Lebanese American University

A Graph Heuristic Approach for the Data Path Allocation Problem

By

Racha Makhoul

A thesis

Submitted in partial fulfillment of the requirements

for the degree of Master of Science in Computer Science

School of Arts and Sciences

December 2022

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

THESIS APPROVAL FORM

Student Name: Racha Makhoul I.D. #: 201202603

Thesis Title: A Graph Theoretical Approach for the Allocation Problem in HLS

Program: Computer Science

Department: Computer Science and Mathematics

School: Arts and Sciences

The undersigned certify that they have examined the final electronic copy of this thesis and approved it in Partial Fulfillment of the requirements for the degree of:

Master of Science in the major of Computer Science

Thesis Advisor's Name: Haidar Harmanani

Signature: [redacted] Date: 21 / 12 / 2022 (Day / Month / Year)

Committee Member's Name: Chadi Nour

Signature: [redacted] Date: 21 / 12 / 2022 (Day / Month / Year)

Committee Member's Name: Nader El-Khatib

Signature: [redacted] Date: 21 / 12 / 2022 (Day / Month / Year)

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

THESIS COPYRIGHT RELEASE FORM

LEBANESE AMERICAN UNIVERSITY NON-EXCLUSIVE DISTRIBUTION LICENSE

By signing and submitting this license, you (the author(s) or copyright owner) grants the Lebanese American University (LAU) the non-exclusive right to reproduce, translate (as defined below), and/or distribute your submission (including the abstract) worldwide in print and electronic formats and in any medium, including but not limited to audio or video. You agree that LAU may, without changing the content, translate the submission to any medium or format for the purpose of preservation. You also agree that LAU may keep more than one copy of this submission for purposes of security, backup and preservation. You represent that the submission is your original work, and that you have the right to grant the rights contained in this license. You also represent that your submission does not, to the best of your knowledge, infringe upon anyone's copyright. If the submission contains material for which you do not hold copyright, you represent that you have obtained the unrestricted permission of the copyright owner to grant LAU the rights required by this license, and that such third-party owned material is clearly identified and acknowledged within the text or content of the submission. IF THE SUBMISSION IS BASED UPON WORK THAT HAS BEEN SPONSORED OR SUPPORTED BY AN AGENCY OR ORGANIZATION OTHER THAN LAU, YOU REPRESENT THAT YOU HAVE FULFILLED ANY RIGHT OF REVIEW OR OTHER OBLIGATIONS REQUIRED BY SUCH CONTRACT OR AGREEMENT. LAU will clearly identify your name(s) as the author(s) or owner(s) of the submission, and will not make any alteration, other than as allowed by this license, to your submission.

Name: Racha Makhoul

Signature: 

Date: 17 / 05 / 2020
Day Month Year

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

PLAGIARISM POLICY COMPLIANCE STATEMENT

I certify that:

1. I have read and understood LAU's Plagiarism Policy.
2. I understand that failure to comply with this Policy can lead to academic and disciplinary actions against me.
3. This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that by acknowledging its sources.

Name: Racha Makhoul

Signature: 

Date: 17 / 05 / 2022
Day Month Year

Acknowledgements

This project would not have been possible without the support of many people. Many thanks to my advisor, Dr. Haidar Harmanani, who read my numerous revisions and helped make some sense of the confusion. Also thanks to my committee members, Dr. Nader El Khatib, and Dr. Chadi Nour, who offered guidance and support. And finally, thanks to my parents, and numerous friends who endured this long process with me, always offering support and love.

A Graph Heuristic Approach for the Data Path Allocation Problem

Racha Makhoul

ABSTRACT

The current escalation in usage and complexity of modern digital systems, and the emergence of Very-Large Scale Integration (VLSI) has led to a huge *design productivity gap* in the chip design industry. It is well established that the annual growth of the number of transistors on a chip surpasses that of transistors handled by a fixed-sized design team. This gap, represented by "Moore's law of engineers", raises the necessity for more efficient approaches to automatically design these advanced frameworks. This paper aims to reduce the design productivity gap by presenting a new algorithm to optimize chip design automation in terms of runtime and solution

quality. More specifically, this automation process known as high-level synthesis (HLS), targets the datapath allocation problem in VLSI design. In this context, datapath allocation represents one of the major steps in HLS along with scheduling and partitioning.

To optimize this allocation problem, a variation of the Fiduccia-Mattheyses (FM) algorithm is presented. The algorithm starts with a schedule as input and partitions it into available hardware resources using a modified version of the Fiduccia-Mattheyses algorithm to fit the datapath allocation problem. The algorithm is tested on various benchmarks and favorable results are reported. The results show that the algorithm performs well while generating sub-optimal solutions.

Keywords: High-Level Synthesis, HLS, Very-Large Scale Integration, VLSI, Chip Design, Datapath Allocation, Scheduling, Partitioning, Fiduccia-Mattheyses.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

First patented in 1959, Integrated Circuits (ICs) have revolutionized electronics by allowing a single chip to hold thousands of elements such as resistors, capacitors, and transistors. The creation of ICs emerged from the need for faster, more efficient, and more accurate methods to design electronic devices, in addition to smaller surface areas for these assemblies [1]. Rather than using traditional circuits which are comprised of multiple components (see definition 2.1.11) such as transistors, resistors, diodes, capacitors, and wires on a board; Integrated Circuits are created using a silicon board that eliminates the need for many components (see definition 2.1.11), and effectively greatly reduces the size of the circuit. The importance of ICs, widely known as chips, lies in allowing for the miniaturization of circuit assemblies and paving the way for all the electronic devices we see today [2].

1.1 Moore's Law

In his manuscript to the patent department of Fairchild Semiconductor in 1965, Dr. in Physical Chemistry Gordon E. Moore stated that, "The promise of integrated electronics is extrapolated into the wild blue yonder to show that there is still much to be done, but that integrated electronics will pervade all of electronics in the future. A curve is shown to suggest that the most economical way to make electronic systems in some ten years will be of the order of 65,000 components per integrated circuit.". This statement along with an attached graph in **figure 1.1** showed Moore's prediction for the future of integrated electronics. More precisely, Moore predicted that the number of transistors that would fit on a chip area would double every year; this was later modified in 1975 to every 18 months. Proving to be accurate, this prediction became known as Moore's law [3].

This fast increase in demand and popularity of ICs created a gap between the human ability to deliver these complex designs, and the rate at which the industry was growing. And thus, the automation of the design process became crucial to minimize this *design productivity gap*, calling for the emergence of Electronic Design Automation (EDA) [1].

1.2 Electronic Design Automation

Due to their vast range of applications and need for different specifications in speed, performance and size, integrated circuits became very complex to design and produce.

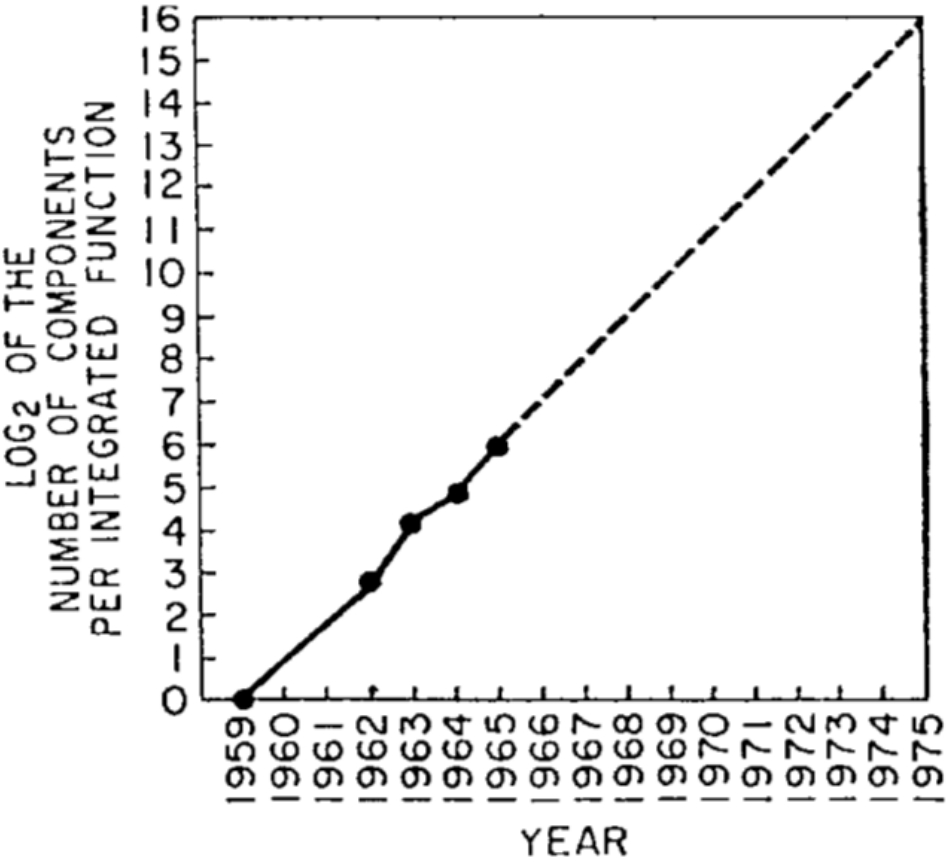


Figure 1.1: Graph representing Moore's Law, which predicts that the number of transistors on Integrated Circuits would double every year. [3]

