

REAL-TIME KERNEL PROCESSING
FOR
CONVOLUTIONAL NEURAL NETWORKS
ON
SOC FPGAS

by
Zackery Adam Backman

A thesis submitted in partial fulfillment
of the requirements for the degree

of
Master of Science
in
Electical Engineering

MONTANA STATE UNIVERSITY
Bozeman, Montana

December 2025

©COPYRIGHT

by

Zackery Adam Backman

2025

All Rights Reserved

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Ross Snider, for his guidance and support throughout this project. Along with him giving me the opportunity to work on this project, so that I could pursue my Master's degree at Montana State University.

This material is based upon work supported in part by the National Science Foundation EPSCoR Cooperative Agreement OIA-2242802. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

1. BACKGROUND.....	1
SMART FireS	3
Wildfires	4
Hyperspectral Imaging	8
Machine Learning	9
Primary Learning Approaches	10
Classification Algorithms	10
Convolutional Neural Networks	11
Hardware Acceleration for Machine Learning	18
FPGAs, SoC FPGAs, and SoMs	19
Field-Programmable Gate Arrays (FPGAs).....	19
System-on-Chip (SoC) FPGAs	22
System-on-Module (SoM)	23
Mercury+ XU1 SoM and Mercury ST1 Base Board	23
Zynq UltraScale+ MPSoC Programmable Logic Details.....	25
Ultra Ram in the Programmable Logic	28
FPGA Hardware Design Concepts and VHDL.....	30
VHDL Primitives	31
Basic Combinational Logic Gates	32
D Flip-Flop	35
Registered Logic	38
Complex Combinational Circuits.....	46
Complex Sequential Circuits	51
Operating System Options for SoC FPGAs	61
Embedded Linux	62
Yocto	63
Rationale for a Linux-Based SoC Platform	66
2. HARDWARE: CUSTOM ACCELERATOR FOR CNN KERNEL OPER- ATIONS.....	67
Design Goals and Data Formatting	69
Design Goals and Constraints	69
Input Data Formatting.....	70
Output Data Formatting	71
Memory Architecture	71
UltraRAM Organization and Memory Mapping.....	72
Memory Controller Implementation.....	77
Data Retrieval and Alignment.....	82

TABLE OF CONTENTS – CONTINUED

Data Slicer Implementation	83
Convolution Pipeline Architecture	86
Value Alignment and Partial Accumulation Strategy	86
Pipelined MAC Architecture	87
Convolution Engine Implementation	89
Performance Of Hardware	93
Computational Throughput	93
Performance Estimation at 100 MHz	100
Resource Estimation	103
Validation and Verification	104
MATLAB Simulation Framework	104
Test Vector Generation	104
Timeseries Creation for Simulink Integration	106
Simulink Simulation Execution	107
Output Capture Using Enabled Subsystem	107
Reference Calculation Methods	108
Verification Results	110
3. SOFTWARE: EMBEDDED LINUX FOR SOC FPGAS	112
Yocto and OpenEmbedded	112
Yocto, OpenEmbedded, and Bitbake Concepts	113
Layers and Directory Structure	113
meta-msu-de10-nano Layer Structure	114
Recipes	116
Metadata and License Information	117
Inheriting Classes	118
Source Code Fetching	118
Build Dependencies	119
Runtime Dependencies	119
Provides Statement	119
Installation Task	120
Installed Files Tracking	120
Systemd Configuration	121
Configuration Files	123
Layer Configuration	123
Machine Configuration	124
Classes	128
Inherit Directive	129
Output Image Formats	130

TABLE OF CONTENTS – CONTINUED

NFS Deployment Variables	130
Custom Task: do_deploy_nfs	130
Task Management	131
Images	132
Board Support Package Components	134
U-Boot Bootloader Configuration	134
Device Tree Specifications	135
FPGA Bitstream Deployment	136
Linux Kernel Configuration	137
Build Utility Files	137
WIC Kickstart Files	138
Building and Deploying the DE10-Nano Image	138
Environment Setup and Layer Configuration	138
Repository Setup	139
Build Directory Initialization	139
Layer Registration	139
Machine and Deployment Configuration	140
Target Machine Selection	140
Deployment Method Configuration	140
Image Build Process	141
Building the Image	141
Build Output Artifacts	141
Deployment Methods	142
SD Card Deployment	142
TFTP/NFS Network Deployment	142
Build System Integration	143
4. CONCLUSION AND FUTURE WORK	144
Hardware Acceleration	144
Software Solution	145
Future Work	146
REFERENCES CITED	147
APPENDICES	154
APPENDIX A : Complete Source Code	155
VHDL Source Files	156
and-gate.vhd	156

TABLE OF CONTENTS – CONTINUED

d-flip-flop.vhd.....	156
async-reset.vhd.....	156
sync-reset.vhd	156
and-gate-registered.vhd.....	157
half-adder.vhd	157
ripple-carry-adder-4bit.vhd.....	157
signed-multiplier.vhd	158
counter-4bit.vhd.....	158
fifo-sync.vhd	159
fifo-async.vhd	160
Yocto Recipe Files.....	162
audiomini-drivers.bb.....	162
audio-mini-image.bbclass.....	162
audio-mini-passthrough.bb	163
audio-mini-combfilter.bb.....	164
APPENDIX B : MATLAB Simulink Hardware Diagrams.....	165

LIST OF TABLES

Table		Page
1.	Table 1 VHDL Modeling Styles	31
2.	Table 2 Synchronous vs. Asynchronous Resets	38
3.	Table 3 FPGA Resource Utilization Summary for Key Digital Circuits (Xilinx UltraScale+)	54
4.	Table 4 Hardware Accelerator System Parameters	70
5.	Table 5 FPGA Resource Estimation for Hardware Accelerator on XCZU15EG	103

LIST OF FIGURES

Figure		Page
1.	Figure 1 Overview of the SMART FIRES program [32].	4
2.	Figure 2 The Fire Triangle [37].	5
3.	Figure 3 The Wildfire Behavior Triangle [38].	6
4.	Figure 4 Wildfire Frequency in the United States, 1983–2022 [61].	7
5.	Figure 5 Wildfire Extent in the United States, 1983–2022 [61].	7
6.	Figure 6 Electromagnetic Spectrum [48].	8
7.	Figure 7 Hyperspectral Imaging Overview [49].	9
8.	Figure 8 LeNet-5 Architecture [26].	11
9.	Figure 9 Step-by-step visualization of a 2D convolution operation with a 3×3 kernel sliding over a 5×5 input matrix (stride=1, padding=0). Each step shows the kernel position (red) on the input and the corresponding output location (orange) in the resulting 3×3 output matrix.	14
10.	Figure 10 Input patch from input data	17
11.	Figure 11 Kernel/Filter (original)	17
12.	Figure 12 Flipped kernel/filter (for convolution)	17
13.	Figure 13 Element-wise multiplication (with flipped kernel/filter)	17
14.	Figure 14 Visual example of the convolution operation for (5, 5, 1) input and (3, 3, 1) kernel[20].	18
15.	Figure 15 Generic FPGA Fabric Diagram [11].	19
16.	Figure 16 Generic FPGA CLB Diagram [11].	20
17.	Figure 17 Generic FPGA Interconnect Diagram [11].	20
18.	Figure 18 Xilinx DSP Slice (DSP48E) [1].	21
19.	Figure 19 Generic FPGA IO Block Diagram [11].	21
20.	Figure 20 Zynq UltraScale+ MPSoC [5].	22
21.	Figure 21 Top View of the Mercury+ XU1 SoM module [13].	24

LIST OF FIGURES – CONTINUED

Figure	Page
22. Figure 22 Bottom View of the Mercury+ XU1 SoM module [13].	24
23. Figure 23 Top View of the Mercury ST1 Base Board [14].	25
24. Figure 24 Synthesized schematic (left) and timing waveform (right) for AND gate.	33
25. Figure 25 Synthesized schematic (left) and timing waveform (right) for OR gate.	33
26. Figure 26 Synthesized schematic (left) and timing waveform (right) for XOR gate.	34
27. Figure 27 Synthesized schematic (left) and timing waveform (right) for NOT gate.	34
28. Figure 28 Synthesized schematic (left) and timing waveform (right) for NAND gate.	34
29. Figure 29 Synthesized schematic (left) and timing waveform (right) for NOR gate.	35
30. Figure 30 Synthesized schematic (left) and timing waveform (right) for XNOR gate.	35
31. Figure 31 Timing waveform for D flip-flop demonstrating edge-triggered behavior.	36
32. Figure 32 Gate-level schematic of D flip-flop showing internal NAND gate structure and SR latch implementation.	37
33. Figure 33 Behavioral schematic of D flip-flop showing the abstracted primitive symbol.	37
34. Figure 34 Synthesized schematic of registered AND gate showing combinational logic followed by flip-flop.	40
35. Figure 35 Timing waveform for registered AND gate demonstrating one clock cycle latency.	40
36. Figure 36 Synthesized schematic of registered OR gate.	41
37. Figure 37 Timing waveform for registered OR gate.	41

LIST OF FIGURES – CONTINUED

Figure		Page
38. Figure 38	Synthesized schematic of registered XOR gate.....	42
39. Figure 39	Timing waveform for registered XOR gate.	42
40. Figure 40	Synthesized schematic of registered NOT gate.....	43
41. Figure 41	Timing waveform for registered NOT gate.	43
42. Figure 42	Synthesized schematic of registered NAND gate.	43
43. Figure 43	Timing waveform for registered NAND gate.....	44
44. Figure 44	Synthesized schematic of registered NOR gate.....	44
45. Figure 45	Timing waveform for registered NOR gate.	45
46. Figure 46	Synthesized schematic of registered XNOR gate.	45
47. Figure 47	Timing waveform for registered XNOR gate.....	46
48. Figure 48	Synthesized schematic (left) and timing waveform (right) for half adder gate.	47
49. Figure 49	Synthesized schematic of 4-bit ripple-carry adder showing cascaded full-adder stages.	49
50. Figure 50	Timing waveform for 4-bit ripple-carry adder demon- strating multi-bit addition.	49
51. Figure 51	Synthesized schematic of 16-bit signed multiplier utilizing DSP48E1 slice.....	51
52. Figure 52	Timing waveform for 16-bit signed multiplier demon- strating multiplication operation.	51
53. Figure 53	Synthesized schematic of 4-bit binary counter with enable and reset.	52
54. Figure 54	Timing waveform for 4-bit counter demonstrating sequential counting behavior.	53
55. Figure 55	Synthesized schematic of 4-bit shift register show- ing cascaded flip-flop chain.	53

LIST OF FIGURES – CONTINUED

Figure	Page
56. Figure 56 Timing waveform for 4-bit shift register demonstrating data propagation through stages.	54
57. Figure 57 Synthesized schematic of synchronous FIFO showing memory array, pointers, and control logic.	57
58. Figure 58 Synthesized schematic of asynchronous FIFO showing dual-clock architecture with Gray code synchronizers.	60
59. Figure 59 Yocto Project Technical Overview [46].	64
60. Figure 60 Top-level diagram of the memory handler.	67
61. Figure 61 Proposed Processing Pipeline of the Smart Sensor.	68
62. Figure 62 Memory Handler (MH) top-level integration of all accelerator subsystems.	69
63. Figure 63 Example of the full data cube.	72
64. Figure 64 Datacube mapped to RAM block groups. Each color represents one RAM block group storing a spatial position slice.	74
65. Figure 65 UltraRAM memory allocation scheme across RAM block groups showing spatial column progression and frame organization.	75
66. Figure 66 Kernel memory allocation across RAM block groups with Z-dimension slicing.	77
67. Figure 67 Memory Manager Internal implementation showing RAM selection and control logic.	78
68. Figure 68 RAM Memory block configured as Ultra RAM primitive.	79
69. Figure 69 Frame Memory Controller Part 1: Input packing and write control logic.	79
70. Figure 70 Frame Memory Controller Part 2: Read control and RAM selection logic.	80
71. Figure 71 Kernel Memory Controller Part 1: Weight input and packing logic.	81

LIST OF FIGURES – CONTINUED

Figure	Page
72. Figure 72 Kernel Memory Controller Part 2: Memory addressing and read control.	81
73. Figure 73 RAM Selector implementing multiplexing between five Ultra RAM blocks.	82
74. Figure 74 Memory row alignment showing 2:3 read ratio between frame and kernel memory.	83
75. Figure 75 Data Slicer top-level implementation coordinating frame and kernel data unpacking.	84
76. Figure 76 72-Bit Unpacker for frame data showing extraction of six 12-bit values.	84
77. Figure 77 72-Bit Unpacker connected to RAM output with timing control.	85
78. Figure 78 Data Point Tracker coordinating unpacked data with convolution timing.	86
79. Figure 79 Stride-5 operation with partial accumulation showing how intermediate windows are computed across stride boundaries.	87
80. Figure 80 Pipelined MAC accumulation structure for a single slice, showing 12 multiply operations feeding into hierarchical accumulation stages, followed by a slice accumulator that collects 170 accumulations. The bottom shows all 5 slice accumulators feeding into the final accumulator that produces the valid kernel result.	89
81. Figure 81 Math Block top-level implementation of the Convolution Engine.	90
82. Figure 82 Multiple Add tree for parallel accumulation of products.	91
83. Figure 83 Multiple Add internal implementation showing adder tree structure.	92
84. Figure 84 Accumulator Offset module for addressing partial sum storage.	93

LIST OF FIGURES – CONTINUED

Figure	Page
85. Figure 85 Yocto Project How It Works [45].....	113

LIST OF LISTINGS

Listing		Page
1. Listing 1	and-gate.vhd	32
2. Listing 2	d-flip-flop.vhd	35
3. Listing 3	async-reset.vhd	37
4. Listing 4	sync-reset.vhd	38
5. Listing 5	and-gate-registered.vhd.....	39
6. Listing 6	half-adder.vhd.....	46
7. Listing 7	ripple-carry-adder-4bit.vhd	47
8. Listing 8	signed-multiplier.vhd	50
9. Listing 9	counter-4bit.vhd.....	51
10. Listing 10	fifo-sync.vhd	55
11. Listing 11	fifo-async.vhd.....	57
12. Listing 12	Kernel_Setup_Script.m	105
13. Listing 13	Kernel_Setup_Script.m	106
14. Listing 14	Kernel_Setup_Script.m	108
15. Listing 15	Kernel_Setup_Script.m	109
16. Listing 16	meta-msu-de10-nano Layer	114
17. Listing 17	audiomini-drivers.bb: 1-7	117
18. Listing 18	audiomini-drivers.bb: 9.....	118
19. Listing 19	audiomini-drivers.bb: 11-14.....	118
20. Listing 20	audiomini-drivers.bb: 17	119
21. Listing 21	audiomini-drivers.bb: 21	119
22. Listing 22	audiomini-drivers.bb: 23	120
23. Listing 23	audiomini-drivers.bb: 25-34.....	120
24. Listing 24	audiomini-drivers.bb: 37	120

LIST OF LISTINGS – CONTINUED

Listing		Page
25. Listing 25	audiomini-drivers.bb: 40-41	121
26. Listing 26	linux-socfpga_%.bbappend: 1-6	121
27. Listing 27	de10-nano-audio-mini-devicetree.bbappend: 1-8	122
28. Listing 28	layer.conf: 1-7	123
29. Listing 29	de10-nano-audio-mini.conf: 1-7	124
30. Listing 30	de10-nano-audio-mini.conf: 9-24	125
31. Listing 31	de10-nano-audio-mini.conf: 45-52	125
32. Listing 32	de10-nano-audio-mini.conf: 55-67	126
33. Listing 33	de10-nano-audio-mini.conf: 77-95	126
34. Listing 34	de10-nano-audio-mini.conf: 111-124	127
35. Listing 35	audio-mini-image-annotated.bbclass: 6	129
36. Listing 36	audio-mini-image-annotated.bbclass: 25	130
37. Listing 37	audio-mini-image-annotated.bbclass: 28-29	130
38. Listing 38	audio-mini-image-annotated.bbclass: 31-51	131
39. Listing 39	audio-mini-image-annotated.bbclass: 54-62	132
40. Listing 40	audio-mini-passthrough.bb	133
41. Listing 41	audio-mini-combfilter.bb	133
42. Listing 42	recipes-bsp/u-boot/	135
43. Listing 43	recipes-bsp/device-tree/	135
44. Listing 44	recipes-bsp/audio-mini-bitstream/	136
45. Listing 45	recipes-kernel/linux/	137
46. Listing 46	wic/	138

ABSTRACT

This work supports the SMART FIRES project, which is an EPSCoR track 1 program with the goal of helping wildfire management. The SMART FIRES initiative aims to develop technologies for wildfire prediction and mitigation, with a particular focus on supporting prescribed fire planning and burning. This thesis contributes to these efforts by enabling the real-time analysis of hyperspectral imagery to identify fuel sources and their conditions directly in the field. This project presents a custom hardware accelerator for convolutional neural networks on Field-Programmable Gate Arrays, implementing memory controllers, data unpacking, and multiply-accumulate operations to process hyperspectral datacubes in real time. The hardware design uses circular buffering and specialized memory allocation to efficiently handle 3D convolution operations on hyperspectral data. An embedded Linux software stack using the Yocto Project creates a custom Board Support Package with bootloader configurations, kernel modifications, device drivers, and deployment tools. This work demonstrates effective hardware-software co-design for embedded systems, allowing for effective development and deployment of hardware and software for embedded systems.

BACKGROUND

Wildfire management systems face a challenge: modernization. There is a lack of quantitative analysis to make informed decisions, and the workers must rely on intuition and qualitative analysis on where and when to allocate resources in specific areas for wildfire prevention and mitigation. Current wildfire management systems are unable to process data in an accurate and timely manner. Forest managers cannot rely on the current systems if they must make a decision in a short timeframe outside the scope of the current systems.

To make quantitative decisions, the data must be processed off-site. This approach is problematic as it can take days to process and analyze the data if there is not already a queue of data to be processed. This delay is not acceptable for many situations, even when the data is intended for maintenance activities like prescribed burns, the processing time needed may be too long to be actionable. By the time the data is processed and analyzed, it may no longer be valid due to changing environmental conditions. This limitation drives the need for a real-time solution.

