A NOVEL APPROACH TO REDUCE SPURIOUS SWITCHING ACTIVITY IN HIGH-LEVEL SYNTHESIS

by

Elie ElAaraj

M.S., Computer Engineering, Lebanese American University, 2008

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

Department of Electrical and Computer Engineering

LEBANESE AMERICAN UNIVERSITY

January 2009

L E B A N E S E   A M E R I C A N   U N I V E R S I T Y

Thesis approval Form

Student Name :	Elie ElAaraj	I.D.: 200104573
Thesis Title :	A Novel Approach to Reduce Spurious Switching Activity in High-Level Synthesis	
Program :	M.S. in Computer Engineering	
Division/Dept :	Electrical and Computer Engineering	
School :	Engineering and Architecture	

Approved/Signed by:

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

Date: January 5, 2009

Plagiarism Policy Compliance Statement

To my parents

Acknowledgment

I would like to thank my advisor Dr. Iyad Ouaiss for his guidance throughout my thesis work. A thanks is also 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 support.

Abstract

Optimizing area and timing have long been considered to be the main design challenges in high-level synthesis. A lot of research has been conducted in this area and many techniques to improve performance have been suggested. However, as design applications become more power sensitive, and with the emergence of portable devices that operate under stringent power constraints, power consumption surfaced as a major issue to consider in the design and optimization processes.

This work studies the effects of binding and scheduling on power consumption in high-level synthesis by analyzing unnecessary switching. The major contribution of this work is to reduce the spurious switching activities in a circuit. For this purpose, all spurious and non-spurious switching inputs in a circuit were identified and many techniques were studied to find the optimal register bindings without inducing any increase in the number of storage elements. Power reduction was attained through altering register bindings using a cool-down simulated annealing approach. In order to test these techniques, a high-level synthesis environment, "Eridanus", was developed and several benchmarks consisting of various complexities have been tested. Using the approach suggested in this work, spurious switching activity was reduced by 40% on average.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

Ever since digital computers emerged and until this day, the number of transistors on a chip has been increasing at an exponential rate. This trend is expected to continue according to Moore's Law. As more devices are being fit on a die, and as multiple integrated circuits are being used in computers, power consumption is becoming a major limiting factor. Power dissipation can be classified as dynamic or static. Dynamic power is the power consumed through the switching of signals while static power is consumed due to the leakage current.

Spurious switching activity is the transitions in the inputs of functional units leading to the computation of unnecessary operations. Power dissipated due to this unnecessary switching is considered dynamic in nature. Many researchers have estimated that spurious switching activity can contribute to 60% of total interconnect power [7] [13].

This said, the design for power has to start at the earliest design stages where the return value is the highest. Nevertheless, conventional high-level synthesis techniques focus on area and latency metrics but lack emphasis on other design metrics. A lot of research has been focused on this problem lately and some have tried to relax the problem and add more resources [3] [4] [6] while others tried to alter the problem and the circuit specification [5] [7] to reduce power consumption. This work explores techniques to reduce spurious switching activity at the binding level. A heuristic technique based on clique partitioning is first explored but found non-feasible paving the way for the use of stochastic search algorithms such as simulated annealing. The suggested approach targets the binding of inputs in storage elements and analyzes the room for improvement in conventional binding techniques. Optimizing variable binding can be done using fast and efficient swapping of inputs using these inputs' lifetimes' information. As the annealing process terminates a good approximate solution to that of the global minima is found.

This work is organized as follows. Chapter 2 explores the literature for the techniques used in high-level synthesis and provides a walkthrough example for the whole synthesis flow. Chapter 3 introduces the Eridanus high-level synthesis environment and its relevance to this work. Chapter 4 discusses the problem of spurious switching activity in depth and provides examples, room for improvement, research work, and finally introduces this work's approach and its strengths. Chapter 5 presents and analyzes the experimental results. Chapter 6 concludes this work and suggests future work and improvements.

Chapter 2: Literature Survey

High-level synthesis also referred to as architectural-level synthesis is the transformation from a behavioral model of a circuit to its structural model. The behavioral model of the circuit is captured through high-level description languages. This description is later translated into a control-data flow graph which is better understood and analyzed using a synthesis tool. After synthesis is done, the structural model of the circuit will describe its datapath using an interconnection of resources, memory elements, and steering logic which route data from resources and memory elements to other parts of the datapath. The structural model of the circuit will also contain a logic-level specification of a controller unit which orchestrates the flow of the data through the datapath.

The synthesis of a specific circuit has a wide range of feasible solutions. Constraining the synthesis problem is the most widely used technique to reduce the solution space size. This can be done by placing bounds on expensive design metrics such as area and timing. Structural models that do not fall within these bounds are often discarded and left unexplored.

High-level synthesis consists of two stages. The first stage is scheduling which assigns operations to time intervals in which they can be executed. The second stage is binding which assigns functional units and memory elements to operations and variables respectively.

For the purpose of illustrating the whole synthesis process, an example has been adapted from one proposed by [1]. This example numerically solves the following differential equation : y'' + 3xy' + 3y = 0 in the interval [0,a] with step-size dx and initial values x(0) = x; y(0) = y; y'(0) = u as suggested by [2].

In a high-level language, an iteration of this example is represented as follows:

```
x1 = x + dx;
u1 = u – ( 3 * x * u * dx ) – ( 3 * y * dx );
y1 = y + u * dx;
c = x1 < a;
```

Before any synthesis is performed on the behavioral description of a circuit, it has to be transformed into an internal format that can be understood, analyzed and easily manipulated by a synthesis tool. The internal format should be able to efficiently represent operations and dependencies among operations. A graph is such a data structure that can be used for this purpose. The structure of the synthesized circuits pertaining to the dependency from one operation to another implies that a directed graph should be used. The directed edges convey the information that one operation consumes the value produced by another operation.

In high-level synthesis this type of graph is known as a control data flow graph, CDFG and is shown in Figure 1. The CDFG is represented by G(V,E) denoting a graph with a set of vertices V and a set of edges E connecting vertices. Throughout this work, the notion of vertices and nodes will be used interchangeably. Two nodes/vertices are added for graph completion purposes. These vertices are the source and sink which have the smallest and the largest indexes respectively. Vertices on the upper part of the graph are predecessors of later vertices if an edge exits between the two vertices. Vertex 2 is the predecessor of vertex 3 and vertex 3 is the successor of vertex 2 in this case. A directed edge exists between those 2 vertices and thus the value produced by vertex 2 will be consumed by vertex 3. The lifetime of this variable starts after vertex 2 and ends as soon as vertex 3 consumes its value.

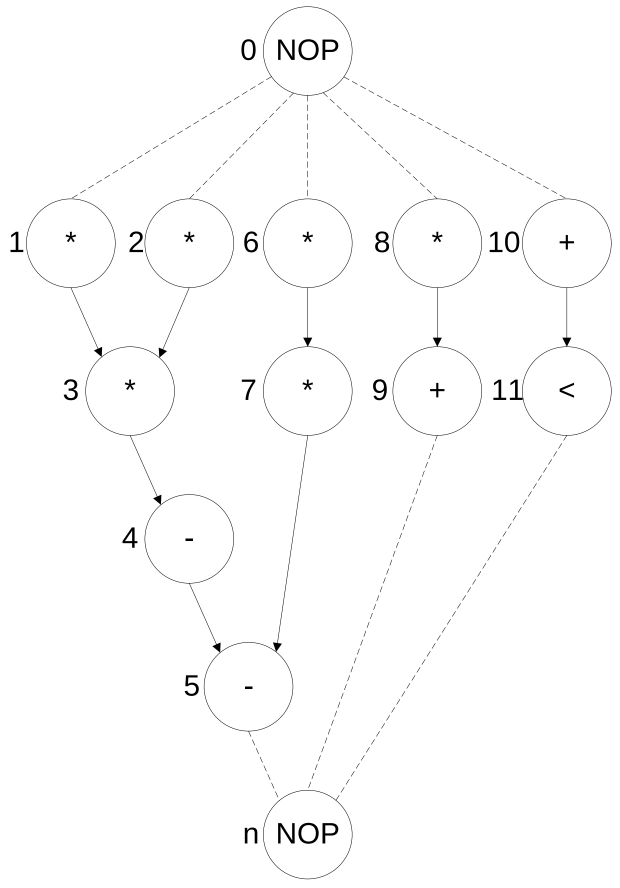


Figure 1: CDFG of the differential equation problem

As mentioned earlier the CDFG will be the main structure used by the synthesis process for the scheduling and binding stages. This is discussed in following sections.

Scheduling

Scheduling is an important part of synthesis since it determines the exact start time of operations and the lifetimes of variables. This step is needed because the CDFG only describes the dependencies among operations and does not contain any timing information. After scheduling is performed, the operations dependencies should remain the same and never change. It can be observed from the differential equation CDFG in Figure 1 that if vertex 2 was scheduled at time 1, then vertex 3 cannot be scheduled at the same time step 1. If so, both vertices would be executed in parallel and this contradicts with the predecessor-successor dependency existing between the 2 operations.

From this discussion, it can be deduced that scheduling determines the concurrency among a circuit’s nodes/operations. It can also be observed that the number of concurrent operations determines a lower bound on the number of resources needed. If fewer resources are used, then the schedule would change and more time would be needed to execute all the operations of the scheduled circuit. One type of scheduling addresses this latter problem. This type of algorithms limits the number of operations that can execute concurrently to the number of available resources but increases the total execution time of the circuit. It can be seen that as the number of resources increase, the total latency of the design decreases and vice versa. This tradeoff is at the heart of the synthesis problem and many solutions are sometimes explored to decide on the tradeoff that best suits the application at hand.

The most known types of scheduling algorithms are presented next. This is in no case a comprehensive discussion of those techniques and one should refer to the literature [2][12] and technical writings concerning those techniques for a detailed discussion.

Unconstrained scheduling

Unconstrained scheduling algorithms are algorithms that are given the freedom to use any number of resources inorder to achieve the minimum scheduling latency.

ASAP scheduling algorithm

ASAP, or as soon as possible, scheduling algorithm assigns the earliest start times of operations. This is done by starting with operations that have no predecessor operations and assigning them start time of 1. All other nodes will have start times based on their predecessors' start times and the time it takes for the predecessor to execute. As an example, assuming that all nodes have a unit delay execution time, a node that has 2 predecessors with start times 1 and 2, will have a start time equals to the maximum of its predecessors' start times plus the unit delay of that predecessor. The ASAP algorithm solves the minimum latency problem due to its unconstrained nature.

The ASAP algorithm is demonstrated using the differential circuit in Table 1 and Figure 2 below.

Table 1: ASAP scheduling steps for the differential circuit.

Step 1	Node 1, Node 2, Node 6, Node 8, Node 10 have no real predecessor nodes and therefore scheduled at time 1. (Note NOP is not considered such a node)
Step 2	All nodes that have all their predecessors scheduled will get scheduled at this time. This step is repeated until all nodes are scheduled.
Step 2 i	Nodes 3, 7, 9, 11 all have their predecessors scheduled and will therefore get scheduled at the maximum of their predecessors' start times. In this case, all of the predecessors of these nodes are scheduled at time 1. Thus Nodes 3,7,9,11 are scheduled at time = 1+1= 2.
Step 2 ii	Node 4 has its predecessors scheduled and will therefore get scheduled at the maximum of their start times. In this case, Node 4 will get scheduled after the start time of node 3 at 2+1 = time 3.
Step 2 iii	Node 5 has its predecessors scheduled and will therefore get scheduled at the maximum of their start times. In this case, Node 5 will get scheduled after the start time of node 4 at 3+1 = time 4.

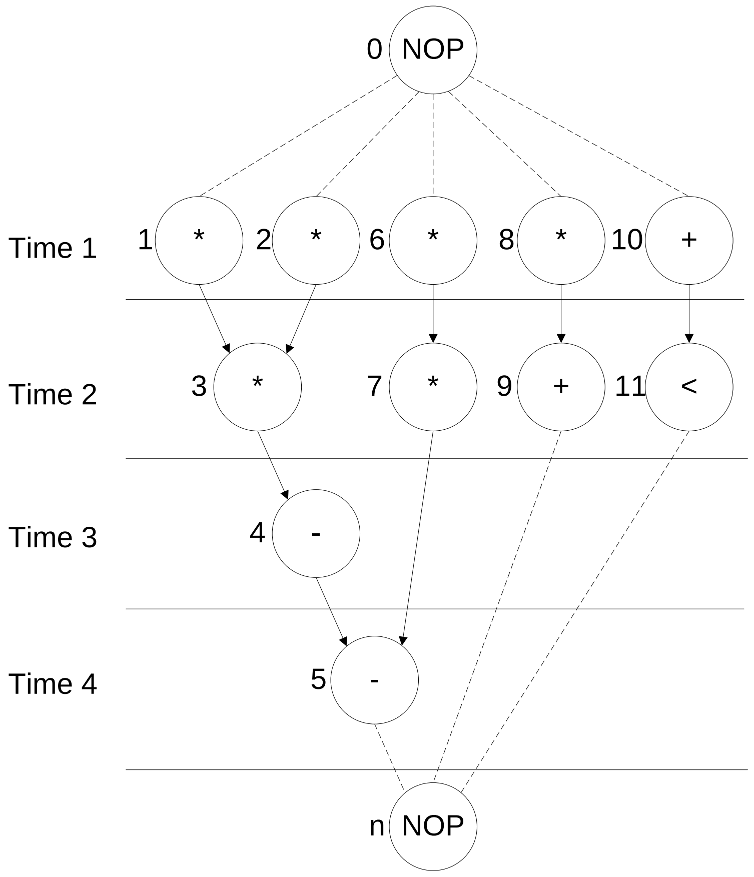


Figure 2: ASAP scheduled differenctial circuit

ALAP scheduling algorithm

ALAP, as late as possible, algorithm explores the range of start times that operations can have while putting a latency bound on the final schedule. Usually the ASAP latency is used to explore the minimum schedule and this schedule is used to find the latest that operations can start. By doing so, the range of start times for all operations is derived by subtracting the ASAP start time from the ALAP start time (ALAP-ASAP) of each operation. This derived numbers, usually referred to as mobility or slack, give information about how much an operation can have flexibility in its execution start time without violating the total latency of the design. As with the ASAP case, the ALAP algorithm is also an unconstrained scheduling algorithm.

The ALAP algorithm is demonstrated using the differential circuit in Table 2 and Figure 3 below.

Table 2: ALAP scheduling steps of the differential circuit

Step 1	All nodes with no successors are scheduled at the maximum allowed time. This is usually set as the upper bound of the schedule latency obtained through ASAP. Nodes 5, 9 and 11 are thus scheduled at time 4. (Note NOP is not considered such a node)
Step 2	All nodes that have all their successors scheduled will get scheduled at this time. This step is repeated until all nodes are scheduled.
Step 2 i	Nodes 4, 7, 8, and 10 all have their successors scheduled and will therefore get scheduled at the minimum of their predecessors' start times. In this case, all of the predecessors of these nodes are scheduled at time 4. Thus Nodes 4, 7, 8, and 10 are scheduled at time = 4-1= 3.
Step 2 ii	Node 3 and 6 have their successors scheduled and will therefore get scheduled at the minimum of their predecessors' start times. In this case, all of the predecessors of these nodes are scheduled at time 3. Thus Nodes 3 and 6 are scheduled at time = 3-1= 2.
Step 2 iii	Node 1 and 2 have their successors scheduled and will therefore get scheduled at the minimum of their predecessors' start times. In this case, all of the predecessors of these nodes are scheduled at time 2. Thus Nodes 1 and 2 are scheduled at time = 2-1= 2.

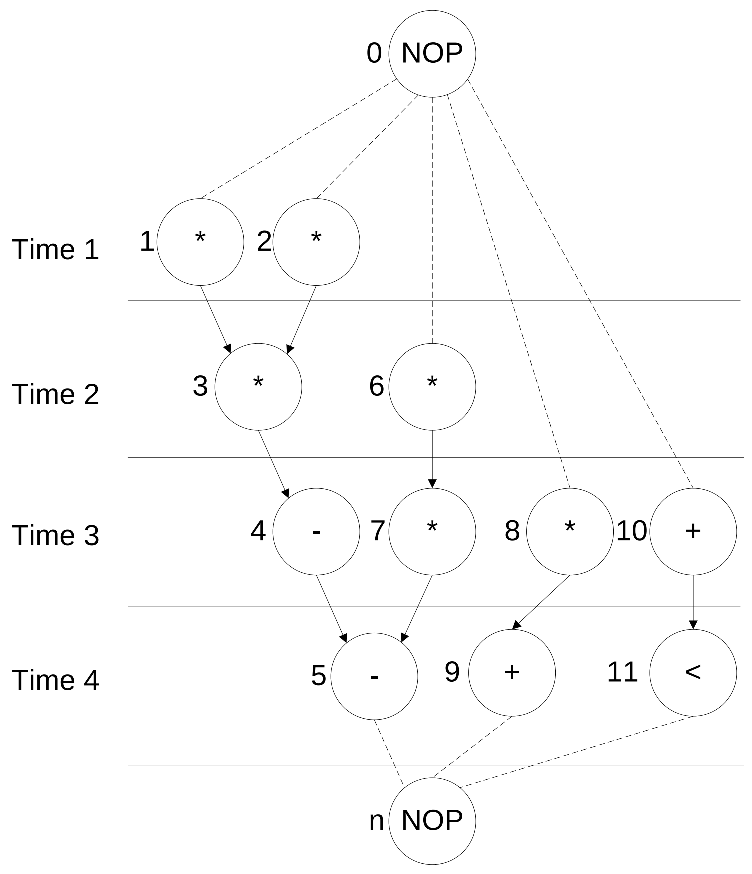


Figure 3: ALAP scheduled differential circuit

Unconstrained scheduling algorithms are the basics of most other scheduling techniques but in most cases constraints will be imposed on design metrics. These constraints could be timing or resource constraints. Time constrained scheduling algorithms are out of the scope of this research and hence will not be discussed. However scheduling under resource constraints is an essential part of this work and will be discussed next.

Constrained Scheduling

Scheduling under resource constraints is intractable and can only be tackled using approximate or heuristic algorithms. This problem is important due to the fact that resource constraints imply area constraints. A designer places a constraint on the circuit area which reflects on resource numbers. As the area is decreased, the number of resources also decreases.

Exact solution methods exist for the constrained scheduling problem nevertheless all methods fail as the problem increases in size. One such method is to model the problem at hand as an Integer Linear Programming model and solve for the optimal solution that reduces the cost which as mentioned earlier could be either time or area.

As the problem size increases heuristics are used to overcome the problem intractability. A family of such algorithms is referred to as List Scheduling algorithms. Again list scheduling can be used to solve the minimal latency resource-constrained and the minimal resource latency-constrained problems. Only minimal latency resource-constrained problems are relevant to this work and will solely be discussed here.

List Scheduling algorithm

The basic idea behind list scheduling algorithms is to select operations based on some criteria and schedule these operations at the current control step only if enough resources exist. Otherwise some of these operations have to be delayed and scheduled at a later control step. The criterion used to select operations (nodes) is what differentiates one family of list algorithms from the other. A priority list, and hence the name list, is used to select nodes with highest priorities based on some cost measure. A common measure used is to sort nodes based on their longest path to the sink and schedule nodes with the longest path. These nodes are assumed to affect the schedule dramatically if delayed and thus have to be scheduled early at the control step or else resource depletion will cause these nodes to get scheduled later and increase the total circuit latency. This is illustrated in the Figure 4. Suppose that only two multipliers are available, if node 6 is given a higher priority than node 2 and therefore scheduled at control step 1, node 2 will have to be delayed until the next control step. Delaying node 2 will also delay node 3 which cannot be scheduled until all its predecessors have been scheduled. Delaying node 3 propagates to all successors and their successors of this node which also happens to be on the critical longest path. This in turn delays nodes 4 and 5 and the total latency of the design is now 5 instead of 4. This shows that nodes with the longest distance to the sink should be scheduled first. Thus the distance to the sink can be used as a measure to indicate priority.

