of step 1 (Q) is provided as an output.

4. If the input operand is negative, the invert flag is set to one to indicate that the final output should be inverted ($\exp(-x) = 1/\exp(x)$).

Input	Output	Flag register					
		Inexact	Invalid	NAN	OVF	EUN	ZD
$+\infty$	$+\infty$	0	0	0	0	0	0
$-\infty$	0	0	0	0	0	0	1
Sig. NAN	Quiet NAN	0	1	1	0	0	0
Quiet NAN	Quiet NAN	0	0	1	0	0	0
zero	1	0	0	0	0	0	0

Table C.17 Special input cases in the exponential function

The function generation step is provided using Table lookup based units using single slope tables. Error variation as the table size changes is shown in Table C.18 and Figure C.22. For this particular function, dividing the table into multiple sub-tables does not result in any reduction in the table size, as all the partitions require the same slope to meet the target accuracy.

Name	Slope	Table entries	Maximum error
exp_main_7_lsi	2^{-11}	1434	3.6241e-8
exp_main_6_lsi	2^{-9}	359	9.2173e-7
exp_main_5_lsi	2^{-8}	180	3.7129e-6
exp_main_4_lsi	2^{-6}	45	6.0123e-5
exp_main_3_lsi	2^{-5}	23	3.390e-4
exp_main_2_lsi	2^{-3}	6	3.900e-3

Table C.18 Maximum error in the exponential generator using a single table and linear interpolation

An iterative series method based on the minimax approximation of the function is also provided. The maximum approximation error for different approximation degrees is illustrated in Table C.19 and the results are summarised in Figure C.23.

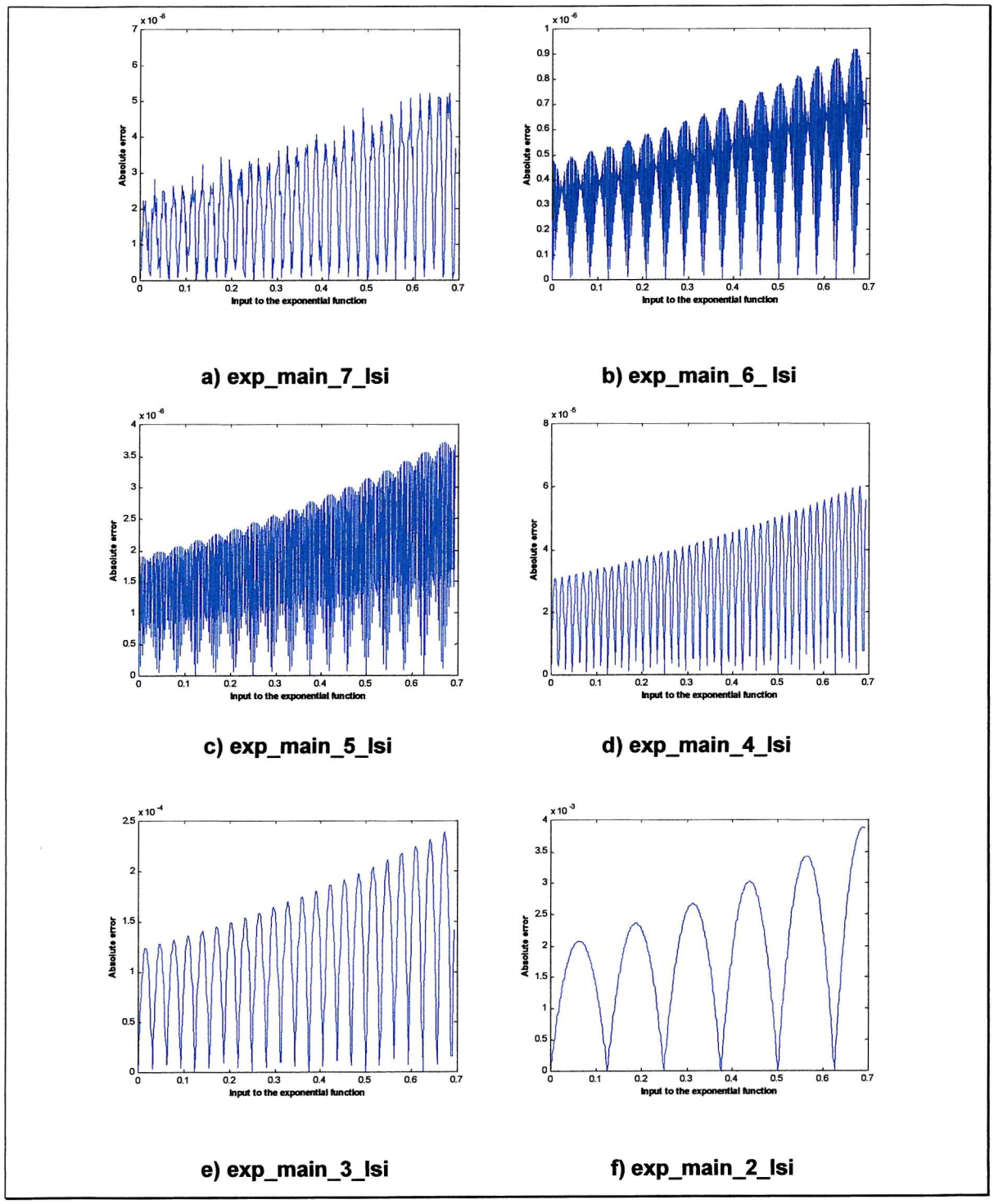


Figure C.22 Error in the exponential generator using a single table and linear interpolation for different table sizes

Name	Approximation degree	Maximum error
exp_main_7_ser	6	2.4737e-8
exp_main_6_ser	5	1.3485e-7
exp_main_5_ser	4	3.9179e-6
exp_main_4_ser	3	1.1176e-4

Table C.19 Maximum error in the exponential generator using the minimax approximation

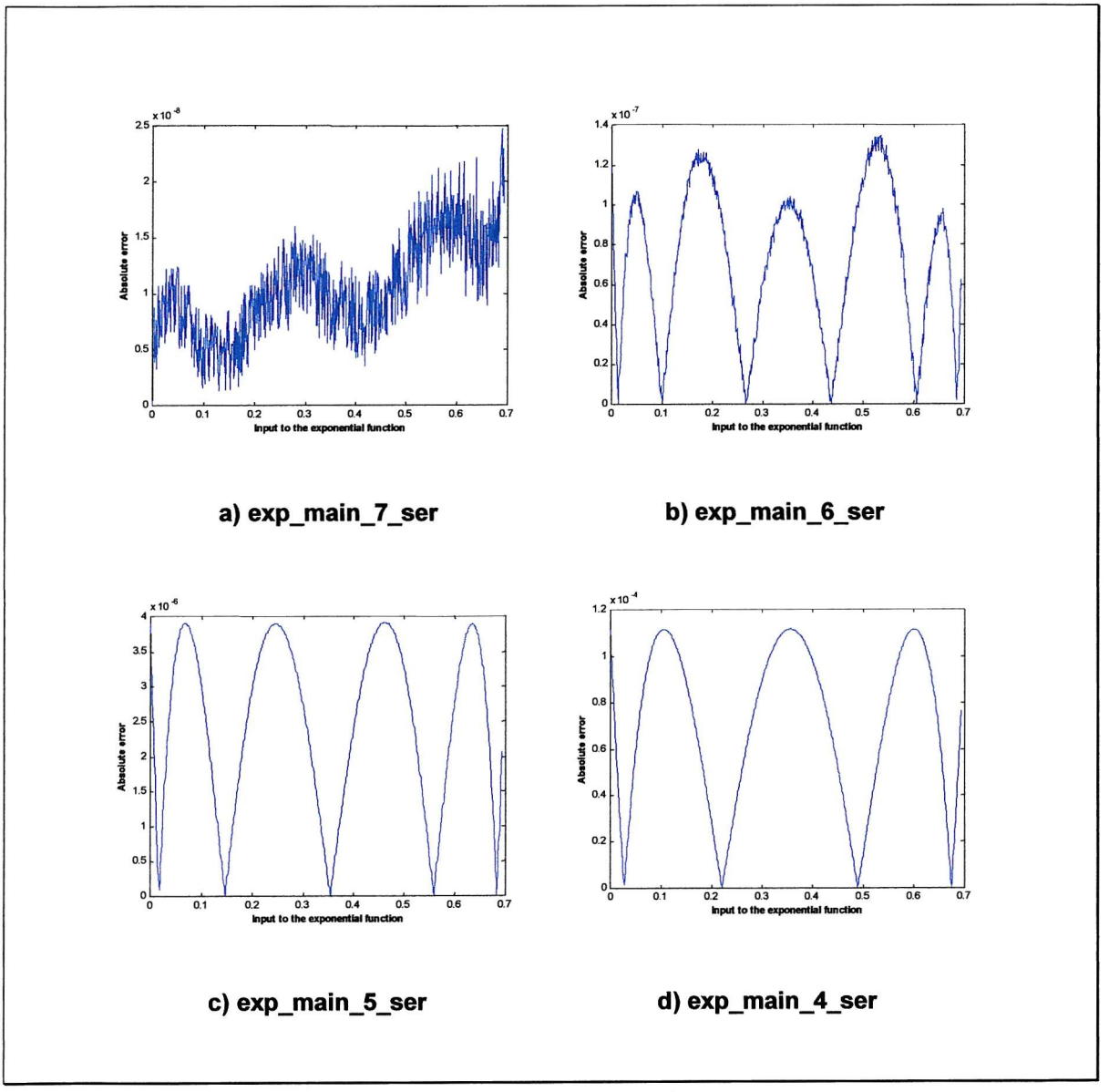


Figure C.23 Error in the exponential generator using the minimax approximation and for different approximation degrees

C.6 Square root function

The square root function generator has a simple pre-processing stage attached to a main function generation block. In addition to the type detection block which detect the cases listed in Table C.20, the pre-processing stage checks the input exponent. If an odd exponent is detected, the exponent is incremented and the fraction is shifted right, allowing the square root to be generated using the general form:

$$\sqrt{F \times 2^{E}} = \sqrt{F} \times 2^{\frac{E}{2}} \qquad ,0.5 < |F| < 2$$

A type detection block monitoring the values listed in Table C.20 is provided prior to the exponent adjustment unit. If any of these values is provided as an input operand, the output is set to a pre-defined value along with an appropriate flag register, and the operation terminates.

Input	Output	Flag register					
		Inexact	Invalid	NAN	OVF	EUN	ZD
+∞	+∞	0	0	0	0	0	0
-∞	0+j∞	0	0	0	0	0	0
Sig. NAN	Quiet NAN	0	1	1	0	0	0
Quiet NAN	Quiet NAN	0	0	1	0	0	0
zero	0	0	0	0	0	0	0

Table C.20 Special input cases in the square root function

For normal operation, two engines are provided to generate the square root function. The first is a table lookup based engine with both a single slope and multi-slope table. A comparison between the total table size for different target accuracies is provided in Table C.21 and Figure C.24.

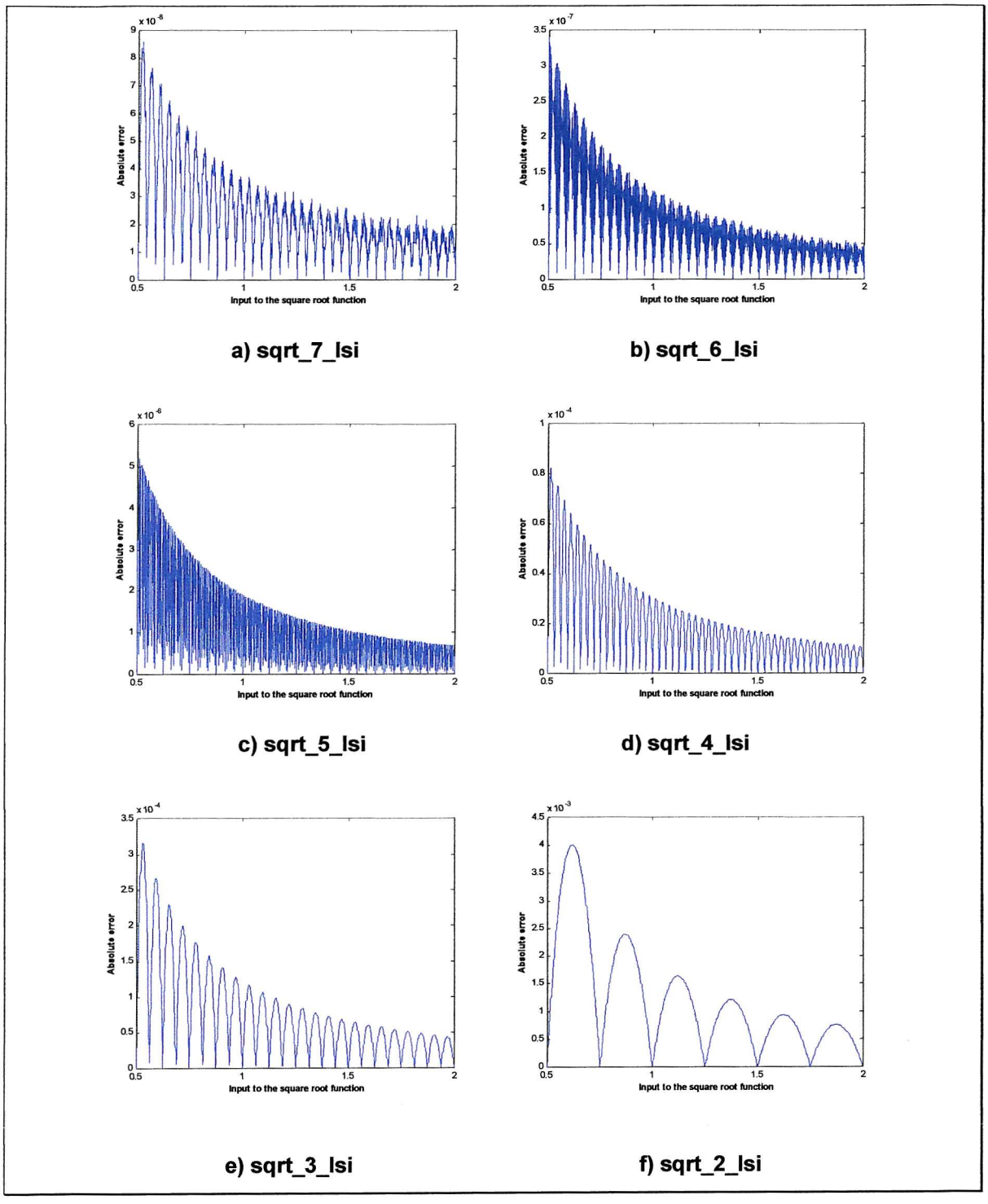
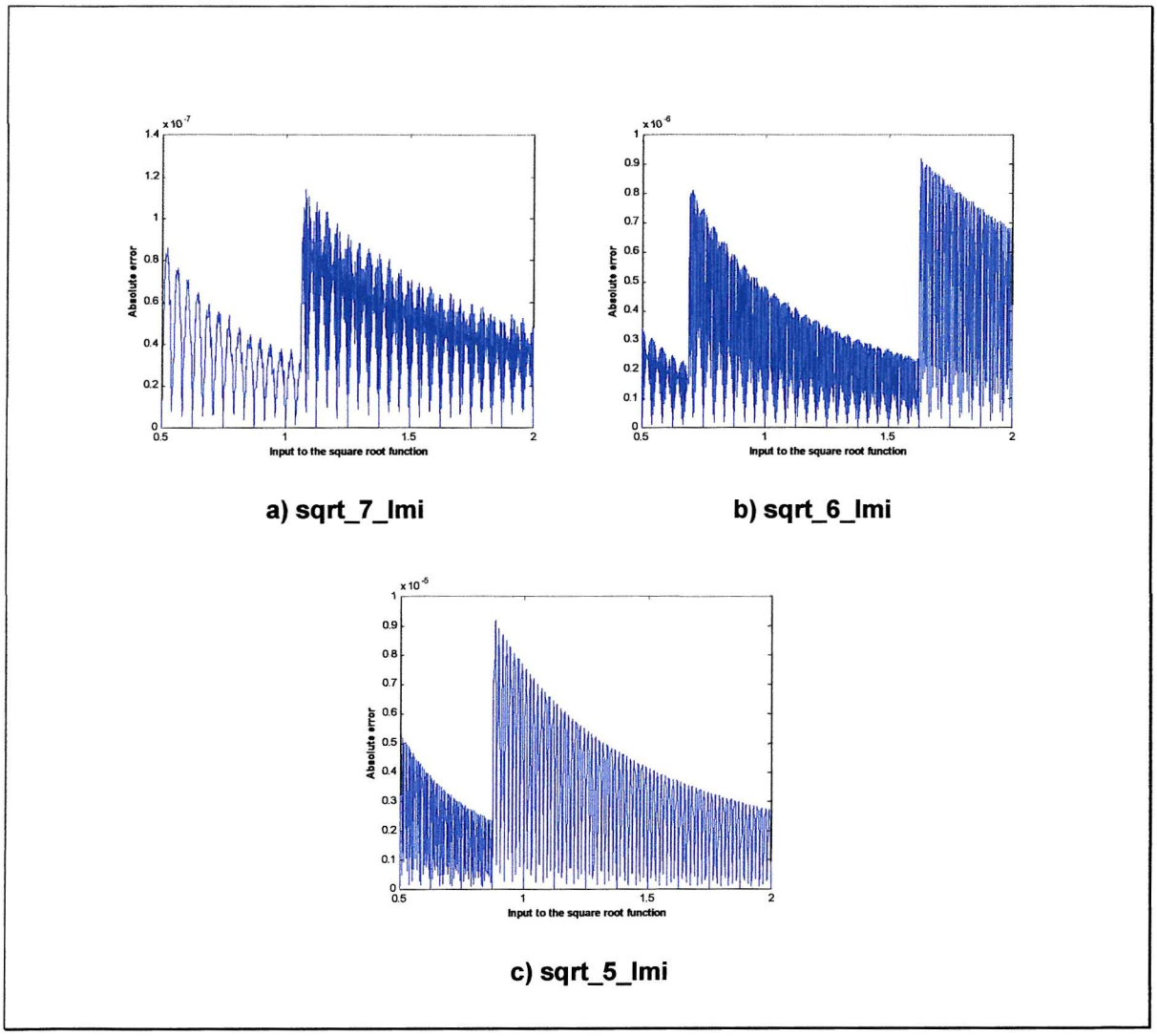


Figure C.24 Error in the square root generator implemented as a single table lookup unit and for different table sizes

Method	Name	Table entries	Maximum error
Single table	sqrt_7_lsi	1536	8.6020e-8
	sqrt_6_lsi	768	3.4048e-7
	sqrt_5_lsi	192	5.3312e-6
	sqrt_4_lsi	48	8.2379e-5
	sqrt_3_lsi	24	3.1566e-4
	sqrt_2_lsi	6	4.000e-3
Partitioned table	sqrt_7_lmi	1056	1.1410e-7
	sqrt_6_lmi	384	9.2021e-7
	sqrt_5_lmi	120	9.1946e-6
	sqrt_4_lmi	Same as sqrt_4_lsi	

Table C.21 Maximum error in the square root generator using a single and partitioned table



a) sqrt_7_lmi **b) sqrt_6_lmi**

c) sqrt_5_lmi

Figure C.25 Error in the square root generator implemented as a partitioned table lookup unit and for different table sizes

The CORDIC algorithm can also be used to generate the square root function. Error variation as the number of iterations change is shown in Table C.22 and Figure C.26. A note of particular interest here is that the angle variation (z variable, see Appendix B) has absolutely no effect of the execution, which implies that the angle calculation as well as the stored rotation values are not required to generate the square root function and can be eliminated completely from the CORDIC procedure that generates the square root.

Name	Number of iterations	Maximum error
sqrt_7_COR	12	6.2357e-8
sqrt_6_COR	10	8.5353e-7
sqrt_5_COR	9	3.4490e-6
sqrt_4_COR	8	1.3908e-5

Table C.22 Maximum error in the square root generator using the CORDIC algorithm

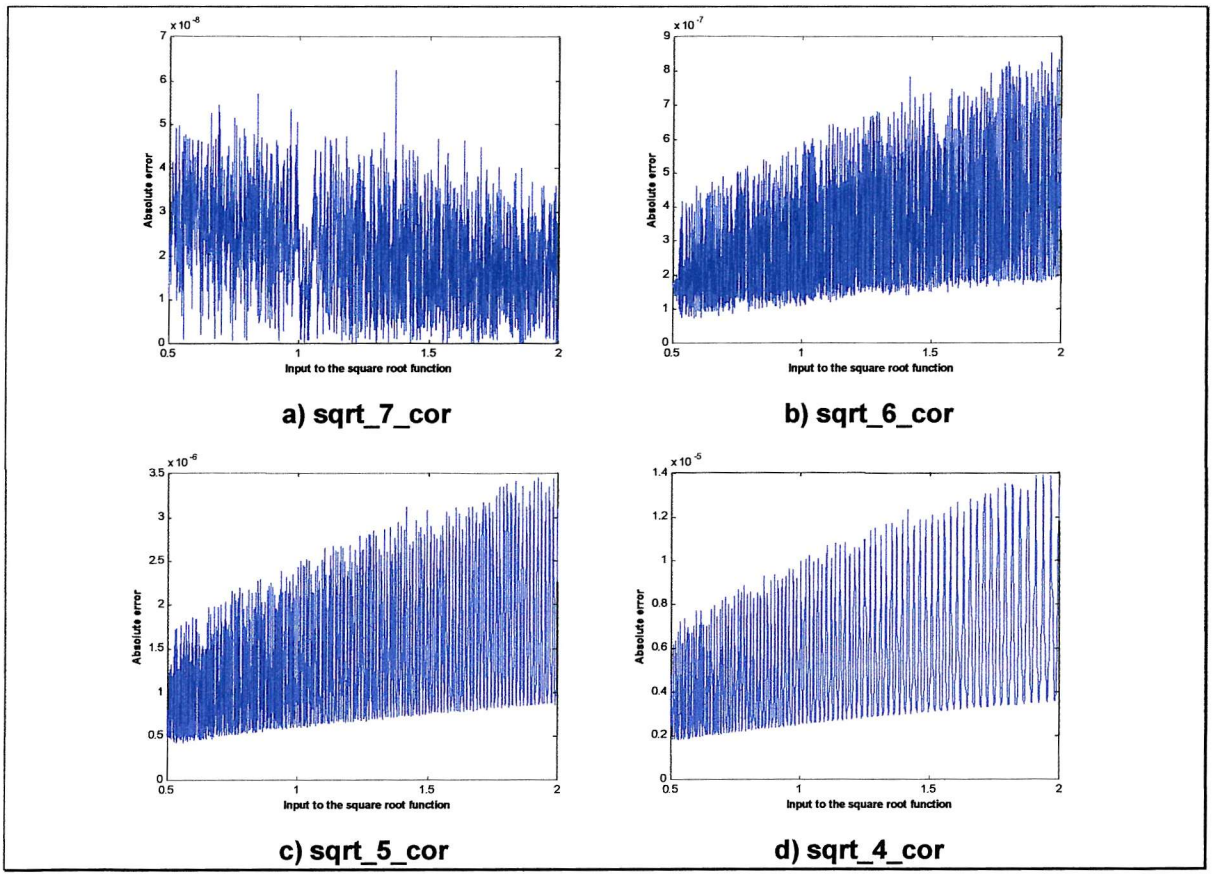


Figure C.26 Error in the square root generator using CORDIC and for different number of iterations

C.7 VHDL library

User access to the floating-point and complex functional units is provided by means of a VHDL package. Floating-point and complex functions and procedures, along with type conversion units are embodied in this package. The floating-point package declaration is provided in Listing C.1.