The current advancement of chip design has especially been possible due to Electronic Design Automation (EDA) tools, that first emerged in 1960. These tools consist of programs used to automatically place components (see definition 2.1.11) on chips while optimizing several aspects of the design [1]. As we moved to larger scales reaching a density of one million components per chip, electronic circuits evolved from ICs to *Very Large-Scale Integration (VLSI)*. **Figure 1.2** shows the progress of EDA throughout the years with chip advancement.

These changes in scaling has driven the industry to opt for the automation of every step of the design flow. Since lower design levels, that include logic simulation (see definition 2.1.1) and synthesis (see definition 2.1.2), became harder to control and

Time Period	Circuit and Physical Design Process Advancements
1950-1965	Manual design only.
1965-1975	Layout editors, e.g., place and route tools, first developed for printed circuit boards.
1975-1985	More advanced tools for ICs and PCBs, with more sophisticated algorithms.
1985-1990	First performance-driven tools and parallel optimization algorithms for layout; better understanding of underlying theory (graph theory, solution complexity, etc.).
1990-2000	First over-the-cell routing, first 3D and multilayer placement and routing techniques developed. Automated circuit synthesis and routability-oriented design become dominant. Start of parallelizing workloads. Emergence of physical synthesis.
2000-now	Design for Manufacturability (DFM), optical proximity correction (OPC), and other techniques emerge at the design-manufacturing interface. Increased reusability of blocks, including intellectual property (IP) blocks.

Figure 1.2: EDA progress throughout years of advancement in chip design. [1]

understand, researchers started looking more into optimizing higher levels, in order to provide a simpler approach and to minimize the number of design objectives examined at later stages. Higher levels of the design look into problems related to circuit layout, including placement (see definition 2.1.8), routing (see definition 2.1.9) and floorplanning (see definition 2.1.10) and are known as *High level synthesis* [4].

1.3 High Level Synthesis

High Level Synthesis (HLS) refers to the EDA process that converts a high-level description of a design to a netlist (see definition 2.1.6). To further understand this concept, we differentiate two additional types of synthesis. **Figure 1.3** shows the synthesis types respectively as follows, layout synthesis, logic synthesis and high-level synthesis. The layout synthesis process takes an abstract structural

description (see definition 2.1.6) of the design and translates it to a physical layout (see definition 2.1.4). Logic synthesis takes a behavioral description (see definition 2.1.5) at register-transfer level (RTL), and turns it into a logic design (see definition 2.1.3). Additionally, high-level synthesis generates a structural RTL implementation of the design from a behavioral description [4].

HLS consists of three main design steps as follows, [4]

- **Partitioning:** deals with dividing the input structures into smaller substructures, based on some design criteria, and solves these substructures separately.
- **Scheduling:** deals with allocating operations 2.1.12 to a specific time-slot in a schedule, based on a set of constraints, that ensure no overlapping between consecutive operations.
- **Allocation:** deals with the mapping of inputs to hardware resources, these inputs are represented as operations (see definition 2.1.12) allocating to functional units (see definition 2.1.15), variables (see definition 2.1.13) allocating to storage units (see definition 2.1.16), and data transfers (see definition 2.1.14) allocating to interconnection units (see definition 2.1.17).

Note that these steps of HLS can be solved concurrently or sequentially without any specific ordering.

As was identified earlier, HLs must satisfy a set of design constraints provided with the input description. Evidently, we can identify a time-resource trade-off in the previously mentioned HLS processes. For example, starting off a design with a scheduling algorithm, and then proceeding to allocation will impose a time-bound

constraint on the allocation problem. On the other hand, starting with allocation will produce a resource-bound schedule [4]. Consequently, in this paper, we focus on a time-bound allocation problem that takes a scheduled operation as an input and allocates resources accordingly.

1.4 Problem Definition

One of the standard approaches to evaluate an algorithm is through its run time complexity (see definition 2.1.19) and solution quality. Datapath allocation is one of the basic operations executed in high-level synthesis; and yet, it's one of the hardest to optimize. The allocation steps consist of NP-Hard problems (see definition 2.1.20) which are most commonly solved using heuristic algorithms (see section 2.2.1).

The issue with heuristic algorithms lies in their high time complexity (see definition 2.1.19) and their poor to near-optimal solutions. Our goal in this work, is to propose a new time efficient algorithm that solves the allocation problem in HLS, and improves the solution quality of the problem.

1.5 Contributions

This research contributes to the advancement of the design automation field in VLSI by suggesting a new approach to solving the allocation problem. The approach looks at solving all subtasks of the problem together and takes advantage of the polynomial runtime of the Fiduccia-Mattheyses algorithm to derive a variation that fits our

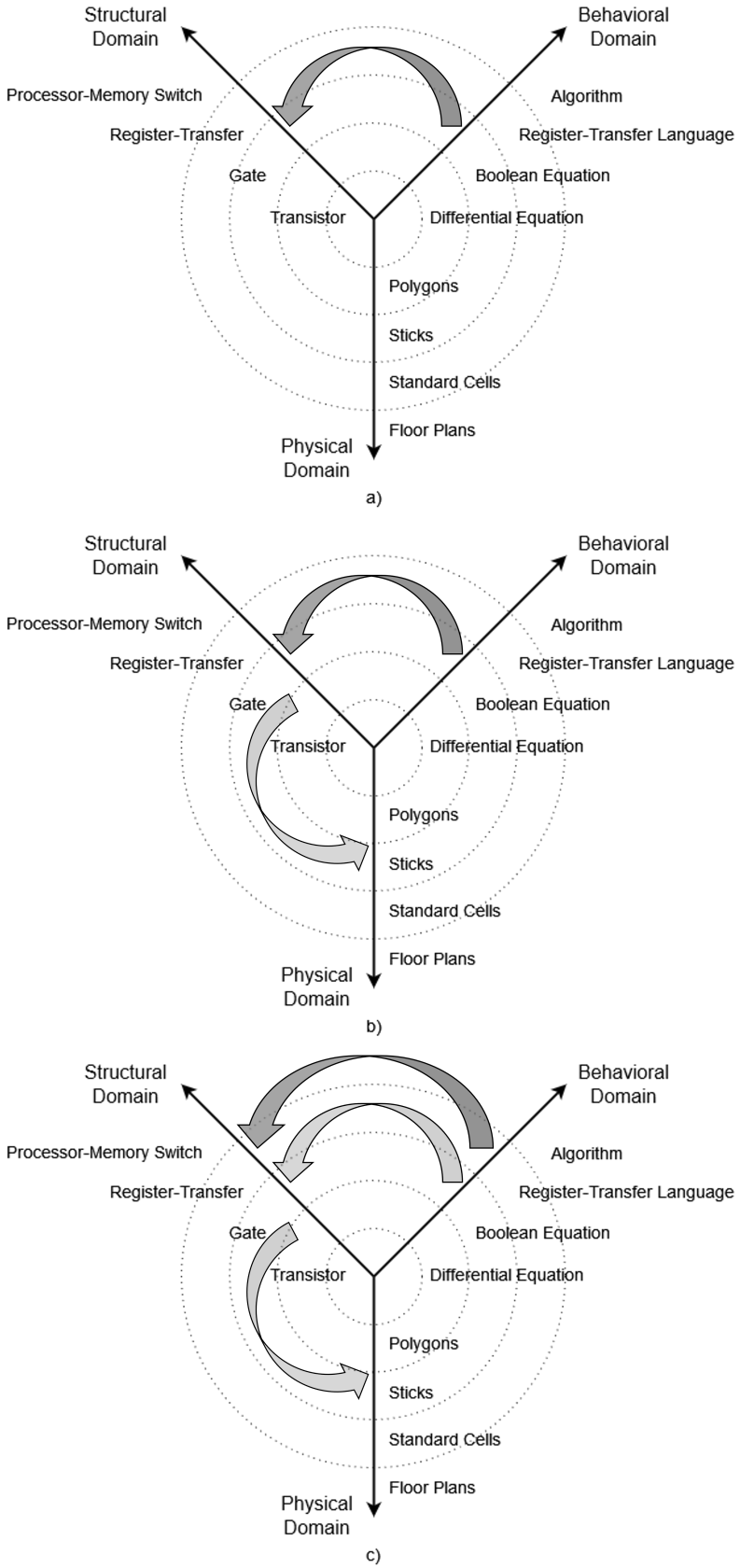


Figure 1.3: Different types of synthesis, a) layout synthesis, b) logic synthesis, c) high-level synthesis.

problem.

