EFFICIENT AREA OPTIMIZATION IN HIGH-LEVEL SYNTHESIS USING PRIORITY-DRIVEN SIMULATED ANNEALING

by

MARIA ABI SAAD

M.S., Computer Engineering, Lebanese American University, 2009

Thesis submitted in partial fulfillment of the requirements for the Degree of Master of Science in Computer Engineering

Division of Computer and Electrical Engineering

LEBANESE AMERICAN UNIVERSITY

January 2009

LEBANESE AMERICAN UNIVERSITY

Thesis approval Form

Student Name:	Maria Abi Saad	I.D.: 200104136
Thesis Title :	Efficient Area Optimization in High-Level Synthesis Using Priority-Driven Simulated Annealing	
Program :	M.S. in Computer Engineering	
Division/Dept :	Electrical and Computer Engineering	
School :	Engineering and Architecture	

Approved/Signed by:

Thesis Advisor	Dr. Iyad Ouaiss	Signatures Redacted
Member	Dr. Zahi Nakad	Signatures Redacted
Member	Dr. Wissam Fawaz	Signatures Redacted

Date: January 5, 2009

Plagiarism Policy Compliance Statement

I certify that I have read and understood LAU's Plagiarism Policy. I understand that failure to comply with this Policy can lead to academic and disciplinary actions against me.

This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that by acknowledging its sources.

Name: Maria Abi Saad

Signature: Signatures Redacted

Date: January 5, 2009

I grant to the LEBANESE AMERCIAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or its students and employees. I further agree that the University may reproduce and provide single copies of the work to the public for the cost of reproduction.

To my parents

Acknowledgment

I would like to thank my advisor Dr. Iyad Ouaiss for his guidance throughout my thesis work. Thanks are also due to Dr. Zahi Nakad and Dr. Wissam Fawaz for being on my thesis committee.

I would like to express my sincere gratitude to the Lebanese American University whose financial support during my graduate studies made it all possible.

Finally, I would like to thank my friends and family for their long and patient support.

Abstract

One of the major enhancements that can be made to the synthesis process is reducing the overall area of a design in order to either decrease the manufacturing costs or introduce more functionality to the design. Optimizing the area of the data path is considered a primary field of research in High-Level Synthesis (HLS). This work proposes an approach to reduce the area by simultaneously tackling the three central tasks of HLS. Scheduling, allocation and binding are performed and the optimal solution based on area reduction is obtained by using simulated annealing with a priority function. The aim of the priority function is to guide the simulated annealing process into finding the best solution while at the same time incurring the least possible execution time. In order to achieve better results than the initial solution, rescheduling, swapping operations between functional units, swapping variables between registers and swapping inputs to functional units are considered in the annealing process. A cost function was devised to evaluate a potential move's success or failure. The simulation environment "Eridanus" was developed in order to support implementation and testing. Several benchmarks were tested and the numerical results consisting of the execution time along with the best solution were recorded to illustrate the performance of the proposed technique. Area reduction was obtained compared to the conventional HLS flow; furthermore, n average substantial reduction in design space exploration time was obtained compared to non-priority based area optimization techniques.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

High-Level Synthesis (HLS) consists of transforming an abstract behavioral description of a digital system into a register-transfer-level design that implements that behavior. Typically, an input file written in a hardware descriptive language is transformed into a data path and a control unit. As designs grew, it became important to automate this task. The tools developed for automation also tried to optimize the design. Optimizing the design typically focuses on three pillars: reducing area, reducing latency and reducing power consumption. As complexity grew, manufacturing costs increased. Hence, optimizing the area of the data path is considered as a primary issue for research.

This work focuses on two issues. The first issue is reducing the area of the data path. Reducing the area is primarily accomplished by reducing the interconnect, represented in this work by the number of multiplexers used. This is accomplished by using simulated annealing to search for a best design cost in terms of area. The neighboring solutions that are potentially used include: rescheduling an operation, changing the register binding combinations, changing the functional unit binding combinations and swapping the input variables of commutative functional units. Area reduction by simulated annealing using these four techniques was previously addressed in the relevant literature. The contribution of this work is reducing the design space exploration time while performing area reduction. This is accomplished by devising a priority-driven simulated annealing approach. The simulated annealing process is guided by a priority function whose components include attributes based on the four techniques mentioned previously. Hence, the priority function takes into account the mobility, the lifetime, the number of

successors and the commutativity of an operation. Thus, simulated annealing in this work is not based on merely selecting a node and a move randomly, but will follow a certain priority to obtain the neighboring solution.

The rest of this paper is organized as follows. Chapter 2 consists of a literature survey. It defines high-level synthesis and explores the main algorithms used in scheduling and binding; it also describes simulated annealing. Chapter 3 introduces Eridanus, the simulation environment that was developed in order to implement and test the techniques used. The methods for designing the software, the supported file formats, the scheduling and binding techniques used, and the VHDL data path and controller code generation are elaborated. Chapter 4 explains in details how simulated annealing was used with a priority function in order to optimize area and reduce the design space exploration time. Chapter 5 lists the results obtained and analyzes their significance. Chapter 6 concludes by summarizing the main achievements and by proposing future work.

Chapter 2: Literature Survey

This chapter introduces the techniques proposed in the literature that are related to high-level synthesis. A general notion of high-level synthesis is presented and its three major tasks: scheduling, allocation and binding are explored. Several scheduling and binding algorithms are discussed by elaborating on their category, their procedure, and their differences. Examples are provided for illustration purposes.

2.1 High-Level Synthesis

High-level synthesis, which is also known as behavioral synthesis, is the process of transforming an abstract behavioral description of a digital system into a register-transfer level design that implements that behavior [3]. Hardware description languages (HDL) are used to describe the behavior of a design and are converted into a control data flow graph (CDFG) for ease of analysis. At the end of the synthesis process, a data path consisting of the storage units, the functional units and the interconnect is generated. A control unit which preserves the correct execution flow of the data path is also generated.

The high-level synthesis process can be decomposed into the following three tasks:

- Scheduling: scheduling is the process of determining the sequence of operation execution without violating any dependency or resource constraints.
- Allocation: allocation is the process of allotting enough components such as functional units, storage elements and interconnection units without violating the resource constraints.

- Binding: binding is the process of assigning operations and variables to functional units and storage elements and connecting these components together to form a well structured data path.

The HAL example in[7] is used throughout this chapter to illustrate the algorithms described. The textual description (or HDL) for HAL is:

```
while (x < a) do
    x1: = x + dx;
    u1: = u – (3 * x * u *dx) – (3 * y * dx);
    y1: = y + (u * dx);
    x : = x1 ;
    u = u1 ;
    y : = y1 ;
endwhile
```

Control Data Flow Graph (CDFG)

As previously mentioned, high-level synthesis reads an HDL description and transforms it into a CDFG. The CDFG is an intermediate form that captures all the control and data flow dependencies of the behavioral high-level description. A CDFG is a directed acyclic graph which can be denoted as G(V, E); where V is a set of nodes and E is a set of edges. Each $v_i \in V$ represents an operation (o_i) in the behavioral description. A directed edge e_{ij} from node v_i to node v_j exists if the node v_i is a predecessor to node v_j. The CDFG for the HAL example is given in Figure 1.

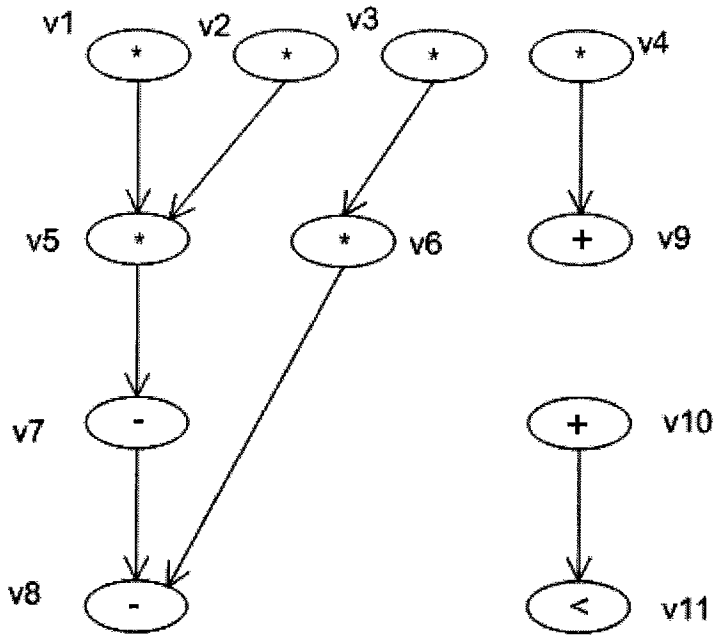

Figure 1: HAL example CDFG

2.2 Scheduling

Scheduling is the process of assigning an order to each operation. This order is referred to as the operation's control step. Each control step can be considered as a sub-graph of the original CDFG discussed above. The assignment of operations into control steps is based primarily on the dependencies between the operations. Hence, in the CDFG of Figure 1, node v6 cannot be scheduled before node v3, since node v6 depends on the output of node v3.

The total number of functional units needed in a design is equivalent to the sum of the maximum number of individual resources within a control step. If more operations are scheduled into each control step, then more functional units will be necessary in the design. This will result in fewer control steps at the expense of an increase in area due to

the increase in functional units. On the other hand, if fewer operations are scheduled within a control step, then more control steps are needed. However, fewer functional units will be required; hence lies the importance of scheduling in high-level synthesis in determining the tradeoff between cost and performance [3].

The scheduling problem can define one of two goals. The first goal is minimizing the number of functional units while keeping a fixed number of control steps. This is known as time-constrained scheduling. The second goal is minimizing the number of control steps for a given design cost consisting of the number of functional and storage units. This is known as resource-constrained scheduling. The most widely used scheduling algorithms can be categorized as shown in Figure 2 [1].

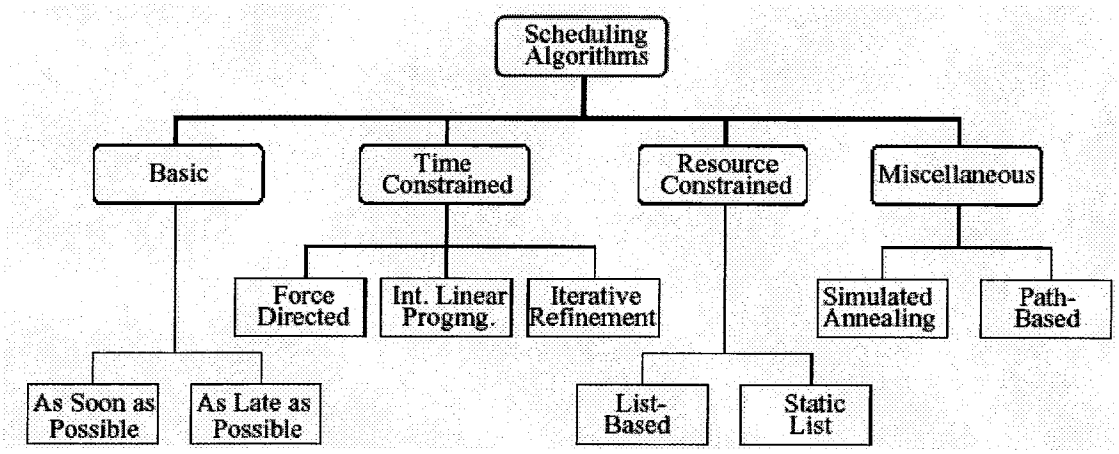

Figure 2: Scheduling algorithms

2.2.1 Basic Scheduling Algorithms

As Soon As Possible (ASAP)

ASAP scheduling consists of assigning the start time of each operation with the least value allowed by the dependencies; hence the name as soon as possible. The algorithm

first schedules the nodes with no predecessors. It then moves down the graph to schedule the other nodes. A node can only execute after its predecessors have finished execution. It is evident that ASAP gives the fastest possible schedule comprising the least possible control steps. However, resource constraints are not considered.

The implementation of the algorithm is given in the pseudo code of Figure 3. G_s represents the graph to be scheduled. E represents the set of edges. V represents the set of vertices or nodes. tj represents the start time of vertex vi; and dj represents the duration of vertex vj [4].

```
ASAP (Gs(V, E)) {
   Schedule v0 by setting t0(s) = 1;
   Repeat {
        Select a vertex vi whose predecessors are all scheduled;
        Select vi by setting ti(s) = max tj(s) + dj; where j: (vj, vi) Є E
             }
    Until (vn is scheduled);
    Return t(s);
             }
```

Figure 3: ASAP algorithm

The ASAP schedule of the HAL example can be found in Figure 4.

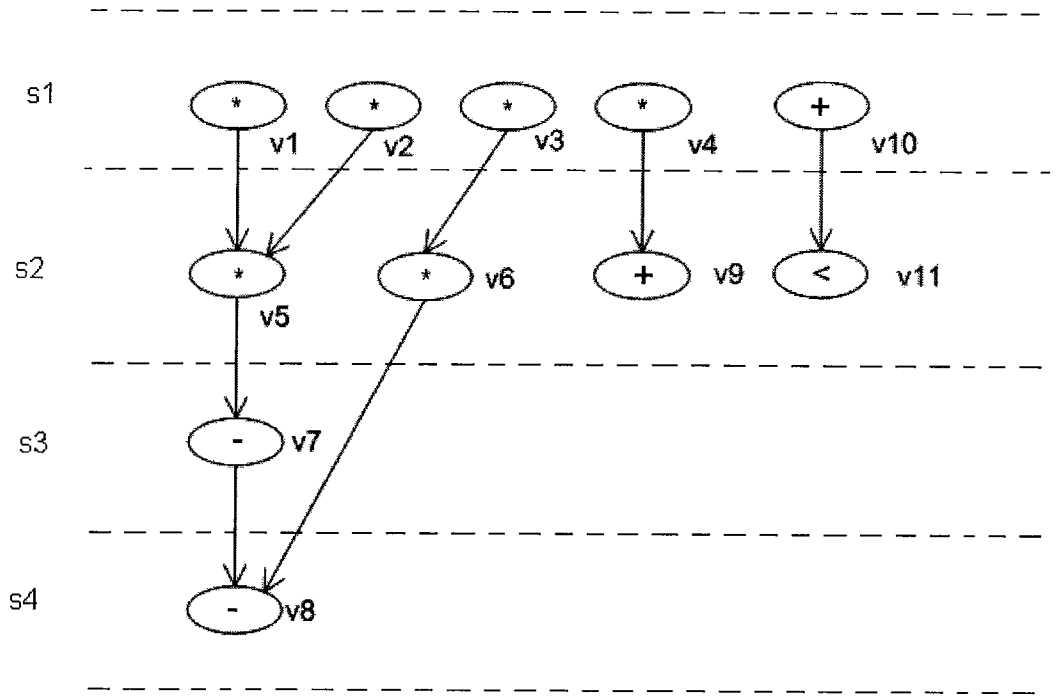

Figure 4: ASAP schedule for the HAL example

As Late As Possible (ALAP)

ALAP scheduling works in a similar manner as ASAP scheduling, but it starts at the bottom of the graph and proceeds upwards. The ALAP schedule defines the latest control step each node can be scheduled at without prolonging the maximum number of execution steps found in the ASAP schedule. This is accomplished by first setting the control step of the nodes with no successors to the ASAP schedule and then moving up the graph and decreasing the level of the predecessor nodes.

The implementation of the ALAP algorithm is given in the pseudo-code of Figure 5. The notations are the same as those for the ASAP algorithm; however, λ represents the ASAP number of control steps, i.e. the total duration of the graph [4].

```
ALAP (Gs(V, E), λ) {
    Schedule vn by setting tn(L) = λ + 1;
    Repeat {
        Select a vertex vi whose successors are all scheduled;
        Select vi by setting ti(L) = min tj(L) - di; where j: (vi, vj) Є E
            }
    Until (v0 is scheduled);
    Return t(L);
            }
```

Figure 5: ALAP algorithm

The ALAP schedule of the HAL example is found in Figure 6.

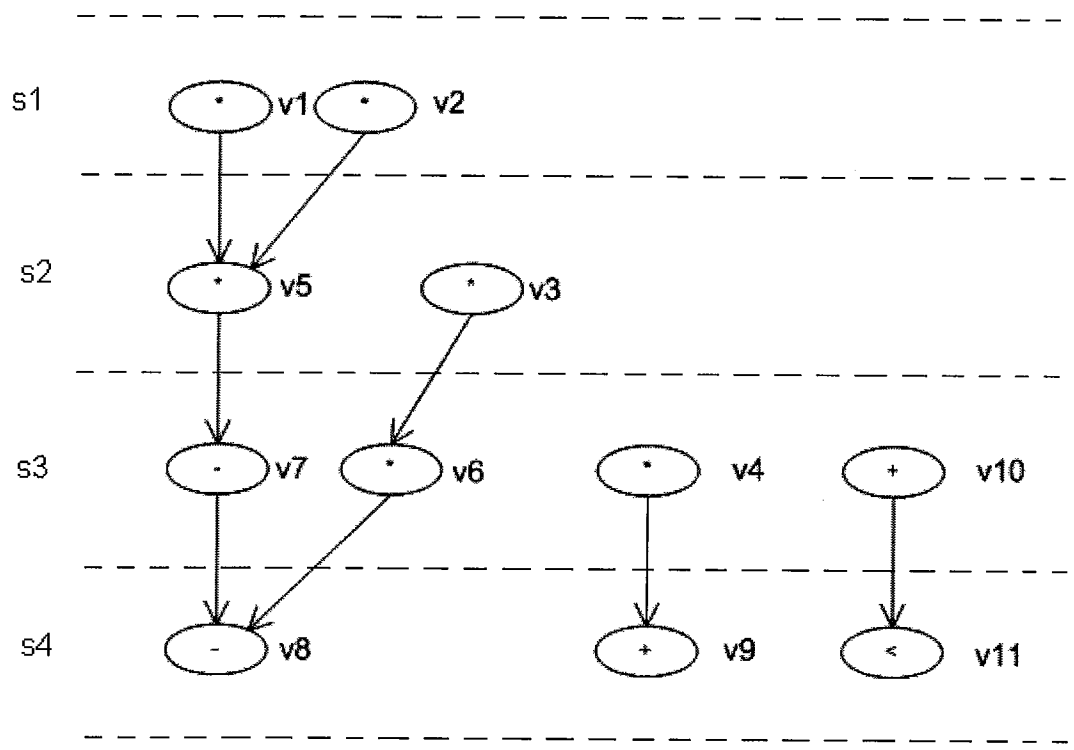

Figure 6: ALAP schedule for the HAL example

2.2.2 Time Constrained Scheduling Algorithms

Time-constrained scheduling is especially important in design applications that represent real-time systems. An example of a real-time system is a Digital Signal Processing

(DSP) system. Since the sampling rate in such a system is fixed, the main objective is to minimize the cost of the hardware. The sampling rate can be expressed in terms of the number of control steps [3].