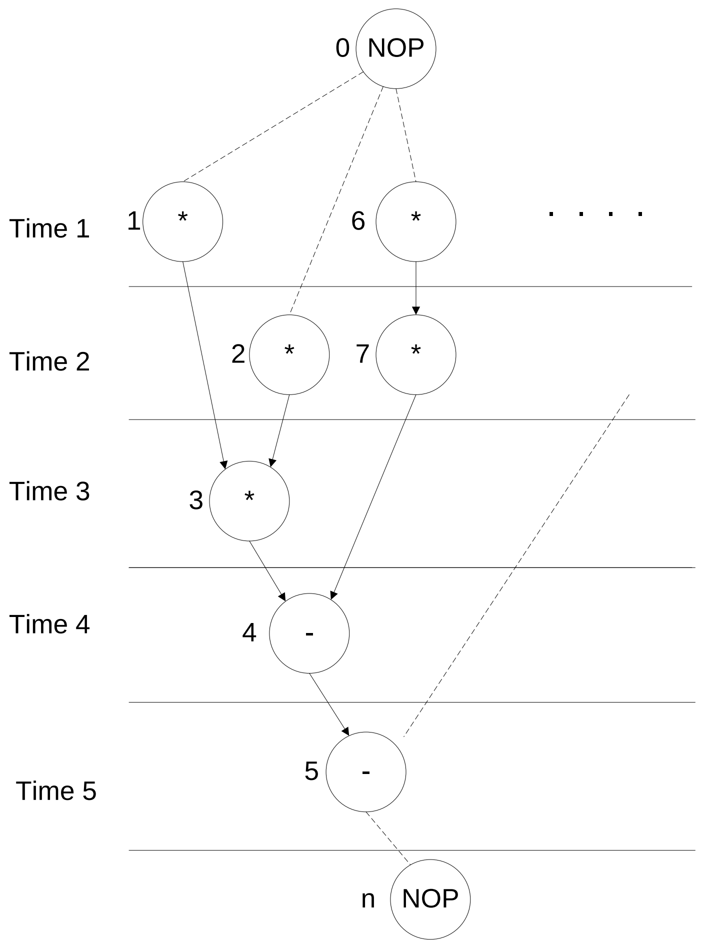


Figure 4: Delaying nodes on the critical path increases total latency

This sets the stage for the list scheduling algorithm. As shown in the pseudo code [2] in Table 3 below, the scheduling loop is repeated until all operations have been scheduled. For each resource type, the inner loop is repeated and candidate nodes are determined. Candidate nodes are nodes that can be scheduled at the current control step *l*. Nodes are considered as candidates if all their predecessors have been scheduled and according to the priority measure discussed earlier. The next step is to determine how many unfinished operations are still executing in the current control step. These operations could be operations that require multiple cycles to execute such as division or multiplication operations. If there are such operations, then this means that a resource of that type is still occupied executing that unfinished operation. All this means that less resources than what is available in the resource bag can be used. The next step is to select the set of operations which are a subset of the candidate operations such that they satisfy this availability constraint. For example if 4 adders are in the resource bag, and 2 unfinished addition operations are still executing in the current control step, only 2 new operations can be scheduled at this control step. After these nodes are scheduled, the control step is incremented to schedule nodes at the next control step and the same process repeats again.

A similar technique is also employed for priorities. This technique is based on the asap and alap schedules. As mentioned earlier, the mobility or the slack due to the difference in both schedules is an important indicator. This indicator can be used by the list scheduling algorithm to select nodes that have zero slacks, i.e. if moved or mobilized will cause an increase in latency, and schedule them first. Thus the lower the slack, the higher the priority will be.

Table 3: List scheduling algorithm

```
LIST_L($G_s(V,E)$,a) {
$l$=1;
    repeat {
        for each resource type $k = 1, 2, \ldots, n_{res}$ {
            Determine candidate operation $U_{l,k}$;
            Determine unfinished operation $T_{l,k}$;
            Select $S_k \subseteq U_{l,k}$ vertices such that $|S_k| + |T_{l,k}| \leq a_k$;
            Schedule the $S_k$ operations at step $l$ by setting $t_i = l \; \forall i : v_i \in S_k$;
        }
        $l = l + 1$;
    }
    until ($v_n$ is scheduled);
    return (t);
}
```

To show how list scheduling algorithms work, the same differential equation circuit will be used. In this example we will assume that 3 multipliers and 1 ALU functional units are available. The execution delays of the multipliers and ALU are 2 and 1 respectively [2].

If the priorities are based on the weight of the longest path to the sink, the operations will be scheduled according to Table 4. The scheduled graph is shown in Figure 5.

Table 4: List scheduling steps of the differential circuit

Multiplier	ALU	Start time
v1, v2, v6	v10	1
OCCUPIED	v11	2
v3, v7, v8	IDLE	3
OCCUPIED	IDLE	4
IDLE	v4	5
IDLE	v5	6
IDLE	v9	7

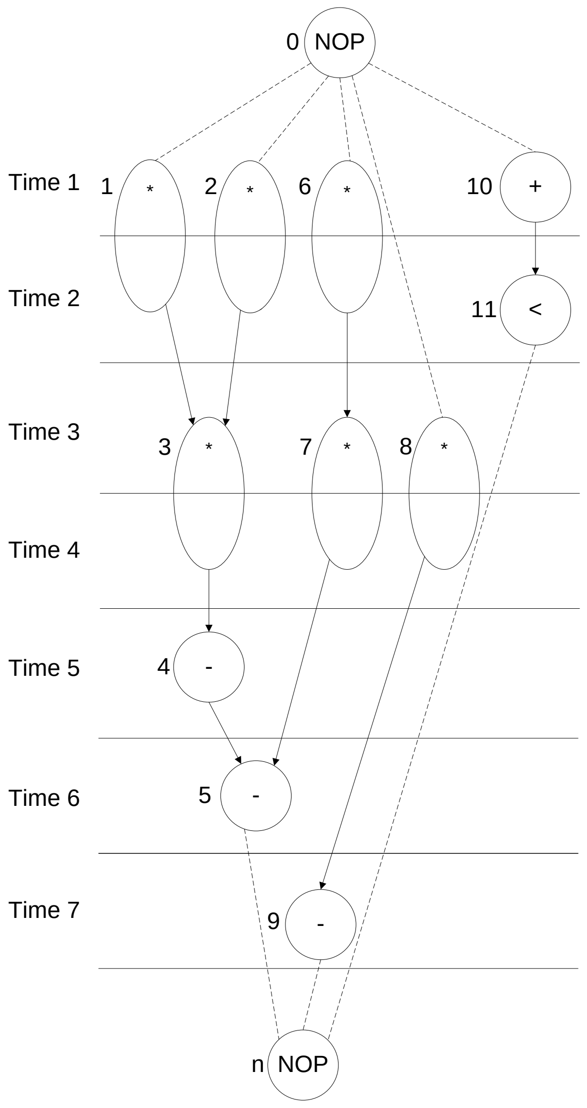


Figure 5: List scheduled differential circuit

The next step after the scheduling stage is the binding stage. The binding stage could have been done before the scheduling stage as well. The flow suggested here is the flow decided when designing and implementing Eridanus. Eridanus will be discussed in depth in a later section of this thesis work.

Binding

Binding is the mapping between operations and resources. Resources are mainly of two types, functional units and storage elements. Whenever constraints are placed on area, resource sharing will become inevitable. Two operations can share a resource if they are non-concurrent. Concurrent operations should exist at the same time and therefore separate resources must be allocated for those types of operations.

When performing binding after resource-constrained scheduling, this process will still affect the area of the final design. Although a large percentage of the area has been determined through constraining the number of resources, many other resources that have not yet been determined at this stage will also play a role in the overall area and performance of the design. These are basically storage elements, steering logic and controller units.

The following rules dictate when operations can be bound to the same resource:

- Operations can be bound to the same resource if they are not concurrent meaning they do not execute simultaneously in that one operation starts after the other ends. Two operations are not concurrent when they are mutually exclusive as in the case of branches.
- Operations should be compatible with the resource types. This means that the resource they are binding to should be able to implement those operations.

This set of rules is used to derive compatibilities between operations. Two operations are said to be compatible if they satisfy those two rules. If two operations are compatible, then they can be bound to the same resource. When all compatibilities are derived, a new graph is constructed that depicts the global compatibilities among all operations of the circuit at hand. This graph is referred as the compatibility graph of the circuit. It can be deduced that for each resource, a separate compatibility graph is constructed. Nodes that are mutually connected by edges represent mutually compatible operations i.e., a clique as [2] suggests. Minimizing the number of needed resources, translates to minimizing the number of cliques in the compatibility graph which is widely known as the clique cover number.

An opposite technique to getting the compatibility graph can be also used. This technique suggests building a conflict graph out of conflicting operations. Operations are said to be in conflict if they are not compatible. It can be deduced that the two graphs, the compatibility and the conflict graphs, are complements of each other. To solve for the minimal number of resources using a conflict graph, the graph coloring technique can be used. Graph coloring tries to color all connected nodes with different colors such that no two connected nodes have the same color. In this case each different color used denotes a resource. Minimizing the number of colors actually minimizes the number of resources.

The clique partitioning and vertex coloring problems are intractable [9] for general graphs and exact and heuristic solution methods have been proposed.

To illustrate how clique partitioning can solve the binding problem, we will use the same differential equation circuit with a predefined schedule. This is shown in Figure 6. The compatibility graphs for the multiplier and the ALU units of this example are shown in Figure 7.

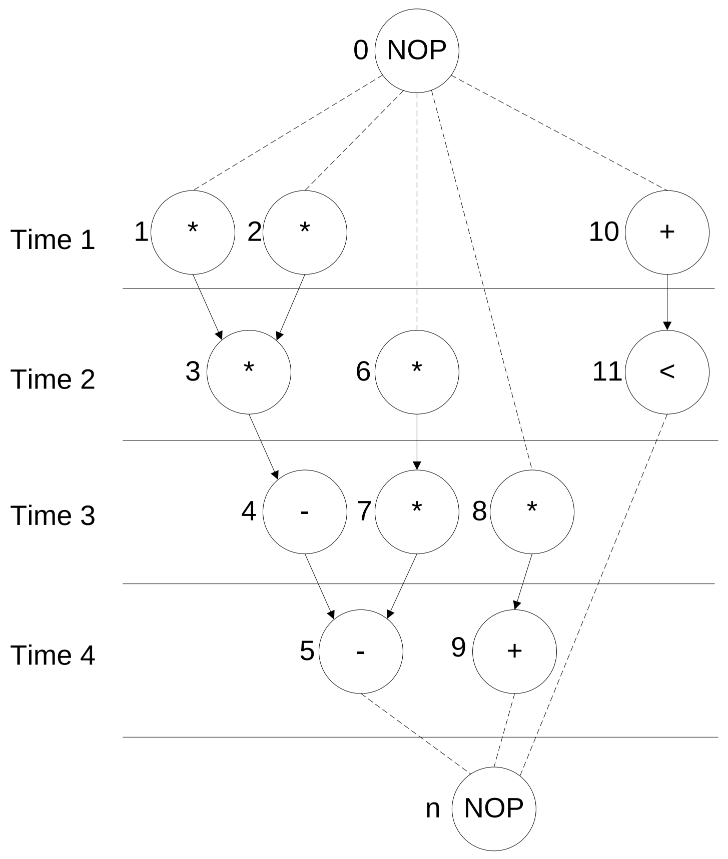


Figure 6: Scheduled CDFG

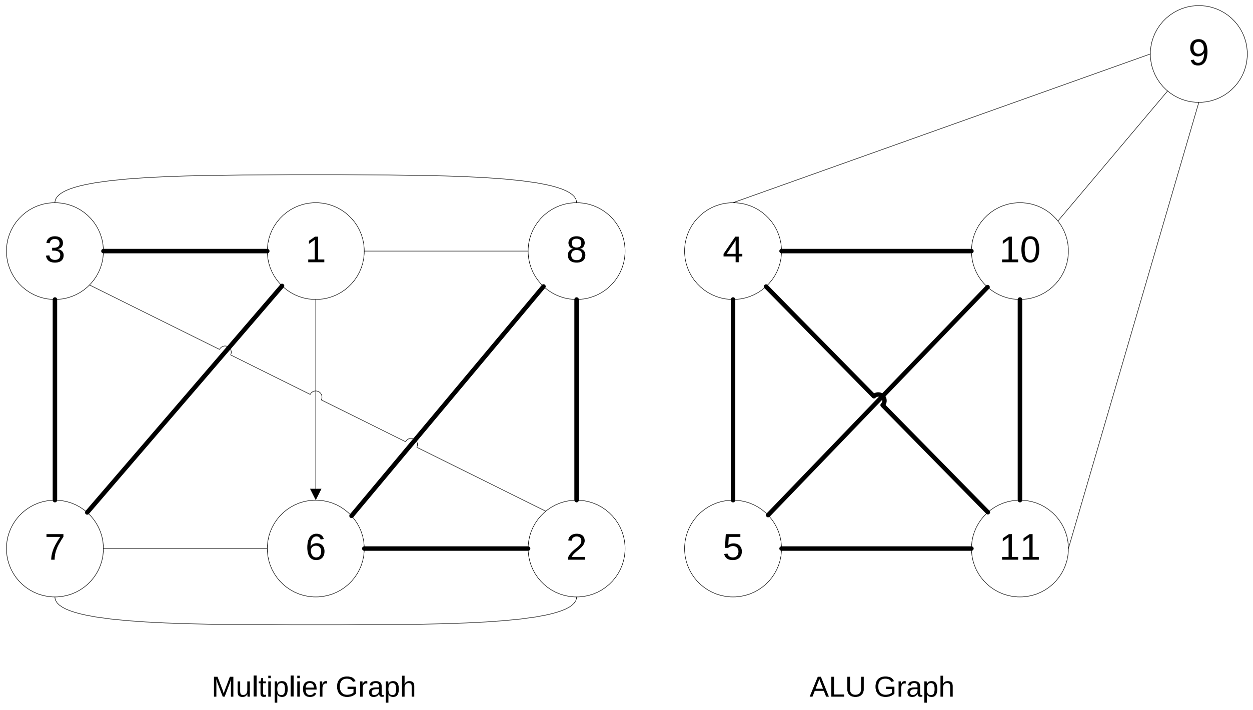


Figure 7: Multiplier and ALU compatibility graphs

To solve for the minimal number of multiplier, the maximal clique cover is found. The cliques {v1, v3, v7} and {v2, v6, v8} in the multiplier graph cover all the vertices in 2 sets. This implies that 2 multipliers are needed and that the operations implemented in the first multiplier are v1, v3 and v7 while the second multiplier implements v2, v6, and v8. The same applies to the ALU compatibility graph. One clique covers {v4, v5, v10, v11} while the other contains the single vertex v9. This also implies that two ALUs are needed. The first ALU implements v4, v5, v10 and v11 while the second ALU implements v9.

Analyzing the results obtained from clique partitioning we can see that v1, v3 and v7 are all non concurrent operations and have been scheduled according to Figure 6 in different control steps. This verifies that they can be bound to the same multiplier. The same applies to the second multiplier. As for the first ALU, v4, v5, v10, and v11 are also scheduled at different control steps and each pair of operations satisfy a predecessor-successor condition which implies that they are non concurrent.

Chapter 3: Eridanus

Many high-level synthesis tools are available in the market in commercial and educational versions but none allows the researcher to easily alter the specifications of the problems and the tool's internals or try to target the synthesis tool towards improving parameters that it is not built to improve. Therefore there was a need to find a tool that does all the above and at the same time targets the whole synthesis flow from accepting a user friendly code as input and generating the synthesized output while keeping the user/researcher seemless of the underlying techniques. Many educational tools were researched and tested for the needed functionalities such as Altera Quartus, but none of those tools provided the needed flexibility.

Due to all those factors, the Eridanus Synthesis project was born. The project aimed at creating a flexible educational environment to synthesize high-level descriptions of circuits. Eridanus provides the basic core functionalities of scheduling algorithms as well as binding algorithms. It also provides the flexibility that other commercial and educational tools lack. This is represented by the extensions to the current code base. The researcher can at any time add new scheduling and binding algorithms and view results of those new algorithms. This makes Eridanus a tool that can be targeted towards improving any synthesis factor whether it is area, power, or a multiple of factors at the same time. After scheduling and binding are performed, the tool then creates a VHDL representation out of the high-level PASCAL-like circuit representation. A datapath and a controller are generated with all the supported components and the required bitwidth sizes.

At the end of the synthesis flow, the tool allows to incorporate optimization algorithms such as simulated annealing that allows the user to further alter the returned results. These new results can then be either used as the final results or compared to the initial result to identify the room for improvements.

Like all CAD tools, and in the design phase of Eridanus, many decisions were made concerning the synthesis flow of the tool. The flow was finalized as follows:

1. High-Level Circuit representation
    a. Syntax highlighting
    b. Language syntax and semantics

2. Creating the Dependency Graph
    a. Expression Parsing
    b. Creating Eridanus Nodes and Edges
    c. Creating Functional statements
    d. Creating Diverge statements
    e. Creating Converge statements

3. Scheduling
    a. ASAP scheduling
    b. ALAP scheduling

    c. LIST scheduling
    d. Add-ons

4. Binding
    a. Generating Compatibility Graphs
    b. Clique Partitioning Algorithm
    c. Left Edge Algorithm
    d. Add-ons

5. Generating Circuit Schematics

6. Generating VHDL code
    a. DataPath hardware structural description
    b. Controller finite state machine description

High-Level Circuit Representation

In an effort to make Eridanus an easy tool to use, a user-input format that resembles the latest prevailing high-level languages was selected. This allows researcher and users to quickly start using Eridanus instead of learning a new language from scratch. The custom language selected for the user-input closely resembles the PASCAL programming language. This allows the user to write input file with functional statements like addition as well as control statements in the form of if-else statements. The syntax and semantics of this language will be explained in detail later in this section.

When a custom language is created, there should be a mechanism to read and understand that particular language. For this purpose, a scanner and parser were created for this language. The Eridanus Lexer scans the user-input for language specific word constructs that can be understood by the Eridanus Parser. The parser receives those word constructs and, depending on the word, performs an action. These actions will be further explained in the "Creating the Dependency Graph" section. In other words, the lexer/scanner reads the user-input file, and the parser understands what should be done with the read statements.

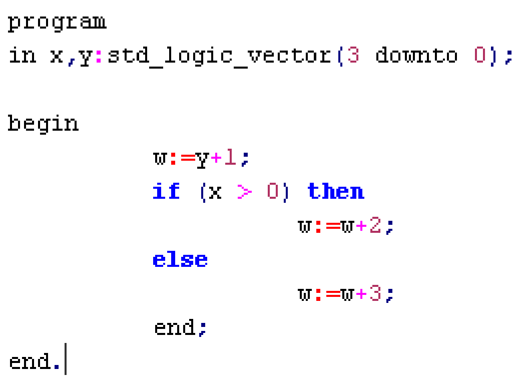
Syntax highlighting

The Eridanus tool offers a custom syntax highlighter. This option helps identify expressions and makes reading large input files an easier task for the user. This following figure shows a raw text user-input file and the Eridanus highlighted version of the same file.

Table 5: Raw-Text file vs. Eridanus syntax highlighting

```
program
in x,y:std_logic_vector(3 downto 0);

begin
        w:=y+1;
        if (x > 0) then
                w:=w+2;
        else
                w:=w+3;
        end;
end.
```



```
program
in x,y:std_logic_vector(3 downto 0);

begin
            w:=y+1;
            if (x > 0) then
                        w:=w+2;
            else
                        w:=w+3;
            end;
end.
```

The highlighted version of the input file shown in Table 5 clearly highlights the code delimiters such as assignment, addition, and comparison operators as well as constants.

Language Syntax and Semantics

The syntax of the custom input-language is similar to the pascal syntax. To create a new user-input file, the first statement of the input should be the ***program*** directive. This tells the Eridanus system that the current file is a program file that contains statements and other words that should be parsed for further details. As in all HDL languages, and to keep the high-level language as close as possible to its hardware counterpart, inputs and output are declared afterwards. A PASCAL similar input/output declaration is used. Instead of using the ***var*** keyword to declare variables, the ***in*** and ***out*** keywords are used to declare inputs and outputs respectively.

To declare more than one input or output, a comma is used to separate the multiple inputs/outputs. The following partial statement shows how multiple inputs can be declared at once.

- *in x , y* – This partial statements shows how x and y can be declared as inputs in the same statement.

To identify the size or width of the input/output, the colon delimiter should be used. This in turn makes the custom language similar to both PASCAL and most HDL languages. To declare the input/output type, syntax similar to HDL has been adopted. This lets the user explicitly specify the type and the size of the input/output. To make things simpler, the std_logic_vector type is the only supported type for this early realease of Eridanus. Using this type, the user can still declare an input of type std_logic_vector by declaring a vector of size 1 as follows, std_logic_vector(0 downto 0). The following statements shows how to declare inputs with vector types and specify the size of that vector.

- *in x , y : std_logic_vector(3 downto 0)* – This statement shows how to declare x and y as inputs with a vector type of size 4.

After all the inputs and outputs have been declared, the user can start describing the circuit in a behavioral high-level description. The same ***begin*** keyword as PASCAL and most HDL languages is also used at this point. This keyword tells the Eridanus system that the circuit description begins at this point. To tell the systems that the description has ended, the "***end.***" keyword is used. Any block of code placed within the begin-end statements is considered as the circuit description. This description can constitute of many different statements that will be explained shortly.

Using the same example in Table 5, an explanation of a circuit description is given. The first statement assigns the value of y+1 to w. The following expression shows how this is done.

- w := y + 1; – This expression shows how w is assigned the value of y+1.

The “:=” assignment operator is used to assign values to variable or outputs. This conforms with the design objective of keeping the syntax as close to PASCAL as possible. The same operator is used in HDL languages to assign values to variables. The next important delimiter operator in the expression is the functional operator. This is represented by one of the supported functions in Eridanus. In this example expression, it is the addition operator “+”. Eridanus currently supports addition (+), subtraction (-), and multiplication. These in turn are synthesized into adders, subtractors and multipliers respectively.

The expression shown earlier contains a constant of value 1. When the Eridanus scanner reads that value, it automatically understands that a variable is being added with a constant. In all cases, it assigns the final value of the left hand side of the expression to the right hand side. To begin a new expression, the semicolon (;) delimiter is used which informs the scanner and parser that the current expression has ended and a new one is in progress.