1.6 Outline

In this chapter, we explained how ICs revolutionized the electronic industry and scaled to VLSI, and how it was all made possible due to design automation and HLS. Chapter 2 will introduce some major terms and definitions needed throughout this work, with more focus on allocation as our main problem in chapter 3. We will then look into state of the art solutions that were made on VLSI design and the allocation problem in chapter 4. Chapter 5 will look into the proposed solution in detail and the reasoning behind it. Chapter 6 will examine the algorithm and the results. And finally we will conclude and consider some improvements and future work in chapter 7.

CHAPTER 2

PRELIMINARIES

2.1 Definitions and Terminology

Definition 2.1.1 (Logic/Circuit Simulation)**.** Logic simulation and circuit simulation are the testing techniques used to predict the output of a circuit and its performance before manufacturing. It is done by solving a set if differential equation representing of the circuit. **Figure 2.1** shows the simulation process to verify a design.

Logic simulations is used to check for any errors in the design during the logic design level. Whereas, circuit simulation is used to test digital cell libraries during the circuit design stage [5].

Definition 2.1.2 (Synthesis)**.** Synthesis or design refinement, is defined as the transformation of a simple behavioral description, to a more detailed structural description that follows a certain set of constraints [4].

Definition 2.1.3 (Logic Design)**.** Logic design is the step in IC design that deals

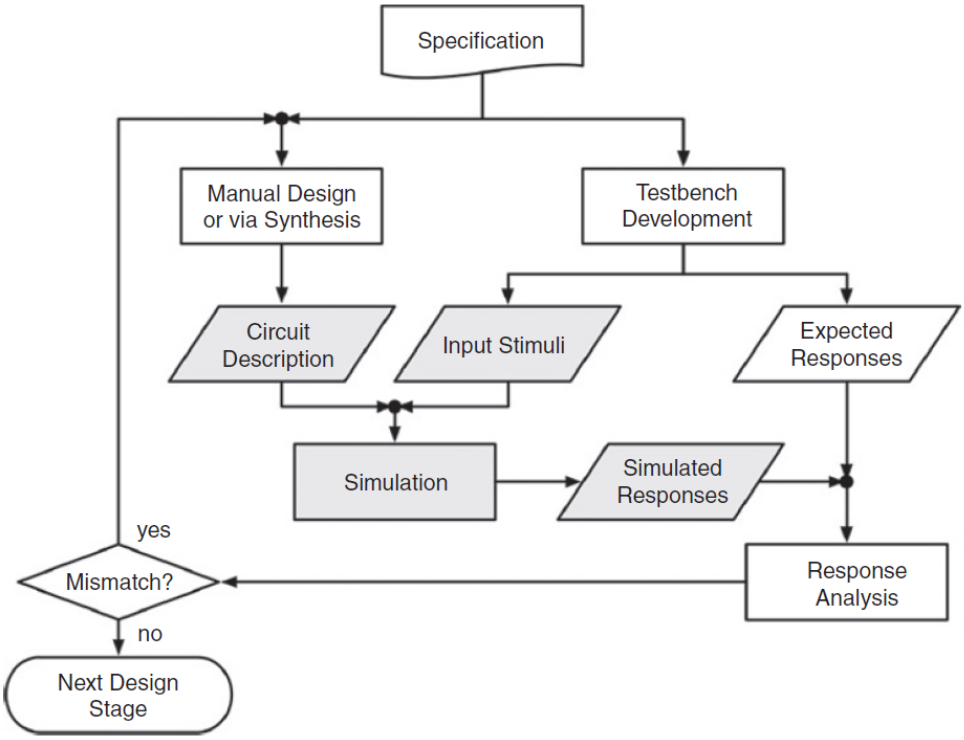


Figure 2.1: Simulation process to verify a design. [5]

with functional specifications and components on chip. It uses memory blocks and operations, verifying that expected output meets the functionality [5].

Definition 2.1.4 (Physical Design)**.** Physical design is the step in IC design that deals with circuit layout, and optimizes component placement to meet area and size specifications [5].

Definition 2.1.5 (Behavioral Description)**.** Behavioral descriptions consists of an algorithmic description of the expected behavior of a circuit. This typically means a set of expected outputs as a response to a certain set of inputs [5].

Definition 2.1.6 (Structural Description/Netlist)**.** A structural description is a description of the circuit board containing detailed component interconnections. It is often called a netlist [5].

Definition 2.1.7 (Physical Description)**.** A physical description disregards as much

as possible the functionality of a circuit, and represents the components in terms of space, area and structure on a board or a silicon [4]. **Figure 2.2** shows an example of each of the above mentioned design descriptions.

Definition 2.1.8 (Placement)**.** Placement defines the position of each component in the circuit to a specific location on the block i.e. a circuit partition [1].

Definition 2.1.9 (Routing)**.** In routing, connections are allocated to routing tracks [1].

Definition 2.1.10 (Floorplanning)**.** Floorplanning establishes the shapes and locations of blocks and circuit partitions within a circuit, additionally to the positioning of external ports [1].

Definition 2.1.11 (Component)**.** A component is basic functional part of a circuit such as a capacitor, resistor or transistor [1].

Definition 2.1.12 (Operation)**.** Operations are a set of tasks that a circuit is expected to perform. For example, in **equation 2.1** the plus sign '+' represents and adding operation.

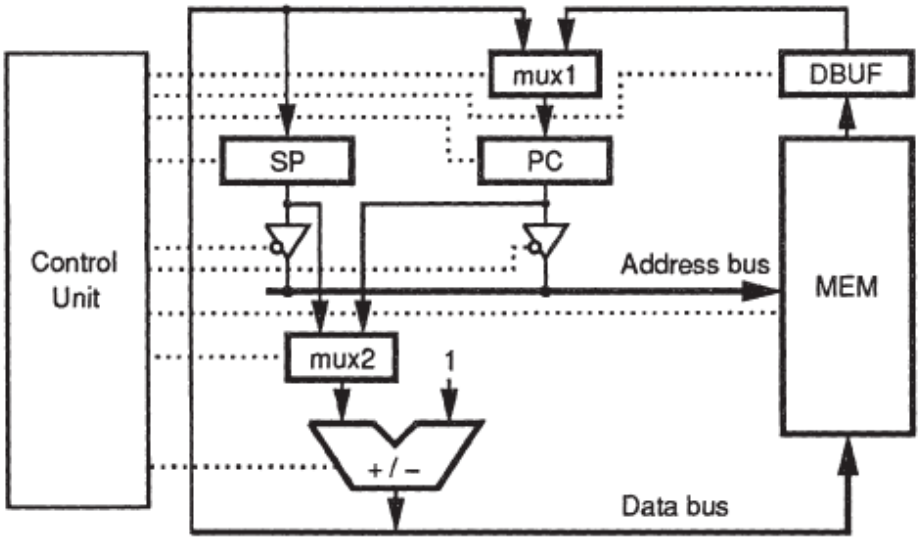
$$MEM(SP) = PC + 1; \tag{2.1}$$

Definition 2.1.13 (Variables)**.** Variables are the data entered as an input to an operation, in order to produce another variable output. This is represented in the above **equation 2.1** as the variables 'PC' and 'MEM(SP)'.