Time-constrained scheduling algorithms can use three techniques:

- mathematical programming, such as integer linear programming
- constructive heuristics, such as force-directed scheduling
- iterative refinement

An example of each technique will be considered next.

Integer Linear Programming Method

The Integer Linear Programming (ILP) method is one of the most popular techniques in mathematical programming. It finds an optimal schedule by using a branch and bound search algorithm that makes use of backtracking. Decisions made in some earlier stages are revisited as the search process proceeds. A simplified form of ILP consists of first calculating the mobility range of each operation as suggested by the following formula:

$M = \{S_j \mid E_k \leq j \leq L_k\}$; where Ek is the ASAP value and Lk is the ALAP value.

ILP makes use of the following equations:

$$Minimize(\sum_{k=1}^{n}(Ck * Nk))$$

$$\sum_{Ei \leq j \leq Li} xi, j = 1$$

$$\forall i, 1 \leq i \leq n$$

where $1 \leq i \leq n$ represents the number of operations, $1 \leq k \leq m$ represent the operation types available, and N_k is the number of functional units of operation type k and C_k is the

cost of each functional unit. Each $x_{i,j}$ is equal to one if the operation "i" is scheduled in control step j and is equal to zero otherwise.

To impose resource and data dependency constraints, ILP makes use of the following equations:

$$\sum_{i=1}^{n} xi, j \leq Ni$$

$$((q * xj, q) - (p * xi, p)) \leq -1, p \prec q$$

where p is the control step assigned to the xi operation and q is the control steps assigned to the operation x_j.

As the number of control steps increases, ILP formulation will rapidly increase. If the number of control steps is incremented by one, this will incur "n" additional "x" variables. Hence the execution time will also increase rapidly. This is why the ILP algorithm is only suitable for problems of very small scale [6].

Force Directed Scheduling

The Force Directed Scheduling (FDS) algorithm is a very popular time-constrained based constructive heuristic. FDS aims at reducing the total number of functional units. This is accomplished by uniformly distributing the operations of the same type into all the available steps. Distributing operations uniformly guarantees that functional units allocated to execute operations in one control step will be efficiently used in the other control steps. This will lead to a high functional unit utilization rate.

The pseudo code for FDS is given and explained briefly below:

Repeat

- Evaluate time frames: find the ASAP and ALAP schedules
- Update the distribution graphs
- Calculate the self forces
- Add the predecessors and successors forces
- Schedule the operation with the least self force

Until all the operations are scheduled

The ASAP and ALAP schedules are calculated to obtain the time frames for the operations: L_i and E_i.

For each type of operations, a distribution graph is constructed to determine the potential control steps for each operation. If an operation can be executed in k steps, then 1/k is added to each of these k steps.

The force for each operator is calculated by using the formula:

$$DG(i) = \sum_{OpnType} \Pr ob(Opn, i)$$

Prob is the probability of an operation to occur in control step "i".

The probability is calculated using the formula:

Prob = 1/ Mobility Range where Mobility Range = $L_i - E_i + 1$.

Each operation's force can be calculated as follows:

Force(i) = DG(i) * x(i); where x(i) is the probability that an operation will occur in control step "i" and DG was defined above.

The force is positive if the operation is being added to control step "i" and is negative if it is being removed from control step "i".

The total force of an operation is obtained by adding the forces of the parents and children nodes along with that operation's self force.

After all the operations' total forces are calculated, the operation with the least force is assigned to the control step under study. The algorithm is done when all the nodes are scheduled [7].

Iterative Refinement Method

The Iterative Refinement (IR) method reschedules one operation at a time. An initial schedule is considered. Each operation is scheduled into either an earlier or a later control step, but conserving data dependencies. A random move is chosen, performed and then temporarily locked in that position. Other moves are made until all the operations are locked. The move that gave the best reduction in cost is selected. All the moves in the sequence up until the chosen move are permanently moved. All the operations are unlocked and the entire process is repeated considering the new schedule. In IR, the quality of the result is dependent on the initial solution [8].

A comparison of the time-constrained scheduling algorithms is given below [1].

Table 1: Comparison of time-constrained scheduling algorithms

	ILP	*FDS*	*IR*
Computational complexity	Very high	Low	High
Quality of schedule	Optimal	Opt/sub optimal	Near optimal
Space complexity	Medium	Low	Very high
Input problem size	Small	Any size	Can't be large
Execution speed	Low	High	Medium
Technique used	Math progression	Constructive	Refinement
Popularity	Least used	Most used	Medium

2.2.3 Resource Constrained Scheduling Algorithms

Resource-constrained scheduling is used in problems where the application is limited by the silicon area. The constraint is based on the resource bag comprising of the number of functional units available. The schedule is gradually constructed, one operation at a time. Data dependencies and resource constraints are maintained throughout scheduling. In what follows, two resource-constrained scheduling algorithms are described: the list-based scheduling method and the static-list scheduling method.

List Based Scheduling Method

List-based scheduling is similar to the ASAP scheduling; however, it takes into consideration the resource constraints while scheduling the operations. List-based scheduling is based on a priority list of ready nodes that is dynamically updated. The list of ready nodes represents the nodes whose predecessors have already been scheduled. The operations in the beginning of the ready list are scheduled in an iteration until all the resources in that state are used. If there are conflicts between ready operations, then the priority function will break the ties. For example, if there are three ready addition operations and the resource bag contains only two adders, then the two operations with higher priority will be scheduled first and the operation with lowest priority will be deferred to the next control step. Once an operation is scheduled, then it might make some other non ready operation eligible to be included in the ready list according to the priority function used.

The quality of the list-based scheduling method depends on the quality of the priority function. Hence, it is important to properly define the priority function. The priority function can be based on several factors. A simple priority function can be based on mobility. Mobility represents the slack that an operation has. The priority function can be proportional to the mobility or have an inverse proportional relation to the mobility. A smaller mobility indicates a higher need to schedule an operation since the schedule will run out of other control steps earlier. Another priority function could use the length of the longest path from the node under scheduling to a node with no immediate successors. Another priority function could use the number of immediate successor nodes as the primary sorting key [3].

Static-List Scheduling Method

The static list scheduling method differs from the list-based scheduling method in that it does not dynamically maintain a priority list. The priority list of the nodes is constructed statically once at the beginning of the algorithm and is used throughout without being updated. Hence, static list scheduling has less time and space complexity.

The algorithm proceeds as follows:

- Find the ASAP and ALAP control steps:

  In order to schedule using List, first the ASAP and ALAP schedules need to be obtained. This is used to sort the nodes.

- Sort the nodes:

  Nodes, or operations, are sorted according to their ASAP and ALAP values. The primary sorting key is the ALAP values. Nodes are first sorted in an increasing order of their ALAP values. Then, these sorted nodes are sorted in a decreasing

order in which their ASAP value is the secondary key. Ties between nodes are broken arbitrarily. Once the nodes are sorted, they are assigned priorities, where the lowest priority corresponds to the first node in the list. Table 2 shows the ASAP, ALAP and priority rankings of the HAL example. It can be noted that the operations v9 and v11 have similar ASAP and ALAP values, but their priorities are different in order to be able to schedule.

Table 2: Priority list for static list scheduling

node	*v8*	*v 9*	*v 11*	*v 7*	*v 6*	*v 4*	*v 10*	*v 5*	*v 3*	*v 1*	*v 2*
ALAP	1	1	1	2	2	2	2	3	3	4	4
ASAP	4	2	2	3	2	1	1	2	1	1	1
priority	1	2	3	4	5	6	7	8	9	10	11

- Schedule the operations:

  Once the operations are sorted and their priorities are determined, they can be scheduled sequentially. Nodes with higher priorities are considered for scheduling first. Since list scheduling is a resource constrained scheduling algorithm, then the resource bag that will be used highly affects the schedule. For the HAL graph in Figure 7, a resource bag containing two multipliers, one adder, one subtractor and one less than operator will be used.

Resource bag
* : 2
+ : 1
- : 1
< : 1

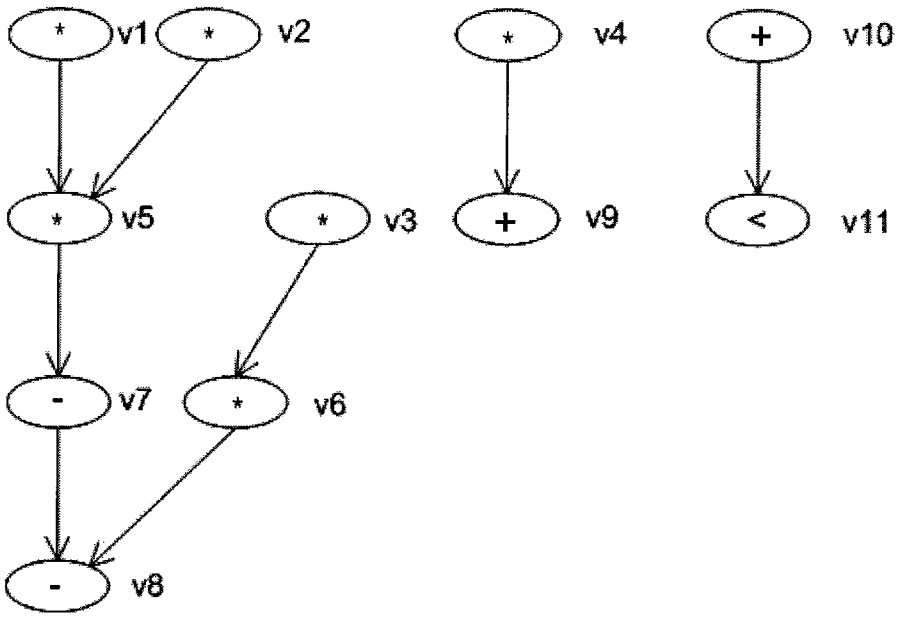

Figure 7: HAL DFG

Operations are scheduled as early as possible. The only constraints are the resource constraints and the data dependencies. The node with highest priority is scheduled first and assigned to the first control step. The number of operators for that node's type is decremented by one in that control step. The second node is visited. If there is an available resource for it in that control step and there are no dependencies, then it can be scheduled in that control step, otherwise, it is scheduled at a later control step.

In the HAL example, operation v2 is considered first and is scheduled in the first control step. Since v2 is a multiplication, then only one more operation that represents a multiplication can be scheduled into that control step. This is because the resource bag contains only two multipliers. The next operation is v1 and can also be scheduled into the first control step. The next operation is v3. it does not have any data dependencies and can therefore be scheduled into the first

control step. However, there are no more free multipliers available. Therefore, v3 is deferred to control step 2. Node v5 is scheduled in control step 2 as well. Node v10 is the next node to be considered. Since it has no predecessors and there is a free adder in control step 1, v10 can be scheduled into the first control step. It is worth noting here that even though v10 has less priority than v3, it was scheduled at an earlier control step. This is due to the resource constraint. An adder was available in control step one, but a third multiplier was not found in the resource bag. The rest of the nodes are scheduled similarly and the final schedule can be found in Figure 8.

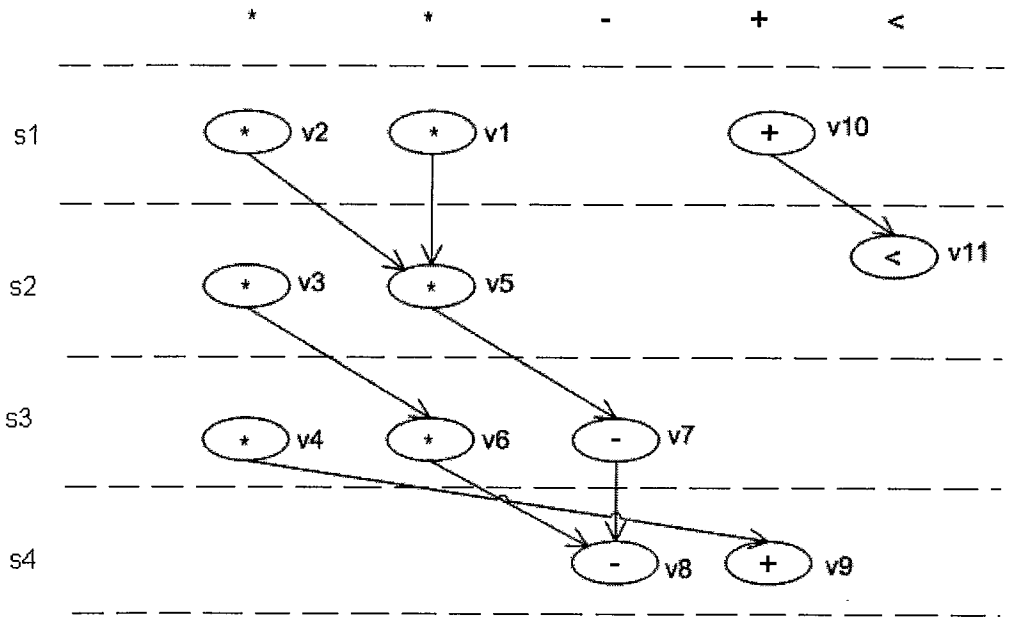

Figure 8: List schedule for HAL example

2.3 Binding

Binding is the process of assigning operations and variables to functional units and storage elements and connecting these components together to form a well-structured data path. The data path is obtained from the register transfers assigned to each control

step. The data path can be considered as a netlist made up of three RT components: functional, storage and interconnection. The functional units are responsible for executing the operations in the HDL. Storage units are responsible for holding the values of the variables throughout execution. Interconnection units are responsible for transporting the data between the functional units and the storage units.

The number and type of components to be used in a design is originally specified in the resource bag. The list scheduling algorithm will take into account the resource bag constraints and will hence allocate the minimum number of resources required in each control step. What remains to be done is to bind the variables and operations in the scheduled graph to the functional units and to the storage units.

Binding is accomplished by decomposing each task and solving each part individually. Hence, assigning variables to registers might be first completed before assigning any operation to a functional unit is done.

The problem of binding can be solved by using the concepts of graph theory. The two binding techniques: clique partitioning and left edge algorithm, are based on graph theory and are explored.

2.3.1 Clique Partitioning

Clique partitioning is a heuristic proposed by [9]. It is based on converting the modules' lifetimes into a compatibility graph. Each node in the graph represents an element to be allocated. An edge is added to the graph between two nodes if the two variables

represented by these two nodes can be bound to the same module. Each fully connected set of nodes will then represent a clique. The total number of cliques obtained will indicate the total number of modules needed for the design.

Clique partitioning can be used for both functional unit binding and for register binding. However, the example given in this section will be based on register binding.

Variable Lifetimes

The lifetime of a variable represents the time that the variable is used or is saved to be used later. A variable's lifetime can be determined by traversing the scheduled graph and searching for the first occurrence and for the last occurrence of that variable. The first occurrence is when that variable was first written. The last occurrence is when that variable was last read in the schedule.

Compatibility Graph

Two variables are said to be incompatible if they overlap at a clock boundary or have intersecting lifetimes. Two incompatible variables cannot share the same register. An incompatibility graph is constructed wherein the nodes represent the variables and an edge connecting two nodes is included if the two nodes represent incompatible variables. A compatibility graph is derived from this incompatibility graph. Connected nodes in a compatibility graph denote that the corresponding variables can be bound to the same register.

Cliques

A complete graph is a graph in which every pair of vertices is connected. A clique derived from the compatibility graph is a complete sub-graph of the compatibility graph. The clique size is defined as the number of vertices forming the clique. Clique partitioning consists of partitioning the graph into a disjoint set of cliques. A maximum clique partition consists of partitioning the compatibility graph into the minimum number of cliques. In order to obtain the maximum clique partition, the clique partitioning algorithm makes use of the "common neighbor" and the "non-common edge" terms. A node is considered to be a common neighbor of a sub-graph if it is not contained in the sub-graph and has an edge with every node of the sub-graph. If a node is not a common neighbor of a sub-graph but has an edge with at least one of the nodes contained in the sub-graph, that edge is labeled a non-common edge.

The clique partitioning algorithm proceeds as follows:

- Locate the edge having the most number of common edges
- Merge the two nodes connected by that edge into a clique node
- Delete all edges that connect the two nodes
- Add edges between the super node and all the other clique nodes that are common neighbors to the two nodes

This process is repeated until there are no more edges. The final number of cliques will give the total number of registers required to store the variables.

2.3.2 Left Edge

The left edge algorithm is mainly used to optimize channel routing for physical design automation. The channel width depends on the number of horizontal tracks. Channel routing aims to reduce the number of horizontal tracks that connect points on the channel boundary.

The left edge algorithm is used in high-level synthesis to solve the register allocation problem. The solution space is mapped to the channel routing problem and the minimum number of registers required is obtained. The variable lifetimes correspond to the horizontal wire segments and the number of registers corresponds to the number of wiring tracks.

The left edge algorithm proceeds in the following manner:

- Calculate the start and end times for all the variables:

  The start time corresponds to the control step in which the variable is first encountered. The end time corresponds to the control step in which the variable is last written. In the HAL example, v1 and v2 are decomposed into v1 and v1' and v2 and v2'. This is done to obtain a better packing density [3]. For example, the variable v6 has its start time in control step 0 and its end time in control step 2. It is needless to say that the start and end times will depend on the scheduling algorithm used.

- Sort the variables:

  Variables are sorted according to their start and end times. First, variables are sorted in increasing order with their start times being the primary key. Then,

these sorted variables are sorted according to their decreasing end times. Figure 9 shows the sorted variables and their lifetimes within the control steps.

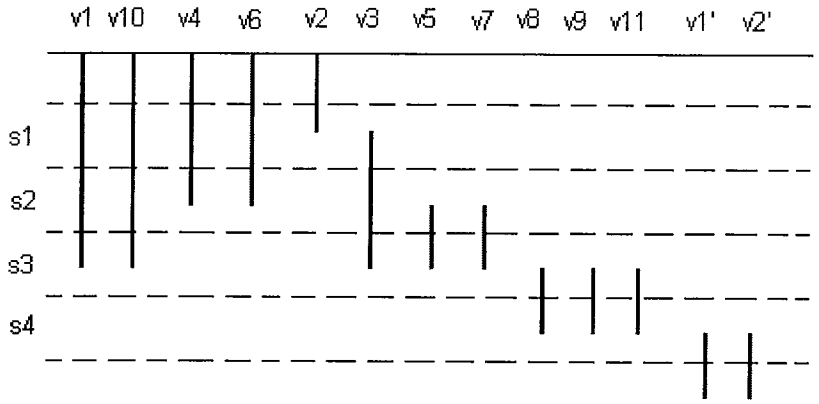

Figure 9: Sorted variable lifetime intervals

- Allocate the variables to registers:

  The sorted registers are traversed in order. The first variable is assigned to a register and is removed from the sorted list of variables. A search is made among the rest of the variables following the sorted order. If a variable's lifetime does not overlap with the initial variable, then the new variable is mapped to the same register and is removed from the list and the register's lifetime is extended. Otherwise, the variable is skipped. Two lifetimes are said to be overlapping if the start time of the next variable is less than or equal to the end time of the first. The list is traversed until there are no more variables that can be mapped to the current register. While there are still elements in the list, a new register is allocated each time the end of the list is reached. The algorithm ends when the list becomes empty.

