Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Electronic Engineering (35^{th}cycle)

VLSI Architectures for Video Processing and RISC-V

By

Walid Walid
******

Supervisor(s):
Prof. Maurizio Martina, Supervisor
Prof. Guido Masera, Co-Supervisor

Doctoral Examination Committee:
Prof. Attilio Fiandrotti, Referee, Università degli Studi di Torino
Prof. Massimo Conti, Referee, Università Politecnica delle Marche
Prof. Mario Casu, Politecnico di Torino
Prof. Marco Vacca, Politecnico di Torino
Dr. Maurizio Capra, Synthara AG

Politecnico di Torino
2023

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Walid Walid
2023

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

I would like to dedicate this thesis to my loving parents

Acknowledgements

First, I would like to thank Almighty Allah (God) for all the blessings in my life and for giving me patience in hard times. Second, I want to thank my supervisors, Prof. Maurizio Martina and Prof. Guido Masera, for their support and professional guidance during this thesis work. It has been very inspiring work under their supervision. I am incredibly thankful to them for their input in my work and for deciding the direction of my work. Third, I thank the Department of Electronics of Politecnico di Torino for providing facilities and online material support for my thesis work. I thank the collaborators for the video processing part, Prof. Muhammad Awais and Prof. Ashfaq Ahmed, Giorgio Armanno, and Sandro Di Paola. I would also like to thank Michele Caon for collaborating on the LEN5 part. Finally, I wish to give my gratitude and thanks to my family and friends for their support.

Abstract

In the internet and smartphone era, video conferencing and live streams are a big part of information transmission. Video processing involves many domains, from high-quality video transmissions to real-time decision-making systems. The usual software-based models and serial approach running on single-issue cores are slow. Hence, there is a need for speed up to make it compatible with real-time applications. The video processing tasks have parallelism that can be exploited for this speedup via the hardware acceleration. Thus, this work is aimed towards achieving this task. In this aspect, two algorithms were targeted for hardware acceleration. In addition, the processor domain is also explored in order to select cores for video processing.

The first algorithm is in the domain of object tracking. Tracking objects in real-time is a crucial step in image processing. Thus the hardware real-time object tracker for several image resolutions should provide good accuracy alongside meeting the frame rate requirements. The sophisticated algorithms employed in the state-of-the-art are computationally demanding for embedded architectures. An algorithm with good performance and high parallelism is discriminative scale space tracking. The parallelism can be exploited for speedup with hardware acceleration. The first part of this work proposes hardware architectures for the major blocks to attain the real-time requirements of a discriminative scale space tracker on FPGA. The main contributions in this work are the improvements in the architecture for the core mathematical functions in the algorithm, including the discrete Fourier transform, QR factorization, and singular value decomposition. Further, for 320×240 image resolution, the proposed architecture obtains a mean 25.38 fps.

The second algorithm is in the domain of video encoding. In recent years, many devices have been using videos for communication. With the adaptation of high-quality video and image content, the need for more memory space is increased. It also demands more data compression from the advancing video coding techniques.

These demands led to the development of AOMedia Video 1 or AV1 by the Alliance for Open Media. AV1 is a royalty-free codec, thus allowing it to be used in many devices without paying extra cost. This work also studies the AV1 model and selects the Wiener filter as a complex block requiring acceleration. Wiener filter is an in-loop restoration tool in AV1. This work presents a separable symmetric normalized architecture for its implementation. The proposed design allows 100 $\times$ reduction in processing time and 5 $\times$ speedup in mega samples per second concerning existing works in literature.

The thesis's third part concerns the processor core for video processing applications. Many Instruction Set Architectures are available but often closed and incremental Instruction Set Architectures. On the contrary, RISC-V is the new open and royalty-free Instruction Set Architecture developed at UC Berkeley in 2010. Apart from being an open architecture, RISC-V is a modular Instruction Set Architecture. Starting from the base Instruction Set Architectures, any other extension can be implemented as per need without implementing all the previous ones. This modularity allows RISC-V to be adaptable for video processing tasks. This work first identifies the core requirement for video processing tasks based on which a processor can be selected. Next, it explores the RISC-V cores in literature based on the selected parameters. Afterward, it compares them and selects those most suitable for video processing. In addition, the work also extends a RISC-V core in literature called LEN5 for video processing.

Contents

List of Figures

List of Tables

Nomenclature

Acronyms / Abbreviations

ALU	Arithmetic Logical Unit
AOMedia	Alliance for Open Media
ASIC	Application Specific Integrated Circuits
AV1	Alliance for Open Media Video 1
AXI	Advanced eXtensible Interface
BE	Back End
BOOM	Berkeley Out-of-Order Machine
BRAM	Block RAM
CDB	Common Data Bus
CORDIC	COordinate Rotation DIgital Computer
CPU	Central Processing Unit
DCF	Discriminative Correlation Filter
DCT	Discrete Cosine Transform
DC	Data Cache
DFT2	Discrete Fourier Transform 2
DFT	Discrete Fourier Transform

DIF	Decimation in Frequency
DIT	Decimation in Time
DP	Diagonal Processor
DSP	Digital Signal Processor
DSST	Discriminative Scale Space Tracking
D	Dimension
FDSST	Fast Discriminative Scale Space Tracking
FE	Front End
FFT	Fast Fourier Transform
FF	Flip Flop
FHOG	Felzenszwalb's Histogram of Gradients
FPGA	Field Programmable Gate Arrays
FPS/fps	Frames Per Second
FSM	Finite State Machine
GPU	Graphical Processor Unit
HEVC	High-Efficiency Video Coding
HLS	High-Level Synthesis
HOG	Histogram of Gradients
HSF	High-Speed Final Architecture
IC	Instruction Cache
IEC	International Electrotechnical Commission
ILP	Instruction Level Parallelism
IoT	Internet-of-Things

IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
ISE	Integrated Software Environment
ISO	International Organization for Standardization
ITU	International Telecommunication Union
L2	Level-2 Cache
LUT	LookUp Table
MEM	Memory
MIMO	Multiple-Input–Multiple-Output
MPEG	Moving Picture Experts Group
OoO	Out-of-Order
PCA	Principal Component Analysis
PSNR	Peak Signal to Noise Ration
QR	QR factorization
RD	Restoring Divider
RISC-V	Reduced Instruction Set Computer (RISC) Five
ROB	Re-order Buffer
RT-DSST	Real-Time Discriminative Scale Space Tracking
RTL	Register Transfer Level
RT	Real-Time
SIMD	Single-Instruction-Multiple-Data
SVD	Singular Value Decomposition
VCEG	Video Coding Experts Group
VLSI	Very Large Scale Integrated Circuits

Chapter 1

Introduction

Humans use the processing of information in many different fields. Processing and sorting to a meaningful form and getting a result out makes information valuable and vital. Humans use the brain (a constant information-processing machine) and tools for information processing. As human society grows, the amount of information to process increases; thus, just using the brain is insufficient and slow to process. In such times, humans turn to tools for rescue. The tools have become more sophisticated through the years, from simple mechanical machines like the abacus to analog devices, simple circuits, and modern digital computers. Digitization started with a primary circuit like a counter to count, timers, and automatic switches, then circuits that could process information from sensors and, based on the transistor, could take decisions to control the actuators [1]. In this case, the processing of information is on a tiny scale. Next, the circuits moved from simple to complex designs that process instructions called processors. A processor is a machine that processes information by taking instructions and operating on some data based on these instructions [2].

With this evolution, the trend was to process more information in less time. Now several scientific fields could take advantage of these tools. The new applications instead started demanding more and more processing. So now, latency and information size have become crucial. When this trend entered medicine, accident prevention, and security, real-time and accuracy became a priority. Recently, most of the information flowing in mainstream media and the Internet is based on images and videos [3]. They are responsible for the majority of the information population. For purposes such as real-time streaming, and accident prevention based on road

scene understanding, there is a need for fast and accurate information processing (in this context, image/video processing). Thus there is a need for fast algorithms and devices to implement them. Based on these motivations, this thesis's work focuses on the implementation and hardware acceleration of image and video processing algorithms.

The variety of applications requires that the processing is tailored to their need. Some required speed, some demanded accuracy, and others required less response time. Consequently, to accommodate all, different objectives are targeted. The processing devices are mainly processors, Application Specific Integrated Circuits (ASIC) devices, or Field Programmable Gate Arrays (FPGA). Processors provide much flexibility with software support but have a cost and limited speed. Also, the software-based implementations are serial. Thus if the algorithm offers high parallelism, the software running on a single-core single-threaded core cannot exploit this. The software running on a multi-core or multi-threaded architecture can perform better but at the cost of high power and resource consumption. On the other hand, FPGAs can utilize parallel processing. They provide high flexibility with reasonable speed with limited resources. ASICs can be customized to get more speed or less area but have less flexibility [4]. An alternative is a Graphical Processor Unit (GPU), which can operate at a very high speed and utilize a high degree of parallelism but is costly. So based on the task at hand, the corresponding device is utilized for implementation. Applications requiring low power, remote usability, and fast implementations use ASIC or FPGAs. This work focuses on ASIC and FPGA-based processing providing customization, portability, low cost, and low power consumption [5].

There needs to be more than merely the selection of a device and acceleration to achieve the target optimization in video and image processing. As applications may use algorithms for better accuracy and performance, the corresponding hardware cost can be high. The goal is also to improve algorithm-to-architecture mapping and resolve the complexity in the algorithm domain keeping in mind the final architecture. So then, architecture-level optimization can achieve further improvement. Furthermore, it will help develop much better final designs as compared to if only architecture-level optimizations are applied [6]. Thus the choice of algorithm is also considered. In this work, the focus is on selecting the algorithm that has a better chance of hardware acceleration.

Thus from the discussion above, three main elements are highlighted. First, information processing in video and image processing requires attention in terms of speed to achieve real-time portable solutions. The target devices for such applications can be ASICs and FPGAs, which offer portability and parallel execution capabilities at low power. The selected algorithms from the literature on a critical path with parallelism to be exploited can help improve the overall system. Thus the goal from here is to review the literature, select the algorithm, analyze the algorithm to find the critical block, and accelerate the critical block with the help of hardware to improve the system's overall performance.

The information processing for security uses a lot of video processing at remote locations like surveillance. Video processing in such environments is essential as they are unsuitable for human presence. Furthermore, streaming video processing in real-time plays a significant role in live sports and events, with a vast potential market. Given all these primary motivations, the focus is on algorithm-to-architecture mapping for these video processing algorithms. Two algorithms were analyzed for this purpose. Fast Discriminative Scale Space Tracking (FDSST) and Wiener filter in video encoding as they present reasonable complexity and have crucial applications [7, 8]. Thus acceleration for them is essential. The first goal was to observe the improvements that the selected algorithms for the implementation of sub-blocks, for the FDSST and Wiener filter, can have on the final architecture. Next, was to focus on how the techniques involved, such as pipelining, parallel processing, unfolding, and folding, can improve the usual parameters at Register Transfer Level (RTL) level. The first topic is in the domain of object tracking. It is considered highly applicable. It was selected as it has one of the most complex implementations. It is introduced in section 1.1. The second topic is the process of video encoding. It is on the critical path for determining performance. It is discussed in section 1.2.

Apart from the hardware acceleration of the two algorithms discussed above, this work also investigates the processor domain. Processors are the core component in many applications and the whole embedded world. The talk of information processing will only be complete if the most used tool for them is discussed. This work focuses on the implementation of Reduced Instruction Set Computer Five (RISC-V) processors available in the literature. A detailed survey is performed to understand the different applications and optimization of processors thoroughly for video processing. The processor core parameters important for video processing are highlighted. Next, the cores are compared on the basis of these parameters

to provide a selection procedure of cores for video processing tasks. Finally, the RISC-V core called LEN5 [9–11] is analyzed in detail with an ASIC implementation. It is discussed in section 1.3.

1.1 Real-Time Discriminative Scale Space Tracker (RT-DSST)

In the domain of videos, visual object tracking is prevalent. It has been on the prominent inquest in recent times. The video is generally obtained in a sequence of frames. The primary mission of visual target tracking is to obtain the object's new location in the current frame of the video, provided the initial location in a prior frame. The major applications are video surveillance, computer vision, automation, etc. Target tracking in real-time is a computationally demanding task. The final performance of the tracker is decided by several elements like the camera's motion, scene background variations, and the object's complex motion. More sophisticated algorithms are needed to resolve these challenges by attaining reasonable accuracy. In addition, the high-quality video from cameras further amplifies the computations needed for fruitful object tracking.

Object tracking is predominantly incumbent upon software-based platforms, typically personal computers and embedded processors. But, the frame rate needs to be faster in these cases to be applicable for real-time applications like accident prevention systems, defense, etc. Furthermore, the variations in scale and the need for multi-target object detection and tracking hinder the use of this serial method for data-centric applications. Thus, massive advancement at the algorithmic level is needed, along with implementations for building real-time, standalone visual tracking devices for high mission-critical systems. FPGA is highly feasible for such applications, as it has massive parallel processing architecture and interfaces with high throughput. The literature shows that significant works in this domain are on either discriminative [12–14] or generative [15, 16] approaches. The former relies on machine learning by using a filter to learn the target's location. In contrast, the later method uses statistical models of the object to get the target location. The authors in [17–19] show that the former performs better, requiring less computational effort. In this aspect, a new multi-aspect detection approach deals with the target location and

size based on two primary techniques. The first technique, joint scale space tracking, uses a 3D correlation filter. The second technique, multi-resolution tracking, uses a 2D filter at multiple resolutions, thus creating a 3D pyramid for object detection. They both are expensive in terms of computational resources, thus insufficient for hardware implementation. Recently, the authors in [7] present an approach called DSST that shows good tracking performance with a reasonable amount of complexity. They use different filters for target translation and scale estimation [7, 20]. Starting with the translation filter to obtain the estimate of the new target location, followed by a scale filter for the object size estimation with an updated location. In the end, for each frame, both filters are updated. This process is iteratively repeated for the whole video sequence. Thus, the high amount of parallelism in the DSST algorithm makes it feasible for hardware implementation. The main operations in the filter are mathematical, involving QR factorization, two-dimensional Discrete Fourier Transform (DFT2), Singular Value Decomposition (SVD), and Histogram of Gradients (HOG). The critical among the minor mathematical operations is windowing, which is the point-wise multiplication of the filter coefficients with all the frame's pixels.

Much literature is software-based, focusing mainly on performance and a minor on hardware resources for high-resolution images. It begs the need for hardware implementation of these operations targeting a complete object tracking system for a standalone device. The authors of [21] also present a survey on hardware implementations of object tracking systems. Thus this work is focused on the FPGA-based realization of the main blocks of the DSST algorithm. The author of this thesis proposes suitable implementation strategies for the core operators in [22]. It is further discussed in Chapter 2. The background of the work is provided in section 2.1. Next, the proposed architecture is presented in 2.2. Then, the results are discussed in 2.3. Finally, the conclusions are given in section 2.4.

1.2 Wiener filter in video encoding

In recent years, the demand for an open media codec has increased with the increase in internet video content. It stems from the idea that the internet is based on essential technologies like operating systems and web browsers, which are open and available to be implemented freely. Thus it urged numerous companies to obtain some

substitutes for codecs with complex and costly royalties. The target was to build a new genesis of video coding to share the videos faster, easier, and cheaply. In this spectacle, Mozilla, Netflix, Google, Cisco, and some hardware vendors like AMD and Intel, co-founded AOMedia in 2015. Three years later, in 2018, released the first draft of Alliance for Open Media Video 1 (AV1) [23, 24], which is a video codec mainly reliant on VP9 [25]. Still, it needed further modification, including several prominent advancements, mainly the complete affinity with Patent Policy [26].

The impulsive increase in the video system's quality has incremented the amount of information to be stored and transmitted. In 2021 a report from Cisco Systems showed that the monthly Internet video traffic was around 187 Exabytes [3]. It can be augmented with the help of better video compression techniques. Video encoding helps compress the videos and reduces the storage and processing of information. Thus for real-time Internet video streaming, video compression is vital. The work here involves starting from the assay of the whole AV1 codec and then focusing on a specific module depending on the "profiling" results of the AV1 Software model [27] to figure out the usage percentage of each module. The focus is turned on the module that requires more study based on usage percentage. The assay pointed towards Wiener filter [8]. It is an in-loop reconstruction filter used in video encoding. The significance of the Wiener filter in image processing is displayed in [28, 29]. It also has many applications in the video processing domain [30, 31]. The other applications of the Wiener filter involve noise reduction, speech processing, deblurring, etc. [32–35]. The Wiener filter decreases the amount of noise and helps in the removal of blurring [36, 37]. It also assists in de-convolution, noise reduction, and signal detection [38]. Finally, it recreates a degraded video frame via a non-causal filter. The author of this thesis provides an implementation of the Wiener filter in [39]. It is further discussed in Chapter 3. The background of the work is provided in section 3.1. Next, the proposed architecture is presented in 3.2. Then, the results are discussed in 3.3. Finally, the conclusions are given in section 3.4.

1.3 RISC-V survey and implementation for video processing

Digital processors have been around for a long time now. They enrich almost every single field of our lives. With time, processors have evolved based on our needs and become more reliable. Until a decade ago, they were all controlled by a few primary silicon design and manufacturing companies. The Instruction Set Architecture (ISA) was not open source, and licensing was required. The previous ISAs were incremental, meaning each new implementation, all the previous extensions need to be implemented. With the introduction of the RISC-V ISA, the game changed. Apart from being a modular ISA, it is also an open-source ISA. The base architecture is frozen and will never change. On top of that, any other extension can be implemented without implementing all previous ones. These features enhanced research on the universities' processors as they can use RISC-V without paying any license and can modify it. Hence since then, there have been many RISC-V designs available in literature like Pulp [40], Ariane [41], Berkeley Out-of-Order Machine (BOOM) [42], etc. The RISC-V ISA is extendable and can be customized. Thus, it can be adapted to cater to their needs in many applications.

Generally, the processor has a simple working principle. The blocks can be categorized into three major parts, the Front End (FE), the Back End (BE), and the memory. The program which is executed is stored as instructions in memory. The FE communicates with the memory to get these instructions and dispatch them to the BE. The BE deals with decoding and assigning the instruction to the proper unit for execution. If it is just a computational instruction, it is assigned to Arithmetic Logical Unit (ALU). The address is calculated and communicated to the memory to load or store data if it is a memory instruction. The data internally in BE are stored in register files and pipeline registers. The performance of the processor depends on the target application. A good and fast computer for one application may mean something different for another. The performance-defining parameters are operating frequency, power consumption, energy dissipation, latency, area, throughput, and reliability.

The simple 5-stage pipelined processors can show good latency with reasonable area and performance. But regarding instructions executed, they predominantly suffer from the data and structural hazards. The hazard resolution involves replicating the

functional unit required or stalling the instruction execution until the unit becomes accessible. Thus, the latency is partially or entirely masked using the forwarding technique, but the hazards are not completely eradicated. The simple 5-stage pipelined processors execute the instructions in-order, meaning they execute them in program order. In contrast, some Out-of-Order (OoO) processors execute the instructions OoO and commit them to avoid data corruption. There are two primary techniques to perform OoO execution, namely, Scoreboarding [43] and Tomasulo technique [44]. The former technique deals with keeping a scoreboard of the instructions that are being dispatched with the help of three major structures. The first one is the register status, which saves the information of the register number where the computation unit will produce the result. The current status of the executing instruction to track the instruction execution. Also, the status of the execution unit is to know when the unit will be available for accepting new instructions. So for each instruction execution, all three structures need to be updated. Also, we need to reorder the instruction before committing to hazard resolution. Thus it becomes too complex to manage the effective execution of instructions. In contrast, the Tomasulo approach comprises a reservation station unit for each execution block. It also consists of a typical data bus where the result produced is placed and broadcasted to all units. For each register, there is a unit called register status to show the availability of results. Finally, a Re-Order Buffer (ROB) at the end to commit the instructions in-order and help resolve the hazards. Thus, it is much simpler than the former approach and inherently resolves the hazards.

The other major part which all processors require is memory. The memory hierarchy allows fast answers to FE with low latency. Typically, the memory system has 1 or 2 levels of cache, and the main memory is arranged in Von-Neumann [45] or Harvard [46] architectures. The first uses a single memory for instructions and data, while the second uses two separate memories for instructions and data. Caches [47] are small and faster memories near the processor to have fast access as they exploit the space and temporal localities to feed a constant data stream.

The RISC-V processors present in the literature use the techniques mentioned in the above discussion and are built on both ASIC and FPGA technologies. The implementations are not only of the RISC-V ISA itself but are for many customized ISA based on applications. In the literature, there are also surveys on RISC-V ISA, which mainly focus on the available open-source cores and contrast them. For instance, one study covers seven open-source cores and implements them on the

same platforms for a fair comparison [48]. Another study shows the procedure of selecting a core for low-cost Internet-of-Things (IoT) applications [49]. However, the literature needs more study on other types of RISC-V cores, especially for a particular application. Thus, this work attempts to review what is available in the state-of-the-art. Mainly it attempts to be as encompassing as possible for what is available in the literature, covering articles about surveys on RISC-V, and core implementations. The cores are then analyzed for video processing tasks. In this context, the parameters concerning video processing are highlighted first. Then the cores displaying suitable performance for these parameters are detailed. Afterwards, they are divided into categories and a comparison is performed among them. This study will aid in the selection of cores for video processing applications.

This work also analyzes an implementation of an OoO RISC-V processor called LEN5 [9–11], a Tomasulo-based architecture with support for virtual memory. Apart from this, it also can be extended to support dedicated hardware accelerators. This work also performs the additional modification to LEN5. It basically adds the features missing in the core pointed out by the authors in [10]. The rest of the discussion on this topic is in Chapter 4. First, section 4.1 provides the literature overview and details the solutions available in the literature. Then, section 4.2 details the comparison between them. Next, in section 4.3 the selected core LEN5 is analyzed and additions are provided. Finally, the conclusions are given in section 4.4.

Chapter 2

Implementation of Real-Time Discriminative Scale Space Tracking Algorithm

2.1 Introduction to the FDSST algorithm

Part of this work is published in the article by the author of this thesis in [22]. This section discusses the particulars of the object tracking algorithm FDSST [7]. In addition, the involved mathematical operations are also introduced. The section also reviews the implementations available in the literature for the mathematical operations involved in FDSST, highlighting the parallelism in the algorithms for the involved operations. The flow of the FDSST algorithm is shown in figure 2.1 while the computational steps are presented in the algorithm 1. At the start, the input image (first frame) is taken as input, along with the target's initial location and the target's initial scale. First, an image patch f is obtained from the input, centered around the target location I. This extraction is performed with the help of a HOG feature extractor with 4×4 cells. The extractor size is set to the initial target size. The HOG extractor outputs image features that are used for learning the translated location of the target utilizing a Discriminative Correlation Filter (DCF). Since the extracted f is arbitrary in domain dimension, the same filter can be used to translate the scale. Let's consider H^l denote a filter whose correlation error between the desired output

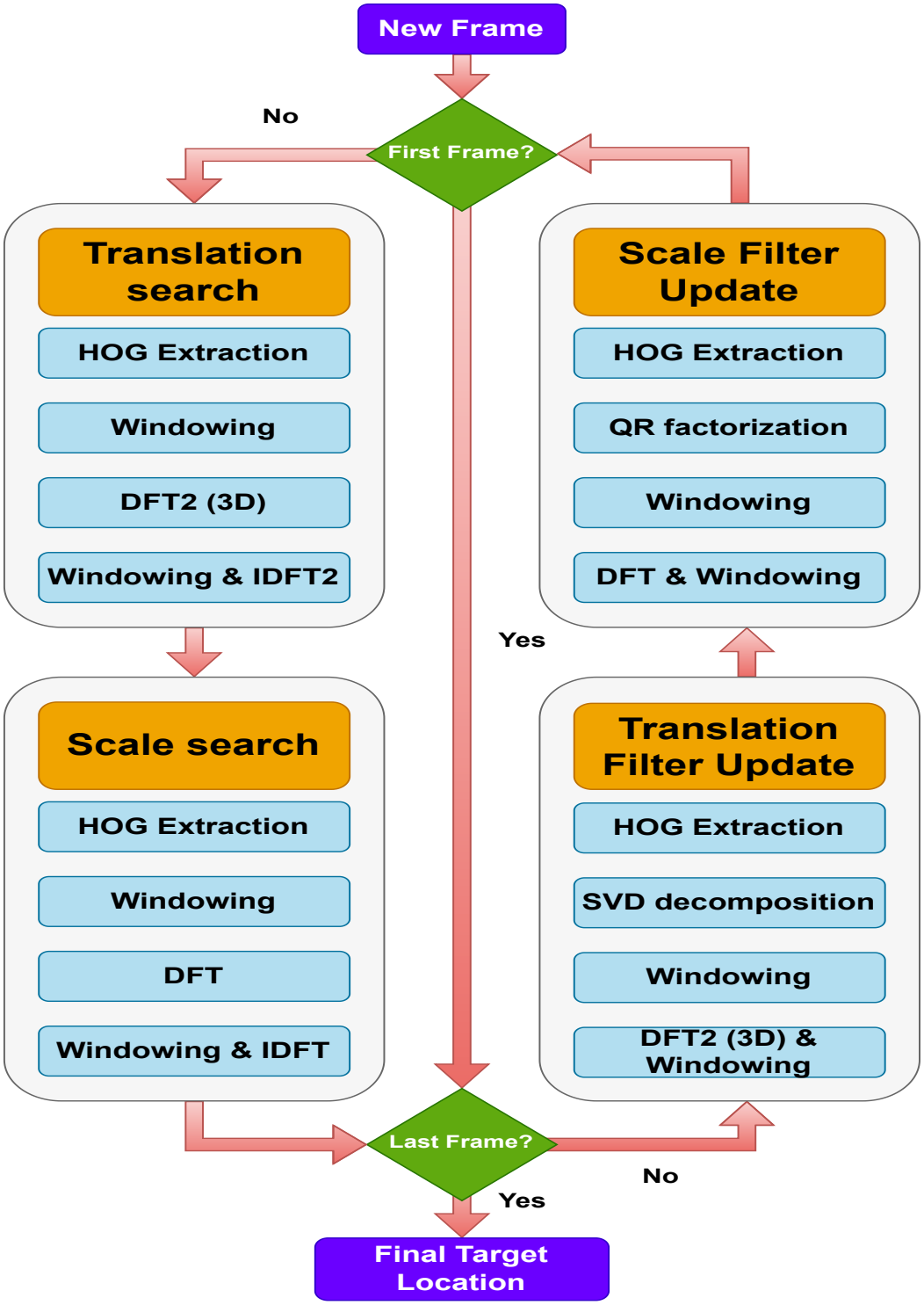


Fig. 2.1 Flow chart of the FDSST algorithm [7, 22].

and the extracted image patch is minimum. The derivation of the filter equation (2.1) is detailed in [7]. The overall filter equations are provided as,

Algorithm 1 FDSST algorithm [7, 22]

Inputs: Image I_t, Prior target position p_{t-1} and scale s_{t-1},
Translation model $A_{t-1,trans}$, $B_{t-1,trans}$,
Scale model $A_{t-1,scale}$, $B_{t-1,scale}$

Outputs: Estimated target position p_t and scale s_t,
Updated translation model $A_{t,trans}$, $B_{t,trans}$,
Updated scale model $A_{t,scale}$, $B_{t,scale}$

for all *frames* t **do**
  if $t \neq 1$ **then**
    Translation estimation:
    Extract $z_{t,trans} \leftarrow I_t$ at $\{p_{t-1}, s_{t-1}\}$ using HOG extractor
    $Z_{t,trans} \leftarrow z_{t,trans}$ using DFT2 and compute correlation scores $y_{t,trans}$ using equation (2.2)
    Set $p_t = \max\{y_{t,trans}\}$
    Scale estimation:
    Extract $z_{t,scale} \leftarrow I_t$ at $\{p_t, s_{t-1}\}$ using HOG extractor
    $Z_{t,scale} \leftarrow z_{t,scale}$ using DFT2 and compute correlation scores $y_{t,scale}$ using equation (2.2)
    Set $s_t = \max\{y_{t,scale}\}$
  end if
  Model update:
  Extract $f_{t,trans} \leftarrow I_t$ at p_{t-1} , $f_{t,scale} \leftarrow p_t$ at s_{t-1} using HOG extractor
  Using SVD compute $P_{t,trans}$, calculate DFT2 of $u_{t,trans}$ and $f_{t,trans}$ then update the translation model $A_{t,trans}$, $B_{t,trans}$ using equations (2.3) and (2.4)
  Using QR compute $P_{t,scale}$, calculate DFT of $u_{t,scale}$ and $f_{t,scale}$ then update scale model $A_{t,scale}$, $B_{t,scale}$ using equations (2.3) and (2.4)
end for

$$H^l = \frac{\overline{G}F^l}{\sum_{k=1}^{d}\overline{F^k}F^k + \lambda}, \qquad l = 1,\ldots,d \tag{2.1}$$

$$Y_t = \frac{\sum_{l=1}^{\tilde{d}}\overline{\tilde{A}_{t-1}^l}\circ\tilde{Z}_t^l}{\tilde{B}_{t-1}+\lambda}, \qquad \forall\, t \tag{2.2}$$

$$\tilde{A}_t^l = \overline{G}\circ\tilde{U}_t^l, \qquad l = 1,\ldots,\tilde{d} \tag{2.3}$$

Table 2.1 Description of the symbols in the DSST algorithm [7, 22]

Symbol	Description
$\bar{}$	Complex conjugate of the quantity
$\tilde{}$	Quantity with compressed dimensions
λ	Regularization parameter
η	Learning rate
$\circ$	Element-wise multiplication
F	Input image extracted features
l	Feature channel dimension
d	Feature length dimension
G	Gaussian function for the correlation filter output
Y_t	Correlation scores
A^l	Numerator of the correlation filter
B_t	Denominator of the correlation filter
Z^l	Image features extracted from new location
P	Projection matrix using PCA
U^l	Iterative compression of features$f$

$$\tilde{B}_t = (1-\eta)\tilde{B}_{t-1} + \eta \sum_{k=1}^{\tilde{d}} \overline{\tilde{F}_t^k} \circ \tilde{F}_t^k, \qquad \forall\, t \tag{2.4}$$

The Fourier transform of the involved quantities is denoted by capital letters. The rest of the quantities are described in Table 2.1. Equation (2.2) is the last version of the equation that defines the correlation filter. Since this method is iterative, the filter is updated for every new frame. The equation (2.3) updates $\tilde{A}_t^l$, the numerator of the filter while equation (2.4) updates $\tilde{B}_t$, the denominator of the filter. As suggested by the authors in [7], standard Principal Component Analysis (PCA) is performed to compress the dimensions. The compressed dimensions help reduce the Discrete Fourier Transform (DFT) sizes, thus realizing the fast DSST. The mathematical aspect of PCA can be achieved by attaining a template as,

$$u_t = (1-\eta)u_t + \eta f_t$$

Using a low-dimension subspace of the features P and performing the windowing with the quantities will produce the *tilde* terms. The eigenvalue decomposition is used on the auto-correlation matrix of u_t to obtain the compressed dimensions. This eigenvalue decomposition, as shown in algorithm 1, is accomplished with QR and

SVD. This step completes the learning or training of the filter. Object detection or the filter usage component is completed by calculating the correlation scores using equation (2.2). The HOG extractor extracts the updated sample z_t using the estimated 2D target location. The translation estimation is completed by performing the inverse transform of Y_t as,

$$y_t = \mathscr{F}^{-1}\{Y_t\}$$

and selecting the one with the maximum correlation score. For the scale estimation part, repeat the above-performed steps for a 1-dimensional scale filter. This estimation is performed with the new target location from 2D estimation. Thus, seeking the scale of the target in an estimated 3D location saves computations. The scaling, translation, filter estimation, and update process in the algorithm are described in (2.1), (2.2), (2.3), and (2.4). As mentioned in section 1.1, the primary mathematical operations of the DSST algorithm are DFT2, QR, SVD, and HOGs. These operations are discussed next.

2.1.1 Discrete Fourier Transform

The DFT transforms an array of uniformly spaced samples of a function into an exact length array of uniformly spaced samples of the discrete-time Fourier transform, a complex function of frequency. The length of the DFT is the reciprocal of the duration of the input array. The well-known equations defining the synthesis and analysis of DFT are given;

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{j2\pi}{N}kn} \tag{2.5}$$

$$x_k = \frac{1}{N}\sum_{n=0}^{N-1} X_n e^{\frac{j2\pi}{N}kn}$$

respectively. Where $W_N^{nk} = e^{-\frac{j2\pi}{N}kn}$ is the twiddle factor and X_n and x_n are complex numbers. The twiddle factor can also be expressed as;

$$W_N^{nk} = \cos(\frac{2\pi n}{N}) - j.\sin(\frac{2\pi n}{N})$$

by adopting Euler's identity.

$$X_k = \sum_{n=0}^{N-1} x_n \left[\cos(\frac{2\pi n}{N}) - j.\sin(\frac{2\pi n}{N}) \right] \tag{2.6}$$

The direct implementation of an N-point DFT unit has operational complexity of the order O(N^2). An improvement on this is a hardware-friendly method called Fast Fourier Transform (FFT) that scales down the DFT operations to O($Nlog_2N$), using the famous Decimation in Time (DIT) and Decimation in Frequency (DIF) techniques [50]. In literature, there are many implementations of the FFT, including parallel [51] and serial approaches [52]. However, the FFT approach requires the condition $N = 2^n$, which is not fixed in this case; thus, instead of FFT, DFT is implemented in this work. Coordinate Rotation Digital Computer (CORDIC) algorithm-based hardware implementation for DFT is provided in [53] where the authors use CORDIC for the twiddle factor calculation. At the same time, the approach in this thesis uses pre-calculated twiddle factors, thus improving performance.

The DFT2 is the 2-dimensional DFT performed on a matrix that is computed using 1-D DFTs. First, apply DFT along the matrix rows and transpose the results. Next, apply DFT along the matrix columns and again transpose the results, which produces the output. The central part of the literature uses 2D FFT to compute DFT2, while this work uses a modified row-column decomposition approach shown by the authors in [54].

2.1.2 QR factorization

QR factorization has many applications in digital signal processing, such as Multiple-Input–Multiple-Output (MIMO) communication, beamforming, echo cancellation, channel equalization, and noise cancellation. The main computational requirement of these applications primarily is QR factorization [55]. QR factorization decomposes a matrix A with row and column dimensions of $m \times n$ into a product of two matrices, the first orthogonal matrix Q of dimensions $m \times m$ and the second upper triangular matrix R of dimensions $m \times n$ [55]. Mathematically, it is given by,

$$A = QR$$

An example of matrices A, Q and R are,

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \\ a_{51} & a_{52} & a_{53} & a_{54} \end{bmatrix}$$

$$Q = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} & q_{15} \\ q_{21} & q_{22} & q_{23} & q_{24} & q_{25} \\ q_{31} & q_{32} & q_{33} & q_{34} & q_{35} \\ q_{41} & q_{42} & q_{43} & q_{44} & q_{45} \\ q_{51} & q_{52} & q_{53} & q_{54} & q_{55} \end{bmatrix}$$

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ 0 & r_{22} & r_{23} & r_{24} \\ 0 & 0 & r_{33} & r_{34} \\ 0 & 0 & 0 & r_{44} \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

In the literature, there are three primary methods for the calculation of QR factorization, namely, Gram-Schmidt approach [56], Householder transformation [57], an approach based on Givens rotation [55, 58]. The Givens method is by far the most used one and is also the one used in this thesis work. It starts by applying the Givens rotations to the lower triangle elements of A and turning them to zero. Continue this rotation until all of the lower elements of A have turned to zero; the result is matrix R. Consequently, applying the same transformation to an identity matrix I in parallel gives the matrix Q^T. The Givens rotation matrix is of the form,

$$G = \begin{bmatrix} c & s \\ -s & c \end{bmatrix},$$

where the elements c and s are given by,

$$c = \frac{a}{\sqrt{a^2 + b^2}}$$

$$s = \frac{b}{\sqrt{a^2 + b^2}}$$

a represents the first element of the row-pair, and b represents the second element, which has to be turned to zero below a. In the matrix A for the rows 5 and 6, a = a_{41} and b = a_{51}. A systolic array-based implementation is proposed in [55]. The solution presented in this thesis is similar, but the focus is on resource optimization rather than performance. The thesis also employs a row-parallel approach, improving performance by offering more parallelism.