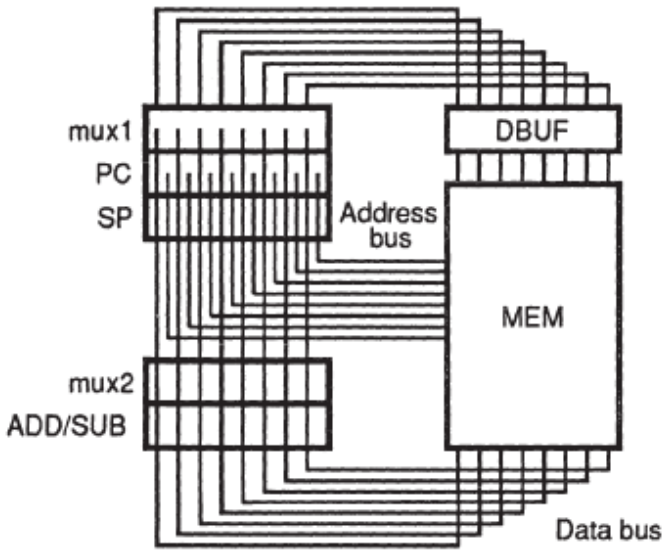
Definition 2.1.14 (Data Transfers)**.** Data transfers are the connections between variables, operations, and their outputs. In **figure 2.2 b)**, data transfers are represented as arrows connecting the operations and variables together.

```
if IR(3) = '0' then
    PC          := PC + 1;
else
    DBUF        := MEM(PC);
    MEM(SP)     := PC + 1;
    SP          := SP – 1;
    PC          := DBUF;
end if;
```

(a)



(b)



(c)

Figure 2.2: Example of the three types of descriptions a) behavioral description b) structural description c) physical description. [4]

Definition 2.1.15 (Functional Units)**.** Functional units (FUs) take data variables as an input and performs some transformations on them. Typical functional units include adders, multipliers, ALUs, comparators, shifters and selectors [4].

Definition 2.1.16 (Storage Units)**.** Storage units preserve the data inputs of FUs over a period of time. Registers, and memories are some basic examples of storage units [4].

Definition 2.1.17 (Interconnection Units)**.** Interconnection units are hardware resources that connect several components on a chip; they include buses, multiplexers and wires [4].

Definition 2.1.18 (Lifetime)**.** A variable's lifetime is defined as the time said variable is first active, until the time its last use. We say the variable is "Live" during this time and "Dead" after that [6].

Definition 2.1.19 (Time Complexity/Run Time)**.** The run time of an algorithm is the time required for an algorithm to finish execution and is used as a measure of the performance of an algorithm. This is typically measured in terms of the frequency of execution of the input, and known as the order of growth. In other words, the order of growth represents how much an algorithm's run time will increase with the increase of the input size.**Figure 2.3** shows an example of how the function of different algorithms is growing with the input, which represents the order of growth [7].

Throughout this work, the run time of an algorithm will be expressed using the big-O notation, which identifies the asymptotic upper bound of the performance of an algorithm, i.e. the run time in the worst case scenario [7].

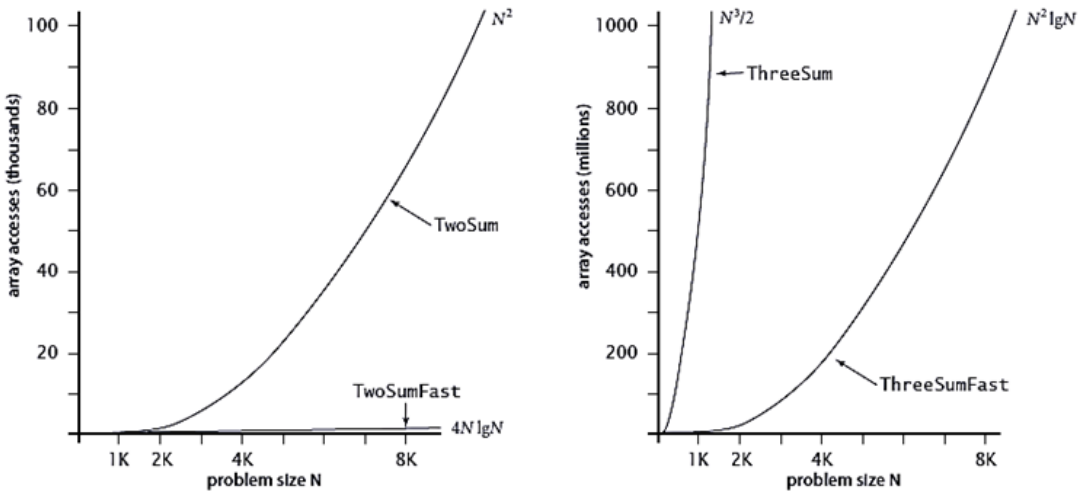


Figure 2.3: Order of growth of different algorithms. [7]

Definition 2.1.20 (NP-Hard Problems)**.** To define the NP-hard problems we start by defining NP problems. NP problems are non-deterministic polynomial problems; which are problems not solvable in polynomial time, but can be verified in that time. We can differentiate other two types of problems belong to the NP set of problems, these problems are polynomial problems (P) which are problems solvable in polynomial time, and NP-complete problems which are NP problems that can be transformed to any other known NP-complete problem. NP-hard problems are NP ones that can not be verified in polynomial time [8]. **Figure 2.4** shows how all P and NP types of problems are related.

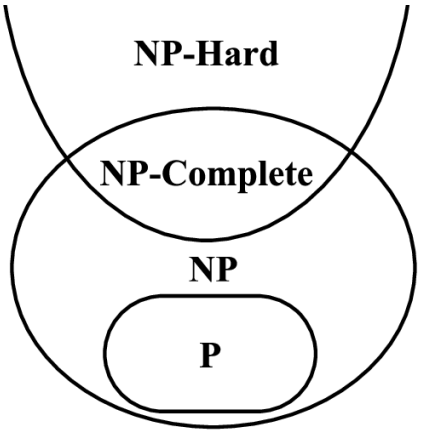


Figure 2.4: P NP Problems.

2.2 Important Algorithms and Concepts

2.2.1 Heuristic Algorithms

Heuristic algorithms are used as optimization techniques to solve problems with high run times, typically NP-complete problems. These algorithm estimate a solution that is not always necessarily optimal, by trading solution accuracy and precision with speed [9].

Heuristic algorithms can be classified as follows, [3]

- **Deterministic:** produces the same solution on every run without any randomness.
- **Stochastic:** has some random variables that can affect the solution on different runs.

Heuristic algorithms can also build the solution in an iterative or a constructive way. Firstly, an iterative solution can be built by looping over an initial solution and improving it until a stopping criterion is reached. Contrarily, a constructive solution is achieved by adding to an initial partial solution [3].

Additionally, heuristics can use exploration and/or exploitation steps. [9]

- **Exploration:** has the goal of widening the search space and looking for solutions in new areas. This improves the solution quality by looking at more globally optimal solutions and avoiding locally optimal ones.

- **Exploitation:** refers to the step that tries to improve the solution by looking at neighboring solutions that maximize the optimization function.

2.2.2 Fiduccia-Mattheyses Algorithm

The Fiduccia-Mattheyses (FM) algorithm, is a partitioning algorithm used to partition operations into two blocks. FM is an iterative heuristic algorithm that starts with a random solution and proceeds with several steps as follows, [10]

1. Starts with a random partition, and attempts to move the nodes one by one from their initial partition to the other one. At this step all nodes are free to move.

2. Chooses the node with highest gain (**equation 5.1**) to be moved to the new partition, by monitoring the solution cost that each move produces.

3. Checks a balance criterion (**equation 2.5**) for the total area of the partitioned blocks. This is done to avoid clustering all nodes to the same block.

4. If the chosen move does not generate balanced partitions, the move is disregarded and a new partition is made.

5. Whenever a move is done, the node moved is fixed, which means it is not allowed to be moved anymore.

6. All above steps are repeated over all unfixed nodes, while keeping track of the partition with highest total gain.

7. Terminates when either all nodes are fixed, or the total gain (**equation 5.2**) becomes less than or equal to zero.

$$\Delta g(c) = FS(c) - TE(c) \tag{2.2}$$

$$G_m = \sum_{i=1}^{m} \Delta g_i \tag{2.3}$$

$$r = \frac{area(A)}{area(A) + area(B)} \tag{2.4}$$

$$[r.area(V) - area_{max}(V)] \leq area(A) \leq [r.area(V) + area_{max}(V)] \tag{2.5}$$

Where,

$g(c)$ is the gain of cell c.

$FS(c)$ is the *moving force* that represents the number of nets connected to c but not connected to any other cell within c's partition.

$TE(c)$ is the *retention force* representing the number of uncut nets connected to c.