For the example above, v1 is considered first and is assigned to register 1. The next node will be v10. However, v10 cannot be assigned to register 1 as well since v1 and v10 have overlapping lifetimes. So v10 is skipped. The next node to consider is v4. As well, v1 and v4 have overlapping lifetimes. So v4 is also skipped. The same applies to each of v6, v2, v3, v5 and v7. Node v8 does not conflict with v1 and can therefore be added to the same register. Now the register's lifetime is extended to s4. But v1' can be added to it. No more variables can be mapped to register 1 without conflicting with a variable already mapped to register 1. Therefore, a new register is allocated and the same procedure follows. At the end of the algorithm, five registers were needed and their bound variables are shown in Figure 10.

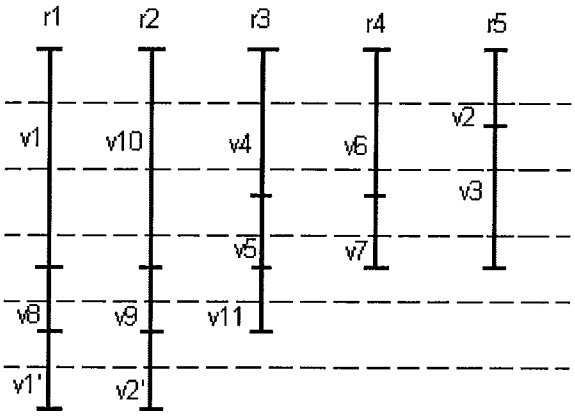

Figure 10: Left edge register allocation result

2.4 Simulated Annealing

Statistical mechanics studies "the behavior of systems with many degrees of freedom in thermal equilibrium at a finite temperature" [5]. There exists a connection between statistical mechanics and combinational optimization. From this similarity emerged the

analogy between annealing solids and optimization of large and complex systems, such as high-level synthesis. Hence, simulated annealing was successful in optimizing the high-level synthesis problem of scheduling and binding [5].

The objective of the simulated annealing is to reach a global minimum. It starts with an initial solution to the problem and iteratively tries to find a best solution. It is also based on an initial high temperature that is decreased with the number of iterations, until a final predefined temperature is reached or a maximum number of trials is attained. Other parameters of the simulated annealing process include the number of trials, the trial factor and the temperature factor. The trial factor is a number greater than one, and will increase the number of trials within a given temperature. It will also increase the outer number of execution loops. The temperature factor is a number slightly less than one and reduces the initial high temperature. The values of these parameters are obtained from experimentation.

An initial solution is first devised and it is considered as the best solution so far. Its cost is also considered as the best cost. A new solution is generated and the new cost is calculated. For now, the current solution is the initial solution as well. If the new cost is less than the current cost, then the current solution will be the new solution. In the case that the new cost is less than the best cost, then the best solution is the new solution and the best cost is updated accordingly. If the new cost is greater than the current cost, then the current solution is updated if it conforms to a certain function discussed below. Otherwise, the new solution is rejected.

The pseudo-code for the simulated annealing algorithm is given below.

```
current solution = initial solution
current cost = initial cost
best solution = current solution
best cost = current cost

    while (time ≤ maximumTime)
    {
        temperature = initialTemperature;
        time = 0.00;

        while (numberOfTrials > 0)
        {
            new solution = neighbor of current solution
            new cost = cost of new solution
            costDifference = newCost - currentCost;

            if(costDifference < 0)
                {
                    if(newCost < bestCost)
                    {
                        best solution = new solution
                        best cost = new cost
                    }
                        current solution = new solution
                        current cost = new cost
                }

            else if (acceptSolution(costDifference,temperature))
                {
                    current solution = new solution
```

```
                    current cost = new cost
                }
                else
                {
                        rollback
                    }
                    numberOfTrials = numberOfTrials - 1.00;

                }
                time = time + numberOfTrials;
                numberOfTrials = numberOfTrials * trialFactor;
                temperature = temperature * temperatureFactor;
            }

            return the best solution and best cost
        }
```

The acceptSolution function determines whether a solution will be accepted if its cost is greater than the best cost. This is necessary so that the simulated annealing process does not get stuck in local minima. This function is defined by calculating the Boltzmann probability: $e^{-\Delta cost/T}$, where $\Delta cost$ represents the difference in cost and T represents the current temperature. If this probability is greater than a random number between zero and one, the solution is accepted, otherwise it is rejected.

Chapter 3: Eridanus

In an attempt to implement all the concepts mentioned in the previous chapter, Eridanus was implemented as part of this work. Eridanus is a high-level synthesis tool that was developed with the aim of automating the entire synthesis process, starting from the reading of input and resource files to obtaining a textual representation of the data path and the controller VHDL codes. In order to do so, Eridanus performs several tasks that can be divided into: scheduling, binding and allocation, floor plan design, data path generation and controller generation.

What distinguishes Eridanus from other industrial and educational tools such as Quartus is its ability to cater for several user-specified parameters. The flexibility that Eridanus offers lies in allowing the user to select the resource bag, the scheduling method and the binding algorithm to be used. This implies that a user can choose several combinations of algorithms in order to select the one that will return an optimal result. Of course, the term "optimal" might vary among users since some will consider an output of minimal area to be optimal while others might consider less scheduling levels to be optimal. The choice of optimality will depend on the application needed and Eridanus can cater for several combinations that will lead to effective testing; hence, its importance. What further distinguishes Eridanus from other tools is that it allows users to write their own scheduling algorithm and easily incorporate it in the tool.

To summarize the flow of operations in Eridanus, the user first specifies an input file and a resource bag file. Once the parsing of these files is done, a dependency graph is

generated. The details of parsing and of the dependency graph can be found in [2]. The user then selects a scheduling algorithm which reorders the dependency graph according to the scheduling method chosen and a binding algorithm in order to allocate and bind registers and functional units. Once user selection is determined, the tool will generate a printable floor plan design that could be saved and the data path and controller VHDL codes that will be inputted to Quartus[1] to obtain a waveform of the results.

In order to thoroughly discuss the capabilities of this tool, Chapter 4 has been divided into several sections. The first section will discuss the file formats that Eridanus supports. The second section will list the scheduling methods that can be used. The third section delves into the details of the binding algorithms offered. The fourth section discusses the floor plan layout. The data path and controller VHDL code generation is addressed in the fifth section.

3.1 File Formats

The file formats to be discussed in this section are the input file and the resource file. The input file contains the sequence of operations that are to be performed and is written in a Pascal-like language. The resource file contains the maximum number of resources that can be used at each control step.

3.1.1 Input File

The input file is written in a Pascal-like language. The beginning of the file is indicated by the keyword "program". The inputs are then specified and are identified by the

[1] The data path and controller codes were implemented based on Quartus-defined syntax; however, they can be used with any other VHDL simulator.

keyword "in". The input type follows the input declaration. A sample file is shown below. In this example, there are two inputs "a" and "b" and each input is of four bits. Inputs are separated by commas. The declaration of outputs is similar, but instead of using the word "in", the word "out" is used.

The start of the operations is indicated by the "begin" keyword. Every statement ends with a semicolon. The end of the file is indicated by the keyword "end" followed by a period. It is worth mentioning that Eridanus supports the use of if-then-else statements and also nested if-then-else statements. Such blocks are terminated by the keyword "end". The example in Figure 11 illustrates this point and will be used throughout the text for illustration purposes.

```
program
in a, b:std_logic_vector(3 downto 0);

begin
        c:=a+3;
        if (b < 0) then
                d:=c+1;
        else
                d:=c+2;
        end;
e:=d+2;
end.
```

Figure 11: Input file for example 1

Once the input file is read, it is parsed for correct processing. The parser converts the variables and the operations into nodes. These nodes are connected together in a graph via edges that represent a dependency among the nodes. Each input and each operation is assigned a node having a unique ID. Constants are also considered as nodes.

A special kind of node is the "Diverge" node. A diverge node represents the if-condition in an if-then-else statement. In example 1, the node b < 0 is a diverge node. The output of this node is either 0 for false or 1 for true.

Another special kind of node is the "Converge" node. A converge node is generated when the end of an if-then-else statement is encountered. The "converge" node acts as a multiplexer that chooses between two values of the same variable depending on the selector which is the if-condition. The output of a "diverge" node is the selector for the converge node.

The implementation of Eridanus was done so as to implement both clauses that are embedded in the if-then-else segment. Hence, in example 1, both operations d = c + 1 and d = c +2 will be mapped to functional units and both results will be obtained. Once the "converge" node on the variable "d" is scheduled, the correct value of "d" depending on the condition imposed by "b" will be considered and used in following operations.

Further details of parsing and of the dependency graph composition can be found in [2].

3.1.2 Resource File

The resource file contains the number of operators that can be used. Of course, operators can be re-used at different control steps. Hence the number of resources available indicates the number of operators that can be used at each control step. A sample resource bag that was used alongside the input file of example 1 is shown in Figure 12. It should be noted that resource bags need not be unique for different input files.

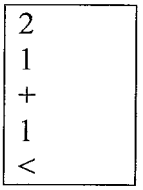

Figure 12: Resource file format for example 1

The resource file starts out by stating the total number of resources available. In the example, there are a total of two resources. It then states the number of a specific resource followed by the resource type on the next line. This is repeated for all the resources available. In the example above, the resource bag contains one adder and one less-than operator.

The resource file format is the same as that used in [1].

3.2 Scheduling Algorithms

Eridanus supports the use of three scheduling algorithms: ASAP, ALAP and list scheduling. It also allows the user to add new scheduling algorithms. ASAP, ALAP and user-defined algorithms were discussed in [2]; however, will be mentioned briefly here. The files used to generate these schedules are the ones pertaining to example 1.

Scheduling will result in a CDFG. The graph generated is a reordered form of the dependency graph obtained after parsing. In the example used, "d" can have either one of two values based on the condition imposed by "b". If "b" is less than zero, "d" is assigned to c + 1. If "b" is greater than or equal to zero, "d" is assigned to c + 2. The "converge" node acts as a multiplexer that chooses between these two values depending on the value of "b". Hence, in example 1, "Converge d" can only be executed after both operations on "d" have been evaluated.

3.2.1 ASAP

In ASAP, each operation is scheduled as early as possible. Figure 13 shows the ASAP schedule of example 1. A successor node can execute only after its parent nodes have finished execution. Hence, "e" cannot be scheduled except after the value of "d" is determined. The file needs four control steps to execute. It is assumed that the inputs are available at time zero. ASAP does not take into account the number of resources available. This is evident in control step two, wherein two additions are performed whereas only one adder is available.

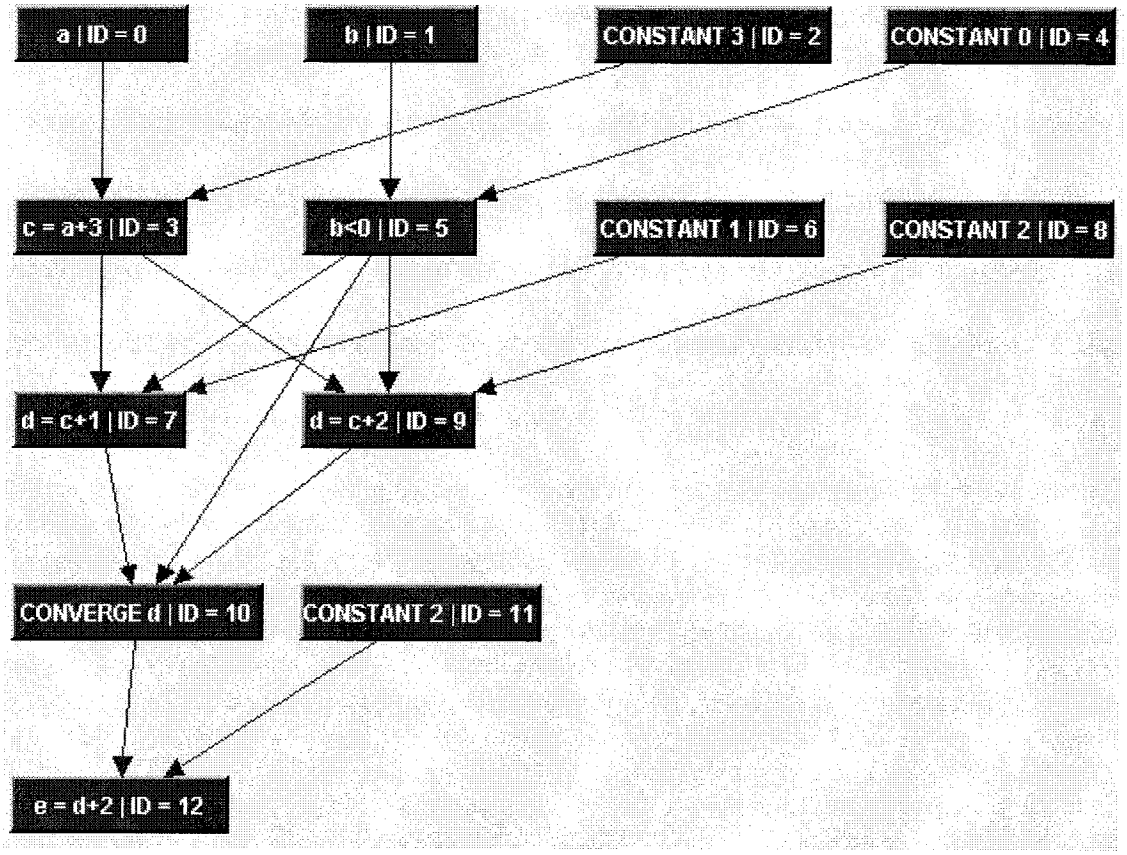

Figure 13: ASAP schedule for example 1

3.2.2 ALAP

In ALAP, each operation is scheduled as late as possible. Figure 14 shows the ALAP schedule of example 1. The schedule is the same as the ASAP schedule since the nodes have no mobility. The file needs four control steps to execute. It is assumed that the inputs are available at time zero. ALAP does not take into account the number of resources available. This is evident in control step two, wherein two additions are performed whereas only one adder is available.

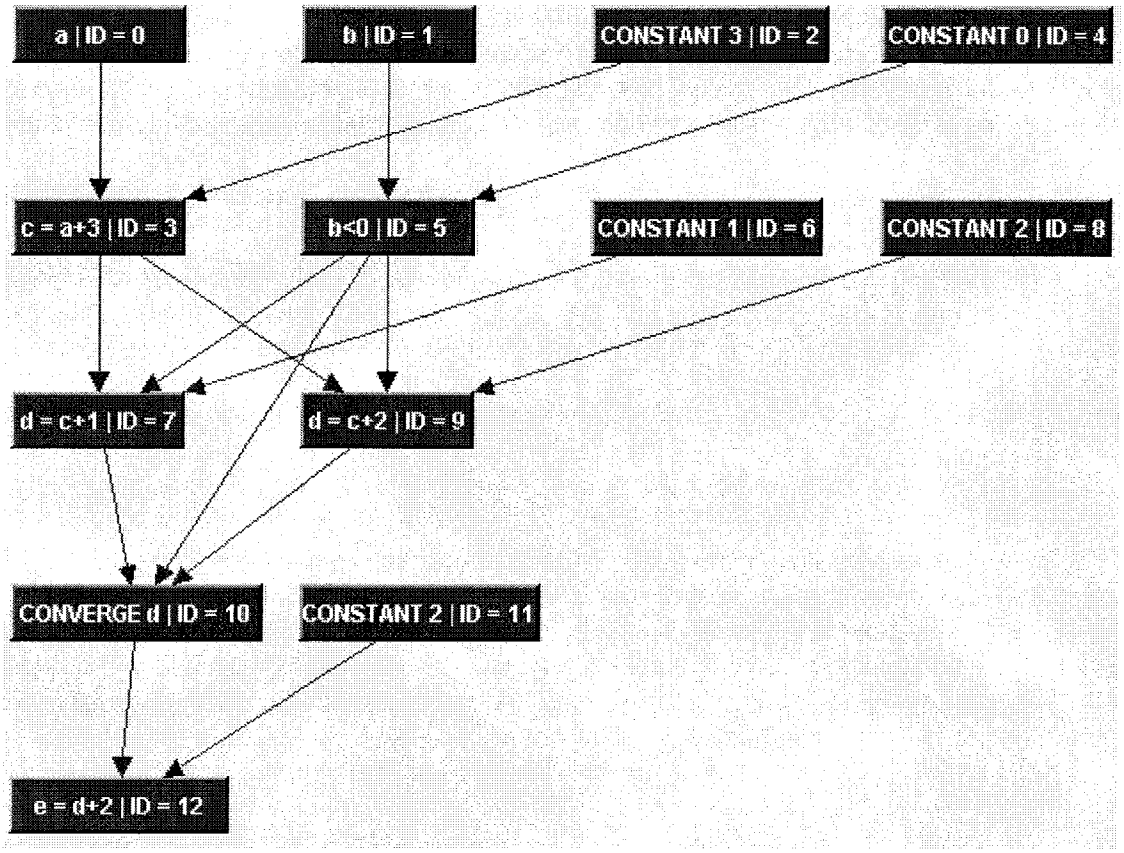

Figure 14: ALAP schedule for example 1

3.2.3 List Scheduling

When scheduled using list scheduling, the previous example takes five control steps to execute instead of four. This is because list scheduling takes into account the number of

resources available. Hence, the operations (d = c +1) and (d = c + 2) can no longer be scheduled at the same control step since only one adder is available in the resource bag chosen.

List scheduling starts out by sorting the nodes based on their ALAP and ASAP values and then assigns priorities to them. Nodes are then scheduled based on their priorities and on the availability of resources. The ALAP and ASAP control steps for the nodes are given in the table below.

Table 3: ASAP and ALAP operations' control steps for example 1

Operation	*ALAP control step*	*ASAP control step*
c = a + 3	1	1
b < 0	1	1
d = c + 1	2	2
d = c + 2	2	2
Converge d	3	3
e = d + 2	4	4

First, nodes are ordered according to their decreasing ALAP values.

Table 4: Order of nodes sorted according to ALAP values

Operation	*ALAP control step*	*Order (ALAP)*
c = a + 3	1	6
b < 0	1	5
d = c + 1	2	3
d = c + 2	2	4
Converge d	3	2
e = d + 2	4	1

Second, the ordered nodes are reordered according to their decreasing ASAP values and their priorities are set such as the node with the highest order has the highest priority and will be scheduled first.

Table 5: Priority of nodes to be scheduled in example 1

Operation	*Order (ALAP)*	*Priority*
c = a + 3	6	1
b < 0	5	2
d = c + 1	3	4
d = c + 2	4	3
Converge d	2	5
e = d + 2	1	6

Lastly, the nodes are scheduled based on their priorities and on the availability of resources at the potential control step. In this example, the first node to be considered is (c = a + 3) which can be scheduled at the first level. The second node is (b < 0). It can also be scheduled at the first level since the resource bag contains a less-than operator. The third node to be scheduled is (d = c + 2). It cannot be scheduled at the first level, since there are not enough adders in the resource bag and since there is a dependency on a node at that same level. Therefore, it is moved to level two. The next node to be considered is (d = c + 1). An attempt is made to schedule it at level two since there is no dependency violation, but since only one adder is available, it is scheduled at a later step. The (converge d) is scheduled at level four since it depends on the nodes (d = c + 1) and (d = c + 2). The last operation is scheduled at level five due to dependency constraints. The List schedule is shown in Figure 15.