2.1.3 Singular Value Decomposition

SVD has many applications in digital signal processing, image processing, speech synthesis, pattern recognition, and biomedical engineering [59]. SVD is the eigenvalue decomposition of a matrix A with rows and columns dimensions $m \times n$ into three matrices U, S, and V. The left eigenvectors matrix U has dimensions $m \times m$, the right eigenvectors V has dimensions $n \times n$, and the diagonal matrix S contains real eigenvalues has dimension $m \times n$. Mathematically, it is given by,

$$A = USV^T$$

An example of matrices A, U, S and V are,

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \\ a_{51} & a_{52} & a_{53} & a_{54} \end{bmatrix}$$

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} & u_{15} \\ u_{21} & u_{22} & u_{23} & u_{24} & u_{25} \\ u_{31} & u_{32} & u_{33} & u_{34} & u_{35} \\ u_{41} & u_{42} & u_{43} & u_{44} & u_{45} \\ u_{51} & u_{52} & u_{53} & u_{54} & u_{55} \end{bmatrix}$$

$$S = \begin{bmatrix} s_{11} & 0 & 0 & 0 \\ 0 & s_{22} & 0 & 0 \\ 0 & 0 & s_{33} & 0 \\ 0 & 0 & 0 & s_{44} \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$V = \begin{bmatrix} v_{11} & v_{12} & v_{13} & v_{14} \\ v_{21} & v_{22} & v_{23} & v_{24} \\ v_{31} & v_{32} & v_{33} & v_{34} \\ v_{41} & v_{42} & v_{43} & v_{44} \end{bmatrix}$$

In the literature, there are many methods for calculating SVD, namely, the one-sided Hestenes–Jacobi method, the QR algorithm, and an approach based on the two-sided Jacobi method. Among them, the two-sided Jacobi approach for parallel implementations, which is crucial for a hardware solution, provides better numerical accuracy [59]. Thus, SVD in hardware is predominately implemented with the two-sided Jacobi method [60]. It divides the whole matrix into small 2×2 matrices. Then applying the Jacobi rotations to the 2×2 sub-blocks of matrix A from left and right turn them to zero. Continue this rotation until all of the non-diagonal elements of A have turned to zero; the result is matrix S. Consequently, applying the same transformation to two identity matrices I in parallel gives the matrices U and V. The Jacobi rotation matrix is of the form,

$$u_{2\times 2} = \begin{bmatrix} c1 & -s1 \\ s1 & c1 \end{bmatrix}$$

$$v_{2\times 2} = \begin{bmatrix} c2 & s2 \\ -s2 & c2 \end{bmatrix}$$

$$a_{2\times 2} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

Thus, by using two-sided rotation,

$$\begin{bmatrix} c1 & -s1 \\ s1 & c1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} c2 & s2 \\ -s2 & c2 \end{bmatrix} = \begin{bmatrix} s_{11} & 0 \\ 0 & s_{22} \end{bmatrix},$$

where the elements c and s are given by,

$$c = \cos(\theta)$$

$$s = \sin(\theta)$$

Finally, theta is given by,

$$\theta_1 = \frac{1}{2}\arctan\frac{a_{12}+a_{21}}{a_{22}-a_{11}}$$

$$\theta_2 = \frac{1}{2}\arctan\frac{a_{12}-a_{21}}{a_{22}+a_{11}}$$

The final matrix obtained is $s_{2\times 2}$. In literature, fixed point-based implementations of the SVD are provided in [60, 61]. This thesis work uses a similar approach, focusing on time optimization. This unit has smaller dimensions in the FDSST algorithm, so it is operated in parallel at the cost of more area.

2.1.4 Histogram of Gradients

The HOG is a feature descriptor used in the recognition of targets. It is obtained with assistance from image gradients and extracts features contained in image pixels. It is a very computationally intensive task, and many approaches are available for its implementation in literature [62–64]. First, the authors of [62] provide a comparative analysis of HOG implementations. Next, an implementation in [63] with low resource consumption is presented but has a lower operating frequency. Finally, the authors in [64] present a solution with a low area cost and a high frame rate of 60 frames per second (fps). The improvements are achieved using integer multiplication and inequality comparisons to replace critical angle computation.

Starting from the initial target location, an image patch around the target is used to extract features. The patch is divided into coarser cells, and a histogram of oriented gradients is computed for the pixels within the cells. This is accomplished by first calculating the difference in pixel values from the neighboring ones in horizontal and vertical directions. Then, an image gradient's direction and magnitude are computed. The magnitude is distributed among the nine available bins based on orientation ranging from 0° to 180° or 0° to 360°. The detection window consists of 8×8 pixels of non-overlapping cells thus, the summation is performed using

the aggregate module of 64 pixels for each bin, and the histogram is obtained. Finally, normalization is performed that allows invariance to changes in illumination. These gradients help describe object appearance and shape within the frame via the distribution of gradient intensities. The detailed algorithm is presented in [64].

2.2 Proposed architectures

This section details the implemented architectures of the mathematical operations in FDSST described in the previous section. The flow of the FDSST process is shown in figure 2.1. It consists of four main steps. First, the 2-dimension (2-D) location of the target is searched in the translation search step. Next, the 1-dimension (1-D) scale search is performed on the updated target 2D location. Then, the translation filter is updated based on this frame's data. Finally, the scale filter is updated for the new frame. These steps are repeated for all the frames.

In the translation search stage, feature extraction is performed via the HOG extractor that takes the window around the target and outputs a 3-dimensional matrix. The extracted features are passed via a windowing phase and fed to the DFT unit. Next, the DFT2 of the 3D matrix is performed, which makes this step computationally expensive, and the output of the DFT block is passed again via a windowing phase. Then, the inverse DFT of the matrix is performed, and the maximum correlation output is selected as the new location. Finally, this new location is fed to the scale search.

In the scale search stage, similar to the translation search, the feature extraction is performed via the HOG extractor but for one dimension. First, the extracted features are passed via a windowing phase and fed to the DFT unit. Next, the DFT of the scale array is performed, and the output of the DFT block is passed again via a windowing phase. Then, the inverse DFT of the matrix is performed, and the maximum correlation output is selected as the new scale. Thus, the new location and scale of the target are tracked. The procedure is stopped if it is the last frame; otherwise, the data is fed to filters.

The feature extraction is performed via the HOG extractor in the translation filter update stage. First, the extracted features are sent to the SVD calculation unit, which is a computationally expensive operation. Next, the output of the SVD is passed via

a windowing phase and fed to the DFT unit. Next, the DFT2 of the 3D matrix is performed, which, again, as mentioned above, is a computationally expensive task. Finally, the output of the DFT block is passed again via a windowing phase. This data is used to update the interpolation and is fed to the scale filter update search.

In the scale filter update stage, similar to the translation update, feature extraction is performed via the HOG extractor, and the extracted features are sent to the QR calculation unit. The number of rows of the input matrix is the product of the target dimension, thus making it computationally expensive. The output of the QR is passed via a windowing phase and fed to the DFT unit. The output of the DFT block is passed again via a windowing phase. This data is used to update the scale interpolation. This step completes the search in one frame.

The computationally expensive blocks in the translation search, i.e., the 2D position of the target and translation filter update steps, involve HOG extraction and DFT2 of 3D matrices with HOG also being expensive and used in all four phases. The scale search and scale filter update involve HOG extraction and DFT of a 2D matrix. The other expensive blocks are SVD and QR, involved in translation filter update and scale filter update steps, respectively. The DFT unit is the most critical block in terms of performance because of operation on the 3D 320×320 matrix.

Vivado High-Level Synthesis (HLS) is used as the base tool for synthesizing and simulation. It is a software tool from Xilinx named Vivado Design Suite and is developed for analyzing and synthesizing hardware designs alongside Xilinx Integrated Software Environment (ISE), with extra features for performing SOC development and, most importantly, high-level synthesis. Thus, the code written in C will be converted to an HDL design. The pragma feature in the HLS permits unrolling, pipeline, or partitioning loops and pieces of code for optimization. It is a handy tool since it controls hardware while using higher-level language.

2.2.1 Discrete Fourier Transform

In this subsection, the proposed architecture for the DFT unit for RT-DSST is presented. The vector DFT unit is an essential building block for the matrix DFT, i.e., DFT2 computation, which is used for 3D matrices as well. The architectures for both are discussed in detail as follows.

DFT

The proposed architecture of the basic unit of vector DFT computational unit is shown in figure 2.2. The basic unit separates the real and imaginary parts as input

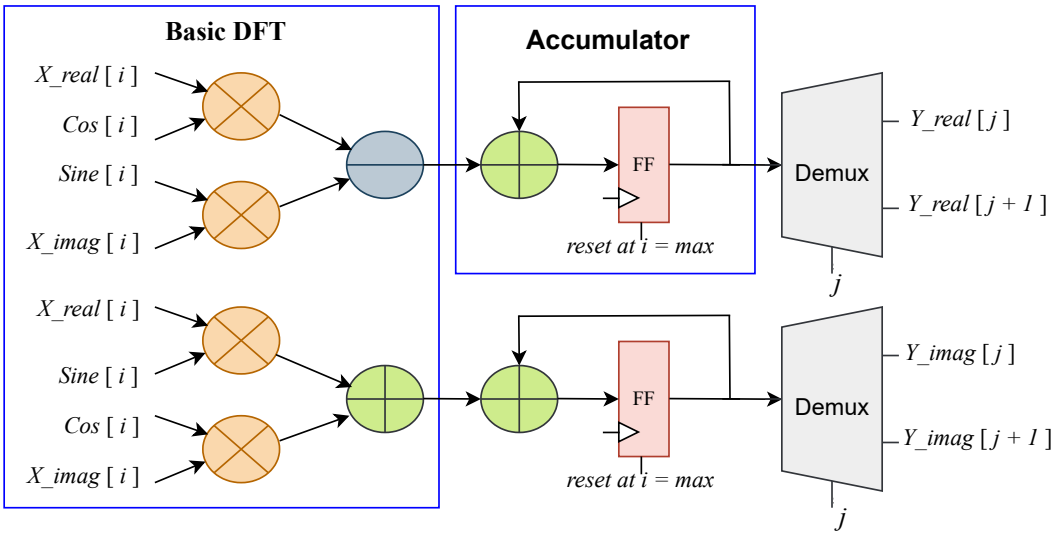


Fig. 2.2 RTL diagram of DFT basic block.

alongside the twiddle factor terms in the form of cos and sine coefficients. Start from the first cycle i the inner loop counter for the DFT, the first current samples $X\_real\,[i]$, $X\_imag\,[i]$ and the initial coefficients $cos\,[i]$ and $sine\,[i]$ are received. Two adders/subtractors with four multipliers are needed to accomplish the basic DFT block's complex multiplication. The accumulator follows this block, which has the other initial input as 0 since the register is at the reset. This step produces the real part of the results for complex multiplication, i.e.,

$$X\_real\,[i]\,.(cos\,[i]) - X\_imag\,[i]\,.(sine\,[i])$$

Similarly, the second accumulator provides,

$$X\_real\,[i]\,.(sine\,[i]) + X\_imag\,[i]\,.(cos\,[i])$$

Afterward, in the following clock cycles, the basic DFT block has the inputs $X\_real\,[i+1]$, $X\_imag\,[i+1]$ along with twiddle factors $cos\,[i+1]$ and $sine\,[i+1]$. Again, it computes the complex multiplication and passes it to the accumulator,

which produces

$$X\_real\,[i+1]\,.(cos\,[i+1]) - X\_imag\,[i+1]\,.(sine\,[i+1])$$

$$X\_real\,[i+1]\,.(sine\,[i+1]) + X\_imag\,[i+1]\,.(cos\,[i+1])$$

Finally, for the last input, when the i reaches N, the basic DFT block has inputs $X\_real\,[N]$, $X\_imag\,[N]$ along with twiddle factors $cos\,[N]$ and $sine\,[N]$. Again, it computes the complex multiplication and passes it to the accumulator, which produces

$$X\_real\,[N]\,.(cos\,[N]) - X\_imag\,[N]\,.(sine\,[N])$$

$$X\_real\,[N]\,.(sine\,[N]) + X\_imag\,[N]\,.(cos\,[N])$$

Currently, j, the output loop counter for the DFT, which is 0 till now, assigns the first output $Y\_real\,[j]$, $Y\_imag\,[j]$ using the demultiplexers. In the end, j is incremented, and the accumulator register is reset to 0. Again after another set on inner loop counter i goes through 0 to N, then the demuliplexers assign $Y\_real\,[j+1]$, $Y\_imag\,[j+1]$. This process is repeated till N outputs are produced, with the last one being $Y\_real\,[N]$, $Y\_imag\,[N]$. This approach is serial as it takes about N cycles to provide one output and meanwhile consumes one input per cycle.

Now, for fast DFT, the architecture needs to be parallelized. The higher dimensions are supported via fast parallel architecture built from the basic DFT discussed above. This parallelization is shown in figure 2.3. Modifying the equation (2.6) for supporting eight elements in parallel gives;

$$\begin{aligned} X_k = \sum_{n=0}^{N/8-1} & x_n \left[\cos(\frac{2\pi n}{N}) - j.\sin(\frac{2\pi n}{N})\right] + x_{n+1} \\ & \left[\cos(\frac{2\pi(n+1)}{N}) - j.\sin(\frac{2\pi(n+1)}{N})\right] + x_{n+2} \\ & \left[\cos(\frac{2\pi(n+2)}{N}) - j.\sin(\frac{2\pi(n+2)}{N})\right] + \cdots + \\ & x_{n+7}\left[\cos(\frac{2\pi(n+7)}{N}) - j.\sin(\frac{2\pi(n+7)}{N})\right] \end{aligned} \tag{2.7}$$

where x_n is complex. For each one of the complex multiplication, i.e.,

$$C_n = x_n.\left[\cos(\frac{2\pi n}{N}) - j.\sin(\frac{2\pi n}{N})\right]$$

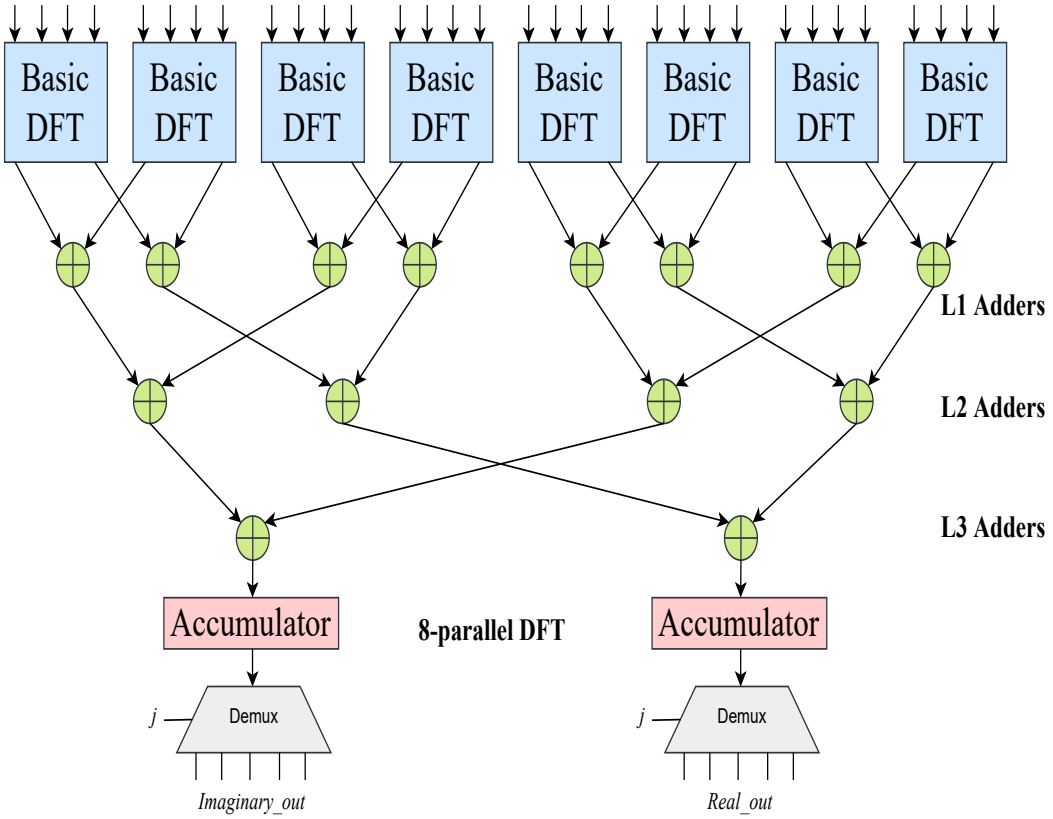


Fig. 2.3 RTL diagram of DFT 1D unit.

is obtained from the basic DFT unit. The outputs of basic DFT are added by the L1 adders in the pairs of $\{C_n, C_{n+1}\}, \{C_{n+2}, C_{n+3}\}, \{C_{n+4}, C_{n+5}\}, \{C_{n+6}, C_{n+7}\}$. Consider that $M_n = C_n + C_{n+2}$ be the output of L1 adders, then L2 adders add the pairs $\{\{M_n, M_{n+2}\}, \{M_{n+1}, M_{n+3}\}\}$. Consider that $P_n = M_n + M_{n+2}$ be the output of L2 adders, then L3 adders add the pairs $\{\{P_n, P_{n+2}\}, \{P_{n+1}, P_{n+3}\}\}$. This step produces the 8-point DFT. The accumulator keeps summing the DFTs until N inputs are processed. Similarly to the serial approach, demultiplexers assign the output after the last input is processed, j is incremented, and accumulators are reset. Hence, with a tree of adders between the basic unit and the accumulator, the 8-parallel DFT is computed. The DFT coefficients are calculated beforehand and stored in the memory to avoid complex computations at runtime. The HLS compiler mostly handles the intermediate computation results. But in some cases, the intermediate values Block RAMs (BRAM)s are instantiated and partitioned so the elements can be processed in parallel. This unit's input and output arrays are partitioned into 8 BRAMS for eight elements to be accessed in parallel. The DFT has a $O(N^2)$ delay. The proposed

architecture improves it, too,

$$delay = O(\frac{N^2}{8}) + pipelinestages \approx O(\frac{N^2}{8})$$

The maximum values of $N = 320$ are used for the implemented architecture. If $N < 320$, a comparator limits the traverses from the DFT block. For further enhancing performance, task-level parallelism is used. It is achieved with the help of Vivado HLS dataflow pragma.

DFT2

The row-column decomposition-based proposed architecture for the DFT2 is shown in figure 2.4. The real and imaginary parts are kept apart, so the throughput can be enhanced by running the operations in parallel. First, it inputs the 2D matrix, and for the first round, mux selects the rows for sending to the DFT calculation blocks, while the coefficients are pre-computed and saved in BRAM blocks. An 8-parallel unit is used for the DFT computation here, and the output is sent to the transpose unit and next to the memory. Finally, the mux selects the columns for the DFT computations for the next round. Thus, it performs the column-wise DFT. The delay of the transpose unit is proportional to N. It is given by,

$$O(N^2 - \sum_{i=1}^{N-1} i))$$

The parallel implementation allows it to achieve,

$$O(\frac{N^2}{8} - \sum_{i=1}^{\frac{N}{8}-1} i))$$

and the total delay is,

$$2M \times (T_{DFT} + T_{Trans}) \approx 2M \times (T_{DFT})$$

where M is the number of rows of the matrix. Since DFT2 lies on the critical path, to reduce the delay, using twice the hardware, the maximum delay is halved, that is

$$2M \times (T_{DFT}) \approx M \times (T_{DFT})$$

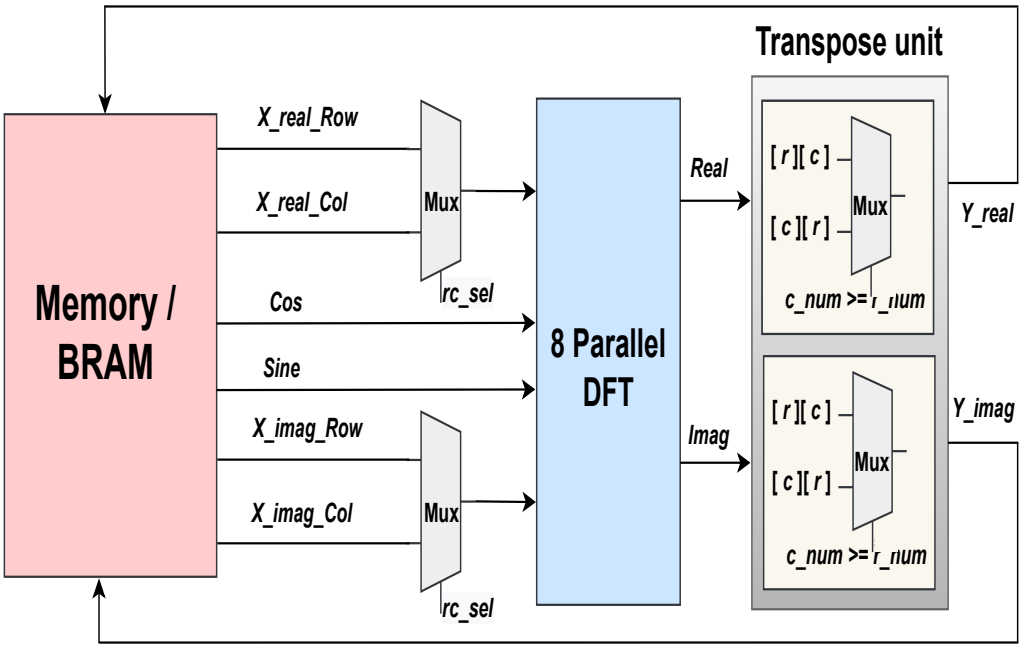


Fig. 2.4 RTL diagram of DFT2 unit.

The DFT2 was synthesized for a maximum size of 320×320. The same input matrix was used to store the results, thus saving the BRAM resource. DFT2 acts as the basic block for the DFT2 (3D). If P is the value of the third dimension, i.e., the number of 2D matrices, the maximum delay of this unit is given by,

$$P \times (T_{DFT2}) \approx P \times M \times (T_{DFT})$$

The maximum value in FDSST of P is 18. For the inverse DFT, the same DFT2 block is utilized with a divisor in the accumulator before the delay element to divide it by N.

2.2.2 QR factorization

In this subsection, the proposed architecture for the QR unit for RT-DSST is presented. The Givens rotation method [55] is used for the QR unit implementation. The Vivado HLS QR factorization library [55] is utilized as a reference unit and modified for RT-DSST architecture. The implementation in this work is for real numbers and is shown in Fig. 2.5. Algorithm 2 features the primary computational steps of QR

factorization. It consists of two major blocks: Givens matrix generation and rotation units. The Givens matrix generation consumes, as input, two elements from the two rows of matrix A and produces the Givens matrix as output. The input 2D matrix A and the Givens matrix G are given by;

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ \overbrace{a_{21}} & a_{22} & a_{23} & a_{14} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ \underbrace{a_{41}} & a_{42} & a_{43} & a_{44} \end{bmatrix} \; and \; G = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}, \tag{2.8}$$

where c and s are

$$c = \frac{a}{M}$$

$$s = \frac{b}{M}$$

Magnitude M is calculated as;

$$M = \sqrt{a^2 + b^2} = x \times \sqrt{1 + y \times y}, \tag{2.9}$$

where x and y are

$$x = \max(a, b)$$

$$y = \frac{\min(a, b)}{x}$$

The Givens matrix generation and rotation blocks are displayed in figures 2.6 and 2.7, respectively. Consider that if $a = a_{31}$ and $b = a_{41}$, using equation (2.9) matrix G is computed. The implementation of equation (2.9) is shown in figure 2.8, which is the magnitude calculator. It starts by considering the maximum using a comparator and two multiplexers, and the y is obtained via division. Finally, M is calculated using a combination of an adder, multiplier, and square root unit. To obtain matrix G, the terms c and s are determined. As the magnitude is in the denominator, the division by zero check is performed using a comparator and a multiplexer. For matrix R, turn the lower triangular elements of matrix A to zero by applying the rotation. Hence, the Givens rotation is applied in the following manner,

$$G \cdot \begin{bmatrix} a_{31} & a_{41} \end{bmatrix}^T = \begin{bmatrix} a_{31}* & 0 \end{bmatrix}^T \tag{2.10}$$

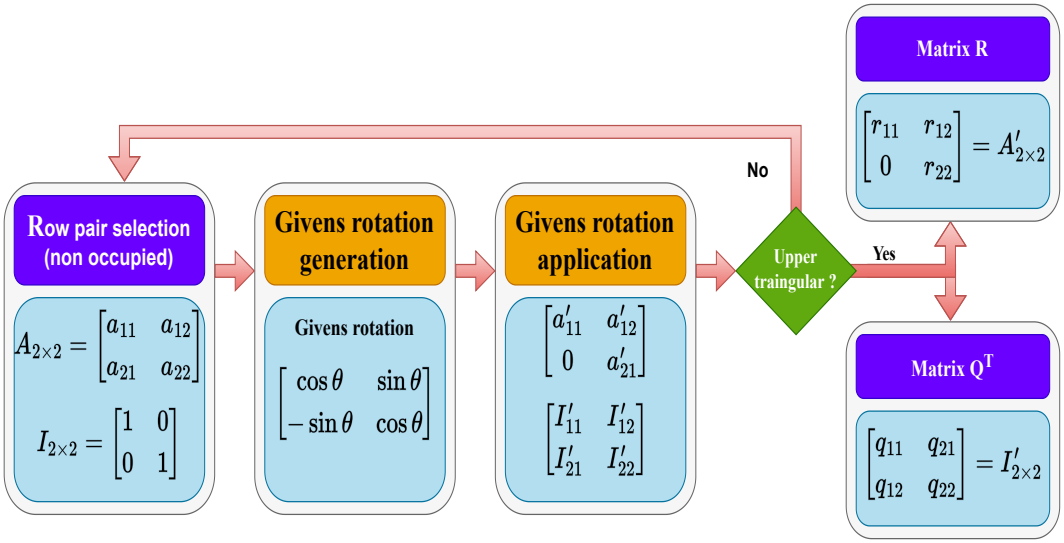


Fig. 2.5 QR factorization using Givens rotations.

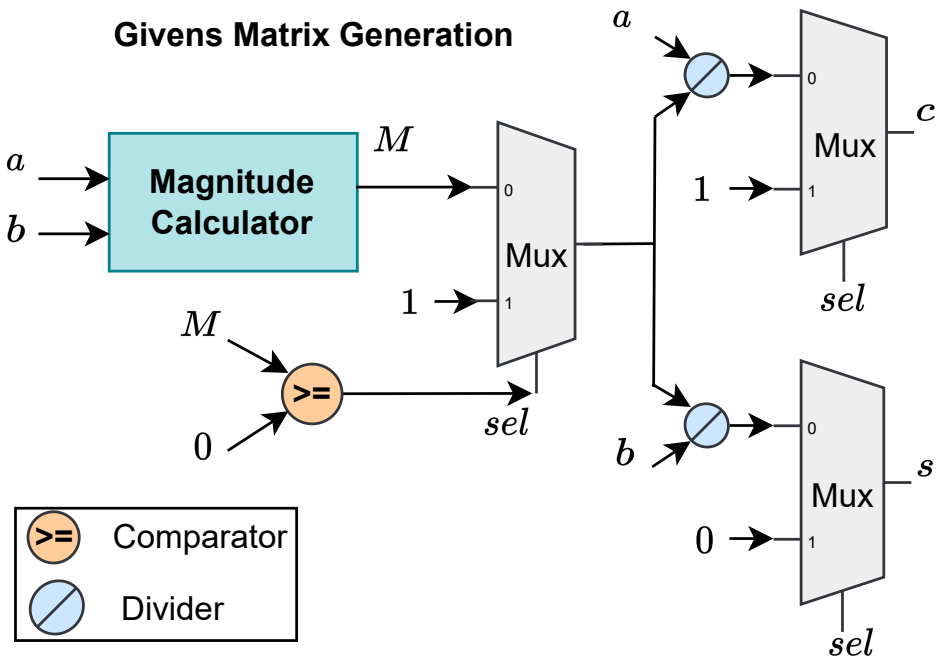


Fig. 2.6 Givens Matrix Generation.

The Givens rotation block is in figure 2.7. It consists of a simple 2×2 matrix to vector multipliers. If the second element b is already zero, Givens rotation is $a = M$

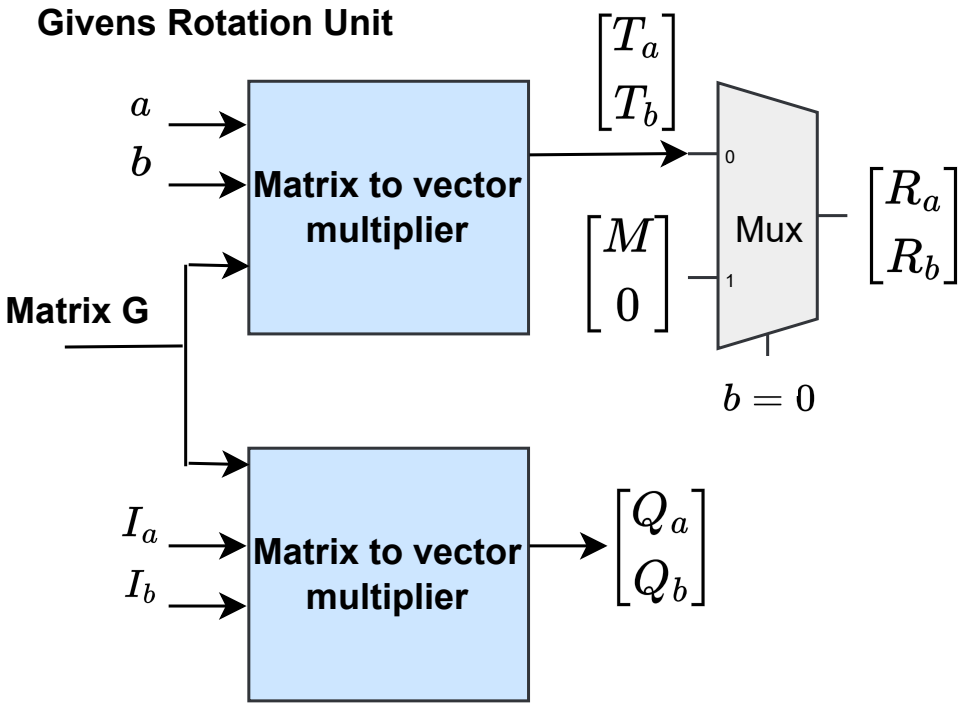


Fig. 2.7 Givens Rotation Unit.

and $b = 0$, and the multiplexers are used for assignments. A comparator accompanies Givens rotation unit to avoid assigning wrong values to zeroed positioned elements shown in figure 2.8. In such a case, the first element is equal to the magnitude. For the complete matrix, R now select a = a_{21} and b =$a_{31}*$ and repeat until all the lower triangular elements are turned to zero as,

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{14} \\ \overbrace{a_{31}} & a_{32} & a_{33} & a_{34} \\ \underbrace{a_{41}} & a_{42} & a_{43} & a_{44} \end{bmatrix} \xRightarrow{G} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ \overbrace{a_{21}} & a_{22} & a_{23} & a_{14} \\ \underbrace{a'_{31}} & a'_{32} & a'_{33} & a'_{34} \\ 0 & a'_{42} & a'_{43} & a'_{44} \end{bmatrix}$$

$$\xRightarrow{G} \begin{bmatrix} \overbrace{a_{11}} & a_{12} & a_{13} & a_{14} \\ \underbrace{a'_{21}} & a'_{22} & a'_{23} & a'_{14} \\ 0 & a''_{32} & a''_{33} & a''_{34} \\ 0 & a'_{42} & a'_{43} & a'_{44} \end{bmatrix} \xRightarrow{G} \begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ 0 & a''_{22} & a''_{23} & a''_{14} \\ 0 & \overbrace{a''_{32}} & a''_{33} & a''_{34} \\ 0 & \underbrace{a'_{42}} & a'_{43} & a'_{44} \end{bmatrix}$$

$$\overset{G}{\Longrightarrow} \begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ 0 & a''_{22} & a''_{23} & a''_{14} \\ 0 & a'''_{32} & a'''_{33} & a'''_{34} \\ 0 & 0 & a''_{43} & a''_{44} \end{bmatrix} \overset{G}{\Longrightarrow} \begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ 0 & a'''_{22} & a'''_{23} & a'''_{14} \\ 0 & 0 & a''''_{33} & a''''_{34} \\ 0 & 0 & a''_{43} & a''_{44} \end{bmatrix}$$

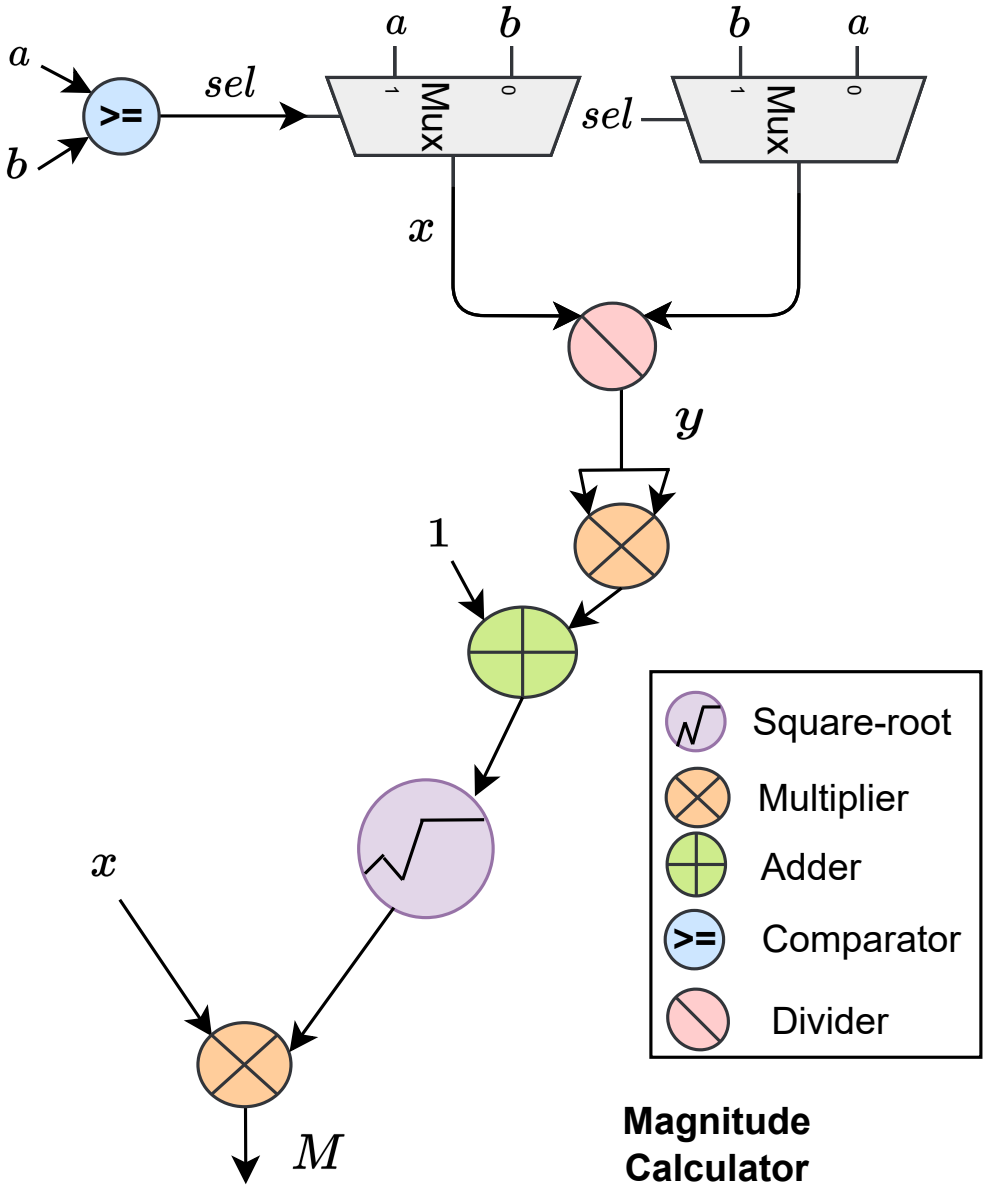


Fig. 2.8 RTL diagram of magnitude calculator for QR.

