LEBANESE AMERICAN UNIVERSITY

Layout Driven Binding In High Level Synthesis Using Reinforcement Learning

By

Marie-Claire Melhem

A thesis

Submitted in partial fulfillment of the requirements

for the degree of Master of Science in Computer Science

School of Arts and Science

July 2021

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

THESIS APPROVAL FORM

Student Name: Marie-Claire Melhem I.D. #: 201604959

Thesis Title: Layout Driven Approach for High Level Synthesis Using Reinforcement Learning

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: 23 / 07 / 2021
Day Month Year

Committee Member's Name: Danielle Azar

Signature: [redacted] Date: 23 / 07 / 2021
Day Month Year

Committee Member's Name: Chadi Nour

Signature: [redacted] Date: 23 / 07 / 2021
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: Marie Claire Melhem

Signature: 

Date: 19 / 7 / 2021
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: Marie Claire Melhem

Signature: 

Date: 19 / 7 / 2021
Day Month Year

Acknowledgement

I would like to thank my advisor Dr. Haidar Harmanani, for his guidance and support throughout this Thesis. Special thanks are also due to my committee members, Dr. Danielle Azar and Dr. Chadi Nour for their encouragement and support. I would like to express heartfelt appreciation and gratitude to the Lebanese American University, especially the Graduate Research Office for their financial support during my graduate studies journey.

Finally, thanks to my parents and friends without whom I would not have been able to do it.

Layout Driven Binding In High Level Synthesis Using Reinforcement Learning

Marie-Claire Melhem

The increase in density that the advent of Very Large Scale Integration (VLSI) has made the move to higher levels of design abstraction imperative. High Level Synthesis (HLS) emerged as a viable approach that has been gaining strides in the EDA industry. This work exploits the tight relation that exists between the allocation process and chip layout in an integrated system level design environment. The approach proposes a layout-driven data path allocation method, and explores design tradeoffs among operators binding, registers assignments, and flooplanning shape functions. The approach uses Deep Reinforcement Learning and proposes new methods and tools for the automatic synthesis of data path at the register-transfer level (RTL). A major effort in this research involves the development of a prototype high-level synthesis system that bridges the gap between high level synthesis and layout information. The goal is to build a model capable of learning optimization steps. The approach has been implemented and several designs were implemented.

Keywords: High-Level Synthesis, Floorplanning, Datapath Synthesis, Machine Learning, Reinforcement Learning.

TABLE OF CONTENTS

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Problem Statement

Very Large Scale Integration (VLSI) has made the packing of billion of transistors in a chip possible [2]. VLSI chips are now embedded in mobile phones, graphic cards, smart TVs, washing machines, light switches, and portable computers.

The VLSI design process can be tackled at three different axes: *behavioral*, *structural*, and *physical synthesis*. Within each dimension, the design process can be further refined. For example, logic synthesis is performed after behavioral synthesis. Typically the final stage includes physical design which places the circuit components on a chip while taking into consideration various constraints. The physical design includes various tasks such as *partitioning*, *floorplanning*, *placement*, *routing*, and *timing optimization*. While circuit partitioning divides the different circuit components to be placed, floorplanning arranges soft blocks with unassigned shapes. Placement allocates the blocks in the chip area. After the physical design, a physical verification step is performed in order to make sure that the physical layout performs the desired behavior and meets the given constraints. Finally, the actual chip is fabricated and tested [13].

VLSI design stages are complex and cannot be solved manually by engineers, hence, electronic design automation (EDA) tools are employed to facilitate the design process and reduce the time-to-market cycle. Additionally, some of these design steps are problems for which a non-deterministic polynomial time solution is known. For instance, area and wire length minimization of the chip layout requires the usage of fast optimization algorithms to approximate the solution [17]. Moreover, machine learning algorithms are being employed in EDA in various VLSI chip design steps. This automation can generalize well over different problem instances in a faster and efficient manner especially with

current computational power [11].

One of the essential steps in EDA is high-level synthesis (HLS). It transforms the given abstract behavioral description into register transfer level (RTL) through scheduling, allocation, and binding. The system behavior is represented using a hardware description language (HDL) then converted to circuit representation known as register transfer level (RTL).

Usually, computer-aided (CAD) tools separate high-level synthesis from physical design steps, since the former precedes the latter. The layout information in the high-level synthesis process is very limited at this stage. Since one design affects the other drastically, researchers directed their work to the incorporation of physical design steps into high-level synthesis. This integration allows the detection of many issues which affect the performance and efficiency of the design, such as congestion, wire delays and power consumption at an earlier stage of the design cycle. Conventionally, the floorplanning process, which is the first step of the physical synthesis, is widely integrated into HLS. Floorplanning on the other side, focuses on placing circuit components into the chip space while minimizing different constraints such as area or wire-length. Researchers focused on developing physically aware HLS methods to address the problem of area [32], power [4] [39], routing congestion and wiring delays [33] [34] [40] or multiple constraints simultaneously[25].

1.2 Thesis Objective and Contribution

This thesis aims to bridge the gap between high-level synthesis and layout information by considering the various trade-offs that might result due to different optimization constraints. The goal is to propose a model capable of learning optimization steps to obtain an efficient design.

To tackle the above mentioned objective, binding and allocation from the high-level synthesis process are combined with floorplanning of the physical level synthesis to obtain an optimized chip design.

The adopted approach is Machine Learning, precisely Deep Reinforcement Learning. This solution approach was not previously proposed in the domain of simultaneous high level synthesis and layout optimization. This works offers a complete automation process of high-level synthesis and layout information in VLSI chip design.

1.3 Thesis Organization

In Chapter Two, a thorough explication of High Level Synthesis and Floor-planning is discussed. In Chapter Three, Reinforcement Learning is introduced. Chapter 4 presents the adopted approach and the obtained results. Finally, we conclude in Chapter Five the thesis contribution and discuss future work.

Chapter 2

Background and Related Work

This chapter discusses relevant concepts that are discussed in the literature. These notions include high-level synthesis *binding* and *allocation*, Floorplanning and Reinforcement Learning. Section 2.1 describes High-Level Synthesis including scheduling, binding, and data path allocation. Section 2.2 introduce floorplanning, its representations and algorithms. Finally, section 2.3 illustrates Reinforcement Learning concepts, and describes Deep Reinforcement Learning as well as its algorithms.

2.1 High-Level Synthesis

High level Synthesis is an automated process converting a behavioral description into a structural description of a circuit. The behavior description is written in a hardware level language (typically Verilog or SystemC), and is transformed using High-Level Synthesis to register-transfer-level (RTL) description that implements the given behavior. In specific, a datapath of functional and storage units with their interconnections, and a control unit [3]. The datapath includes functional units which perform different operations (multipliers, adders, arithmetic logic units (ALU), logic gates), storage units which hold data (registers, register files) and components which transfer the data between storage and functional units (buses, multiplexers). High level synthesis includes three main tasks:

- Scheduling: assigns operations in the design to clock cycles while respecting the precedence constraint of the operations
- Allocation: determines the number of functional units to be used in the design, as well as storage units and their interconnections
- Binding: maps operations and variables to their corresponding functional units and storage units

Representation

High Level Synthesis models the behavioral description of the system through an intermediate representation known as a control/data flow graph (CDFG). A CDFG is an acyclic directed Graph $G = (V, E)$, where $v \in V$ is a vertex or node representing an operation in the behavior and $e \in E$ represents a precedence or successor relation between these operations or vertices. Throughout this chapter, we will use the differential equation behavior that is shown in Figure 2.1 in order to illustrate the high-level synthesis process.

The differential equation benchmark solves the following second order differential equation:

$$y'' + 3xy^3 + 3y = 0 \tag{2.1}$$

The above equation can be modeled using the behavior shown in Figure 2.1. The iteration of the equation 2.1 is shown in the following algorithm.

Algorithm 1: Iteration of the Differential Equation

```
Input: a
while x < a do
    xl = x +dx;
    ul = u - (3*x*u*dx) - (3*y*dx);
    yl = y + u +dx;
    x = xl;
    y = yl;
    u = ul;
end
```

Figure 2.1 shows the CDFG representing the algorithmic behavior of the differential equation where nodes represent various operations including addition, subtraction, and comparison.

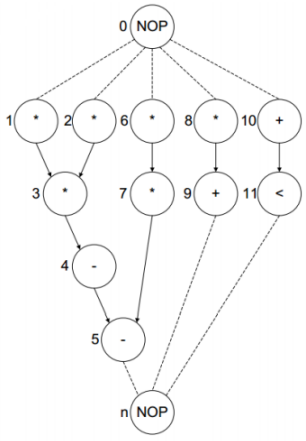

Figure 2.1: CDFG of the Differential Equation Example

2.1.0.1 Scheduling

Scheduling maps each node or operation in the DFG graph to a clock cycle which determines the operation's start execution time while taking into consideration the dependency between these operations. In some cases, latency and resource constraints are introduced. Latency constraints set a maximum number of clock cycles for the operations to be executed at, while resource constraints limit the number of resources of each type in each clock step. The number of needed resources is reflected by the maximum number of resources in each control step of each type: multipliers, adders, subtractors and comparators.

Scheduling Algorithms

Scheduling Algorithms can be be implemented using constrained or unconstrained algorithms. Constrained scheduling algorithms produce schedules according to latency or resource constraints.

A. Unconstrained Scheduling Algorithms: ASAP and ALAP

1. As Soon As Possible (ASAP)

ASAP scheduling starts by scheduling vertices v ∈ V with no predecessors, Pred_{v_i}, and setting the ASAP clock cycle of v to E_i to 1. The remaining vertices are examined to check that their predecessors have been scheduled using all_pred_scheduled(v_i, E). Upon finding a vertex with scheduled predecessors, the maximum ASAP time of its predecessor is calculated using MAX(Pred_{v_i} E).

Algorithm 2: ASAP

```
Input: G (V,E)
for v_i ∈ V do
    if Pred_{v_i} = ∅ then
        E_i = 1;
        V= V - v_i;
    end
    else
        E_i = 0;
    end
end
while V ≠ ∅ do
    for v_i ∈ V do
        if all_pred_scheduled(v_i, E) then
            E_i= MAX(Pred_{v_i} E) + 1;
            V= V - v_i;
        end
    end
end
```

Figure 2.1.0.1 shows the scheduled CDFG of the previous behaviour using the ASAP scheduling algorithm.

