Hardware Acceleration for Quantum Decoding

A Dissertation Submitted in Partial Fulfilment of the Requirements

for the Degree of

Master of Technology

by

Sai Pavan Krishna Nagam

01-02-04-10-51-24-1-24071

under the guidance of

Prof. Utsav Banerjee

DEPARTMENT OF

INSTRUMENTATION AND APPLIED PHYSICS

INDIAN INSTITUTE OF SCIENCE

BENGALURU - 560012

May 2026

DECLARATION

I, **Sai Pavan Krishna Nagam (01-02-04-10-51-24-1-24071)**, hereby declare that the work presented in this M.Tech thesis entitled **"Hardware Acceleration for Quantum Decoding"** is the result of the work performed by me in the **Department of Instrumentation and Applied Physics, Indian Institute of Science, Bengaluru**, under the supervision of **Prof. Utsav Banerjee** and that it has not been submitted elsewhere for any degree or diploma. In keeping with the general practice, due acknowledgements are made wherever the work is based on findings of other investigations.

Bengaluru - 560012 **Sai Pavan Krishna Nagam**

May 2026

CERTIFICATE

This is to certify that the work contained in this project report entitled **"Hardware Acceleration for Quantum Decoding"**, submitted by **Sai Pavan Krishna Nagam**, towards the partial requirement of **Master of Technology** in **Quantum Technology**, has been carried out by him under my supervision at the Department of Instrumentation and Applied Physics, Indian Institute of Science, Bengaluru, and that no part of it has been previously submitted for a degree, diploma, or any other qualification at this university or any other institution.

Bengaluru- 560012 — Prof. Utsav Banerjee

May 2026 — Project Supervisor

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my thesis guide, Prof. Utsav Banerjee, for his constant support, encouragement, and guidance throughout this work. I am especially thankful for the freedom he provided to explore ideas independently, continuously learn, and grow both academically and professionally. I also deeply appreciate his guidance beyond the scope of this thesis toward my future endeavors and opportunities.

I would also like to thank the SINESys Lab for their constant support whenever I was stuck — whether it was debugging issues, setting up hardware, helping me find resources, or contributing through collaborative discussions during team meetings. The learning environment within the lab greatly enriched this journey.

I am grateful to Prof. Baladitya Suri, my program advisor, for his mentorship and valuable advice regarding coursework, academic direction, and opportunities throughout the program.

I would also like to thank IQTI and QuRP for providing the necessary tools, infrastructure, and resources required for conducting my research. The talks, workshops, conclaves, and fests organized through these

initiatives made learning beyond academics engaging and helped me understand the field from a broader perspective.

I would also like to acknowledge the availability of modern AI-assisted tools such as Anthropic's Claude, OpenAI's ChatGPT, Google Gemini, and GitHub Copilot, which were valuable in accelerating code generation, debugging, resource discovery, visualizations, and brainstorming during the course of this work. These tools enabled me to spend more time learning, exploring ideas, and designing solutions rather than being limited by implementation overhead.

Finally, I would like to acknowledge the Ministry of Education, Government of India, for providing financial support through the M.Tech stipend.

Sai Pavan Krishna Nagam

ABSTRACT

A fault-tolerant quantum computer requires a classical decoder that is both accurate enough to suppress logical errors and fast enough to keep pace with the quantum error-correction cycle. This thesis investigates three major surface-code decoder families—Minimum-Weight Perfect Matching (MWPM), Union–Find (UF), and Belief Propagation (BP)—from this combined algorithmic and hardware perspective. The work integrates two study streams: a kernel-level RTL, ASIC, FPGA, and optimisation study, and a distance-scaling study that measures decoding quality, latency, resource utilisation, maximum frequency, and a High-Level Synthesis (HLS) baseline on the Xilinx ZCU104.

The algorithmic layer uses Stim-generated rotated surface-code experiments, PyMatching, and `ldpc`-based BP variants to evaluate logical error rate across code distances $d = 3, 5, \ldots, 31$. Under depolarising code-capacity noise, MWPM exhibits a clean threshold crossing near $p \approx 15\%$, consistent with published estimates; Union–Find tracks MWPM within a small constant factor; and plain min-sum BP fails to suppress logical error rate with increasing distance because of degeneracy and short cycles in the surface-code Tanner graph. Order-0 ordered-statistics post-processing restores BP performance, bringing BP+OSD close to MWPM. Under circuit-level noise, weighted Union–Find improves the high-error-regime logical error rate relative to unweighted growth.

The hardware layer profiles the decoder implementations, identifies the dominant computational kernels, and implements them in synthesisable Verilog-2005. The kernels are verified against independent golden models and then evaluated through a 45 nm Nangate Open Cell ASIC synthesis flow, a PYNQ-compatible ZCU104 FPGA implementation, distance-scaled Vivado synthesis, and a Vitis HLS baseline. The results show that BP processing-element replication converts almost directly into wall-clock speed-up, while naively collapsing MWPM frontier extraction or Union–Find parent traversal into a single combinational cycle can reduce cycle count but destroy achievable frequency. The central conclusion is therefore structural: decoder accuracy is moved by algorithmic strategies such as OSD and weighted growth, whereas hardware speed is moved by carefully structured parallelism, pipelining, and memory mapping rather than by cycle-count reduction alone.

Keywords: Quantum Error Correction, Surface Codes, Decoder Hardware Acceleration, RTL, FPGA, ASIC, High-Level Synthesis, MWPM, Union–Find, Belief Propagation, OSD.

DEDICATION

To the person I became through this journey.

To myself, for persevering through a highly interdisciplinary field and navigating a domain completely new to me with persistence, patience, and trust in the process.

To my family, for their unwavering support and encouragement throughout this journey. Special thanks to my sister for always listening, supporting me through difficult moments, and motivating me to keep moving forward.

And lastly, to my friends, for their constant encouragement and for bringing lightness to this journey.

Contents

List of Figures xv

List of Tables xviii

Abbreviations xx

1 Introduction 1

1.1 The Quantum Decoding Problem 1

1.2 The Real-Time Wall 2

1.3 Two Questions, One Study 3

1.4 Scope of this Thesis 3

1.5 Thesis Organisation 5

2 Quantum Error Correction 7

2.1 Surface Codes 8

2.2 Decoding Pipeline 9

2.2.1 Syndrome Extraction 9

2.2.2 Decoding Graph Construction 10
2.2.3 Applying Correction 10
2.3 Decoder Algorithms 10
2.3.1 Minimum-Weight Perfect Matching 11
2.3.2 Union–Find 13
2.3.3 Belief Propagation 15
2.4 Performance Metrics 16
2.4.1 Latency 17
2.4.2 Logical Error Rate 18
2.4.3 Scalability 18

3 Hardware System Design **20**
3.1 Hardware Design Methodology 20
3.2 Register Transfer Level (RTL) Design 21
3.3 High-Level Synthesis (HLS) 22
3.4 Functional Verification 24
3.5 Tools and Development Stack 25
3.6 Performance Metrics 26
3.6.1 Timing Performance 26
3.6.2 Resource Utilisation 27
3.6.3 Area and Power 28

4 Efficient Hardware Decoder Mappings **30**

4.1 Union–Find in Hardware 30
4.2 Matching-Based Decoding in Hardware 32
4.3 Belief Propagation in Hardware 33
4.4 Summary of Hardware Mapping Strategies 34

5 Methodology **35**
5.1 Overview of the Evaluation Flow 35
5.2 Decoder Variants Considered 36
5.3 Software Profiling and Algorithmic Analysis 37
5.4 Surface-Code Simulation Framework 38
5.5 Golden Reference Model 39
5.6 RTL Design and Functional Verification 39
5.7 ASIC-Oriented Hardware Flow 40
5.8 FPGA and High-Level Synthesis Flow 41
5.9 Performance Evaluation Metrics 42

6 Kernel Identification, RTL Design, and Verification **44**
6.1 Profiling-Based Kernel Selection 45
6.2 RTL Targets and Strategy Knobs 46
6.3 Common RTL Design Conventions 47
6.3.1 Start–Done Interface 48
6.3.2 Fixed-Point Format 48
6.3.3 Graph Representation 48

6.4 MWPM Hardware Architecture 49
6.4.1 Dijkstra Shortest-Path Kernel 49
6.4.2 Bitmask Dynamic-Programming Matcher 50
6.5 Union–Find Hardware Architecture 50
6.5.1 Disjoint-Set Engine 50
6.5.2 Cluster Growth and Peeling 51
6.6 Belief-Propagation Hardware Architecture 51
6.7 Functional Verification 52

7 Hardware Implementation, Synthesis, and Optimisation Results 54

7.1 ASIC Synthesis Flow 55
7.2 Baseline ASIC Results 55
7.3 ZCU104 FPGA Implementation 57
7.3.1 AXI4-Lite Register Map 57
7.3.2 Functional FPGA Sweep 58
7.4 Post-Implementation FPGA Results 58
7.5 Kernel Optimisation Study 60
7.5.1 Optimisations Applied 60
7.6 Chapter Summary 63

8 Decoding Quality, Distance Scaling, and HLS Comparison 64

8.1 Decoding-Quality Study 64

8.1.1 Logical Error Rate versus Code Distance 65
8.1.2 Threshold Behaviour 66
8.1.3 Weighted Union–Find under Circuit-Level Noise . 67
8.2 RTL Latency versus Code Distance 67
8.3 FPGA Resource Utilisation and Maximum Frequency . . 69
8.4 Hand-Written RTL versus Vitis HLS 71
8.5 Chapter Summary 73

9 Conclusion and Future Work **74**
9.1 Summary of Contributions 74
9.2 Main Findings 75
9.3 Limitations 76
9.4 Future Work 77
9.5 Closing Remark 78

Appendices **80**

A Stabilizers, Syndromes, and the Decoding Graph **80**
A.1 Stabilizer Formalism 80
A.2 Syndromes and the Detection-Event Picture 82

B Error Models and Threshold Estimates **84**
B.1 Code-Capacity Noise Model 84
B.2 Phenomenological and Circuit-Level Noise 85

B.3 Known Threshold Estimates 85

C Decoder Algorithms (Pseudocode) 87

D Representative RTL Listings 90

D.1 MWPM Dijkstra Kernel (Frontier Reduction) 90

D.2 Union–Find Disjoint-Set Engine (HOPS Parameter) . . 91

D.3 BP Min-Sum Check-Node Update (NPE Replication) . . 92

D.4 AXI4-Lite Wrapper Register Map (Common to All PYNQ Cores) 93

E Reproducibility Notes 95

F Finite-State Machine Diagrams for the RTL Kernels 97

Bibliography 99

List of Figures

2.1 Rotated surface-code lattice 8
2.2 Decoding pipeline 9
2.3 MWPM decoding workflow 12
2.4 Union–Find decoding workflow 14
2.5 BP decoding workflow 15

6.1 Runtime bottleneck profile of the three decoder families. The hardware study targets the routines that dominate per-shot execution. 46

7.1 Baseline ASIC characterisation of the three complete decoder tops. MWPM is the largest design, Union–Find is intermediate, and BP is the most compact under the 45 nm standard-cell flow. 56

7.2 ZCU104 utilisation and achieved frequency for the PYNQ-wrapped decoder cores. All three designs fit comfortably, but none closes the aggressive 250 MHz target after full implementation. 59
7.3 Cycle-count speed-up compared with net wall-clock speed-up. The difference between the two exposes optimisations whose critical-path cost cancels their cycle benefit. . . . 62
7.4 Critical-path depth of the baseline and optimised kernels. The 904-stage Dijkstra spike corresponds to an over-aggressive unpipelined parallel-relaxation cone. 62

8.1 LER versus code distance under code-capacity noise . . 65
8.2 LER versus physical error rate 67
8.3 Weighted UF under circuit-level noise 68
8.4 RTL decode latency in clock cycles versus code distance. BP processing-element replication scales nearly linearly; Union–Find multi-hop traversal gives only modest cycle reduction; MWPM parallel extraction dramatically reduces cycle count. 68
8.5 ZCU104 post-synthesis resource utilisation versus code distance. BP grows rapidly when synthesised as one flat whole-code instance; MWPM and Union–Find remain modest over the measured range. 69

8.6 Post-synthesis $F_{\max}$ at $d = 15$. The MWPM combinational extraction and the multi-hop Union–Find traversal reduce cycle count but lengthen the critical path substantially. 70

8.7 Area comparison between the best hand-written RTL configurations and Vitis HLS at $d = 11$. HLS infers block RAM for memories that the current hand RTL maps into registers or LUT storage. 72

F.1 Top-level FSM of the MWPM decoder kernel (Dijkstra plus bitmask matching DP). 97

F.2 Top-level FSM of the Union–Find decoder kernel (cluster growth, union, peeling). 98

F.3 Top-level FSM of the BP min-sum decoder kernel (check-node, variable-node, convergence check). 98

List of Tables

4.1 Mapping of prior hardware-decoder literature to the optimisation strategies evaluated in this work. 34

5.1 Decoder variants evaluated in this study. 37

6.1 Measured runtime split used for kernel selection at $d = 7$. The dominant routines (bold) become the RTL targets. 45

6.2 RTL modules and architectural strategy knobs used in the thesis. 47

6.3 RTL verification coverage for the kernel-level implementation study. 52

7.1 Baseline ASIC synthesis results using the Nangate 45 nm library. 56

7.2 Common AXI4-Lite register map used by the PYNQ decoder wrappers. 58

7.3 Post-implementation ZCU104 results for the PYNQ cores 59

7.4 Kernel optimisation results: 45 nm area, timing depth, and net wall-clock speed-up. 61

8.1 Decode latency endpoints from the distance sweep. The speed-up is baseline divided by the fastest strategy at $d = 31$. 68

8.2 ZCU104 post-synthesis results at $d = 15$. Wall-clock time is $T = \text{cycles}/F_{\max}$. 70

8.3 Hand-written RTL versus Vitis HLS at $d = 11$ on the ZCU104. 71

B.1 Reference threshold estimates for the rotated surface code. 85

Abbreviations

Acronym	Expansion
ABC	A System for Sequential Synthesis and Verification (logic synthesis engine used in Yosys)
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface (ARM AMBA bus protocol)
BP	Belief Propagation
BP+OSD	Belief Propagation with Ordered Statistics Decoding post-processing
BRAM	Block Random-Access Memory (FPGA primitive)
CNOT	Controlled-NOT gate
CSR	Compressed Sparse Row (graph storage format)
DP	Dynamic Programming
DSP	Digital Signal Processing (FPGA primitive)
DSU	Disjoint-Set Union data structure
FF	Flip-Flop
FPGA	Field-Programmable Gate Array
FSM	Finite-State Machine
HDL	Hardware Description Language
HLS	High-Level Synthesis
IP	Intellectual Property (in FPGA/ASIC context, a reusable hardware module)
LDPC	Low-Density Parity-Check (code family)
LER	Logical Error Rate

Acronym	Expansion
LLR	Log-Likelihood Ratio
LUT	Look-Up Table (FPGA primitive)
MWPM	Minimum-Weight Perfect Matching
NMS	Normalised Min-Sum (BP variant)
NPE	Number of Processing Elements
OSD	Ordered Statistics Decoding
PE	Processing Element
PL	Programmable Logic (FPGA fabric of Zynq SoC)
PS	Processing System (ARM cores of Zynq SoC)
PVT	Process–Voltage–Temperature
PYNQ	Python productivity for Zynq
QEC	Quantum Error Correction
QECC	Quantum Error-Correcting Code
RTL	Register Transfer Level
SoC	System on Chip
SRAM	Static Random-Access Memory
UF	Union–Find
URAM	UltraRAM (UltraScale+ FPGA memory primitive)
VHDL	VHSIC Hardware Description Language
WNS	Worst Negative Slack

Symbols and Notation

Symbol	Meaning
d	Code distance of the surface code
p	Physical error rate (per data-qubit per cycle)
p_L	Logical error rate per round (LER)
p_{th}	Threshold physical error rate of the code+decoder pair
n	Number of physical qubits in a code block
k	Number of encoded logical qubits in a code block
$[\![n, k, d]\!]$	Quantum stabilizer code with n physical qubits, k logical qubits, distance d
$\mathcal{S}$	Stabilizer subgroup of the Pauli group
$\mathcal{N}(\mathcal{S})$	Normaliser of the stabilizer group (logical Pauli operators)
H	Parity-check matrix
$\mathbf{s}$	Syndrome vector
$\mathbf{e}$	Error vector (Pauli string)
w	Edge weight $w_e = \log\big((1-p_e)/p_e\big)$
α	Min-sum normalisation factor (set to 0.75 in this work)
$F_{\max}$	Maximum achievable clock frequency after synthesis
T	Wall-clock latency, $T = \text{cycles}/F_{\max}$
N	Number of detection events or graph nodes
Λ	Error-suppression factor per distance increment

Physical Constants

Quantity	Symbol	Value
Planck constant (reduced)	$\hbar$	$1.054\,571\,817 \times 10^{-34}$ J·s
Boltzmann constant	k_B	$1.380\,649 \times 10^{-23}$ J/K
Electron volt	eV	$1.602\,176\,634 \times 10^{-19}$ J
Speed of light in vacuum	c	$2.997\,924\,58 \times 10^{8}$ m/s (exact)

Chapter 1

Introduction

1.1 The Quantum Decoding Problem

Quantum computers are inherently susceptible to noise arising from decoherence, imperfect gate operations, and measurement errors. To enable reliable large-scale quantum computation, quantum error correction (QEC) encodes a logical qubit across multiple physical qubits such that errors can be detected and corrected without directly measuring the quantum information itself [1], [2], [3], [4], [5]. Among the various QEC schemes, the surface code descended from Kitaev's toric code has emerged as the leading architecture due to its compatibility with two-dimensional nearest-neighbour hardware layouts and its comparatively high error threshold [6], [7].

In the surface code, stabilizer measurements are repeatedly performed to detect the presence of errors. Changes in stabilizer outcomes between consecutive rounds generate *detection events*, collectively referred to as the syndrome. A classical decoder processes this syndrome and determines the most probable correction operation to apply. Since the decoder only

observes the syndrome and not the underlying physical errors directly, decoding is fundamentally a probabilistic inference problem [4].

Several decoder families have been proposed for surface codes, with Minimum-Weight Perfect Matching (MWPM), Union–Find (UF), and Belief Propagation (BP) being among the most widely studied [8], [9], [10]. These approaches differ significantly in decoding accuracy, computational complexity, and suitability for hardware acceleration.

1.2 The Real-Time Wall

In a fault-tolerant quantum computer, decoding is not an offline computation but part of the real-time execution loop. Modern superconducting quantum processors perform QEC cycles on the order of $1\,\mu\mathrm{s}$, continuously generating fresh syndrome data [11]. If the decoder cannot process syndromes at the rate they are produced, the backlog of unprocessed data grows without bound, eventually causing an exponential slowdown of the logical clock [12], [13].

This stringent latency requirement has transformed quantum decoding into a hardware systems problem. General-purpose processors often struggle to satisfy the timing demands associated with large-distance surface codes, motivating the development of FPGA- and ASIC-based decoders capable of exploiting parallelism and custom datapaths [13], [14], [15], [16]. Consequently, the practical usefulness of a decoder now depends not only on its algorithmic performance but also on how efficiently it can be mapped onto hardware.