Listing C.1 Floating-point and complex package declaration

```
--------------------------------------------------------------------------------
--           MOODS FLOATING-POINT AND COMPLEX SYNTHESIS LIBRARY               --
--------------------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;

PACKAGE FLP_OPS IS
 ------------------------------------------------------------------------------
 -- TYPE DECLARATION
 TYPE FLOAT is array (31 downto 0) of STD_LOGIC;
 TYPE CMPLX is array (63 downto 0) of STD_LOGIC;
 TYPE CMPLX_POLAR is array (63 downto 0) of STD_LOGIC;
 TYPE STATUS is array (5 downto 0) of STD_LOGIC;
 --TYPE STD_LOGIC IS STD_LOGIC;
 ------------------------------------------------------------------------------
 -- return the real part of a complex variable
 FUNCTION RE(input : IN CMPLX) return FLOAT;
 FUNCTION RE(input : IN FLOAT) return FLOAT;

 -- return the imaginary part of a complex variable
 FUNCTION IMAG(input : IN CMPLX) return FLOAT;
 FUNCTION IMAG(input : IN FLOAT) return FLOAT;

 -- return the magnitude of a complex polar variable
 FUNCTION MAGN(input : IN CMPLX_POLAR) return FLOAT;
 FUNCTION MAGN(input : IN FLOAT) return FLOAT;

 -- return the angle of a complex polar variable
 FUNCTION ARG(input : IN CMPLX_POLAR) return FLOAT;
 FUNCTION ARG(input : IN FLOAT) return FLOAT;

 -- return the conjugate
 FUNCTION CONJ(input : IN CMPLX_POLAR) return CMPLX_POLAR;
 FUNCTION CONJ(input : IN CMPLX) return CMPLX;
 FUNCTION CONJ(input : IN FLOAT) return FLOAT;

 -- converts a complex input argument to a complex polar
 FUNCTION COMPLEX_TO_POLAR (input : IN CMPLX) return CMPLX_POLAR;
 -- same functionality but with a STD_LOGIC register support
 PROCEDURE COMPLEX_TO_POLAR_F
 ( input : IN CMPLX; output : OUT CMPLX_POLAR; FLAG_REG : OUT STATUS);


 -- converts a complex polar input argument to a complex
 FUNCTION POLAR_TO_COMPLEX (input : IN CMPLX_POLAR) return CMPLX;


 -- same functionality but with a STD_LOGIC register support
 PROCEDURE POLAR_TO_COMPLEX_F
 ( input : IN CMPLX_POLAR; output : OUT CMPLX; FLAG_REG : OUT STATUS);
```

```
-- VHDL type real and integer to float , cmplx or cmplx_polar
FUNCTION to_float (input : IN integer) return FLOAT;
FUNCTION to_float (input : IN REAL) return FLOAT;

FUNCTION to_complex (input1 : IN integer;input2 : IN integer) return CMPLX;
FUNCTION to_complex (input1 : IN real;input2 : IN real) return CMPLX;
FUNCTION to_complex (input1 : IN FLOAT;input2 : IN FLOAT) return CMPLX;
FUNCTION to_complex (input1 : IN integer;input2 : IN real) return CMPLX;
FUNCTION to_complex (input1 : IN integer;input2 : IN FLOAT) return CMPLX;
FUNCTION to_complex (input1 : IN real;input2 : IN integer) return CMPLX;
FUNCTION to_complex (input1 : IN real;input2 : IN FLOAT) return CMPLX;
FUNCTION to_complex (input1 : IN FLOAT;input2 : IN integer) return CMPLX;
FUNCTION to_complex (input1 : IN FLOAT;input2 : IN real) return CMPLX;
------------------------------------------------------------------------
-- Addition operations
FUNCTION "+" (in1, in2 : FLOAT) return FLOAT;
FUNCTION "+" (in1, in2 : CMPLX) return CMPLX;
FUNCTION "+" (in1, in2 : CMPLX_POLAR) return CMPLX_POLAR;
FUNCTION "+" (in1 : CMPLX; in2 : FLOAT) return CMPLX;
FUNCTION "+" (in1 : CMPLX_POLAR; in2 : FLOAT) return CMPLX_POLAR;


PROCEDURE FLP_ADD
( in1, in2 : IN FLOAT; output : OUT FLOAT);
PROCEDURE FLP_ADD_F
( in1, in2 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);

PROCEDURE CMPLX_ADD
( in1, in2 : IN CMPLX; output : OUT CMPLX);
PROCEDURE CMPLX_ADD_F
( in1, in2 : IN CMPLX; output : OUT CMPLX; FLAG_REG : OUT STATUS);

PROCEDURE CMPLX_ADD
(in1 : CMPLX; in2 : FLOAT; output : OUT CMPLX);
PROCEDURE CMPLX_ADD_F
(in1 : CMPLX; in2 : FLOAT; output : OUT CMPLX; FLAG_REG : OUT STATUS);


PROCEDURE POLAR_ADD
( in1, in2 : IN CMPLX_POLAR; output : OUT CMPLX_POLAR);
PROCEDURE POLAR_ADD_F
( in1, in2 : IN CMPLX_POLAR; output : OUT CMPLX_POLAR; FLAG_REG : OUT STATUS);

PROCEDURE POLAR_ADD
(in1 : CMPLX_POLAR; in2 : FLOAT; output : OUT CMPLX_POLAR);
PROCEDURE POLAR_ADD_F
(in1 : CMPLX_POLAR; in2 : FLOAT; output : OUT CMPLX_POLAR; FLAG_REG : OUT
STATUS);

------------------------------------------------------------------------
-- Subtraction operations
FUNCTION "-" (in1, in2 : FLOAT) return FLOAT;
FUNCTION "-" (in1, in2 : CMPLX) return CMPLX;
FUNCTION "-" (in1, in2 : CMPLX_POLAR) return CMPLX_POLAR;
FUNCTION "-" (in1 : CMPLX; in2 : FLOAT) return CMPLX;
FUNCTION "-" (in1 : CMPLX_POLAR; in2 : FLOAT) return CMPLX_POLAR;


PROCEDURE FLP_SUB
( in1, in2 : IN FLOAT; output : OUT FLOAT);
PROCEDURE FLP_SUB_F
( in1, in2 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);

PROCEDURE CMPLX_SUB
( in1, in2 : IN CMPLX; output : OUT CMPLX);
PROCEDURE CMPLX_SUB_F
( in1, in2 : IN CMPLX; output : OUT CMPLX; FLAG_REG : OUT STATUS);
```

```
PROCEDURE CMPLX_SUB
(in1 : CMPLX; in2 : FLOAT; output : OUT CMPLX);
PROCEDURE CMPLX_SUB_F
(in1 : CMPLX; in2 : FLOAT; output : OUT CMPLX; FLAG_REG : OUT STATUS);

PROCEDURE POLAR_SUB
( in1, in2 : IN CMPLX_POLAR; output : OUT CMPLX_POLAR);
PROCEDURE POLAR_SUB_F
( in1, in2 : IN CMPLX_POLAR; output : OUT CMPLX_POLAR; FLAG_REG : OUT STATUS);

PROCEDURE POLAR_SUB
(in1 : CMPLX_POLAR; in2 : FLOAT; output : OUT CMPLX_POLAR);
PROCEDURE POLAR_SUB_F
(in1 : CMPLX_POLAR; in2 : FLOAT; output : OUT CMPLX_POLAR; FLAG_REG : OUT
STATUS);
--------------------------------------------------------------------------
-- Multiplication operations
FUNCTION "*" (in1, in2 : FLOAT) return FLOAT;
FUNCTION "*" (in1, in2 : CMPLX) return CMPLX;
FUNCTION "*" (in1, in2 : CMPLX_POLAR) return CMPLX_POLAR;
FUNCTION "*" (in1 : CMPLX; in2 : FLOAT) return CMPLX;
FUNCTION "*" (in1 : CMPLX_POLAR; in2 : FLOAT) return CMPLX_POLAR;

PROCEDURE FLP_MULT
( in1, in2 : IN FLOAT; output : OUT FLOAT);
PROCEDURE FLP_MULT_F
( in1, in2 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);

PROCEDURE CMPLX_MULT
( in1, in2 : IN CMPLX; output : OUT CMPLX);
PROCEDURE CMPLX_MULT_F
( in1, in2 : IN CMPLX; output : OUT CMPLX; FLAG_REG : OUT STATUS);

PROCEDURE CMPLX_MULT
(in1 : CMPLX; in2 : FLOAT; output : OUT CMPLX);
PROCEDURE CMPLX_MULT_F
(in1 : CMPLX; in2 : FLOAT; output : OUT CMPLX; FLAG_REG : OUT STATUS);

PROCEDURE POLAR_MULT
( in1, in2 : IN CMPLX_POLAR; output : OUT CMPLX_POLAR);
PROCEDURE POLAR_MULT_F
( in1, in2 : IN CMPLX_POLAR; output : OUT CMPLX_POLAR; FLAG_REG : OUT STATUS);

PROCEDURE POLAR_MULT
(in1 : CMPLX_POLAR; in2 : FLOAT; output : OUT CMPLX_POLAR);
PROCEDURE POLAR_MULT_F
(in1 : CMPLX_POLAR; in2 : FLOAT; output : OUT CMPLX_POLAR; FLAG_REG : OUT
STATUS);
--------------------------------------------------------------------------
-- Division operations
FUNCTION "/" (in1, in2 : FLOAT) return FLOAT;
FUNCTION "/" (in1, in2 : CMPLX) return CMPLX;
FUNCTION "/" (in1, in2 : CMPLX_POLAR) return CMPLX_POLAR;
FUNCTION "/" (in1 : CMPLX; in2 : FLOAT) return CMPLX;
FUNCTION "/" (in1 : CMPLX_POLAR; in2 : FLOAT) return CMPLX_POLAR;

PROCEDURE FLP_DIV
( in1, in2 : IN FLOAT; output : OUT FLOAT);
PROCEDURE FLP_DIV_F
( in1, in2 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);

PROCEDURE CMPLX_DIV
( in1, in2 : IN CMPLX; output : OUT CMPLX);
PROCEDURE CMPLX_DIV_F
( in1, in2 : IN CMPLX; output : OUT CMPLX; FLAG_REG : OUT STATUS);
PROCEDURE CMPLX_DIV
(in1 : CMPLX; in2 : FLOAT; output : OUT CMPLX);
PROCEDURE CMPLX_DIV_F
(in1 : CMPLX; in2 : FLOAT; output : OUT CMPLX; FLAG_REG : OUT STATUS);
```

```
PROCEDURE POLAR_DIV
( in1, in2 : IN CMPLX_POLAR; output : OUT CMPLX_POLAR);
PROCEDURE POLAR_DIV_F
( in1, in2 : IN CMPLX_POLAR; output : OUT CMPLX_POLAR; FLAG_REG : OUT STATUS);

PROCEDURE POLAR_DIV
(in1 : CMPLX_POLAR; in2 : FLOAT; output : OUT CMPLX_POLAR);
PROCEDURE POLAR_DIV_F
(in1 : CMPLX_POLAR; in2 : FLOAT; output : OUT CMPLX_POLAR; FLAG_REG : OUT
STATUS);
--------------------------------------------------------------------------
-- Logarithm
FUNCTION LN (in1 : FLOAT) return FLOAT;
FUNCTION LOG10 (in1 : FLOAT) return FLOAT;
FUNCTION LOG2 (in1 : FLOAT) return FLOAT;
FUNCTION LOG (in1 : FLOAT; base : FLOAT) return FLOAT;

PROCEDURE LN_F
( in1 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);
PROCEDURE LOG10_F
( in1 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);
PROCEDURE LOG2_F
( in1 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);
PROCEDURE LOG_F
( in1 : IN FLOAT; base : FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);

FUNCTION LN (in1 : CMPLX) return CMPLX;
FUNCTION LOG10 (in1 : CMPLX) return CMPLX;
FUNCTION LOG2 (in1 : CMPLX) return CMPLX;
FUNCTION LOG (in1 : CMPLX; base : FLOAT) return CMPLX;

PROCEDURE LN_F
( in1 : IN CMPLX; output : OUT CMPLX; FLAG_REG : OUT STATUS);
PROCEDURE LOG10_F
( in1 : IN CMPLX; output : OUT CMPLX; FLAG_REG : OUT STATUS);
PROCEDURE LOG2_F
( in1 : IN CMPLX; output : OUT CMPLX; FLAG_REG : OUT STATUS);
PROCEDURE LOG_F
( in1 : IN CMPLX; base : FLOAT; output : OUT CMPLX; FLAG_REG : OUT STATUS);

FUNCTION LN (in1 : CMPLX_POLAR) return CMPLX_POLAR;
FUNCTION LOG10 (in1 : CMPLX_POLAR) return CMPLX_POLAR;
FUNCTION LOG2 (in1 : CMPLX_POLAR) return CMPLX_POLAR;
FUNCTION LOG (in1 : CMPLX_POLAR; base : FLOAT) return CMPLX_POLAR;

PROCEDURE LN_F
( in1 : IN CMPLX_POLAR; output : OUT CMPLX_POLAR; FLAG_REG : OUT STATUS);
PROCEDURE LOG10_F
( in1 : IN CMPLX_POLAR; output : OUT CMPLX_POLAR; FLAG_REG : OUT STATUS);
PROCEDURE LOG2_F
( in1 : IN CMPLX_POLAR; output : OUT CMPLX_POLAR; FLAG_REG : OUT STATUS);
PROCEDURE LOG_F
( in1 : IN CMPLX_POLAR; base : FLOAT; output : OUT CMPLX_POLAR;
FLAG_REG : OUT STATUS);

--------------------------------------------------------------------------
-- Trigonometric
FUNCTION SIN (in1 : FLOAT) return FLOAT;
FUNCTION COS (in1 : FLOAT) return FLOAT;
FUNCTION TAN (in1 : FLOAT) return FLOAT;
FUNCTION ASIN (in1 : FLOAT) return FLOAT;
FUNCTION ACOS (in1 : FLOAT) return FLOAT;
FUNCTION ATAN (in1 : FLOAT) return FLOAT;

PROCEDURE SIN_F
( in1 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);
PROCEDURE COS_F
( in1 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);
```

```
PROCEDURE TAN_F
( in1 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);
PROCEDURE ASIN_F
( in1 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);
PROCEDURE ACOS_F
( in1 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);
PROCEDURE ATAN_F
( in1 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);

FUNCTION SIN (in1 : CMPLX) return CMPLX;
FUNCTION COS (in1 : CMPLX) return CMPLX;

PROCEDURE SIN_F
( in1 : IN CMPLX; output : OUT CMPLX; FLAG_REG : OUT STATUS);
PROCEDURE COS_F
( in1 : IN CMPLX; output : OUT CMPLX; FLAG_REG : OUT STATUS);

FUNCTION SIN (in1 : CMPLX_POLAR) return CMPLX_POLAR;
FUNCTION COS (in1 : CMPLX_POLAR) return CMPLX_POLAR;

PROCEDURE SIN_F
( in1 : IN CMPLX_POLAR; output : OUT CMPLX_POLAR; FLAG_REG : OUT STATUS);
PROCEDURE COS_F
( in1 : IN CMPLX_POLAR; output : OUT CMPLX_POLAR; FLAG_REG : OUT STATUS);
------------------------------------------------------------------------
-- Hyperbolic
FUNCTION SINH (in1 : FLOAT) return FLOAT;
FUNCTION COSH (in1 : FLOAT) return FLOAT;
FUNCTION TANH (in1 : FLOAT) return FLOAT;
FUNCTION ASINH (in1 : FLOAT) return FLOAT;
FUNCTION ACOSH (in1 : FLOAT) return FLOAT;
FUNCTION ATANH (in1 : FLOAT) return FLOAT;

PROCEDURE SINH_F
( in1 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);
PROCEDURE COSH_F
( in1 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);
PROCEDURE TANH_F
( in1 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);
PROCEDURE ASINH_F
( in1 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);
PROCEDURE ACOSH_F
( in1 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);
PROCEDURE ATANH_F
( in1 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);

FUNCTION SINH (in1 : CMPLX) return CMPLX;
FUNCTION COSH (in1 : CMPLX) return CMPLX;

PROCEDURE SINH_F
( in1 : IN CMPLX; output : OUT CMPLX; FLAG_REG : OUT STATUS);
PROCEDURE COSH_F
( in1 : IN CMPLX; output : OUT CMPLX; FLAG_REG : OUT STATUS);

FUNCTION SINH (in1 : CMPLX_POLAR) return CMPLX_POLAR;
FUNCTION COSH (in1 : CMPLX_POLAR) return CMPLX_POLAR;

PROCEDURE SINH_F
( in1 : IN CMPLX_POLAR; output : OUT CMPLX_POLAR; FLAG_REG : OUT STATUS);
PROCEDURE COSH_F
( in1 : IN CMPLX_POLAR; output : OUT CMPLX_POLAR; FLAG_REG : OUT STATUS);

------------------------------------------------------------------------
-- Exponential
FUNCTION EXP (in1 : FLOAT) return FLOAT;
PROCEDURE POWER
( in1 : IN FLOAT; pow : IN FLOAT; output : OUT FLOAT);
```

```
  PROCEDURE EXP_F
  (in1 : FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);
  PROCEDURE POWER_F
  ( in1 : IN FLOAT; pow : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);

  FUNCTION EXP (in1 : CMPLX) return CMPLX;
  PROCEDURE POWER
  ( in1 : IN CMPLX; pow : IN FLOAT; output : OUT CMPLX);

  PROCEDURE EXP_F
  (in1 : CMPLX; output : OUT CMPLX; FLAG_REG : OUT STATUS);
  PROCEDURE POWER_F
  ( in1 : IN CMPLX; pow : IN FLOAT; output : OUT CMPLX; FLAG_REG : OUT STATUS);

  FUNCTION EXP (in1 : CMPLX_POLAR) return CMPLX_POLAR;
  PROCEDURE POWER
  ( in1 : IN CMPLX_POLAR; pow : IN FLOAT; output : OUT CMPLX_POLAR);

  PROCEDURE EXP_F
  (in1 : CMPLX_POLAR; output : OUT CMPLX_POLAR; FLAG_REG : OUT STATUS);
  PROCEDURE POWER_F
  ( in1 : IN CMPLX_POLAR; pow : IN FLOAT; output : OUT CMPLX_POLAR;
  FLAG_REG : OUT STATUS);
  ------------------------------------------------------------------------
  -- Square root
  PROCEDURE SQRT
  ( in1 : IN FLOAT; output : OUT FLOAT; imaginary : OUT STD_LOGIC);
  PROCEDURE SQRT_F
  ( in1 : IN FLOAT; output : OUT FLOAT; imaginary : OUT STD_LOGIC; FLAG_REG : OUT
  STATUS);

  FUNCTION CBRT( in1 : IN FLOAT)return FLOAT;
  PROCEDURE CBRT_F
  ( in1 : IN FLOAT; output : OUT FLOAT; FLAG_REG : OUT STATUS);

  FUNCTION SQRT ( in1 : IN CMPLX) return CMPLX;
  PROCEDURE SQRT_F
  ( in1 : IN CMPLX; output : OUT CMPLX; FLAG_REG : OUT STATUS);

  FUNCTION SQRT (in1 : IN CMPLX_POLAR) return CMPLX_POLAR;
  PROCEDURE SQRT_F
  ( in1 : IN CMPLX_POLAR; output : OUT CMPLX_POLAR;FLAG_REG : OUT STATUS);

END FLP_OPS;
```

Appendix D

Implementation details

This appendix provides a range of information concerning the floating-point library development and provides a quick reference to add new building blocks and hierarchical units to the floating-point library.

The appendix is divided into four sections: section D.1 introduces a number of file formats, namely the ICODE instruction database (inst.icd), the floating-point instruction database (flplib.ficd), the floating-point module library (flplib.mlib) and the floating-point expanded instruction set (.fxi). Section D.2 describes the ICODE file format. Section D.3 represents the ICODE file modification performed in the floating-point manipulation stage to generate the ICODE+ file. Finally, section D.4 summarises the steps required to develop and integrate a new floating-point instruction into the library.

D.1 File formats

This section describes three file formats used in the integration of the floating-point library, along with a brief description of the MOODS ICODE instruction database.

D.1.1 ICODE instruction database

MOODS ICODE instructions are defined in an ICODE instruction database file. Each entry in that file represents a new ICODE instruction and is composed of:

1. Instruction name (e.g. PLUS, MINUS, FLP_SIN).
2. A unique ICODE instruction number.
3. A datapath module instruction number representing the function required to implement this instruction from the low level module library.

4. The instruction I/O port definition.

A fragment of the ICODE instruction database file is shown in Figure D.1. The file defines ten ICODE instructions. Comments in the file are indicated by a preceding semicolon. The first three parameters in an instruction declaration can be easily identified. For example, the first instruction is a **PLUS** instruction with a unique instruction number of **14** and the function required to implement this instruction in the low level module library is function number **14**.

I/O port definition provides information on the number of I/O ports available and the width of each port in terms of the primary instruction width. For the PLUS instruction, two input ports and one output port are available. In order to specify the width of these ports, four different notations are provided:

1. ***Primary (p)***: defines the port that represents the primary width of the instruction. For example, adding two 16-bit numbers will require a plus instruction with a primary width of 16 which meets the width of the two input ports, which is why the two ports are indicated by p in the port declaration.

2. ***Fixed (f)***: defines a port that always has the same width indicated by the numerical value attached to it. The MINUSC instruction in Figure D.1 has a fixed input port of 1-bit represented by (f1), which is the carry-in port in this case.

3. ***Dependent (d)***: defines a port with a width related to the primary width. The nature of the relation is specified by the numerical value attached to it. Three possible values are available: 1 implies that the port width equals the primary width; 2 implies a width equal to the primary width + 1; and 3 indicates twice the width of the primary width. An example of a dependent port is the output port in the MULT instruction. Multiplication generates a result that is twice as wide as the primary input port, therefore the output port is defined as (d3).

4. ***Independent (i):*** defines a port of an arbitrary width. The port width in this case is the same as the width of the variable connected to it. An example of this case is the first output in the SRAMREAD instruction. The output represents a variable width address bus and is defined as (i).

```
; ICODE instruction database file
; Format of definitions is:
; <CODE name> <ICODE number> <DP fn> <No. I/P> <No. O/P> <I/O spec.>

PLUS       14   14   2 1    p p   d2
MINUS      15   15   2 1    p p   d2

MINUSC     151 15   3 2    p p f1   d1 f1
MULT       18   18       2 1    p p   d3
NE         23   23       2 1    p p   f1

ROMREAD    100 10000    2 3    i p   i f1 d1
SRAMREAD   101 10001    2 4    i p   i f1 d1

sin_cos_6_lsi 704 10704 5 2 p f1 f1 f1 f6 d1 f6
sin_cos_6_lmi 706 10706 5 2 p f1 f1 f1 f6 d1 f6
sin_cos_6_lme 707 10707 7 4 p f1 f1 f1 f6 f14 f28 d1 f6 f1 f14
```

Figure D.1 ICODE instruction database file

D.1.2 Floating-point instruction database

The floating-point instruction database file provides information that allows manipulation of the floating-point instruction in the floating-point pre-processor. A preceding semicolon indicates comment in this file. Each floating-point instruction is identified using an entry providing the following definitions:

1. A unique instruction name.

2. Instruction number.

3. A flag to indicate if the unit is part of the low level floating-point building block database or a hierarchical decomposition of a number of units.

4. A number of figures identifying the location of the external ROM interface ports in the unit I/O port list.

Figure D.2 shows an example of the floating-point instruction database with three floating point units declarations. FLP_MULT is instruction number 59, it is part of the floating-point module library and therefore the hierarchical flag is assigned to N. The floating-

point multiplier does not require an external ROM which is indicated by assigning zero to all the external ROM interface port locations. The SIN_COS unit on the other hand has a possible implementation that utilises an external ROM: an external ROM interface is defined for it. To interface to an external ROM four ports are required:

1. ***Bias register*:** defining the starting point of the function table within the external ROM. In the SIN_COS case it is port number six.

2. ***Address bus:*** an output port connects directly to the external ROM address bus. It is port number seven in the SIN_COS function.

3. ***Data bus:*** another output port that connects to the to the external ROM data bus. Port number ten in the SIN_COS unit is assigned to that bus.

4. ***Output enable:*** a control signal that controls the read operation of the external ROM. Port number eleven in the SIN_COS unit provides this signal.

Note that the hierarchical flag in the FLP_CBRT declaration is assigned to Y. This indicates that the FLP_CBRT is a hierarchical unit composed of a number of functional units and the unit should be expanded within the ICODE structure before any further processing.

```
; ICODE instruction database file
; Format of definitions is:
; <inst. name> <number> <hier. flag> <bias> <address> <data> <ctrl>

FLP_MULT   49 N 0 0 0 0
SIN_COS    157 N 6 7 10 11
FLP_CBRT   142 Y 0 0 0 0
```

Figure D.2 Floating-point instruction database file

D.1.3 Floating-point module library

The floating-point module library provides essential information on the cost of different engines provided to implement a floating-point function. Figure D.3 provides an example of the floating-point library declaring the SIN_COS instruction. Each floating-point instruction is defined by:

1. Instruction name that matches the name in the floating-point ICODE database.

2. The number of units provided to implement this function.

This is followed by entries that define the area and delay cost of each of the engines that implement the floating-point instruction. This includes:

1. Module number.

2. Accuracy figure defining the maximum error in the output result (6 implies a maximum error of 10^{-6}).

3. Total on_chip area cost in μm^2.

4. Total number of external ROM entries required.

5. An average number of clock cycles required executing the engine. The data is based on simulation results of the optimised floating-point blocks.

6. A Figure indicating increase in area cost when the unit is shared (i.e. the multiplexing cost). Comparing area costs of a number of testbenches incorporating shared floating-point units is carried out to get a close estimation of this figure.

7. ICODE unit name that indicates the name of the ICODE instruction that represents this possible implementation of the main function.

8. Names of fixed-point units that are utilised in the design and have a major effect on the total design area and/or delay cost. For example a fixed point multiplier, a fixed point divider, or a barrel shifter.

```
; <instruction name>
; <number of modules>
; <unit number> <accuracy> <area cost> <ext. ROM> <delay> <sharing cost>
; <unit name> <fixed_point units>
sin_cos
26
1 6 105616 805  30 33000 sin_cos_6_lsi fixed_mult
2 6 109909 679  34 33000 sin_cos_6_lmi fixed_mult
3 6 469000 0    20 25875 sin_cos_6_lsi fixed_mult
4 6 387000 0    24 25875 sin_cos_6_lmi fixed_mult
5 6 88000  0    76 24840 sin_cos_6_ser fixed_mult
```

Figure D.3 Floating-point Module library file

D.1.4 Floating-point expanded instruction

A floating-point expanded instruction is a sequential implementation of a floating-point function, which is dynamically expanded within the internal design representation during the floating-point pre-processing stage and prior to the optimisation phase. This evolved from the need to generalise the implementation of a hierarchical functional unit and split it up into components to reduce the complexity that faces the optimisation routine. An expanded ICODE instruction format (fxi) is provided to facilitate this decomposition. Figure D.4 [1]shows an example of a *fxi* file. It consists of five main parts:

1. Header declaring the expanded instruction argument. Three arguments are provided in this case: input, output, and flag_reg.

2. Alias declaration defines a slice of an I/O port or an internal register. It has the general format:

   ALIAS <name> <lsb> <msb> <from> <lsb> <msb>

   For example, line 7 declares a slice of the second port named (%2) with an ascending 0 to 31 range. The alias is used as an alternative name to the port with any modification to the alias resulting in a similar modification to the port.

3. Register declaration defines a new internal register. It has the general format:

   REG <name> <lsb> <msb>

[1] Note that the line numbers in the figure are for illustration purposes only and are not part of the file format.

For example, line 12 declares a register named (%6) with an ascending 0 to 31 range.

4. The instructions block defines a sequence of ICODE operations on the declared aliases and internal registers. Each instruction is provided as an opcode followed by a list of operands. Binary constants can be used as operands using the (#) operator. Each instruction within the block is either an original ICODE instruction or a newly added floating-point operation.

5. The final line in the file provides the error propagation information, which indicate the contribution each building block has on the total instruction error. These figures are utilised by the floating-point pre-processing units to decide the accuracy of each building block based on the target accuracy of the hierarchical unit.

```
1.  input output flag_reg
2.  -- %1 = variable or alias name
3.  -- 0 31 = lsb msb
4.  -- 1   = from input number 1
5.
6.  alias %1 0 31 1 0 31  -- input
7.  alias %2 0 31 2 0 31  -- output
8.  alias %3 0 5 3 0 5    -- flag_reg
9.  alias %4 0 0 1 31 31  -- input_sign
10. alias %5 0 30 1 0 30  -- input_rest
11. alias %17 0 0 2 31 31 -- output_sign
12. reg %6 0 31
13. reg %10 0 31
14. reg %11 0 30
15. reg %12 0 0
16. reg %13 0 0
17. reg %14 0 31
18. reg %15 0 31
19. {
20.    move #00111110101010101010101010101010 %10
21.    move %4 %13
22.    move %5 %11
23.    move #0 %12
24.    concat %12 %11 %6
25.    flp_ln_f %6 %14 %3
26.    flp_mult_f %10 %14 %15 %3
27.    flp_exp_f %15 %2 %3
28.    move %4 %17
29. }
30. 0 0 0 0 0 2 0 1 0
```

Figure D.4 Expanded ICODE instruction file

D.2 The ICODE format

The ICODE format is a textural representation of the behaviour of the system at the register transfer level. The system is represented by a number of *modules*, with the top level identified by a special *program* declaration. Each module has an optional IO parameter list, defining the module interface to the higher level. A module contains a number of ICODE *processes*. Each process consists of an *instruction* and an *activation list*, which defines the processes to be activated once the current process concludes. ICODE instructions operate on explicitly declared variables (*register*, *alias*, *counter*, *memory*), and/or temporary variables. It may be thought of as a kind of hardware assembly language. In MOODS, the high level behavioural input (VHDL). ICODE is "source language neutral", in that translation from other high level languages (ANSI-C, SystemC) is just as feasible.

Name	Format
Program declaration	**PROGRAM** *program_name io_list [info]*
Module declaration	**MODULE** *module_name io_list [info]*
Port declaration	**INPORT** | **OUTPORT** *port_name port_range*
Register declaration	**REGISTER** *register_name register_range*
Counter declaration	**COUNTER** | **COUNTDOWN** *counter_name counter_range*
Alias declaration	**ALIAS** *alias_name alias_range* **FROM** *source_name source_sub_range*
Constant declaration	**#** *integer value*
Integer value	*decimal* | **%***binary_value* | **&***octal_value* | **$***hex_value*
Information (info)	*{specifier : value}*
ROM declaration	**ROM** *name data_range* **ADDRESS** *address_range* **DATA** *rom_content*
RAM declaration	**RAM** *name data_range* **ADDRESS** *address_range*
Activation list	*Instruction_label [,Instruction_label]*
Unconditional activation	**ACT** *activation_list*
Activate if true	**ACTT** *activation_list*
Activate if false	**ACTF** *activation list*
Collect instruction	**COLLECT** *number_of_collects*
Conditional instruction	**IF** | **IFNOT** *variable_name act_if_true act_if_false [info]*
Count instruction	**COUNT** *counter, [step], limit act_if_true act_if_false [info]*
Decode instruction	**DECODE** *variable [info]* *{***CASE** *constant unconditional_activation [info]}*
Switch instruction	**SWITCHON** *variable [info]* *{***CASE** *constant unconditional_activation [info]}* **DEFAULT** *unconditional_activation [info]*
Module call instruction	**MODULEAP** *module_name io_list [info]*
Memory read instruction	**MEMREAD** *memory_variable_name, address, output [info]*
Memory write instruction	**MEMWRITE** *input, memory_variable_name, address [info]*
General instruction	**EQ** | **NE** | **GR** | **GE** | **LS** | **LE** | **AND** | **OR** | **XOR** | **NOT** | **NEG** | **PLUS** | **MINUS** | **MULT** | **DIV** | **LSHIFT** | **RSHIFT** | **ROR** | **ROL** | **MOVE** | **SETTRUE** | **HIGHZ** | **CONCAT**

Table D.1 ICODE format definition

System execution starts with the first process in the top-level program. Other modules are executed using the MODULEAP instruction, which takes as parameters the module name and a list of variables to interface to the IO ports. Table D.1 provides a complete definition

of the ICODE format, while the listing in Figure D.5 illustrates most of the ICODE features.

```
PROGRAM   dummy   a , b , c , enable , sel , result          main program declaration

                                                            IO port declarations
INPORT      a           [0:6]
INPORT      b           [0:6]
INPORT      c           [0:6]
INPORT      enable      [1:1]
INPORT      sel         [0:2]
OUTPORT     result      [0:6]

                                                            register declarations
REGISTER    t_a         [0:6]
REGISTER    t_b         [0:6]
REGISTER    t_c         [0:6]
REGISTER    temp        [0:6]


COUNTER     i           [1:2]
                                                                                internal ROM
rom         internal_rom  [0:6] address [0:4] data [#%1000110,#%1101100,#%1101111,#%1100001,#%1110100]

process                                  constant binary value
label
.L2         move        #%0000000 , t_a
            move        #%0000000 , t_b
            move        #%0000000 , t_c
                                                    temporary
.L5         move        enable , 17                 variable

.L6         ne          17 , enable , 18
            if          18     actt L8     actf L6      {pt:0.8, pf:0.2}
.L8         eq          enable , #1 , 19
            if          19     actt L10    actf L5      {pt:0.8, pf:0.2}
.L10        move        #0 , i
.L11        plus        t_a , a , t_a
            plus        t_b , b , t_b
            plus        t_c , c , t_c
                                        activate if true        activate if false
            count       i  , #2     actt L15    actf L11     {pt:0.25, pf:0.75}
.L15        move        sel , 20
            switchon    20
                                integer in decimal
            case        #3      act L27      {pt:0.2}
            case        #2      act L26      {pt:0.2}
            case        #1      act L25      {pt:0.2}
            case        #0      act L23      {pt:0.2}
            default             act L28      {pt:0.2}
            endcase
.L23        moduleap  average   t_a , t_b , t_c , temp

            move        temp , result   act L2      unconditional activation

.L25        move        t_a , result     act L2
.L26        move        t_b , result     act L2
.L27        move        t_c , result     act L2
.L28        memread     internal_rom[sel],  21
            move        21 , result      act L2
            endmodule   dummy       module label matches declaration

MODULE    average   in1 , in2 , in3 , avr          sub-module declaration


INPORT      in1         [0:6]
INPORT      in2         [0:6]
INPORT      in3         [0:6]
OUTPORT     avr         [0:6]

            plus        in1 , in2 , 22
            plus        22 , in3 , 23
            div         23 , #3 , avr
            endmodule   average
```

Figure D.5 Example ICODE file

D.3 ICODE+

The ICODE+ file is the floating-point optimiser output that contains all the necessary information required by MOODS to implement the circuit. ICODE+ generation is a four stage process. The first two stages occur before the optimisation algorithm, and the final two stages are required once the functional unit mapping is decided:

1. Initially, a global flag register port is added (if applicable) as an output port; this is connected to the floating-point unit internal flag register to indicate any exception during the unit execution.

2. In the second stage, hierarchical units are expanded into sub-blocks. The operation involves declaring a set of temporary variables and aliases to provide a communication path between the unit sub-components.

3. At this stage, each floating-point functional unit is replaced with the appropriate expanded module name within the floating-point module library.

4. The external ROM interface (if required) is provided at this stage. It involves declaring the *address bus*, the *data bus* and the ROM *control signal* and interfacing them to the appropriate floating-point unit. An *address bias* constant, will also be assigned to each floating-point unit to indicate the lookup-table location within the external ROM.

By way of an example, consider the VHDL behavioural description in Figure D.6 along with its ICODE file. The equivalent ICODE+ file is represented in Figure D.7. Initially, a flag register is declared as an output port (line 8) and is interfaced to the cubic root unit[2]. Then the *cbrt()* unit is expanded into its sub components (lines 46 to 60). Note that the exponential and natural logarithm functions within the *cbrt()* unit are again expanded into further building blocks (lines 52 to 54, and lines 56 to 58 respectively). The stage also involves declaring a number of temporary registers (lines 12 to 27) and a number of aliases (lines 30 to 41) declaring sub-ranges of internal variables. Finally, once the optimisation is performed, the floating-point functional units are replaced with expanded

[2] Note that the *cbrt()* unit has been expanded into its sub-components, which hides the flag register interface.

module declarations. In this case the natural logarithm and the exponential functions are replaced with external table lookup based implementations named ln_pre_7_lse, and exp_main_7_lse.

Finally, the external ROM interface is implemented within the design. Two output ports and one input port are declared. The output ports represent the ROM address bus (line 10) and the ROM control port (line 9) and the input port representing the ROM data bus (line 5). A register representing an *address bias* (line 28) is also required to indicate the starting point of the natural logarithm and exponential units lookup tables within the external ROM. The register is connected to the two units and is assigned a *constant* value each time a unit is executed.

VHDL

```
1. Entity test is
2. port (input  : in FLOAT;
3.       enable : in bit;
4.       ready : out bit;
5.       result : out FLOAT);
6. end;
7. Architecture behave of test is
8. Begin
9.    Process
10.   Begin
11.     ready <= '0';
12.     wait until enable = '1';
13.     result <= cbrt(input);
14.     ready <= '1';
15.     wait until enable = '0';
16.   End Process;
17. End behave;
```

ICODE

```
1. PROGRAM    test   input, enable,  ready, result
2.
3. inport     input        [0:31]
4. inport     enable       [1:1]
5. outport    ready        [1:1]
6. outport    result       [0:31]
7.
8. .PR1           move   #%0,  ready
9. .if4_false_3               eq      enable, #1,  5
10.               if       5               ACTT if4_true_5    ACTF if4_false_3   {pt:0.8, pf:0.2}
11. .if4_true_5   cbrt     input,  result
12.               move     #%1,  ready
13. if8_false_7   eq       enable, #0,  6
14.               if       6               ACTT PR1    ACTF if8_false_7   {pt:0.8, pf:0.2}
15.               ENDMODULE          test
```

Figure D.6 Example VHDL and ICODE files

ICODE+

```
1.  PROGRAM  test  input , enable , rom_data_bus , ready , result , global_flag_reg , rom_en , rom_address_bus
2.
3.  INPORT    input                 [0:31]
4.  INPORT    enable                [1:1]
5.  INPORT    rom_data_bus          [0:27]
6.  OUTPORT   ready                 [1:1]
7.  OUTPORT   result                [0:31]
8.  OUTPORT   global_flag_reg       [0:5]
9.  OUTPORT   rom_en      [0:0]
10. OUTPORT   rom_address_bus       [0:13]
11.
12. REGISTER  fxi_reg1      [0:31]
13. REGISTER  fxi_reg2      [0:31]
14. REGISTER  fxi_reg3      [0:30]
14. REGISTER  fxi_reg4      [0:0]
15. REGISTER  fxi_reg5      [0:0]
16. REGISTER  fxi_reg6      [0:31]
17. REGISTER  fxi_reg7      [0:31]
18. REGISTER  fxi_reg8      [0:31]
19. REGISTER  fxi_reg9      [0:0]
20. REGISTER  fxi_reg10     [0:7]
21. REGISTER  fxi_reg11     [0:5]
22. REGISTER  fxi_reg12     [0:31]
23. REGISTER  fxi_reg13     [0:7]
24. REGISTER  fxi_reg14     [0:0]
25. REGISTER  fxi_reg15     [0:0]
26. REGISTER  fxi_reg16     [0:5]
27. REGISTER  fxi_reg17     [0:0]
28. REGISTER  rom_address_bias      [0:13]
29.
30. ALIAS     fxi_alias1      [0:31]  from    input               [0:31]
31. ALIAS     fxi_alias2      [0:31]  from    result              [0:31]
32. ALIAS     fxi_alias3      [0:5]   from    global_flag_reg     [0:5]
33. ALIAS     fxi_alias4      [0:0]   from    input               [31:31]
34. ALIAS     fxi_alias5      [0:30]  from    input               [0:30]
35. ALIAS     fxi_alias6      [0:0]   from    result              [31:31]
36. ALIAS     fxi_alias7      [0:31]  from    fxi_reg1                    [0:31]
37. ALIAS     fxi_alias8      [0:31]  from    fxi_reg6                    [0:31]
38. ALIAS     fxi_alias9      [0:5]   from    fxi_alias3                  [0:5]
39. ALIAS     fxi_alias10     [0:31]  from    fxi_reg7                    [0:31]
40. ALIAS     fxi_alias11     [0:31]  from    fxi_alias2                  [0:31]
41. ALIAS     fxi_alias12     [0:5]   from    fxi_alias3                  [0:5]
42.
43. .L2     move            #%0 , ready
44. .L3     eq              enable , #1 , 5
45.         if              5    actt L5    actf L3    {pt:0.8, pf:0.2}
46. .L5     move            #%00111110101010101010101010101010 , fxi_reg2
47.         move            fxi_alias4 , fxi_reg5
48.         move            fxi_alias5 , fxi_reg3
49.         move            #%0 , fxi_reg4
50.         concat          fxi_reg4 , fxi_reg3 , fxi_reg1
51.         move            #%00000000000000 , rom_address_bias
52.         ln_pre_7_lse    fxi_alias7 , rom_address_bias , rom_data_bus , fxi_reg8 , fxi_reg9 , fxi_reg10 ,\
53.                         fxi_reg11 , rom_en , rom_address_bus
54.         ln_post         fxi_reg8 , fxi_reg10 , fxi_reg9 , fxi_reg11 , fxi_alias8 , fxi_alias9
55.         flp_mult_f      fxi_reg2 , fxi_reg6 , fxi_reg7 , fxi_alias3
56.         exp_pre         fxi_alias10 , fxi_reg12 , fxi_reg13 , fxi_reg15 , fxi_reg14 , fxi_reg17 , fxi_reg16
57.         move            #%00010000000000 , rom_address_bias
58.         exp_main_7_lse fxi_reg12 , fxi_reg13 , fxi_reg15 , fxi_reg14 , fxi_reg17 , fxi_reg16 , rom_address_bias ,\
59.                         rom_data_bus , fxi_alias11 , fxi_alias12 , rom_en , rom_address_bus
60.         move            fxi_alias4 , fxi_alias6
61.         move            #%1 , ready
62. .L19    eq              enable , #0 , 6
63.         if              6    actt L2    actf L19    {pt:0.8, pf:0.2}
64.       endmodule     test
```

Figure D.7 Example ICODE+ file

D.4 Adding a new instruction

Two types of floating-point unit can be integrated within the floating-point synthesis library: a normal floating-point functional unit, and a hierarchical floating-point functional unit. In both cases, knowledge of the nature of the function is required by the system in order to be able to handle the new function. To achieve this, a number of steps are required:

1. Provide an entry in the floating-point ICODE instruction database file to declare the new instruction and assign it a new unique instruction number.