The goal of the SoS team is to develop a "Smart Optical Sensor" that can collect data in real-time and analyze/classify it immediately to support decisions based on both quantitative and qualitative analysis. To process this data in real-time, hardware acceleration is required, and an SoC FPGA is the device for this application. The SoC FPGA serves as the "Smart" component of the "Smart Optical Sensor".

- **Reducing Response Time:** Developing systems capable of processing and analyzing data in real time to allow for faster and more accurate decisions. Real Time for this application will be defined as being able to capture and process data within the capture rate of the optical sensor the SoC FPGA is connected to.

- **Improving Accuracy:** Using machine learning models and hyperspectral data to improve the precision of wildfire predictions.
- **Eliminating Offline Dependencies:** Transitioning from offline post-processing to on-the-fly and on-site data analysis using hardware acceleration.

The problem addressed in this work is the implementation and acceleration of real-time machine learning classification for forestry maintenance to classify the fuel characteristics of the area. Specifically, this will involve using hardware acceleration for kernels to process hyperspectral data. The collected hyperspectral data must be processed immediately to achieve a real-time response. The specific problems include:

- **Data Movement Bottlenecks:** Transferring data between CMOS, DRAM, FPGA fabric, and Linux user space.
- **Kernel Design:** Designing hardware-accelerated kernels tailored for ingesting data from the CMOS and transferring results to the DRAM.
- **Memory Management:** Managing the hyperspectral data cube in the FPGA fabric.

The objectives of this research are as follows:

- **Develop a 3D Circular Buffer:** Design a data management system to handle hyperspectral data cubes.
- **Implement Hardware-Accelerated Kernel Operations:** Develop kernel operations that use the FPGA fabric for real-time processing.
- **Validate the Framework:** Test and validate the framework using a variety of inputs to test edge cases and ensure the design is correct and functional.

- **Operating System Integration:** Propose an operating system and development pipeline for the SoC FPGA for a development environment that integrates with existing or future software and hardware designs.

SMART FireS

Sensors, Machine Learning, and Artificial Intelligence (AI) in Real-Time (SMART) for Fire Science (FireS) is a research program funded by the National Science Foundation (NSF) to develop a real-time wildfire detection and management system. This program is an Established Program to Stimulate Competitive Research (EPSCoR) Track-1 project. A collaboration between multiple Montana universities and colleges, its goal is to develop a system that can be used to identify and mitigate wildfire destruction through prescribed burns. To achieve this, the research is organized into four areas: Fire and Smoke Science, to understand the air-quality impacts of prescribed-fire emissions; Smart Optical Sensors (SOS), to develop new ground and UAS-based sensors for real-time data collection; Social Psychology, Economics, and Ethics (SPEE), to improve how the benefits and risks of prescribed fires are communicated to the public; and Artificial Intelligence and Machine Learning (AIML), to process data and create predictive models for fire behavior. Partners in this initiative include the Montana Department of Environmental Quality, the U.S. Forest Service Missoula Fire Sciences Laboratory, and other state agencies. The research in this thesis advances the objectives of the Smart Optical Sensors (SOS) team [32]. An overview of the SMART FIRES program is shown in Figure 1.

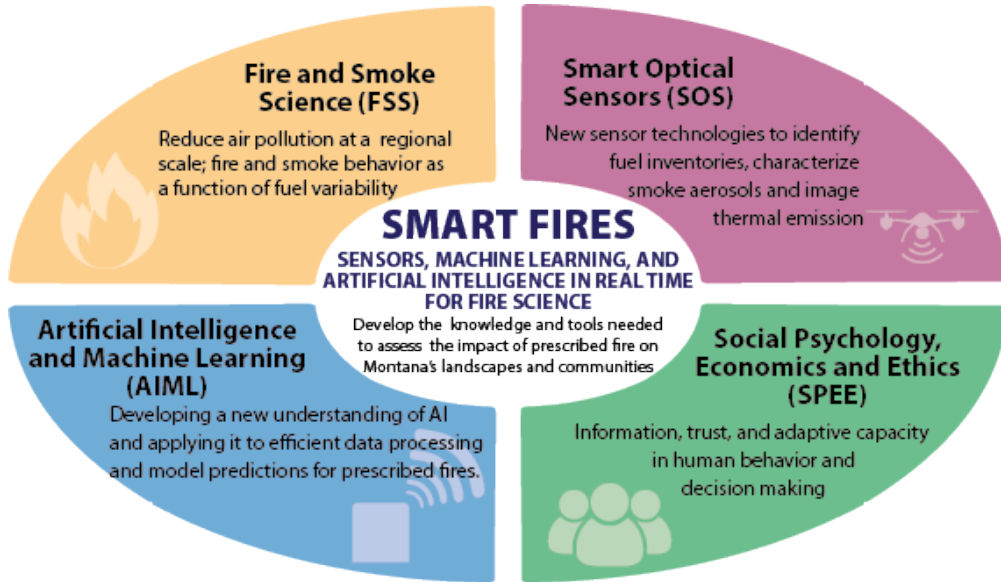


Figure 1: Overview of the SMART FIRES program [32].

Wildfires

Wildfires, also known as forest or bush fires, are uncontrolled fires that rapidly spread across vegetation. These fires are influenced by factors including weather conditions, fuel availability, topography, and human activity. The most significant predictor of fire intensity is the amount of available fuel such as grasses, trees, and other organic matter [51].

Understanding the behavior of wildfires is needed for management and mitigation. For a fire to begin, there must be a source of ignition, a fuel to burn, and oxygen. Without one of these three elements, a fire cannot start. These three elements are described as the fire triangle as seen in Figure 2 [36].



Figure 2: The Fire Triangle [37].

The science of fire behavior is explained through the fire-behavior triangle, which consists of three components: fuel, topography, and weather, as shown in Figure 3 [38]. Fuel encompasses grass, shrubs, and timber, with flammability determined by moisture content, size, and fuel-bed continuity (horizontal and vertical connectivity). Topography contributes to fire spread; for example, fires accelerate uphill due to rising heat that preheats fuels above. Weather represents the most variable component, with wind serving as the driver of fire spread, temperature affecting fuel preheating, and low relative humidity increasing fuel flammability [38].

While wildfires present challenges, they also play a role in fire-adapted ecosystems [36]. Natural fire cycles provide ecological benefits, including nutrient cycling, reduction of dead biomass that acts as fuel, control of competing vegetation, and stimulation of seed germination for fire-adapted species [36]. Wildfires can be harmful when they occur in inappropriate locations and have severe consequences, including loss of life, property destruction, and infrastructure damage [63]. Additional impacts include degraded air quality from smoke, which poses public health risks, and watershed damage that can lead to erosion

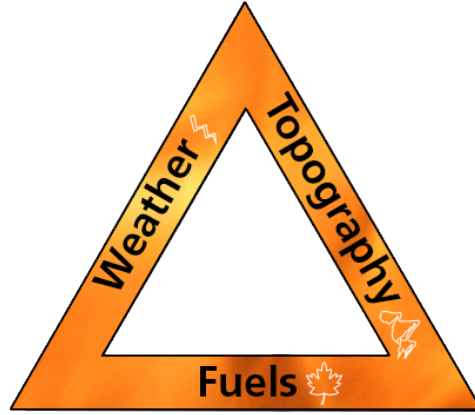


Figure 3: The Wildfire Behavior Triangle [38].

and flooding [36]. These events also result in economic disruption across affected regions [63].

Over the years, wildfire damage has increased, while the total number of fires per year has remained consistent. This trend is illustrated in Figures 4 and 5 below [61]. Contributing factors include climate change, human activity, natural disasters, and a lack of fire management.

Current wildfire monitoring approaches utilize remote sensing technologies, including satellite imagery and airborne systems, to detect and track fire activity in real time. NASA, for example, has developed remote sensing technologies that improve the speed and accuracy of information relayed to firefighting decision-makers [33]. However, these systems lack the capability to deliver on-the-ground analysis in real time due to their reliance on post-processing and data parsing, which can delay decision-making processes. The research outlined in this paper addresses this limitation by developing solutions that minimize processing time for wildfire management efforts.

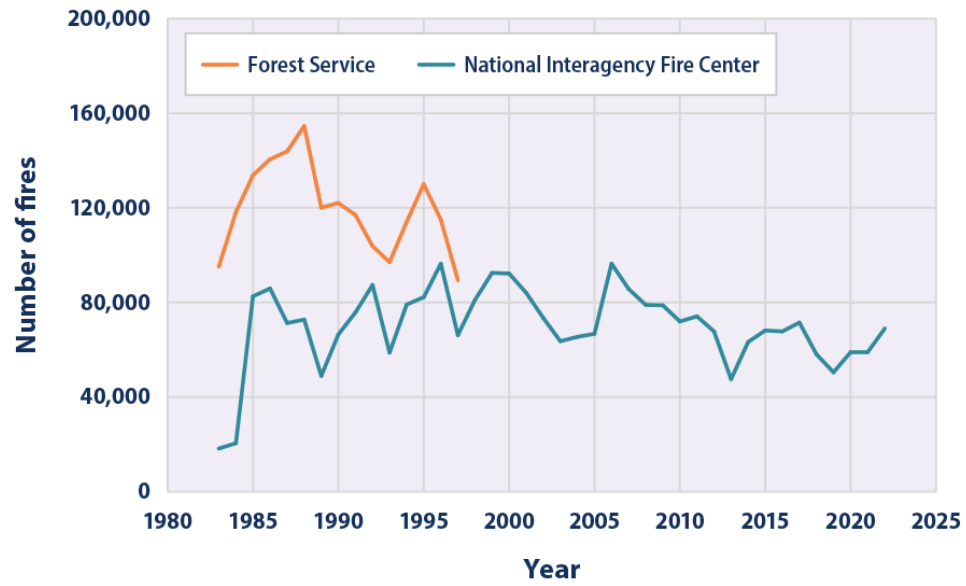


Figure 4: Wildfire Frequency in the United States, 1983–2022 [61].

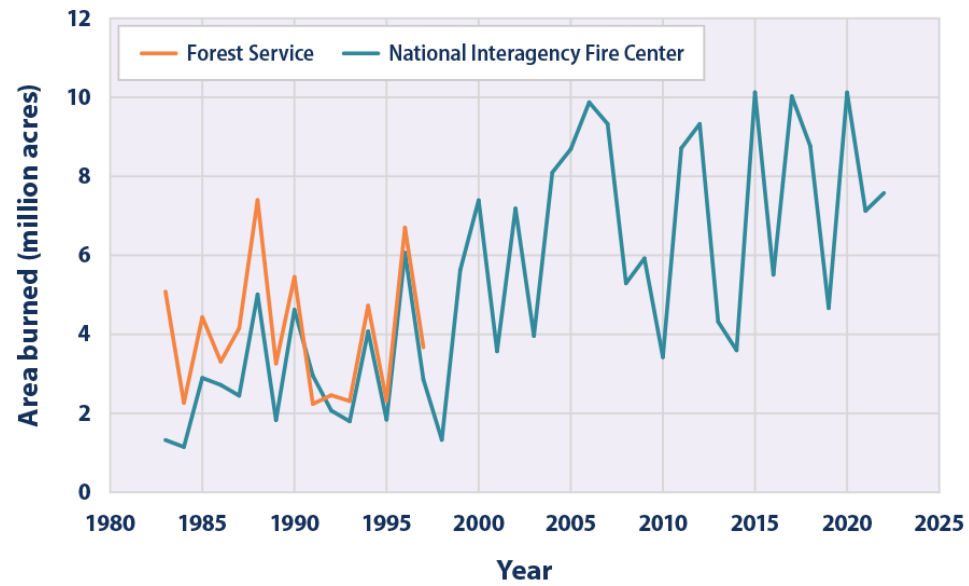


Figure 5: Wildfire Extent in the United States, 1983–2022 [61].

Hyperspectral Imaging

Hyperspectral imaging (HSI) is an imaging technology that captures more spectral information than conventional cameras. While standard cameras record images using three broad color bands (red, green, and blue), HSI systems can detect hundreds of narrow spectral bands across the ultraviolet, visible, and infrared spectrum, as seen in Figure 6 [49]. This allows the characterization of different materials based on their spectral properties [65].

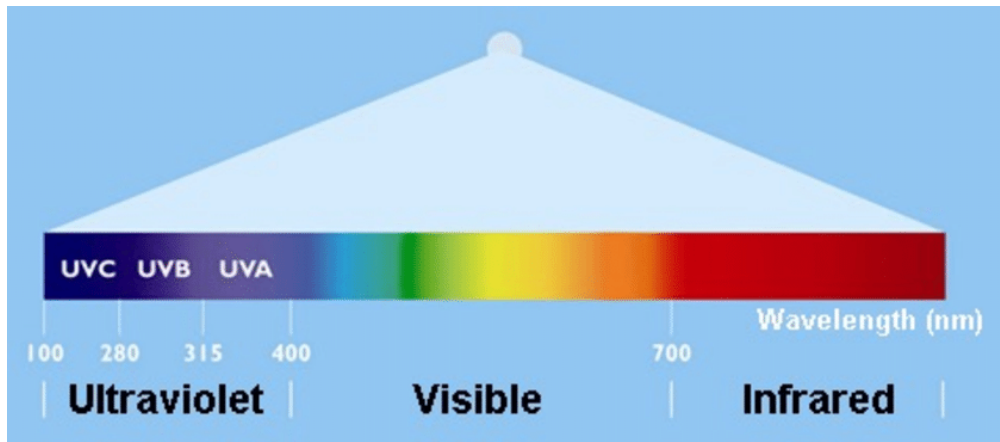


Figure 6: Electromagnetic Spectrum [48].

HSI produces a three-dimensional dataset known as a hyperspectral data cube, which combines spatial imaging with spectral information [49] [65]. This cube contains two spatial dimensions that form the image, plus a third spectral dimension that provides a complete light spectrum for every pixel in the image [57]. This data makes it possible to identify and analyze materials that appear visually similar but have different chemical or physical properties [65].

A challenge in wildfire management is understanding what makes an area likely to burn. Fire managers need to know about the fuel [50]. HSI technology can provide information on fuel in an area.

With HSI, it is possible to identify vegetation types—distinguishing not just between

trees and grasses, but even among plant species—without the need for a ground survey [54] [30] [60]. HSI can differentiate live, healthy vegetation from dead, dry fuels and estimate fuel moisture [54] [53]. This information helps fire managers create accurate models of wildfire behavior, allowing more quantitative decisions about where to focus prevention efforts and how to plan prescribed burns [60] [53].

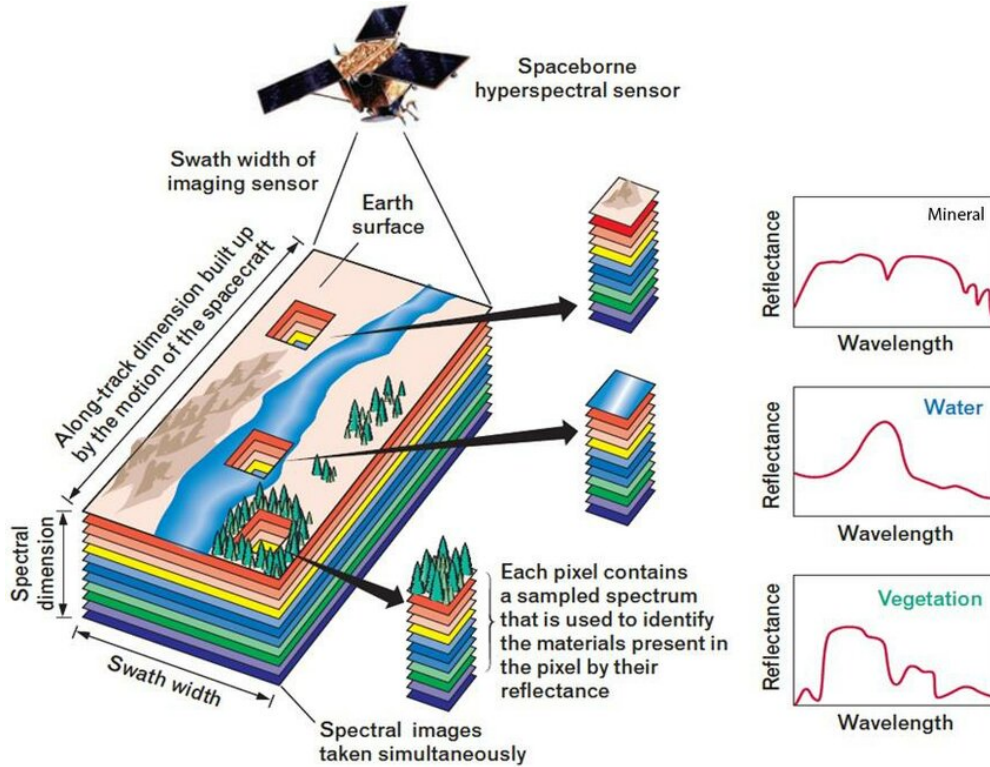


Figure 7: Hyperspectral Imaging Overview [49].

Machine Learning

Machine learning (ML) is a field of study that gives computers the ability to generate outputs based on inputs without being explicitly programmed. The quality of a machine learning model is measured by its ability to make accurate predictions on new, unseen data, not just the data it was trained on. The process of applying a trained model to new data to make these predictions is known as inference [17].

Primary Learning Approaches

Machine learning is typically categorized into three main categories: supervised learning, unsupervised learning, and reinforcement learning.

Supervised learning A model is trained on a dataset containing labeled examples, where the correct output is already known. The goal is to learn a function that maps inputs to outputs [18].

- *Classification*: Predicts a discrete, categorical label (e.g., classifying an email as “spam” or “not spam”).
- *Regression*: Predicts a continuous, numerical value (e.g., forecasting the price of a house).

Unsupervised learning Uses datasets with no corresponding output labels to discover hidden patterns or intrinsic structures, such as via clustering which can be used to find groups of similar data points [18].

Reinforcement learning An autonomous agent learns to make decisions by interacting with an environment, receiving rewards or penalties, and developing a strategy to increase rewards over time [17].