G_m is the *maximum positive gain* that represents the cumulative cell gain of m moves with minimum cut cost.

r is the *ratio factor* that represents the relative balance between two partitions with respect to the cell area.

$area(A)$ and $area(B)$ are the total areas of partitions A and B respectively.

A and B are partitions of V into two.

and $area_{max}(V)$ is the maximum cell area.

2.2.3 Clique Partitioning

Let $G = (V, E)$ be a graph of V and E edges. A clique is a sub-graph $S = (V', E')$ of G such that V' and E' are subsets of V and E respectively [11].

The Clique partitioning problem refers to partitioning a graph into the least number of cliques such that every vertex in the graph is represented in one clique exactly [11]. **Figure 5.1** shows different ways to partition a graph into cliques.

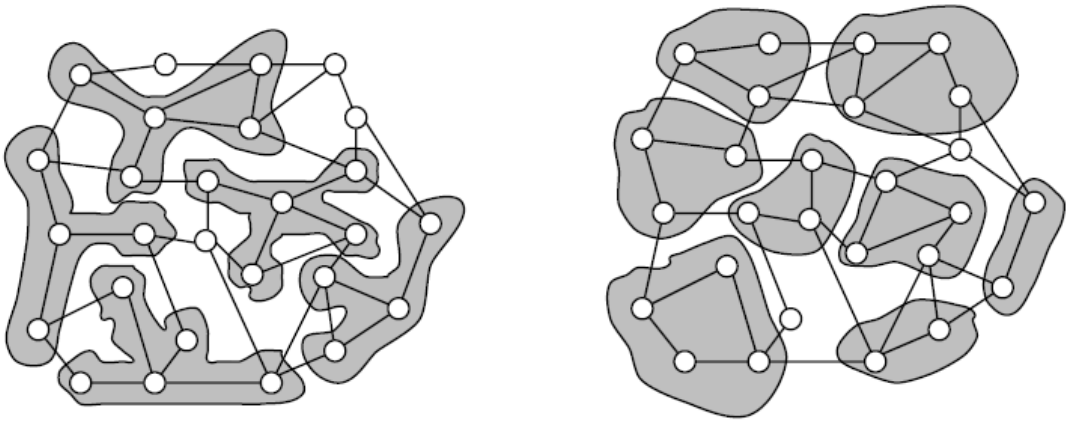


Figure 2.5: Graph partitioned into different cliques. [11]

2.2.4 The Left Edge Algorithm

The left edge algorithm is a well known algorithm, used for solving the allocation problem. The algorithm takes as inputs a list of operations or variables along with their corresponding lifetime, and attempts to allocate them to functional or storage units. This is done by first sorting the operations according to their lifetimes then making several passes through the list and allocating the variables and operations in such a way that the ones allocated to the same FU or SU don't have overlapping lifetimes. This step is repeated until all variables are allocated.

This algorithm is a polynomial time algorithm and it finds the minimum number of registers and operations. This algorithm however, allocates operations and variables separately and cannot take into account the impact of the allocation on connections.

CHAPTER 3

ALLOCATION

3.1 Definition

Datapath allocation, along with scheduling and partitioning, form the three major steps in high-level synthesis (see section 1.3). Datapath allocation focuses on operations, variables and data transfers (see definitions 2.1.12, 2.1.13 and 2.1.14), by allocating them to different resources such as functional, storage and interconnection units (see definitions 2.1.15, 2.1.16 and 2.1.17). This is performed in a way that minimizes costs in terms of the number of components (see definition 2.1.11) used. This is where share-ability comes in handy, as it allows compatible operations to share resources while still respecting the scheduling and partitioning constraints set, i.e., while still respecting time conflicts and area constraints [12]. **Figure 3.1** shows the reduction in the number of components used in a circuit after applying different allocation techniques. part a of the figure represents the scheduled operations which indicates a time-constrained input. On the other hand, parts b, c and d represent the

circuit after different allocations. A huge reduction in the number of hardware used can be seen in the figure leading to a great decrease in the cost of manufacturing.

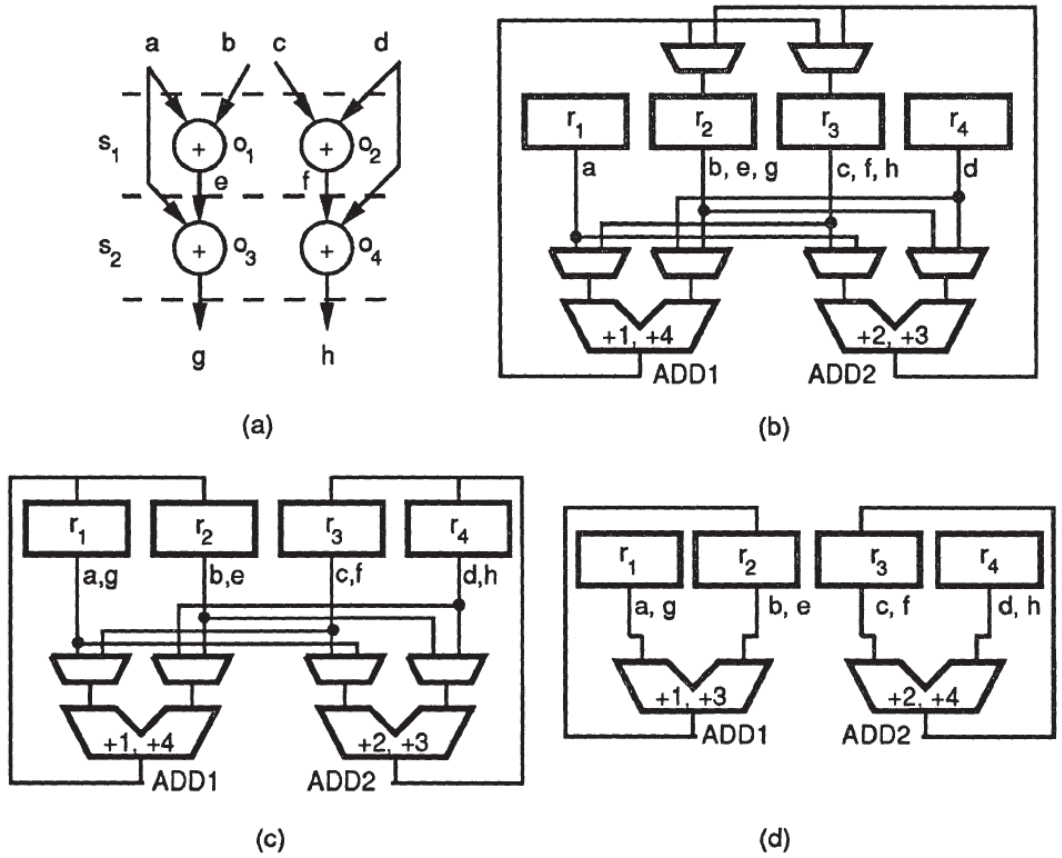


Figure 3.1: Example showing the reduction of the number of components in a circuit after different types of allocations (parts b, c and d). part a) shows a schedule as the initial input of the process. [4]

3.2 Allocation Tasks

Allocation consists of two main tasks, unit selection and unit assignment or binding. In this section, we will go into the details of these tasks and discuss the interdependence between the three unit types [4].

3.2.1 Unit Selection

Mainly, unit selection specifies the numbers and types of hardware resources available. This is done using a component library that can be user defined or specified automatically. This library contains different hardware characteristics, such as size or power, different operation implementations, and several categories of functional units. For example, and addition operation can be performed using an adder, an adder/subtracter or an ALU. [4].

3.2.2 Unit Binding

Unit binding also known as unit assignment divides the operations, variables and data transfers into different functional units, registers and interconnection units in a way that reduces the number of resources used. One basic rule applies for the allocation of all three types of units. Typically, any two operations or variables or data transfers can be assigned to the same hardware component if and only if they do not occur concurrently or at the same time step [4]. As an example, variables a and b in **figure 3.1** have the same lifetime (see definition 2.1.18) and thus can not be bound to the same register; whereas variables a and g take place in different time steps and where accordingly assigned to register r_1. In addition, the operation type plays a role in binding since two operations that perform different tasks can not be assigned to the same hardware, if the hardware component does not accommodate the task being performed or is not specified in the component library created on unit selection task [4].

All these sub-tasks in binding, consisting of the allocation of all three types of units (function, storage and interconnection), are closely dependent on each other and can be done consecutively or in parallel. However, one should keep in mind that performing the allocation of one unit type before another can greatly affect the decisions in the allocation of subsequent unit types. This can significantly affect the quality and performance of the task [4].

