



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF MICROELECTRONICS

ÚSTAV MIKROELEKTRONIKY

SEMI-AUTOMATED DESIGN OF HIGH-PERFORMANCE DIGITAL CIRCUITS WITH XILINX FPGAS

POLOAUTOMATIZOVANÝ NÁVRH VYSOCE VÝKONNÝCH ČÍSLICOVÝCH OBVODŮ S XILINX FPGA

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR

AUTOR PRÁCE

Bc. David Houška

SUPERVISOR

VEDOUCÍ PRÁCE

Ing. Martin Štáva, Ph.D.

BRNO 2021

Master's Thesis

Master's study program **Microelectronics**

Department of Microelectronics

Student: Bc. David Houška

ID: 170858

**Year of
study:** 2

Academic year: 2020/21

TITLE OF THESIS:

Semi-automated Design of High-performance Digital Circuits with Xilinx FPGAs

INSTRUCTION:

Design a suitable representation of the delay tree compiled from the static timing analysis reports of Xilinx ISE/Vivado. Create a program/script for automatic delay optimization in FPGA-based sequential digital circuits developed under Xilinx ISE/Vivado. Verify the optimization method on selected circuits.

RECOMMENDED LITERATURE:

Podle pokynů vedoucího práce

**Date of project
specification:** 8.2.2021

Deadline for submission: 25.5.2021

Supervisor: Ing. Martin Štáva, Ph.D.

doc. Ing. Lukáš Fucík, Ph.D.
Chair of study program board

WARNING:

The author of the Master's Thesis claims that by creating this thesis he/she did not infringe the rights of third persons and the personal and/or property rights of third persons were not subjected to derogatory treatment. The author is fully aware of the legal consequences of an infringement of provisions as per Section 11 and following of Act No 121/2000 Coll. on copyright and rights related to copyright and on amendments to some other laws (the Copyright Act) in the wording of subsequent directives including the possible criminal consequences as resulting from provisions of Part 2, Chapter VI, Article 4 of Criminal Code 40/2009 Coll.

ABSTRACT

This master's thesis deals with sequential digital circuit design optimization concerning delay optimization. Two techniques commonly used for the optimization are described in the thesis – a brief description of the *retiming* technique and a more in-depth description of the *pipelining* technique. A form of abstraction of sequential digital circuits using Directed Acyclic Graphs (DAGs) was developed in the practical part of the thesis. This abstraction represents the circuit in a more manageable way for transformations. At the same time, a tool for semi-automatic digital circuit optimization using pipelining is introduced. This tool is compatible with Xilinx ISE Design Suite.

KEYWORDS

Delay optimization, FPGA, latency, pipelining, register balancing, sequential digital circuits, Xilinx ISE

ABSTRAKT

Tato diplomová práce se zabývá návrhem sekvenčních digitálních obvodů s ohledem na optimalizaci zpoždění. V práci je popsána problematika dvou technik, které jsou běžně používané při optimalizaci – stručně je popsána technika tzv. synchronizace registrů (angl. *retiming*), větší pozornost je však věnována technice tzv. zřetězení (angl. *pipelining*). V rámci praktické části byla vypracována forma abstrakce sekvenčních digitálních obvodů pomocí acyklických orientovaných grafů. Obvod je tak přenesen do roviny, ve které je jednodušší jej transformovat. Zároveň je představen nástroj pro poloautomatickou optimalizaci digitálních obvodů vyvíjených v prostředí Xilinx ISE Design Suite využitím techniky zřetězení.

KLÍČOVÁ SLOVA

Balancování registrů, FPGA, latence, optimalizace zpoždění, sekvenční digitální obvody, Xilinx ISE, zřetězení

ROZŠÍŘENÝ ABSTRAKT

V oblasti návrhu číslicových obvodů zůstává nepříjemným trnem v oku většiny hardwarových návrhářů splnění časových požadavků. I přes mnoho desetiletí intenzivního výzkumu a významného vlivu komerční sféry v automatizaci návrhů zůstává mnoho aspektů, které je nutné při návrhu stále ladit ručně tak, aby byly tyto požadavky splněny.

Jedním z takových aspektů je tzv. zřetězení (angl. *pipelining*), které se dá stručně popsat jako implementace řady operací, při kterých je více výpočtů paralelizováno. Zřetězení je v oblasti návrhu digitálních obvodů velmi hojně využíváno, protože umožňuje vyřešit problémy s časováním bez funkčního zásahu do výpočetních algoritmů nebo hardwarové mikroarchitektury, a protože může snížit celkovou plochu návrhu v porovnání s bezkompromisní paralelizací obvodu.

Technika zřetězení je tak jedním z nejúčinnějších způsobů, kterých lze využít ke zvýšení efektivity výkonnostních aplikací programovatelných hradlových polí (FPGA, angl. *Field Programmable Gate Array*). Ruční implementace zřetězení však často vyžaduje značné úsilí ze strany návrhářů, kteří musí prozkoumat a ověřit různé zásahy do meziregistrových přenosů (RTL, angl. *Register Transfer Logic*). K dosažení žádaných výsledků je obvykle nutný iterativní přístup a v bezesporu drtivě většině případů trpělivá a soustředěná mysl.

V teoretické části práce je stručně popsána technika tzv. synchronizace registrů (angl. *retiming*), větší pozornost je však věnována právě technice zřetězení. Algoritmů využívaných při řetězení obvodu je nepřehledné množství, v práci jsou tak stručně charakterizovány pouze reprezentativní příklady. Zvláštní pozornost je pak věnována technikám řetězení obvodů se zpětnou vazbou – tato problematika je velmi aktuální a je předmětem intenzivního výzkumu.

V rámci první sekce praktické části byla vypracována forma abstrakce obvodu pomocí acyklických orientovaných grafů (DAGs, angl. *Directed Acyclic Graphs*). Obvod je tak přenesen do roviny, ve které je jednodušší jej řetězit (a obecně transformovat i jinými způsoby). Takový obvod sestávající z logických elementů (registrů, multiplexorů, vyhledávacích tabulek [LUTs, angl. *Lookup Tables*], atp.) a uzlů propojujících tyto elementy je tak možné modelovat právě pomocí orientovaného grafu $G = \langle V, E \rangle$ tvořeného vrcholy V a hranami E . Jednotlivé vrcholy $v \in V$ grafu reprezentují fyzické piny logického elementu, hrany $e \in E$ poté propojují tyto piny. Všechny hrany jsou rozděleny do dvou kategorií: buďto vnitřní, které propojují piny uvnitř logického elementu (vstupní pin na výstupní pin), anebo vnější, které propojují piny jednotlivých logických elementů (výstupní pin jednoho logického elementu na vstupní pin druhého). Každá hrana je charakterizována vahou w , která vystihuje propagační zpoždění spoje. Výsledkem je tak graf $G = \langle V, E, w \rangle$, který z časového hlediska plnohodnotně vystihuje fyzický obvod.

Aby bylo možné obvod převést do abstraktní roviny, je nejprve nutné získat data o jednotlivých propojích v obvodu uložených v netlistu. Této problematice se věnuje druhá sekce praktické části. Xilinx ISE Design Suite ukládá netlist do souboru v enkódované podobě, je tak nutné nejprve soubor převést do čitelného formátu, a to sice pomocí skriptovacího jazyka TCL (angl. *Tool Command Language*). Získáním čitelné podoby netlistu lze extrahovat data o jednotlivých logických elementech a propojích obvodu s využitím regulárních výrazů. Pomocí jednotlivých regulárních výrazů jsou tak extrahovány jednotlivé logické buňky, jejich instance, a jednotlivé uzly propojující tyto instance. Každá instance logického elementu nese informaci o buňce, kterou je tvořena, a každá z těchto buněk nese informaci o vstupních a výstupních pinech. Určité piny a cesty lze ignorovat a nepodrobovat je tak zřetězení – lze jmenovat například hodinové cesty, ze strany pinů pak zejména hodinový pin klopného obvodu typu D. Efektivně jsou tak k dispozici veškeré informace nutné k sestavení náhradního obvodu.

Dále jsou v praktické části extrahována propagační zpoždění jednotlivých cest ze zpráv statické časové analýzy (angl. *timing reports*). Tuto analýzu lze provést po syntéze obvodu, mnohem přesnějších výsledků je však dosaženo po provedení analýzy až po umístění logických elementů na konkrétní čip (angl. *post Place and Route*). Tuto analýzu lze vhodně nastavit tak, aby bylo vypsáno dostatečné množství kritických cest (tedy cest s největším propagačním zpožděním). Tyto cesty jsou řetězeny prioritně, protože mají největší vliv na výsledné zvýšení rychlosti obvodu. Rovněž je však potřeba brát v úvahu zdroj a cíl cesty – všechny cesty, které nevedou mezi registry, jsou ignorovány.

S informacemi o netlistu a propagačních zpožděních nejkritičtějších cest je možné sestavit abstraktní reprezentaci obvodu. Sestavený obvod je následně možné řetězit, čemuž se věnuje třetí sekce praktické části. Vstupním argumentem řetězicího algoritmu je seznam všech hran $e \in E$ seřazených sestupně od nejkritičtější po nejméně kritickou. Samotný algoritmus je upravenou formou algoritmu implementovaného ve vývojovém prostředí Xilinx Vivado Design Suite, verze 2016.1. Tento algoritmus je velmi škálovatelný s lineární časovou složitostí $O(n)$, což jej činí velmi použitelným.

Výstupními daty celého procesu jsou pozice (hrany) v obvodu, na které je nutné vložit řetězicí registry tak, aby byla zachována koherence dat. Proces je verifikován na obvodu bez zpětnovazebních cest, jak je ukázáno v rámci poslední sekce praktické části.

Přestože byla práce koncipována jako doplněk k vývojovému prostředí Xilinx ISE Design Suite, byl navržen s dostatečnou flexibilitou. Po mírné úpravě jej tak lze aplikovat i na návrhy vyvíjené v jiných prostředích.

HOUŠKA, David. *Semi-automated Design of High-performance Digital Circuits with Xilinx FPGAs*. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Microelectronics, 2021, 66 p. Master's Thesis. Advised by Ing. Martin Štáva, Ph.D.

Author's Declaration

Author: Bc. David Houška
Author's ID: 170858
Paper type: Master's Thesis
Academic year: 2020/21
Topic: Semi-automated Design of High-performance
Digital Circuits with Xilinx FPGAs

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the paper and listed in the comprehensive bibliography at the end of the paper.

As the author, I furthermore declare that, with respect to the creation of this paper, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll. of the Czech Republic, Section 2, Head VI, Part 4.

Brno
author's signature*

*The author signs only in the printed version.

Contents

| | |
|---|-----------|
| Introduction | 14 |
| 1 Preliminaries | 15 |
| 1.1 Time and Space Complexity | 15 |
| 1.2 Directed Graphs | 17 |
| 1.3 Primary I/O, Fanin, Fanout | 19 |
| 2 Retiming and Pipelining | 20 |
| 2.1 Retiming | 20 |
| 2.1.1 Minimum Clock Period Retiming | 21 |
| 2.1.2 Minimum Register Retiming | 23 |
| 2.2 Pipelining | 24 |
| 2.2.1 Pipelining in ISA Design | 24 |
| 2.2.2 Pipelining in ASIC Design | 25 |
| 2.2.3 Power-Oriented Pipelining | 27 |
| 2.2.4 Feedforward Circuits Pipelining | 29 |
| 2.2.5 Looped Circuits Pipelining | 31 |
| 3 Circuit Abstraction | 35 |
| 3.1 Vertices and Edges | 35 |
| 3.2 Paths | 36 |
| 4 Data Extraction, Graph Assembly, and Pipelining Algorithms | 38 |
| 4.1 Data Extraction | 39 |
| 4.1.1 Timing Reports Data Extraction | 39 |
| 4.1.2 Netlist Extraction | 43 |
| 4.2 Circuit Representation Assembly | 46 |
| 4.2.1 Internal Edges Connection | 48 |
| 4.2.2 Reference Cell Data Copy | 49 |
| 4.2.3 External Edges Connection | 49 |
| 4.2.4 Propagation Delay and Circuit Assembly | 50 |
| 4.3 Pipelining Implementation | 52 |
| 4.3.1 Legal Edge List Compilation | 52 |
| 4.3.2 Pipelining Algorithm | 54 |
| 5 Experimental Results | 55 |
| Summary and Reflection | 57 |

| | |
|-----------------------------------|----|
| Conclusion | 58 |
| Bibliography | 59 |
| Symbols and abbreviations | 62 |
| A Schematics and waveforms | 63 |
| B List of Used Python Libraries | 65 |
| C Content of Enclosed Data Medium | 66 |

List of Figures

| | | |
|------|---|----|
| 1.1 | Graphs of functions commonly used in the analysis of algorithms . . . | 16 |
| 1.2 | A digraph | 17 |
| 1.3 | A weighted acyclic digraph | 18 |
| 1.4 | Transitive fanin and transitive fanout | 19 |
| 2.1 | Retiming example | 20 |
| 2.2 | Modeling two-way fanout with an extra vertex having zero delay . . . | 23 |
| 2.3 | Pipelining example | 24 |
| 2.4 | Timing requirements | 25 |
| 2.5 | Relationship between pipeline depth and area cost | 26 |
| 2.6 | A one-stage array multiplier | 28 |
| 2.7 | A four-stage array multiplier | 29 |
| 2.8 | Performance of the extra simple cut pipelining method | 30 |
| 2.9 | Sequential loop pipelining | 31 |
| 2.10 | Loop splitting example | 32 |
| 2.11 | Associative refactoring example | 32 |
| 2.12 | Associative refactoring of a function | 33 |
| 2.13 | Adder chain pipelined by the combining technique | 33 |
| 2.14 | Pipelining a loop inside an FSM by changing a condition | 34 |
| 3.1 | Abstract representation of an instance | 36 |
| 3.2 | Instances connected by external edges | 36 |
| 3.3 | A small circuit and its abstraction into a directed graph | 37 |
| 4.1 | Process of adding a new pipeline stage | 38 |
| 4.2 | Netlist extraction process | 44 |
| 4.3 | Circuit building algorithms and netlist extraction algorithms | 47 |
| 4.4 | Absolute difference between ideal half-cut delay and gradual delay . . | 53 |
| 5.1 | Simple design schematic | 55 |
| 5.2 | Execution time as a function of number of LUTs in the design | 56 |
| A.1 | Pipelined 8-bit multiplier | 63 |
| A.2 | Waveforms of a simple circuit before and after pipelining | 64 |

List of Tables

| | | |
|-----|--|----|
| 1.1 | Common time complexities | 16 |
| 1.2 | The adjacency matrix for DAG from Figure 1.3 | 18 |
| 1.3 | The incidence matrix for DAG from Figure 1.3 | 18 |
| 2.1 | Comparison of retiming algorithms | 22 |
| 5.1 | Pipelining results | 56 |
| B.1 | List of used Python libraries | 65 |

List of Algorithms

| | | |
|------|--|----|
| 2.1 | Minimum period retiming | 22 |
| 2.2 | Half-cut pipelining procedure | 26 |
| 2.3 | Search half point in critical delay path | 27 |
| 2.4 | Simple cut pipelining procedure | 30 |
| 4.1 | Synthesis timing reports data extraction | 40 |
| 4.2 | Place and Route timing reports data extraction | 42 |
| 4.3 | Cell data extraction | 45 |
| 4.4 | Instance data extraction | 45 |
| 4.5 | Net data extraction | 46 |
| 4.6 | Connecting internal edges | 48 |
| 4.7 | Copying cell reference data | 49 |
| 4.8 | Connecting external edges | 50 |
| 4.9 | DAG assembly | 51 |
| 4.10 | Adding delays to DAG edges | 51 |
| 4.11 | Calculate ideal half-cut position | 53 |
| 4.12 | Finalize legal edge list | 54 |
| 4.13 | Computing a pipeline stage | 54 |

Listings

| | | |
|-----|--|----|
| 4.1 | An example of Xilinx ISE Synthesis timing results. | 40 |
| 4.2 | An example of Xilinx ISE Place and Route timing results. | 41 |
| 4.3 | Xilinx ISE Place and Route timing results encoded in HTML. | 42 |
| 4.4 | Syntax used in edf netlist files. | 44 |

Introduction

Meeting timing requirements is one of the perennial challenges in hardware design. Despite many decades of research and a significant commercial presence in the design automation, many aspects of circuit design are still done manually.

One example is pipelining, which refers to implementing a series of operations where multiple computations are parallelized during execution. Pipelining is particularly important because it can allow timing problems to be resolved without changing the underlying algorithm or hardware micro-architecture and because it can involve less additional hardware than pure parallelization.

Pipelining is one of the most effective ways to improve FPGA (Field Programmable Gate Array) performance applications. However, manual pipelining often requires significant efforts from FPGA designers to explore various changes in the RTL (Register Transfer Logic) and re-run the flow iteratively to achieve the ideal results.

This thesis deals with implementing pipelining algorithms in Xilinx ISE Design Suite for FPGAs where no register balancing techniques are available except for retiming. Adding pipelining stages has to be done manually and, more importantly, designers have to choose which paths to pipeline manually. In this regard, the thesis aims to achieve a semi-automated pipelining process in Xilinx ISE Design Suite. This is done by extracting netlist data, modelling an abstract circuit with Directed Acyclic Graphs (DAGs), reading timing reports provided by ISE Design Suite and finally executing pipelining algorithms that show the designer which paths need to be pipelined and where.