Classification Algorithms

Classification is the process of categorizing data points into predefined groups, called classes, using a trained model. A few algorithms include:

Support Vector Machines (SVM) Find a separating hyperplane that increases the margin between classes, a property that supports strong generalization [18].

Decision Trees and Random Forests Decision Trees provide interpretable, rule-based models [18].

Artificial Neural Networks (ANN) Computational models composed of interconnected layers of neurons. Deep networks excel at modeling complex, non-linear patterns but can be harder to interpret [17].

Convolutional Neural Networks

In the context of ANNs, Convolutional Neural Networks (CNNs) are a specialized type designed to process data for classification and computer vision tasks. They are primarily useful due to utilizing matrix multiplication to identify patterns in data, especially in images [20].

Multiple different architectures of CNNs have been developed over the years. LeNet-5 is the first useful CNN architecture that was developed in 1998 by Yann LeCun [26]. Its main focus was to classify handwritten digits in the Modified National Institute of Standards and Technology (MNIST) dataset [27]. It did this by implementing seven layers, three convolutional layers, two sub-sampling layers, and two fully connected layers. Its architecture is shown in Figure 8.

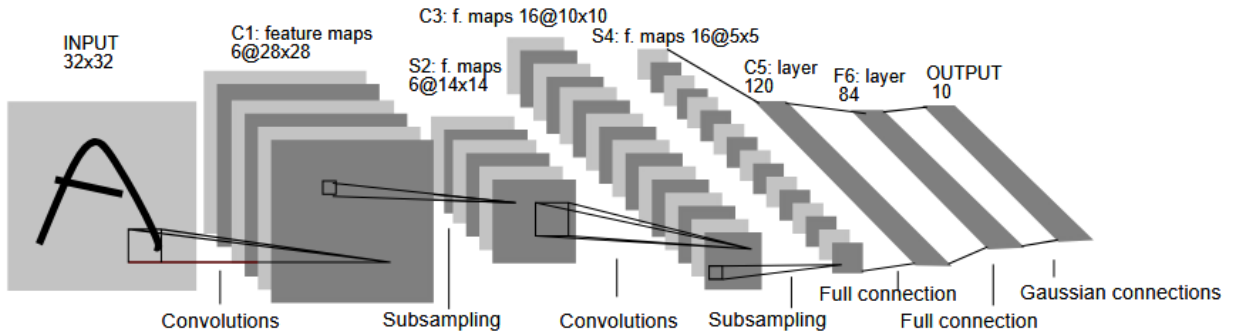


Figure 8: LeNet-5 Architecture [26].

Convolutional layers are the main component of CNNs that are responsible for extracting features from the input data. The layer requires a few inputs to be able to perform the convolution operation. These inputs are the input data, the kernel, the stride,

and the padding. The input data is traditionally an image of some kind, where it represents 3 dimensions, height, width, and color channels, red, green, and blue. The kernel, also known as a filter, is a smaller matrix of values that is used to parse over the input data and retrieve features. The kernel must move across the input data in a specific manner determined by the stride. This process is known as a convolution operation and the stride determines how many positions the kernel moves across the input data. Padding is the process of adding zeros to the edges of the input data to prevent the kernel from moving off the edges of the input data, and to maintain the same size of the output data if desired. The output of the convolution operation is a feature map, which is a matrix of values that represents the features extracted from the input data.

Depending on the input size, the kernel size, the stride, and the padding, the output size of the convolution operation can be calculated. The output size is calculated by the following equation 1 [12].

$$output_{size} = \frac{input_{size} + 2 \times padding - kernel_{size}}{stride} + 1 \quad (1)$$

So, for example, let us calculate the output size for the following parameters. Let the input matrix be (5, 5, 1), the kernel matrix be (3, 3, 1), the stride be 1, and the padding be 0. Using equation 1 to calculate the size of each dimension. The output size for the x dimension is calculated in 2, the y dimension is calculated in 3, and the z dimension is calculated in 4.

$$\begin{aligned}
output_{x_{size}} &= \frac{5 + 2 \times 0 - 3}{1} + 1 \\
&= \frac{5 + 0 - 3}{1} + 1 \\
&= \frac{2}{1} + 1 \\
&= 2 + 1 \\
&= 3
\end{aligned} \tag{2}$$

$$\begin{aligned}
output_{y_{size}} &= \frac{5 + 2 \times 0 - 3}{1} + 1 \\
&= \frac{5 + 0 - 3}{1} + 1 \\
&= \frac{2}{1} + 1 \\
&= 2 + 1 \\
&= 3
\end{aligned} \tag{3}$$

$$\begin{aligned}
output_{z_{size}} &= \frac{1 + 2 \times 0 - 1}{1} + 1 \\
&= \frac{1 + 0 - 1}{1} + 1 \\
&= \frac{0}{1} + 1 \\
&= 0 + 1 \\
&= 1
\end{aligned} \tag{4}$$

The output size for the x dimension is 3, the y dimension is 3, and the z dimension is 1. So, the output size of the convolution operation is (3, 3, 1). For a visual step-by-step example of the convolution output size matching the equation, see Figure 9. This figure shows all 9 steps of the convolution operation, where the 3×3 kernel (shown in red on the input matrix) slides across the 5×5 input matrix. At each step, the kernel position on the input corresponds to one output value in the 3×3 output matrix (shown in orange on the output).

2D Convolution: Stride=1, Padding=0

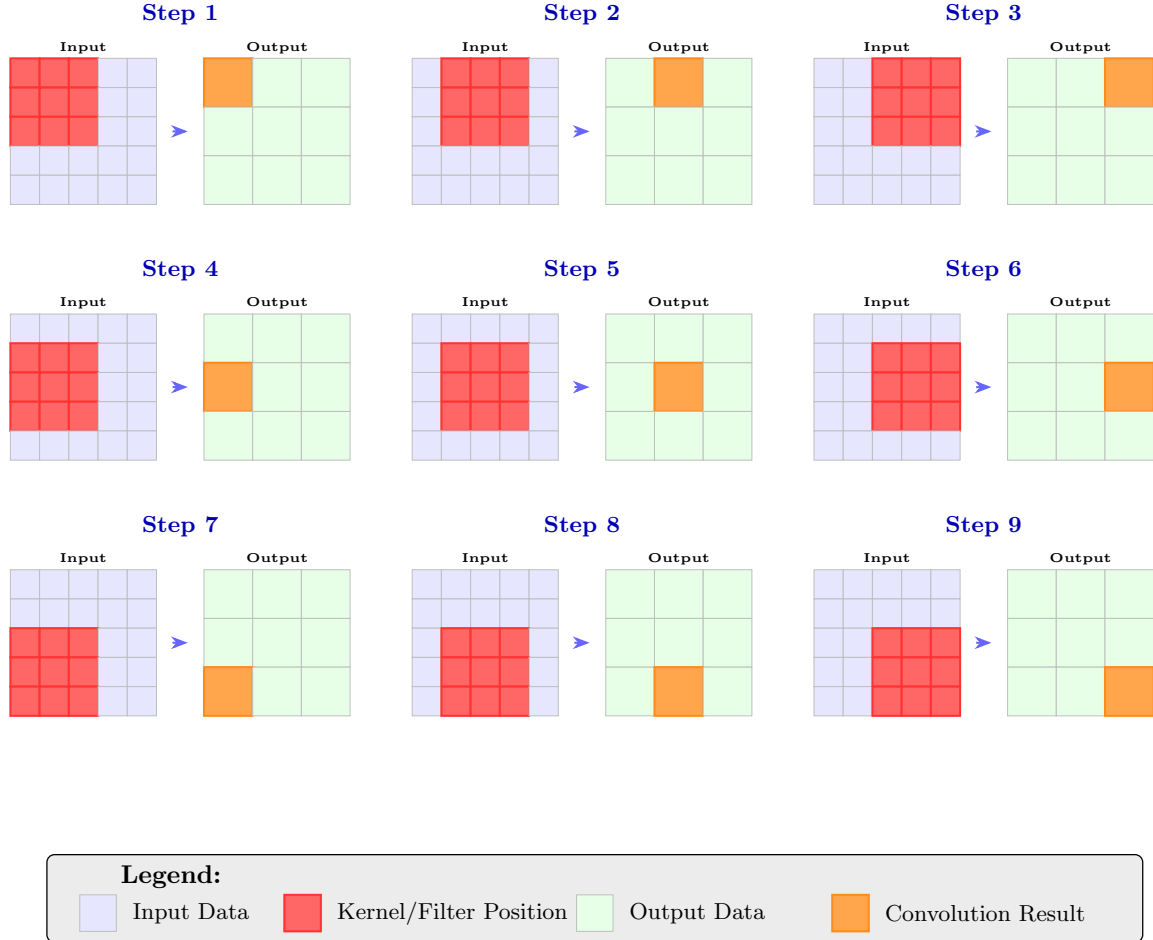


Figure 9: Step-by-step visualization of a 2D convolution operation with a 3×3 kernel sliding over a 5×5 input matrix (stride=1, padding=0). Each step shows the kernel position (red) on the input and the corresponding output location (orange) in the resulting 3×3 output matrix.

When we have the output size of $(3, 3, 1)$, this means that we must perform a convolution operation 9 separate times to obtain the entire output matrix. This is because the kernel slides to 9 distinct positions (3 horizontally $\times$ 3 vertically) to cover the valid regions of the input matrix (assuming, for example, a 5×5 input and a 3×3 kernel with

stride 1 and no padding). This is shown in Figure 9. Each convolution output can be modeled as a general function based on an arbitrary dimension size; this can be seen in equation 5 and, for a discrete convolution operation, in equation 6, where both equations are for an M-dimensional convolution operation [17].

$$y(n_1, n_2, \dots, n_M) = (x * h)(n_1, n_2, \dots, n_M) \quad (5)$$

$$output_{n_1, n_2, \dots, n_M} = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} \dots \sum_{k_M=-\infty}^{\infty} h(k_1, k_2, \dots, k_M) x(n_1 - k_1, n_2 - k_2, \dots, n_M - k_M) \quad (6)$$

In regards to equations 5 and 6, the y or *output* is the final data like a filtered image. The x is the input, which is the original data. The term $x(n-k)$ (or $x(n_1-k_1, n_2-k_2, \dots, n_M-k_M)$ in multiple dimensions) shows how to get input values. Here, n is the spot in the output, and k loops over the kernel. $n-k$ shifts to the right input spot for each k . This lets the kernel slide over the input and flips it for the math. For example, in 1D with kernel size 3 ($k = -1, 0, 1$) at $n = 2$, it gets $x(3)$, $x(2)$, $x(1)$. In real use, sums are only over the kernel's size, and padding handles edges. The h is the kernel, a set of weights for operations. In the equations, it appears as h in the shorthand notation and as $h(k)$ for specific weights in the summation. The n is the output index, appearing as (n) in the shorthand and serving as the anchor point in the summation. The k is the kernel index, hidden by the $*$ symbol in the shorthand and appearing as the summation index Σ_k in the detailed form. We can model a 3-dimensional discrete convolution for an input of $(5, 5, 1)$ and a kernel of $(3, 3, 1)$ as shown in equation 7. This is technically a 2-dimensional convolution (since the third dimension is 1), but it illustrates how the general M-dimensional form in equation 6 applies by explicitly showing all three dimensions.

$$output_{n_1, n_2, n_3} = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} \sum_{k_3=-\infty}^{\infty} h(k_1, k_2, k_3) x(n_1 - k_1, n_2 - k_2, n_3 - k_3) \quad (7)$$

The convolution operation is performed by sliding the kernel over the input data and computing the dot product of the kernel and the overlapping input patch at each position. The dot product is performed by element-wise multiplication of the kernel and the input patch, followed by summing the results. This is shown in equation 8.

To illustrate the dot product as a matrix operation for the 3D case (with the trivial third dimension of depth 1), consider an example input patch and kernel, both $3 \times 3 \times 1$. The dot product computes:

$$output(n_1, n_2, n_3) = \sum_{k_1=0}^2 \sum_{k_2=0}^2 \sum_{k_3=0}^0 h(k_1, k_2, k_3) \cdot x(n_1 - k_1, n_2 - k_2, n_3 - k_3) \quad (8)$$

For a specific example, the process starts with the input patch from the overlapping region of the input data, as shown in Figure 10. This patch represents the portion of the input that aligns with the kernel at a given position n . Next, the original kernel (or filter) is depicted in Figure 11, containing the weights used for the filtering operation. Due to the mathematical definition of convolution involving a flip (from the $n - k$ term), the kernel is reversed along each dimension before multiplication, resulting in the flipped kernel shown in Figure 12. Then, element-wise multiplication is performed between the input patch and the flipped kernel, yielding the matrix in Figure 13, where each corresponding pair of elements is multiplied. Finally, these products are summed to produce a single scalar value for the output at position (n_1, n_2, n_3) : Sum: $ax + bw + cv + du + et + fs + gr + hq + ip$. The figure 14 shows a visual example of the convolution operation for $(5, 5, 1)$ input and $(3, 3, 1)$ kernel

with numbers filled in.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

Figure 10: Input patch from input data

$$\begin{pmatrix} p & q & r \\ s & t & u \\ v & w & x \end{pmatrix}$$

Figure 11: Kernel/Filter (original)

$$\begin{pmatrix} x & w & v \\ u & t & s \\ r & q & p \end{pmatrix}$$

Figure 12: Flipped kernel/filter (for convolution)

$$\begin{pmatrix} ax & bw & cv \\ du & et & fs \\ gr & hq & ip \end{pmatrix}$$

Figure 13: Element-wise multiplication (with flipped kernel/filter)

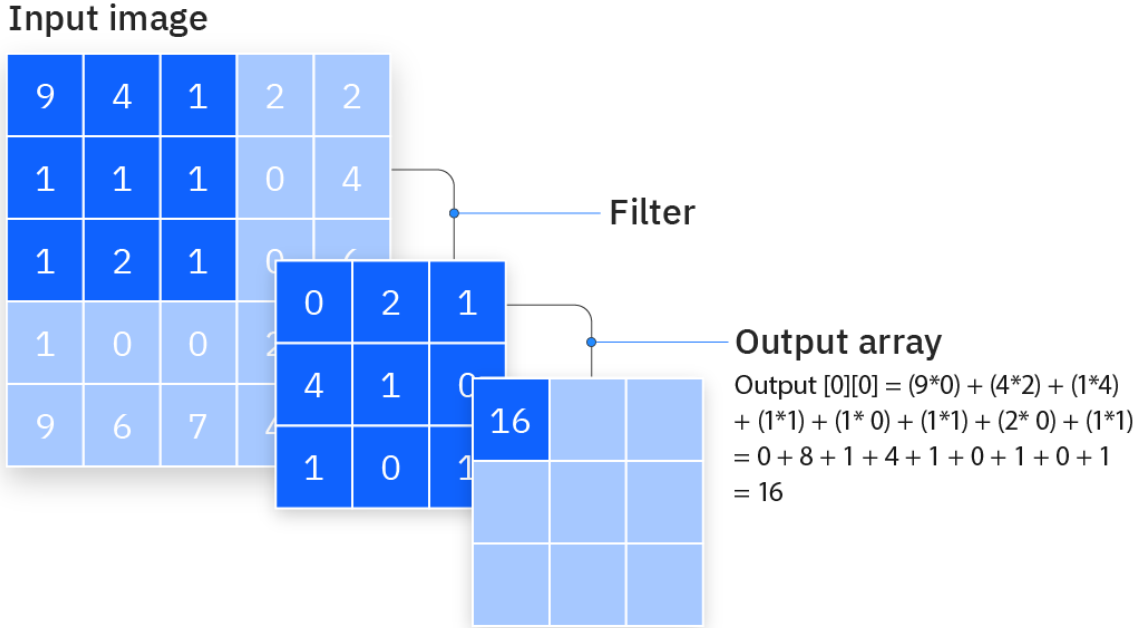


Figure 14: Visual example of the convolution operation for (5, 5, 1) input and (3, 3, 1) kernel[20].

Hardware Acceleration for Machine Learning

Hardware acceleration is a technique that improves the performance of a machine learning model by using specialized hardware to perform the computations. These improvements can be in training time or inference time. These hardware speedups can be accomplished through the use of Application-Specific Integrated Circuits (ASICs). For modern deep learning, it is very common to use hardware acceleration with Graphics Processing Units (GPUs), which are well suited to the linear algebra operations common in neural networks along with the wealth of software suites provided by NVIDIA and AMD [17]. However, GPUs are not the only hardware acceleration option. Field-Programmable Gate Arrays (FPGAs) are also a popular choice for hardware acceleration; however, they require more specialized hardware and software to be used effectively.

FPGAs, SoC FPGAs, and SoMs

Field-Programmable Gate Arrays (FPGAs)

A Field-Programmable Gate Array (FPGA) is a semiconductor device that is essentially a configurable Application-Specific Integrated Circuit (ASIC). Its defining feature is that its internal hardware is not set at the factory and can be reprogrammed by the developer to meet the needs of the application. This allows for a high degree of prototyping flexibility without the lead time and cost of an ASIC.

An FPGA is a generic piece of silicon made up of thousands of interconnected hardware elements that can be programmed to form basic hardware components; the general term for this combination of hardware is called a fabric. An example of this combination of hardware can be seen in Figure 15.

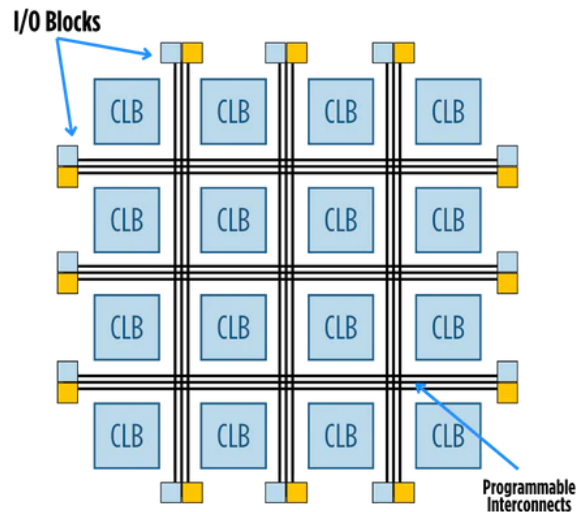


Figure 15: Generic FPGA Fabric Diagram [11].