3.3 Allocation Approaches

Many methods and algorithms were developed and made state of the art (see chapter 4) solutions for the datapath allocation problem. However, in this section we follow three main approaches widely adopted for this problem.

3.3.1 Greedy Constructive Approaches

A greedy constructive algorithm constructs a solution gradually from an empty one, and starts allocating operations one by one starting with ones that minimize the cost of the datapath the most. For each operation, the algorithm tries to find an existing functional unit that can perform the operation and to which no conflicting operations have been allocated i.e. operations that exist in the same control step of the operation being allocated. A new functional unit is added to the datapath if no existing FU fits the criteria. Similarly, variables and data transfers are assigned to existing registers and interconnection units respectively, if no allocated variables have overlapping intervals. One key factor for evaluating the datapath in this approach

is a cost function, this function can be calculated by looking at the cost of adding a hardware component to the datapath, for example the cost of adding and adder might be less than that of adding an ALU [4].

We can also differentiate between two ordering techniques according to which the operations are allocated in a greedy constructive approach. The first ordering technique, known as static, is one where all operations are ordered before starting any datapath construction and is never changed throughout it. Another technique is the dynamic one, in which no ordering is done but instead, every unallocated operation is evaluated and the one yielding minimum cost is selected. In the dynamic approach, unallocated operations are re-evaluated after every step.

Although greedy constructive algorithms can solve all sub-tasks of the allocation problem simultaneous and are also widely used for their simple implementation and good time complexity, their solution quality is far from optimal due to their greedy nature.

3.3.2 Decomposition Approaches

In contrast to greedy constructive algorithms, in the decomposition algorithms the allocation problem is divided into a series of unrelated tasks, each solved separately. Since all these allocation sub-tasks are connected, decomposition can be reflected in the quality of the solution, since solving one task before another can affect the optimization of the following components. One example of a decomposition approach is clique partitioning (see section 2.2.3) [4].

In order to apply clique partitioning algorithm to the allocation problem, we need to be able to transform our input to a graph. To emphasize, Any two operations, variables or data transfers can be allocated to same unit only on the premises that they satisfy the conditions mentioned earlier. As such, we can transform the allocation problem to a clique partitioning problem by converting the RT level components to nodes in a graph, and an edge can be added between any two nodes that are compatible together [4].

3.3.3 Iterative Refinement Approaches

Additionally to greedy constructive approaches and decomposition approaches, iterative refinement is also widely used to further optimize a synthesized datapath. This method is used after applying one of the previously mentioned approaches, in order to improve and refine the results. This is done by performing some modifications to the datapath and evaluating the cost of the new solution. An example of the modifications that can be applied is swapping operation or variable assignment and re-evaluating the solution [4].

Some issues related to iterative refinement may arise, and these are mostly related to the type of modification applied and to the determination of the process termination criteria.For instance, performing one refinement method and not the other may not allow for exploration of the entire solution space and has a risk of being stuck at a local minimum [4].

CHAPTER 4

RELATED WORK

4.1 Rule-Based Systems

Initial work on the allocation problem created mathematical and algebraic models as was done by Louis Hafer and Alice Parker [13]. Parker and Hafer expressed the behavioral description of the model at the register-transfer (RT) level as a system of algebraic relations. This was solved as a mixed-integer linear programming (MILP) problem, and results were verified by creating a synthesis model at the RT level of the system. Analysis of the solution showed that this method is task specific and needs to be adjusted for different tasks, additionally to being time expensive, as very large solution times were reported for small data. In [14] and [15] a Design Automation Assistant (DAA) system is implemented, by using a set of rules gathered from expert designers. The DAA system explored the allocation problem on operations, variables and data transfers.The system showed satisfactory results, but is input specific and can not be globally applied to other design descriptions as is the case with many

other rule-based techniques. Similar results were presented in [16] and [17], that implement rule-based systems for automatic synthesis of data-path.

4.2 Global Algorithmic Approaches

In [18], Chia-Jeng Tseng and Daniel P. Siewiorek presented an algorithm called 'Facet' that optimizes unit, storage, and interconnection allocation problems separately. This work solves the clique partitioning problem by grouping nodes with the maximum number of common neighbours to find the minimum number of cliques in a graph. Tseng and Siewiorek continue their work in [19], as they develop a design generator called 'Emerald' that produces functional level structures from a behavioral description. The designs generated from this approach for a mainframe computer, the IBM System/370 was restricted to the ISPS description, and showed great similarities to designs created by human experts.

4.3 Heuristic and Meta-Heuristic Approaches

Several papers tried to solve the allocation problem using heuristics and meta-heuristics approaches. In their paper, "Heuristic Search Based Approach To Scheduling, Allocation And Binding In Data Path Synthesis"[20] Kumar, A., Kumar, A., & Balakrishnan, M. used an approach called VITAL. This approach has been developed in a way that unifies various sub-tasks of the Datapath allocation or scheduling. And it's a search-based approach that supports several design styles, design constraints and component types. It uses heuristics search with three variations which

are VITAL-NS (no search), VITAL-SS (selective search) and VITAL-EX (extended search). And it also uses, three generic heuristic functions which estimate cost for the variety of situations in Datapath allocation. The experiments were tested on several benchmarks and yielded good results. Another approach used is of a GA-based technique[21], that addresses some of the deficiencies of earlier GA-based approaches to high-level synthesis. The algorithm performs the combined subtasks of scheduling and allocation in high-level behavioural synthesis and concurrently searches the space of data path schedules and module/storage allocations. In addition, the inherent parallelism of GAs allows efficient design space exploration during a single optimization iteration. The algorithm incorporates three features that enables it to efficiently perform design space exploration. Which are, a multi-chromosome encoding to concurrently handle a search in the space both the schedules and allocations. Additionally, an efficient list-scheduling heuristic was used to decode chromosomes into valid schedules, and concurrently perform register minimization in the Datapath, in addition to functional allocation using a variant of the left-edge algorithm. A Moth flame optimization algorithm was also used in[22]. This approach performs scheduling, allocation, and binding steps simultaneously. And it uses three generic heuristic functions to estimate the cost depending on different situations. This paper showed a good improvement in the delay rate compared to the GA algorithm and it optimized the occupied area with an improvement of 6.58%. Additionally to an improvement on power consumption of 6.48%.

4.4 Other Approaches

Additionally, several other approaches were made on this problem, some important ones include a pipelined synthesis program by Nohbyung Park and Alice Parker [23]. This approach represents several tasks divided into smaller sub-tasks and executed during a clock cycle. This is done using instructions on computational tasks and any conflicting tasks or dependencies are delayed until the execution of previous ones.This research showed that pipeline synthesis is crucial to generate good designs and can be repeated and modified due its fast runtime. Another important research is one on a program called MAHA by Parker, A.C. and Pizarro, J. and Mlinar, M. [24], which implements an algorithm that assigns critical nodes in a path using linear hardware assignment and evaluates node freedom for cost based assignment. This algorithm solves some weaknesses found in other approaches such as flexibility and ability to adapt based on several constraints such as speed or cost.

CHAPTER 5

PROPOSED APPROACH

The proposed approach to solve the allocation problem in VLSI is inspired by the Fiduccia-Mattheyses (FM) algorithm. The algorithm starts from a schedule as input and an empty solution space and attempts to allocate operations, variables and connections concurrently by looping over the nodes in the schedule and clustering compatible nodes in hardware units. This approach attempts to join compatible nodes together by assigning them to the same resource. **Figure 5.1** shows how several nodes are joined together to form one super-node, or a cluster of nodes which can also be recognized as one connected sub-graph. Each clustered node in this case represents a unit that contains allocated operations.

At each loop the algorithm attempts to add a node to existing compatible resources in the solution space and evaluates the maximum gain g generated from this operation, along with the maximum cumulative gain G_m of the solution space. If no such resource exists a new resource is added to the solution space and evaluated e.i. a new unit is added.