Furthermore, designs with feedback loops are particularly hard to pipeline, and the problematics are only very recently getting untangled. For this reason, the thesis also describes some of the novel approaches that tackle the issue of feedback loops pipelining.

1 Preliminaries

This chapter focuses mainly on time and space complexity and directed graphs. Some of the definitions used later in the thesis are described here to understand the problematics deeper.

1.1 Time and Space Complexity

In computer science, the *time complexity* (or *running time*) describes the amount of computing time it takes to run an algorithm. The complexity of algorithms in terms of space (memory) required is called *space complexity*. The performance of an algorithm is assessed based on time and space complexity when the size of the input is changed. In other words, both complexities are functions of the size of the input [1].

It is usually assumed that computations are performed by a Turing machine (i.e. some abstract computer) to make the time complexity or the space complexity measurement sufficiently universal [1]. Both time complexity and space complexity do not measure actual quantities; they are merely abstract representations.

Algorithm's running time or space requirements may vary among different inputs of the same size. Therefore, three analysis forms are defined [1]:

- *worst-case analysis* where the longest-running time or the largest space of all inputs of a particular length is considered,
- *average-case analysis* where the average of all the running times or space requirements of inputs of a particular length is considered, and
- *best-case analysis* where the best result is considered.

The worst-case analysis is usually utilized.

The exact running time or space requirement is generally difficult to compute precisely. Therefore, it is usually estimated [1]. One form of this estimation is called *asymptotic analysis* where the time or space complexity of the algorithm is calculated on large inputs [1]. This is done by considering only the highest order term of the expression. Therefore, the coefficient of that term and any lower-order terms are disregarded. For example, the function $f(n) = 6n^3 + 2n^2 + 20n + 45$ has four terms with the highest order term $6n^3$. Disregarding the coefficient 6, the function $f(n)$ is asymptotically at most n^3 [1]. Table 1.1 summarizes some commonly encountered time complexities. Graphs of the functions are shown in Figure 1.1.

The *asymptotic* notation (often called the *big-O* notation) for the particular example above is $f(n) = O(n^3)$. Meaning, the specific complexity is $O(n^3)$ for the length of input n . Note that for the best-case analysis, the *Omega* notation $\Omega(f(n))$ is sometimes used instead of the $O(f(n))$ notation. Arithmetic operations can be

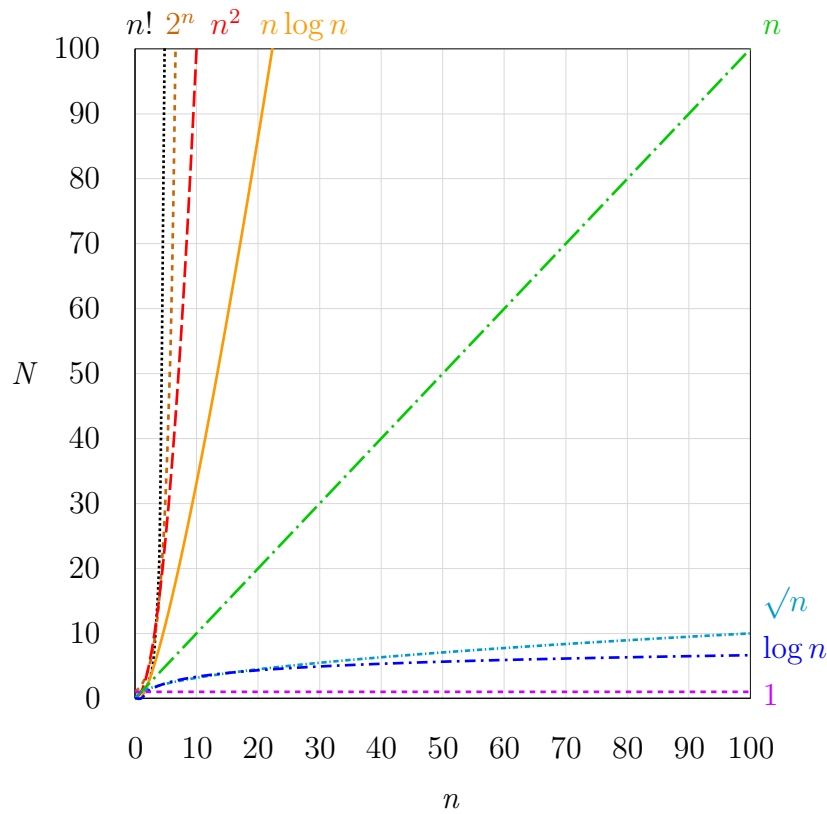


Figure 1.1: Graphs of functions commonly used in the analysis of algorithms; showing the number of operations N versus input size n for each function [2].

Table 1.1: Common time complexities [2].

| Name | Notation | Example algorithms |
|-------------------|---------------|--|
| Constant time | $O(1)$ | Finding the median value in a sorted array of numbers |
| Logarithmic time | $O(\log n)$ | Binary search |
| Linear time | $O(n)$ | Linear search |
| Linearithmic time | $O(n \log n)$ | Fast Fourier transform |
| Exponential time | $O(2^n)$ | Solving the matrix chain multiplication via brute-force search |
| Quadratic time | $O(n^2)$ | Bubble sort |
| Cubic time | $O(n^3)$ | Naive multiplication of two $n \times n$ matrices |
| Factorial time | $O(n!)$ | Solving the traveling salesman problem via brute-force search |

done on big-O notations, for example $f(n) = O(n^2) + O(n)$. In this case, the $O(n^2)$ term dominates the $O(n)$ term, and the expression is equivalent to $f(n) = O(n^2)$ [1].

While the big-O notation says that one function is asymptotically no more than another, the *small-o* notation says that one function is asymptotically less than another [1]. The difference between the big-O and small-o notations is analogous to the difference between \leq and $<$.

The exponential time complexity typically arise when using *brute-force search* to solve problems, as demonstrated in Table 1.1 [1]. Sometimes the brute-force search can be avoided through a deeper understanding of the problem and redesign of the algorithm. Therefore, the algorithm's time complexity can be transformed from exponential to polynomial (where polynomial time is denoted as $O(n^k)$ for $k > 0$). This borders with the well-known *P-hard* versus *NP-hard* problem, which is beyond the scope of this work.

1.2 Directed Graphs

In graph theory, a *directed graph* (abbreviated a *digraph*) is a graph $G = \langle V, E \rangle$ connecting a set of vertices V by edges E , where the edges have a direction associated with them [3]. The vertices $(u, v) \in V$ are adjacent if there exists an edge $e \in E$ between them. Henceforth, this edge will be notated as $u \xrightarrow{e} v$, where the first vertex u is the *tail* of the edge and the second vertex v is the *head* of the edge. An example of a directed graph is shown in Figure 1.2.

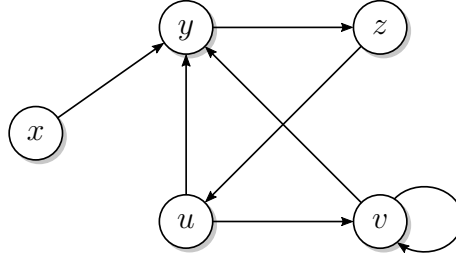


Figure 1.2: A digraph.

An *edge-weighted* digraph is a digraph G with a mapping $w : E(G) \rightarrow \mathbb{R}$ [3]. An edge-weighted digraph is thus $G = \langle V, E, w \rangle$. A *vertex-weighted* digraph is a digraph G with a mapping $d : V(G) \rightarrow \mathbb{R}$ and thus $G = \langle V, E, d \rangle$ [3].

For a vertex $v \in V$, the *out-degree* is the number of edges (except for loops) with tail v . Similarly, the *in-degree* is the number of edges (except for loops) with head v . For a digraph, the out-degree of a vertex equals the number of *out-neighbours* of this vertex [3]. Analogously, the in-degree of a vertex equals the number of *in-neighbours* of the vertex. If loops are counted as well, then the number of all edges with tail v

is called the *out-pseudodegree*, and the number of all edges with head v is called the *in-pseudodegree* [3].

A digraph G is *acyclic* if it has no cycles (loops). Acyclic digraphs (DAGs for short) are a well-explored family of digraphs because every DAG has a vertex of in-degree zero as well as a vertex of out-degree zero (i.e. it has at least one vertex with zero heads and at least one vertex with zero tails) [3]. An example of an edge-weighted and vertex-weighted DAG $G = \langle V, E, d, w \rangle$ is shown in Figure 1.3. By contrast, if a digraph has cycles (loops) as is shown in Figure 1.2, it is called a *directed pseudograph* [3].

Digraphs are very often represented with an *adjacency matrix*. For a digraph $G = \langle V, E \rangle$ with vertices labeled as v_1, v_2, \dots, v_n , the adjacency matrix $M(G) = [m_{ij}]$ of the digraph G is an $n \times n$ matrix such that $m_{ij} = 1$ if there exists an edge between vertices $v_i \rightarrow v_j$ and $m_{ij} = 0$ otherwise [3]. As an example, Table 1.2 lists the incidence matrix for the digraph shown in Figure 1.3. This representation is a fast and convenient tool for verifying whether two vertices are connected with an edge. On the other hand, the time complexity of an algorithm verifying all adjacencies is $\Omega(n^2)$ at best [3]. This means that most algorithms using the adjacency matrix cannot have a time complexity lower than $O(n^2)$.

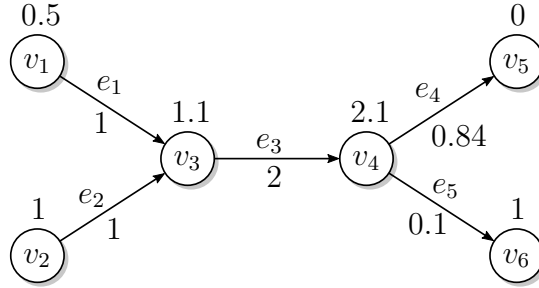


Figure 1.3: A weighted acyclic digraph.

Table 1.2: The adjacency matrix for DAG from Figure 1.3.

| | v_1 | v_2 | v_3 | v_4 | v_5 | v_6 |
|-------|-------|-------|-------|-------|-------|-------|
| v_1 | 0 | 0 | 1 | 0 | 0 | 0 |
| v_2 | 0 | 0 | 1 | 0 | 0 | 0 |
| v_3 | 0 | 0 | 0 | 1 | 0 | 0 |
| v_4 | 0 | 0 | 0 | 0 | 1 | 1 |
| v_5 | 0 | 0 | 0 | 0 | 0 | 0 |
| v_6 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 1.3: The incidence matrix for DAG from Figure 1.3.

| | e_1 | e_2 | e_3 | e_4 | e_5 |
|-------|-------|-------|-------|-------|-------|
| v_1 | 1 | 0 | 0 | 0 | 0 |
| v_2 | 0 | 1 | 0 | 0 | 0 |
| v_3 | -1 | -1 | 1 | 0 | 0 |
| v_4 | 0 | 0 | -1 | 1 | 1 |
| v_5 | 0 | 0 | 0 | -1 | 0 |
| v_6 | 0 | 0 | 0 | 0 | -1 |

An alternative to the adjacency matrix is an *incidence matrix*. The rows of the incidence matrix $S(G) = [s_{ij}]$ of a digraph $G = \langle V, E \rangle$ are labelled by the vertices of V and columns are labelled by the edges of E . Then, the entry s_{v_i, e_j} equals 1 if the edge e_j has tail v_i , -1 if e_j has head v_i , and 0 otherwise [3]. As an example, Table 1.3 lists the incidence matrix for the digraph shown in Figure 1.3.

1.3 Primary I/O, Fanin, Fanout

The vertices of in-degree zero (i.e. sources) of a DAG are called its *primary inputs* (PIs), and the vertices of out-degree zero (i.e. sinks) are called its *primary outputs* (POs) [4].

An edge $u \xrightarrow{e} v$ has its tail u and its head v , as already mentioned earlier. The tail is referred to as a *fanout* in digital electronics and the head as a *fanin* [4].

If there is a path from vertex u to v , then u is in the *transitive fanin* (TFI) of v and v is in the *transitive fanout* (TFO) of u [4]. The TFI of a vertex u includes vertex u and its TFI, including the PIs. Similarly, the TFO of a vertex u includes vertex u and its TFO, including the POs. A TFI and TFO example is shown in Figure 1.4.

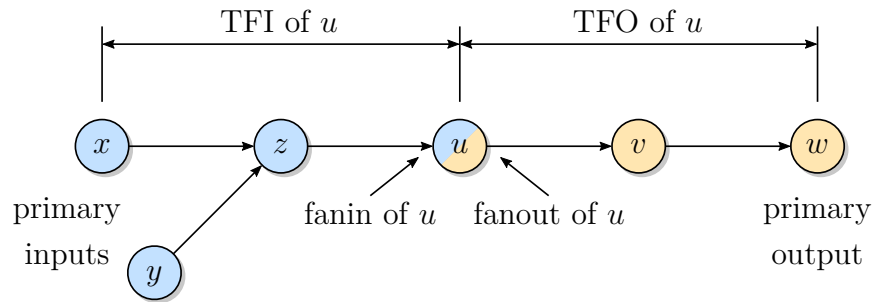


Figure 1.4: Transitive fanin and transitive fanout.

2 Retiming and Pipelining

This chapter of the thesis introduces register balancing techniques, including register retiming and pipelining and several standard algorithms used for optimal technique implementation.

2.1 Retiming

Retiming was proposed in 1983 [5] and further developed in 1991 [6] by Leiserson and Saxes as a technique for optimizing sequential circuits. Generally, a retiming of a circuit can be viewed as inserting and deleting registers, effectively relocating them, and leaving the combinatorial cells untouched, as is shown in Figure 2.1. Retiming does not increase the circuit latency or change the functionality of the circuit [5]. The technique of retiming can be used to optimize a circuit for one of several objective functions [7,8]: either a minimal clock period (as is shown in Figure 2.1), a minimal number of registers, or a combination of the two (i.e. minimization of the number of registers subject to a maximum constraint on the clock period).

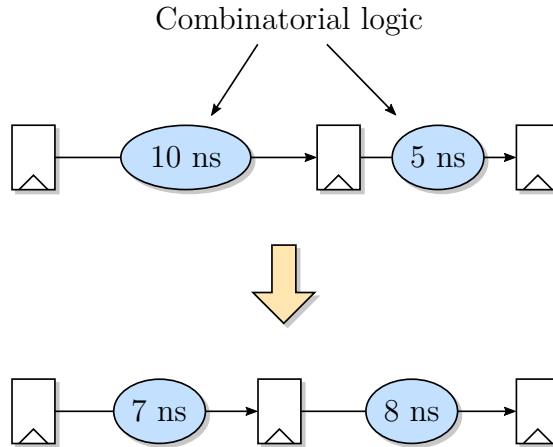


Figure 2.1: Retiming example.

Leiserson and Saxes show that the problem of finding a retiming of a circuit can be modelled as a transformation of a finite, vertex-weighted and edge-weighted $G = \langle V, E, d, w \rangle$ DAG [6]. In this case, the vertices V represent *asynchronous* combinatorial blocks and the directed edges E represent a series of *registers* or *latches*. Each vertex $v \in V$ is weighted with its numerical propagation delay $d(v)$ and each edge $e \in E$ is weighted with a register count $w(e)$ (i.e. the number of registers along with the connection).

A path p in a digraph G is an alternating sequence of vertices and edges. The *path weight* for a path $p = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} \dots \xrightarrow{e_{k-1}} v_k$ is defined as the sum of the weights of the edges of the path [6]:

$$w(p) = \sum_{i=0}^{k-1} w(e_i). \quad (2.1)$$

Similarly, the *path delay* for a path $p = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} \dots \xrightarrow{e_{k-1}} v_k$ is defined as the sum of the delays of the vertices of the path [6]:

$$d(p) = \sum_{i=0}^k d(v_i). \quad (2.2)$$

For a digraph G , the *minimum clock period* $\Phi(G)$ is defined as the maximum amount of propagation delay through which any signal must ripple between clock ticks, formally represented by the equation [6]:

$$\Phi(G) = \max\{d(p) : w(p) = 0\}. \quad (2.3)$$

Then, retiming $r : V \rightarrow \mathbb{Z}$ is defined as changing the graph G into a new graph $G_r = \langle V, E, d, w_r \rangle$ where the edge weighting w_r is defined for an edge $u \xrightarrow{e} v$ by the equation [6]:

$$w_r(e) = w(e) + r(v) - r(u). \quad (2.4)$$

This can be viewed as adding and removing registers. When a circuit G is retimed to produce a new circuit G_r , the new circuit is functionally equivalent. A proof can be found in [5].

2.1.1 Minimum Clock Period Retiming

Several algorithms for retiming a circuit were proposed by Leiserson and Saxes to minimize either the clock period $\Phi(G_r)$ of the retimed circuit G_r or the number of registers and fanout [6]. The most efficient algorithm for minimal clock period proposed by Leiserson and Saxes is shown in Algorithm 2.1.