Algorithm 2 QR computation algorithm.

Inputs: Matrix: $A_{m\times n}$, Identity matrix $I_{m\times n}$
Outputs: Orthogonal matrix: $Q_{m\times m}$, Lower triangular matrix: $R_{m\times n}$

Givens rotation Generation:
for all $r, c \in A$ **do**
  if $r > c$ *and* $A[r][c] \neq 0$ **then**
    for all non overlapping $[ra, rb]$ pairs **do**
      Compute magnitude M using equation (2.9)
      Generate the matrix G using c = $\frac{a}{M}$, s = $\frac{b}{M}$
    end for
  end if
end for
Givens rotation Application:
for all $r, c \in A$ **do**
  if $r > c$ *and* $A[r][c] \neq 0$ **then**
    for all non overlapping $[ra, rb]$ pairs **do**
      for all $c \in [ca, c_{\max}]$ **do**
        Triangular matrix R computation:
        Obtain matrix R by applying equation (2.10) to [ra, rb] pairs
        Orthogonal matrix Q computation:
        Generate matrix Q by applying equation (2.10) to I
      end for
    end for
  end if
end for

$$\xrightarrow{G} \begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ 0 & a'''_{22} & a'''_{23} & a'''_{14} \\ 0 & 0 & a'''''_{33} & a'''''_{34} \\ 0 & 0 & 0 & a'''_{44} \end{bmatrix}$$

The matrix Q is obtained by performing the same generated rotations to an identity matrix I. A small address generation unit is utilized for the row selection and the row pairs are selected to be operated in parallel. The rotations are applied to all the columns of the selected rows. Operating non-occupied rows in parallel speeds up the

process. It is given as,

$$A = \begin{bmatrix} \overbrace{a_{11}} & a_{12} & a_{13} & a_{14} \\ \underbrace{a_{21}} & a_{22} & a_{23} & a_{14} \\ \overbrace{a_{31}} & a_{32} & a_{33} & a_{34} \\ \underbrace{a_{41}} & a_{42} & a_{43} & a_{44} \end{bmatrix} \xRightarrow{G} \begin{bmatrix} \overbrace{a'_{11}} & \overbrace{a'_{12}} & a'_{13} & a'_{14} \\ 0 & a'_{22} & a'_{23} & a'_{14} \\ \underbrace{a'_{31}} & a'_{32} & a'_{33} & a'_{34} \\ 0 & \underbrace{a'_{42}} & a'_{43} & a'_{44} \end{bmatrix}$$

$$\xRightarrow{G} \begin{bmatrix} a''_{11} & \overbrace{a''_{12}} & a''_{13} & a''_{14} \\ 0 & a''_{22} & a''_{23} & a''_{14} \\ 0 & \underbrace{a''_{32}} & a''_{33} & a''_{34} \\ 0 & 0 & a''_{43} & a''_{44} \end{bmatrix} \xRightarrow{G} \begin{bmatrix} a''_{11} & a''_{12} & a''_{13} & a''_{14} \\ 0 & a'''_{22} & \overbrace{a'''_{23}} & a'''_{14} \\ 0 & 0 & a'''_{33} & a'''_{34} \\ 0 & 0 & \underbrace{a''_{43}} & a''_{44} \end{bmatrix}$$

$$\xRightarrow{G} \begin{bmatrix} a''_{11} & a''_{12} & a''_{13} & a''_{14} \\ 0 & a'''_{22} & a'''_{23} & a'''_{14} \\ 0 & 0 & a''''_{33} & a''''_{34} \\ 0 & 0 & 0 & a'''_{44} \end{bmatrix}$$

A single matrix is used for input A and output R to reduce the number of resources. The critical path is the magnitude unit in the Givens generation block, which has a division and square root operation. The generated Verilog code can be modified at the RTL level to pipeline the architecture to reduce the critical path. The number of parallel rotations impacts resources and performance. Resource optimization is employed as QR is not along the critical path of the algorithm being in the scale filter update stage. The generation and rotation blocks are parallelized by a factor of two and pipelined with an Initiation Interval of 4 using the HLS pragma.

2.2.3 Singular Value Decomposition

In this subsection, the proposed architecture for the SVD unit for RT-DSST is presented. The two-sided Jacobi method [60] is used for the SVD unit implementation and the Vivado HLS SVD library [55] is utilized as a reference unit and modified for RT-DSST architecture. The implementation in this work is for real numbers and is shown in figure 2.9 while the RTL diagram is displayed in figure 2.10. Algorithm 3 features the primary computational steps of SVD execution. SVD computation consists of a Diagonal Processor (DP) and a non-diagonal processor. The DP takes

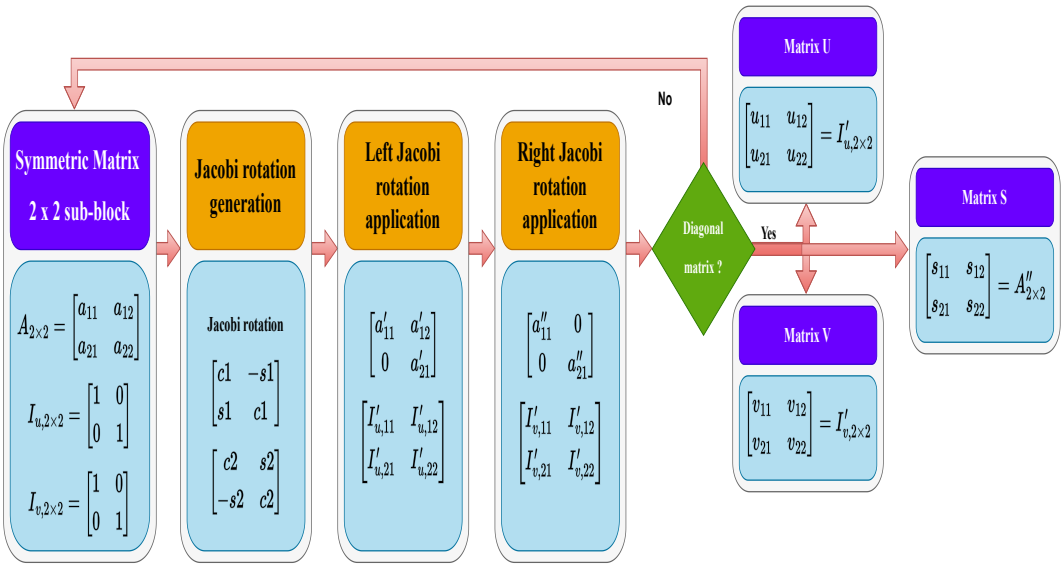


Fig. 2.9 SVD using Jacobi rotations.

as input the 2D matrix A, divided into $N/2$ 2×2 submatrices. Matrix A is the same as equation 2.8 and Jacobi left and right matrices are given by;

$$u = \begin{bmatrix} c1 & -s1 \\ s1 & c1 \end{bmatrix} \text{ and } v = \begin{bmatrix} c2 & s2 \\ -s2 & c2 \end{bmatrix},$$

where c = cos(θ) and s= sin(θ). For θ we have θ_1 and θ_2 as shown in [60] and are given as,

$$\theta_1 = \frac{1}{2}\arctan\frac{b+c}{d-a}$$

$$\theta_2 = \frac{1}{2}\arctan\frac{b-c}{d+a}$$

to avoid calculation of arctan consider;

$$\tan(2\theta) = \frac{b+c}{(d-a)} \tag{2.11}$$

The angle unit implementation is displayed in figures 2.11, 2.12, and 2.13. A divisor displayed in the figures generates tan(2θ). Then, with the help of trigonometric

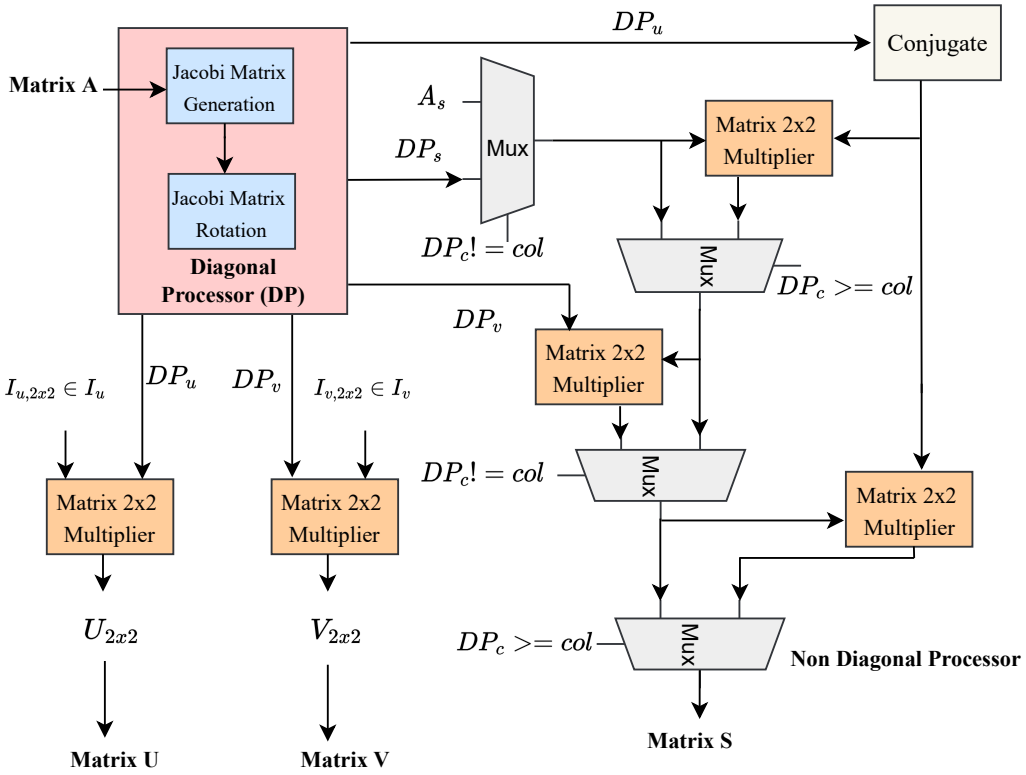


Fig. 2.10 RTL diagram of 2×2 SVD.

identities, cos and sin are derived. They are as under;

$$\begin{aligned} \cos(\theta) &= \frac{1}{\sqrt{1+(\tan^2(\theta))}} \\ \sin(\theta) &= \cos(\theta).\tan(\theta) \\ \tan(\frac{\theta}{2}) &= \frac{(1-\cos(\theta))}{\sin(\theta)} \end{aligned} \tag{2.12}$$

The computation of half-angle and full-angle identities are shown in figure 2.11 and 2.12, respectively. Angle calculator is shown on the right of figure 2.13, which compromises a tree of multiplexers to select the correct angle based on whether the number is real, imaginary, or complex.

Figure 2.14 shows the Jacobi matrices generation. The numerator and denominator of the equation (2.11) are calculated via adder and subtractor and sent to the angle calculation unit. The Jacobi matrices are generated using $c = \cos(\alpha \mp \beta)$ and

$s = \cos(\alpha \pm \beta)$ identities. The half-angle quantities calculated before are the α and β and are implemented using a simple vector multiplier. The Jacobi rotations are given by;

$$\begin{bmatrix} c1 & -s1 \\ s1 & c1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} c2 & s2 \\ -s2 & c2 \end{bmatrix} = \begin{bmatrix} a'_{11} & 0 \\ 0 & a'_{22} \end{bmatrix} \tag{2.13}$$

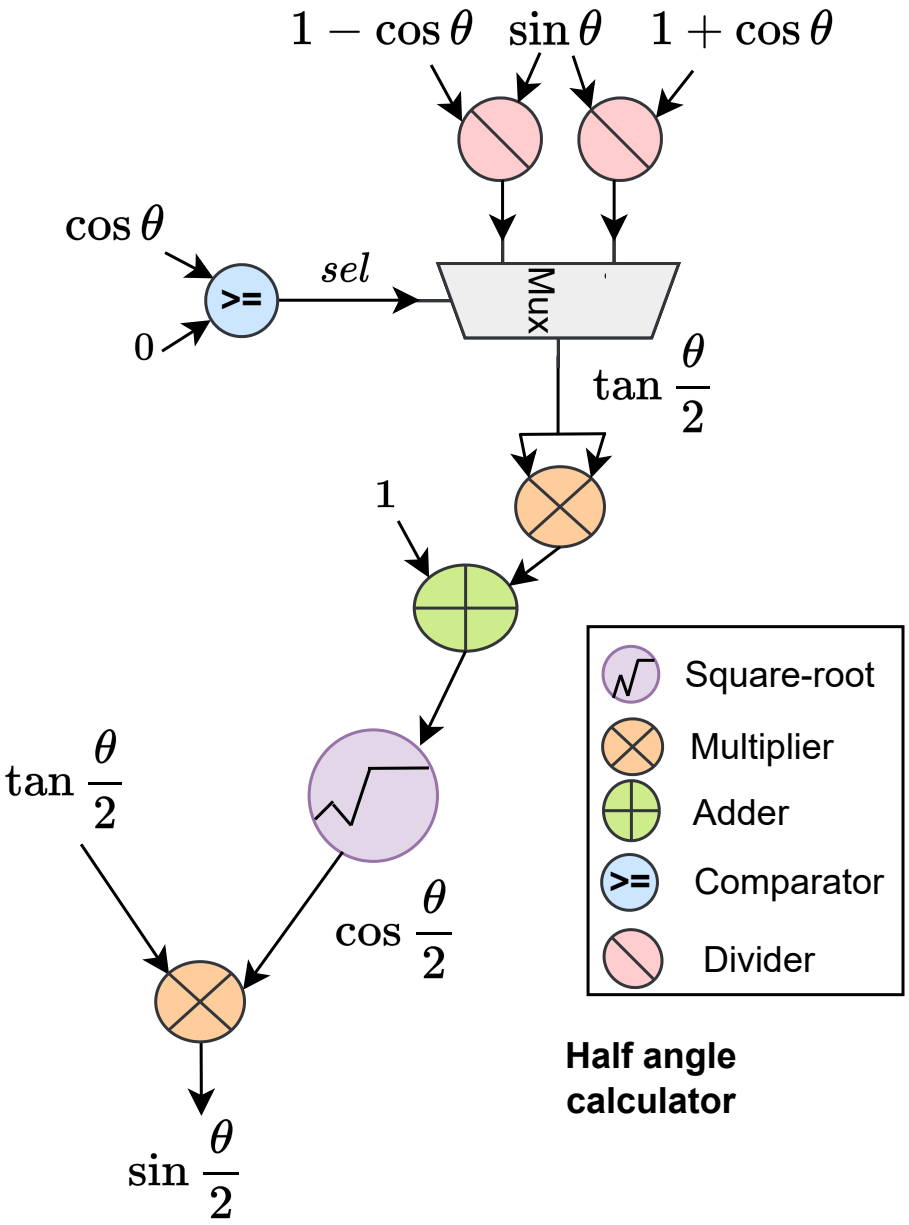


Fig. 2.11 RTL diagram of Half angle calculator.

Hence, the Jacobi rotations are just matrix multiplications realized by vector multipliers in figure 2.15.

After describing the building blocks, the discussion on the computation of U, S, and V matrices is next. As depicted in figure 2.10 SVD consists of DP and non-DP. The symbols of the quantities used are defined as A_s denotes submatrix on the main diagonal of A, while the terms with subscript DP mean newly updated submatrix from DP unit, the terms with I indicate they are from the identity matrix. Finally, DP_c and col represent 2×2 submatrix in the current iteration and selected column

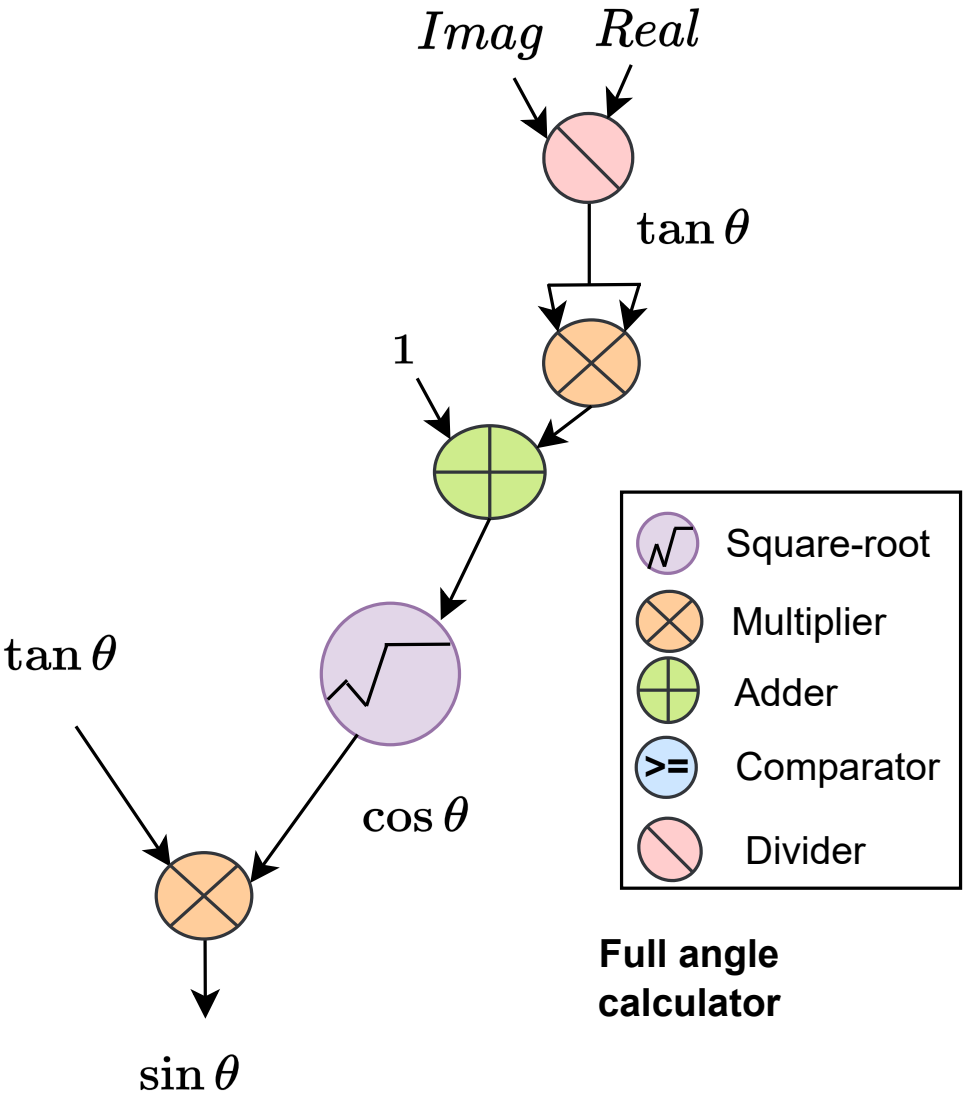


Fig. 2.12 RTL diagram of Full angle calculator.

pairs, respectively. The procedure is repeated for a certain number of iterations to obtain convergence. Literature suggests that most of the 6 to 10 iterations are sufficient. In this work, for a dimension of 32, the iteration is limited to 6. The table in [60] determines the iteration factor.

The main component diagonal processor accepts a 2×2 diagonal submatrix A_s of matrix A. DP generates the Jacobi matrices and then applies the rotations. First, A_s is rotated by DP_v, then post-multiplied by DP_u, and the non-diagonal elements of A_s are turned to zero. DP now outputs the new matrices DP_s, DP_u and DP_v to the non-DP unit. The Non-DP in figure 2.10 receives 2 identity matrices U, V and matrix

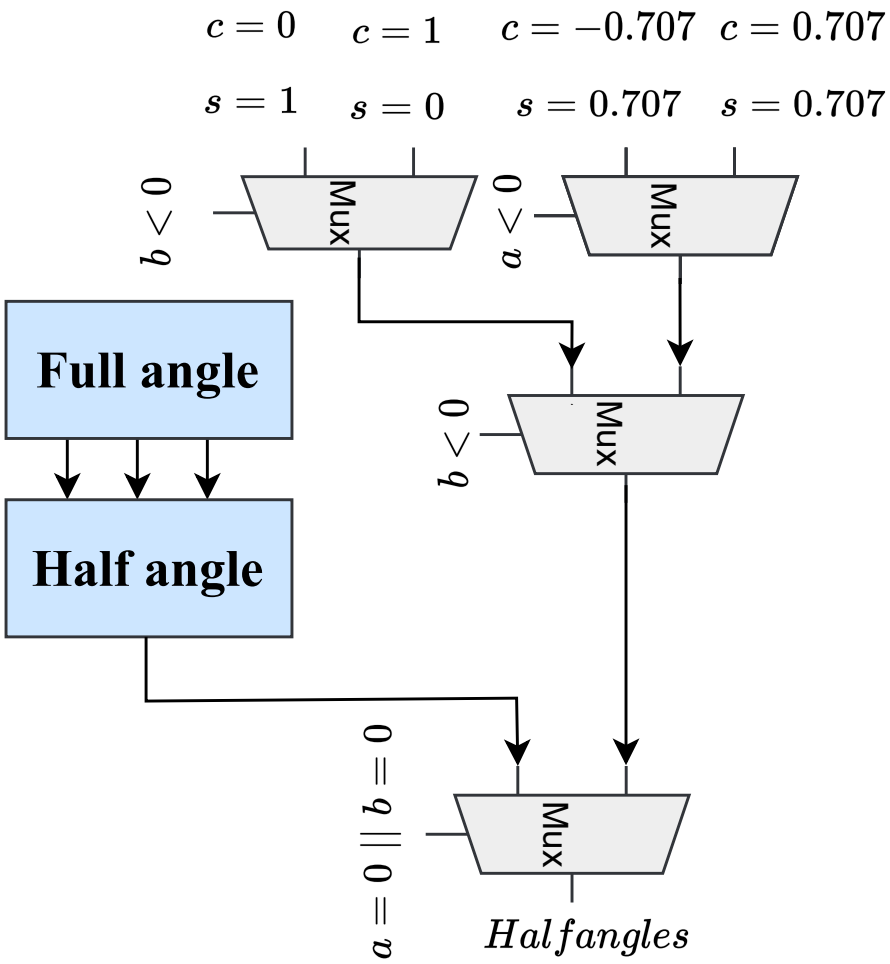


Fig. 2.13 RTL diagram of angle calculator.

A alongside DP_s, DP_u and DP_v from the DP. In non-DP, two subcases can occur; the current 2×2 submatrix DP_c is smaller than current column indices col, and DP_c is greater than col. In the first scenario, A_s is pre-rotated by Hermitian transposed DP_u and then post-rotated by DP_v. In the second situation, A_s is pre-rotated by DP_v then post-processed by Hermitian transposed DP_u. In case of the overlapping with the submatrix DP_s, DP_s values from DP are used from matrix A. This process is repeated until the matrix A is diagonalized. This diagonalized matrix is the matrix S containing the eigenvalues. Matrix U (left eigenvectors) and V (right eigenvectors) are obtained by applying the same DP_u and DP_v rotations to identity matrices I_u and I_v, respectively. It is given as,

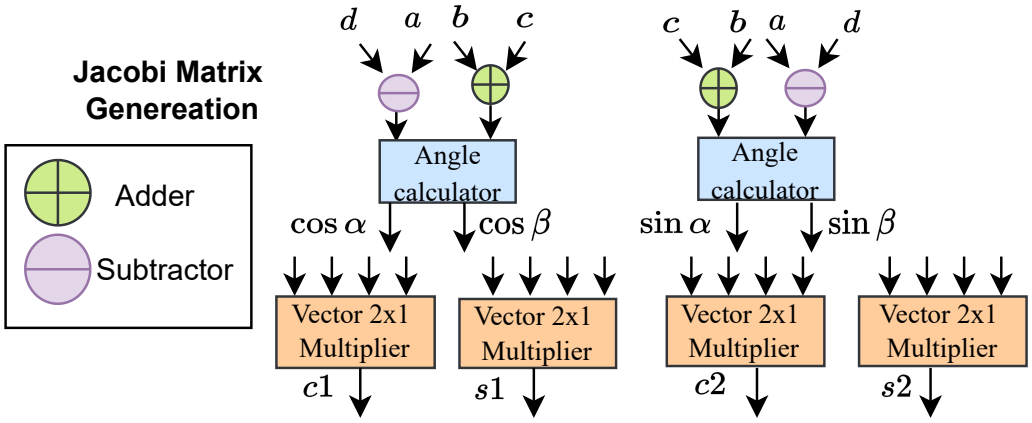


Fig. 2.14 RTL diagram of 2×2 Jacobi matrix generation.

$$A = \begin{bmatrix} \overbrace{a_{11}} & \overbrace{a_{12}} & a_{13} & a_{14} \\ \underbrace{a_{21}} & \underbrace{a_{22}} & a_{23} & a_{14} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \xRightarrow{J_{2\times2}} \begin{bmatrix} a'_{11} & 0 & a_{13} & a_{14} \\ 0 & a'_{22} & a_{23} & a_{14} \\ a_{31} & a_{32} & \overbrace{a_{33}} & \overbrace{a_{34}} \\ a_{41} & a_{42} & \underbrace{a_{43}} & \underbrace{a_{44}} \end{bmatrix}$$

$$\xRightarrow{J_{2\times2}} \begin{bmatrix} \overbrace{a'_{11}} & 0 & a_{13} & \overbrace{a_{14}} \\ 0 & a'_{22} & a_{23} & a_{14} \\ a_{31} & a_{32} & a'_{33} & 0 \\ \underbrace{a_{41}} & a_{42} & 0 & \underbrace{a'_{44}} \end{bmatrix} \xRightarrow{J_{2\times2}} \begin{bmatrix} a''_{11} & 0 & a_{13} & 0 \\ 0 & \overbrace{a'_{22}} & \overbrace{a_{23}} & a_{14} \\ a_{31} & \underbrace{a_{32}} & \underbrace{a'_{33}} & 0 \\ 0 & a_{42} & 0 & a''_{44} \end{bmatrix}$$

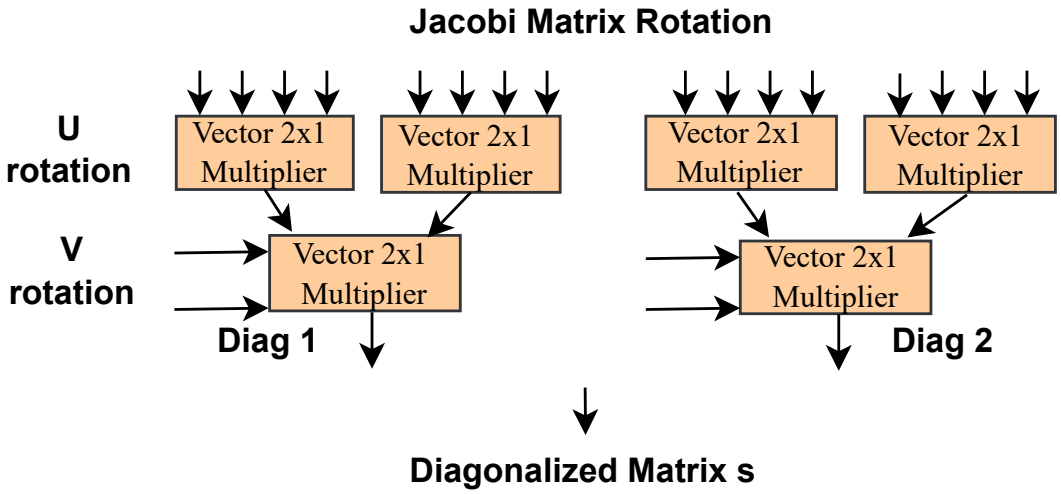


Fig. 2.15 RTL diagram of 2×2 Jacobi matrix rotation.

$$\xrightarrow{J_{2\times2}} \begin{bmatrix} \overbrace{a''_{11}} & 0 & \overbrace{a_{13}} & 0 \\ 0 & a''_{22} & 0 & a_{14} \\ \underbrace{a_{31}} & 0 & \underbrace{a''_{33}} & 0 \\ 0 & a_{42} & 0 & a''_{44} \end{bmatrix} \xrightarrow{J_{2\times2}} \begin{bmatrix} a'''_{11} & 0 & 0 & 0 \\ 0 & \overbrace{a''_{22}} & 0 & \overbrace{a_{14}} \\ 0 & 0 & a'''_{33} & 0 \\ 0 & \underbrace{a_{42}} & 0 & \underbrace{a''_{44}} \end{bmatrix}$$

$$\xrightarrow{J_{2\times2}} \begin{bmatrix} a'''_{11} & 0 & 0 & 0 \\ 0 & a'''_{22} & 0 & 0 \\ 0 & 0 & a'''_{33} & 0 \\ 0 & 0 & 0 & a'''_{44} \end{bmatrix}$$

A single matrix is used for input A and output U to reduce the number of resources. The critical path is the DP block because of the angle calculator since this block has a division and square root operation. The generated Verilog code can be modified at the RTL level to pipeline the architecture to reduce the critical path. The number of parallel rotations impacts resources and performance. At the cost of resources, frequency and latency optimization is employed. The two Jacobi generation and rotation blocks are used in parallel and pipelined with the initiation interval of 8 using the HLS pragma.

In conclusion, initially, Jacobi's left and right rotations are generated. Then, they are applied to the original matrix A to obtain S. Left rotations on identity matrix I produce matrix U while the right rotations produce matrix V^T. But this approach can only be applied to Symmetric matrices. If the matrix is not symmetric, a further

Algorithm 3 SVD computation algorithm.

Inputs: Matrix: $A_{m\times n}$, Identity matrices $I_{m\times m}$ and $I_{n\times n}$
Outputs: Diagonalized eigenvalues matrix: $S_{m\times n}$
Left eigenvectors matrices: $U_{m\times m}$
Right eigenvectors matrix: $V_{n\times n}$

1: **repeat**
2: **for all** $c \in A$ **do**
3: **Diagonal Processor:**
4: **Jacobi matrices generation:**
5: Compute half angles $\cos\alpha, \cos\beta$ and $\sin\alpha, \sin\beta$ using equations (2.11) and (2.12)
6: Generate the matrix DP_u and DP_v by using $c = \cos\alpha \mp \beta$ and $s = \sin\alpha \pm \beta$ identities
7: **Jacobi two-sided rotation:**
8: Matrix $S_{diagonal}$ is computed by applying the Jacobi rotations on A using equation (2.13)
9: **Non Diagonal Processor:**
10: **for all** 2×2 $DP_c \in \left[0, \frac{columns}{2}\right]$ **do**
11: **Jacobi rotations for matrix** S**:**
12: **if** $DP_c < col$ **then**
13: Pre-rotate A by Hermitian DP_u and post-rotate by DP_v
14: **end if**
15: **if** $DP_c > col$ **then**
16: Pre-rotate A by DP_v and post-rotate by Hermitian DP_u
17: **end if**
18: **Jacobi rotations for matrices** U **and** V**:**
19: For matrix U rotate identity matrix I_u by DP_u
20: For matrix V rotate identity matrix I_v by DP_v
21: **end for**
22: **end for**
23: **until** minimum number of iterations to converge

step is needed to symmetrize the matrix. The symmetrization can be done with the help of Givens rotations.

2.2.4 Histogram of Gradients

The authors of [64] provide the architecture implementation of the HOG unit. It receives a 2D grayscale image as input and computes the image pixel gradients p_x and p_y in x and y orientation with the help of two subtractors. Thus the gradient

magnitude and direction are given by;

$$M = \sqrt{p_x^2 + p_y^2}$$

$$\tan(\theta) = \frac{p_y}{p_x}$$

The angle provided by the computation 0° to 180° helps determine the assignment

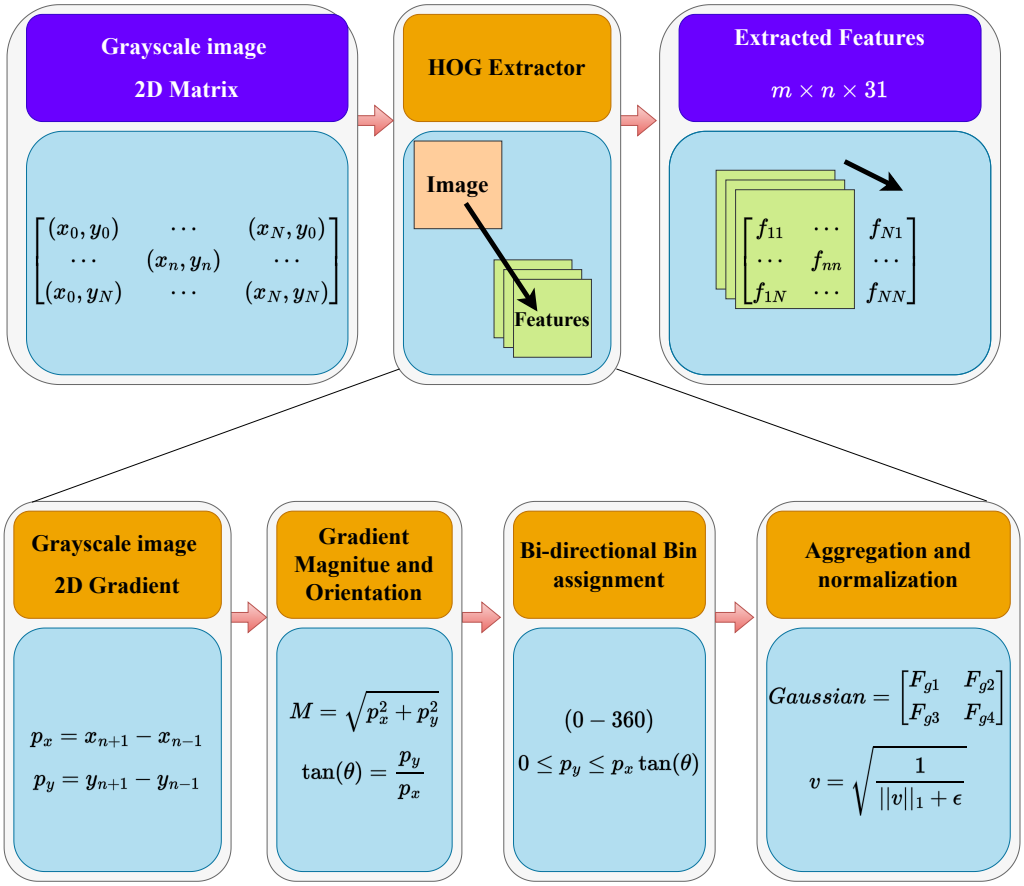


Fig. 2.16 HOG extractor [64].

of gradient magnitude to one of the nine available bins [63]. These bins are used to create the histogram. The angle is computed using integer multiplications rather than a division to save computational resources. It is achieved by performing both angle calculation and bin assignments together. Thus, considering that the angle lies between one of the nine available bins, that gradient is attributed to the corresponding bin. As shown by authors of [64], the gradient assignment to bin one is performed if

the following inequality holds.

$$0 \leq \frac{p_y}{p_x} \leq \tan(20)$$

$$0 \leq p_y \leq p_x \tan(20)$$

Thus, a simple multiplication of integer constants to satisfy the inequality is required. It helps save resources as it avoids a calculation of $\tan(\theta)$, which involves divisions. Next, the gradients are assigned to nine different slots based on the angles, which are the contrast-insensitive slots. Further, they are averaged for smoothening among all bins, and the L1-norm is used for normalization. Compared to the L2-norm, the L1-norm avoids squaring and saves hardware resources without sacrificing accuracy too much. Two further steps are required to obtain Felzenszwalb's Histogram of Gradients (FHOG). First, in the bin assignment, the gradients are assigned to 18 bins based on orientation 0° to 360° which are contrast-sensitive bins. These 18 bins from each block are averaged together, and the previously calculated nine bins are averaged for each block. The nine normalized bin elements are also averaged for the final four blocks. The 18 directional, nine non-directional, and four normalization bins form the 31 third-dimensional features of FHOG. After this step, the output is assigned. The implementation diagram is demonstrated in [64] and in figure 2.16.