Another type of expression supported in Eridanus is the Converge-Diverge expression. This expression is represented by if-else block statements. To indicate the end of the if-else block an ***end*** statement should be used. The mechanisms behind parsing and interpreting the if-else block will be explained in detail in the next section, “Creating the Dependency Graph”.

We will use the following example to explain the syntax of the if-else block:

```
if (x > 0) then
        … expressions
else
        … expressions
end;
```

1. Use the ***if*** keyword to start the if-else block
2. Use open-parenthesis to start the condition to be checked upon.
3. Write single comparison condition. Multiple comparisons are not directly supported in Eridanus. To compare and check on multiple conditions, use nested if-else blocks. An example of this technique is shown later.
4. Use close-parenthesis to end the condition
5. Write the block of if-expressions which will get executed when the condition is satisfied.
6. Use the ***else*** keyword to end the if-statement and start the else block.
7. Write the block of else-expressions which will get executed when the condition is dissatisfied.
8. End the if-else block using the ***end*** keyword as mentioned earlier.

Eridanus does not directly support multiple comparisons or condition checks. Inorder to do that, the user must explicitly write multiple nested if-else statements. To illustrate this, the following example shows a direct multiple condition and its counterpart in Eridanus.

Table 6: Direct multiple comparisons vs. Eridnaus nested if-else blocks

```
if ( x > 0 and x < 10 ) then
   …
else
   …
end;
```

```
if ( x > 0 ) then
    if ( x < 10 ) then
       …
    else
       …
    end;
else
    …
end;
```

Creating the Dependency Graph

After laying the foundation for the Eridanus input-file format, and after the system checks for the file format compatibility, the tool starts scanning and parsing the input for token. Tokens can be either characters or words indicating variables such as inputs and outputs. They can also take the form of numbers which will be considered as constants. Another important type of a token is the operation and delimiter tokens. Operation tokens are the symbols that represent operations such as addition (+). Delimiter token are symbols which inform the parser that an action should be performed. The assignment operator (:=) is an example of a delimiter. This delimiter informs the parser that the evaluated left hand side of the expression should be assignment to the right hand side of the expression.

Eridanus most important subsystems are the Scanner and the Parser. These two subsystems work hand in hand to read and interpret the user input. After the input is read by the scanner, the parser interprets the semi expressions and builds the circuit dependency graph gradually as the user input is being read. Walk-through example of this procedure will be shown next. The same example shown in Table 5 is used for consistency throughout this discussion.

As the scanner reads the input file, it identifies token and passes them to the parser for interpretation. The first token encountered by the scanner is the ***program*** token. This token is passed to the parser which in turn identifies that a new input file describing a particular circuit is at hand. For this, the parser creates a new dependency graph for that circuit. After the program token, and according to the same example, the ***in*** token is encountered. The parser interprets each token scanned after the ***in*** (as well the ***out***) token and creates a corresponding node for that token. The following expression shows how the scanner and parser work simultaneously to build the dependency graph.

in x,y:std_logic_vector(3 downto 0);

This expression is interpreted in the following sequence:

1. Scanner reads the ***in*** token and identifies that each following token corresponds to an input.
2. Scanner reads x, passes it to the parser which in turn creates an input node with the identifier being “x”.
3. Scanner reads semicolon indicating that another token follows.

4. Scanner reads y, passes it to the parser which also creates an input node with a “y” identifier.
5. Scanner reads a semicolon indicating that the input token have finished. This tells the parser that no more input token are available and any following token are setting tokens.
6. Scanner reads ***std_logic_vector*** and informs the parser that the types of the previous tokens, being x and y, should be the scanned token or std_logic_vector in this case.
7. Scanner reads open-parenthesis indicating that more tokens are to come.
8. Scanner reads 3, and passes this value as a number to the parser. This number is understood by the parser as the upper index of the vector.
9. Scanner reads ***downto***, and passes it to the parser. The parser interprets this token and realizes that the downto token gives a range property to the vector. The parser then awaits for the lower index of the vector.
10. Scanner reads 0, and this value to the parser. This number is then interpreted as the number the parser was waiting for which is the lower index of the vector.
11. Scanner reads the close-parenthesis meaning that the range tokens have ended.
12. Scanner reads a semicolon, and passes it to the parser. This token is considered a line delimiter and therefore the parser realizes that the current line has ended and the expression contains no more token. The parser now expects tokens for the next expression.

begin – This token is read by the scanner and passed to the parser which identifies the beginning of the circuit description section.

w:=y+1;

This expression is interpreted in the following sequence:

1. Scanner reads w, passes it to the parser which identifies w as being an intermediate temporary variable because it was neither declared as an input nor an output. A node is created having w as an output. This node type is yet to be decided depending on the upcoming token.
2. Scanner reads the assignment operator :=, and passes it to the parser which recognizes that this token means that the evaluated version of the following token is to be assigned to the w variable.
3. Scanner reads y, and passes it to the parser. The parser here checks if y has been declared earlier and identifies it as an input. Since the value of y will directly affect the value of w, a dependency edge is created between y and w. This edge is a directed edge incident from y and incident to w. This is shown in Figure 8.
4. Scanner reads the addition operator +, and the parser identifies that the operation is an addition operation. The parser sets the w node type as an addition node and awaits the next input for the w node.
5. Scanner reads the constant value 1, and the parser creates a constant node with a value 1. This node is then set as the second input of the w node and an edge is created accordingly as with the case of y.

6. Scanner reads the semicolon delimiter and the parser finalized the current expression and awaits the tokens of the next expression.

The following figure shows the result of scanning and parsing this expression.

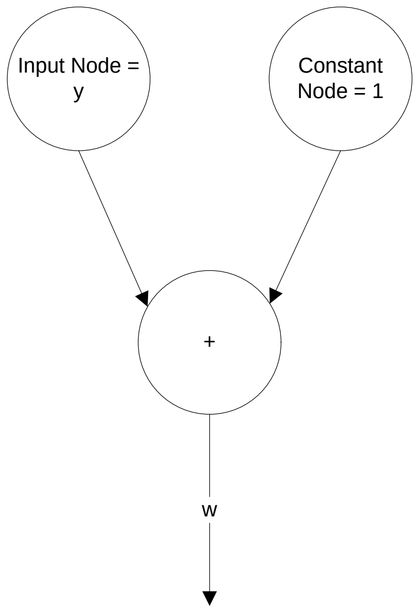


Figure 8: Visual representation of parsing w:=y+1.

```
if (x > 0) then
        w:=w+2;
else
        w:=w+3;
end;
```

The above if-else block is interpreted in the following sequence:

1. Scanner reads the ***if*** token and passes if to the parser. The parser creates a diverge node whose type if yet to be determined depending on the type of the condition in the if-statement. Diverge nodes in Eridanus are always mapped to if-else statements because separate if-statements are not supported. If the user wants to write an if-statement without an else statement, then she has to provide and empty else block.
2. Scanner reads the open-parenthesis token which is identified by the parser as the start of the if-condition.
3. Scanner reads the x token, which is identified by the parser as an input. The result of the diverge node is affected by the value of x and therefore the parser creates a directed edge whose head is the x input node and whose tail is the diverge node.
4. Scanner reads the comparsion operator >, and passes it to the parser. The parser recognizes the operator as existing inside a condition expression and thus assigns the diverge node the GREATER THAN type.
5. Scanner reads the constant value 0, and the parser creates a constant node for that value. Then it creates a directed edge from the constant node to the diverge node.
6. Scanner reads the close-parenthesis delimiter token and the parser identifies the end of the comparison expression.
7. Scanner reads the ***then*** token and the parser awaits the following expressions which will be descendents of the diverge node. The sequence of operations performed so far is shown in Figure 9.

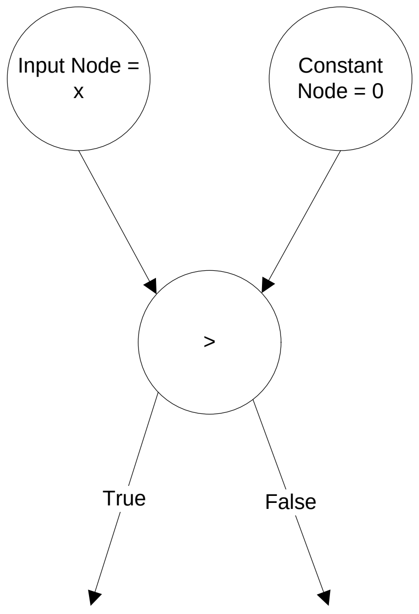


Figure 9: Partial visual representation of the if-else block

8. Scanner reads and interprets all expressions in the if-block until no more expressions are available. All those expressions are parsed in the same way as regular expressions. The only difference is that all those expressions will be tagged as belonging to a path that has a unique number. This number (0) denotes that the node should be placed on the true path or branch.
9. Scanner scans the else token, and the parser identifies that the if-block statements have ended and the next set of expressions are the else-expressions which get executed when the condition is not satisfied. All those expressions are parsed in the same way as regular expressions. The only difference is that all those expressions will be tagged as belonging to a path that has a unique number. This number (1) denotes that the node should be placed on the true path or branch. The numbers 0 and 1 are the unique path number for if-else blocks and any other

numbers are invalid because for example Eridanus does not support if-elsif-else blocks.

10. Scanner reads the end block and the parser recognizes that all else-statements have ended. The parser begins searching for common variables. Common variables are variables that have been edited or modified in both paths/branches. If such variables exist, the parser will create a converge node for each common variable. In this example, the variable w has been modified in both the true and the false paths.

The following figure shows a visual representation of scanning and parsing the if-else block.

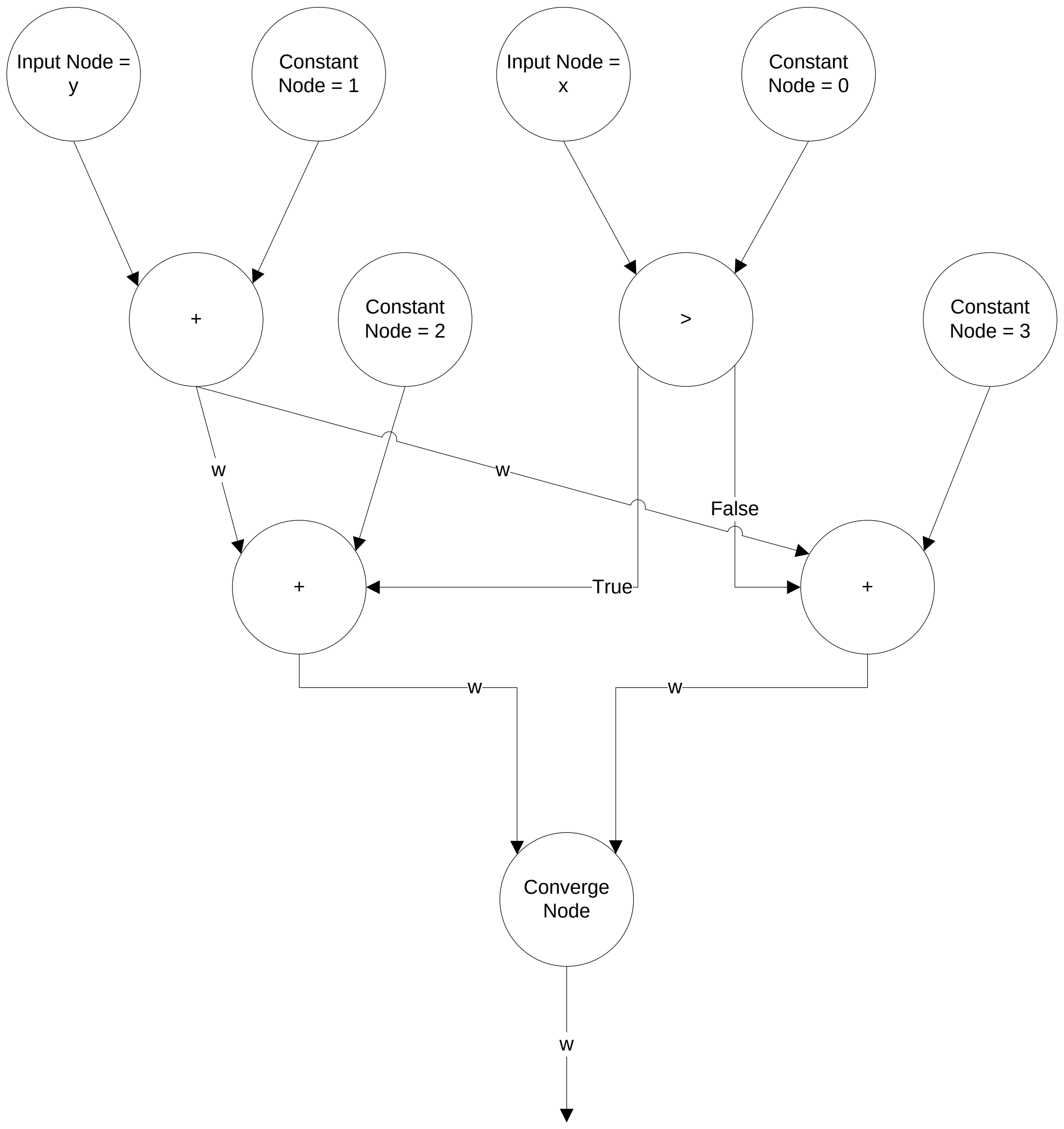


Figure 10: A visual representation of parsing the if-else block in Eridanus.

Scheduling

After the dependency graph (control data flow graph) has been created, the next step in the execution flow is the scheduling task. Eridanus offers three options for scheduling. They are listed as follows:

ASAP scheduling

As mentioned earlier in “Literature Survey”, the ASAP algorithm schedules the CDFG and its nodes as soon as possible. The same walk-through example as in Table 5 will be used here for illustration.

Whenever an input node or a constant is encountered in the scheduling phase, this node is skipped. The remaining nodes are scheduled normally.

Assuming that all inputs and constants have been iterated and skipped, the scheduler searches for nodes with no predecessors and encounters the w:=y+1 operation. The operation is to be scheduled at the soonest after the maximum of its parents control steps. In this case, both it’s parents are inputs and constants which will have control step equal to zero. Thus the scheduler schedules the operation at control step 0 + 1 = 1. The complete scheduling of this example is listed below in Table 7 and represented in Figure 11.

Table 7: ASAP scheduling steps

Node/Operation	Control Step	Reason
x, y	0	Input nodes are scheduled at step 0 at all times.
w:=y+1	1	Its predecessor y is scheduled at step 0. Thus w is scheduled at 0+1 = 1
x>0	1	Its predecessor x is scheduled at step 0. Thus x>0 is scheduled at 0+1 = 1
w:=w+2	2	It predecessor w is scheduled at step 1 and x>0 is also scheduled at 1. Thus the maximum of the 2

		steps is 1. The new w is scheduled at 1+1 = 2.
w:=w+3	2	It predecessor w is scheduled at step 1 and x>0 is also scheduled at 1. Thus the maximum of the 2 steps is 1. The new w is scheduled at 1+1 = 2.
Converge w	3	The maximum of its predecessors' control steps is 2. Therefore the converge node is scheduled at 2+1 = 3.

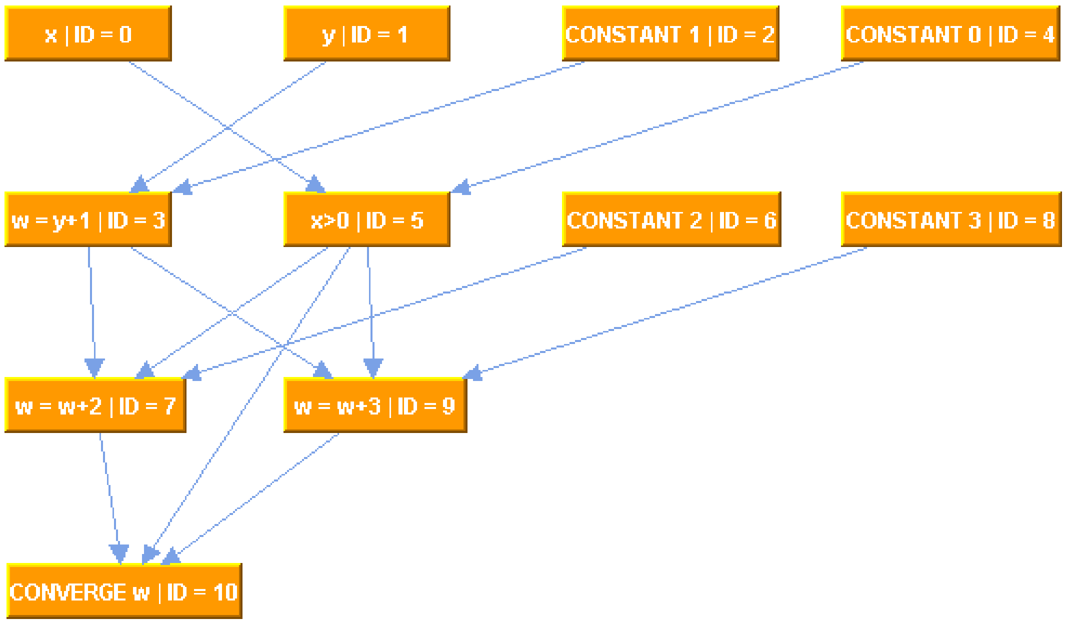


Figure 11: ASAP scheduling

ALAP scheduling

As mentioned earlier in "Literature Survey", ALAP algorithm schedules the CDFG and its nodes as latest as possible. The same walk-through example as in Table 5 will be used here for illustration.

As is the case with ASAP, whenever an input node or a constant is encountered in the scheduling phase, this node is skipped. The remaining nodes are scheduled normally.

Again assuming that all inputs and constants have been iterated and skipped, the scheduler searches for nodes with no successors and encounters the CONVERGE w operation. The operation is to be scheduled at the latest (control step 3) which is the final control step of the ASAP scheduling algorithm. The next step is to search for operations and schedule them at the minimum - 1 of its successors'' control steps. The complete scheduling of this example is listed below in Table 8 and represented in Figure 12.

Table 8: ALAP scheduling steps

Node/Operation	Control Step	Reason
Converge w	3	Node with no successors scheduled at the final ASAP step.
w:=w+2	2	Has one successor only at step 3. Thus scheduled at 3-1 = 2.
w:=w+3	2	Also has one successor only at step 3. Thus scheduled at 3-1 = 2.
w:=y+1	1	The minimum of its successors' control steps is 2. Thus scheduled at 2-1 = 1.
x>0	1	The minimum of its successors' control steps is 2. Thus scheduled at 2-1 = 1.
x, y	0	Input nodes scheduled at step 0.

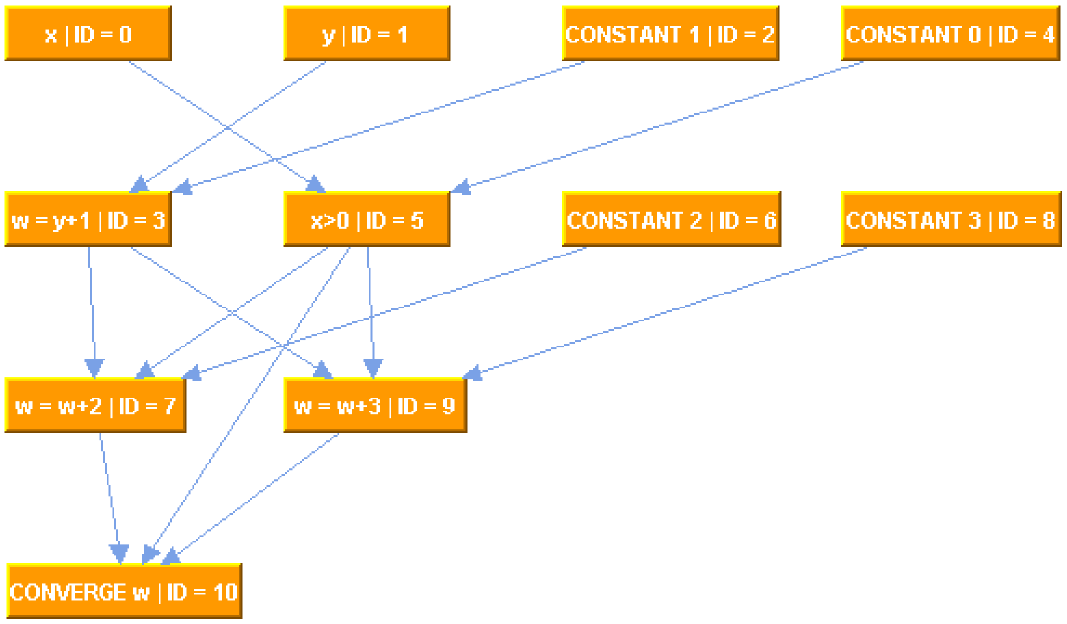


Figure 12: ALAP scheduling

LIST scheduling

LIST scheduling algorithm schedules the CDFG according to a resource bag and thus is considered a resource constraint technique. The same walk-through example as in Table 5 will be used here for illustration.

For the LIST scheduling algorithm to work, the ASAP and ALAP algorithms have to be executed first. Those two algorithms create a gap which can be used by the LIST scheduler to decide on the best schedule given the resource constraints. This gap is represented as the ALAP control step – ASAP control step.

The complete scheduling of the same example is listed below in Table 9 and represented in Figure 13. Further details of this algorithm and its implementation are available in [16].