However, the algorithms proposed by Leiserson and Saxes have polynomial complexity with $O(n^3)$ time complexity and $O(n^2)$ space complexity [6]. This renders them ineffective for retiming circuitry with more than roughly 500 combinatorial cells. Shenoy and Rudell addressed this issue in [3, 7]. Their novel method can be applied to circuits with as many as 10,000 combinatorial cells. Shenoy and Rudell proposed that the retiming algorithm can be accelerated using the Bellman-Ford algorithm and detecting if a clock period is feasible or not before completing the

```

Input: A circuit  $G = \langle V, E, d, w \rangle$  and a desired clock period  $c$ 
Output: A retimed circuit  $G_r$  with clock period  $\Phi(G_r) \leq c$ , if such a retiming
exists

1 foreach  $v \in V$  do
2   | set  $r(v) = 0$ 
3 end
4 for  $|V| - 1$  times do
5   | Compute retimed edge weights (Equation 2.4)
6   | Compute largest delay  $\Delta(v)$  for all  $v \in V$ 
7   | foreach  $v \in V$ , such that  $\Delta(v) > c$  do
8     |  $r(v)++$ 
9   | end
10 end
11 Compute retimed edge weights (Equation 2.4)
12 if  $\max_{v \in V} \{\Delta(v)\} > c$  then
13   | No feasible retiming
14 else
15   | The current  $r$  yields a legal retiming
16 end

```

Algorithm 2.1: Minimum period retiming as proposed by Leiserson and Saxes [6].

requisite $|V| - 1$ iterations [7]. Experimental comparison of Shenoy's and Rudell's algorithm to Leiserson's and Saxes's algorithm is shown in Table 2.1; for more detailed information on how the algorithm works, the reader can refer to [7].

Table 2.1: Comparison of retiming algorithms: Leiserson's and Saxes's (Original), and Shenoy's and Rudell's (New) [7].

| Number of gates | CPU (in seconds) | | Clock period | |
|-----------------|------------------|-------|--------------|-------|
| | Original | New | Before | After |
| 386 | 96.2 | 2.1 | 17.4 | 17.4 |
| 384 | 101.0 | 3.9 | 36.6 | 31.4 |
| 887 | 527.6 | 13.7 | 11.9 | 10.4 |
| 1,107 | 159.8 | 13.8 | 19.9 | 12.9 |
| 1,854 | 1,973.2 | 28.8 | 23.2 | 21.4 |
| 2,240 | 2,856.1 | 37.4 | 40.1 | 22.9 |
| 7,882 | 39,025.9 | 306.2 | 35.5 | 34.1 |

2.1.2 Minimum Register Retiming

Leiserson and Saxes showed that retiming can also be solved in polynomial time for reducing the total number of registers [6]. Minimizing the number of registers can save area and power. Moreover, the minimum register retiming reduces the number of state variables in verification [9].

For a given circuit $G = \langle V, E, d, w \rangle$, this can be done by determining a retiming r such that the total state $S(G_r) = \sum_{e \in E} w_r(e)$ of the retimed circuit G_r is minimized [6]. This *state-minimization* problem may or may not be bound on the clock period c so that, for example, the total state of a circuit can be minimized after the defined clock period c was achieved. The *minimum-cost flow* algorithm can be used to solve the minimum register retiming in both cases [6]. The reader may want to see how these algorithms operate in polynomial time in [10].

It is advantageous to add a single register along a one-bit wide path than adding multiple registers along a multi-bit wide data bus. This can be modelled by assigning a breadth $\beta(e)$ to each edge e proportional to the cost of adding a register along e [6]. Suppose a signal fanning out to several functional elements. In that case, it is better to add a single register before the path forks rather than adding multiple registers after the path forks. Leiserson and Saxes dealt with this problem by introducing a dummy vertex \hat{u} with zero propagation delay [6]. This vertex then models the fork of the interconnection, as is shown in Figure 2.2.

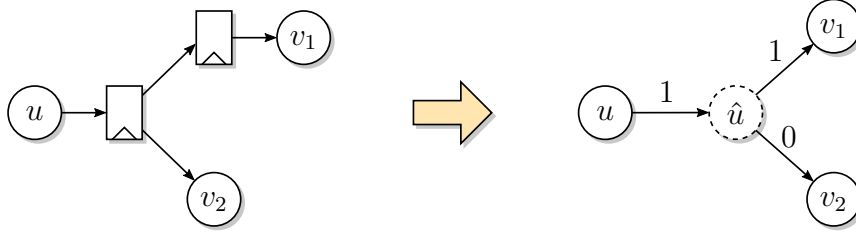


Figure 2.2: Modeling two-way fanout with an extra vertex having zero delay [6].

Hurst et al. proposed a novel algorithm for the minimum register retiming via *binary maximum-flow* [8]. Maximum-flow is a practically simpler problem to solve than minimum-flow. In fact, the worst-case bound of maximum-flow is asymptotically faster than the best known bound on the minimum-cost flow problem [8]. In their proposal, Hurst et al. construct a residual graph from the flow problem where the flow source is connected to all register outputs, and the register inputs are connected to the flow sink. This graph is then used to generate a corresponding minimum cut which is equivalent to finding a solution of minimizing the number of registers. This problem is beyond the scope of this work. Therefore, the reader can refer to [8] for further details.

2.2 Pipelining

Pipelining is one of the most effective (and therefore profusely used) technique to improve the performance of FPGAs, ASICs, and other digital circuits (MCUs, CPUs, etc.). It improves the maximal achievable frequency by introducing registers on the most critical paths and slicing them into several shorter paths. However, to preserve the consistency and coherency of the signals, a whole stage must be added by introducing registers in the non-critical paths. Figure 2.3 shows the pipelining of a simple circuit which effectively doubles the operational frequency by adding a register into the critical path from LUT A to LUT C. Consistency is preserved by adding an extra register in the non-critical path from LUT B to LUT C.

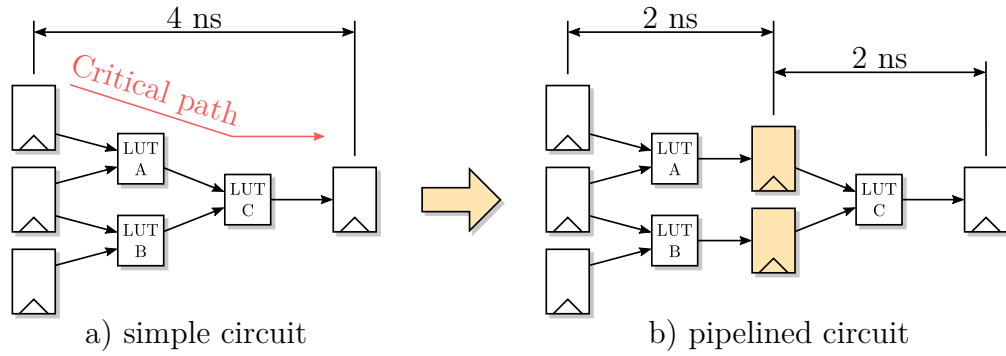


Figure 2.3: Pipelining example [11].

A fundamental relation underlying the speed at which a design can be run is described as [12]:

$$t_{cp} \geq t_{c2q} + t_{cd} + t_s + t_{cu}, \quad (2.5)$$

where t_{cp} is the clock period, t_{c2q} represents the clock-to-output delay of a register, t_{cd} is the delay of combinatorial logic between registers, t_s is the setup time for a register, and t_{cu} represents the clock skew and jitter (clock uncertainty). A graphic representation is shown in Figure 2.4. Typically, t_{c2q} , t_s , and t_{cu} are parameters given by fabrication technology. Therefore, t_{cd} is the only factor designers can adjust [12]. By splitting the combinatorial path in half, as is demonstrated in Figure 2.3, t_{cd} is lowered, effectively increasing the maximal operating frequency.

2.2.1 Pipelining in ISA Design

Several methods for automatic pipelining have been developed over the years. Most methods dealing with Instruction Set Architecture (ISA; as opposed to datapath-oriented designs, including most of FPGA's applications) are focusing on maximizing

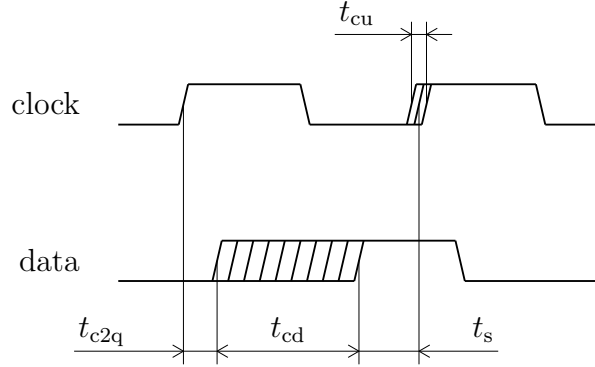


Figure 2.4: Timing requirements [13].

performance as measured in terms of instructions per clock [12]. For example, Marinescu and Rinard described an algorithm that repeatedly determines the critical path, chooses an operator or function on it, and moves the chosen operator or function to a newly created stage that is then connected to the original circuit [14]. Galceran-Oms et al. described an iterative approach [15]:

1. One or more bypasses are inserted into a design.
2. Retiming of the design with the added bypass is performed.
3. Overall performance improvement with the added pipeline stage is checked.
4. If improved, the iteration is either repeated or stopped.

More ISA pipelining methods can be found in [16, 17].

2.2.2 Pipelining in ASIC Design

Pipelining can also be used to optimize area cost while maintaining the required timing. This approach is very desirable in Application Specific Integrated Circuit (ASIC) design where the cost of an integrated circuit rises with every logic element added. In ASIC design, cells driving larger load capacitances (i.e. higher fanout) need a larger area for placement [12]. One can use modern RTL synthesis tools to optimize the design to meet timing requirements and to optimize area. However, when the clocking requirements become tight enough, the synthesis tools cannot successfully find a design meeting the constraints [12]. This can be addressed using pipelining. However, this introduces another problem. In FPGAs, there is a fixed amount of Lookup Tables (LUTs) and flip-flops, and therefore the area is fixed. By contrast, adding new flip-flops in ASIC design increases the area cost, and so over-pipelining an ASIC design (as shown in Figure 2.5) leads to excessive cost with little to no benefit added.

One of the heuristic approaches proposed by Kim et al. [12] is based on cuts, i.e. splitting the DAG that represents the circuitry into two. At each iteration of the

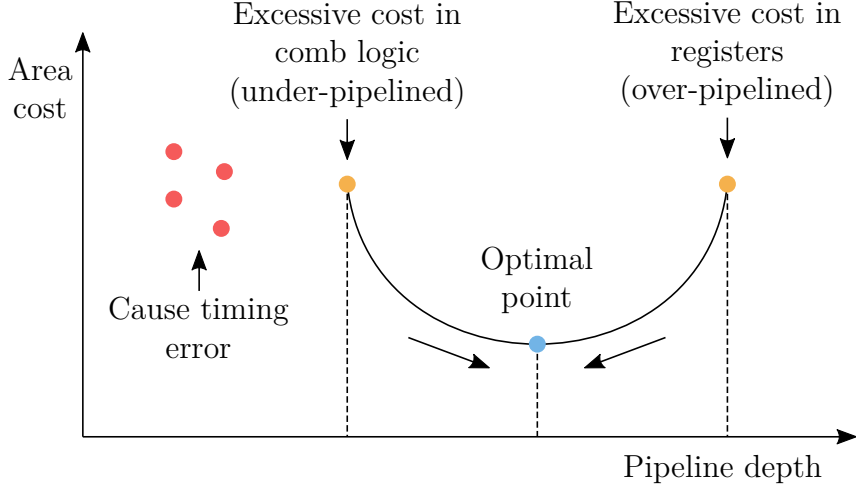


Figure 2.5: Relationship between pipeline depth and area cost [12].

algorithm, the current critical path is calculated and cut roughly at its mid-point delay. All paths parallel to the critical path also need to be inserted in the cut to maintain data coherency. Algorithm 2.2 summarizes the pipelining procedure of the proposed method. Algorithm 2.3 tracks all logic elements in the critical delay path p_c and adds the delay of each element d to the accumulated delay d_{acc} .

| | |
|--|--|
| Input: A DAG representing a circuit | |
| Output: The pipelined design | |
| 1 | Generate an RTL file from current DAG |
| 2 | Synthesize the RTL with the user specified clock constraint and target cell library |
| 3 | Analyze synthesis results to find the most critical delay path and extract timing and area information |
| 4 | if no timing error then |
| 5 | if area cost < current minimum area then |
| 6 | Save the pipelined DAG as a new solution design |
| 7 | Search half point in the critical delay path (Algorithm 2.3) |
| 8 | Insert a pipeline stage in DAG |
| 9 | else |
| 10 | return the saved solution design with its design information |
| 11 | end |
| 12 | else |
| 13 | Search half point in the critical delay path (Algorithm 2.3) |
| 14 | Insert a pipeline stage in DAG |
| 15 | end |

Algorithm 2.2: Half-cut pipelining procedure [12].

| |
|--|
| <p>Input: A critical path p_c including logic elements in series</p> <p>Output: Every point which needs to insert a pipeline stage</p> <pre> 1 $d_{acc} = 0$ // d_{acc}...accumulated delay 2 foreach logic element i in the critical path p_c do 3 if $d_{acc} + d(i) = p_c/2$ then // $d(i)$...delay of logic element i 4 return position of the logic element i 5 else if $d_{acc} + d(i) > p_c/2$ and $d_{acc} < p_c/2$ then 6 if $(d_{acc} + d(i) - p_c/2) > (p_c/2 - d_{acc})$ then 7 return position of the logic element $i - 1$ 8 else 9 return position of the logic element i 10 end 11 else 12 $d_{acc} = d_{acc} + d(i)$ 13 end 14 end </pre> |
|--|

Algorithm 2.3: Search half point in critical delay path [12].

Kim et al. also proposed the multicut algorithm which is more suitable for designs with a relatively longer critical path and/or with a target clock period relatively shorter [12]. The multicut method adds multiple pipeline stages into the critical path at a time. However, the drawback is that the designs are generally larger [12]. Nevertheless, both the multicut method and the half-cut method yield good results (close to optimal) with significant speed-up and linear complexity. Although they are primarily focused on ASIC designs, they are applicable to FPGA designs as well [12].

2.2.3 Power-Oriented Pipelining

Pipelining is usually considered to be a method for improving performance of the design. It is then a somehow surprising result that it can also reduce power consumption in FPGAs. Even though adding extra pipelining registers increases the total power, adding extra pipeline stages can reduce glitches or spurious transitions by orders of magnitude which, in return, decreases the power consumption [18, 19].

Two primary forms of power consumption occur in FPGAs (and in CMOS circuits in general). The first form is the static power which is the leakage power when transistors are 'off'. The second form, dynamic power, refers to the power consumption related to signal transitions given by the equation [18]:

$$P = \sum_{i=1}^n C_{\text{tot}}(i) V_{\text{DD}}^2 f, \quad (2.6)$$

where P is the power, n is the number of circuit elements (e.g. an inverter, pass transistor, etc.), C_{tot} is the capacitance of the i -th element's output node, V_{DD} is the supply voltage, and f is the switching frequency. Each time a transistor switches to a different logic state, the circuit consumes power.

Glitches occur when signals arrive at a logic element (LUTs in FPGAs) at different times. Paths for the LUTs used in FPGAs are generally unbalanced. Therefore, a LUT is forced to switch when one of the signals arrives ahead of the others. This can propagate further in a domino effect, forcing all the LUTs in the original LUTs transitive fanout to switch (demonstrated in Figure 2.6). Even though these glitches do not add errors to well-designed synchronous circuits, they contribute significantly to the dynamic power consumption. Glitch-related power consumption can be responsible for more than 80 % of the dynamic power in an FPGA [18, 19].

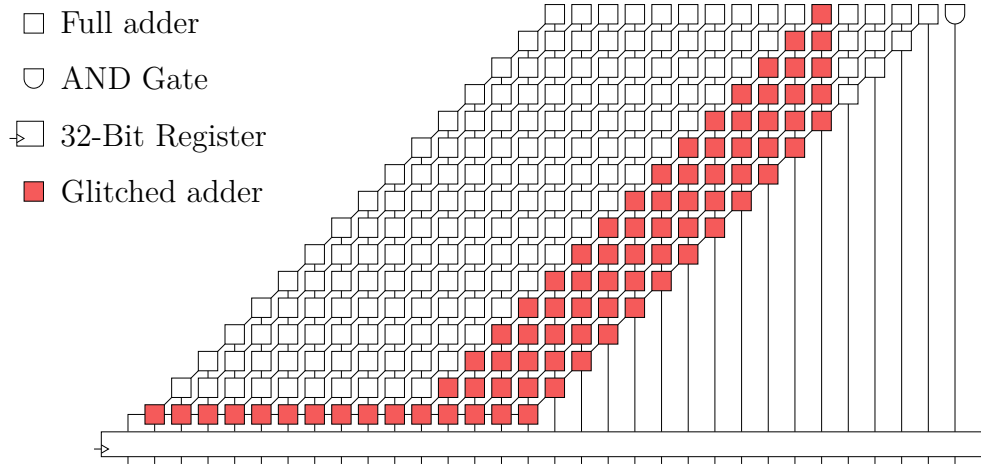


Figure 2.6: A one-stage array multiplier showing the domino effect of glitches [18].