2.3 Implementation results and discussion

The proposed RT-DSST datapath is specified using the Vivado HLS tool. The prototyping for this is performed on ZedBoard with Xilinx Zynq xc7z020clg484-1 System-on-Chip. The results obtained based on the block-wise implementation are discussed as follows.

2.3.1 Discrete Fourier Transform

For 1D DFT, the results are compared with the implementation in [53] for Spartan 3E FPGA. The 1D DFT has a maximum size of $N = 320$. Vivado HLS-based post-synthesis results in terms of timing and resources utilized, referred to as HLS, are shown in Tables 2.2, 2.3 and 2.4. The HLS-based timing results outperform [53] by 92 % due to using pre-computed twiddle factors and an unrolled architecture.

The two solutions provided here are based on serial and parallel approaches. The consumption of resources of serial DFT has fewer Look Up Tables (LUTs) than [53] but uses twice the Digital Signal Processors (DSPs) or DSP48E units indicated in Table 2.3. This higher consumption of resources is due to the separate hardware for imaginary and real parts. Thus, our resource utilization for the eight parallel DFTs is also higher than [53]. The choice of higher resources is justifiable as optimization is done for performance in the case of higher dimensions. The maximum operating frequency value for the eight parallel DFTs is 112 MHz while [53] operates at 50 MHz. It can be further enhanced with pipelining in the Verilog code generated at the RTL level. However, the frequency cannot be augmented in Vivado HLS with the pipeline pragma. The pipeline pragma applied to the region of the code to reduce the critical path also unrolls the architecture in the scope it is applied. Consequently, the original architecture is modified and uses more resources. Thus, to avoid this, only inner loops are pipelined. The authors in [51] present radix-8 parallel FFT. They only provide the LUT results for the implementation. The two implementations on FPGAs SPARTAN 6Q and VERTEX 5Q utilizes 3 % and 8 %, respectively. While the HLS-based implementations, series and parallel utilizes 0.7 % and 7 %, respectively. Thus utilizing fewer resources than the FFT solution as well. The verification of results is performed as well. The simulation from Vivado

Table 2.2 Timing results for the FPGA-based DFT.

DFT SIZE	**Latency [cycles]**			**Time [µs]**		
	Spartan 3E [53]	**ZedBoard HLS**		**Spartan 3E [53]**	**ZedBoard HLS**	
		Serial	**Parallel**		**Serial**	**Parallel**
10	24179	260	247	484	2.2	2.4
12	28999	358	342	580	3.3	3.32
20	96509	910	883	1930	7.7	8.6

was compared against MATLAB-generated golden matrices. The relative percentage error is calculated by,

$$E = \frac{(R_{MATLAB} - R_{HLS})}{R_{MATLAB}} \cdot 100,$$

where E denotes the error. The golden matrices used were obtained by considering the intermediate values from the DSST algorithm at the input of each block. They

Table 2.3 Area results for the DFT implementation in FPGA.

DFT SIZE	LUT			MULT	DSP48E	
	Spartan 3E [53]	ZedBoard HLS		Spartan 3E [53]	ZedBoard HLS	
		Serial	Parallel		Serial	Parallel
10	616	395	3702	4	8	32
12	648	421	3710	4	8	32
20	776	460	3823	4	8	32

Table 2.4 Area and timing results of DFT and DFT2.

DFT (ZedBoard)		Time [ms]	LUT	DSP48E
SIZE	TYPE			
320	DFT	0.475	1722	32
320×320	DFT2	273	11352	32
320×320	DFT2 (2 parallel)	170	14934	64

were then applied to HLS-based units. Next, the outputs of both are compared. The average error values obtained were 6.52 and 6.9 for real and imaginary matrices, respectively. This is due to the usage of double precision in MATLAB. HLS results are still approximate enough as double precision in hardware will consume four times more resources. Also, it will affect latency, making it slower. The DFT2 implementation is for maximum dimensions of 320×320. The obtained results for DFT and DFT2 are reported in Table 2.4. It also has a maximum frequency of 112 MHz. The authors in [54] present multi-dimensional FFT IP on Xilinx VIRTEX-5 FPGA. It utilizes 25 % of Slices and 53 % DSP48E, while the HLS-based solution for DFT2 consumes 28 % of Slices and 29.1 % DSP48E. In terms of timing the solution in [54] only consumes 11 ms while the HLS-based one takes 170 ms for a dimension of 320×320, which is expected as the HLS-based is for DFT while the one in [54] is for FFT only with the frequency slightly lower at 100 MHz.

2.3.2 QR factorization

For the QR unit, the results are compared with the implementation in [55] for Virtex-6 XV6VLX240T speed-2 FPGA, which is for a 4×4 matrix. The QR has a maximum size of 800×17. The authors of [55] use fully unrolled fixed-point iterations while the approach is based on floating-point. The comparison is presented with the 32-bit fixed-point version. Vivado HLS-based post-synthesis results regarding timing and resources utilized, referred to as HLS, are shown in Tables 2.5. The HLS-based timing results take four times more clock cycles compared to [55], while it performs better in terms of the area. This solution uses 2.3 times fewer DSP48E resources since this is not the critical block; thus, resource optimization was the target. The operating frequency is almost similar to the non-pipelined version. The HLS-based approach is generic compared to [55] is a fixed size. The worst-case delay for maximum input size, i.e., 800×17, 337 µs for QR economy, and it consumes 26 DSP48Es resources with the maximum frequency of 116 MHz. The average error is computed in the same procedure as in section 2.3.1, and its value is 0.034 for matrix Q since only matrix Q is used for the DSST algorithm. HLS-based results are reasonable enough since double precision MATLAB solution in hardware will require four times more resources. The authors in [58] present 4×4 QR using Givens rotations on TSMC 90 nm. It operates at 214 MHz with a latency of 4 cycles, while the HLS-based solution operates at 115.9 MHz with a latency of 467 cycles on an FPGA. The latency result is justified as the solution in [58] has a maximum dimension of 4×4 and is optimized for it, while the HLS-based one is flexible and the maximum dimension is 800×17. Also, as mentioned, this is implemented with lower area utilization.

Table 2.5 Timing and area for FPGA based 4×4 QR.

Architecture	**Device**	**Frequency**	**Latency**	**Resources**		
		[MHz]	**[cycles]**	**DSP48E**	**FF**	**LUT**
Mult A [55]	**Virtex-**6	117.1	116	48	10844	11337
Mult B [55]	**Virtex-**6	377.6	140	48	11520	11225
HLS	**ZedBoard**	115.9	467	21	6054	8824

2.3.3 Singular Value Decomposition

For the SVD unit, the results are compared with the implementations in [60] for Xilinx xc6s lx 100t-4 FPGA, for low values of SVD size, while for others, they are compared with [61] for Spartan-3e XC3S500E speed grade-5 FPGA. The SVD has the maximum size of 32×32. The authors of [55] use fixed-point iterations while the approach is based on floating-point. Vivado HLS-based post-synthesis results in terms of timing, referred to as HLS, are shown in Tables 2.6. The HLS-based solution performs better than both in literature for all sizes except for 4×4 with [60]. The improvement in timing is by factors of 3.7 and 4.8 as compared to [61] and [60], respectively. The cost is paid in terms of twice the area, as shown in Table 2.7. The main reason is that the HLS angle calculation unit has division and the square root of floating-point numbers, while [60, 61] has the CORDIC algorithm. The worst-case delay for a maximum input of size 32×32 is 4 ms, while it consumes 30 DSP48Es resources. The maximum frequency is 108 MHz. Similar to QR, the error calculation is used from 2.3.1. The average error value is 7.156 for matrix U since only matrix U is used for the DSST algorithm. HLS-based results are reasonable enough since double precision MATLAB solution in hardware will require more resources. Also, the latency will be higher. The authors in [65] present 4×4 SVD on Virtex-5 FPGA. It operates at 233 MHz with 54 072 slices, while the HLS-based solution operates at 108 MHz with 10 110 slices. The low-frequency result is justified as the solution in [65] has a maximum dimension of 4×4, the data width of 12 and is optimized for it, while the HLS-based one is flexible, and the maximum dimension is 32×32

Table 2.6 Timing results for SVD implementation in FPGA.

SVD SIZE	**Time [µs]**		
	xc6s lx 100t [61]	**Spartan-3e [60]**	**ZedBoard HLS**
4×4	-	12.1	35
10×10	3570	1001	207
20×20	4280	12100	1135
30×30	12550	-	3445
40×40	26860	-	7799
50×50	-	69500	14872

and the data width of 32. Also, as mentioned, this is implemented with lower area utilization.

Table 2.7 Area results for SVD implementation in FPGA.

Architecture [4 × 4]	Device	LUT	BRAM	DSP48E
Implementation [61]	**xc6s lx 100t**	5283	8	12
Implementation [60]	**Spartan-3e**	1504	3	16
HLS	**ZedBoard**	10110	14	30

2.3.4 Histogram of Gradients

For the HOG extractor, the results are provided from [64] for Xilinx Virtex-5 XC5VFX200T speed grade-5 FPGA. The HOG has a maximum size of 320 × 240. The maximum operating frequency is 270 MHz. The speed is 60fps for an image size of 1920 × 1080, while it only consumes 12 DSP blocks. The authors in [62] present 640 × 480 HOG on Cyclone V. It operates at 50 MHz with 38 DSPs and speed of 78fps. Thus, the one in [64] has better operating frequency and fewer DSPs, hence more suitable for the task at hand.

Table 2.8 Area, fps, and power results of the proposed architecture for DSST algorithm.

Unit	Resources				FPS	Power
	BRAM	DSP	FF	LUT		[mW]
DFT	0	32	4419	6305	6153	258
DFT2	192	64	11922	18343	173	662
QR	33	26	6662	5837	2948	238
SVD	16	30	8294	10808	248	358
HOG[1]	7	12	3642	3924	60	244
Misc	0	8	591	1545	4392	156
Used	248	172	35530	46762	-	1916
(%)	(88.6)	(78.2)	(33.4)	(87.9)	-	-
Total	280	220	106400	53200	-	-

[1]The resources and FPS reported here are from the implementation in [64]. But the authors didn't provide power consumption, thus the power estimation reported is calculated using the implementation provided in https://github.com/nikkatsa7/HOG_Zedboard.git

2.3.5 Overall resources and speed

The table 2.8 displays the maximum resources FDSST units utilize in terms of DSP48E, BRAM, Flip Flops (FF), and LUTs. Alongside this, the maximum power for each unit is displayed as well. Power is reported by using power reports of post-place-and-route from VIVADO HLS. The values reported are for the maximum dimensions of each unit in the FDSST. Each block's speed/ FPS is computed separately, and the average FPS is obtained for each block using a range of sizes as input. The FPS is calculated as follows;

$$FPS = \frac{F_{max}}{Cycles},$$

where cycles denote the number of clock cycles required to process one whole frame. The means FPS is reported in Table 2.8. The architecture is divided into four stages: Scale Search, Scale Filter, Translation Search, and Translation Filter. Table 2.9 depicts the mean FPS values of each stage. The total FPS for a stage is computed by summing the reciprocal FPS of the units involved in the corresponding stage. Regarding FPS, the most critical stages are the translation search and filter stages since they involve DFT2 units. The most critical stage (with minimum FPS) dictates the mean FPS of the whole architecture. Thus, using the two parallel architectures for critical block, the FPS is improved and shown in Table 2.9. Finally, with an image size of 320×240, the RT-DSST can fit in a Zync Zed board with a means frame rate of 25.38fps. The maximum operating frequency is 108 MHz. Enhancing the size of the image using the same FPGA would decrease FPS. If it doubles, the FPS is lowered by a factor of 1.5. But for large image sizes with a different FPGA, the same FPS can be attained at the cost of higher resources; for the input and output images in Vivado, the (hls::Mat) from HLS is used. Then, they are transformed into an 8-bit integer matrix for processing.

Table 2.10 compares the whole HLS-based RT-DSST architecture against other tracker implementations in the literature. HLS frame rate is nearly equal to the original DSST [7] but is lower than the fDSST tracker. The power of HLS-based implementation is better as the fDSST runs on Xeon Central Processing Unit (CPU), thus consuming more resources. A correlation filter-based tracker [66] is provided for IoT applications, and the implementation is for edge devices. It uses the server for computations, which allows it to be faster than the HLS-based approach. Again, considering power HLS solution is much better. The authors of the Rotation Aware

DSST tracker [67] use DSST to integrate rotation awareness for accurate scale estimation. In comparison, the HLS solution dominates both speed and power. MOSSE [68] is not the DSST-based tracker but uses programmable logic. It has a higher speed for 4K resolution images, but it is presented for power comparison here. Again HLS solution has a lower power consumption. Regarding FPS, the Moving Target Tracker MTT [69], which uses a Single-Instruction-Multiple-Data

Table 2.9 FPS results for proposed DSST architecture for 240 × 320 images.

Target Unit	Scale		Translation			
	Search	Filter	Search	Filter	2 parallel	
					Search	Filter
HOG	60	60	60	60	60	60
DFT	6153	6153	-	-	-	-
IDFT2	-	-	29.9	-	59.8	-
DFT3	-	-	86.52	43.26	173	86.52
SVD	-	-	-	248	-	248
QR	-	1474	-	-	-	-
Misc	4392	4392	4392	4392	4392	4392
Total	58.63	56.38	16.16	22.71	25.38	30.78

Table 2.10 comparison against other implementations.

Algorithm	Platform	Power [W]	FPS
DSST [7]	Intel Xeon 2 core CPU	-	25.4
fDSST [7]	2.66GHz 16GB RAM	-	54.3
RPCF [66]	Xilinx Soc Zynq dual core Cortex A9 + Artix7 FPGA	-	39.3
RADSST [67]	Intel i7 6700k CPU 64GB	-	20
MOSSE [68]	Zynq US+ MPSoc (4k)	8.84	60
VDSP [69]	Vision DSP 4GB RAM	-	165
Mini-Tracker [70]	Zync ZedBoard	1.28	18.6
Gabor [71]	Virtex-5 + Xilinx Microblaze processor	-	30
HLS	Zync ZedBoard	1.92	25.4

(SIMD) based DSP platform, performs best. It also uses 4 GB of RAM. So again, regarding power, resources, and portability, the HLS solution is better. The authors in [70] present a Mini-tracker implemented on ZedBoard with a lower FPS of 18.6 and power consumption slightly lower than the HLS-based one. The authors in [71] present a Gabor filtering-based tracker implemented on Virtex-5 and Microblaze-core with an FPS of 30. But the implementation in [70] and [71] utilize 611 and 288 DSPs, respectively. While the HLS approach uses only 172 DSPs. Compared to all these trackers, the HLS-based solution consumes less power except the Mini-Tracker. The speed is comparable to some of the trackers, but the fewer resources make it feasible to operate on the field.

2.4 Conclusions

This chapter presents the RTL implementation of the significant blocks of FDSST [7]. The RTL implementation is given for the significant mathematical operations involved, including SVD, QR, DFT2, and HOG extractor. The DFT unit is implemented by 8-parallel architecture, which is the base for the DFT2 unit.Further a 2-parallel architecture of DFT2 is used for 3D-DFT. This approach improves the timing by 92 % with increased resources as compared to the state-of-the-art. For QR, the resource utilization is improved by a factor of 2.3 compared to [55]. The literature mostly consists of non-scalable implementations. At the same time, the implemented solution here is scalable from 4×4 to higher dimensions. For SVD, the timing is enhanced by a factor of nearly 3.8 compared to [61]. The solution is scalable and performs better with higher dimensions in contrast to other implementations in the literature. The HOG extractor is taken from the literature as it provided a higher frame rate with low resource consumption. Finally, an image of 320×240 size can fit in Zync Zed board with a mean frame rate of 25.38fps, thus can be operated as a standalone unit. The overall power dissipation is estimated to be 1.92 W. This is much lower as compared to high-speed computer and server-based solutions. Further research on this can be performed to optimize the operations involved. Also, the work can involve the integration of the whole algorithm and interfacing with a real camera to test it on the field. Ultimately, as a future work, the overall accuracy can be compared to the available databases.

Chapter 3

VLSI Architecture of Wiener Filter for Video Encoding

3.1 Background

Part of this work is published in the article by the author of this thesis in [39]. This section discusses the video encoding and the particulars of the Wiener filter algorithm [27, 31]. The transmission of high-quality videos requires a lot of resources, including all the pixels. The transmission of videos with a resolution of 4K in real-time would require a very high cost. However, all the video content is not necessary for transmission as there is much redundancy available in the frames. Only the necessary part of the video needs to be transmitted, and when required, the video is reconstructed at the receiver end. Thus, the video needs to be encoded and then sent. The real-time video transmission will be feasible if the encoding is fast and the reconstruction is reliable. Also, the same encoding procedure can be applied to store the video, and when required, the desired video is reconstructed. Thus the central concept of video encoding is to compress the video into bitstreams, and then the encoded bitstream is transmitted. The compression is possible due to the spatial correlation of the pixels or the temporal correlation between two consecutive frames, and the nearby frames mostly have the same content. The compression has losses; thus, the accuracy of the video after reconstruction changes and depends on the used codec as per parameters. The parameter is the amount of data required to represent the video, called the bit rate, the complexity of the encoder, and the noise sensitivity.

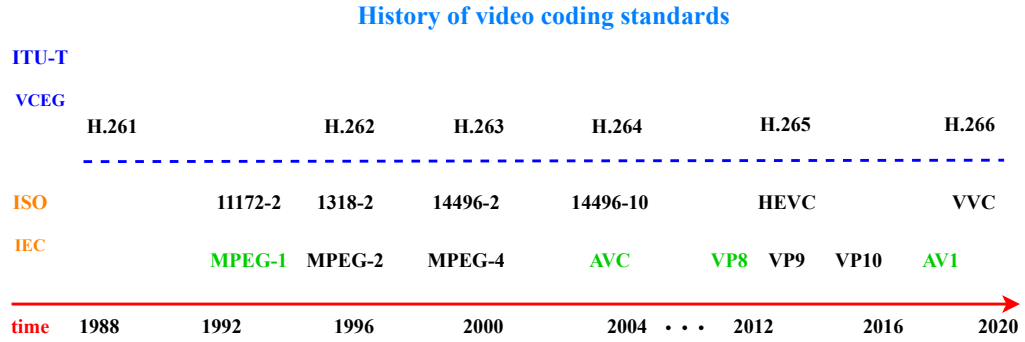


Fig. 3.1 Coding standards timeline [72].

In 1984, the first digital video coding technology standard developed by the International Telecommunication Union (ITU) called H.120 was launched. Since then, many other standards have come into existence, and with each new standard, new algorithms were introduced in the video encoding process. The video encoding standards timeline [72] is displayed in figure 3.1. Among the recent ones are the VP9 video codec launched in 2013 and the High-Efficiency Video Coding (HEVC) by the International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC) Moving Picture Experts Group (MPEG) and ITU-T Video Coding Experts Group (VCEG). In 2015, the consortium of many companies co-founded the Alliance for Open Media (AOMedia). It was targeted to be open and royalty-free. Thus, a new open video coding format, AV1, evolved from the previous VP9 and can compete with HEVC. The main concern is improving scalability to make it compatible with many modern devices and improving the performance in terms of compression. The AV1 encoder block diagram is shown in figure 3.2. The encoder accepts the video as a sequence of frames and starts compressing them. The previously encoded frames are utilized for predicting the new one. Also, the coding algorithm computes the difference between the predicted and current frames, which is processed via a transform coding algorithm, typically the Discrete Cosine Transform (DCT). Finally, the quantization is performed, and the resultant bitstream is sent as output.

As mentioned in the section, the Wiener filter is used for noise reduction and removing blurring effects [36, 37]. Other applications involve signal processing applications like de-convolution and signal detection [38]. In video coding, in particular, in AV1, it is used as an in-loop denoising filter to restore frames and

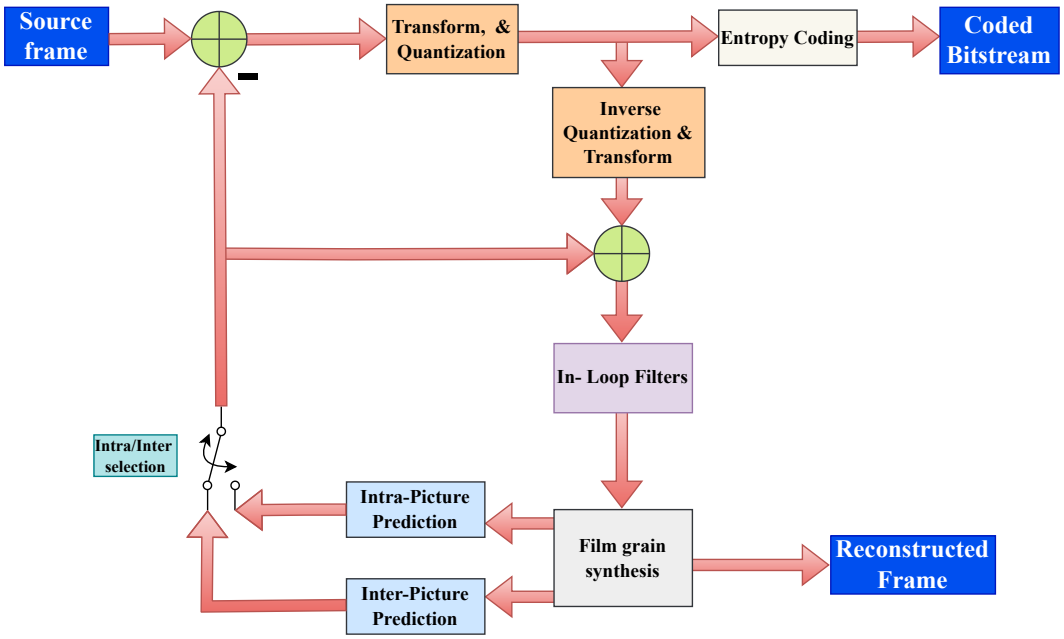


Fig. 3.2 The block diagram of AV1 encoder [72].

enhance the quality close to edges. Every one of the pixels in a frame is encompassed with a $w \times w$ window around it. The size w is an odd number such that

$$w = 2r + 1$$

where r is an integer denoting the radius of the encompassing window [27, 31]. Thus, the filtering is performed by using separable Wiener coefficients, using the symmetry and normalization constraints; this helps reduce the number of filtered parameters sent. So, the processed version of the input taps is used instead of $w \times w$ or w^2 taps. These taps are available in matrices H and M. In particular, H is given by

$$H = E[XX^T] \tag{3.1}$$

that is the auto-covariance of X. X is the column-vectorized version of the w^2 input taps, where $E[\cdot]$ corresponds to the expectation operation. M is given by

$$M = E[YX^T] \tag{3.2}$$

that is the cross correlation between X and the source pixel Y. The procedure needs to send w^2 values for each filtered pixel, thus increasing both the bit rate cost and the decoder complexity. Instead, some constraints are imposed [27, 31]:

- The implemented filter has to be separable;
- The filter in both horizontal and vertical must be symmetric;
- Both filter coefficients, horizontal and vertical, have limits on the values they can take. The sum of them should be exactly S, where S is a constant value that, for the AV1 implementation, equals 2^{16}.

These imposed constraints make it feasible to send, for each filter, just r values instead of w. Further, as the filter is now symmetric, it spans over only the computation of the first r elements. Hence, the implementation complexity is reduced. From here on, both the vertical and horizontal filters are called a and b, respectively, and are reconstructed from the r values. They can be derived as follows:

$$a(i) = a(w-1-i), i = 0,1,..r-1 \tag{3.3}$$

$$a(r) = S - 2\sum_{i=0}^{r-1} a(i) \tag{3.4}$$

$$b(i) = b(w-1-i), i = 0,1,..r-1 \tag{3.5}$$

$$b(r) = S - 2\sum_{i=0}^{r-1} b(i) \tag{3.6}$$

The process is iterative: it initiates with an initial value for horizontal and vertical filters. Then, it optimizes one of them (a in this case) while the other is kept constant (b_{in}). Next, the first *r-taps* version of the filter is obtained, which is reconstructed using Equations (3.3)–(3.6). Finally, it is used as input for the other filter processing. For example, figure 3.3 represents the Wiener filter process.

This work provides a hardware implementation of the Wiener filter process for the AOMedia AV1 video coding [39]. In addition, a high-speed implementation can be used for real-time data processing. This is possible due to the high frame rates achieved by the hardware implementation. First, the following section 3.2 presents the architecture implementation. Then, section 3.3 presents the final results and a real-time filter evaluation. Finally, the conclusions are given in Section 3.4.

3.2 Architectural Implementation

Starting from the mathematical description of the algorithm performed in [31] and then observing the implementation of the AV1 codec, an architecture for the Wiener filter is proposed. The inputs to the architecture are:

- The matrix H, where each element of the H_{ij} matrix has the dimensions 7×7.
- The matrix M, which has the dimensions of 7×7.
- The first guess vector b_{in}, that is composed of 7 elements.

The horizontal and vertical filters, represented by the vectors a and b, are the outputs. Each of the vectors comprises seven elements. The process consists of two major blocks, *update a* and *update b*. Each block further comprises many sub-blocks. The data flow is shown in figure 3.3. The first stage, called *update a*, accepts the first vector b and calculates matrix B and vector A with the dimensions of matrix B and the vector A are 4×4 and 4, respectively. They are obtained as follows:

$$B = \sum_{i=0}^{w-1} \sum_{j=0}^{w-1} H_{ij} b(i) b(j) \tag{3.7}$$

$$A = \sum_{i=0}^{w-1} M_i b(i) \tag{3.8}$$

The final output of the Wiener filter is generated by solving a linear system of equations such that:

- The coefficients of the system are in matrix B
- The constant terms are in the vector A

The result of the solution of linear equations has the output values of filter a. As described in the AV1 software model, $r = 3$ is a necessary condition for processing them by the *Enforcement block* that helps reduce the dimensions. The reduced dimensions for matrix B and array A are 3×3 and 3, respectively. Hence, the linear system of equations with proper dimensions is achieved. The standard Gaussian elimination method is exploited to solve the system of equations. The steps

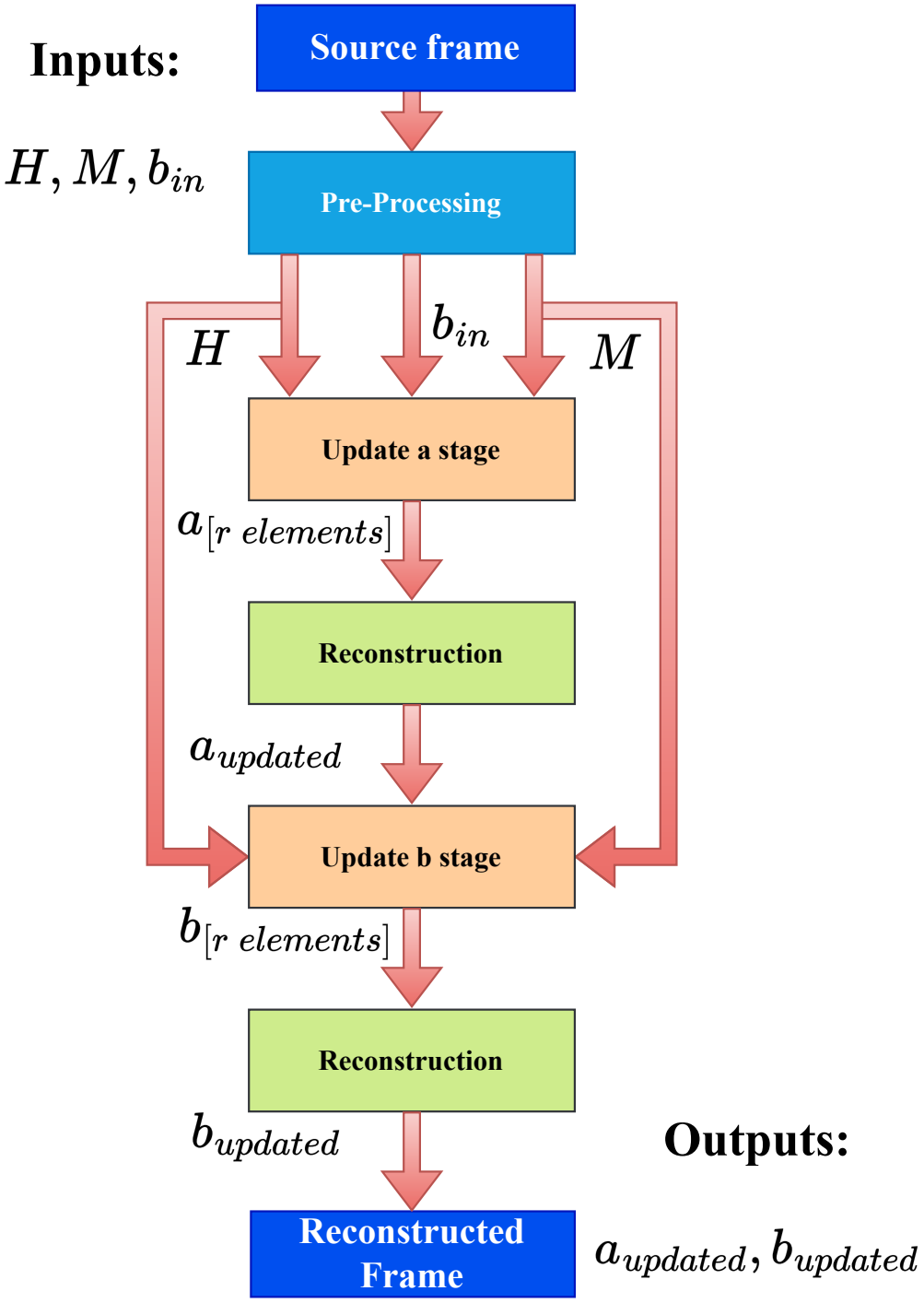


Fig. 3.3 Wiener filter process.

comprise *Partial Pivoting*, *Forward Elimination*, and *Back-Substitution*. Similarly,

the blocks involved in the implementation are *Partial Pivoting*, *Forward Elimination*, and *Back-Substitution*, respectively. The post-processed result is the output vector X that compromises three elements. Next, using the symmetry property, the updated values of vector a are reconstructed to the original dimension w. Similarly, for the block *update b*, starting from the new and updated vector a, vector b is generated by following the same sequence of steps. A matrix storing mechanism is used instead of applying a feedback approach for matrix B and vector A. The following equations can summarize the operations performed:

$$B = \sum_{i=0}^{w-1} \sum_{j=0}^{w-1} H_{ij} a(i) a(j) \tag{3.9}$$

$$A = \sum_{i=0}^{w-1} M_i a(i) \tag{3.10}$$

Again, by using the same constraints as for vector a, the updated vector b vector is also reconstructed. A detailed description of the sub-blocks mentioned above is reported in the following subsections. The main idea of this work was to start from the basic implementation and move toward an optimized implementation. The architecture presented in section 3.2.5 is based on the same data path elements. However, each sub-block is implemented depending on the kind of optimization required. In addition, each architecture contains a specific FSM for control.

3.2.1 Enforcement block

It is responsible for compressing the inputs. The compression is to adapt them to the dimensional linear system of equations. For example, the components of vector A compute the enforced output described in figure 3.4. Similarly, for the matrix B taking 16 components, reducing them to 9, i.e., 3×3. From here onward, the B matrix will be called A, and vector A will be called b to be coherent with the C model.

3.2.2 Partial Pivoting

It is the most simple sub-block in the process as it only assists in interchanging matrix rows. The hardware architecture is depicted in figure 3.5, where $k = 0$ denotes the

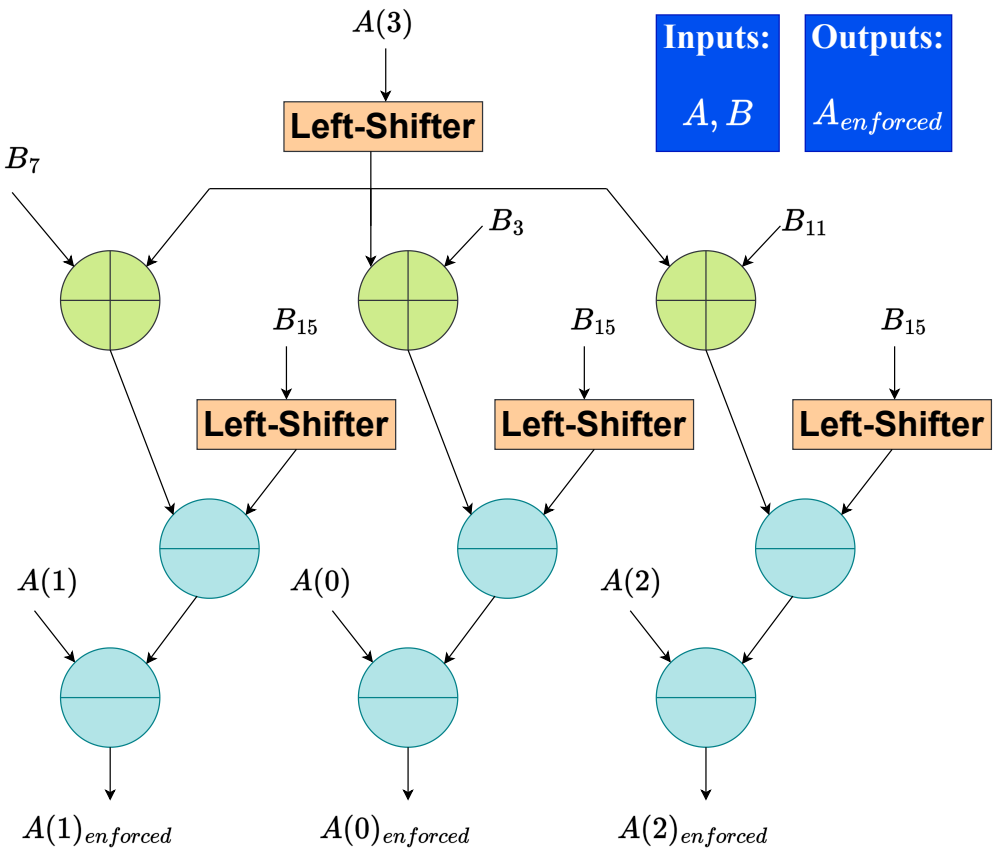


Fig. 3.4 Enforcement architecture.

first step of *Partial Pivoting*, while $k = 1$ represents the second one. In the first stage, the absolute values of $|A_0|$, $|A_4|$, and $|A_8|$ are computed and then compared two at a time to find the largest one. Next, applying the *Swap Rows block*, interchanging the b elements based on the results from the comparators. Finally, in the second stage, the absolute values of $|A_5|$, $|A_9|$ are compared and then swapped to make the matrix in shape to be solved with the Gaussian Elimination Method.