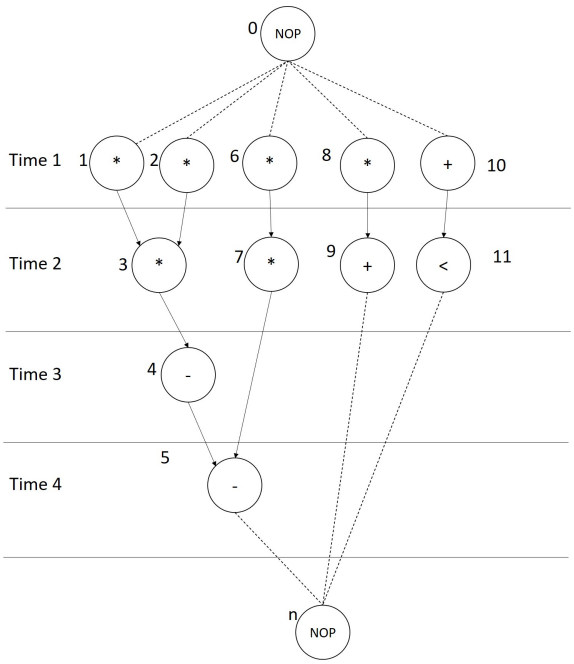

Figure 2.2: ASAP Schedule

2. As Late As Possible (ALAP)

ALAP starts by scheduling operations with no successor into the last possible time step. Remaining operations are scheduled as late as possible. It is important to note that the maximum clock cycles is determined using ASAP scheduling, that is the maximum latency bound T for the ALAP schedule and this solving a latency-constrained problem.

ALAP starts by scheduling vertices v $\in$ V with no successors $\mathrm{Succ}_{\mathrm{v_i}}$ into the last possible clock cycle by setting the ALAP time of v to $\mathrm{L_i}$ to 1. The remaining vertices are examined to check that their successors have been scheduled using all_succ_scheduled($\mathrm{v_i}$, E), once a vertex with scheduled successor is found, the maximum ALAP time of its successors is calculated using MAX($\mathrm{Succ}_{\mathrm{v_i}}$ E).

Figure 2.1.0.1 shows the scheduled CDFG of the previous behaviour using the ALAP scheduling algorithm.

Algorithm 3: ALAP

```
Input: G (V,E), T
for v_i ∈ V do
    if Succ_{v_i} = ∅ then
        L_i = T;
        V= V - v_i;
    end
    else
        L_i = 0;
    end
end
while V ≠ ∅ do
    for v_i ∈ V do
        if all_succ_scheduled(v_i, L) then
            L_i= MAX(Succ_{v_i},L) + 1;
            V= V - v_i;
        end
    end
end
```

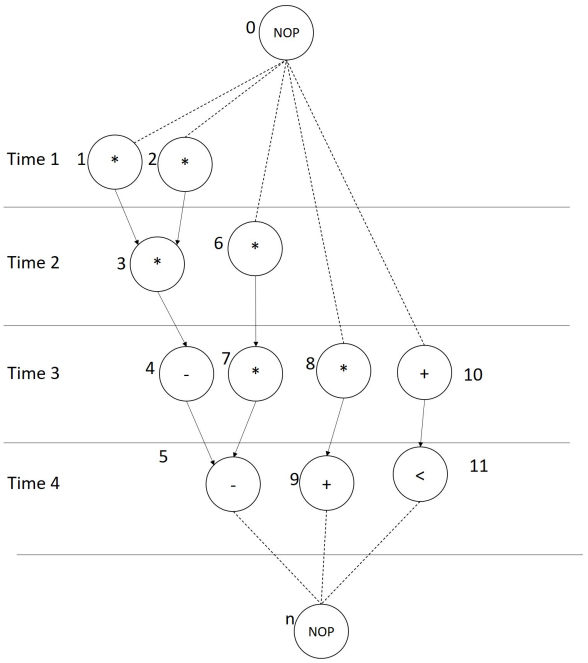

Figure 2.3: ALAP Schedule

Mobility

The mobility of an operation is defined by the difference of ALAP and ASAP schedule. The mobility represents the time frame of possible clock cycles that the operation can be scheduled without violating

the predecessor and successor dependency. The critical path is the longest path on the schedule, operations located on the critical path have 0 mobility, for instance operations 1, 2, 3, 4 and 5. Operations 6 and 7 have a mobility of 1. The rest of the operations have a mobility of 2 as shown in the figures 2.1.0.1 and 2.1.0.1.

B. Constrained Scheduling

1. Force Directed Scheduling

Force directed Scheduling (FDS) is a heuristic scheduling method for minimizing resources according to latency constraints. Suggested by Paulin and Knight [24], the goal of FDS is to balance concurrency of the operations by respecting the latency constraint. FDS has three main steps: determining the time frames or intervals of each operation in the CDFG, calculate the probability and distribution graphs of the operations and calculate the force to schedule each operation.

Definitions:

- **Operation Interval or Time frame**: mobility as determined by the difference of ALAP and ASAP schedule.

- **Operation Probability**: probability of an operation i being scheduled at time-step l:

$$p_{\mathrm{i}}(l) = 1/(mobility + 1) \tag{2.2}$$

  inside the interval of possible time steps as computed previously, and 0 elsewhere.

- **Operation type distribution, $\mathbf{q_k(l)}$**

$$\sum_{m=t'_i}^{i} q_k(m)\,(\delta_{lm} - p_i(m)) \tag{2.3}$$

- **Self force**: operation type distribution * probability of being assigned to a certain time step.

- **Total Force**: self-force + successor-force + predecessor-force

Algorithm 4: Force Directed Scheduling Algorithm

Input : CDFG

Output: Scheduled CDFG

while *not all operations are scheduled* **do**

  Compute time frames

  Compute operations probability

  Compute resource type probability

  Compute self forces and total forces

  Schedule operations with least force and update time frames

end

return *schedule*;

Computing Timeframes and Operation Probability:

- Operation 1: mobility 0, p1(1) = 1, p1(2) = 0, p1(3) = 0, p1(4) = 0
- Operation 2: mobility 0, p2(1) = 1, p2(2) = 0, p2(3) = 0, p2(4) = 0
- Operation 3: mobility 0, p3(1) = 0, p3(2) = 1, p3(3) = 0, p3(4) = 0
- Operation 4: mobility 0, p4(1) = 0, p4(2) = 0, p4(3) = 1, p4(4) = 0
- Operation 5: mobility 0, p5(1) = 0, p5(2) = 0, p5(3) = 0, p5(4) = 1
- Operation 6: mobility 2, Timeframe [1,2], p6(1) = 0.5, p6(2) = 0.5, p6(3) = 0, p6(4) = 0
- Operation 7: mobility 2, Timeframe [2,3], p7(2) = 0.5, p7(3) = 0.5, p7(4) = 0
- Operation 8: mobility 3, Timeframe [1,2,3], p8(1) = 0.3, p8(2) = 0.3, p8(3) = 0.3, p8(4) = 0
- Operation 9: mobility 3, Timeframe [2,3,4], p9(1) = 0, p9(2) = 0.3, p9(3) = 0.3, p9(4) = 0.3
- Operation 10: mobility 3, Timeframe [1,2,3], p10(1) = 0.3, p10(2) = 0.3, p10(3) = 0.3, p10(4) = 0
- Operation 11: mobility 3, Timeframe [2,3,4], p9(1) = 0, p9(2) = 0.3, p9(3) = 0.3, p9(4) = 0.3

Computing resource Probability:

- Multiplier k=1 at timestep 1: $q_1(1)$: 1+1+0.5+0.3=2.8
- Multiplier k=1 at timestep 2: $q_k(2)$: 1+0.5+0.5+0.3=2.3

Taking the example of operation number 6, which can be scheduled at timestep 1 or 2; timeframe [1,2]:

- probability: $p_6(1)$= 0.5 and $p_6(2)$= 0.5
- distribution of multipliers at timestep 1 and 2: $q_1(1)$= 2.8 and $q_1(2)$= 2.3
- self-force(1)= 2.8*0.5 - 2.3*0.5= 0.25 and self-force(2)= -2.8*0.5+ 2.3*0.5= -0.25.
  It is important to consider the modification in the timeframe of operation 7 if operation 6 is tentatively scheduled at clock cycle 2, in this case operation 7 force is calculated. Self-force of operation 7 at timestep 3 : 2.3 * (0 - 0.5) + 0.8 * (1 - 0.5) = - 0.75
- total-force(1): 0.25; total-force(2)= -1.

Then operation 6 is scheduled at clock cycle 2 which has the least force and then operation 7 is scheduled at clock cycle 2.

3. List Scheduling Algorithm

List scheduling is a heuristic algorithm that can be adopted to solve the minimal resource-latency constrained problem or the minimal latency-resource constrained problem. It uses a priority list of operations that can be scheduled.

A. List Scheduling for minimum latency-resource constraints (ML-RC).

The list scheduling algorithm creates a list of candidate operations for each resource type in each time step.

Algorithm 5: List Scheduling for ML-RC

Input : CDFG, vector of resource number of each type *a*
Output:
l=1;
while *not all* v_n *are scheduled* **do**
  for *each resource type k=1,2,3...*n_{res} **do**
    Create list of candidate operations $U_{l,k}$;
    Create list of unfinished operations $T_{l,k}$;
    Select S_k vertices, s.t $S_k + T_{l,k} <= a_k$;
    Schedule S_k operations at time step l;
  end
  l=l+1;
end

Figure 2.1.0.1 shows the List Scheduling algorithm with 2 multipliers (MULT) and 2 arithmetic logic units (ALU).

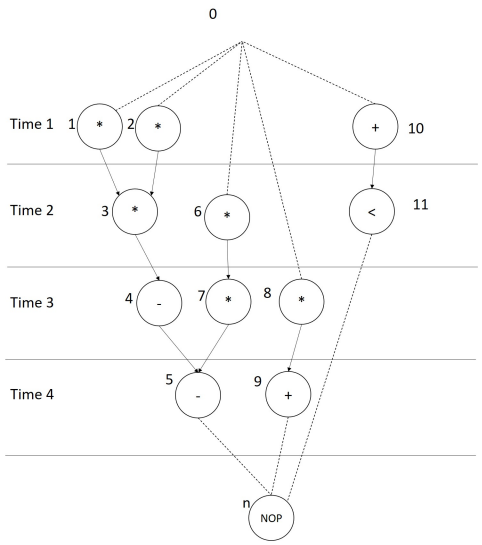

Figure 2.4: List Scheduling Algorithm for the Differential Equation, ML-RC (2 Mult and 2 ALUs)

Step 1 - $U_{1,1}$=1,2,6,8, select operation 1,2 and schedule

- $U_{1,2}$= 10, select operation 10 and schedule

Step 2 - $U_{2,1}$=3,6,8, select operation 3 and 6 and schedule

- $U_{2,2}$= 11, select operation 11 and schedule

Step 3 - $U_{3,1}$= 7,8, select operation 7 and 8 and schedule

- $U_{3,2}$= 4, select and schedule

Step 4 - $U_{4,2}$= 5,9, select operation 5 and 9 and schedule.

