HARDWARE ACCELERATED BASECALLING FOR MOBILE NANPORE DNA SEQUENCING

ABEL BEYENE

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTERS OF APPLIED SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO

APRIL 2023

Abstract

Nanopore Sequencing is characterized by its high-throughput and long reads which makes it amenable to sequencing genomes in real-time. The vast amounts of data generated, however, imposes serious requirements on any hardware tasked with performing the back-end data processing necessary to convert the electrical signal inputs to base-pair label outputs. The literature contains several examples of machine learning algorithms targeted at nanopore sequencing and demonstrated on server class CPUs and GPUs. Such hardware would be poorly suited to mobile environments however which have tight power and area constraints. To demonstrate the potential of custom accelerators for sequencing tasks, this work presents a complete RISC-V System-on-Chip (SoC) equipped with a bioinformatics hardware accelerator and taped out in Global Foundries 22nm process. It's able to achieve a 13X speed-up when basecalling over the ARM cortex A53 while only consuming 20 mW power.

Table of Contents

List of Tables

List of Figures

Abbreviations

BHT	Branch History Table
BTB	Branch Target Buffer
ONT	Oxford Nanopore Technologies
NGS	Next Generation Sequencing
PLIC	Platform Level Interrupt Controller
RAS	Return Address Stack
SoC	System-on-Chip

Chapter 1

Introduction

Genomic sequencing refers to the process of identifying the precise order of nucleotides within DNA molecules. The ability to decode and analyze genomic content plays a foundational role across a wide spectrum of disciplines, including molecular biology, medicine, and forensic science [1, 2, 3, 4, 5]. Although contemporary sequencing platforms offer high levels of accuracy and throughput, their substantial size, complexity, and cost limit their deployment in resource-constrained or remote environments. This limitation arises from the inherently multi-stage nature of DNA sequencing pipelines, which involve signal acquisition, basecalling, data conversion, and downstream computational analysis. Integrating this entire pipeline into a portable system remains a challenging but critical engineering goal.

Recent developments in compact sequencing devices have opened the possibility for real-time, in-field genome analysis. However, progress remains constrained by multiple technical bottlenecks. Engineering advances in chemistry, sensor design, and microelectronics are all required to realize the full potential of portable sequencing systems. In particular, enhancing the computational capacity of such systems represents a key area for innovation. Presently, most miniature sequencers offload the bulk of their computational tasks to external hosts and rely on conventional, server or cloud-based compute platforms for high-throughput processing.

This dependence limits and undermines their autonomy and physical portability.

Moreover, to-date, no custom computing designs for miniature sequencing devices have been seriously discussed or any potential candidates experimentally demonstrated in the open literature. Previous works have focused on alternative algorithms [6, 7], simulation-based methodologies [8] or the use of commodity hardware [9, 10, 11, 12] Thus, to address this gap, the research presented in this work introduces a fully integrated system-on-chip (SoC) bioinformatics processor implemented in Global Foundries 22FDX fully-depleted silicon-on-insulator (FD-SOI) technology. The proposed SoC architecture features heterogeneous processing cores, domain-specific hardware accelerators tailored for DNA sequencing workloads, and high-bandwidth interfaces for off-chip memory access. The design is optimized to support both real-time data input and computationally intensive tasks within the power and form-factor constraints of mobile sequencing platforms.

The SoC presented here is intended to demonstrate the feasibility and advantages of embedding dedicated, power-efficient computation within next-generation mobile sequencers. The SoC integrates purpose-built accelerator units for core bioinformatics tasks while also providing general-purpose processing blocks capable of executing high-level application code. This hybrid architecture enables both flexibility and performance scaling and allows the device to support a range of sequencing tasks without sacrificing programmability. Importantly, the entire system is designed to operate within a strict energy budget, making it suitable for battery-powered field deployment.

In addition to its targeted impact on mobile DNA sequencing, the design principles underpinning this SoC—namely, domain-specific acceleration, heterogeneous processing integration, and energy-conscious computation—can generalize to other application domains. The ability to offload intensive workloads onto custom accelerators without relinquishing software programmability is increasingly relevant across edge computing, portable diagnostics, and embedded AI workloads. As such, this architecture serves not only as a sequencing-

specific proof of concept but also as a blueprint for how emerging embedded systems can reconcile high computational throughput with constrained power envelopes.

In sum, beyond its contributions to the field of bioinformatics, the proposed embedded SoC holds significant potential for applications in the broader consumer electronics landscape. This work aims to illustrate how thoughtful co-design of algorithms, hardware, and power management strategies can unlock new capabilities for mobile computation and pave the way for a class of embedded systems that are both computationally capable and practically deployable in the field.

Chapter 2

Background and Algorithm

2.1 DNA Sequencing Overview

Current sequencing technologies represent a diverse and advanced set of methods and chemistries that have evolved considerably since the earliest approaches were introduced. Among these foundational methods, the Sanger technique was the most influential early contribution. Published in 1977 alongside the work of Gilbert and Maxam [13], Sanger's method laid the groundwork for modern sequencing by introducing the core principle of sequencing by synthesis (SBS). Later refinements built on this concept and extended its utility across a range of high-throughput applications.

With Sanger's approach, the DNA sequence is determined by synthesizing complementary strands using an enzyme (DNA polymerase) and incorporating fluorescently labeled chain-terminating dideoxynucleotides (ddNTPs). The process begins with denaturation, where the double-stranded DNA template is heated to separate it into single strands. Next, during the annealing step, a short single-stranded primer binds to a specific region on the template strand. In the extension step, DNA polymerase synthesizes a new strand by adding deoxynucleotides (dNTPs), but occasionally incorporates a ddNTP which lacks a 3-OH group and therefore

terminates chain elongation at that point. Because ddNTPs are fluorescently labeled and randomly incorporated at each of the four bases (A, T, C, G), this results in a mixture of DNA fragments of various lengths, each ending in a labeled base. Finally, the fragments are separated by capillary electrophoresis, and a laser detector reads the fluorescence to determine the terminal base at each position, thus reconstructing the DNA sequence from shortest to longest fragment. This technique has contributed to almost all current knowledge of gene and genome sequences.

However, despite its status as the gold standard for sequencing, the Sanger technique featured several limitations which drove the development of next-generation sequencing (NGS) technologies. Particularly, it suffered from low throughput (one DNA fragment per reaction), high cost (requires many individual reagents and primers) and a labour intensive workflow (many manual steps including primer design, thermal cycling, electrophoresis). It also performs poorly for detecting low frequency variants. These shortcomings motivated the development of Next-Generation Sequencing (NGS) techniques which, while still based on sequencing by synthesis principles, focused more heavily on parallelization and throughput.

In NGS-based technologies, the original DNA strand is broken into smaller pieces and special adapter sequences are attached to both ends. These fragments are then placed on a solid surface like a flow cell and copied many times through a process called bridge amplification. This creates millions of tiny clusters where each cluster contains many identical copies of one DNA piece. In other words, NGS-based platforms allow for the sequencing of millions to billions of short DNA fragments (typically 50-300 base pairs) across an entire genome or a specific region in parallel. During the main sequencing step, a DNA polymerase enzyme adds fluorescently labeled building blocks (dNTPs) one at a time. Each added base, under stimulation by a laser, emits a light signal that's indicative of which base was added. A chemical blocker which is present on each added base is then removed so the next base can be added. Once the last fluorescent marker is read, images are analyzed to determine the

DNA sequence of the original strand. The results are then compared to a known genome or assembled from scratch to find genetic differences (variants).

The first commercial demonstration of an NGS-based platform was by 454 Life Sciences in 2005 [14]. They introduced a variant of NGS referred to as pyrosequencing - the detection of pyrophosphate release during DNA synthesis. Additional methods soon emerged including reversible dye terminator sequencing (used in platforms like Illumina) and proton detection (used by Ion Torrent). This marked the beginning of the next-generation sequencing (NGS) era.

Alas, while NGS technologies represented a significant breakthrough for researchers, they too possessed several limitations that eventually led to the development of third generation platforms. A key shortcoming was the short read length, typically limited to a few hundred base pairs. That made it difficult to resolve repetitive regions and detect large structural variants. NGS also required PCR amplification and thus introduced bias and limiting the ability to analyze epigenetic modifications. The complex and time-consuming sample preparation process further slowed turnaround times. Additionally, the short reads produced by NGS imposed high computational demands during follow up data analysis. These challenges motivated the shift toward a third generation sequencing technologies based on long reads, real-time analysis, and direct sequencing of native DNA or RNA without amplification.

2.2 Third Generation Sequencing

By 2010, third-generation sequencing devices began to emerge, the most prominent of which was from Oxford Nanopore Technologies (ONT). They debuted their nanopore sequencing platform which has since become a dominant approach due to its portability and scalability. Nanopore sequencing involves threading strands of DNA through nanometer-sized holes (referred to as nanopores) embedded in membranes suspended within an electrically biased

conducting fluid (see Figure 2.1). As nucleotides pass through the pores, they create characteristic disruptions in electrical current. Each disruption, specific to a nucleotide type (A, C, G, or T), generates a signal that is decoded using advanced bioinformatics algorithms in a process called basecalling. The result is a high-throughput, real-time output of DNA sequences.

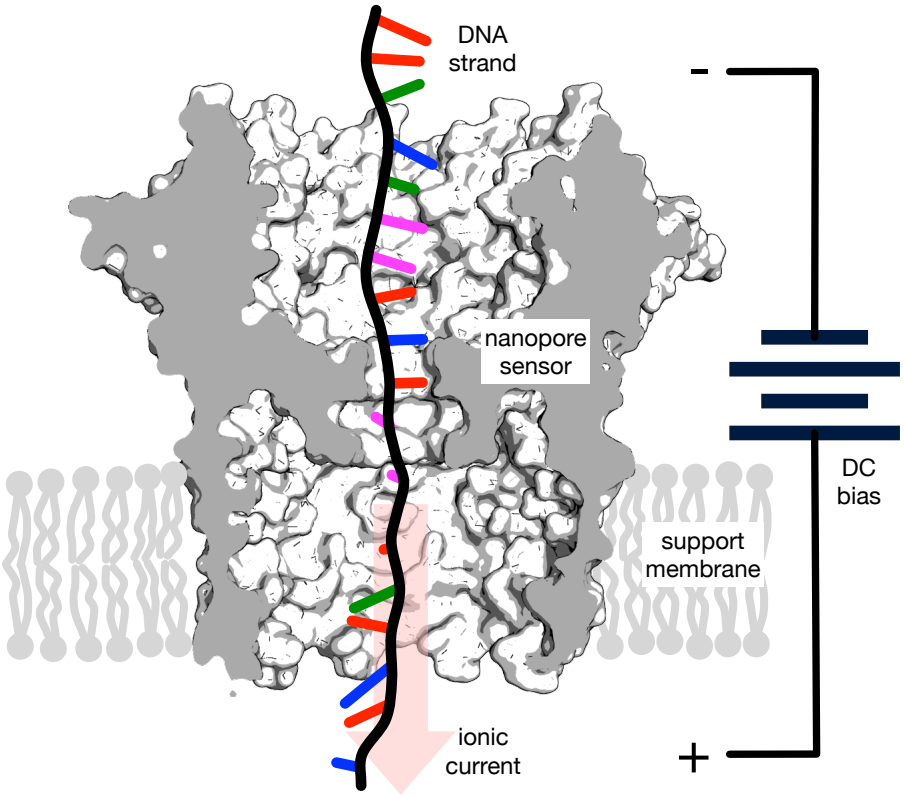

Figure 2.1: DNA Strand passing through Nanopore Sensor

Table 2.2 provides a brief comparison of three generations of sequencing technologies. Sanger sequencing offers high accuracy and read lengths of around 500 to 1000 base pairs but is slow and costly for large-scale projects, thus making it best suited for small-scale or targeted sequencing tasks. Next-generation sequencing (NGS) technologies provide much higher throughput at a lower cost per base but produce shorter reads typically between 50 to 300 base pairs. That places a heavier burden on downstream steps in the computational