3.2.3 Forward Elimination

It is one of the mathematical steps in solving a linear system that performs multiplication, division, and subtraction to properly combine two rows of the matrix, thus transforming it as close as possible into an upper triangular form. Figure 3.6 reports the hardware implementation of the *Forward Elimination* operation for b vector.

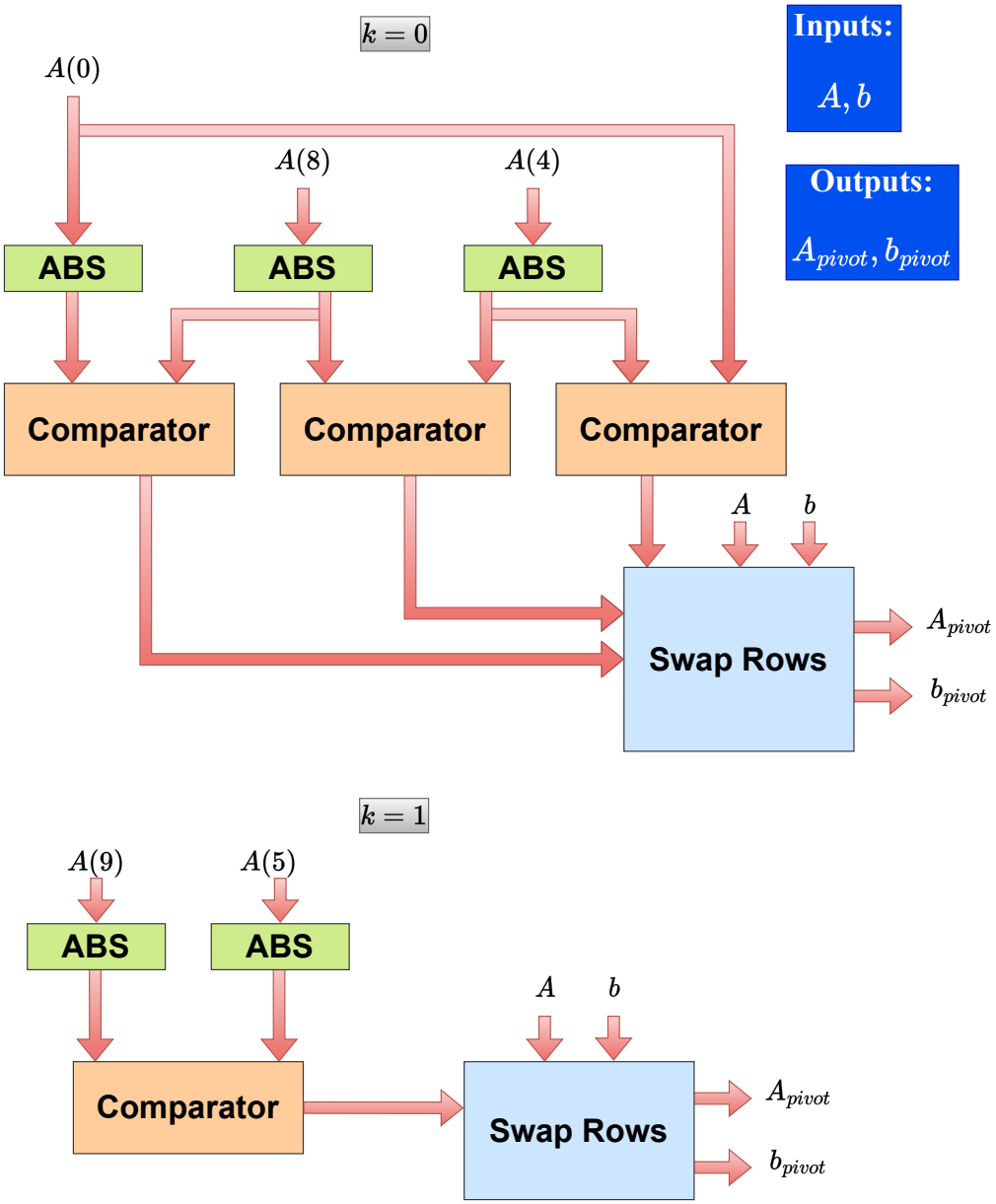


Fig. 3.5 Partial Pivoting architecture.

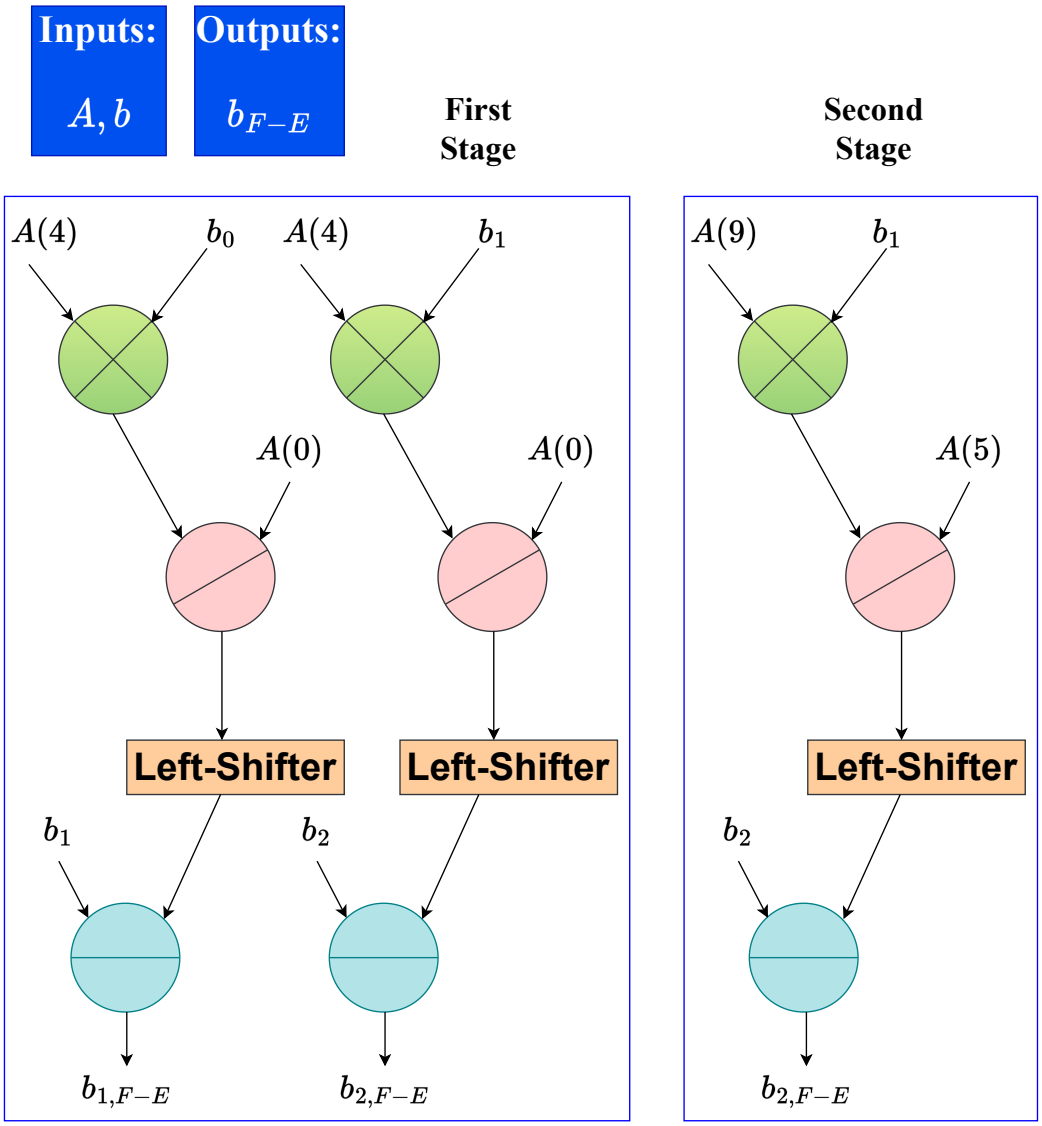


Fig. 3.6 Forward Elimination architecture.

3.2.4 Back-Substitution

After the previous steps, the linear system now takes the form;

$$\begin{cases} A_0 \cdot a_0 + A_1 \cdot a_1 + A_2 \cdot a_2 = b_0 \\ A_5 \cdot a_1 + A_6 \cdot a_2 = b_1 \\ A_{10} \cdot a_2 = b_2 \end{cases}$$

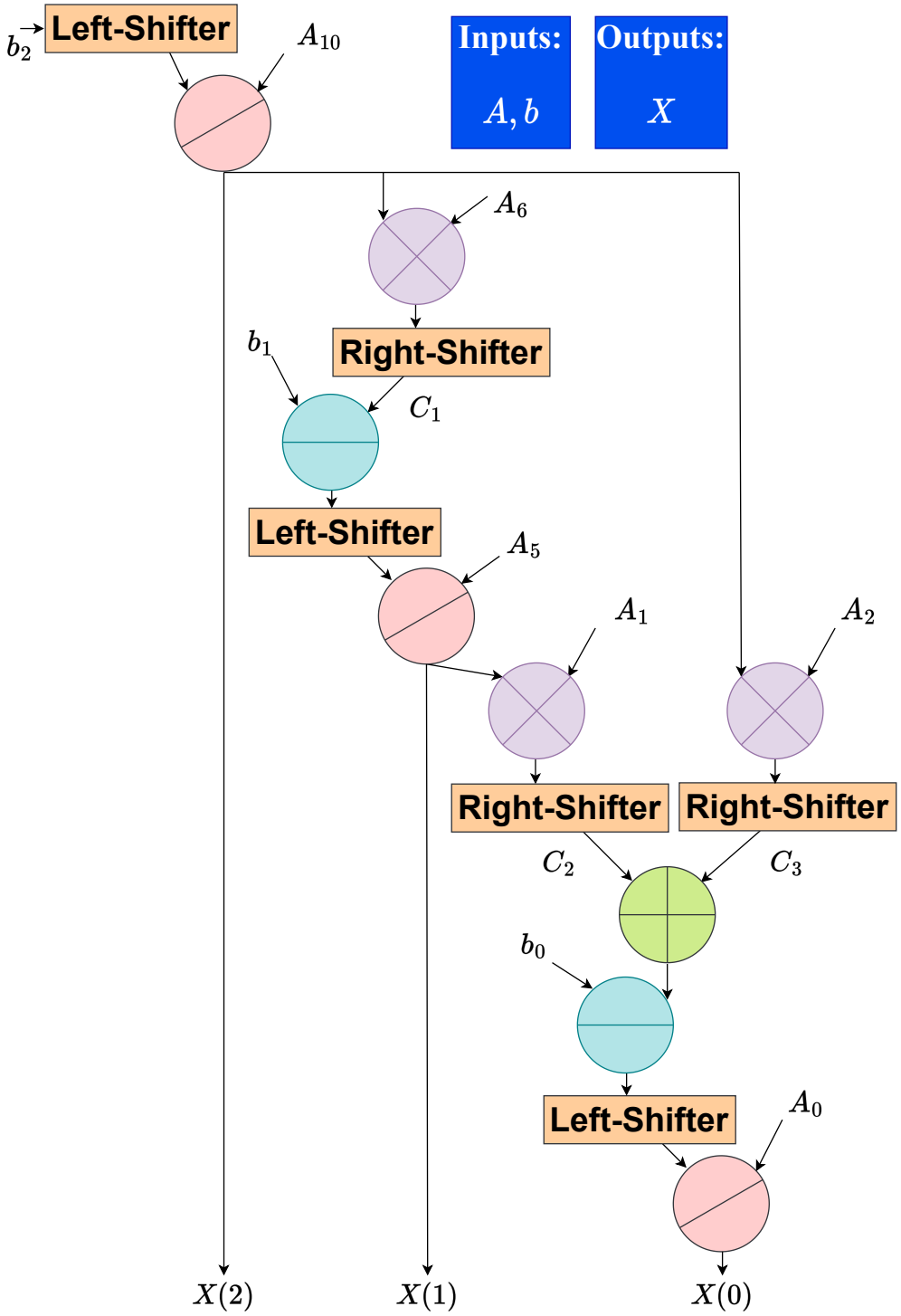


Fig. 3.7 Back-Substitution and Storing architecture.

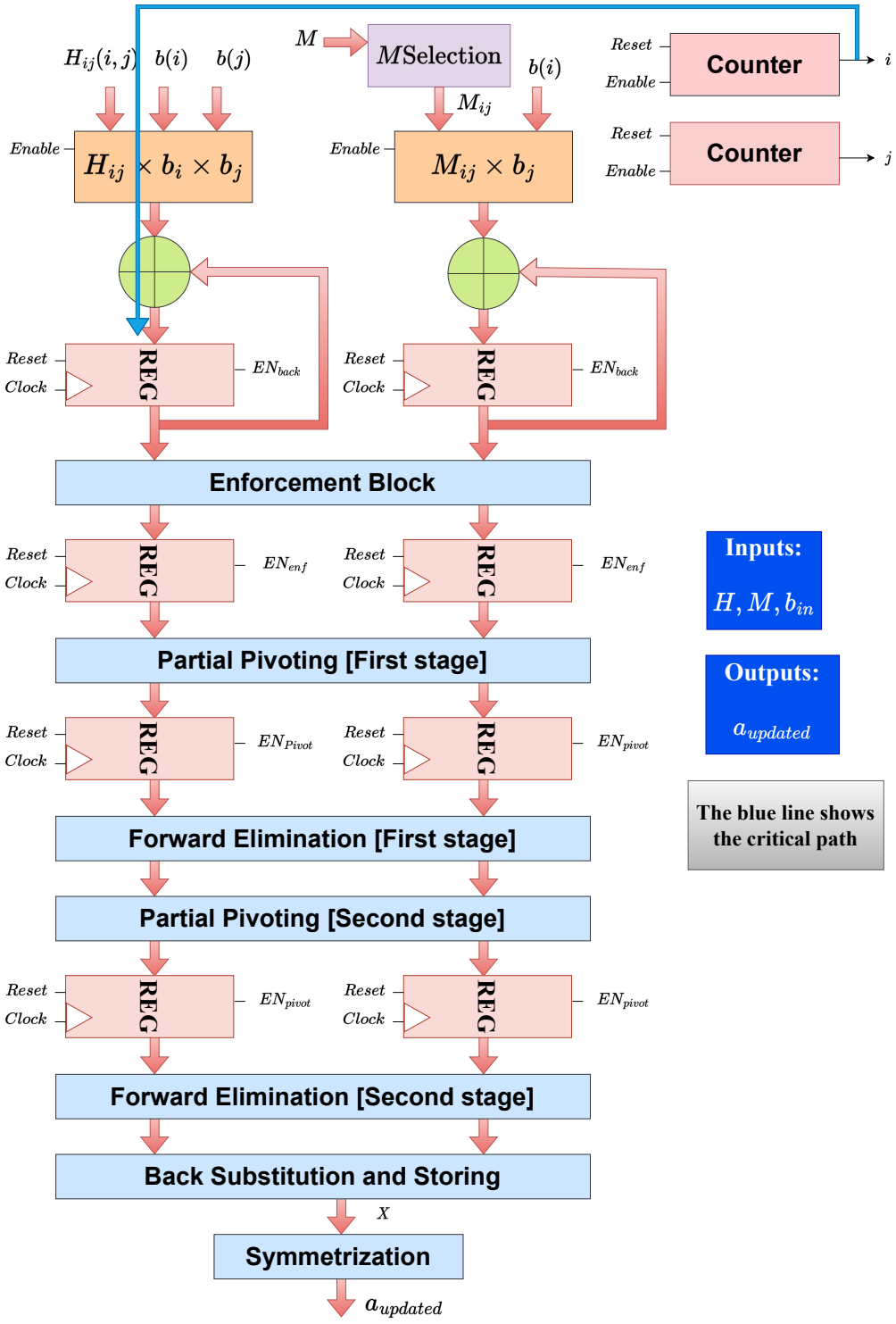


Fig. 3.8 Update a data path.

Now, the system must be solved by using the *Back Substitution* and *storing*. The architecture implementation is shown in Figure 3.7. It is a complex block, performing several expensive operators like dividers and multipliers. Also, the figure 3.8 showing the *update a* data path highlights with an arrow the critical path. It goes from counter i to the adder on the top left. It is due to the term

$$H_{ij} \times b_i \times b_j$$

as it involves cascaded multipliers.

3.2.5 High Speed Architecture

Modern architectures' primary objective is to process data in the shortest possible time, which means operating at high frequency. In addition, the throughput is also an essential parameter since matrix processing is involved. Hence, the proposed architecture needs to be accelerated. Post-synthesis timing results are utilized to recognize the critical blocks limiting the speedup. Finally, improvements were made to the original architecture to reduce clock time and enhance frequency and throughput. This analysis identified the two critical points of the structure:

- The total length of the combinational paths;
- The combinational dividers.

The first critical point identified is due to many operators present along different combinational paths. That means the time desired to elaborate a single piece of data is very long, thus slowing the clock down. The second critical point identified is because of the structure of the primary dividers that perform a division between two 64-bit operands. The dividers in the basic architecture are implemented in a purely combinatorial way. Significantly, the one used here performs an n-bit division exploiting $2n$ consecutive addition and subtraction operations. Thus, each divisor compromises 128 operators of adders and subtractors along the same combinational path. It also slows down the clock as well as affects the throughput. Inserting the pipeline registers along the combinational paths using the typical Restoring division algorithm [73, 74] is reported in Figure 3.9. It improves the clock frequency. The pipelining is applied to the *Back-Substitution*, *Forward Elimination* basic block,

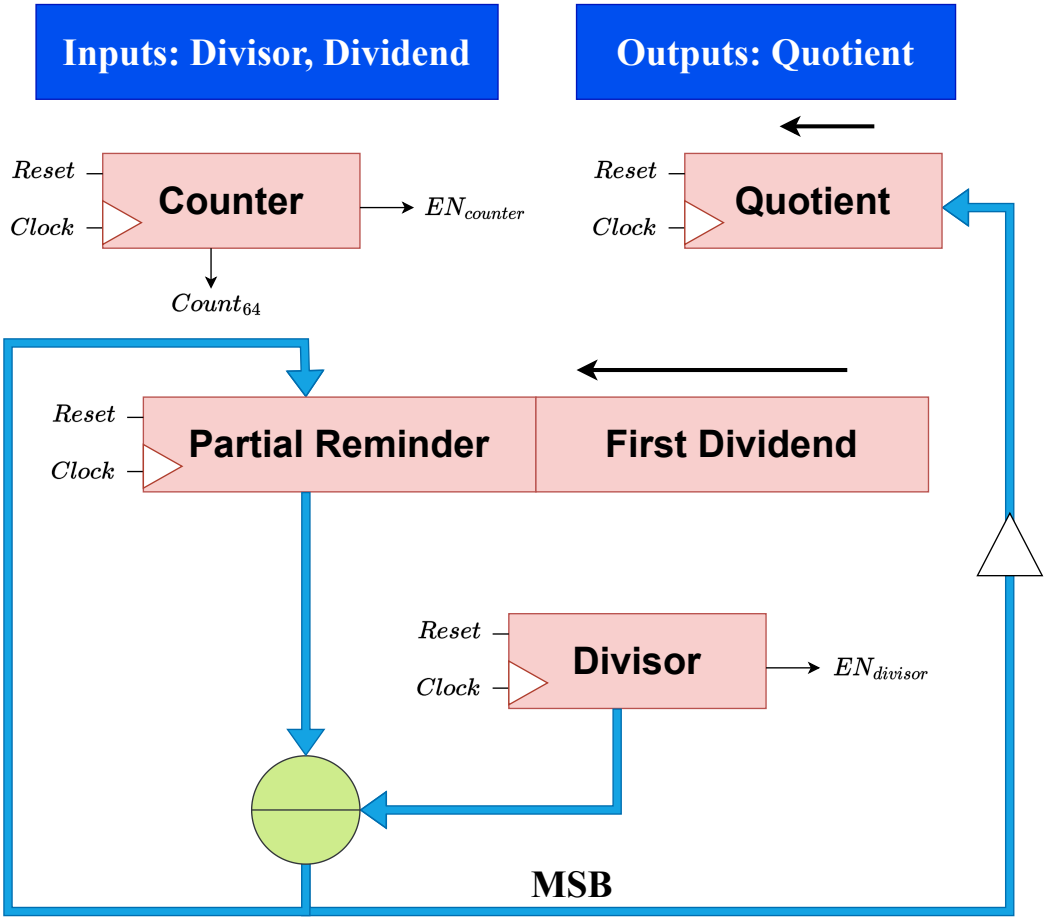


Fig. 3.9 Optimized division data path.

update a, and *update b* top-level architectures, thus reducing the critical path to a single operator block. In addition, all the previous dividers are replaced with optimized Restoring Dividers (RD) [73, 74]. The restoring algorithms store the dividend and divisor in respective registers shown in Figure 3.9. Then, they are shifted and subtracted. The MSB of the result is complemented and shifted in the quotient register, and the counter is decremented. The result is ready in the quotient register when the counter reaches zero. Now, the critical path is reduced to a single adder. Thus, the final structure proposed can work at a higher frequency and provide a reasonable throughput.

3.3 Implementation Results

The Wiener filter is designed in the VHDL language. Modelsim6.2 is used as a tool for behavioral simulation. The design implementation and synthesis are achieved using Synopsis Design Compiler and Innovus 20.11 with UMC NAND Gate 45 nm technology. The implementation is also done for Xilinx Vivado 16.1 to estimate area results regarding LUT and DSPs on an FPGA. The Zync evaluation board, i.e., Zync ZedBoard (xc7z020clg484-1), is the target device. For verification, the video data are streamed to the ASIC implementation of the Wiener filter. The architecture is tested in testbench with matrices produced by the AV1 software model and the outputs are compared with the outputs of the AV1 model. The FPGA-based implementation is just performed to compare resources against state-of-the-art fairly. For simulation, the video is sent to the ASIC using a script to convert the video to pixel values, which are then sent as matrices. It is done via dual-port static RAMs, which act like a buffer. The output from the filter is compared in the testbench with that of the AV1 model. The video can also be streamed to the FPGA using the Vivado video library as a sequence of frames. Each frame is treated as a matrix input to the Wiener filter.

Table 3.1 Timing and area results for the restoring divider.

Divider	**Frequency [MHz]**	**Latency [µs]**	**Area [µm^2]**
Combinational	15.15	29.91	87,258.107
Restoring	153.6	1.59	3426.346
RD1 [74]	-	577	740.44
RD2 [75]	-	-	2799

3.3.1 High Speed Architecture

The proposed architecture is tested against the AV1 software model. The authors report the error in terms of Peak Signal to Noise Ratio (PSNR) values in [27]. The results reported here include timing, area, and power reports. The results are reported in table 3.1 for the divider implementation. The primary combinational divider has a low frequency and high area consumption, while the RD is better in both aspects. Compared to the dividers in literature [74, 75], this work has timing results better by

a factor of 100, while it suffers in terms of area. It is due to the design of a 64-bit integer divider while the one in [74] is a 12×12 array divider and [75] is a 16-bit divisor.

The overall resource, timing, and area are shown in table 3.2. The solution in this work is referred to as High-Speed Final Architecture (HSF). The results are provided for both post-synthesis and post-place-and-route. The die aspect ratio is set to 1.0×0.6 with 5 µm set as die margins. The fixCap and fixTran are kept accurate for the clock setting, with a 10 ns period to be satisfied by the tool. The results show that the final post-Place-and-Route optimized version has a maximum frequency of 100 MHz. The power consumption of the implementation is 1011 mW. The area is higher as the final architecture is extensively unrolled for parallel execution to target high performance. The ASIC and FPGA implementation layouts are displayed in figures 3.10 and 3.11, respectively. The post-Place-and-Route timings for the technology corners, i.e., fast and slow, are 9.95 ns and 24.128 ns, respectively. The fast corner consumes 1519 mW of power while the slow one consumes 1196 mW. The fast one consumes more power as it works at a higher frequency.

Table 3.2 Area, timing, and power results for the Wiener filter ASIC implementation.

High Speed Architecture	**Clock [ns]**	**Frequency [MHz]**	**Area [mm^2]**	**Power [mW]**
Post Synthesis	13.29	75.24	1,988,927.227	12.816
Post Place-and-Route	10.03	99.7	1,966,505.5	1011

Table 3.3 displays the timing and area results of the proposed solution for the FPGA implementation. The clock cycle information is extracted from the simulation for the latency result. The product of clock cycles and clock frequency obtains the latency. Regarding timing, the proposed Wiener filter performs 1000 better against state-of-the-art CPU- and FPGA-based solutions. However, the price is paid in terms of area. It consumes more DSPS, LUT, and FFs than other FPGA-based solutions. But it is better against software and hardware solutions for latency at the cost of 10 times higher area consumption. The LUT and DSP area results were obtained with the help of the Xilinx Vivado tool to have a reasonable comparison for the proposed architecture. A better parameter for comparison is the product of Area-Latency (A-L), which is obtained in terms of FPGA cells consumed. As one FPGA cell contains two LUTs and two FF, half of the maximum from LUT and FF

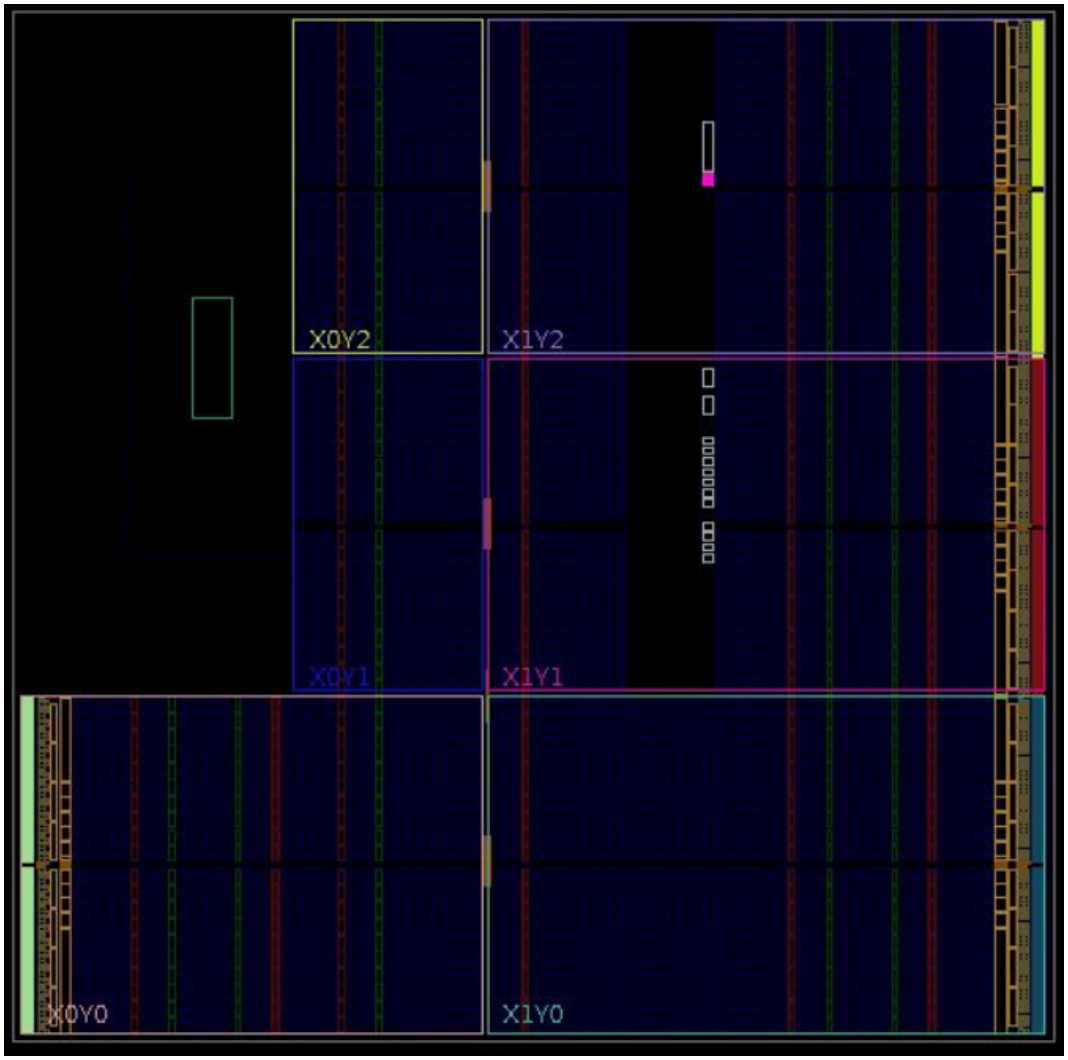


Fig. 3.10 FPGA layout of the Wiener filter.

is considered as the number of cells for this calculation. This is because each cell has two FFs and two LUTs. Therefore, A-L is two orders of magnitude better than the other solutions. Thus, the area penalty is rectified here. In addition, this solution is for a 3-times-higher resolution, as shown in Table 3.4, with four times more bit precision. Hence, high resource consumption is justifiable. Finally, the frequency determines the latency; a low frequency means a high area-latency product, resulting in a poor solution. A higher frequency decreases overall latency, thus decreasing the area-latency product, giving a better solution. For Wiener Filter coding the average time required per call in the AV1 model as per profiling is 490 ms while the proposed architecture can perform this in 1.59 ms. If instead of the VHDL, HLS solution was

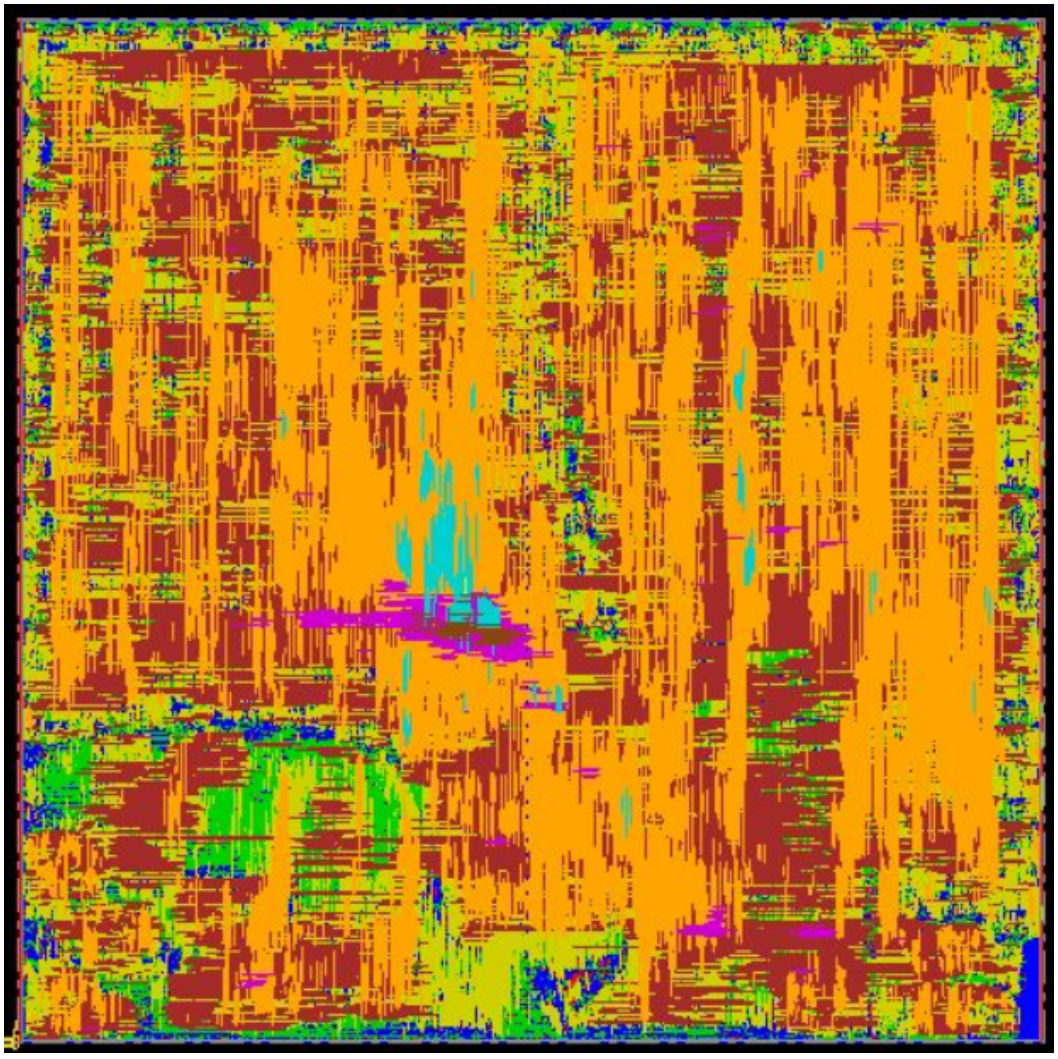


Fig. 3.11 Post place-and-route layout of the Wiener filter.

used to implement the Wiener filter, then the results would have been impacted as follows. In terms of frequency, the HLS solution will not be able to achieve a higher frequency as it would be harder to direct HLS to optimize the critical path [76]. In terms of latency, the HLS pragmas of the pipeline also unrolling the architecture in the scope in which it is applied will not help attain the desired latency in each case. Regarding area, if HLS is directed to implement a simple multiplier in LUTs, it will most probably use the DSP48Es inside the FPGA. But in terms of programmer effort and time to code, the HLS solution will be significantly faster than the VHDL solution.

Table 3.3 Timing and area results for the Wiener filter.

Arch	Device	Freq	Latency	Resources			Area-Latency
		[MHz]	[ms]	LUT	FF	DSP	$[\frac{\text{cells}}{\text{sec}}]$
HSF	ASIC / ZedBoard	99.7	0.00159	565,181	27,423	166	0.449
WF [77]	Intel CPU B830	1800	406.7	-	-	-	-
WF [78]	ZedBoard	100	10	3912	4109	14	20.545
WF [79]	Zynq 7000 SoC	150	4	-	-	-	-
WF [80]	Spartan 3E	-	25	6721	6186	16	84.013
WF [81]	Virtex-5	89	-	8360	2385	13	-

Table 3.4 Video sequences fps.

Architecture	Device	Resolution	fps	$\frac{\text{Megasamples}}{\text{sec}}$ [1]
High-Speed Final	ASIC	720×480	254.452	87.94
		720×576	211.864	87.86
		1280×720	95.328	87.85
		1920×1080	42.372	87.86
WF [78]	FPGA	512×640	-	3.2
WF [79]	FPGA	256×256	250	16.4
WF [81]	FPGA [2]	32×32	4541	4.6

[1] The $\frac{Megasamples}{sec}$ reported are calculated by taking the product of fps and resolution.

[2] The fps reported is extracted from the latency information and sample size given in the article.