B. List Scheduling for Minimum Resources with latency constraints (MR-LC)

The list scheduling algorithm can solve the minimum resources- latency constraint problem by using the mobility of the operations as calculated by the ASAP and ALAP schedules. The mobility value is used as indicator of priority for scheduling the given operation. Hence, nodes with lower mobility will be scheduled first since they have highest priority, such as nodes on the critical path. The remaining operations will be scheduled next if they do not violate the resource constraints.

For each resource type, a candidate list of ready operations is created with operations sorted based on their mobility value.

Algorithm 6: List Scheduling for MR-LC

Input: G (V,E), γ
//γ latency constraint
//a: vector containing the resources with different resource type
a=1;
// l is the current clock cycle
l=1;
while *not all* v_n *are scheduled* **do**
  for *each resource type k=1,2,3...*n_{res} **do**
    Create a list of candidate operations $U_{l,k}$;
    Compute the slack or mobility of each operation

$$\left\{ s_i = t_i^L - l \forall v_i \in U_{lk} \right\} \tag{2.4}$$

    Schedule the candidate operations with zero slack and update a;
    Schedule the candidate operations that do not require additional resources;
  end
  l=l+1;
end

Figure 2.1.0.1 shows the List Scheduling algorithm for 4 clock cycles.

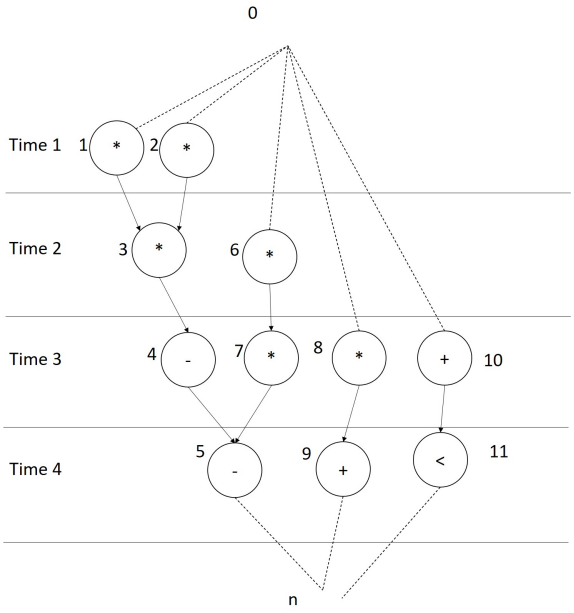

Figure 2.5: List Scheduling Algorithm for the Differential Equation, ML-LC (4 clock cycles)

4. Force Directed List Scheduling (FDLS)

The FDLS algorithm proposed by [23] solves the minimum latency with resource constraint problem. It employs the force directed and list scheduling approaches presented previously. The force calculation is used as priority attribute in the list of ready operations.

The algorithm starts by finding the time-frame of each operation, then for each time step in the time constraints, operations that can be scheduled are added to the ready-operation list. If the ready operations cannot be scheduled due to resource constraints, then force calculation is employed to determine which operation to defer out of the ready-operation list. Critical operations have a force equal to infinity which implies that these operations remain in the ready operation list to be scheduled. Other operations with least force are deferred out of the list. Finally, operations in the ready operation list are scheduled.

1) Initialize time constraint to length of critical path.
2) **for** c-step **from** 1 **to** time constraint **do**:
   2.1) Determine time frames.
   2.2) Determine *ready* operations in c-step (i.e. opns. whose time frame intersects the current c-step).
   2.3) **while** (no. of *ready* opns. > no. of FU's) **do**:
      - **if** all opns. on critical path **then**
        - extend time constraint by 1 c-step,
        - reevaluate time frames.
      - Calculate forces for possible deferrals.
      - Defer operation with lowest force.
      - Remove it from *ready* opns.

      end;
   2.4) Schedule remaining *ready* opns. in current c-step.
end;

Figure 2.6: Force-Directed List Scheduling Algorithm

2.1.0.2 Allocation and Binding

Resource Allocation specifies the type and number of needed hardware resources to execute the given behavior. Resources can be functional units, registers and interconnections. Binding assigns each operation to the allocated functional unit, as well as variables to the corresponding storage units that hold their values. Interconnections allow the flow of data between storage and functional units.

Representation

A compatibility graph is employed to represent the operations that are compatible and could be bound to the same functional unit: $G_{+}(V,E)$ where V represents the set of operations and E represent compatible vertices. Two vertices v_i and v_j are connected by an edge if they are compatible, i.e. scheduled at different time-steps and are executed by the same resource type.

For variables compatibility, the lifetime or lifespan of a variable is defined by the time it is saved in a register for its value to used.The lifetime of a variable can be determined by the the first time it is read into the register and the last time it was written into the register in the scheduled DFG. Similarly, variables that can be bound to the same register are represented by $G_{+}(V,E)$, V represents the variables and E compatible variables with non-overlapping lifetimes.

Figure 2.1.0.2 represents the compatibility graph of the given DFG.

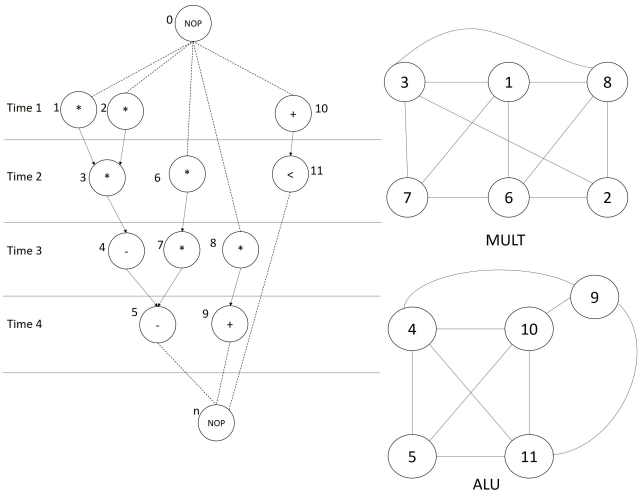

Figure 2.7: Compatibility Graphs of the given DFG

Another way of representing compatibility of operations is to employ the complement of the compatibility graph, by representing conflict operations using a conflict Graph $G_{-}(V,E)$, where V represents the set of operations and E represent conflict operations. Two vertices v_i and v_j are connected in G_{-}

if they are scheduled at the same time steps and are executed by the same resource type. Variables as well can be represented by conflict graphs.

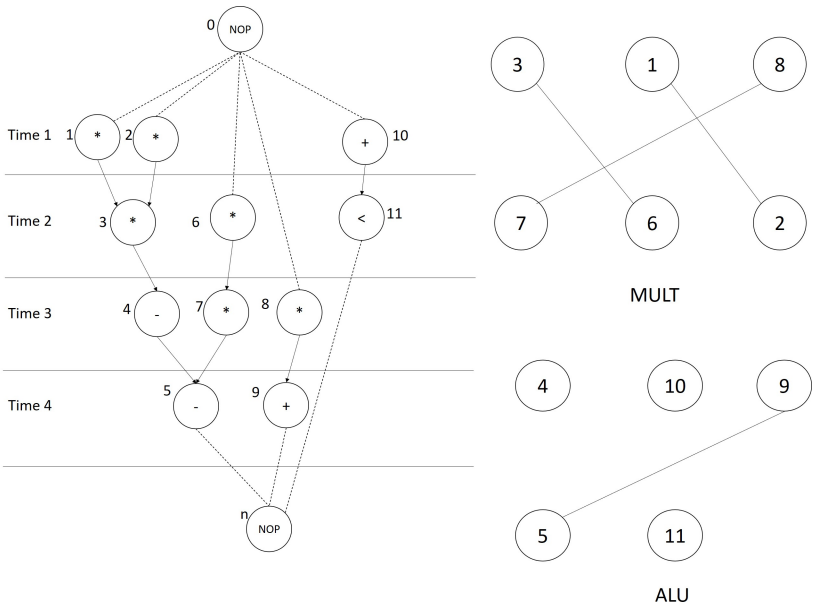

Figure 2.8: Conflict Graphs of the given DFG

Allocation and Binding Algorithms

The goal of binding is to find the minimum number of needed resources in the datapath.

1.Clique Partitioning

A Clique is a maximal complete sub-graph.In the case of compatibility graphs, finding the binding of operations and the number of required resources is achieved by partitioning the graph into a minimum number of cliques. The total number of cliques in the compatibility graph indicates the number of functional units needed for the design. This algorithm is used for both operations and variables binding, i.e to bind operations to functional units and variables to registers.

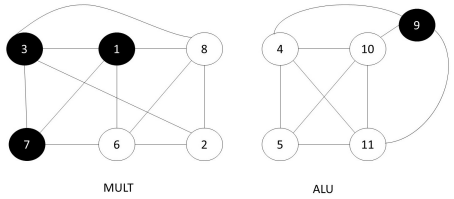

Figure 2.9: Clique Partitioning

2. Graph Coloring

In the case of conflict graphs, the goal is to coloring the graph with the minimum of colors.

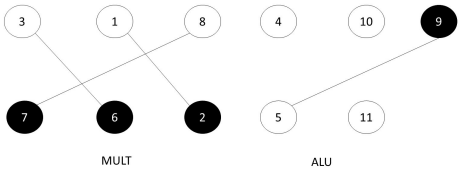

Figure 2.10: Conflict Graphs Coloring

3. Left-Edge algorithm

The Left Edge algorithm proposed by Hashimoto and Stevens [9] is mainly employed in optimizing channel assignment in the routing stage of physical design. Applied to the register allocation problem, the Left-Edge algorithm assigns a register for each set of compatible variables. After determining the variable's lifetime start and end, variables are sorted in ascending order according to their start time. Just like wire segments having an upper and lower interval bound as specified by [9], the search considers intervals with greatest upper bound less than the previously chosen lower bound interval. The interval is the analogy of variable's lifetime in this case.

Figure 2.1.0.2 shows the lifetime of each variable and how the Left-Edge Algorithm assigns variables to registers.