Table 9: LIST scheduling steps

Node/Operation	Control Step	Reason
x, y	0	Input nodes are scheduled at step 0 at all times.

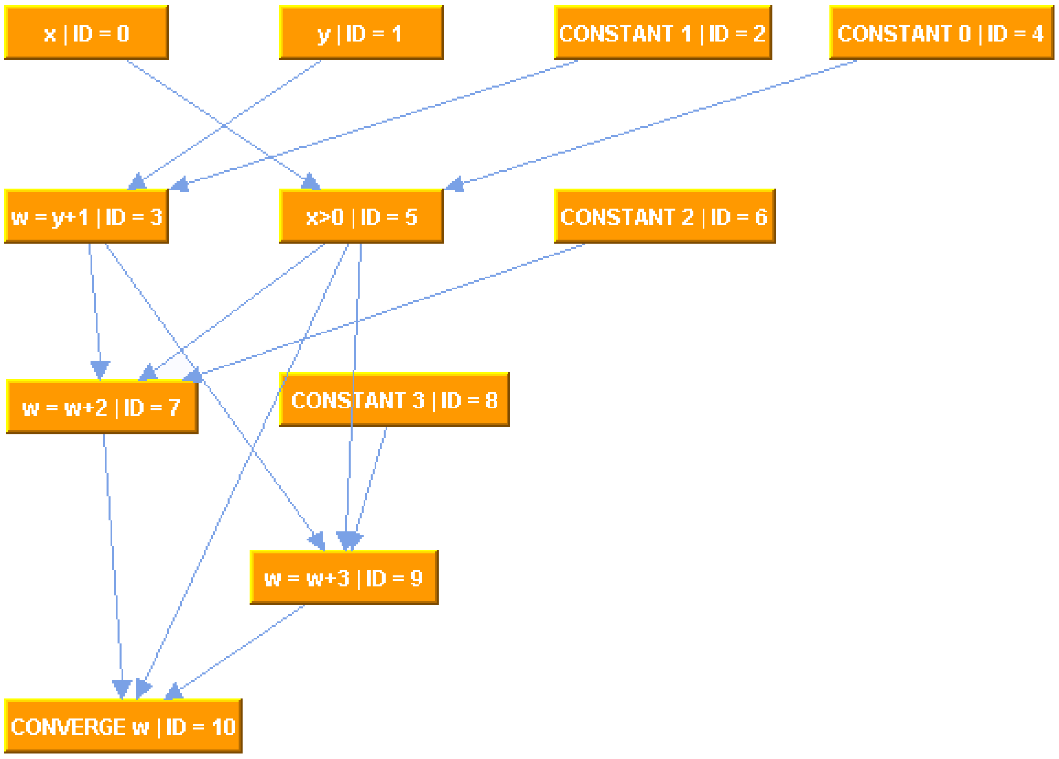


Figure 13: LIST scheduling

Adds-Ons

The three previous scheduling algorithms are initially available for the users to select from. Nevertheless, the researcher can at any time add a new scheduling algorithm and perform scheduling on a CDFG with that algorithm. This stems from the fact that the Eridanus tool was designed to be flexible and to incorporate later additions as easily and transparently as possible. All that the user has to do is to extend the Eridanus Algorithm core services and implement the needed algorithm.

Binding

After scheduling the CDFG, Eridanus will move to the next stage in the synthesis flow. For binding, the user can select one of the two predeveloped algorithms, Clique partitioning and Left Edge algorithms. Clique Partitioning can be used to perform binding on functional units as well as storage elements, whereas left edge can only be used for storage elements binding. As with the case of scheduling, the user can also extend the base core of binding algorithms by creating a new algorithm and extending the Eridanus Algorithm core services.

The following table shows how the left edge algorithm uses the start and end times of variables to determine the binding of variables to storage elements. This information is also useful to derive inputs compatibility in order to build the register compatibility graph. As mentioned earlier, two variables or inputs are compatible if they are non concurrent. In the case of variables and register binding, variables are compatible when they do not have overlapping lifetimes. This means that if one variable is written to a register, then the other variable can also occupy the same register because none will overwrite the value of the other while that value is still needed elsewhere in the circuit. This said, every two nodes, representing variables, can now be connected by an edge if they are compatible. As this is done for all the nodes, the register compatibility graph is constructed. At this point, clique partitioning algorithm can be used to derive the clique cover number and hence the minimal number of needed registers.

The left-edge algorithm [11] can also be used to derive the minimal number of registers. The variables are sorted with respect to their start times and the algorithm proceeds from time 0. Any variable with the smallest start time is selected and a second variable with a start time greater than the first variable's end time is selected to be bound to the same register. If ties exist in the start times of variables, an arbitrary variable is selected. The concept of left-edge is similar to the clique partitioning algorithm but the former does not use any compatibility graph. This reduces the execution time by the time taken to generate needed to create the register compatibility graph. Implementation details and illustrative examples of left-edge are available in [16].

A similar procedure is used to construct the functional units' compatibility graphs. In this case also, each pair of operations is checked for compatibility and non concurrency. An edge is added between nodes that are scheduled at different control steps and do not have overlapping execution times. Once the compatibility graph is constructed, clique partitioning is used to derive the operations that will be bound to each resource type instance.

The results shown in Table 10, Table 11, and Table 12 are generated by Eridanus for the example suggested in Table 5.

Table 10: Calculated nodes' start and end times.

Variable	Start time	End time
X	0	1
y	0	1
Constant 1	0	1

W = y+1	1	3
Constant 0	0	1
x>0	1	4
Constant 2	1	2
W = w+2	2	4
Constant 3	2	3
W = w+3	3	4
Converge w	4	5

Table 11: Register binding result.

Register	Bound variables

Register 1	w = y + 1
	w = w +3

Register 2	X > 0 y
Register 3	x Converge W W =w +2

Analyzing the start and end times of some variables will help verify the above result obtained in Table 11. Looking at register 1, it can be seen that w = y+1 and w=w+3 are bound to this register. We will refer to those two operations and their results as the first and the second variables respectively. The first variable has a start time equals to 1 and an end time equals to 3. The second variable on the other hand has a start time equals to 3 and an end time equals to 4. This implies that the two variables are non concurrent and can occupy the same resource. The same reasoning applies to the other two registers.

Table 12: Functional unit binding solution.

Functional units	Bound operations
Comparator 1 (GREATER THAN)	X > 0

Adder 1	W = y + 1
	W = w + 2
	W = w + 3

Converge 1 (multiplexer)	Converge w

Again to verify the functional unit binding solution, we will analyze the adder binding. As mentioned earlier, the first operation is scheduled at control step 1. The second operation is scheduled at step 2 while the third operation at 3. This implies that all three operations can be bound to the same adder as was obtained in Table 12.

A note on this example is that the operations seem streamlined one after the other although the second and third operations can be concurrent because of their mutually exclusive branching characteristic. Nevertheless what caused this streamlined behavior is the bounding of the problem. This example is scheduled using only one resource instance for each resource type. This causes all operations of the same type to occupy the same resource instance and thus streamline the whole binding process. Thus as mentioned earlier, binding does not affect the area of the circuit through the number of resources because that is constrained earlier in the scheduling phase. Binding only maps operations to resources at this point, but this mapping can later be dramatic regarding area due to its influence on routing logic and the controller unit synthesis.

Data Path and Controller Generation

After binding is performed, all the basic information needed to connect resources together in a datapath is now available. At this point, basic connectivity synthesis is performed. Steering logic is used to connect functional units to register and register to functional units whenever necessary.

Let’s demonstrate how this is done in Eridanus. For this, we will use the adder functional unit and derive the needed multiplexers. Finally, the Eridanus-generated datapath will be shown in graphical and textual forms.

The adder implements three operations W = y + 1, W = w + 2, W = w + 3. The first operands for the three operations are y, w and w respectively. The second operands are the constants 1, 2 and 3 respectively. Looking at register bindings to locate where each variable is bound, we can identify that at least 2 operands y and w are bound to different registers. This calls for the user of steering logic which in the case of Eridanus will be multiplexers rather than other steering logic types such as busses. The same applies to the constants. Thus 2 multiplexers are needed to route the first and the second set of operands to the inputs of the adder. Another set of multiplexers is needed to route the output of the adder to the registers. This is needed because other functional units may be writing values to the same registers at different control steps. The suggested datapath structure is shown in Figure 14.

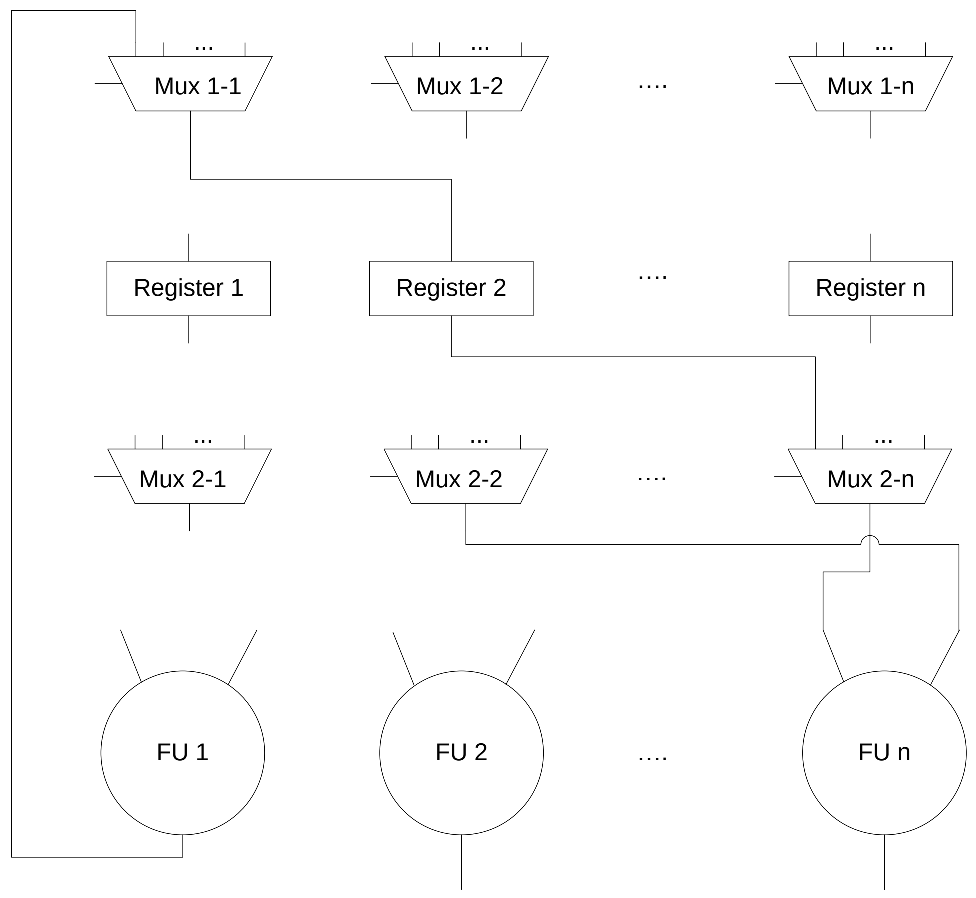


Figure 14: Suggested datapath structure.

The synthesized datapath for our example is shown in Figure 15. The corresponding VHDL code is included in Appendix A.

At the same time, the datapath is generated; the controller unit is also being produced. The controller is a finite state machine that has 2 inputs, the clock and the reset signal. The outputs of the controller are the selectors, and enables of datapath multiplexers and registers respectively. The controller is responsible of routing the variables from registers to functional unit input ports and routing the outputs of functional units to the register that they are bound to. The controller was synthesized to make use of two processes. The first process is sensitive on the falling edge of the clock. This process is used to jump from one state to the other. The second process is sensitive on the states and this process is used to generate the outputs of the controller depending on the states and hence the control steps. A complete walkthrough controller synthesis and simulation example is provided in [16].

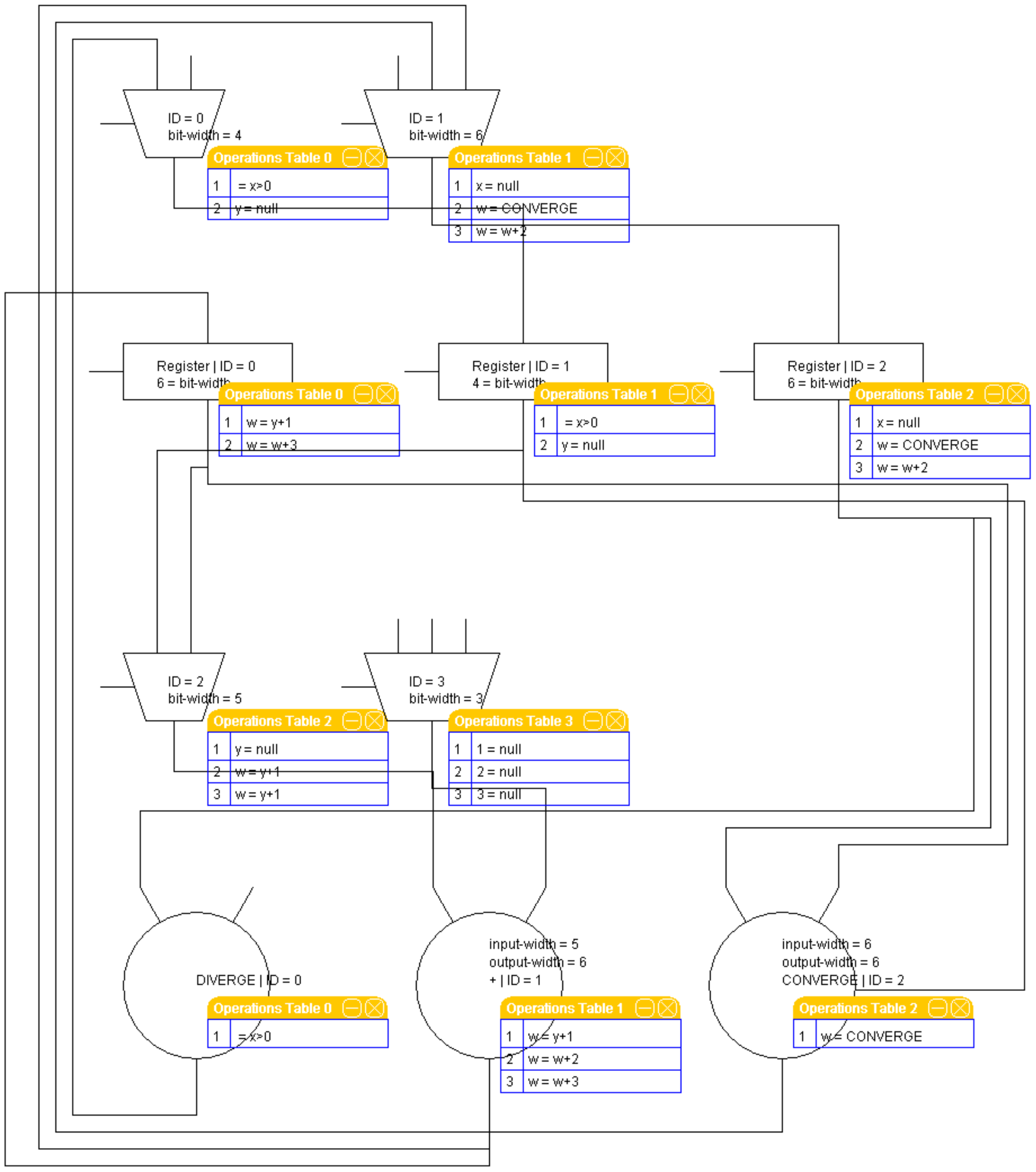


Figure 15: Eridanus synthesized datapath

Chapter 4: Reducing Spurious Switching Activity

The past years have seen an enormous jump in the number of transistors that can fit on a die. As this number increases, many design factors that were deemed as secondary became primary and crucial factors that needed urgent attention. One of those factors is power consumption.

Traditional binding and scheduling techniques do not consider the power factor during the synthesis flow and therefore the resulting schedules and binding solutions may not be efficient in terms of power consumption.

Power consumption (dissipation) is challenging to calculate because it can be divided into static and dynamic power dissipation. In brief, dynamic power dissipation occurs when switching charging and discharging output loads. Dynamic dissipation also occurs when a short-circuit exists in the circuit. This typically happens in CMOS VLSI when an input signal changes and a direct path exists between the pull-up and the pull-down circuits. On the other hand, static power dissipation occurs because of leakage sources in transistors. This in turn includes subthreshold conduction and reverse bias pn-junction leakage [14] [15].

When performing high-level synthesis, static and dynamic power cannot be tackled directly. These factors are better understood and manipulated when examined at the physical level rather than at higher levels such as RTL and logic levels. One of the things that can be indirectly approached in high-level synthesis is the dynamic power consumption. Since dynamic power is closely-tied to the change in the circuit inputs, an arrangement of inputs at the functional units can reduce the switching at those inputs. This in turn minimizes dynamic power dissipation and total power consumption.

While many techniques have added the number of registers to decrease the spurious switching activity at the functional units, our goal is to keep the number of the registers and hence the area constant without any increase. The room for improvement is obvious when a benchmark circuit is synthesized without any power considerations. As this synthesized circuit is observed it can be deduced that some variable bound to certain registers could have been swapped without increasing the number of registers. As variables are swapped, the routing of those variables through multiplexers is changed and thus the controller states are also modified. This research only studies the reduction in spurious switching activity due to variable swapping and does not include analyzing the design area after those swaps. Not a single component is added to the synthesized circuit, but an area increase could be identified when variable swaps move wider variables to registers of smaller bit-width. This in turn increases the size of register being moved to and hence increases the total area of the design. This is illustrated in Figure 16. The initial area in terms of bitwidth was 13 + 6 = 19 bits. After swapping Z and X, the second register now holds a variable that is 11 bits wide. Thus the area for that register in bits would be 11. This increases the design area from 19 to 13 + 11 = 24 bits.

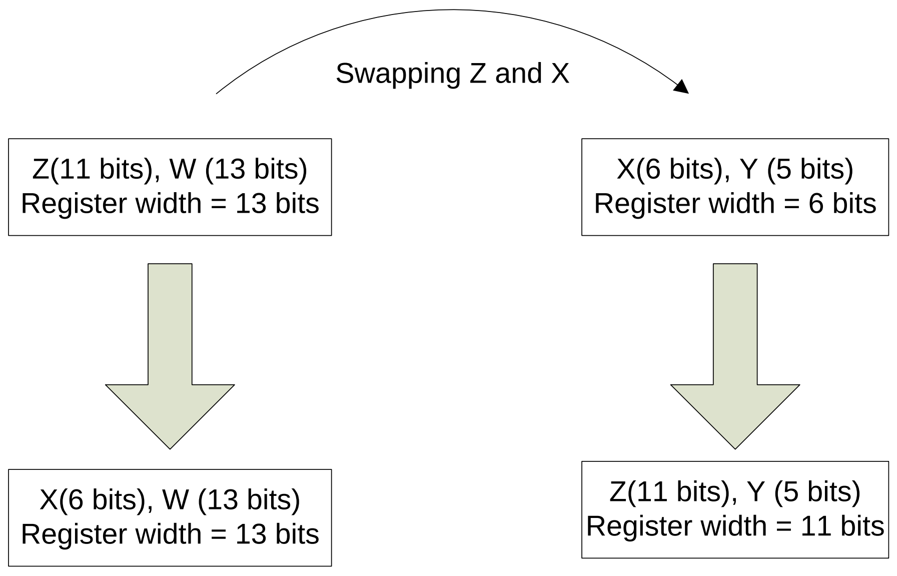


Figure 16: Swapping variables may increase design area.

Motivation

Let us now analyze the differential equation (DiffEq) benchmark circuit to try to find room for improving power consumption by reducing the spurious switching activity. After normal synthesis of this benchmark, the functional unit and register bindings are as follows:

Register bindings:

Table 13: Diffeq register bindings

Register	Bound variables
Register 1:	yinport
Register 2:	dxport
Register 3:	x _var = xinport+dxport

	t1 = uinport*dxport
Register 4:	e3 t3 = e3*yinport
Register 5:	t6 = uinport-t4 xinport uinport

Register 6:	y1 = u_var*dxport
	
	y_var = yinport+y1
	
	u_var = t6-t5
	
	t5 = dxport*t3
	
	t4 = t1*t2
	
	t2 = e3*xinport

Functional unit bindings:

Table 14: Diffeq functional unit bindings

Functional unit	Bound operations

Adder 1:	x_var = xinport+dxport
	
	y_var = yinport+y1

Subtractor 1:	t6 = uinport-t4
	
	u_var = t6-t5

Multiplier 1:	t2 = e3*xinport
	
	t1 = uinport*dxport

	t3 = e3*yinport
	
	t4 = t1*t2
	
	t5 = dxport*t3
	
	y1 = u_var*dxport

With the above configuration, and using only one resource for each functional unit type (1 adder, 1 subtractor, and 1 multiplier), the total switching activity of the design is 20. This is illustrated in detail in the following table.