1.3 Two Questions, One Study

Surface-code decoders are typically evaluated along two major dimensions: decoding quality and hardware efficiency.

Decoding quality is commonly measured using the *logical error rate* (LER), which represents the probability that a logical failure remains after error correction. A strong decoder suppresses logical errors rapidly as the code distance increases below the threshold error rate [7].

Hardware efficiency is measured through metrics such as decode latency, resource utilisation, and achievable operating frequency. A decoder with low latency and efficient resource usage is essential for real-time fault-tolerant operation.

These two objectives are often competing. Highly accurate algorithms may introduce substantial computational overhead, while highly optimised hardware implementations may sacrifice decoding performance. This thesis therefore studies both aspects together by analysing three major decoder families — MWPM, UF, and BP — from algorithmic as well as hardware implementation perspectives.

1.4 Scope of this Thesis

This work investigates the trade-offs between decoding performance and hardware efficiency for surface-code decoders implemented on FPGA and ASIC-oriented hardware platforms. The study is organised as a two-layer evaluation framework combining algorithmic analysis with hardware implementation and profiling.

The first layer evaluates decoding quality through Monte Carlo simu-

lations of rotated surface codes using the Stim simulator together with software decoding libraries such as PyMatching and the `ldpc` package [8], [17], [18]. Logical error rates are analysed across multiple code distances and physical error rates for different decoder variants.

In addition to decoding-quality evaluation, algorithmic profiling is performed on CPU-based baseline implementations of each decoder family to identify the computational bottleneck subroutines that dominate execution time. These bottleneck kernels are then isolated and targeted for hardware acceleration and architectural optimisation.

The second layer investigates hardware implementation. The dominant computational kernels of each decoder are implemented as parameterised and synthesisable Verilog-2005 RTL designs. Their latency, resource utilisation, and maximum operating frequency are evaluated through cycle-accurate simulation (Icarus Verilog) and out-of-context FPGA synthesis (Vivado 2025.2) targeting the Xilinx Zynq UltraScale+ ZCU104 platform [19]. To extend the study beyond FPGA mapping, the same RTL implementations are also synthesised using Yosys [20] with the Nangate 45 nm Open Cell Library [21] to analyse gate-level area and timing characteristics.

Furthermore, equivalent implementations are developed using Vitis HLS 2025.2 to compare hand-written RTL against HLS-generated hardware [22], [23]. The study therefore investigates the relationship between decoding performance, hardware efficiency, architectural optimisation, and hardware development methodology across multiple implementation flows.

The scope of this work is limited to synthesis-level evaluation and

functional verification. FPGA place-and-route, bitstream generation, and physical ASIC backend design are outside the scope of this thesis.

1.5 Thesis Organisation

The remainder of this thesis is organised as follows:

- **Chapter 2: Quantum Error Correction** - Introduces surface codes, the decoding pipeline, major decoding algorithms, and the performance metrics associated with quantum decoding.
- **Chapter 3: Hardware System Design** - Discusses RTL design, High-Level Synthesis (HLS), verification methodology, development tools, and hardware performance metrics.
- **Chapter 4: Literature Review on Efficient Hardware Mapping of Decoders** - Reviews existing FPGA- and ASIC-based implementations and optimisation strategies for quantum decoders.
- **Chapter 5: Methodology** - Presents the two-layer evaluation framework, including decoding simulations, profiling methodology, hardware implementation flow, and synthesis setup.
- **Chapter 6: Kernel Identification, RTL Design, and Verification** - Presents the profiling results, selected decoder kernels, common RTL conventions, per-kernel microarchitectures, and verification methodology.
- **Chapter 7: Hardware Implementation, Synthesis, and Optimisation Results** - Presents the Study-1 ASIC synthesis, PYNQ

FPGA implementation, and kernel-optimisation results.

- **Chapter 8: Decoding Quality, Distance Scaling, and HLS Comparison** - Integrates the Study-2 logical-error-rate, RTL latency, FPGA resource, maximum-frequency, and HLS baseline results.
- **Chapter 9: Conclusion and Future Work** - Summarises the contributions, limitations, and future directions for scalable hardware decoders.

Chapter 2

Quantum Error Correction

Quantum computers are highly susceptible to decoherence, gate imperfections, measurement noise, and environmental interactions. Unlike classical bits, qubits cannot be copied directly due to the no-cloning theorem, making conventional redundancy-based error correction inapplicable [3], [5]. Quantum Error Correction (QEC) addresses this challenge by encoding logical quantum information across multiple physical qubits using the *stabilizer formalism* [1], [2]: errors are detected indirectly through commuting measurements of stabilizer generators, and the recovery operation is chosen so that the encoded logical state is preserved without ever measuring the encoded information itself. Appendix A reviews this formalism in more detail.

Among the various QEC architectures, the surface code – a planar variant of Kitaev's toric code [6] – has emerged as one of the most practical approaches for scalable fault-tolerant quantum computation due to its high threshold and compatibility with nearest-neighbour hardware layouts [4], [7].

2.1 Surface Codes

Surface codes arrange physical qubits on a two-dimensional lattice consisting of data qubits and ancilla qubits. Stabilizer measurements are repeatedly performed using ancilla qubits to detect bit-flip and phase-flip errors occurring on nearby data qubits.

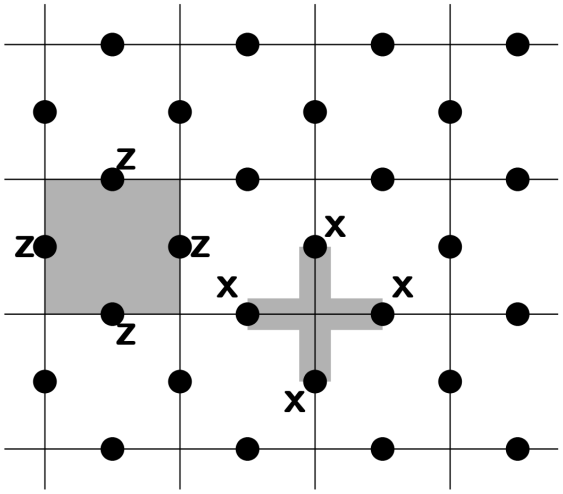

Figure 2.1: Illustration of a rotated surface-code lattice showing plaquette-based Z-type stabilizers (shaded square) and vertex-based X-type stabilizers (shaded cross). Data qubits sit at lattice vertices, while ancilla qubits (not shown) are placed at face centres and are used to measure each stabilizer [7], [24].

The surface code is particularly attractive because every stabilizer involves at most four neighbouring data qubits, so a planar layout suffices and the protocol matches the nearest-neighbour connectivity of superconducting transmon arrays [7], [24]. The code distance, denoted by d, is the minimum weight of a non-trivial logical operator: at least $\lceil (d+1)/2 \rceil$ correlated physical errors must occur before a logical fault is possible. Increasing d improves error suppression at the cost of $\mathcal{O}(d^2)$ additional physical qubits.

A stabilizer measurement cycle produces syndrome information indicating whether an error likely occurred between successive rounds. These syndrome measurements form the basis of the decoding pipeline.

2.2 Decoding Pipeline

The decoding pipeline converts raw stabilizer measurements into corrective operations applied to the quantum state. A typical decoding cycle consists of syndrome extraction, decoding graph construction, decoding, and correction application.

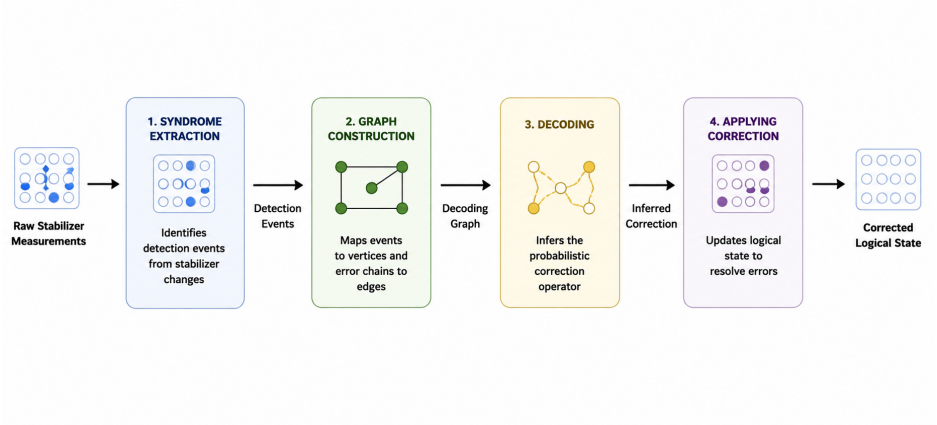

Figure 2.2: The four-stage decoding pipeline used throughout the thesis: stabilizer (syndrome) extraction, decoding-graph construction, decoder execution (MWPM / UF / BP), and application of the inferred Pauli correction to the logical state.

2.2.1 Syndrome Extraction

Syndrome extraction is performed by repeatedly measuring stabilizer operators using ancilla qubits. A change in stabilizer outcome between consecutive rounds produces a *detection event*, indicating that an error may have occurred nearby in space-time [4].

These detection events collectively form the syndrome observed by the decoder. Since the decoder does not directly observe the underlying physical error, decoding becomes a probabilistic inference problem.

2.2.2 Decoding Graph Construction

The extracted syndrome is transformed into a decoding graph representation suitable for the selected decoder algorithm. In this graph, detection events are represented as vertices, while edges correspond to possible error chains connecting them.

Edge weights are typically derived from the probability of physical errors occurring on the quantum hardware. Different decoder families use different graph representations and data structures depending on their algorithmic strategy.

2.2.3 Applying Correction

After decoding, the inferred correction operation is applied logically to the encoded qubits. The objective is not necessarily to identify the exact physical error that occurred, but rather to apply a correction belonging to the same equivalence class such that no logical failure remains [7].

A successful decoding cycle suppresses logical errors while maintaining real-time operation within the quantum error-correction cycle budget.

2.3 Decoder Algorithms

Several decoder algorithms have been proposed for surface codes, each providing different trade-offs between decoding quality, computational

complexity, and hardware efficiency.

2.3.1 Minimum-Weight Perfect Matching

Minimum-Weight Perfect Matching (MWPM) is the reference-quality decoder for the surface code. Syndrome detection events become vertices of a weighted graph in which an edge connects every pair of detectors that could be triggered by the same elementary fault chain; the edge weight is $w_e = \log\big((1-p_e)/p_e\big)$, with p_e the probability of that fault. The decoder then computes a minimum-weight perfect matching, which by the standard MAP-decoding argument identifies the maximum-likelihood Pauli string consistent with the syndrome [4], [25], [26].

Figure 2.3 illustrates the overall MWPM decoding workflow. The process begins with syndrome extraction on the lattice, followed by the construction of a weighted graph between detection events. Edge weights are assigned using lattice distances and per-edge fault probabilities, after which Edmonds' Blossom algorithm [25], [26] determines an optimal pairing configuration. These matched pairs are then mapped back onto the lattice as candidate error chains, and the corresponding Pauli corrections neutralise the syndrome defects.

Modern implementations such as PyMatching [8] and Sparse Blossom [27] have significantly improved the runtime performance and scalability of MWPM decoding through sparse graph representations and incremental Blossom-based matching.

Although MWPM provides strong decoding performance and near-optimal logical error suppression, its graph-construction and matching operations become increasingly computationally expensive as the code

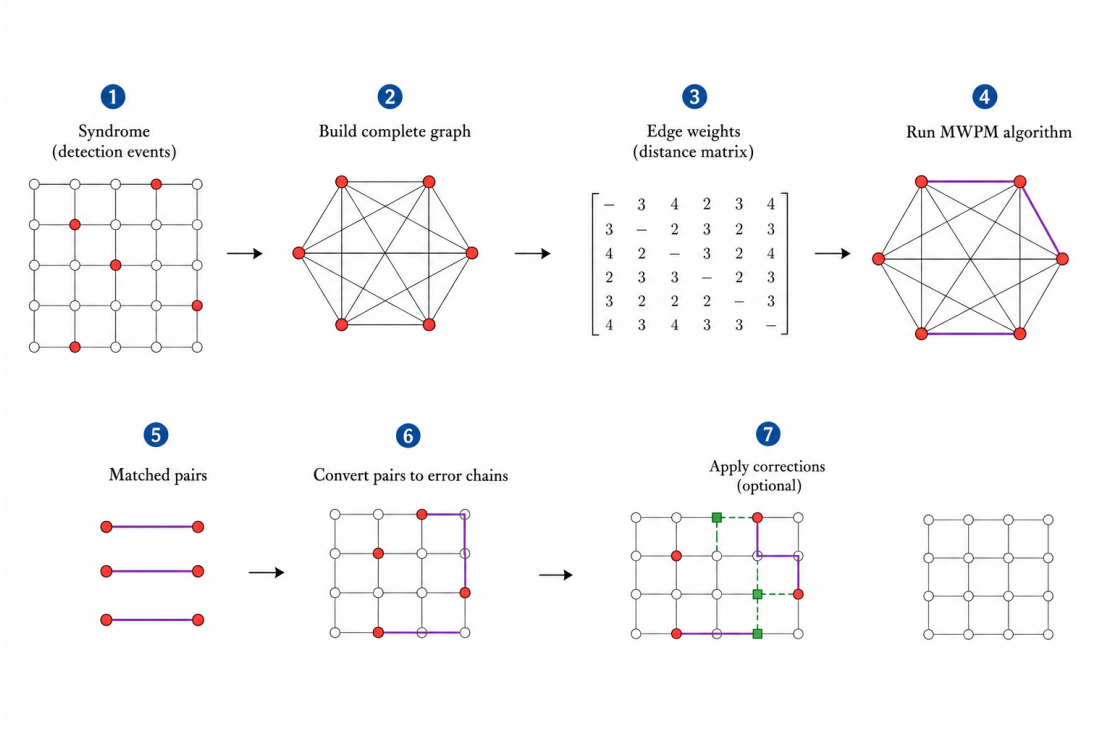

Figure 2.3: Step-by-step workflow of Minimum-Weight Perfect Matching (MWPM) decoding for surface codes, showing syndrome extraction, graph construction, weighted matching, and correction generation.

distance and syndrome volume grow.

2.3.2 Union–Find

Union–Find (UF) decoding is a cluster-based technique that achieves almost-linear time complexity in the number of detectors by dynamically growing and merging clusters of detection events [9]. Instead of solving a global matching problem, the decoder grows clusters on the lattice until each cluster's syndrome parity is satisfied, after which a peeling pass produces a concrete recovery within each cluster. The underlying disjoint-set data structure was introduced by Tarjan and Galler–Fisher and has near-constant amortised cost per operation [28].

Figure 2.4 illustrates the overall workflow of Union–Find decoding. The process begins with syndrome extraction, where detection events are identified on the lattice. Singleton clusters are initialised around each defect and grown iteratively across neighbouring lattice edges. As clusters expand, overlapping regions are merged using the disjoint-set data structure, which efficiently tracks connectivity and cluster membership. Once every cluster has even parity (or contacts a boundary), a peeling pass extracts a concrete correction within the grown forest.

Compared to MWPM, UF decoding typically provides significantly lower computational overhead and maps efficiently onto parallel and hardware-oriented architectures such as FPGAs [14], [15], [29]. The localised nature of cluster growth and merging operations also enables scalable implementations for large-distance surface codes. However, its logical error suppression performance is in general slightly lower than MWPM under circuit-level noise unless edge weights are used (weighted

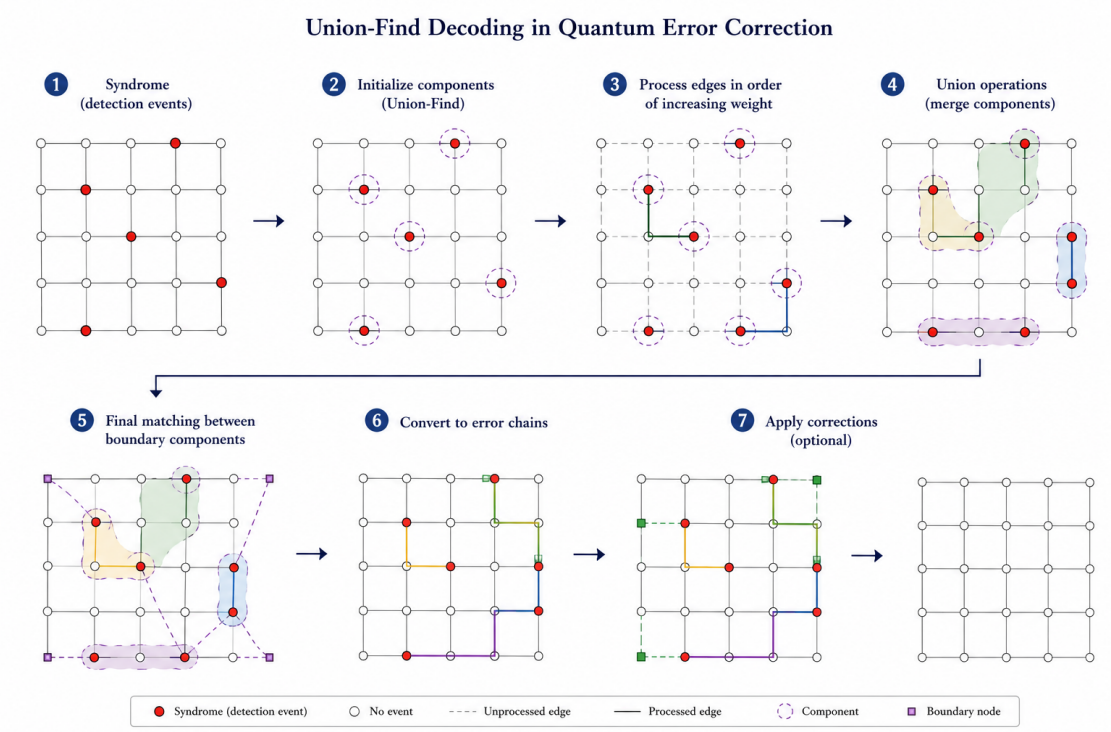

Figure 2.4: Step-by-step workflow of Union–Find (UF) decoding for surface codes: syndrome extraction, cluster initialisation, cluster growth, union of overlapping clusters, and peeling-based correction generation.

UF) [29].

2.3.3 Belief Propagation

Belief Propagation (BP) is an iterative probabilistic decoding technique originally developed by Gallager for classical low-density parity-check (LDPC) codes [30], [31]. In BP decoding, log-likelihood-ratio messages are iteratively exchanged between variable nodes and parity-check nodes on the Tanner graph of the parity-check matrix H to estimate the most likely underlying error configuration.