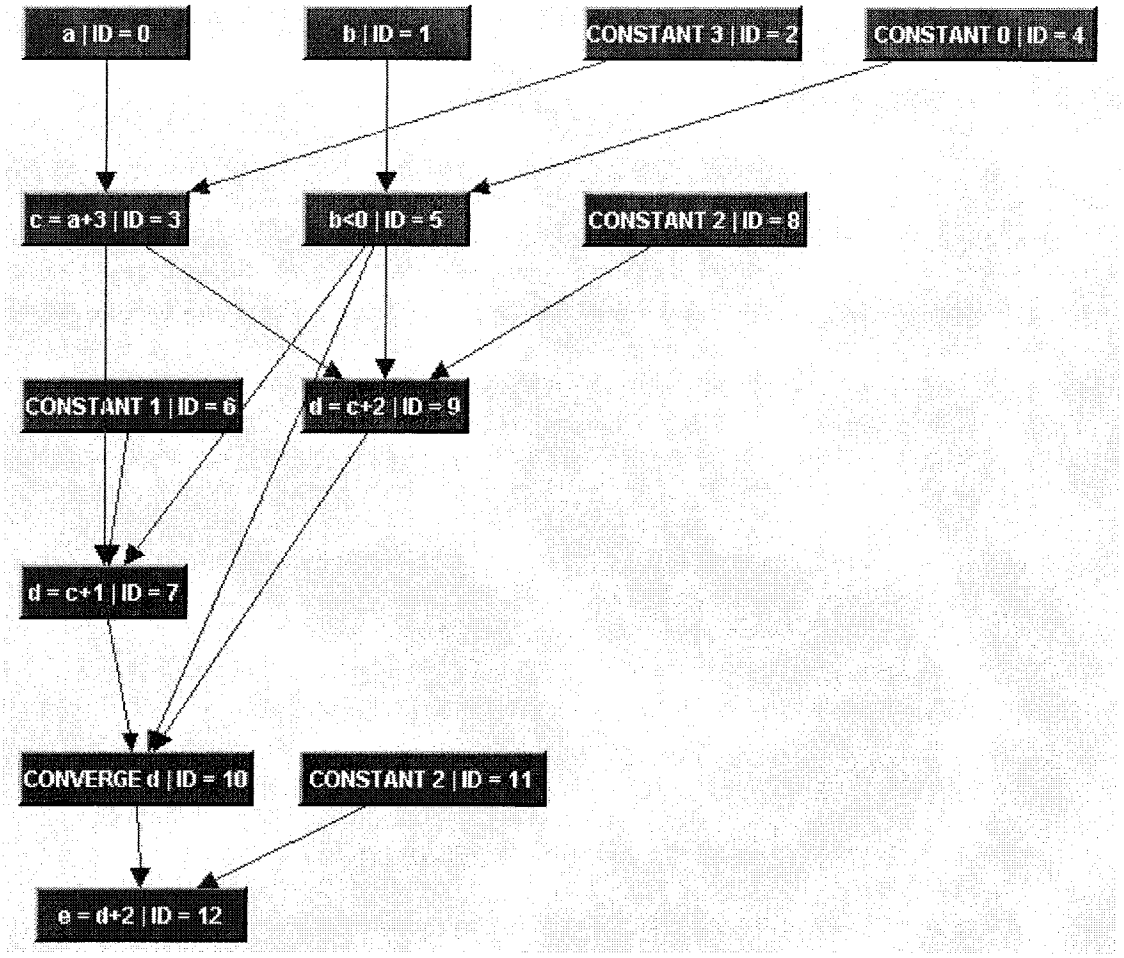

Figure 15: List schedule for example 1

3.3 Binding Algorithms

The binding algorithms were used to bind operations to registers and to bind operations to functional units. The binding algorithms that Eridanus supports are clique partitioning and left edge.

3.3.1 Clique Partitioning

The clique partitioning algorithm was used in order to bind variables to registers and to bind operations to functional units. The implementation of clique partitioning was

detailed in [2]. It is mentioned here as a continuation of the illustration of the previous example.

With respect to the functional unit binding, since the resource file contains only one adder and one less-than operator, then all the addition operations are bound to that adder and the only less-than operation is bound to that comparator.

With respect to register binding, the clique partitioning algorithm identified the usage of three registers. The variables bound to these registers are shown below.

Register 1	*Register 2*	*Register 3*
a	c = a + 3	b
d = c + 2	d = c + 1	b < 0
Converge d		
e = d + 2		

Figure 16: Register allocation for example 1 using clique partitioning

3.3.2 Left Edge

The left edge algorithm was used to bind variables to registers. Input variables are also to be bound to registers. Constants, on the other hand, will not be bound to registers. The binding of variables to functional units was done by using clique partitioning.

The left edge algorithm first starts out by ordering the nodes based on their start and end times. Nodes are sorted based on their increasing start times and then based on their decreasing end times. The ordering is shown in the table below.

Table 6: Start and End times for nodes in example 1

Operation	*Start time*	*End time*	*Order*
a	0	1	1
b	0	1	2
c = a + 3	1	3	4
b < 0	1	4	3
d = c + 2	2	4	5
d = c + 1	3	4	6
Converge d	4	5	7
e = d + 2	5	6	8

Variables are bound to registers if their lifetimes do not overlap. The algorithm assigns the first node to a register and walks down the ordered list. If the next node's start time does not conflict with the previous node's end time, the two nodes can be bound to the same register. Otherwise, a new register is allocated and the latter node is bound to it. In the previous example, the left edge algorithm identified that three registers were needed. The distribution of variables in registers is shown below. The same number of registers was obtained using clique partitioning, however, the distribution of variables within the registers differed.

Register 1	*Register 2*	*Register 3*
a	b	d = c + 2
b < 0	c = a + 3	
Converge d	d = c + 1	
e = d + 2		

Figure 17: Register allocation for example 1 using left edge

3.4 Floor plan

After performing scheduling and binding for registers and functional units, Eridanus will perform the interconnect binding. Interconnect represents the wires and the multiplexers that connect the different components together. The multiplexers can be divided into two levels. Level-one multiplexers connect the outputs of the functional units to the inputs of the registers. Level-two multiplexers connect the outputs of the registers to the inputs of the functional units. Once interconnect assignment is done, Eridanus generates a visual representation of the floor plan corresponding to the design under the selected scheduling and binding methods. The floor plan for example 1 is given in Figure 18.

The floor plan can be considered to be composed of four horizontal layers: level-one multiplexers, registers, level-two multiplexers and functional units. Each component's bit width is also shown within the component's geometrical representation. An operations table appears next to each component. It contains the operations that are bound to that component. For example, the register with ID = 1 holds the operations: d = c + 1 and c = a + 3. These operations tables can be minimized in order not to clutter the design view. Inputs that seem to float represent constants. The floor plan shown below was a result of scheduling example 1 using the list schedule and the binding of both registers and functional units was done using clique partitioning.

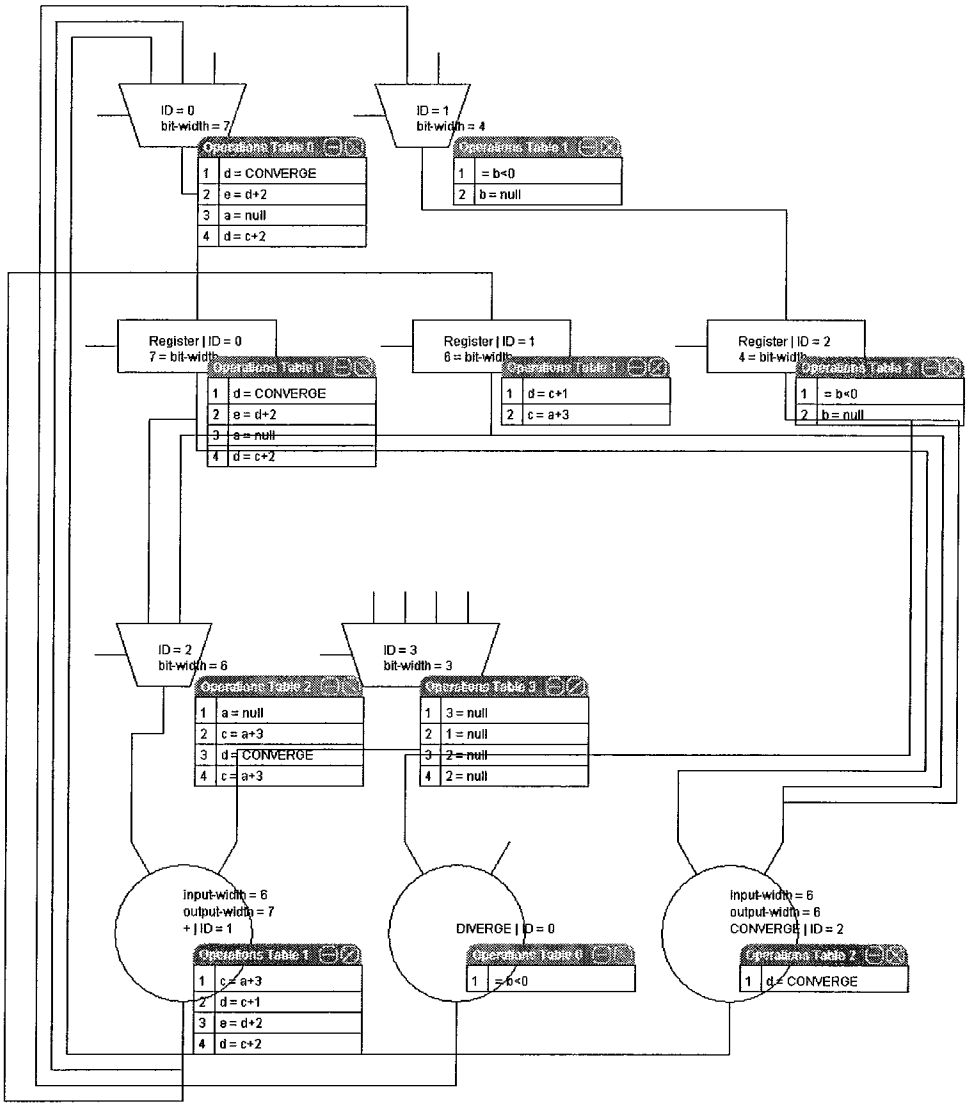

Figure 18: Floor plan for example 1

3.5 VHDL Code Generation

In addition to scheduling, binding and floor plan representation, Eridanus generates the VHDL data path and controller codes. The data path and controller generation is based on the floor plan as well as on the dependency graph.

3.5.1 Data Path

The floor plan mentioned above is merely a visual representation of the data path. The VHDL code for the data path consists of first declaring the entity by determining the input and output signals. The architecture section declares the components used in the design. Components are named following the convention: component type_number of bits for functional units and registers. For multiplexers, the naming also includes the number of inputs. It should be noted that to be able to correctly compile the data path, the components that are declared should be implemented and compiled first. The next part of the architecture is to assign signals. Signals were used in order to correctly propagate bitwidth and to create wrapper signals that will translate signal width to the correct number of bits. Constants are also declared and their binary values are hard coded. The data path begins by connecting the signals to the components' inputs and outputs. The final output is also assigned. The data path code that was generated for example 1 is found in the Appendix.

3.5.2 Controller

The inputs to the controller are the clock and the reset signal. The output signals are the multiplexer and register strobes that are connected to the data path. When reset is high, all the registers' enable signals are disabled.

In order to generate the controller code for the data path design, the level-one multiplexers, the level-two multiplexers and the registers' enable signals need to be set accordingly. The controller will make use of two processes: a state process dependent on reset and on the clock and used for changing the state; and an output process dependent

on the state and used for setting the output within each state. The state process works on the falling edge of the clock. As shown in Figure 19, the high and the low levels of the clock are not equally distributed. Since all the output strobes are set at the falling edge of the clock, the fall time of the clock should be greater than the rise time to be able to accommodate for the delays generated by the components. Registers are enabled at the rising edge of the clock. Since register strobes are determined at the falling edge of the clock, the register enables are effectively taken into account at the rising edge of the next clock.

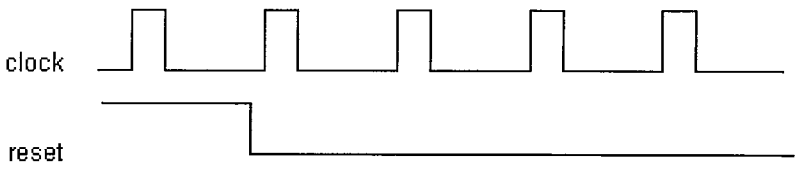

Figure 19: Clock and reset signals

The VHDL controller code for example 1 is given in the Appendix. Each state other than the start state corresponds to a control step and is expanded in what follows.

Start
This is the first state in which reset in enabled and all register strobes are disabled.

reg_0strobe <= '0'	Register 0 is disabled
reg_1strobe <= '0'	Register 1 is disabled
reg_2strobe <= '0'	Register 2 is disabled

State 0

This is the state that will initialize the inputs.

mux_level1_0strobe <= "10"	"a" is selected from the mux

mux_level1_1strobe <= "1"	"b" is selected from the mux
reg_0strobe <= '1'	Register 0 contains "a" and is enabled to be processed at the next rising edge
reg_2strobe <= '1'	Register 2 contains "b" and is enabled
reg_1strobe <= '0'	Register 1 is disabled

State 1

This is the state that will perform b < 0 and c = a +3.

mux_level1_1strobe <= "0"	b < 0 which is the output of the less-than operator is sent to the mux
mux_level2_2strobe <= "0"	Mux will send out value of "a"
mux_level2_3strobe <= "00"	Mux will send out the constant "3"
reg_1strobe <= '1'	Register 1 will contain the value of c = a + 3
reg_2strobe <= '1'	Register 2 will contain the value of b < 0
reg_0strobe <= '0'	Register 0 is disabled

State 2

This is the state that will perform d = c + 2.

mux_level1_0strobe <= "01"	d = c + 2 which is the output of the adder is sent to the mux
mux_level2_2strobe <= "1"	Mux will contain the value from register 1 (which is c)
mux_level2_3strobe <= "11"	Mux will send out the constant "2"
reg_0strobe <= '1'	Register 0 will contain the value of d = c + 2
reg_1strobe <= '0'	Register 1 is disabled
reg_2strobe <= '0'	Register 2 is disabled

State 3

This is the state that will perform d = c + 1.

mux_level2_2strobe <= "1"	Mux will contain the value from register 1 (which is c)
mux_level2_3strobe <= "01"	Mux will send out the constant "1"
reg_1strobe <= '1'	Register 1 will contain the value of d = c + 1
reg_0strobe <= '0'	Register 0 is disabled
reg_2strobe <= '0'	Register 2 is disabled

State 4

This is the state that will perform converge d.

mux_level1_0strobe <= "00"	Converge d which is the output of the converger is sent to the mux
reg_0strobe <= '1'	Register 0 will contain the value of the converged d
reg_1strobe <= '0'	Register 1 is disabled
reg_2strobe <= '0'	Register 2 is disabled

State 5

This is the state that will perform e = d + 2.

mux_level1_0strobe <= "01"	e = d + 2 which is the output of the adder is sent to the mux
mux_level2_2strobe <= "0"	The converge output through register 0 is selected
mux_level2_3strobe <= "10"	Mux will send out the constant "2"
reg_0strobe <= '1'	Register 0 will contain the value of e = d + 2
reg_1strobe <= '0'	Register 1 is disabled
reg_2strobe <= '0'	Register 2 is disabled

3.5.3 Block Diagram and Simulation Waveform

Block Diagram

The output of the controller is the input for the data path. Both data path and controller use the same clock and the same reset signals. The block diagram is given in Figure 20.

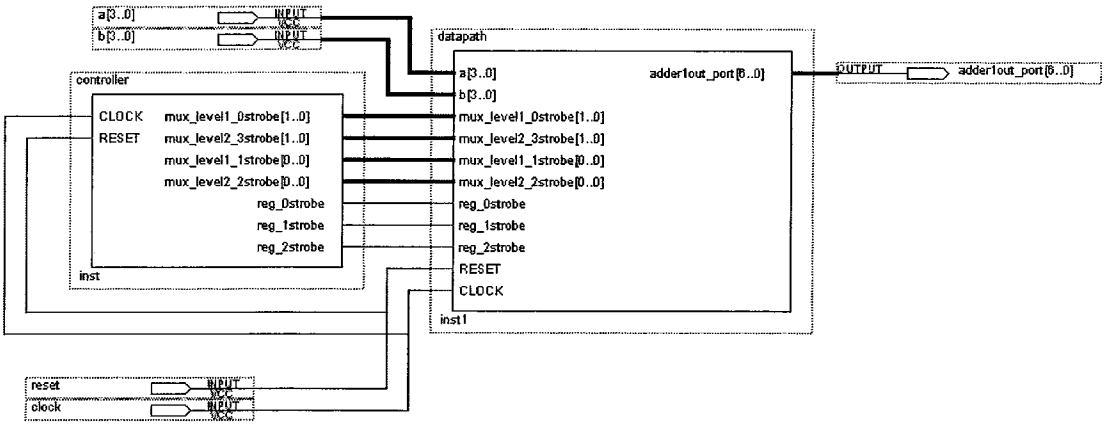

Figure 20: Block diagram for example 1

Simulation Waveform

The circuit was simulated with a = 2 and b = 4 using Altera Quartus II, version 6.0.

The operations and their outputs are shown below:

$c = a + 3$	$c = 2 + 3 = 5$
$b < 0$	False $\rightarrow 0$
$d = c + 2$	$d = 5 + 2 = 7$
$d = c + 1$	$d = 5 + 1 = 6$
Converge d	$d = 7$
$e = d + 2$	$e = 7 + 2 = 9$

The vertical red line in the simulation waveform shows the result which is obtained after six states including the reset state. Since there is only one adder in the resource bag, it

will perform all the additions, this is why d = c + 1 = 6 is also shown even though its value does not propagate further throughout the successive operations.