Table 15: Diffeq switching couples

Functional Unit	Switching couples
Subtractor 1:	xinport :junk
	uinport:t2
	uinport:t4
	t6:t5
	t6:u_var
	t6:y1
	t6:y_var
Adder 1:	xinport:dxport
	uinport:dxport
	t6:dxport
	yinport:y1

	yinport:y_var
Multiplier 1:	e3:xinport
	uinport:dxport
	e3:yinport
	t1:t2
	dxport:t3
	u_var:dxport
	y1:dxport
	y_var:dxport

Observations

From the above Diffeq benchmark circuit, it can be observed that unnecessary switching exists in the following cases:

1. At the first operation executed in a functional unit
2. At intermediate operations executed in a functional unit
3. At the final operation executed in a functional unit

The first case occurs when the operation is the first operation bound to the functional unit. The operands of this operation are bound to different registers because they are not compatible in the compatibility graph. The noncompatibility stems from the fact that both operands have intersecting lifetimes especially because they may have similar start times. Because of this, the second level multiplexers should have their selectors set in a way to allow the routing of those operands to the proper inputs of the functional unit.

As an example, comparing the second and the third switching couples of the subtractor in Table 15, it can be seen that t6 = uinport-t4 does not get calculated until a switching between uinport:t2 occurs. After that the effective switching that calculates t6 (uinport:t4) occurs.

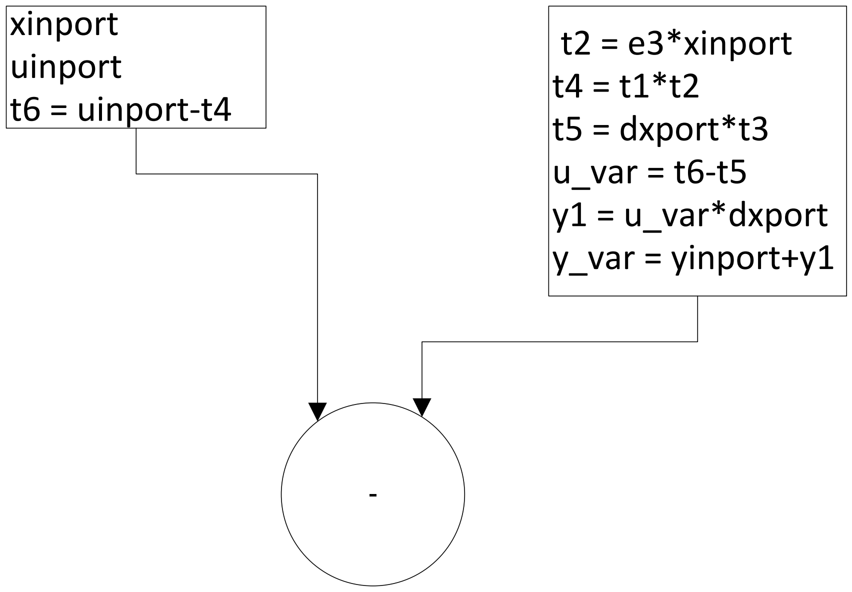


Figure 17: First operation switching

The first operation, as mentioned earlier, is t6:=uinport-t4. But the start time of variable xinport is 0 while that of uinport is 1. Because the controller has strobed the selectors of the functional unit's input multiplexers, the values from those 2 input registers will be read from time 0. Thus the first value available at the input of the functional unit is xinport. On the second register, the first variable bound to the register is t2. This variable has a start time of 1 and thus the register lacks a variable that has a start time equal to 0. This implies that the value in the second register could be some junk value of an initial value set when resetting the circuit. From this discussion we can build a switching table similar to that in Table 15.

Manual switching couple	Reason	Step
Xinport:junk	The first input register contains Xinport which has a start time = 0 The second register contains junk value at step 0	0
Uinport:t2	The first input register now contains uinport which has a start time = 1 and end time = 5 The second register now contains variable t2 which also has a start time = 1 and end time = 4	1
Uinport:t4	The first input register still contains uinport which has a start time = 1 and an endtime = 5 The second register now contains variable t4 which has a start time = 4	4

A timeline description of the above table clearly shows the overlapping lifetimes and clarifies the switching activity on the subtractor.

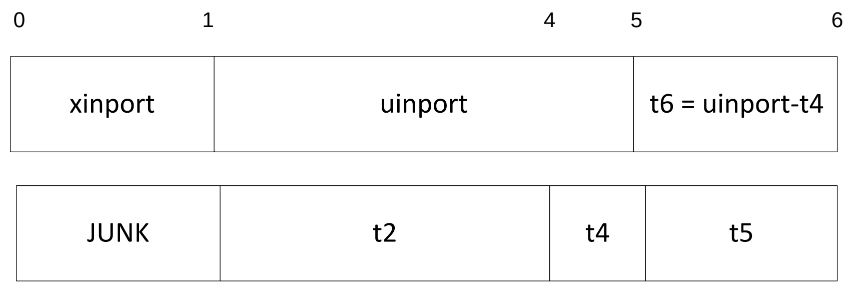


Figure 18: Register 1 and 2 timelines

The second case of spurious switching occurs in between operations. This happens frequently because not all operands (variable) of an operation have same start times. Sometimes even if they may have similar start times, the registers connected to the inputs of the functional unit do not get changed through multiplexer routing. This causes all the variables bound to a certain register to be available at the inputs of that functional unit. An illustration diagram is provided in Figure 19 below.

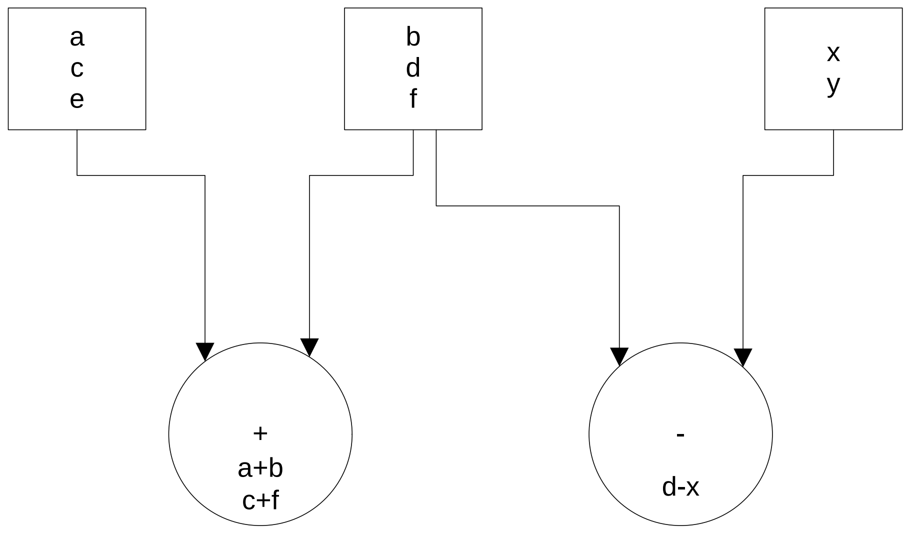


Figure 19: Spurious switching in-between operations

The example in Figure 19 clarifies inter-operation spurious switching. This figure is a partial example used for illustrative purposes and variables e and y are used in other functional units that are not shown here for simplicity. The adder in this example performs 2 operations. The first operation adds a and b. In this case switching will occur between those 2 couples (a:b). For the second operation bound to this adder, the operands are c and f. At this point the multiplexer selectors at the input of the functional unit will not change and because of this, the following variables will be available at the second input of the adder in the following order: b, d, f. Variable d is not a valid input for this adder. Thus depending on the lifetimes of a and c, variable d will switch with either of those variables. Two scenarios are provided in Table 16 to illustrate this.

Table 16: Inter-switching scenarios

Setting 1	Switching couples	Reason
a: start time = 0, end time = 2 c: start time = 2, end time = 3 b: start time = 0, end time = 1 d: start time = 1, end time = 2 f: start time = 2, end time = 3	a:b	Effective switching to calculate a+b
	a:d	Spurious switching due to the overlapping lifetimes of a and d.
	c:f	Effective switching to calculate c+f
Setting 2		
a: start time = 0, end time = 1 c: start time = 1, end time = 3 b: start time = 0, end time = 1 d: start time = 1, end time = 2 f: start time = 2, end time = 3	a:b	Effective switching to calculate a+b
	c:d	Spurious switching due to the overlapping lifetimes of c and d.
	c:f	Effective switching to calculate c+f

Multiplexers in this example have been omitted because the partial example shown here does not need multiplexers to route variables from different registers to an input of the functional input such as the adder. A similar switching scenario can occur when looking at the example with the second register as a reference. All in all, variable changes in-between operations also is a major source of spurious switching and should therefore be addressed when trying to reduce power consumption.

Analyzing the same Diffeq benchmark, it can be found that inter-operation switching is also a major part of the switching couples. This example also shows how multiplexer routing affects switching on the inputs of the functional units. The functional unit used in this example is the adder functional unit and its partial datapath and inputs are shown in Figure 20.

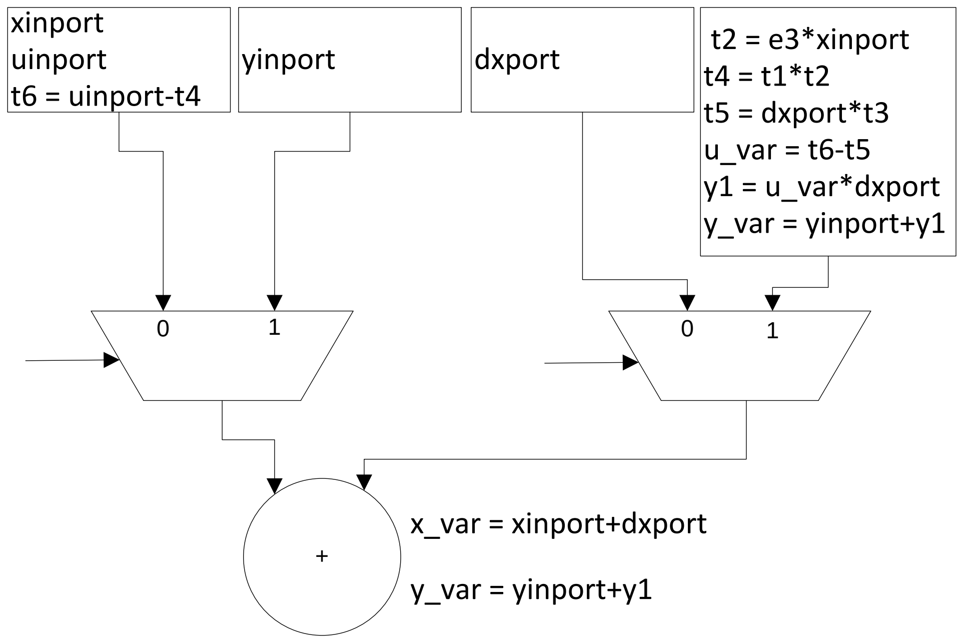


Figure 20: Adder 1 input's partial datapath

The inter-operation spurious switching is better explained in this example. The multiplexers' selectors are initially set to 0. This configuration routes the values in the first and the third registers to inputs of the adder. Hence, xinport and dxport are available at the first and second input of the adder respectively. The switching due to those two inputs allows the calculation of x_var. Any switching occurring after this point will be referred to as inter-operations spurious switching.

Because the next operation y_var = yinport + y1, is scheduled at control step 7, the multiplexers' selectors will not change until that control step. This means that all variables that will be written to the first and third registers will be consumed by the functional units connected to those registers. In this case, the adder will calculate unnecessary operations and therefore produce spurious switching. Thus after the first switching couple (xinport:dxport), the variable uinport will be written to the first register at control step 1. Knowing that dxport has a lifetime from control step 0 till control step 7 when it is no more used, this variable will switch with variables in the first register. In fact dxport will live in the third register till the end of the example lifetime because no value is written to that register as shown in Table 13. As a consequence, uinport, which has a start time of 1 and end time of 4, will switch once with dxport (uinport:dxport). The same scenario occurs between t6 and dxport(t6:dxport). Since t6 is the final variable written to the first register, it will also be available beyond its end time as is the case with dxport. It can be seen that although one operation, and therefore 1 switching activity, was needed, three switching activities were actually performed while waiting for the next operation to be executed. At control step 7, the multiplexers' selectors will change to 1 to consume the values of the next operation operands. At this point, yinport will be available at the adder's first input, and y1 at the second input causing the yinport:y1 switching to occur. This calculates y_var which is the final operation bound to this functional unit.

The above example's lifetime and switching couples are shown Figure 21 and Table 15 respectively.

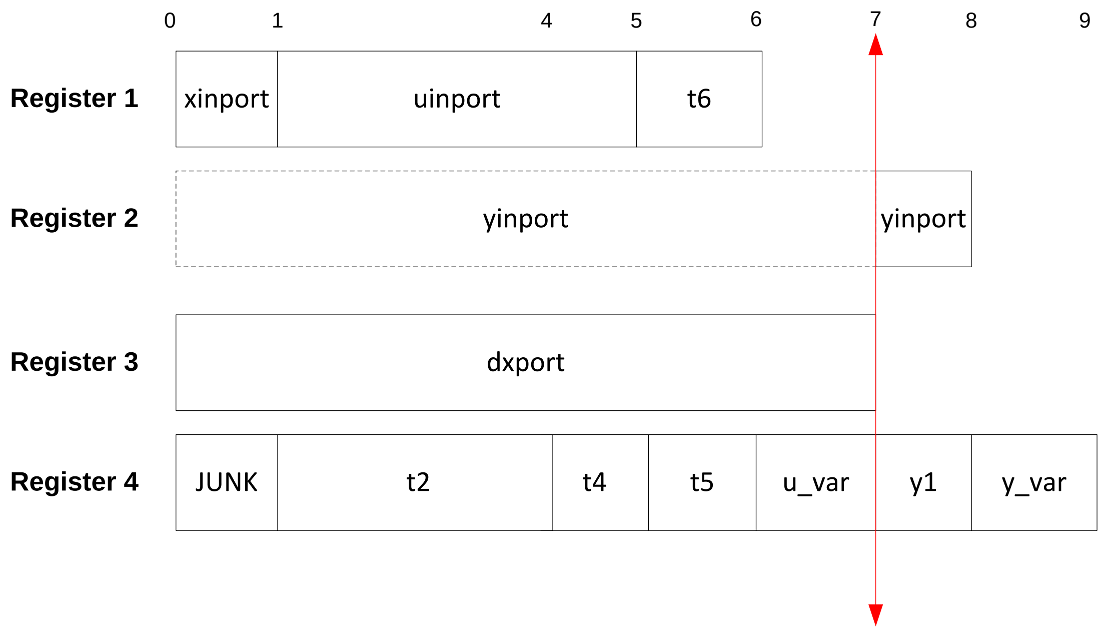
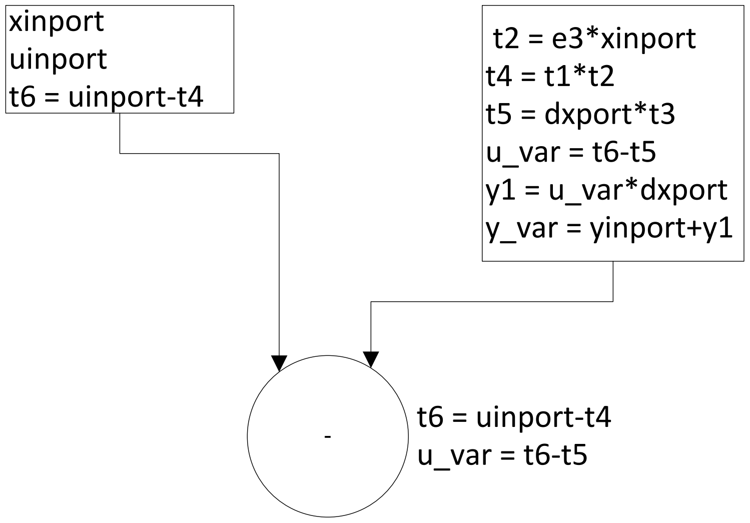


Figure 21: Adder's inputs overlapping lifetimes and switching couples

The third case of spurious switching is quite similar to the first type of spurious switching. The last type of spurious switching can be observed in the previous example shown in Figure 20 and Figure 21. The last operation in the adder is y_var = yinport + y1. After this operation, the y_var result will be written to the same register as y1. Since the adder is not performing any other operations or using different operands, the multiplexers' selectors have not changed and therefore the same registers will still be connected to the inputs of the adder. New values written to those registers will be available at the inputs of the adder and will cause spurious switching as in the case with yinport:y_var. The intensity of this type of spurious switching is not shown clearly in this example and will be illustrated using the subtractor functional unit of the DiffEq benchmark circuit.

The partial datapath of the subtractor functional unit is replicated below for explanatory purposes. It can be seen that the last operation performed in the subtractor is the u_var = t6 –t5 operation. Because t6 is the last variable written to the first register, this value will remain alive throughout the lifetime of the circuit. Since u_var = t6 – t5 is the last operation, the selectors of the multiplexers (omitted from the figure for simplicity) at the inputs of the subtractor will not change. As new values are written to the second register, these variables will be directly consumed by the subtractor and will therefore cause the subtractor to switch spuriously.

In this example, the operation u_var = t6 – t5 is scheduled at control step 5. The total control steps needed to execute the DiffEq circuit is 8 control steps as calculated using the LIST scheduling technique with one adder, one subtractor and one multiplier. The latter means that after u_var is calculated, the circuit still has a 3 control step lifetime and during this time many variables can be written to the registers. A simple case is observed in the subtractor circuit. As u_var is calculated, it is saved to the registers that contained one of its operands, t5. As the value in this register is updated, it will be consumed by the subtractor which will calculate t6 – u_var. Similarly y1 is written to the register at control step 7 causing the subtractor to also calculate t6 – y1. Again as y_var is written to the second register, the subtractor will calculate t6 – y_var. This case alone causes the subtractor to switch 3 times more than needed, not to mention the switching caused by the first and second type of spurious switching. All this explanation concludes that final operation switching can have dramatic influence on the number of total and spurious switching in a functional unit and in this example it caused 150 % more switching than necessary.



The variables' overlapping lifetimes of the first and second registers is shown in Figure 22. The figure clearly shows how t6 will switch with t5, u_var, y1 and y_var consecutively.

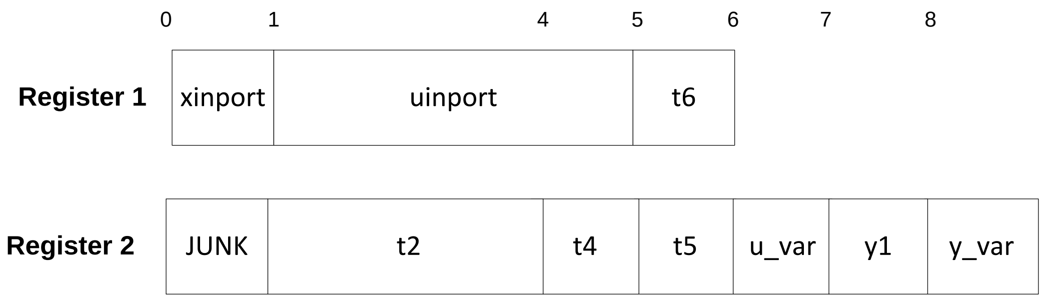


Figure 22: Subtractor last operation operand lifetimes and switching couples

Power Research

As power consumption has emerged as a primary critical issue in high-level synthesis, a lot of research has been put into exploring the means to optimize this design metric. Many solutions have been proposed to reduce Spurious Switching Activity (SSA) and below is a listing of some:

Spurious switching activity can be reduced by changing and constraining variable assignments [3]. This is done through using more registers. As shown earlier in our discussion of the three types of switching activities, registers stacked with variables can cause excessive spurious switching. As more registers are added, the variable assignments are relaxed and consequently less variables are written to each register. This is demonstrated using the same subtractor example of Figure 17. Looking at Figure 22, it can be seen that u_var, y1 and y_var written to register 2 at control steps 6, 7, and 8 repsectively are not used by the subtractor functional unit and can therefore be moved to another register. If an extra register is available, then those 3 variables can be bound to that register and therefore the switching between t6 and u_var, t6 and y1, and t6 and y_var will be eliminated. Of course, the movement of the 3 variables could induce switching with other variables but if done properly and globally could reduce switching dramatically. However, this technique obviously increases area due to additional registers and may induce area increase in steering logic such as multiplexers. It also implies that more power is consumed due to the additional registers.

Another technique to reduce spurious switching at the inputs of functional units was proposed by [4]. This technique suggests adding transparent latches at the input ports of the functional units. Targeting idle components, it can reduce unnecessary switching at those functional units. This is possible because the scheduling and binding phases provide valuable information of when a functional unit is used as well as when it is idle. In the control steps when the functional unit is supposed to be idle, those transparent latches would inhibit any changes at the input ports. This in turn reduces spurious switching. However, this technique also increases area due to the additional latches. It also implies that the total power consumption will increase due to the newly added latches. Another drawback of adding latches is the added complexity in the controller. New signals have to be setup to enable and disable those latches whenever necessary.

Extending the life of protected variables is another approach suggested by [5]. In this work, the authors set variables and path as protected. Protected paths and variables are critical to power management in that they can cause spurious switching when left unprotected. The protection concept extends the life of variables that are used as operands to the functional units in previous control steps so as to provide stable non-changing inputs to that functional unit in idle control steps. This in turn reduces the second type of switching activity mentioned earlier. Extending the lifetimes of variables is in essence similar to the work in [4] suggesting the addition of transparent latches to preserve input values at the inputs of the functional units.