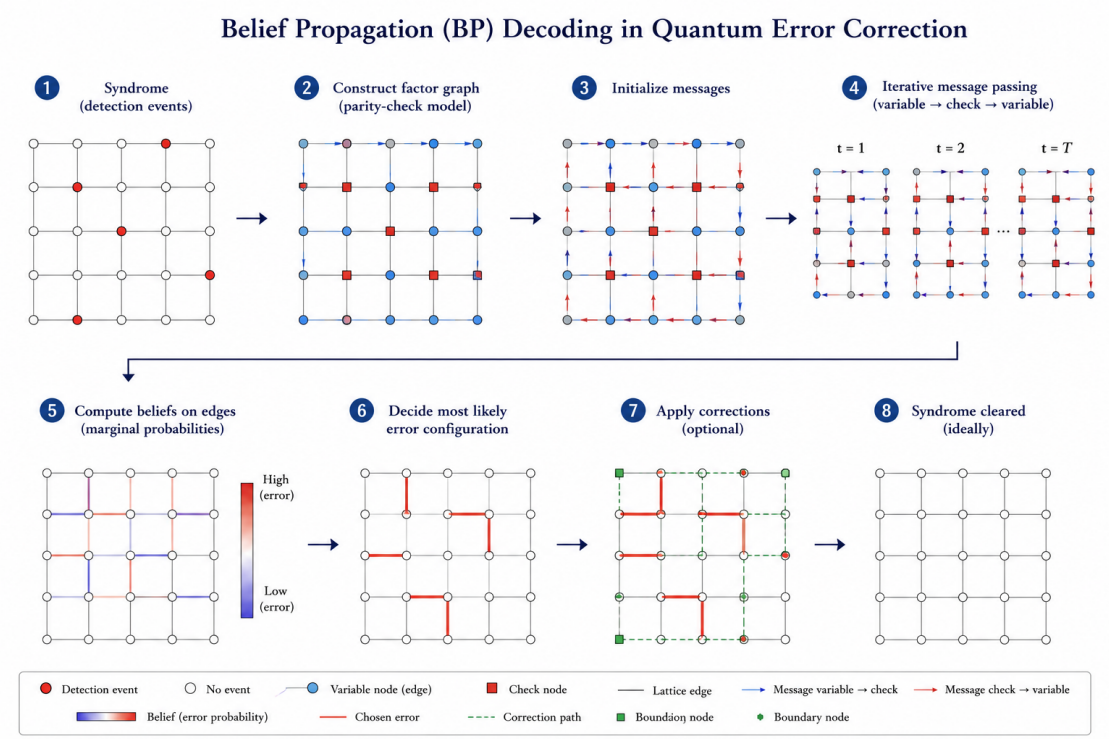

Figure 2.5: Step-by-step workflow of Belief Propagation (BP) decoding for quantum error correction: Tanner-graph construction, iterative variable-to-check and check-to-variable message passing, belief update, and final hard decision.

Figure 2.5 illustrates the workflow of Belief Propagation decoding for

quantum error correction. After syndrome extraction, a Tanner graph is built from the parity-check matrix H. Initial messages are seeded from the per-qubit prior log-likelihood ratios, and check-to-variable and variable-to-check messages are exchanged according to the min-sum or sum-product rules. The marginal error probabilities (*beliefs*) are progressively refined; after convergence or a fixed maximum iteration count, a hard decision is made and the implied Pauli correction is applied.

Quantum adaptations of BP have demonstrated strong performance for sparse quantum LDPC codes [10], [32]. However, the surface-code Tanner graph contains many length-four cycles, which causes plain BP to converge poorly because of code degeneracy — distinct errors yield equivalent stabilizer cosets and the local message-passing rules cannot resolve which representative to choose [10]. Combining BP with an Ordered Statistics Decoding (OSD) post-processing step [32], [33] restores competitive accuracy on the surface code.

2.4 Performance Metrics

Surface-code decoders are evaluated using a combination of algorithmic, architectural, and hardware-oriented performance metrics. Since quantum error correction must operate continuously alongside the quantum processor, an effective decoder must not only provide high decoding accuracy but also satisfy strict real-time execution constraints. The most commonly used evaluation metrics include decode latency, logical error rate, and scalability.

2.4.1 Latency

Decode latency refers to the total time required for a decoder to process a syndrome measurement and produce a corresponding correction decision. In software implementations, latency is typically measured in execution time, while in hardware accelerators such as FPGAs and ASICs, it is commonly measured in clock cycles, pipeline depth, and end-to-end processing delay.

Latency is a critical metric in practical fault-tolerant quantum computing because decoding must be completed within the duration of a single Quantum Error Correction (QEC) cycle. If the decoder cannot keep pace with syndrome generation, errors may accumulate faster than they can be corrected, leading to backlog and eventual logical failure. In superconducting quantum hardware, QEC cycles are typically expected to operate on microsecond timescales, placing extremely stringent timing constraints on the classical decoding system [11].

Different decoding algorithms exhibit significantly different latency characteristics. For example, MWPM decoders often incur higher latency due to global graph construction and matching operations, whereas Union–Find decoders achieve lower latency through localised cluster-growth procedures. Belief Propagation decoders may additionally experience latency overhead due to multiple iterative message-passing rounds before convergence. In hardware implementations, latency is further affected by memory access patterns, routing congestion, communication overhead, and achievable operating frequency.

2.4.2 Logical Error Rate

The Logical Error Rate (LER) represents the probability that a logical failure remains after the decoding process. It is the primary metric used to evaluate the error-correction capability and overall decoding quality of a quantum decoder. A logical failure occurs when the combined effect of physical errors and imperfect decoding results in an undetected logical operation on the encoded qubit.

LER is typically evaluated as a function of the physical error probability p and the code distance d. Below the threshold p_{th}, the LER decays as $p_L \sim A(p/p_{\text{th}})^{\lceil (d+1)/2 \rceil}$, so doubling d roughly squares the suppression factor [4], [7]. Above p_{th} the trend reverses and larger codes accumulate too many errors for the decoder to correct. Threshold estimates for the surface code under the noise models used in this thesis are summarised in Appendix B.

The logical error rate depends not only on the decoding algorithm itself but also on the underlying noise model, syndrome extraction fidelity, and decoder approximations. For instance, MWPM generally achieves near-optimal logical error suppression for surface codes, while faster decoders such as Union–Find may trade some decoding accuracy for improved runtime performance. Belief Propagation decoders can perform effectively for sparse quantum codes but may experience degraded performance in highly correlated or loopy graph structures.

2.4.3 Scalability

Scalability refers to the ability of a decoder to maintain acceptable decoding performance, latency, and hardware cost as the quantum code

distance and system size increase. Since the number of physical qubits and syndrome measurements grows rapidly with code distance, scalable decoding algorithms must efficiently manage increasing graph complexity, communication overhead, and computational workload.

From an algorithmic perspective, scalability is commonly analysed through asymptotic time and memory complexity. Decoders with lower computational complexity are generally more suitable for large-scale fault-tolerant quantum computing. For example, Union–Find decoding achieves near-linear time complexity, making it attractive for large hardware implementations, whereas MWPM decoding can become increasingly computationally expensive due to global matching operations on large graphs.

For hardware implementations, scalability is additionally influenced by FPGA resource utilisation, memory bandwidth, routing complexity, interconnect overhead, and achievable clock frequency. As decoder size increases, maintaining timing closure and efficient data movement becomes increasingly difficult, particularly for highly parallel architectures. Iterative decoders such as Belief Propagation may also encounter scalability challenges due to repeated message exchanges and synchronization overhead across processing elements.

Scalable decoding architectures must therefore balance decoding accuracy, throughput, resource consumption, and energy efficiency while remaining capable of operating within strict real-time QEC constraints. Efficient scalable decoding is considered one of the key requirements for practical large-scale fault-tolerant quantum computing systems [12], [13].

Chapter 3

Hardware System Design

This chapter presents the hardware design methodology, development flow, and evaluation framework used for implementing quantum error-correction decoders on reconfigurable and custom hardware platforms. The chapter covers hardware description methodologies, verification strategies, synthesis workflows, development tools, and performance evaluation metrics used throughout the implementation process. Particular emphasis is placed on FPGA-based acceleration and the long-term implications for Application-Specific Integrated Circuit (ASIC) implementations targeting large-scale fault-tolerant quantum computing systems.

3.1 Hardware Design Methodology

The hardware implementation of quantum error-correction decoders requires careful consideration of parallelism, latency, memory organisation, communication overhead, and scalability. Since quantum decoding must operate continuously alongside the quantum processor under strict real-time constraints, specialised hardware accelerators are often required to

achieve sufficiently low decode latency.

Field-Programmable Gate Arrays (FPGAs) provide an attractive prototyping and deployment platform due to their inherent parallelism, reconfigurability, and rapid development cycle. However, large-scale fault-tolerant quantum computing systems may eventually require custom ASIC implementations to achieve significantly lower latency, higher energy efficiency, and improved scalability.

3.2 Register Transfer Level (RTL) Design

Register Transfer Level (RTL) design refers to the low-level hardware description methodology used to model digital systems in terms of registers, combinational logic, finite-state machines, and clocked data movement between hardware components. RTL descriptions are commonly written using Hardware Description Languages (HDLs) such as Verilog or VHDL.

RTL-based implementations provide fine-grained control over datapath organisation, memory structures, pipelining, clocking behaviour, and hardware parallelism. This level of architectural control is particularly important for latency-critical quantum decoding systems, where deterministic timing behaviour and highly optimised dataflow are required.

In this work, RTL design techniques are used to implement performance-critical modules of the decoder architecture. Particular emphasis is placed on syndrome ingestion, graph-processing logic, cluster management, message propagation, pipeline scheduling, and memory-efficient hardware mapping. The RTL approach additionally enables detailed optimisation of hardware resources such as Look-Up Tables (LUTs), Flip-Flops (FFs),

Block RAMs (BRAMs), and Digital Signal Processing (DSP) blocks in FPGA implementations.

From an ASIC perspective, RTL descriptions also form the foundation of the standard digital integrated-circuit design flow. Synthesizable RTL can be mapped onto standard-cell libraries corresponding to specific semiconductor technology nodes such as 28 nm, 14 nm, 7 nm, or advanced sub-5 nm processes. Technology scaling enables higher transistor density, increased operating frequency, and reduced energy consumption, which are critical for implementing large-scale quantum decoding accelerators.

However, advanced technology nodes also introduce significant challenges, including increased routing complexity, higher leakage power, clock distribution challenges, process variability, and substantially higher fabrication costs. Therefore, decoder architectures intended for ASIC deployment must be designed with scalability, timing closure, and power efficiency in mind from the RTL stage itself.

Although RTL design provides high hardware efficiency and precise architectural optimisation, it also increases development complexity and verification effort compared to higher-level abstraction methodologies.

3.3 High-Level Synthesis (HLS)

High-Level Synthesis (HLS) enables hardware accelerators to be developed using high-level programming languages such as C/C++ instead of traditional HDL-based RTL descriptions. HLS tools automatically translate algorithmic descriptions into synthesisable hardware architectures while performing scheduling, pipeline generation, resource allocation, and

interface synthesis [22].

HLS significantly improves development productivity and reduces hardware design complexity, making it attractive for rapidly prototyping quantum decoding architectures and exploring algorithmic variations. Architectural modifications can be implemented more efficiently compared to manual RTL development, enabling faster design-space exploration and optimisation.

In this work, HLS-based methodologies are explored for implementing selected decoder components using Vitis HLS 2025.2 [23]. Optimisation directives such as loop pipelining, loop unrolling, array partitioning, and dataflow scheduling are used to improve throughput and reduce execution latency. The generated hardware architectures are subsequently synthesised and analysed using FPGA development environments.

From an ASIC perspective, HLS can also accelerate early-stage architectural exploration before transitioning to fully optimised RTL implementations. However, HLS-generated circuits may exhibit reduced timing performance, increased area overhead, and less predictable resource utilisation compared to manually optimised RTL designs [22]. These limitations become increasingly significant for deeply pipelined and latency-sensitive quantum decoding accelerators.

Therefore, this thesis analyses the trade-offs between development productivity, hardware efficiency, timing performance, and scalability across both HLS-based and RTL-based implementation methodologies.

3.4 Functional Verification

Functional verification is a critical stage in the hardware development process that ensures the correctness and reliability of the implemented decoder architecture before deployment on physical hardware.

The verification flow used in this work includes behavioural simulation, testbench-driven validation, waveform analysis, timing verification, and comparison against software reference models. Hardware-generated correction outputs are compared with golden-reference software simulations to verify functional correctness across a wide range of syndrome patterns and operating conditions.

Simulation environments are additionally used to validate pipeline behaviour, state transitions, cluster evolution, graph-processing correctness, message-passing operations, and memory access patterns depending on the decoder architecture under consideration. Corner-case testing is also performed to analyse decoder behaviour under high syndrome density and worst-case operating scenarios.

For ASIC-oriented workflows, verification additionally plays a major role in ensuring design reliability prior to fabrication, since post-fabrication modifications are extremely costly and often impractical. ASIC verification flows therefore typically include formal verification, static timing analysis, clock-domain-crossing verification, power analysis, and gate-level simulation.

Comprehensive verification is essential for real-time quantum error-correction systems because incorrect decoder behaviour can directly propagate into logical failures in the quantum processor.

3.5 Tools and Development Stack

The hardware development workflow used in this thesis integrates multiple software and hardware tools for design entry, simulation, synthesis, implementation, timing analysis, and performance evaluation.

RTL development uses Verilog-2005 with Icarus Verilog as the open-source simulator and Yosys [20] for ASIC-oriented synthesis against the Nangate 45 nm Open Cell Library [21]. FPGA synthesis, place-and-route, and bitstream generation are carried out using AMD Xilinx Vivado 2025.2 targeting the Zynq UltraScale+ ZCU104 evaluation board [19], and high-level synthesis baselines are produced using Vitis HLS 2025.2 [23]. The PYNQ overlay [34] provides Python-based control of the on-board ARM processing system for register access and result read-back.

The development stack additionally includes software simulation frameworks (Stim [17], PyMatching [8], the `ldpc` package [18]) for syndrome generation and decoder validation, scripting environments for automation, GTKWave/Surfer for waveform inspection, and Python-based preprocessing utilities for benchmarking and architectural analysis.

For ASIC-oriented analysis, additional considerations include synthesis using standard-cell libraries, physical design workflows, timing closure across process–voltage–temperature (PVT) corners, floorplanning, routing congestion analysis, and power estimation under different technology nodes. The integration of these tools enables rapid prototyping, architectural exploration, systematic optimisation, and scalable performance evaluation of quantum decoding hardware architectures.

3.6 Performance Metrics

The proposed decoder implementations are evaluated using multiple architectural and hardware-oriented performance metrics. These metrics are used to analyse decoding efficiency, real-time capability, hardware scalability, and implementation feasibility across both FPGA and ASIC deployment scenarios.

3.6.1 Timing Performance

Timing performance refers to the execution speed of the hardware decoder and is commonly characterised using metrics such as clock frequency, critical path delay, throughput, and decode latency.

For FPGA implementations, timing analysis determines the maximum achievable operating frequency after synthesis and routing. Decode latency is measured both in clock cycles and absolute execution time. Since quantum error correction operates continuously during computation, the decoder must complete syndrome processing within the duration of a single QEC cycle.

Pipeline depth, routing congestion, communication overhead, graph-processing complexity, and memory-access latency significantly influence timing performance. Iterative decoders such as Belief Propagation may additionally incur increased latency due to repeated message-passing rounds, while graph-based decoders such as MWPM often experience timing bottlenecks from global matching operations.

From an ASIC perspective, timing performance is strongly influenced by the semiconductor technology node. Advanced technology nodes such

as 7 nm or 5 nm can provide substantially higher operating frequencies and lower switching delays compared to older nodes such as 28 nm or 65 nm. However, deep submicron scaling also introduces challenges related to wire delays, clock distribution, process variation, and power density.

Achieving timing closure in large-scale quantum decoding accelerators therefore requires careful architectural optimisation, pipelining, and interconnect-aware design methodologies.

3.6.2 Resource Utilisation

Resource utilisation measures the amount of hardware resources consumed by the decoder implementation.

For FPGA-based systems, commonly evaluated resources include:

- Look-Up Tables (LUTs)
- Flip-Flops (FFs)
- Block RAMs (BRAMs)
- UltraRAM (URAM)
- Digital Signal Processing (DSP) blocks

Efficient resource utilisation is important for supporting larger decoder architectures and higher code distances on a single FPGA device. Different decoder algorithms exhibit different resource-consumption characteristics depending on their graph structures, memory organisation, degree of parallelism, and communication patterns.

From an ASIC perspective, resource utilisation is commonly represented in terms of gate count, standard-cell area, SRAM usage, and

interconnect complexity. As code distance increases, the required decoder hardware can scale rapidly, making efficient area management increasingly important.

Resource analysis therefore provides insight into decoder scalability, architectural efficiency, and long-term feasibility for large-scale fault-tolerant quantum computing systems.

3.6.3 Area and Power

Area and power consumption are critical considerations for practical deployment of large-scale quantum decoding systems, particularly for future tightly integrated quantum-classical control architectures.

Area refers to the total silicon footprint or FPGA fabric occupancy of the decoder architecture, including logic resources, memory blocks, routing infrastructure, and communication interconnects. Highly parallel architectures may improve throughput and latency performance but often increase area consumption significantly.

Power consumption consists of both static and dynamic components. Dynamic power is heavily influenced by switching activity, operating frequency, memory accesses, and communication overhead. In FPGA implementations, routing resources and clock networks can contribute substantially to overall power consumption.

For ASIC implementations, technology scaling generally reduces dynamic power per operation but may increase leakage power at advanced nodes due to reduced transistor dimensions. Thermal density and power delivery also become increasingly challenging in highly parallel decoder architectures implemented at modern technology nodes.

Power-efficient decoding architectures are especially important for future cryogenic control systems, where thermal budgets are severely constrained. Minimising power consumption while maintaining low-latency operation is therefore a key requirement for scalable fault-tolerant quantum computing hardware.

The trade-offs between decoding performance, hardware area, operating frequency, and power efficiency are analysed throughout this thesis as part of the overall hardware evaluation framework.

Chapter 4

Efficient Hardware Decoder Mappings

The hardware-acceleration strategies evaluated in this study are motivated by recent advances in quantum error-correction decoder architectures reported in the literature. Rather than proposing entirely new decoding algorithms, this work focuses on identifying computational bottlenecks within established decoder families and mapping them efficiently onto parallel hardware platforms such as FPGAs and, potentially, ASICs.

This chapter reviews prior hardware-oriented decoder implementations and highlights the architectural techniques that directly motivate the optimisation strategies explored in this thesis. Particular emphasis is placed on latency reduction, scalable parallelism, and hardware-efficient execution.

4.1 Union–Find in Hardware

One of the most influential FPGA implementations of Union–Find decoding is the **Helios** architecture by Liyanage *et al.* [14]. Helios organises

parallel processing elements into a hybrid tree–grid structure and implements an almost-local version of the Union–Find algorithm, with multiple PEs concurrently managing local regions of the decoding lattice.

A key result reported by Helios is that, with $\mathcal{O}(d^3)$ parallel hardware resources, the *average* decoding time per measurement round decreases as the code distance increases, reaching approximately $11.5\,\text{ns/round}$ at $d = 21$ under 0.1% phenomenological noise on a Xilinx VCU129. The follow-up distributed Union–Find architecture [15] extends this to a multi-FPGA tree–grid topology with sub-microsecond merging latency.

Earlier work such as the **AFS** decoder by Das *et al.* [35] demonstrated a low-latency and scalable Union–Find microarchitecture targeted at cryogenic control, while **LILLIPUT** [36] showed that lookup-table-based decoding can achieve nanosecond-scale latency for small-distance surface codes.