2. At this point, if we are adding a new hierarchical instruction composed of pre-defined building blocks, all that is necessary is to provide an expanded ICODE instruction file describing the sequence of data execution within the new instruction, an example of which is provided in Figure D.4.

3. In the more general case of dealing with a new instruction, a number of possible implementations of the instruction in the form of a set of expanded modules should be provided. Details about generating expanded modules are provided in Chapter 4.

4. A block defining the parameters of all possible implementations of the new function should be added to the floating-point module library file. This is an important step, since the information provided here will be used to guide the optimisation procedure during the high level binding process. An example of the floating-point module library file is available in Figure D.3.

5. Each possible implementation of the function should be assigned a unique ICODE instruction in the ICODE instruction database file, in order to allow the MOODS synthesis system to handle the expanded module expansion and optimisation process. For example, the last three entries in Figure D.1 define three different implementations for the SIN_COS instruction, each represented by a different expanded module and therefore assigned to a separate ICODE instruction.

6. If a module is to be implemented using an external ROM, a file that contains an ASCII text format of the ROM entries which has the same name as the expanded module and with a (.ROM) extension should be provided to be used in generating the external ROM data.

By following these steps, the new instruction can be integrated within the floating-point synthesis library. It is worth mentioning that the user should try to preserve the hierarchy of the floating-point functional unit before generating the expanded model. For example, during the floating-point library development, an optimised fixed-point multiplier and fixed-point divider are provided as expanded modules in the MOODS template library. The currently available floating-point building blocks invoke these modules every time a multiplier and divider is required. This approach tends to produce better results at the final synthesis stage since it allows maximal sharing of the two expensive fixed-point units. The user is encouraged to take a similar approach rather than implementing a multiplication or division procedure every time it is required at the VHDL level. Note that the multiplier and divider are only an example and this note applies to any relatively expensive units that might be used more than once in a number of floating-point implementations.

Appendix E

Example details

This appendix provides additional information regarding the FPGA prototyping board and the cubic equation solver discussed in Chapter 6. It is organised in three sections: Section E.1 provides additional data for the FPGA prototyping board. Section E.2 provides additional information on the VGA display adapter used to drive the VGA screen in the cubic equation solver design. Finally, section E.3 contains VHDL source listings of the designs in Chapter 6.

E.1 FPGA prototyping board data

E.1.1 FPGA pin-out

The prototyping board was designed to support the Xilinx XC40125XVPG559, XC4085XVPG559, and XC40250XVPG559 FPGA. These are members of the Xilinx XC4000 series devices based on a programmable architecture of Configurable Logic Blocks (CLBs). Each device is programmed by loading the configuration data into internal memory cells. A top view of the FPGA pin-out is provided in Figure E.1. The XC40125XV for example, is based on a CLB array of 68 x 68 unit providing a total number of 4624 CLBs. It is claimed that the device is capable of implementing designs in the gate range 80,000 to 265,000 gates. The estimation is provided by Xilinx and is based on 20-30% of the CLBs used as RAMs. Further details on these devices can be found on [101].

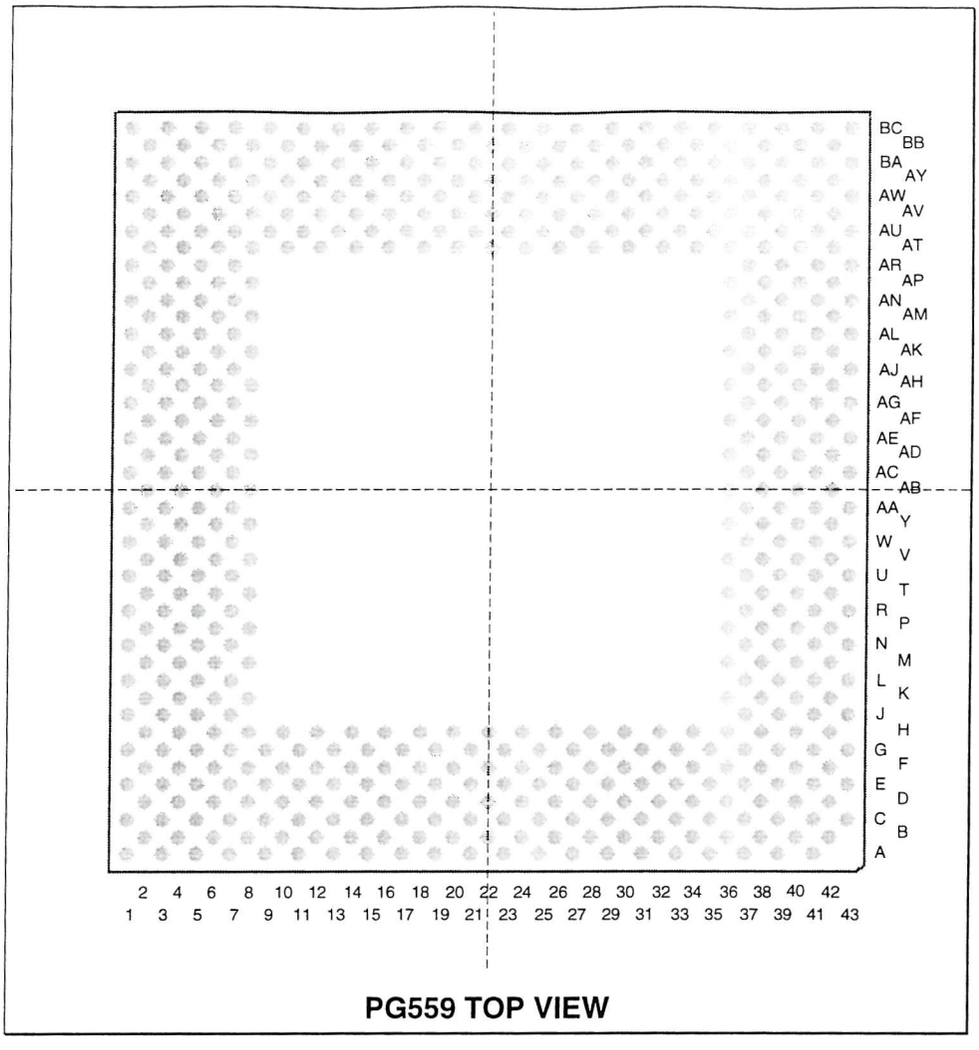


Figure E.1 FPGA package for the Xilinx FPGA used in the board

E.1.2 Device programming

Two methods may be employed to programme the device. Serial programming from a PC using a download cable or parallel programming based on an external ROM driven by the FPGA. Note that the device needs to be programmed whenever it is powered up. This suggests that the serial method may be used to programme the design during the implementation phase, while it is desirable to use the parallel mode for the final version of the design. A set of switches is provided on the board to enable one of these two modes. The default serial mode is active if the switches are off. Figure E.2 shows the PC cable connector provided. Details on the functionality of each pin on the connector are available in [101].

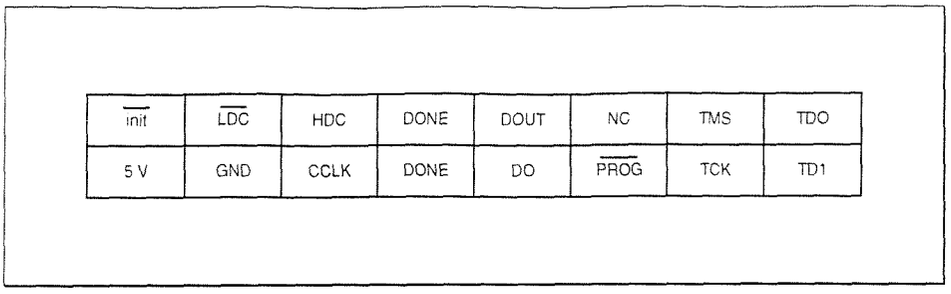
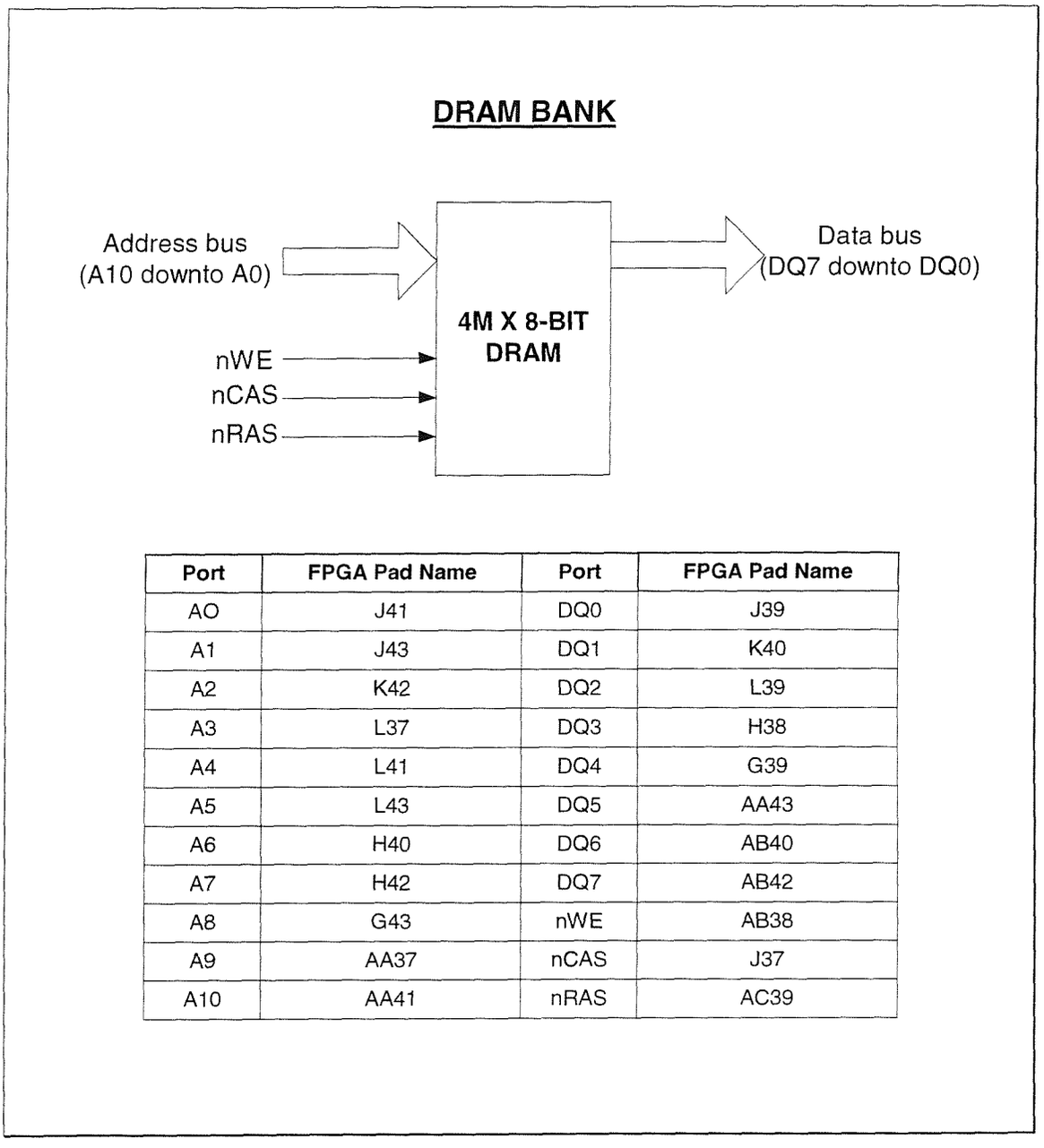


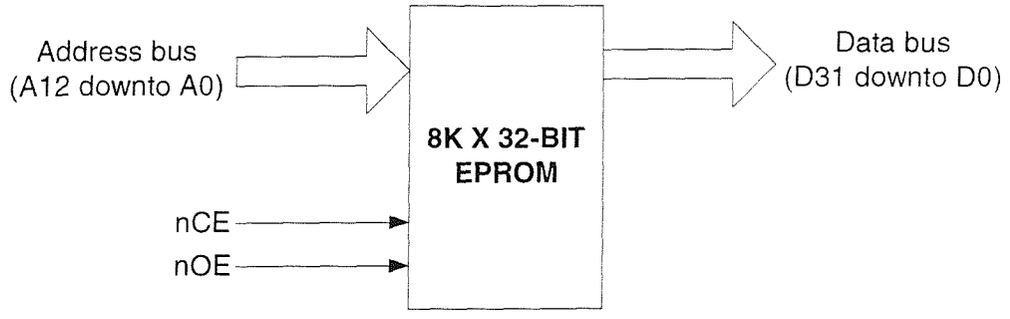
Figure E.2 Serial programming cable connector

E.1.3 Device pin-assignment



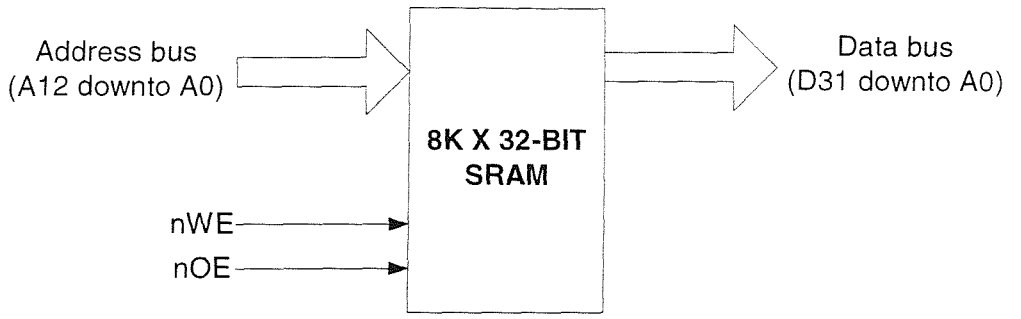
Port	FPGA Pad Name	Port	FPGA Pad Name
AO	J41	DQ0	J39
A1	J43	DQ1	K40
A2	K42	DQ2	L39
A3	L37	DQ3	H38
A4	L41	DQ4	G39
A5	L43	DQ5	AA43
A6	H40	DQ6	AB40
A7	H42	DQ7	AB42
A8	G43	nWE	AB38
A9	AA37	nCAS	J37
A10	AA41	nRAS	AC39

EPROM BANK



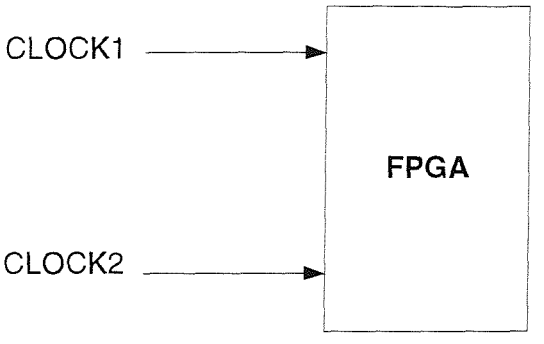
Port	FPGA Pad Name	Port	FPGA Pad Name
AO	C9	D20	B20
A1	C11	D19	B18
A2	C13	D18	B16
A3	C15	D17	B14
A4	C17	D16	B10
A5	C21	D15	B8
A6	C23	D14	B6
A7	C27	D13	B4
A8	C29	D12	A41
A9	C31	D11	A37
A10	C33	D10	A35
A11	C35	D9	A33
A12	C43	D8	A29
D31	C5	D7	A27
D30	B42	D6	A23
D29	B40	D5	A21
D28	B38	D4	A17
D27	B36	D3	A15
D26	B34	D2	A9
D25	B30	D1	A7
D24	B28	D0	A3
D23	B26	nCE	F36
D22	B24	nOE	G33
D21	B22		

SRAM BANK



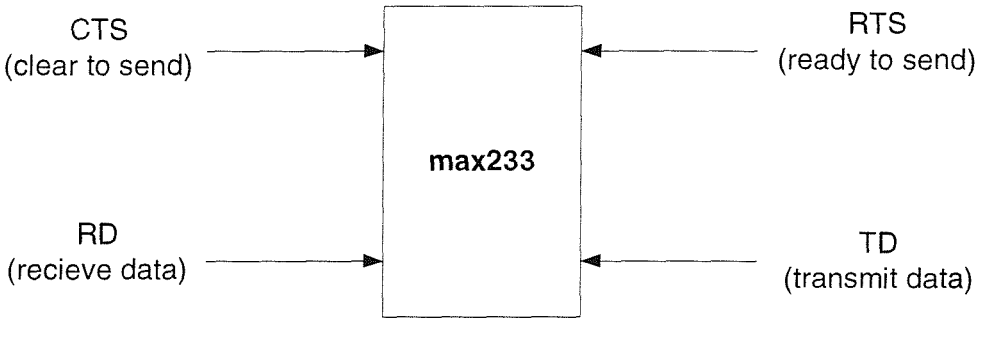
Port	FPGA Pad Name	Port	FPGA Pad Name
AO	C9	D20	B20
A1	C11	D19	B18
A2	C13	D18	B16
A3	C15	D17	B14
A4	C17	D16	B10
A5	C21	D15	B8
A6	C23	D14	B6
A7	C27	D13	B4
A8	C29	D12	A41
A9	C31	D11	A37
A10	C33	D10	A35
A11	C35	D9	A33
A12	C43	D8	A29
D31	C5	D7	A27
D30	B42	D6	A23
D29	B40	D5	A21
D28	B38	D4	A17
D27	B36	D3	A15
D26	B34	D2	A9
D25	B30	D1	A7
D24	B28	D0	A3
D23	B26	nWE	F40
D22	B24	nOE	F42
D21	B22		

CLOCK GENERATORS



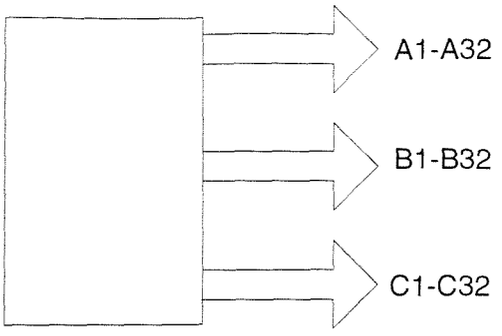
Port	FPGA Pad Name	Port	FPGA Pad Name
CLOCK1	F38	CLOCK2	E37

SERIAL PORT INTERFACE



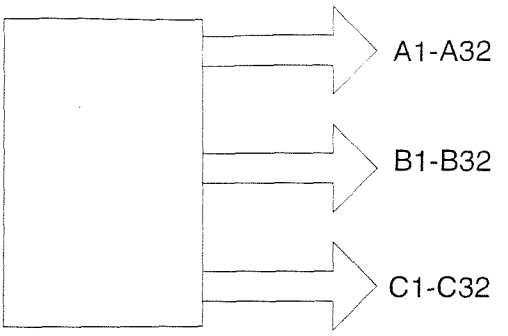
Port	FPGA Pad Name	Port	FPGA Pad Name
CTS	AP4	RTS	AT6
RD	AP2	TD	AP8

EXTERNAL PORT A

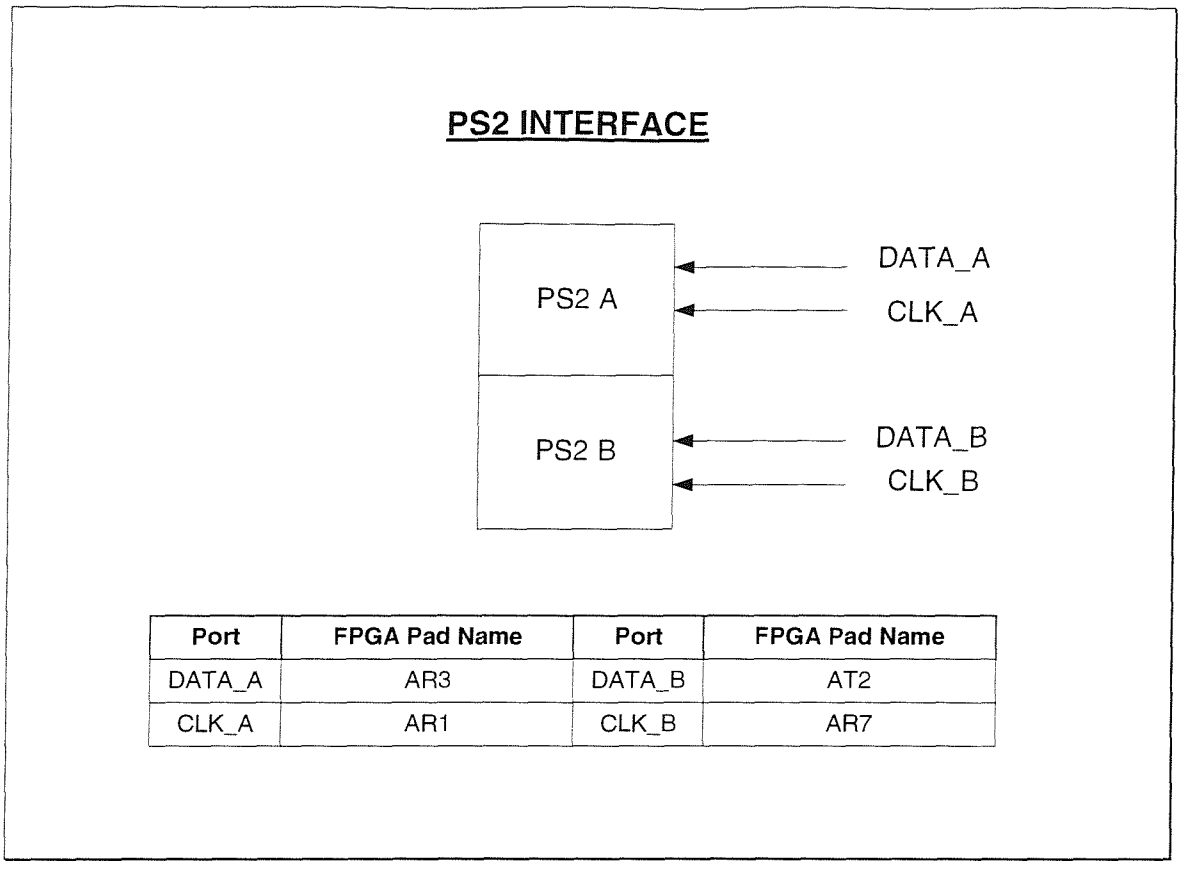


Port	FPGA Pad Name	Port	FPGA Pad Name	Port	FPGA Pad Name	Port	FPGA Pad Name
A1	A0	A25	D2	B17	TCK	C9	E11
A2	A1	A26	D3	B18	TMS	C10	E13
A3	A2	A27	D4	B19	nRS	C11	E15
A4	A3	A28	D5	B20	D2	C12	E17
A5	A4	A29	D6	B21	D6	C13	E19
A6	A5	A30	D7	B22	D8	C14	E21
A7	A6	A31	nPROG	B23	D10	C15	E23
A8	A7	A32	DONE	B24	D12	C16	E25
A9	A8	B1	M0	B25	D14	C17	E27
A10	A9	B2	M1	B26	D16	C18	E29
A11	A10	B3	M2	B27	D18	C19	E31
A12	A11	B4	DOUT	B28	D20	C20	E33
A13	A12	B5	nINIT	B29	D22	C21	E35
A14	A13	B6	nLDC	B30	D24	C22	E41
A15	A14	B7	HDC	B31	D26	C23	F2
A16	A15	B8	CCLK	B32	D28	C24	F6
A17	A16	B9	RDY	C1	D30	C25	F8
A18	A17	B10	nCS0	C2	D32	C26	F12
A19	A18	B11	GCK2	C3	D34	C27	F18
A20	A19	B12	GCK3	C4	D36	C28	F20
A21	A20	B13	GCK4	C5	D40	C29	F22
A22	A21	B14	GCK5	C6	D42	C30	F24
A23	D0	B15	TD0	C7	E7	C31	F26
A24	D1	B16	TD1	C8	E9	C32	F32

EXTERNAL PORT B



Port	FPGA Pad Name	Port	FPGA Pad Name	Port	FPGA Pad Name	Port	FPGA Pad Name
A1	BA39	A25	BC15	B17	AY30	C9	AU39
A2	BA41	A26	BC17	B18	AY32	C10	AU43
A3	BA43	A27	BC21	B19	AY34	C11	AV2
A4	BB2	A28	BC23	B20	AY36	C12	AV4
A5	BB6	A29	BC27	B21	AY38	C13	AV8
A6	BB8	A30	BC33	B22	AY40	C14	AV12
A7	BB10	A31	BC35	B23	BA11	C15	AV18
A8	BB14	A32	BC37	B24	BA13	C16	AV20
A9	BB16	B1	AW27	B25	BA15	C17	AV24
A10	BB18	B2	AW31	B26	BA17	C18	AV26
A11	BB20	B3	AW33	B27	BA21	C19	AV32
A12	BB22	B4	AW35	B28	BA27	C20	AV36
A13	BB24	B5	AW37	B29	BA29	C21	AV40
A14	BB26	B6	AY2	B30	BA31	C22	AV42
A15	BB28	B7	AY4	B31	BA33	C23	AW3
A16	BB30	B8	AY8	B32	BA35	C24	AW7
A17	BB34	B9	AY10	C1	GND	C25	AW11
A18	BB36	B10	AY12	C2	SUPPLY	C26	AW13
A19	BB38	B11	AY14	C3	AU23	C27	AW15
A20	BB40	B12	AY18	C4	AU25	C28	AW17
A21	BC3	B13	AY20	C5	AU27	C29	AW19
A22	BC7	B14	AY22	C6	AU29	C30	AW21
A23	BC9	B15	AY26	C7	AU31	C31	AW23
A24	BC11	B16	AY28	C8	AU33	C32	AW25



Port	FPGA Pad Name	Port	FPGA Pad Name
DATA_A	AR3	DATA_B	AT2
CLK_A	AR1	CLK_B	AR7

E.2 VGA adapter

The interface to the VGA adapter[1] is provided via an 8-bit input port and a 1-bit output ready signal. The input port is split into two fields: a 7-bit instruction occupying the bottom 7-bits of the port, and a single bit strobe signal. The VGA adapter drives a VGA display at a resolution of 640 x 480 pixels. This requires a 10-bit variable to identify the x location and a 9-bit variable to identify the y location. The VGA adapter instructions are listen in Table E.1.

The Set palette instruction allows the user to set the RGB ratios of 16 different colours. A unique 4-bit binary number allowing 16 different colours to be located will identify each colour, and each colour may be recalled by using the set colour instruction.

The Set point instruction sets the locations of one of two points p1 and p0. Both points should be located to allow drawing lines from p0 to p1. The two points also designate the

[1] The adapter is a contribution from a different research project within the same research group [113]

top left corner (p0) and the bottom right corner (p1) in the rectangle drawing mode. On the text drawing mode, only the point p0 is required to specify the top left corner of the ASCII character.

The Set mode instruction defines the VGA drawing mode. Four modes are available, designated by two bit binary variables:

1. Mode = 00 is a direct draw mode on both the foreground and the back ground (text drawing mode).
2. Mode = 01 is a direct drawing mode on the foreground.
3. Mode = 10 is an XOR drawing mode on both the foreground and the background.
4. Mode = 11 is an XOR drawing mode on the foreground.

Accessing the VGA adapter is a five-stage process:

1. Set the input port MSB to zero at the initialisation stage.
2. Set the port MSB to one along with the required VGA instruction.
3. Wait until an acknowledge is received (busy signal = 1).
4. Set the input port MSB to zero.
5. The instruction is now executed, any further commands are performed by looping back to stage two.

An example representing the functionality of the VGA adapter is represented in Figure E.3. It shows a sequence of commands along with the expected output.

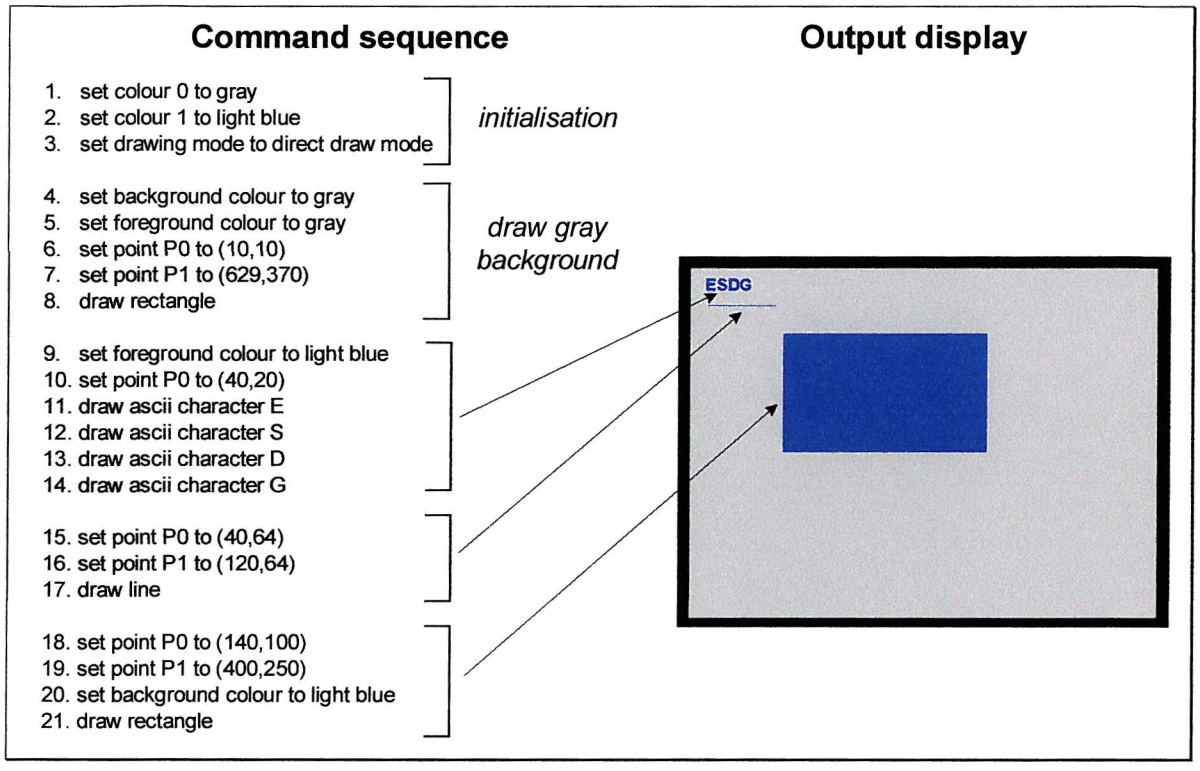


Figure E.3 VGA adapter example

Instruction	Instruction length	Detailed bit field
Set point [p1, p0, x(9:0), y(8:0)]	4	"0000X",p1,p0 "XX", x(9:5) x(4:0), y(8:7) y(6:0)
Set page [front, page(1:0)]	1	"0001", front, page(1:0)
Set made [mode(1:0)]	1	"0010X", mode(1:0)
Set palette [colour(3:0),R(3:0),G(3:0),B(3:0)]	3	"0011", colour(3:1) colour(0), "X", R(3:0), G(3) G(2:0), B(3:0)
Set colour [foreground, colour(3:0)]	1	"01", foreground, colour(3:0)
Draw line	1	"1001XXX"
Draw rectangle	1	"1010XXX"
Wait for vertical blanking	1	"1011XXX"
Draw character [xsize(1:0), ysize(1:0), ASCII(7:0)]	2	"11", xsize(1:0),ysize(1:0),ASCII(7) ASCII(6:0)

Table E.1 VGA adapter instruction set

E.3 IO stage details

E.3.1 Input stage

Before examining the operation of the keyboard interface unit, first consider Figure E.4 which represents the keyboard sequential data along with what is called the *scancode* of the keys in the numerical keypad. Every time a key is pressed, the keyboard generates a scancode. Each key has a unique scancode consisting of one or more 8-bit words. The scancodes related to each key in the numerical keypad are represented in Figure E.4c in hexadecimal. When the key is released, the keyboard regenerates the scancode preceded by hex F0. For example the scancodes generated when (num lock) key is pressed and released are 45 F0 45.