Technology	Generation	Sample Size	Read Length	Accuracy
Sanger Sequencing	1st	Large	500-1000 [15]	>99.99
Illumina	2nd	Small	50-300 [16]	99.5 [17]
Oxford Nanopore	3rd	Small	5000-50000 [18]	85–95 [19]
PacBio	3rd	Small	10000-30000 [17]	85-92 [20]

Table 2.1: Summary of Sequencing Methods

pipeline. NGS is ideal for whole-genome sequencing population studies and transcriptomics. Nanopore sequencing enables very long reads often exceeding tens of thousands of base pairs and supports real-time analysis with relatively low setup costs though its raw read accuracy has historically been lower than that of Sanger or NGS platforms; it is especially useful for de novo assembly, structural variant analysis and field-based sequencing where portability is important.

2.3 Challenges for Mobile Sequencing

While sequencing technologies have advanced tremendously, challenges remain. A critical factor is the sample volume required by different methods influences cost and accessibility. However, sanger sequencing has evoloved enough in methodology, automation, and commercialization that it remains the most appropriate sequencing method for many current applications [21]. Insufficient standardization of NGS still limits its use in clinical medicine. Furthermore, despite the accuracy of current technologies, achieving higher sequencing fidelity often requires sequencing individual genomes multiple times and that drives up the cost and complexity of projects.

The ability of nanopores to produce data on singular DNA molecules at a time allows them to dramatically improve upon the read length and accuracy of pre-existing sequencing methods. Their miniscule size, however, is not sufficiently small to prevent more than one

base from passing through at a time. In reality, a group of k bases (known as a $k-mer$) will occupy a pore at any given moment where k depends on the pore's physical dimensions. This implies that nanopores exists in one of 4^k possible states (where 4 represents one of four possible bases A, C, G or T) that each sample of the current signal must map too. The larger the value for k, the more complex the backend algorithms must be to solve the mapping problem. Moreover, as DNA is fragmented into individual strands before translocation, the muliple reads produced by the sensor (which the basecaller converts to multiple text strings) must eventually be assembled and aligned to a reference. Together with randomness in the strand translocation rates and noise inherent in the sensor electronics, processing raw nanopore data poses serious challenges to mobile platforms limited in computing resources.

The dominant approach to the basecalling problem involves hidden markov model (HMM) or neural network based machine learning [22][23][24]. Such solutions are software-based however and are typically deployed on powerful servers or cloud services. To reduce the excess runtimes associated with model training and prediction, GPUs are often exploited for their highly parallel architectures [25]. The literature has shown many examples of GPU-accelerated neural network basecalling yielding fast (< 1 min), highly accurate ($> 90\%$) results.

What the literature has not shown, however, is a solution suited to mobile platforms. The high-throughput requirement for real-time sequencing would constrain the choice of GPU to prohibitively power-intensive models [26]. And the potential of quantized basecalling networks has only been lightly explored [27]. Thus, this work argues that the next step towards realizing mobile, real-time sequencing requires a custom ASIC solution that overcomes the limitations of traditional embedded systems.

2.4 Design Requirements

Embedded computing in miniature DNA sequencers must strike a balance between high data throughput, low power consumption, and strict form factor constraints. These requirements are closely tied to the sequencing method, data generation characteristics, SoC architecture, and end-user application. For this study, the MinION device from Oxford Nanopore Technologies (ONT) is used as the reference platform due to its commercial maturity and relevance in portable sequencing.

MinION uses 512 nanopore channels operating in parallel, each sampled at 4 kHz with 10-bit resolution [9], resulting in a raw data stream of approximately 20 Mbits/s from the mixed-signal front end. This data must undergo preprocessing steps such as baseline correction, event detection, and possibly initial basecalling or data compression, all in real time. These operations place significant demands on the embedded processor, both in terms of compute capability and memory bandwidth.

The total power consumption of MinION averages around 1 W, with an estimated 200 mW (20%) allocated to embedded digital processing. Within this budget, the system must accommodate CPU cycles, memory accesses, and potentially hardware accelerators (e.g. DSP blocks or NPUs) while still maintaining thermal and energy efficiency under mobile operating conditions such as USB-powered laptops or field-deployable devices.

Latency is another important factor. Real-time sequencing workflows require prompt data handling to avoid buffer overflows or sampling interruptions. Additionally, local processing helps reduce the need for continuous high-bandwidth host-device communication, which is especially critical when operating in bandwidth-limited or disconnected environments.

Beyond raw performance, area constraints impose further design pressure. The embedded system must fit within a highly compact enclosure alongside analog front-end components, nanopore sensor arrays, and thermal control systems. Together, the data rate, processing

Requirements	
Performance	20 Mbits/s
Power	200 mW
Area	4 mm x 3 mm

Table 2.2: ASIC Requirements

budget, power constraints, and form factor converge to define the key design requirements, summarized in Table 3.1, for developing an SoC architecture tailored to real-time nanopore sequencing.

2.5 Algorithm

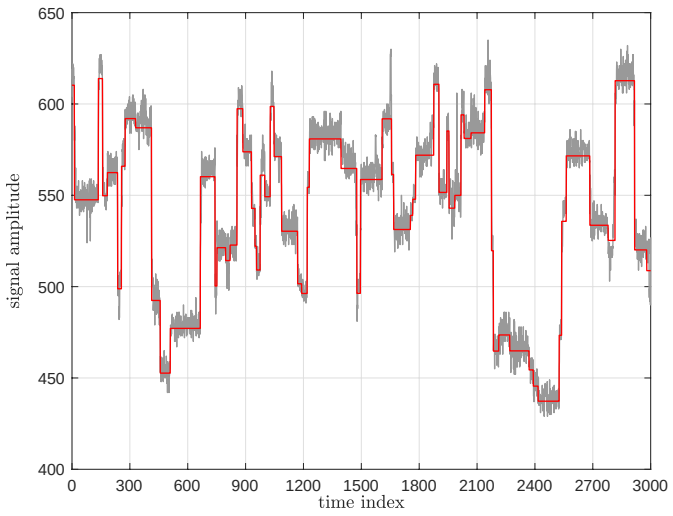

Figure 2.2: Electrical Characteristics of an event

In practical terms, basecalling can be defined as the problem of mapping a sequence of measurements to a sequence of symbols. The sequence can be of arbitrary length and will be exhibit certain characteristics such as an expected amplitude and resolution. Due to their grounding in a rigorous probabalistic framework, ability to quantify uncertainty and inherent

handling of noise and variability, Markov Chains are a natural choice for modeling biological data. This work takes a Markov Chain based approach to solving the basecalling problem.

2.5.1 Markov Chains

Markov chains are mathematical models used to describe systems that move between different states over time where the next state depends only on the current state as opposed to the full history. This property is called the Markov property and makes Markov chains particularly tractable for analysis and widely applicable across various disciplines including physics, biology, economics, computer science, and engineering. Markov chains began as a mathematical curiosity in the early 1900's but became central to many bioinformatics tools due to their power in modeling biological sequences and evolutionary processes.

Formally, a discrete-time Markov chain is a sequence of random variables $\{X_n\}$ defined on a discrete-time index set $T = \{0, 1, 2, ...\}$ such that each random variable can take a value from a countable set of possible states called the state space, $S = \{s_1, s_2, ..., s_n\}$. Further, the sequence exhibits what is called the Markov property, expressed as:

$$P(X_{n+1} = j | X_0 = i_0, X_1 = i_1, \ldots, X_n = i_n) = P(X_{n+1} = j | X_n = i_n)$$

The probabilities of transitioning from one state to another are defined by the transition probabilities. For a time-homogeneous Markov chain, these probabilities are constant over time. Let p_{ij} denote the probability of moving from state i to state j in one step:

$$p_{ij} = P(X_{n+1} = j | X_n = i)$$

These probabilities can be organized into a transition matrix P, where the entry in the i-th row and j-th column is p_{ij}.

$$\mathbf{P} = \begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1k} \\ p_{21} & p_{22} & \cdots & p_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ p_{k1} & p_{k2} & \cdots & p_{kk} \end{pmatrix}$$

Each row in the matrix must sum to 1, as it represents all possible transitions from a given state. That is, for each row i:

$$\sum_{j=1}^{N} P_{ij} = 1$$

2.5.2 Hidden Markov Chains

Hidden Markov Models (HMMs) are a powerful extension of Markov chains where the system has hidden (unobserved) states that produce observable outputs. In bioinformatics, HMMs are widely used because biological sequences often show patterns that depend on underlying but unobservable biological states. An HMM is formally defined by the following components:

- N: The number of hidden states in the model. We denote the set of states as $S = \{s_1, s_2, \ldots, s_N\}$.
- L: The number of possible observation symbols per state. We denote the set of observation symbols as $V = \{v_1, v_2, \ldots, v_L\}$.
- A: The state transition probability distribution, an $N \times N$ matrix.
- B: The observation (or emission) probability distribution, an $N \times L$ matrix.
- π: The initial state probability distribution, a $1 \times N$ vector.

Let $\{\psi_0, \psi_1, \psi_2, \ldots, \psi_M\}$ represent a series of **state vectors** where $\psi_m = (\psi_m^0, \psi_m^1, ..., \psi_m^{N-1})$ and $\psi_m^k \in S$. Further, let $\mathbf{x} = \{x_1, x_2, \ldots, x_M\}$ be a sequence of observations of length M, where $x_k \in V$. The probability of transitioning from state s_i to state s_j is given by

$$A = \{a_{ij}\}$$

where

$$a_{ij} = P(\psi_{t+1} = s_j \mid \psi_t = s_i), \quad 1 \leq i, j \leq N$$

. And for any state s_i, the sum of probabilities of transitioning to all possible next states must be 1:

$$\sum_{j=1}^{N} a_{ij} = 1, \quad 1 \leq i \leq N$$

As for emission probabilities, the probability of observing symbol v_k given that the system is in state s_j is:

$$B = \{b_j(k)\}$$

where

$$b_j(k) = P(x_t = v_k \mid \psi_t = s_j), \quad 1 \leq j \leq N, 1 \leq k \leq M$$

And for any state s_j, the sum of probabilities of emitting all possible observation symbols must be 1:

$$\sum_{k=1}^{M} b_j(k) = 1, \quad 1 \leq j \leq N$$

Hidden Markov Models (HMMs) are well-suited for nanopore sequencing because they provide a structured way to deal with two major challenges in nanopore sequencing: indirect

observation and noisy and ambiguous signals.