Introducing extra pipeline stages keeps the glitches confined to a smaller portion of a chip's logic, as shown in Figure 2.7. Consequently, the probability of glitches is reduced [19]. This issue is less significant in ASICs because FPGAs usually use gates with more inputs (4-6 input LUTs) instead of mostly 1-3 input gates in ASICs [11].

Computer-Aided Design (CAD) algorithms that optimize circuit implementations can reduce FPGA power consumption by approximately 20 % [20]. Nevertheless, pipelining can save typically 40-80 % of the energy needed by the FPGA [20, 21]. Note that up to 98 % reduction in consumption was reported [21]. This enables, for example, the reduction of the size of power supplies and/or system costs by eliminating fans or heat sinks in FPGA-based hand-held devices [18].

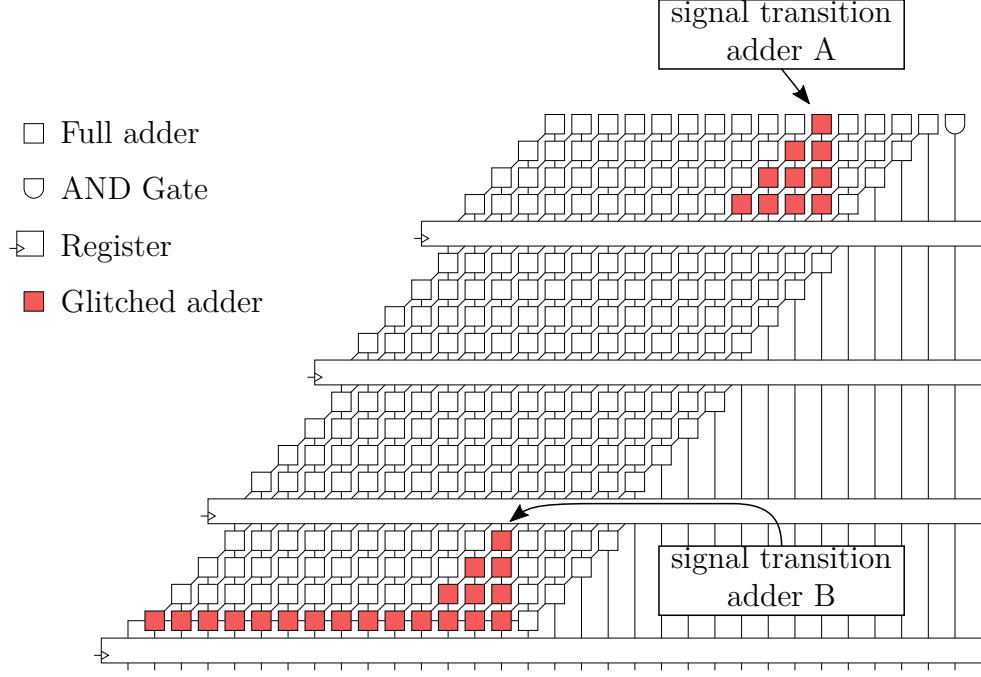


Figure 2.7: A four-stage array multiplier showing how the pipelining can reduce the domino effect of glitches [18].

2.2.4 Feedforward Circuits Pipelining

Many algorithms for feedforward circuits (i.e. circuits without loops) pipelining have been developed over the years. One example for computing a pipeline stage is a simple cut algorithm proposed by Ganusov et al. [11]. Results of their work were implemented as a tool for pipeline analysis in *Xilinx Vivado Design Suite* [22] release 2016.1. The algorithm works only for paths without loops. However, the linear time complexity $O(n)$ of this algorithm makes it very scalable. This also renders the algorithm as a perfect candidate to be exploited in the practical part of this thesis. The algorithm proposed by Ganusov et al. is described in Algorithm 2.4. This algorithm works as follows [11]:

1. All design paths are classified into feedforward (pipelinable) and loop (non-pipelinable).
2. All vertices in the design are sorted by their priority (given by the path delay these vertices are in) in an ascending order.
3. A new vertex $s \in S$ is added in the place of the most critical vertex $v \in V$ of the original DAG.
4. TFI and TFO vertices of the vertex v are discarded as candidates for the next pipeline register insertion.

Steps three and four are then repeated until all the vertices in the design have been visited.

Input: A circuit $G = \langle V, E, d, w \rangle$
Output: A list of pins (vertices) V_{stage}

- 1 Collect all the legal critical pins of the circuit into V_{legal}
- 2 Sort the pins in V_{legal} by increasing slack as primary goal and decreasing slack improvement as secondary goal
- 3 Append the remaining legal (non-critical) pins of the circuit to the end of V_{legal}
- 4 $V_{\text{stage}} = \{\}, V_{\text{TFI}} = \{\}, V_{\text{TFO}} = \{\}$
- 5 **foreach** pin $v \in V_{\text{legal}}$ **do**
- 6 **if** $(v \in V_{\text{TFI}}) \vee (v \in V_{\text{TFO}})$ **then**
- 7 **continue**
- 8 **end**
- 9 Add v to V_{stage}
- 10 Add the pins in the Transitive Fanin (TFI) of v into V_{TFI}
- 11 Add the pins in the Transitive Fanout (TFO) of v into V_{TFO}
- 12 **end**
- 13 **return** V_{stage}

Algorithm 2.4: Simple cut pipelining procedure [11].

The algorithm works because each path from primary input to primary output contains exactly one new vertex $s \in S$. A direct consequence is that there cannot be any path between two vertices of S ; mathematical proof can be found in [11]. This inherently ensures that the added latency affects all paths in the same way and that the data flow and functionality of the design are preserved.

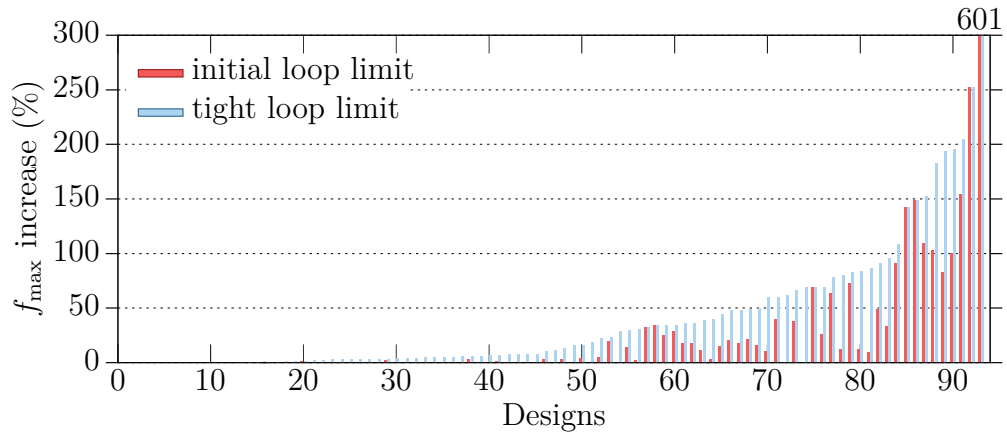


Figure 2.8: Performance of the extra simple cut pipelining method [11]. Initial loop limit corresponds to the timing within the loops after Place and Route. Tight loop limit corresponds to the timing after Place and Route when all paths, except the ones inside a loop, are disabled.

Experimental results show 19-29 % increase of f_{\max} on average in designs varying from 8,000 to 464,000 LUTs in size. Note that the highest frequency increase is 601 % in the best case. See Figure 2.8 for more detailed results.

2.2.5 Looped Circuits Pipelining

Pipelining looped circuits may lead to functional errors. Figure 2.9 demonstrates a loop where introducing a new pipelining stage breaks the functionality. The original loop in Figure 2.9a represents a counter initialized to 0 with incrementation by 1 on every cycle. However, the pipelined loop in Figure 2.9b produces a different sequence of values from the original. Fixing this requires a more complex transformation than simple insertion of flip-flops.

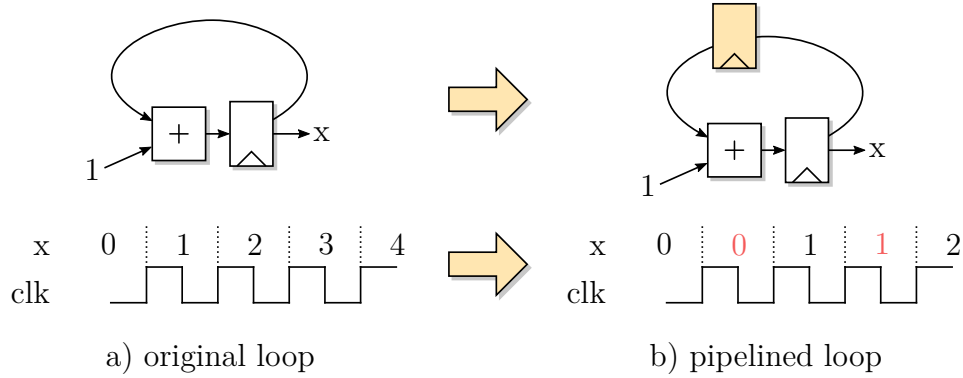


Figure 2.9: Sequential loop pipelining [11].

Slow loops in the designs are usually the main reason for FPGAs running below their capabilities [23]. Designers often have to manually parallelize and pipeline the design with great efforts to achieve the desired frequency. Another approach is to use High-Level Synthesis (HLS) where software programs written in languages such as C/C++ can be more easily optimized and compiled to RTL hardware design. An example of such tool is *Xilinx Vivado HLS* [24]. However, a netlist produced from HLS can still contain many timing-critical sequential loops [23]. Note that a netlist is a description of the connectivity of an electronic circuit.

HLS has become an essential FPGA design tool due to the high development productivity. For this reason, loops pipelining is widely supported in many state-of-art HLS tools [24]. Contrarily, automating low-level loop optimization for FPGAs as a very relevant topic is still unexplored. As mentioned before, it is not possible to simply add pipeline registers to sequential loops. More sophisticated transformations need to be used to improve f_{\max} . Four loop transformation techniques are listed and described here.

Loop Splitting

As shown in Figure 2.9, pipelining the counter changes the sequence of the output and its functionality. We can split the counter into two separate loops (odd and even) with increments of 2, as shown in Figure 2.10. This eliminates the loop as a critical path, even though the frequency of the counters remains the same. This technique can be easily automated, although it may increase the design area [23].

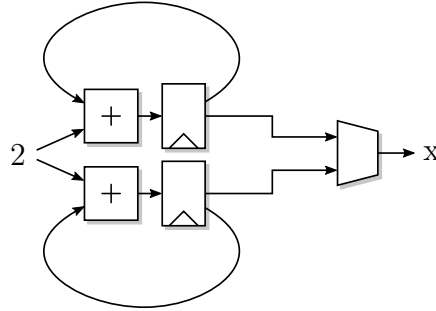


Figure 2.10: Loop splitting example [23].

Omidian et al. argue that loops can be generated from different parts in the whole RTL project. Therefore, it may be challenging to keep track of the loops manually [23]. Furthermore, developers need to add one more clock to the design manually which adds one more constraint to the design [23]. Automating the process of loop splitting can save a lot of resources to be utilized elsewhere.

Associative Refactoring

Figure 2.11a shows another simplified loop that is commonly used. Since the add operation is associative, the order can be changed. Consequently, the whole loop can be transformed to an adder chain with an accumulator at the end (shown in Figure 2.11b) [23].

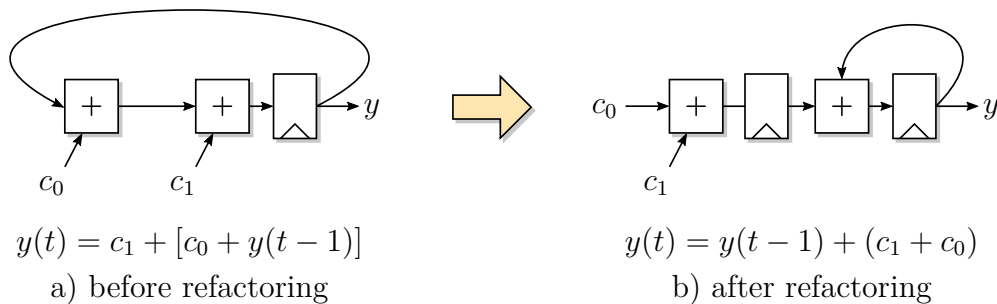


Figure 2.11: Associative refactoring example [23].

This technique can generally be used on any associative function [23]. Figure 2.12 shows a generalized refactored diagram.

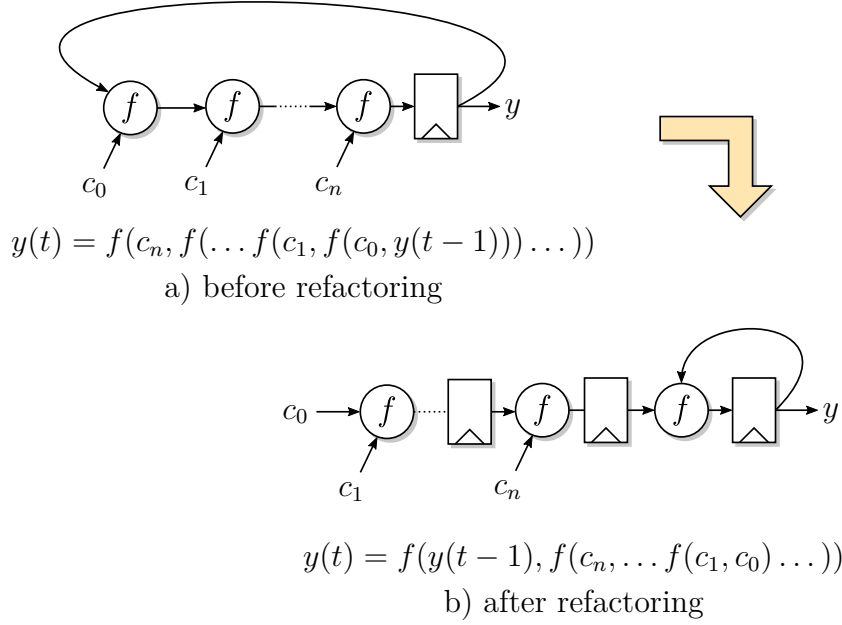


Figure 2.12: Associative refactoring of a function [23].

Combining

Another approach is to add pipeline stages inside the loop to increase the frequency. This technique is similar to *C-slow retiming* which enhances conventional retiming by simply replacing every register with a sequence of c separate registers before retiming occurs [25]. The easiest way to utilize this technique is to simply multiplex and demultiplex c separate data streams, as depicted in Figure 2.13 [25]. In this example, two loop patterns similar to those in Figure 2.9 are combined and a valid output for each of them is generated on alternate clock cycles in a *round-robin* manner.

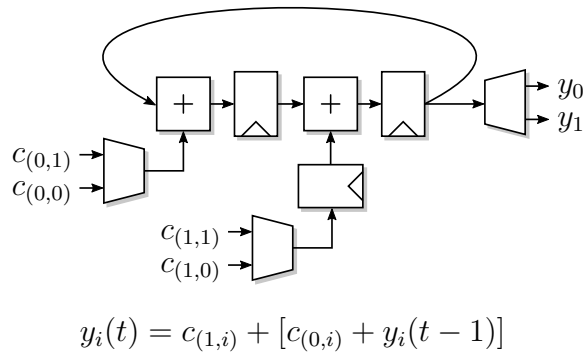


Figure 2.13: Adder chain pipelined by the combining technique [23].

The combining technique is the only one that may decrease the area cost of the design [23]. If the function f is associative, it is possible to transform it into a

function chain, as shown in Figure 2.12b. Regardless of whether the function f is associative or not, pattern matching can be used to find similar patterns and use combining [23].

Early Conditions

The last typical pattern described, early conditions, is a part of a simple Finite-State Machine (FSM). In many cases, each FSM state waits for a fixed amount of time using a counter followed by initialization of the current state. Several levels of logic (divided into condition logic and initialization logic) are often generated with big counters which slow the circuit down [23]. A pipeline stage can be added between condition logic and initialization logic. However, this results in the final condition being generated one clock cycle too late [23].

A new condition computed one cycle early can be added to correct the circuit. Combined with a pipeline delay, this results in a functionally correct logic [23]. A part of a simple FSM is shown in Figure 2.14a. The waveform of the counter and waveforms of conditions before and after transformation are shown in Figure 2.14b.

This technique can be automated for simple patterns, as the one demonstrated in Figure 2.14 [23]. Nevertheless, more complex cases may require manual modifications by designers.