Register binding with retentive multiplexers [6] is another such approach to preserve values at functional units' input ports. Retentive multiplexers preserve the value of their previous select signals in the control steps in which those select signals are don't cares. Because synthesis tools sometimes choose to set values for don't cares in order to better utilize a specific type of encoding, this may cause unnecessary changing in input values resulting in spurious switching. When retentive multiplexers are added, previous signal values will be preserved when the new signal values are don’t cares. This inhibits spurious switching by keeping inputs stable.

A novel approach to eliminate spurious switching and leakage power was introduced in [7]. This could be achieved through power islands. A power island is a cluster of logic that is independently powered from the rest of the circuit. This work proposes scheduling, binding and placement techniques to classify logic and separate them into different clusters. When all the logic in an island is idle, the whole island could be powered down thus stopping all switching activity and eliminating power consumption due to leakage power.

Room for improvement

To set the motive for this work, we will alter the DiffEq benchmark used so far in order to show how power management can be incorporated. It can be seen from the binding solution provided in Table 13, that variables t1 and x_var are bound to register 3. Variable x_var has a start time of 1 and end time of 2 while t1 has a start time of 2 and an end time of 4. The total latency of the schedule as mentioned earlier is 9. Thus register 3 can hold new values that might be causing spurious switching elsewhere. Consider variables y1 and y_var bound to register 6. Those variables have a combined lifetime with a start time of 7 and end time of 9. Since register 3 has no values at this interval, then y1 and y_var can be moved there and thus reducing the number of changing values in register 6. Doing this would reduce the switching activity at the subtractor to 5 instead of 7. The new power aware register bindings are shown in Table 17.

Table 17: New power-aware register bindings

Register	Bound variables
Register 1:	yinport
Register 2:	dxport

Register 3:	x _var = xinport+dxport
	
	t1 = uinport*dxport
	
	y1 = u_var*dxport
	
	y_var = yinport+y1

Register 4:	e3 t3 = e3*yinport
Register 5:	t6 = uinport-t4 xinport uinport

Register 6:	u_var = t6-t5
	
	t5 = dxport*t3
	
	t4 = t1*t2
	
	t2 = e3*xinport

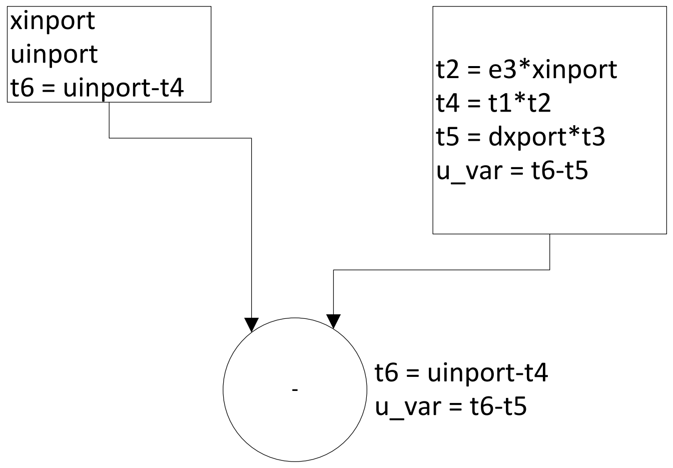


Figure 23: Subtractor datapath after power optimization

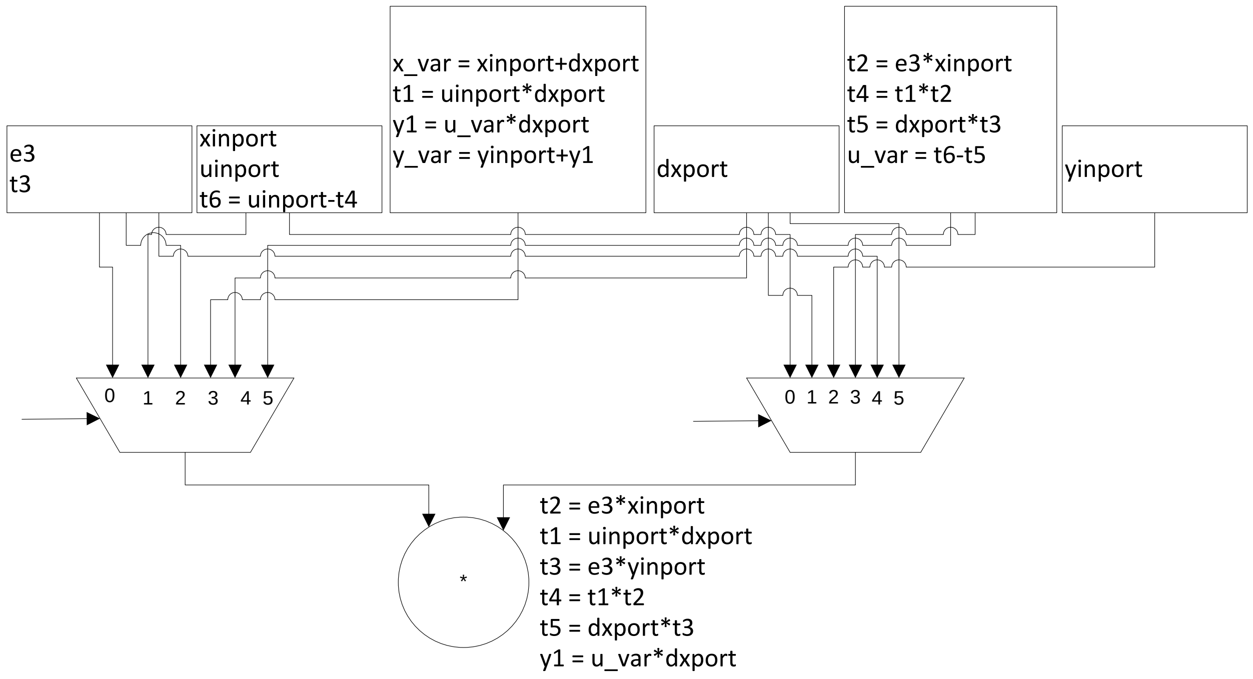


Figure 24: Multiplier datapath after power optimization

The resulting switching couples due to this power-aware variable swapping are shown in Table 18. This table clearly shows reduced switching from the one in Table 15. The reduction is from 20 to 16 couples which accounts to 20% in switching activity. Spurious switching on the other hand was reduced from 10 to 6 which accounts to 40%.

Table 18: New reduced switching couples

Functional Unit	Switching couples
Subtractor 1:	xinport :junk
	uinport:t2
	uinport:t4
	t6:t5
	t6:u_var
Adder 1:	xinport:dxport
	uinport:dxport
	t6:dxport
	yinport:y1
	yinport:y_var
Multiplier 1:	e3:xinport
	uinport:dxport
	e3:yinport
	t1:t2
	dxport:t3
	u_var:dxport

A New Approach

Our work focuses on similar objectives and that is to reduce power consumption through reducing spurious switching activity without any overhead in the number of resources. This is achieved first through obtaining the regular schedule and then performing the regular binding. The regular binding is then analyzed for improvements. The binding optimization procedure makes use of a cool-down simulated annealing to explore the solution space. Finally a binding for the minimal number of registers with the minimal switching activity is obtained.

The technique proposed in this work targets the three types of switching activities. To reduce switching activity without increasing the number of registers, the number of registers had to be fixed and the binding of variables to registers has to be explored. This exploration is based on the idea of swapping variables in registers in order to calculate the best (minimal) switching activity.

Before introducing the cool-down simulated annealing optimization process, we will introduce other techniques that were explored for this same purpose. The first and earliest step to reducing switching activity could be at the initial binding phase. If a feasible technique could be found to reduce switching at this phase, then execution time and complexity of the synthesis tool can be reduced. Nevertheless, the clique partitioning problem is NP-Complete and thus no solution to solve the problem in polynomial time exists. Heuristic algorithms need to be used and the available heuristic algorithms do not have the flexibility to incorporate several design metrics to influence the partitioning problem. In the case of Eridanus and this work in general, the clique partitioning heuristic implemented and used is the Tseng-Siewiorek algorithm [10]. This algorithm merges neighbor nodes which share the largest number of neighbors. It is therefore power unaware and only tries to produce a solution that minimizes the number of resources. Whenever ties in the number of neighbors are encountered, Tseng's algorithm breaks ties arbitrarily. Trying to break ties differently was one of the targets of this work. Instead of exploring all other solutions by merging arbitrary nodes together, this work tried to explore tied solutions and proceed from that point. This would definitely explore the power consumption of all bindings corresponding to all tied nodes. The complexity of this technique stems from the fact that the switching cost cannot be calculated until the graph has been partitioned completely. This means that if two nodes are tied at the first iteration, then both have to be explored and two graphs have to be generated for each merge. Moving on to the next iteration, other graphs have to be calculated for each tie. Obviously as the problem size and the number of ties increase, this problem increases exponentially in complexity and eventually drains the memory heap.

Because of all these factors, post binding optimization techniques were explored. Simulated annealing is one technique that can be used for the global optimization problem as it locates a good approximation to the global minimum of a certain function.

Moving from one solution to the other is a matter of changing variable binding in registers. This is done by interchanging or swapping two or more variables. Our approach swaps variables according to two ways with equal probabilities to keep the annealing process as unbiased and random as possible. These two methods are:

Swapping two variables

This method explores swapping variables with same lifetimes including start and end times. This is a purely random process that selects any node and tries to find nodes with same start and end times. After doing so, a random variable is selected from the list of matching candidates. The two variables are swapped and the switching cost is calculated after that.

The advantage of using this method is that it allows exploring the various slots and registers where a certain variable can be bound. Since it works on a node (variable) by node basis, it is a fast method and contributes to the switching optimization.

Swapping a set of variables

In most of the cases, swapping two variables will not contribute much to reducing spurious switching activity of a circuit. In some cases, not a lot of variables with same lifetimes exist in the same circuit. Thus other alternatives have to be used. The second method focuses on swapping a set of variables with one variable. This could have covered the case implemented in the first method, but it was a design consideration to separate them for better performance that will be explained later.

This method first identifies a variable that has a lifetime greater than one control step. In doing so, it eliminates searching for variables that could be covered using the first method. The next step is to identify a set of variables that have lifetimes subset of the first variable's lifetime. An example illustrating the acceptable cases is shown in Figure 25. Consider that variable b was selected to be swapped. The process then searches for other nodes in other registers for possible swaps. Since variable b is going to be replaced with the new set of variables, then each variable in the new set must have a start time greater than or equal to that of b, and an end time less than or equal to b. Variables d and i do not satisfy this condition and therefore cannot be included in the set of swapped variables. Only variables e, f, g, and h will be swapped with variable b.

Combining the two methods, we can even achieve solutions where if compared to the initial solution, would seem as if the annealing process is moving variables without even swapping. This is due to the swapping and shuffling behavior. As one set of variables is swapped with one variable, a subset of those variables may also be swapped with another variable and may eventually end up in the same initial place after obtaining an approximation of the global minimum. This was previously illustrated in Figure 24 when y1 and y_var were moved to the third register without any swapping traces. In fact a set of variable swapping in the background led to this movement behavior.

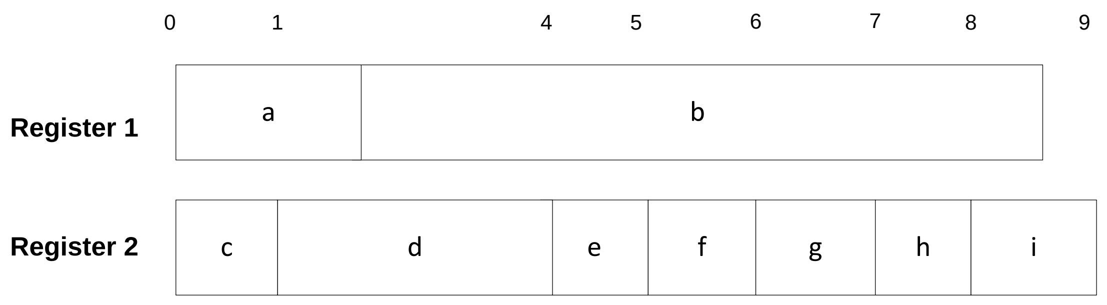
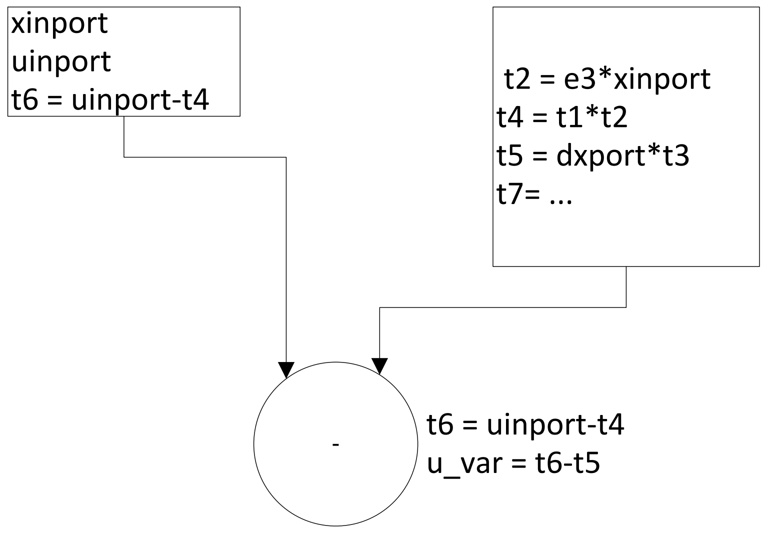


Figure 25: Set-swapping example

Let us demonstrate how this technique can help reduce the spurious switching activity by analyzing the subtractor functional unit shown in Figure 17. The last operation is u_var = t6-t5 and thus the last operands used by this subtractor are the t6 and t5 operands. The other variables, u_var, y1, y_var, written to the second register after t5 are not needed and will therefore cause spurious switching activity as mentioned earlier.

However, these variables are definitely needed somewhere else in the datapath and should therefore be written to some register. Consider a variable t7 residing in some other register. After the switching optimization annealing process is started, that variable t7 was randomly chosen to be swapped. Suppose that the start and end times of t7 are 6 and 9 respectively. The annealing process identifies the variables u_var, y1 and y_var as candidates for swapping considering that their combined lifetimes equal to that of t7. The process then completes the swap. It is obvious here that the 3 variables were swapped with one and now the second register at the second input port of the subtractor now has one variable that spans control steps 6 to 9. In other words, that register will have only one stable value throughout that time and only one value will be written to that register after t5 is written to it. Recalculating the switching inputs of the subtractor, we notice that the last three spurious switchings are now eliminated and replaced by one spurious switching activity (t6 : t7).



Simulated Annealing

Keeping in mind that a fast, easy to implement, and a non-exhaustive method needs to be used, simulated annealing was selected for the purpose. As mentioned earlier, simulated annealing is used to explore the design space and solve the global minimization problem by locating a good approximation to the global minimum. The pseudo code for the simulated annealing heuristic is provided in Table 19.

Table 19: Simulated annealing pseudo code

```
state = initialState
cost = getCost(state)
bestState = state
bestCost = cost
while (time < maximumTime)
{
        newState = getNeighbor(state);
        newCost = getCost(state);
        if  (newCost < bestCost)
        {
                bestState = newState;
                bestCost = newCost;
        }
        else if ( Probabilty of accepting the solution at current temperature > random)
        {
                state = newState;
                cost = newCost;
        }
        temperature = temperature * temperatureFactor;
        time = time + time * timeFactor;
}
```

In brief, at each iteration simulated annealing will identify a neighbor solution to the current solution. As the initial description of simulated annealing suggests, the process will probabilistically decide to move the system to the new solution or leave the current solution. The probabilities are chosen so that the system eventually tends to move to a lower cost. [8] suggests that simulated annealing will eventually find some of the many near-optimal solutions that exist for a certain function.

For each new neighbor, the cost for that solution is explored and if this new cost is better than the best cost found so far, the new cost becomes the best cost of the problem. The new state also becomes the best state. If the cost is not better than the best cost so far, the new state will be used as the current state depending with a certain probability. It is this probability that increases the chances of a stochastic process to find near-optimal solutions. This is because it helps simulated annealing escape from a local minima in the solution space. If this probability is not used, then as new solutions are explored using the local minima solution, all these new solutions will have worse costs that the local minima's cost. This would exhaust the time for annealing and eventually return the local minima as the best solution. To keep the acceptance as random as possible, the probability will be compared with a newly generated random number at each iteration. If the probability is greater than that random number, the new state will be used as the current solution from which other solutions/states will be generated. The probability highly depends on the new cost and the current temperature. As the temperature drops, this indicates that the annealing process is nearing the end. At this stage the amount of perturbation to the current state is not acceptable as it alters the solution dramatically.

Whereas at higher temperatures, moves that increase the cost rather than decreasing it can be accepted as the simulated annealing process still has time to refine the best solution. The cost also plays an important role in the acceptance probability. As the cost difference increases, this means that the new state is much worse than the current state and therefore such as move is not preferable. As the cost tends to be small, such jumps or moves can be considered in search for a better state. Finally at each iteration, the time is increased by a factor and the temperature is also reduced by another factor. This assures that the annealing process eventually cools down after a certain time limit.

Fast Cost Update

The cost function of the simulated annealing and its recalculation are a major part of the execution. Therefore an accurate and efficient cost function is needed. The cost function implemented is the function that generates switching couples for the current state which corresponds to the current binding solution. As the annealing process generates new solutions, the cost function is calculated each time. In an attempt to make this recalculation faster, recalculation was eliminated and instead a cost update technique was used. Knowing that a new solution corresponds to a couple of variables swaps, we can observe that switching will decrease on one side and decrease on the other side. Thus instead of calculating the switching couples of the new solution, we just update the old couples by recalculating couples at the newly changed sites. This dramatically reduces execution time and makes the annealing faster and capable of running more iterations to further refine the search. Thus instead of running iterations using the number of variables as the upper bound for the swaps, we can double that value and give the annealing process more time and more trials to achieve a better solution. Consider the example shown in Table 15 and Table 18. The two tables correspond to the same example with one difference that variables y1 and y_var have been moved to a register other than their initial register. The new switching couples can be obtained with little effort using the update feature just mentioned. All functional units connected to the sixth register will have all switching couples containing y1 and y_var removed. All functional units connected to the third register will have their switching couples recalculated to obtain the newly added variables. This leaves all other switching couples intact.

Chapter 5: Experimental Results

To validate the effectiveness of the annealing process provided so far, 11 benchmarks of various complexities were synthesized and power optimized. The results have shown that on average the total power savings due to reducing spurious switching activity were 18% while spurious switching activity was reduced by an average of 40%.

In this section we will provide benchmark description and CDFG figures along with simulation results showing initial switching activity with no power management and annealed results with power management.

Note: Circuits will be classified according to their size in an ascending order with circuits having the least number of nodes first. Resource bags are named according to the number of available resources in the bag. For example, resource1 indicates that one resource of each required type is available, while resource 10 indicates that 10 are available. As mentioned earlier, due to the fast recalculation of the cost function, the simulated annealing was given more than enough time to execute. The execution time was set as 20 million iterations and the trial and temperature factors were set to 0.99 to slowly decrease both parameters.

Poly Design (7 nodes)

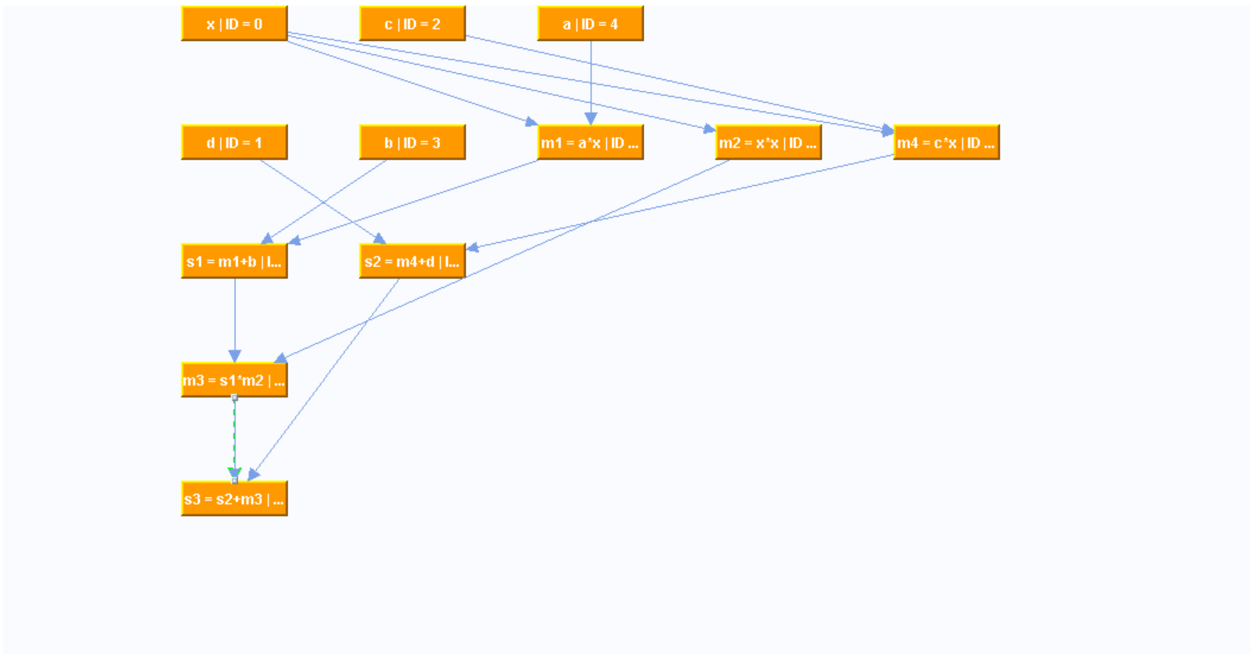


Figure 26: poly_design design cdfg

The results in Table 20 show that on average the improvement in spurious switching is approximately 14% in this case. The first column lists the resource bags that were used in scheduling while the second and third column show the switching costs for the circuit using normal binding and RSSA power managed binding respectively. The last column shows the percentage improvement in switching cost.