A more recent cluster-based contribution is the **Collision Clustering** decoder of Barber *et al.* [13], which grows clusters of detection events until collisions occur and demonstrates scalable, resource-efficient decoding on both FPGA and ASIC platforms; it sustains megahertz decoding speed up to surface codes of 881 (FPGA) and 1 057 qubits (ASIC).

Architectural motivation. A key observation from these works is that the disjoint-set `find` operation — traditionally implemented as a serial pointer traversal through a parent tree — becomes a major latency bottleneck in hardware. Rather than resolving a single parent hop per clock cycle, the operation can be pipelined or partially unrolled so that multiple parent traversals are resolved concurrently. Additionally, cluster growth operations are inherently parallel and map efficiently onto

distributed processing structures.

Motivated by these observations, this thesis adopts a pipelined multi-hop `find` architecture as the primary optimisation strategy for the Union–Find decoding kernel.

4.2 Matching-Based Decoding in Hardware

Minimum-Weight Perfect Matching (MWPM) decoding remains one of the most computationally demanding quantum decoding techniques due to the irregular and graph-intensive nature of Blossom-based matching algorithms.

One of the most advanced hardware-oriented MWPM implementations is **Micro Blossom** [16], which partitions the decoding workflow between software and a programmable accelerator that deploys dedicated processing units at every graph vertex and edge. Micro Blossom is the first publicly reported MWPM decoder to achieve sub-microsecond mean decoding latency, demonstrating decoding under circuit-level noise within the QEC cycle budget at $d = 13$.

Architectural motivation. A recurring bottleneck in graph-based matching algorithms is the serial frontier-selection stage present in shortest-path computations such as Dijkstra-like graph traversal. A naïve implementation performs a sequential scan across all candidate distances, requiring N cycles for N nodes. In contrast, a balanced comparator-tree reduction structure can determine the minimum value within a single cycle while introducing only logarithmic logic depth.

Motivated by these hardware-oriented optimisations, this thesis adopts

balanced-tree frontier extraction as the primary acceleration strategy for MWPM-related shortest-path kernels.

4.3 Belief Propagation in Hardware

Belief Propagation (BP) and min-sum decoding architectures for quantum LDPC codes have also been extensively explored in FPGA-based systems. Valls *et al.* [37] implemented and compared syndrome-based min-sum and Ordered Statistics Decoding (OSD)-assisted decoders, analysing both decoding performance and hardware feasibility for real-time quantum error correction.

Unlike graph-matching algorithms, BP decoding consists primarily of repeated message-passing operations between check nodes and variable nodes. Since these updates are independent given the current message state, the algorithm naturally exposes substantial parallelism and maps efficiently onto arrays of identical processing elements.

Architectural motivation. The most direct hardware acceleration strategy for BP decoding is processing-element replication. Multiple node-update engines can operate concurrently on different graph regions without introducing significant control dependencies. Since each processing element performs identical computations, latency can be reduced by increasing parallelism without substantially increasing critical-path complexity.

Based on these observations, this thesis adopts processing-element replication as the primary optimisation strategy for Belief Propagation decoding kernels.

4.4 Summary of Hardware Mapping Strategies

An additional observation common across nearly all hardware-decoder literature is the separation between algorithmic evaluation and hardware evaluation. Logical error rate is generally measured using software-level simulations of the decoding algorithm, while latency, frequency, area, and resource utilisation are evaluated using hardware synthesis and implementation results. The decoding quality itself remains fundamentally an algorithmic property and is typically unaffected by hardware-level optimisations such as pipelining, retiming, or processing-element replication, provided that algorithmic correctness is preserved.

Table 4.1: Mapping of prior hardware-decoder literature to the optimisation strategies evaluated in this work.

Decoder Family	**Relevant Literature**	**Strategy Used in This Work**
Union–Find / cluster-based	Helios [14], distributed UF [15]	pipelined multi-hop `find`
	AFS [35]	parallel cluster growth
	weighted UF [29]	weighted cluster growth
	Collision Clustering [13]	scalable graph-processing
MWPM	Micro Blossom [16]	balanced-tree frontier extraction
Belief Propagation	FPGA min-sum [37]	processing-element replication
	BP+OSD [10], [32]	ordered-statistics post-processing

This thesis follows the same two-layer methodology, separately evaluating decoding quality and hardware performance while ensuring functional equivalence between software and hardware implementations.

Chapter 5

Methodology

This chapter presents the methodology adopted for evaluating hardware-oriented quantum decoder architectures for surface-code error correction. The study integrates software-level algorithmic analysis with hardware-oriented architectural evaluation to investigate both decoding performance and implementation feasibility across FPGA and ASIC-oriented workflows.

The evaluation framework is structured to clearly separate algorithmic behaviour from hardware implementation characteristics. This separation enables decoding quality metrics, such as logical error rate, to be analysed independently from hardware-oriented metrics including latency, operating frequency, and resource utilisation, while ensuring functional equivalence between software reference models and hardware implementations.

5.1 Overview of the Evaluation Flow

The study begins with software-based implementations of the selected decoder families. These implementations are used for profiling, behavioural

analysis, and extraction of computational bottlenecks that dominate decoder execution time.

Based on the profiling results, hardware-oriented implementations are developed for the dominant computational structures using Register Transfer Level (RTL) design and High-Level Synthesis (HLS). The generated hardware modules are then verified against software-generated reference outputs before being synthesised for both FPGA and ASIC-oriented technology flows. The overall workflow followed in this thesis is summarised below:

1. Construction of surface-code simulation framework
2. Decoder implementation and profiling in C++
3. Generation of golden-reference outputs using Python
4. Functional verification using RTL testbenches
5. FPGA synthesis using Vivado and Vitis HLS
6. ASIC-oriented synthesis using open-source RTL flows
7. Timing, latency, and resource evaluation

5.2 Decoder Variants Considered

Three major decoder families are considered in this work are Minimum-Weight Perfect Matching (MWPM), Union–Find (UF), Belief Propagation (BP). To study both decoding quality and hardware behaviour, multiple variants of these decoder families are evaluated.

Table 5.1: Decoder variants evaluated in this study.

Variant	Decoder Family	Technique	Reference
`mwpm`	MWPM	Blossom minimum-weight matching	[8], [25]
`uf`	Union–Find	synchronous unweighted cluster growth	[9]
`uf_weighted`	Union–Find	weighted cluster growth	[29]
`bp_ms`	Belief Propagation	plain min-sum	[31]
`bp_nms`	Belief Propagation	normalised min-sum	[38]
`bp_osd`	Belief Propagation	min-sum with order-0 OSD	[10]

The MWPM decoder implementations are based on PyMatching [8], [27], while the Union–Find and Belief Propagation variants are implemented using custom software frameworks together with the `ldpc` package [18] for BP and BP+OSD.

5.3 Software Profiling and Algorithmic Analysis

The study begins with software-level decoder implementations developed in C++. These implementations are used to analyse execution behaviour, identify dominant computational bottlenecks, and evaluate scalability across different code distances.

Profiling experiments are performed on an Apple Silicon M1 system running macOS. The profiling stage focuses on identifying computational kernels that dominate runtime behaviour within each decoder family. Particular emphasis is placed on graph traversal operations, shortest-path

computations, cluster-growth procedures, memory-access behaviour, and iterative message-passing operations.

The profiling results are subsequently used to guide hardware-oriented architectural exploration and determine which components are most suitable for acceleration using RTL and FPGA-based implementations.

5.4 Surface-Code Simulation Framework

The decoding-quality evaluation is performed using a Monte-Carlo simulation framework built around rotated surface-code memory experiments generated using Stim [17].

Stim is used to generate stabilizer circuits and detector error models corresponding to the syndrome-generation process. The extracted detector-error models are converted into decoding graphs shared across all decoder implementations. Two noise models are considered:

- **Code-Capacity Noise Model**

  A simplified model consisting of data-qubit depolarising noise followed by perfect syndrome measurements.

- **Circuit-Level Noise Model**

  A fault-tolerant noise model including noisy syndrome extraction, gate errors, reset errors, and measurement errors.

For each combination of code distance and physical error probability, multiple syndrome samples are generated and decoded independently. Logical error rates are then computed as the fraction of logical failures observed during simulation.

5.5 Golden Reference Model

To ensure correctness across all hardware implementations, a software-generated golden-reference model is used throughout the verification workflow.

The reference outputs are generated using Python-based decoder implementations and validated software simulations. These golden-reference outputs contain expected syndrome-processing results, correction outputs, and intermediate decoding states depending on the decoder architecture under evaluation.

The golden-reference data serves as the primary correctness baseline for both RTL simulation and HLS-generated hardware verification.

5.6 RTL Design and Functional Verification

Hardware implementations are developed using synthesizable Verilog-2005 RTL descriptions. The RTL modules are designed to model the dominant computational structures identified during the profiling stage.

To validate correctness, dedicated testbenches are constructed for each RTL module. These testbenches provide syndrome inputs, decoding-graph structures, and control signals corresponding to the software-generated reference data.

The RTL outputs are then compared cycle-by-cycle against the golden-reference outputs to verify functional equivalence between software and hardware implementations.

Functional simulation is performed using Icarus Verilog, while waveform-level debugging and signal analysis are carried out using standard RTL

simulation tools available within the open-source Verilog toolchain ecosystem. The verification flow additionally validates:

- state transitions
- pipeline behaviour
- graph-processing correctness
- memory-access ordering
- synchronization behaviour
- decoder output correctness

Comprehensive functional verification is essential because incorrect decoder behaviour directly translates into logical failures in the quantum error-correction system.

5.7 ASIC-Oriented Hardware Flow

In addition to FPGA-oriented evaluation, the study also explores ASIC-oriented synthesis methodologies using open-source digital design tools available on macOS platforms.

The ASIC workflow is built around the following toolchain:

- Icarus Verilog for RTL simulation,
- Yosys for RTL synthesis [20],
- ABC for technology mapping (invoked from within Yosys),

- the Nangate 45 nm Open Cell Library as the reference standard-cell library [21],
- GTKWave/Surfer for waveform inspection.

The Nangate 45 nm standard-cell library is used because it is fully open and includes calibrated timing arcs and leakage data, making the synthesis results reproducible and comparable across designs. This enables estimation of gate-level area, timing behaviour, and logic complexity under a standard-cell flow.

Although full physical implementation and tapeout-level analysis are outside the scope of this thesis, the ASIC-oriented flow provides insight into the scalability and feasibility of future custom quantum-decoder accelerators.

5.8 FPGA and High-Level Synthesis Flow

FPGA-oriented hardware evaluation is performed using the AMD Xilinx development ecosystem.

RTL implementations are synthesised using Vivado 2025.2 targeting the Zynq UltraScale+ XCZU7EV-2FFVC1156E device on the ZCU104 evaluation board [19]; this device is representative of the low-latency, mid-cost FPGAs used by current quantum-control stacks. Timing analysis, resource utilisation, and synthesis reports are extracted from out-of-context (OOC) Vivado synthesis with a 4 ns clock target; the achievable frequency is reported as $F_{\max} = 1000/(4 - \mathrm{WNS})$ MHz.

To compare manual RTL development against higher-level hardware-generation methodologies, equivalent implementations are additionally

developed using Vitis HLS 2025.2 [23].

The HLS implementations are written in C++ with `#pragma HLS PIPELINE` on inner loops and `#pragma HLS ARRAY_PARTITION` on small look-up arrays only. The resulting HLS-generated architectures are compared against the hand-written RTL implementations in terms of latency, operating frequency, and hardware resource utilisation.

5.9 Performance Evaluation Metrics

The proposed implementations are evaluated using both algorithmic and hardware-oriented metrics.

The primary algorithmic metric considered is:

- Logical Error Rate (LER)

The hardware-oriented metrics include:

- Decode latency
- Clock frequency ($F_{\max}$)
- LUT utilisation
- Flip-Flop utilisation
- DSP utilisation
- Block RAM utilisation
- ASIC gate-level area estimates

These metrics are analysed across multiple decoder families, implementation strategies, and hardware flows in order to evaluate the trade-offs between decoding quality, latency, scalability, and hardware cost.

Chapter 6

Kernel Identification, RTL Design, and Verification

The preceding chapters establish the algorithmic and methodological context for evaluating quantum error-correction decoders. This chapter turns that context into hardware targets. The central design decision is deliberately conservative: rather than attempting to place an entire software decoder directly into hardware, the study first identifies the computational kernels that dominate execution time and then implements those kernels as small, testable, synthesisable RTL blocks.

This kernel-oriented approach has two advantages. First, it ties the hardware effort to measured software bottlenecks rather than intuition. Second, it allows MWPM, Union–Find, and BP to be compared through their natural microarchitectural shapes: shortest-path search and subset dynamic programming for MWPM, disjoint-set cluster growth for Union–Find, and iterative message passing for BP.

6.1 Profiling-Based Kernel Selection

The software profiling stage used C++ reference implementations of the three decoder families driven by Stim-generated rotated surface-code instances. Per-routine timers decomposed each decoder execution into its major computational stages. The objective was not to obtain a final software benchmark, but to identify which part of each decoder should be moved into hardware first.

Table 6.1 summarises the measured bottlenecks at distance $d = 7$. The profile shows that each decoder has a small number of dominant routines. MWPM is dominated by the matching dynamic program, Union–Find splits work between cluster growth and peeling, and BP is dominated by node-update computation, especially in the sum-product form.

Table 6.1: Measured runtime split used for kernel selection at $d = 7$. The dominant routines (bold) become the RTL targets.

Decoder	**Routine**	**Arithmetic pattern**	**Runtime share**
MWPM	Dijkstra	irregular graph traversal	25.0%
MWPM	matching DP	subset dynamic programming	**75.0%**
Union–Find	cluster growth	find/union, pointer traversal	**53.0%**
Union–Find	peeling	tree walk, XOR accumulation	47.0%
BP sum-product	check-node update	sparse update, non-linear functions	**87.9%**
BP min-sum	check-node update	min, sign, sparse accumulation	**50.1%**
BP min-sum	variable-node update	sparse accumulation	49.9%

Three conclusions follow from the profile. First, MWPM cannot be

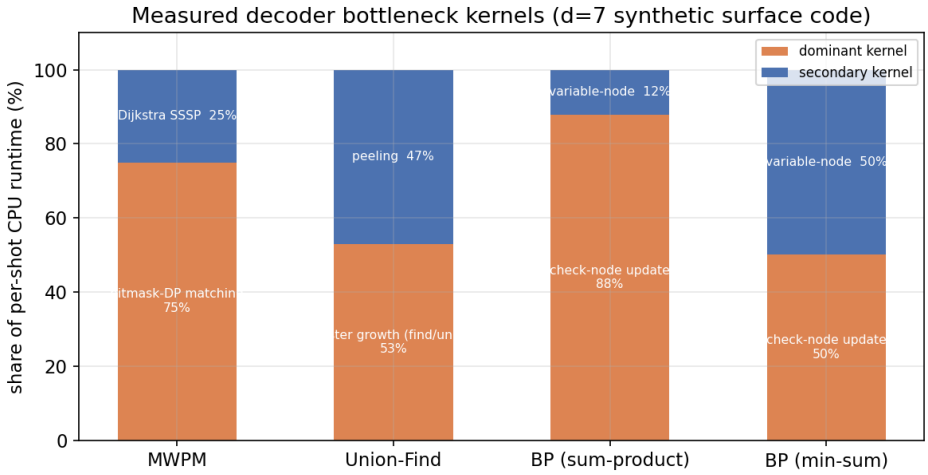

Figure 6.1: Runtime bottleneck profile of the three decoder families. The hardware study targets the routines that dominate per-shot execution.

treated as a single opaque algorithm: its shortest-path and matching stages have different hardware structures and must be analysed separately. Second, Union–Find is sufficiently balanced that both cluster growth and peeling are relevant to a complete implementation, even though the growth stage is the primary optimisation target. Third, BP should be implemented in the min-sum form rather than sum-product, since min-sum replaces transcendental functions with compare, sign, and scaling operations while preserving the message-passing structure relevant to hardware [31].

6.2 RTL Targets and Strategy Knobs

Two related sets of RTL modules are used in this thesis. The first set, developed in the kernel-level implementation study, contains complete decoder tops and their sub-kernels. The second set, used for

the distance-scaling study, contains compact parameterised kernels with explicit strategy knobs. Table 6.2 combines the two views.

Table 6.2: RTL modules and architectural strategy knobs used in the thesis.

Decoder	**Study-1 RTL modules**	**Distance-scaled kernel**	**Strategy knob**
MWPM	`mwpm_dijkstra`, `mwpm_dp_match`, `mwpm_top`	`mwpm_dijkstra_k`	`PAR`$\in \{0,1\}$: linear scan or parallel frontier extraction
Union–Find	`uf_dsu`, `uf_cluster`, `uf_peeling`, `uf_top`	`uf_grow`	`HOPS`$\in \{1,2,4,8\}$: parent hops resolved per cycle
Belief Propagation	`bp_check_node`, `bp_var_node`, `bp_top`	`bp_minsum`	`NPE`$\in \{1,2,4,8\}$: replicated processing elements

The strategy knobs are not arbitrary tuning parameters. They are direct hardware translations of the literature reviewed in Chapter 4: processing-element replication for BP, multi-hop or pipelined `find` for Union–Find, and parallel frontier extraction for MWPM.

6.3 Common RTL Design Conventions

All modules are written in synthesisable Verilog-2005. This choice keeps the same RTL acceptable to Icarus Verilog for simulation, Yosys for ASIC-oriented synthesis, and Vivado for FPGA synthesis. Each module is structured as a finite-state machine (FSM) with explicit registered state and combinational datapath logic; latches, multiple-driver nets, and tool-specific SystemVerilog constructs are avoided.

6.3.1 Start–Done Interface

Every kernel exposes a common request/acknowledge interface:

`clk`, `rst`, `start`, `done`.

The host asserts `start`; the kernel runs until its internal FSM reaches completion; and `done` remains asserted until `start` is deasserted. Holding `done` avoids the single-cycle race in which a testbench or host samples the status signal on the wrong clock edge. This convention is also convenient for the AXI4-Lite FPGA wrapper discussed in Chapter 7.

6.3.2 Fixed-Point Format

Edge weights and BP log-likelihood-ratio messages use signed Q8.8 fixed-point arithmetic. A 16-bit two's-complement value represents $x/256$, giving a resolution of $1/256$ and a range close to $[-128, 128)$. This is sufficient for the edge weights $w = \log((1-p)/p)$ used in the surface-code decoding graph and for accumulated BP messages. The sentinel value `0x7FFF` is used as positive infinity in shortest-path computations.

6.3.3 Graph Representation

The decoding graph is stored in compressed sparse row (CSR) form. Directed edge arrays store source, destination, weight, and observable-parity data, while `adj_start` and `adj_count` locate the outgoing edge segment for each detector node. Since Verilog-2005 cannot pass unpacked arrays through module ports, multi-element structures are flattened into wide buses at the boundary and unpacked internally using generate loops.

This convention keeps the external interface portable while preserving readable array-style logic inside each module.

6.4 MWPM Hardware Architecture

MWPM has two hardware-relevant stages. The first is shortest-path computation from a defect to all other graph nodes. The second is the matching problem over the resulting dense defect-cost matrix.