Nanopore devices don't read DNA letters directly. Instead, they detect changes in electrical current as a segment of DNA (usually a few bases at once) passes through a pore. These current patterns are the only observable data but they do not tell you the DNA sequence directly. The true sequence is hidden and HMMs are designed to uncover the hidden information by working backward from the observations.

Further, the current measured at each moment is influenced by multiple adjacent bases (not just one) and can vary due to small shifts or irregular movement through the pore. Multiple different k-mers might produce similar signals. An HMM accounts for this by assigning probabilities to how likely a particular current signal is for each possible k-mer.

The problem can be described as being given an HMM and a sequence of observations and attempting to determine the single most probable sequence of hidden states that could have generated those observations. A brute-force approach of trying every possible hidden state sequence would be computationally intractable for even moderately long observation sequences.

2.5.3 Trellis Construction

Thus, the basecalling algorithm chosen was motivated by the choice of model for the nanopore from which the raw data (i.e current signal samples) would be generated. As explored in [28], the nanopore sensor was characterized by an HMM with M observable events x_i drawn from one of 4^k probability distributions ϵ_j. An event here represents an aggregate of the raw current signal samples over the time period of a single k-mer. The problem was to find the sequence of states π_i which maximized the likelihood of the event observations. Mathematically, given the model $\lambda = (A, B, \pi)$ and an observation sequence $\mathbf{x} = \{x_1, x_2, \ldots, x_M\}$, find the most

```
1  for m = 0 to M-1 {  // *(1) event loop start
2    for n = 0 to N-1 { // *(2) state loop start
3      *transidx = GatherTrans(n)
4      for t = 0 to T-1 { // *(3) trans loop start
5        idx = transidx[t]
6        trans[t] = α_{m-1}[idx] + tprob[t]
7      } // end trans loop (3)*
8      minidxT = FindMin(trans, T)
9      β[n][m-1] = minidxT
10     α'_m[n] = Post(x[m],trans[minidxT],μ[n],σ[n])
11   } // end state loop (2)*
12   minidxN = FindMin(α'_m,N)
13   minprob = prob[minidxN]
14   for n = 0 to N-1 { // *(4) norm loop start
15     α_m[n] = α'_m[n]-minprob
16   } // end norm loop (4)*
17 } // end event loop (1)*
```

Figure 2.3: Viterbi trellis construction algorithm.

likely sequence of hidden states $\Psi = \{\psi_1, \psi_2, \ldots, \psi_M\}$ that produced $\mathbf{x}$.

$$\{\psi_i \mid 1 \leq i \leq M\} = \max_{\nu_j \in \{0,..,4^k-1\}} P(\nu_1, ..., \nu_N, x_1, ..., x_n)$$

This is the well-known HMM decoding problem for which a dynamic program technique known as the *Viterbi algorithm* is typically used. This algorithm is shown in Figure 2.3.

The Viterbi algorithm uses dynamic programming to efficiently solve the problem by building up the solution iteratively. The algorithm turns an event series into a $N \times M$ matrix (i.e., the trellis) of integer *trellis pointers* β`[n][m]`. As explained below, the pointer values denote probable steps through the states composing the trellis and can collectively be used to identify the optimal (i.e most likely) path through the trellis. The N term denotes the number of possible k-mers that can be associated with any one event. In general, since DNA consists of four monomers (`A`, `C`, `G`, `T`), the number of possible k-mers for a sequencing with sensing resolution of k is $N{=}4^k$. As shown in Fig. 2.3, for-loop iterations over M and N comprise the two outermost loops of the trellis construction algorithm as it builds the

trellis one pointer calculation at-a-time. A follow-up traceback algorithm (discussed below in §Traceback) traverses the $N \times M$ trellis via the pointers β`[n][m]` to produce the final sequence of monomers that constitute a DNA read.

As noted above, the pointers collectively define a set of optimal paths through the trellis spanning contiguous routes from `m` indexes $M-1$ to 0. More formally, these paths traverse the trellis's k-mer *state vectors* $\Psi_0, \ldots, \Psi_{M-1}$ where $\Psi_m = (\psi_m^0, \ldots, \psi_m^{N-1})$ and where each ψ_m^n represents a unique k-mer *state*[1]. For example, if β`[n][m-1]` $= l$ is computed, this means that the state ψ_m^n is part of a path through the trellis who's preceding state is most likely to be ψ_{m-1}^l. The calculations behind these estimates are discussed below. The path corresponding to the minimum cost is ultimately identified by a *traceback algorithm* (discussed below) that outputs the final read sequence estimate from the original input sequence.

Before further detailing the Fig. 2.3 code, its worth outlining the manner in which it would be expected to operate. In a typical processing use-case, many event streams will be presented to a detector in parallel. For example, in existing portable sequencing devices, over 500 parallel channels are capable of simultaneously producing event streams. Although the length of these streams may vary depending on the DNA sample that produced them, for easier management and without loss of generality, they can be partitioned into equal chunks of length M to present a consistent detection processing load. Since the origin and location of each such chunk within a given stream is known, splicing these back into complete reads is straightforward[2]. More importantly, to avoid the need for 100s of parallel detector instantiations, it is preferred that the code of Fig. 2.3 can process channels in a rapid time-multiplexed manner. Specifically, if there are C channels and each is able to produce data at a rate of R events per second the algorithm in Fig. 2.3 has to complete its outer loop

[1]For example, for k=3, if the states denote 3-mers in terms of lexicographical order, then ψ_m^0 represents the 3-mer `AAA` , ψ_m^1 is `AAC` , and ψ_m^{63} represents `TTT` for all m. For brevity, we also refer to states at event index `m` by their index term `n`.

[2]In contrast, recombining the different streams to form a contiguous genome requires a separate set of algorithms, not considered in this work.

in a time of $M/(C{\cdot}R)$ to keep-up, it is this performance pressure that motivates the SoC accelerators discussed in this paper.

Returning to the code, the algorithm's job is to form probabilistic relationships between events and k-mer states. That's done through the construction of a path metric through all the state vectors Ψ comprising the trellis structure:

$$p(\boldsymbol{x}|\Psi)P(\Psi) = \prod_{m=1}^{M-1} [p(x\texttt{[m]}|\Psi_m)P(\Psi_m|\Psi_{m-1})] \tag{2.1}$$

where $P(\Psi_m|\Psi_{m-1})$ denotes a probability of *transition* between state vectors at adjacent event indexes and where $p(x\texttt{[m]}|\Psi_m)$, denotes the *observation* likelihood of a k-mer state vector being associated with a given measured event $x[m]$. For improved hardware efficiency, the negative logarithm of these terms is computed in the form of the (log) posteriors

$$\{\alpha_m\texttt{[n]}\}_0^{N-1} = -\log [p(x\texttt{[m]}|\Psi_m)P(\Psi_m|\Psi_{m-1})] \tag{2.2}$$

Thus, for each event $x\texttt{[m]}$, the trellis computation updates N terms $\alpha_m\texttt{[0]}$ to $\alpha_m\texttt{[N-1]}$. For each newly computed posterior $\alpha_m\texttt{[n]}$, a corresponding trellis pointer $\beta\texttt{[n][m-1]}$ is also calculated. To help clarify, a picture associating these variables to the trellis structure is offered in Fig. 2.4. The posterior and pointer calculations in the trellis construction algorithm outlined in Fig. 2.3 consists of four nested for-loops. The outermost loop (1) - referred to as the *event loop* (lines 1–17) - sequentially processes incoming events $x\texttt{[m]}$. This initiates the calculation of each new set of N states within a state vector Ψ_m. The state calculations are handled within the (2) *state loop* (lines 2-11). Within the state loop, the (3) *trans loop* (lines 4-7) computes transition probabilities between states. After the state loop, a normalization (4) *norm loop* (lines 12-16) is run to re-adjust the computed posterior terms to prevent overflow.

The state loop consist of three main phases. The first is on line 3 of Fig. 2.3. There the

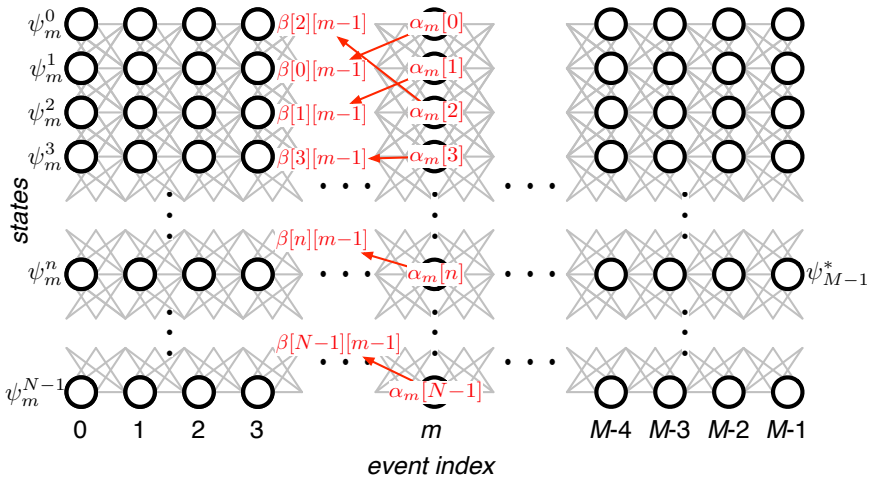

Figure 2.4: A trellis representation and its construction via state progression.

`GatherTrans` function computes and gathers the addresses for all the states at the previous index `m-1` that may transition into the state `n` at the present index `m` (i.e., the current state ψ_m^n). The starting address of the set of states computed by `GatherTrans` is indicated by pointer `transidx`. In theory, all N preceding states at `m-1` may transition into any one current state at `m`. That would require computing N^2 total transitions for each iteration of the event loop. But sequence constraints allow only a unique subset of $T<N$ states to be considered as transition states into ψ_m^n. Hence, only a total of $N{\cdot}T$ transitions must be identified for each event loop iteration. Identifying the particular transitions for each state is the job of `GatherTrans`.

In particular, for any ψ_m^n, only $T = 21$ preceding states need to be accounted for. These 21 states reflect so-called *stay* (1 in total), *step* (4 in total), and *skip* (16 in total) transition types. The stay transition types reflect the possibility that consecutive events at `m` and `m-1` are just separate measurements of exactly the same k-mer. Hence, they simply account for a transition between identical states (i.e., ψ_{m-1}^n to ψ_m^n). The step types reflect the possibility that consecutive events reflect 1-mer shifts, for example, like the transition from ψ_{m-1}^6 to ψ_m^{27} which reflects `ACG` "stepping" to `CGT`. And skip types reflect the possibility that only a coarse

2-mer jump has been captured by consecutive events (e.g., `ACG` "skipping" to `GAC`, a jump from ψ_{m-1}^{6} to ψ_{m}^{33}). Generally, the relationship between the state labels for steps and skips can be as expressed as

$$x = l \cdot 4^{k-1} + \lfloor y/4 \rfloor \text{ for } \psi_{m-1}^{x} \xrightarrow{\text{step}} \psi_{m}^{y} \tag{2.3}$$

$$x = L \cdot 4^{k-2} + \lfloor z/4^{2} \rfloor \text{ for } \psi_{m-1}^{x} \xrightarrow{\text{skip}} \psi_{m}^{z} \tag{2.4}$$

where $l \in \{0, 1, 2, 3\}$ and $L \in \{0, 1, \ldots, 4^2 - 1\}$. These relationships play an important role in § 3.2.3 when considering hardware design for traceback acceleration.