The generated scancode is provided as serial data on the keyboard data line, synchronised by a clock signal provided on the keyboard clock line with a new bit outputted every falling edge on the clock line. Note that the keyboard outputs groups of 9-bit data: a start bit indicating the beginning of a new word precedes the 8-bit word.

The flowchart in Figure E.5 illustrates the keyboard interface process. The process waits to detect a falling edge on the keyboard_clock line, and once detected, the data on the keyboard_data line is latched into an internal register. The loop iterates nine times until the whole 8-bit word is detected (the start bit is ignored). The next step involves decoding the scancode to identify the pressed key. This stage involves the following operations:

1. If a F0 code is detected, the following scancode is ignored, since this would be a release code.

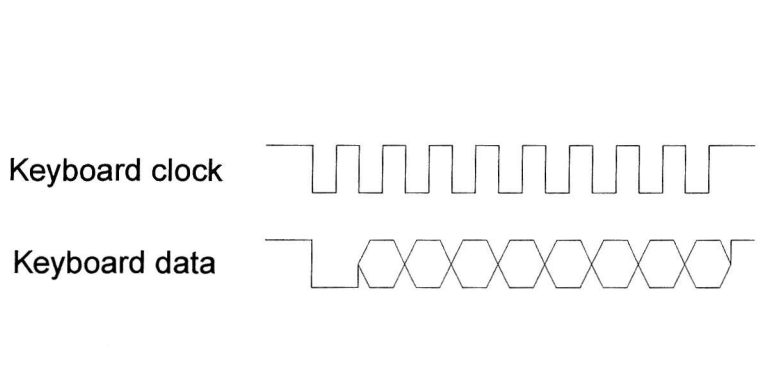
2. If an E0 is detected, another word is read before decoding, as E0 indicates an extended word.

3. If the scancode represent a key within the recognised set (shaded in Figure E.4b) decode it.

4. If the pressed key is (numlock), toggle the initialise line low and pass it to the core unit and the output stage to initialise the system.

5. The divide, add, and multiply keys in the keypad are ignored.

6. The minus key is used to invert the sign of the current parameter. The input number is assumed to be positive. Every time the minus key is pressed the number sign is inverted.



a) Keyboard serial outputs

num lock	/	*	-
7	8	9	+
4	5	6	
1	2	3	Enter
0		.	

b) Numerical keypad

Key	Code	Key	Code	Key	Code
num lock	45	9	49	2	50
/	E035	+	4E	3	51
*	37	4	4B	0	52
-	4A	5	4C	.	53
7	47	6	4D	Enter	E01C
8	48	1	4F		

c)Key scan codes

Figure E.4 Keyboard Information

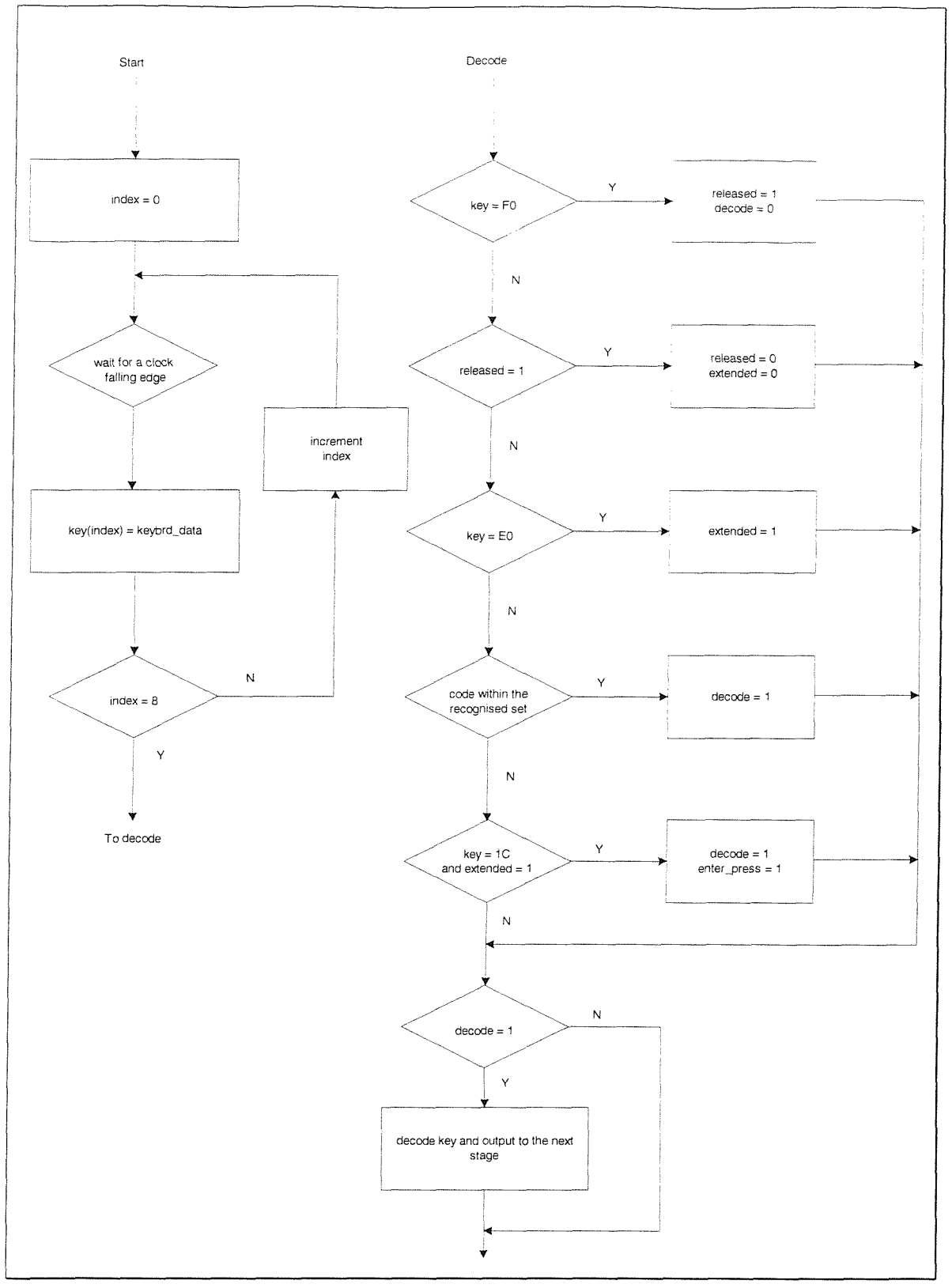


Figure E.5 Keyboard interface flowchart

Each numerical parameter is expected to be input as a set of decimal values followed by an (enter). Every time a related key is pressed, the decoded key is passed to the output

stage to be displayed and also to the format conversion stage. The format conversion stage converts a set of binary coded decimal values in to a binary single precision floating point number and passes it to the core unit.

The functionality of the format conversion is illustrated by the flowchart in Figure E.6. It consists of two main blocks: the first block generates a binary representation of the integer part of the input operand, the second generates the fraction part. At each step, two operations are performed:

1. Multiply the integer accumulator by 10_{10} (1010_2).

2. Add the input value to the accumulator.

To illustrate the functionality of this block a simple example is provided where the sequence 2, 5, 6 is provided indicating a decimal value of 256. The sequence of execution is:

$$
\begin{aligned}
&acc = 0 \\
&acc = acc \times 1010 = 0 \\
&acc = acc + 0010 = 0010 \\
&acc = acc \times 1010 = 10100 \\
&acc = acc + 0101 = 11001 \\
&acc = acc \times 1010 = 111111010 \\
&acc = acc + 0110 = 100000000_2 = 256_{10}
\end{aligned}
$$

Note that the internal register that holds the integer part of the input parameter is a 63-bit register allowing a maximum entry of (±9223372036854775808) for the integer part. The execution continues in the first block until the maximum number of digits is reached or the decimal point is encountered or the (enter) key is pressed.

Once the decimal point is encountered, execution moves to the second block, which is responsible for generating the fraction of the input parameter. At this stage, the digits to the right of the decimal points are pushed into a stack until the (enter) key is pressed or the maximum number of digits is received (seven digits in this case). Once the fraction digit accumulation is completed the conversion operation starts. The operation involves the three following steps:

1. Divide the fraction accumulator by 10.

2. Divide the input digit by 10.

3. Add the results in 1 and 2 and save it in the fraction accumulator.

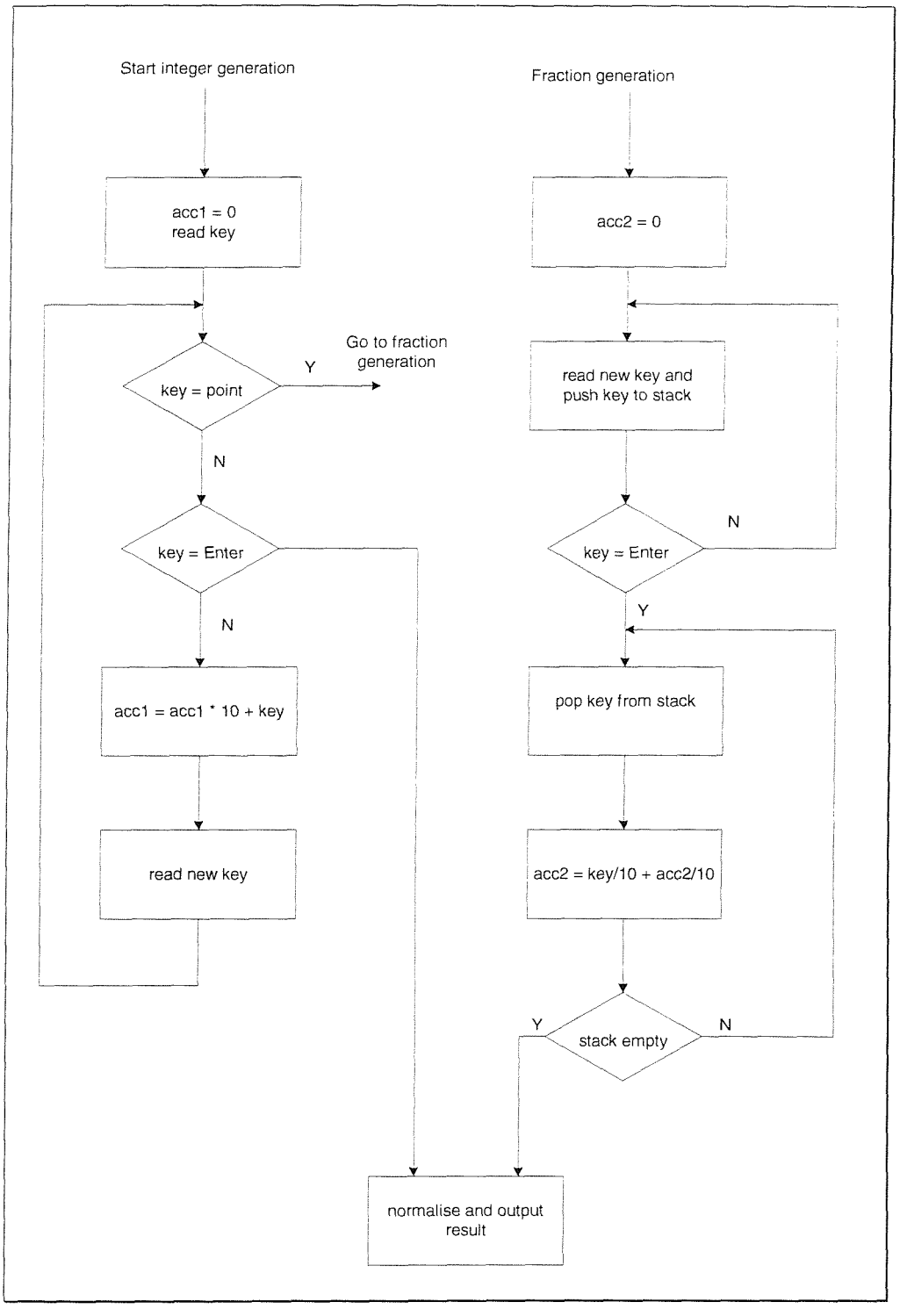


Figure E.6 Format conversion unit flowchart

Once the second stage is concluded, two internal variables will hold the integer and fraction part of the input parameter in a binary format. These two numbers are then treated as a single fixed point variable which is normalised to fit into the output format and the output sign and exponent value are assigned.

E.3.2 Output stage

The unit splits into two blocks executing before and after the roots calculation in the core unit.

The first block performs two main duties. It is responsible for creating the static elements of the VGA display (e.g. title, background, variable names). It also monitors the data input stage to display the decimal values of the input parameters.

The second block, monitors the core unit for the root values and displays them on the VGA screen. This stage involves a simple type conversion to convert the binary representation of the floating-point number to the displayed representation.

The VGA display adapter[2] [113] that drives the VGA screen interfaces to the system via an 8-bit command port and a 1-bit busy signal. A low busy signal indicates that the adapter is ready to receive a new instruction. Each instruction is 7-bits long. A new instruction is latched into the VGA adapter by loading the instruction to the lower seven bits of the input port and setting the most significant bit. The adapter provides a set of basic instructions that supports writing to the VGA screen. The instructions are set point, set page, set mode, set palette, set colour, draw rectangle, draw line, and draw text.

A simple technique is adopted to create the display of the static elements on the screen. The required set of instructions is developed and stored in internal ROMs. A loop is then provided to iterate through these ROMs and output the VGA commands to the adapter. Two internal ROMs are provided. The first is a 47 x 7-bit ROM provided to store the initialisation commands such as setting the colour palette, setting the drawing mode, and drawing the background and the title underline. The second ROM is 84 x 7-bit responsible for drawing the static characters on the screen (the title, the inputs and the output names).

---

[2] The adapter was synthesised using the MOODS synthesis system and implemented on an FPGA.

Once the screen is initialised, the output stage starts monitoring the input keys and displaying them on the appropriate location on the screen. Upon receiving the third parameter, the output stage starts monitoring the core unit to receive the output results and display them on the screen.

To perform the last step and display the output result, the output stage needs to convert the binary representation of the floating-point number into another representation that can be read easily. A number of possible methods can be used to print the floating-point numbers [102]. However, a fairly simple approach is taken due to the limited hardware resources available, illustrated in the flow chart of Figure E.7.

The conversion operation starts by detecting any possible symbolic representations such as NAN or infinity and displaying the equivalent ASCII representation. If none of these symbols are detected, execution moves to the second stage. The second stage starts by displaying the result sign. The following step displays the fraction field, starting by displaying the implicit one and the decimal point. Then the decimal digits of the fraction are displayed sequentially where at each step the fraction is multiplied by 10 and the integer part of the result is displayed until the fraction equals zero.

The final step in the conversion operation displays the exponent. After removing the bias, the actual exponent passes through five stages:

1. If the exponent is less than zero, a negative sign is displayed and the exponent is complemented.

2. If the exponent is greater than or equal to 100, a one is displayed and 100 is subtracted from the exponent, a flag (flag1) is set at this stage to indicate that the exponent is $\geq$ 100.

3. The third stage involves counting the number of tens contained within the exponent and displaying it as a decimal number, a flag (flag2) is set here to indicate that the remaining exponent is $\geq$ 10.

4. A special case when (flag1 = 1 and flag2 = 0) is detected here and a zero is displayed before displaying the last digit.

5. At this stage, the exponent will have a value between 0 and 9, which is displayed directly.

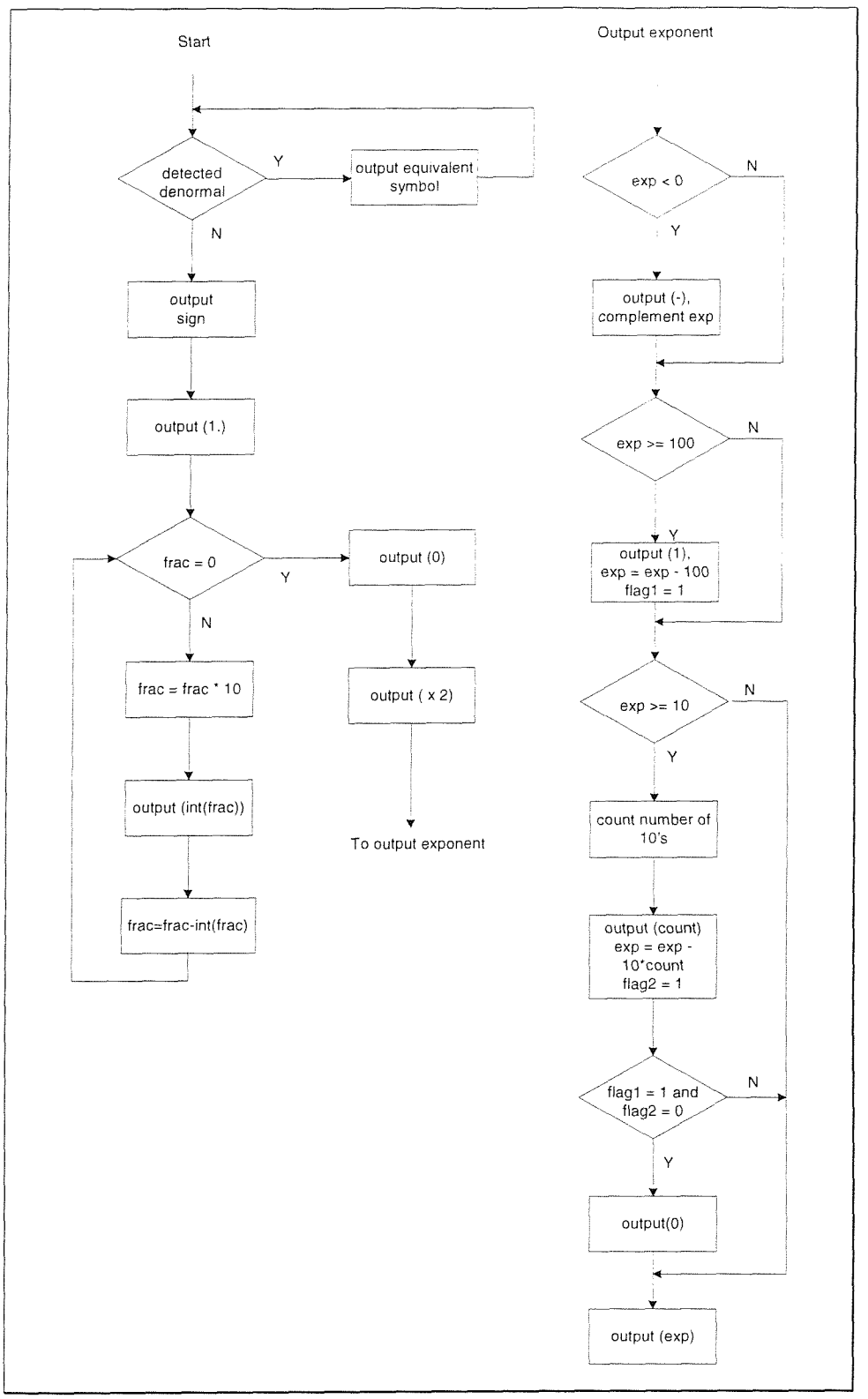


Figure E.7 Output stage type conversion flowchart

E.4 Source code listings

Listing E.1 Input stage VHDL behavioural description

```
1  ---------------------------------------------------------------------------
2  -- The main input stage. Provides an interface to the keyboard unit and transfer
3  -- the input data to the core unit and the output stage. It also provide a
4  -- system reset entry to reset the whole system via the new_entry output
5  ---------------------------------------------------------------------------
6
7   package InputConst is
8    -- scancodes for various keys
9    constant rel_code   : bit_vector (7 downto 0) := "11110000";
10   constant ext_code   : bit_vector (7 downto 0) := "11100000";
11   constant num_code   : bit_vector (7 downto 0) := "01110111";
12   constant minus_code : bit_vector (7 downto 0) := "01111011";
13   constant point_code : bit_vector (7 downto 0) := "01110001";
14   constant enter_code : bit_vector (7 downto 0) := "01011010";
15   constant zero_code  : bit_vector (7 downto 0) := "01110000";
16   constant one_code   : bit_vector (7 downto 0) := "01101001";
17   constant two_code   : bit_vector (7 downto 0) := "01110010";
18   constant three_code : bit_vector (7 downto 0) := "01111010";
19   constant four_code  : bit_vector (7 downto 0) := "01101011";
20   constant five_code  : bit_vector (7 downto 0) := "01110011";
21   constant six_code   : bit_vector (7 downto 0) := "01110100";
22   constant seven_code : bit_vector (7 downto 0) := "01101100";
23   constant eight_code : bit_vector (7 downto 0) := "01110101";
24   constant nine_code  : bit_vector (7 downto 0) := "01111101";
25   -- internal representation of keys
26   constant num_val    : bit_vector (7 downto 0) := "01010";
27   constant minus_val  : bit_vector (7 downto 0) := "01101";
28   constant point_val  : bit_vector (7 downto 0) := "10000";
29   constant enter_val  : bit_vector (7 downto 0) := "01111";
30   constant zero_val   : bit_vector (7 downto 0) := "00000";
31   constant one_val    : bit_vector (7 downto 0) := "00001";
32   constant two_val    : bit_vector (7 downto 0) := "00010";
33   constant three_val  : bit_vector (7 downto 0) := "00011";
34   constant four_val   : bit_vector (7 downto 0) := "00100";
35   constant five_val   : bit_vector (7 downto 0) := "00101";
36   constant six_val    : bit_vector (7 downto 0) := "00110";
37   constant seven_val  : bit_vector (7 downto 0) := "00111";
38   constant eight_val  : bit_vector (7 downto 0) := "01000";
39   constant nine_val   : bit_vector (7 downto 0) := "01001";
40  end InputConst;
41
42  use work.InputConst.all;
43  entity in_stage is
44   port (key_clk,key_data : in bit;
45     float_output      : out bit_vector(31 downto 0);
46     key_out           : out bit_vector(4 downto 0);
47     stb_core          : out bit;
48     ack_core          : in bit;
49     stb_out           : out bit;
50     ack_out           : in bit;
51     new_entry         : out bit
52     );
53  end;
54
55  architecture behave of in_stage is
56   ---------------------------------------------------------------------------
57   -- an array is declared to act as a stack for the fraction digits
58   ---------------------------------------------------------------------------
59   type in_array is array(0 to 6) of bit_vector(3 downto 0);
60   begin
61   process
62    -- a counter for the number of serial bits received form the keyboard
63    variable bit_count : bit_vector(3 downto 0);
64    -- int_part holds the integer value of the input
65    variable int_part : bit_vector(62 downto 0);
66    -- frac_part holds the fraction value of the input
67    variable frac_part : bit_vector(23 downto 0);
```

```
68      variable extended , released : bit;
69      variable decode  : bit;
70      variable key_val : bit_vector(4 downto 0);
71      variable done_press,new_press,enter_press,minus_press : bit;
72      -- a flag that indicates the decimal point press while monitoring
73      -- the integer part we are receiving the integer part
74      variable frac : bit;
75      variable key_word : bit_vector(7 downto 0);
76      variable frac_count : integer range 0 to 7;
77      variable int_count  : integer range 0 to 19;
78      -- temporary variables
79      variable div_result1,div_result2 : bit_vector (31 downto 0);
80      -- the stack that hold the fraction digits
81      variable frac_inputs : in_array;
82      begin
83      -- initialise all the control and the handshaking signals
84      -- along with the accumulators
85      new_entry <= '1';
86      stb_core <= '1';
87      stb_out <= '1';
88      key_out <= "00000";
89      frac_count := 0;
90      int_count := 0;
91      frac := '0';
92      new_press := '0';
93      enter_press := '0';
94      minus_press := '0';
95      float_output(31) <= '0';
96      decode := '0';
97      extended := '0';
98      released  := '0';
99      bit_count  := "1111";
100     int_part := convert_int2bv(0,63);
101     frac_part := convert_int2bv(0,24);
102     wait for 0 ns;
103     ------------------------------------------------------------------------
104     -- The main loop that reads the keyboard entries and converts them to
105     -- a floating-point number.
106     ------------------------------------------------------------------------
107     loop
108      ---------------------------------------------------------------------
109      -- The first loop reads the keyboard serial data and converts
110      -- it to a single word
111      ---------------------------------------------------------------------
112      loop
113       -- wait for the keyboard clock to go low
114       wait until key_clk = 1;
115       wait until key_clk = 0;
116       -- enter the bit into the key_word
117       if bit_count(3) = '0' then
118        key_word := "0" & key_word(7 downto 1) ;
119        key_word(7) := key_data;
120       end if;
121       -- exit the loop
122       exit when bit_count = "1001";
123       -- next bit
124       bit_count := bit_count + "0001";
125      end loop;
126      -- reset the bit_count to its starting position
127      -- set the initial bit count to 15 (this is so the start bit is ignored)
128      bit_count := "1111";
129      if key_word = rel_code then
130       -- ignore the next word sent
131       decode := '0';
132       released := '1';
133      elsif released = '1' then
134       -- the last character was a release code
135       -- ignore the present code and reset the released flag
136       released := '0';
```

```
137         -- also reset the extended flag for release of extended keys
138         extended := '0';
139        elsif key_word = ext_code then
140         -- check the extended bit
141         extended := '1';
142        elsif key_word = num_code then
143         decode := '1';
144         key_val := num_val;
145         new_press := '1';
146        elsif key_word = minus_code then
147         decode := '1';
148         minus_press := '1';
149         float_output(31) <= NOT float_output(31);
150         key_val := minus_val;
151        elsif key_word = point_code then
152         decode := '1';
153         frac := '1';
154         key_val := point_val;
155        elsif key_word = zero_code  then
156         decode := '1';
157         key_val := zero_val;
158        elsif key_word = one_code then
159         decode := '1';
160         key_val := one_val;
161        elsif key_word = two_code then
162         decode := '1';
163         key_val := two_val;
164        elsif key_word = three_code then
165         decode := '1';
166         key_val := three_val;
167        elsif key_word = four_code then
168         decode := '1';
169         key_val := four_val;
170        elsif key_word = five_code then
171         decode := '1';
172         key_val := five_val;
173        elsif key_word = six_code then
174         decode := '1';
175         key_val := six_val;
176        elsif key_word = seven_code then
177         decode := '1';
178         key_val := seven_val;
179        elsif key_word = eight_code then
180         decode := '1';
181         key_val := eight_val;
182        elsif key_word = nine_code then
183         decode := '1';
184         key_val := nine_val;
185        elsif key_word = enter_code and extended = '1' then
186         decode := '1';
187         key_val := enter_val;
188         enter_press := '1';
189       end if;
190       -- a key entry part of the numerical pad is received if decode = 1
191       if decode = '1' then
192        decode := '0';
193        -- output stage is ready to receive an entry
194        wait until ack_out = '1';
195        if (enter_press = '1' or new_press = '1' or minus_press = '1') then
196         new_entry <= not new_press;
197         key_out <= key_val;
198         stb_out <= '0';
199         wait for 0 ns;
200         -- check for the decimal point
201        elsif key_val /= point_val then
202         key_out <= key_val;
203         stb_out <= '0';
204         wait for 0 ns;
205
```

```
206          ------------------------------------------------------------------
207          -- decimal to float for the integer part is performed here
208          -- it involves multiplying the accumulator by 10 and adding the
209          -- keyboard value to it. Note that the multiply by 1010 is achieved
210          -- by a simple shift and add operation
211          ------------------------------------------------------------------
212          if (frac = '0' and int_count /= 19) then-- still in the integer part
213           int_count := int_count + 1;
214           -- multiply by "1010"
215           int_part:= int_part(61 downto 0)&"0" + int_part(59 downto 0)&"000";
216           -- add the input value
217           int_part := int_part + key_val;
218          else
219           -- if receiving the fraction digits just push them in the stack
220           frac_inputs(frac_count) := key_val (3 downto 0);
221           frac_count := frac_count + 1;
222          end if;
223         else
224          key_out <= key_val;
225          wait for 0 ns;
226          stb_out <= '0';
227         end if;
228         wait until ack_out = '0';
229         stb_out <= '1';
230         wait for 0 ns;
231         minus_press := '0';
232         if (enter_press = '1') then -- output the float_output to the core
233          -- first generate the number and normalise it
234          float_output (30 downto 23) <= "01111111"; --initialise the exponent
235         wait for 0 ns;
236          if (frac = '1') then
237           frac_count := frac_count - 1;
238           loop
239            -- generate the binary equivalent of the fraction digits
240            fixed_div ("0" & frac_inputs (frac_count) & convert_int2bv(0,27),
241            "01010000000000000000000000000000",div_result1);
242            fixed_div ("00000" & frac_part & "000",
243            "01010000000000000000000000000000",div_result2);
244            frac_part(23 downto 0) := div_result1 (26 downto 3)
245            + div_result2(26 downto 3);
246            wait for 0 ns;
247            Exit when frac_count = 0;
248            frac_count := frac_count - 1;
249           end loop;
250         end if;
251         -- normalise the integer part and adjust the exponent
252         if (int_part /= convert_int2bv(0,63)) then
253          loop
254           exit when int_part(62 downto 1) = convert_int2bv(0,62);
255           frac_part:= int_part(0) & frac_part(23 downto 1);
256           int_part := "0" & int_part (62 downto 1);
257           float_output(30 downto 23) <= float_output (30 downto 23) + "1";
258           wait for 0 ns;
259          end loop;
260          -- then number is less than one
261         elsif (frac_part /= convert_int2bv(0,24)) then
262          loop
263           exit when int_part(0) = "1";
264           int_part(0) := frac_part(23);
265           frac_part := frac_part(22 downto 0) & "0";
266           float_output(30 downto 23) <= float_output (30 downto 23) - "1";
267           wait for 0 ns;
268          end loop;
269         else -- the entry is zero
270          float_output(31 downto 23) <= convert_int2bv(0,9);
271          wait for 0 ns;
272         end if;
273         float_output (22 downto 0) <= frac_part (23 downto 1);
274         -- output the floating-point entry to the core
```

```
275          wait until ack_core = '1';
276          stb_core <= '0';
277          wait until ack_core = '0';
278          stb_core <= '1';
279          Exit;
280        elsif new_press = '1' then
281          new_entry <= '0';
282          wait until ack_core = '1';
283          stb_core <= '0';
284          wait until ack_core = '0';
285          stb_core <= '1';
286          Exit;
287        end if;
288        enter_press := '0';
289        new_press := '0';
290        new_entry <= '1';
291      end if;
292    end loop;
293  end process;
294 end behave;
```