The elements of FPGA fabric include:

Configurable Logic Blocks (CLBs) The basic units of logic, replicated thousands or

millions of times. Each CLB contains Lookup Tables (LUTs), which are small memory blocks that can be programmed to implement Boolean logic functions, flip-flops, which store state and enable sequential logic, and other hardware elements depending on the vendor and the FPGA family [6]. Seen in Figure 16.

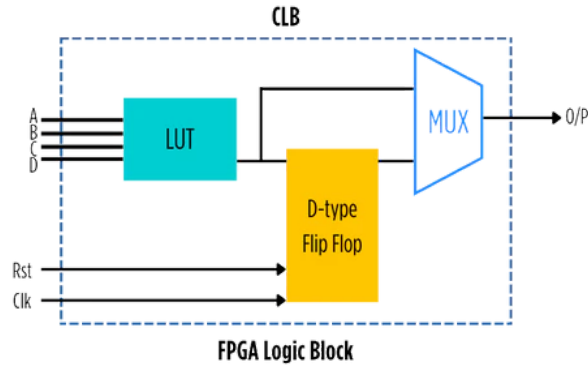


Figure 16: Generic FPGA CLB Diagram [11].

Programmable interconnects A vast network of configurable wires and switches that connect the CLBs and other resources to form a custom circuit [6]. Seen in Figure 17.

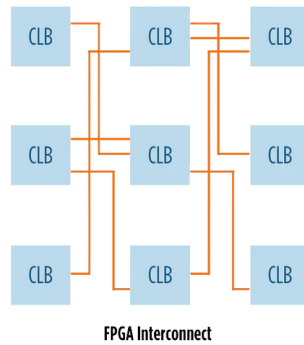


Figure 17: Generic FPGA Interconnect Diagram [11].

DSP Slices High-performance blocks specialized for mathematical operations such as multiplication, subtraction, and addition [1]. Seen in Figure 18.

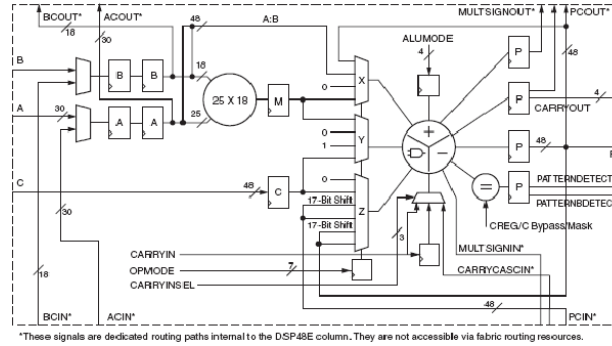


Figure 18: Xilinx DSP Slice (DSP48E) [1].

Block RAM (BRAM) These are dedicated memory blocks that can be used to store data within the FPGA's fabric. This can be dedicated hardware or built up with the CLBs [6].

Input Output (I/O) Blocks These control the signals flowing into and out of the FPGA.

Seen in Figure 19.

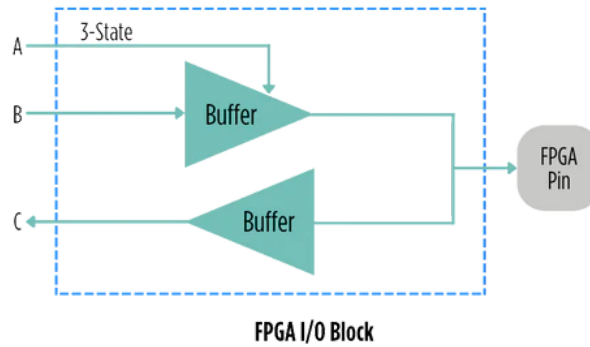


Figure 19: Generic FPGA IO Block Diagram [11].

FPGAs are programmed using a Hardware Description Language (HDL), which describes the physical circuit itself. This description is compiled through a synthesis and place-and-route process that physically maps the logical design onto the FPGA's resources, generating a final bitstream file for configuration. The two most common HDLs are System Verilog and VHDL. Using these HDLs, FPGAs can achieve massive parallelism. Unlike a CPU that processes instructions sequentially, an FPGA implements thousands of custom,

dedicated circuits that all operate simultaneously on every clock cycle. This makes them ideal for high-performance, low-latency tasks like high-frequency trading and real-time signal processing.

System-on-Chip (SoC) FPGAs

A System-on-Chip (SoC) FPGA represents the combination of a complete, hardened Processing System (PS) with the traditional reconfigurable Programmable Logic (PL) on a single silicon die. This creates a powerful hybrid device for building complete embedded systems. An example of a SoC FPGA is the Zynq UltraScale+ MPSoC seen in Figure 20 below [5].

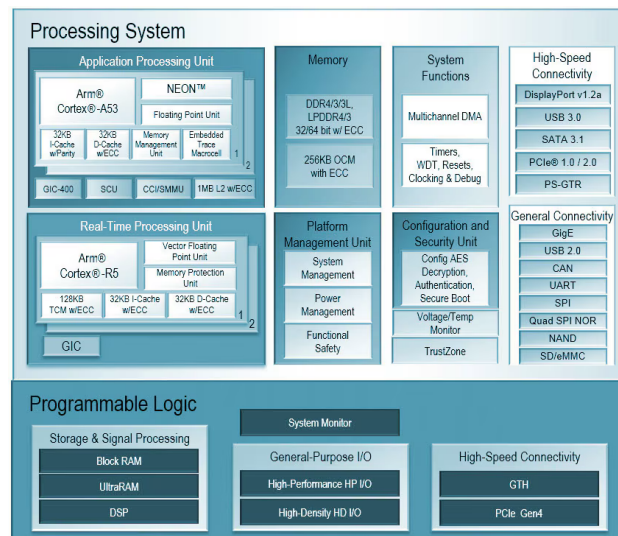


Figure 20: Zynq UltraScale+ MPSoC [5].

Processing System (PS) A hardened microprocessor subsystem, not built from FPGA logic. It typically includes one or more ARM Cortex-A cores, caches, memory controllers, and standard peripherals (Ethernet, USB, etc.). It can run a full operating system like Linux [7].

Programmable Logic (PL) The traditional FPGA fabric, which acts as a powerful, reconfigurable accelerator for the PS.

The PS and PL interact through high-bandwidth, low-latency on-chip buses (e.g., AXI). This allows hardware/software co-design workflows where software on the PS can offload computationally intensive tasks to custom accelerators in the PL. This happens commonly through the use of Direct Memory Access (DMA).

System-on-Module (SoM)

While an SoC FPGA integrates functions at the chip level, a System-on-Module (SoM) integrates them at the board level. An SoM is a small, production-ready PCB that contains an SoC FPGA, DDR memory, flash storage, power management, and other essential support circuitry [2]. The primary value of a SoM is that it reduces hardware development risk. Designing a custom board for a modern SoC FPGA is complex due to high-speed interfaces needed for Double Data Rate (DDR) Memory and Peripheral Component Interconnect Express (PCIe) devices.

Mercury+ XU1 SoM and Mercury ST1 Base Board

The research project utilizes the Mercury+ XU1 SoM module and the Mercury ST1 Base Board. The Mercury+ XU1 SoM module is a small, low-power, and low-cost module that contains an FPGA and other components. The Mercury ST1 Base Board is a larger board that utilizes the Mercury+ XU1 SoM module and pipes out peripherals for the system to use for programming, development, and testing of the FPGA hardware and software. The top of the Mercury+ XU1 SoM module is shown in Figure 21, the bottom of the Mercury+ XU1 SoM module is shown in Figure 22, and the top of the Mercury ST1 Base Board is shown in Figure 23.

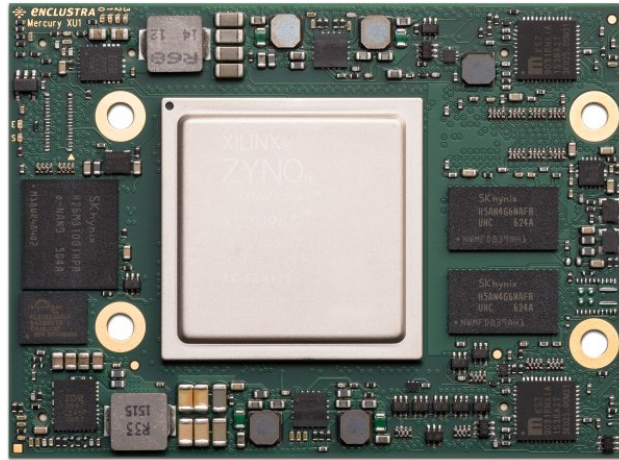


Figure 21: Top View of the Mercury+ XU1 SoM module [13].

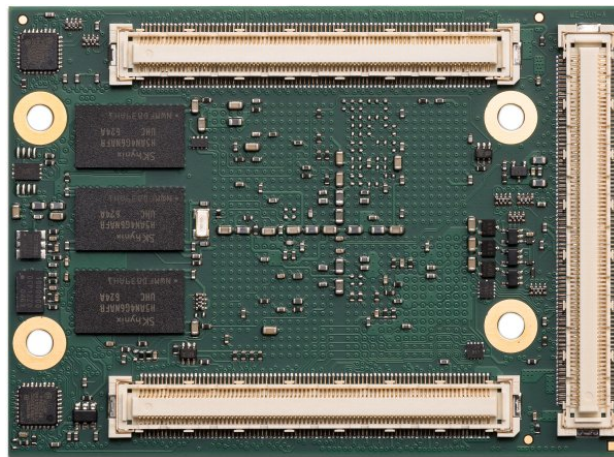


Figure 22: Bottom View of the Mercury+ XU1 SoM module [13].

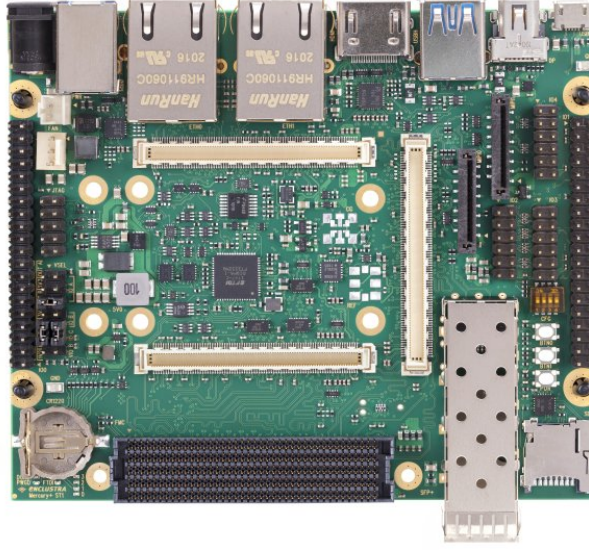


Figure 23: Top View of the Mercury ST1 Base Board [14].

The specific version of the Mercury+ XU1 used in the research project is the ME-XU1-15EG-1E-D12E-G1. This system-on-module (SOM) uses the AMD Xilinx Zynq UltraScale+ XCZU15EG-1FFVC900E MPSoC which is a System-on-Chip (SoC) FPGA [68]. It integrates processing, system memory, and interfaces in a 74 x 54 mm form factor. The XCZU15EG MPSoC has a processing system (PS) and programmable logic (PL). The PS includes a quad-core ARM Cortex-A53 clocked at 1.333 GHz, dual-core ARM Cortex-R5F clocked at 533 MHz, and Mali-400 MP2 GPU clocked at 667 MHz [68]. The SoM provides 4GB of DDR4 SDRAM clocked at 2400 MT/s for the PS/PL to access. The XCZU15EG PL is equipped with 746,550 logic cells, 340,560 CLB LUTs, 681,120 CLB Flip-Flops, 3,528 DSP Slices (up to 2,276 GMAC/s), 26.2 Mb Block Ram, 31.5 Mb Ultra Ram, and 9.8 Mb Distributed Ram [68].

Zynq UltraScale+ MPSoC Programmable Logic Details

The Programmable Logic (PL) section of the Zynq UltraScale+ MPSoC is built on Xilinx's UltraScale+ architecture, offering a versatile FPGA fabric [68]. This fabric supports custom hardware accelerators, specialized I/O interfaces, and tailored digital logic to meet

specific application requirements.

Configurable Logic Blocks (CLBs) form the core of the PL. Each CLB includes:

- **Look-Up Tables (LUTs):** These flexible memory elements can implement any Boolean logic function based on a set number of inputs. In the UltraScale+ architecture, LUTs are typically configured as 6-input LUTs or as two 5-input LUTs with shared inputs. They can also function as 64-bit distributed Ram or 32-bit shift registers (SRL32). Each CLB generally contains eight LUTs.
- **Flip-Flops:** These provide storage for sequential logic, with each CLB typically housing 16 flip-flops.
- **Carry Logic and Cascadable Adders:** Dedicated fast carry chains connect CLBs, allowing arithmetic operations like addition, subtraction, and comparisons.

The number of “System Logic Cells” (a term often tied to LUT count) varies widely across the MPSoC family, ranging from approximately 81,000 in smaller devices like the ZU1CG to over 1.1 million in larger ones like the ZU19EG. This scalability ensures the PL can handle custom logic of varying complexity.

For significant on-chip data storage, the PL includes dedicated 36Kb Block RAMs. These are true dual-port memories, allowing simultaneous read and write operations from two independent ports. Each Block Ram can support data widths up to 72 bits or be split into two 18Kb dual-port RAMs. They also feature Error Correction Code (ECC) for improved data reliability.

With ECC enabled, Block RAMs can operate at up to 718 MHz. The total Block Ram capacity varies by device, from 5.3 Mb in the ZU2EG to 34.6 Mb in the ZU19EG. These resources are used for implementing FIFOs, data buffers, and coefficient storage in PL designs.

Larger Zynq UltraScale+ MPSoC devices often include Ultra Ram blocks, which are high-density, dual-port synchronous memories offering 288Kb of storage per block with a 72-bit data path. Like Block RAMs, Ultra Ram supports ECC for data integrity and can be cascaded to create large on-chip memory arrays, up to 36Mb in some cases.

Ultra Ram is typically found in mid-to-high-end devices, such as the ZU4EV (14.0Mb) and the ZU19EG (36.0Mb). Smaller CG and some EG devices may lack Ultra Ram. Without output register pipelining and with ECC enabled, Ultra Ram achieves a maximum frequency of around 435 MHz. Ultra Ram is suitable for applications needing large on-chip buffers, like video frame storage or extensive datasets, where Block Ram might require too many blocks.

For math-intensive tasks like digital signal processing, the PL includes dedicated DSP slices. Each slice typically features:

- A 27×18-bit signed multiplier.
- A 48-bit adder/accumulator.
- A 27-bit pre-adder.

These low-power hardware blocks offer superior performance and efficiency for multiply-accumulate (MAC) operations and similar arithmetic tasks compared to general-purpose CLB logic. The number of DSP slices varies, from 216 in the ZU1CG to over 2,500 in high-end devices like the ZU9CG. They are critical for applications like wireless communications, image processing, radar, and any field requiring intensive filtering or matrix operations.

The PL provides connectivity options for interfacing with external devices.

- **Programmable I/O Blocks:** These support various I/O standards, including LVCMOS, LVDS, and SSTL, with voltage levels from 1.0V to 3.3V. They include programmable delay elements and integrated serializer/deserializer circuits (ISERDES for input, OSERDES for output) for high-speed parallel bus interfaces. I/O blocks are

divided into High-Performance (HP) and High-Density (HD) banks, each with distinct electrical characteristics.

- **High-Speed Serial Transceivers:** For high-bandwidth communication, the PL includes multi-gigabit transceivers:
 - **GTH Transceivers:** Support data rates up to 16.3 Gb/s per lane.
 - **GTY Transceivers:** Support data rates up to 32.75 Gb/s per lane.

The number of GTH and GTY transceivers depends on the device. These transceivers support standard protocols like PCI Express (up to Gen3, or Gen4 with GTY in some cases), 10/25/40/50/100 Gigabit Ethernet, Interlaken, and Serial RapidIO, as well as custom serial links. Integrated hard IP blocks for PCIe, 100G Ethernet, and Interlaken reduce PL resource usage and simplify design for these high-speed interfaces. Together, the flexible parallel I/Os and advanced serial transceivers allow the PL to connect with a wide range of external components and systems.

Ultra Ram in the Programmable Logic

Ultra Ram is designed for high storage density and differs significantly from the more flexible Block Ram (BRAM) found in AMD devices [68]. Each Ultra Ram block is a synchronous, 288-kilobit (Kb) memory with a fixed configuration of 4,096 words by 72 bits, often denoted as 4K×72. Unlike BRAM, which supports flexible width and depth configurations, Ultra RAM's structure is rigid and non-configurable. To create memory structures that don't match this aspect ratio or exceed its capacity, designers must combine multiple Ultra Ram blocks. Ultra Ram operates using a single clock signal for both of its access ports, Port A and Port B. This simplifies the clocking architecture compared to BRAM, which can function as a true dual-clock memory. However, this design choice limits Ultra Ram's use in applications requiring asynchronous access across different clock domains.

Ultra Ram blocks cannot be initialized with user-defined values through the FPGA configuration bitstream. Upon power-up or a global device reset, the entire memory array is cleared to all zeros. This contrasts with BRAM, which can be pre-loaded with specific data. Applications needing non-zero initial values in Ultra Ram must include logic to write these values after FPGA configuration. Although Ultra Ram is described as a dual-port Ram, its access model differs from a true dual-port memory. Understanding its behavior is key to using it effectively. Ultra Ram is built on a single-port memory cell array but achieves dual-port functionality by time-multiplexing access within a single clock cycle. Port A operations always execute before Port B operations. This sequential execution resembles an enhanced version of Simple Dual-Port (SDP) Ram functionality. The fixed, sequential execution order ensures that simultaneous accesses to the same memory address by both ports produce predictable results, eliminating race conditions common in traditional memory collisions. The term “address collision is not possible” refers to the absence of unpredictable outcomes, not the prevention of simultaneous access. The hardware follows these deterministic rules:

- **Write on Port A, Read on Port B:** If Port A writes to an address and Port B reads from the same address in the same clock cycle, Port B retrieves the new data written by Port A.
- **Read on Port A, Write on Port B:** If Port A reads from an address and Port B writes to the same address in the same clock cycle, Port A retrieves the old data that existed before Port B’s write operation.
- **Write on Port A, Write on Port B:** If both ports write to the same address in the same clock cycle, Port B’s data is stored, overwriting Port A’s data by the end of the cycle.