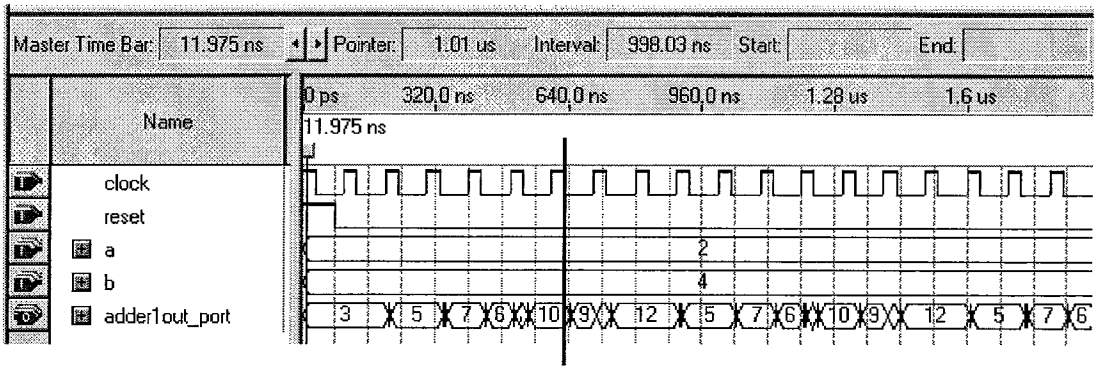

Figure 21: Simulation waveform for example 1

Chapter 4: Priority-Driven Simulated Annealing

This work consisted of reducing the area of a design while trying at the same time to reduce the execution time. Optimizing the area was obtained by using simulated annealing to determine the best cost. A cost function consisting of the total number of LUTs in the data path along with the total number of control steps in the schedule was devised. The neighboring solution in the annealing process consists of first selecting a certain node and second of selecting a potential move. A move can consist of rescheduling a node, swapping variables between registers, swapping operations between functional units or swapping the order of operands to a functional unit. Which node and which move to select will depend on a priority function. The priority function guides the simulated annealing process into finding the best solution while at the same time incurring the least possible execution time.

This section first presents the motivation behind this work by exposing the related published literature. The approach used in this work is also explained. This section also shows how the priority and cost functions are obtained.

4.1 Previous Work

Over the past few years, many algorithms have been devised to improve the solution quality by using stochastic optimization techniques that aim to solve the high-level synthesis tasks simultaneously. For example, [11] proposed a technique based on simulated annealing to concurrently perform scheduling and allocation. A global optimization over the entire data path was obtained. The cost of the data path was

calculated based on area parameters and is represented by the following equation: cost = p1* number of ALUs + p2*execution time + p3*number of registers + p4*number of multiplexers. The potential move consisted of interchanging two code operations, displacing a code operation from one location to another, or interchanging the variables in a symmetric operation. The choice of which move to select was based on generating two random numbers n1 and n2. The range of n1 was chosen from one to the number of operations. The range of n2 was chosen from one to the number of operations multiplied by a factor, which was typically equal to five. These random numbers were compared. If n2 was found to be less than the number of nodes, then n1 and n2 were interchanged. If a constraint violation occurred, then if either n1 or n2 is symmetric, the input variables in one of them are interchanged. If n2 is greater than the number of operations, an attempt is made to reschedule n1. During the end of the annealing process at low temperatures, states are generated in a different manner so that they become more likely to be acceptable. However, this technique does not take into account swapping variables between registers and also is based purely on random selection. [10] proposed a technique based on simultaneous scheduling, allocation and binding also based on simulated annealing. The quality of a solution is evaluated using the equation: $c(i) = \sum_{t \in type} ct*M(t, i) + cx*X(i) + cs * S(i)$. In the previous equation, i represents a solution, M(t, i) represents the number of used functional units and registers and X(i) represents the number of equivalent 2-1 multiplexers. S (i) describes the quality of a schedule and can be found in [13]. The actual hardware cost is represented by the constants ct and cx, while cs represents the weight of the schedule quality. An operation is chosen randomly and one of four potential moves is applied. The first technique involved scheduling the selected operation into a new control step. This will require maintaining module and

register binding. The second technique involved binding the selected value to a new register. A new register is selected randomly from a set of available registers. The third technique consisted of binding the selected operation to a new functional unit. This will require new register binding. The fourth technique consisted of swapping module inputs of the selected operation if it was commutative. However, this work is based on a random approach for selecting the neighboring solution in the annealing process. [12] applies simulated evolution to simultaneously solve scheduling and allocation. This SE-based algorithm is based on genetic algorithms and consisted of repeatedly ripping up parts of the design using a probabilistic approach. These parts were then reconstructed based on application specific heuristics. This approach provides a feasible technique, however, further interconnect optimization is required.

4.2 Approach

As mentioned earlier, four techniques were devised in an attempt to reduce area. This section first shows how these techniques will optimize the size of the design by illustrative examples. The details of the implementation decisions are also elaborated in this section.

Swapping Operations between Registers

The first technique to be considered consists of swapping variables between registers. Swapping operations bound to registers can reduce the area by mainly reducing the number of multiplexers or by reducing the number of inputs to a multiplexer. Consider the following HDL file that consists of only three operations:

o1 = v1 + v2;

o2 = o1 + v4;

o3 = o2 + v2;

It is assumed that the register bag contains only one adder.

A possible register binding combination could be:

R1: v2, o3

R2: v1, v4, o2

R3: o1

The corresponding data path is shown in Figure 22. It shows that the design needs two multiplexers to be bound to the inputs of the adder.

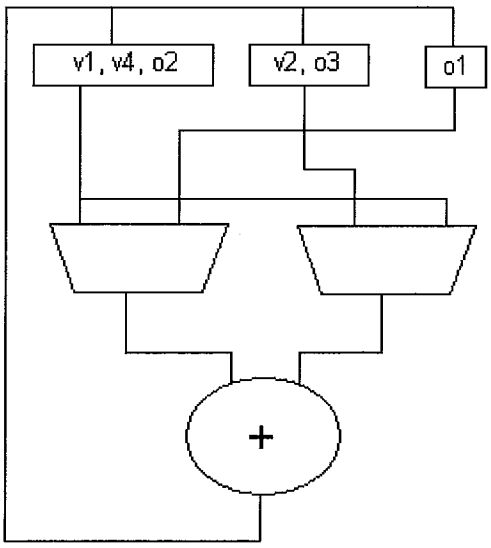

Figure 22: Data path before swapping operations between registers

Now, suppose that operations o1 and v4 were swapped. This is possible since o1 does not conflict with either of the remaining operations bound to the first register as in the figure. In other words, o1's lifetime does not overlap with the lifetimes of v1 or o2 and can hence be added to the same register. From v4's side, there is also no problem, since

originally o1 does not share a register with any other operation. The swap will generate the following variable distribution within registers:

R1: v2, o3

R2: v1, o1, o2

R3: o4

The corresponding data path is given in Figure 23. The data path shows that in this case the design used only one multiplexer connected to the adder. Hence, the swap was able to reduce the number of multiplexers by one.

It is also worth noting that the wire length also decreased[2]. In the initial design, the output of the adder was connected to all three registers. In the optimized design, the output of the adder is connected to two registers out of three, thus saving on wire length.

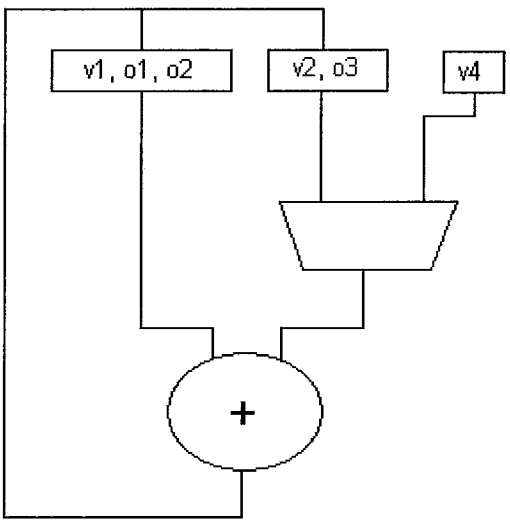

Figure 23: Data path after swapping operations between registers

[2] This work does not take into account the wire length while calculating area. The cost metrics are discussed in Section 4.4.

When a node is selected to be bound to a different register, a list of possible swaps is composed. This list can be divided into three possible ways to be populated. First, two nodes can be swapped if they have equal lifetimes. Let n be the chosen node. All nodes that have the same lifetime as n are added to this list. The second set of nodes that can be added to this list is the set of cliques whose operations do not conflict with n. Nodes n1 and n2 are not considered to be in conflict if their lifetimes do not intersect. In other words if the end time of n1 $\leq$ start time of n2 or if the end time of n2 $\leq$ start time of n1, then there is no conflict between n1 and n2. If node n does not conflict with any node in the clique, then it can be added to the clique. The third set of nodes that can be added is the set of individual nodes that do not conflict with n. Hence, n can be added to these nodes to form a clique. Once this list is populated, then n can be swapped or added to either item in the list. A random number is generated. Based on this number, one of the nodes in the list is chosen and the nodes are swapped. It is evident that upon swapping, the interconnection will change. Hence a new data path is generated and the new cost is calculated for comparison with the best cost.

Swapping Operations between Functional Units

The second technique to be considered consists of swapping operations between functional units. Swapping operations bound to functional units can reduce the area by mainly reducing the number of multiplexers or by reducing the number of inputs to a multiplexer. Consider the following circuit:

a = b + c;

d = e + f;

g = e + d;

h = a + c;

It is assumed that the register bag contains two adders.

A possible functional unit binding could end up with the following assignments:

Adder 1: a = b + c and g = e + d

Adder 2: d = e + f and h = a + c

The data path is shown in Figure 24. It shows that four multiplexers are needed.

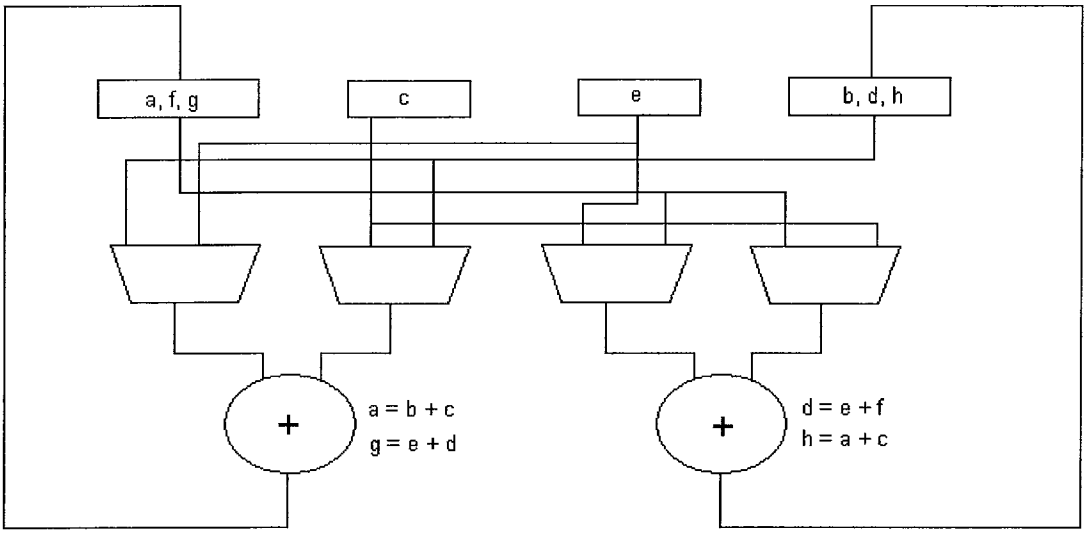

Figure 24: Data path before swapping operations between functional units

Suppose that operations g and h were swapped. Hence, now the functional unit assignments will be:

Adder 1: a = b + c and h = a + c

Adder 2: d = e + f and g = e + d

The corresponding data path is given in Figure 25. the number of multiplexers was reduced to two instead of four. A considerable decrease in wire length can also be found in the area between the registers and the multiplexers. Two horizontal tracks are now needed instead of four.

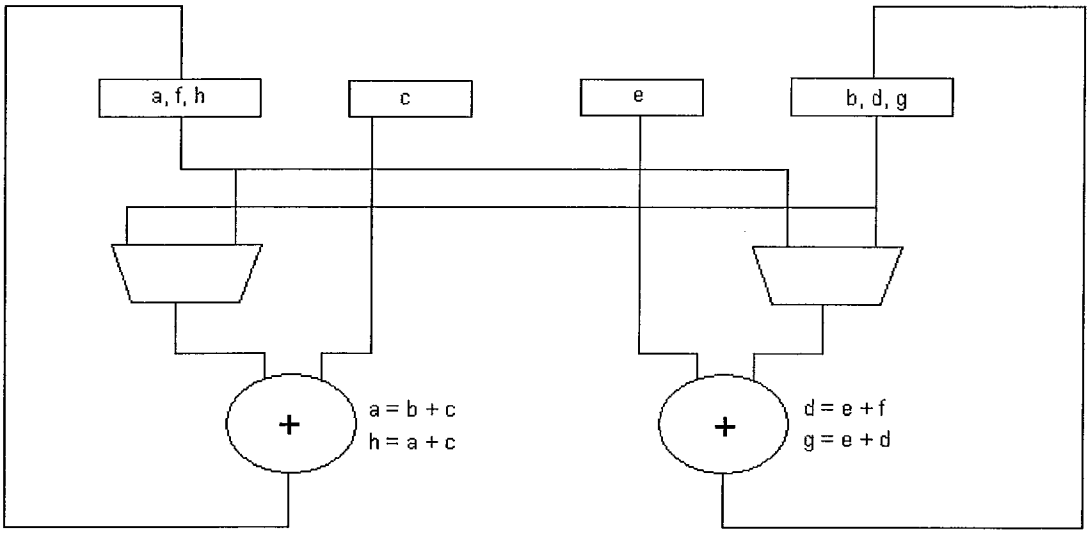

Figure 25: Data path after swapping operations between functional units

When a node is selected to be bound to a different functional unit, a list of possible swaps is composed. This list can be divided into three possible ways to be populated. First, two operations can be swapped if they belong to the same type. In other words, an addition mapped to adder1 can be swapped with an addition to adder2 but not to a multiplication bound to multiplier1. As well, these two operations should belong to the same list control step. Let n be the chosen operation. The second set of nodes that can be added to this list is the set of cliques whose operations do not conflict with n. If node n does not conflict with any node in the clique, then it can be added to the clique. The third set of nodes that can be added is the set of individual nodes that do not conflict with n. Hence, n can be added to these nodes to form a clique. Another move to be considered is the addition of an unused functional unit in the design. Suppose that the resource bag contains three adders and optimally Eridanus was able to allocate just two adders; a possible move could make use of this third adder that was unused initially but that is available in the resource bag. Once this list is populated, then n can be swapped or added to either item in the list. A random number is generated. Based on this number, one of

the operations in the list is chosen and the nodes are swapped. Upon swapping, the register assignment and the interconnection will change. Hence a new data path is generated and the new cost is calculated for comparison with the best cost.

Swapping Inputs to Operations

The third technique to be considered consists of swapping the order of inputs in commutative functional units. Swapping inputs in operations can reduce the area by mainly reducing the number of multiplexers or by reducing the number of inputs to a multiplexer. The example provided here is not complete but represents a partial circuit.

x = a + b;

y = b + c;

For the purpose of illustration, it is assumed that the variables a, b and c each belong to a different register and that only one adder is available. The data path corresponding to the mentioned assumptions is shown in Figure 26.

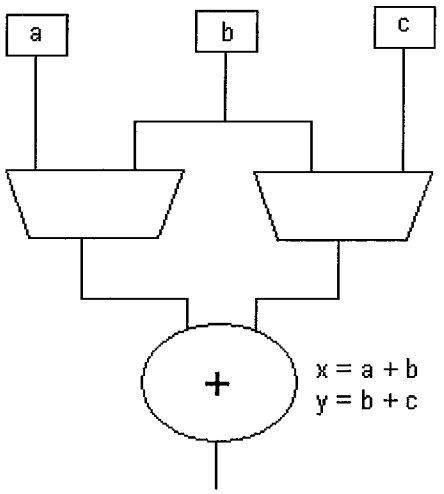

Figure 26: Partial data path before swapping inputs to operations

Suppose that the order of the inputs to the operation, y, are now swapped. Hence, y is represented by:

y = c + b;

In this case, one less multiplexer is used and the corresponding data path is shown below.

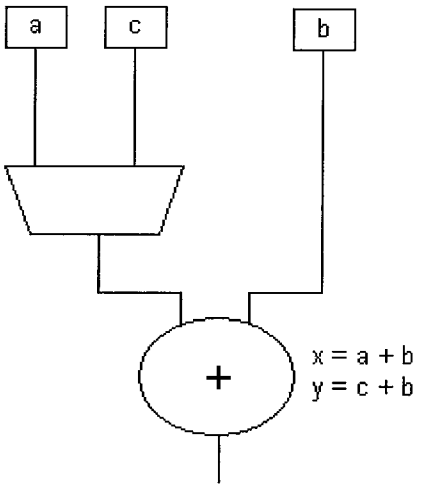

Figure 27: Partial data path after swapping inputs to operations

If the selected node is commutative, then its inputs can be swapped. This will incur different binding assignments of registers, functional units and multiplexers. Hence a new data path is generated and the new cost is calculated for comparison with the best cost.

Rescheduling an Operation

The fourth technique to be considered consists of rescheduling an operation into a control step different from that assigned to it by the list scheduling algorithm. Rescheduling an operation can reduce the area by mainly decreasing the number of functional units. Consider for example a resource bag that contains two multipliers. Two

multiplications, n1 and n2, can be scheduled into the same control step if of course they do not contain any data dependencies. If n2 was to be rescheduled into a later control step, then only one multiplier will be required and will be reused. This will not necessary imply a longer schedule if for example, n2's mobility is greater than one. Hence, the area will be reduced. The figure below illustrates this example.

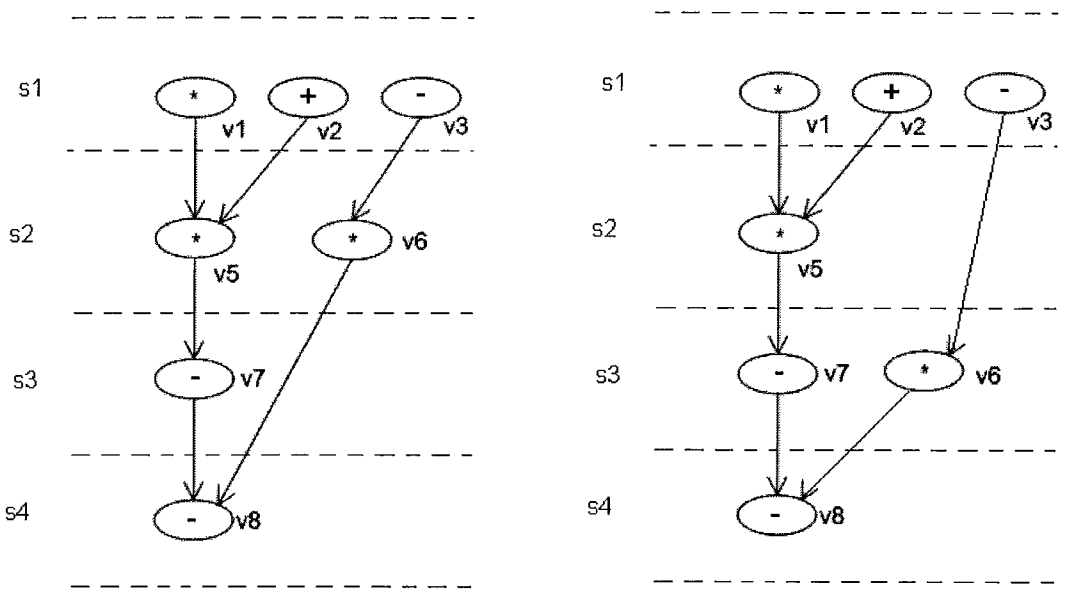

Figure 28: DFG before and after rescheduling

When a node is selected to be rescheduled, its mobility is checked. A random number is generated to determine at which step it will be rescheduled. For example, if a node has a mobility of three, then it can be rescheduled into three consecutive control steps. The random number will select to which step to assign the node. Since list scheduling depends on a priority function to sort nodes in order to be scheduled, then once a node is rescheduled, the rest of the nodes with lower priorities should be rescheduled as well.