Algorithm 7: Left Edge Algorithm

Input : Elements E
Output: Assigned elements
//Sort elements in E in a list L of ascending order;
c=0;
while *some interval is not colored in a color c* **do**
    S= ∅; r=0;
    while *s in L such that* $l_s > r$ **do**
        s=First element in the list L with l_s>r;
        S=S ∪s;
        r= r_s; Remove s from L;
    end
    c=c+1;
    Label elements of S with color c;
end

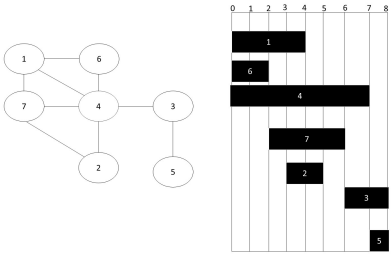

Figure 2.11: Left Edge Algorithm

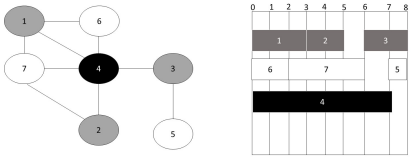

Figure 2.12: Left Edge Algorithm

2.2 Floorplanning

Floorplanning is concerned with the arrangement of blocks into a floorplan or bounding box and is completed right before placement. This phase takes into account soft blocks that have multiple aspect ratios, and ensures that each module has a defined shape and position and pins are assigned a location to route nets. A floorplanning instance includes all potential aspect ratios of each module (height and width).

The floor-planning stage can have multiple optimization goals:

- minimize the global bounding box which contains all the blocks of the floorplan.
- minimize the total wire-length which can be estimated at this stage using the Manhattan distance for example. Area and Wire-length minimization in the floorplan can be combined as in 2.5

$$\alpha \cdot \text{area}(F) + (1 - \alpha) \cdot L(F) \tag{2.5}$$

- minimize signal delays of connecting wires which are highly affected by the positions and shapes of the blocks in the floorplan.

Representation

Floorplans are categorized into slicing and non-slicing. A slicing floor-plan is a floor-plan which is obtained by recursively dissecting a rectangle by vertical or horizontal lines. A non-slicing floor-plan, on the other hand, cannot be cut by horizontal and vertical lines, these floor-plans include wheels. Figure 2.2 shows on the right a non-slicing floorplan and on the left a slicing instance.

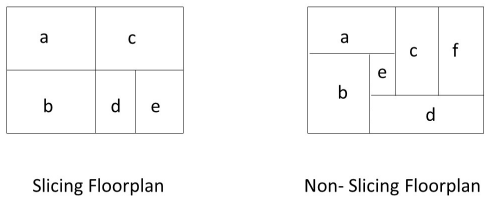

Figure 2.13: Slicing and Non-Slicing Floor-plans

Slicing floorplan can be represented using slicing trees or polish expressions. A Slicing Tree is a binary tree with n leaves and $n - 1$ internal nodes. Leaves represent blocks and internal nodes

represent a horizontal or vertical cut. A floorplan might have one or more slicing trees.To represent slicing trees, polish expressions are employed. A polish expression is obtained by traversing the slicing tree in a post-order manner or left right root order. The length of the polish expression is $2n-1$ with n the number of leaves in the slicing tree. Figure 2.2 shows a slicing tree of the floorplan and the corresponding polish expression.

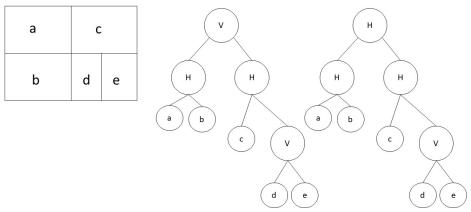

Figure 2.14: Slicing Tree

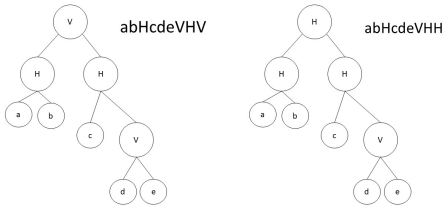

Figure 2.15: Polish expression

Other floorplanning representations include vertical and horizontal constraint graphs and sequence pairs. Constraint graphs are a pair of two directed acyclic graphs, vertical constraint graph (VCG) and horizontal constraint graph (HGC), which represent the relation between block positions. In a constraint graph $G = (V, E)$, vertices $|V| = n+2$ are connected through edges E, and a weight $w(v)$ of a block resents the dimension of the block (width or height). In a vertical constrained graphs (VCG) as in figure 2.2, two nodes $\mathrm{v_i}$ and $\mathrm{v_j}$, representing blocks i and j is below block j. The weights on the vertices of the VCG represent the heights of the blocks. The longest path in the VCG corresponds to the minimum height required to represent the blocks in a floorplan.

In a horizontal constrained graph as in figure , two nodes $\mathrm{v_i}$ and $\mathrm{v_j}$, representing blocks i and$j$ respectively, are connected by an edge from $\mathrm{v_i}$ to $\mathrm{v_j}$ if block i is on the left of block j. The weights on the vertices of the HCG represent the width of the blocks. The longest path in the HCG corresponds to the minimum width required to represent the blocks in a floorplan.

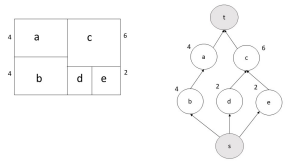

Figure 2.16: Vertical Constraint Graph

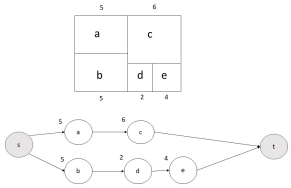

Figure 2.17: Horizontal Constraint Graph

The Sequence Pair is another floorplan representation which can generalize to both slicing and non-slicing cases. Sequence pairs $(S+, S-)$ are permutations of relations between every pair of blocks i and j. If i appears before j in both $S+$ and $S-$, then i is to the left of j. Otherwise, if i appears before j in $S+$ but not in $S-$, then i is above j. Figure 2.2 shows the Sequence Pair of the given floorplan. Blocks A and C appear in the same order in $S + (A...C..)$ and $S - (A...C..)$ then A is on the left of C. However, A and B appear in different order in $S+$ and $S-$. Block A appears before B in $S + (A..B)$ but after B in $S - (B...A)$, hence A is above B in the floorplan.

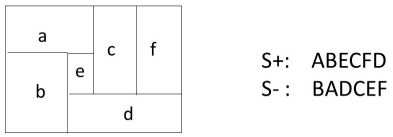

Figure 2.18: Sequence Pair

Floorplanning Algorithms

1. Floorplan sizing

Floorplan sizing calculates the minimum-area floorplan and finds the corresponding dimensions (width,height) of the blocks. This algorithm proposed by Stockmeyer [28], takes as input the flexible dimension and shapes of every block to calculate the minimal top-level floorplan. This algorithm is employed to find the minimum area of a slicing floorplan in polynomial time. However, finding the minimum area floorplan of a non-slicing floorplan is an NP-Hard problem.

2. Sequence Pair Evaluation

Building a floorplan from S+ and S- as well as obtaining the coordinates of the blocks in the floorplan is equivalent to processing the weighted Longest Common Sub-sequence (LCS) of S_+) and S_-. To determine the x-coordinates of each block , LCS(S_+,S_-) is computed. As for finding the y-coordinates of each block the LCS($S_+{}^R$,S_-) where $S_+{}^R$ is the reverse of S_+.

A fast approach is adopted using Binary Search Trees (BST) in $O(nlogn)$ [37]. The location of the blocks in sequence Y is saved in match array. The BST manages the length of the common sub-sequence where nodes include the index and length of blocks. The index and length key represent the nodes attributes where the index is the primary key. The index marks the location of the current block in sequence Y and the length denotes the length of candidates of the longest common sub-sequence. Nodes whose length is not increasing are disregarded since they make no contribution to the LCS.

Algorithm 8: Fast Evaluation of Sequence Pair [37]

```
Input  : set of all blocks B, cost function C
Output: optimized floorplan based on cost function C
Initialize Match Array match;
Initialize BST with a node (0,0);
for block i n do
    //get current block at index i
    b = X[i];
    //index of block b in Y
    p = match[b].y;
    //update P array
    //find the greatest index in BST which is less than p and return the corresponding length
    P[b] = findBST(p);
    t = P[b] + w(b);
    insert(p,t) in BST;
    discard the nodes with greater index than p and less length than t;
end
return findBST(n)
```

2.3 Reinforcement Learning and Deep Reinforcement Learning

Reinforcement Learning is an area machine learning where an agent learns a sequence of actions in an autonomous manner. The agent interacts with the environment where it observes a state, performs an action and gets a reward. The goal is to learn a policy or state-action sequence maximizing the total reward. As mentioned by [30], the reinforcement learning field involves an agent that learns what action to perform in order to maximize the long-term reward, since one action affects the subsequent ones drastically.

Reinforcement learning has been widely applied in the field of games such as Chess [15], Tic-Tac-Toe, Go [16] and Atari [22]. Additionally, Reinforcement Learning has been proposed in the fields of autonomous driving, finance, robotics and healthcare.

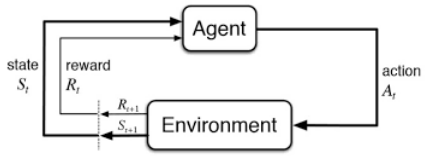

Figure 2.19: Representation of Reinforcement Learning

The formulation of a Reinforcement Learning problem is achieved through Markov Decision Process (MDP). MDPs state that the current action and state are independent of the previous states and actions [26]. A Markov Decision Process is defined by :

- S: finite set of states
- A: finite set of actions
- T : the transition function which assigns a transition probability for each action when going from state s to s'.
- R: reward function
- discount factor γ in [0,1)

Considering the example of Tic-Tac-Toe Game, which is a 3x3 grid of two players, the MDP is defined as follows:

- S: Board configuration, set of Xs and Os available on the board after each move.
- A: Placing an X or O at a position in the grid.

- T : Probability function of transitioning from state to another, depends on O's turn. Can be defined as the possible available positions where X can be placed on the board.
- R: Can be defined as 1 if X won, 0 if O won and 0.5 for braking ties.
- discount factor γ in [0,1)

The aim of the agent in the reinforcement learning environment is to find the sequence of actions resulting in the maximal reward value. This sequence is a mapping of State-Action pairs and known as policy π. A policy π is defined as a probability distribution over the set of actions. The agent differentiate between different combinations of state-action values. These values are known as state values and state-action values. First, the state value function V returns the expected reward of selecting a state s by following the policy π.

$$V^{\pi}(s_t) \equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{2.6}$$

Hence, the optimal expected return is defined as:

$$V^*(s) = \max_{\pi \in \Pi} V^{\pi}(s) \tag{2.7}$$

Second, the action-state value function Q returns the expected reward of a state-action pair by following the policy π.

$$Q^*(s,a) = \max_{\pi \in \Pi} Q^{\pi}(s,a) \tag{2.8}$$

The state function and state-action function are closely related as such:

$$Q_*(s,a) = \mathbb{E}\left[R_t + \gamma V_*(S_{t+1}) \mid S_t = s, A_t = a\right] \tag{2.9}$$

where $\mathbb{E}$ is the expected reward. Hence, the optimal policy can be obtained:

$$\pi^*(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q^*(s,a) \tag{2.10}$$

To find the optimal Q value, the Bellman equation is employed 2.11.

$$Q_*(s,a) = E\left[R_{t+1} + \gamma \max_{a'} Q_*(s',a')\right] \tag{2.11}$$

Reinforcement Learning algorithms can be Model-based or Model-free. In model-based algorithms,

the reward functions and transition probabilities (model) are given or learnt by the agent. Dynamic Programming is an example of mode-based RL . As for model-free algorithms, the agent predicts the optimal policy without previous knowledge about the transition probability and reward function of the MDP. This can be achieved either through approximating the policy, the value-function V or Q value from which an optimal policy is derived [12]. Q-learning is a model-free method.