Listing E.2 Original design VHDL behavioural description

```
1  ------------------------------------------------------------------------------
2  -- Floating-point Cubic equation solver core.
3  -- All the floating-point operations are performed within the core. This is
4  -- a direct translation of the mathematical equations
5  ------------------------------------------------------------------------------
6  package CoreConst is
7   constant con1 : real := 0.866025404; -- sqrt(3)/2
8   constant con2 : real := 2.094395102; -- 2Pi/3
9   constant con3 : real := 4.188790204; -- 4Pi/3
10 end;
11
12 use work.CoreConst.all;
13 entity core is
14  port (
15   input     : in float;
16   stb_in    : in bit;
17   ack_in    : out bit;
18   new_entry : in bit;
19   stb_out   : out bit;
20   ack_out   : in bit;
21   data_out  : out float
22   );
23 end;
24 architecture behave of core is
25  begin
26  process
27   variable a1,a2,a3,S,T : float;
28   variable R,Q,R_sq,Q_cu,D, sqrt_D : float;
29   variable X1 : float;
30   variable Temp1,Temp2,theta3 : float;
31   variable X2,X3 : cmplx;
32   ------------------------------------------------------------------------------
33   -- a  procedure to read the three input parameters and store them in a1,a2,a3
34   ------------------------------------------------------------------------------
35   procedure get_input_data  is
36    begin
37    wait until stb_in = '0';
38    a1 := input;
39    ack_in <= '0';
40    wait until stb_in = '1';
41    ack_in <= '1';
42    wait until stb_in = '0';
43    a2 := input;
44    ack_in <= '0';
45    wait until stb_in = '1';
46    ack_in <= '1';
47    wait until stb_in = '0';
48    a3 := input;
49    ack_in <= '0';
50    wait until stb_in = '1';
51    ack_in <= '1';
52   end get_input_data;
53
54   ------------------------------------------------------------------------------
55   -- a  procedure to deliver results to the output stage
56   ------------------------------------------------------------------------------
57   procedure send_output_result  is
58    begin
59    data_out <= X1;
60    wait until ack_out = '1';
61    stb_out <= '0';
62    wait until ack_out = '0';
63    stb_out <= '1';
64    data_out <= RE(X2);
65    wait until ack_out = '1';
```

```
66        stb_out <= '0';
67        wait until ack_out = '0';
68        stb_out <= '1';
69        data_out <= IMAG(X2);
70        wait until ack_out = '1';
71        stb_out <= '0';
72        wait until ack_out = '0';
73        stb_out <= '1';
74        data_out <= RE(X3);
75        wait until ack_out = '1';
76        stb_out <= '0';
77        wait until ack_out = '0';
78        stb_out <= '1';
79        data_out <= IMAG(X3);
80        wait until ack_out = '1';
81        stb_out <= '0';
82        wait until ack_out = '0';
83        stb_out <= '1';
84      end send_output_result;
85      -- core process see Figure 6.5
86      begin
87      get_input_data;
88      Q := ((TO_FLOAT(3.0)*a2)-(a1*a1))/TO_FLOAT(9.0);
89      R := ((TO_FLOAT(9.0)*a1*a2)-(TO_FLOAT(27.0)*a3)
90           -(TO_FLOAT(2.0)*a1*a1 *a1))/TO_FLOAT(54.0);
91      R_sq := R * R;
92      Q_cu := Q * Q * Q;
93      D := R_sq + Q_cu;
94      if (D = TO_FLOAT(0.0)) then
95        S := CBRT(R);
96        Temp1 := a1/TO_FLOAT(3.0);
97        X1 := TO_FLOAT(2.0)*S-Temp1;
98        X2 := TO_COMPLEX(-S-Temp1,TO_FLOAT(0.0));
99        X3 := X2;
100     elsif (D > TO_FLOAT(0.0)) then
101       sqrt_D := SQRT(D);
102       S := CBRT(R+sqrt_D);
103       T := CBRT(R-sqrt_D);
104       Temp1 := S+T;
105       Temp2 := a1/TO_FLOAT(3.0);
106       X1 := Temp1-Temp2;
107       X2 := TO_COMPLEX((-Temp1/TO_FLOAT(2.0))-Temp2,(S-T)*TO_FLOAT(con1));
108       X3 := CONJ(X2);
109     else
110       theta3 := ACOS(R/SQRT(-Q_cu))/TO_FLOAT(3.0);
111       Temp1 := a1/TO_FLOAT(3.0);
112       Temp2 := TO_FLOAT(2.0)*SQRT(-Q);
113       X1 := Temp2*COS(theta3)-Temp1;
114       X2 := TO_COMPLEX(Temp2*COS(theta3+TO_FLOAT(con2))-Temp1,TO_FLOAT(0.0));
115       X3 := TO_COMPLEX(Temp2*COS(theta3+TO_FLOAT(con3))-Temp1,TO_FLOAT(0.0));
116     end if;
117     send_output_result;
118   end process;
119 end;
```

Listing E.3 controller VHDL behavioural description

```
1  ------------------------------------------------------------------------
2  -- The unit acts as a master in a master slave combination that generates the
3  -- three roots of the cubic equation. The unit uses the arithmetic processor as
4  -- to generate a number of functions as well as floating-point
5  -- multiplication. The control over the arithmetic processor  is provided via a
6  -- 3-bit control vector and two handshaking signals (stb_c2,ack_c2)
7  -- the control signal is defined as follows:
8  -- control          Reaction
9  -- -------          --------
10 --  000               Multiply two operands
11 --  001               Square a single operand
12 --  010               multiply three operands
13 --  011               multiply four operands
14 --  100               Square root
15 --  101               Cubic root
16 --  110               cosine function
17 --  111               inverse cosine function
18 ------------------------------------------------------------------------
19 package UnitConst is
20
21   constant mult2_op  : bit_vector (2 downto 0) := "000";
22   constant square_op : bit_vector (2 downto 0) := "001";
23   constant mult3_op  : bit_vector (2 downto 0) := "010";
24   constant mult4_op  : bit_vector (2 downto 0) := "011";
25   constant sqrt_op   : bit_vector (2 downto 0) := "100";
26   constant cbrt_op   : bit_vector (2 downto 0) := "101";
27   constant cos_op    : bit_vector (2 downto 0) := "110";
28   constant acos_op   : bit_vector (2 downto 0) := "111";
29
30   constant con1 : real := 0.866025404; -- sqrt(3)/2
31   constant con2 : real := 2.094395102; -- 2Pi/3
32   constant con3 : real := 4.188790204; -- 4Pi/3
33 end;
34
35 use work.UnitConst.all;
36 entity controller is
37  port (
38   input      : in float;
39   stb_in     : in bit;
40   ack_in     : out bit;
41   new_entry  : in bit;
42   ack_c2     : in bit;
43   stb_c2     : out bit;
44   c2_data    : out bit_vector (7 downto 0);
45   c2_result  : in bit_vector (7 downto 0);
46   control    : out bit_vector (3 downto 0);
47   stb_out    : out bit;
48   ack_out    : in bit;
49   data_out   : out float
50   );
51 end;
52 architecture behave of controller is
53  begin
54  process
55   -- a number of floating-point variables to hold intermediate results
56   variable a1,a2,a3,S,T : float;
57   variable R,Q,R_sq,Q_cu,D, sqrt_D : float;
58   variable X1 : float;
59   variable Temp1,Temp2,Temp3,Temp4,theta3,core2_result : float;
60   variable X2,X3 : cmplx;
61   ------------------------------------------------------------------------
62   -- a  procedure to read the three input parameters and store them in a1,a2,a3
63   ------------------------------------------------------------------------
64
65   procedure get_input_data  is
66    begin
67    wait until stb_in = '0';
```

```
68        a1 := input;
69        ack_in <= '0';
70        wait until stb_in = '1';
71        ack_in <= '1';
72        wait until stb_in = '0';
73        a2 := input;
74        ack_in <= '0';
75        wait until stb_in = '1';
76        ack_in <= '1';
77        wait until stb_in = '0';
78        a3 := input;
79        ack_in <= '0';
80        wait until stb_in = '1';
81        ack_in <= '1';
82      end get_input_data;
83
84      ------------------------------------------------------------------------
85      -- a  procedure to deliver results to the output stage
86      ------------------------------------------------------------------------
87      procedure send_output_result  is
88        begin
89        data_out <= X1;
90        wait until ack_out = '1';
91        stb_out <= '0';
92        wait until ack_out = '0';
93        stb_out <= '1';
94        data_out <= RE(X2);
95        wait until ack_out = '1';
96        stb_out <= '0';
97        wait until ack_out = '0';
98        stb_out <= '1';
99        data_out <= IMAG(X2);
100       wait until ack_out = '1';
101       stb_out <= '0';
102       wait until ack_out = '0';
103       stb_out <= '1';
104       data_out <= RE(X3);
105       wait until ack_out = '1';
106       stb_out <= '0';
107       wait until ack_out = '0';
108       stb_out <= '1';
109       data_out <= IMAG(X3);
110       wait until ack_out = '1';
111       stb_out <= '0';
112       wait until ack_out = '0';
113       stb_out <= '1';
114     end send_output_result;
115     ------------------------------------------------------------------------
116     -- The procedure sends a floating-point variable to the slave unit over four
117     -- iterations. It provides the strobe signal and monitors the acknowledge
118     ------------------------------------------------------------------------
119     procedure send_to_core2 ( data : bit_vector (31 downto 0)) is
120       begin
121       wait until ack_c2 = '1';
122       c2_data <= data (31 downto 24);
123       stb_c2 <= '0';
124       wait until ack_c2 = '0';
125       stb_c2 <= '1';
126       wait until ack_c2 = '1';
127       c2_data <= data (23 downto 16);
128       stb_c2 <= '0';
129       wait until ack_c2 = '0';
130       stb_c2 <= '1';
131       wait until ack_c2 = '1';
132       c2_data <= data (15 downto 8);
133       stb_c2 <= '0';
134       wait until ack_c2 = '0';
135       stb_c2 <= '1';
136       wait until ack_c2 = '1';
```

```
137       c2_data <= data (7 downto 0);
138       stb_c2 <= '0';
139       wait until ack_c2 = '0';
140       stb_c2 <= '1';
141      end send_to_core2;
142
143      ------------------------------------------------------------------------
144      -- The procedure receives the floating-point result of a certain operation
145      -- from the arithmetic processor. It is based on monitoring a transition on
146      -- acknowledge signal to indicate a new result which it recieves over four
147      -- iterations
148      ------------------------------------------------------------------------
149      Procedure get_from_core2 is
150       begin
151       wait until ack_c2 = '0';
152       core2_result(31 downto 24 ) := c2_result;
153       stb_c2 <= '0';
154       wait until ack_c2 = '1';
155       stb_c2 <= '1';
156       wait until ack_c2 = '0';
157       core2_result(23 downto 16 ) := c2_result;
158       stb_c2 <= '0';
159       wait until ack_c2 = '1';
160       stb_c2 <= '1';
161       wait until ack_c2 = '0';
162       core2_result(15 downto 8 ) := c2_result;
163       stb_c2 <= '0';
164       wait until ack_c2 = '1';
165       stb_c2 <= '1';
166       wait until ack_c2 = '0';
167       core2_result(7 downto 0 ) := c2_result;
168       stb_c2 <= '0';
169       wait until ack_c2 = '1';
170       stb_c2 <= '1';
171      end get_from_core2;
172
173      begin
174      -- initialise control ports
175      ack_in  <= '1';
176      stb_c2  <= '1';
177      stb_out <= '1';
178      control <= "000";
179      wait for 0 ns;
180      get_input_data;
181      control <= mult2_op;
182      wait for 0 ns;
183      send_to_core2(TO_FLOAT(3.0));
184      send_to_core2(a2);
185      get_from_core2;
186      Temp1 := core2_result;
187      control <= square_op;
188      wait for 0 ns;
189      send_to_core2(a1);
190      get_from_core2;
191      Temp2 := core2_result;
192      Q := ((Temp1)-(Temp2))/TO_FLOAT(9.0);
193      control <= mult3_op;
194      wait for 0 ns;
195      send_to_core2(TO_FLOAT(9.0));
196      send_to_core2(a1);
197      send_to_core2(a2);
198      get_from_core2;
199      Temp1 := core2_result;
200      control <= mult2_op;
201      wait for 0 ns;
202      send_to_core2(TO_FLOAT(27.0));
203      send_to_core2(a3);
204      get_from_core2;
205      Temp2 := core2_result;
```

```
206       control <= mult4_op;
207       wait for 0 ns;
208       send_to_core2(TO_FLOAT(2.0));
209       send_to_core2(a1);
210       send_to_core2(a1);
211       send_to_core2(a1);
212       get_from_core2;
213       Temp3 := core2_result;
214       R := ((Temp1)-(Temp2)-(Temp3))/TO_FLOAT(54.0);
215       control <= square_op;
216       wait for 0 ns;
217       send_to_core2(R);
218       R_sq := core2_result;
219       control <= mult3_op;
220       wait for 0 ns;
221       send_to_core2(Q);
222       send_to_core2(Q);
223       send_to_core2(Q);
224       get_from_core2;
225       Q_cu := core2_result;
226       D := R_sq + Q_cu;
227       if (D = TO_FLOAT(0.0)) then
228        control <= cbrt_op;
229        wait for 0 ns;
230        send_to_core2(R);
231        get_from_core2;
232        S := core2_result;
233        Temp1 := a1/TO_FLOAT(3.0);
234        control <= mult2_op;
235        wait for 0 ns;
236        send_to_core2(TO_FLOAT(2.0));
237        send_to_core2(S);
238        get_from_core2;
239        Temp2 := core2_result;
240        X1 := Temp2-Temp1;
241        X2 := TO_COMPLEX(-S-Temp1,TO_FLOAT(0.0));
242        X3 := X2;
243      elsif (D > TO_FLOAT(0.0)) then
244        control <= sqrt_op;
245        wait for 0 ns;
246        send_to_core2(D);
247        get_from_core2;
248        sqrt_D := core2_result;
249        control <= cbrt_op;
250        wait for 0 ns;
251        send_to_core2(R+sqrt_D);
252        get_from_core2;
253        S := core2_result;
254        send_to_core2(R-sqrt_D);
255        get_from_core2;
256        T := core2_result;
257        Temp1 := S+T;
258        Temp2 := a1/TO_FLOAT(3.0);
259        X1 := Temp1-Temp2;
260        control <= mult2_op;
261        wait for 0 ns;
262        send_to_core2(S-T);
263        send_to_core2(TO_FLOAT(con1));
264        get_from_core2;
265        Temp3 := core2_result;
266        X2 := TO_COMPLEX((-Temp1/TO_FLOAT(2.0))-Temp2,Temp3);
267        X3 := CONJ(X2);
268      else
269        control <= sqrt_op;
270        wait for 0 ns;
271        send_to_core2(-Q_cu);
272        get_from_core2;
273        Temp3 := core2_result;
274        control <= acos_op;
```

```
275         wait for 0 ns;
276         send_to_core2(R/Temp3);
277         get_from_core2;
278         Temp4 := core2_result;
279         theta3 := Temp4/TO_FLOAT(3.0);
280         Temp1 := a1/TO_FLOAT(3.0);
281         control <= sqrt_op;
282         wait for 0 ns;
283         send_to_core2(-Q);
284         get_from_core2;
285         Temp3 := core2_result;
286         control <= mult2_op;
287         wait for 0 ns;
288         send_to_core2(TO_FLOAT(2.0));
289         send_to_core2(Temp3);
290         get_from_core2;
291         Temp2 := core2_result;
292         control <= cos_op;
293         wait for 0 ns;
294         send_to_core2(theta3);
295         get_from_core2;
296         Temp3 := core2_result;
297         control <= mult2_op;
298         wait for 0 ns;
299         send_to_core2(Temp3);
300         send_to_core2(Temp2);
301         get_from_core2;
302         Temp4 := core2_result;
303         X1 := Temp4-Temp1;
304         control <= cos_op;
305         wait for 0 ns;
306         send_to_core2(theta3+TO_FLOAT(con2));
307         get_from_core2;
308         Temp3 := core2_result;
309         control <= mult2_op;
310         wait for 0 ns;
311         send_to_core2(Temp3);
312         send_to_core2(Temp2);
313         get_from_core2;
314         Temp4 := core2_result;
315         X2 := TO_COMPLEX(Temp4-Temp1,TO_FLOAT(0.0));
316         control <= cos_op;
317         wait for 0 ns;
318         send_to_core2(theta3+TO_FLOAT(con3));
319         get_from_core2;
320         Temp3 := core2_result;
321         control <= mult2_op;
322         wait for 0 ns;
323         send_to_core2(Temp3);
324         send_to_core2(Temp2);
325         get_from_core2;
326         Temp4 := core2_result;
327         X3 := TO_COMPLEX(Temp4-Temp1,TO_FLOAT(0.0));
328       end if;
329       send_output_result;
330     end process;
331   end;
```

Listing E.4 Arithmetic processor VHDL behavioural description

```
1  ---------------------------------------------------------------------------
2  -- The unit ack as a slave in a master slave combination. It is
3  -- a floating-point arithmetic unit that performs one of eight floating-point
4  -- operations based on a control vector provided as an input port
5  -- the control signal is defined as follows:
6  -- control          Reaction
7  -- -------          --------
8  --  000               Multiply two operands
9  --  001               Square a single operand
10 --  010               multiply three operands
11 --  011               multiply four operands
12 --  100               Square root
13 --  101               Cubic root
14 --  110               cosine function
15 --  111               inverse cosine function
16 -- Once the result is generated it is transferred back to the master unit
17 -- using the same handshaking signals but in reverse order (i.e. acknowledge
18 -- acts as strobe and vice versa)
19 ---------------------------------------------------------------------------
20
21 package UnitConst is
22   constant mult2_op  : bit_vector (2 downto 0) := "000";
23   constant square_op : bit_vector (2 downto 0) := "001";
24   constant mult3_op  : bit_vector (2 downto 0) := "010";
25   constant mult4_op  : bit_vector (2 downto 0) := "011";
26   constant sqrt_op   : bit_vector (2 downto 0) := "100";
27   constant cbrt_op   : bit_vector (2 downto 0) := "101";
28   constant cos_op    : bit_vector (2 downto 0) := "110";
29   constant acos_op   : bit_vector (2 downto 0) := "111";
30 end;
31
32 use work.UnitConst.all;
33 entity arith_pro is
34  port(input    : in bit_vector (7 downto 0);
35    stb_core1 : in bit;
36    ack_core1 : out bit;
37    control   : in bit_vector (2 downto 0);
38    result    : out bit_vector (7 downto 0));
39 end;
40 architecture behave of arith_pro is
41  begin
42  process
43    ---------------------------------------------------------------------
44    -- temporary variables to hold the input operands and the output result
45    ---------------------------------------------------------------------
46    variable out_data,x1,x2,x3,x4: float;
47    variable in_data : float;
48    ---------------------------------------------------------------------
49    -- a simple procedure that reads the data from unit1_core governed by
50    -- two handshaking signals over four iterations
51    ---------------------------------------------------------------------
52    Procedure read_data is
53     begin
54     wait until stb_core1 = '0';
55     in_data (31 downto 24) := input;
56     ack_core1 <= '0';
57     wait until stb_core1 = '1';
58     ack_core1 <= '1';
59     wait until stb_core1 = '0';
60     in_data (23 downto 16) := input;
61     ack_core1 <= '0';
62     wait until stb_core1 = '1';
63     ack_core1 <= '1';
64     wait until stb_core1 = '0';
65     in_data (15 downto 8) := input;
66     ack_core1 <= '0';
67     wait until stb_core1 = '1';
```

```
68        ack_core1 <= '1';
69        wait until stb_core1 = '0';
70        in_data (7 downto 0) := input;
71        ack_core1 <= '0';
72        wait until stb_core1 = '1';
73        ack_core1 <= '1';
74       end read_data;
75       begin
76       ack_core1 <= '1';
77       wait for 0 ns;
78       read_data;
79       x1 := in_data;
80       case control is
81        when mult2_op =>
82        read_data;
83        x2 := in_data;
84        out_data := x1 * x1;
85        when square_op =>
86        out_data := x1 * x1;
87        when mult3_op =>
88        read_data;
89        x2 := in_data;
90        read_data;
91        x3 := in_data;
92        out_data := x1 * x2 * x3;
93        when mult4_op =>
94        read_data;
95        x2 := in_data;
96        read_data;
97        x3 := in_data;
98        read_data;
99        x4 := in_data;
100       out_data := x1 * x2 * x3 * x4;
101       when sqrt_op =>
102       out_data := SQRT(x1);
103       when cbrt_op =>
104       out_data := CBRT(x1);
105       when cos_op =>
106       out_data := COS(x1);
107       when acos_op =>
108       out_data := ACOS(x1);
109       when others =>
110       null;
111     end case;
112     ------------------------------------------------------------------------
113     --  output out_data over 4 iteration to the output stage
114     --  starting with the MSBs
115     ------------------------------------------------------------------------
116     result <= out_data (31 downto 24);
117     wait for 0 ns;
118     ack_core1 <= '0';
119     wait until stb_core1 = '0';
120     ack_core1 <= '1';
121     wait until stb_core1 = '1';
122
123     result <= out_data (23 downto 16);
124     wait for 0 ns;
125     ack_core1 <= '0';
126     wait until stb_core1 = '0';
127     ack_core1 <= '1';
128     wait until stb_core1 = '1';
129
130     result <= out_data (15 downto 8);
131     wait for 0 ns;
132     ack_core1 <= '0';
133     wait until stb_core1 = '0';
134     ack_core1 <= '1';
135     wait until stb_core1 = '1';
136     result <= out_data (7 downto 0);
```

```
137       wait for 0 ns;
138       ack_core1 <= '0';
139       wait until stb_core1 = '0';
140       ack_core1 <= '1';
141       wait until stb_core1 = '1';
142     end process;
143   end behave;
```

Listing E.5 Output stage VHDL behavioural description

```
1  ---------------------------------------------------------------------------
2  -- The output stage is responsible for driving the VGA adapter that connects
3  -- to the VGA screen. starts by initializing the screen static components
4  -- such as titles and borders. Then it starts monitoring the input stage to
5  -- display the entries provided by the keyboard. The final stage includes
6  -- monitoring the core to get the floating-point outputs, performs
7  -- type conversion and display them on the screen.
8  ---------------------------------------------------------------------------
9  entity out_stage is
10  port ( key : in bit_vector (4 downto 0);
11    stb_in : in bit;
12    ack_in : out bit;
13
14    float_in : in bit_vector (31 downto 0);
15    stb_c : in bit;
16    ack_c : out bit;
17
18    vga_data : out bit_vector (7 downto 0);
19    ready : in bit
20    );
21 end;
22 architecture behave of out_stage is
23  ---------------------------------------------------------------------------
24  -- initial commands that initialises the VGA adapter and draws the static
25  -- components on the screen are provided in goups of internal ROMs. A control
26  -- loop passes through these ROMS and output the commans in order
27  ---------------------------------------------------------------------------
28  ---------------------------------------------------------------------------
29  -- ROM_SetPage initialises the VGA adapter by setting the pallete, setting the
30  -- drawing mode and the drawing page.It also draws the back groun rectange
31  -- and any other static lines.
32  ---------------------------------------------------------------------------
33  type ROM_SetPage is array(0 to 35) of bit_vector(6 downto 0);
34
35  ---------------------------------------------------------------------------
36  -- ROM_CharSet holds all the static ASCII characters such as the main title
37  -- and the variables names
38  ---------------------------------------------------------------------------
39  type ROM_CharSet is array(0 to 83) of bit_vector(6 downto 0);
40
41  ---------------------------------------------------------------------------
42  -- ROM_resetSc holds the command sequence required to reset the output results
43  -- by drawing a rectangle with the same colour as the back ground over
44  -- the output result
45  ---------------------------------------------------------------------------
46  type ROM_resetSc is array(0 to 19) of bit_vector(6 downto 0);
47
48
49  begin
50  main_process : process
51   variable adrs_set : integer range 0 to 47;
52   variable adrs_char : integer range 0 to 84;
53   variable CharSet   : ROM_CharSet := (
54   "1000110", -- F (70)
55   "1101100", -- l (108)
56   "1101111", -- o (111)
57   "1100001", -- a (97)
58   "1110100", -- t (116)
59   "1101001", -- i (105)
60   "1101110", -- n (110)
61   "1100111", -- g (103)
62   "0100000", -- space (32)
63   "1010000", -- p (80)
64   "1101111", -- o (111)
65   "1101001", -- i (105)
66   "1101110", -- n (110)
67   "1110100", -- t (116)
```

```
68      "0100000", -- space (32)
69      "1010011", -- S (83)
70      "1111001", -- y (121)
71      "1101110", -- n (110)
72      "1110100", -- t (116)
73      "1101000", -- h (104)
74      "1100101", -- e (101)
75      "1110011", -- s (115)
76      "1101001", -- i (105)
77      "1110011", -- s (115)
78
79      "1000001", -- A (65)-- address = 24
80      "0110001", -- 1 (49)
81      "0100000", -- space (32)
82      "0111101", -- = (61)
83      "0100000", -- space (32)
84
85      "1000001", -- A (65)-- address = 29
86      "0110010", -- 2 (50)
87      "0100000", -- space (32)
88      "0111101", -- = (61)
89      "0100000", -- space (32)
90
91      "1000001", -- A (65)-- address = 34
92      "0110011", -- 3 (51)
93      "0100000", -- space (32)
94      "0111101", -- = (61)
95      "0100000", -- space (32)
96
97      "1011000", -- X (88)-- address = 39
98      "0110001", -- 1 (49)
99      "0100000", -- space (32)
100     "0100000", -- space (32)
101     "0100000", -- space (32)
102     "0100000", -- space (32)
103     "0100000", -- space (32)
104     "0111101", -- = (61)
105     "0100000", -- space (32)
106
107     "1010010", -- R (82)-- address = 48
108     "1100101", -- e (101)
109     "0101000", -- ( (40)
110     "1011000", -- X (88)
111     "0110010", -- 2 (50)
112     "0101001", -- ) (41)
113     "0100000", -- space (32)
114     "0111101", -- = (61)
115     "0100000", -- space (32)
116
117     "1001001", -- I (73)-- address = 57
118     "1101101", -- m (109)
119     "0101000", -- ( (40)
120     "1011000", -- X (88)
121     "0110010", -- 2 (50)
122     "0101001", -- ) (41)
123     "0100000", -- space (32)
124     "0111101", -- = (61)
125     "0100000", -- space (32)
126
127     "1010010", -- R (82)-- address = 66
128     "1100101", -- e (101)
129     "0101000", -- ( (40)
130     "1011000", -- X (88)
131     "0110011", -- 3 (51)
132     "0101001", -- ) (41)
133     "0100000", -- space (32)
134     "0111101", -- = (61)
135     "0100000", -- space (32)
136
```

```
137      "1001001", -- I (73)-- address = 75
138      "1101101", -- m (109)
139      "0101000", -- ( (40)
140      "1011000", -- X (88)
141      "0110011", -- 3 (51)
142      "0101001", -- ) (41)
143      "0100000", -- space (32)
144      "0111101", -- = (61)
145      "0100000"  -- space (32)
146      );
147      variable SetPage  : ROM_SetPage := (
148      "0011000",  -- set the palette
149      "0011011",  -- colour 0
150      "1011101",  -- to grey ( Backgroud color)
151
152      "0011000",  -- colour 1 light blue (title and underline)
153      "1000011",
154      "0111110",
155
156      "0011001",  -- colour 2 dark blue (a1,a2,a3)
157      "0000100",
158      "0101001",
159
160      "0011001",  --colour 3 black
161      "1000000",
162      "0000000",
163
164      "0010000",  -- mode = direct draw
165      "0001100",  -- set raster page to 0
166      "0001000",  -- set render page to 0
167
168      --draw the background rectangle
169      "0100011",  -- set background color to black
170      "0110011",  -- set fore color to black
171      "0000001",  -- set point 0 to (0,0)
172      "0000000",
173      "0000000",
174      "0000000",
175      "0000010",  -- set point 1 to (639,479)
176      "0010011",
177      "1111111",
178      "1011111",
179      "1010000",  -- draw rectangle
180
181      "0100000",  -- set background color to grey
182      "0110000",  -- set fore color to grey
183      "0000001",  -- set point 0 to (10,10)
184      "0000000",
185      "0101000",
186      "0001010",
187      "0000010",  -- set point 1 to (629,370)
188      "0010011",
189      "1010110",
190      "1110010",
191      "1010000",  -- draw rectangle
192
193      "0110001",  -- set colour to light blue
194      "0000001",  -- set point 0 to (40,64)
195      "0000001",
196      "0100000",
197      "1000000",
198      "0000010",
199      "0001101",
200      "0100000",
201      "1000000",  -- set point 1 to (424,64)
202      "1001000"   -- draw line
203      );
204      variable initialise : bit := '1';   -- initialise the vga screen
205      variable x_val,a_sign_x : bit_vector (9 downto 0);
```

```
206      variable y_val : bit_vector (8 downto 0);
207      variable number : bit_vector (6 downto 0);
208      variable sign : bit;
209      variable float_val : bit_vector (31 downto 0);
210      variable temp : bit_vector (26 downto 0);
211      variable exponent : bit_vector (8 downto 0);
212      variable count : bit_vector (3 downto 0);
213      variable current_key : bit_vector (4 downto 0);
214      variable out_hund,out_ten : bit;
215
216     ------------------------------------------------------------------------
217     -- A procedure to output a single VGA command provided as an input argument
218     -- to the VGA screen.
219     ------------------------------------------------------------------------
220     procedure send2vga(inst : in bit_vector(7 downto 0)) is
221      begin
222      wait until ready = '0';
223      vga_data(7 downto 0) <= inst(7 downto 0);
224      wait until ready ='1';
225      vga_data(7) <= '0';
226     end send2vga;
227
228     ------------------------------------------------------------------------
229     -- A procedure to draw an ASCII character provided as an input argument
230     -- to the location specified by x1,y1. The character size is also provided by
231     -- the xsize and ysize arguments
232     ------------------------------------------------------------------------
233     procedure DrawChar (x1 : in bit_vector(9 downto 0);
234      y1 : in bit_vector(8 downto 0);
235      char : in bit_vector(6 downto 0);
236      xsize : in bit_vector(1 downto 0);
237      ysize : in bit_vector(1 downto 0)) is
238      begin
239      --set point 0
240      send2vga("10000001");
241      send2vga("100" & x1(9 downto 5));
242      send2vga("1" & x1(4 downto 0) & y1(8 downto 7));
243      send2vga("1" & y1(6 downto 0));
244      -- draw character
245      send2vga("111" & xsize(1 downto 0) & ysize(1 downto 0) & "0");
246      send2vga("1" & char(6 downto 0));
247     end DrawChar;
248     ------------------------------------------------------------------------
249     -- A simple procedure to read a key entry from the input stage
250     ------------------------------------------------------------------------
251     procedure get_key is
252      begin
253      wait until stb_in = '0';
254      current_key := key;
255      ack_in = '0';
256      wait until stb_in = '1';
257      ack_in <= '1';
258     end get_key;
259
260     ------------------------------------------------------------------------
261     -- A simple procedure to output the sign of the input variables
262     ------------------------------------------------------------------------
263     procedure output_sign (sign : in bit; x : in bit_vector (9 downto 0);
264      y : in bit_vector (8 downto 0)) is
265      begin
266      if (sign = '0') then -- output blank in the location
267       DrawChar(x,y,"0100000","00","00");
268      else  -- output (-) (45) in the location
269       DrawChar(x,y,"0101101","00","00");
270      end if;
271     end output_sign;
272
273
274
```

```
275     ------------------------------------------------------------------------
276     -- A  procedure to read a floating-point value from the core
277     -- and save in an internal variable (float_val)
278     ------------------------------------------------------------------------
279     procedure get_float is
280      begin
281      wait until stb_c = '0';
282      float_val := float_in;
283      ack_c <= '0';
284      wait until stb_c = '1';
285      ack_c <= '1';
286     end get_float;
287
288     ------------------------------------------------------------------------
289     -- A  procedure to display the sign of the floating-point result (the roots)
290     ------------------------------------------------------------------------
291     procedure f_output_sign is
292      begin
293      if (float_val(31) = '0') then -- the sign is plus
294       DrawChar (x_val,y_val,"0101011","00","00");
295      else
296       DrawChar (x_val,y_val,"0101101","00","00");
297      end if;
298      x_val := x_val + convert_int2bv(8,10);
299     end f_output_sign;
300
301     ------------------------------------------------------------------------
302     -- A  procedure to display the mantissa of the floating-point result
303     -- (the roots)
304     ------------------------------------------------------------------------
305     procedure f_output_mantissa is
306      begin
307      DrawChar (x_val,y_val,"0110001","00","00");     -- output the implicit 1
308      x_val := x_val + convert_int2bv(8,10);
309      DrawChar (x_val,y_val,"0101110","00","00");     -- output the decimal point
310      x_val := x_val + convert_int2bv(8,10);
311      temp := "0000" & float_val(22 downto 0);
312      -- convert the fraction to its equivelent sequence of ASCII digits
313      for j in 0 to 12 loop
314       -- multipliy by 1010 then output temp(26 downto 23)
315       -- then set temp(26 downto 23) to "0000"
316       temp := temp(25 downto 0) & "0" + temp(23 downto 0) & "000";
317       number := "000" & temp(26 downto 23);
318       -- the equivelent ascii character conversion
319       number := convert_int2bv(48,7) + number;
320       DrawChar (x_val,y_val,number,"00","00");
321       x_val := x_val + convert_int2bv(8,10);
322       temp(26 downto 23) := "0000";
323      end loop;
324     end f_output_mantissa;
325
326     ------------------------------------------------------------------------
327     -- A  procedure to display the exponent of the floating-point result
328     -- (the roots)
329     ------------------------------------------------------------------------
330     procedure f_output_exponent is
331      begin
332      -- output (space * space )
333      DrawChar (x_val,y_val,"0100000","00","00");
334      x_val := x_val + convert_int2bv(8,10);
335      DrawChar (x_val,y_val,"0101010","00","00");
336      x_val := x_val + convert_int2bv(8,10);
337      DrawChar (x_val,y_val,"0100000","00","00");
338      x_val := x_val + convert_int2bv(8,10);
339
340      DrawChar (x_val,y_val,"0110010","00","00");       -- output 2
341      x_val := x_val + convert_int2bv(8,10);
342      y_val := y_val - convert_int2bv(8,10);
343      -- now final thing output the exponent
```

```
344       exponent := "0" & float_val(30 downto 23) - convert_int2bv(127,9);
345       if (exponent (8) = '1') then -- negative exponent
346        exponent := NOT exponent + "000000001";
347        DrawChar (x_val,y_val,"0101101","00","00");
348        x_val := x_val + convert_int2bv(8,10);
349       end if;
350       if (exponent >= convert_int2bv(100,9)) then
351        -- output 1 and subtract 100
352        DrawChar (x_val,y_val,"0110001","00","00");
353        x_val := x_val + convert_int2bv(8,10);
354        exponent := exponent - convert_int2bv(100,9);
355        out_hund := '1';
356       end if;
357       if (exponent >= convert_int2bv(10,9)) then
358        count := "0000";
359        out_ten := '1';
360        loop
361         exit when exponent < convert_int2bv(10,9);
362         exponent := exponent - convert_int2bv(10,9);
363         count := count + "0001";
364        end loop;
365        number := "000" & count;
366        number := convert_int2bv(48,7) + number;
367        DrawChar (x_val,y_val,number,"00","00");
368        x_val := x_val + convert_int2bv(8,10);
369       end if;
370       if (out_hund = '1' and out_ten = '0') then
371        number := convert_int2bv(48,7);
372        DrawChar (x_val,y_val,number,"00","00");
373        x_val := x_val + convert_int2bv(8,10);
374       end if;
375       -- output the BCD LSB
376       number := exponent (6 downto 0);
377       number := convert_int2bv(48,7) + number;
378       DrawChar (x_val,y_val,number,"00","00");
379      end f_output_exponent;
380
381      ------------------------------------------------------------------------
382      -- A  procedure to control displaying the floating-point result on the VGA
383      -- screen. It checks for demornal situations and then display the number
384      -- based on three procedures declared earlier (f_output_sign,
385      -- f_output_mantissa,f_output_exponent)
386      ------------------------------------------------------------------------
387      procedure output_float is
388       begin
389       if float_val(30 downto 23) = "00000000" then     -- result = zero     (48)
390        DrawChar (x_val,y_val,"0110000","00","00");
391        -- e=255 is preserved for NaN and infinity
392       elsif float_val(30 downto 23) = "11111111" then
393        -- detected infinity
394        if float_val(22 downto 0) = "00000000000000000000000" then
395         if (float_val(31) = '0') then -- +inf
396          DrawChar (x_val,y_val,"0101011","00","00");
397         else -- -inf
398          DrawChar (x_val,y_val,"0101101","00","00");
399         end if;
400         x_val := x_val + convert_int2bv(8,10);
401         DrawChar (x_val,y_val,"1101001","00","00");
402         x_val := x_val + convert_int2bv(8,10);
403         DrawChar (x_val,y_val,"1101110","00","00");
404         x_val := x_val + convert_int2bv(8,10);
405         DrawChar (x_val,y_val,"1100110","00","00");
406        else -- NAN
407         DrawChar (x_val,y_val,"1001110","00","00");
408         x_val := x_val + convert_int2bv(8,10);
409         DrawChar (x_val,y_val,"1000001","00","00");
410         x_val := x_val + convert_int2bv(8,10);
411         DrawChar (x_val,y_val,"1001110","00","00");
412        end if;
```

```
413       else -- normal case
414         f_output_sign;
415         f_output_mantissa;
416         f_output_exponent;
417       end if;
418     end output_float;
419     ------------------------------------------------------------------------
420     -- main control sequence
421     ------------------------------------------------------------------------
422     begin
423     if (initialise = '1') then
424      adrs_set = 0;
425      out_hund := '0';
426      out_ten := '0';
427      vga_data <= "00000000";
428      wait for 0 ns;
429      -- set the pallete, draw the background and draw the underline
430      loop
431       send2vga("1" & SetPage(adrs_set));
432       exit when adrs_set = 35;
433       adrs_set = adrs_set + 1;
434      end loop;
435      adrs_char := 0;
436      initialise = '0';
437
438      -- now draw the fixed characters (title in light blue)
439      -- (a1,a2,a3) in dark blue
440      send2vga("1" & "0100000");  -- set back ground color to grey
441      send2vga("1" & "0110001");  -- set foreground color to light blue.
442      x_val := convert_int2bv(40,10);
443      y_val := convert_int2bv(32,9);
444      loop -- draw the title  -- x_size = y_size = "01";
445       DrawChar (x_val,y_val,CharSet(adrs_char),"01","01");
446       exit when adrs_char = 23;
447       adrs_char := adrs_char + 1;
448       x_val := x_val + convert_int2bv(16,10);
449      end loop;
450      send2vga("1" & "0110010");  -- set foreground color to dark blue.
451      x_val := convert_int2bv(40,10);
452      y_val := convert_int2bv(88,9);
453      adrs_char := 24;
454      loop                          -- draw a1,a2,a3
455       DrawChar (x_val,y_val,CharSet(adrs_char),"00","00");
456       exit when adrs_char = 38;
457       adrs_char := adrs_char + 1;
458       x_val := convert_int2bv(8,10);
459       if (adrs_char = 29) then
460        x_val := convert_int2bv(40,10);
461        y_val := convert_int2bv(120,9);
462       elsif (adrs_char = 34) then
463        x_val := convert_int2bv(40,10);
464        y_val := convert_int2bv(152,9);
465       end if;
466      end loop;
467      send2vga("1" & "0110011");  -- set foreground color to black.
468      x_val := convert_int2bv(40,10);
469      y_val := convert_int2bv(200,9);
470      adrs_char := 39;
471      loop -- draw x1,rex2,imx2,rex3,imx3
472       DrawChar (x_val,y_val,CharSet(adrs_char),"00","00");
473       exit when adrs_char = 83;
474       adrs_char := adrs_char + 1;
475       x_val := convert_int2bv(8,10);
476       if (adrs_char = 48) then
477        x_val := convert_int2bv(40,10);
478        y_val := convert_int2bv(232,9);
479       elsif (adrs_char = 57) then
480        x_val := convert_int2bv(40,10);
481        y_val := convert_int2bv(256,9);
```

```
482         elsif (adrs_char = 66) then
483           x_val := convert_int2bv(40,10);
484           y_val := convert_int2bv(288,9);
485         elsif (adrs_char = 75) then
486           x_val := convert_int2bv(40,10);
487           y_val := convert_int2bv(312,9);
488         end if;
489       end loop;
490     end if;
491     -- initialisation is done the process will start monitoring the input stage
492     -- to display the three input paramemters a1,a2,a3 and display them digit
493     -- by digit on the VGA screen
494     ack_in <= '1';
495     ack_c <= '1';
496     wait for 0 ns;
497     -- get a1
498     sign = '0';
499     x_val := convert_int2bv(104,10); -- 96 for the sign
500     y_val := convert_int2bv(88,9);
501     a_sign_x := convert_int2bv(96,10);
502     -- set back ground color to grey and foreground color to dark blue.
503     send2vga("1" & "0100000");
504     send2vga("1" & "0110010");
505     for i in 0 to 2 loop
506       if (i = 1) then            -- recieving the second variable (a2)
507         sign = '0';
508         x_val := convert_int2bv(104,10); -- 96 for the sign
509         y_val := convert_int2bv(120,9);
510         a_sign_x := convert_int2bv(96,10);
511       elsif (i = 2) then       -- receiving the third variable (a3)
512         sign = '0';
513         x_val := convert_int2bv(104,10); -- 96 for the sign
514         y_val := convert_int2bv(120,9);
515         a_sign_x := convert_int2bv(96,10);
516       end if;
517       loop
518         get_key;
519         if (current_key = "01101") then    -- minus
520           sign := not sign;
521           output_sign (sign, a_sign_x,y_val);
522         elsif (current_key = "10000") then -- point (46)
523           DrawChar (x_val,y_val,"0101110","00","00");
524           x_val := x_val + convert_int2bv(8,10);
525         elsif (current_key = "01111") then -- enter
526           exit;
527         else -- a digit is received generate the equielent ASCII character
528           -- and output it
529           number := "00" & current_key;
530           number := convert_int2bv(48,7) + number;
531           DrawChar (x_val,y_val,number,"00","00");
532           x_val := x_val + convert_int2bv(8,10);
533         end if;
534       end loop;
535     end loop;
536     -- the final stage is reading the roots from the core unit and display them
537     x_val := convert_int2bv(128,10); -- X1
538     y_val := convert_int2bv(200,9);
539     get_float;
540     output_float;
541     x_val := convert_int2bv(128,10); -- RE(X2)
542     y_val := convert_int2bv(232,9);
543     get_float;
544     output_float;
545     x_val := convert_int2bv(128,10); -- IM(X2)
546     y_val := convert_int2bv(256,9);
547     get_float;
548     output_float;
549     x_val := convert_int2bv(128,10); -- RE(X3)
550     y_val := convert_int2bv(288,9);
```

```
551       get_float;
552       output_float;
553       x_val := convert_int2bv(128,10); -- IM(X3)
554       y_val := convert_int2bv(312,9);
555       get_float;
556       output_float;
557     end process;
558   end behave;
```