Rescheduling a node implies that register binding and functional unit binding need to be re-performed. Hence a new data path is generated and the new cost is calculated for comparison with the best cost.

4.3 Priority Function

Since simultaneous scheduling, binding and allocation were considered, the priority function to be devised had to take into account all these tasks. As well, the priority function had to take into account the potential optimization incurred by performing one of the moves: rescheduling, swapping variables between registers, swapping operations between functional units, and swapping inputs to functional units. Hence, the priority function calculates the priority of a node and is divided into four terms:

$P = P1 + P2 + P3 + P4$

The higher the probability that an operation has, the higher the likelihood that that operation will give an area reduction once subjected to a move. The priority function will also take into consideration the number of bits that a node has relative to the maximum number of bits attained at the end of the computations.

Calculating P1

P1 denotes the priority obtained based on scheduling.

The first term in the priority function is based on the schedule and is calculated by finding the mobility of an operation. The mobility of a node is found by computing the slack between the list and ALAP values. Mobility can be calculated as such:

Mobility = max level – min level

The min level represents the list control step, whereas the max level represents the ALAP control step, only if it is greater than the list level. Otherwise, the max level will represent the list level. This calculation is due to resource constraints. Even though the list scheduling aims to schedule the nodes as early as possible, the availability of resources is another primary determinant in scheduling. Therefore, in some cases, the ALAP level of a node might be less than the list level of that node.

Nodes with high mobility can be rescheduled into more control steps than nodes with less mobility and will not likely affect the other nodes. As well, nodes with more freedom of movement will probably have fewer operations bound to them. Hence, it is important to reschedule those nodes with smaller mobility first. P1 calculates the priority as an inverse function with respect to mobility. Nodes with smaller mobility are assigned to higher priorities. If a node has zero mobility, then there is no room for rescheduling. Therefore, its P1 value will be set to zero.

Let n denote a node. The P1 value of n is represented by:

P1 [n] = (1 / mobility[n]) * factor

The term "factor" in the previous equation represents the bit width of the operation with respect to the maximum number of bits required for the output. It is calculated as such:

Factor = width[n] / max width

This is needed to break ties between two operations with the same slack but that have different bitwidths. Higher priority is given to operations with greater bitwidth since optimizing such operations will lead to better results.

Calculating P2

P2 denotes the priority obtained based on register binding.

The second term in the priority function is based on the register assignment and is calculated by finding the lifetime of an operation. The lifetime of a node is found by computing the difference between that node's end time and start time. The start time of a node represents the control step in which the variable was first encountered. The end time of a node represents the control step in which the node was last used. Hence the lifetime of node n is computed as follows:

Lifetime [n] = end time[n] – start time [n]

Variables with short lifetime spans are given higher priorities than variables with longer lifetime spans. This is because variables with short lifetimes can be swapped more frequently and more easily between registers than variables with long lifetimes that dominate most of a register's lifetime. Hence, it is important to schedule nodes with short lifetime spans first. P2 calculates the priority as an inverse function with respect to a node's lifetime. Nodes with shorter lifetimes are assigned to higher priorities.

Let n denote a node. The P2 value of n is represented by:

P2 [n] = (1 / lifetime[n]) * factor

The term "factor" in the previous equation denotes the same value as for P1.

This is needed to break ties between two operations with the same lifetimes but that have different bitwidths. Higher priority is given to operations with greater bit width since optimizing such operations will lead to better results. Consider for example two

operations that have the same lifetimes, one consists of five bits; the other consists of eight bits. Suppose that the reassignment of these operations will result in the elimination of an input to a multiplexer. Then reassigning the operation with eight bits will lead to more area reduction that reassigning the operation with five bits. This is because the multiplexer area grows with the increase in the number of bits of its inputs.

Calculating P3

P3 denotes the priority obtained based on functional unit binding.

The third term in the priority function is based on the functional unit assignment. It is calculated by finding the number of nodes connected to an operation. This is found by calculating the number of edges that originate from a node, since that value will represent the number of connections a node has. Let n denote a node; if n is the source node of an edge, then that node has an edge incident to n. The total number of edges converging from node n represents the total number of functional units whose inputs share the output of n.

The more operations have common inputs and outputs with other operations, the higher their priority. This is because they will be used more frequently than other operations, and an optimization in this case will be likely to affect a greater number of nodes. Hence it is possible to bind operations with the same inputs or outputs to the same functional units. In order to normalize the value of P3, the total number of edges out of a node is divided by the maximum number of edges that originate out of all the nodes. P3 calculates the priority as a directly proportional function with respect to the number of common edges. Nodes with greater number of connections are assigned to higher priorities.

Let n denote a node. The P3 value of n is represented by:

P3 [n] = (edges / max edges) * factor

The term "factor" in the previous equation denotes the same value as for P1.

This is needed to break ties between two operations with the same number of incident edges but that have different bit widths. Higher priority is given to operations with greater bit width since optimizing such operations will lead to better results.

Calculating P4

P4 denotes the priority obtained based on the type of the operation.

The fourth term in the priority function is based on input assignment to operations. It is calculated by finding whether an operation is commutative or not. Commutative operations, such as adders and multipliers can have the order of their inputs swapped without affecting the correctness of execution. Non-commutative operations on the other hand do not have that property. Non-commutative operations include subtraction, division, and comparison. If an operation is commutative, its "commutative" value is set to one. Otherwise, it is set to zero. P4 calculates the priority as a directly proportional function with respect to whether a node is commutative. It is evident that commutative nodes are assigned higher priorities than non commutative nodes.

Let n denote a node. The P4 value of n is represented by:

P4 [n] = commutative * factor

The term "factor" in the previous equation denotes the same value as for P1.

This is needed to break ties between commutative operations. Higher priority is given to operations with greater bit width since optimizing such operations will lead to better results.

4.3.1 Selecting a Node

Once the priority function is calculated, a node is selected to undergo the annealing process. The selection of a node is done in five ways and the best solution and its corresponding execution time was calculated and recorded. These five selection methods were done for the sake of comparison. This work emphasizes the fifth method; however, all five are explained in what follows.

The first method is a random one that does not take priority into account. Hence a node is selected at random and a potential move is also selected at random. This is the same method used in [10]. The second method consists of iterating over all the nodes in the inner loop of the simulated annealing process. The order of iteration is based on the priority of the nodes. The third method is similar to the second, but the nodes are selected according to their inverse priority. The fourth method consists of iterating over all the nodes without taking the priority of the nodes into account. The fifth and proposed method consists of iterating the inner loop over only 80% of the prioritized nodes. The measure of 80% was obtained by trial and error and was found to be the most suitable value.

4.3.2 Selecting a Move

After a node is selected as the node to consider for a possible move, the kind of move should be determined. Suppose that node n was chosen. What remains is to select whether node n should be rescheduled into a new control step, whether it should be bound to a different register, whether it should be bound to a different functional unit, or whether its inputs should be inverted. In order to select which option to apply, the deviation of each node from the average deviations of each priority component is calculated. The four deviations of each node are then sorted in decreasing order and the method corresponding to the highest deviation is considered to be the neighboring solution in the annealing process.

For each node, an array of four locations is maintained. The first location contains the deviation of that node with respect to its P1 value. The second location contains the deviation with respect to the P2 value. The third location contains the deviation with respect to the P3 value and the last location contains the deviation with respect to the P4 value. To calculate the deviations for a node, the range of the corresponding priority is calculated and the average value of that priority. Let Pk denote the term of the priority function corresponding to method k; where method k can be rescheduling (k = 1), swapping variables between registers (k = 2), swapping operations between functional units (k = 3) or swapping inputs to operations (k = 4). The range represents the difference between the maximum value and the minimum value that Pk has over all the nodes. For example, if node n has the greatest lifetime of five and node m has the least lifetime of one, then the range for P2 will be 5 – 1 = 4. Of course, this value is inaccurate

since the P-components are multiplied by a factor based on bitwidth and are normalized. The average value of Pk is the summation of all the Pk values in the schedule over the total number of nodes. Therefore, to calculate the deviation of a node n with respect to k, the following equation is used:

deviationPk[n] = (Pk[n] – average Pk) / range Pk

Each node has four deviation values which are calculated according to the previous equation. Once all four deviations are calculated, they are sorted in decreasing order. The method corresponding to the highest value is selected as the neighbor solution in the annealing process.

4.4 Cost Function

The cost function is the primary indicator of a move's success or failure. In order to determine whether a new solution is better than the preceding ones, its cost should be less. Hence, calculating the cost should be done as accurately as possible. Since optimization is targeted through four techniques, then the cost function should not exclude any elements pertaining to scheduling, allocation and binding. Therefore, the cost function takes into account the number of control steps, the number of input bits for each component and the component types.

The cost can be expressed as a function of two variables as such:

cost = c_steps + A

where c_steps denotes the number of control steps required for a given schedule, and A denotes the total area of the layout design.

The variable c_steps can be easily obtained by finding how many control steps the specified input and resource files produced using the list algorithm.

The total area, A, on the other hand, is not obtained as easily. As mentioned earlier, A is the total area of the design, comprising of functional units, registers and multiplexers. Wire length is not included in the equation. Area is measured in terms of LUTs fr representative FPGAs (Family: FLEX10K). Individual components were built on Quartus II and simulated taking into account the number of input bits. For registers, adders and subtractors, the number of LUTs was proportional to the number of bits. With respect to multiplexers and multipliers, several cases were considered and an interpolation formula was generated using Excel. The number of LUTs for each component can be found in the tables below. The tables are a representation of the number of LUTs needed according to the number of input bits. To simplify, let n denote the number of input bits, and l denote the number of LUTs needed.

The total area, A, will then be expressed as follows:

$A = l_{(registers)} + l_{(multiplexers)} + l_{(functional\ units)}$

Registers

Table 7: Number of LUTs for registers

Number of input bits (n)	*Number of LUTs (l)*
1	1
2	2
3	3
4	4
8	8
16	16

A 4-bit register will need 4 LUTs. Hence the equation obtained is:

$l = n$.

Multiplexers

The number of LUTs needed for 1-bit inputs to multiplexers is shown below. There is no direct formula; hence, one was derived based on the graph in Figure 29.

Table 8: Number of LUTs for 1-bit inputs to multiplexers

Number of inputs (k)	*Number of LUTs (l)*
2	1
3	2
4	2
5	3
6	4
7	5
8	5
9	6
10	7
12	8
14	9
16	10
20	13
32	21

The x-axis represents the number of inputs to the multiplexer and the y-axis represents the number of LUTs needed. Hence, for 1-bit inputs, the following equation was obtained:

$l = 0.659*k - 0.1089$

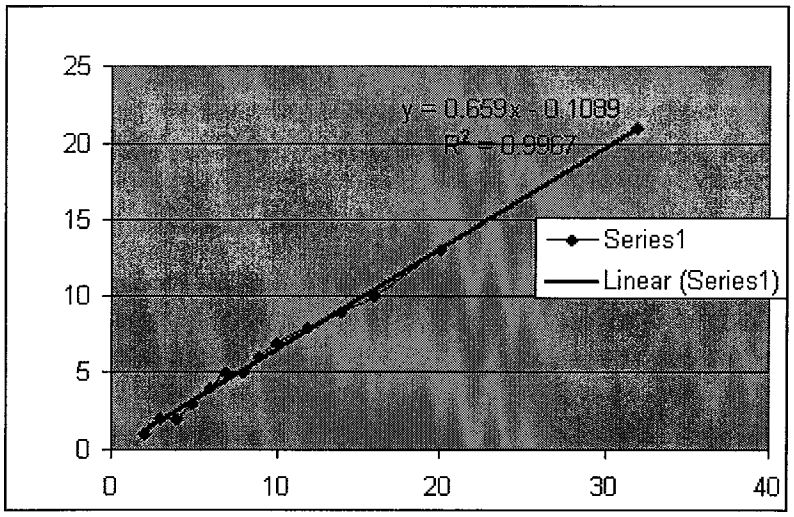

Figure 29: Graph for calculating number of LUTs for muxes with 1-bit inputs

The following tables summarize the results for 2-bit, 3-bit, 4-bit and 8-bit inputs.

Table 9: Number of LUTs for muxes

Number of input bits	*Number of inputs (k)*	*Number of LUTs (l)*
2-bit inputs	2	2
	4	4
	8	10
	16	20
	32	42
3-bit inputs	2	3
	4	6
	8	15
	16	30
	32	63
4-bit inputs	2	4
	4	8
	8	20
	16	40
	32	84
8-bit inputs	2	8
	4	16
	8	40
	16	80
	32	168

Let b denote the number of input bits. The number of LUTs needed will be calculated form the following equation:

l = b * area of the 1-bit inputs muxes with the same number of inputs

Adders/Subtractors

Table 10: Number of LUTs for adders and subtractors

Number of input bits (n)	*Number of LUTs (l)*
1	1
2	2
3	3
4	4
8	8
16	16

An 8-bit adder will need 8 LUTs and a 2-bit subtractor will need 2 LUTs. Hence the equation obtained is:

l = n.

Multipliers

Table 11: Number of LUTs for multipliers

Number of input bits (n)	*Number of LUTs (l)*
1	1
2	4
3	21
4	33
6	72
8	130
10	175
12	276
14	361
16	462

The data in the table cannot be easily interpreted into an equation. Therefore, the values obtained were plotted in a graph using Excel and the following equation was produced based on the graph of Figure 30.

$l = 1.6513 * n^2 + 2.8869 * n - 4.8414$

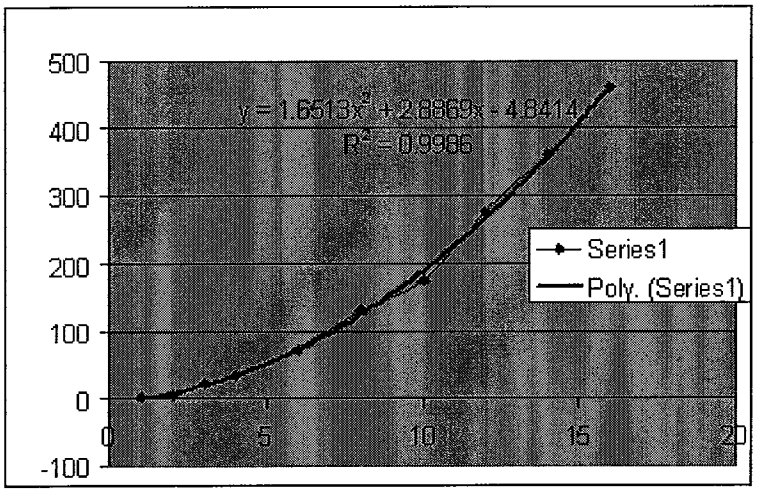

Figure 30: Graph for calculating number of LUTs for multipliers

Chapter 5: Results and Analysis of Data

The results obtained are tabulated in the following tables. The list schedule for each benchmark is also shown given that the resource bag contains two adders, two multipliers and one subtractor. The input file for the poly_design benchmark is shown. The input files used in the other benchmarks can be found in the Appendix. Method 1 corresponds to random selection. Method 2 corresponds to the priority selection. Method 3 corresponds to inverse priority. Method 4 corresponds to no priority. Method 5 corresponds to 80% of the nodes.

Poly design

The input size is 5 bits for each input.

The initial cost is 513.

```
program
in x, d, c, b, a :std_logic_vector(4 downto 0);

begin

m1 := a * x;
s1 := m1 + b;
m2 := x * x;
m3 := s1 * m2;
m4 := c * x;
s2 := m4 + d;
s3 := s2 + m3;

end .
```

Table 12: Poly Design results

		Method 1			*Method 2*		
Resource bag	**Initial cost**	**Best cost**	**Best cost time**	**Execution time**	**Best cost**	**Best cost time**	**Execution time**
1*, 1+	421	386	5461	29773	386	4273.5	29750
2*, 1+	511	488	4282	32468	488	6359.5	32891
2*, 2+	513	428	5593	31703	428	6023.5	32546.5
3*, 3+	511	488	6462	34304	488	6078	34453

Table 13: Poly Design results (cont'd)

Method3			*Method 4*			*Method 5*		
Best cost	**Best cost time**	**Execution time**	**Best cost**	**Best cost time**	**Execution time**	**Best cost**	**Best cost time**	**Execution time**
386	6250	29797	386	6250	29828	386	5070	26968.5
488	4429.5	32023	488	5570	31765.5	488	5383	28258
428	6242.5	32891	428	4953.5	31070.5	433	6531.5	30141
488	5305	32148	488	5875.5	31664.5	489	4336	28359.5

Improvement with respect to method 1:

Table 14: Poly Design improvement wrt Method 1

	Method 2	*Method 3*	*Method 4*	*Method 5*	
	% of time	**% of time**	**% of time**	**% of area**	**% of time**
	21.75	21.75	-14.45	0.13	7.16
	-48.54	-48.54	-30.10	-0.20	-25.73
	-7.68	-7.68	11.44	-1.29	-16.77
	5.93	5.93	9.06	-0.20	32.89
Average	-7.14	-7.14	-6.01	-0.39	-0.61

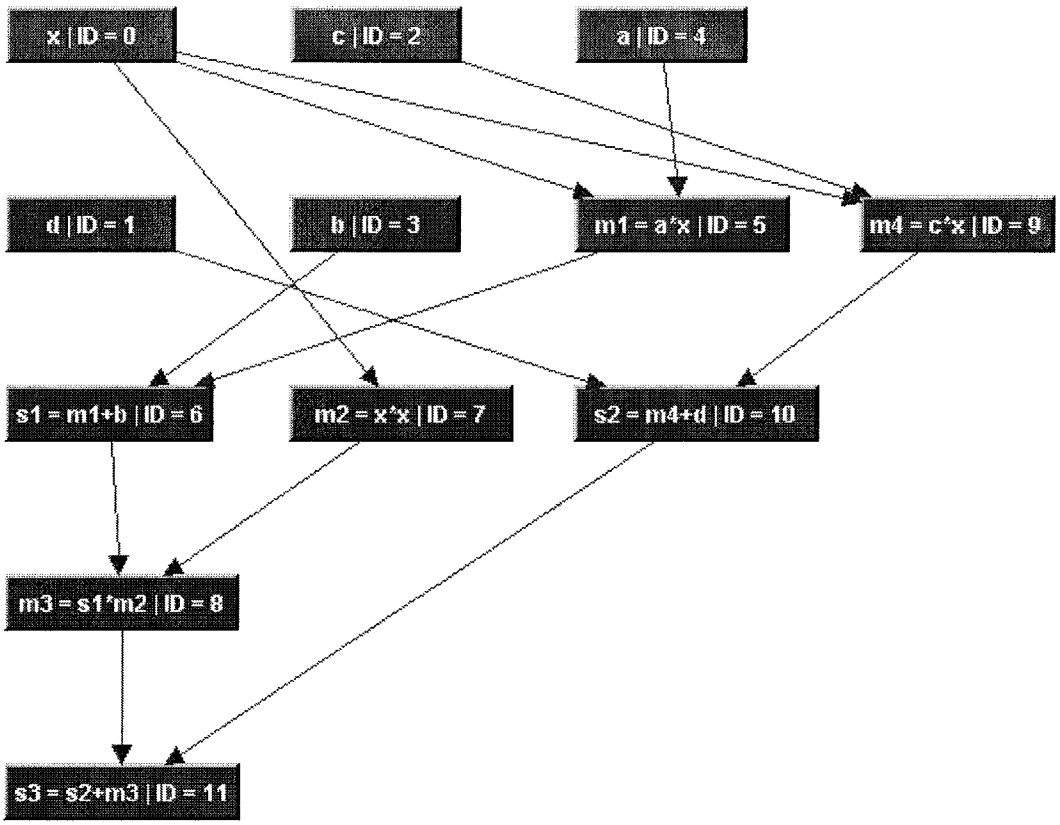

Figure 31: List schedule for poly design benchmark

Diffeq

The input size is 5 bits for each input.
The initial cost is 1304.