2.3.0.1 Deep Reinforcement Learning

Deep Learning is a subset of Machine Learning that employs multiple layers of Artificial Neural Networks.

In Deep Reinforcement Learning, the agent is implemented using a deep neural network. Specifically, the network is given as input the state information and learns its features to make adequate decisions [41].

1. Deep Q-Learning

In usual Q-learning algorithms, the values of the Q-value function are saved in a look-up table for later reference. The agent employs previously saved values in the Q-table or picks a random action to balance exploitation and exploration.

In huge state-spaces, it is inefficient to store every state-action pair in a two dimensional table. For that reason, it is conventional to use a function approximate to estimate Q-values. Deep Q learning has been first proposed by [21] to play the Atari Game. At each step, and according to an ϵ selects the maximum Q value or performs a random action according to 1-ϵ. The current state, action, reward and next state (s, a, r, s') are recorded in a replay buffer. Sometimes, two neural networks are employed, a Q-network which calculates the Q-value at the current step and a target network which calculates the Q-value for the subsequent step. The saved information in the replay buffer are used to update the target-network weights via back propagation[8].

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 2.20: Deep Q Learning with Experience Replay [21]

2. Deep Policy Gradient

As opposing to the Q-learning methods, policy gradient focus on learning directly the optimal policy. The policy π is associated with a parameter θ, this parameter is learned using a deep neural network in the case of deep reinforcement learning. A deep policy network is employed to output action probabilities and input is the current state. The network's weights are the policy parameters,θ. As the name implies, policy optimization is guided by the gradient approach to update the parameters [31]. Policy based methods and Value based methods can be combined through an actor-critic algorithm involving an actor which learns a policy π and a critic which evaluates the action using a value-function.

2.4 Related Work

The first part of this section presents previous research effort related to the integration of high-level synthesis and floorplanning. The second part demonstrates research adopting Reinforcement Learning as solution to problems associated with chip design stages.

2.4.1 Binding, Allocation and Floorplanning

Simultaneous binding and floorplanning have been addressed in the literature mainly using heuristic methods. Simulated Annealing is a stochastic global optimization technique, inspired by metal annealing. At each step, the algorithm searches the solution space for more optimized solutions while sometimes accepting worst solution to escape the local optimum. In the work [27], nested Simulated Annealing is employed to minimize area and power. An outer loop performs binding moves such as swapping and sharing functional units, followed by a floorplan update. An inner loop performs floorplan optimization for interconnect power.

Similarly, FloM is another approach employing an outer-level Simulated Annealing process to optimize the total power of the floorplan components. An inner Simulated Annealing loop performs voltage scaling, functional unit rebinding or swapping, register rebinding or swapping [38]. Sequential Simulated Annealing is also employed where one Simulated Annealing loop optimize the binding through different moves followed by a second Simulated Annealing loop optimizing the floorplan to obtain an efficient interconnect wire-delay design [14].

As for the Branch and Bound algorithm, it consists of enumerating all possible solutions by constructing a rooted tree and pruning branches violating the given bound constraints. Branch and bound is another method used to deal with simultaneous binding and floorplanning. A probabilistic gain is assigned for binding operation-functional units and register-variable pairs.Pairs with maximum gains are selected and the floorplan is updated. The feasibility of the solution is checked, solutions with maximum clock cycle constraint violation are disregarded [29].

2.4.2 Reinforcement Learning

Reinforcement learning has been employed in different VLSI design stages, where the model learns from experience without human intervention to output efficient designs.

Placement from the physical design level in VLSI has been addressed in [20] by employing a Proximal Policy Optimization, which is a policy based RL technique. The agent's actions are based on the possible locations a macro can be placed. The reward is a weighted sum of the wire-length, congestion, and density. A Graph Convolution Neural Network is employed to learn the state's features, mainly netlist and macro information. Similarly, Deep Reinforcement Learning is applied for mapping the nodes of a graph to resources [7]. Tensor-Flow computational graphs are mapped to TPU or GPU, ASIC net-lists to grid cells, and FPGAs to configurable logic blocks. This placement will be tackled using a gradient policy optimization approach. The network will output a probability of assigning each node to a device according to the graph's type. Another work [35] applied Reinforcement Learning combined with heuristics for the placement problem. The agent of the policy network will learn to select a candidate block to swap with an initial block in the sequence pair representation. After several iterations, the Simulated Annealing algorithm is performed for a couple of other iterations to optimize the placement. A Proximal Policy Actor-Critic Network is adopted and the algorithm is tested on the ami49 benchmark instance. Additionally, the work [6] proposed the use of Deep Reinforcement Learning for FPGA placement. The agent chooses between types of blocks to be randomly swapped. This move is evaluated in a Simulated Annealing algorithm and a reward is given to the agent afterwards.

Global routing from the physical design of VLSI chips is tackled as well. Routing involves connecting a number of nets through pins using wires. The work [19] proposed a Deep Q-Learning Network route nets. Precisely, the agent perceives the state, a vector representing its location, the target pin location. The agent's action is to choose a direction to adopt and move to. The authors suggested the use of the A^* algorithm as a burn-in memory for DQN and as a benchmark for later comparison.

Floorplanning from the physical design has been addressed through deep Reinforcement learning, precisely through deep Q-learning [10]. Neighboring solutions are obtained by modifying the sequence pair representation of the floorplan through six actions. The agent of the RL problem will choose either to accept one neighboring solution or to reject the move. This approach is tested on the MCNC and GCRS benchmarks.

Scheduling in HLS for FPGAs is tackled using Deep Reinforcement Learning by [1]. The first step is supervised learning, where generated Data-flow Graphs are scheduled by expert demonstration. The

expert will move operations in their time-frame using Integer Linear Programming (ILP). State-action pairs are given as input to a policy network which learns the movement of operations in the time frame done by the expert. This first policy network is trained using policy gradient to learn which action to choose. Since one policy network is not able to generalize over all states, another policy network is used Then, another policy network will be trained on a set of random graphs with initial weights as calculated by the previous policy network.

Chapter 3

Layout Driven Binding Using Deep Reinforcement Learning

Since High-Level Synthesis is at the higher level of abstraction where circuit components are decided upon, it is logical for researchers to combine layout information at this earlier stage of chip design. Considering layout information can reduce the area of the chip in a radical manner by making decision upon chip components at an earlier stage. By introducing layout information into scheduling, binding and allocation of High-Level Synthesis, the focus will not only be directed to the number of needed resources but also to their shape and location. As mentioned by Xing and Jong [39], floorplanning of the physical synthesis provides a good estimate of the area occupied by the circuit components. This explains the integration of floorplanning and High-Level synthesis, especially that it is less computationally exhaustive than other stages of physical design [39]. Also, floorplanning has a great influence on routing which connects the circuit components in the chip.

To combine layout information with High-Level Synthesis, a datapath representation should be employed. This is achievable through modeling the different operations and variables using appropriate data structures and binding them functional units. Simultaneously, the floorplan representation is needed to model the arrangement of the needed resources in the datapath. Figure 3 shows a datapath formed of ALUs, Multipliers and Registers. These functional units are placed in the floorplan.

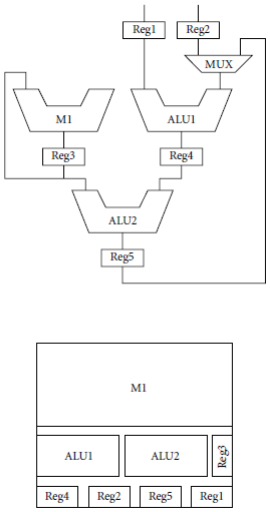

Figure 3.1: Datapath and its corresponding Floorplan from [38]

The tight relation between the datapath itself and layout information is modeled through possible actions performed on the datapath. Deciding on allocating a certain functional unit to operations infer changes on the floorplan and blocks' arrangement. For instance, assigning two operations the same multiplier can reduce the number of functional units used in the datapath and hence the area of the floorplan.

As previously seen, there do previous work adopted a machine learning approach to tackle the layout driven binding problem. Earlier research focused on employing heuristic methods that optimize the given design directly. Heuristic methods learn where the optimal solution is in the solution space, whereas machine learning methods applied to optimization problems learn how to find this optimum [18].

3.1 Methodology

In this section we define the approach used to tackle layout driven binding. The first step is to model the datapath using compatibility graphs. Second, the environment should be defined to be able to apply the adopted reinforcement learning algorithm. Finally, choosing the suitable parameters and training the RL model is accomplished and results are obtained over different examples.

3.1.1 Data Preprocessing

The given behavior modeled as DFG description is parsed into operations and variables used as inputs and outputs for the these operations. The parsed DFG is scheduled by employing the Force-Directed Scheduling Algorithm [23] previously explained, a compatibility graph is built for each type of operations and registers. The compatibility of operations is defined by their schedulability in different clock cycles. As for variables, their lifetime defines registers compatibility. Operations and variables are represented using compatibility graphs as in 2.1.0.2 and the floorplan using the sequence pair S+ and S- representation mentioned in 2.2.

3.1.2 Deep Reinforcement Learning

3.1.2.1 Environment Definition

The Markov Decision Process (MDP) framework for Reinforcement Learning is defined. The MDP describes the State space, Action space, transition and reward functions.