The second major phase of the state loop concerns execution of its trans loop (lines 4-7) block. This part computes the (log) probability of the transition term, `trans[:]`$=P(\Psi_m|\Psi_{m-1})$ in equation (2.2). Computationally, it is an application of the probability chain rule expressed as an addition (line 6) between the (log) posterior α_{m-1} and the (log) probability of transition between states, `tprob`. This latter value is determined by some preliminary model training procedure.

The third major phase (lines 8-10) of the state loop completes the main trellis construction for event `m` by updating posterior and the corresponding set of trellis pointers for all states. Specifically, via the `FindMin` function (line 8), this part of the state loop identifies `minidxT`, the most likely of T preceding states to transition into the current state `n`. This allows the corresponding trellis pointer β`[n][m-1]` to be assigned (line 9). The new log posterior of state `n`, α'_m`[n]`, is computed with the `Post` function which effectively completes the calculation summarized in equation (2.2) as

$$\alpha'_m\texttt{[n]} = \texttt{trans[minidxT]} - \sigma\texttt{[n]} + (x\texttt{[m]} - \mu\texttt{[n]})^2 \tag{2.5}$$

where μ and σ are another set of model parameters that may be simultaneously determined

```
1 state_path = []
2 prev_state = minidxN
3 Cat(state_path,minidxN)
4 for m = M-2 to 0 {  // *(1) traceback loop start
5   prev_state = β[prev_state][m]
6   Cat(state_path,prev_state)
7 } // end traceback loop (1)*
```

Figure 2.5: The traceback algorithm.

alongside `tprob` as noted above. Finally, as noted above, to prevent overflow, a normalization loop (lines 14-15) is executed to produce a scaled set of posteriors α_m`[n]` to process in the following iteration of `m`.

2.5.4 Traceback

The traceback procedure initiates by instantiating the optimal path sequence array, denoted as *state_path*. Drawing upon the outputs of the trellis construction algorithm (specifically, Fig. 1, line 12, which refers to the Viterbi algorithm's final step), the traceback module retrieves the most probable terminal state, identified as minidxN or $\psi_M - 1$. This state is then prepended to the *state_path* array via a concatenation operation.

Subsequently, the traceback loop (lines 4-7) commences a pointer-chasing operation through successive dereferencing of the β array (representing the trellis pointers). This iterative process effectively reconstructs the most likely preceding sequence of M-1 states, denoted as *prev_state* or ψ_m. As each state in this optimal sequence is identified, it is progressively concatenated to the *state_path* array until the entire optimal hidden state sequence is fully assembled in reverse chronological order.

Chapter 3

Hardware Design

3.1 Digital Processor

In place of a GPU, this project integrated a custom basecalling hardware accelerator into a RISC-V system-on-chip (SoC). The core was based on the open-source Chipyard design *Rocket* and implemented the RV64IMAFD RISC-V instruction set. It was equipped with an MMU supporting page-based virtual memory, a non-blocking data cache, and a non-blocking instruction cache complete with branch prediction, a branch target buffer (BTB), a branch history table (BHT), and a return address stack (RAS). It also contained a custom Berkeley floating-point unit and supported the RISC-V machine, supervisor, and user privilege levels. Each RocketTile represents a CPU core instance. Among the many functional blocks within a RocketTile are the core's exclusive L1 Cache (I&D separately) and a page table walker (PTW). The SystemBus is a Network on Chip (NoC) to connect cores with other components. DRAM connects to the CPU through the AXIMem interface, because of the different protocol between the CPU and DRAM, the TileLink (TL) to AXI converter is inserted. For MMIO, the SystemBus connects to two inner buses, one is the ControlBus, connecting some standard peripherals such as BootROM that includes the first code when processor is powered on or is

Table 3.1: System-on-Chip Processor and Accelerator Parameters

Design Parameter	Value	Physical Parameter	Value
ISA	RISCV64G	**Technology**	GF 22-nm FDSOI
Data width	64 B	**Proc. area**	0.61 mm^2
I/D-Cache (4-way)	16 KiB	**Accel. area**	0.67 mm^2
Line size	64 B	**Core supply**	0.49-0.80 V
TLB entry	8	**Clock freq.**	200 MHz
BHT entry (2-b)	4096	**Gates**	400K
BTB entry	62		
RAS entry	2		
Accel. mem.	36 KiB		

reset, Platform Level Interrupt Controller (PLIC), Core Local Interrupts (CLINT) and debug unit linking to JTAG. Another is PeripheryBus, a bus to connect additional peripherals like NIC or Block Device (HDD). Last, the FrontBus is the used to connect AXI4 DMA devices. The RoCC interface empowers the processor's communication with RoCC accelerators through a set of custom protocols and custom non-standard ISA instructions reserved in the RISC-V ISA encoding space [**chipyard20**]. It also provides interfaces for custom accelerators to access every SoC component, such as L1 data cache, floating point unit (FPU), page table walker (PTW) and main memory.

3.2 Accelerator Design

A system-level outline of the accelerator implemented in this work is given in Fig. 3.2. As previously mentioned, it is tasked tasked with handling the trellis construction algorithm of Fig. 2.3 as well as the traceback code of Fig. 2.5. As shown in the chip measurement discussions of Chapter 5, this addition brings substantial benefits to the SoC's performance.

The operation of the hardware starts with the processor receiving a series of accelerator

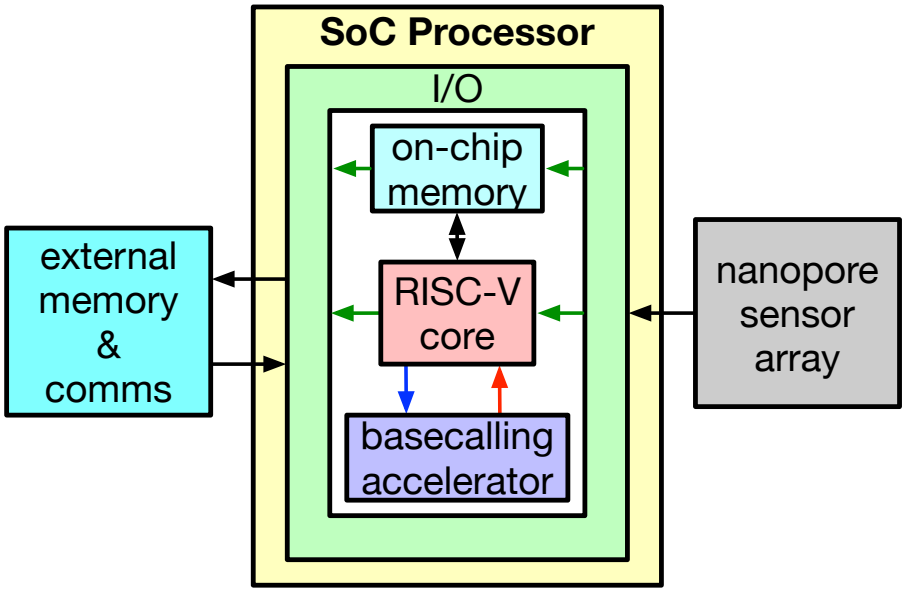

Figure 3.1: Application Processor with RISC-V core and I/O

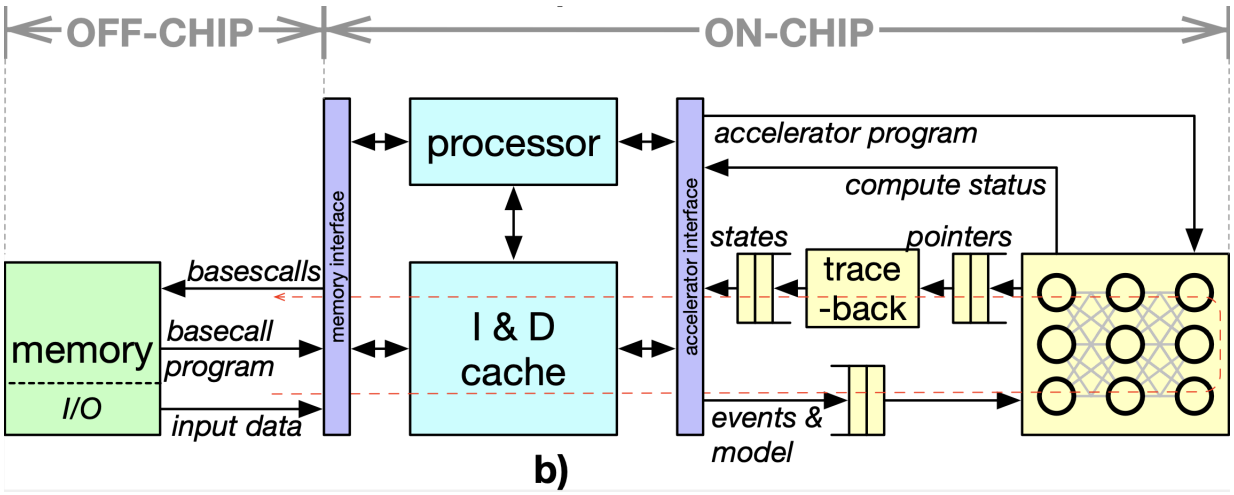

Figure 3.2: Accelerator merging trellis and traceback functions

program commands from the core. The commands set the number, M, of events $\boldsymbol{x}$ to process and the location of the events in memory. The memory locations of model parameters `tprob`, `mu` and `sigma` are also conveyed as is the memory location for data returned by the accelerator. Once the program is loaded the accelerator starts operation.

This involves feeding in events from memory through a 4-KiB SRAM *event buffer* and storing trellis pointers in a 32-KiB SRAM *pointer buffer*. After a set of trellis pointers is computed by the trellis constructor hardware, they are then sent to a traceback hardware block that computes the corresponding state sequence. This sequence is then sent back to memory along with a compute status signal to the RISC-V core. This status signal enables the core software to initiate another set of events for processing by the accelerator. More details on the manner in which signals are exchanged between the core and the accelerator are provided next.

3.2.1 Accelerator Software Interface

The accelerators used in this work both connect to the core and its D cache using the Rocket Custom Coprocessor (RoCC) interface - another open-source Chipyard component. RoCC facilitates 64-bit communication between accelerators and the processor using sets of request/response channels. Four RoCC ports are used in this work, *cmd* and *resp* for control and status signal exchange between the RISC-V core and the accelerators and *mem.req* and *mem.resp* for data load/store between the D cache and the accelerators. The RoCC ports support 64-bit data paths to and from memory as well as 40-bit memory address ports.