This predictable, prioritized access scheme offers significant advantages. For example, a designer can build a data pipeline where one stage writes a value via Port A, and the next

stage reads the updated value via Port B within the same clock cycle. This can simplify control logic and reduce pipeline depth, enhancing design efficiency.

FPGA Hardware Design Concepts and VHDL

Shifting from software programming to hardware design is a paradigm shift. Software involves writing sequential instructions for a CPU to execute on fixed hardware. In contrast, designing for a Field-Programmable Gate Array (FPGA) means defining the hardware sequence itself. Traditional software development ignores concurrency (multiple operations happening simultaneously) and timing in the physical sense, VHDL explicitly models these physical aspects. In a circuit, all logic gates operate simultaneously. VHDL reflects this by assuming all components are active concurrently, mimicking their physical behavior.

VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language), developed in the 1980s, is a verbose, case-insensitive, IEEE-standardized language [22]. VHDL code isn't executed line-by-line; synthesis tools translate it into a netlist (a blueprint of circuit components like gates and wires and their connections) of gates, memory, and wires to configure the FPGA's programmable fabric into a custom circuit.

- **Entity and Architecture:** An entity defines a component's interface (ports); the architecture describes its behavior. Multiple architectures allow varied implementations.
- **Signals vs. Variables:** Signals represent wires with delayed assignments; variables, used in processes, update immediately but don't persist.
- **std_logic Type System:** The IEEE 1164 `std_logic` type supports values like '0', '1', 'Z' (high impedance), and 'X' (unknown), with `std_logic_vector` for buses.

VHDL offers three styles, often mixed in complex designs:

- **Dataflow:** Uses concurrent assignments (e.g., `Y <= A and B;`) for combinational logic like multiplexers.

- **Behavioral:** Employs sequential code in processes (e.g., `if-then-else`) for sequential logic like counters.
- **Structural:** Builds circuits from interconnected components, ideal for modular designs.

Table 1: VHDL Modeling Styles

Style	Description	Constructs	Best For
Dataflow	Concurrent flow	<code><=</code> , <code>when-else</code>	Combinational logic
Behavioral	Sequential algorithms	<code>process</code> , <code>if-then-else</code>	Sequential logic
Structural	Hierarchical components	<code>component</code> , <code>port map</code>	Modular systems

VHDL Primitives

Digital systems combine combinational (output from current inputs) and sequential (output from inputs and state) logic [22]. When synthesized for Xilinx UltraScale+ FPGAs, these designs map to specific hardware primitives including LUTs (Look-Up Tables) for combinational logic, flip-flops such as FDRE (D-type Flip-Flop with clock Enable and synchronous Reset) and FDCE (D-type Flip-Flop with clock Enable and asynchronous Clear) for sequential logic, CARRY4 (dedicated fast carry logic) for arithmetic operations, and DSP48E2 slices for multiplication and multiply-accumulate operations.

Toolchain and Methodology: The schematics, timing waveforms, and resource usage estimates presented in this section were generated using an open-source toolchain. The VHDL source code is first analyzed and synthesized using GHDL (a VHDL compiler and simulator), which generates a structural netlist in Verilog format. This netlist is then processed by Yosys (an open-source synthesis framework) to optimize and convert it to a JSON representation. The JSON netlist is visualized as a schematic using netlistsvg, which

renders the circuit topology with standard gate symbols. For timing waveforms, GHDL simulates the corresponding testbench, generating waveform data in FST (Fast Signal Trace) or VCD (Value Change Dump) format. This data is converted to WaveDrom JSON format using `vcd2wavedrom`, and finally rendered as timing diagrams using `wavedrom-cli`. Resource usage estimates (LUTs, flip-flops, DSP slices) are extracted from Yosys synthesis reports, providing insight into the hardware footprint of each primitive on Xilinx UltraScale+ FPGAs. This automated pipeline ensures consistency and reproducibility across all presented designs.

Basic Combinational Logic Gates Logic gates form the foundation of digital circuits and are described using dataflow modeling [22]. The basic logic gates include AND, OR, NOT, NAND, NOR, XOR, and XNOR. These gates implement Boolean logic functions using VHDL operators such as `and`, `or`, `not`, `nand`, `nor`, `xor`, and `xnor`. Listing 1 shows a 2-input AND gate example:

and-gate.vhd

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  entity AND_GATE is
4      port( A, B : in std_logic;
5            Y : out std_logic);
6  end AND_GATE;
7  architecture DATAFLOW of AND_GATE is
8  begin
9      Y <= A and B;
10 end DATAFLOW;
```

Code Listing 1: VHDL AND gate (dataflow)

Figure 24 shows both the synthesized schematic and timing behavior for the AND gate.

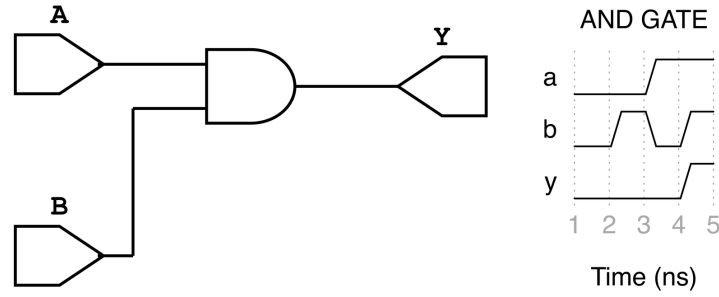


Figure 24: Synthesized schematic (left) and timing waveform (right) for AND gate.

The AND gate outputs '1' only when both inputs are '1', otherwise it outputs '0'. As shown in Figure 24, the output Y is high only when both A and B are simultaneously high. The AND gate requires 1 LUT and 1 Logic Element for Xilinx UltraScale+ FPGAs.

Similarly, other basic gates follow the same dataflow pattern. The OR gate uses the `or` operator, shown in Figure 25.

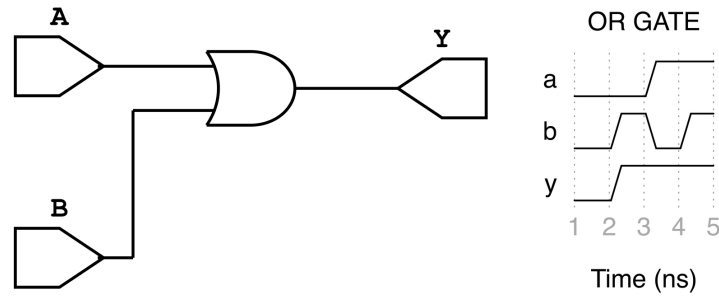


Figure 25: Synthesized schematic (left) and timing waveform (right) for OR gate.

The OR gate outputs '1' when at least one input is '1', and outputs '0' only when both inputs are '0'. Figure 25 demonstrates that the output Y is high whenever either A or B (or both) is high. The OR gate requires 1 LUT and 1 Logic Element for Xilinx UltraScale+ FPGAs.

The XOR gate implements exclusive-OR logic, shown in Figure 26.

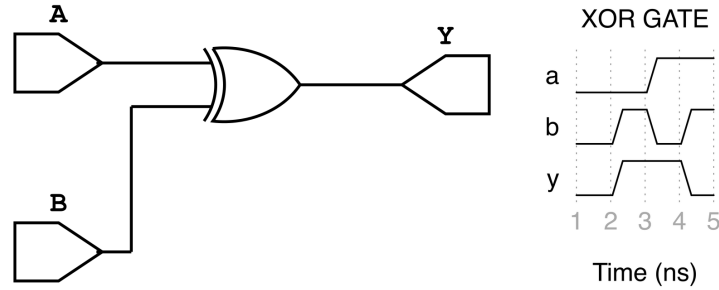


Figure 26: Synthesized schematic (left) and timing waveform (right) for XOR gate.

The XOR gate outputs '1' when the inputs differ (one is '1' and the other is '0'), and outputs '0' when both inputs are the same. Figure 26 shows that the output Y is high only when A and B have different values. The XOR gate requires 1 LUT and 1 Logic Element for Xilinx UltraScale+ FPGAs.

The NOT gate inverts its input, shown in Figure 27.

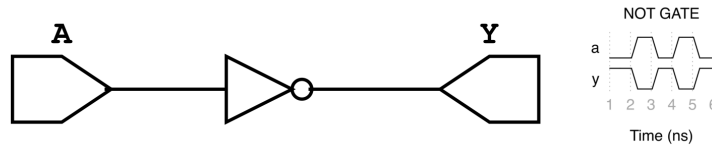


Figure 27: Synthesized schematic (left) and timing waveform (right) for NOT gate.

The NOT gate inverts its input, outputting '1' when the input is '0' and vice versa. Figure 27 illustrates that the output Y is always the opposite of input A. The NOT gate requires 0 LUTs (uses INV primitive) and 0 Logic Elements for Xilinx UltraScale+ FPGAs.

The NAND gate combines AND with inversion, shown in Figure 28.

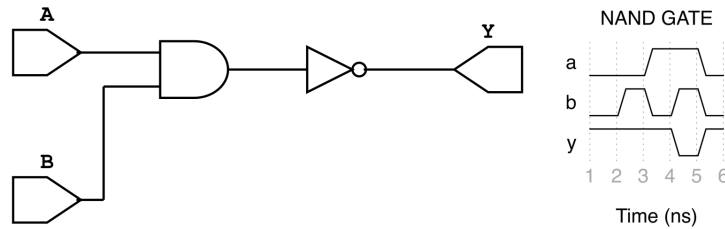


Figure 28: Synthesized schematic (left) and timing waveform (right) for NAND gate.

The NAND gate is the inverse of AND, outputting '0' only when both inputs are '1', and '1' otherwise. Figure 28 shows that the output Y is low only when both A and B are simultaneously high. The NAND gate requires 1 LUT and 1 Logic Element for Xilinx UltraScale+ FPGAs.

The NOR gate combines OR with inversion, shown in Figure 29.

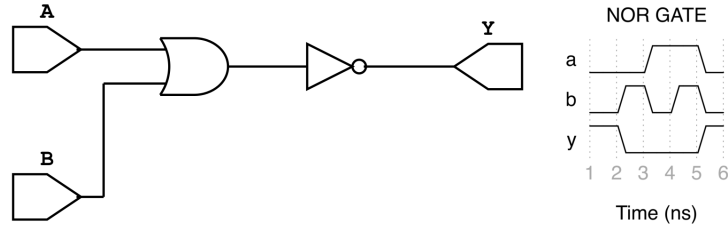


Figure 29: Synthesized schematic (left) and timing waveform (right) for NOR gate.

The NOR gate is the inverse of OR, outputting '1' only when both inputs are '0', and '0' otherwise. Figure 29 demonstrates that the output Y is high only when both A and B are low. The NOR gate requires 1 LUT and 1 Logic Element for Xilinx UltraScale+ FPGAs.

The XNOR gate implements exclusive-NOR logic, shown in Figure 30.

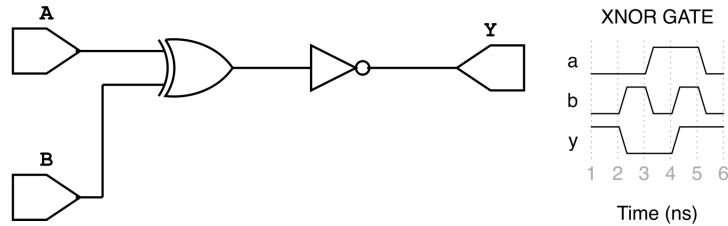


Figure 30: Synthesized schematic (left) and timing waveform (right) for XNOR gate.

The XNOR gate is the inverse of XOR, outputting '1' when both inputs are the same (both '0' or both '1'), and '0' when the inputs differ. Figure 30 shows that the output Y is high only when A and B have matching values. The XNOR gate requires 1 LUT and 1 Logic Element for Xilinx UltraScale+ FPGAs.

D Flip-Flop The D-type flip-flop (DFF) stores a bit, capturing input (D) on a clock edge (CLK) [22], as shown in Listing 2:

d-flip-flop.vhd

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  entity D_FLIP_FLOP is
4      port( CLK : in std_logic;
5            D  : in std_logic;
6            Q  : out std_logic);
7  end D_FLIP_FLOP;
8  architecture BEHAVIORAL of D_FLIP_FLOP is
9  begin
10     process(CLK)
11     begin
12         if rising_edge(CLK) then
13             Q <= D;
14         end if;
15     end process;
16 end BEHAVIORAL;

```

Code Listing 2: VHDL D flip-flop (behavioral)

The `rising_edge(CLK)` ensures synthesis infers a flip-flop. Figure 31 shows the timing behavior of the D flip-flop.

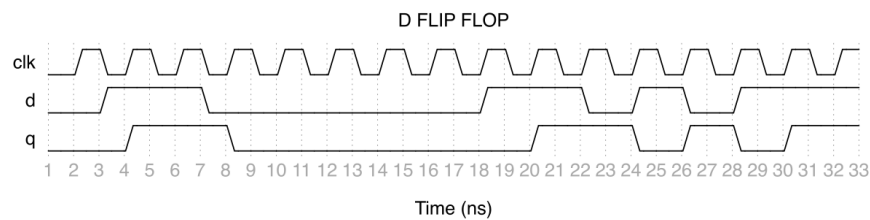


Figure 31: Timing waveform for D flip-flop demonstrating edge-triggered behavior.

The D flip-flop requires 1 FDRE flip-flop, 0 LUTs, and 0 DSP slices for Xilinx UltraScale+ FPGAs.

Figure 32 shows the gate-level implementation of the D flip-flop, revealing the internal structure using NAND gates to form an SR latch (Set-Reset latch, a basic memory element that can

store one bit by cross-coupling two gates). This lower-level view demonstrates how the abstract flip-flop primitive is constructed from basic logic gates. Figure 33 shows the same flip-flop synthesized as a single primitive symbol, which is the typical representation used in higher-level designs.

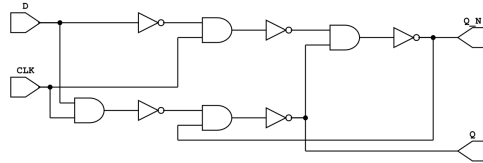


Figure 32: Gate-level schematic of D flip-flop showing internal NAND gate structure and SR latch implementation.

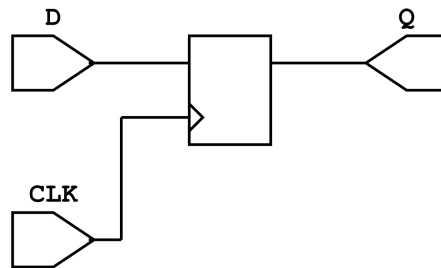


Figure 33: Behavioral schematic of D flip-flop showing the abstracted primitive symbol.

Resets can be implemented in two ways:

- **Asynchronous Reset:** Acts instantly, overriding clock, as demonstrated in Listing 3:

async-reset.vhd

```

1  process(CLK, RST)
2  begin
3      if RST = '1' then
4          Q <= '0';
5      elsif rising_edge(CLK) then
6          Q <= D;
7      end if;
8  end process;

```

Code Listing 3: VHDL asynchronous reset template

- **Synchronous Reset:** Acts on clock edge, as shown in Listing 4:

```

sync-reset.vhd

1  process(CLK)
2  begin
3      if rising_edge(CLK) then
4          if RST = '1' then
5              Q <= '0';
6          else
7              Q <= D;
8          end if;
9      end if;
10 end process;

```

Code Listing 4: VHDL synchronous reset template

Table 2 compares the key differences between synchronous and asynchronous resets.

Table 2: Synchronous vs. Asynchronous Resets

Feature	Synchronous Reset	Asynchronous Reset
VHDL Template	if rising_edge(CLK) then if RST='1'...	if RST='1'... elsif rising_edge(CLK)...
Hardware	MUX before D-input	Flip-flop clear pin
Latency	Clock edge	Immediate
Timing	Synchronous, safe	Risks metastability (unreliable output due to input changing too close to a clock edge)
Use Case	Internal logic	Global resets

Registered Logic Combinational logic gates can be combined with flip-flops to create registered (or pipelined) versions [22]. A registered gate includes a flip-flop at its output, storing the result on each clock edge. This approach offers several advantages: it breaks long combinational paths to improve maximum clock frequency, provides predictable timing by synchronizing outputs to the clock, and enables pipelined architectures where data flows through multiple stages. These

registered versions are needed in digital designs where timing closure and pipeline throughput are critical. The trade-off is increased latency (one clock cycle) and additional flip-flop resources, but the benefits in timing and design modularity often outweigh these costs.