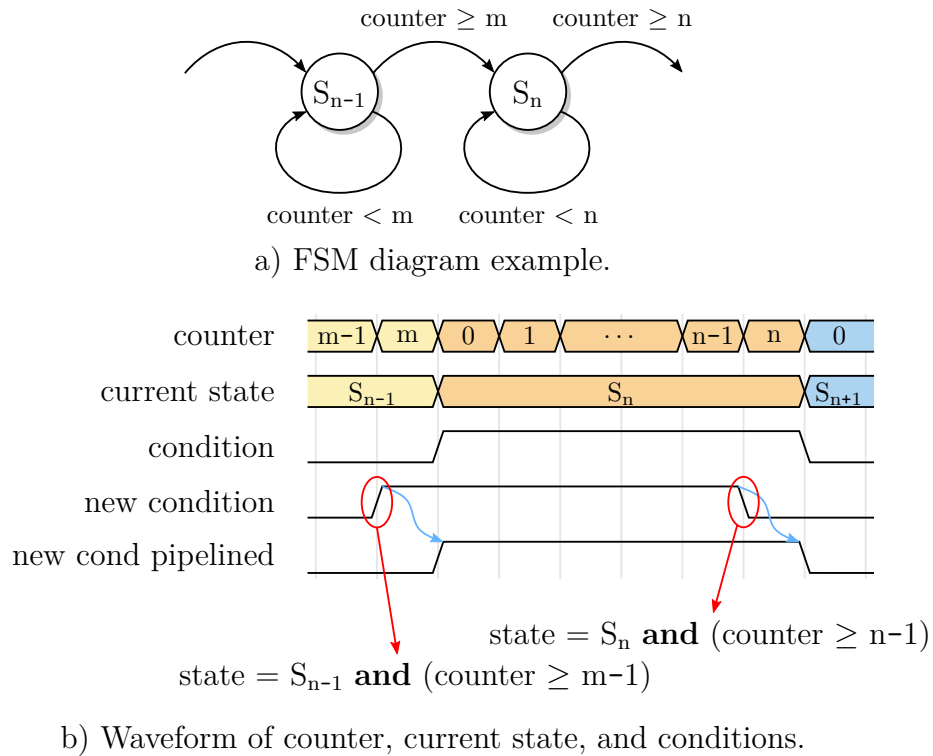


Figure 2.14: Pipelining a loop inside an FSM by changing a condition [23].

3 Circuit Abstraction

As already described in section 2.1, a circuit without feedback loops can be modelled as a finite $G = \langle V, E, d, w \rangle$ DAG. A very similar approach is utilized in this thesis. The proposed representation models the circuit as a $G = \langle V, E, w \rangle$ DAG. Vertices V represent pins of instances and edges E represent nets and internal paths of individual instances. Each edge $e \in E$ has a propagation delay w .

3.1 Vertices and Edges

Each instance stored in the netlist file is described, among others, by the cell type used (e.g. a D-type flip-flop or a 4-input LUT), and each of these cells is described by its input and output pins. These pins are represented by vertices V in the DAG. This thesis adopts an approach where pins are modelled without any weights (i.e. delays) and stores all delay information into the edge weights w instead.

Input and output vertices of each instance are connected by an edge. The direction of this edge is inherently always from input to output; no edge connecting two inputs or two outputs exists. A DAG that models the circuit as a graph representing propagation delays is proposed. In other words, the internal structures of instances and logical responses to the stimuli are irrelevant. Only the time it takes to propagate the signal from an instance input to its output is valuable. This enables the edge e to be created as a pair of input pin i and output pin o , formally notated as $i \xrightarrow{e} o$. All edges describing internal connections of an instance have the same propagation delay. These edges are later referred to as internal edges.

Each input is connected to every output of the specific instance, as is shown in Figure 3.1. The out-degree of each input vertex equals to the number of output vertices of the same instance. Similarly, the in-degree of each output vertex equals to the number of input vertices of the same instance. An exception to this rule are the vertices representing clock pins (e.g. CLK pin of a D-type flip-flop) because these pins (and corresponding clock paths) are not submitted to the pipelining. One crucial property of input vertices rises from how digital circuits work. The in-degree of all input vertices always equals to one. This is because only one signal may drive a pin.

Analogically to internal edges, edges connecting individual instances are referred to as external edges — these edges model physical nets in the design. A simple example is shown in Figure 3.2. Each physical net may be described by more than one edge. External edges representing a single net do not need to have the same propagation delay as opposed to the internal edges modelling the internal delay structure of an instance.

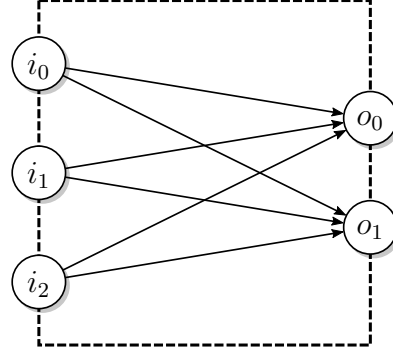


Figure 3.1: Abstract representation of an instance (edge weights are omitted).

Equally to clock pins and corresponding internal edges being excluded from the instance abstraction, external edges connecting ground and/or power vertices are also excluded from the final design abstraction. Therefore, these edges are not submitted to the pipelining either.

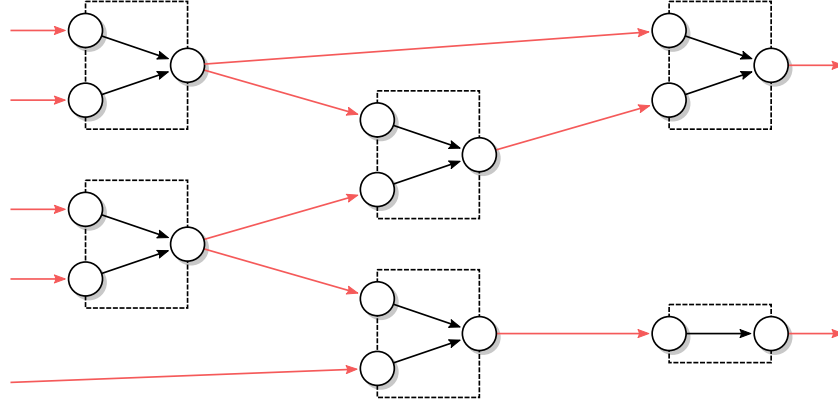
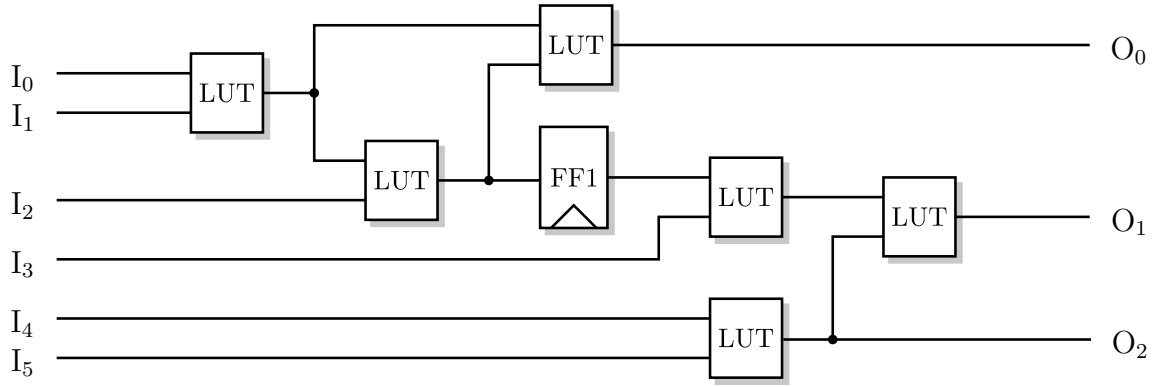


Figure 3.2: Instances connected by external edges (colored red; edge weights are omitted).

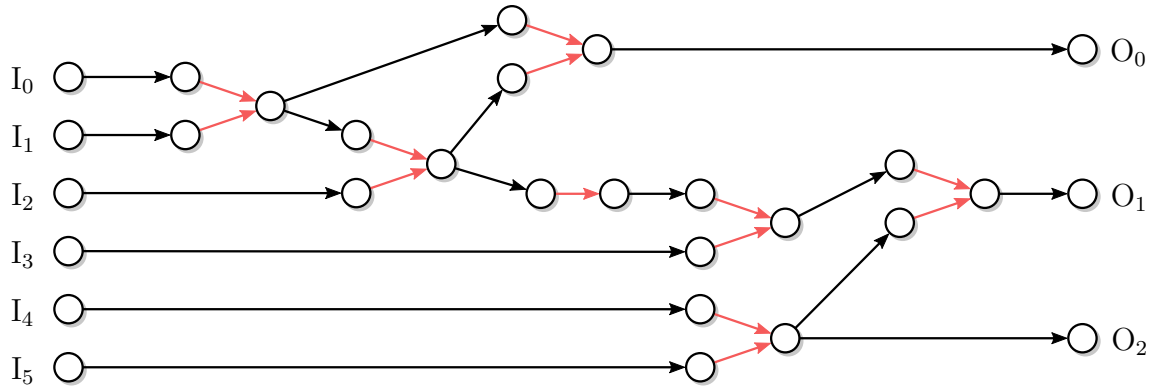
3.2 Paths

A path is defined as an alternating sequence of vertices and edges. Delay of the path is the sum of its partial delays. The proposed abstraction differentiates between internal and external edges. Edges in a path alternate between the two types. Two adjacent edges of the same type can be truncated into a single edge with total delay calculated as the sum of their partial delays. In practice, this never happens because CAD tools solve this problem before the design abstraction is constructed.

As an example, a small circuit and its abstraction are shown in Figure 3.3. This circuit consists of 7 instances and a total of 13 possible paths.



a) Small circuit.



b) Circuit abstraction.

Figure 3.3: A small circuit and its abstraction into a directed graph. The edges between vertices of the same instances are colored in red [11].

4 Data Extraction, Graph Assembly, and Pipelining Algorithms

This chapter of the thesis deals with timing results and netlist extraction from *Xilinx ISE[®] Design Suite* release 14.7 [26] (Xilinx ISE or simply ISE hereafter). The abstract circuit representation assembly and pipelining algorithms implementation are also introduced in this chapter. Scripts used in the thesis are written in *Python* release 3.8 [27] language unless noted otherwise.

The whole process is depicted in Figure 4.1. Following sections discuss timing results and netlist extraction from ISE. Afterwards, algorithms constructing an abstract circuit representation described in chapter 3 are introduced. Finally, the path-forward pipelining algorithm implementation is presented and described.

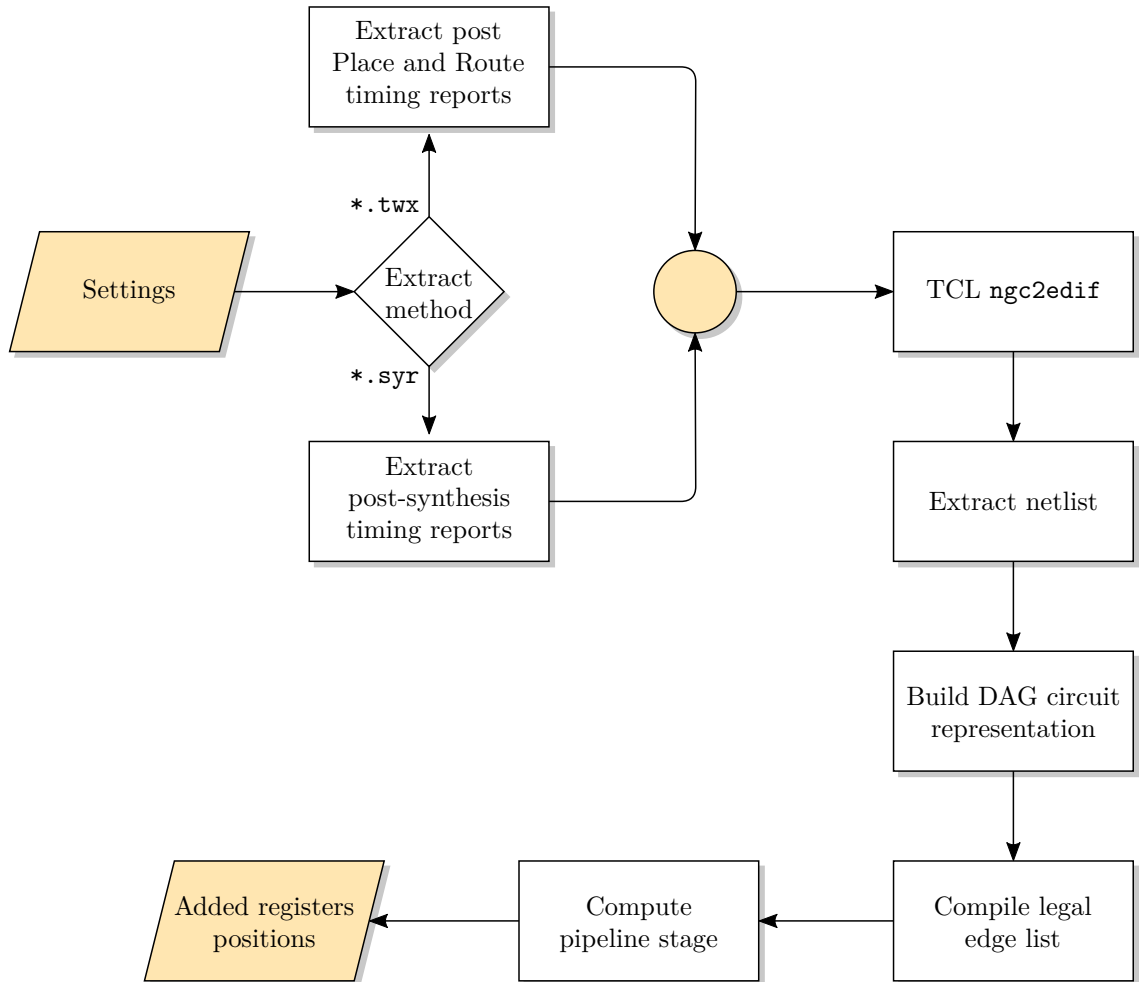


Figure 4.1: Process of adding a new pipeline stage.

4.1 Data Extraction

This section introduces the process of timing reports and netlist data extraction from specific files generated by Xilinx ISE. A file-reading script is utilized to choose the file for data extraction (i.e. extension `*.syx` or `*.twx` for timing reports and `*.edf` or `*.ndf` for netlist).

4.1.1 Timing Reports Data Extraction

Xilinx ISE provides the user with timing reports generated after:

- the design is synthesized,
- the design is mapped to the physical hardware,
- the design is placed and routed (PaR).

Two scripts were written to extract these timing results – the first for the synthesized design and the second for the placed and routed design.

Timing Reports after Synthesis

Synthesis reports contain, among other helpful information, timing details of the synthesized design. By default, only the slowest path is printed in the report. The number of critical paths obtained from the synthesis results has to be increased to a higher value to gather sufficient data. Even though these reports can be helpful, more accurate reports can be extracted after the Place and Route process.

Detail of circuit timing gathered from the timing report after synthesis is shown in Listing 4.1. The critical path shown is from a flip-flop `q_a` to a flip-flop `Y` through two LUTs internally labelled as `d_y11` and `d_y21`.

The script parses the timing reports using *regular expressions* to extract a specific text from the `*.syx` file where synthesis results are stored by Xilinx ISE. The script first identifies where the timing detail starts and ends. It then reads the information in the header – path's total delay, source/destination names, and logical element types. Only paths from a flip-flop to a flip-flop are considered. Paths leading from and/or to pads, DSPs (Digital Signal Processors), and others are ignored. An error is raised if the logical elements are not flip-flops. Path's details, including gate delays, net delays, and logical and net names, are then extracted and stored in an internal path object alongside with all data from the header. The pseudocode of the script is shown in Algorithm 4.1.

Timing Reports after Place and Route

Timing reports after the Place and Route process yield more accurate information compared to the timing reports generated after synthesis. Advanced analysis has to

Listing 4.1: An example of Xilinx ISE Synthesis timing results.

```

=====
Timing constraint: Default period analysis for Clock 'CLK'
Clock period: 4.381ns (frequency: 228.258MHz)
Total number of paths / destination ports: 11 / 2
=====

Delay:                4.381ns (Levels of Logic = 2)
Source:               q_a (FF)
Destination:          Y (FF)
Source Clock:         CLK rising
Destination Clock:    CLK rising

Data Path: q_a to Y

      Cell:in->out      fanout      Gate      Net
                        Delay      Delay      Logical Name (Net Name)
-----
      FD:C->Q            1    0.720    1.140    q_a (q_a)
      LUT4:I0->0          2    0.551    1.216    d_y11 (d_y1)
      LUT4:I0->0          1    0.551    0.000    d_y21 (d_y2)
      FD:D                1    0.203          Y
-----

Total                4.381ns (2.025ns logic, 2.356ns route)
                        (46.2% logic, 53.8% route)

```

Input: A synthesis results file

Output: A list of objects containing the information about path's delay, source, and destination

```

1 Find timing detail start
2 Find and store total delay  $t_{tot}$ 
3 Find and store source name  $s_{name}$  and destination name  $d_{name}$ 
4 Find and store source type  $s_{type}$  and destination type  $d_{type}$ 
5 if ( $s_{type} = FF$ )  $\wedge$  ( $d_{type} = FF$ ) then
6   | continue
7 else
8   | error
9 end
10 repeat
11   | Find and store gate delay  $g_{delay}$ 
12   | Find and store net delay  $n_{delay}$ 
13   | Find and store gate name  $g_{name}$ 
14   | Find and store net name  $n_{name}$ 
15   | Move to next element
16 until  $\nexists$  next element

```

Algorithm 4.1: Synthesis timing reports data extraction.

be enabled in order to obtain detailed reports including the critical paths entries. This is achieved either in GUI under *Generate Post-Place & Route Static Timing* process properties or in the terminal via the TCL (Tool Command Language) command `trce`. Argument `-n` followed by an integer in this command specifies the total number of paths generated. For further information about the TCL commands, the reader can refer to [28].