6.4.1 Dijkstra Shortest-Path Kernel

The baseline `mwpm_dijkstra` kernel implements a deterministic linear-scan priority queue. For each extraction step, the FSM scans all unsettled nodes, selects the node with minimum tentative distance, relaxes its outgoing edges, and records the accumulated observable parity. This avoids heap control logic and dynamic memory allocation, making the hardware small and predictable for modest graph sizes. Its cost, however, scales poorly: the extraction scan is $O(N)$ and repeated $O(N)$ times.

The optimised distance-scaled kernel `mwpm_dijkstra_k` therefore exposes `PAR`. With `PAR=0`, frontier extraction uses the baseline scan. With `PAR=1`, extraction is collapsed into a combinational reduction. Chapter 8 shows that this dramatically reduces cycle count, while Chapter 7 explains why the reduction must be structured as a balanced or pipelined tree if the clock frequency is to survive.

6.4.2 Bitmask Dynamic-Programming Matcher

The `mwpm_dp_match` kernel solves the exact minimum-weight matching problem for up to eight defects using a subset dynamic program. For a set S of unmatched defects, $dp[S]$ is the minimum cost to match all defects in S. If i is the lowest-indexed defect in S, the recurrence is

$$dp[S] = \min\left(b_i + dp[S \setminus \{i\}],\ \min_{j \in S,\, j>i} (c_{ij} + dp[S \setminus \{i,j\}])\right),$$

where b_i is the boundary cost and c_{ij} is the pair cost. The DP and choice tables are small enough to be implemented as register arrays, enabling later parallel candidate evaluation in the optimisation study.

6.5 Union–Find Hardware Architecture

The Union–Find implementation is divided into a disjoint-set engine, a cluster-growth controller, and a peeling stage.

6.5.1 Disjoint-Set Engine

The `uf_dsu` module stores `parent[]` and `rank[]` arrays. A baseline `find` walks one parent link per cycle and applies path-halving as it proceeds. A `union` compares the roots and ranks, then merges the two sets. The distance-scaled `uf_grow` kernel exposes the `HOPS` parameter, allowing several parent links to be resolved in one cycle. This is useful only when parent chains are deep enough to justify the added combinational delay, a point that becomes important in the measured results.

6.5.2 Cluster Growth and Peeling

The `uf_cluster` FSM scans edges, determines whether either endpoint belongs to a non-neutral cluster, grows the edge when required, and merges clusters through the DSU. Cluster metadata records syndrome parity and boundary contact. Once all clusters are neutral, the `uf_peeling` module constructs a correction by walking the grown forest from leaves inward, toggling syndrome bits and accumulating the observable-parity mask.

This structure mirrors the algorithmic Union–Find decoder [9]: growth determines the connected correction region, and peeling extracts a concrete correction from that region.

6.6 Belief-Propagation Hardware Architecture

The BP hardware implements the normalised min-sum update. In the check-node stage, incoming variable-to-check messages are decomposed into magnitudes and signs; the smallest and second-smallest magnitudes are selected; and outgoing check-to-variable messages are formed using the global sign parity and the normalisation factor $\alpha = 0.75$. In the variable-node stage, incoming check messages are summed with the prior LLR to form an updated belief, and extrinsic variable-to-check messages are emitted by subtracting the contribution of the target edge.

The baseline `bp_top` time-multiplexes one check-node and one variable-node engine over the graph. The distance-scaled `bp_minsum` kernel exposes `NPE`, which instantiates multiple identical processing elements. Because each PE is a copy of the baseline datapath, this is expected to

reduce cycle count without increasing the critical path. The measured results confirm this expectation.

6.7 Functional Verification

Hardware correctness is treated as a prerequisite for quoting any synthesis or latency result. Each kernel is tested against an independent golden model written in Python or against a baseline RTL implementation in a differential testbench. The golden models are intentionally written in a direct, non-RTL style so that they do not copy microarchitectural mistakes from the Verilog.

Table 6.3: RTL verification coverage for the kernel-level implementation study.

Testbench	**Check performed**	**Result**
`tb_mwpm_dijkstra`	shortest paths and observable masks on a toy graph	PASS
`tb_mwpm_dp_match`	exact matching over multiple cost matrices	PASS
`tb_mwpm_top`	complete MWPM decode on reference syndromes	PASS
`tb_uf_dsu`	union/find root sequence and path-update behaviour	PASS
`tb_uf_cluster`	cluster growth and grown-edge set	PASS
`tb_uf_peeling`	tree peeling and correction extraction	PASS
`tb_uf_top`	complete Union–Find decode	PASS
`tb_bp_check_node`	min-sum check-node vectors	PASS
`tb_bp_var_node`	variable-node extrinsic-message vectors	PASS
`tb_bp_top`	complete BP decode over three syndromes	PASS

The verification stage caught several design issues before synthesis: an array-indexing bug in BP message accumulation, an incorrectly sized

subset iterator in the MWPM matcher, and a one-cycle `done` pulse that could be missed by a polling testbench. These fixes became part of the design rules above. The optimisation study later uses differential verification: the baseline and optimised kernels are instantiated side by side, driven by identical vectors, and asserted to produce identical outputs. This ensures that any performance difference reported in Chapters 7 and 8 is not caused by a change in decoder function.

Chapter 7

Hardware Implementation, Synthesis, and Optimisation Results

Chapter 6 established the RTL kernels and the verification discipline. This chapter reports the first hardware study built on those verified modules. The same RTL is characterised in three ways: logic-only ASIC synthesis using an open 45 nm standard-cell library, FPGA implementation on the Xilinx ZCU104 through a PYNQ-compatible wrapper, and kernel-level optimisation with measured area–speed trade-offs.

The chapter is intentionally evidence-driven. The goal is not merely to show that the designs synthesise, but to determine which architectural transformations reduce real wall-clock latency and which only reduce clock cycles while making timing worse.

7.1 ASIC Synthesis Flow

The ASIC-oriented evaluation targets the Nangate 45 nm Open Cell Library at the typical $(1.10\,\mathrm{V},\ 25\,^{\circ}\mathrm{C})$ corner [21]. This open standard-cell library provides cell area, leakage power, and timing arcs, making the synthesis results reproducible. The flow is logic-only and stops before physical place-and-route; therefore the reported area is pre-layout cell area and the timing estimate is a consistent, first-order comparison rather than sign-off static timing.

Yosys [20] reads and elaborates the Verilog, extracts and optimises FSMs, maps registers through `dfflibmap`, and uses ABC to map combinational logic to the 45 nm library. The synthesis logs are parsed for cell count, total cell area, flip-flop count, critical-path depth, and leakage. Dynamic power is estimated from a documented switched-capacitance model and added to leakage to obtain the total power estimate.

7.2 Baseline ASIC Results

Table 7.1 gives the baseline ASIC synthesis results for the principal kernels and the algorithm-level tops. Figure 7.1 visualises the three complete decoders.

The results are consistent with the algorithmic structure. MWPM is the heavyweight: `mwpm_top` occupies approximately $107\,000\,\mu\mathrm{m}^2$, more than twice the area of `uf_top` and more than five times the area of `bp_top`. The exact matching DP contributes strongly to this cost because the subset DP and choice tables are held in flip-flops. Union–Find is balanced across cluster growth and peeling, while BP is compact because

Table 7.1: Baseline ASIC synthesis results using the Nangate 45 nm library.

Design	**Cells**	**Area** (μm^2)	**Flops**	$F_{\max}$ (MHz)	**Power** (μW)
MWPM					
`mwpm_dijkstra`	18 342	32 212	2 311	66.1	768
`mwpm_dp_match`	26 610	53 631	5 494	54.4	1 200
`mwpm_top`	—	106 803	20 458	58.6	4 260
Union–Find					
`uf_dsu`	7 856	12 711	886	45.1	317
`uf_cluster`	7 769	22 820	2 794	48.7	1 000
`uf_peeling`	13 494	26 441	2 119	85.1	603
`uf_top`	—	49 428	7 060	116.2	2 041
Belief Propagation					
`bp_check_node`	6 051	8 057	269	201.6	210
`bp_var_node`	2 835	4 519	299	346.0	112
`bp_top`	3 972	19 523	2 168	263.9	850

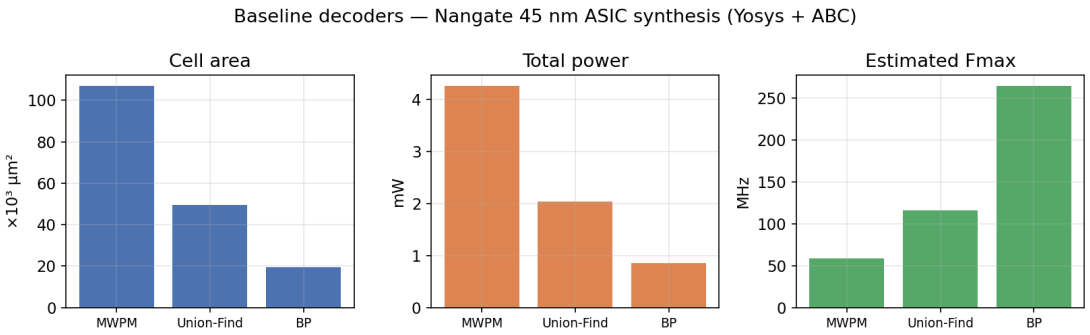

Figure 7.1: Baseline ASIC characterisation of the three complete decoder tops. MWPM is the largest design, Union–Find is intermediate, and BP is the most compact under the 45 nm standard-cell flow.

min-sum avoids transcendental arithmetic.

7.3 ZCU104 FPGA Implementation

The FPGA implementation takes the same verified RTL and wraps it in an AXI4-Lite interface suitable for the Xilinx Zynq UltraScale+ ZCU104 evaluation board [19]. The ZCU104 contains a processing system (PS) running software and programmable logic (PL) holding the decoder core. A PYNQ-style integration [34] therefore lets Python code running on the ARM processor configure the decoder, write graph and syndrome data, start execution, poll status, and read the cycle count and correction result.

7.3.1 AXI4-Lite Register Map

A reusable AXI4-Lite slave implements a 4 KB memory-mapped register window. All decoder wrappers share the same logical register map: a control register pulses `start`, a status register exposes `done` and `busy`, a cycle-count register reports hardware-measured latency, result registers expose the observable or diagnostic outputs, and an input window stores the syndrome or graph data.

The wrapper is deliberately thin: it changes the control interface but not the decoder function. This separation is important because it allows the same core RTL to be used in testbenches, ASIC synthesis, and FPGA integration.

Table 7.2: Common AXI4-Lite register map used by the PYNQ decoder wrappers.

Offset	Name	Dir.	Meaning
0x000	CTRL	W	bit 0 pulses `start`
0x004	STATUS	R	bit 0 `done`, bit 1 `busy`, bit 2 BP convergence flag
0x008	CYCLE_COUNT	R	cycles taken by the last decode
0x00C	RESULT0	R	predicted observable-flip mask
0x010	RESULT1	R	total cost, iteration count, or diagnostic counter
0x014	RESULT2	R	union count or reserved diagnostic field
0x080+	INPUT_WINDOW	W	syndrome, defect list, or CSR graph data

7.3.2 Functional FPGA Sweep

Before interpreting FPGA resource numbers, the wrapped designs were exercised on randomised syndrome instances. For Union–Find and MWPM, the sweep covered code distances $d = 3, 5, \ldots, 21$, five physical error rates, and 25 random syndromes per setting. Across 2 500 total decodes, the RTL outputs matched the Python reference exactly for both implemented decoders. This establishes that the AXI wrapper and FPGA-facing data layout did not introduce functional errors.

7.4 Post-Implementation FPGA Results

Table 7.3 and Figure 7.2 report the ZCU104 post-implementation results for the PYNQ-wrapped decoder cores.

The three cores fit within the device with substantial margin. The largest, `mwpm_pynq`, uses 31.0% of LUTs and less than 10% of flip-flops. The important caveat is that the LUT count is strongly influenced by the common AXI4-Lite register bank, especially the wide read multiplexer

Table 7.3: Post-implementation ZCU104 results for the PYNQ-wrapped decoder cores.

IP core	**LUTs**	**LUT %**	**FFs**	**DSP**	$F_{\max}$ (MHz)	**Power** (W)
`mwpm_pynq`	71 325	31.0	43 748	0	163.4	1.092
`uf_pynq`	67 370	29.2	37 108	0	204.2	1.041
`bp_pynq`	63 937	27.8	34 467	4	90.3	0.984

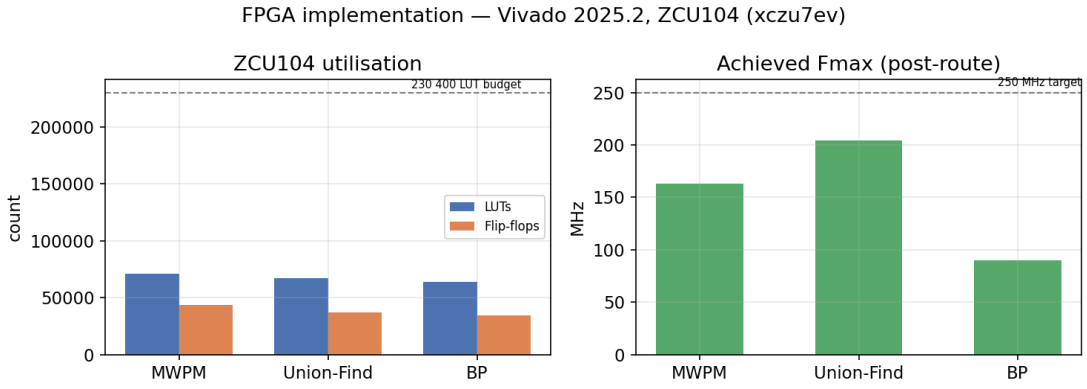

Figure 7.2: ZCU104 utilisation and achieved frequency for the PYNQ-wrapped decoder cores. All three designs fit comfortably, but none closes the aggressive 250 MHz target after full implementation.

over the 4 KB register window. A BRAM-backed register file would reduce this overhead and is a natural implementation improvement.

The achieved frequencies, 90–204 MHz, are lower than the 250 MHz target. This is not surprising: the cores are FSM-oriented, include deep combinational paths, and are wrapped by a large register interface. The post-implementation numbers are therefore more honest than optimistic pre-synthesis expectations.

7.5 Kernel Optimisation Study

The optimisation study re-engineers selected bottleneck kernels and evaluates whether cycle-count reductions survive timing. The metric used throughout is net wall-clock speed-up:

$$\text{net speed-up} = \frac{\text{cycles}_{\text{base}}}{\text{cycles}_{\text{opt}}} \times \frac{F_{\max,\text{opt}}}{F_{\max,\text{base}}}.$$

This product matters because a kernel that halves its cycle count but also halves its clock frequency is not faster in wall-clock time.

7.5.1 Optimisations Applied

Six optimised kernels were evaluated:

- parallel candidate reduction in the MWPM matching DP,
- two versions of MWPM Dijkstra parallelisation,
- combinational multi-hop `find` in the Union–Find DSU,
- Union–Find cluster growth retargeted to the faster DSU,

- four-way processing-element replication in BP.

All optimised kernels passed differential verification against their baselines. The trade-offs are therefore architectural rather than functional.

Table 7.4: Kernel optimisation results: 45 nm area, timing depth, and net wall-clock speed-up.

Kernel	**Area** (μm^2)	**Depth**	$F_{\max}$ (MHz)	**Cycle reduction**	**Net speed-up**	**Verdict**
`mwpm_dijkstra` base	43 512	87	249.1	—	—	baseline
`mwpm_dijkstra_par`	150 724	904	24.5	3.8×	0.4×	regression
`mwpm_dijkstra_par2`	45 400	104	209.2	2.5×	**2.1**×	win
`mwpm_dp_match` base	67 779	57	375.2	—	—	baseline
`mwpm_dp_match_par`	74 131	147	148.9	11.7×	**4.6**×	win
`uf_dsu` base	18 928	48	442.5	—	—	baseline
`uf_dsu_par`	27 817	181	121.3	2.5×	0.7×	regression
`uf_cluster` base	32 947	48	442.5	—	—	baseline
`uf_cluster_par`	40 096	181	121.3	1.9×	0.5×	regression
`bp_top` base	21 579	363	60.8	—	—	baseline
`bp_parallel`	65 563	363	60.8	3.6×	**3.6**×	win

Three results are most important. First, BP processing-element replication is a clean area-for-throughput trade: the cycle count falls by $3.6\times$, the critical path is unchanged, and the net speed-up is also $3.6\times$. Second, the MWPM matching DP benefits strongly from parallel candidate reduction: even after the frequency penalty, the net speed-up is $4.6\times$. Third, the negative results are as informative as the wins. A wide unpipelined scatter/reduction cone can make a design slower even when it uses fewer cycles. The refined Dijkstra implementation shows the remedy: a balanced structure can recover a $2.1\times$ net win where a chain-like implementation regresses.

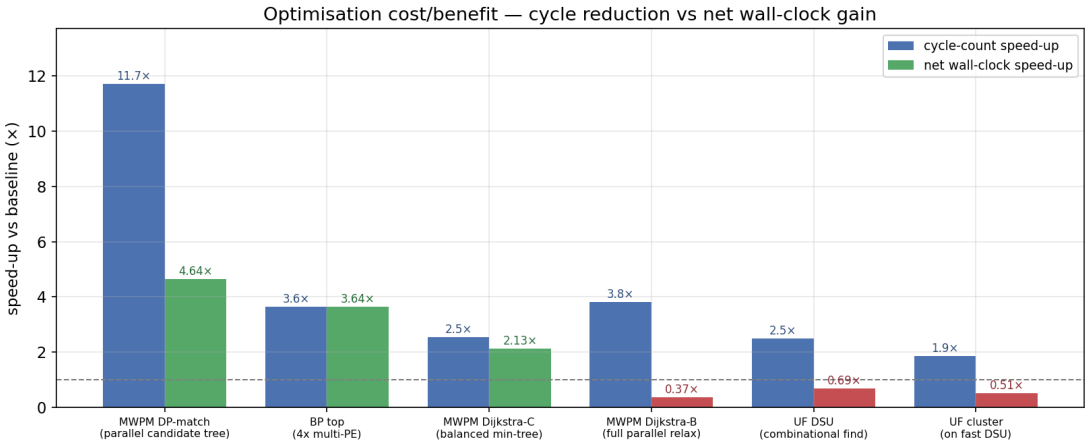

Figure 7.3: Cycle-count speed-up compared with net wall-clock speed-up. The difference between the two exposes optimisations whose critical-path cost cancels their cycle benefit.

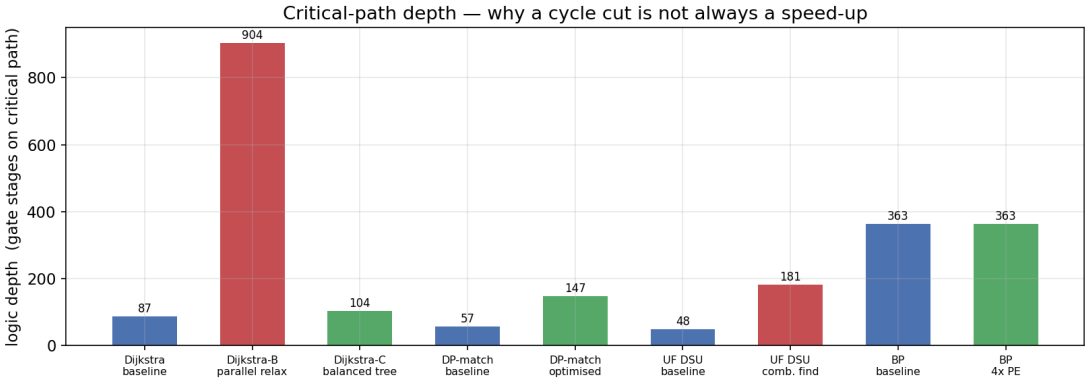

Figure 7.4: Critical-path depth of the baseline and optimised kernels. The 904-stage Dijkstra spike corresponds to an over-aggressive unpipelined parallel-relaxation cone.