3.3.2 Elaboration for a Real-Time Video Sequence

For further analysis of the results, the effect of the obtained improvements has been measured for a target real-time application. By evaluating the throughput, computed as the number of samples processed per second, it is possible to define how many frames can be elaborated in real time for a specific target application. This analysis is conducted for the final high-speed architecture. To better understand, the best-known video formats were analyzed: SD and HD; many resolutions were analyzed for approximated fps to measure the implementation speed. A better parameter for comparison is mega samples per second ($\frac{M_{samples}}{sec}$), which is the product of frame size (height $\times$ width) and the reciprocal of latency. It also takes into account the resolution of the frames. Both fps and $\frac{M_{samples}}{sec}$ are reported in Table 3.4. The fps

of [81] is much better but at a minimal resolution. Regarding $\frac{M_{samples}}{sec}$, the proposed solution outperforms all the others in the literature by a factor of 5. Table 3.4 shows that the proposed architecture can sustain very high frame rates for SD and HD video resolution.

3.4 Conclusions

This work provides an algorithm-to-architecture mapping for the Wiener filter of AOMedia AV1 video coding. The AV1 codec is analyzed, and the Wiener filter is identified as the critical block. The complexity of the Wiener filter is considered in detail for an optimized solution. A separable and symmetric algorithm is considered for its implementation. This allows for fewer computations and reconstruction of the other filter coefficients from the computed ones. The implementation is broken into a series of steps, namely, *Enforcement*, *Partial Pivoting*, *Forward Elimination*, and *Back-Substitution*. A high-speed implementation for better speed is explained to be compatible with many applications. Also, it is well within reach to efficiently exploit the architecture to improve the working frequency. Thus, in terms of throughput, the solution is much better than the state-of-the-art. This is due to the usage of highly parallel architecture and inline RDs. The design choice presented in this work aims to achieve a special-purpose application coherent in terms of data, parallelism, and operations with the C implementation of the Wiener filter [82]. Future works include the implemented filter's overall power analysis relative to the literature.

Chapter 4

RISC-V survey and implementation for video processing

4.1 Literature Overview

In this section, the background and selected articles on RISC-V are presented. In the literature, there are many implementations of RISC-V. This work serves as a summary of them for video processing tasks. A Google Scholar search of RISC-V implementations in hardware, i.e., ASIC or FPGA-based, is performed as a starting point. Among the results, the papers that presented or discussed the RISC-V core implementation were selected. The main comparison parameters used are frequency, area (resources in the case of FPGA), power, and Instructions Per Cycle (IPC), as they are provided for most implementations. The ones that don't provide at least one among frequency, area, or power results were discarded. Next, the main features are highlighted and divided into four categories for comparison. In addition, this work also analysis LEN5, an OoO RISC-V core for academic research being developed at Politecnico di Torino.

Before diving into the literature, a brief introduction to RISC-V unprivileged ISA is provided here [83]. The base ISA RV32I, RV64I, RV64E, and RV128I contain the arithmetic, load/store, memory, branch, jump and register instructions. Other ISA extensions include M (Integer Multiplication and Division), A (Atomic), Zicsr (control and status register), F (Floating point single precision), D (Floating point double precision), Q (Floating point quad precision), C (compressed), B (Bit

manipulation), V (Vector operations). Finally, the ISA extension G, called general, includes RV32I or RV64I with IMAFD, Zicsr, and Zifencei.

Before analyzing the cores for video processing, let's first consider the parameters that affect the speed of video processing algorithms in the processor core. The details are provided in [84]. As a given, the higher number of processing cores would mean more parallel processing power, thus positively affecting the performance. Multi-core processors can distribute the tasks among many cores and achieve higher throughput through parallel execution. The operating frequency is another critical aspect in this context. It directly affects performance. The tasks in video encoding, like transform and quantization and in-loop filters, are compute-bound. They can finish earlier on a fast processor or if many tasks are executed in more cores simultaneously. Thus, they can take advantage of this solution, but a limiting factor of high frequency and multi-core solutions would be the dependencies and scheduling of the tasks. Typically, for small power-efficient cores, there is in-order execution, while for high-performance cores, there is OoO execution. Task scheduling, in the latter case, has high significance as it can improve performance by exploiting the instruction level parallelism (ILP). In real-time systems, the responsiveness of the systems depends on latency. Latency masking can be done with the help of pipelining and pre-fetching from memory. The IPC defines the throughput in the case of processors. The higher IPC will help achieve a higher frame rate in video processing. Further, a large amount of memory size will help accommodate high-resolution videos. Also, high-speed memory will enhance processing performance. In this context, using a cache hierarchy will allow further fast access to data and enhance the speedup. Finally, power is an important parameter as it defines the battery life. Power consumption is of utmost importance for remote embedded cores utilized for surveillance. The low area can reduce power at the cost of performance. Ultimately, performance, flexibility, and power cost are a trade-off.

Other approaches can be considered to provide a better solution for video processing, including the improvements in processors above. It includes using dedicated hardware for complex tasks and adding special ISA instructions for target applications. RISC-V easily accommodates both of these. Furthermore, it is very flexible for new ISA to be included above the base RV32I instruction set. Also, the modularity allows the use of dedicated hardware units for speedup.

The selected cores are BOOM [42], ISP [85], RSD [86], OPA [86], Rocket [87], Ariane [88], Shakti [89], RI5CY [90], preDRAC [91], mRISC-V [92, 93], Low Power core [94], DSP core [95], PicoRV32 [96], ORCA [97], VexRiscv [98], Freedom [99, 100], Taiga [101], Adept-3/5 [102]. Also in addition LEN5 [9–11] is analyzed. The details of these cores are presented below:

BOOM: One of the most used implementations of RISC-V in many academic kinds of research is BOOM [42]. It is an OoO superscalar processor that can be parameterized as per need. The ISA is RV64IMAFD with a 6-stage pipeline. It supports Floating points, virtual memory, and atomic instructions. An updated BOOM version is given in [103]. It involves separate integer and floating point register files, a large Branch target buffer, and separate issue logic for memory, integer, and floating point instructions. Thus, it allows more opportunities for research, including branch-prediction studies and multiple issues. OoO processors are compared in [86] that utilizes BOOM with RV64GC / RV32IMAC ISA. Also, a comparison between many open-source RISC-V cores and BOOM is given in [88, 89] that utilizes BOOM with MAFDC.

ISP: It is an in-order fetch, OoO execution, and in-order completion RISC-V processor. The ISA is RV32I+F with a 5-stage pipeline. The authors [85] present it as an FPU-based coprocessor. It also involves the hazard resolution units.

RSD: It is an OoO RISC-V core with support for speculation provided by the authors of [86]. Speculation allows for higher IPC by providing more instruction to be executed. It is also equipped with memory dependence predictors. It supports the base RV32IM ISA for integers.

OPA: It is an OoO RISC-V soft core optimized for FPGA [86]. It supports the base RV32IM ISA without division and CSR instructions.

Rocket: Rocket [87] is a 5-stage scalar in-order RV32/64G RISC-V core with privilege and virtual memory support written in Chisel. It can also activate other ISA extensions. For example, the front end complies with a branch prediction unit consisting of a Return Address Stack, Branch History Table, and Branch Target Buffer. Also, support for Level-1 (L1) cache, virtual memory, floating point, and coprocessor interface is provided. Among the many Rocket implementations in the literature, a comparison of OoO core with Rocket is provided in [42], and the authors

in [89] instantiate Rocket for both ASIC and FPGA to have a comparison with other similar cores. The authors in [104] instantiate Rocket with ISA RV32IMAFD / RV64IMAFD for low-power applications. An FPGA-based implementation with ISA RV64IMAC for comparison against a softcore is provided in [101]. Finally, the Rocket cores for YOLO with customized instructions and DNN applications are provided in [105] and [106], respectively.

Ariane: CVA6/Ariane [88] is a 6-stage single issue 64-bit in-order implementation of RISC-V. It is SystemVerilog based with RV64GC support. The other ISA extensions can be activated as well. The front end has cache, Branch prediction, and TLB support. Ariane also supports the SV39 virtual memory architecture. To fill the 6-stage, it takes the help of a branch predictor. In [41], the authors provide a Linux booting Ariane RV64GC implementation. In [107], the authors use Ariane RV64GC to reconfigure FPGAs partially. Finally, the authors in [88, 89] instantiate Ariane RV64MAFDC for both ASIC and FPGA to compare with similar cores.

Shakti: Shakti C-Class [89] processor is a 5-stage in-order implementation of RISC-V. It is SystemVerilog based with RV32I and RV64I support. The other ISA extensions can be activated as well. The front end has a cache and branch prediction (GShare two-level). Shakti-E [108] RISC-V Processor Core is RV32I in-order 3-stage RISC-V with memory protection unit.

RI5CY: RI5CY is a RISC-V in-order 32-bit 4-stage pipeline core [90]. The authors in [90] use RI5CY as the processing core for MPSoC with RV32I ISA + DSP extensions. It also has 32 vector interrupts. The authors in [99] evaluate Pulpino RI5CY with RV32IMC against similar cores. RI5CY is the base core for many applications in [109–112].

preDRAC: It [91] a RISC-V CPU capable of supporting the Linux operating system called preDRAC is presented. It is a 64-bit 5-stage single-core RV64IMA in-order RISC-V core with branch predictor and 2-level caches.

mRISC-V: mRISC-V [92, 93] is a RISC-V Processor Core with RV32IM ISA. It is used inside a microcontroller as the main master of the system.

Low-Power core: The work in [94] presents a micro-architecture with low power consumption. It comprises a 4-stage pipeline RV32I with a customized multiplier, divisor, and branch predictor.

DSP core: It is an in-order 4-stage RV32IMFC RISC-V [95]. It is designed for DSP applications in near-threshold tightly coupled memory clusters.

PicoRV32: PicoRV32 is RV32IMC 3/6-stage core [96, 113]. It consists of interrupt and coprocessor units. The articles [96, 104, 113, 114] implement PicoRV32 with RV32IMC 1-stage, RV32IMC 1-stage, RV32IM 3/6-stage and RV32IMC, respectively. The article [96, 104, 113, 114] discusses its comparison with other cores and concludes that it is a small and efficient core that operates at high frequency.

ORCA: ORCA [97] is RV32I/M 5-stage cached RISC-V core. It is used for comparison in articles [104, 111, 113] with ISA RV32I/RV64I, RV32IM and RV32I, respectively. It shows a relatively higher frequency. ORCA is used to study clock gating in [115].

VexRiscv: VexRiscv [98] is RV32I 2/5-stages RISC-V softcore that can be extended for M and other ISA extensions but written in SpinalHDL. It can also be configured for Linux-running systems with the assistance of MMU.

Freedom: Freedom [99, 100] an RV32IMAC single issue 5-stage in-order core from rocket-chip. It is also used for comparison in [111] with ISA RV32IMAC.

Taiga: Taiga [101, 113] is the first attempt at variable pipeline softcore on FPGA. Taiga is a 32-bit RV32IMA RISC-V core. It is configurable and supports shared memory and Linux with parallel execution units. The modification in [116] also allows for OoO commits.

Adept: Adept [102] is a RIC-V core written in Chisel3. It has two versions: a 3-stage pipeline with RV32I ISA and a 5-stage pipeline with RV32IM ISA.

LEN5: LEN5 [9–11] is an OoO execute and in-order commit processor supporting floating-point, virtual memory, and 2-level cache hierarchy. It implements RV32/64I with support for MC and is also extendable for customs. It has a 6-stage pipeline. For further details, see section 4.3.

4.2 Comparisons

In this section, the previously mentioned articles are compared for their architectural implementation. The results are divided into four parts. The primary division

involves in-order and OoO cores, further divided into FPGA and ASIC implementations. The results are provided as given in the articles. The result comprises core technology, given in nm, for ASIC, while the device's name is displayed for FPGA. Then, the implemented ISA is provided along with operating frequency in MHz, power in mW, number of pipeline stages, and the instructions per cycle (IPC). For ASIC, the area is provided in mm^2, while for FPGA, it is provided in terms of several resources, i.e., LUTs, FFs, BRAMs, and DSPs. In the tables shown, F means FPGA, and A means ASIC.

4.2.1 RISC-V in-order Processors with ASIC implementations

The in-order RISC-V cores AISC implementations are presented in table 4.1. The pipeline range is from 4 – 6 stages. The technology solutions range from 28 nm to 130 nm. The IPC is only provided for limited implementations. The highest IPC is achieved by Rocket core in [42] along with the highest frequency of 1.5 GHz and reasonable area. But it is only for RV64 ISA. The cores in Rocket [87, 89], Shakti [89, 108] and preDRAC [91] achieve moderate IPC.

The comparison is performed in terms of area, power, and frequency. The best solution for video processing is one with low power, low area, and high frequency. The more ISA implementation, the better, as the floating point operations and division are used in video encoding. High frequency is achieved by Rocket [87, 89], Ariane [88, 89] and Shakti [89, 108]. Then moderate frequency is displayed by DSP core [94, 95], RI5CY [90], preDRAC [91] and Low power core [94]. preDRAC [91] has the highest area followed by Shakti [89, 108] and Ariane [88, 89]. preDRAC [91] also has the highest power followed by Ariane [88, 89], Shakti [89, 108], Rocket [106].

The power, frequency, and area results are shown in figures 4.4, 4.2, and 4.3, respectively. Figure 4.1 shows the combined area, power, and frequency triangles for the in-order cores. The area and power axis are changed to a logarithmic scale for better visibility. The cores with high-performance are Rocket [87, 89], Ariane [88, 89] and Shakti [89, 108] with moderate area and power. The best among them is Rocket [87, 89]. In the moderate range, the best is DSP core [94, 95] with high frequency, low area, and moderate power.

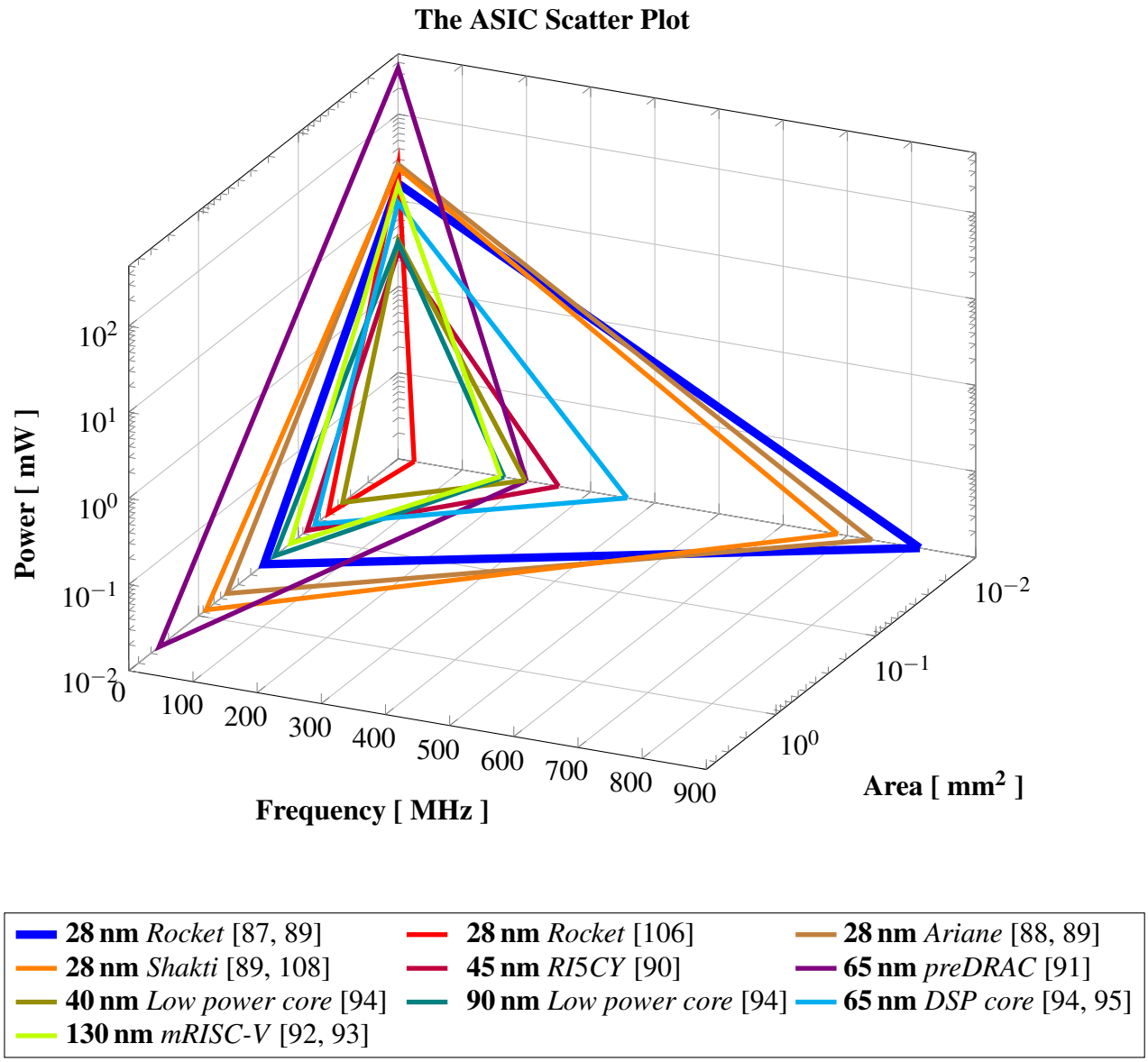


Fig. 4.1 The ASIC in-order cores for Video Processing

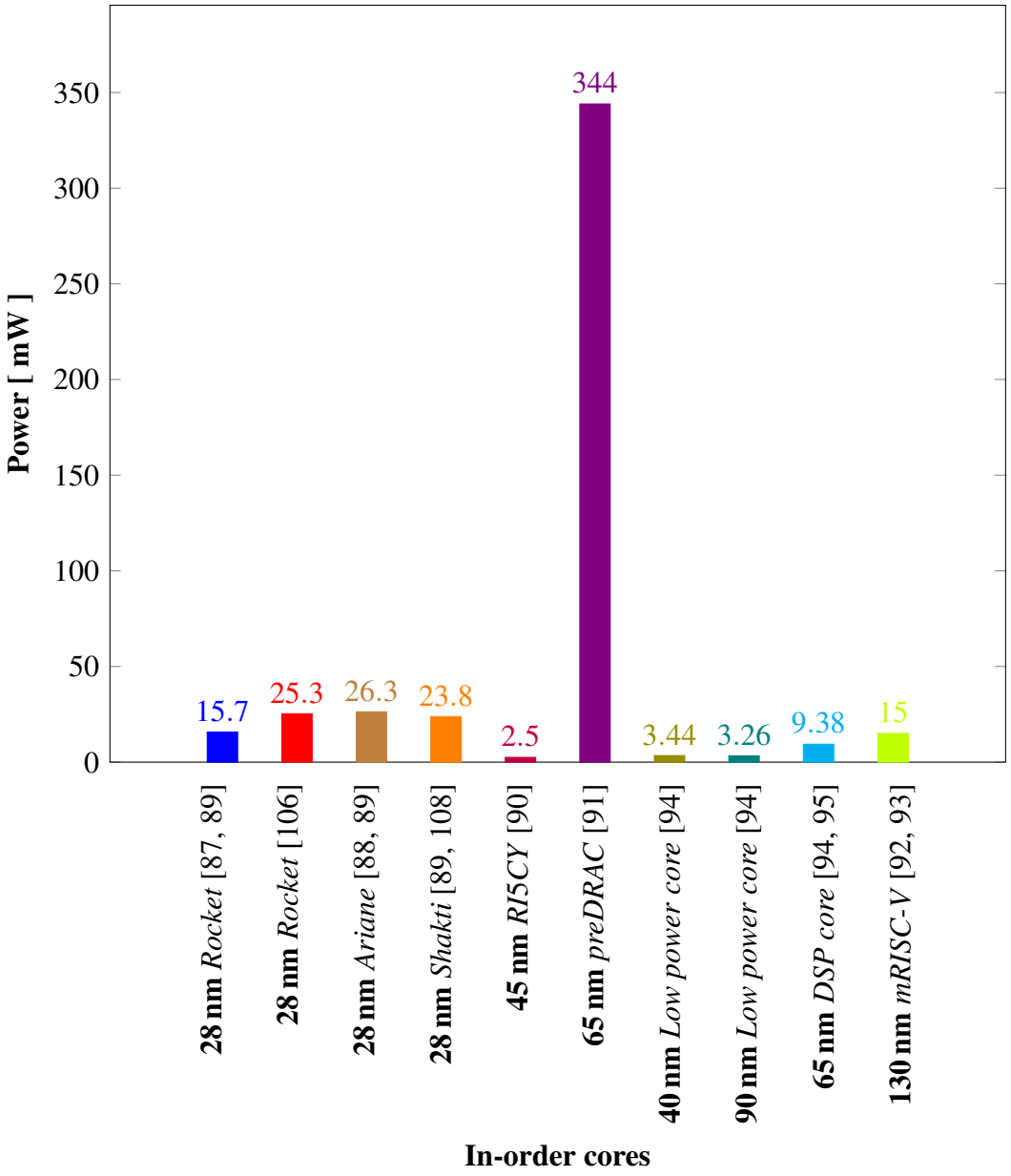


Fig. 4.2 The power of ASIC in-order cores for Video Processing

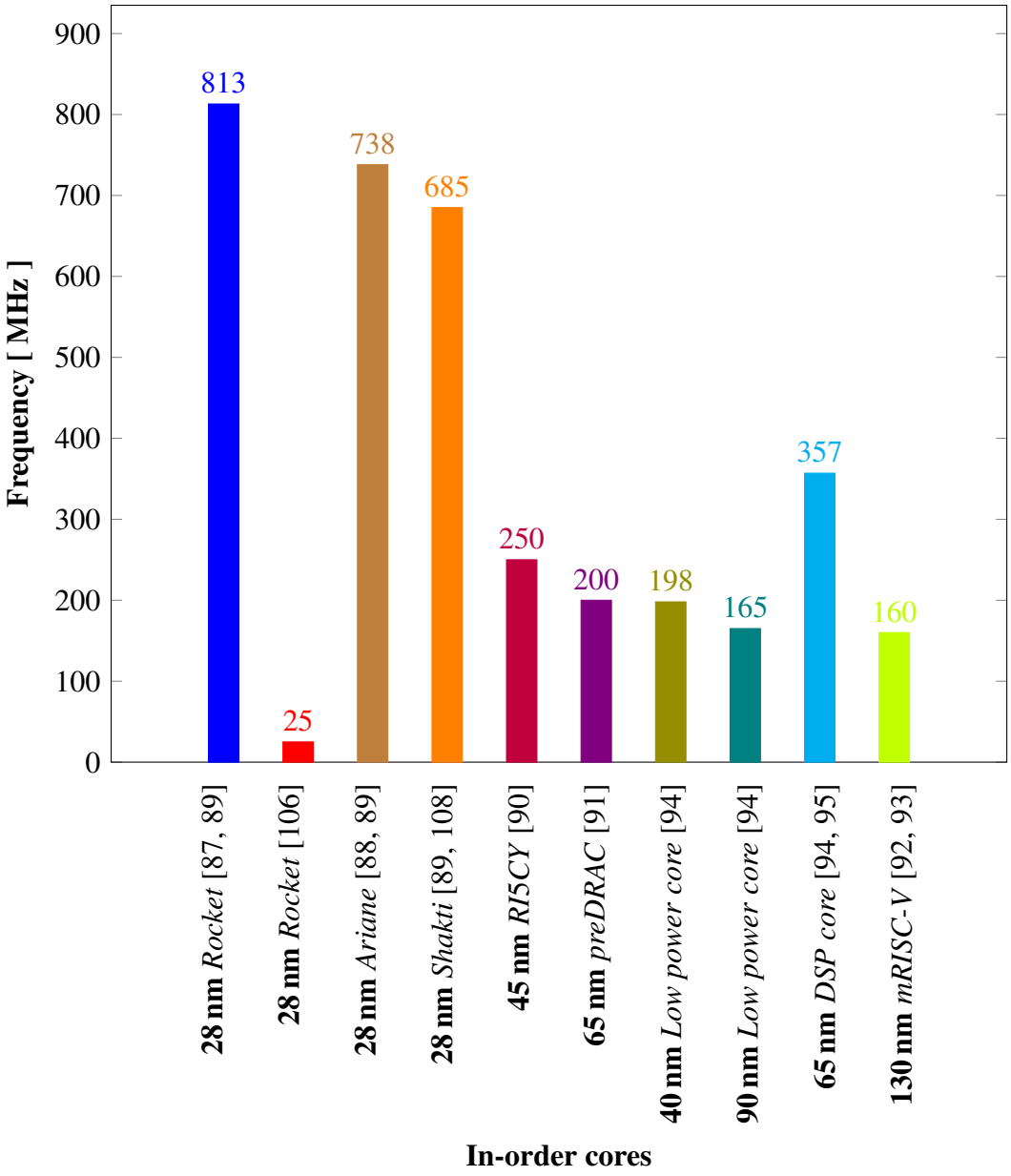


Fig. 4.3 The frequencies of ASIC in-order cores for Video Processing

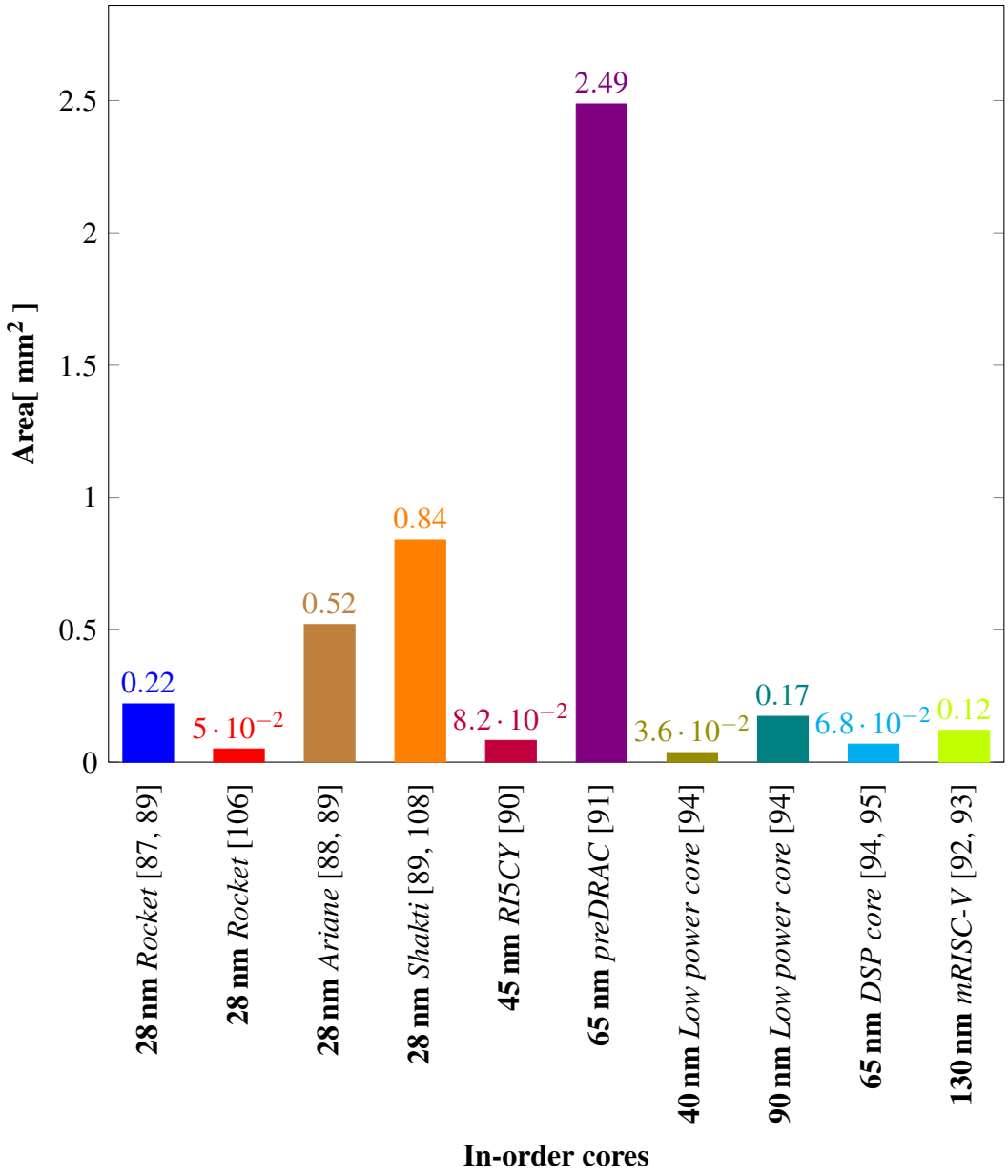


Fig. 4.4 The area of ASIC in-order cores for Video Processing

Table 4.1 RISC-V Processors ASIC implementation results.

Core	T nm	ISA	Freq MHz	IPC	Pipe	A mm^2	P mW
Rocket							
[42]	45	RV64	1500	0.76	-	0.5	-
[87, 89]	28	RV32/64I +MAFDC	813	0.33	5	0.22	15.7
[106]	28	RV32/64IMF	25	-	5	0.053	25.3
Ariane / CVA6							
[88, 89]	28	RV64GC +MAFD	738	-	6	0.52	26.3
Shakti							
[89, 108]	28	RV32/64I +MAFDC	685	0.23	5	0.84	23.8
RI5CY							
[90]	45	RV32I	250	-	4	0.082	2.5
preDRAC							
[91]	65	RV64IMA	200	0.33	5	2.487	344
mRISC-V							
[92, 93]	130	RV32IM	160	-	-	0.12	15
Low power Core							
[94]	40 90	RV32IM	198 165	-	4	0.036 0.173	3.44 3.26
DSP Core							
[94, 95]	65 65	RV basic RV ext	357	-	4	0.068 0.077	9.38 9.52

4.2.2 RISC-V in-order Processors with FPGA implementations

The in-order RISC-V cores FPGA implementations are presented in table 4.2. The pipeline range is from 1 – 6 stages. Many XC7Z020-based solutions are present. The IPC is only provided for limited implementations. The highest is IPC is achieved Rocket core in [87, 89] followed by PicoRV32 [114] and Shakti [89, 108].

The comparison is performed in terms of resources, power, and frequency. High frequency is achieved by PicoRV32 [96, 113] followed by VexRiscv [98, 113], Taiga [101, 113] and ORCA [97, 113]. Then moderate frequency is displayed by PicoRV32 [96, 104], ORCA [104, 115] and Rocket [87, 89]. Ariane [41] has the highest LUT consumption, thus consuming more power, followed by Rocket [87, 89]. Ariane [41] has the highest DSP consumption, thus allowing fewer opportunities for acceleration by adding dedicated hardware for video processing, followed by Rocket [87, 89].

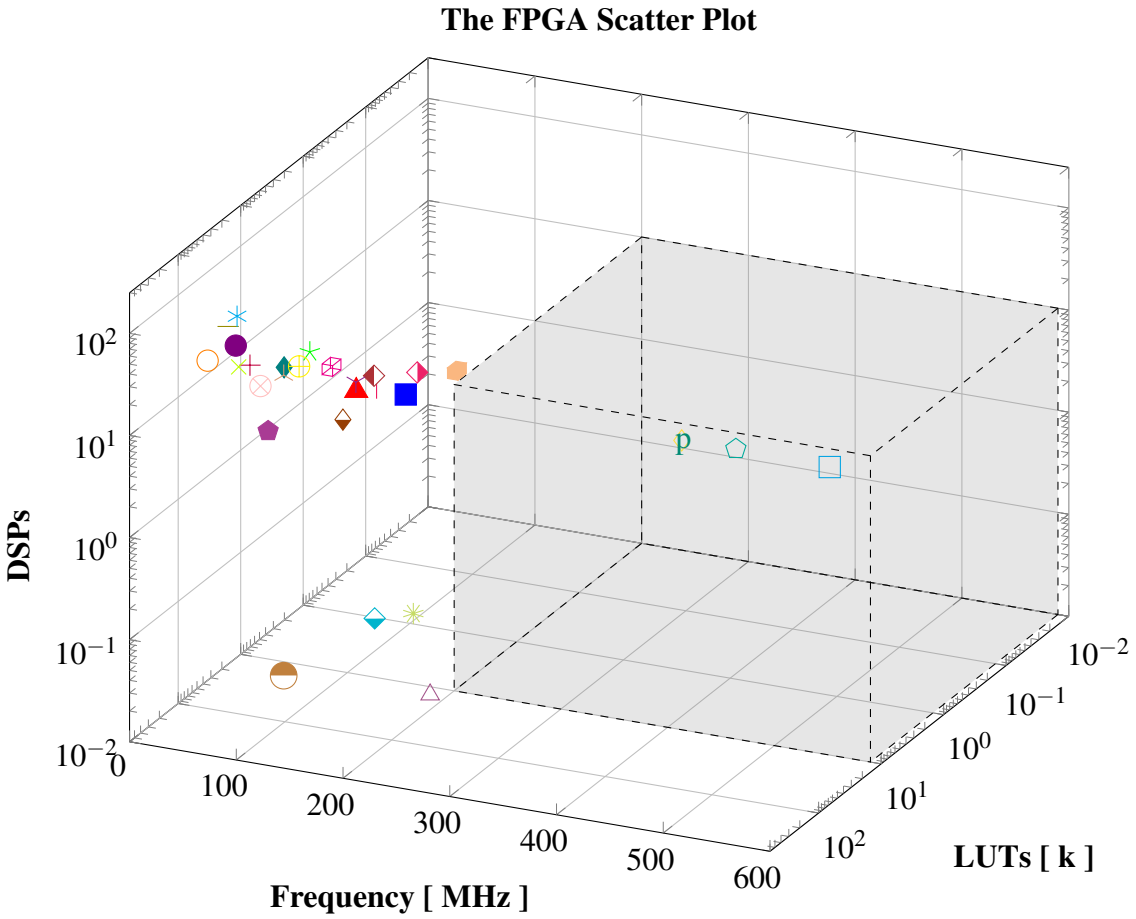
Figure 4.5 shows the in-order cores' LUTs, DSPs, and frequency scatter plots. The LUT and DSP axis is changed to the logarithmic scale for better visibility. The cores with high-performance are PicoRV32 [96, 113] followed by VexRiscv [98, 113], Taiga [101, 113] and ORCA [97, 113] with moderate LUTs consumption and and low DSP usage. The best among them is PicoRV32 [96, 113]. The best is Rocket [87, 89] with high frequency and low resources in the moderate range.

Figure 4.6 shows the power consumption of some cores in the form of a bar graph. The cores with low power are PicoRV32 [96, 104], followed by Rocket [106], ORCA [104, 115] and Rocket [104].