A. State Space:

The state space is represented as:

- b a vector representing an encoding of the binding of operations to functional units and variables to registers, such as $resource_{\mathrm{j}}$ is assigned to $operation_{\mathrm{i}}$. With n is the total number of operations and variables and m number of needed resources:

$$b = \begin{bmatrix} op_i & res_j \\ op_{i+1} & res_{j+1} \\ \vdots & \\ op_n & res_m \end{bmatrix} \tag{3.1}$$

B. Action Space:

1. Share functional unit: based on the gain value on the edges of the compatibility graph of operations, two operations with maximum gain are assigned the same functional unit.

$$gain_{\mathrm{i,j}} = N_{\mathrm{i,j}} + comm_{\mathrm{i,j}} + F_{\mathrm{i,j}} + Mux \tag{3.2}$$

where:

$N_{i,j}$ = number of inputs or outputs of operation$_{i,j}$ residing in the same register

$comm_{i,j}$ = 1 if both operations are commutative (multiplication or addition)

$F_{i,j}$ = indicating the gain of functional units when merging the operations

Mux = need of a mutiplexer

2. Change Variables Binding: based on the gain value on the edges of the compatibility graph of variables, two variables with maximum gain are assigned the same register.

$$gain_{i,j} = R_{i,j} - C_{i,j} + Mux, gain_{j,i} = R_{j,i} - C_{j,i} + Mux \tag{3.3}$$

where:

$R_{i,j}, R_{j,i}$ = number of registers saved

$C_{i,j}, C_{j,i}$ = number of conflict with variables already residing in the register

Mux = need of a multiplexer

3. Swap Binding of two compatible operations: pick two compatible operations and swap their binding to functional units.

4. Perform Hill Climbing: this algorithm is performed as a local search procedure to optimize the floorplan. Given the sequence-pair representation S+ and S-, two random components of the sequence-pair are chosen to be swapped. The algorithm is executed for a number of iterations.

Algorithm 9: Hill Climbing

```
Input  : initial solution, number of iterations
Output: best solution
// s = initial solution
s = sequence pair of the current floorplan S+ and S-
bestscore = initial deadspace value
for i in iterations do
    S+', S-' = modify(S+,S-);
    newscore = evaluate s';
    if newscore ≤ bestcore then
        bestsolution = s' ;
        bestscore = newscore;
    end
end
```

Algorithm 10: Modify Sequence pair

Input : S+,S-

Output: modified : S+,S-

i = random index(S+);

j= random index(S+);

//swap S+[i] , S+[j] = S+[j] , S+[i] ;

k = random index(S-);

l = random index(S-);

//swap

S-[k] , S-[l] = S-[l] , S-[k] ;

C. Transition from s to s$^{'}$: Deep Neural network with input the state s_t and output the probability associated with each action. The next action is the one with maximum probability value.

D. Reward: δ area + α * (δ dead-space). Where $\alpha = 0.2$. It is important to note that these values are normalized on a same scale of [0,1]. The choice of the reward is a weighted sum of the goal to be minimized. The agent is encouraged to perform a positive change by reducing the area and dead-space of the design.

The agent will learn to optimize the datapath through binding of functional units and registers to operations and variables respectively. The agent's action will be driven by the need to minimize the design area through the floorplan representation. This is achieved through performing the chosen action on operations and variables with maximum gain. The gain formulas focus on minimizing the number of components used in the datapath, mainly registers, functional units and multiplexers. It is important to note that when merging functional units, multiplexers are introduced. These circuit components act as data selectors given a certain set of inputs.

After performing a certain action, the reward value is determined by calculating the area and induced dead-space. The area of the floorplan is determined through the Longest Common Subsequence (LCS) of the sequence pair [37]. As for the Dead-Space percentage is calculated using 3.4

$$Dead - Space = \left(\frac{A - \sum_{i=1}^{n} A_i}{A} \right) \times 100 \tag{3.4}$$

with A the total area of the floorplan and A_i is the area of block i.

3.1.2.2 Deep Reinforcement Learning Algorithm

After defining the MDP framework of the problem, a RL algorithm is chosen to train the agent. The Policy Gradient algorithm which is a model-free policy-based algorithm will be adopted. Policy-based methods are guaranteed to converge faster as compared to value-based methods [5]. **Policy Gradient**

Policy Gradient methods optimize a policy π using Gradient Descent. The Reinforce algorithm is adopted to achieve this optimization [36]. Specifically, a parameter θ is associated with the policy π to be optimized. An objective function J(θ) is defined(3.5), this objective function J(θ) reflects the goal of obtaining the expected discounted total reward by following the parameterized policy π. Hence, the goal is to learn θ which maximizes the function J(θ).

$$J(\theta) = E_{\pi\theta}\left[\sum \gamma r\right] \tag{3.5}$$

The Reinforce algorithm is a Monte Carlo method, thus it learns from the set of episodes, or series of steps, performed by the agent during its experience without having prior knowledge about the transition function of the MDP framework.

The vector of parameters θ is learned through a Deep Neural Network. This network is known as Policy Gradient Network with weights representing the θ parameters. The Policy Network, or agent, is a Deep Neural Network with a set of hidden layers. The activation function of the last layer of this network is 'Softmax', since the output of the network is a probability distribution of the environment actions. The calculated Gradient is adopted to update the weights theta (3.6). The policy gradient algorithm moves the parameters θ in the direction that returns actions with highest reward.

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{3.6}$$

With α the learning rate.

Deriving (3.5) results in (3.7) which will be adopted in updating the value of the parameter θ. It is important to note the use of logarithm which is useful while deriving a probability function.

$$\nabla_\theta J(\theta) = E_\pi \left[\nabla_\theta (\log \pi(\tau \mid \theta)) R(\tau)\right] \tag{3.7}$$

To train the policy network, the Reinforce algorithm is employed as shown in algorithm 11. At each step of the episode, the Policy Network will predict a probability distribution assigned to the different actions available in the environment given the state description. The set of state, action, reward and

next state in each episode are recorded. The set of discounted reward is employed to calculate the gradient and update the weights of the Policy Network.

Algorithm 11: Reinforce Algorithm

Input : number of *episodes*, number of *steps*
Output: Policy π
Initialize θ (policy network weights) randomly;
for *e in episodes* **do**
  for *s in steps* **do**
    Perform an action as predicted by the policy network;
    Record s, a, r, s$^{'}$;
    Calculate the gradient as per eq.3.7;
  end
  Update θ as per eq.3.6;
end

Chapter 4

Experimental Results

The proposed approach was implemented using Python and using the following libraries:

- Keras for Deep Learning
- Gym for Reinforcement Learning
- Matplotlib for Plotting
- Numpy for arrays and matrices
- Pandas for Data analysis and manipulation

Google Colaboratory is adopted to run the algorithm. The parameters set for the Deep Policy Network of the Deep RL algorithm are shown in table 4.1

In the remainder of this chapter, we test the proposed approach using the high-level synthesis benchmarks using the algorithmic behaviors discussed in Appendix 5. The results are evaluated based on the number of resources, the floorplan area, and the floorplan dead-space.

Parameter	**Value**
Number of Deep Neural Layers	2
Number of Neurons per layer	8
Number of Steps	100
Number of Episodes	500
Learning Rate α	0.005
Gamma γ	0.99

Table 4.1: Parameters of the Deep Reinforcement Learning

For every benchmark, we show the initial floorplan, compacted by a 2D bin packing algorithm along with the final solutions and their floorplan representations. We use a simple scheduling algorithm, the ASAP, for illustration purposes as any other scheduling method can be used and would most likely improve the results in terms of resources. Finally, the data path resources such as multipliers, arithmetic logic units, multiplexers and registers are used with the widths and heights shown in Table 4.2 as inspired by [38].

Resource Type	**Dimension (in unit)**
Multiplier	5x20
ALU	3x8
Register	2x4
Mux 2:1	2x1
Mux 4:1	2x5
Mux 8:1	5x4

Table 4.2: Resource Library and Dimensions

The proposed algorithm is applied to different behaviors: the Second Order Differential equation (Diffeq) in section4.1.1, Poly-design (poly), Auto-Regressive filter (AR), and Discrete Cosine Transform (DCT). The previously mentioned benchmarks have the same format as shown in 5.
Results over the number of episodes are evaluated based on the lowest area and dead-space values.

4.1 High-Level Synthesis Benchmarks Results

4.1.1 Differential Equation (Diffeq)

The first benchmark is the differential equation iterative solver. This solver has 6 multiplications, 2 substractions and 1 addition. Figure 4.1 shows its behavior.

```
t1 := uinport * dxport
t2 := 3 * xinport
t3 := 3 * yinport
t4 := t1 * t2
t5 := dxport * t3
t6 := uinport - t4
u_var := t6 - t5
y1 := u_var * dxport
y_var := yinport + y1
x_var := xinport + dxport
xoutport := x_var
youtport := y_var
uoutport := u_var
```

Figure 4.1: Differential Equation Behavior

Num of Operations	Latency	Num of Mult	Num of ALU	Num of Reg	Num of Mux
10	7	2	1	8	3

Table 4.3: DiffEQ Resources and Latency

The initial floorplan is shown in figure 4.2 with area 840 units. The resulting floorplan is shown in figure 4.3 with an area of 360 units (10x36) and dead-space 9%.

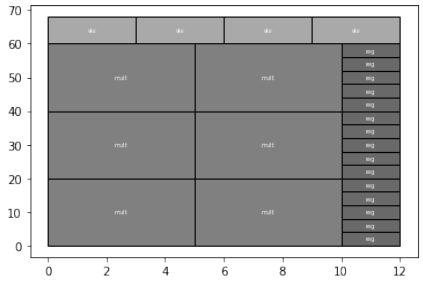

Figure 4.2: Initial Floorplan

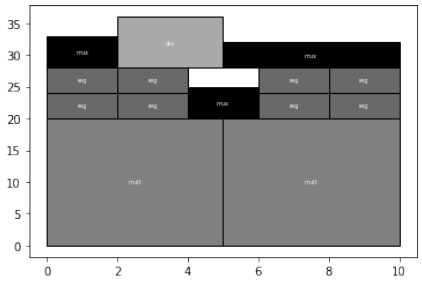

Figure 4.3: Diffeq Floorplan

4.1.2 Polynomial Design (Poly)

The Polynomial Design outputs a polynomial equation of the third degree given the coefficients a, b, c and d. This design has 7 operations, 4 multiplications and 3 additions. Figure 4.4 shows its behavior.

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

Figure 4.4: Polynomial Design Behavior

Num of Operations	Latency	Num of Mult	Num of ALU	Num of Reg	Num of Mux
7	4	1	1	6	2

Table 4.4: Poly-Design Resources and Latency

The initial floorplan is shown in figure 4.5 with area 576 units. The resulting floorplan is shown in figure 4.6 with an area 225 units and dead-space 7%.