Listing E.6 Interface unit in the first FPGA

```
1  ---------------------------------------------------------------------------
2  -- The unit is part of the final modification to the cubic equation solver
3  -- as a result of moving the output stage to the second FPGA.
4  -- The data is converted into blocks of 6-bit output and passed to interface2
5  -- in the second FPGA
6  ---------------------------------------------------------------------------
7  entity interface1 is
8   port (  key : in bit_vector (4 downto 0);
9     stb_in : in bit;
10    ack_fpga2 : in bit;
11    float_in : in bit_vector (31 downto 0);
12    stb_c : in bit;
13    ack_c : out bit;
14    stb_fpga2 : out bit;
15    ctrl_fpga2 : out bit;
16    ack_in : out bit;
17    fpga2_data : out bit_vector (5 downto 0)
18    );
19 end;
20 architecture behave of interface1 is
21  begin
22  process
23    variable count : integer range 0 to 3;
24    variable in_key : bit_vector (4 downto 0);
25    begin
26    -- first initialise all ports
27    ack_in <= '1';
28    ack_c  <= '1';
29    stb_fpga2 <= '1';
30    ctrl_fpga2 <= '0';     -- '0' means initialise the screen for a new entry
31    -- now negotiate with fpga2 to initialise screen
32    wait until ack_fpga2 = '1';
33    stb_fpga2 <= '0';
34    wait until ack_fpga2 = '0';
35    stb_fpga2 <= '1';
36    ctrl_fpga2 <= '1';
37    count := 0;                    -- count three enters
38    loop
39     wait until stb_in = '0';
40     in_key := key;
41     ack_in <= '0';
42     wait until stb_in = '1';
43     ack_in <= '1';
44     if (in_key = "01111") then   -- if key is enter
45      count := count + 1;
46     End if;
47     fpga2_data(4 downto 0) <= in_key;
48     wait until ack_fpga2 = '1';  -- send the key to the output
49     stb_fpga2 <= '0';
50     wait until ack_fpga2 = '0';
51     stb_fpga2 <= '1';
52     exit when count = 3;         -- three enters mean three parameters
53    End loop;
54    -- now receiving the five floating point variables
55    loop
56     wait until stb_c = '0';
57     fpga2_data <= float_in (5 downto 0);
58     wait until ack_fpga2 = '1';
59     stb_fpga2 <= '0';
60     wait until ack_fpga2 = '0';
61     stb_fpga2 <= '1';
62
63     fpga2_data <= float_in (11 downto 6);
64     wait until ack_fpga2 = '1';
65     stb_fpga2 <= '0';
66     wait until ack_fpga2 = '0';
67     stb_fpga2 <= '1';
```

```
68        fpga2_data <= float_in (17 downto 12);
69        wait until ack_fpga2 = '1';
70        stb_fpga2 <= '0';
71        wait until ack_fpga2 = '0';
72        stb_fpga2 <= '1';
73
74        fpga2_data <= float_in (23 downto 18);
75        wait until ack_fpga2 = '1';
76        stb_fpga2 <= '0';
77        wait until ack_fpga2 = '0';
78        stb_fpga2 <= '1';
79
80        fpga2_data <= float_in (29 downto 24);
81        wait until ack_fpga2 = '1';
82        stb_fpga2 <= '0';
83        wait until ack_fpga2 = '0';
84        stb_fpga2 <= '1';
85
86        fpga2_data <= "0000" & float_in (31 downto 30);
87        wait until ack_fpga2 = '1';
88        stb_fpga2 <= '0';
89        wait until ack_fpga2 = '0';
90        stb_fpga2 <= '1';
91        ack_c <= '0';
92        wait until stb_c = '1';
93        ack_c <= '1';
94      end loop;
95    end process;
96  end behave;
```

Listing E.7 Interface unit in the second FPGA

```
1  ------------------------------------------------------------------------------
2  -- The unit is part of the final modification to the cubic equation solver
3  -- as a result of moving the output stage to the second FPGA.
4  -- The data is received from the first FPGA in blocks of 6-bit, which gets
5  -- adjusted to the appropriate formate and passed to the output stage
6  ------------------------------------------------------------------------------
7
8  entity interface2 is
9   port (  initialise : out bit;
10    key : out bit_vector (4 downto 0);
11    stb_in : out bit;
12    ack_in : in bit;
13    float_data : out bit_vector (31 downto 0);
14    stb_core : out bit;
15    ack_core : in bit;
16    ack_fpga : out bit;
17    stb_fpga : in bit;
18    ctrl_fpga : in bit;
19    fpga_data : in bit_vector (5 downto 0)
20    );
21 end;
22 architecture behave of interface2 is
23  begin
24  process
25    variable enter_count : integer range 0 to 3;
26    variable data : bit_vector (5 downto 0);
27    variable ctrl : bit;
28    begin
29    -- first initialise all ports
30    stb_in <= '1';
31    stb_core  <= '1';
32    ack_fpga <= '1';
33    initialise <= '1';
34    protect;
35    initialise <= '0';   -- initialise VGA driver
36    -- get key from FPGA1
37    enter_count = '0';
38    loop
39     wait until stb_fpga = '0';
40     ctrl := ctrl_fpga;
41     data := fpga_data;
42     ack_fpga <= '0';
43     wait until stb_fpga = '1';
44     ack_fpga <= '1';
45     exit when ctrl = '0';
46     -- here ctrl does not equal 0
47     -- send the key to VGA
48     if (enter_count /= 3) then
49      key <= data (4 downto 0);
50      wait until ack_in = '1';
51      stb_in <= '0';
52      wait until ack_in = '0';
53      stb_in <= '1';
54      -- key is sent to output stage now check if it is an enter
55      if (data(4 downto 0) = "01111") then
56       enter_count := enter_count + 1;
57      end if;
58     else       -- we are receiving floating-point results five of them
59      float_data (5 downto 0) <= data;
60      wait until stb_fpga = '0';
61      ctrl := ctrl_fpga;
62      float_data (11 downto 6) <= fpga_data;
63      ack_fpga <= '0';
64      wait until stb_fpga = '1';
65      ack_fpga <= '1';
66      exit when ctrl = '0';
67      wait until stb_fpga = '0';
```

```
68          ctrl := ctrl_fpga;
69          float_data (17 downto 12) <= fpga_data;
70          ack_fpga <= '0';
71          wait until stb_fpga = '1';
72          ack_fpga <= '1';
73          exit when ctrl = '0';
74          wait until stb_fpga = '0';
75          ctrl := ctrl_fpga;
76          float_data (23 downto 18) <= fpga_data;
77          ack_fpga <= '0';
78          wait until stb_fpga = '1';
79          ack_fpga <= '1';
80          exit when ctrl = '0';
81          wait until stb_fpga = '0';
82          ctrl = ctrl_fpga;
83          float_data (29 downto 24) <= fpga_data;
84          ack_fpga <= '0';
85          wait until stb_fpga = '1';
86          ack_fpga <= '1';
87          exit when ctrl = '0';
88          wait until stb_fpga = '0';
89          ctrl := ctrl_fpga;
90          float_data (31 downto 30) <= fpga_data (1 downto 0);
91          ack_fpga <= '0';
92          wait until stb_fpga = '1';
93          ack_fpga <= '1';
94          exit when ctrl = '0';
95          -- the float variable is recieved now send it
96          wait until ack_core = '1';
97          stb_core <= '0';
98          wait until ack_core = '0';
99          stb_core <= '1';
100       end if;
101     end loop;
102   end process;
103 end behave;
```

Appendix F

Papers

The paper contained in this Appendix, "Floating-point Behavioural Synthesis" submitted to the IEEE Transactions on Computer-Aided Design, describes the floating-point synthesis capabilities of the MOODS synthesis system. It briefly surveys the infrastructure of the floating-point optimisation algorithm, along with a description of the way the system handles the high-level floating-point binding decisions based on a set of pre-determined interactions.

Floating point behavioural synthesis

Z. Baidas, A.D. Brown *(Senior Member)* and A.C. Williams
Department of Electronics and Computer Science
University of Southampton
Hampshire SO17 1BJ
England

Abstract

Traditionally, the data processed by a synthesised digital design is fixed (occasionally variable) width integer, and the functional units available are concomitantly simple (adders, subtractors, multipliers, multiplexers and so on). The aims of this work are two-fold: (1) to provide a library of high-level floating point functions (trigonometric, transcendental, complex) to support the synthesis of behavioural designs incorporating complicated sets of floating point operations, and (2) to incorporate this into an optimising behavioural synthesis environment. Floating point units are large and cumbersome, and an optimisation technique which allows the internal substructures of these units to be shared (in both space and time) produces a dramatic decrease in the overall hardware resources required to support a design.

The floating point modules themselves are each implemented in several ways: as an iterative series, by table lookup and using the CORDIC algorithm. The choice of implementation is left to the optimiser, which makes individual binding choices based on global knowledge of the overall design.

This paper describes the library and the optimisation algorithm and demonstrates the overall system use with an exemplar: a floating point quadratic equation solver capable of delivering complex roots, realised using 30% of a Xilinx 40125XV FPGA.

I. Introduction

Floating point number representation can simultaneously provide a large range of numbers and a high degree of precision. However, the manipulation of floating point numbers is considerably more complicated than the corresponding fixed point operations - as a consequence, a portion of modern microprocessors is devoted to dedicated hardware for floating point computation. Increasing hardware capacity and increasing power of optimisation techniques now make it possible to sensibly synthesise systems containing floating point operations on an ASIC or FPGA.

Behavioural synthesis works on a description that specifies the *relationship* between system inputs and outputs by describing abstract data structures and functions to manipulate them. The physical structure is not described, as the emphasis is on what a design *does* and not *how* it is done. In addition, the data flow manipulation aspects for a synthesis system are not generally concerned with the data *type;* the limitations of integer arithmetic are imposed simply by the lack of functional units for more complicated data types.

I.1 Existing system

The MOODS (*M*ultiple *O*bjective *O*ptimisation for *D*ata and Control path *S*ynthesis) system has been described in detail elsewhere[1-5]. In précis, this is a synthesis system which implements global optimisation of a design dataflow and control graph by the repeated application of small, reversible (behaviour preserving) transforms, under the control of a simulated annealing algorithm. The system is designed to support overall optimisation with respect to widely differing criteria; currently these are area, delay and power dissipation. The operation of the system is usually characterised by a *design trajectory* - the entire structural design is represented by its values of area, delay and power dissipation, and these numbers form the coordinates of a point in *design space*. The algorithm moves the design through this space, as in figure 1, from an initial point, created from a line-by-line translation of the user behaviour, towards a user defined goal (typically minimum area, delay and dissipation). The speed of this process allows the designer to interactively study the tradeoffs possible between the three criteria.

I.2 The floating point enhancement

An overview of the entire system is given in figure 2. The shaded boxes in figure 2 represent the aspects of the system described in this paper. The floating point optimiser

makes strategic decisions about the high level binding for each floating point unit (i.e. table lookup, iterative series or CORDIC), taking into account such issues as the type and number of each floating point operation required, and the availability and capacity of any off-chip ROM available to the system. This 'coarse design' is then presented to the simulated annealing algorithm. The front end of *this* subsystem consists of a 'module-ripping' unit[6], which allows the simulated annealing based optimiser to capitalise on similarities in the internal structures of the floating point units. . The definition of the floating point number underpinning this work is the IEEE single precision floating point standard 754-1985[9].

I.3 Other systems

Work elsewhere has implemented floating point capabilities by introducing a non-standard format[10] - this results in a reduction in both the accuracy and dynamic range available - and in the introduction of a dedicated chip to handle floating point operations[11] - effectively a discrete ALU. A set of block-diagram-based commercial tools[12,13] allows users to create graphical representations of their systems using an assortment of functions provided in block libraries. These tools mainly support fixed-point format for hardware design, with a limited support of some standard floating point operations (addition, subtraction, multiplication, and division). Global optimisation is not supported. The key point of behavioural synthesis is that the designer should not concerned with the structure, just the system functionality.

I.4 Overall strategy

The high level behavioural description (which may contain fixed and floating point operations) is defined in VHDL, parsed and translated into ICODE. This is a hardware equivalent of assembly language; the principle significant difference being that ICODE supports multiple 'control' threads and parallelism. (The rationale for a file-based step in the overall dataflow is the same as for a software development environment: an overall project may consist of many source files, only some of which will be modified at each edit cycle. Further, ICODE is language neutral - parts of a project may be described using HDLs other than VHDL, and ICODE allows the support of a mixed language design environment.)

The ICODE representation is then input to the floating point optimiser, which is the subject of this paper. It is described in detail in section III. The main synthesis

optimisation block - controlled by a simulated annealing algorithm - operates on the control and dataflow graphs of the design. Local, reversible, small transforms are applied quasi-randomly (see [1-5]), which have the effect of iteratively moving the design from the initial point to a point as close as possible to the user defined objectives (see figure 1). This approach is successful because (from the perspective of the simulated annealing algorithm) the transforms are *local* (i.e. they affect only a few other nodes each in the dataflow and control graphs) and *independent*. Attempts to use this philosophy on the floating point optimiser produced disappointing results, because each floating point mapping choice has *system-wide* repercussions; there is a very strong 'follow-me' effect. This makes iterative techniques unsuitable and constructive methods more attractive. A number of different procedures were explored; the heuristic described in section III permits MOODS to produce the 'best' results, although of course other algorithms may be just as suitable.

Section II describes the implementation of the floating point library itself (the functional units are much more complex than their fixed point counterparts), and section III describes how these structures are manipulated by the floating point optimisation algorithm. Finally, two examples are analysed, illustrating the behaviour of the system on non-trivial designs.

II. The floating point library

The floating point modules currently supported are given in table 1. (There is no reason in principle why the complex counterparts of all the functions cannot be supported; we chose to support those recommended in the IEEE standard 1076.2[14]) Each functional unit consists of three building blocks:

1. Range reduction.
2. Function evaluation.
3. Post evaluation rounding and normalisation.

II.1 Range reduction

The large dynamic range provided by a floating point representation introduces a problem when designing systems to handle floating point arithmetic. Some evaluation methods, such as iterative series, converge over a wide range of arguments. However, achieving a good accuracy over that range requires taking many terms into account. Other methods, such as the CORDIC[11,15] algorithm have an in-built limited range of

convergence. Having a suitable technique to reduce the range of the input operand(s) is therefore essential. Each function has its own range reduction unit: periodic and symmetric functions have obvious reductions, and others include shifting and scaling (for example, $\ln(M.2^E) \Rightarrow \ln(M) + E.\ln(2)$)

II.2 Function evaluation

Three different *base techniques* (table lookup, iterative series and CORDIC) are used to implement the functional units; these techniques generate modules with significantly different physical properties.

II.2.1 Table lookup

Look-up tables are frequently and trivially used to evaluate mathematical functions. This scheme has often been rejected in practical cases because of the large table sizes required for acceptable accuracy. However, combining range reduction techniques with a dedicated interpolation procedure gives rise to a large reduction in table size, often to the point that it may be reduced to an *on-chip* set of static registers, rather than an external ROM. For example, evaluating $sin(x)$ to an accuracy of 1% , using linear interpolation, requires a table with just six entries. Further reduction in table size can be achieved by partitioning a table into several smaller sub-tables that handle intervals of a function, each with its own scaling factor appropriate to that sub-table[16]. Even greater reduction in table size may be achieved by replacing the linear interpolation procedure with a quadratic or even higher order interpolation procedure, but the additional costs of the interpolation engine usually outweigh this advantage (although of course the interpolation unit can be shared amongst an arbitrary number of tables).

Finally, the scaling factor and intervals in the previous discussion can be trivially forced to be a power of two, so that all the division operations during interpolation may be replaced by fast binary right shift operations.

II.2.2 Iterative series

In this method the value of a function $f(x)$ is obtained by an iterative process; the value of x is inserted into some formula and after a number of operations the value is obtained. This is attractive when the relationship between adjacent terms in the series is

simple, or when the accuracy requirements are low. For example, the Taylor expansion of

$$sin(x) \text{ is } \sin(x) = \sum_{n=0}^{\infty} t_n; \quad t_n = (-1)^n . \frac{x^{2n+1}}{(2n+1)!} \quad and \quad t_{n+1} = t_n \left(\frac{-1}{2} \right) \frac{x^2}{(n+1)(2n+3)} \forall x$$

The main issue in calculating functions using power series is the number of terms that need to be taken in order to ensure that the result is accurate to the desired precision.

II.2.3 The CORDIC algorithm

The CORDIC (*Co-Ordinate Rotation DIgital Computer*) algorithm[11,15,17] was introduced as the basis for a navigational computer. Its principal advantages are that it requires no multipliers, and can generate two function results simultaneously.

It is an iterative process, applied to a set of input variables (x, y, z) for n iterations, to generate a result accurate to n digits. Each iteration involves a shift, an add and an add constant operation. The capabilities of the algorithm are summarised in figure 3.

II.3 Post evaluation rounding and normalisation

At this stage, the output result is adjusted to comply with the IEEE 754 standard. This involves

1. Rounding the quotient by conditionally adding one to the least significant bit. All the floating point library modules work to an internal accuracy of 28 bits - the IEEE standard has 23.

2. Normalising the quotient by shifting and adjusting the exponent field. The standard saves a bit by assuming the most significant bit of the fraction field is always set (which means it need not be saved) and modifying the exponent accordingly.

3. Providing the special symbols to represent unusual events (infinity, NAN). Finally, any range reduction effects are inverted.

III. The floating point optimisation block

The task of this block is to assign a base technique (i.e. one of the three implementation methods above) to each floating point functional unit in the design. The aim of the process is *not* to produce directly an implementation that will meet the global design parameters specified by the user, rather to produce an intermediate implementation that makes it likely that the simulated annealing based optimiser will be able to approach the design objectives.

III.1 Function implementation interactions

The attributes of each function implementation considered in isolation are easy to compare: to generate *sin(x)* with a table requires the table itself (which may be internal, or external, requiring an interface), plus an interpolation engine. To generate it with a series requires a cumulative adder plus a term generator, which may require a table, but no interpolation engine. All these elements have easily quantifiable area and speed costs. However, when a number of functions are required, new interactions become important:

- There is an overhead to interfacing an ASIC/FPGA to an external ROM or RAM, but it is fixed and independent of the number of external function tables.
- The CORDIC algorithm can generate two function results simultaneously.
- Once an iterative series generator has been instantiated, the cost of switching between different functions is relatively small.
- Once a complex function is implemented, the equivalent real function is virtually free in most cases.
- Some functions are built as a hierarchical composition of other functional units. If these units are already available, the total cost is reduced.
- Some functions can be realised as functions of other functions. If these are already available, part of the required behaviour can be 'bootstrapped'.
- Some function tables are subsets of others.
- If the external ROM size is limited, the *distribution* of tables onto the ROM affects the overall area and speed.
- A low accuracy functional unit is a complete subset of any higher accuracy counterpart.
- If a high number of functional units are to mapped onto an external ROM, the multiplexer costs can become comparable to the cost of an alternative base technique.

Diverse interactions such as these require a dedicated optimisation algorithm.

III.2 Practical function implementation

The overall synthesis system operates by instantiating sequential multi-cycle technology-independent functional blocks, which are inline expanded in the internal design structure during synthesis. These expanded modules act as templates, and enable the implementation of functional units not available in the MOODS technology library[6]. The floating point functional units are provided to the synthesis system as a set of

expanded modules. This enables inter-module optimisation at the sub-module level, allowing greater opportunities for functional unit sharing. In addition, a pre-processing stage handles floating point functional unit binding to the base technique expanded modules to help the main synthesis core to reach an optimum that meets the users supplied objectives.

The pre-processing stage performs three tasks:

1. Hierarchical unit expansion
2. Exception register allocation
3. Base technique optimisation

III.2.1 Hierarchical unit expansion

Many floating point and complex functional units in the library are provided as a hierarchical structure of common building blocks. This approach allows the final synthesis stage to share the common building blocks of different arithmetic units, which results in a significant reduction of the total area cost. In addition, partitioning the arithmetic units into a number of building blocks allows effective pipelining. This results in a reduction of the total delay and increases the throughput of the whole system. As an example, consider the pseudo-code of figure 4.

The *sine* function is expanded into two sub-blocks, the range reduction stage (*sin_cos_pre()*), and the function evaluation stage (*sin_cos_main()*) - figure 4b. A large number of sub-blocks are common to more than one floating point unit. They communicate with each other by means of (automatically generated) temporary buffers, which are initialized by the system to allow the sub-blocks to know which floating point unit they are actually representing. For example, *buf1* in figure 4b will be initialized to tell *sin_cos_pre()* it is representing a *sin()*, and *sin_cos_pre()* may write the range reduction details into *buf1* to be picked up by *sin_cos_main()*. The complex type conversion function *polar_to_complex()* is expanded into further building blocks (*sine*, *cosine*, two floating point multipliers and two type converters) - figure 4c. The *sine* and *cosine* functions are then further expanded (figure 4d). This approach makes it easy for the main annealing algorithm to exploit functional unit duplication.

Note that RE() and IM() in figure 4c are similar to PL/1 pseudofunctions: if they appear on the right hand side of an assignment, they return a *value*, if they appear on the left hand side, they provide *access*.

As an aside, it is useful at this point to review the unit hierarchy utilised by the system; this will put the numeric results presented later into context. The hierarchy is shown in table 2. It is the job of the floating point optimiser (this paper, section III) to realise floating point library units (level 1) in terms of floating point primitives (level 2). It is the job of the simulated annealing optimiser[1-8] to realise the floating point primitives in terms of MOODS functional units (level 3). Finally, the MOODS functional units will be realised in terms of (for the purposes of this paper) FPGA CLBs. This is done by the low-level logic optimiser supplied by the FPGA manufacturer. (Alternatively, the system can target ASICs directly - in this case, low level logic optimisation and placement and routing are needed.) At each step in the process, virtual units at one level can *share* physical units at the underlying level.

III.2.2 Exception status register

Implementing floating point operations necessarily implies supporting some kind of 'exception notification', to handle illegal operation attempts, over- and underflow, and so on. The IEEE floating point standard[9] defines the behaviour of a floating point system in pathological situations, both in terms of bit patterns in the floating point variable itself, and in a *status register*. The status register is a six-bit register that indicates the integrity of a floating point operation. Asserting one of the flags (bits) is analogous to throwing an exception; it is the responsibility of the user to handle (capture) the exception.

From a broad perspective, there are two sensible places for such a register in a VHDL design. It is possible to have a single, global status register that can be accessed by any instruction within a process - the user must provide a dedicated monitor process for the register, and must decide what action (if any) is to be taken if a flag is raised. Equally, each floating point unit may have its own local register, and handle problems in its own way. Overloads of the floating point functions allow the user to use either (or both) technique(s) - if a flag register is supplied as an actual argument to a function instantiation, it is used; otherwise the existence of a global register is assumed. (This register will be *provided* automatically by the system, but any *monitoring* process is the responsibility of the user.)