Detail of circuit timing gathered from the timing report after Place and Route (PaR) is shown in Listing 4.2. The critical path shown in the report leads from a flip-flop `q_c` to a flip-flop `Y` through two LUTs internally labelled as `d_y11` and `d_y21` located in their respective slices.

Listing 4.2: An example of Xilinx ISE Place and Route timing results.

| | | | |
|-----------------------------|---|--------------------------------|--|
| ----- | | | |
| Slack (setup path): | 16.542ns | | |
| | (requirement - (data path - clock path skew + uncertainty)) | | |
| Source: | q_c (FF) | | |
| Destination: | Y (FF) | | |
| Requirement: | 20.000ns | | |
| Data Path Delay: | 3.457ns (Levels of Logic = 2) | | |
| Clock Path Skew: | -0.001ns (0.041 - 0.042) | | |
| Source Clock: | CLK_BUFGP rising at 0.000ns | | |
| Destination Clock: | CLK_BUFGP rising at 20.000ns | | |
| Clock Uncertainty: | 0.000ns | | |
| Maximum Data Path: q_c to Y | | | |
| Location | Delay type | Delay(ns) | Physical Resource Logical Resource(s) |
| ----- | | | |
| J13.IQ1 | Tiockiq | 0.259 | C |
| | | | q_c |
| SLICE_X38Y17.G1 | net (fanout=1) | 1.220 | q_c |
| SLICE_X38Y17.Y | Tilo | 0.608 | q_y |
| | | | d_y11 |
| SLICE_X38Y17.F3 | net (fanout=1) | 0.015 | d_y1 |
| SLICE_X38Y17.X | Tilo | 0.608 | q_y |
| | | | d_y21 |
| K12.01 | net (fanout=1) | 0.719 | d_y2 |
| K12.0TCLK1 | Tioock | 0.028 | Y |
| | | | Y |
| ----- | | | |
| Total | | 3.457ns | |
| | | (1.503ns logic, 1.954ns route) | |
| | | (43.5% logic, 56.5% route) | |

Apart from more accurate results, Place and Route results are also stored in HTML format. Therefore, the extraction is much simpler. A sample of PaR timing results (in HTML format) is shown in Listing 4.3.

The script parsing the Place and Route timing report (`*.twx` file) uses Python library `Beautiful Soup` [29]. Pseudocode of this script is shown in Algorithm 4.2.

Listing 4.3: Xilinx ISE Place and Route timing results encoded in HTML.

```

...
<twSrc BELType='FF'>q_c</twSrc>
<twDest BELType='FF'>Y</twDest>
<twLogLvls>2</twLogLvls>
<twSrcSite>J13.ICLK1</twSrcSite>
<twSrcClk twEdge ="twRising" twArriveTime ="0.000">CLK_BUFGP</twSrcClk>
<twPathDel>
  <twSite>J13.IQ1</twSite>
  <twDelType>Tiockiq</twDelType>
  <twDelInfo twEdge="twRising">0.259</twDelInfo>
  <twComp>C</twComp>
  <twBEL>q_c</twBEL>
</twPathDel>
<twPathDel>
  <twSite>SLICE_X38Y17.G1</twSite>
  <twDelType>net</twDelType>
  <twFanCnt>1</twFanCnt>
  <twDelInfo twEdge="twRising">1.220</twDelInfo>
  <twComp>q_c</twComp>
</twPathDel>
...

```

Input: A Place and Route timing results file

Output: A list of objects containing information about path's delay, source and destination

```

1 Get all paths  $P$  from the timing results file
2 foreach path  $p \in P$  do
3   Find and store source name  $s_{\text{name}}$  and destination name  $d_{\text{name}}$ 
4   Find and store source type  $s_{\text{type}}$  and destination type  $d_{\text{type}}$ 
5   if  $(s_{\text{type}} \neq \text{FF}) \vee (d_{\text{type}} \neq \text{FF})$  then
6     continue
7   end
8   Get path resources  $R$ 
9   foreach resource  $r \in R$  do
10    Find and store resource name  $r_{\text{name}}$ 
11    Find and store resource delay  $r_{\text{delay}}$ 
12    Find and store resource type  $r_{\text{type}}$ 
13  end
14 end

```

Algorithm 4.2: Place and Route timing reports data extraction.

All the paths in the timing report are identified at first and the path source and destination are extracted afterwards. If either the source or destination is not a flip-flop logic element, the path is dropped and the algorithm proceeds to the following path. This is because a list of edges is compiled and sorted by delays in a descending order. For example, if a path leads from or to a pad, this path will have a higher priority in the pipeline stage computation algorithm over the path that is not listed in the timing results. In other words, the path connected to the pad will be preferred over an unlisted path even if the design would benefit more from adding the pipelining register to the unlisted path.

Each path consists of nets and logic resources. Information about each resource, including the name, type, and delay, is extracted and stored.

As opposed to the timing details extracted from the synthesis results file, this script does not raise an error if a path does not originate or lead to a flip-flop. This is because it is reasonably simple to change the number of generated paths (and their corresponding timing details). Consequently, enough detail about the circuit can be extracted to build a comprehensive timing model. Several paths not coming or leading to a flip-flop can be dropped, and yet an accurate timing model of the circuit can still be built.

4.1.2 Netlist Extraction

Information about the synthesized design is stored as a netlist and can be accessed in an encoded format in an `*.ngc` type file. Prior to the netlist data extraction, the user has to first convert the file to a readable `*.edf` (or an equivalent `*.ndf`) file format. This can be done using the `ngc2edif` TCL command. Entries in an `*.edf` file encompass individual cells with specific inputs and outputs, their instantiation, and nets connecting individual instantiated cells. An example showing `*.edf` file format syntax can be found in Listing 4.4.

Xilinx ISE always lists all used cell types first. Every cell is described by its interface (i.e. inputs and outputs). These cells can range from D-type flip-flops, n-input LUTs, multiplexors to buffers, invertors, memory blocks etc. The generated `*.edf` file lists only cells used in the design. This property is leveraged by the circuit abstraction building algorithm described in section 4.2.

Specific instances are listed right after cells. Each of these instances has assigned a cell type and a unique name. An internal property "rename" is also assigned to each instance. Each instance identified by its "rename" property is connected by several nets. Each net with its unique name specifies the interface pin of the instance it is connected to.

First, the netlist is first converted from an `*.ngc` to an `*.edf` format in the

Listing 4.4: Syntax used in `edf` netlist files.

```

...
# Cell syntax example
(cell FD
  (cellType GENERIC)
  (view view_1
    (viewType NETLIST)
    (interface
      (port C
        (direction INPUT)
      )
      (port D
        (direction INPUT)
      )
      (port Q
        (direction OUTPUT)
      )
    )
  )
...
# Cell instantiation syntax example
(instance (rename q_a_renamed_2 "q_a")
  (viewRef view_1 (cellRef FD (libraryRef UNISIMS)))
  (property XSTLIB (boolean (true)) (owner "Xilinx"))
  (property IOB (string "true") (owner "Xilinx"))
...
# Net syntax example
(net CLK_BUFGP
  (joined
    (portRef C (instanceRef q_b_renamed_0))
    (portRef C (instanceRef Y_renamed_1))
    (portRef C (instanceRef q_c_renamed_4))
    (portRef O (instanceRef CLK_BUFGP_renamed_11))
  )
)
...

```

thesis. Then, the netlist information is extracted from the `*.edf` file using the script presented in Algorithm 4.3. This script is used to list all the cell data. Afterwards, instances are extracted from the netlist file using the script described in Algorithm 4.4. Finally, nets data are obtained by the script shown in Algorithm 4.5. As shown in Figure 4.2, the scripts are run sequentially and the extracted data is stored in three lists – cell-types, instances, and nets.

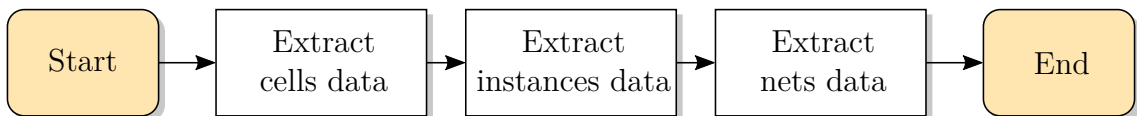


Figure 4.2: Netlist extraction process.

Input: A netlist file
Output: List of cells

```

1 Create a blank list of cells
2 repeat
3   Find and store cell name  $c_{\text{name}}$ 
4   Create a blank list of cell inputs  $C_{\text{inputs}}$ 
5   repeat
6     Append cell input  $c_{\text{input}}$  into list of cell inputs  $C_{\text{inputs}}$ 
7     Go to next cell input  $c_{\text{input}}$ 
8   until  $\nexists$  cell input  $c_{\text{input}}$ 
9   Create a blank list of cell outputs  $C_{\text{outputs}}$ 
10  repeat
11    Append cell output  $c_{\text{output}}$  into list of cell outputs  $C_{\text{outputs}}$ 
12    Go to next cell output  $c_{\text{output}}$ 
13  until  $\nexists$  cell output  $c_{\text{output}}$ 
14  Create new cell object and fill in  $c_{\text{name}}$ ,  $C_{\text{inputs}}$ ,  $C_{\text{outputs}}$ 
15  Append the cell object into list of cells
16  Go to next cell
17 until  $\nexists$  cell  $c$ 

```

Algorithm 4.3: Cell data extraction.

Input: A netlist file
Output: List of instances

```

1 Create a blank list of instances
2 repeat
3   Find and store instance name  $i_{\text{name}}$ 
4   Find and store instance rename  $i_{\text{rename}}$ 
5   if  $\nexists i_{\text{rename}}$  then // if does not exist
6      $i_{\text{rename}} = i_{\text{name}}$  // use original name instead
7   end
8   Find and store cell reference  $i_{\text{cell}}$ 
9   Create new instance object and fill in  $i_{\text{name}}$ ,  $i_{\text{rename}}$ ,  $i_{\text{cell}}$ 
10  Append the instance object into list of instances
11  Go to next instance
12 until  $\nexists$  instance  $i$ 

```

Algorithm 4.4: Instance data extraction.

| |
|---|
| <p>Input: A netlist file</p> <p>Output: List of nets</p> <pre> 1 Create a blank list of nets 2 repeat 3 Find and store net name n_{name} 4 Create a blank list of instance references I_{ref} 5 repeat 6 Append pair $[i_{\text{ref}}, p_{\text{ref}}]$ into I_{ref} 7 Go to next reference 8 until $(\nexists \text{ instance reference } i_{\text{ref}}) \vee (\nexists \text{ port reference } p_{\text{ref}})$ 9 Create new net object and fill in $n_{\text{name}}, I_{\text{ref}}$ 10 Append the net object into list of nets 11 Go to next net 12 until $\nexists \text{ net } n$ </pre> |
|---|

Algorithm 4.5: Net data extraction.

4.2 Circuit Representation Assembly

This section describes the building sequence of an abstract circuit representation. As has already been introduced in chapter 3, physical circuits are modelled as finite DAGs in this thesis. Nodes of these DAGs represent pins of instances, and edges represent causal connections between these pins. Edges are either external or internal.

The building sequence is divided into several phases. These phases are tightly linked to the netlist data extraction scripts. Although the process of netlist data extraction is described as an independent procedure in section 4.1.2, it is embedded in the algorithms assembling the DAG. This works in a tick-tock way. In other words, each time the individual list is extracted from the netlist, the algorithm building the corresponding part of the DAG is executed. The whole procedure continues until the DAG is fully assembled. The whole process is pictured in Figure 4.3.

This section builds upon the netlist extraction process described earlier. The lists of cells, instances, and nets extracted from the netlist enter the circuit assembly algorithms as input variables.

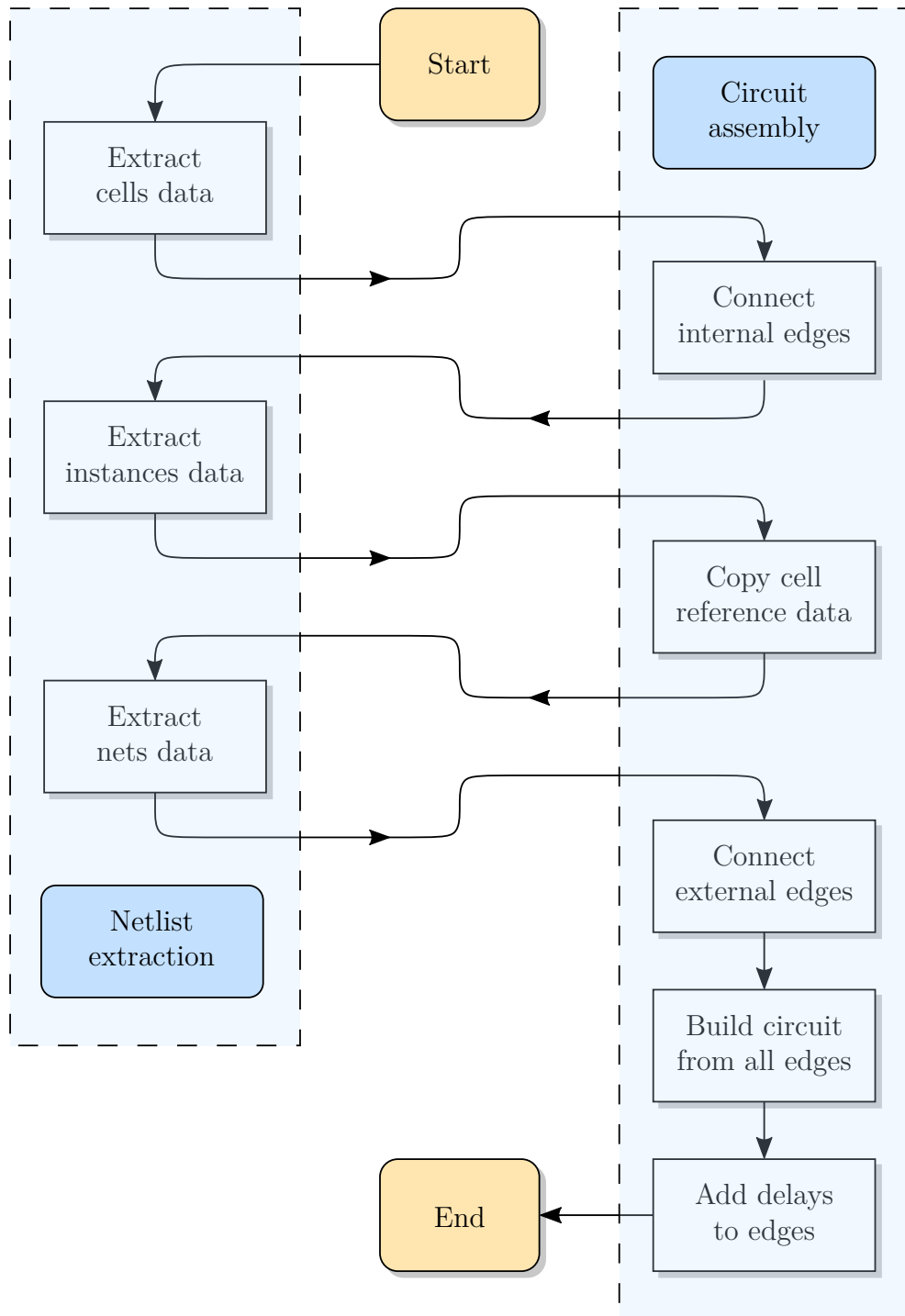


Figure 4.3: Circuit building algorithms and netlist extraction algorithms running in a tick-tock way.

4.2.1 Internal Edges Connection

Each cell is represented as an object with name, inputs, and outputs attributes. This section describes adding another attribute to each cell – a list of internal edges. The process of connecting the cell's inputs to outputs that create these edges is shown in Algorithm 4.6. This algorithm connects individual input pins to each and every output pin.

The user can exclude input pins of a D-type flip-flop, such as set, reset or preset pins in the settings file. There are cases where it is advantageous to do so (e.g. asynchronous reset). However, these pins can be pipelined if necessary (e.g. in a counter).

The algorithm first verifies whether the parsed cell type is a D-type flip-flop. If so, user-defined pins and clock pins are ignored. Afterward, the algorithm creates an input-output pair and appends it to a list of input-output pairs for the corresponding cell. This list is then stored as an attribute of the parsed cell.