Table 4.2 RISC-V Processors FPGA implementation results

Core	Device	ISA	Freq	IPC	Pipe	Resources				P
			MHz			LUT	FF	BRAM	DSP	mW
Rocket										
[87, 89]	Virtex US+	RV32/64I +MAFDC	198	0.33	5	55k	46k	47	18	1820
[104]	XC7Z020	RV32/64IMAFD	91	-	5	5073	2008	0	4	92
[101]	X7CZ020	RV64IMAC	54	-	-	17144	9058	10	0	-
[106]	Zynq-7000	RV32/64IMF	25	-	5	26.7k	9.7k	-	-	36.8
Ariane / CVA6										
[41]	Artix 7	RV64GC	30	-	6	110k	75k	66	27	-
	Kintex 7		67			99k	75k	66	27	
	2×1		67			167k	120k	124	54	
	Virtex 7		60			114k	77k	63	27	
	3×1		60			268k	169k	179	81	
	Virtex US+		100			103k	84k	89	30	
	4×2		100			583k	399k	495	219	
[107]	Kintex-7	RV64GC	100	-	6	39940	22500	36	27	-
[88, 89]	Virtex US+	RV64GC +MAFD	112	-	6	95k	77k	69	31	1995
Shakti										
[89, 108]	Virtex US+	RV32/64I +MAFDC	136	0.23	5	74k	55k	112	30	1660
[104, 108]	XC7Z020	RV32I	12.7	-	3	7948	1813	0	4	100
PicoRV32										
[96, 104]	XC7Z020	RV32IMC	200	-	3	904	566	0	0	13
[96, 113]	AU250	RV32IMC	530	-	1	4037	3341	32	4	-
	PYNQ					4143	3363	32	4	
	VC709					249	3296	32	4	
	ZCU102					4233	3345	32	4	
[114]	Kintex 7T	RV32IMC	400	0.25	-	2k	-	-	-	-
[96, 99]	XC7A35T	RV32IMC	115	-	-	2123	1509	48	-	-

ORCA										
[104, 115]	XC7Z020	RV32/64I	185.2	-	5	1354	746	1	0	69
[97, 113]	AU250	RV32IM	390	-	4/5	3822	3139	32	4	-
	PYNQ					3917	3100	32	4	
	VC709					4019	3115	32	4	
	ZCU102					3882	3115	32	4	
[111]	-	RV32I	185.4	-	-	1354	746	-	-	-
VexRiscv										
[98, 113]	AU250	RV32IMCA	440	-	2/5	3797	3328	32	4	-
	PYNQ					3777	3305	33	4	
	VC709					3795	3329	33	4	
	ZCU102					3906	3378	33	4	
Freedom										
[111]	-	RV32IMAC	32.5	-	-	2692	1311	-	-	-
[99, 100]	XC7A35T	RV32IMAC	32.5	-	5	13062	7662	6	2	-
Taiga										
[101]	X7CZ020	RV32IMA	104	-	-	3,998	2,942	10	4	-
[101, 113]	AU250	RV32IMA	395	-	V	4515	3239	32	4	-
	PYNQ					4504	3176	34	4	
	VC709					4576	3207	34	4	
	ZCU102					4565	3216	34	4	
[116]	X7CZ020	RV32I	109	-	-	16821	959	1	4	-
Adept-3/5										
Adept 3 [102]	ARRIA V	RV32IM	93	-	3	670	201	133	0	-
	K-7		68			1041	18	133	0	
Adept 5 [102]	ARRIA V	RV32IM	120	-	5	1675	1470	266	4	-
	Kintex-7		90			2533	18	1405	4	
RI5CY										
[110]	Zynq U+	RV32IM	100	-	4	7322	1451	32	6	1180
[111]	-	RV32IMC	50	-	-	6748	2577	-	-	
[99, 109]	XC7A35T	RV32IMC	50	-	4	14616	8959	16	8	-



Virtex US+ *Rocket* [87, 89]	**XC7Z020** *Rocket* [104]	**XC7Z020** *Rocket* [101]
Artix 7 *Ariane* [41]	**Kintex 7** *Ariane* [41]	**2 × 1** *Ariane* [41]
Virtex 7 *Ariane* [41]	**3 × 1** *Ariane* [41]	**Virtex US+** *Ariane* [41]
4 × 2 *Ariane* [41]	**Kintex-7** *Ariane* [107]	**Virtex US+** *Ariane* [88, 89]
Virtex US+ *Shakti* [89, 108]	**XC7Z020** *Shakti* [104, 108]	**XC7Z020** *PicoRV32* [96, 104]
ZCU102 *PicoRV32* [96, 113]	**XC7Z020** *ORCA* [104, 115]	**VC709** *ORCA* [97, 113]
ZCU102 *VexRiscv* [98, 113]	**XC7A35T** *Freedom* [99, 100]	**X7CZ020** *Taiga* [88]
VC709 *Taiga* [101, 113]	**X7CZ020** *Taiga* [116]	**ARRIA V** *Adept*-3 [102]
K-7 *Adept*-3 [102]	**ARRIA V** *Adept*-5 [102]	**Kintex-7** *Adept*-5 [102]
Zynq U+ *RI5CY* [110]	**XC7A35T** *RI5CY* [99, 109]	

Fig. 4.5 The FPGA in-order cores for Video Processing

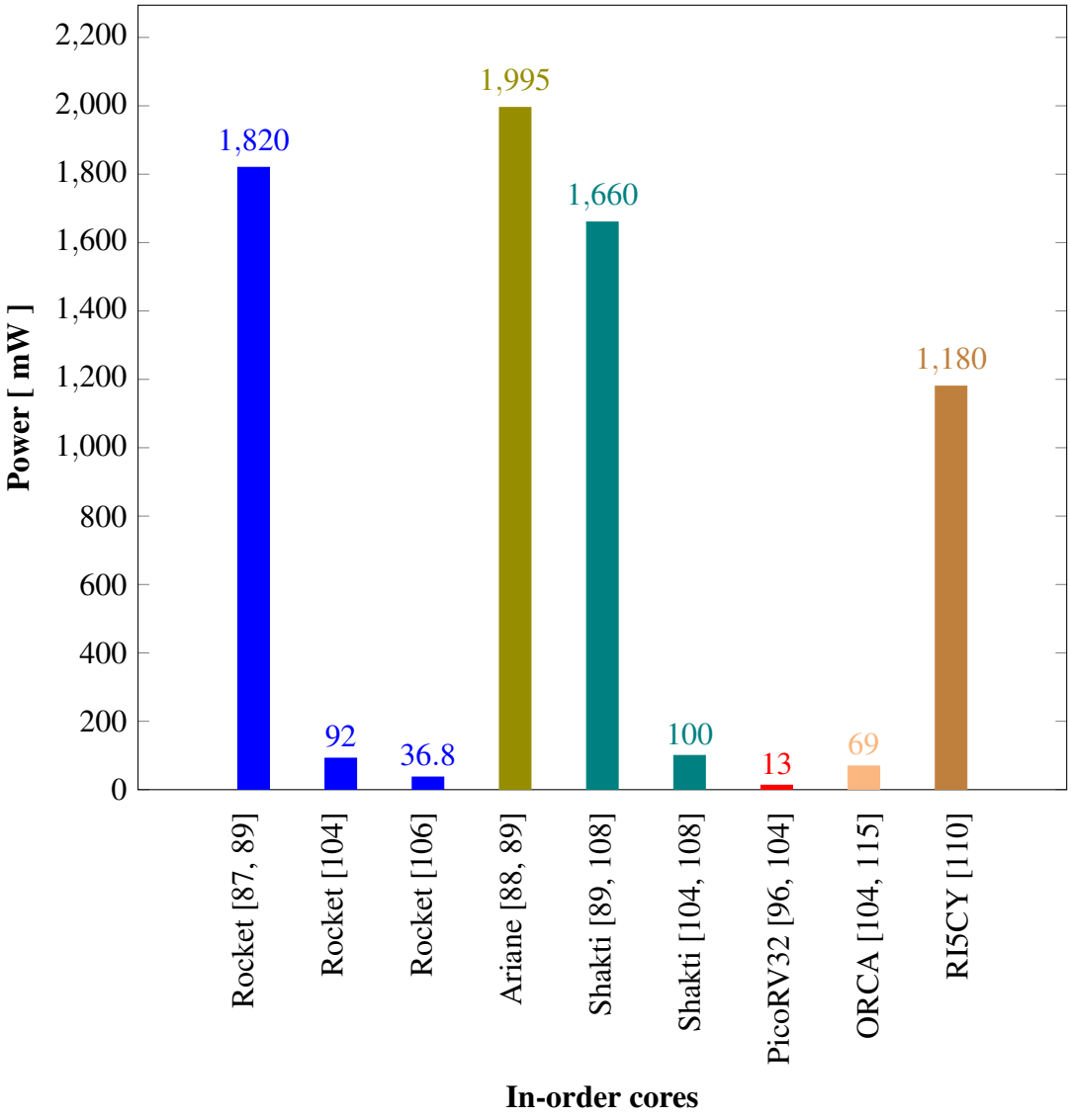


Fig. 4.6 The power of FPGA in-order cores for Video Processing

Table 4.3 RISC-V OoO Processors implementation results.

Core	T	F	Device	ISA	Freq	IPC	Pipe	Resources				A	P
	nm	A			MHz			LUT	FF	BRAM	DSP	mm^2	mW
BOOM													
v4 [42]	45	A	-	RV64	1500	1.5	-	-	-	-	-	1.4	-
v2 [42]	45	A	-	RV64	1500	1.25	-	-	-	-	-	1.1	-
[86]	-	F	XC7Z020	RV64/32 GCIMAC	71.7	-	-	43k	22k	-	-	-	-
[88, 89]	-	F	Virtex	RV64GC	88	0.5	10	261k	123k	43	39	-	3030
	28	A	US+	+MAFD	943			-	-	-	-	0.92	139
ver2 [103, 104]	-	F	XC7Z020	RV64IMAFD	34.7	-	6	49865	25205	6	40	-	188
ISP													
[85]	65	F/A	Virtex 6	RV32I+F	-	-	5	46530	11759	-	32	0.561	-
RSD													
[86]	-	F	XC7Z020	RV32IM	95.3	-	-	15k	8k	-	-	-	-
OPA													
[86]	-	F	XC7Z020	RV32IM	134	-	-	20k	9k	-	-	-	-
LEN5													
[9–11]	65	A	-	RV32/64I	537	1	6	-	-	-	-	0.45[1]	-

[1] The area doesn't include the execution units.

4.2.3 RISC-V OoO Processors with FPGA implementations

The OoO RISC-V cores implementations are presented in table 4.3. The pipeline range is from 5 – 6 stages. Many XC7Z020-based solutions are present. The IPC is not provided for FPGA-based solutions.

The comparison is performed regarding resources and frequency, as power is provided for only two implementations. High frequency is achieved by BOOM [86] followed by BOOM [103, 104], and BOOM [88, 89]. The LUT and DSP consumption is similar. The power consumption of BOOM [103, 104] is high followed by BOOM [88, 89].

Figure 4.7 shows the LUTs and frequency scatter plot for the OoO cores. The cores with high performance are OPA [86] and RSD [86] with low LUT consumption. In the moderate frequency range, the best is BOOM [86] with low resources.

4.2.4 RISC-V OoO Processors with ASIC implementations

The OoO RISC-V cores implementations are presented in table 4.3. The pipeline range is from 5 – 10 stages. The technology solutions range from 28 nm to 65 nm. The IPC is only provided for limited implementations. The highest is IPC is achieved BOOM v4 [42], followed by BOOM v2 [42], and BOOM [88, 89].

The comparison is performed regarding area and frequency, as power is only provided for one solution. High frequency is achieved by BOOM v4 [42] and BOOM v2 [42] with also the highest area. BOOM displays the moderate frequency [88, 89] and LEN5 [9–11] with the high and low areas, respectively.

Figure 4.7 shows the OoO cores' area and frequency scatter plot. The cores with high performance are BOOM v4 [42], BOOM v2 [42], and BOOM [88, 89] but at the cost of the high area. In the moderate frequency range, the best is LEN5 [9–11] with low area and, ultimately, low power. In contrast to in-order ASIC cores, it achieves a higher IPC, with moderate frequency and higher area, thus high power consumption.

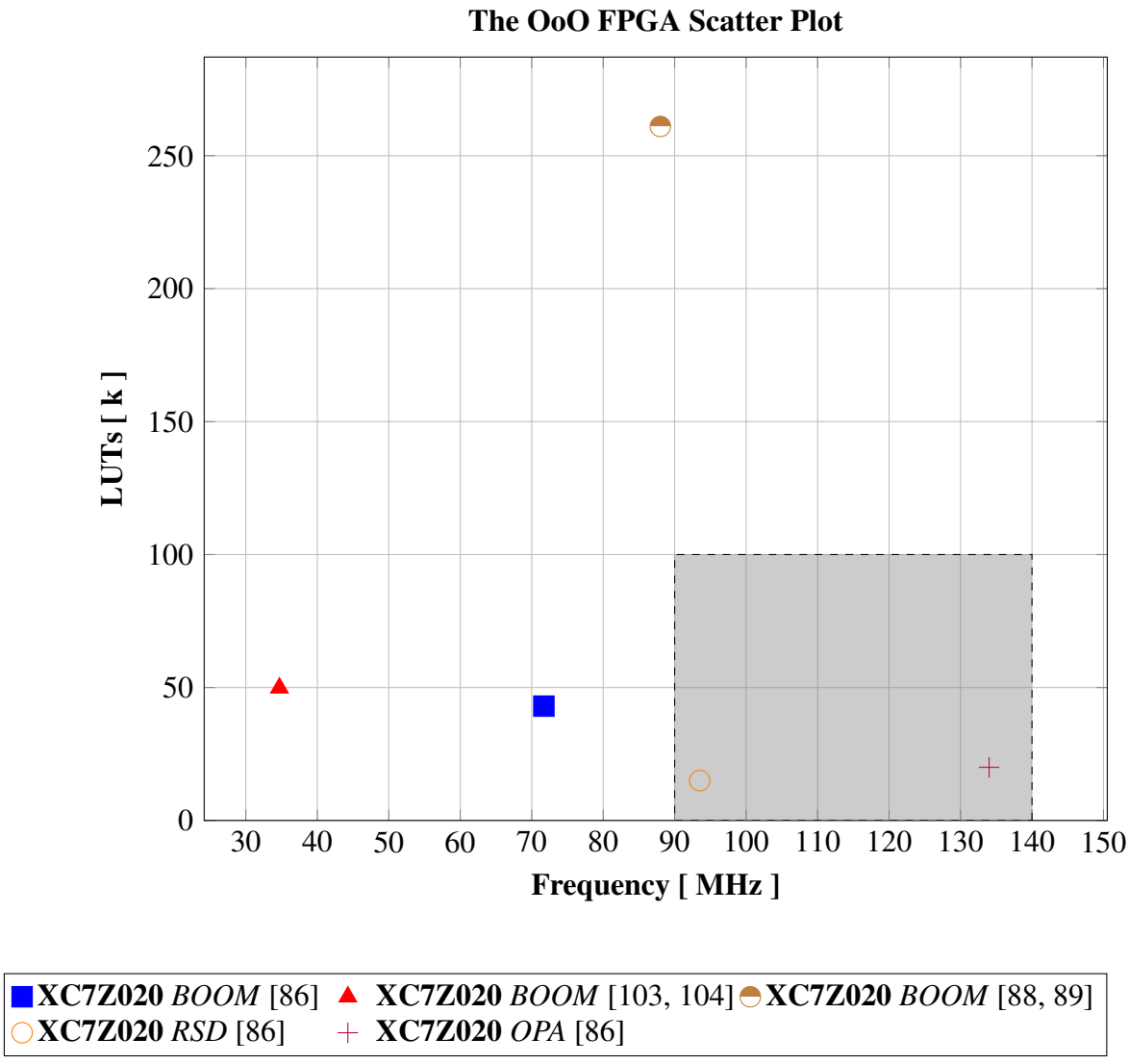


Fig. 4.7 The FPGA OoO cores for Video Processing

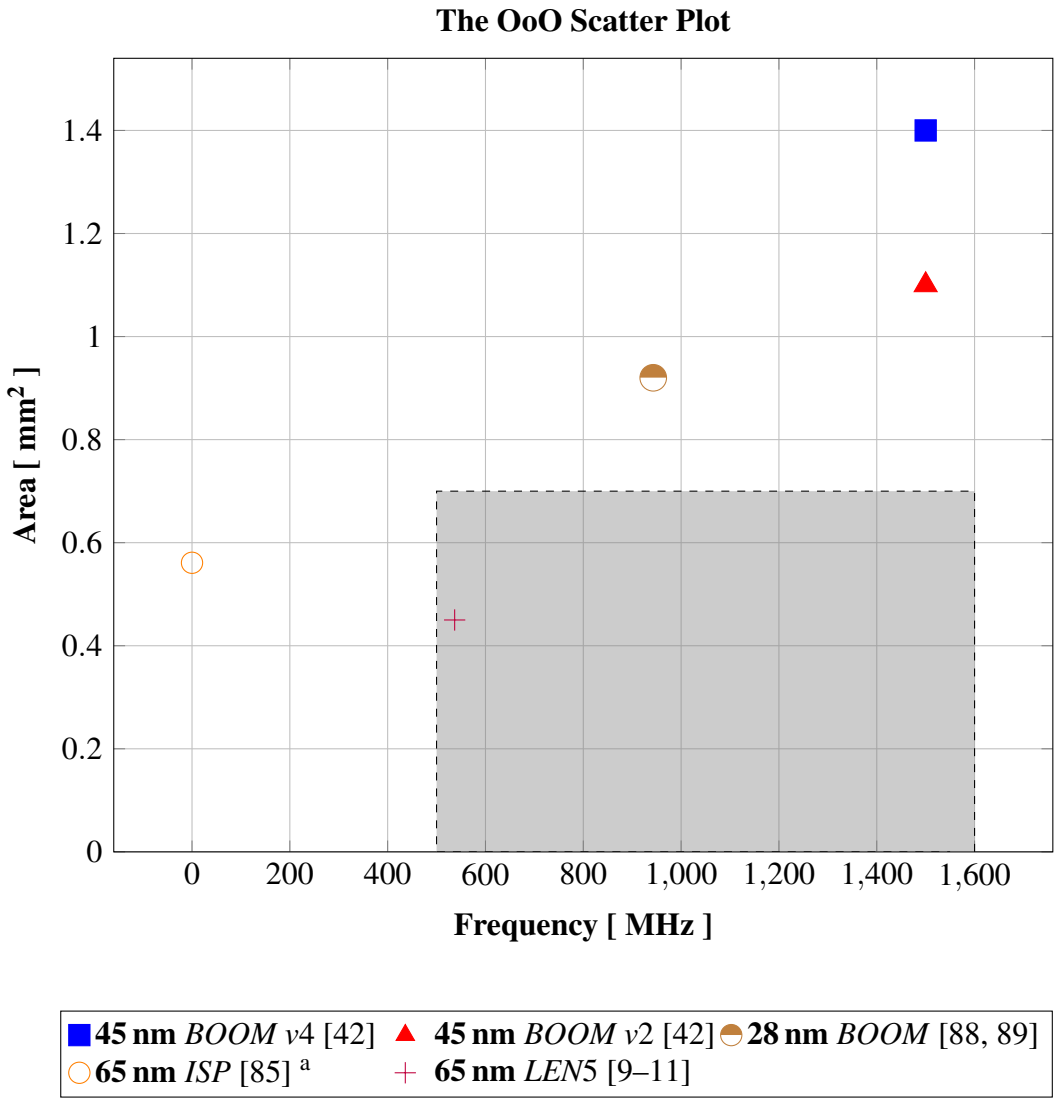


Fig. 4.8 The ASIC OoO cores for Video Processing

[a] No frequency information was provided. The frequency is set to 0, just to show the area results.

4.3 LEN5

As pointed out in section 4.1, cores with high frequency, low area, and power are more suitable for video processing tasks. A memory hierarchy can allow fast access and keep the pipeline feed continuous with new instructions. An OoO core can help achieve a higher IPC. Thus, among the four categories, OoO cores are more suitable. The ones which have preference are the ASIC solutions as they have a significantly higher frequency. In ASIC OoO cores, LEN5 [9–11] has a low area and reasonably high frequency. Thus, it can utilize parallelism in video processing algorithms. It has a frequency 500 MHz, with the lowest area among ASIC solutions, 0.45 mm^2 without the execution units. Thus, the low area makes it suitable for low power. It also has a hierarchical cache to have fast memory access. Thus, from here on, the LEN5 is investigated further, and additions are provided for this architecture.

LEN5 is an OoO execute and in-order commit processor supporting floating-point, virtual memory, and 2-level cache hierarchy. It uses Tomasulu's architecture for OoO execution and ROB for in-order commits. It executes instructions in steps similar to other processors in literature. The stages are the issue of the instruction, instruction decode, fetching of operands, instruction execution or accessing the memory system,

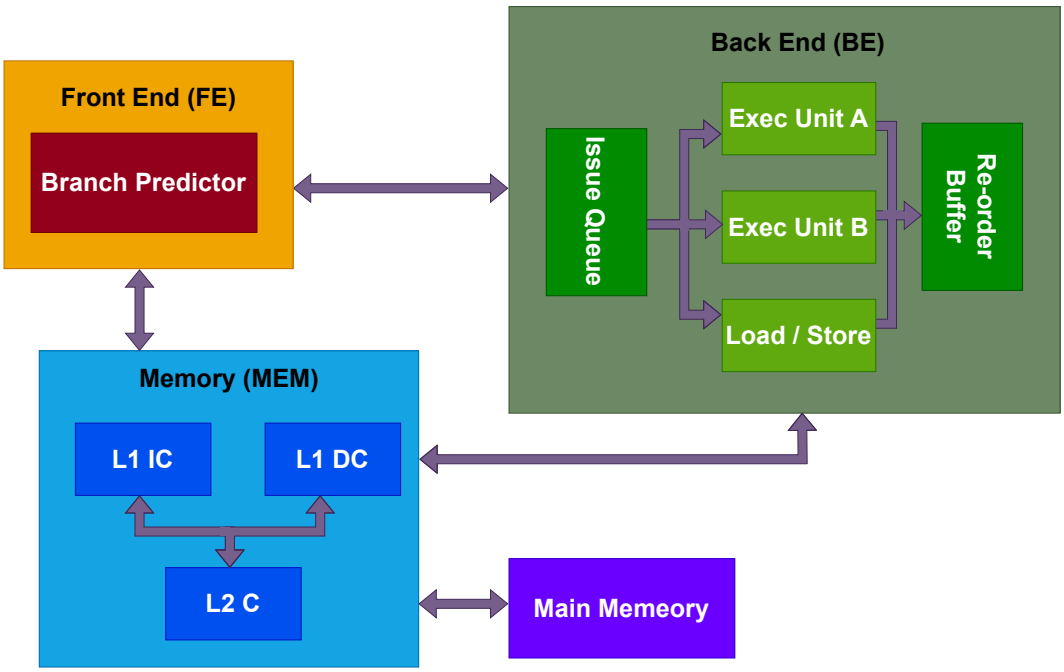


Fig. 4.9 LEN5 block diagram [9–11].

and finally, committing the instruction. Figure 4.9 shows the block diagram of the LEN5. The main blocks in LEN5 are FE [9], BE [10], memory [11]. It implements RV32/64I with support for M and is also extendable for customs ISA. An Advanced eXtensible Interface (AXI)-like handshaking protocol is applied throughout the system for unit communication. The BE structure is based on Tomasulo's approach for dynamic scheduling, with expansion including support for speculation and precise exceptions. The FE concerns the instruction address generation, branch target address prediction, and new instruction fetching from memory.

The instruction execution in LEN5 is as follows. The new instruction address is generated in FE and transmitted to Memory (MEM). The memory consists of two levels of cache. The first level is divided between instruction and data cache. The instruction cache receives the request for new instruction and sends it to the level 2 cache. The new instruction is then sent back to FE. It is then inserted to BE in an instruction queue for decode. After the decode stage, the instructions are sent to execution units via allocated reservation stations. Finally, the results from execution units are sent to the ROB to commit in order via a Common Data Bus (CDB). The results are checked for possible exceptions. If non found, the commit logic updates the results in the register file. This is the whole instruction execution in LEN5.

4.3.1 LEN5 completion

The LEN5 shown for comparison is a work in progress. The authors of LEN5 [9–11] also describe the missing parts in the core. The BE is incomplete as it is missing the commit logic and the execution units. Furthermore, the CDB must be instantiated to all units in BE and generate respective reservation stations for execution units. Apart from this, the three parts BE, FE, and MEM need to be combined with the design of the control unit. Thus, this work is focused on overcoming these limitations and implementing the described units to make a complete LEN5.

The additions are shown in red in figure 4.10. This work provides the additions for RISC-V LEN5. The main additions combine BE into one central unit, thus correctly instantiating the CDB. It also implements the limitations provided by the authors as the commit logic, control unit, and ROB. This work combined the BE [10] into one unit and the RTL description of execution units. Then, the CDB is instantiated to connect all the units. Finally, the whole FE, BE, and MEM are joined

to make the core functional. All these additions are made to the LEN5 core. This work is only preliminary tested.

The commit logic is shown in figure 4.11. Once an instruction is issued from the issue queue, the ROB logic pushes the entry into the ROB data structure. An entry number is assigned to it. Then, it waits for the completion of the instruction by the execution stage. The head and tail counter count the position of instructions in the data structure. When an instruction is completed, and the ROB entry is on the top of the ROB, the instruction is sent to the commit logic for completion. The commit logic is responsible for handling the instruction data update on the register files and the handling of the exception raised in the execution and memory stage. If no exception has occurred and the register file is available, the update signal is used to give the handshake to the register file for update. The control unit is notified in case of an exception, and the code is made to switch to the exception-handling routine. Finally, the commit decoder takes care of all the types of instructions, and the necessary units are communicated with the respective information of committing. It is how the commit is implemented.

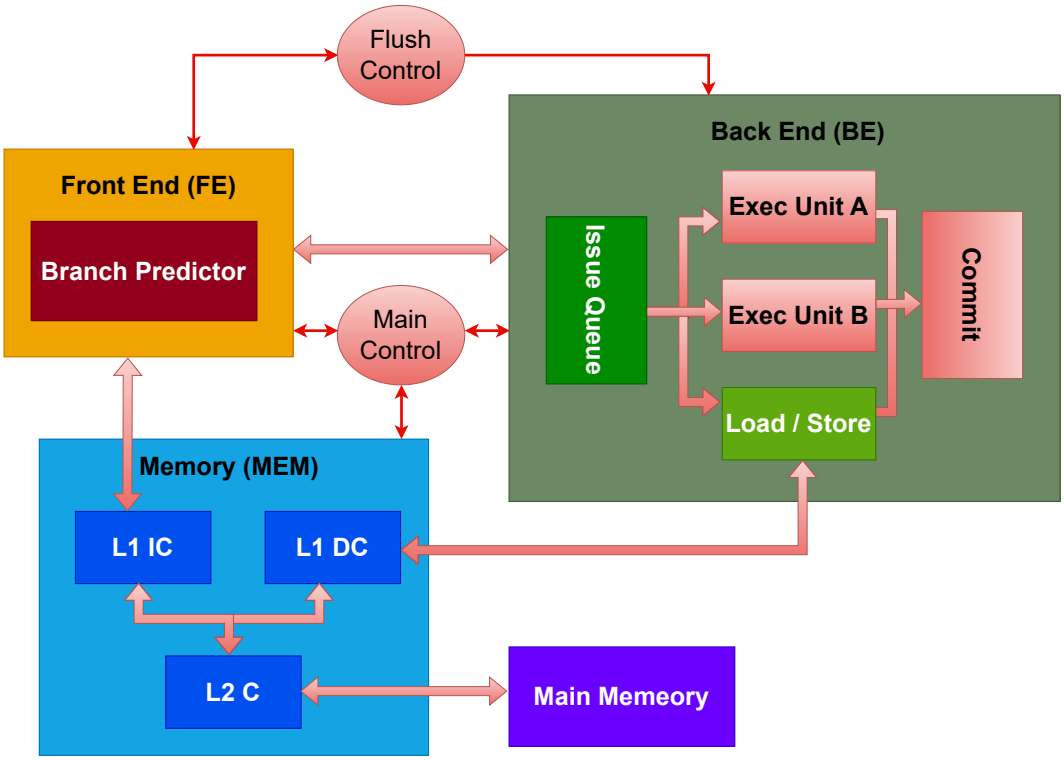


Fig. 4.10 Updated LEN5 block diagram [9–11].

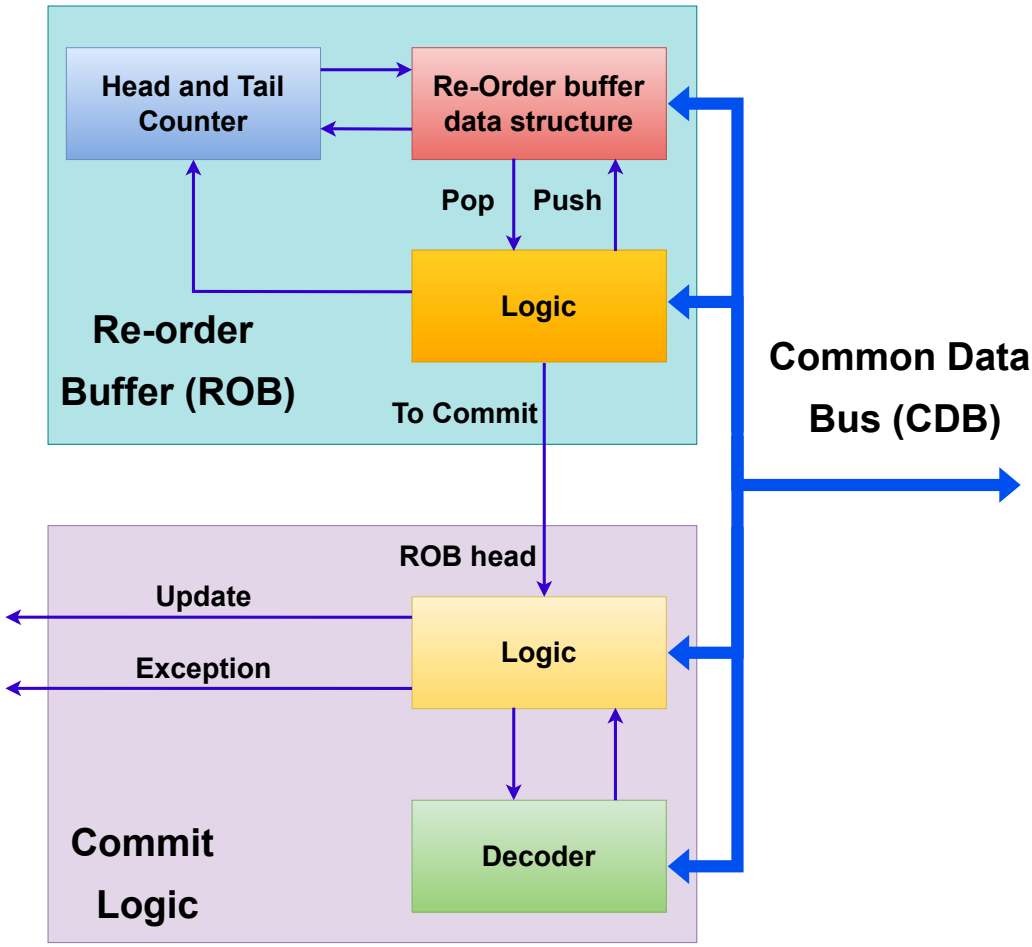


Fig. 4.11 LEN5 Commit Logic.

In the execution units, the RTL design is performed for the ALU for LEN5. It includes all the arithmetic instructions in RV64I ISA and the M multiplication. They are described in a behavioral manner using SystemVerilog.

The CDB is extended to all reservation stations, the issue queue, ROB, and commit logic. They communicate via a two-way, ready-valid handshake protocol. Also, the three parts FE, BE, and MEM are instantiated with the MEM interface. They are controlled via the control unit described next.

The control part deals with maintaining coherency between all blocks and resolving exceptions. The control unit is divided into two parts, as shown in 4.10, the main control and the flush control. The main control manages the whole pipeline and exceptions, while the flush control manages the stall and pipeline flush. The

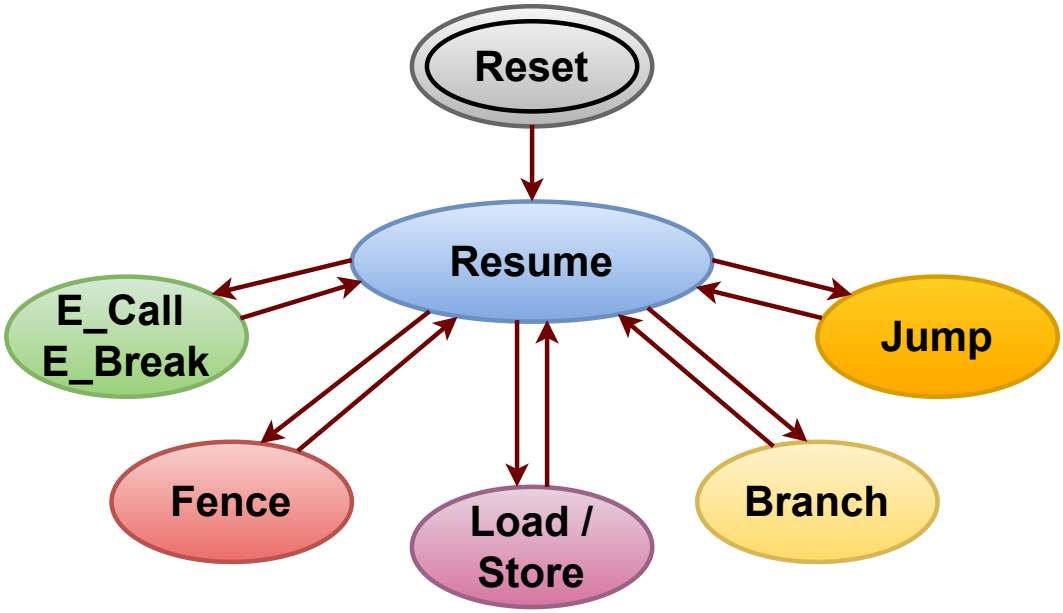


Fig. 4.12 LEN5 Main Control unit.

main control is depicted in figure 4.12. It is a Finite State Machine (FSM)-based control unit. The states are described as follows:

***Reset*:** The first state is *Reset*, responsible for flushing the pipeline and front end. It is also responsible for providing the initial program counter value.

***Resume*:** After the *Reset* state, the control moves to the *Resume* state. It corresponds to the standard pipeline execution for all instructions. It handles the fetch, decode, and execution instructions for all arithmetic instructions.

***Jump*:** The jump instruction is handled in this state. It involves stalling the pipeline and avoiding further fetch and issue of instructions as the pre-fetch instructions are the wrong ones.

***Branch*:** The branch instruction is handled in this state. Since the branch prediction is available, it executes like the *Resume* state. In case of the wrong prediction, the commit logic raises the exception late. So, it involves stalling the pipeline and avoiding further fetch and issue of instruction, as the pre-fetch instructions are the wrong ones in case of misprediction.