III 2.3 Accuracy considerations

The introduction of a floating point capability gives rise to a fourth gross design parameter - that of *accuracy*. This cannot be treated on an equal footing with the other three dimensions of design space because the effects of changing the accuracy of a functional unit cannot be localised - a change to any module in the dataflow graph will threaten all operations predicated upon it. Errors propagate and interact non-linearly, and furthermore the extent and form of the interactions are invariably data dependent. It is not difficult to construct a process where a change of component accuracy ultimately affects behaviour. The system supports user specification of floating point accuracy at two levels: it is possible to assert an overall accuracy on a process, (each individual floating point operation in the process will deliver this accuracy) and it is possible to override this and assign individual accuracies to each floating point operation. Within each operator, a differential error propagation model[18] is employed to calculate the necessary accuracies of each of the sub-blocks, given the required accuracy of the parent operator itself. Where sub-blocks are shared between operators later by the system, the accuracy of each shared sub-block is promoted to the value of the most accurate.

Figure 5 shows the design space trajectories for a large process (example 1), with a variety of user constraints and goals, optimised with a number of different accuracy requirements. The original behavioural VHDL process description contains sin(), arctan(), exp(), ln(), arcsin() and sqrt(). Each trajectory consists of five points: 0.0001% accuracy (the end marked with a solid point), 0.001%, 0.01%, 0.1% and 1%. Trajectories T1..T3 are optimised with respect to area - changing the accuracy requirements impacts almost entirely on the *delay*. Trajectories T4..T6 are optimised with respect to delay, and changing accuracy requirements are traded off against system *area*. Trajectory parameters are given in table 3 - the trajectories in figure 5 are the final, physical characteristics of alternative structures delivering the same behaviour. "Delay" (table 3) is plotted against "Physical CLBs (datapath)" + "Physical CLBs (controller). The floating point functions (table 2, level 1) are implemented in terms of 10 virtual floating point primitives (level 2). The floating point optimiser cannot share any of these units because they are all different, hence 10 physical floating point primitives are required. (The point of this example is to demonstrate the effects of changing *accuracy*.) Depending on the amount of off-chip ROM available and the (user imposed) accuracy requirements, differing base technique bindings are asserted, which give rise to the "Virtual functional unit" column. The simulated annealing algorithm maps these onto a reduced number of physical functional

units (table 2, level 3). These are implemented in terms of virtual CLBs, which are mapped onto physical CLBs by the low level optimiser/router supplied with the FPGAs. Comparing the "Virtual CLB" column with the "Physical CLB (datapath)" column shows that this step does not gain much.

Some points of particular note are:

- Any user instructions for accuracies in excess of 0.0001% are ignored (i.e. treated as 0.0001%), as the floating point internal structure cannot support the results.
- If a very low accuracy is required (less than 1%) the resource impact of the function generation cores becomes negligible. The area requirements are dominated by the range reduction and post processing units.
- In trajectories T3 and T6, the whole design is realised as a set of table lookup modules utilising an external ROM; accuracy variation has no effect on the system parameters.
- The trajectories do not all terminate at exactly the same point because of numerical noise - recall that the principal optimisation process is controlled by a simulated annealing algorithm.

III.2.4 Base technique optimisation

The floating point optimiser (figure 2) operates on the floating point and complex functions within the design, binding each floating point operation to a suitable base technique component from the floating point module library.

The algorithm consists of two non-interacting phases: *external memory utilisation* and *on-chip optimisation*. Empirical results indicate that by far the best results (in terms of area and delay) can be obtained by utilising table lookup implemented on off-chip ROM to its fullest extent; the system therefore attempts to do this before attempting to handle other interactions.

III.2.4.1 External memory utilisation

Each module in the expanded module library and technology dependent library has two figures of merit associated with it: the *delay* and the *area*. In the floating point library, these are expanded: the area factor is split into the *on-chip* area and the *off-chip* area. The approach here is simple: the algorithm performs an exhaustive search of all possible combinations of table lookup mapping to see which utilises the ROM most effectively. Note that this does not lead to a combinatorial explosion: a table is necessary for each floating point module *type*, not *instance*, and in practise, subtable isomorphism within the

list of table 1 means that the largest number of off-chip tables ever considered cannot be larger than six.

III.2.4.2 On-chip optimisation

The flowchart of this phase of the system is shown in figure 6.

Step (1): All remaining floating point modules are mapped onto a table lookup base technique, implemented on an infinite, virtual, internal (on-chip) RAM. If this meets the user area constraints, and fits the physical system, the base technique mapping is complete and successful.

Step (2): Select the biggest (irrespective of user requirements) floating point functional unit. (Step (1) gave the *fastest* possible mapping - here we are iteratively trading speed against area until we can deliver the user requirements.)

Step (3): Increment the mapping for that unit. (The base technique mappings are ordered: 1. Linear table, 2. Piecewise linear table, 3. Iterative series, 4. CORDIC. Note that step (1) maps all units to 1 (linear table), and attempting to increment past 4 has no effect. Not all mappings are allowed for all floating point module types - see table 1.)

Step (4): The effect on the overall area of the mapping change is estimated. If the area is not reduced, goto step (5). Otherwise, the new mapping is accepted, and if the overall user requirements are satisfied, the algorithm terminates successfully.

Step (5): If all the floating point functional units are mapped onto the CORDIC base technique, and the user requirements are not met, then the algorithm terminates in failure. Otherwise, return to step (2).

The shaded decision boxes in figure 6 represent an invocation of the 'Estimator'. This is a subsystem that predicts, from the current state of the design dataflow and control graph, what *further* improvements the simulated annealing based optimisation (see figure 2) will be able to make. The Estimator is a heuristic algorithm, which takes as input the statistical properties of the design (for example, variable and operator count - both fixed and floating point, control constructs and so on) and *predicts* the compression that the simulated annealing phase will be able to achieve with a reasonable degree of accuracy (90-95%).

The design of this system is derived from observations of base technique interactions. Some points of particular interest are:

- Functions based on table lookup implemented on off-chip ROM share a single ROM controller and a single I/O port.

- Expanding the hierarchical (real and complex) functions before the optimisation phase permits substructure sharing. If both the complex and real instances of a function are required, this delivers significant cost reductions.

- Mapping a function onto a CORDIC base technique makes subsequent mappings to that implementation more likely.

- Two or more functions having the same table (for example *sin()* and *cos()*) have only one physical table.

- The cost of an iterative series generator can be significantly changed by the prior availability of its primitive subunits (multiplier, divider). Equally, the selection of this base technique reduces the cost of other operations by providing these units.

Figure 7 shows the effects of this process, with the accuracy criterion set to 10^{-6} on the process of figure 5. In figure 7a, the user requirements are optimisation with respect to delay alone, and the set of histograms shows the distribution of the functional units between the three base techniques, as a function of off-chip ROM capacity. The tradeoff between the two table-based implementations as the external ROM resource becomes scarcer is clear, with CORDIC and iterative series only coming into play as a last resort.

Figure 7b shows the same design, this time optimised for minimum area. Here, the main tradeoff is between external ROM usage and iterative series, as the latter obviously consumes less area than the on-chip table lookup, although there is a delay price to pay.

IV. Quadratic equation solver (example 2)

The power of this level of description is illustrated by implementing a complex quadratic equation solver. This takes as input the three (real) quadratic coefficients and delivers the (possibly complex) roots. The VHDL behavioural description of the design is listed in figure 8. The design space accuracy trajectories for this process are virtually identical, as the design has only one non-linear operator, the square root. The process is optimised from (area, delay) coordinates of (7800,1800) to a small region centered around (1542,376), an improvement of a factor of 5 in both dimensions. Note that the floating point optimiser has realised 12 floating point functions with only 4 physical floating point primitives. Quantitative details are given in table 4 for a target accuracy of 10^{-6}.

V. Final remarks

V.1 Comparison with other published results

Comparable studies are hard to find.

[10] reports a study of floating point arithmetic on FPGAs, but the authors have their own (reduced bit width) floating point format, the synthesis is RTL based, and there is no optimisation - by using RTL, the designer is forcing the cycle-by-cycle timing, which is an important degree of freedom exploited by MOODS. Speed is reported in terms of time, not clock cycles.

[11] describes the *manual* design and characterisation of two versions of a single real floating point square root system implemented on Xilinx XC4000 series FPGAs. Table 5 shows a comparison of this functional unit with the corresponding unit from the MOODS library. The third row is an implementation that can be switched between real and complex by a single control line. (The behaviour is the same as setting the imaginary part of the operand to zero, but this implementation is significantly smaller than the sum of the other two.) The data taken from [11] is hard to interpret, because the authors distinguish between "CLB functional generators" and "CLB flip-flops". Each Xilinx CLB is composed of two function generators and two flip-flops, and it is not clear how these figures map onto the total CLB usage on the chip.

The key point of *this* work, however, is that the MOODS floating point library is designed to exploit module sharing and support large designs consisting of many invocations of units from within the library.

V.2 Conclusions

Even with the increasing size and processing power of silicon systems, the difficulties of synthesising sizeable circuits containing floating point operations are great. The system described here allows a designer to manipulate floating point and complex variables *on an equal footing* with all other data types (fixed point, access operations, control stuctures), and the additional complexities arising are hidden from the user. Module decomposition, space- and time multiplexing of submodules and accuracy considerations are all controlled by three simple user specified parameters: the desired area, the desired speed and the desired accuracy. The designer is free to concentrate on the functionality of the design, and does not have to worry about the implementation details, which, of course, is the goal of every behavioural synthesis system.

Acknowledgements

This work was supported by EPSRC grant reference GR/L28494, "High level floating point synthesis library".

References

1. Baker K.R., Currie A.J. and Nichols K. G., "Multiple objective optimisation in a behavioural synthesis system", *IEE Proc.-Circuits Devices & Systems*, 140, August 1993, pp 253-260.

2. Brown, A.D., Baker K. R. and Williams, A.C., "Online testing of statically and dynamically scheduled synthesized systems", *IEEE-CAD* 16, no 1, pp 47-57, 1997.

3. Williams A. C., "A behavioural VHDL synthesis system using data path optimisation", *PhD thesis*, University of Southampton, UK, July 1997.

4. Baker K. R., Brown A. D. and Currie A. J., "Optimisation efficiency in behavioural synthesis", *IEE Proc.-Circuits Devices & Systems*, 141, no. 5, pp 399-406, 1994

5. Baker K. R., "Multiple objective optimisation of data and control paths in a behavioural silicon compiler", *PhD thesis*, University of Southampton, UK, September 1992.

6. Williams, A.C, Brown, A.D and Baidas, Z., "Hierarchical module expansion using templates', *FDL'98 conference, Swiss Federal Institute of Technology, Lausanne,* September 1998.

7. Nijhar, T.P.K, and Brown, A.D., "Source level optimisation of VHDL for behavioural synthesis", *IEE proceedings on Computers and Digital Techniques,* 144, no 1, January 1997, pp 1-6.

8. Nijhar, T.P.K, and A.D. Brown, A.D., "HDL-specific source level behavioural optimisation", *IEE proceedings on Computers and Digital Techniques*, 144, no 2, March 1997, pp 138-144.

9. IEEE Standard for Binary Floating point Arithmetic, ANSI/IEEE Std 754-1985

10. Shirazi N., Walters, A.L. and Athanas, P. "Quantitative analysis of floating point arithmetic on FPGA based custom computing machines", Report, Virginia state University, January 1995.

11. Wakamatsu A. "Implementation of single precision floating point square root on FPGAs", *Fifth annual IEEE SYMPOSIUM on field-programmable custom computing machines*, 1997, pp 226-232.

12. Barbara T., "Finally, behavioural synthesis is production-ready", *Computer Design,* 36, no. 7, pp 57-63, July 1997.

13. Barbara T., "Behavioural synthesis yet to prove itself beyond DSP", *Computer Design*, 34, no. 6, pp 88-96, June 1995.

14. Standard VHDL Language Mathematical Packages (MATH_REAL and MATH_COMPLEX), 1076.2-1996

15. Volder J. E., "The CORDIC trigonometric computing technique", *IRE Trans. Electron. Comput.* 8, 1959, pp 330-334.

16. Chance R. J., "The effect of processor architecture on an efficient floating point table look-up algorithm", *Microprocessors and Microsystems*, 15, no. 8, October 1991, pp 411-415.

17. Mazenc C., Merrheim X. and Muller J., "Computing functions $\cos^{-1}$and $\sin^{-1}$ using CORDIC", *IEEE Transactions on Computers*, 42, no. 1, January 1993, pp.118 121.

18. Mutrie M, and Bartels R, "An approach to floating point error analysis using computer algebra", *ACM Trans. Math.*, 7, 1992, pp 284-293.

Figure and table captions

	REAL			COMPLEX#		
FUNCTION	**Table**	**Iterative**	**CORDIC**	**Table**	**Iterative**	**CORDIC**
*		**			**	
/		**			**	
+		**			**	
-		**			**	
ln(z)	y	y	n	y	y	n
$\log_{10}(z)$	y	y	n	y	y	n
$\log_2(z)$	y	y	n	y	y	n
$\log_n(z)$	y	y	n	y	y	n
sin(z)	y	y	y	y	y	y
cos(z)	y	y	y	y	y	y
tan(z)	y	y	y	n	n	n
arcsin(z)	n	y	y	n	n	n
arccos(z)	n	y	y	n	n	n
arctan(z)	y	y	y	n	n	n
sinh(z)	y	y	n	y	y	y
cosh(z)	y	y	n	y	y	y
tanh(z)	y	y	n	n	n	n
arcsinh(z)	y	y	n	n	n	n
arccosh(z)	y	y	n	n	n	n
arctanh(z)	y	y	n	n	n	n
e^z	y	y	n	y	y	y
$z1^{z2}$	y	y	n	y	y	y
sqrt(z)	y	y	n	y	y	y
conj(z)		*			**	
real(z)		*			**	
imag(z)		*			**	
magn(z)		*			**	
arg(z)		*			**	
complex_to_polar(z)		n/a			y	
polar_to_complex(z)		n/a			y	
to_float( )		##			n/a	
to_complex(,)		n/a			##	

** These are implemented using a single base technique/functional unit.

* These return trivial results

The complex functions are all overloaded to allow input as real, complex polars or complex Cartesians

Type changing functions support translation between VHDL type *real* and *integer*[14] and IEEE *float* and *complex*[9].

Table 1: The floating point library

Level	Name	Number of units (inc. overloads)	Examples
1	Floating point functions (table 1)	53	complex arcsin(), $z1^{z2}$
2	Floating point primitives	125	sin_cos_pre(), sin_cos_main()
3	MOODS functional units	16	fixed point multiply, MUX, shift
4	CLBs	--	Xilinx function generators, flip-flops

Table 2: The unit hierarchy

Trajectory	Optimisation objective	External ROM size (kbyte)	Accuracy	Delay (clock cycles)	"Area" (unit count)								
					Floating point functions*	Virtual floating point primitives*	Physical floating point primitives*	Virtual functional units*	Physical functional units*	Virtual CLBs*	Physical CLBs (datapath)*	Physical CLBs (controller)
Original	n/a	n/a	10^{-6}	523	6	10	10	525		36204	32307	1113
T1	area	0.000	10^{-2}	140				537	283	2515	2020	558
			10^{-3}	140				537	283	2544	2044	558
			10^{-4}	614				544	395	3624	2983	584
			10^{-5}	781				550	397	3733	3065	590
			10^{-6}	946				550	398	3848	3213	596
T2		3.410	10^{-2}	160				514	275	2602	1908	538
			10^{-3}	160				514	275	2602	1908	538
			10^{-4}	160				514	275	2602	1908	538
			10^{-5}	204				551	295	2989	2244	587
			10^{-6}	694				547	310	3159	2593	592
T3		∞	10^{-2}	160				514	275	2602	1908	538
			10^{-3}	160				514	275	2602	1908	538
			10^{-4}	160				514	275	2602	1908	538
			10^{-5}	160				514	275	2602	1908	538
			10^{-6}	160				514	275	2602	1908	538
T4	delay	0.000	10^{-2}	140				519	285	2622	2148	513
			10^{-3}	140				519	285	2651	2177	513
			10^{-4}	140				519	285	3774	3300	513
			10^{-5}	140				519	285	5698	5124	513
			10^{-6}	159				555	305	11134	10181	550
T5		3.410	10^{-2}	160				514	278	2648	2105	532
			10^{-3}	160				514	278	2648	2105	532
			10^{-4}	160				514	278	2648	2105	532
			10^{-5}	157				527	298	4139	3524	549
			10^{-6}	155				528	299	7682	7058	542
T6		∞	10^{-2}	160				514	278	2648	2105	532
			10^{-3}	160				514	278	2648	2105	532
			10^{-4}	160				514	278	2648	2105	532
			10^{-5}	160				514	278	2648	2105	532
			10^{-6}	160				514	278	2648	2105	532

* Up to the point of final realisation, the 'area' cost refers to data path only. The controller is held in an abstract form until the final implementation - it is optimised by MOODS *on an equal basis* to the datapath elements.

Table 3: Parameters for the design space trajectories of figure 5.

Trajectory	Optimisation objective	External ROM size (kbyte)	Delay (clock cycles)	"Area" (unit count)							
				Floating point functions*	Virtual floating point primitives*	Physical floating point primitives*	Virtual functional units*	Physical functional units*	Virtual CLBs*	Physical CLBs (datapath)*	Physical CLBs (controller)
Original	n/a	n/a	1800	12	12	4	968		5983	5560	2240
T1	area	0.000	321				974	190	1058	939	741
T2		0.700	325				973	187	857	745	765
T3		1.100	323				967	180	851	753	747
T4	delay	0.000	256				968	207	1287	1156	729
T5		0.700	261				973	210	875	773	752
T6		1.100	261				967	204	878	776	734

* Up to the point of final realisation, the 'area' cost refers to data path only.

Table 4: Parameters for the design space trajectories of the quadratic equation solver (example 2).

	Performance		CLB function generator	CLB flip-flops
	Latency (clock cycles)	**Issue** (clock cycles)		
Iterative	25	24	82	138
Pipeline	15	1	408	675

(i) Floating point square root FPGA implementation details (from [11]).

	Speed (clock cycles)	**CLB usage**
Real	20	297
Complex polar	25	314
Real & complex polar	26	363

(ii) *Isolated* floating point (real and complex polar) square root FPGA MOODS implementation details.

Table 5: Comparison between the MOODS square root unit and that from [11].

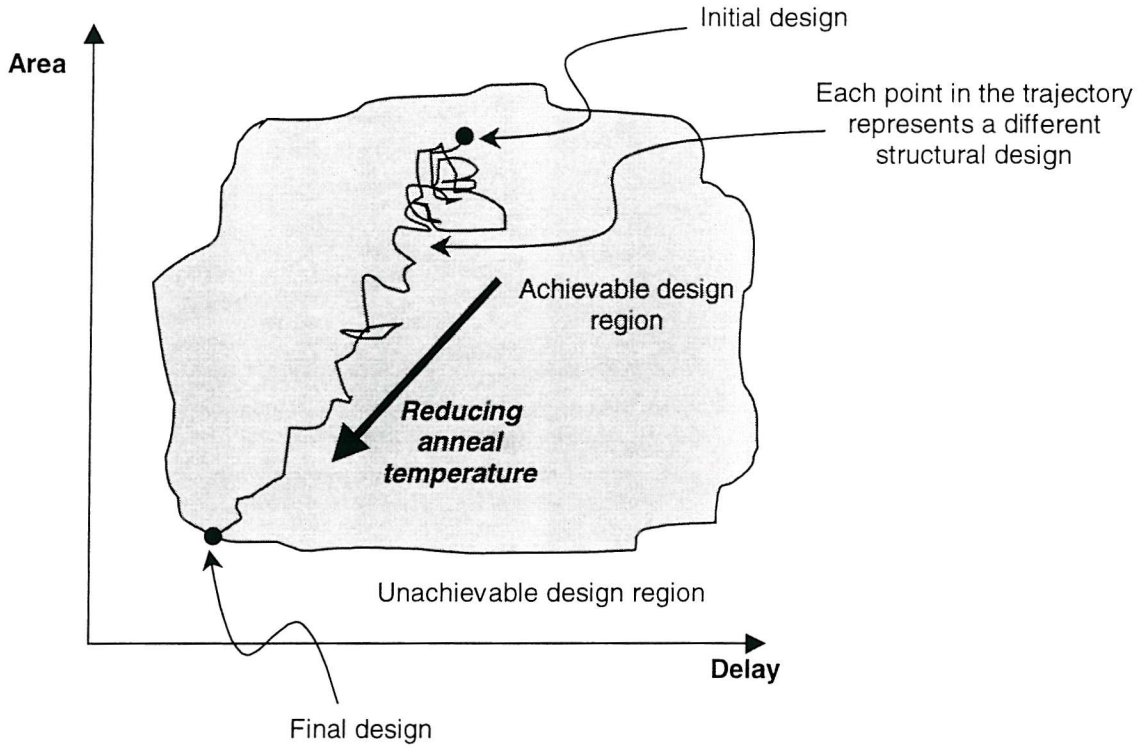


Figure 1: A two-dimensional projection (area/delay) of behavioural design space.

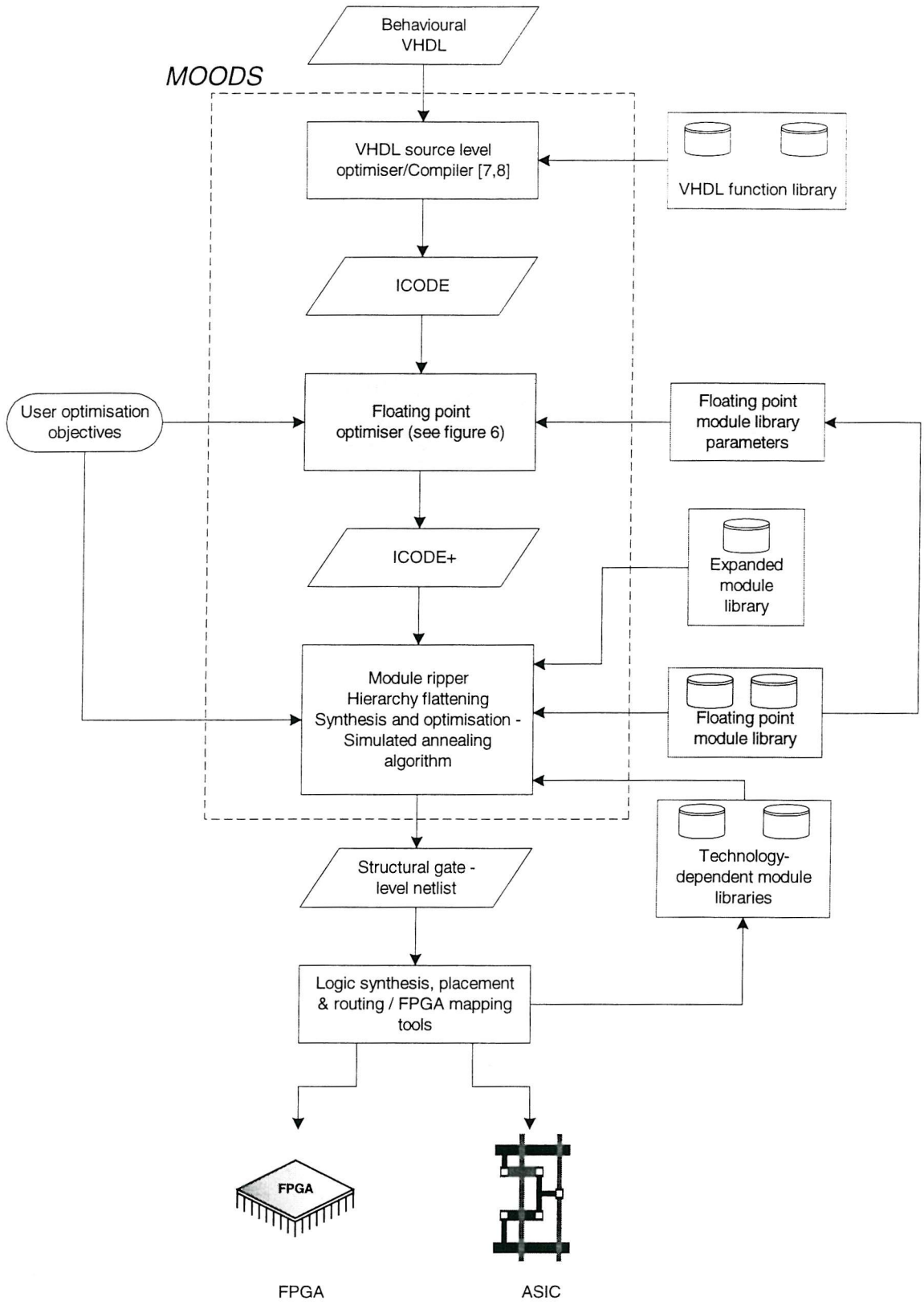


Figure 2: The overall synthesis system

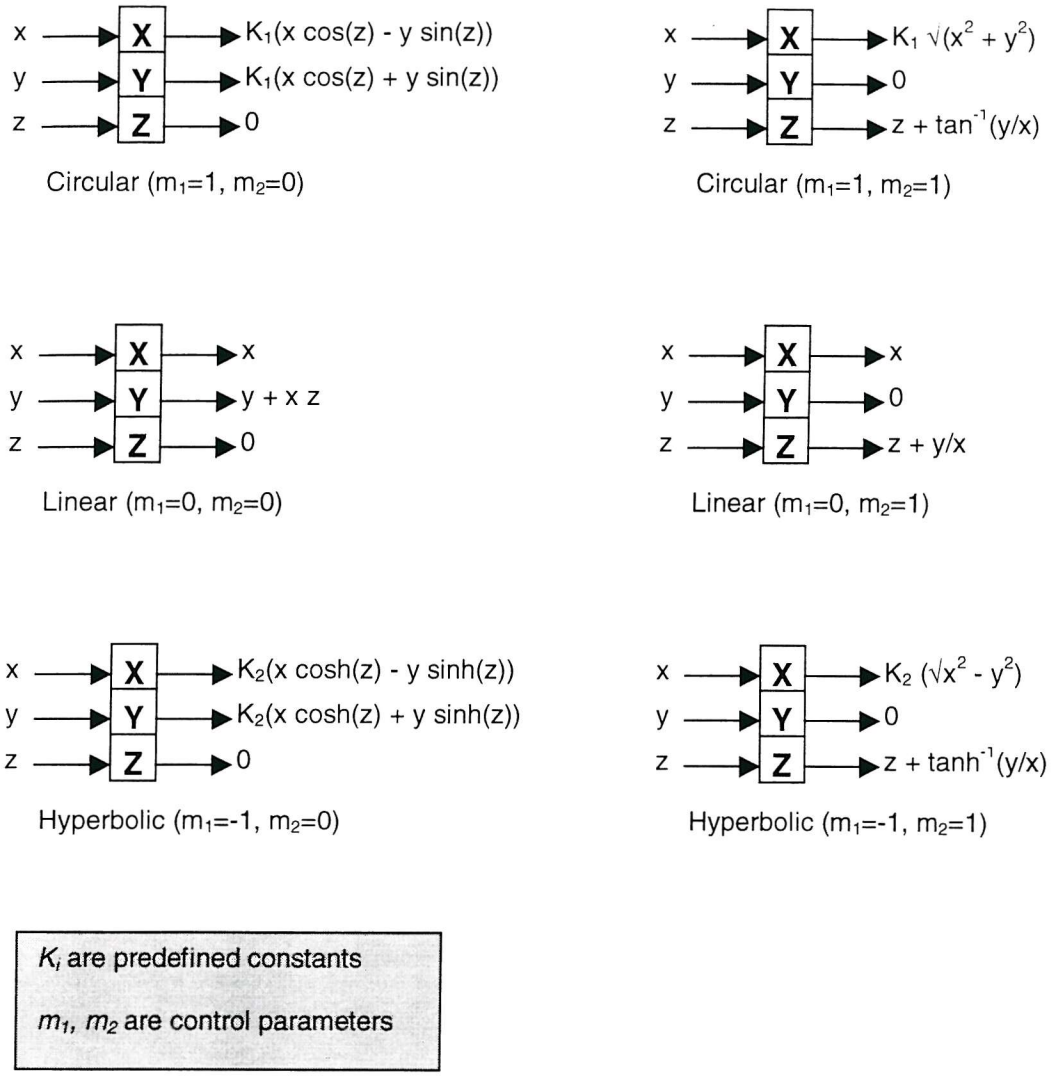


Figure 3: The CORDIC algorithm

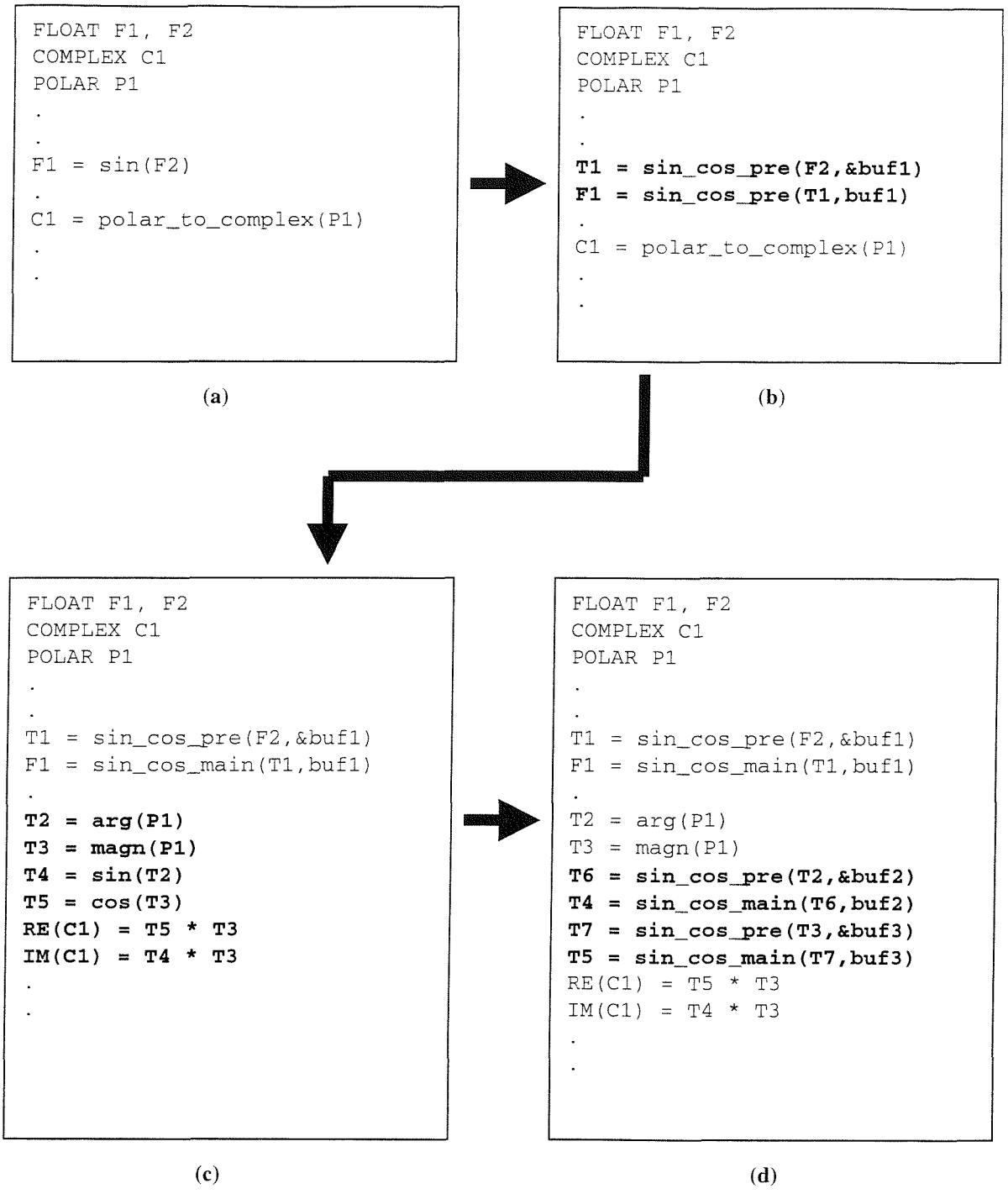


Figure 4: Hierarchical floating point unit expansion

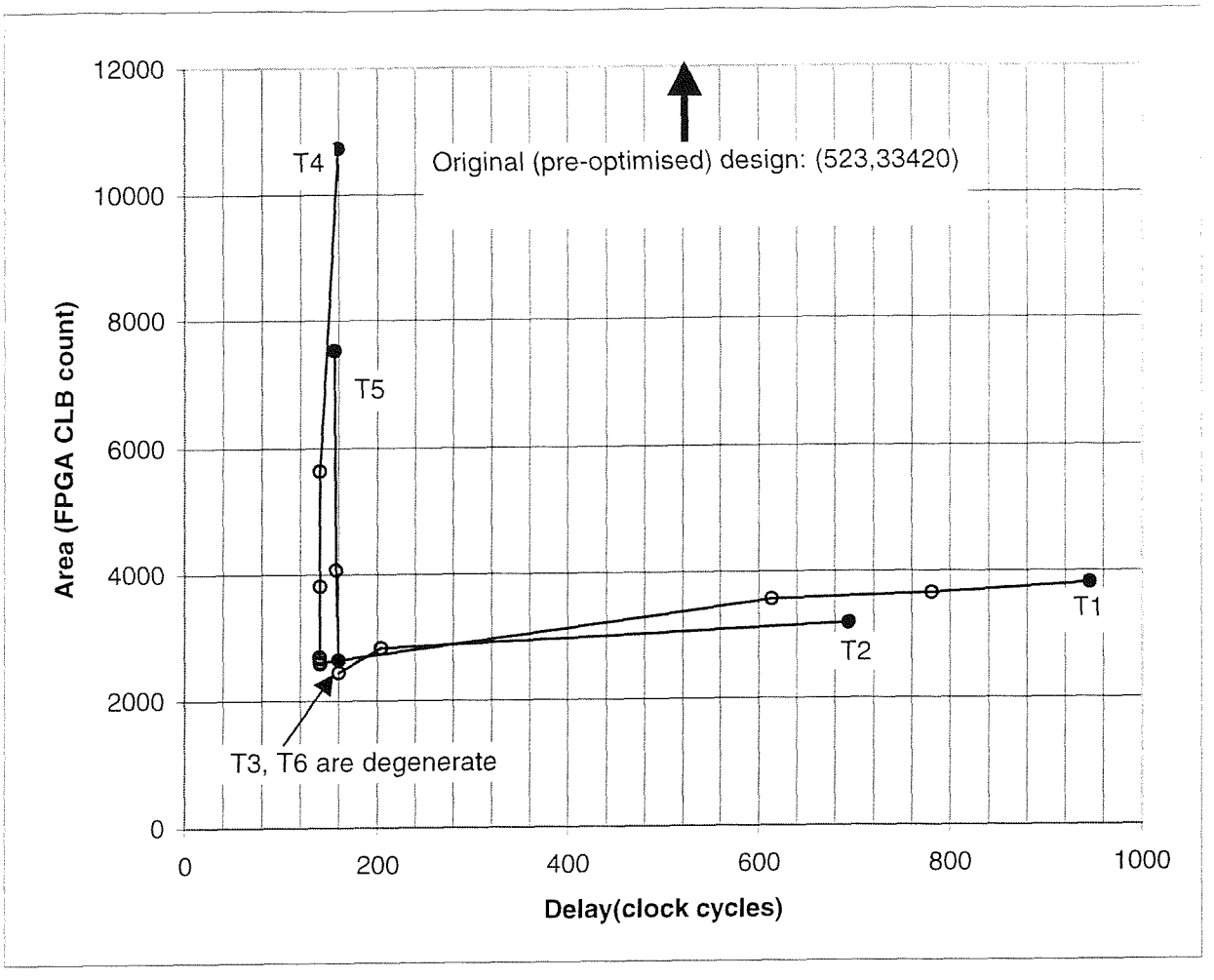


Figure 5: **Design space trajectories, showing the movement of a complex design as user accuracy requirements change.**