A library of all logical elements (marked as the list C) used in the circuit is created by the algorithm. The library is then passed to another algorithm described in section 4.2.2.

| |
|---|
| <p>Input: List of all cells C, user settings S</p> <p>Output: Modified list C (list of edges connecting cell inputs to outputs added)</p> <pre>1 foreach cell $c \in C$ do 2 Create a blank list C_{edges} for storing input-output pairs 3 foreach cell input pin c_{input} do 4 if (cell name $c_{\text{name}} = \text{"FD"} \wedge (c_{\text{input}} \in S)$) then 5 continue // Skip user-defined pins 6 else if $c_{\text{input}} = \text{"CLK"}$ then 7 continue // Skip clock pins 8 else 9 foreach cell output pin c_{output} do 10 Append pair $[c_{\text{input}}, c_{\text{output}}]$ to list C_{edges} 11 end 12 end 13 end 14 Store list C_{edges} as an attribute of cell c 15 end</pre> |
|---|

Algorithm 4.6: Connecting internal edges.

4.2.2 Reference Cell Data Copy

Previous subsection described the process of assignment of internal edges to instances. This list of cells C forms a library of all possible types of logical elements in the circuit. Each instance has a reference to a cell it is made of. Edges and their respective inputs and outputs are copied from the cell linked to the corresponding instance, as shown in Algorithm 4.7.

The algorithm finds corresponding cell references and the attributes of the cell are copied into the respective instance. The algorithm also adds internal edges of the corresponding instance to a list of internal edges E_{int} . This list is subsequently used in the final circuit building algorithm alongside a list E_{ext} containing external edges (refer to section 4.2.3).

| |
|---|
| <p>Input: List of instances I, list of cells C</p> <p>Output: Modified list of instances I (attributes appended), list of internal edges E_{int}</p> <pre>1 Create blank list of all internal edges E_{int} 2 foreach instance $i \in I$ do 3 Create blank sub-list of internal edges $E_{\text{int,sub}}$ 4 foreach cell $c \in C$ do 5 if cell name $c_{\text{name}} =$ instance reference i_{ref} then 6 Copy list of inputs C_{inputs} to I_{inputs} 7 Copy list of outputs C_{outputs} to I_{outputs} 8 Copy list of edges C_{edges} to $E_{\text{int,sub}}$ 9 break // found, no need to continue... 10 end 11 end 12 Add sub-list of internal edges $E_{\text{int,sub}}$ and instance name i_{name} to list E_{int} 13 end</pre> |
|---|

Algorithm 4.7: Copying cell reference data.

4.2.3 External Edges Connection

Each net connects a minimum of two different instances. Therefore, the net consists of at least one external edge connecting these instances. External edges forming the net are directed from an instance output to another instance's input. Generally, for a net connecting n instances, there exists $n - 1$ external edges (reason described in section 3.1).

Each net extracted from the netlist is parsed by Algorithm 4.8. The algorithm first finds the instance this net is connected to in the list of instances I . If the net is connected to the input pin of the instance, the algorithm creates an edge to this pin (i.e. head of the edge). Otherwise, it creates an edge from this pin (tail of the edge).

A list containing exactly one output pin and at least one input pin is created after the whole net is parsed by the algorithm. List of external edges E_{ext} is produced by the algorithm shown in Algorithm 4.8. Lists of internal and external edges, denoted by E_{int} and E_{ext} respectively, are exploited by the final circuit building algorithm.

| |
|--|
| <p>Input: List of nets N, list of instances I, settings S</p> <p>Output: Modified list of nets N (attributes appended), list of external edges E_{ext}</p> <pre> 1 Create blank list of all external edges E_{ext} 2 foreach net $n \in N$ do 3 Create blank sub-list of external edges $E_{\text{ext,sub}}$ 4 foreach instance reference $i_{\text{ref}} \in I_{\text{ref}}$ do // I_{ref} is a net attribute 5 foreach instance $i \in I$ do 6 if $i_{\text{ref}} = i_{\text{rename}}$ then // $i_{\text{rename}} \dots \text{instance rename}$ 7 if pin p of i_{ref} is input pin then // each i_{ref} has only one pin p 8 Add pin as first term to sub-list of external edges $E_{\text{ext,sub}}$ 9 else 10 Add pin as last term to sub-list of external edges $E_{\text{ext,sub}}$ 11 end 12 break // found instance, no need to continue... 13 end 14 end 15 end 16 Add sub-list of external edges $E_{\text{ext,sub}}$ and net name n_{name} to list E_{ext} 17 end </pre> |
|--|

Algorithm 4.8: Connecting external edges.

4.2.4 Propagation Delay and Circuit Assembly

In this subsection, the final assembly of the DAG circuit abstraction is described. The subsection also covers the algorithm adding propagation delays to the built graph. Propagation delays are extracted from timing details (described in section 4.1.1) and stored in a list of paths P .

The assembly of the DAG circuit abstraction is described in Algorithm 4.9. Specific vertices can be excluded from the graph assembly by listing them in the

settings file. By default, the ground and power source vertices are excluded because they supply a constant logical zero or logical one to the circuit, and therefore do not submit to the pipelining.

| |
|---|
| <p>Input: List of internal edges E_{int}, list of external edges E_{ext}, list of vertices to be deleted V_{tbd}</p> <p>Output: A DAG $G = \langle V, E \rangle$</p> <pre> 1 Add edges to G from E_{int} 2 Add edges to G from E_{ext} 3 foreach vertex to be deleted $v_{\text{tbd}} \in V_{\text{tbd}}$ do 4 Remove vertices succeeding node v_{tbd} 5 Remove vertices preceding node v_{tbd} 6 end </pre> |
|---|

Algorithm 4.9: DAG assembly.

Each path extracted from timing details consists of several logical elements and nets (generally referred to as resources) repeating in an alternating pattern (as described in section 3.2). Names of the resources match the names used in the netlist. The algorithm in Algorithm 4.10 finds the resource by the name given in the netlist and assigns the propagation delay of the resource as a weight of the corresponding edge. Propagation delays of logical elements are assigned to internal edges, and propagation delays of nets are assigned to external edges. Zero propagation delay is assigned to the resource if the resource is not in the list of paths P .

| |
|---|
| <p>Input: List of paths P with delays, a built DAG G</p> <p>Output: A DAG $G = \langle V, E, w \rangle$</p> <pre> 1 Assign zero propagation delay to all edge weights 2 foreach path $p \in P$ do 3 foreach resource $r \in p$ do 4 Find resource name r_{name} in graph G edges 5 if resource type $r_{\text{type}} = \text{edge type}$ then 6 Assign resource delay r_{delay} to the edge weight 7 end 8 end 9 end </pre> |
|---|

Algorithm 4.10: Adding delays to DAG edges.

4.3 Pipelining Implementation

The algorithm enabling the user to add a pipelining stage quickly and efficiently is the cornerstone of this thesis. However, the pipelining algorithm and the procedure of adding a new pipeline stage is preceded by the compilation of the legal edge list.

4.3.1 Legal Edge List Compilation

Paths between registers with the largest propagation delay have the highest priority when adding a pipeline stage. Propagation delays of these critical paths are extracted by algorithms shown in subsection 4.1.1. This section describes calculating the most ideal positions for inserting pipeline registers in these paths.

Consider the following example: A critical path leading from point U to point V consists of several resources R divided into logical elements and nets connecting these elements. Each resource $r \in R$ has its partial propagation delay t_{delay} . This path has a total propagation delay t_{tot} calculated as the sum of its partial delays $\sum_{j=1}^n t_{\text{delay},j}$. The ideal half-cut delay is $t_{\text{tot}}/2$. If we insert a new register into this half-cut position, the benefit of the added register is maximal. However, to insert the register in this ideal position is not always possible. The main problem with this approach is that it is illegal to insert pipelining registers into logical elements (i.e. internal edges). New registers can only be added into nets (i.e. external edges), splitting them in half in the process.

The procedure shown in Algorithm 4.11 calculates the ideal position for a pipelining register with respect to the positions that are unfeasible. The algorithm gradually adds the partial propagation delay of each resource and subtracts the sum of delays from the ideal half-cut delay $t_{\text{tot}}/2$. A minimal absolute difference between the ideal half-cut delay and the gradual delay is the best position for the new pipeline register. The algorithm also verifies whether this position is a part of an external edge. If the edge is external, this edge is the best candidate for pipeline register insertion. Otherwise, the algorithm compares neighbouring edges (they are external by definition) and chooses the one with the smaller absolute difference between the ideal half-cut delay and the gradual delay as the best position for pipeline register. A graphic representation is shown in Figure 4.4.

The legal edge list is therefore a list of feasible external edges where it is most effective to insert pipeline registers. The list is sorted according to the total propagation delay t_{tot} of paths these edges are part of. The first edge in the list is part of the most critical path and hence will be pipelined first by the pipelining algorithm. Edges that are part of paths with zero propagation delays are appended to the end of this list in the final step. Therefore, the compiled legal edge list contains external

Input: Paths P extracted from timing results
Output: Legal edge list E_{legal}

```

1 foreach path  $p \in P$  do
2   Calculate ideal half-cut delay of the path  $p$  as  $t_{\text{half}} = t_{\text{tot}}/2$ 
3   foreach resource  $r \in p$  do
4     Store previous delays
5      $t_{\text{grad}} = \sum_{j=1}^r t_{\text{delay},j}$  //  $t_{\text{delay}}$ ...propagation delay of resource  $r$ 
6      $t_{\text{diff}} = |t_{\text{grad}} - t_{\text{half}}|$ 
7     if previous  $t_{\text{diff}} <$  current  $t_{\text{diff}}$  then
8       if previous resource  $r$  is external edge then
9         Add previous  $r$  to  $E_{\text{legal}}$ 
10      else if pre-previous  $t_{\text{diff}} <$  current  $t_{\text{diff}}$  then
11        Add pre-previous  $r$  to  $E_{\text{legal}}$ 
12      else
13        Add current  $r$  to  $E_{\text{legal}}$ 
14      end
15      break // found, no need to continue in this path
16    end
17  end
18 end

```

Algorithm 4.11: Calculate ideal half-cut position.

edges sorted from the most critical to the least critical. The procedure is shown in Algorithm 4.12.

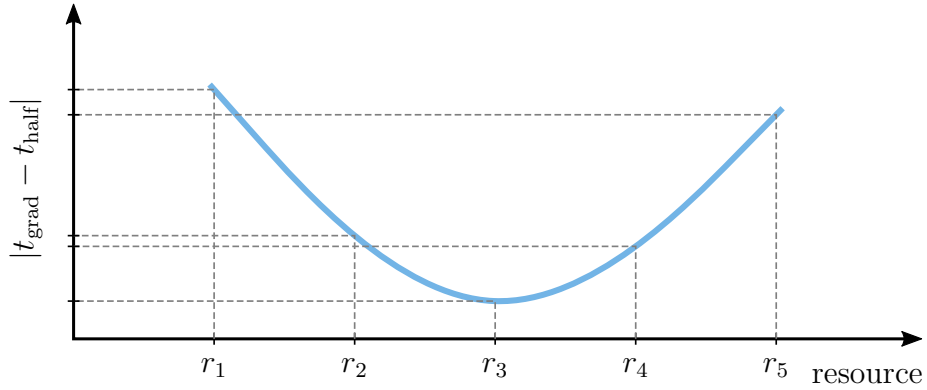


Figure 4.4: Absolute difference between ideal half-cut delay and gradual delay.

| |
|---|
| <p>Input: Legal edge list E_{legal}, list of external edges E_{ext}</p> <p>Output: Sorted legal edge list E_{legal}</p> <pre> 1 Sort resources in E_{legal} by the path's total delay these resources are part of 2 foreach external edge $e_{\text{ext}} \in E_{\text{ext}}$ do 3 if $e_{\text{ext}} \notin E_{\text{legal}}$ then 4 Append e_{ext} to the end of E_{legal} 5 end 6 end </pre> |
|---|

Algorithm 4.12: Finalize legal edge list.

4.3.2 Pipelining Algorithm

The pipeline stage is computed by a modified algorithm proposed by Ganusov et al. [11] introduced in subsection 2.2.4. The main difference is that the algorithm proposed by Ganusov et al. uses a legal pin list instead of legal edge list employed by the modified algorithm proposed in this thesis. The advantage is that the resulting register locations are external edges (i.e. nets) and not vertices (i.e. pins). This makes it more evident for the user where to add the pipeline registers. The proposed algorithm is described in Algorithm 4.13. The process of finding TFI and TFO edges exploits the BFS (Breadth-First Search) algorithm.

| |
|--|
| <p>Input: A circuit $G = \langle V, E, w \rangle$, legal edge list E_{legal}</p> <p>Output: A list of edges (nets) E_{stage}</p> <pre> 1 $E_{\text{stage}} = \{\}, E_{\text{TFI}} = \{\}, E_{\text{TFO}} = \{\}$ 2 foreach edge $e \in E_{\text{legal}}$ do 3 if $(e \in E_{\text{TFI}}) \vee (e \in E_{\text{TFO}})$ then 4 continue 5 end 6 Add e to E_{stage} 7 Find TFI edges of e using BFS algorithm and add them to E_{TFI} 8 Find TFO edges of e using BFS algorithm and add them to E_{TFO} 9 end 10 return E_{stage} </pre> |
|--|

Algorithm 4.13: Computing a pipeline stage.

5 Experimental Results

The script was functionally verified on selected circuits. The time of execution was also measured on a single CPU thread (AMD Ryzen™ 3600). Only the relative execution time of the script related to the number of LUTs is relevant, because absolute values may differ when executed on different systems.

A simple circuit (referred to as *circuit 1*) is shown in Figure 5.1. The script was fed with the netlist of this circuit in an *.edf file format and timing report in an *.twx file format. Results suggest inserting a new pipeline stage consisting of three pipeline registers. Resulting increase in operating frequency f_{\max} as well as execution time is shown in Table 5.1. Waveforms confirming data consistency is shown in Figure A.2.

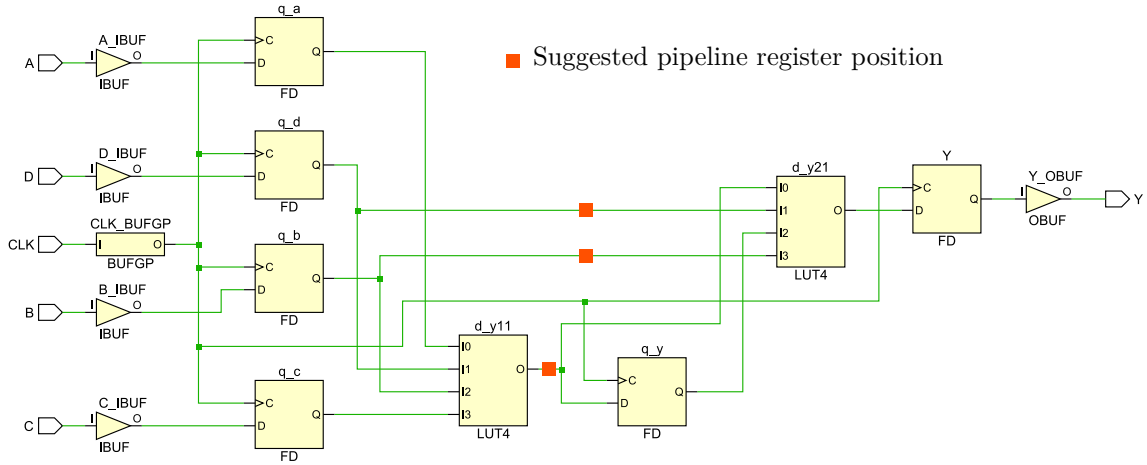


Figure 5.1: Simple design schematic.

An 8-bit multiplier (referred to as *circuit 2*) was also verified (schematic shown in Figure A.1). The script was fed with the netlist of this circuit in an *.ndf file format and timing report in an *.twx file format. The timing report generated 24 paths with timing details. Results suggest inserting a new pipeline stage consisting of 14 pipeline registers. This circuit was not simulated. However, inspection of the pipelined paths and timing report suggests increase in operating frequency f_{\max} by 22.7 %. Resulting increase in operating frequency f_{\max} as well as execution time is shown in Table 5.1.

Several other designs were pipelined using the developed script in order to measure the execution time. Results are summed up in Table 5.1. Execution time as a function of number of LUTs in the design is shown in Figure 5.2. The execution time is linearly dependent on the number of LUTs in the design which makes the script very scalable.