Listing 5 shows a registered AND gate implementation:

and-gate-registered.vhd

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity AND_GATE_REGISTERED is
5      port( CLK : in std_logic;
6            A  : in std_logic;
7            B  : in std_logic;
8            Y  : out std_logic);
9  end AND_GATE_REGISTERED;
10
11 architecture BEHAVIORAL of AND_GATE_REGISTERED is
12     signal and_result : std_logic;
13 begin
14     -- Combinational AND logic
15     and_result <= A and B;
16
17     -- Register the output
18     process(CLK)
19     begin
20         if rising_edge(CLK) then
21             Y <= and_result;
22         end if;
23     end process;
24 end BEHAVIORAL;

```

Code Listing 5: VHDL registered AND gate

The registered AND gate combines the combinational AND operation with a flip-flop to store the result. Figure 34 shows the synthesized schematic with both the gate-level logic and the behavioral representation. Figure 35 demonstrates the timing behavior, showing how the output is

delayed by one clock cycle compared to a purely combinational gate.

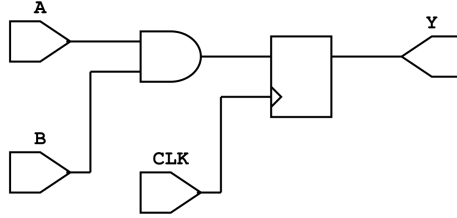


Figure 34: Synthesized schematic of registered AND gate showing combinational logic followed by flip-flop.

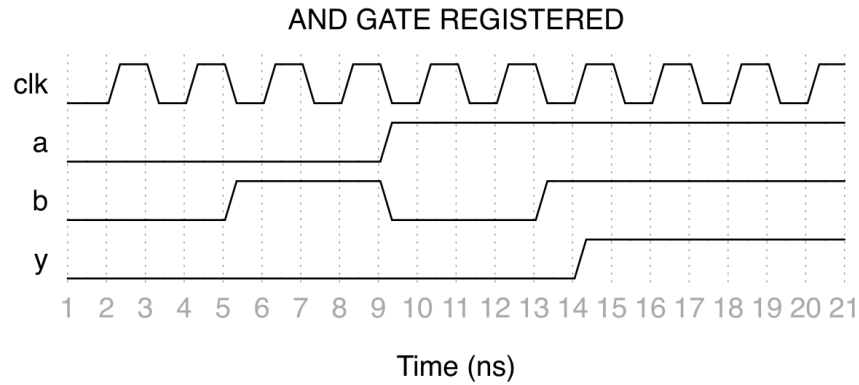


Figure 35: Timing waveform for registered AND gate demonstrating one clock cycle latency.

The registered AND gate requires 1 LUT, 1 FDRE flip-flop, and 1 Logic Element for Xilinx UltraScale+ FPGAs.

Similarly, all other basic gates can be registered. The registered OR gate is shown in Figure 36 with timing in Figure 37.

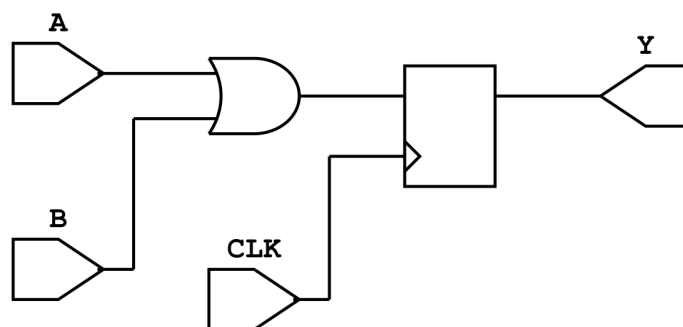


Figure 36: Synthesized schematic of registered OR gate.

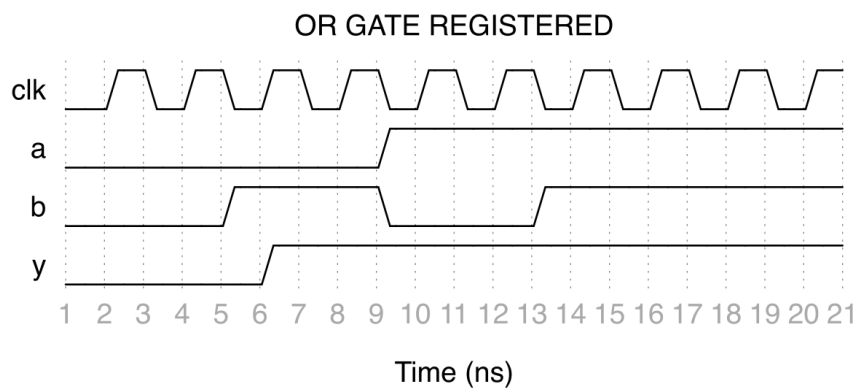


Figure 37: Timing waveform for registered OR gate.

The registered OR gate requires 1 LUT, 1 FDRE flip-flop, and 1 Logic Element for Xilinx UltraScale+ FPGAs.

The registered XOR gate is shown in Figure 38 with timing in Figure 39.

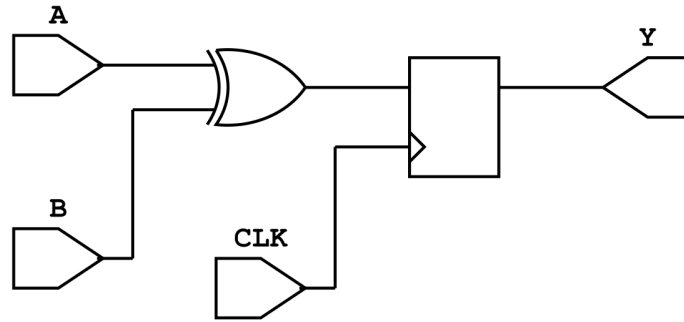


Figure 38: Synthesized schematic of registered XOR gate.

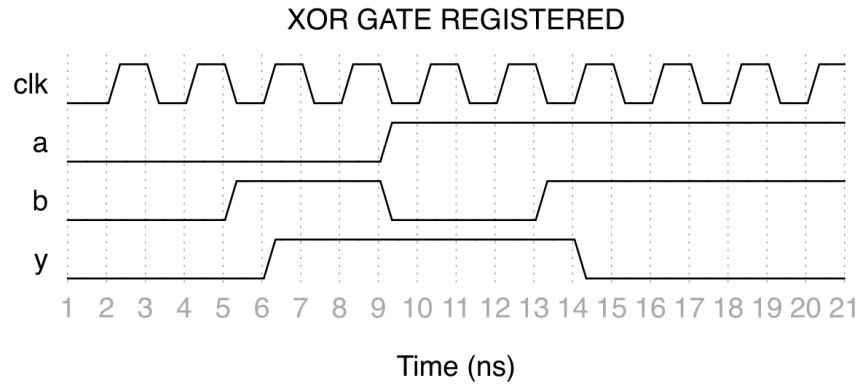


Figure 39: Timing waveform for registered XOR gate.

The registered XOR gate requires 1 LUT, 1 FDRE flip-flop, and 1 Logic Element for Xilinx UltraScale+ FPGAs.

The registered NOT gate is shown in Figure 40 with timing in Figure 41.

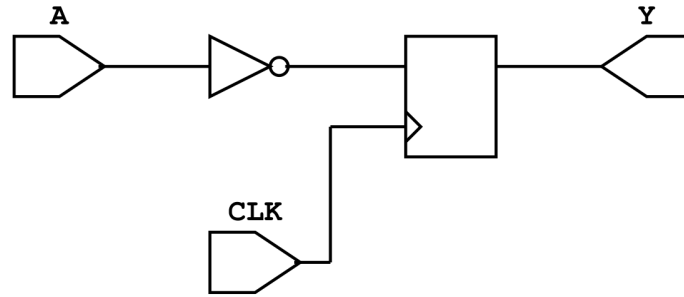


Figure 40: Synthesized schematic of registered NOT gate.

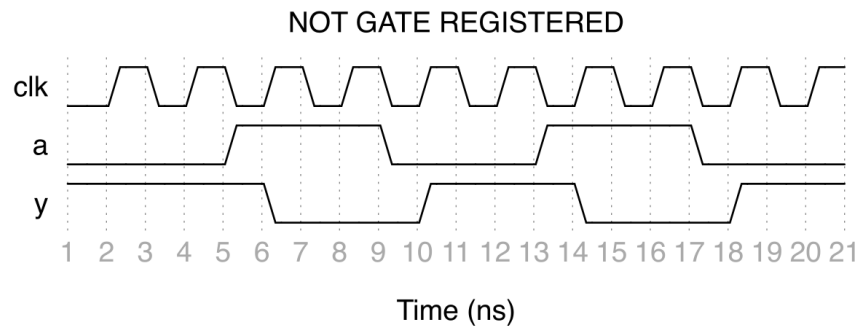


Figure 41: Timing waveform for registered NOT gate.

The registered NOT gate requires 0 LUTs, 1 FDRE flip-flop, and 0 Logic Elements for Xilinx UltraScale+ FPGAs.

The registered NAND gate is shown in Figure 42 with timing in Figure 43.

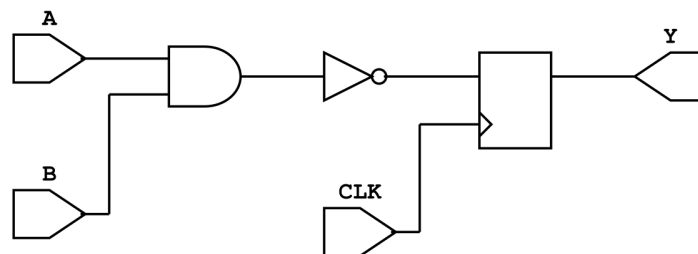


Figure 42: Synthesized schematic of registered NAND gate.

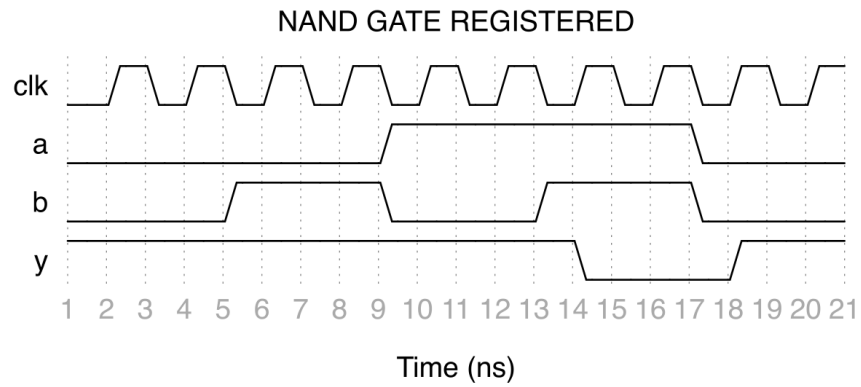


Figure 43: Timing waveform for registered NAND gate.

The registered NAND gate requires 1 LUT, 1 FDRE flip-flop, and 1 Logic Element for Xilinx UltraScale+ FPGAs.

The registered NOR gate is shown in Figure 44 with timing in Figure 45.

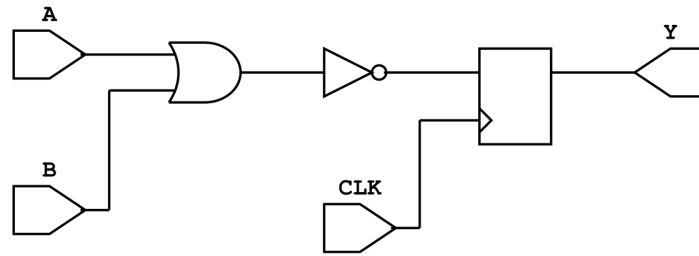


Figure 44: Synthesized schematic of registered NOR gate.

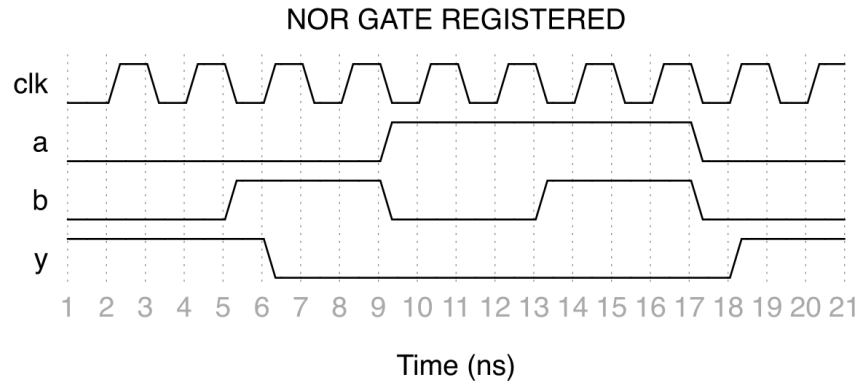


Figure 45: Timing waveform for registered NOR gate.

The registered NOR gate requires 1 LUT, 1 FDRE flip-flop, and 1 Logic Element for Xilinx UltraScale+ FPGAs.

The registered XNOR gate is shown in Figure 46 with timing in Figure 47.

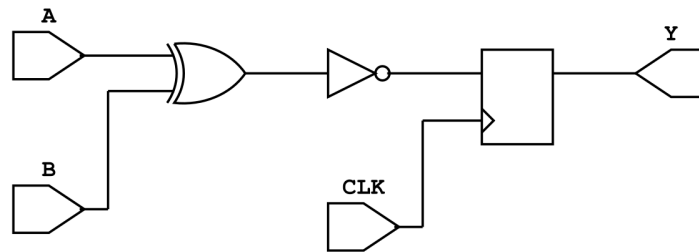


Figure 46: Synthesized schematic of registered XNOR gate.

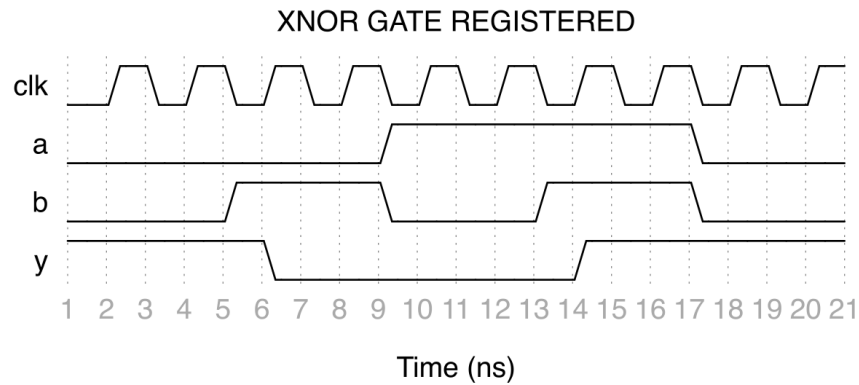


Figure 47: Timing waveform for registered XNOR gate.

The registered XNOR gate requires 1 LUT, 1 FDRE flip-flop, and 1 Logic Element for Xilinx UltraScale+ FPGAs.

Complex Combinational Circuits Arithmetic circuits showcase VHDL's modeling and synthesis-aware design [22]. Half and full adders are fundamental building blocks:

- **Half-Adder:** Adds two bits, producing SUM ($A \oplus B$) and CARRY ($A \cdot B$), as shown in Listing 6:

```
half-adder.vhd
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  entity HALF_ADDER is
4      port( A, B : in std_logic;
5            SUM : out std_logic;
6            CARRY: out std_logic);
7  end HALF_ADDER;
8  architecture DATAFLOW of HALF_ADDER is
9  begin
10     SUM <= A xor B;
11     CARRY <= A and B;
```

Code Listing 6: VHDL half-adder (dataflow)

- **Full-Adder:** Adds three bits (A , B , Cin), producing SUM ($A \oplus B \oplus Cin$) and $CARRY$ ($(A \cdot B) + (A \cdot Cin) + (B \cdot Cin)$). Structurally, it can be composed of two half-adders and an OR gate, leveraging modularity.

Figure 48 shows both the synthesized schematic and timing behavior.

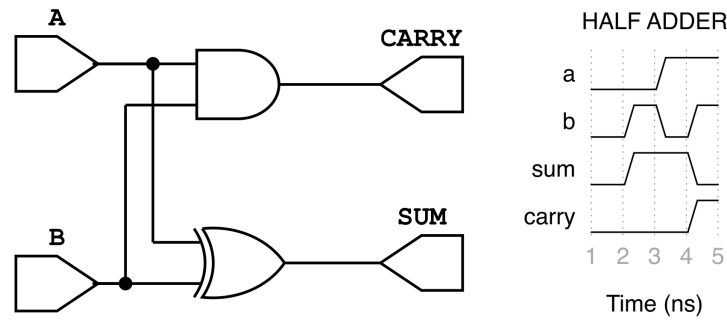


Figure 48: Synthesized schematic (left) and timing waveform (right) for half adder gate.

The half-adder requires 2 LUTs and 1 Logic Element for Xilinx UltraScale+ FPGAs.

Multi-bit adders can be implemented in several ways. A Ripple-Carry Adder (RCA) chains multiple full-adders together, where each full-adder adds one bit position. The carry output ($Cout$) from each full-adder connects to the carry input (Cin) of the next full-adder, creating a "ripple" effect as the carry propagates through all bit positions. For a 4-bit RCA, four full-adders are connected in sequence: bit 0's $Cout$ feeds bit 1's Cin , bit 1's $Cout$ feeds bit 2's Cin , and so on. This structure is demonstrated in Listing 7:

ripple-carry-adder-4bit.vhd

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  entity RIPPLE_CARRY_ADDER_4BIT is
4      port( A, B : in std_logic_vector(3 downto 0);
5            Cin : in std_logic;
6            SUM : out std_logic_vector(3 downto 0);
7            Cout : out std_logic);
```

```

8  end RIPPLE_CARRY_ADDER_4BIT;
9  architecture STRUCTURAL of RIPPLE_CARRY_ADDER_4BIT is
10     component FULL_ADDER is
11         port( A, B, Cin : in std_logic;
12              SUM, Cout : out std_logic);
13     end component;
14     signal C : std_logic_vector(4 downto 0);
15 begin
16     C(0) <= Cin;
17     FA0 : FULL_ADDER port map(A=>A(0), B=>B(0), Cin=>C(0), SUM=>SUM(0), Cout=>C(1)
18         );
19     FA1 : FULL_ADDER port map(A=>A(1), B=>B(1), Cin=>C(1), SUM=>SUM(1), Cout=>C(2)
20         );
21     FA2 : FULL_ADDER port map(A=>A(2), B=>B(2), Cin=>C(2), SUM=>SUM(2), Cout=>C(3)
22         );
23     FA3 : FULL_ADDER port map(A=>A(3), B=>B(3), Cin=>C(3), SUM=>SUM(3), Cout=>C(4)
24         );
25     Cout <= C(4);
26 end STRUCTURAL;

```

Code Listing 7: VHDL 4-bit ripple-carry adder (structural)

RCA's are simple to implement but slow because each bit must wait for the previous bit's carry to be computed, resulting in linear delay to the number of bits. Carry-Lookahead Adders (CLAs) improve performance by computing carries in parallel. Synthesis can infer CLAs using the `+` operator with `ieee.numeric_std`.

Figure 49 shows the synthesized schematic. Figure 50 shows the timing behavior.