Knowing that the circuit contains 7 nodes corresponding to 7 operations, the non spurious switching needed to calculate those 7 operations are 7. Using this information the spurious switching activity improvement can be obtained by subtracting 7 from the obtained switching cost. The spurious switching costs are shown in Table 21. The average reduction in spurious switching is expected to be higher because as discussed earlier, spurious switching is a subset of the total switching of a circuit.

Table 20: poly_design total switching costs results

Resource bag	Normal	RSSA	Improvement
resource1	12	12	0.00
resource2	18	15	16.67
resource3	20	17	15.00
resource4	20	17	15.00
resource5	20	17	15.00
resource6	20	17	15.00
resource7	20	17	15.00
resource10	20	17	15.00
resource15	20	17	15.00
resource20	20	17	15.00
Average			13.67

Table 21: poly_design spurious switching costs

Normal	RSSA	Improvement
5	5	0.00
11	8	27.27
13	10	23.08
13	10	23.08
13	10	23.08
13	10	23.08
13	10	23.08
13	10	23.08
13	10	23.08
13	10	23.08
		21.19

To further analyze our results, it can be observed in the lower resource bags that the improvement is much lower than the results for the bigger resource bags. It can even be seen that sometimes there is no improvement. This is because whenever fewer resources are available, then all the operations have to be mapped to those scarse resources. Let's consider the first resource bag results. As only one resource is available in here, then all addition operations must be mapped to the one available adder. Similarly all subtraction operations have to be mapped to that single subtractor. This implies that the operations will be streamlined and thus the functional unit implementing these operations will rarely be idle. Thus operations will be stacked throughout the lifetime of the schedule and that functional unit will be possible executing a new operation every control step. All this means that the second type of switching activity, inter-operation switching, will be negligible and the other two types of switching will contribute mainly to the switching costs. Thus trying to power optimize the circuit may not sometimes improve the cost but it will never increase it. This example clearly shows the mentioned case when using the first resource bag. As more resources are added to the bag, the cost seems to stabilize. This is because the poly design example is a small example and all that is needed is three multipliers and 2 adders to obtain the minimal schedule latency. After the third resource bag, resource3, the switching costs stabilize.

The discussion presented here applies to all upcoming circuits and therefore will not be repeated in the next examples.

DiffEq circuit (10 nodes)

The diffeq circuit is the second order differential equation solver benchmark and contains 10 operations. The CDFG, the total switching, and the spurious switching results for this circuit are shown in Figure 27, Table 22, and Table 23 respectively.

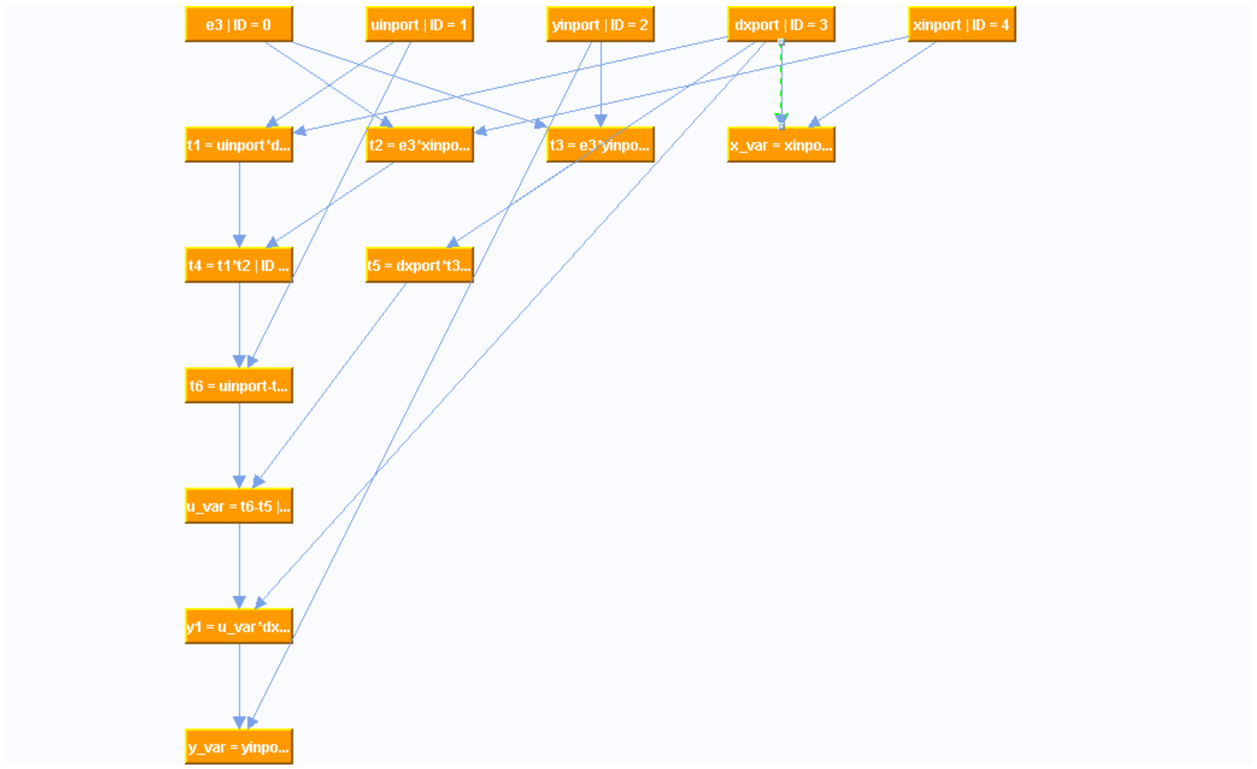


Figure 27: Diffeq circuit cdfg

Table 22: diffeq total switching costs results

Resource bag	Normal	RSSA	Improvement
resource1	20	16	20.00
resource2	26	17	34.62
resource3	26	23	11.54
resource4	26	23	11.54
resource5	26	23	11.54
resource6	26	23	11.54
resource7	26	23	11.54
resource10	26	23	11.54
resource15	26	23	11.54
resource20	26	23	11.54
Average			14.69

Table 23: diffeq spurious switching costs

Normal	RSSA	Improvement
10	6	40.00
16	7	56.25
16	13	18.75
16	13	18.75
16	13	18.75
16	13	18.75
16	13	18.75
16	13	18.75
16	13	18.75
16	13	18.75
		24.63

4pDCT circuit (15 nodes)

The 4pDCT circuit is the four point discrete cosine transform benchmark and contains 15 operations. The CDFG, the total switching, and the spurious switching results for this circuit are shown in Figure 28, Table 24, and Table 25 respectively.

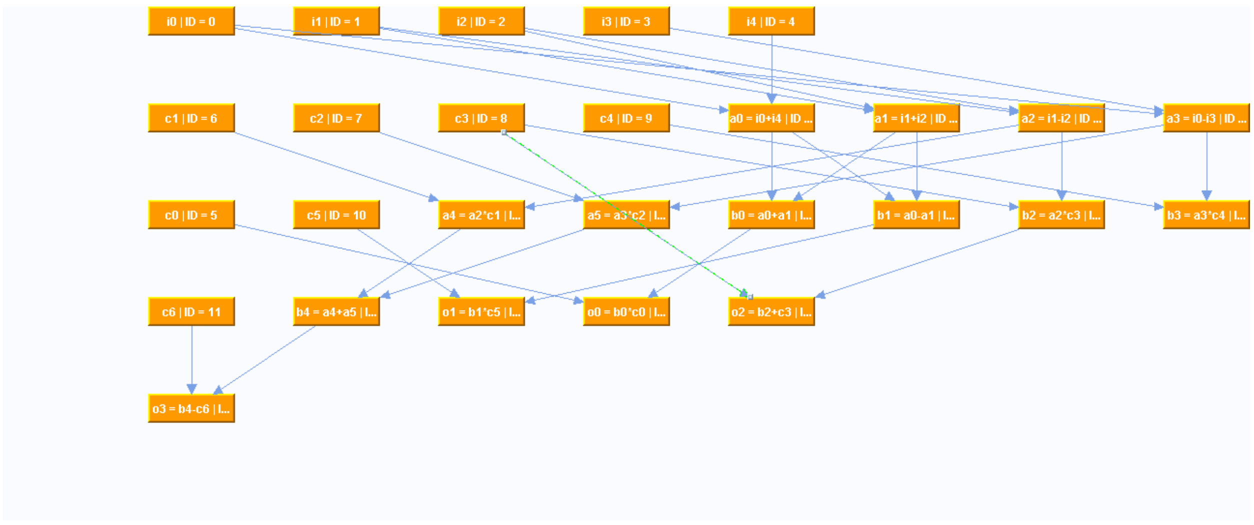


Figure 28: 4pDCT circuit cdfg

Table 24: 4pDCT total switching costs results

Resource bag	Normal	RSSA	Improvement
resource1	23	18	21.74
resource2	28	26	7.14
resource3	29	26	10.34
resource4	33	29	12.12
resource5	33	29	12.12
resource6	33	29	12.12
resource7	33	29	12.12
resource10	33	29	12.12
resource15	33	29	12.12
resource20	33	29	12.12
Average			12.41

Table 25: 4pDCT spurious switching costs

Normal	RSSA	Improvement
8	11	62.50
13	11	15.38
14	14	21.43
18	14	22.22
18	14	22.22
18	14	22.22
18	14	22.22
18	14	22.22
18	14	22.22
18	11	22.22
		25.49

DiffEq4 circuit (17 nodes)

The DiffEq4 circuit is a fourth order differential equation solver and contains 17 operations. The CDFG, the total switching, and the spurious switching results for this circuit are shown in Figure 29, Table 26, and Table 27 respectively.

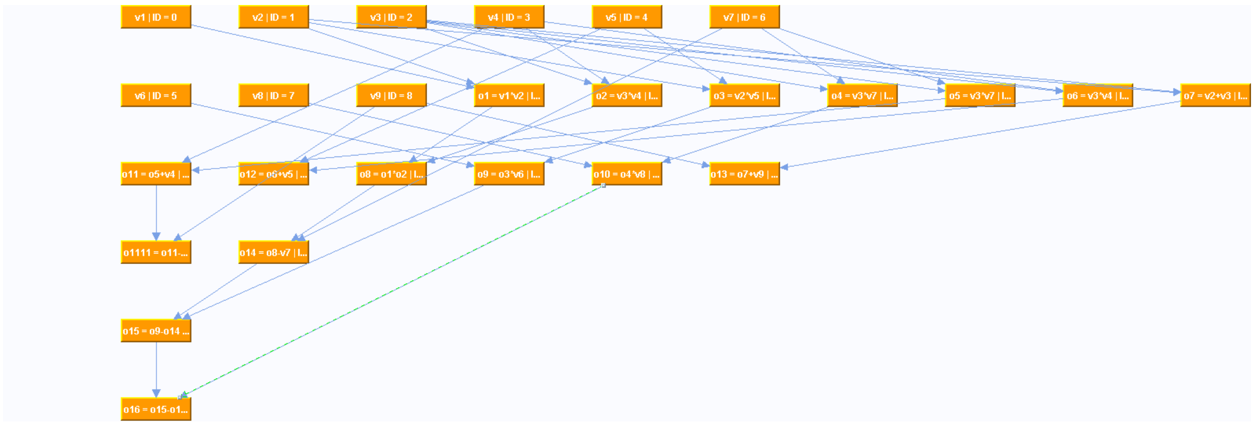


Figure 29: DiffEq4 circuit cdfg

Table 26: DiffEq4 total switching costs results

Resource bag	Normal	RSSA	Improvement
resource1	28	20	28.57
resource2	30	24	20.00
resource3	27	21	22.22
resource4	30	26	13.33
resource5	33	27	18.18
resource6	40	31	22.50
resource7	40	32	20.00
resource10	40	31	22.50
resource15	40	32	20.00
resource20	40	32	20.00
Average			20.73

Table 27: DiffEq4 spurious switching costs

Normal	RSSA	Improvement
11	3	46.15
13	7	60.00
10	4	30.77
13	9	37.50
16	10	39.13
23	14	34.78
23	15	39.13
23	14	34.78
23	15	34.78
23	15	46.15
		42.98

ArFilter circuit (28 nodes)

The ArFilter circuit is the AR lattice filter and contains 28 operations. The CDFG, the total switching, and the spurious switching results for this circuit are shown in Figure 30, Table 28, and Table 29 respectively.

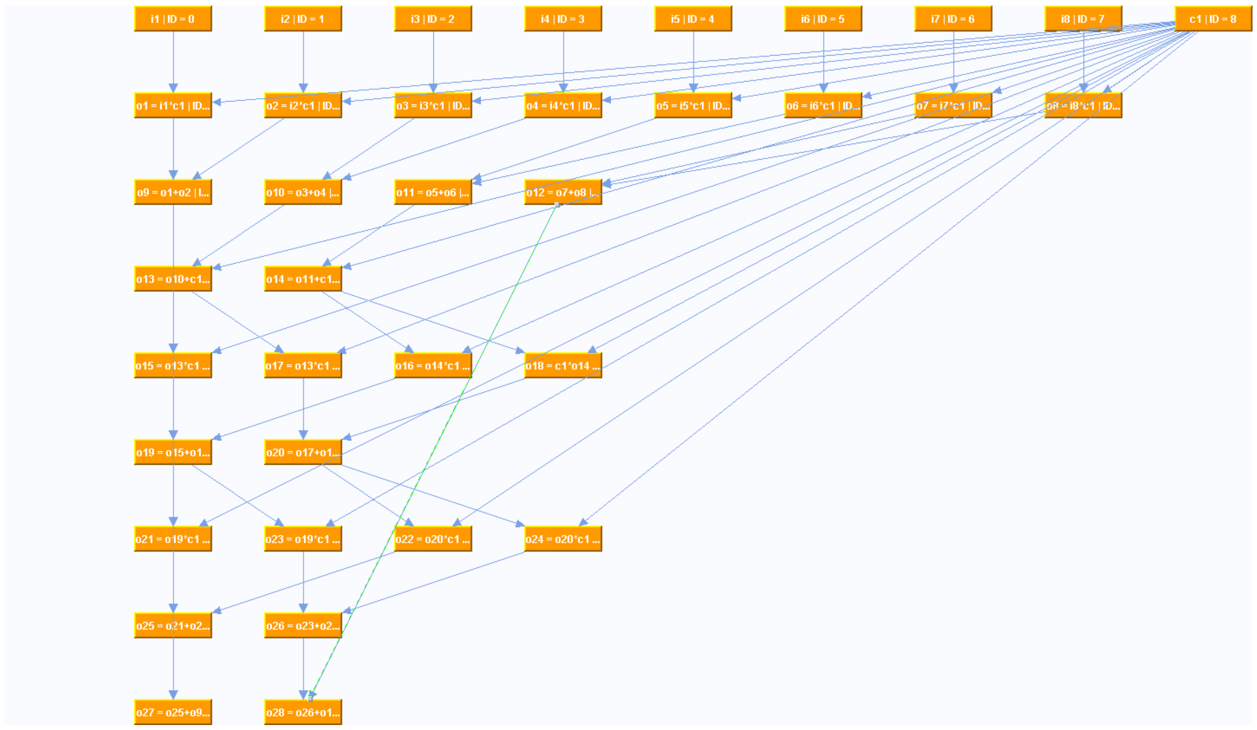


Figure 30: ARFilter circuit cdfg

Table 28: ARFilter total switching costs results

Resource bag	Normal	RSSA	Improvement
resource1	34	33	2.94
resource2	41	33	19.51
resource3	45	40	11.11
resource4	61	51	16.39
resource5	76	51	32.89
resource6	77	56	27.27
resource7	79	51	35.44
resource10	87	60	31.03
resource15	87	59	32.18
resource20	87	60	31.03
Average			23.98

Table 29: ARFilter spurious switching costs

Normal	RSSA	Improvement
6	5	16.67
13	5	61.54
17	12	29.41
33	23	30.30
48	23	52.08
49	28	42.86
51	23	54.90
59	32	45.76
59	31	47.46
59	32	45.76
		42.67

Elliptic circuit (34 nodes)

The Elliptic circuit is the fifth order elliptic wave filter and contains 34 operations. The CDFG, the total switching, and the spurious switching results for this circuit are shown in Figure 31, Table 30, and Table 31 respectively.

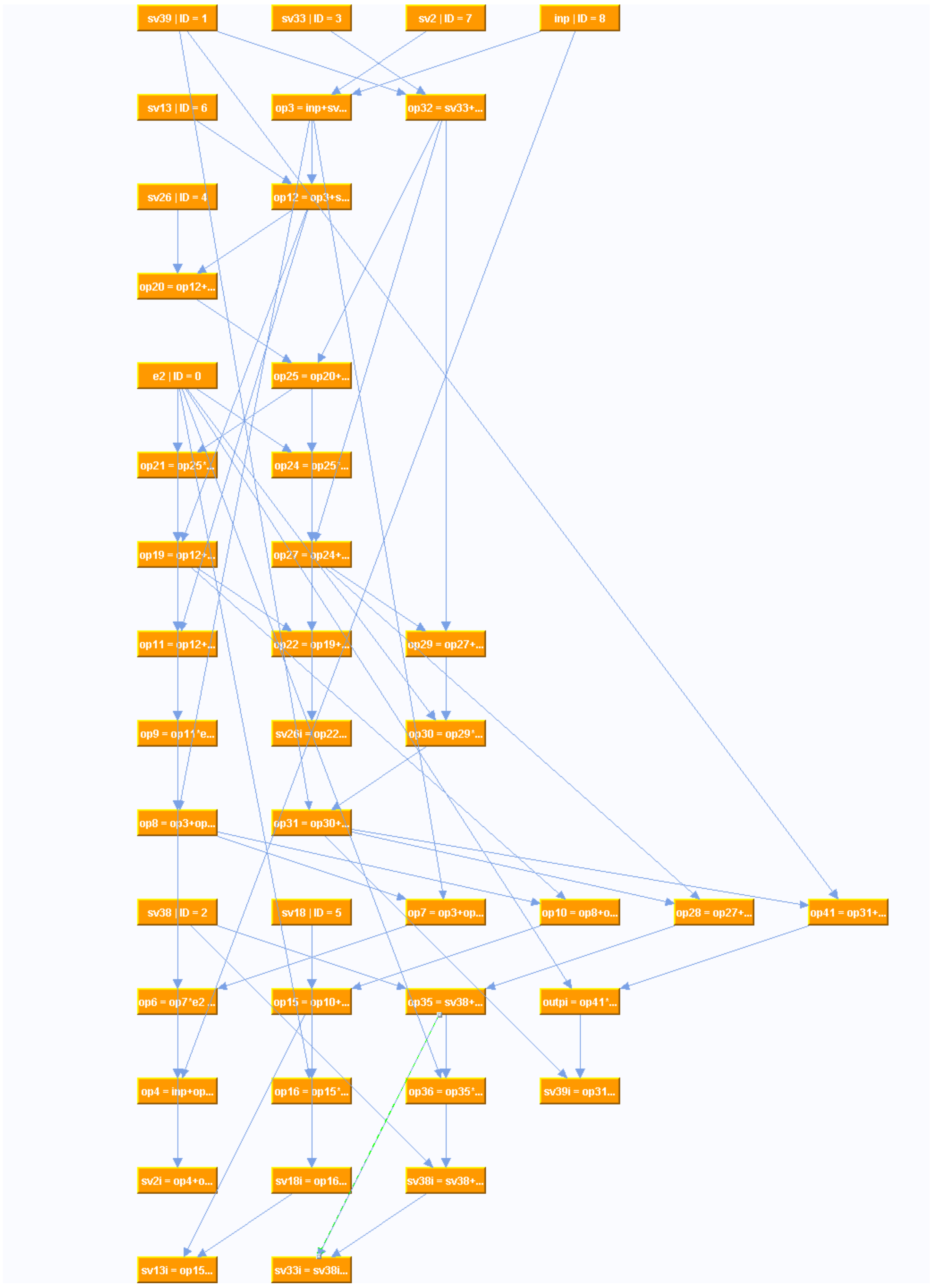

sv39 | ID = 1
sv33 | ID = 3
sv2 | ID = 7
inp | ID = 8
sv13 | ID = 6
op3 = inp+sv...
op32 = sv33+...
sv26 | ID = 4
op12 = op3+s...
op20 = op12+...
e2 | ID = 0
op25 = op20+...
op21 = op25*...
op24 = op25*...
op19 = op12+...
op27 = op24+...
op11 = op12+...
op22 = op19+...
op29 = op27+...
op9 = op11*e...
sv26i = op22...
op30 = op29*...
op8 = op3+op...
op31 = op30+...
sv38 | ID = 2
sv18 | ID = 5
op7 = op3+op...
op10 = op8+o...
op28 = op27+...
op41 = op31+...
op6 = op7*e2 ...
op15 = op10+...
op35 = sv38+...
outpi = op41*...
op4 = inp+op...
op16 = op15*...
op36 = op35*...
sv39i = op31...
sv2i = op4+o...
sv18i = op16...
sv38i = sv38+...
sv13i = op15...
sv33i = sv38i...