Table 5.1: Pipelining results.

| Name | CPU | LUTs | f_{\max} | | |
|-----------|----------|------|------------|------------|----------|
| | | | Before | After | Increase |
| circuit 1 | 60 ms | 2 | 228.26 MHz | 371.75 MHz | 62.9 % |
| circuit 2 | 230 ms | 45 | 353.61 MHz | 457.67 MHz | 29.4 % |
| circuit 3 | 1,320 ms | 308 | — | — | — |
| circuit 4 | 1,990 ms | 457 | — | — | — |

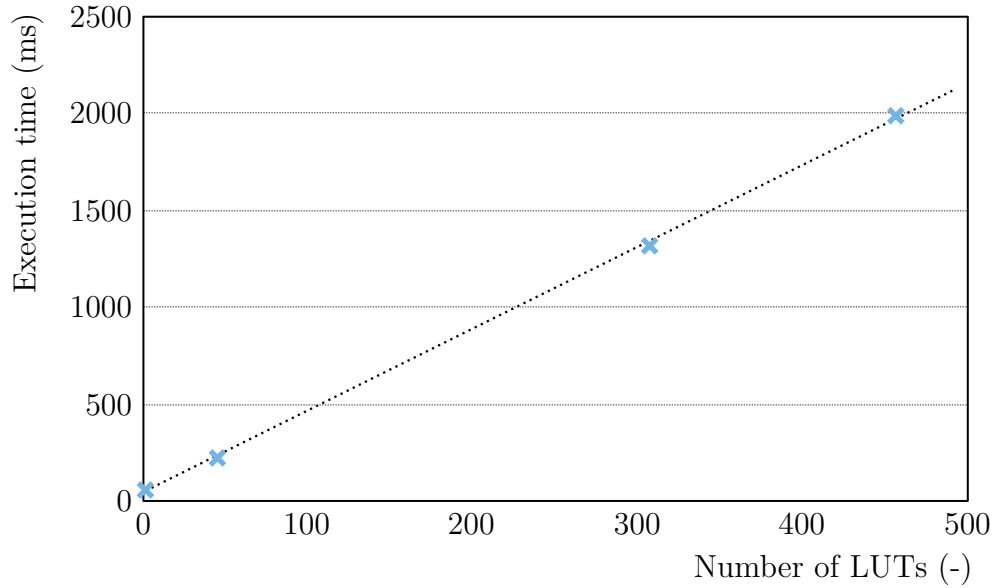


Figure 5.2: Execution time as a function of number of LUTs in the design.

Summary and Reflection

This master's thesis deals with the pipelining of digital designs without any feed-back loops. The thesis exploits a modified algorithm based on the work of Ganusov et al. [11]. Their algorithm is implemented in Xilinx Vivado Design Suite, release 2016.1, and offers great scalability and time complexity of $O(n)$. However, another complementary algorithm for looped circuits could be designed, the theoretical background was already described in subsection 2.2.5. For this to work, all paths need to be labelled either as *feed-forward* or as *looped*. The feed-forward paths then can be pipelined using the algorithm already described in the practical part of this thesis. By contrast, looped paths could be analyzed, labelled, and transformed by one or more of the techniques described in subsection 2.2.5.

Circuits with looped paths are notoriously hard to pipeline. Although this thesis builds a DAG representation of the circuit from available netlist, the ultimate goal is to read and extract the design's VHDL source files and build the circuit from these extracted data. The benefit is that it is much easier to find patterns, such as FSMs, adders, counters, and others, in the design's source files. Structures written in VHDL are usually well-defined and so these patterns are more easily recognized there as opposed to the netlist.

The practical work of this thesis could be optimized on several fronts. For example, some of the used Python libraries could be replaced (namely networkx library for building and storing the DAG) by more optimized alternatives. Many of the used libraries are very well documented which is the reason of their exploitation. On the other hand, the semi-automatic process developed in the thesis is still several orders faster than manual pipelining.

Conclusion

This thesis describes the methodology and algorithms used in pipelining of sequential digital circuits and presents a method for semi-automatic design pipelining.

Tools for the timing results extraction and netlist extraction were created. Tools operate with the formats generated by Xilinx Design Suite ISE. However, support of formats of other design development environments may be easily added which makes this work fairly universal.

A circuit abstraction was introduced. This abstraction models a circuit as a finite DAG Directed Acyclic Graph with vertices representing pins of individual logical elements, and edges representing connections between these pins. Each edge has its weight representing propagation delay. From the timing perspective, this abstraction models circuits simply yet accurately.

An algorithm with linear time complexity used for pipelining feed-forward paths was used. The algorithm employs circuit abstract models and outputs recommended locations where registers creating a pipeline stage should be added. New pipeline stages are created with performance in mind because flip-flops are usually very abundant in modern FPGAs and hence the design area requirements are a less important issue compared to the time requirements.

Finally, selected circuits were pipelined using the developed scripts verifying their function.

Bibliography

- [1] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.
- [2] Alan Cobham. The intrinsic computational difficulty of functions. 1965.
- [3] Jørgen Bang-Jensen and Gregory Z Gutin. *Digraphs: theory, algorithms and applications*. Springer Science & Business Media, 2009. doi:10.1007/978-1-84800-998-1.
- [4] A. Mischenko and R. K. Brayton. Verification after synthesis. In *IWLS*, pages 263–267, Berkley, CA, 2006. Department of EECS, University of California, Berkley. URL: <https://people.eecs.berkeley.edu/~alanmi/publications/>.
- [5] Charles E. Leiserson, Flavio M. Rose, and James B. Saxe. Optimizing synchronous circuitry by retiming (preliminary version). In *Third Caltech Conference on Very Large Scale Integration*, pages 87–116, Berlin, Heidelberg, 1983. Springer Berlin Heidelberg.
- [6] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, Jun 1991. doi:10.1007/BF01759032.
- [7] Narendra Shenoy and Richard Rudell. *Efficient Implementation of Retiming*, pages 615–630. Springer US, Boston, MA, 2003. doi:10.1007/978-1-4615-0292-0_49.
- [8] A. P. Hurst, A. Mishchenko, and R. K. Brayton. Fast minimum-register retiming via binary maximum-flow. In *Formal Methods in Computer Aided Design (FMCAD’07)*, pages 181–187. IEEE, 2007. doi:10.1109/FAMCAD.2007.31.
- [9] Andreas Kuehlmann and Jason Baumgartner. Transformation-based verification using generalized retiming. In *Computer Aided Verification*, pages 104–117, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
- [10] James B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Operations Research*, 41(2):338–350, 1993. doi:10.1287/opre.41.2.338.
- [11] I. Ganusov, H. Fraise, A. Ng, R. T. Possignolo, and S. Das. Automated extra pipeline analysis of applications mapped to Xilinx UltraScale+ FPGAs. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–10, 2016. doi:10.1109/FPL.2016.7577344.

- [12] Lok-Won Kim, Dong-U Lee, and John Villasenor. Automated iterative pipelining for ASIC design. *ACM Transactions on Design Automation of Electronic Systems*, 20(2), March 2015. doi:10.1145/2660768.
- [13] E. Salman, A. Dasdan, F. Taraporevala, K. Kucukcakar, and E. G. Friedman. Exploiting setup–hold-time interdependence in static timing analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(6):1114–1125, 2007. doi:10.1109/TCAD.2006.885834.
- [14] Maria-Cristina V. Marinescu and Martin Rinard. High-level automatic pipelining for sequential circuits. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, page 215–220, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/500001.500053.
- [15] M. Galceran-Oms, J. Cortadella, D. Bufistov, and M. Kishinevsky. Automatic microarchitectural pipelining. In *2010 Design, Automation Test in Europe Conference Exhibition*, pages 961–964. IEEE, 2010. doi:10.1109/DATE.2010.5456910.
- [16] J. P. L. Campbell and N. A. Day. High-level optimization of pipeline design. In *Eighth IEEE International High-Level Design Validation and Test Workshop*, pages 43–48, 2003. doi:10.1109/HLDVT.2003.1252473.
- [17] Daniel Kroening and Wolfgang J Paul. Automated pipeline design. In *Proceedings of the 38th annual Design Automation Conference*, pages 810–815, 2001.
- [18] S. Bard and N. I. Rafla. Reducing power consumption in FPGAs by pipelining. In *2008 51st Midwest Symposium on Circuits and Systems*, pages 173–176. IEEE, 2008. doi:10.1109/MWSCAS.2008.4616764.
- [19] Eduardo Boemo, Juan P. Oliver, and Gabriel Caffarena. Tracking the pipelining-power rule along the FPGA technical literature. In *Proceedings of the 10th FPGAworld Conference*, FPGAworld '13, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2513683.2513692.
- [20] J. Lamoureux and S. J. E. Wilton. On the interaction between power-aware FPGA CAD algorithms. In *ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No.03CH37486)*, pages 701–708. IEEE, 2003. doi:10.1109/ICCAD.2003.159755.
- [21] Steven J. E. Wilton, Su-Shin Ang, and Wayne Luk. The impact of pipelining on energy per operation in field-programmable gate arrays. In *Field Programmable*

- Logic and Application*, pages 719–728, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. doi:10.1007/978-3-540-30117-2_73.
- [22] Xilinx. Vivado Design Suite 2016.1, 2016. [Accessed 11-May-2021]. URL: <https://www.xilinx.com/products/design-tools/vivado.html>.
 - [23] H. Omidian and G. G. F. Lemieux. Low-level loop analysis and pipelining of applications mapped to Xilinx FPGAs. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 391–396, 2019. doi:10.1109/FPL.2019.00068.
 - [24] J. Liu, J. Wickerson, and G. A. Constantinides. Loop splitting for efficient pipelining in high-level synthesis. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 72–79. IEEE, 2016. doi:10.1109/FCCM.2016.27.
 - [25] Nicholas Weaver, Yury Markovskiy, Yatish Patel, and John Wawrzynek. Post-placement C-slow retiming for the Xilinx Virtex FPGA. In *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, FPGA ’03, page 185–194, New York, NY, USA, 2003. Association for Computing Machinery. doi:10.1145/611817.611845.
 - [26] Xilinx. ISE Design Suite 14.7, 2013. [Accessed 18-April-2018]. URL: <https://www.xilinx.com/products/design-tools/ise-design-suite.html>.
 - [27] Python Software Foundation. Python 3.8, 2019. [Accessed 5-October-2020]. URL: <https://www.python.org/>.
 - [28] Command Line Tools User Guide. Xilinx, 2013. [Online; accessed 13-February-2020]. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/devref.pdf.
 - [29] Beautiful Soup Documentation. Leonard Richardson, 2020. [Online; accessed 18-March-2020]. URL: <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>.

Symbols and abbreviations

| | |
|-------------|---|
| ASIC | Application Specific Integrated Circuit |
| BFS | Breadth-First Search |
| CAD | Computer-Aided Design |
| CPU | Central Processing Unit |
| DAG | Directed Acyclic Graph |
| DSP | Digital Signal Processor |
| FPGA | Field Programmable Gate Array |
| FSM | Finite-State Machine |
| GUI | Graphical User Interface |
| HLS | High-Level Synthesis |
| HTML | Hypertext Markup Language |
| IC | Integrated Circuit |
| IDE | Integrated Development Environment |
| ISA | Instruction Set Architecture |
| LUT | Lookup Table |
| MCU | Microcontroller Unit |
| PaR | Place and Route |
| PI | Primary Input |
| PO | Primary Output |
| RTL | Register Transfer Logic |
| TCL | Tool Command Language |
| TFI | Transitive Fanin |
| TFO | Transitive Fanout |
| VHDL | VHSIC-HDL, Very High Speed Integrated Circuit Hardware Description Language |

A Schematics and waveforms

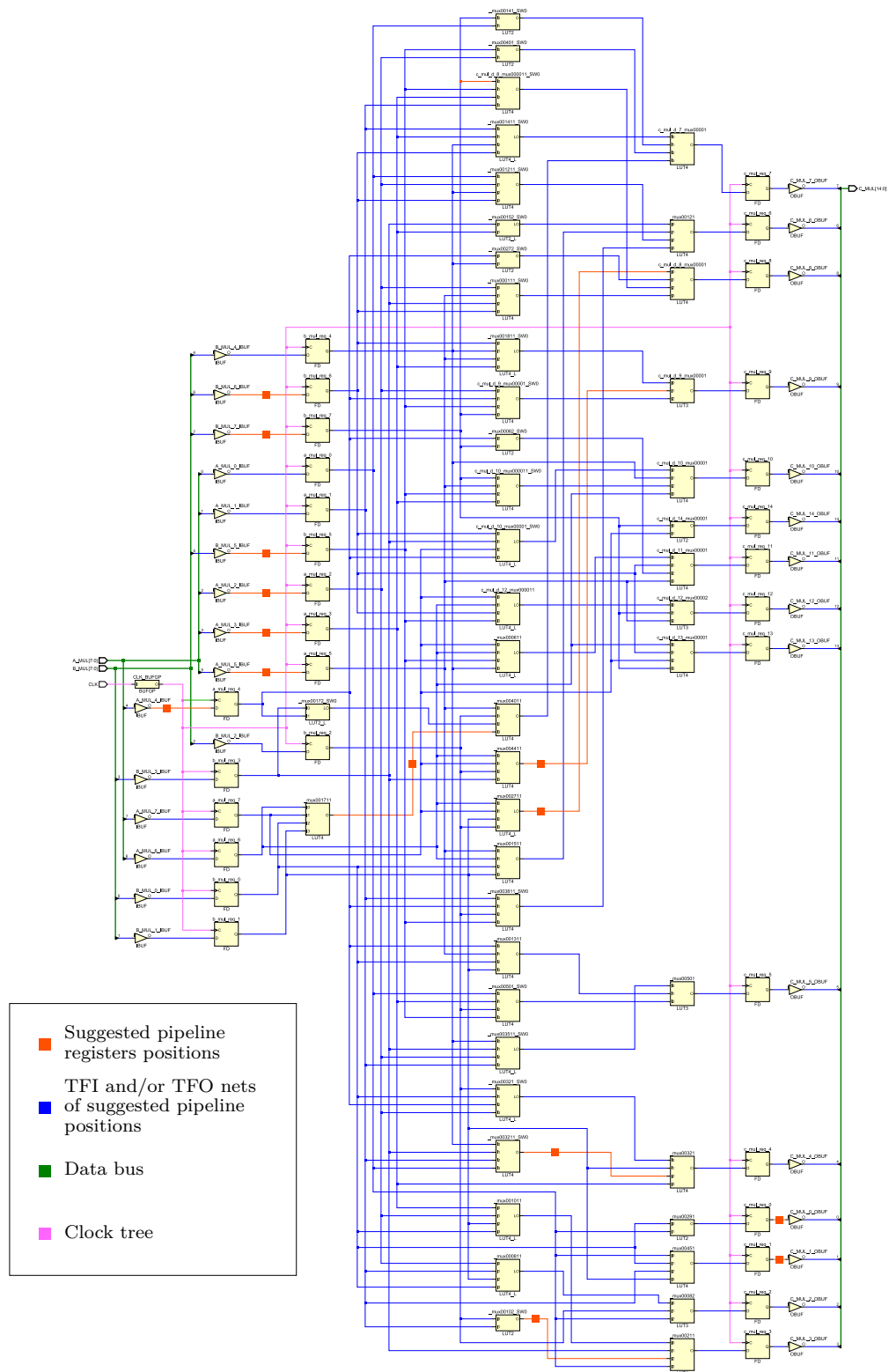


Figure A.1: Pipelined 8-bit multiplier.

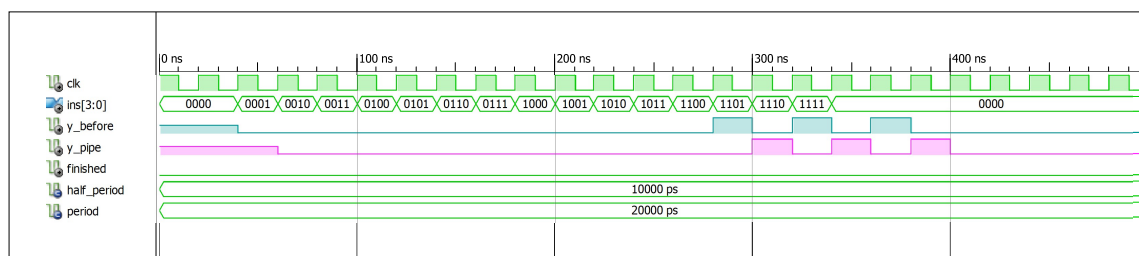


Figure A.2: Waveforms of a simple circuit before and after pipelining.

B List of Used Python Libraries

Table B.1: List of used Python libraries

| Library name | Documentation |
|-----------------------------|---|
| <code>beautifulsoup4</code> | https://pypi.org/project/beautifulsoup4/ |
| <code>collections</code> | https://docs.python.org/3/library/collections.html |
| <code>copy</code> | https://docs.python.org/3/library/copy.html |
| <code>glob</code> | https://docs.python.org/3/library/glob.html |
| <code>matplotlib</code> | https://matplotlib.org/ |
| <code>networkx</code> | https://networkx.org/ |
| <code>numpy</code> | https://numpy.org/ |
| <code>os</code> | https://docs.python.org/3/library/os.html |
| <code>re</code> | https://docs.python.org/3/library/re.html |
| <code>sys</code> | https://docs.python.org/3/library/sys.html |
| <code>time</code> | https://docs.python.org/3/library/time.html |

C Content of Enclosed Data Medium

```
/ ..... root directory
├── DP_Houska_170858.pdf ..... pdf file of the thesis
├── readme.txt ..... eternal wisdom is stored here
└── src
    ├── example_designs ..... vhdl examples source files
    │   ├── basic_rtl
    │   │   ├── top_basic_rtl.vhd
    │   │   ├── top_basic_rtl.twx
    │   │   ├── top_basic_rtl.syr
    │   │   └── top_basic_rtl.edf
    │   ├── multiplier
    │   │   ├── top_multi.vhd
    │   │   ├── top_multi.twx
    │   │   ├── top_multi.syr
    │   │   └── top_multi.ndf
    └── script ..... python script source files
        ├── files.py
        ├── graph.py
        ├── main.py
        ├── process_file.py
        └── settings.py
```