***Load/Store*:** The memory instructions are handled in this state. Since the cache might not be ready for the new instruction, this state creates the wait for memory

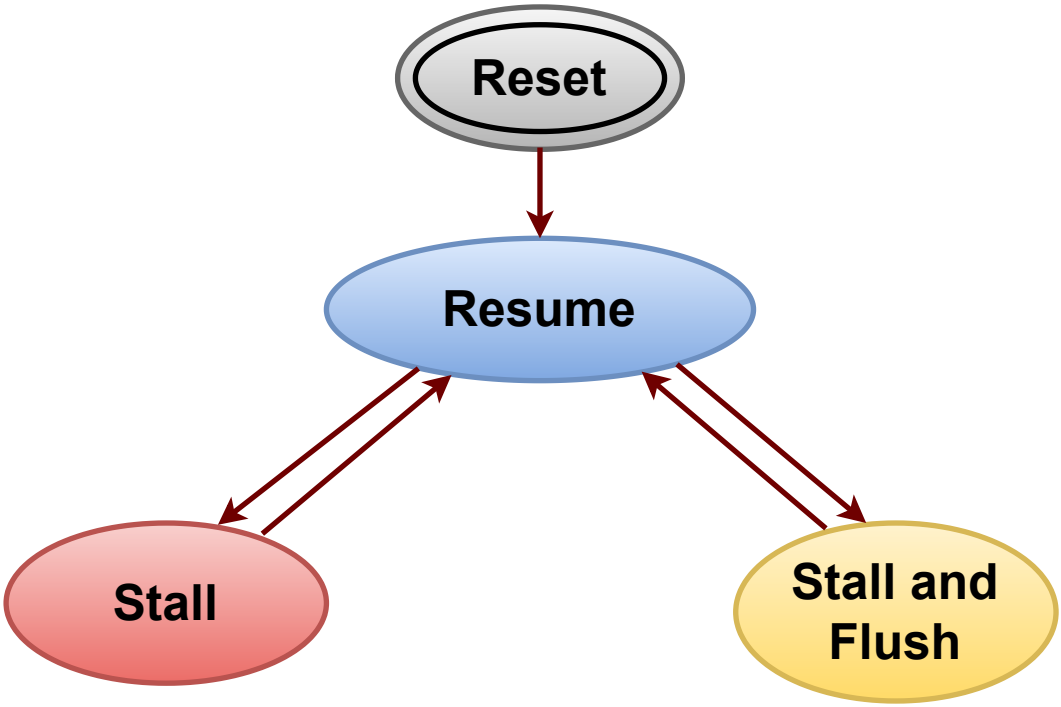


Fig. 4.13 LEN5 Flush Control.

while allowing other instructions to go no. Once the memory is ready, the instruction is replayed.

***Fence*:** The memory synchronization instructions are handled in this state. It involves stalling the pipeline, flushing everything, and reloading the IC and DC to synchronize the data with the Level-2 (L2) cache.

***E_Call/E_Break*:** The change in the mode of the processor for exception handling is performed in this state. It involves passing the correct Interrupt Vector Table address to the program counter.

Figure 4.13 shows the flush control logic. The states are described as follows:

***Reset*:** The first state is *Reset*, responsible for flushing the pipeline and front end.

***Resume*:** After the *Reset* state, the control moves to the *Resume* state. It corresponds to the standard pipeline execution for all instructions. It handles the fetch, decodes, and execution instructions for all instructions.

***Stall*:** The stall information is communicated from the main control in case of memory exception or jump or fence via the FE. Thus, it avoids fetching new instructions to save power.

***Stall and Flush*:** The stall and flush information are communicated from the main control for exception or prediction. It is the regular stall and flush of the pipeline. Also, it flushes the execution pipeline to save power by avoiding unnecessary computations.

The video processing accelerators can be added to LEN5. Figure 4.14 shows the LEN5 with the accelerator highlighted in a dashed line in dark red. The corresponding custom ISA is added, and the decoder is updated with this information.

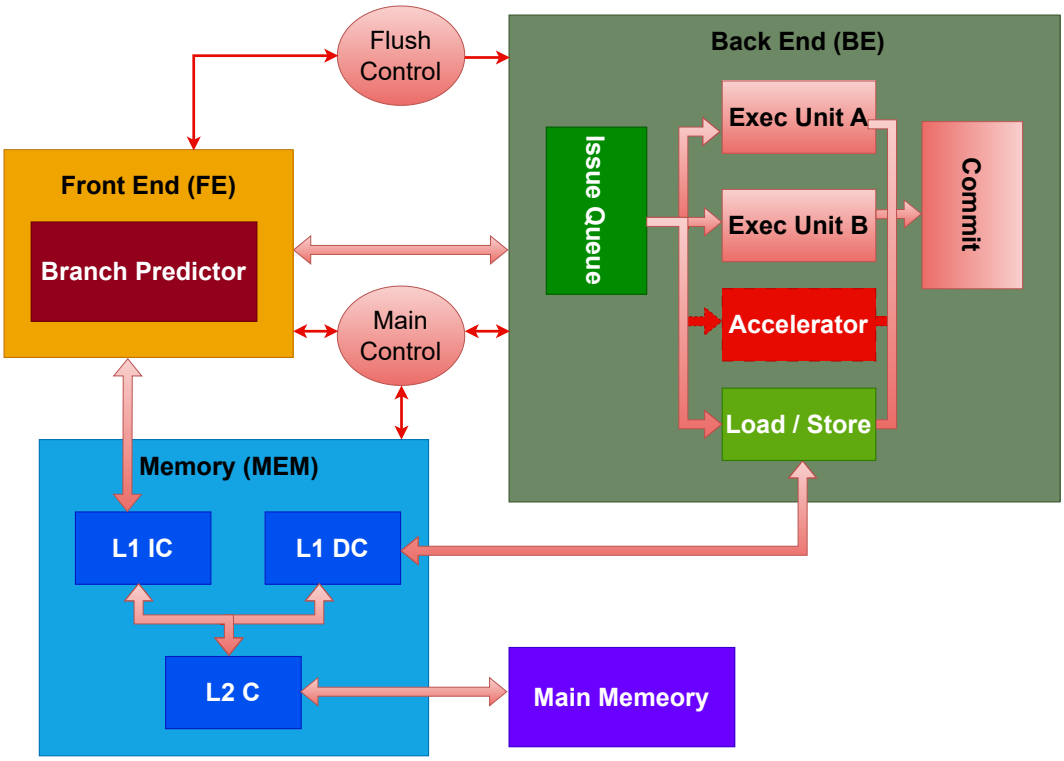


Fig. 4.14 Updated LEN5 block diagram with accelerator [9–11].

4.4 Conclusions

This work starts by providing a brief introduction to RISC-V ISA and its importance. RISC-V ISA being modular and open makes it more powerful than its predecessors. The discussion is then diverted toward determining the main parameters for selecting a core for a video processing application. The effect of each core parameter is considered for video processing algorithms. The necessary parameters are highlighted: low power, low area, and high frequency. Lower area or resources allow less power and portability. Low power consumption allows longer battery life. While high frequency allows compute-bound critical tasks to be completed faster. In the case of FPGA-based cores, low DSP consummations allows them to be used for video processing accelerators. OoO cores are more suitable as they can achieve IPC > 1. Then, it provides a survey of the literature for suitable cores. It selects and compares those cores by distributing them into four categories: in-order ASIC and FPGA and OoO ASIC and FPGA core implementations. Then, it provides the most suitable core in each case. For in-order ASIC Rocket [87, 89] is best for high-performance and DSP core [94, 95] for moderate range. For in-order FPGA, PicoRV32 [96, 113] is best for high performance and Rocket [87, 89] for moderate range. For OoO FPGA, OPA [86] is best for high performance and BOOM [86] for moderate range. For OoO ASIC BOOM v4 [42] is best for high performance and LEN5 [9–11] for moderate range. It finally chooses one of the cores LEN5 for detailed analysis as it is OoO with a low area and suitable for another accelerator extension. It also provides the necessary modification for LEN5 to be adopted for video processing.

Chapter 5

Conclusions

This thesis work explores the VLSI architectures in the domain of video processing and RISC-V processor cores. It provides the importance of video processing algorithms, which account for a significant portion of the internet data. For real-time streaming and processing, they need to be much faster. This work highlights that the parallelism available in video processing algorithms can help speed up using hardware acceleration. The devices available for this purpose are ASIC, FPGA, and Processor cores. The ASICs provide fast solutions but are less flexible. The FPGAs are more flexible but offer fewer resources and operating frequency. The processor cores are serial but can take help from OoO execution and dedicated hardware accelerators for high performance. The work follows two algorithms and provides their VLSI implementations and improvements in their architectures. Then, it provides a survey on RISC-V base processor cores for video processing. It contrasts the cores and provides a selection procedure for video processing tasks. Then, a RISC-V core from the literature is chosen, and the implementation is provided for video processing. Finally, the algorithm-to-architecture mapping of two algorithms is performed.

The first algorithm, FDSST, is in visual object tracking. The literature is reviewed for visual object tracking algorithms. The FDSST algorithm is selected for object tracking as it has a separate translation and scale filter. The complexity is resolved as the scale filter is applied only to the predicted target location rather than the whole frame. The algorithm is analyzed, and the critical blocks of complexity and delay are considered. The identified blocks are SVD, QR, DFT2, and HOG extractor.

The available parallelism in QR, SVD, and DFT2 is exploited for acceleration. The DFT2 is on a critical path. The DFT2 is implemented with the row-column decomposition algorithm and unfolds to an 8-parallel 1D-DFT architecture for fast processing. The DFT2 inside 1D-DFT has an accumulator with a register to create a pipeline to reduce critical paths. Using 2-parallel architecture helps speed up the FDSST fps. The QR factorization is implemented using the givens rotation algorithm to exploit the parallelism available by operating on multiple rows simultaneously. The SVD decomposition is implemented using the two-sided Jacobi method and 2×2 sub-blocks to exploit the parallelism available by operating on multiple blocks simultaneously. This work allows the algorithm the achieve a mean frame rate of 25.38fps for an image of 320×240 in a standalone device.

The second one, the Wiener filter, is in the domain of video encoding. It is used in AOMedia AV1 video coding. The AV1 video coding is reviewed for video encoding, and the critical block in terms of complexity and delay is highlighted. The Wiener filter algorithm is on the critical path. The selected algorithm for implementation is separable and symmetric. The complexity is resolved by reducing the calculation using symmetric reconstruction. Then RTL implementation is provided for this Wiener filter interior blocks for the overall speedup. The algorithm uses simple mathematical operations, with the most complex being the divider. Using a fast divider helped improve throughput.

This thesis also provides a survey on the RISC-V cores in the literature that can be targeted for video processing. The work highlights the main parameters for core selection. Then, the selector cores are compared for these parameters, and the best solution is provided for each category. The RISC-V cores are distributed in literature as in-order and OoO cores. The AISC solutions are fast but have high power consumption, while the FPGA solutions are slow but with low power. The high frequency, low area, and power are identified as the parameters in cores for video processing. The memory hierarchy is crucial for fast access. OoO is essential to augment the IPC / throughput available. Moderate frequency and low-power FPGA-based cores are suitable, while high-performance OoO ASIC cores are more accommodating. It also selects one of the cores LEN5 for detailed analysis as it is OoO with a low area and suitable for another accelerator extension. It also provides the necessary modification for LEN5 to be adopted for video processing.

Further research on this can be on the inclusion of the SVD, QR, and DFT2 hardware blocks inside the LEN5 core. Then, the execution of these algorithms in the core and the core with these units can help show the speedup it provides. A study can also be performed on instantiating the whole DSST block in the processor core. The performance of the core executing the DSST as a sub-block can help understand the co-execution of object-tracking tasks. Similar work can be performed for video encoding filters.

References

[1] Jean Pierre Vasseur. *Properties and applications of transistors*. Elsevier, 2016.

[2] William Aspray. The intel 4004 microprocessor: What constituted invention? *IEEE Annals of the History of Computing*, 19(3):4–15, 1997.

[3] Cisco Systems. Vni complete forecast highlights, 2018. https://www.cisco.com/c/dam/m/en_us/solutions/service-provider/vni-forecast-highlights/pdf/Global_2021_Forecast_Highlights.pdf.

[4] John Villasenor and William H Mangione-Smith. Configurable computing. *Scientific American*, 276(6):66–71, 1997.

[5] Amara Amara, Frederic Amiel, and Thomas Ea. Fpga vs. asic for low power applications. *Microelectronics journal*, 37(8):669–677, 2006.

[6] Vivienne Sze and Anantha P Chandrakasan. Joint algorithm-architecture optimization of cabac to increase speed and reduce area cost. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1577–1580. IEEE, 2011.

[7] Martin Danelljan, Gustav Häger, Fahad Shahbaz Khan, and Michael Felsberg. Discriminative scale space tracking. *IEEE transactions on pattern analysis and machine intelligence*, 39(8):1561–1575, 2016.

[8] JS Goldstein, IS Reed, LL Scharf, and JA Tague. A low-complexity implementation of adaptive wiener filters. In *Conference Record of the Thirty-First Asilomar Conference on Signals, Systems and Computers (Cat. No. 97CB36136)*, volume 1, pages 770–774. IEEE, 1997.

[9] Marco Andorno. Design of the frontend for len5, a risc-v out-of-order processor. Master's thesis, Politecnico di Torino, 2019. http://webthesis.biblio.polito.it/13198/.

[10] Michele Caon. Design of the execution pipeline for len5, a risc-v out-of-order processor. Master's thesis, Politecnico di Torino, 2019. http://webthesis.biblio.polito.it/13205/.

[11] Matteo Perotti. Design of an os compliant memory system for len5, a risc-v out of order processor. Master's thesis, Politecnico di Torino, 2019. http://webthesis.biblio.polito.it/13231/.

[12] Sam Hare, Stuart Golodetz, Amir Saffari, Vibhav Vineet, Ming-Ming Cheng, Stephen L Hicks, and Philip HS Torr. Struck: Structured output tracking with kernels. *IEEE transactions on pattern analysis and machine intelligence*, 38(10):2096–2109, 2015.

[13] Martin Danelljan, Fahad Shahbaz Khan, Michael Felsberg, and Joost Van de Weijer. Adaptive color attributes for real-time visual tracking. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1090–1097, 2014.

[14] Anton Varfolomieiev. Channel-independent spatially regularized discriminative correlation filter for visual object tracking. *Journal of Real-Time Image Processing*, 18:233–243, 2021.

[15] Xu Jia, Huchuan Lu, and Ming-Hsuan Yang. Visual tracking via adaptive structural local sparse appearance model. In *2012 IEEE Conference on computer vision and pattern recognition*, pages 1822–1829. IEEE, 2012.

[16] Laura Sevilla-Lara and Erik Learned-Miller. Distribution fields for tracking. In *2012 IEEE Conference on computer vision and pattern recognition*, pages 1910–1917. IEEE, 2012.

[17] David S Bolme, J Ross Beveridge, Bruce A Draper, and Yui Man Lui. Visual object tracking using adaptive correlation filters. In *2010 IEEE computer society conference on computer vision and pattern recognition*, pages 2544–2550. IEEE, 2010.

[18] Joao F Henriques, Rui Caseiro, Pedro Martins, and Jorge Batista. Exploiting the circulant structure of tracking-by-detection with kernels. In *Computer Vision–ECCV 2012: 12th European Conference on Computer Vision, Florence, Italy, October 7-13, 2012, Proceedings, Part IV 12*, pages 702–715. Springer, 2012.

[19] Tobias Böttger and Carsten Steger. Accurate and robust tracking of rigid objects in real time. *Journal of Real-Time Image Processing*, 18(3):493–510, 2021. https://doi.org/10.1007/s11554-020-00978-9.

[20] Ce Li, Xingchao Liu, Xiangbo Su, and Baochang Zhang. Robust kernelized correlation filter with scale adaption for real-time single object tracking. *Journal of Real-Time Image Processing*, 15:583–596, 2018.

[21] Al-Hussein A El-Shafie and Serag ED Habib. Survey on hardware implementations of visual object trackers. *IET Image Processing*, 13(6):863–876, 2019.

[22] Walid Walid, Muhammad Awais, Ashfaq Ahmed, Guido Masera, and Maurizio Martina. Real-time implementation of fast discriminative scale space tracking algorithm. *Journal of Real-Time Image Processing*, 18(6):2347–2360, 2021.

[23] Mozilla Research. Expanding the foundations of the open web. https://research.mozilla.org/av1-media-codecs/.

[24] Yue Chen, Debargha Murherjee, Jingning Han, Adrian Grange, Yaowu Xu, Zoe Liu, Sarah Parker, Cheng Chen, Hui Su, Urvang Joshi, et al. An overview of core coding tools in the av1 video codec. In *2018 picture coding symposium (PCS)*, pages 41–45. IEEE, 2018.

[25] Mário Saldanha, Marcel Corrêa, Guilherme Corrêa, Daniel Palomino, Marcelo Porto, Bruno Zatt, and Luciano Agostini. An overview of dedicated hardware designs for state-of-the-art av1 and h. 266/vvc video codecs. In *2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 1–4. IEEE, 2020.

[26] W3C Patent Policy. https://www.w3.org/Consortium/Patent-Policy-20040205/.

[27] Debargha Mukherjee, Shunyao Li, Yue Chen, Aamir Anis, Sarah Parker, and James Bankoski. A switchable loop-restoration with side-information framework for the emerging av1 video codec. In *2017 IEEE International Conference on Image Processing (ICIP)*, pages 265–269. IEEE, 2017.

[28] J Scott Goldstein, Irving S Reed, and Louis L Scharf. A multistage representation of the wiener filter based on orthogonal projections. *IEEE Transactions on Information Theory*, 44(7):2943–2959, 1998.

[29] Marcello LR de Campos, Stefan Werner, and JA Apolinario. On an efficient implementation of the multistage wiener filter through householder reflections for ds-cdma interference suppression. In *GLOBECOM'03. IEEE Global Telecommunications Conference (IEEE Cat. No. 03CH37489)*, volume 4, pages 2350–2354. IEEE, 2003.

[30] Jianpeng Dong and Nam Ling. An iterative method for frame-level adaptive wiener interpolation filters in video coding. In *2006 IEEE Workshop on Signal Processing Systems Design and Implementation*, pages 113–117. IEEE, 2006.

[31] Mischa Siekmann, Sebastian Bosse, Heiko Schwarz, and Thomas Wiegand. Separable wiener filter based adaptive in-loop filter for video coding. In *28th Picture Coding Symposium*, pages 70–73. IEEE, 2010.

[32] A Elnady, A Noureldin, and Yan-Fei Liu. Implementation of the wiener filter for extracting power quality disturbances. In *2007 IEEE Power Electronics Specialists Conference*, pages 1116–1120. IEEE, 2007.

[33] Hilman Pardede, Kalamullah Ramli, Yohan Suryanto, Nur Hayati, and Alfan Presekal. Speech enhancement for secure communication using coupled spectral subtraction and wiener filter. *Electronics*, 8(8):897, 2019.

[34] Fei Wu, Wenxue Yang, Limin Xiao, and Jinbin Zhu. Adaptive wiener filter and natural noise to eliminate adversarial perturbation. *Electronics*, 9(10):1634, 2020.

[35] Piotr Musznicki, Jean-Luc Schanen, Pierre Granjon, and Piotr J Chrzan. The wiener filter applied to emi decomposition. *IEEE Transactions on Power Electronics*, 23(6):3088–3093, 2008.

[36] K Raja Rajeswari, K Murali Krishna, V Jagan Naveen, and A Vamsidhar. Performance comparison of wiener filter and cls filter on 2d signals. In *2009 Sixth International Conference on Information Technology: New Generations*, pages 1244–1249. IEEE, 2009.

[37] Smit Trambadia and Paresh Dholakia. Design and analysis of an image restoration using wiener filter with a quality based hybrid algorithms. In *2015 2nd International Conference on Electronics and Communication Systems (ICECS)*, pages 1318–1323. IEEE, 2015.

[38] GVP Chandra, Sekhar Yadav, B Ananda Krishna, and M Kamaraju. Performance of wiener filter and adaptive filter for noise cancellation in real-time environment. *International journal of computer applications*, 97(15), 2014.

[39] Walid Walid, Giorgio Armanno, Sandro Di Paola, Massimo Ruo Roch, Guido Masera, and Maurizio Martina. Vlsi architectures of a wiener filter for video coding. *Electronics*, 10(16):1961, 2021.

[40] Tim Fritzmann, Uzair Sharif, Daniel Müller-Gritschneder, Cezar Reinbrecht, Ulf Schlichtmann, and Johanna Sepulveda. Towards reliable and secure post-quantum co-processors based on risc-v. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1148–1153. IEEE, 2019.

[41] Jonathan Balkind, Katie Lim, Fei Gao, Jinzheng Tu, David Wentzlaff, Michael Schaffner, Florian Zaruba, and Luca Benini. Openpiton+ ariane: The first open-source, smp linux-booting risc-v system scaling from one to many cores. In *Workshop on Computer Architecture Research with RISC-V (CARRV)*, pages 1–6, 2019.

[42] Krste Asanovic, David A Patterson, and Christopher Celio. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. Technical report, University of California at Berkeley Berkeley United States, 2015.

[43] Chao Wang, Xi Li, Peng Chen, Xiaojing Feng, Junneng Zhang, and Xuehai Zhou. Detecting data hazards in multi-processor system-on-chips on fpga. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 282–287. IEEE, 2012.

[44] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[45] Rudolf Eigenmann and David J Lilja. Von neumann computers. *Wiley Encyclopedia of Electrical and Electronics Engineering*, 23:387–400, 1998.

[46] Sue Nelson. The harvard computers. *Nature*, 455(7209):36–37, 2008.

[47] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[48] Carsten Heinz, Yannick Lavan, Jaco Hofmann, and Andreas Koch. A catalog and in-hardware evaluation of open-source drop-in compatible risc-v softcore processors. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2019.

[49] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8. IEEE, 2017.

[50] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[51] Ruchira Shirbhate, Tejaswini Panse, and Chetan Ralekar. Design of parallel fft architecture using cooley tukey algorithm. In *2015 International Conference on Communications and Signal Processing (ICCSP)*, pages 0574–0578. IEEE, 2015.

[52] Muhammad Awais, Marco Vacca, Mariagrazia Graziano, and Guido Masera. Fft implementation using qca. In *2012 19th IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2012)*, pages 741–744. IEEE, 2012.

[53] Debaprasad De, GK Gumar, Archisman Ghosh, and Anurup Saha. Fpga implementation of discrete fourier transform using cordic algorithm. *Adv. Model. Anal. B*, 60(2):332–337, 2017.

[54] Chi-Li Yu, Kevin Irick, Chaitali Chakrabarti, and Vijaykrishnan Narayanan. Multidimensional dft ip generator for fpga platforms. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 58(4):755–764, 2010.

[55] Sergio D Muñoz and Javier Hormigo. High-throughput fpga implementation of qr decomposition. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62(9):861–865, 2015.

[56] Jian Wu, Shu Fang, Lei Li, and Yang Yang. Qr decomposition and gram schmidt orthogonalization based low-complexity multi-user mimo precoding. *IET*, 2014.

[57] Farhad Merchant, Tarun Vatwani, Anupam Chattopadhyay, Soumyendu Raha, SK Nandy, and Ranjani Narayan. Achieving efficient qr factorization by algorithm-architecture co-design of householder transformation. In *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, pages 98–103. IEEE, 2016.

[58] Wen Fan and Amir Alimohammad. Givens rotation-based qr decomposition for mimo systems. *IET Communications*, 11(12):1838–1845, 2017.

[59] Swanirbhar Majumder, Anil Kumar Shaw, and Subir Kumar Sarkar. Hardware implementation of singular value decomposition. *Journal of The Institution of Engineers (India): Series B*, 97:227–231, 2016.

[60] Aidin Shiri and Ghader Karimian Khosroshahi. An fpga implementation of singular value decomposition. In *2019 27th Iranian Conference on Electrical Engineering (ICEE)*, pages 416–422. IEEE, 2019.

[61] Ramanarayan Mohanty, Gonnabhaktula Anirudh, Tapan Pradhan, Bibek Kabi, and Aurobinda Routray. Design and performance analysis of fixed-point jacobi svd algorithm on reconfigurable system. *IERI Procedia*, 7:21–27, 2014.

[62] Vinh Ngo, David Castells-Rufas, Arnau Casadevall, Marc Codina, and Jordi Carrabina. Low-power pedestrian detection system on fpga. *Multidisciplinary Digital Publishing Institute Proceedings*, 31(1):35, 2019.

[63] Cédric Bourrasset, Luca Maggiani, Jocelyn Sérot, and François Berry. Dataflow object detection system for fpga-based smart camera. *IET Circuits, Devices & Systems*, 10(4):280–291, 2016.

[64] Michael Hahnle, Frerk Saxen, Matthias Hisung, Ulrich Brunsmann, and Konrad Doll. Fpga-based real-time pedestrian detection on high-resolution images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 629–635, 2013.

[65] Yue Wang, Kevin Cunningham, Prawat Nagvajara, and Jeremy Johnson. Singular value decomposition hardware for mimo: State of the art and custom design. In *2010 International Conference on Reconfigurable Computing and FPGAs*, pages 400–405. IEEE, 2010.

[66] Hong Zhang, Zeyu Zhang, Lei Zhang, Yifan Yang, Qiaochu Kang, and Daniel Sun. Object tracking for a smart city using iot and edge computing. *Sensors*, 19(9):1987, 2019.

[67] Seyed Mojtaba Marvasti-Zadeh, Hossein Ghanei-Yakhdan, and Shohreh Kasaei. Rotation-aware discriminative scale space tracking. In *2019 27th Iranian Conference on Electrical Engineering (ICEE)*, pages 1272–1276. IEEE, 2019.

[68] Marcin Kowalczyk, Dominika Przewlocka, and Tomasz Kryjak. Real-time implementation of adaptive correlation filter tracking for 4k video stream in zynq ultrascale+ mpsoc. In *2019 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 53–58. IEEE, 2019.

[69] Xuan Gong, Zichun Le, Hui Wang, and Yukun Wu. Study on the moving target tracking based on vision dsp. *Sensors*, 20(22):6494, 2020.

[70] Bingyi Zhang, Xin Li, Jun Han, and Xiaoyang Zeng. Minitracker: a lightweight cnn-based system for visual object tracking on embedded device. In *2018 IEEE 23rd International Conference on Digital Signal Processing (DSP)*, pages 1–5. IEEE, 2018.

[71] Ehsan Norouznezhad, Abbas Bigdeli, Adam Postula, and Brian C Lovell. Object tracking on fpga-based smart cameras using local oriented energy and phase features. In *Proceedings of the Fourth ACM/IEEE International Conference on Distributed Smart Cameras*, pages 33–40, 2010.

[72] Marco Jacobs and Jonah Probell. A brief history of video coding. *ARC International*, 1:6, 2007.

[73] Gustavo Sutter, Jean-Pierre Deschamps, Gery Bioul, and Eduardo Boemo. Power aware dividers in fpga. In *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation: 14th International Workshop, PATMOS 2004, Santorini, Greece, September 15-17, 2004. Proceedings 14*, pages 574–584. Springer, 2004.

[74] Seong-Wan Kim and Earl E Swartzlander. Restoring divider design for quantum-dot cellular automata. In *2011 11th IEEE International Conference on Nanotechnology*, pages 1295–1300. IEEE, 2011.

[75] Suganthi Venkatachalam, Elizabeth Adams, and Seok-Bum Ko. Design of approximate restoring dividers. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2019.

[76] Roberto Millón, Emmanuel Frati, and Enzo Rucci. A comparative study between hls and hdl on soc for image processing applications. *arXiv preprint arXiv:2012.08320*, 2020.

[77] Lora Petkova and Ivo Draganov. Noise adaptive wiener filtering of images. In *2020 55th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*, pages 177–180. IEEE, 2020.

[78] S Malarvizhi, R Kayalvizhi, Amit Kumar, and Anita Topkar. Raw data processing using modern hardware for inspection of objects in x-ray baggage inspection systems. *IEEE Transactions on Nuclear Science*, 68(6):1296–1303, 2021.

[79] Tugay Doner and Dincer Gokcen. Fpga-based infrared image deblurring using angular position of ir detector. *The Visual Computer*, 37(7):2039–2050, 2021.

[80] A Neetu. Implementation of wiener filter on fpga using xilinx system generator for speech enhancement. *Int. J. Sci. Innov. Res. Stud*, 6:1–12, 2018.

[81] A Yasodai and AV Ramprasad. Noise degradation system using wiener filter and cordic based fft/ifft processor. *Journal of Central South University*, 22:3849–3859, 2015.

[82] Google Git. Alliance for open media. https://aomedia.googlesource.com/aom/.

[83] RISC-V Organisation. Risc-v specifications, 2021. https://riscv.org/technical/specifications/.

[84] Shahriar Akramullah. *Digital video concepts, methods, and metrics: quality, compression, performance, and power trade-off analysis*. Springer Nature, 2014.

[85] Aneesh Raveendran, Vinayak Baramu Patil, David Selvakumar, and Vivian Desalphine. A risc-v instruction set processor-micro-architecture design and analysis. In *2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA)*, pages 1–7. IEEE, 2016.

[86] Susumu Mashimo, Akifumi Fujita, Reoma Matsuo, Seiya Akaki, Akifumi Fukuda, Toru Koizumi, Junichiro Kadomoto, Hidetsugu Irie, Masahiro Goshima, Koji Inoue, et al. An open source fpga-optimized out-of-order risc-v soft processor. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 63–71. IEEE, 2019.

[87] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 4, 2016.

[88] Florian Zaruba and Luca Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, 2019.

[89] Alexander Dörflinger, Mark Albers, Benedikt Kleinbeck, Yejun Guan, Harald Michalik, Raphael Klink, Christopher Blochwitz, Anouar Nechi, and Mladen Berekovic. A comparative survey of open-source application-class risc-v processor implementations. In *Proceedings of the 18th ACM International Conference on Computing Frontiers*, pages 12–20, 2021.

[90] Mahmoud A Elmohr, Ahmed S Eissa, Moamen Ibrahim, Mostafa Khamis, Sameh El-Ashry, Ahmed Shalaby, Mohamed AbdElsalam, and M Watheq El-Kharashi. Rvnoc: A framework for generating risc-v noc-based mpsoc. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 617–621. IEEE, 2018.

[91] Jaume Abella, Calvin Bulla, Guillem Cabo, Francisco J Cazorla, Adrián Cristal, Max Doblas, Roger Figueras, Alberto González, Carles Hernández, César Hernández, et al. An academic risc-v silicon implementation based on open-source components. In *2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6. IEEE, 2020.

[92] Ronny Garcia-Ramirez, Alfonso Chacon-Rodriguez, Roberto Molina-Robles, Reinaldo Castro-Gonzalez, Egdar Solera-Bolanos, Gabriel Madrigal-Boza, Marco Oviedo-Hernandez, Diego Salazar-Sibaja, Dayhana Sanchez-Jimenez, Melissa Fonseca-Rodriguez, et al. Siwa: A custom risc-v based system on chip (soc) for low power medical applications. *Microelectronics Journal*, 98:104753, 2020.

[93] Ckristian Duran, D Luis Rueda, Giovanny Castillo, Anderson Agudelo, Camilo Rojas, Luis Chaparro, Harry Hurtado, Juan Romero, Wilmer Ramirez, Hector Gomez, et al. A 32-bit risc-v axi4-lite bus-based microcontroller with 10-bit sar adc. In *2016 IEEE 7th Latin American Symposium on Circuits & Systems (LASCAS)*, pages 315–318. IEEE, 2016.

[94] Satyajit Bora and Roy Paily. A high-performance core micro-architecture based on risc-v isa for low power applications. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 68(6):2132–2136, 2020.

[95] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K Gürkaynak, and Luca Benini. Near-threshold risc-v core with dsp extensions for scalable iot end-point devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, 2017.

[96] Clifford Wolf. Picorv32 - a size-optimized risc-v cpu, 2019. https://github.com/cliffordwolf/picorv32.

[97] VectorBlox Computing Inc. Orca, 2019. https://github.com/VectorBlox/orca.

[98] SpinalHDL. Vexriscv, 2019. https://github.com/SpinalHDL/VexRiscv.

[99] Roland Höller, Dominic Haselberger, Dominik Ballek, Peter Rössler, Markus Krapfenbauer, and Martin Linauer. Open-source risc-v processor ip cores for fpgas—overview and evaluation. In *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–6. IEEE, 2019.

[100] Freedom. https://github.com/sifive/freedom.

[101] Eric Matthews and Lesley Shannon. Taiga: A new risc-v soft-processor framework enabling high performance cpu architectural features. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.

[102] Luís Fiolhais, Fernando Gonçalves, Rui P Duarte, Mário Véstias, and José T de Sousa. Low energy heterogeneous computing with multiple risc-v and cgra cores. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2019.

[103] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A Patterson, and Krste Asanovic. Boomv2: an open-source out-of-order risc-v core. In *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.

[104] Dennis Agyemanh Nana Gookyi and Kwangki Ryoo. Selecting a synthesizable risc-v processor core for low-cost hardware devices. *Journal of Information Processing Systems*, 15(6):1406–1421, 2019.

[105] Guohe Zhang, Kepeng Zhao, Bin Wu, Yiqun Sun, Li Sun, and Feng Liang. A risc-v based hardware accelerator designed for yolo object detection system. In *2019 IEEE International Conference of Intelligent Applied Systems on Engineering (ICIASE)*, pages 9–11. IEEE, 2019.

[106] Abdelrahman S Hussein and Hassan Mostafa. Asic-fpga gap for a risc-v core implementation for dnn applications. In *2021 3rd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*, pages 385–388. IEEE, 2021.

[107] Najdet Charaf, Ahmed Kamaleldin, Martin Thümmler, and Diana Göhringer. Rv-cap: Enabling dynamic partial reconfiguration for fpga-based risc-v system-on-chip. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 172–179. IEEE, 2021.

[108] Neel Gala, Arjun Menon, Rahul Bodduna, GS Madhusudan, and V Kamakoti. Shakti processors: An open-source hardware initiative. In *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, pages 7–8. IEEE Computer Society, 2016.

[109] Davide Rossi, Francesco Conti, Andrea Marongiu, Antonio Pullini, Igor Loi, Michael Gautschi, Giuseppe Tagliavini, Alessandro Capotondi, Philippe Flatresse, and Luca Benini. Pulp: A parallel ultra low power platform for next generation iot applications. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–39. IEEE Computer Society, 2015.

[110] Muhammad Ali, Pedram Amini Rad, and Diana Göhringer. Risc-v based mpsoc design exploration for fpgas: area, power and performance. In *Applied Reconfigurable Computing. Architectures, Tools, and Applications: 16th International Symposium, ARC 2020, Toledo, Spain, April 1–3, 2020, Proceedings 16*, pages 193–207. Springer, 2020.

[111] Iván Gamino del Río, Agustín Martínez Hellín, Óscar R Polo, Miguel Jiménez Arribas, Pablo Parra, Antonio da Silva, Jonatan Sánchez, and Sebastián Sánchez. A risc-v processor design for transparent tracing. *Electronics*, 9(11):1873, 2020.

[112] Michael Gautschi, Michael Schaffner, Frank K Gürkaynak, and Luca Benini. An extended shared logarithmic unit for nonlinear function kernel acceleration in a 65-nm cmos multicore cluster. *IEEE Journal of Solid-State Circuits*, 52(1):98–112, 2016.

[113] Carsten Heinz, Yannick Lavan, Jaco Hofmann, and Andreas Koch. A catalog and in-hardware evaluation of open-source drop-in compatible risc-v softcore processors. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2019.

[114] Nguyen My Qui, Chang Hong Lin, and Poki Chen. Design and implementation of a 256-bit risc-v-based dynamically scheduled very long instruction word on fpga. *IEEE Access*, 8:172996–173007, 2020.

[115] Vazgen Melikyan, Eduard Babayan, Anush Melikyan, Davit Babayan, Poghos Petrosyan, and Edvard Mkrtchyan. Clock gating and multi-vth low power design methods based on 32/28 nm orca processor. In *2015 IEEE East-West Design & Test Symposium (EWDTS)*, pages 1–4. IEEE, 2015.

[116] Eric Matthews, Zavier Aguila, and Lesley Shannon. Evaluating the performance efficiency of a soft-processor, variable-length, parallel-execution-unit architecture for fpgas using the risc-v isa. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–8. IEEE, 2018.

Appendix A

Publications

List of Publications

	Published papers
1	Walid Walid, Muhammad Awais, Ashfaq Ahmed, Guido Masera, and Maurizio Martina. Real-time implementation of fast discriminative scale space tracking algorithm. Journal of Real-Time Image Processing, 18(6):2347–2360, 2021.
2	Walid Walid, Giorgio Armanno, Sandro Di Paola, Massimo Ruo Roch, Guido Masera, and Maurizio Martina. Vlsi architectures of a wiener filter for video coding. Electronics, 10(16):1961, 2021.