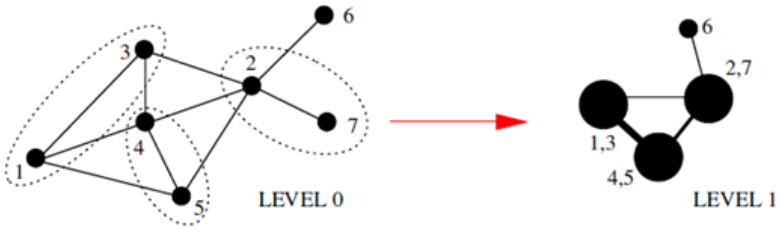


Figure 5.1: Clustering

The gains of the graphs are calculated as follows.

$$g = 1/(L_{FU} + L_{SU} + L_{IU}) \tag{5.1}$$

$$G_m = \sum_{i=1}^{m} \Delta g \tag{5.2}$$

Where,

L_{FU} is the losses from adding a functional unit represented in **equation 5.3**.

$$L_{FU} = Num_{FU} \times Cost_{FU} \tag{5.3}$$

L_{SU} is the losses from adding a storage units represented in **equation 5.4**.

$$L_{SU} = Num_{SU} \times Cost_{SU} \tag{5.4}$$

and L_{IU} is the losses from adding an interconnection unit represented in **equation 5.5**.

$$L_{IU} = Num_{IU} \times Cost_{IU} \tag{5.5}$$

All resources' costs are assumed to be equal with a value of 1. To understand this further we will look at an example represented in **figure 5.2**

We start with a schedule and an empty solution space. Let $S = \{\}$ be our initial solution space. So a schedule contains several operations, which are represented here by the numbers $[1 - 6]$. These operations are divided into different timesteps represented by the dotted lines. The inputs and outputs of these operations are represented by the variables $[a - m]$. And these variables are stored in registers and connected to the functional units through connections or wires . We can say that two nodes are compatible or can be assigned together in the same resource if they do not occur concurrently or if they do not occur in the same timestep.

We start our algorithm with a random node, in this case we choose node 4. Since the solution space is empty, we will have to add a new functional unit to which node 4 will be assigned. Additionally, variables g, h and k will need to be assigned to their corresponding units as well. In this case since g and h are not compatible, they can't be assigned to the same storage unit, but we can assign k with either g or h since no time conflicts exist between them. This will lead to two storage units in the solution space.

So now that we added the functional unit and the storage units, we still need to add the interconnection units, this is basically needed to connect the new registers to the functional unit, so in this case we need a connection between F_1, R_1 and F_1, R_2.

The gain of this assignment is then calculated to $\frac{1}{5}$ according to the equation shown above.

This first iteration led to the assignment of node 4 and variables g, h and k with one functional unit , two storage units and two interconnection units in the solution space.

In the next iteration, we will loop over each unassigned node and check the gain that adding this node to the solution space will generate. Adding nodes 1 and 2 in the solution space does not require new resources since all the inputs are compatible with the existing units. This leads to a gain of $\frac{1}{5}$ for each. Node 3 on the other hand, along with its inputs e and f are compatible with existing resources, but has conflicting outputs i and j. This results in adding two registers and two connections which gives a gain of$\frac{1}{8}$. Similarly, nodes 5 and 6 need additional resources resulting in a gain of $\frac{1}{10}$ and $\frac{1}{7}$ respectively.

In this iteration the highest gain we got was from the assignments of either nodes 1 or 2. We choose node 1, fix it and continue to the next iteration.

So we now have two nodes and five variables fixed. And our solution space still contains one functional unit, two storage units and two interconnection units.

In the next iterations, We will go similarly over all unfixed nodes. In the example shown above, node 6 generates the highest gain in iteration 2, so we fix it and move to the next iteration. We now have three assigned nodes and seven assigned variables. This is done by adding one register and one connection to the solution space.

The algorithm keeps looping over all unassigned nodes and terminates when all nodes

and variables are fixed.

Figure 5.3 shows the pseudocode of the proposed approach and its runtime. The algorithm has a complexity of $O(N^2)$ which indicates a polynomial time algorithm. This proves a competitive runtime for solving the allocation problem.

Figure 5.2: Example of the proposed approach

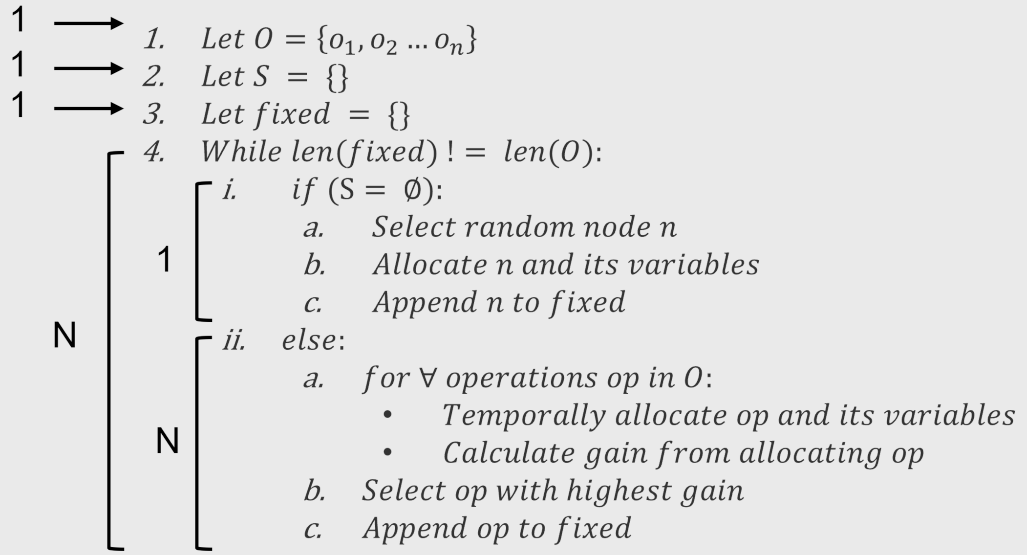


Figure 5.3: Algorithm pseudo-code

CHAPTER 6

RESULTS AND ANALYSIS

6.1 Benchmarks

Experiments were made on eleven different High Level Synthesis benchmarks [25], with different number of inputs in each benchmark. Each line in the data available represents a connection between two operations. The name of an operation is separated from another by an arrow, that represents data transfer between the two operations. The example below shows how the data is represented. We can see four different operations 1, 2 3 and 31, where 1 and 2 are predecessors to 3 and the later is connected to 31. These relationships between the nodes are indicated by an arrow.

$$1 -> 3;$$

$$2 -> 3;$$

$$3 -> 31;$$

Additionally, the type of each operation (MUL, ADD, etc...) is indicated separately. These benchmarks are used to validate the results of our proposed algorithm, and findings are compared with other algorithms to determine the feasibility and quality of our solution.

6.2 Experimental Setup

The algorithm is implemented on a Jupyter notebook, using python version 3.10.4. We ran the experiments on a Windows 11 machine with a 64-bit operating system, x64-based processor, provided with an Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz and a 6 cores CPU, additionally to a 16.0 GB RAM and 2TB storage.

6.3 Experimental Results

The Benchmarks we used to validate our algorithm are used in several papers on the allocation problem. Additionally, we use a configuration file to determine compatible node types in the algorithm. This step represents the data selection step in allocation. Our configuration file is a .txt file that represents all compatible types of operations in one line. For example if a MUL (multiplier) operation type and a ADD (adder) type are on different lines in the configuration file, this indicates that these two types are not compatible and have to be assigned to different resources.

Results reported in several state of the art approaches differ between one another. There is no standard way these results are compared and analyzed. For instance, some

papers report their findings in terms of area, power and delay; while others report the number of resources used with different timesteps. In our case, the timesteps calculated are fixed, and resources are allocated accordingly. These differences in reported results and implementations used made it difficult for us to test the efficiency of the algorithm compared to existing approaches. One way to normalize all these factors that might affect the solution was to implement the left edge algorithm (see Section 2.2.4), a well known approach to solving the allocation problem, and compare our results to that generated by it. The left edge algorithm can't solve all unit types concurrently, so in this case it was applied on operations and variables and results were tabulated and analyzed.

Most tables in the reported results show similar results for both algorithms. The left edge algorithm however, proved to have much faster runtime. We also generated better results in **Table 6.4** where we were able to allocate 44 operations in 34 functional units in comparison to the Left Edge Algorithm that allocated them to 44 functional units. However, the Left Edge Algorithm performed better on **Table 6.11**, as it was able to allocate 106 operations in 56 functional units rather than 62 in our algorithm. These results suggest that the proposed approach performs really well on most instances as we can see a great reduction in the allocation, for example, **Table 6.10** shows up to 67% reduction between existing operations and allocated functional units. Keeping in mind that the proposed approach solves all three subtasks of allocation concurrently while the left edge algorithm solves them separately. This can be a great advantage in this approach and additional refinement can drive the solution to a more optimal state.