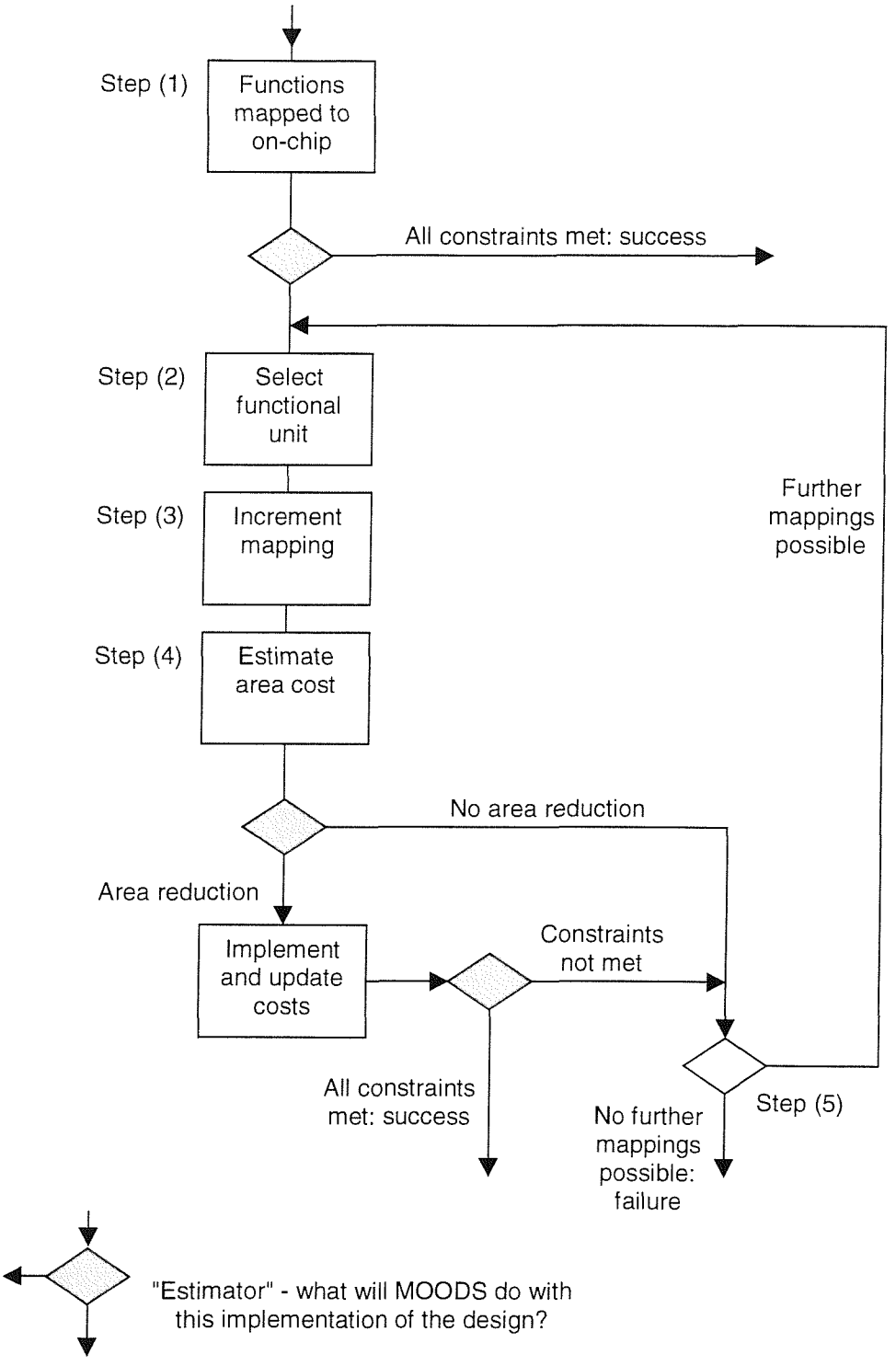


Figure 6: On chip optimisation algorithm

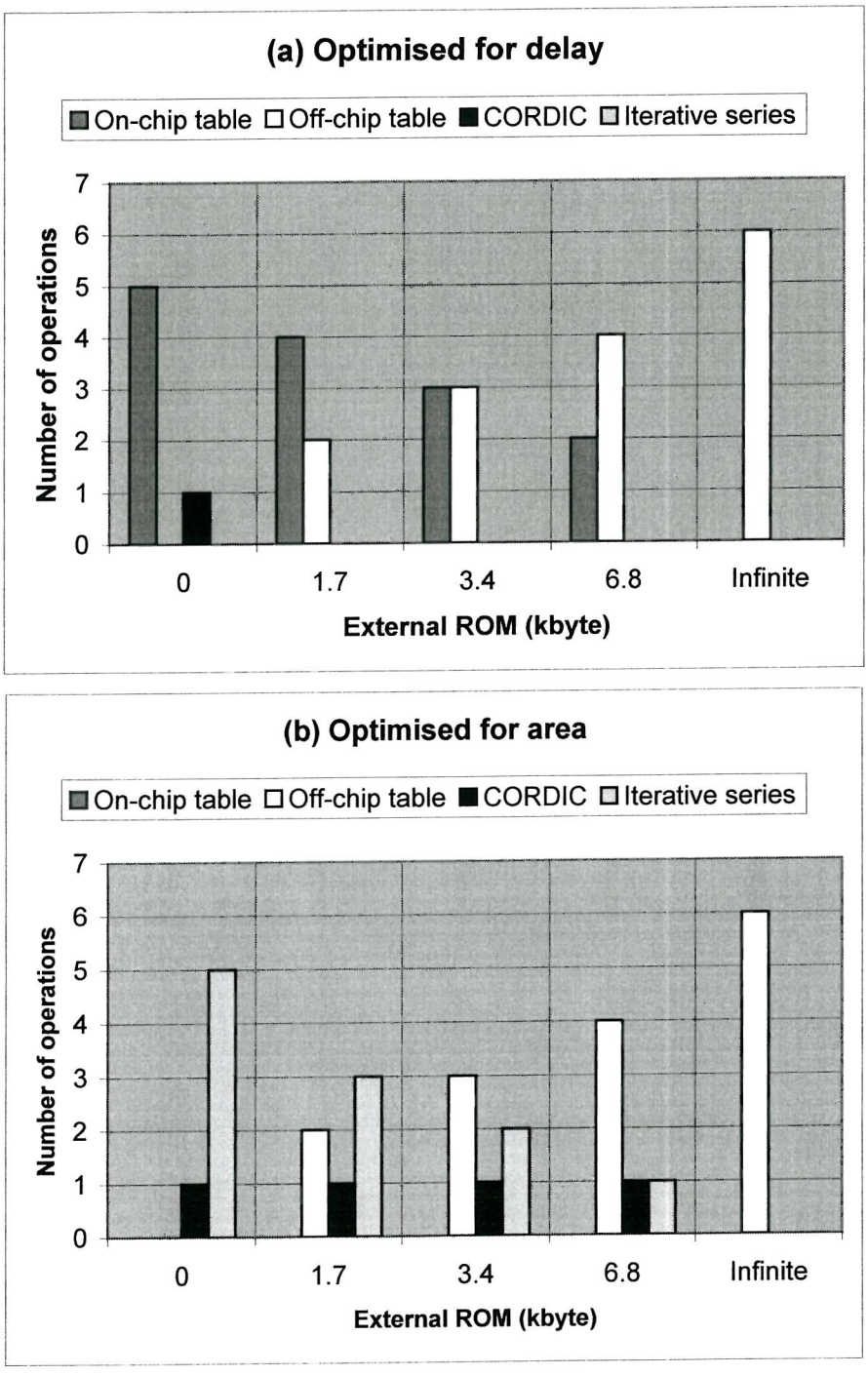


Figure 7: **Distribution of functional unit bindings between the three base techniques as a function of external ROM size.**

```
ENTITY quad IS
PORT (a,b,c            : IN  float;
      single_root      : OUT bit;
      output1,output2  : OUT complex;
      ready            : OUT bit
     );
END;

ARCHITECTURE behave OF quad IS
BEGIN
  PROCESS
  VARIABLE t1 : float;
  VARIABLE t2 : complex;
  BEGIN
    ready <= '0';
    WAIT FOR 10 ns;
    IF (a = to_float(0)) THEN
      single_root <= '1';
      t1 := -c/b;
      output1 <= float_to_complex(t1,to_float(0));
    ELSE
      single_root <= '0';
      t2 := sqrt(b*b - to_float(4)*a*c);
      output1 <= (float_to_complex(-b,to_float(0))-t2)/(to_float(2)*a);
      output2 <= (float_to_complex(-b,to_float(0))+t2)/(to_float(2)*a);
    ENDIF;
    ready <= '1';
    WAIT FOR 10 ns;
  END PROCESS;
END behave;
```

Figure 8: Quadratic equation solver behavioural description

References

1. Baker, K.R. - Currie, A.J. - Nichols, K.G., "Multiple Objective Optimisation in a Behavioural Synthesis System", IEE Proceedings - G, Vol. 140, No.4 August 1993, pp. 253-260.

2. Baker, K.R. - Brown, A.D. - Currie, A.J., "Optimisation Efficiency in Behavioural Synthesis", IEE Proceedings on Circuits, Devices and Systems, Vol. 141, No. 5, October 1994, pp. 399-406.

3. Brown, A.D. - Baker, K.R. - Williams, A.C., "On-Line Testing of Statically and Dynamically Scheduled Synthesized Systems", IEEE Transactions on Computer-Aided Design, Vol. 16, No. 1, January 1997, pp. 47-57.

4. Baker, Keith R., "Multiple Objective Optimisation of Data and Control Paths in a Behavioural Silicon Compiler", PhD Thesis, University of Southampton, September 1992.

5. Baker, Keith R., "The MOODS Synthesis System - User Manual v2.xx", University of Southampton, July 1993.

6. "Standard VHDL Reference Manual, IEEE Std 1076-1993", IEEE Catalog No. SH16840, 1993.

7. McFarland, M.C. - Parker, A.C. - Camposano, R., "The High-Level Synthesis of Digital Systems", Proceedings of the IEEE, Vol. 78, February 1990, pp. 301-318.

8. Camposano, R., "From Behavior to Structure: High-Level Synthesis", IEEE Design and Test of Computers, Vol. 7, No. 5, October 1990, pp. 8-19.

9. Rushton, A., "VHDL for Logic Synthesis", McGraw-Hill, 1995, ISBN: 0-070-09092-6.

10. Lin, Youn-Long, "Recent Development in High-Level Synthesis", ACM Transactions on Design Automation of Electronic Systems, Vol. 2, No. 1, January 1997, pp. 2-21.

11. Gajski, Daniel D. - Ramachandran, Loganath, "Introduction to High-Level Synthesis", IEEE Design and Test of Computers, Vol. 11, No. 4, Winter 1994, pp. 45-54.

12. Micheli, Gionanni, "High Level Synthesis of Digital Circuits", IEEE Design and Test of Computers, Vol. 7, No. 5, October 1990, pp. 6-7.

13. Camposano, R. - Saunders, L.F. - Tabet, R.M., "VHDL as Input for High Level Synthesis", IEEE Design and Test of Computers, Vol. 8, No. 1,March 1991, pp. 43-49.

14. Eles, Petru - Kuchcinski, Krzysztof - Minea, Marius, "Compiling VHDL into a High-level Synthesis Design Representation", EURO-DAC 92 : European Design Automation Conference, Ch. 121, 1992, pp. 604-609.

15. McFarland, M.C. - Parker, A.C. - Camposano, R., "Tutorial on High-Level Synthesis", Proceedings of the 25th ACM/IEEE Design Automation Conference, 1988, pp. 330-336.

16. Nijhar, T.P.K. - Brown, A.D., "HDL-Specific Source Level Behavioural Optimisation", IEE Proceedings-Computers and Digital Techniques, Vol. 144, No. 2, 1997, pp. 138-144.

17. Nijhar, T.P.K. - Brown, A.D., "Source Level Optimisation of VHDL for Behavioural Synthesis", IEE Proceedings-Computers and Digital Techniques, Vol. 144, No. 1, 1997, pp.1-6.

18. Camposano, Raul, "Behavioral Synthesis", Design Automation Conference, Ch. 161, 1996, pp. 33-34.

19. Williams, A.C., "A Behavioural VHDL Synthesis System using Data path Optimisation", PhD Thesis, University of Southampton, July 1997.

20. De Micheli, Giovanni, "Synthesis and Optimisation of Digital Circuits", McGraw-Hill, 1994, ISBN: 0-071-13271-6.

21. McFarland, M.C., "Reevaluating the Design Space for Register-Transfer Hardware Synthesis", IEEE International Conference on Computer-Aided Design, ICCAD-87 - Digest of Technical Papers, 1987, Ch. 119, pp. 262-265.

22. Brewer, Forrest - Gajski, Daniel, "Chippe: A System for Constraint Driven Behavioural Synthesis", IEEE Transactions on Computer-Aided Design, Vol.9, No. 7, July 1990, pp. 681-694.

23. Camposano, Raul - Rosenstiel, Wolfgang, "Synthesizing Circuits From Behavioral Descriptions", IEEE Transactions on Computer-Aided Design, Vol. 8, No. 2, February 1989, pp. 171-180.

24. Walker, Robert A. - Chaudhuri, Samit, "Introduction to the Scheduling Problem", IEEE Design and Test of Computers, Vol. 12, No. 2, Summer 1995, pp. 60-69.

25. Xia, C. - Cheng, H.D., "High-Level Synthesis: Current Status and Future Prospects", Circuits Systems Signal Processing, Vol. 14, No. 3, 1995, pp. 351-400.

26. Paulin, Pierre G. - Knight, John P., "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", IEEE Transactions on Computer-Aided Design, Vol.8, No. 6, June 1989, pp. 661-678.

27. Tseng, Chia-Jeng - Siewiorek, Daniel P., "Automated Synthesis of Data paths in Digital Systems", IEEE Transactions on Computer-Aided Design, Vol. 5, No. 3, July 1986, pp. 379-395.

28. Septien, J. - Mozos, D. - Tirado, J.F. - Hermida, R. - Fernandez, M. - Mecha, H., "FIDIAS: An Integral Approach to High-Level Synthesis", IEE Proceedings - Circuits, Devices and Systems, Vol. 142, No. 4, August 1995.

29. "Quick Start Guide for Xilinx Alliance Series 1.5", Xilinx, Version 1.5, 1998.

30. "LeonardoSpectrum User's Guide", Exemplar Logic, Inc. Version 1999.1, 1999.

31. "Synergy VHDL Synthesizer and Optimizer Tutorial", Cadence Design Systems, Version 2.2, June 1995.

32. Baker, Keith R., "Writing Behavioural VHDL for MOODS Synthesis - User Manual v1.xx", University of Southampton, July 1993.

33. Rutenbar, Rob A., "Simulated Annealing Algorithms: An Overview", IEEE Circuits and Devices, Vol. 5, No. 1, January 1989, pp. 19-26.

34. Nahar, S. - Sahni, S. - Shragowitz, E., "Simulated Annealing and Combinatorial Optimisation", 23rd Design Automation Conference, 1986, pp. 293-299.

35. Kirkpatrick, Scott, "Optimization by Simulated Annealing: Quantitative Studies", Journal of Statistical Physics, Vol. 34, No. 5/6, 1984, pp. 975-986.

36. Kirkpatrick, S. - Gelatt Jr., C.D. - Vecchi, M.P., "Optimization by Simulated Annealing", Science, 13 May 1983, Vol. 220, No. 4598, pp. 671-680.

37. Metropolis, N. - Rosenbluth, A. - Teller, A. - Teller, E., "Equation of State Calculations by Fast Computing Machines", Journal of Chemical Physics, Vol. 21, 1087, 1953.

38. Williams, A.C. - Brown, A.D. - Baidas, Z.A., "Optimisation in Behavioural Synthesis using Hierarchical Expansion: Module Ripping", In Preparation.

39. Williams, A.C. - Brown, A.D. - Baidas, Z.A., "Hierarchical Module Expansion in a VHDL Behavioural Synthesis System", FDL'98, September 1998.

40. Goldberg, David, "The Design of Floating-Point Data Types", ACM Letters on Programming Languages and Systems, Vol. 1, No. 2, June 1992, pp. 138-151.

41. "IEEE Standard for Binary Floating-Point Arithmetic, IEEE Std 754-1985", 1985.

42. "IEEE Standard for Radix-Independent Floating-Point Arithmetic, IEEE Std 854-1987", 1987.

43. Advanced Micro Devices Inc., "IEEE Floating-Point Format", Microprocessors and Microsystems, Vol. 12, No. 1, 1988, pp. 13-23.

44. Henkel, Hartmut, "Improved Addition for Logarithmic Number system", IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. 37, No. 2, February 1989.

45. Lai, Fang-shi, "A Hybrid Number System Processor with Geometric and Complex Arithmetic Capabilities", IEEE Transactions on Computers, Vol. 40, No. 8, August 1991.

46. Das, Debasish - Mukhopadhayaya, Krishnendu - Sinha, Bhabani P., "Implementation of Four Common Functions on an LNS Co-Processor", IEEE Transactions on Computers, Vol. 44, No. 1, January 1995, pp. 155-161.

47. Coleman, J.N. - Chester, E.I., "A 32-bit Logarithmic Arithmetic Unit and its Performance Compared to Floating Point", 14th IEEE Symposium on Computer Arithmetic, Proceedings, Ch. 32, 1999, pp. 142-151.

48. Oberman, Stuart Franklin, "Design Issues in high Performance Floating point Arithmetic Units", Technical report, Stanford University, Reference No. CSL-TR-96-711, December 1996.

49. Booth, Andrew D., "A Signed Binary Multiplication Technique", Quarterly Journal of Mechanics and Applied Mathematics, Vol. 4, 1951, pp. 236-240.

50. Al_Tawaijry, Hesham - Flynn, Michael, "Performance/Area Tradeoffs in Booth Multipliers", Technical report, Stanford University, Reference No. CSL-TR-95-684, November 1995.

51. Bewick, Gray W., "Fast multiplication: Algorithms and Implementation", PhD Thesis, Stanford University, February 1994.

52. Wilson, J.B. - Ledley, R.S., "An Algorithm for Rapid binary Division", IRE Transactions on Electronic Computers Vol. 16, 1961, pp. 224-226.

53. Ledely, Robert Steven, "Digital Computer and Control Engineering", McGraw-Hill, 1960.

54. Oberman, Stuart Franklin, "Design Issues in High Performance Floating Point Arithmetic Units", PhD Thesis, Stanford University, November 1996.

55. Volder, J.E., "The CORDIC Trigonometric Computing Technique", IRE Transactions on Electronic Computers, Vol. EC-8, 1959,pp.330-334.

56. Walther, J.S., "A Unified Algorithm for Elementary Functions", Spring Joint Computer Conference Proceedings, Vol. 38, 1971, pp.379-385.

57. Mazenc, Christophe - Merrheim, Xavier - Muller, Jean-Michel, "Computing Functions $\cos^{-1}$ and $\sin^{-1}$ Using Cordic", IEEE Transactions on Computers, Vol. 42, No. 1, January 1993, pp. 118-122.

58. Wong, W.F. - Goto, E., "Fast Evaluation of The Elementary Functions in Single Precision", IEEE Transactions on Computers, Vol. 44, No. 6, March 1995, pp. 453-457.

59. Gal, Shmuel - Bachelis, Boris, "An Accurate elementary Mathematical Library for the IEEE Floating Point Standard", ACM Transactions on Mathematical Software, Vol. 17, No. 1, 1991, pp. 26-45.

60. Atkinson, Kendall E., "An Introduction to Numerical Analysis", John Wiley & Sons, 1978, ISBN: 0-471-02985-8.

61. Muller, Jean-michel, "Elementary Functions, Algorithms and Implementation", Birkhauser, 1997, ISBN: 0-817-63990-X.

62. Ligon, Walter B. - McMillan, Scott - Monn, Greg, "A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs", IEEE Symposium on FPGAs for Custom Computing Machines Proceedings, Ch .65, 1998, pp .206-215.

63. Li, Yamin - Chu, Wanming, "Implementation of Single Precision Floating Point Square Root on FPGAs", 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, Ch. 32, 1997, pp. 226-232.

64. Louca, Loucas - Cook, Todd A. - Johnson, Willian H., "Implementation of IEEE single Precision Floating Point Addition and Multiplication on FPGAs", ", IEEE Symposium on FPGAs for Custom Computing Machines Proceedings, Ch. 24, 1996, pp. 107-116.

65. Shirazi, Nabil - Walters, Al - Athanas, Peter, "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines".

66. Xing, Shanzhen - Yu, William Wing Hong, "FPGA-Based Floating-Point Datapath Design for Geometry Processing", SPIE Conference on Configurable Computing: Technology and Application, November 1998.

67. Brunvand, Erik - Novak, Joe H., "Using FPGAs to Prototype Self-Timed Floating-Point Co-Processors", IEEE 1994 Custom Integrated Circuits Conference, 1994.

68. Scheelen, J., "Floating-Point DSP Primitives for the ASA Silicon compiler", International Conference on DSP Applications and Technology, Ch. 10, 1991.

69. Kyrloglou, N.A. - Kouforavlou, O.G. - Goutis, C.E., "Number Format conversion: Algorithm and VLSI Module Generator", Int. J. Electronics, Vol. 73, No. 1, 1992, pp.145-156.

70. Houelle, A. - Mehrez, H., "On Portable Macro-Cell FPU Generators Using the Fully 754-IEEE Standard", IEEE Transactions on VLSI Systems, Vol. 6, No. 1, 1998, pp. 1749-1754.

71. Aberbour, Mourad - Houelle, Alain - Mehrez, Habeb - Vaucher, Nicolas - Guyot, Alain, "On Portable Macrocell FPU Generators for Division and Square Root Operators Complying to the Full IEEE-754 Standard", IEEE Transactions on VLSI Systems, Vol. 6, No. 1, March 1998, pp. 114-121.

72. Compan, A. - Debaud, P. - Delorme, V. - Francois, J.A. - Mehrez, H. - Pecheux, F., "GAF : A Portable Standard-Cell Floating Point Adder Generator Using The CXgen Function Library", Microprocessing and Microprogramming, Vol. 32, 1991, pp. 637-644.

73. "COSSAP Design Environment Datasheet", Synopsys, Inc., 1999.

74. "Behavioural Compiler Datasheet", Synopsys, Inc., 1999.

75. "Design Compiler Datasheet", Synopsys, Inc., 1999.

76. "Datasheet Signal Processing Workstation with Convergence Simulation Architecture", Cadence Design Systems, 1998.

77. "Datasheet Visual Architect", Cadence Design Systems, 1997.

78. "Datasheet SPW Floating-Point Communications Library", Cadence Design Systems, 1997.

79. Barbara, T., "Finally, Behavioural Synthesis is Production Ready", Computer Design, Vol. 36, No. 7, July 1997, pp. 57-63.

80. Stiefel, Eduard L., "An Introduction to Numerical Mathematics", Academic Press, 1963.

81. Chance, R.J., "The Effect of Processor Architecture on an Efficient Floating-Point Table look-up Algorithm", Microprocessors and Microsystems, Vol. 15, No. 8, October 1991, pp. 411-415.

82. Hahn, Helmut - Timmermann, Dirk - Hosticka, Bedrich J. - Rix, Bernold, "A Unified and Division-Free CORDIC Argument Reduction Method with Unlimited Convergence Domain Including Inverse Hyperbolic Functions", IEEE Transactions on Computers, Vol. 43, No. 11, November 1994, pp. 1339-1344.

83. Hu, Yu Hen, "The Quantization Effects of the CORDIC Algorithm", IEEE Transactions on Signal Processing, Vol. 40, No. 4, April 1992, pp. 834-844.

84. Kota, Kishore - Cavallaro, Joseph R., "Numerical Accuracy and Hardware Tradeoffs for CORDIC Arithmetic for Special -Purpose Processors", IEEE Transactions on Computers, Vol. 42, No. 7, July 1993, pp. 769-779.

85. Spiegel, Murray R., "Theory and Problems of Complex Variables", McGraw-Hill, 1974, ISBN: 0-070-84382-1.

86. Weltner, K. - Grosjean, J. - Schuster, P. - Weber, W.J., "Mathematics for Engineers and Scientists", Stanley Thornes, 1995, ISBN: 0-859-50120-5.

87. Char,B.W. - Geddes, K.O. - Gonnet, G.H. - Leong, B.L. - Monagan, M.B. - Watt, S.M., "Maple V Library Reference Manual", Springer-Verlag, 1991.

88. Hennessy, John L. - Patterson, David A., "Computer Organization and Design, The Hardware/Software Interface", Morgan Kaufmann, 1994, ISBN: 1-558-60282-8.

89. "Standard VHDL Language Mathematical Package (MATH_REAL and MATH_COMPLEX), IEEE P1076.2", 1996.

90. Nhon, T.Q. - Flynn, M., "An Improved Algorithm for High-Speed Floating-Point Addition", Technical report, Stanford University, Reference No. CSL-TR-90-442, August 1990.

91. Al-Twaijry, Hesham Abdulaziz, "Area and Performance Optimised CMOS Multiplier", PhD Thesis, Stanford University, August 1997.

92. Oberman, Stuart F. - Flyyn, Michael J., "Design Issues in Floating-Point Division", Technical report, Stanford University, Reference No. CSL-TR-94-647, December 1994.

93. Oberman, Stuart F. - Flyyn, Michael J., "An Analysis of Division Algorithms and Implementations", Technical report, Stanford University, Reference No. CSL-TR-95-675, December 1996.

94. Oberman, Stuart F. - Flyyn, Michael J., "Division Algorithms and Implementation", IEEE Transactions on Computers, Vol. 46, No. 8, 1997, pp. 833-854.

95. Churchhouse, R.F., "Handbook of Applicable Mathematics", Vol. 3, John Wiley and Sons, 1981, ISBN: 0-471-27947-1.

96. Swartzlander, Earl E. [editor], "Computer Arithmetic", Dowden, Hutchinson & Ross Inc. 1980, ISBN 0-879-33350-2.

97. Mutrie, Mark P.W. - Bartels, Richard H. - Char, Bruce W., "An Approach for Floating-Point Error Analysis using Computer Algebra", ISSAC '92. Papers from the international symposium on Symbolic and algebraic computation, 1992, pp. 284-293.

98. Bauer, F.L., "Computational Graphs and Rounding Error", Siam Journal of Numerical Analysis, Vol. 11, No. 1, March 1974, pp. 87-96.

99. Molenkamp, J.H.J. - Goldman, V.V. - Hulzen, Van, "An Improved Approach to Automatic Error Cumulation Control", Proceedings of the 1991 international symposium on Symbolic and algebraic computation , 1991, pp. 414-418.

100. Hulshof, B.J.A. - Van Hulzen, J.A., "Automatic Error Cumulation Control", EUROSAM 84: International Symposium on Symbolic and Algebraic Computation, Ch. 37, 1984, pp. 260-271.

101. "The Programmable Logic Data Book", Xilinx, 1998, PN 0010323.

102. Burger, Robert G. - Dybvig, R. Kent, "Printing Floating-Point Numbers Quickly and Accurately", ACM Sigplan Notices, Vol. 31, No. 5, 1996, pp. 108-116.

103. Wakerly, John F., "Digital Design Principles and Practices", 2nd ed., Prentice Hall, 1994, ISBN: 0-130-59973-5.

104. Langet, S.H., "A Comparison of the Floating-Point Performance of Current Computers", Computers in Physics, Vol. 12, No. 4, July/August 1998, pp. 338-345.

105. Goldberg, David, "What every computer scientist should know about floating-point arithmetic", ACM Computing Surveys, Vol. 23, No. 1, March 1991, pp. 5-48.

106. Alfred, V. Aho - Jeffrey, D. Ullman, "Principles of compiler design", Addison-Wesley, 1977, ISBN: 0-201-00022-9.

107. Kreyszig, Erwin, "Advanced Engineering Mathematics", 7th ed., Jhon Wiley & Sons, 1993, ISBN: 0-471-59989-1.

108. Press, W.H. - Teukolsky, S.A. - Vetteling, W.T. - Flannery, B.P., "Numerical Recipes in C: The Art of Scientific Computing", Cambridge University Press, 2nd edition, 1992, ISBN: 0-521-43108-5.

109. Davenport, J.H. - Siret, Y. - Tournier, E., "Computer Algebra Systems and Algorithms for Algebraic Computation", Academic Press, 1988, ISBN: 0-122-04230-1.

110. "3DNOW Technology Manual", Advanced Micro Devices, Inc., 1998.

111. Thakkar, Shreekant - Huff, Tom, "The Internet Streaming SIMD Extensions", Intel Technology Journal Q2, 1999.

112. Dewar, R.B.K. - Smosna, M., "Microprocessors a Programmer's View", McGraw-Hill, 1990, ISBN: 0-070-16638-2.

113. Milton, D.J.D., "Memory Allocation within Hardware Synthesis", Transfer Thesis, University of Southampton, August 1999.