Software communicates with the accelerator through extended 32-bit assembly instructions that map directly to instructions in the RISC-V ISA designed to forward commands to the RoCC logic and custom accelerator hardware. As a result, commands such as

```
asm volatile ("custom0 %[rd],%[rs1],%[rs2],0" : \
```

```
[rd]"=r"(z) : [rs1]"r"(x), [rs2]"r"(y));
```

can be included in developer C code. Thus, contents in the RISC-V core's register file locations, `x` and `y` may be sent to the accelerator (with results written back from the accelerator to a core register `z`). Wrapping such commands in C macros allows their low-level details to be hidden from software developers.

The accelerator in this work can be operated using a series of 6 macro commands (the aforementioned accelerator program) that form the following programming sequence: 1) accelerator reset; 2) set number of events, M, to process; 3-5) starting memory addresses for model parameters; 6) starting memory address for events x`[0]` and starting memory address for computational results. With the successful receipt of the sixth command the accelerator initiates hardware-based execution of the algorithms in Figs. 2.3 and 2.5. This process is conducted between the accelerators and memory units (cache and DRAM) until all the events are processed and associated outputs are generated. Since the RISC-V processor uses virtual addressing, it conveniently integrates the accelerator's load/store actions within the standard memory space. Thus, programming for the exchange of data between the processor's main memory and the accelerator is straightforward.

3.2.2 Trellis Constructor Hardware Design

The trellis construction accelerator executes loop (1) from Fig. 2.3 sequentially due to the data dependency across iterations. Specifically, each state vector Ψ_m relies on prior values Ψ_{m-1}, necessitating a serial traversal of the event loop. However, by increasing the execution rate R of this loop, the architecture can exploit parallelism across C buffered input channels, enabling scalable processing of multiple input streams.

To support higher throughput, the accelerator implements fully unrolled versions of loops (2) through (4), leveraging the independence of operations within these regions. Loops (2) and

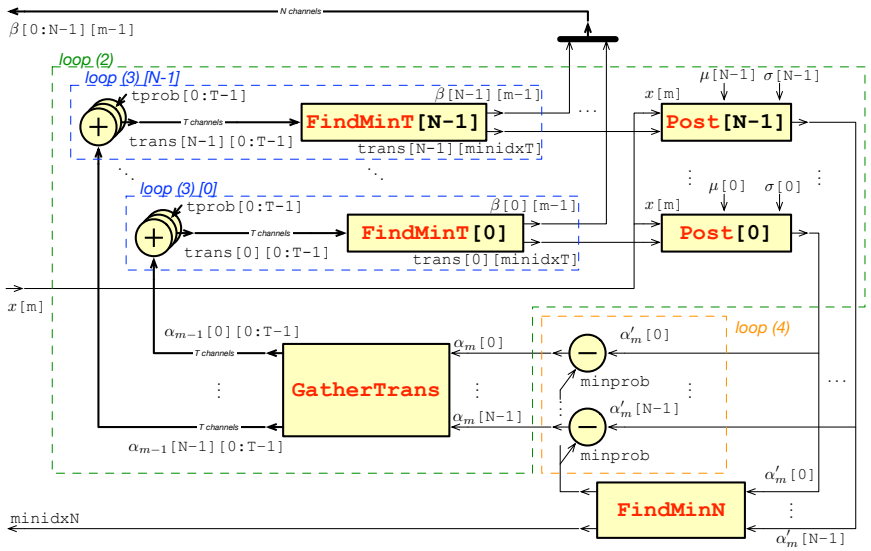

Figure 3.3: Trellis construction hardware design

(3), responsible for state and transition computations, execute in parallel across N elements, followed by the normalization stage defined in loop (4). For this implementation, parameters were fixed at $T = 21$ and $N = 64$, aligning with the DNA sequencing application described in [29]. The system can accommodate a maximum of $M = 512$ input events per segment. For read lengths exceeding this limit, the RISC-V host processor divides event sequences into $M \leq 512$ chunks for accelerated processing. Reassembly into complete reads follows after decoding.

The hardware architecture supporting this functionality is illustrated in Fig.3.3, with components corresponding closely to loops (2) and (3) from the algorithm in Fig.2.3. The design begins at the `GatherTrans` block, which outputs T-length posterior vectors α_{m-1}`[n][0:T-1]` to each of the N loop (3) datapaths. Because the output destinations are fixed, `GatherTrans` is realized as a hardwired redistribution network.

Each loop (3) unit comprises T adders and a `FindMinT[n]` module implementing the `FindMin(trans,`T`)` operation from line 8 of Fig.2.3. The adders calculate the unrolled summation in line 6, with results forwarded to the `FindMinT` blocks. This produces N trellis

pointers β[0:N-1][m-1] buffered for traceback.

Concurrently, each loop (3) block generates trans[n][minidxT] values. These, along with event observations x[m], are input to the Post blocks, which compute posterior updates as defined by equation (2.5). The N parallel posterior computations complete loop (2) and feed into an unrolled implementation of loop (4). Normalization is performed using a FindMinN block, implementing FindMin(α'_m,N) (line 12, Fig. 2.3), which also outputs minidxN, the final state used to initiate traceback.

A complete loop (1) iteration spans 18 cycles. Of these, seven are allocated to loop (3) and Post operations, seven to normalization via loop (4) and FindMinN, and four to internal communication and handshaking, including buffer management.

The architecture of the FindMinT block aligns with line 6 of the algorithm. 21 adders combine α_{m-1} with transition probabilities tprob to generate 21 trans terms. A pipelined comparator tree (C1 to C5) identifies the minimum trans value and corresponding index in five cycles. These relative trellis pointers β simplify hardware but shift global resolution complexity to the traceback stage, addressed in §3.2.3.

The Post block operates in parallel with FindMinT, combining the selected trans values with observation data x and model parameters μ and σ to produce posterior scores. These outputs are subsequently normalized. The FindMinN block mirrors the FindMinT design but employs six comparator stages to find the minimum of 64 Post outputs.

3.2.3 Traceback Hardware Design

The traceback block builds on the trellis construction engine by incorporating dedicated hardware to accelerate the traceback phase of the Viterbi algorithm. As depicted in Fig. 2.5, the traceback unit is positioned between two memory buffers: an input pointer buffer and an output state buffer.

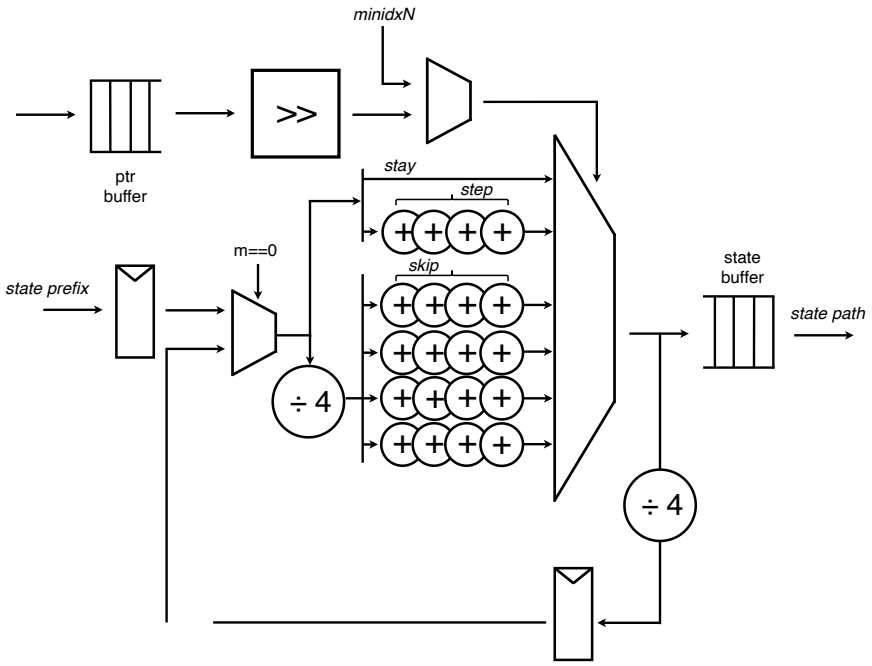

Figure 3.4: Traceback hardware design

The pointer buffer is a 32-KiB SRAM which stores the relative trellis pointers (β) generated by the trellis engine. Conversely, the state buffer is a 384-byte register file that accumulates the final global state sequence. Upon completion, the sequence (denoted *state_path*) is streamed to the RISC-V memory via the RoCC interface.

Traceback starts when the trellis engine signals completion all M trellis pointer outputs for a given event segment. The traceback hardware subsequently executes M-1 iterations of the loop described in Fig. 2.5, proceeding in reverse chronological order. In each iteration, N relative pointers are retrieved from the pointer buffer starting at address $m = M - 2$. The specific pointer for the current iteration is selected by right-shifting the fetched row by *prev_state* bytes. The least-significant byte resulting from this shift provides the relative trellis pointer utilized for tracing back through the state sequence. The initial *prev_state* is provided by the minidxN output from the trellis engine.

Given that the trellis engine emits relative pointers indexed locally from 0 to 20, the

traceback hardware must map them to global state indices ranging from 0 to 63. This transformation is achieved using expressions (2.3) and (2.4) from § **??**. The requisite arithmetic operations are implemented using adder banks and 2-bit shift blocks, which perform integer division by four. Each iteration computes 21 potential global states that could transition into the current *prev_state*. These states are then presented to a state multiplexer which selects the correct global state based on the recently fetched relative pointer.

In each iteration, the selected global state is written to the state buffer and concurrently passed back as the new *prev_state* for the subsequent step, forming a continuous feedback loop until the traceback operation concludes. This architecture achieves efficient and fully pipelined traceback execution by performing on-the-fly conversion of relative pointers to global indices and minimizing control overhead. The outcome is a hardware-accelerated state sequence reconstruction tightly integrated with the upstream trellis engine.

Chapter 4

Hardware Implementation

The DNA sequencing SoC described above is implemented in Global Foundries 22-nm fully-depleted system-on-insulator (FDSOI) CMOS process. The hardware was designed using Synopsys Design Compiler for synthesis, Synopsys IC Compiler 2 for place and route and Altium Designer for PCB work. The chip is wirebonded to a ball grid array (BGA) package substrate and placed within an elastomer contact BGA socket shown in Figure 4.1.

4.1 PCB Platform

The hardware system is distributed across two printed circuit boards - a host board implemented on the Zynq Evaluation Development Board (ZedBoard) and the target board called Real Trouble (RT). In addition, there are 2 FPGAs responsible for a large part of the computing and communications in the system. They are:

- Gateway FPGA XC6SLX150-FGG676 (GW): The FPGA chip used to mediate communications between ZB and RT.
- Zynq-7000 XC7Z020-1CLG484CES (Z7K): The FPGA chip used to link the ASIC to certain external software as well as DRAM.