7.6 Chapter Summary

The Study-1 hardware results establish a baseline for the rest of the thesis. The verified RTL is small enough to synthesise and integrate, and the ASIC and FPGA results rank the decoder families in ways consistent with their algorithmic structure. At the same time, the optimisation study shows that hardware acceleration must be judged by wall-clock latency, not cycle count alone. Structured reductions, pipelining, and PE replication are productive; unstructured combinational collapse is not.

Chapter 8

Decoding Quality, Distance Scaling, and HLS Comparison

Chapter 7 established that the hand-written RTL kernels can be verified, synthesised, integrated, and optimised. This chapter adds the distance-scaled study: the same three decoder families are evaluated over code distances $d = 3, 5, \ldots, 31$ for decoding quality, RTL cycle count, FPGA resource utilisation, maximum frequency, and HLS comparison.

The purpose of separating this chapter from the implementation chapter is methodological. Logical error rate is a property of the decoder algorithm. Cycle count, resource usage, and maximum frequency are properties of the hardware mapping. A real-time decoder must satisfy both, but measuring them separately makes clear which strategy improves which axis.

8.1 Decoding-Quality Study

The decoding-quality layer uses Stim-generated [17] rotated surface-code memory experiments and evaluates six decoder variants: MWPM,

unweighted Union–Find, weighted Union–Find, plain min-sum BP, normalised min-sum BP, and BP with order-0 ordered-statistics decoding (BP+OSD) [10], [32], [33]. The primary noise model is depolarising code-capacity noise (Appendix B), with a separate circuit-level sweep used to assess weighted Union–Find [29]. MWPM points use 5×10^4 Monte-Carlo shots, while the other decoder families use 4×10^3 shots per point in the code-capacity sweep.

8.1.1 Logical Error Rate versus Code Distance

Figure 8.1 plots logical error rate as a function of code distance at sub-threshold physical error rates. The desired fault-tolerant signature is a downward-sloping line on the logarithmic axis: as d increases, logical error rate should fall exponentially below threshold.

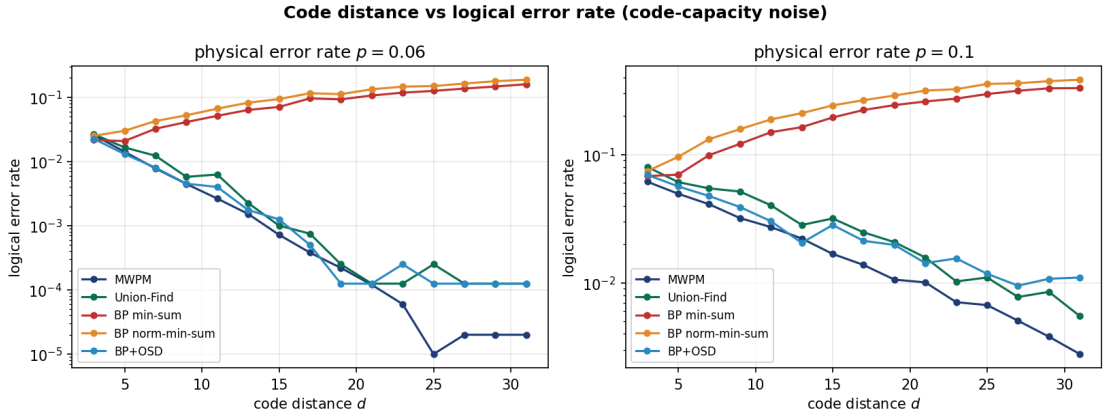

Figure 8.1: Logical error rate versus code distance under code-capacity noise. MWPM, Union–Find, and BP+OSD suppress logical error rate with distance, while plain BP variants fail on the surface-code graph.

The measured behaviour separates the decoders into three groups. MWPM suppresses logical failures cleanly: at $p = 0.06$, its logical error rate falls from 2.582×10^{-2} at $d = 3$ to 2.64×10^{-3} at $d = 11$ and

1.2×10^{-4} at $d = 21$. Union–Find follows the same qualitative trend with a small constant-factor penalty; for example, at $p = 0.08$ and $d = 21$, Union–Find gives 3.5×10^{-3} compared with MWPM's 1.38×10^{-3}.

Plain min-sum BP fails in the expected way on the surface code. At $p = 0.06$, the min-sum BP logical error rate rises from 2.15×10^{-2} at $d = 3$ to 1.59×10^{-1} at $d = 31$. The larger code performs worse because short loops and degeneracy in the surface-code decoding graph prevent plain message passing from converging to a consistent correction. BP+OSD fixes this failure mode: at $d = 11$ and $p = 0.06$, BP+OSD gives 4.0×10^{-3}, close to the MWPM result.

8.1.2 Threshold Behaviour

Figure 8.2 sweeps physical error probability. The MWPM family of curves crosses near $p \approx 15\%$, consistent with the published depolarising code-capacity threshold of $\approx 15.5\%$ for independent X/Z MWPM decoding [39] (cf. Appendix B). Below the crossing, increasing code distance improves logical error rate; above it, larger codes accumulate too many errors for the decoder to correct effectively.

The threshold crossing is visible numerically. At $p = 0.14$, the $d = 31$ MWPM logical error rate is 0.07734, below the $d = 7$ rate of 0.10564. At $p = 0.16$, the order reverses: $d = 31$ gives 0.1833, while $d = 7$ gives 0.14622. This validates the simulator and reference decoding flow before hardware conclusions are drawn from the same graph family.

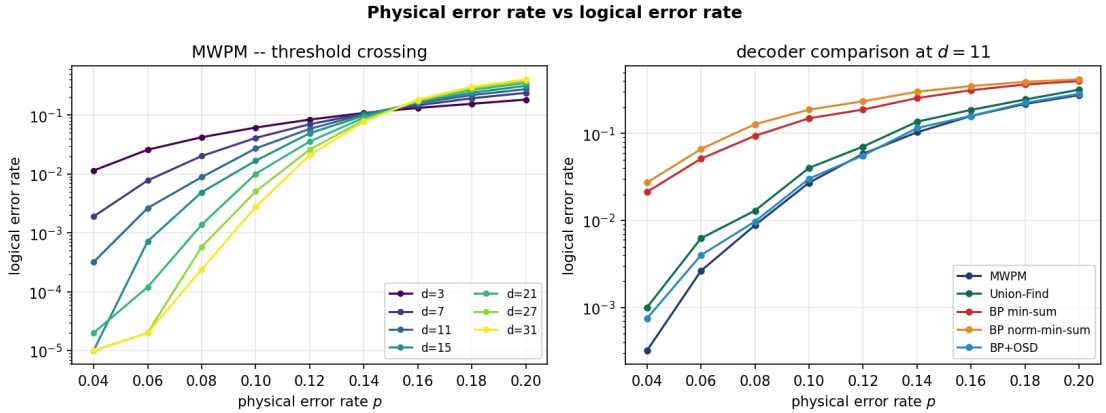

Figure 8.2: Logical error rate versus physical error rate. The MWPM curves show a threshold crossing near $p \approx 15\%$; at fixed distance, BP+OSD is competitive with MWPM, Union–Find is slightly worse, and plain BP variants are much weaker.

8.1.3 Weighted Union–Find under Circuit-Level Noise

Weighted Union–Find is evaluated under circuit-level noise because depolarising code-capacity noise gives nearly uniform edge weights, leaving little for weighted growth to exploit. Figure 8.3 shows that weighted growth improves high-error-regime performance. At $d = 9$ and $p = 1.2 \times 10^{-2}$, unweighted Union–Find gives a logical error rate of 0.3395, while weighted Union–Find [29] reduces it to 0.285, a relative improvement of approximately 16%. MWPM remains stronger at the same operating point, with $p_L = 0.2091$.

8.2 RTL Latency versus Code Distance

The hardware layer measures latency using cycle-accurate Icarus Verilog simulation. For every code distance, the corresponding graph size is emitted as memory images, the parameterised kernel is elaborated, and the hardware cycle counter reports latency.

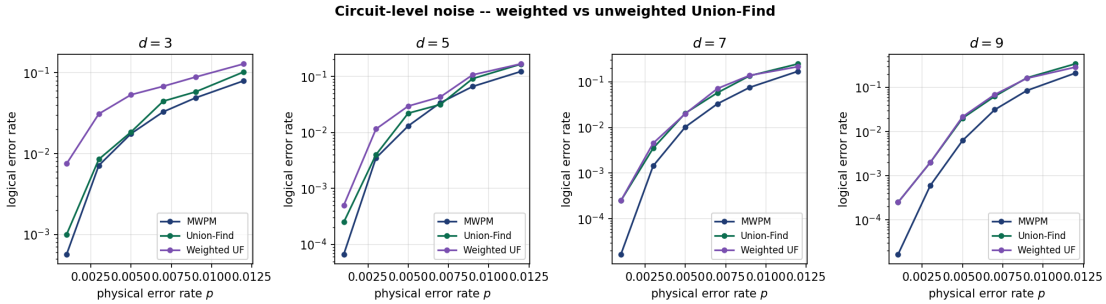

Figure 8.3: Circuit-level noise comparison of MWPM, unweighted Union–Find, and weighted Union–Find. Weighted growth improves the high-error-regime logical error rate by using non-uniform edge probabilities.

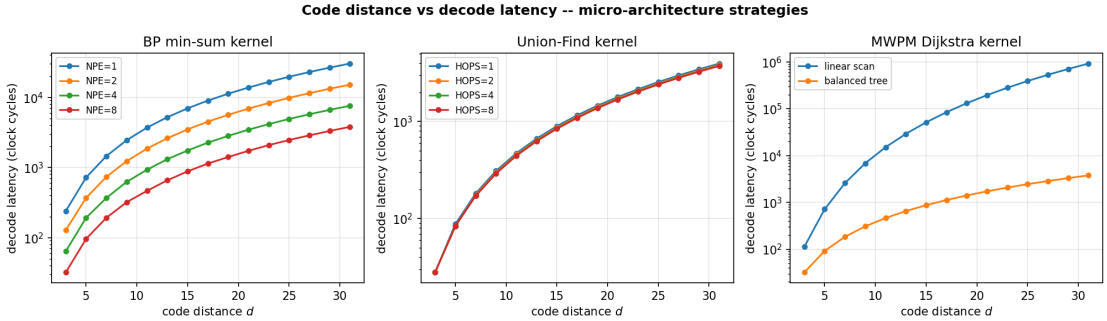

Figure 8.4: RTL decode latency in clock cycles versus code distance. BP processing-element replication scales nearly linearly; Union–Find multi-hop traversal gives only modest cycle reduction; MWPM parallel extraction dramatically reduces cycle count.

Table 8.1: Decode latency endpoints from the distance sweep. The speed-up is baseline divided by the fastest strategy at $d = 31$.

Kernel	**Strategy**	$d = 3$	$d = 31$	**Speed-up**
`bp_minsum`	`NPE=1`	240	30 256	—
`bp_minsum`	`NPE=8`	32	3 792	7.98×
`uf_grow`	`HOPS=1`	28	3 962	—
`uf_grow`	`HOPS=8`	28	3 740	1.06×
`mwpm_dijkstra_k`	`PAR=0`	113	927 305	—
`mwpm_dijkstra_k`	`PAR=1`	32	3 784	245×

The BP result is the cleanest cycle-count result: the latency is essentially proportional to $1/\texttt{NPE}$. Union–Find behaves differently. Even HOPS=8 gives only a $1.06\times$ cycle reduction at $d = 31$, indicating that parent chains are not the dominant latency source for these local surface-code graphs. MWPM shows the most dramatic cycle-count improvement: replacing the linear scan with parallel frontier extraction converts an $O(d^4)$-like curve into an $O(d^2)$-like curve. Whether that cycle reduction is a wall-clock speed-up depends on $F_{\max}$, measured next.

8.3 FPGA Resource Utilisation and Maximum Frequency

Every distance-scaled RTL implementation is synthesised out of context using Vivado 2025.2 for the ZCU104 target part, with a 4 ns clock target. The flow reports LUTs, flip-flops, BRAM use, and worst negative slack, from which $F_{\max} = 1000/(4 - \mathrm{WNS})$ MHz is computed.

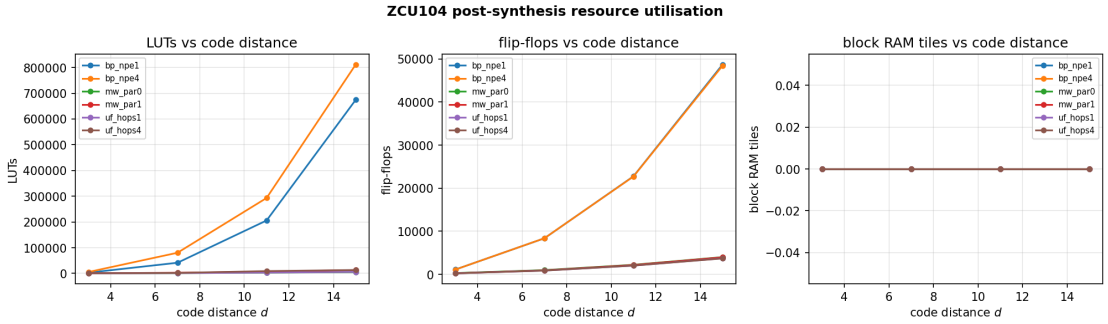

Figure 8.5: ZCU104 post-synthesis resource utilisation versus code distance. BP grows rapidly when synthesised as one flat whole-code instance; MWPM and Union–Find remain modest over the measured range.

The FPGA results sharpen the lesson from Chapter 7. BP PE repli-

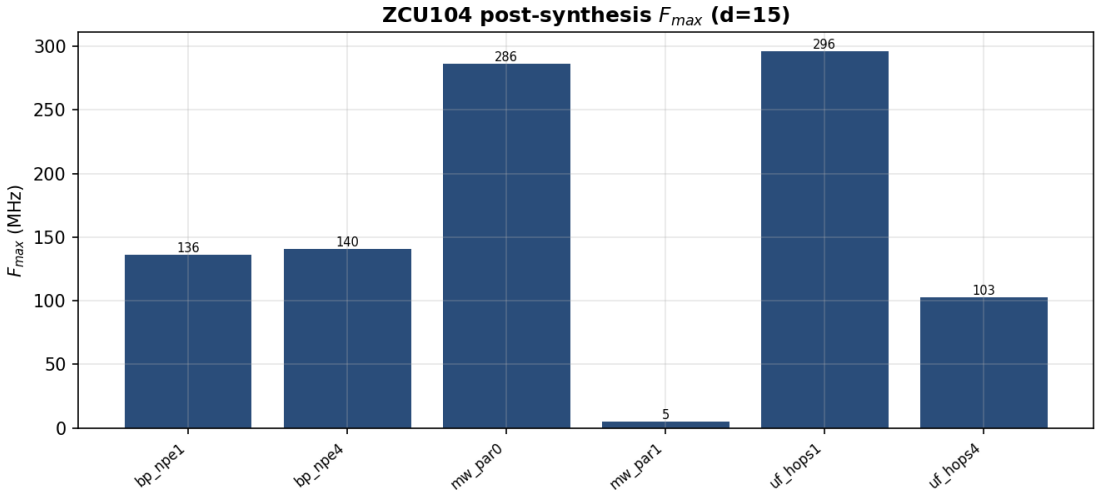

Figure 8.6: Post-synthesis $F_{\max}$ at $d = 15$. The MWPM combinational extraction and the multi-hop Union–Find traversal reduce cycle count but lengthen the critical path substantially.

Table 8.2: ZCU104 post-synthesis results at $d = 15$. Wall-clock time is $T = \text{cycles}/F_{\max}$.

Implementation	**LUTs**	**FFs**	$F_{\max}$ (MHz)	**Cycles**	T (μs)
`bp_minsum`, `NPE=1`	674 355	48 686	136.05	6 960	51.16
`bp_minsum`, `NPE=4`	811 427	48 417	140.49	1 744	12.41
`uf_grow`, `HOPS=1`	6 013	3 737	296.03	898	3.03
`uf_grow`, `HOPS=4`	13 319	3 707	102.75	852	8.29
`mwpm_dijkstra_k`, `PAR=0`	5 243	4 004	286.20	51 497	179.93
`mwpm_dijkstra_k`, `PAR=1`	13 096	3 959	4.79	872	182.05

cation is a genuine wall-clock win: T improves from 51.16 to 12.41 μs at $d = 15$ because $F_{\max}$ is preserved. MWPM `PAR=1` is a cycle-count win but not a wall-clock win: the cycle count falls by nearly $60\times$, but $F_{\max}$ collapses to 4.79 MHz, making the net time essentially equal to the baseline. Union–Find `HOPS=4` is worse in wall-clock time because the cycle reduction is small and the frequency penalty is large.

The area result is also important. A flat BP instance sized for an entire distance-d code reaches 205 845 LUTs at $d = 11$ and exceeds the ZCU104 capacity at $d = 15$. This does not mean BP is unusable; it means that a deployable design must be tiled or memory-backed rather than synthesised as one flat whole-code instance.

8.4 Hand-Written RTL versus Vitis HLS

To compare manual RTL with high-level hardware generation, each kernel is also implemented in C++ and synthesised with Vitis HLS 2025.2 at $d = 11$. The hand-RTL rows use the best wall-clock RTL configuration for each algorithm at that distance.

Table 8.3: Hand-written RTL versus Vitis HLS at $d = 11$ on the ZCU104.

Algorithm	**Source**	**LUTs**	**FFs**	**BRAM**	$F_{\max}$ (MHz)	T (μs)
BP min-sum	hand RTL, `NPE=4`	293 233	22 715	0	112.15	8.27
BP min-sum	Vitis HLS	2 118	444	4	344.95	23.39
Union–Find	hand RTL, `HOPS=1`	3 242	2 036	0	333.33	1.42
Union–Find	Vitis HLS	701	327	1	319.59	49.40
MWPM Dijkstra	hand RTL, `PAR=0`	3 142	2 226	0	309.02	48.88
MWPM Dijkstra	Vitis HLS	1 531	550	0	345.42	51.16

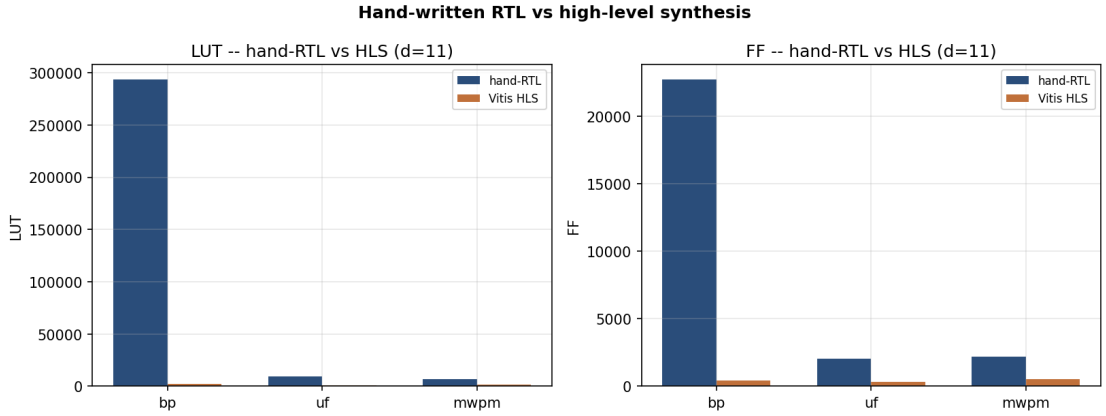

Figure 8.7: Area comparison between the best hand-written RTL configurations and Vitis HLS at $d = 11$. HLS infers block RAM for memories that the current hand RTL maps into registers or LUT storage.