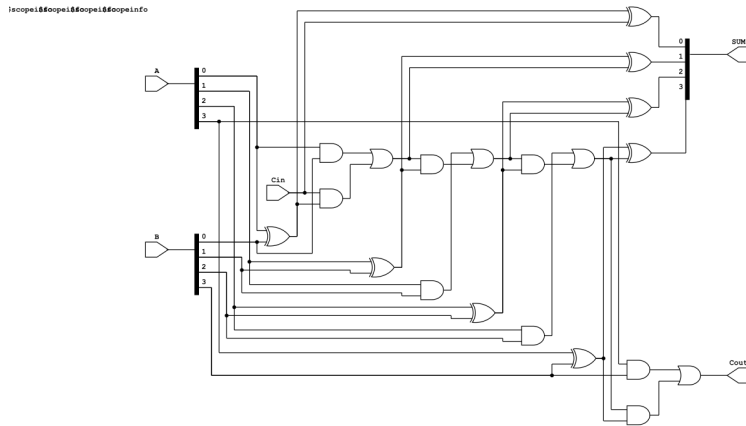


Figure 49: Synthesized schematic of 4-bit ripple-carry adder showing cascaded full-adder stages.

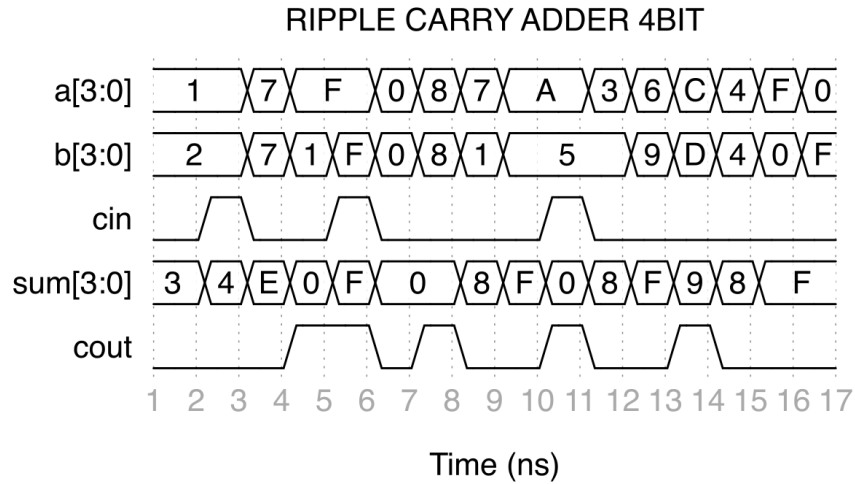


Figure 50: Timing waveform for 4-bit ripple-carry adder demonstrating multi-bit addition.

The 4-bit ripple-carry adder requires 8 LUTs for Xilinx UltraScale+ FPGAs.

Subtraction ($A - B$) uses $A + (\text{NOT } B + 1)$, implemented with XOR gates and a carry-in of 1. For a 4-bit subtractor, each bit's B input is XORed with a subtract signal (SUB=1 for subtraction), inverting B, and Cin=1 adds the +1. This reuses the RCA structure, adding 4 LUTs for XOR gates. The result is a unified adder/subtractor, switching modes via SUB.

Binary multiplication can be implemented using sequential shift-and-add algorithms or by

using DSP slices. The shift-and-add method checks each multiplier bit, adding the shifted multiplicand if the bit is '1', requiring N cycles for an N -bit multiplier. For a 16-bit multiplier, this takes 16 cycles. The control logic, implemented as a finite state machine, uses 2-3 LUTs and 1-2 flip-flops. Alternatively, Listing 8 shows how using `ieee.numeric_std.all` maps multiplication to DSP slices:

signed-multiplier.vhd

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4  entity SIGNED_MULTIPLIER is
5      port( A, B : in std_logic_vector(15 downto 0);
6            PRODUCT : out std_logic_vector(31 downto 0));
7  end SIGNED_MULTIPLIER;
8  architecture BEHAVIORAL of SIGNED_MULTIPLIER is
9  begin
10     PRODUCT <= std_logic_vector(signed(A) * signed(B));
11 end BEHAVIORAL;
```

Code Listing 8: VHDL signed multiplier using numeric_std

This infers a DSP48E2 slice, performing a 16x16-bit signed multiplication in one cycle, using one DSP slice and no LUTs. DSP slices support up to 27x18-bit multiplications, with a 48-bit accumulator for multiply-accumulate operations, critical for signal processing.

Figure 51 shows the synthesized schematic. Figure 52 shows the timing behavior.

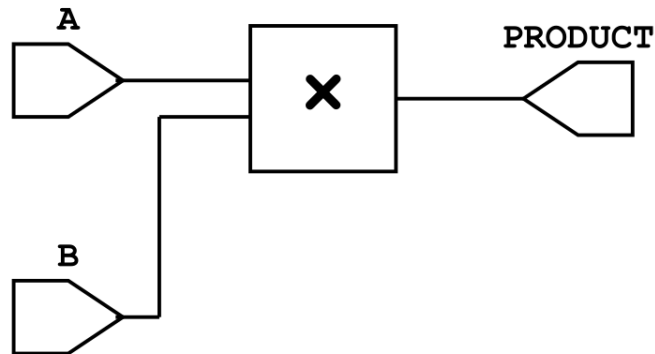


Figure 51: Synthesized schematic of 16-bit signed multiplier utilizing DSP48E1 slice.



Figure 52: Timing waveform for 16-bit signed multiplier demonstrating multiplication operation.

The 16-bit signed multiplier requires 0 LUTs, 0 Logic Elements, and 1 DSP slice for Xilinx UltraScale+ FPGAs.

Complex Sequential Circuits Sequential logic circuits build upon the basic flip-flop primitive to create more complex state machines and counters [22]. A 4-bit binary counter increments its output value on each clock cycle, as shown in Listing 9:

counter-4bit.vhd

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;

```

```

4
5  entity COUNTER_4BIT is
6      port( CLK    : in std_logic;
7            RST    : in std_logic;
8            EN     : in std_logic;
9            COUNT  : out std_logic_vector(3 downto 0));
10 end COUNTER_4BIT;
11
12 architecture BEHAVIORAL of COUNTER_4BIT is
13     signal count_int : unsigned(3 downto 0) := (others => '0');
14 begin
15     process(CLK, RST)
16     begin
17         if RST = '1' then
18             count_int <= (others => '0');
19         elsif rising_edge(CLK) then
20             if EN = '1' then
21                 count_int <= count_int + 1;
22             end if;
23         end if;
24     end process;

```

Code Listing 9: VHDL 4-bit binary counter

Figure 53 shows the synthesized schematic. Figure 54 shows the timing behavior.

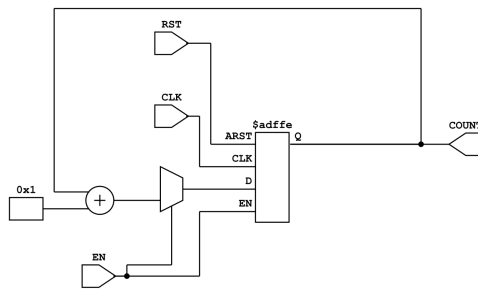


Figure 53: Synthesized schematic of 4-bit binary counter with enable and reset.

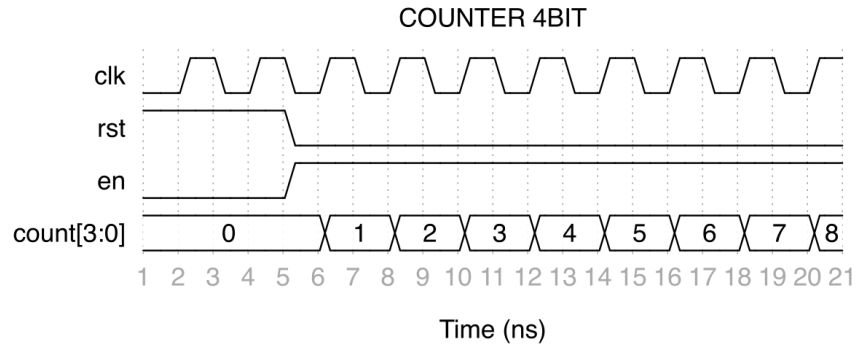


Figure 54: Timing waveform for 4-bit counter demonstrating sequential counting behavior.

The 4-bit counter requires 4 FDCE flip-flops, 1 CARRY4 carry chain, 0 LUTs, and 0 DSP slices for Xilinx UltraScale+ FPGAs.

Shift registers are another essential sequential circuit, used for serial-to-parallel conversion, data buffering, and delay lines. A 4-bit shift register shifts data through a chain of flip-flops on each clock cycle. Figure 55 shows the synthesized schematic. Figure 56 shows the timing behavior.

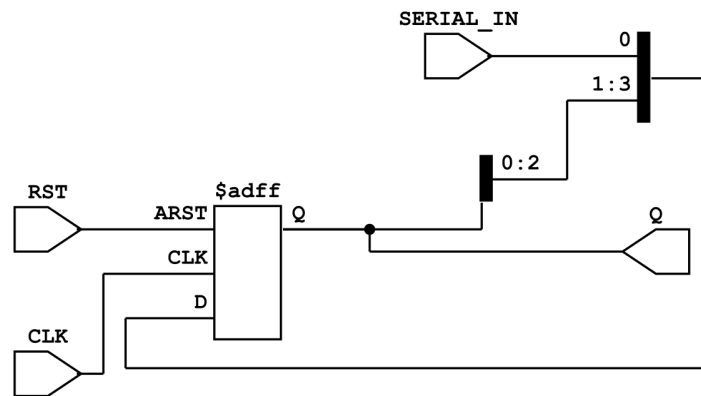


Figure 55: Synthesized schematic of 4-bit shift register showing cascaded flip-flop chain.

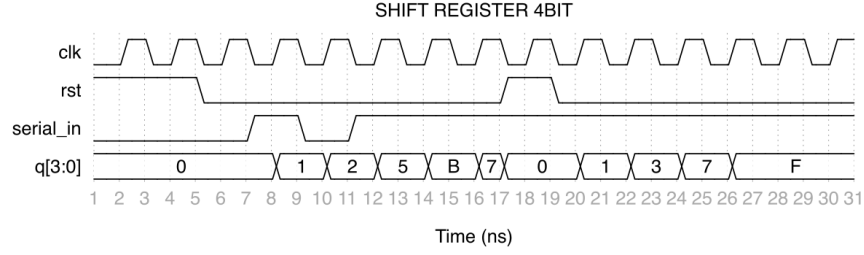


Figure 56: Timing waveform for 4-bit shift register demonstrating data propagation through stages.

The 4-bit shift register requires 4 FDCE flip-flops, 0 LUTs, and 0 DSP slices for Xilinx UltraScale+ FPGAs.

Table 3 summarizes the FPGA resource utilization for the key digital circuits.

Table 3: FPGA Resource Utilization Summary for Key Digital Circuits (Xilinx UltraScale+)

Circuit	LUTs	Flip-Flops	DSP Slices	Carry Logic	Est. LCs
AND Gate	1 (LUT2)	0	0	0	1
Half-Adder	2 (LUT2)	0	0	0	1
4-bit RCA	8 (LUT3)	0	0	0	—
D Flip-Flop	0	1 (FDRE)	0	0	0
4-bit Counter	0	4 (FDCE)	0	1 (CARRY4)	0
4-bit Shift Register	0	4 (FDCE)	0	0	0
16-bit Signed Multiplier	0	0	1 (DSP48E2)	0	0

FIFOs are essential for buffering and clock domain crossing (transferring data safely between circuit parts using different clock speeds) [22]. FIFOs support:

- **Rate Matching:** Buffering between different rates, e.g., a high-speed ADC producing data faster than a processor, using a 512-word FIFO to store bursts.
- **Buffering:** Smoothing data flow, e.g., handling 4K video frames for a processing pipeline with variable latency.
- **Clock Domain Crossing:** Safe data transfer across clocks, e.g., from a sensor to a network interface, preventing data corruption.

A synchronous FIFO operates within a single clock domain, as shown in Listing 10:

fifo-sync.vhd

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4
5  -- Synchronous FIFO with single clock domain (2 words deep, 2-bit wide)
6  entity FIFO_SYNC is
7      port(
8          CLK : in std_logic;
9          RST : in std_logic;
10         WR_EN : in std_logic;
11         RD_EN : in std_logic;
12         DIN : in std_logic_vector(1 downto 0);
13         DOUT : out std_logic_vector(1 downto 0);
14         FULL : out std_logic;
15         EMPTY : out std_logic
16     );
17 end FIFO_SYNC;
18
19 architecture BEHAVIORAL of FIFO_SYNC is
20     type memory_type is array (0 to 1) of std_logic_vector(1 downto 0);
21     signal memory : memory_type := (others => (others => '0'));
22     signal wr_ptr : unsigned(0 downto 0) := (others => '0');
23     signal rd_ptr : unsigned(0 downto 0) := (others => '0');
24     signal count : unsigned(1 downto 0) := (others => '0');
25 begin
26     process(CLK)
27     begin
28         if rising_edge(CLK) then
29             if RST = '1' then
30                 wr_ptr <= (others => '0');
31                 rd_ptr <= (others => '0');
32                 count <= (others => '0');
33             else

```

```

34          -- Write operation
35          if WR_EN = '1' and count < 2 then
36              memory(to_integer(wr_ptr)) <= DIN;
37              wr_ptr <= wr_ptr + 1;
38          end if;
39
40          -- Read operation
41          if RD_EN = '1' and count > 0 then
42              rd_ptr <= rd_ptr + 1;
43          end if;
44
45          -- Update count
46          if WR_EN = '1' and RD_EN = '0' and count < 2 then
47              count <= count + 1;
48          elsif WR_EN = '0' and RD_EN = '1' and count > 0 then
49              count <= count - 1;
50          end if;
51      end if;
52  end if;
53  end process;
54
55  DOUT <= memory(to_integer(rd_ptr));
56  FULL <= '1' when count = 2 else '0';
57  EMPTY <= '1' when count = 0 else '0';
58
59  end BEHAVIORAL;

```

Code Listing 10: VHDL synchronous FIFO

Synchronous FIFOs include memory, write/read pointers, and status flags. Empty when pointers are equal; full uses counters or extended pointers. Figure 57 shows the synthesized schematic of a synchronous FIFO.

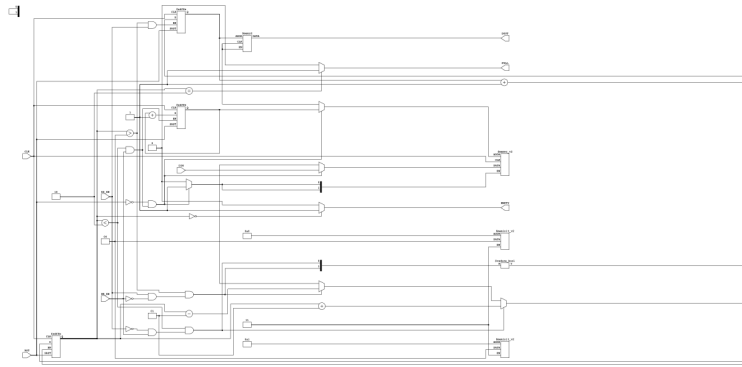


Figure 57: Synthesized schematic of synchronous FIFO showing memory array, pointers, and control logic.

An asynchronous FIFO handles dual clock domains using Gray code for safe pointer synchronization, as shown in Listing 11:

`fifo-async.vhd`

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4
5  -- Asynchronous FIFO with dual clock domains using Gray code (2 words deep, 2-bit
   wide)
6  entity FIFO_ASYNC is
7      port(
8          WR_CLK : in std_logic;
9          RD_CLK : in std_logic;
10         RST : in std_logic;
11         WR_EN : in std_logic;
12         RD_EN : in std_logic;
13         DIN : in std_logic_vector(1 downto 0);
14         DOUT : out std_logic_vector(1 downto 0);
15         FULL : out std_logic;
16         EMPTY : out std_logic
17     );
18 end FIFO_ASYNC;
```

```

19
20 architecture BEHAVIORAL of FIFO_ASYNC is
21     type memory_type is array (0 to 1) of std_logic_vector(1 downto 0);
22     signal memory : memory_type := (others => (others => '0'));
23
24     signal wr_ptr_bin : unsigned(1 downto 0) := (others => '0');
25     signal rd_ptr_bin : unsigned(1 downto 0) := (others => '0');
26     signal wr_ptr_gray : unsigned(1 downto 0) := (others => '0');
27     signal rd_ptr_gray : unsigned(1 downto 0) := (others => '0');
28
29     signal wr_ptr_gray_sync1 : unsigned(1 downto 0) := (others => '0');
30     signal wr_ptr_gray_sync2 : unsigned(1 downto 0) := (others => '0');
31     signal rd_ptr_gray_sync1 : unsigned(1 downto 0) := (others => '0');
32     signal rd_ptr_gray_sync2 : unsigned(1 downto 0) := (others => '0');
33
34     function bin_to_gray(bin : unsigned) return unsigned is
35         variable gray : unsigned(bin'range);
36     begin
37         gray := bin xor shift_right(bin, 1);
38         return gray;
39     end function;
40
41 begin
42     -- Write clock domain
43     process(WR_CLK)
44     begin
45         if rising_edge(WR_CLK) then
46             if RST = '1' then
47                 wr_ptr_bin <= (others => '0');
48                 wr_ptr_gray <= (others => '0');
49             else
50                 if WR_EN = '1' and FULL = '0' then
51                     memory(to_integer(wr_ptr_bin(0 downto 0))) <= DIN;
52                     wr_ptr_bin <= wr_ptr_bin + 1;
53                     wr_ptr_gray <= bin_to_gray(wr_ptr_bin + 1);
54                 end if;
55             end if;
56

```

```

57      -- Synchronize read pointer to write clock domain
58      rd_ptr_gray_sync1 <= rd_ptr_gray;
59      rd_ptr_gray_sync2 <= rd_ptr_gray_sync1;
60  end if;
61 end process;
62
63 -- Read clock domain
64 process(RD_CLK)
65 begin
66     if rising_edge(RD_CLK) then
67         if RST = '1' then
68             rd_ptr_bin <= (others => '0');
69             rd_ptr_gray <= (others => '0');
70         else
71             if RD_EN = '1' and EMPTY = '0' then
72                 rd_ptr_bin <= rd_ptr_bin + 1;
73                 rd_ptr_gray <= bin_to_gray(rd_ptr_bin + 1);
74             end if;
75         end if;
76
77         -- Synchronize write pointer to read clock domain
78         wr_ptr_gray_sync1 <= wr_ptr_gray;
79         wr_ptr_gray_sync2 <= wr_ptr_gray_sync1;
80     end if;
81 end process;
82
83 DOUT <= memory(to_integer(rd_ptr_bin(0 downto 0)));
84 FULL <= '1' when wr_ptr_gray = (rd_ptr_gray_sync2 xor "10") else '0';
85 EMPTY <= '1' when rd_ptr_gray = wr_ptr_gray_sync2 else '0';
86
87 end BEHAVIORAL;

```

Code Listing 11: VHDL asynchronous FIFO with Gray code

Asynchronous FIFOs use Gray codes (a binary system where only one bit changes between consecutive numbers) to safely synchronize pointers across clock domains. For a 2-word FIFO, 2-bit binary pointers are converted to Gray code, ensuring only one bit changes per increment.