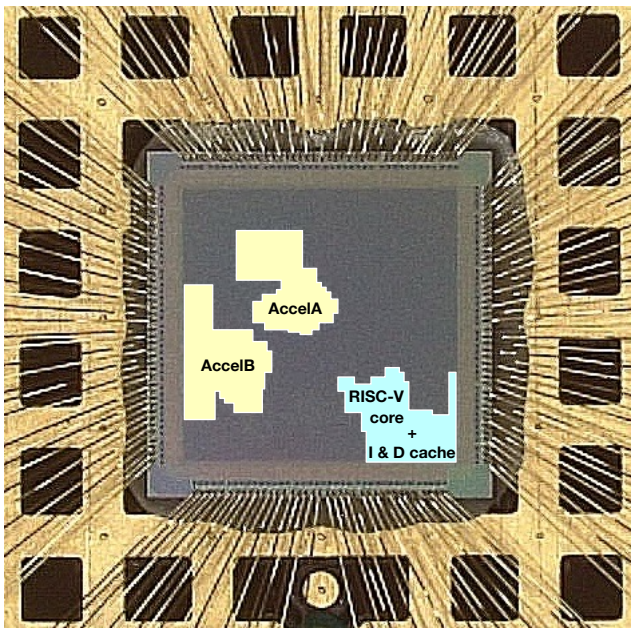

Figure 4.1: Wired-bonded die

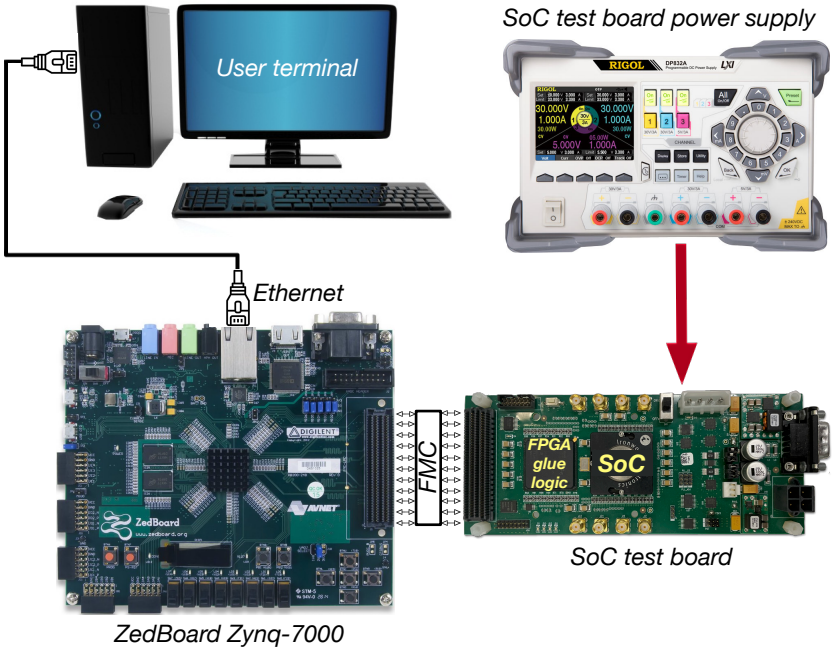

Figure 4.2: Test Platform

Z7K links up to GW via a FPGA Mezzanine Card (FMC) connection and the GW connects to the ASIC via printed circuit board (PCB) traces. The physical layer communications between Z7K and GW take on a 1.8-V low-voltage differential signalling (LVDS) form while the communications between GW and AF take on a 3.3-V low-voltage transistor-transistor logic (LVTTL) form.

The ZB connects the Z7K to a low-pin-count (LPC) FMC connector. In total, 68 single-ended connections exist between the ZB's Z7K and the FMC. These connections can also be configured as 34 differential connections as per the FMC specification.

There are 88 lines carrying signals (and their communication support data) between GW and ASIC, 44 lines in each direction. In each direction 32 lines hold actual data while among the remaining 12 links there are 4 clocks for synchronization purposes; 4 flow control lines (to indicate valid status); 4 credit lines (to implement back-pressure signals). Each of one of the 4 clock, control and credit lines helps look after an 8-b subset of the 32-b being exchanged between GW and ASIC.

4.2 Operation

At a high level, execution of the sequencing algorithm consists of three main parts:

- System reset
- Main Memory initialization
- Program Execution

System reset involves asynchronous reset signals being asserted across all key blocks on ZB and RT. The communication blocks between GW and ASIC then undergo an initial calibration process before the rest of the system (host software running on the ARM Cortex

A9 CPU, Microblaze CPU on the GW and RISC-V SoC) begins operation. After calibration, the target program for the RISC-V is loaded into the RISC-V memory space. This happens via special software running on the ARM sending HTIF commands to the ASIC which are interpreted by the uncore logic in the Rocket core that provides access to the RISCV's shared memory.

With the program in main memory, system operation begins with an HTIF command sent to uncore specifying that the RISC-V must be pulled out of reset. Figure 4.2 contains a depiction of the complete system including i) the interface x86 PC, ii) the Zedboard, iii) Real Trouble.

4.2.1 Interface x86 CPU

The interface CPU's main function is primarily to allow a human tester to run the system. A standard x86 PC is used here. Via a remote terminal window on the PC, a user can `ssh` into the host CPU (described in the next section) through an Ethernet connection and thus can operate the host CPU from a shell. The host CPU in this platform is the ARM Cortex A9 processor present in the Zynq FPGA chip that is featured in the Zedboard development board.

4.2.2 Host ARM CPU

For testing the RISC-V SoC in this setup there is one program that is of interest for running on the host ARM Cortex A9 - the front-end server program, `fesvr`. This program allows the host to communicate with the RISC-V target CPU. This communication facilitates a number of important services for the target. These services include starting up the target (i.e. controlling resets), loading programs into memory, and facilitating the ability of the target to access those programs and their associated data from the host's filesystem.

The host is enabled with this capability by putting all necessary software on a flash memory (an SD card in the case of this test system) and subsequent transferal of this software 'into' the host ARM using the resources already available in the Zedboard development system. In particular, from the flash memory a Linux kernel is installed into the ARM's memory system as well as `fesvr`. Not only does the Linux kernel allow convenient setting up of `ssh` connections to the host, but it also serves to ultimately deal with system calls that may be coming from the program (i.e., the application) that's transferred into the target RISC-V CPU.

Among its functions, the `fesvr` program defines the memory system for the target RISC-V SoC and places programs that the RISC-V must execute into its memory space. In essence, with `fesvr` we map the target's memory system onto a portion of the Zedboard's DRAM and thus allow the target to use this memory for its computing needs.

Thus, in its simplest use, `fesvr` loads the application we wish to run on the target into the Zedboard's DRAM and then frees the RISC-V target to fetch and run this application in a bare-metal form starting at a prearranged program counter (PC) setting.

The `fesvr` program maintains control communications between the host and the target via intermittent commands implementing the host-target interface (HTIF) protocol. For instance, via HTIF the host could set and reset the target, tell it where its program is stored, allow the target to start executing, and stop the target when it is told by the target that the program is finished. In parallel with these control messages the target could be communicating with the host's Linux kernel and memory controller to fetch code and data from the space allotted to it by `fesvr`.

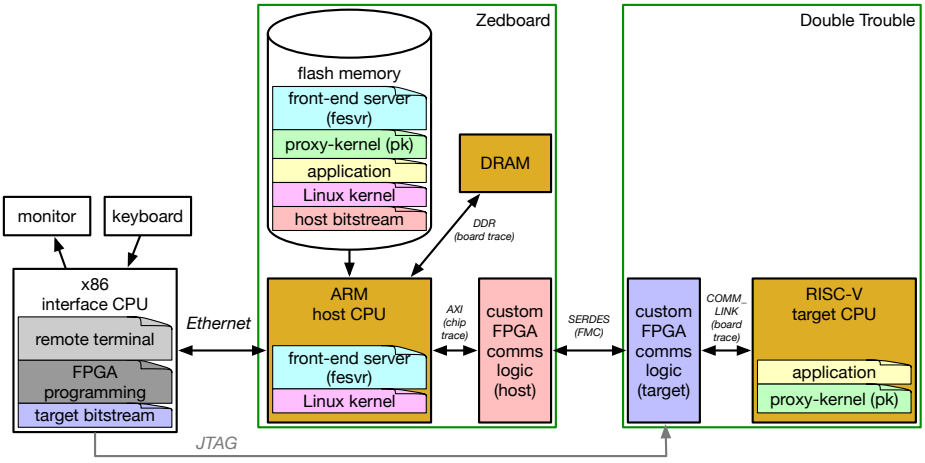

Figure 4.3: ASIC Test Setup

4.2.3 Target RISC-V SoC

The target RISC-V SoC is the custom computer which represents the design under test. It runs any suitably compiled RISC-V application, specifically the previously discussed basecalling algorithm. Two modes of program exeuction are supported: baremetal mode and operating system based execution. In the baremetal case, a physical addressing environment is employed.

If local operating system like services are required by the running application however, execution in a user-mode (U-mode) environment with virtual addressing is also supported. In this case, the target RISC-V's memory space requires not only the application of interest but also a supervisor program called `pk`, the proxy kernel. The `pk` provides operating system-like services such as paging and handles I/O related system calls by forwarding (proxying) them to the host's Linux kernel.

`pk` runs in supervisor mode (S-mode) and shares the same virtual address space as the application that it is tasked with loading. The lower 2 GiB of the virtual address space is reserved for the application and the `pk` is mapped after `0x80000000` (i.e., above 2 GiB). Figure 4.3 illustrates the complete software/hardware stack.

Chapter 5

Experimental Results

The system described across Figs. 3.2 was implemented in Global Foundries 22FDX CMOS process within a 3-mm on-a-side die. As noted above, the basecalling accelerator was coupled to a Linux-capable microprocessor with on-chip memory and standard I/O support. This allowed for evaluating the hardware within the context of a realistic computing system complete with off-chip and off-board peripherals. The implementation was based on an open-source multi-board test environment made available by BSG [30]. The processing core and accelerator consumed areas of 0.355 mm^2 and 0.401 mm^2 respectively.

5.1 Functional Verification

Our evaluation of the hardware commenced with a functional assessment. As previously established, the fixed capacity of the accelerator's internal buffers necessitates the segmentation and sequential processing of input event sequences into M-sized discrete chunks. To ascertain the resultant impact on detection accuracy and to corroborate functional correctness against a reference software implementation, detection operations were executed on a dataset comprising 1800 Matlab-generated base sequences. These sequences were subjected to event generation

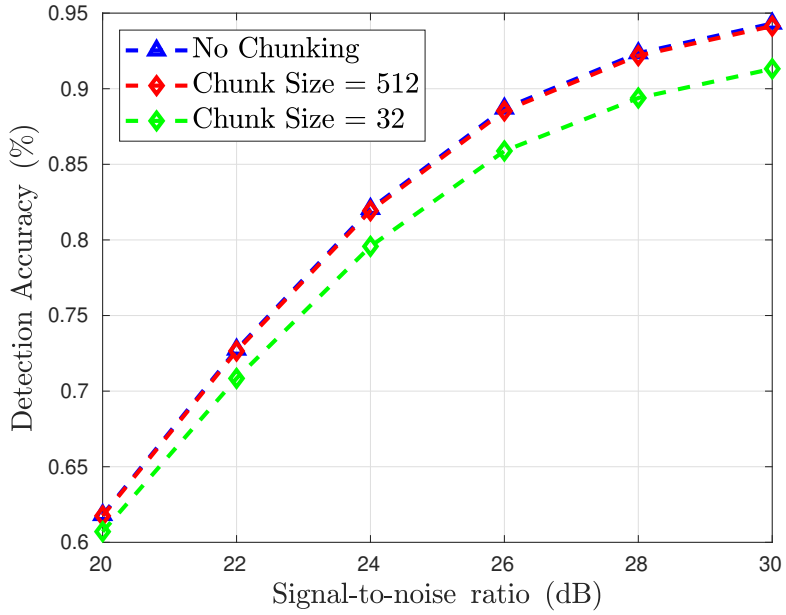

Figure 5.1: Accuracy as a function of chunk size

utilizing predictive nanopore 3-MER models.