HLS is much smaller, especially for BP. The HLS BP implementation uses 2 118 LUTs and four BRAM tiles, while the hand-RTL BP instance uses 293 233 LUTs and no BRAM. This difference is mainly memory inference: HLS recognises the large message arrays as memories, whereas the hand RTL currently exposes many of them as register-array or LUT-RAM structures. Manual RTL wins on latency, especially for Union–Find, where hand-written FSM control handles data-dependent `find` loops more efficiently than the conservative HLS schedule.

The comparison does not imply that one methodology dominates the other. HLS is attractive for area-efficient memory mapping and rapid exploration. Hand RTL remains preferable when deterministic low latency is the binding constraint. The most promising path combines both lessons: keep the latency-critical control explicit in RTL, but refactor large arrays into BRAM-friendly access patterns.

8.5 Chapter Summary

The distance-scaled study produces three thesis-level conclusions. First, decoder accuracy is moved by algorithmic choices: OSD rescues BP, weighted growth improves Union–Find under non-uniform noise, and MWPM remains the reference-quality decoder. Second, wall-clock latency is not the same as cycle count; BP replication converts directly into speed, while unpipelined MWPM and Union–Find combinational shortcuts do not. Third, scalable hardware mapping is fundamentally a memory and tiling problem. Flat whole-code kernels are useful for measurement, but deployable large-distance decoders require tiled processing elements and explicit memory architecture.

Chapter 9

Conclusion and Future Work

This thesis studied the hardware acceleration of surface-code quantum error-correction decoders from two connected viewpoints: the algorithmic ability to suppress logical errors and the hardware ability to meet a real-time latency budget. The three decoder families considered–MWPM, Union–Find, and BP–span the central trade-off in the field: MWPM gives strong decoding quality but uses irregular graph computation, Union–Find is hardware-friendly and near-linear but slightly less accurate, and BP maps naturally to parallel message-passing hardware but needs additional algorithmic support on surface-code graphs.

9.1 Summary of Contributions

The work makes the following contributions.

- A profiling-driven kernel selection study identified the dominant computational routines of each decoder family and used those measurements to define the RTL targets.
- The selected kernels were implemented in synthesisable Verilog-2005

with common start–done control, Q8.8 fixed-point arithmetic, CSR graph input formatting, and portable flat-bus interfaces.

- The RTL was verified against independent golden models and differential testbenches before any synthesis result was interpreted.
- The verified kernels and decoder tops were characterised using a 45 nm Nangate ASIC synthesis flow and integrated into PYNQ-compatible ZCU104 FPGA cores.
- A kernel optimisation study quantified net wall-clock speed-up rather than cycle-count speed-up alone, exposing both genuine wins and negative results.
- A distance-scaled study measured logical error rate, RTL latency, FPGA resource utilisation, maximum frequency, and HLS comparison over code distances $d = 3, 5, \ldots, 31$.

9.2 Main Findings

The first major finding is that decoding quality is controlled by algorithmic strategy. MWPM shows the expected depolarising code-capacity threshold near $p \approx 15\%$ [39] and remains the reference-quality decoder. Union–Find tracks MWPM within a small constant factor while offering a simpler hardware structure. Plain min-sum BP fails on the surface code because degeneracy and short cycles in the Tanner graph prevent reliable convergence [10], [32], but BP+OSD restores performance close to MWPM. Weighted Union–Find [29] improves logical error rate under circuit-level noise near the high-error regime where edge weights matter.

The second major finding is that hardware speed must be measured in wall-clock time. BP processing-element replication is the cleanest acceleration: it reduces cycle count in proportion to the number of processing elements while preserving the critical path. In contrast, naive single-cycle combinational reductions can be misleading. MWPM frontier extraction reduced the cycle count by orders of magnitude, but when written as an unpipelined behavioural reduction it collapsed the FPGA frequency enough to erase the wall-clock benefit. Union–Find multi-hop traversal similarly reduced cycles too little to justify the frequency penalty on the local surface-code graphs considered here.

The third finding is that scalable implementation is a memory architecture problem as much as a logic problem. The HLS comparison shows that automatic memory inference can reduce LUT use dramatically by moving large arrays into BRAM, while hand-written RTL still provides lower latency on control-heavy kernels. Future decoder hardware should therefore combine explicit RTL control for latency-critical paths with BRAM-friendly storage and tiled processing-element layouts.

9.3 Limitations

The study has several deliberate boundaries. The ASIC results are pre-layout synthesis results and do not include extracted parasitics or sign-off static timing. The distance-scaled FPGA flow stops at out-of-context synthesis, so place-and-route effects are not captured in those results. The full PYNQ implementation study demonstrates integration but is limited to modest kernel configurations. The logical-error-rate sweep

uses code-capacity noise as the primary model; circuit-level noise is evaluated only for the weighted Union–Find comparison. Finally, the hardware kernels accelerate dominant routines rather than implementing a complete production decoder stack with streaming I/O, scheduler, memory hierarchy, and live quantum-control integration.

9.4 Future Work

Four directions follow directly from the results.

1. **Pipeline the regressing reductions.** The MWPM frontier extraction and Union–Find multi-hop traversal should be rewritten as balanced, registered pipelines. This would preserve much of the cycle-count advantage while restoring clock frequency.

2. **Introduce tiled decoder architectures.** Flat whole-code kernels are useful for measurement, but large-distance decoders require tiled processing elements similar to Helios-style distributed Union–Find and systolic BP message-passing arrays.

3. **Refactor memories into BRAM-friendly structures.** The HLS study shows that the largest area savings come from memory inference. The hand RTL should be reworked to use explicit two-port memory access patterns for BP messages, parent arrays, and edge data.

4. **Close the end-to-end real-time loop.** The next implementation step is to stream syndrome data through a placed-and-routed FPGA overlay, measure end-to-end latency including host/control overhead,

and compare it directly against the microsecond-scale QEC cycle budget.

9.5 Closing Remark

A practical fault-tolerant quantum computer will need a decoder that is not merely accurate in simulation or fast in an isolated kernel benchmark, but both accurate and fast in a hardware system. This thesis shows how to evaluate those requirements separately and then combine them honestly. The central engineering lesson is simple but strict: measure the algorithm for correctness and logical error rate, measure the hardware for cycles and frequency, and trust only the wall-clock result that survives both.

Appendices

Appendix A

Stabilizers, Syndromes, and the Decoding Graph

This appendix collects the structural facts about stabilizer codes, syndrome generation, and the surface-code decoding graph that are used (often implicitly) throughout the thesis. The presentation follows Gottesman [2] and the surface-code papers [4], [6], [7].

A.1 Stabilizer Formalism

Let $\mathcal{P}_n$ denote the n-qubit Pauli group, generated by tensor products of $\{I, X, Y, Z\}$ and the phases $\{\pm 1, \pm i\}$. A *stabilizer group* $\mathcal{S} \subset \mathcal{P}_n$ is an Abelian subgroup that does not contain $-I$. A stabilizer code is the joint $+1$ eigenspace of $\mathcal{S}$:

$$\mathcal{C} \;=\; \{\, |\psi\rangle \in \mathbb{C}^{2^n} : g\,|\psi\rangle = |\psi\rangle \;\; \forall g \in \mathcal{S} \,\}.$$

If $\mathcal{S}$ has $n-k$ independent generators $\{g_1, \ldots, g_{n-k}\}$, the code encodes k logical qubits in n physical qubits and is written $[\![n, k, d]\!]$, where the

code distance d is the minimum weight of any non-trivial logical operator in $\mathcal{N}(\mathcal{S}) \setminus \mathcal{S}$.

A Pauli error $E \in \mathcal{P}_n$ either commutes or anti-commutes with each stabilizer generator. The bit pattern of anti-commutations is the *syndrome* $\mathbf{s}(E) \in \{0,1\}^{n-k}$. Two errors E, E' are indistinguishable by syndrome iff $E^{-1}E' \in \mathcal{N}(\mathcal{S})$; they are correctable to the same logical operation iff $E^{-1}E' \in \mathcal{S}$. Decoding is therefore the task of inferring, from $\mathbf{s}$, an error $\hat{E}$ such that $E^{-1}\hat{E}$ lies in $\mathcal{S}$ rather than $\mathcal{N}(\mathcal{S}) \setminus \mathcal{S}$.

Surface-Code Stabilizers

On a rotated $d \times d$ surface code, data qubits sit on the vertices of a square lattice. There are two families of stabilizers:

$$A_v = \prod_{q \in v} X_q \qquad \text{(vertex / star-type generator)},$$

$$B_p = \prod_{q \in p} Z_q \qquad \text{(plaquette / face-type generator)}.$$

Each generator acts on the (up to four) qubits incident to a vertex v or face p. The two families commute because every X- and Z-type stabilizer share an even number of qubits. There are $d^2 - 1$ independent generators, leaving $k = 1$ logical qubit. Logical operators are non-trivial strings of X (or Z) operators connecting opposite boundaries; the shortest such string has length d.

A.2 Syndromes and the Detection-Event Picture

In hardware experiments, stabilizers are measured every QEC cycle through ancilla-mediated Hadamard+CNOT circuits. Because measurements are noisy, the relevant decoding signal is not the raw measurement outcome but a *detection event*: a stabilizer measurement that differs from the previous round on the same ancilla.

Errors map to detection events through a linear-algebraic relation

$$\mathbf{s} \;=\; H\,\mathbf{e} \pmod 2,$$

where H is the parity-check matrix of the code (extended in time when syndromes are noisy). For the surface code, H is sparse: each row has weight at most four, and each column has weight at most four. The sparse, locally connected structure is precisely what lets cluster decoders such as Union–Find run in near-linear time.

The Matching Graph

The MWPM and Union–Find decoders both operate on the *matching graph* $G_M = (V, E)$:

- V: one vertex per stabilizer (detector). For boundary-touching edges, a single *virtual boundary node* is added per family (X- and Z-type).
- E: one edge for every elementary fault that triggers at most two detectors. Edge weights are $w_e = \log((1 - p_e)/p_e)$, so that the minimum-weight matching maximises the joint likelihood of the

implied error string.

A defect is a vertex whose detection event is 1. The decoding problem reduces to finding a minimum-weight set of edges $T \subseteq E$ whose endpoints exactly cover the defect set — equivalently, a minimum-weight perfect matching on the complete graph of pairwise shortest-path distances. Figure F.1 in Appendix F reproduces the finite-state-machine view of the matching graph used by the RTL implementation.

The Tanner Graph for Belief Propagation

For BP decoding, the same parity-check matrix H is interpreted as a bipartite Tanner graph $G_T = (V_v \cup V_c, E_T)$: V_v contains one variable node per qubit (or per fault location), V_c contains one check node per stabilizer, and an edge $(v, c) \in E_T$ exists iff $H_{cv} = 1$. Messages flow along these edges during decoding (Algorithm C). The surface code is a notoriously difficult instance for plain BP because the Tanner graph contains many short cycles of length four, which violate the tree-assumption that justifies BP convergence [10], [32].

Appendix B

Error Models and Threshold Estimates

B.1 Code-Capacity Noise Model

The code-capacity model assumes perfect syndrome extraction and applies independent Pauli errors only on data qubits. The two standard variants used in this thesis are:

- **Bit-flip channel:** each qubit suffers X with probability p, I otherwise.
- **Depolarising channel:** each qubit suffers each of $\{X, Y, Z\}$ with probability $p/3$, I with probability $1 - p$.

The depolarising channel is the relevant code-capacity model for single-decoder X/Z inference because it produces correlated X and Z syndromes on a fraction of qubits.

B.2 Phenomenological and Circuit-Level Noise

The phenomenological noise model adds independent measurement flips with probability p on top of the code-capacity errors. The circuit-level model further injects:

- single-qubit depolarising errors after each idle slot,
- two-qubit depolarising errors after every CNOT,
- Pauli-X errors before each measurement and after each reset.

The Stim detector-error model used in this thesis follows the circuit-level prescription described in [11], [17].

B.3 Known Threshold Estimates

Table B.1 summarises the published threshold estimates that calibrate the decoding-quality results in Chapter 8.

Table B.1: Reference threshold estimates for the rotated surface code.

Noise model	**Decoder**	**Threshold p_{th}**
Code capacity, bit-flip	MWPM (optimal)	$\approx 10.9\%$ [4]
Code capacity, depolarising	MWPM (independent X/Z)	$\approx 15.5\%$ [39]
Code capacity, depolarising	Optimal (correlated)	$\approx 18.9\%$ [39]
Phenomenological	MWPM	$\approx 2.9\%$ [4], [40]
Circuit level (Google CZ)	MWPM/UF	≈ 0.7–1.0% [7], [11]

The MWPM threshold crossing observed near $p \approx 15\%$ in Chapter 8 is consistent with the depolarising code-capacity estimate. The lower

circuit-level threshold motivates the real-time hardware requirement that drives the entire thesis: at $p \sim 10^{-3}$, a working code distance must be tens of qubits deep, and the decoder must process the corresponding syndrome volume each microsecond-scale QEC cycle [11], [13].

Appendix C

Decoder Algorithms (Pseudocode)

The pseudocode below reflects the actual decoder implementations used in the thesis. Notation: $G_M = (V, E)$ is the matching graph, $D \subseteq V$ is the set of defects, H is the parity-check matrix, $\mathbf{s}$ is the syndrome, $\mathbf{m}_{v \to c}$ and $\mathbf{m}_{c \to v}$ are the BP messages.

Algorithm 1: Dijkstra Shortest Path (MWPM kernel)

```
input  : graph G_M, source s
output : dist[v] for all v in V, parent[v], obs_xor[v]
1  for each v in V do dist[v] := +inf; parent[v] := -1; obs_xor[v] := 0
2  dist[s] := 0
3  while exists an unsettled vertex with dist < +inf do
4      u := unsettled vertex with min dist[u]    // linear scan or tree-reduce
5      mark u settled
6      for each edge (u,v,w,obs) in G_M do
7          if dist[u] + w < dist[v] then
8              dist[v]    := dist[u] + w
9              parent[v]  := u
10             obs_xor[v] := obs_xor[u] XOR obs
11 return dist, parent, obs_xor
```

Algorithm 2: Bitmask DP Matching (MWPM kernel)

```
input  : pair-cost matrix C[1..K,1..K], boundary-cost vector B[1..K]
output : min-cost perfect matching value and choice table
1  dp[empty set] := 0
2  for each subset S of {1..K} in increasing popcount order do
3      if S is empty then continue
4      i := lowest-indexed defect in S
5      dp[S] := B[i] + dp[S \ {i}]                        // match i to boundary
6      choice[S] := ("boundary", i)
7      for each j in S with j > i do
8          c := C[i,j] + dp[S \ {i,j}]
9          if c < dp[S] then
10             dp[S]     := c
11             choice[S] := ("pair", i, j)
12 return dp[full set], choice
```

Algorithm 3: Synchronous Union–Find Decoder

```
input  : matching graph G_M = (V,E), defect set D
output : correction set T (subset of E)
1  initialise DSU; create singleton cluster {v} for each v in D
2  for each cluster C: parity:=1; grown_edges:=empty; boundary:=false
3  while there exists an odd-parity cluster do
4      for each odd-parity cluster C in parallel do
5          for each unfilled edge incident to C do
6              advance the edge growth counter by 1/2
7              if the edge becomes fully grown then
8                  add it to C.grown_edges
9                  let v be the new vertex reached
10                 if v is a boundary node then C.boundary := true
11                 else union(C, cluster_of(v))    // may merge clusters
12     update C.parity := XOR of defects in C and C.boundary
13 T := peeling(C.grown_edges for each cluster)
14 return T
```

Algorithm 4: Normalised Min-Sum Belief Propagation

```
input  : check matrix H, prior LLRs L, syndrome s, max iters T_max, alpha
output : hard decision e_hat, convergence flag
1  for each edge (v,c) in Tanner graph do m_{v->c} := L[v]
2  for t = 1 to T_max do
3      // ---------- check-node update ----------
4      for each check node c do
5          let mins[c]   := smallest |m_{v->c}| over v ~ c
6          let mins2[c]  := second-smallest |m_{v->c}| over v ~ c
7          let prod[c]   := product of sign(m_{v->c}) over v ~ c
8          let sigma[c]  := (-1)^{s[c]} * prod[c]
9          for each v ~ c do
10             mag := (|m_{v->c}| == mins[c]) ? mins2[c] : mins[c]
11             m_{c->v} := alpha * sigma[c] * sign(m_{v->c}) * mag
12     // ---------- variable-node update -------
13     for each variable node v do
14         belief[v] := L[v] + sum_{c ~ v} m_{c->v}
15         e_hat[v]  := (belief[v] < 0) ? 1 : 0
16         for each c ~ v do
17             m_{v->c} := belief[v] - m_{c->v}
18     if H * e_hat == s then return (e_hat, converged = true)
19 return (e_hat, converged = false)
```

Algorithm 5: Order-0 OSD Post-Processing

```
input  : parity-check matrix H, BP belief vector belief, syndrome s
output : recovered error e_hat such that H * e_hat = s
1  sort columns of H by descending reliability |belief[v]|
2  apply Gaussian elimination on the reordered H to obtain
   an information set I of cardinality rank(H)
3  set e_hat[v] := 0 for all v not in I
4  solve H[:, I] * e_hat[I] = s by back-substitution
5  un-permute columns and return e_hat
```

Appendix D

Representative RTL Listings

This appendix reproduces compact, synthesis-ready Verilog excerpts that implement the three central kernels described in Chapter 6. Comments above each block name the corresponding algorithm. The fixed-point convention is Q8.8 throughout, with `INF = 16'sh7FFF` used as the shortest-path sentinel.