Table 15: Diffeq results

		Method 1			*Method 2*		
Resource bag	**Initial cost**	**Best cost**	**Best cost time**	**Execution time**	**Best cost**	**Best cost time**	**Execution time**
1*, 1+, 1-	1221	1142	414146	415244	1142	92088	355984
2*, 2+, 2-	1304	1169	462735	463948	1142	104140	365703
2*, 2+, 1-	1304	1162	443324	444400	1137	129102	381922
2*, 1+, 1-	1304	1164	429953	431109	1168	108447	378406
3*, 3+, 3-	1329	1222	504266	505380	1221	95453	425406

Table 16: Diffeq results (cont'd)

Method3			*Method 4*			*Method 5*		
Best cost	**Best cost time**	**Execution time**	**Best cost**	**Best cost time**	**Execution time**	**Best cost**	**Best cost time**	**Execution time**
1145	388782	389025	1142	213336	322638	1142	81704	307968
1134	436208	436457	1144	301550	423479	1154	59781	246875
1144	426811	427077	1144	192628	322031	1134	68532	240781
1166	450576	450803	1170	272272	395769	1176	52828	276907
1222	391652	391823	1223	221878	393900	1222	52265	292234

Improvement with respect to method 1:

Table 17: Diffeq improvement wrt Method 1

	Method 2	*Method 3*	*Method 4*	*Method 5*	
	% of time	**% of time**	**% of time**	**% of area**	**% of time**
	77.76	6.12	48.49	0.00	80.27
	77.49	5.73	34.83	1.25	87.08
	70.88	3.72	56.55	2.41	84.54
	74.78	-4.80	36.67	-1.06	87.71
Average	76.40	6.62	46.51	0.51	85.85

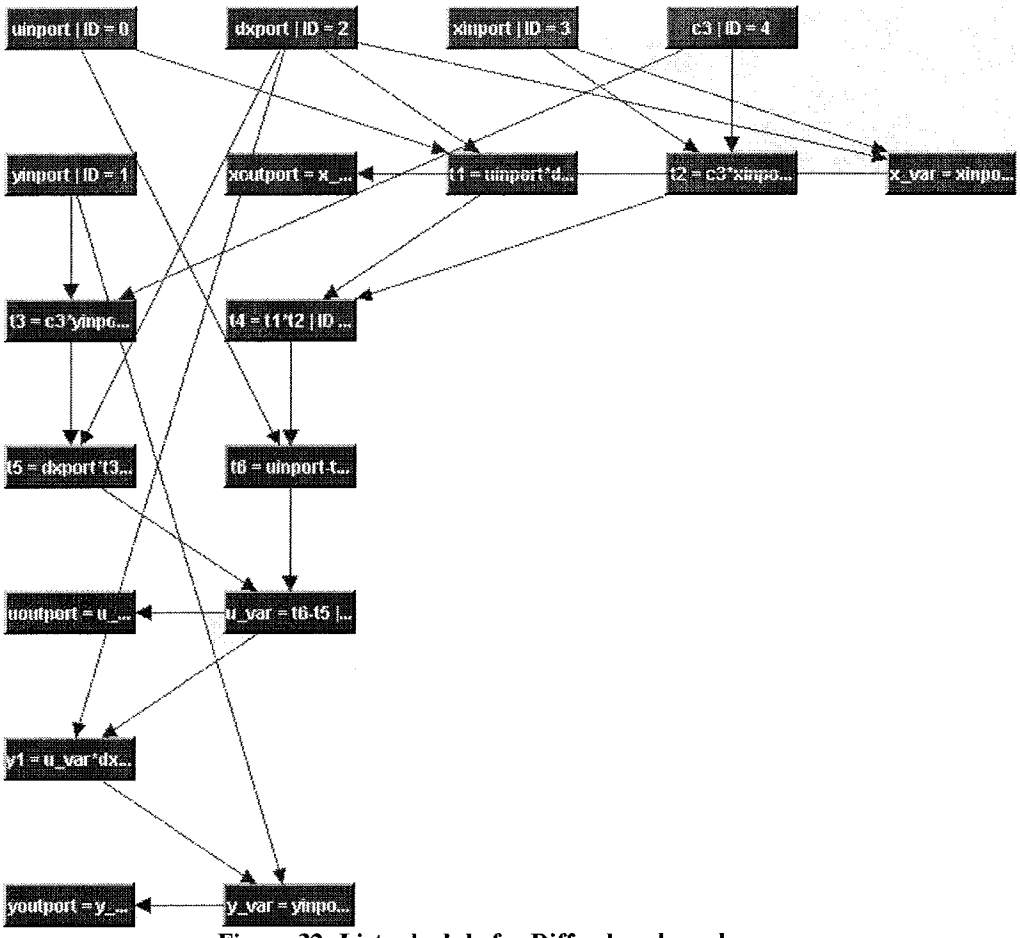

Figure 32: List schedule for Diffeq benchmark

4pt DCT

The input size is 4 bits for each input.
The initial cost is 447.

Table 18: 4pt DCT results

		Method 1			*Method 2*		
Resource bag	**Initial cost**	**Best cost**	**Best cost time**	**Execution time**	**Best cost**	**Best cost time**	**Execution time**
2*, 2+, 2-	468	341	675145	804983	324	644898	986721
2*, 2+, 1-	447	339	1062229	1181669	347	603418	729965
2*, 1+, 2-	444	349	742654	776076	353	571388	877561
3*, 2+, 1-	536	380	1209785	1430449	384	561209	657105
3*, 2+, 2-	508	397	822662	833940	394	724650	728542

Table 19: 4pt DCT results (cont'd)

Method3			*Method 4*			*Method 5*		
Best cost	**Best cost time**	**Execution time**	**Best cost**	**Best cost time**	**Execution time**	**Best cost**	**Best cost time**	**Execution time**
319	512744	597914	319	605860	643568	322	383508	440851
345	545805	631945	335	707356	722418	336	476926	565573
350	636117	681257	354	613076	632480	351	530859	663312
381	631539	681257	371	601151	745189	386	602141	637875
391	647914	716647	394	614815	740634	395	456879	520785

Improvement with respect to method 1:

Table 20: 4pt DCT improvement wrt Method 1

	Method 2	*Method 3*	*Method 4*	*Method 5*	
	% of time	**% of time**	**% of time**	**% of area**	**% of time**
	4.48	24.05	10.26	5.43	43.20
	43.19	48.62	33.41	0.88	55.10
	23.06	14.35	17.45	-0.57	28.52
	53.61	47.80	50.31	-1.58	50.23
Average	27.25	31.21	27.34	0.81	44.30

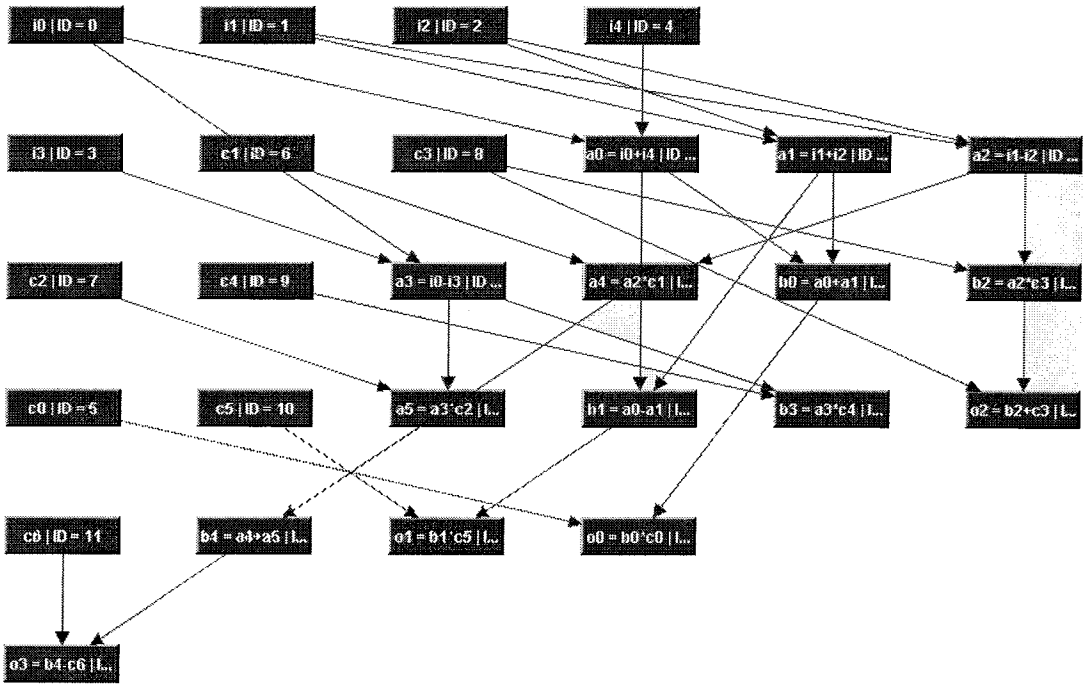

Figure 33: List schedule for 4pt DCT benchmark

AR

The input size is 4 bits for each input.
The initial cost is 1557.

Table 21: AR results

		Method 1			*Method 2*		
Resource bag	**Initial cost**	**Best cost**	**Best cost time**	**Execution time**	**Best cost**	**Best cost time**	**Execution time**
2*, 2+	1557	1456	1375063	1924760	1428	912847	1508937
3*, 3+	1916	1841	1305406	1783055	1836	1164508	1547547
3*, 2+	1990	1860	977280	1789917	1827	923178	1936633
5*, 2+	2330	2266	1030720	1784226	2232	1916159	2347312
4*, 3+	2354	2266	1452920	1667091	2264	608070	1477610

Table 22: AR results (cont'd)

Method3			*Method 4*			*Method 5*		
Best cost	**Best cost time**	**Execution time**	**Best cost**	**Best cost time**	**Execution time**	**Best cost**	**Best cost time**	**Execution time**
1462	621540	1490302	1477	541716	1481288	1457	690281	1056938
1853	998067	1399052	1849	590010	1403950	1865	1011532	1077515
1870	812523	1153192	1882	459009	1185138	1827	921158	1936537
2278	539814	1352493	2253	664739	1389887	2250	974406	1024406
2214	688066	1429688	2282	291897	1229399	2247	64000	1043813

Improvement with respect to method 1:

Table 23: AR improvement wrt Method 1

	Method 2	*Method 3*	*Method 4*	*Method 5*	
	% of time	**% of time**	**% of time**	**% of area**	**% of time**
	33.61	54.80	60.60	-0.07	49.80
	10.79	23.54	54.80	-1.30	22.51
	5.54	16.86	53.03	1.76	5.74
	-85.90	47.63	35.51	0.68	5.46
Average	4.44	39.09	56.77	0.04	43.34

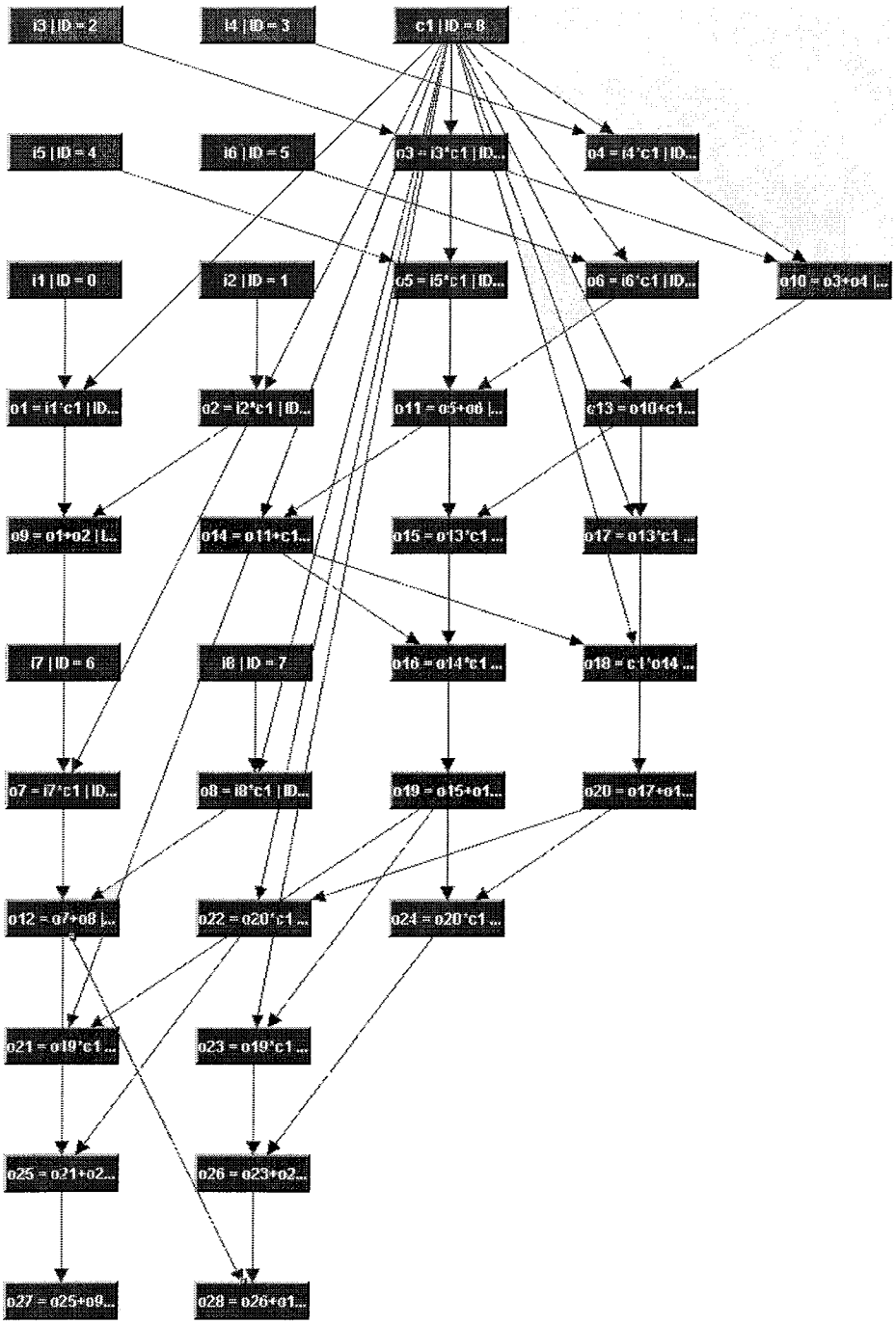

Figure 34: List schedule for AR benchmark

Elliptic

The input size is 4 bits for each input.
The initial cost is 2799.

Table 24: Elliptic results

		Method 1			*Method 2*		
Resource bag	**Initial cost**	**Best cost**	**Best cost time**	**Execution time**	**Best cost**	**Best cost time**	**Execution time**
2*, 2+	2799	2357	2620413	2672199	2376	2502024	2633769
3*, 3+	3607	3394	3094039	3122857	3403	1871742	3100976
1*, 2+	1878	1698	3106210	3133331	1602	2938648	3109193
2*, 1+	2799	2543	2595359	2600053	2369	2461339	2660618
2*, 3+	2793	2708	2987927	2996748	2624	1643472	3001543

Table 25: Elliptic results (cont'd)

Method3			*Method 4*			*Method 5*		
Best cost	**Best cost time**	**Execution time**	**Best cost**	**Best cost time**	**Execution time**	**Best cost**	**Best cost time**	**Execution time**
2381	2580826	2688124	2375	2536519	2685437	2335	2628957	2629785
3371	2722293	3092414	3283	2675162	3077422	3400	3054032	3116862
1625	2763113	3087318	1636	2905359	3059285	1650	2279502	3082274
2414	2557752	2692039	2362	2529405	2635598	2383	2329512	3173275
2610	2533890	3054715	2589	2069427	3037840	2564	2504532	3017386

Improvement with respect to method 1:

Table 26: Elliptic improvement wrt Method 1

	Method 2	*Method 3*	*Method 4*	*Method 5*	
	% of time	**% of time**	**% of time**	**% of area**	**% of time**
	4.52	1.51	3.20	0.91	-0.33
	39.50	12.01	13.54	-0.19	1.29
	5.39	11.05	6.47	0.44	26.61
	5.16	1.45	2.54	6.29	10.24
Average	19.92	8.24	11.30	1.62	10.94

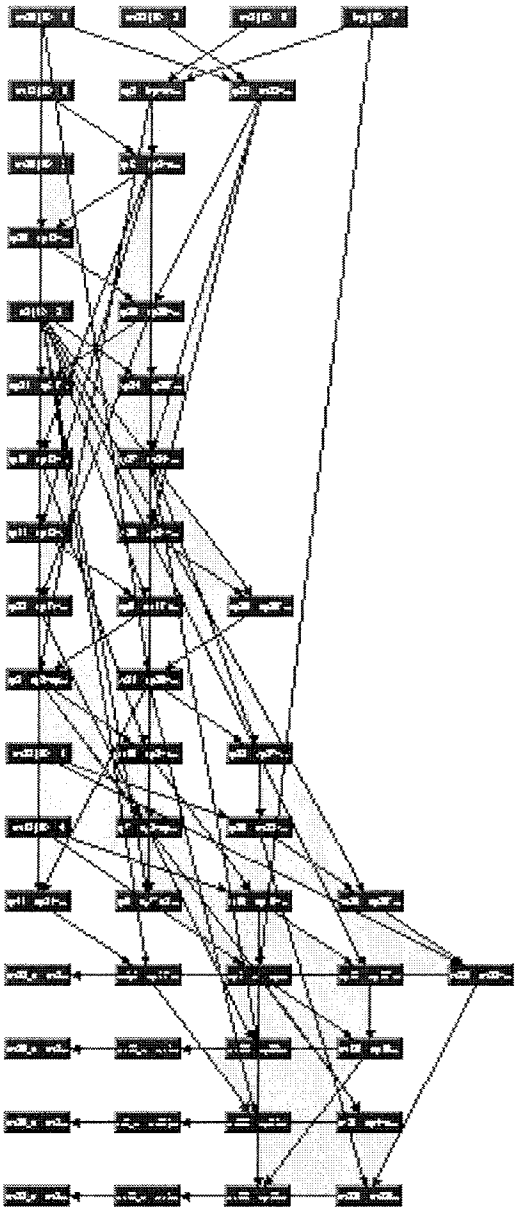

Figure 35: List schedule for Elliptic benchmark

Table 27 gives the average improvement in execution time with respect to method 1 and the values obtained are the averages over all the benchmarks.