The outcomes of this assessment are graphically represented in Fig. 5.1, which illustrates the detection accuracy as a function of the nanopore signal-to-noise ratio (quantified as the intrinsic noise level within an individual event measurement). The blue curve denotes the idealized operational scenario, wherein the algorithmic processing of the input sequence is performed in its entirety, obviating any form of chunking. Conversely, the red and green curves delineate the performance for event sequences processed with chunk sizes of 512 and 32 events, respectively, reporting the aggregated accuracy of the reconstructed output. Each data point presented signifies the arithmetic mean accuracy derived from 100 distinct input samples. As evinced by the graphical representation, detection sustained accuracy levels exceeding 90%, even when employing chunk sizes as diminutive as 32. Furthermore, the concordance between the hardware's output and that of the reference software was absolute, thereby unequivocally confirming its functional correctness.

5.2 Performance

Figure 5.2 presents the basecalling throughput of the SoC's application processor, expressed in kilobases per second, as a function of system clock frequency. While simulation results suggest that the chip architecture supports clock rates exceeding 1 GHz, practical testing was constrained by two factors: the use of an external clock source and limitations inherent in the I/O standard cell libraries available at the time of tapeout.

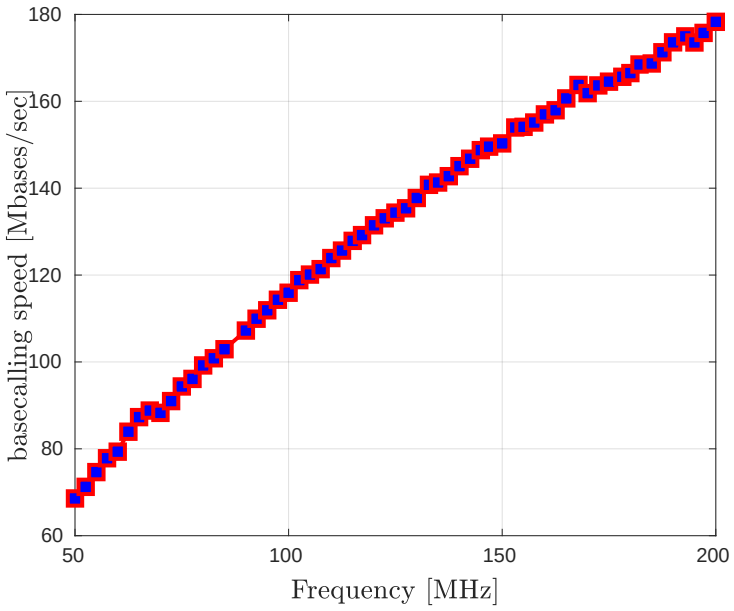

Figure 5.2: Speed vs Frequency

The measured performance of the SoC exhibits a monotonic increase with clock frequency as it scales efficiently up to a tested maximum of 200MHz. At this upper bound the system achieves a peak basecalling rate of 6.9 million bases per second (Mb/s). For practical context, this rate corresponds to processing approximately 3,700 complete SARS-CoV-2 viral genomes per hour. If extrapolated to a clock frequency of 1GHz, the projected throughput reaches nearly 18,600 viral genomes per hour which equates to approximately 17% of a full human genome. A minor performance taper is observable at higher frequencies. Specifically,

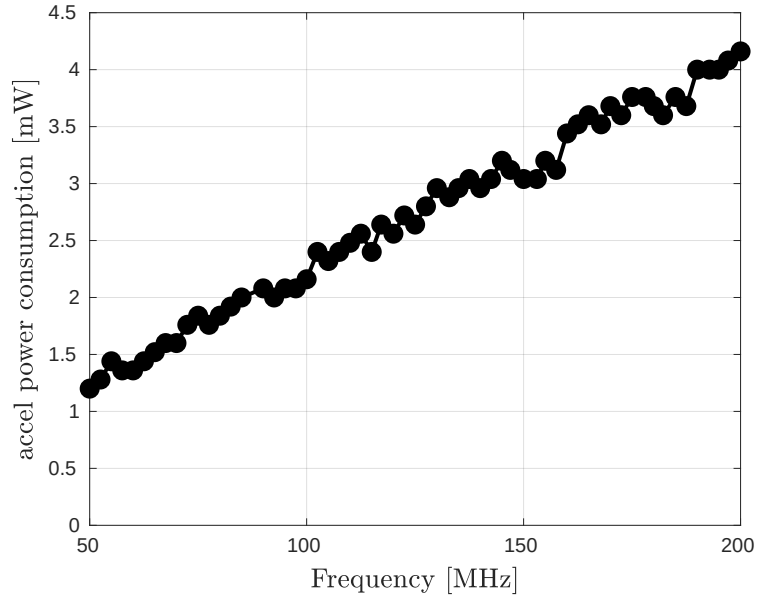

Figure 5.3: Power vs Frequency

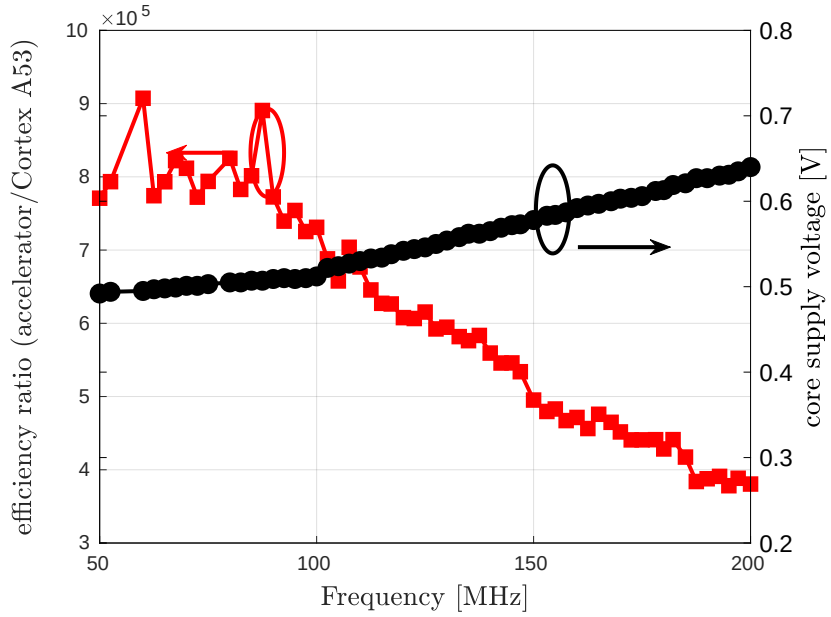

Figure 5.4: Efficiency vs Frequency

Table 5.1: Performance comparison of basecalling hardware implementations

Reference	Node	System	Efficiency	ASIC Eff. Boost
ASIC Accelerated (200-MHz)	**22-nm**	**SoC**	**37.4 Mb/s/W**	–
Embedded Proc. (Extrapolated, 200 MHz)	16-nm	ARM A53	157 Kb/s/W	238×
Embedded Proc. (Actual, 1.0 GHz)	16-nm	ARM A53	182 Kb/s/W	206×
FPGA Accelerated[9]	28-nm	x86+Virtex7	352 Kb/s/W	106×
Desktop Proc. (2.4 GHz)[29]	22-nm	Intel Xeon	3 Kb/s/W	12.5K×

between 100MHz and 200MHz, the system exhibits an approximate 1% drop from ideal linear scaling. This deviation is attributable to increasing memory subsystem limitations, a well-documented phenomenon wherein memory bandwidth becomes a performance bottleneck as core frequency rises without a proportional increase in memory access rates [**Hennessy17**]. The performance loss reflects the growing contribution of memory-induced stalls during high-frequency operation.

For comparative analysis, Fig.5.2 also includes the basecalling performance of a commercial-grade processor: the ARM Cortex-A53 embedded in the Xilinx Zynq UltraScale+ MPSoC (16-nm process technology). The same basecalling workload was executed on this platform. Although the Cortex-A53 supports dynamic frequency scaling up to 1.2GHz, hardware constraints limited its operational floor to 300MHz. To facilitate a direct comparison across the full frequency spectrum, performance values for the A53 below 300MHz were extrapolated using empirical data collected between 300 and 1200 MHz.

As illustrated in the figure, the proposed hardware accelerator consistently outperforms the Cortex-A53 at matched frequencies and achieves an average speedup of 10.3× across the measured range. Even when comparing the 200-MHz accelerator output to the A53 operating at its peak frequency of 1.2 GHz, the accelerator maintains a performance advantage of 1.8×. This underscores the efficiency of the design despite its operation in an older process node.

5.3 Power

The power consumption characteristics of the custom accelerator block during execution of the basecalling algorithm are presented in Fig. 5.3. The figure shows average power dissipation measured across a range of operating frequencies for the accelerator's clock. Each data point represents the mean of three independent power measurements. All measurements were performed under the nominal supply voltage of the 22-nm technology node, $V_{DD} = 0.8$ V.

Observed variations in power draw are primarily attributable to dynamic activity within the on-chip cache memory and fluctuations in traffic through the uncore communication infrastructure. The latter includes interface logic responsible for managing data transfers between the accelerator and external DRAM.

To evaluate the potential for improved energy efficiency, we also investigated operation under reduced supply voltages. Manual testing revealed that the accelerator remains functionally correct at voltages significantly below the nominal V_{DD}. The results of this low-voltage operation are shown in Fig. 5.4, where the black curve (referenced to the right y-axis) denotes the minimum supply voltage required to sustain correct operation at various clock frequencies.

As indicated in the figure, stable execution was observed down to VDD = 0.49 V at a 50-MHz clock rate and down to VDD = 0.64V at 200MHz. These findings suggest that with appropriate voltage scaling the power budget of the accelerator can be constrained to under 1mW while maintaining full functionality up to 200MHz. Such operational flexibility highlights the energy efficiency of the design and its suitability for power-constrained environments.

5.4 Efficiency

The performance efficiency of the accelerator is also quantified where efficiency is defined as the number of bases processed per second per watt of power consumed. Specifically, the ratio

between the efficiency of the accelerator-based system and the peak efficiency achieved by the ARM Cortex-A53 processor is depicted by the red curve (left y-axis) in Fig. 5.4.