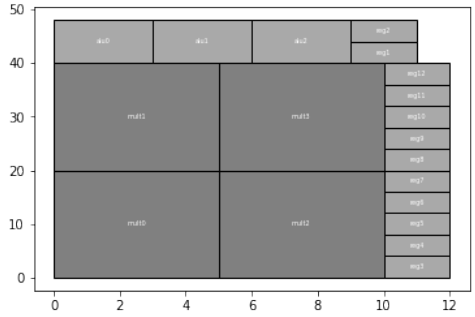

Figure 4.5: Initial Floorplan

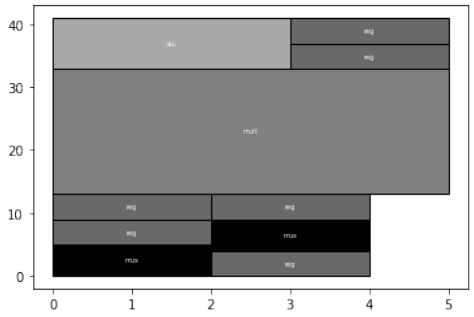

Figure 4.6: Poly-Design Floorplan

4.1.3 Auto-Regressive Filter (AR)

The Auto-Regressive Filter or Infinite Impulse Response is used in Signal Processing. This behavior filters a signals filter where the current output is dependent on previous values. The number of operations is 28 for this benchmark.Figure 4.7 shows its behavior.

```
m1 := i0 * i1
m2 := i2 * i3
m3 := i4 * i5
m4 := i6 * i7
m5 := i8 * i9
m6 := i10 * i11
m7 := i12 * i13
m8 :=  i14 * i15
a9 := m1 + m2
a10 := m3 + m4
a11 := m5 + m6
a12 := m7 + m8
a13 := a11 + i16
a14 := a12 + i17
m15 := a14 * i18
m16 := a13 * i19
m17 := a13 * i20
m18 := a14 * i21
a19 := m15 + m16
a20 := m17 + m18
m21 := a20 * i22
m22 := a19 * i23
m23 := a19 * i24
m24 := a20 * i25
a25 := m21 + m22
a26 := m23 + m24
a27 := a9 + a25
a28 := a10 + a26
```

Figure 4.7: AR-Filter Behavior

Num of Operations	Latency	Num of Mult	Num of ALU	Num of Reg	Num of Mux
28	17	4	2	31	12

Table 4.5: Ar-Filter Resources and Latency

The initial floorplan is shown in figure 4.8 with an area 2040 units. The resulting floorplan is shown in figure 4.9 with area 1148 units and dead-space 27%.

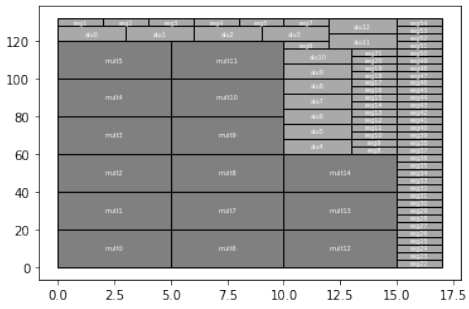

Figure 4.8: Initial Floorplan

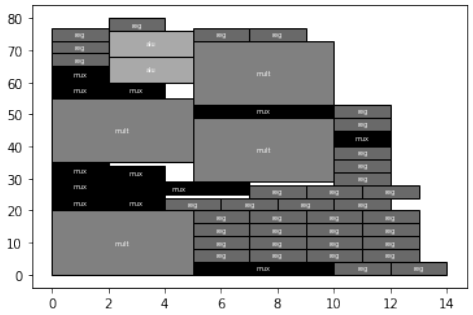

Figure 4.9: Ar-Filter Floorplan

4.1.4 Discrete Cosine Transform (DCT)

The Discrete Cosine Transform is employed to transform a signal into numeric data. In image processing, it transforms an image into a sum of sinusoidal frequencies.Figure 4.10 shows its behavior.

```
u12 := d2 + d5
u11 := d1 + d6
u13 := d3 + d4
u10 := d0 + d7
v12 := d2 - d5
v11 := d1 - d6
v13 := d3 - d4
v10 := d0 - d7
v21 := u12 - u11
v20 := u13 - u10
u21 := u11 + u12
u20 := u13 + u10
w210 := v21 + v20
w111 := v11 + v12
w110 := v10 + v13
w120 := w111 + w110
z3 := 2 * v21
z2 := 2 * w210
z4 := 2 * v20
z1 := 2 * u21
z0 := 2 * u20
z5 := 2 * v12
z6 := 2 * v12
z7 := 2 * w111
z8 := 2 * v11
z9 := 2 * v11

z10 := 2 * w120
z11 := 2 * v13
z12 := 2 * v13
z13 := 2 * w110
z14 := 2 * v10
z15 := 2 * v10
p6 := z2 + z3
p2 := z2 + z4
p0 := z1 + z0
p4 := z0 - z1
p5l := z5 + z7
p5r := z10 + z14
p5 := p5l + p5r
p3l := z7 + z8
p3r := z10 + z11
p3 := p3l + p3r
p7l := z9 + z10
p7r := z12 + z13
p7 := p7l + p7r
p1l := z6 + z10
p1r := z13 + z15
p1 := p1l + p1r
```

Figure 4.10: DCT Behavior

Constraints for the Discrete Cosine Filter behavior are shown in table 4.6.

Num of Operations	Latency	Num of Mult	Num of ALU	Num of Reg	Num of Mux
48	19	6	6	44	16

Table 4.6: DCT Resources and Latency

The initial floorplan is shown in figure 4.11 with area 3000 units. The resulting floorplan is shown in figure 4.12 with area 1587 units and dead-space 19%.

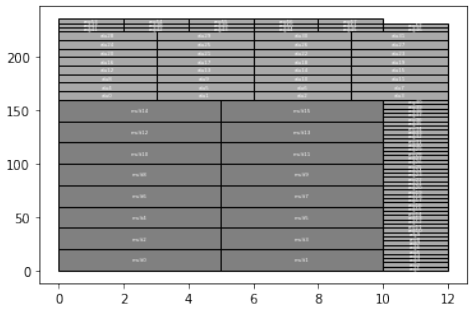

Figure 4.11: Initial Floorplan

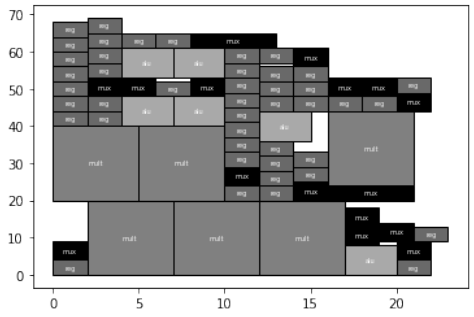

Figure 4.12: DCT Floorplan

4.2 Discussion

Starting by the scheduling algorithm, using the Force-Directed List scheduling Algorithm aims at balancing the concurrency and resource usage in the datapath. Hence, resource allocation is limited by the bound set for the scheduling algorithm as compared to initial allocation of the behavior. Moving to the binding step, the agent focuses on assigning the given operations and variables to the available resources given their dimensions. Finally, the layout is optimized using a local search algorithm. During the training process, the agent learns to perform alternating binding and layout optimization moves guided by the change in the floorplan.

The layout is represented through the sequence pair (S+,S-), hence while removing components from the floorplan, some dead-space is introduced. For that reason, a compaction algorithm is implemented when needed. It is also noticed that when the behavior gets more complex, more layout components are needed hence a wider solution space to search via local search optimization. This is controlled via the number of iterations the local search algorithm performs.

The run-time shown in table 4.7 for of the total number of episodes is influenced by the benchmark as well. In general, training time is between 3 and 6 hours for the shown benchmarks. With more complex behavior, an increased number of operations will have to be bound to resources by the agent. As well as more components to optimize in the floorplan.

Benchmark	**Time in s**
Diffeq	15600
Poly	15300
AR-Filter	19800
DCT	21600

Table 4.7: Run-Time in Seconds

As for the comparison of the produced results with previous works, this step was not mentioned since previous research considering on simultaneous binding and floorplanning focused on power consumption without considering registers and multiplexers placement in the layout.

4.3 Conclusion

In this chapter, the proposed algorithm is thoroughly explained. Starting by the Reinforcement Learning setting and moving to the detailed algorithm. Additionally, the model implementation is discussed and applied to different High-Level Synthesis behaviors. Results show optimal binding and allocation with a balance of latency and resources constraints as well as layout representation.

Chapter 5

Conclusion

This work presents an automated layout driven High-Level Synthesis technique in VLSI. A Deep Reinforcement Learning Approach is adopted, where an agent chooses an action that will reduce the area of the design given its behavior. The model's target is to preform an optimal datapath binding by arranging a floorplan with minimum area and dead-space. The agent's action focus on binding operations and variables to functional units and registers, then optimizing the layout representation which is the floorplan. The built prototype learns optimization steps by employing the Policy Gradient Reinforcement Learning algorithm. Several algorithmic behaviors are tested, such as Differential Equation Behavior and Auto-Regressive Filter. Results show further area reduction through optimized binding. The introduced method considers multiplexers in the layout unlike previously mentioned work. Further work might introduce scheduling as part of the agent's action. Since scheduling influence in a considerable way the compatibility of operations and variables, it affects as well the allocation and placement of different resources in the layout. Additionally, wire-length estimation can be added as optimization goal in the floorplan.

Bibliography

[1] H. Chen and M. Shen. A deep-reinforcement-learning-based scheduler for fpga hls. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019.

[2] Sanjay Churiwala and Sapan Garg. *Introduction to VLSI RTL Designs*, pages 1–20. Springer New York, New York, NY, 2011.

[3] Philippe Coussy, Daniel Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *Design and2 Test of Computers, IEEE*, 26:8–17, 09 2009.

[4] A. Davoodi and A. Srivastava. Power-driven simultaneous resource binding and floorplanning: a probabilistic approach. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(8):934–942, 2005.

[5] Li Deng and Dong Yu. Deep learning: Methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, 2014.

[6] Mohamed A. Elgammal, Kevin E. Murray, and V. Betz. Learn to place: Fpga placement using reinforcement learning and directed moves. *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 85–93, 2020.

[7] Anna Goldie and Azalia Mirhoseini. Placement optimization with deep reinforcement learning. In *Proceedings of the 2020 International Symposium on Physical Design*, ISPD '20, page 3–7, New York, NY, USA, 2020. Association for Computing Machinery.

[8] Shanghang Zhang Hao Dong, Zihan Ding. *Deep Reinforcement Learning: Fundamentals, Research, and Applications*. Springer Nature, 2020.

[9] Akihiro Hashimoto and James Stevens. Wire routing by optimizing channel assignment within large apertures. In *Proceedings of the 8th Design Automation Workshop*, DAC '71, page 155–169, New York, NY, USA, 1971. Association for Computing Machinery.

[10] Z. He, Y. Ma, L. Zhang, P. Liao, N. Wong, B. Yu, and M. D. F. Wong. Learn to floorplan through acquisition of effective local search heuristics. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 324–331, 2020.