Table 6.1: Comparison of our approach with the left edge algorithm on ARF DFG

	# Operations	# Variables	# FUs	# SUs	# IUs	time in s
Our approach	28	48	8	16	12	0.646
Left edge Algorithm			8	16	-	0.007

Table 6.2: Comparison of our approach with the left edge algorithm on COSINE2 DFG

	# Operations	# Variables	# FUs	# SUs	# IUs	time in s
Our approach	82	161	50	62	62	18.455
Left edge Algorithm			54	62	-	0.139

Table 6.3: Comparison of our approach with the left edge algorithm on EWF DFG

	# Operations	# Variables	# FUs	# SUs	# IUs	time in s
Our approach	34	64	4	12	9	1.250
Left edge Algorithm			4	12	-	0.007

Table 6.4: Comparison of our approach with the left edge algorithm on FIR DFG

	# Operations	# Variables	# FUs	# SUs	# IUs	time in s
Our approach	44	88	34	44	44	2.865
Left edge Algorithm			44	44	-	0.074

Table 6.5: Comparison of our approach with the left edge algorithm on H2V2 DFG

	# Operations	# Variables	# FUs	# SUs	# IUs	time in s
Our approach	51	85	18	32	31	3.906
Left edge Algorithm			18	32	-	0.022

Table 6.6: Comparison of our approach with the left edge algorithm on HAL DFG

	# Operations	# Variables	# FUs	# SUs	# IUs	time in s
Our approach	11	19	6	10	8	0.051
Left edge Algorithm			6	10	-	0.003

Table 6.7: Comparison of our approach with the left edge algorithm on IDCTCOL DFG

	# Operations	# Variables	# FUs	# SUs	# IUs	time in s
Our approach	114	208	18	58	89	36.916
Left edge Algorithm			18	58	-	0.063

Table 6.8: Comparison of our approach with the left edge algorithm on MATMUL DFG

	# Operations	# Variables	# FUs	# SUs	# IUs	time in s
Our approach	109	168	25	48	47	36.194
Left edge Algorithm			25	48	-	0.096

Table 6.9: Comparison of our approach with the left edge algorithm on MOTION DFG

	# Operations	# Variables	# FUs	# SUs	# IUs	time in s
Our approach	32	60	14	28	27	1.082
Left edge Algorithm			14	28	-	0.012

Table 6.10: Comparison of our approach with the left edge algorithm on SMOOTH DFG

	# Operations	# Variables	# FUs	# SUs	# IUs	time in s
Our approach	197	332	65	128	125	224.261
Left edge Algorithm			65	128	-	0.374

Table 6.11: Comparison of our approach with the left edge algorithm on WRITE DFG

	# Operations	# Variables	# FUs	# SUs	# IUs	time in s
Our approach	106	182	62	76	74	36.729
Left edge Algorithm			56	76	-	0.158

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

In conclusion, the automation of VLSI design proved to be crucial in the advancement of chip design due to the complexity of all the design steps involved and their large scaling. The allocation problem in VLSI design, plays a huge role in these steps, and solving it in an effective way is fundamental in having high quality, and efficient chips. In this research, we presented a new graph heuristic approach for the data path allocation problem. We stared by introducing major terms in this field and went deeper into allocation with its subtasks and existing approaches. We then presented some existing solutions in the literature and continued with our proposed approach.The approach, inspired by the Fiduccia-Mattheyses algorithm, attempts to solve the subtasks of the allocation problem concurrently, by assigning, operations,

variables and connections to hardware resources and evaluating the solution using a gain function. At each step of the algorithm all operations are assigned to compatible hardware resources along with their variables and connections. And the operation generating the highest gain is fixed and can not move again. This process is repeated until all operations, variables and connections are fixed. Our algorithm was tested on existing benchmarks and compared to the classic left edge algorithm. The proposed approach generated competitive results and proved to be efficient in terms of time complexity and solution quality.

7.2 Future Work

Many improvements and optimizations can be made to further improve this approach or refine the solution. As mentioned earlier, in this approach the cost of adding any hardware resource is assumed to be equal to 1. We suggest investigating how changing this cost, and using unequal cost of resources can affect the solution. For example the cost of adding a functional unit can be double of that of a storage unit. Additionally, In the current solution, the algorithm starts from a random operation. We can look at how optimizing the starting point of the algorithm can affect the end solution. This can also be explored by changing the order of assigning our resources. For instance, we can start by allocating variables instead of operations and look into how performing one allocation before the other can affect the results.

Bibliography

[1] A. Kahng, J. Lienig, I. Markov, and J. Hu, *VLSI Physical Design: From Graph Partitioning to Timing Closure*. 01 2011.

[2] A. N. Saxena, *Invention Of Integrated Circuits: Untold Important Facts (International Series on Advances in Solid State Electronics)*. World Scientific Publishing Company, 2009.

[3] . M. G. E. Brock, D. C., *Understanding Moore's Law : Four Decades of Innovation*. 2006.

[4] A. C.-H. W. S. Y.-L. L. Daniel Gajski, Nikil Dutt, *High — Level Synthesis: Introduction to Chip and System Design*. 02 1992.

[5] K.-T. T. C. Laung-Terng Wang, Yao-Wen Chang, *Electronic Design Automation: Synthesis, Verification, and Test (Systems on Silicon)*. Morgan Kaufmann, 2009.

[6] C.-J. Tseng and D. Siewiorek, "Automated synthesis of data paths in digital systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 5, pp. 379 – 395, 08 1986.

[7] K. W. Robert Sedgewick, *Algorithms, 4th Edition: Essential Information about Algorithms and Data Structures*. Addison-Wesley, 2011.

[8] O. Goldreich, *P, NP, and NP-Completeness: The Basics of Computational Complexity*. Cambridge University Press, 1 ed., 2010.

[9] F. R. (auth.), *Design of Modern Heuristics: Principles and Application*. Natural Computing Series, Springer, 1 ed., 2011.

[10] A. Caldwell, A. Kahng, and I. Markov, "Design and implementation of the fiduccia-mattheyses heuristic for vlsi netlist partitioning," vol. 1619, 02 1999.

[11] J. Bhasker and T. Samad, "The clique-partitioning problem," *Computers Mathematics With Applications - COMPUT MATH APPL*, vol. 22, pp. 1–11, 12 1991.

[12] Z. Baruch, "Datapath allocation," 07 2008.

[13] L. Hafer and A. Parker, "A formal method for the specification, analysis, and design of register-transfer level digital logic.," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 2, pp. 4–18, 01 1983.

[14] T. Kowalski and D. Thomas, "The vlsi design automation assistant: Prototype system.," pp. 479–483, 01 1983.

[15] T. Kowalski and D. Thomas, "The vlsi design automation assistant: what's in a knowledge base," pp. 252–258, 01 1985.

[16] J. Rajan and D. Thomas, "Synthesis by delayed binding of decisions," pp. 367–373, 01 1985.

[17] C. III and D. Thomas, "A method of automatic data path synthesis," pp. 484–489, 07 1983.

[18] C.-J. Tseng and D. Siewiorek, "Facet: A procedure for the automated synthesis of digital systems," pp. 490 – 496, 07 1983.

[19] C.-J. Tseng and D. Siewiorek, "Emerald: A bus style designer," pp. 315– 321, 07 1984.

[20] A. Kumar, A. Kumar, and M. Balakrishnan, "Heuristic search based approach to scheduling, allocation and binding in data path synthesis.," pp. 75–80, 01 1995.

[21] V. Krishnan and S. Katkoori, "A genetic algorithm for the design space exploration of datapaths during high-level synthesis," *Evolutionary Computation, IEEE Transactions on*, vol. 10, pp. 213 – 229, 07 2006.

[22] M. Esmaeili, S. Zahiri, and S. Razavi, "A novel method for high-level synthesis of datapaths in digital filters using a moth-flame optimization algorithm," *Evolutionary Intelligence*, vol. 13, pp. 1–16, 09 2020.

[23] N. Park and A. Parker, "Sehwa: A program for synthesis of pipelines," in *23rd ACM/IEEE Design Automation Conference*, pp. 454–460, 1986.

[24] A. Parker, J. Pizarro, and M. Mlinar, "Maha: A program for datapath synthesis," in *23rd ACM/IEEE Design Automation Conference*, pp. 461–466, 1986.

[25] "Express Benchmark Suit. University of California, Santa Barbara." `https://web.ece.ucsb.edu/EXPRESS/benchmark/`.

[26] E. Girczyc, "Automatic generation of microsequenced data paths to realize ada circuit descriptions," 05 2022.