For the ARM A53, the maximum observed efficiency was 182 kilobases per second per watt (Kb/s/W) at a 1-GHz clock rate. This figure was derived by isolating the dynamic power consumption attributable solely to execution of the basecalling workload, subtracting background power from total power during active computation. Across the measured frequency range of the accelerator, a performance efficiency advantage exceeding 170× relative to the A53's peak value was consistently. This gain reflects the substantial improvements enabled by application-specific hardware integration.

While some variability in efficiency arises from memory subsystem interactions—mirroring the memory-induced performance fluctuations discussed previously—the overall efficiency advantage remains substantial. At lower frequencies where power demands scale more favorably with voltage, the advantage peaks around 300×. Even at the upper end of the frequency spectrum, where increased supply voltage begins to impact energy efficiency more significantly, the advantage remains above 170×.

A broader comparison of various hardware platforms capable of executing basecalling workloads is provided in Table 5.1. The first row summarizes the performance of the proposed system - which to our knowledge represents the first custom SoC designed specifically to support nanopore-based sequencing in embedded environments. As no directly comparable ASIC designs currently exist, comparisons are drawn against representative FPGA-accelerated and embedded processor-based solutions.

When compared against ARM A53 processors implemented in more advanced CMOS process nodes, the accelerator demonstrates a 238× improvement in basecalling efficiency relative to an A53 extrapolated to operate at 200MHz (as in Fig.5.2). Against the A53's peak measured efficiency at 1 GHz, the system still achieves a 206× improvement.

In addition, relative to the FPGA-based hybrid solution described in [9]—which distributes

basecalling computation across an x86 host processor and a Virtex-7 FPGA—our SoC achieves a 106× gain in computational efficiency. Finally, when compared to a conventional desktop CPU implementation reported in [29], the system presented in this work offers over four orders of magnitude improvement in efficiency.

Taken together, these comparisons highlight the effectiveness of the custom accelerator in delivering a favorable balance between computational throughput and energy consumption. The results affirm the potential of domain-specific SoC architectures in advancing the feasibility of portable, low-power genomic sequencing systems.

Chapter 6

Conclusion and Future Work

Recent advancements in DNA sequencing technology have enabled significant miniaturization, thus making mobile sequencing applications increasingly feasible. However, realizing the full potential of such applications requires integrating computational systems that combine sufficient processing capability with high energy efficiency. In response to this need, an experimental low-power computing platform tailored for future mobile DNA sequencing systems is presented. These systems are expected to deliver acceptable levels of computational performance while maintaining minimal power consumption and retaining the programmability necessary to support auxiliary sequencing tasks.

To meet these objectives, we developed a Linux-capable system-on-chip (SoC) based on a RISC-V processing core. The SoC features integrated on-chip cache memory, interfaces for external memory and peripheral communication and a specialized hardware accelerator designed to significantly enhance basecalling throughput. Compared to a commercial-grade mobile processor (ARM Cortex A53), the system presented here achieves nearly a 10× extrapolated performance speed-up at a 200-MHz clock while running from a 0.64-V power supply. Perhaps even more importantly our processor can achieve a 238× improvement in computational efficiency over the A53 at this frequency (and 206× relative to an optimally

clocked A53).

In summary, this work highlights the significant promise of semi-specialized low-cost processors in advancing mobile DNA sequencing. The results underscore the substantial performance gains achievable through heterogeneous computing architectures tailored for domain-specific workloads. The system presented offers a practical and immediate opportunity to accelerate basecalling operations in resource-constrained environments.

While the current evaluation was limited by reliance on external clock sources and low-speed I/O circuitry, these constraints do not reflect the full capabilities of the design. The use of 22-nm fully-depleted silicon-on-insulator (FDSOI) CMOS technology presents considerable potential for further enhancement. This process technology supports higher clock frequencies and enables advanced power management techniques such as dynamic voltage and frequency scaling (DVFS) and body bias control. These capabilities offer promising avenues for improving both performance and energy efficiency in future iterations of the system, reinforcing the viability of domain-specific SoCs for embedded genomic analysis.

Bibliography

[1] Yunhao Wang et al. "Nanopore sequencing technology, bioinformatics and applications". In: *Nature biotechnology* 39.11 (2021), pp. 1348–1365.

[2] John E Gorzynski et al. "Ultrarapid nanopore genome sequencing in a critical care setting". In: *New England Journal of Medicine* 386.7 (2022), pp. 700–702.

[3] Kaire Loit et al. "Relative performance of MinION (Oxford Nanopore Technologies) versus Sequel (Pacific Biosciences) third-generation sequencing instruments in identification of agricultural and forest fungal pathogens". In: *Applied and Environmental Microbiology* 85.21 (2019), e01368–19.

[4] Kaikai Chen et al. "Nanopore-based DNA hard drives for rewritable and secure data storage". In: *Nano letters* 20.5 (2020), pp. 3754–3760.

[5] Qiang Liu et al. "DNA-based molecular computing, storage, and communications". In: *IEEE Internet of Things Journal* 9.2 (2021), pp. 897–915.

[6] Ardhendu Sarkar, Som Banerjee, and Surajeet Ghosh. "An Energy-Efficient Pipelined-Multiprocessor Architecture for Biological Sequence Alignment". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.12 (2020), pp. 2598–2611. DOI: 10.1109/TVLSI.2020.3015138.

[7] Mohaddeseh Sharei et al. "GEMA: A Genome Exact Mapping Accelerator Based on Learned Indexes". In: *IEEE Transactions on Biomedical Circuits and Systems* 18.3 (2024), pp. 523–538. DOI: 10.1109/TBCAS.2023.3348152.

[8] Haiyu Mao et al. "GenPIP: In-Memory Acceleration of Genome Analysis via Tight Integration of Basecalling and Read Mapping". In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2022, pp. 710–726. DOI: 10.1109/MICRO56248.2022.00056.

[9] Z. Wu et al. "FPGA-Accelerated 3rd Generation DNA Sequencing". In: *IEEE Transactions on Biomedical Circuits and Systems* 14.1 (Feb. 2020), pp. 65–74. ISSN: 1940-9990. DOI: 10.1109/TBCAS.2019.2958049.

[10] Karim Hammad et al. "A Scalable Hardware Accelerator for Mobile DNA Sequencing". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 29.2 (2021), pp. 273–286. DOI: 10.1109/TVLSI.2020.3044527.

[11] Neng Huang et al. "SACall: A Neural Network Basecaller for Oxford Nanopore Sequencing Data Based on Self-Attention Mechanism". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 19.1 (2022), pp. 614–623. DOI: 10.1109/TCBB.2020.3039244.

[12] Di Xu et al. "TransCaller: An End-to-end Accelerated Transformer-based Nanopore Basecaller on GPUs". In: *2023 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. 2023, pp. 704–711. DOI: 10.1109/BIBM58861.2023.10386023.

[13] Frederick Sanger, Steven Nicklen, and Alan R Coulson. "DNA sequencing with chain-terminating inhibitors". In: *Proceedings of the national academy of sciences* 74.12 (1977), pp. 5463–5467.

[14] Jose Antonio Garrido-Cardenas et al. "DNA Sequencing Sensors: An Overview". In: *Sensors* 17.3 (2017). ISSN: 1424-8220. DOI: 10.3390/s17030588. URL: https://www.mdpi.com/1424-8220/17/3/588.

[15] Timothy Beck and et al. "Systematic evaluation of Sanger validation of next-generation sequencing variants". In: *BMC Genomics* 15 (2014). DOI: 10.1186/1471-2164-15-844.

[16] Josephine B. Oehler et al. "The application of long-read sequencing in clinical settings". In: *Human Genomics* 17.1 (Aug. 2023), p. 22. DOI: 10.1186/s40246-023-00522-3.

[17] Ting Hon et al. "Highly accurate long-read HiFi sequencing data for five complex genomes". In: *Scientific Data* 7 (2020). DOI: 10.1038/s41597-020-00743-4.

[18] J. G. Caporaso, M. P. Cummings, and R. Knight. "Nanopore sequencing technology, bioinformatics and applications". In: *Nature Biotechnology* 39 (2021), pp. 789–807. DOI: 10.1038/s41587-021-01108-x.

[19] Cédric Delahaye and Jean Nicolas. "Sequencing DNA with nanopores: Troubles and biases". In: *PLOS ONE* 16.10 (2021), e0257521. DOI: 10.1371/journal.pone.0257521.

[20] Aaron M. Wenger et al. "Accurate circular consensus long-read sequencing improves variant detection and assembly across human genomes". In: *Nature Reviews Genetics* 21 (2020), pp. 125–148. DOI: 10.1038/s41576-019-0177-0.

[21] Barton E. Slatko, Andrew F. Gardner, and Frederick M. Ausubel. "Overview of Next-Generation Sequencing Technologies". In: *Current Protocols in Molecular Biology* 122.1 (2018), e59. DOI: https://doi.org/10.1002/cpmb.59. eprint: https://currentprotocols.onlinelibrary.wiley.com/doi/pdf/10.1002/cpmb.59. URL: https://currentprotocols.onlinelibrary.wiley.com/doi/abs/10.1002/cpmb.59.

[22] Neng Huang et al. "SACall: A Neural Network Basecaller for Oxford Nanopore Sequencing Data Based on Self-Attention Mechanism". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 19.1 (2022), pp. 614–623. DOI: 10.1109/TCBB.2020.3039244.

[23] Xuan Lv et al. "An End-to-end Oxford Nanopore Basecaller Using Convolution-augmented Transformer". In: *2020 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. 2020, pp. 337–342. DOI: 10.1109/BIBM49941.2020.9313290.

[24] C N Ramachandra et al. "ONT-X: An FPGA Approach to Real-time Portable Genomic Analysis". In: *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2021, pp. 268–269. DOI: 10.1109/FCCM51124.2021.00054.

[25] S Reddy, LH Hung, and O Sala-Torra. "A graphical, interactive and GPU-enabled workflow to process long-read sequencing data". In: 22.1 (2021), p. 626. DOI: doi.org/10.1186/s12864-021-07927-1.

[26] Kanokwan Rungsuptaweekoon, Vasaka Visoottiviseth, and Ryousei Takano. "Evaluating the power efficiency of deep learning inference on embedded GPU systems". In: *2017 2nd International Conference on Information Technology (INCIT)*. 2017, pp. 1–5. DOI: 10.1109/INCIT.2017.8257866.

[27] Gagandeep Singh et al. *A Framework for Designing Efficient Deep Learning-Based Genomic Basecallers*. 2022. arXiv: 2211.03079 [cs.AR].

[28] W Timp, J Comer, and A Aksimentiev. "DNA base-calling from a nanopore using a Viterbi algorithm". In: 102.10 (2012), pp. 37–39. DOI: 10.1016/j.bpj.2012.04.009.

[29] W. Timp, J. Comer, and A. Aksimentiev. "DNA Base-calling from a Nanopore using a Viterbi algorithm". In: *Biophysical J.* 102.10 (May 2012), pp. L37–L39.

[30] Michael Bedford Taylor. "BaseJump STL: SystemVerilog Needs a Standard Template Library for Hardware Design". In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465909.