[11] Guyue Huang, Jingbo Hu, Yifan He, Jianhui Liu, Mingyuan Ma, Zhaoyang Shen, Juejian Wu, Y. Xu, Hengrui Zhang, K. Zhong, Xuefei Ning, Yuzhe Ma, H. Yang, Bei Yu, Huazhong Yang, and Yu Wang. Machine learning for electronic design automation: A survey. *ArXiv*, abs/2102.03357, 2021.

[12] Q. Huang. Model-based or model-free, a review of approaches in reinforcement learning. In *2020 International Conference on Computing and Data Science (CDS)*, pages 219–221, 2020.

[13] Andrew B. Kahng, Jens Lienig, Igor L. Markov, and Jin Hu. *VLSI Physical Design: From Graph Partitioning to Timing Closure*, chapter 1. Springer Publishing Company, Incorporated, 1st edition, 2011.

[14] V. Krishnan and S. Katkoori. Minimizing wire delays by net-topology aware binding during floorplan- driven high level synthesis. In *2007 IFIP International Conference on Very Large Scale Integration*, pages 99–104, 2007.

[15] Matthew Lai. Giraffe: Using deep reinforcement learning to play chess. *ArXiv*, abs/1509.01549, 2015.

[16] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 2140–2146. AAAI Press, 2017.

[17] Lalin L. Laudis, Shilpa Shyam, V. Suresh, and Ajay Kumar. A study: Various np-hard problems in vlsi and the need for biologically inspired heuristics. In Pankaj Kumar Sa, Sambit Bakshi, Ioannis K. Hatzilygeroudis, and Manmath Narayan Sahoo, editors, *Recent Findings in Intelligent Computing Techniques*, pages 193–204, Singapore, 2018. Springer Singapore.

[18] Ke Li and Jitendra Malik. Learning to optimize, 2016.

[19] Haiguang Liao, Wentai Zhang, Xuliang Dong, B. Póczos, K. Shimada, and L. Kara. A deep reinforcement learning approach for global routing. *ArXiv*, abs/1906.08809, 2019.

[20] A. Mirhoseini, Anna Goldie, M. Yazgan, J. Jiang, Ebrahim M. Songhori, S. Wang, Youngjoon Lee, E. Johnson, Omkar Pathak, Sungmin Bae, Azade Nazi, Jiwoo Pak, Andy Tong, Kavya Srinivasa,

W. Hang, E. Tuncer, Anand Babu, Quoc V. Le, J. Laudon, R. Ho, R. Carpenter, and Jeff Dean. Chip placement with deep reinforcement learning. *ArXiv*, abs/2004.10746, 2020.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *NIPS Deep Learning Workshop 2013*, 2013. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.

[23] P.G. Paulin and J.P. Knight. Force-directed scheduling for the behavioral synthesis of asics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–679, 1989.

[24] Pierre Paulin and J.P. Knight. Force-directed scheduling in automatic data path synthesis. pages 195– 202, 07 1987.

[25] D. S. H. Ram, M. C. Bhuvaneswari, and Shanthi S. Prabhu. A novel framework for applying multiobjective ga and pso based approaches for simultaneous area, delay, and power optimization in high level synthesis of datapaths. *VLSI Design*, 2012, 2012. Copyright - Copyright © 2012 D. S. Harish Ram et al. D. S. Harish Ram et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited; Last updated - 2018-10-08.

[26] Mohit Sewak. *Deep Reinforcement Learning - Frontiers of Artificial Intelligence*. Springer, 2019.

[27] A. Stammermann, D. Helms, M. Schulte, A. Schulz, and W. Nebel. Binding allocation and floorplanning in low power high-level synthesis. In *ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No.03CH37486)*, pages 544–550, 2003.

[28] Larry Stockmeyer. Optimal orientations of cells in slicing floorplan designs. *Information and Control*, 57(2):91–101, 1983.

[29] V. Sundaresan and R. Vemuri. A novel approach to performance-oriented datapath allocation and floorplanning. In *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*, pages 6 pp.–, 2006.

[30] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[31] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 2000.

[32] S. Tarafdar, M. Leeser, and Zixin Yin. Integrating floorplanning in data-transfer based high-level synthesis. In *1998 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (IEEE Cat. No.98CB36287)*, pages 412–417, 1998.

[33] M. Tatsuoka, R. Watanabe, T. Otsuka, T. Hasegawa, Qiang Zhu, R. Okamura, Xingri Li, and T. Takabatake. Physically aware high level synthesis design flow. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015.

[34] Masato Tatsuoka and M. Kaneko. Wire congestion aware high level synthesis flow with source code compiler. *2018 International Conference on IC Design & Technology (ICICDT)*, pages 101–104, 2018.

[35] Dhruv Vashisht, Harshit Rampal, Haiguang Liao, Y. Lu, D. Shanbhag, E. Fallon, and L. Kara. Placement in integrated circuits using cyclic reinforcement learning and simulated annealing. *ArXiv*, abs/2011.07577, 2020.

[36] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, May 1992.

[37] Xiaoping Tang, Ruiqi Tian, and D. F. Wong. Fast evaluation of sequence pair in block placement by longest common subsequence computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(12):1406–1413, 2001.

[38] Xianwu Xing and Ching Jong. Floorplan-driven multivoltage high-level synthesis. *VLSI Design*, 2009, 08 2009.

[39] Xianwu Xing and Ching Jong. Floorplan-driven multivoltage high-level synthesis. *VLSI Design*, 2009, 08 2009.

[40] Y. W. Yunfeng Wang, J. B. Jinian Bian, Q. W. Qiang Wu, and H. H. Heng Hu. Reallocation and rescheduling after floor-planning for timing optimization. In *ASIC, 2003. Proceedings. 5th International Conference on*, volume 1, pages 212–215 Vol.1, 2003.

[41] Alexander Alexander Zai. *Deep Reinforcement Learning in Action.* Manning Publications Company, 2020.

Appendix

DFG Description of : ./poly_design.vhdl

Input : x d c b a

Output : output

InOut :

Preset Variables :

Preset Values :

Constants :

Constant Values :

Bit Width : 4

--

m1 := a * x

s1 := m1 + b

m2 := x * x

m3 := s1 * m2

m4 := c * x

s2 := m4 + d

s3 := s2 + m3

output := s3

DFG Description of : DIFFEQ.vhdl

Input : uinport yinport aport dxport xinport

Output : uoutport youtport xoutport

InOut :

Preset Variables :

Preset Values :

Constants : 3

Constant Values : 3

Bit Width : 4

--

t1 := uinport * dxport

t2 := 3 * xinport

t3 := 3 * yinport

t4 := t1 * t2

t5 := dxport * t3

t6 := uinport - t4

u_var := t6 - t5

y1 := u_var * dxport

y_var := yinport + y1

x_var := xinport + dxport

xoutport := x_var

youtport := y_var

uoutport := u_var

DFG Description of : ellip.vhdl

Input : sv39 sv38 sv33 sv26 sv18 sv13 sv2 inp

Output : sv39_o sv38_o sv33_o sv26_o sv18_o sv13_o sv2_o outp

InOut :

Preset Variables :

Preset Values :

Constants :

Constant Values :

Bit Width : 16

--

n1 := inp + sv2

n2 := sv33 + sv39

n3 := n1 + sv13

n4 := n3 + sv26

n5 := n4 + n2

n8 := n3 + n5

n9 := n5 + n2

n10 := n3 + n8

n11 := n8 + n5

n12 := n2 + n9

n15 := n1 + n10

n16 := n12 + sv39

n17 := n1 + n15

n18 := n15 + n8

n19 := n9 + n16

n20 := n16 + sv39

n22 := n18 + sv18

```
n23 := sv38 + n19
n25 := inp + n17
n28 := n22 + sv18
n29 := n23 + sv38
sv2_o := n25 + n15
sv13_o := n17 + n28
sv18_o := n28
sv26_o := n9 + n11
sv38_o := n29
sv33_o := n19 + n29
sv39_o := n16 + n20
outp := n20
```

```
DFG Description of : Arfilter
Input           : i0 i1 i2 i3 i4 i5 i6 i7 i8 i9 i10 i11 i12 i13 i14
Output          :
InOut           :
Preset Variables   :
Preset Values     :
Constants         :
Constant Values    :
Bit Width         : 16
--
m1 := i0 * i1
m2 := i2 * i3
m3 := i4 * i5
m4 := i6 * i7
m5 := i8 * i9
m6 := i10 * i11
m7 := i12 * i13
m8 :=  i14 * i15
a9 := m1 + m2
a10 := m3 + m4
a11 := m5 + m6
a12 := m7 + m8
a13 := a11 + i16
a14 := a12 + i17
m15 := a14 * i18
m16 := a13 * i19
m17 := a13 * i20
m18 := a14 * i21
a19 := m15 + m16
```

```
a20 := m17 + m18
m21 := a20 * i22
m22 := a19 * i23
m23 := a19 * i24
m24 := a20 * i25
a25 := m21 + m22
a26 := m23 + m24
a27 := a9 + a25
a28 := a10 + a26
```

DFG Description of : nestor.vhdl

Input : d7 d6 d5 d4 d3 d2 d1 d0

Output : p7 p6 p5 p4 p3 p2 p1 p0

InOut :

Preset Variables :

Preset Values :

Constants : 2

Constant Values : 2

Bit Width : 16

--

u12 := d2 + d5

u11 := d1 + d6

u13 := d3 + d4

u10 := d0 + d7

v12 := d2 - d5

v11 := d1 - d6

v13 := d3 - d4

v10 := d0 - d7

v21 := u12 - u11

v20 := u13 - u10

u21 := u11 + u12

u20 := u13 + u10

w210 := v21 + v20

w111 := v11 + v12

w110 := v10 + v13

w120 := w111 + w110

z3 := 2 * v21

```
z2 := 2 * w210
z4 := 2 * v20
z1 := 2 * u21
z0 := 2 * u20
z5 := 2 * v12
z6 := 2 * v12
z7 := 2 * w111
z8 := 2 * v11
z9 := 2 * v11
z10 := 2 * w120
z11 := 2 * v13
z12 := 2 * v13
z13 := 2 * w110
z14 := 2 * v10
z15 := 2 * v10
p6 := z2 + z3
p2 := z2 + z4
p0 := z1 + z0
p4 := z0 - z1
p5l := z5 + z7
p5r := z10 + z14
p5 := p5l + p5r
p3l := z7 + z8
p3r := z10 + z11
p3 := p3l + p3r
p7l := z9 + z10
p7r := z12 + z13
p7 := p7l + p7r
p1l := z6 + z10
```

p1r := z13 + z15

p1 := p1l + p1r