D.1 MWPM Dijkstra Kernel (Frontier Reduction)

```
// mwpm_dijkstra_k.v  --  optimised frontier extraction (PAR=1)
// Selects the unsettled vertex with minimum dist[] using a
// balanced log-depth comparator tree instead of a linear scan.

module argmin_tree #(parameter N = 64, parameter W = 16) (
    input  signed [W*N-1:0] vals,
    input              [N-1:0] valid,
    output reg signed [W-1:0]  min_val,
    output reg [$clog2(N)-1:0] min_idx
);
    integer i;
    reg signed [W-1:0] v_i;
```

```
    always @* begin
        min_val = 16'sh7FFF;     // +INF
        min_idx = {($clog2(N)){1'b0}};
        for (i = 0; i < N; i = i + 1) begin
            v_i = vals[W*i +: W];
            if (valid[i] && v_i < min_val) begin
                min_val = v_i;
                min_idx = i[$clog2(N)-1:0];
            end
        end
    end
endmodule

// Per-cycle relaxation pipeline stage
always @(posedge clk) if (state == RELAX) begin
    for (k = 0; k < OUT_DEG; k = k + 1) begin
        v   = adj_dst[u_idx][k];
        wgt = adj_w  [u_idx][k];
        if (!settled[v] && (dist[u_idx] + wgt) < dist[v]) begin
            dist     [v] <= dist[u_idx] + wgt;
            parent   [v] <= u_idx;
            obs_xor  [v] <= obs_xor[u_idx] ^ adj_o[u_idx][k];
        end
    end
end
```

D.2 Union–Find Disjoint-Set Engine (HOPS Parameter)

```
// uf_grow.v  --  combinational multi-hop find (HOPS = 1, 2, 4, 8)
// Resolves up to HOPS parent links per cycle, applying path halving.

module uf_find #(parameter HOPS = 4, parameter N = 64) (
    input  wire [$clog2(N)-1:0] x,
    input  wire [$clog2(N)*N-1:0] parent_flat,
    output reg  [$clog2(N)-1:0] root
);
```

```
    integer h;
    reg [$clog2(N)-1:0] cur, par;
    always @* begin
        cur = x;
        for (h = 0; h < HOPS; h = h + 1) begin
            par = parent_flat[$clog2(N)*cur +: $clog2(N)];
            if (par != cur) cur = par;          // climb
        end
        root = cur;                  // not full root if HOPS too small
    end
endmodule

// Union by rank, executed in a single FSM state
always @(posedge clk) if (state == UNION) begin
    if (rank_a > rank_b)       parent[rb] <= ra;
    else if (rank_a < rank_b)  parent[ra] <= rb;
    else begin parent[rb] <= ra; rank_a <= rank_a + 1; end
end
```

D.3 BP Min-Sum Check-Node Update (NPE Replication)

```
// bp_check_node.v  --  normalised min-sum, alpha = 0.75 = (3/4)
// Streams 'deg' incoming variable-to-check messages, emits 'deg'
// check-to-variable messages. Parallelised by instantiating NPE copies.

module bp_check_node #(parameter DEG = 8, parameter W = 16) (
    input  signed [W*DEG-1:0] m_in,    // variable -> check
    input                     synd_c,  // syndrome bit for this check
    output reg signed [W*DEG-1:0] m_out // check -> variable
);
    integer i;
    reg signed [W-1:0] x, mag;
    reg [W-1:0] absv [0:DEG-1];
    reg sgn [0:DEG-1];
    reg signed [W-1:0] min1, min2;
    reg sigma;
```

```
    always @* begin
        min1 = 16'sh7FFF; min2 = 16'sh7FFF; sigma = synd_c;
        for (i = 0; i < DEG; i = i + 1) begin
            x      = m_in[W*i +: W];
            sgn[i] = x[W-1];
            absv[i]= x[W-1] ? -x : x;
            sigma  = sigma ^ sgn[i];
            if (absv[i] < min1) begin min2 = min1; min1 = absv[i]; end
            else if (absv[i] < min2) min2 = absv[i];
        end
        for (i = 0; i < DEG; i = i + 1) begin
            mag = (absv[i] == min1) ? min2 : min1;
            // alpha = 3/4 implemented as (mag - mag>>>2)
            mag = mag - (mag >>> 2);
            m_out[W*i +: W] = (sigma ^ sgn[i]) ? -mag : mag;
        end
    end
endmodule
```

D.4 AXI4-Lite Wrapper Register Map (Common to All PYNQ Cores)

```
// axi_decoder_regs.v  --  4 KB register window
localparam OFF_CTRL       = 12'h000;   // bit 0: pulse start
localparam OFF_STATUS     = 12'h004;   // bit 0: done; bit 1: busy; bit 2: conv
localparam OFF_CYCLES     = 12'h008;   // measured decode latency
localparam OFF_RESULT0    = 12'h00C;   // predicted observable mask
localparam OFF_RESULT1    = 12'h010;   // total cost / iteration count
localparam OFF_RESULT2    = 12'h014;   // union count / diagnostic
localparam OFF_INPUT_BASE = 12'h080;   // syndrome / CSR graph data

always @(posedge clk) begin
    if (axi_write && axi_waddr[11:0] == OFF_CTRL && axi_wdata[0])
        start <= 1'b1;
    else
        start <= 1'b0;
```

```
end
```

Appendix E

Reproducibility Notes

- **Software stack.** Stim 1.13, PyMatching 2.1, `ldpc` 0.1.55 (Python); Apple Clang 15 for C++ profiling.
- **RTL simulation.** Icarus Verilog 12.0 with VCD waveform dumps; differential testbenches drive baseline and optimised RTL with identical stimuli and assert bit-exact equality of outputs and `done` timing.
- **ASIC flow.** Yosys 0.40 with `synth`, `dfflibmap`, and ABC commands targeting the Nangate 45 nm OpenCell Library at the typical $(1.10\,\mathrm{V}, 25\,^{\circ}\mathrm{C})$ corner [20], [21]; cell area, leakage, and flop count parsed from the synthesis log.
- **FPGA flow.** Vivado 2025.2 out-of-context synthesis for the ZCU104 XCZU7EV part [19]; 4 ns target clock; $F_{\max} = 1000/(4-\mathrm{WNS})$ MHz reported.
- **HLS flow.** Vitis HLS 2025.2 [23]; identical C++ algorithm files compiled once per kernel with `#pragma HLS PIPELINE` on inner loops and `#pragma HLS ARRAY_PARTITION` on small look-up arrays only.

- **PYNQ overlay.** Generated against the PYNQ 3.0 ZCU104 base image [34]; AXI4-Lite slave attached to the PS-PL high-performance bus.

Appendix F

Finite-State Machine Diagrams for the RTL Kernels

For completeness, the high-level FSM views of the three RTL decoder kernels are reproduced below. They show the control structure that sequences the algorithmic operations described in Appendix C and that is implemented by the Verilog excerpts in Appendix D.

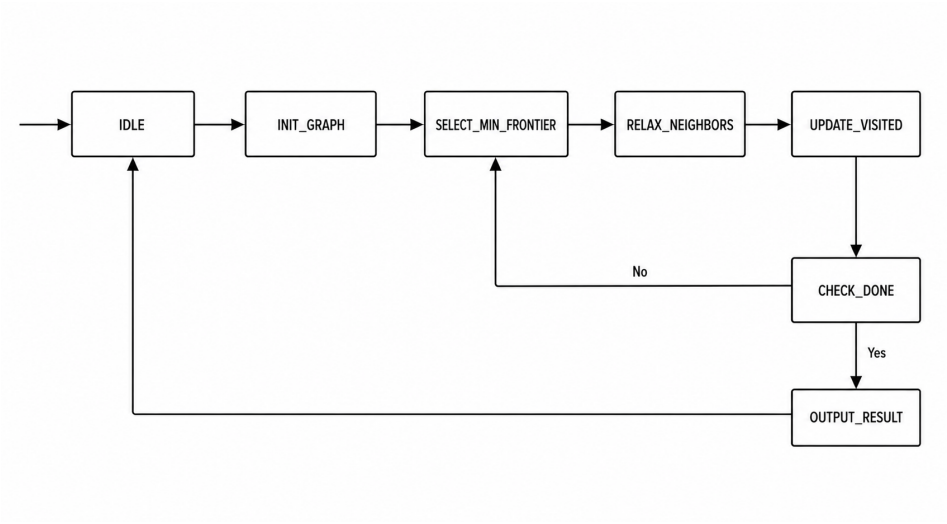

Figure F.1: Top-level FSM of the MWPM decoder kernel (Dijkstra plus bitmask matching DP).

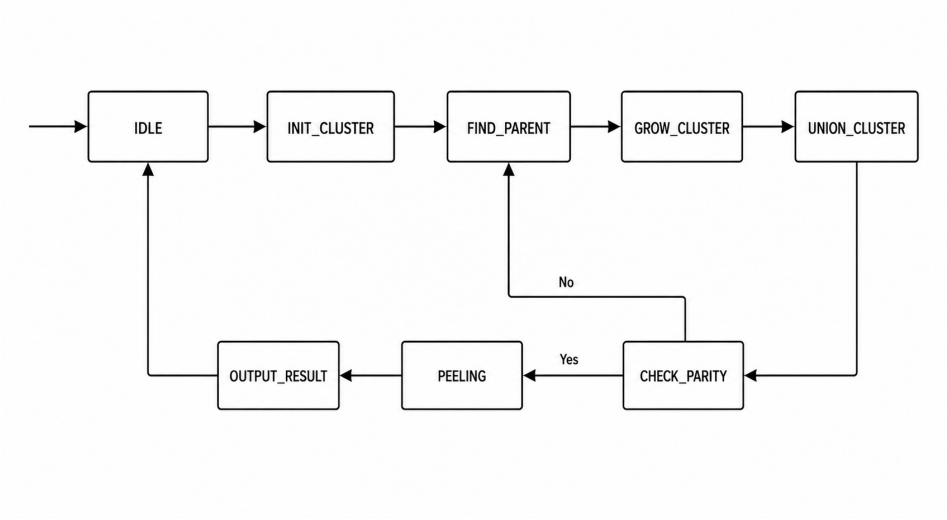

Figure F.2: Top-level FSM of the Union–Find decoder kernel (cluster growth, union, peeling).

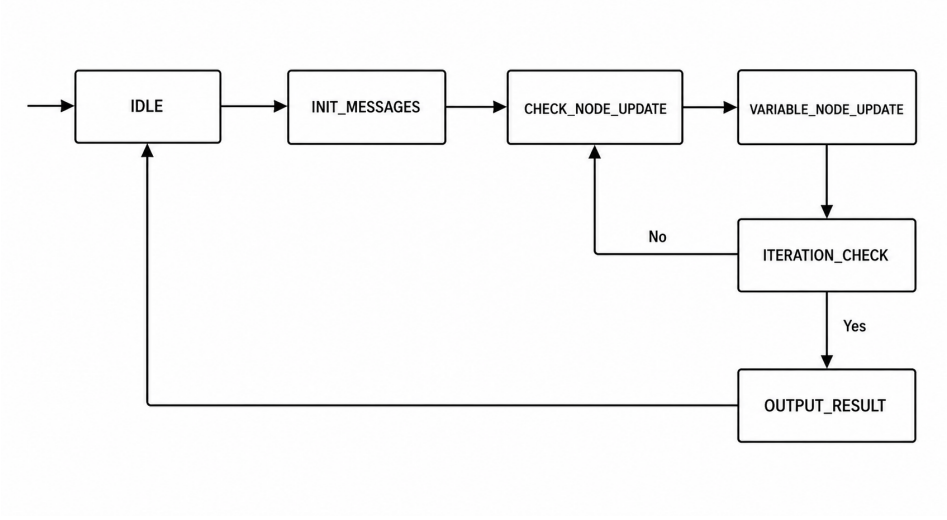

Figure F.3: Top-level FSM of the BP min-sum decoder kernel (check-node, variable-node, convergence check).

Bibliography

[1] P. W. Shor, "Scheme for reducing decoherence in quantum computer memory," *Physical Review A*, vol. 52, no. 4, R2493–R2496, 1995. DOI: 10.1103/PhysRevA.52.R2493.

[2] D. Gottesman, "Stabilizer codes and quantum error correction," arXiv:quant-ph/9705052, Ph.D. dissertation, California Institute of Technology, 1997.

[3] M. A. Nielsen and I. L. Chuang, "Quantum computation and quantum information: 10th anniversary edition," 2010.

[4] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill, "Topological quantum memory," *Journal of Mathematical Physics*, vol. 43, no. 9, pp. 4452–4505, 2002. DOI: 10.1063/1.1499754.

[5] B. M. Terhal, "Quantum error correction for quantum memories," *Reviews of Modern Physics*, vol. 87, no. 2, pp. 307–346, 2015. DOI: 10.1103/RevModPhys.87.307.

[6] A. Y. Kitaev, "Fault-tolerant quantum computation by anyons," *Annals of Physics*, vol. 303, no. 1, pp. 2–30, 2003. DOI: 10.1016/S0003-4916(02)00018-0.

[7] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, "Surface codes: Towards practical large-scale quantum computation," *Physical Review A*, vol. 86, no. 3, p. 032 324, 2012. DOI: 10.1103/PhysRevA.86.032324.

[8] O. Higgott, "Pymatching: A python package for decoding quantum codes with minimum-weight perfect matching," *ACM Transactions on Quantum Computing*, vol. 3, no. 3, pp. 1–16, 2022. DOI: 10.1145/3505637.

[9] N. Delfosse and N. H. Nickerson, "Almost-linear time decoding algorithm for topological codes," *Quantum*, vol. 5, p. 595, 2021. DOI: 10.22331/q-2021-12-02-595.

[10] J. Roffe, D. R. White, S. Burton, and E. Campbell, "Decoding across the quantum low-density parity-check code landscape," *Physical Review Research*, vol. 2, no. 4, p. 043 423, 2020. DOI: 10.1103/PhysRevResearch.2.043423.

[11] R. Acharya et al., "Quantum error correction below the surface code threshold," *Nature*, vol. 638, pp. 920–926, 2025. DOI: 10.1038/s41586-024-08449-y.

[12] L. Skoric, D. E. Browne, K. M. Barnes, N. I. Gillespie, and E. T. Campbell, "Parallel window decoding enables scalable fault tolerant quantum computation," *Nature Communications*, vol. 14, p. 7040, 2023. DOI: 10.1038/s41467-023-42482-1.

[13] B. Barber et al., "A real-time, scalable, fast and resource-efficient decoder for a quantum computer," *Nature Electronics*, vol. 8, pp. 84–91, 2025. DOI: 10.1038/s41928-024-01319-5.

[14] N. Liyanage, Y. Wu, A. Deters, and L. Zhong, "Scalable quantum error correction for surface codes using FPGA," in *Proc. IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2023, pp. 57–67. DOI: 10.1109/FCCM57271.2023.00016.

[15] N. Liyanage, Y. Wu, S. Tagare, and L. Zhong, "FPGA-based distributed union–find decoder for surface codes," *IEEE Transactions on Quantum Engineering*, vol. 5, pp. 1–16, 2024. DOI: 10.1109/TQE.2024.3401260.

[16] Y. Wu, N. Liyanage, and L. Zhong, "Micro Blossom: Accelerated minimum-weight perfect matching decoding for quantum error correction," in *Proc. 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025, pp. 782–798. DOI: 10.1145/3676641.3716005.

[17] C. Gidney, "Stim: A fast stabilizer circuit simulator," *Quantum*, vol. 5, p. 497, 2021. DOI: 10.22331/q-2021-07-06-497.

[18] J. Roffe, *LDPC: Python tools for low density parity check codes*, Software package, version 0.1.x, 2022. [Online]. Available: https://pypi.org/project/ldpc/.

[19] Xilinx, Inc., *ZCU104 evaluation board user guide (UG1267)*, 2018. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug1267-zcu104-eval-bd.

[20] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free Verilog synthesis suite," in *Proc. 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[21] Si2/Silvaco, *Nangate 45 nm open cell library*, 2011. [Online]. Available: https://si2.org/open-cell-library/.

[22] R. Nane et al., "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016. DOI: 10.1109/TCAD.2015.2513673.

[23] AMD/Xilinx, *Vitis high-level synthesis user guide (UG1399)*, Version 2025.2, 2025.

[24] C. Horsman, A. G. Fowler, S. Devitt, and R. Van Meter, "Surface code quantum computing by lattice surgery," *New Journal of Physics*, vol. 14, no. 12, p. 123 011, 2012. DOI: 10.1088/1367-2630/14/12/123011.

[25] J. Edmonds, "Paths, trees, and flowers," *Canadian Journal of Mathematics*, vol. 17, pp. 449–467, 1965. DOI: 10.4153/CJM-1965-045-4.

[26] V. Kolmogorov, "Blossom v: A new implementation of a minimum cost perfect matching algorithm," *Mathematical Programming Computation*, vol. 1, no. 1, pp. 43–67, 2009. DOI: 10.1007/s12532-009-0002-8.

[27] O. Higgott and C. Gidney, "Sparse Blossom: Correcting a million errors per core second with minimum-weight matching," *Quantum*, vol. 9, p. 1600, 2025. DOI: 10.22331/q-2025-01-20-1600.

[28] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM*, vol. 22, no. 2, pp. 215–225, 1975. DOI: 10.1145/321879.321884.

[29] S. Huang, M. Newman, and K. R. Brown, "Fault-tolerant weighted union-find decoding on the toric code," *Physical Review A*, vol. 102, no. 1, p. 012 419, 2020. DOI: 10.1103/PhysRevA.102.012419.

[30] R. G. Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, 1962. DOI: 10.1109/TIT.1962.1057683.

[31] M. P. C. Fossorier, M. Mihaljevic, and H. Imai, "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation," *IEEE Transactions on Communications*, vol. 47, no. 5, pp. 673–680, 1999. DOI: 10.1109/26.768759.

[32] P. Panteleev and G. Kalachev, "Degenerate quantum LDPC codes with good finite length performance," *Quantum*, vol. 5, p. 585, 2021. DOI: 10.22331/q-2021-11-22-585.

[33] M. P. C. Fossorier and S. Lin, "Soft-decision decoding of linear block codes based on ordered statistics," *IEEE Transactions on Information Theory*, vol. 41, no. 5, pp. 1379–1396, 1995. DOI: 10.1109/18.412683.

[34] AMD/Xilinx, *PYNQ: Python productivity for Zynq*, 2022. [Online]. Available: https://www.pynq.io/.

[35] P. Das et al., "AFS: Accurate, fast, and scalable error-decoding for fault-tolerant quantum computers," in *Proc. IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 259–273. DOI: 10.1109/HPCA53966.2022.00027.

[36] P. Das et al., "LILLIPUT: A lightweight low-latency lookup-table decoder for near-term quantum error correction," in *Proc. 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022, pp. 541–553. DOI: 10.1145/3503222.3507707.

[37] J. Valls, F. Garcia-Herrero, N. Raveendran, and B. Vasic, "Syndrome-based min-sum vs OSD-0 decoders: FPGA implementation and analysis for quantum LDPC codes," *IEEE Access*, vol. 9, pp. 138 734–138 743, 2021. DOI: 10.1109/ACCESS.2021.3118490.

[38] J. Chen and M. P. C. Fossorier, "Density evolution for two improved BP-based decoding algorithms of LDPC codes," *IEEE Communications Letters*, vol. 6, no. 5, pp. 208–210, 2002. DOI: 10.1109/4234.1001666.

[39] D. S. Wang, A. G. Fowler, and L. C. L. Hollenberg, "Surface code quantum computing with error rates over 1%," *Physical Review A*, vol. 83, no. 2, 020302(R), 2011. DOI: 10.1103/PhysRevA.83.020302.

[40] A. M. Stephens, "Fault-tolerant thresholds for quantum error correction with the surface code," *Physical Review A*, vol. 89, no. 2, p. 022 321, 2014. DOI: 10.1103/PhysRevA.89.022321.