Figure 31: Elliptic circuit cdfg

Table 30: Elliptic total switching costs results

Resource bag	Normal	RSSA	Improvement
resource1	42	37	11.90
resource2	46	38	17.39
resource3	61	47	22.95
resource4	68	45	33.82
resource5	68	45	33.82
resource6	68	46	32.35
resource7	68	45	33.82
resource10	68	45	33.82
resource15	68	45	33.82
resource20	68	46	32.35
Average			28.61

Table 31: Elliptic spurious switching costs

Normal	RSSA	Improvement
8	3	62.50
12	4	66.67
27	13	51.85
34	11	67.65
34	11	67.65
34	12	64.71
34	11	67.65
34	11	67.65
34	11	67.65
34	12	64.71
		64.87

DCT1 circuit (35 nodes)

The DCT1 circuit is a simplified form of the discrete cosine transform benchmark and contains 35 operations. The CDFG, the total switching, and the spurious switching results for this circuit are shown in Figure 32, Table 32, and Table 33 respectively.

Figure 32: DCT1 circuit cdfg

Table 32: DCT1 total switching costs results

Resource bag	Normal	RSSA	Improvement
resource1	42	39	7.14
resource2	52	41	21.15
resource3	56	45	19.64
resource4	57	47	17.54
resource5	59	48	18.64
resource6	67	53	20.90
resource7	53	48	9.43
resource10	79	55	30.38
resource15	94	74	21.28
resource20	96	80	16.67
Average			18.28

Table 33: DCT1 spurious switching costs

Normal	RSSA	Improvement
7	4	42.86
17	6	64.71
21	10	52.38
22	12	45.45
24	13	45.83
32	18	43.75
18	13	27.78
44	20	54.55
59	39	33.90
61	45	26.23
		43.74

Nestor circuit (48 nodes)

The Nestor circuit contains 48 operations. The CDFG, the total switching, and the spurious switching results for this circuit are shown in Figure 33, Table 34, and Table 35 respectively.

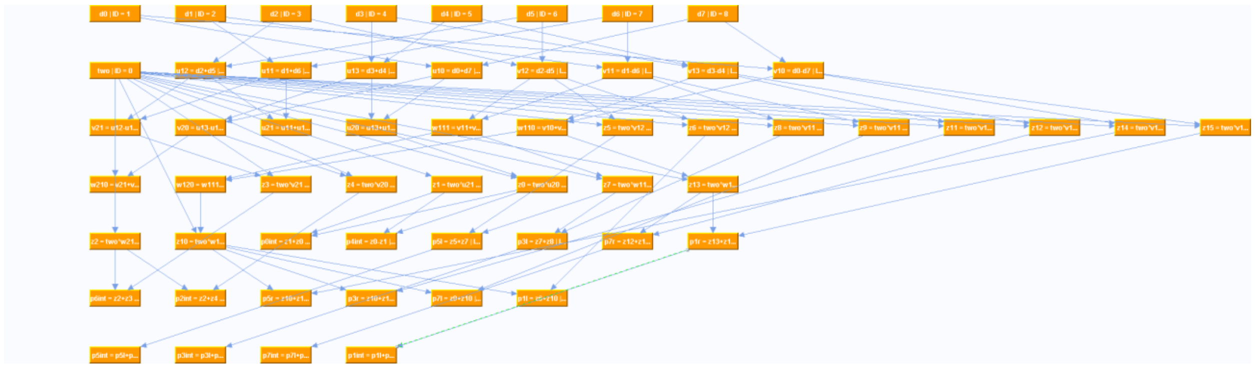


Figure 33: Nestor circuit cdfg

Table 34: Nestor total switching costs results

Resource bag	Normal	RSSA	Improvement
resource1	56	54	3.57
resource2	60	56	6.67
resource3	70	61	12.86
resource4	74	70	5.41
resource5	91	73	19.78
resource6	94	84	10.64
resource7	104	87	16.35
resource10	108	88	18.52
resource15	108	89	17.59
resource20	108	90	16.67
Average			12.80

Table 35: Nestor spurious switching costs

Normal	RSSA	Improvement
8	6	25.00
12	8	33.33
22	13	40.91
26	22	15.38
43	25	41.86
46	36	21.74
56	39	30.36
60	40	33.33
60	41	31.67
60	42	30.00
		30.36

DCT circuit (70 nodes)

The DCT circuit is the discrete cosine transform benchmark and contains 70 operations. The CDFG, the total switching, and the spurious switching results for this circuit are shown in Figure 34, Table 36, and Table 37 respectively.

Figure 34: DCT circuit cdfg

Table 36: DCT total switching costs results

Resource bag	Normal	RSSA	Improvement
resource1	80	73	8.75
resource2	86	75	12.79
resource3	89	74	16.85
resource4	95	76	20.00
resource5	108	81	25.00
resource6	99	82	17.17
resource7	106	87	17.92
resource10	107	97	9.35
resource15	108	107	0.93
resource20	132	119	9.85
Average			13.86

Table 37: DCT spurious switching costs

Normal	RSSA	Improvement
10	3	70.00
16	5	68.75
19	4	78.95
25	6	76.00
38	11	71.05
29	12	58.62
36	17	52.78
37	27	27.03
38	37	2.63
62	49	20.97
		52.68

FFT circuit (113 nodes)

The DCT circuit is the fast fourier transform benchmark and contains 113 operations. The CDFG, the total switching, and the spurious switching results for this circuit are shown in Figure 35, Table 38, and Table 39 respectively.

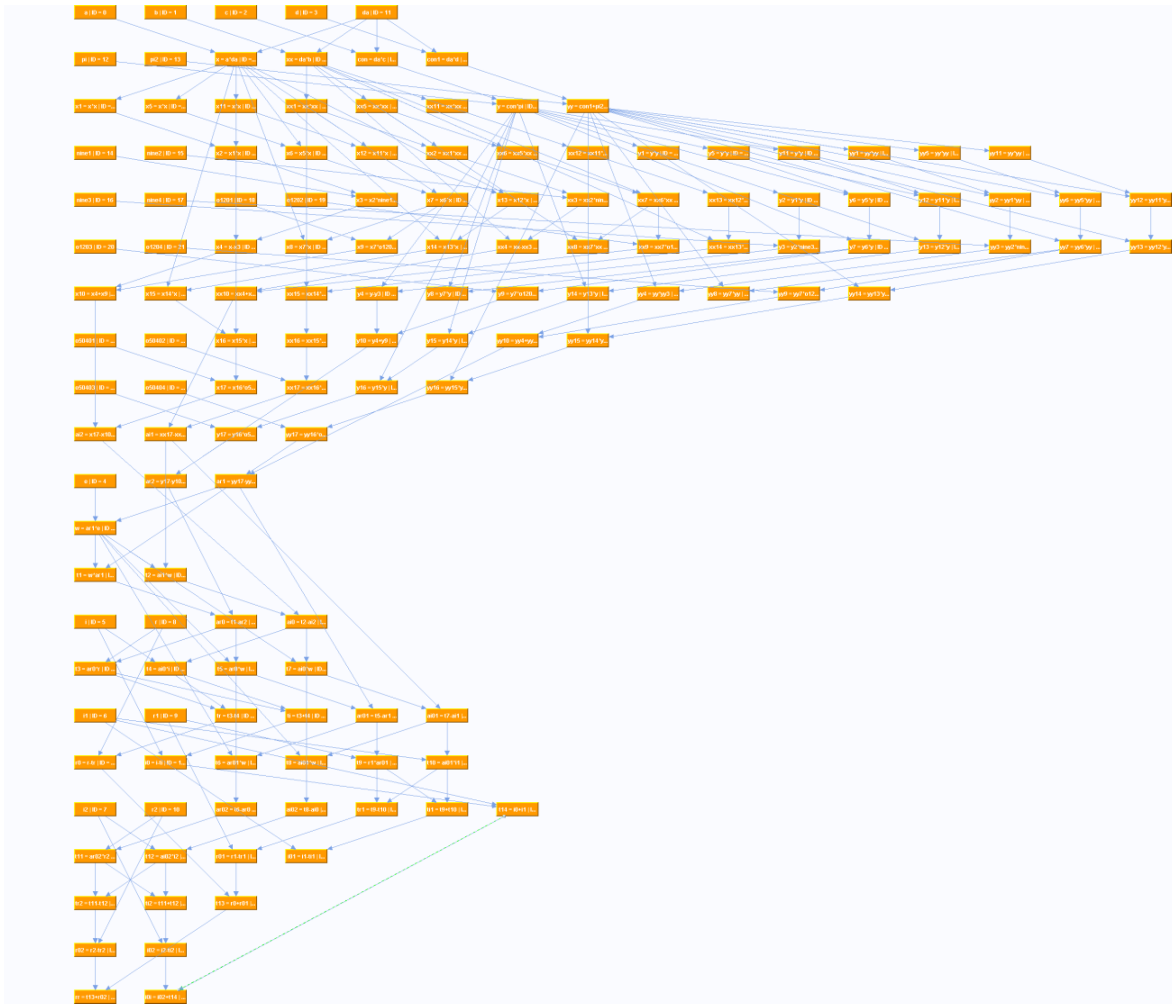


Figure 35: FFT circuit cdfg

Table 38: FFT total switching costs results

Resource bag	Normal	RSSA	Improvement
resource1	153	129	15.69
resource2	178	138	22.47
resource3	191	151	20.94
resource4	196	149	23.98
resource5	191	161	15.71
resource6	188	163	13.30
resource7	201	168	16.42
resource10	239	194	18.83
resource15	256	206	19.53
resource20	256	203	20.70
Average			18.76

Table 39: FFT spurious switching costs

Normal	RSSA	Improvement
40	16	60.00
65	25	61.54
78	38	51.28
83	36	56.63
78	48	38.46
75	50	33.33
88	55	37.50
126	81	35.71
143	93	34.97
143	90	37.06
		44.65

All circuits show improvements in both total-switching and spurious-switching activities. The averages of these improvements are 17.86% and 39.22% respectively.

Chapter 6: Conclusions

In this work, we introduced a new approach to power manage circuits by reducing spurious switching activity in functional units through proper register binding. For this purpose, we explored how conventional register binding can affect the spurious switching and hence the total power consumption. Three type of spurious switching were identified and the room for improvement in regular binding techniques was discussed. To test this new approach, we developed a synthesis environment and implemented and integrated a cool-down simulated annealing process to probe a good approximate to the minimal switching cost. Results showed that, on average, spurious switching activity is reduced by 40 % which in turn reduces total switching by 18%. Future work will focus on further improving spurious switching by better targeting the first and third types of switching activities discussed earlier and incorporating area cost into the simulated annealing process.

Bibliography

[1] Pauling, P. & Knight, J. (1989, July). Force directed scheduling for the behavioral synthesis of ASIC's. *IEEE Transactions on CAD/ICAS*, *CAD-8* (6), 661-679.

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

[3] Lakshminarayana, G., Raghunathan, A., Jha, N. K., & Dey, S. (1999, March). Power management in high-level synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 7* (1).

[4] Mussoll, E. & Cortadella, J. (1995, April). High-level synthesis techniques for reducing the activity of functional units. In *Proceedings of the International Symposium on Low-Power Design* (pp.99–104).

[5] Lakshminarayana, G., Raghunathan, A., Jhat, N. K., & Dey, S. (1998, November). Transforming control-flow intensive designs to facilitate power management. In *Proceedings of the International Conference Computer-Aided Design* (657–664).

[6] Luo, J., Zhong, L., Fei, Y., & Jha, N. K. (2004, August). Register binding-based RTL power management for control-flow intensive designs. *IEEE Transactions on CAD/ICAS, 23* (8).

[7] Dal, D., Kutagulla, D., Nunez, A., & Mansouri, N. (2005). Power islands: A high-level synthesis technique for reducing spurious switching activity

and leakage. In *IEEE International Midwest Symposium on Circuits and Systems*.

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

[9] Garey, M. & Johnson, D. (1979). *Computers and intractability*. New York: Freeman.

[10] Tseng, C. J. & Siewiorek, D. (1986, July). Automated synthesis of data paths in digital systems. *IEEE Transactions on CAD/ICAS*, *CAD-5* (3), 379-395.

[11] Hashimoto, A. & Stevens, J. (1971). Wire routing by optimizing channel assignment within large apertures. In *Proceedings of the 8th Design Automation Workshop* (155-163).

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

[13] Zhong, L. & Jha, N. K. (2002). Interconnect-aware high-level synthesis for low power. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided design*.

[14] Chandrakasan, A. P., Sheng, S., & Broadersen, R. W. (1992, April). Low power CMOS digital design. *IEEE Journal of Solid State Circuits*, *27*(4), 472-484.

[15] Pedram, M. (2007, October). *Leakage power modeling and minimization*: *ICCAD 2004 Tutorial*. Retrieved July 12, 2008, from the University of Southern California, Department of Electrical Engineering Systems Web site: http://atrak.usc.edu/~massoud/Papers/pedram-tutorial-iccad04.pdf

[16] Abi Saad, M. (2009, January). Efficient area reduction in high-level synthesis using priority-driven simulated annealing. *Unpublished master's thesis, Lebanese American University*, *Byblos, Lebanon*.

Appendix A

Eridanus Generated Datapath code

```
LIBRARY IEEE;

USE IEEE.STD_LOGIC_1164.ALL;



ENTITY datapath IS

PORT(

x , y : IN STD_LOGIC_VECTOR(3 downto 0);

mux_level1_1strobe , mux_level2_3strobe : IN STD_LOGIC_VECTOR(1 downto 0);

mux_level1_0strobe , mux_level2_2strobe : IN STD_LOGIC_VECTOR(0 downto 0);

reg_0strobe , reg_1strobe , reg_2strobe : IN STD_LOGIC;

RESET , CLOCK : IN STD_LOGIC;

converger2out_port :OUT STD_LOGIC_VECTOR(5 downto 0)

);

END datapath;



ARCHITECTURE structural OF datapath IS

COMPONENT greater_4

PORT(D0, D1 : IN     STD_LOGIC_VECTOR(3 downto 0);

    Q : OUT     STD_LOGIC_VECTOR(0 downto 0)

);
```

```
END COMPONENT;

COMPONENT adder_5

PORT(A, B: IN          STD_LOGIC_VECTOR(4 downto 0);

    Sum           : OUT  STD_LOGIC_VECTOR(5 downto 0)

);

END COMPONENT;

COMPONENT converger_6

PORT(A, B: IN          STD_LOGIC_VECTOR(5 downto 0);

    Sel: IN  STD_LOGIC;

    Q : OUT     STD_LOGIC_VECTOR(5 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_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_3inputs_6bits

PORT(D0 , D1 , D2: IN         STD_LOGIC_VECTOR(5 downto 0);

    Sel: IN       STD_LOGIC_VECTOR(1 downto 0);

    Q : OUT     STD_LOGIC_VECTOR(5 downto 0)

);

END COMPONENT;

COMPONENT multiplexer_2inputs_5bits

PORT(D0 , D1: IN     STD_LOGIC_VECTOR(4 downto 0);

    Sel: IN       STD_LOGIC_VECTOR(0 downto 0);

    Q : OUT     STD_LOGIC_VECTOR(4 downto 0)

);

END COMPONENT;

COMPONENT multiplexer_3inputs_3bits

PORT(D0 , D1 , D2: 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 greater0_input0 , greater0_input1 , reg1out , mux_level1_0out ,
multiplexer0_input0 : STD_LOGIC_VECTOR(3 downto 0);
SIGNAL adder1out , converger2out , converger2_input2 , reg0out , reg2out ,
mux_level1_1out , multiplexer1_input0 : STD_LOGIC_VECTOR(5 downto 0);
SIGNAL greater0out : STD_LOGIC_VECTOR(0 downto 0);
SIGNAL mux_level2_3out , multiplexer3_input0 : STD_LOGIC_VECTOR(2 downto
0);
SIGNAL adder1_input1 , mux_level2_2out , multiplexer2_input0 , multiplexer2_input1
: STD_LOGIC_VECTOR(4 downto 0);
CONSTANT constant_1 :STD_LOGIC_VECTOR(1 downto 0):="01";
CONSTANT constant_0 :STD_LOGIC_VECTOR(1 downto 0):="00";
CONSTANT constant_2 :STD_LOGIC_VECTOR(2 downto 0):="010";
CONSTANT constant_3 :STD_LOGIC_VECTOR(2 downto 0):="011";

BEGIN

--Creating a wrapper signal multiplexer0_input0 for greater0out
multiplexer0_input0<=(3 downto 1=>greater0out(0)) & greater0out;
mux_level1_0 :multiplexer_2inputs_4bits PORT MAP(multiplexer0_input0 , y ,
mux_level1_0strobe , mux_level1_0out);


--Creating a wrapper signal multiplexer1_input0 for x
multiplexer1_input0<=(5 downto 4=>x(3)) & x;
mux_level1_1 :multiplexer_3inputs_6bits PORT MAP(multiplexer1_input0 ,
converger2out , adder1out , mux_level1_1strobe , mux_level1_1out);


reg0 :register_6 PORT MAP(adder1out , RESET , CLOCK , reg_0strobe , reg0out);
reg1 :register_4 PORT MAP(mux_level1_0out , RESET , CLOCK , reg_1strobe ,
reg1out);
reg2 :register_6 PORT MAP(mux_level1_1out , RESET , CLOCK , reg_2strobe ,
reg2out);



--Creating a wrapper signal multiplexer2_input0 for reg1out
multiplexer2_input0<=(4 downto 4=>reg1out(3)) & reg1out;


--Creating a wrapper signal multiplexer2_input1 for reg0out
multiplexer2_input1<=reg0out(4 downto 0);
mux_level2_2 :multiplexer_2inputs_5bits PORT MAP(multiplexer2_input0 ,
multiplexer2_input1 , mux_level2_2strobe , mux_level2_2out);


--Creating a wrapper signal multiplexer3_input0 for constant_1
```

```
multiplexer3_input0<=(2 downto 2=>constant_1(1)) & constant_1;

mux_level2_3 :multiplexer_3inputs_3bits PORT MAP(multiplexer3_input0 , constant_2
, constant_3 , mux_level2_3strobe , mux_level2_3out);



--Creating a wrapper signal greater0_input0 for reg2out
greater0_input0<=reg2out(3 downto 0);

--Creating a wrapper signal greater0_input1 for constant_0
greater0_input1<=(3 downto 2=>constant_0(1)) & constant_0;
greater0 :greater_4 PORT MAP(greater0_input0 , greater0_input1 , greater0out);

--Creating a wrapper signal adder1_input1 for mux_level2_3out
adder1_input1<=(4 downto 3=>mux_level2_3out(2)) & mux_level2_3out;
adder1 :adder_5 PORT MAP(mux_level2_2out , adder1_input1 , adder1out);

--Creating a wrapper signal converger2_input2 for reg1out
converger2_input2<=(5 downto 4=>reg1out(3)) & reg1out;
converger2 :converger_6 PORT MAP(reg2out , reg0out , converger2_input2 ,
converger2out);

converger2out_port<=converger2out;
```

```
END structural;
```

Eridanus Generated Controller code

```
LIBRARY IEEE;

USE IEEE.STD_LOGIC_1164.ALL;



ENTITY controller IS

PORT(

CLOCK , RESET : IN STD_LOGIC;

mux_level1_1strobe , mux_level2_3strobe : OUT STD_LOGIC_VECTOR(1 downto 0);

mux_level1_0strobe , 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);

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<=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_1strobe <= "00";

mux_level1_0strobe <= "1";

reg_2strobe <= '1';

reg_1strobe <= '1';

reg_0strobe <= '0';


WHEN state1 =>

mux_level1_0strobe <= "0";

mux_level2_2strobe <= "0";

mux_level2_3strobe <= "00";

reg_0strobe <= '1';

reg_1strobe <= '1';

reg_2strobe <= '0';


WHEN state2 =>

mux_level1_1strobe <= "10";

mux_level2_2strobe <= "1";

mux_level2_3strobe <= "01";

reg_2strobe <= '1';

reg_0strobe <= '0';

reg_1strobe <= '0';
```

```
WHEN state3 =>

mux_level2_2strobe <= "1";

mux_level2_3strobe <= "10";

reg_0strobe <= '1';

reg_1strobe <= '0';

reg_2strobe <= '0';


WHEN state4 =>

mux_level1_1strobe <= "01";

reg_2strobe <= '1';

reg_0strobe <= '0';

reg_1strobe <= '0';

END CASE;

END PROCESS;


END behavioral;
```