Table 27: Average improvement with respect to Method 1

	Method 2	*Method 3*	*Method 4*	*Method 5*
Average	24.17	15.61	27.18	36.76

It is evident from the values in Table 27 that using method 3 which consists of using the inverse priority yielded the least improvement among the other techniques when compared to the random approach.

On average, an 11.4% area reduction was obtained by using method 5 in comparison to the conventional HLS flow.

As well, on average, 37% reduction in design space exploration time was obtained compared to non-priority based area optimization techniques.

Chapter 6: Conclusion

This work presented a technique to reduce the area in high-level synthesis while at the same time minimizing the overall execution time. A priority-driven simulated annealing approach was used to come up with an optimal solution. The techniques pertaining to the potential moves in the annealing process consisted of rescheduling, swapping variables between registers, swapping operations between functional units and alternation the order of inputs to commutative functional units. For this, a priority function was derived based on the four techniques. As well, a cost function to measure the quality of a move was calculated. Five simulation techniques were tested on the available benchmarks. A random approach for choosing the neighboring solution was first implemented. Second, all nodes in the priority function were considered. Third, the inverse form of the priority function was used. Fourth, an approach with no priority function but that includes all the nodes was performed. Lastly, a solution consisting of using the priority function applied on 80% of the nodes was devised. This last solution gave an improvement of 11.4% area reduction as compared to conventional high-level synthesis techniques. Also, it obtained an average of 37% reduction in execution time as compared to the random approach.

Future work could be to try to expand on the input file so as to include loop-based instructions. Hence, inter-process optimization might be performed and expanded to include the whole design in an efficient manner. Another issue has to do with interconnect area estimation. More accurate estimations of routing area that include the wire length could be incorporated to improve the quality of the results.

References

[1] Bassil, L. (2005, June). High level Synthesis: Optimization of area through optimization of functional unit binding and force directed scheduling. *Unpublished Senior Project Report, Lebanese American University, Byblos, Lebanon.*

[2] Elaaraj E. (2009, January). A novel approach to reduce spurious switching activity in high-level synthesis. *Unpublished Master's Thesis, Lebanese American University, Byblos, Lebanon.*

[3] Gajski, D., Dutt, N., Wu, A., & Lin, S. (1992) High-level synthesis. Boston: Kluwer Academic Publishers.

[4] Michelli, De., G. (1994). Synthesis and optimization of digital circuits. New York: McGraw-Hill.

[5] Kikpatrick, S., Gelatt, C.D., & Vecchi M.P. (1983) Optimization by simulated annealing. *Science, 220* (4598), 671-680.

[6] Lee J., Hsu Y., & Lin Y. (1989) A new integer linear programming formulation for the scheduling problem in data path synthesis. *Proceedings of the International Conference on Computer-Aided Design*, 20-23.

[7] Paulin P.G. & Knight, J.P. (1989, June). Force directed scheduling for the behavioral synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 8* (6), 661-679.

[8] Park, I.C. & Kyung, C.M. (1991). Fast and near optimal scheduling in automatic data path synthesis. *Proceedings of the 28th Design Automation Conference*, 680-685.

[9] Tseng, C.J, & Siewiorek, D.P. (1986, July). Automated synthesis of data paths on digital systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, CAD-5* (3), 379-395.

[10] Kollig, P. & Al-Hashimi, B.M. (1997, August). Simultaneous scheduling, allocation and binding in high level synthesis. *IEE Electronics Letter, 33* (18), 1516-1518.

[11] Devadas, S. & Newton, A.R. (1989, July). Algorithms for hardware allocation in data path synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 8* (7), 768-781.

[12] Ly, T.A. & Mowchenko, J.T. (1983, March). Applying simulated evolution to high level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 12* (3), 389-409.

[13] Kollig, P., Al-Hashimi, B.M., & Abbott, K.M. (1997, March). Efficient scheduling of behavioral descriptions in high level synthesis. *IEE Proceedings Computer Digital Technology, 144* (2), 75-82.

Appendix

Data Path VHDL Code for Example 1

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY datapath IS
PORT(
a , b : IN STD_LOGIC_VECTOR(3 downto 0);
mux_level1_0strobe , mux_level2_3strobe : IN STD_LOGIC_VECTOR(1 downto
0);
mux_level1_1strobe , mux_level2_2strobe : IN STD_LOGIC_VECTOR(0 downto
0);
reg_0strobe , reg_1strobe , reg_2strobe : IN STD_LOGIC;
RESET , CLOCK : IN STD_LOGIC;
adder1out_port :OUT STD_LOGIC_VECTOR(6 downto 0)
);
END datapath;

ARCHITECTURE structural OF datapath IS
COMPONENT adder_6
PORT(A, B: IN     STD_LOGIC_VECTOR(5 downto 0);
     Sum    : OUT STD_LOGIC_VECTOR(6 downto 0)
);
END COMPONENT;
COMPONENT less_4
PORT(D0, D1 : IN  STD_LOGIC_VECTOR(3 downto 0);
     Q : OUT      STD_LOGIC_VECTOR(0 downto 0)
);
END COMPONENT;
COMPONENT converger_6
PORT(A, B: IN     STD_LOGIC_VECTOR(5 downto 0);
     Sel: IN  STD_LOGIC_VECTOR(0 downto 0);
     Q : OUT      STD_LOGIC_VECTOR(5 downto 0)
);
```

```
END COMPONENT;
COMPONENT register_7
PORT(D : IN STD_LOGIC_VECTOR(6 downto 0);
     Resetn, Clock, Enable : IN STD_LOGIC;
     Q : OUT      STD_LOGIC_VECTOR(6 downto 0)
);
END COMPONENT;
COMPONENT register_6
PORT(D : IN STD_LOGIC_VECTOR(5 downto 0);
     Resetn, Clock, Enable : IN STD_LOGIC;
     Q : OUT      STD_LOGIC_VECTOR(5 downto 0)
);
END COMPONENT;
COMPONENT register_4
PORT(D : IN STD_LOGIC_VECTOR(3 downto 0);
     Resetn, Clock, Enable : IN STD_LOGIC;
     Q : OUT      STD_LOGIC_VECTOR(3 downto 0)
);
END COMPONENT;
COMPONENT multiplexer_3inputs_7bits
PORT(D0 , D1 , D2: IN   STD_LOGIC_VECTOR(6 downto 0);
     Sel: IN      STD_LOGIC_VECTOR(1 downto 0);
     Q : OUT      STD_LOGIC_VECTOR(6 downto 0)
);
END COMPONENT;
COMPONENT multiplexer_2inputs_4bits
PORT(D0 , D1: IN  STD_LOGIC_VECTOR(3 downto 0);
     Sel: IN      STD_LOGIC_VECTOR(0 downto 0);
     Q : OUT      STD_LOGIC_VECTOR(3 downto 0)
);
END COMPONENT;
COMPONENT multiplexer_2inputs_6bits
PORT(D0 , D1: IN  STD_LOGIC_VECTOR(5 downto 0);
     Sel: IN      STD_LOGIC_VECTOR(0 downto 0);
     Q : OUT      STD_LOGIC_VECTOR(5 downto 0)
);
END COMPONENT;
COMPONENT multiplexer_4inputs_3bits
```

```
PORT(D0 , D1 , D2 , D3: IN    STD_LOGIC_VECTOR(2 downto 0);
     Sel: IN      STD_LOGIC_VECTOR(1 downto 0);
     Q : OUT      STD_LOGIC_VECTOR(2 downto 0)
);
END COMPONENT;


SIGNAL less0_input1 , reg2out , mux_level1_1out , multiplexer1_input0 :
STD_LOGIC_VECTOR(3 downto 0);
SIGNAL adder1_input1 , converger2out , converger2_input1 , reg1out ,
REG1_input , mux_level2_2out , multiplexer2_input0 : STD_LOGIC_VECTOR(5
downto 0);
SIGNAL less0out, converger2_input2 : STD_LOGIC_VECTOR(0 downto 0);
SIGNAL mux_level2_3out , multiplexer3_input1 : STD_LOGIC_VECTOR(2
downto 0);
SIGNAL adder1out , reg0out , mux_level1_0out , multiplexer0_input0 ,
multiplexer0_input2 : STD_LOGIC_VECTOR(6 downto 0);
CONSTANT constant_3 :STD_LOGIC_VECTOR(2 downto 0):="011";
CONSTANT constant_0 :STD_LOGIC_VECTOR(1 downto 0):="00";
CONSTANT constant_1 :STD_LOGIC_VECTOR(1 downto 0):="01";
CONSTANT constant_2 :STD_LOGIC_VECTOR(2 downto 0):="010";

BEGIN

--Creating a wrapper signal multiplexer0_input0 for converger2out
multiplexer0_input0<=(6 downto 6=>converger2out(5)) & converger2out;

--Creating a wrapper signal multiplexer0_input2 for a
multiplexer0_input2<=(6 downto 4=>a(3)) & a;
mux_level1_0 :multiplexer_3inputs_7bits PORT MAP(multiplexer0_input0 ,
adder1out , multiplexer0_input2 , mux_level1_0strobe ,
mux_level1_0out);

--Creating a wrapper signal multiplexer1_input0 for less0out
multiplexer1_input0<=(3 downto 1=>less0out(0)) & less0out;
mux_level1_1 :multiplexer_2inputs_4bits PORT MAP(multiplexer1_input0 ,
b , mux_level1_1strobe , mux_level1_1out);
```

```
reg0 :register_7 PORT MAP(mux_level1_0out , RESET , CLOCK , reg_0strobe
, reg0out);

--Creating a wrapper signal REG1_input for adder1out
REG1_input<=adder1out(5 downto 0);
reg1 :register_6 PORT MAP(REG1_input , RESET , CLOCK , reg_1strobe ,
reg1out);
reg2 :register_4 PORT MAP(mux_level1_1out , RESET , CLOCK , reg_2strobe
, reg2out);


--Creating a wrapper signal multiplexer2_input0 for reg0out
multiplexer2_input0<=reg0out(5 downto 0);
mux_level2_2 :multiplexer_2inputs_6bits PORT MAP(multiplexer2_input0 ,
reg1out , mux_level2_2strobe , mux_level2_2out);

--Creating a wrapper signal multiplexer3_input1 for constant_1
multiplexer3_input1<=(2 downto 2=>constant_1(1)) & constant_1;
mux_level2_3 :multiplexer_4inputs_3bits PORT MAP(constant_3 ,
multiplexer3_input1 , constant_2 , constant_2 , mux_level2_3strobe ,
mux_level2_3out);


--Creating a wrapper signal adder1_input1 for mux_level2_3out
adder1_input1<=(5 downto 3=>mux_level2_3out(2)) & mux_level2_3out;
adder1 :adder_6 PORT MAP(mux_level2_2out , adder1_input1 , adder1out);

--Creating a wrapper signal less0_input1 for constant_0
less0_input1<=(3 downto 2=>constant_0(1)) & constant_0;
less0 :less_4 PORT MAP(reg2out , less0_input1 , less0out);

--Creating a wrapper signal converger2_input1 for reg0out
converger2_input1<=reg0out(5 downto 0);

--Creating a wrapper signal converger2_input2 for reg2out
--converger2_input2<=(5 downto 4=>reg2out(3)) & reg2out;
converger2_input2<=reg2out(0 downto 0);
converger2 :converger_6 PORT MAP(reg1out , converger2_input1 ,
```

```
converger2_input2 , converger2out);

adder1out_port<=adder1out;

END structural;
```

Controller VHDL Code for Example 1

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY controller IS
PORT(
CLOCK , RESET : IN STD_LOGIC;
mux_level1_0strobe , mux_level2_3strobe : OUT STD_LOGIC_VECTOR(1 downto
0);
mux_level1_1strobe , mux_level2_2strobe : OUT STD_LOGIC_VECTOR(0 downto
0);
reg_0strobe , reg_1strobe , reg_2strobe: OUT STD_LOGIC
);
END controller;

ARCHITECTURE behavioral OF controller IS

TYPE state_type IS (start , state0 , state1 , state2 , state3 , state4
, state5);
SIGNAL state : state_type;

BEGIN
state_process : PROCESS(CLOCK , RESET)
BEGIN
IF RESET = '1' THEN
state<=start;
ELSIF CLOCK'EVENT AND CLOCK = '0' THEN
CASE state IS
WHEN start => state<=state0;
WHEN state0 => state<=state1;
WHEN state1 => state<=state2;
WHEN state2 => state<=state3;
WHEN state3 => state<=state4;
WHEN state4 => state<=state5;
WHEN state5 => state<=start;
END CASE;
```

```
END IF;
END PROCESS;

output_process : PROCESS(state)
BEGIN
CASE state IS

WHEN start =>
reg_0strobe <= '0';
reg_1strobe <= '0';
reg_2strobe <= '0';

WHEN state0 =>
mux_level1_0strobe <= "10";
mux_level1_1strobe <= "1";
reg_0strobe <= '1';
reg_2strobe <= '1';
reg_1strobe <= '0';

WHEN state1 =>
mux_level1_1strobe <= "0";
mux_level2_2strobe <= "0";
mux_level2_3strobe <= "00";
reg_1strobe <= '1';
reg_2strobe <= '1';
reg_0strobe <= '0';

WHEN state2 =>
mux_level1_0strobe <= "01";
mux_level2_2strobe <= "1";
mux_level2_3strobe <= "11";
reg_0strobe <= '1';
reg_1strobe <= '0';
reg_2strobe <= '0';

WHEN state3 =>
mux_level2_2strobe <= "1";
mux_level2_3strobe <= "01";
```

```
reg_1strobe <= '1';
reg_0strobe <= '0';
reg_2strobe <= '0';

WHEN state4 =>
mux_level1_0strobe <= "00";
reg_0strobe <= '1';
reg_1strobe <= '0';
reg_2strobe <= '0';

WHEN state5 =>
mux_level1_0strobe <= "01";
mux_level2_2strobe <= "0";
mux_level2_3strobe <= "10";
reg_0strobe <= '1';
reg_1strobe <= '0';
reg_2strobe <= '0';
END CASE;
END PROCESS;

END behavioral;
```

Benchmarks

Diffeq

```
program
in uinport, yinport, dxport, xinport,c3:std_logic_vector(4 downto 0);
out xoutport, youtport, uoutport:std_logic_vector(12 downto 0);

begin

t1 := uinport * dxport;
t2 := c3 * xinport;
t3 := c3 * yinport;
t4 := t1 * t2;
t5 := dxport * t3;
t6 := uinport - t4;
u_var := t6 - t5;
y1 := u_var * dxport;
y_var := yinport + y1;
x_var := xinport + dxport;
xoutport:= x_var;
youtport:= y_var;
uoutport:= u_var;



end .
```

4pt DCT

```
program
in i0,i1,i2,i3,i4,c0,c1,c2,c3,c4,c5,c6:std_logic_vector(3 downto 0);
begin

a0:=i0+i4;
a1:=i1+i2;
a2:=i1-i2;
a3:=i0-i3;
```

```
a4:=a2*c1;
a5:=a3*c2;
b0:=a0+a1;
b1:=a0-a1;
b2:=a2*c3;
b3:=a3*c4;
b4:=a4+a5;
o1:=b1*c5;
o3:=b4-c6;
o0:=b0*c0;
o2:=b2+c3;

end .
```

AR

```
program
in i1,i2,i3,i4,i5,i6,i7,i8,c1:std_logic_vector(3 downto 0);

begin

o1:=i1*c1;
o2:=i2*c1;
o3:=i3*c1;
o4:=i4*c1;
o5:=i5*c1;
o6:=i6*c1;
o7:=i7*c1;
o8:=i8*c1;
o9:=o1+o2;
o10:=o3+o4;
o11:=o5+o6;
o12:=o7+o8;
o13:=o10+c1;
o14:=o11+c1;
o15:=o13*c1;
o17:=o13*c1;
```

```
o16:=o14*c1;
o18:=c1*o14;
o19:=o15+o16;
o20:=o17+o18;
o21:=o19*c1;
o23:=o19*c1;
o22:=o20*c1;
o24:=o20*c1;
o25:=o21+o22;
o26:=o23+o24;
o27:=o25+o9;
o28:=o26+o12;

end .
```

Elliptic

```
program
in sv39,sv38,sv33,sv26,sv18,sv13,sv2,inp, c2:std_logic_vector(3 downto
0);
out sv39_o,sv38_o,sv33_o,sv26_o,sv18_o,sv13_o,sv2_o:std_logic_vector(8
downto 0);

begin

op3 := inp + sv2;
op32 := sv33 + sv39;
op12 := op3 + sv13;
op20 := op12 + sv26;
op25 := op20 + op32;
op21 := op25 * c2;
op24 := op25 * c2;
op19 := op12 + op21;
op27 := op24 + op32;
op11 := op12 + op19;
op22 := op19 + op25;
op29 := op27 + op32;
op9 := op11 * c2;
```

```
sv26i := op22 + op27;
sv26_o := sv26i;
op30 := op29 * c2;
op8 := op3 + op9;
op31 := op30 + sv39;
op7 := op3 + op8;
op10 := op8 + op19;
op28 := op27 + op31;
op41 := op31 + sv39;
op6 := op7 * c2;
op15 := op10 + sv18;
op35 := sv38 + op28;
outpi := op41 * c2;
op4 := inp + op6;
op16 := op15 * c2;
op36 := op35 * c2;
sv39i := op31 + outpi;
sv39_o := sv39i;
sv2i := op4 + op8;
sv2_o := sv2i;
sv18i := op16 + sv18;
sv18_o := sv18i;
sv38i := sv38 + op36;
sv38_o := sv38i;
sv13i := op15 + sv18i;
sv13_o := sv13i;
sv33i := sv38i + op35;
sv33_o := sv33i;

end .
```