Each pointer is synchronized to the other clock domain using a 2-stage flip-flop synchronizer. The synchronized Gray pointers are compared to determine full/empty status. Figure 58 shows the synthesized schematic.

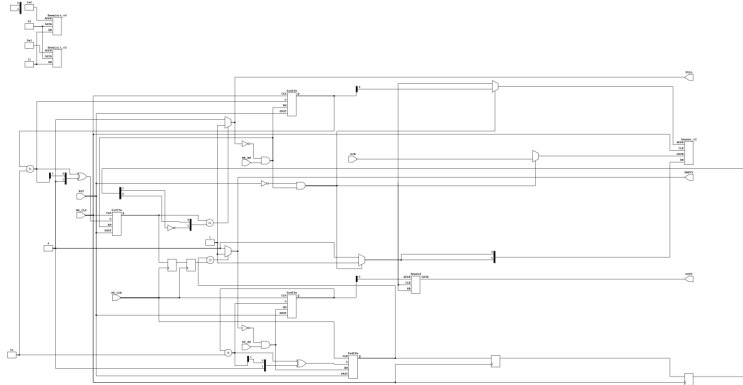


Figure 58: Synthesized schematic of asynchronous FIFO showing dual-clock architecture with Gray code synchronizers.

Operating System Options for SoC FPGAs

An operating system (OS) is a complex piece of software that manages hardware resources and provides an interface for other software to interact with the hardware and run on top of the OS. It is responsible for managing memory, process scheduling, device drivers, and more. In the context of integration, OSes abstract hardware complexities, enabling applications to run efficiently and securely. There are three main approaches to OS design:

- **Bare-Metal (No OS):** The term “bare-metal” refers to running software directly on hardware without an additional OS layer. This approach is typically used for systems designed to execute a single application or firmware, with no support for multitasking or resource sharing. It is the simplest and least complex setup, common in resource-constrained embedded devices like basic microcontrollers.
- **Real-Time Operating System (RTOS):** An RTOS is designed to meet specific real-time constraints, ensuring tasks complete within predefined deadlines through priority-based, preemptive scheduling. These systems prioritize determinism and low latency. They are divided into two categories: hard real-time (where missing a deadline causes system failure, e.g., in avionics) and soft real-time (where occasional misses are tolerable, e.g., in multimedia streaming). Commonly used RTOS examples include FreeRTOS, VxWorks, and Zephyr.
- **General-Purpose Operating System (GPOS):** A GPOS is built for versatility, handling a wide variety of tasks without strict timing guarantees. It emphasizes overall throughput, fairness in resource allocation, and support for multiple users and applications (e.g., running a web server alongside a video editing program). Examples include Linux, Windows, and macOS.

For this project, we will be using a minimal GPOS, specifically a GNU/Linux-based OS. We are choosing this over a traditional RTOS due to the need for a flexible and scalable development environment that can integrate with existing or future software and hardware. With an RTOS,

significant changes often require recompiling the OS image, which is not ideal for a development environment supporting a variety of applications and hardware. The GNU/Linux-based OS will provide greater flexibility. For our flavor of GNU/Linux, we will utilize the Yocto Project build system to create a custom GNU/Linux OS for the SoC FPGA. This allows for a tailored operating system that supports diverse applications and hardware designs.

Embedded Linux

The GNU/Linux operating system consists of two main components: the kernel and the userspace. It is referred to as GNU/Linux because Linux denotes just the kernel, while GNU represents the system software that provides libraries and tools to create a usable operating system [24]. The kernel handles hardware abstraction, process scheduling, memory allocation, I/O devices, interrupt handling, security mechanisms, and kernel modules.

- **Hardware Abstraction:** This layer manages communication with the physical hardware of the device. Essentially, the kernel provides a set of hardware-agnostic interfaces (via system calls) to software, allowing applications to interact with hardware without knowing its specifics. When an application uses these interfaces, the kernel translates the request, communicates with the hardware, and returns the result. This enables portability across different hardware platforms.
- **Process Scheduling:** This layer manages the assignment of processes to the CPU based on priorities and available resources. It also handles context switching to enable efficient multitasking and system responsiveness.
- **Memory Allocation:** This layer oversees memory distribution to processes based on their needs and available RAM. It manages virtual memory, including paging to disk for efficient RAM usage and protection against unauthorized access.
- **I/O Devices:** This layer facilitates communication with input/output devices, such as keyboards, mice, or webcams, ensuring applications can use them effectively.

- **Interrupt Handling:** This layer processes signals from hardware (e.g., keyboard presses or timer events) via the interrupt controller, allowing timely responses to events.
- **Security Mechanisms:** This layer enforces access controls, user permissions, and features like SELinux to protect against threats and support multi-user environments.
- **Kernel Modules:** This layer supports dynamic loading and unloading of modules to extend kernel functionality, such as adding drivers for new hardware without rebooting.

The userspace is the layer of the OS where users interact with applications and system software. Userspace communicates with the kernel via system calls to retrieve or send requests to the hardware. It runs programs in an isolated environment, preventing direct hardware access and minimizing the risk of system crashes from program errors. Programs in userspace operate within virtual memory spaces managed by the kernel, enabling multiple applications to run simultaneously. When a program requests or allocates memory, it uses this virtual space, with the kernel handling the physical mapping. Key userspace components include shells (e.g., Bash), utilities (e.g., from GNU coreutils like `ls` and `grep`), and libraries (e.g., glibc for system call wrappers).

Yocto

The Yocto Project is an open-source collaboration project that provides tools and templates to enable developers to create custom Linux-based systems for devices, mainly targeted at embedded devices. It is not a pre-built Linux distribution that runs directly on hardware but rather a toolkit for building tailored, reproducible Linux images optimized for specific architectures and applications. This makes it particularly suitable for projects like ours, where we are using an SoC FPGA and might need to add hardware-specific drivers and configurations to the OS[45].

Yocto is composed of three key components that work together to facilitate the build process:

- **Poky (pronounced “Pock-ee”):** This is a reference distribution that the Yocto Project provides as a starting point for creating custom distributions. It is a complete example of a Yocto project so that developers can see how to create a custom distribution. It offers default configurations that can be extended or modified[45].

- **OpenEmbedded:** This is the core build framework of the Yocto Project, often referred to as OpenEmbedded-Core (OE-Core). It manages metadata, including recipes for software packages, configuration files, and dependencies. OpenEmbedded handles the logic for fetching source code, applying patches, cross-compiling for target architectures, and packaging components into a cohesive system image[45].
- **BitBake:** BitBake is responsible for parsing recipes and configuration files, scheduling tasks, and executing the build steps. It supports parallel processing, dependency resolution, and incremental builds, making it efficient for complex projects. BitBake interprets the metadata from OpenEmbedded layers to compile software, generate packages, and produce bootable images[45].

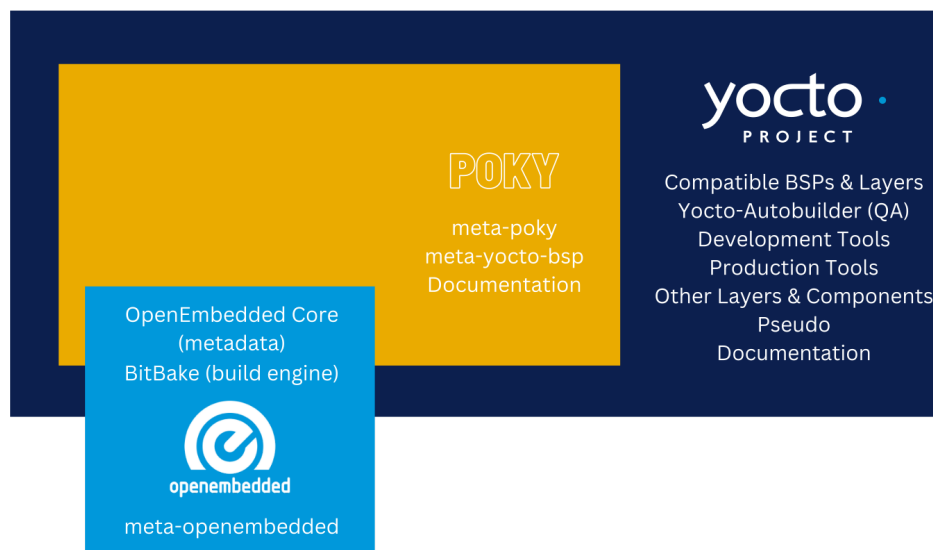


Figure 59: Yocto Project Technical Overview [46].

The Yocto Project employs a layered architecture to allow for modular customization. Essentially this allows you to stack metadata on top of metadata. This is useful for example if you want to build a specific fork of a software package, you don't have to make a brand new instruction set for Yocto to use, you can simply stack the metadata for the fork on top of the metadata for the original software package. This allows you to use the original software package's

instructions and have it be recognized by any software package that is dependent on it. Yocto allows this layering to happen in a few specific ways.

- **Recipes:** These are files (with a `.bb` and `.bbappend` extension) that define how to fetch, configure, build, and install individual software components, such as the Linux kernel, libraries, or applications[45].
- **Meta Layers:** Collections of related recipes and configurations, organized into directories. Layers allow modular customization for example, adding a hardware-specific BSP (Board Support Package) layer for the SoC FPGA[45].
- **Build Process:** Developers configure the build via files like `local.conf` which will specify the computer architecture, compilers, and other features. Running BitBake commands `bitbake core-image-minimal` will then automates the entire process, from source retrieval to image generation[45].

The key benefit of all this is that it allows for a scalable, maintainable OS that can simply be updated with future hardware and software requirements. The Yocto build system eliminates the need to manually compile the OS from source and allows for a more reproducible and consistent build process. This makes it easier to pull in patches, updates, and customizations to the final operating system without the need to setup scripts to do so. The key drawback of using Yocto is that it is complex and requires a lot of knowledge to use effectively. Yocto also requires a lot of system resources to build the OS image, as Yocto builds every tool that is needed to complete the build pipeline. For example if I want to build a software package compiled for the ARM architecture on a x86 machine, Yocto will first need to pull down the compiler for my x86 machine and compile all the tools that are needed for my machine even if I already have them installed on my machine. This is because Yocto needs to build the toolchain for the host and target architectures separately. Then Yocto will build the software package for the target architecture. This is a lot of overhead and can take a lot of time to build the OS image, however this means that reproducible builds are very easy to achieve and are very consistent across different machines and build environments[45].

Rationale for a Linux-Based SoC Platform

Real-time wildfire fuel classification necessitates both high-performance processing and a versatile platform for system integration. While the FPGA fabric provides the computational throughput required for processing hyperspectral datacubes, a hardware-only solution lacks the flexibility needed for a modern edge device.

To address these requirements, this work implements a hybrid architecture using the Zynq UltraScale+ MPSoC, pairing a custom FPGA accelerator with a custom Embedded Linux OS. The integration of the operating system is driven by the need to utilize standard drivers and services, facilitating capabilities that are complex to implement in a hardware-only environment:

- **Standard Peripherals:** The inclusion of Embedded Linux provides native support for the MPSoC’s hard-IP peripherals, enabling access to Gigabit Ethernet networking, USB connectivity, and PCIe interfaces for high-speed NVMe storage.
- **System Flexibility:** A standard POSIX environment facilitates future development by allowing high-level applications (in Python, C++, etc.) to interact with the hardware accelerator. This architecture supports the integration of fabric-based acceleration into broader software frameworks, offering the potential to offload computationally intensive tasks—such as machine learning inference—to the FPGA through mechanisms like Python-wrapped C drivers interfacing with the Linux kernel.

This establishes a pipeline where the FPGA handles the real-time data stream, while the Linux OS provides the necessary connectivity, storage infrastructure, and other necessary services.

HARDWARE: CUSTOM ACCELERATOR FOR CNN KERNEL OPERATIONS

This chapter describes the hardware implementation of a custom CNN accelerator for real-time hyperspectral image processing. The accelerator consists of three main subsystems, each with specialized subcomponents:

- **Memory Manager:** Manages read/write operations to Ultra RAM blocks and packs data into 72-bit words.
- **Data Slicer/Unpacker:** Retrieves packed words from memory and unpacks them into individual values for computation.
- **Convolution Engine:** Performs multiply-accumulate operations and organizes results into the output stream.

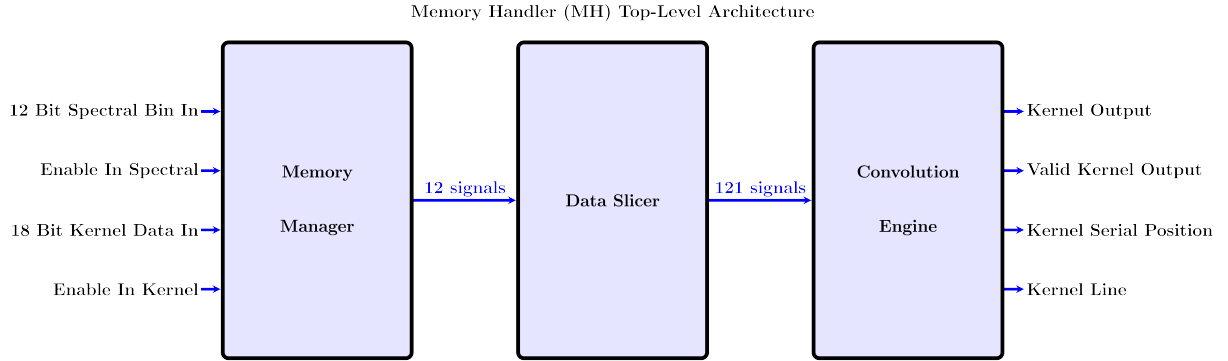


Figure 60: Top-level diagram of the memory handler.

The accelerator integrates within a larger processing pipeline shown in Figure 61. This chapter focuses on the "Convolution Processing" stage. Other pipeline stages include:

- **Receive Values From CMOS Sensor:** Captures 1100×1600 images at 180.6 frames per second from the hyperspectral sensor.
- **Apply Gain and Offset Values:** Normalizes sensor data to appropriate ranges.

- **Convert Upwelling Radiance to Reflectance:** Transforms raw measurements into reflectance values.
- **Bin Adjacent Pixels:** Reduces spectral dimensions by $4\times$, producing 1100×400 images for convolution.
- **Convolution Processing:** Delivers formatted data to the accelerator (primary focus of this chapter).
- **Results to Shared Memory:** Stores convolution results for SoC access.

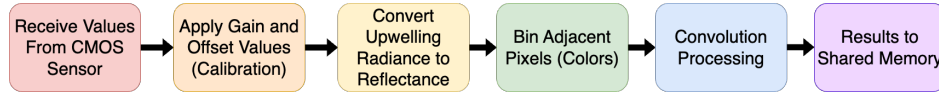


Figure 61: Proposed Processing Pipeline of the Smart Sensor.

The accelerator was designed using Vitis Model Composer 2024.1 for MATLAB Simulink 2023b [1, 29] rather than writing VHDL directly. This visual design approach allows easier handoff to future developers and maintains consistency with the upstream binning subsystem already implemented in Simulink. Simulink provides clear visualization of hardware interconnections and data flow while generating synthesizable VHDL for FPGA implementation. The only vendor-specific primitive is the Xilinx UltraRAM block for storing frame data and kernel weights. All source code for this hardware design, including the Simulink model, MATLAB configuration scripts, and generated VHDL, can be found in the project repository [10].

The Memory Handler (MH) module (Figure 62) shows the top-level integration of all accelerator subsystems. This module coordinates data flow from memory through unpacking to convolution output, connecting the Frame and Kernel Memory Controllers, Data Slicer, and Convolution Engine to manage the complete pipeline from data input to convolution result output.

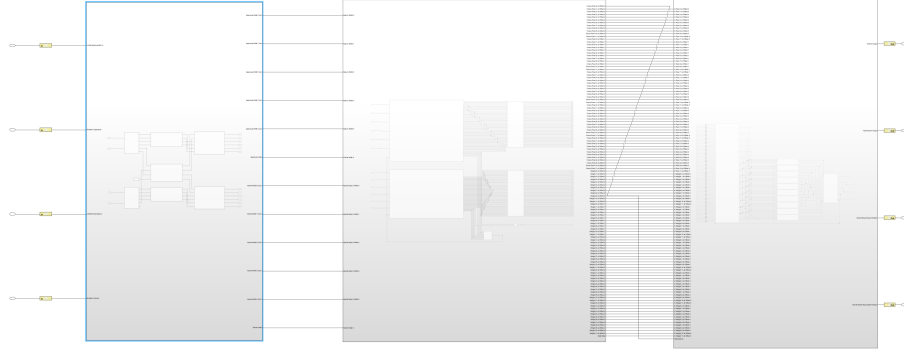


Figure 62: Memory Handler (MH) top-level integration of all accelerator subsystems.

Note: All MATLAB Simulink block diagrams in this chapter are also available in full-page landscape format in Appendix B for improved readability.

Design Goals and Data Formatting

This section describes the design goals, constraints, and data formatting requirements that drive the hardware accelerator architecture.

Design Goals and Constraints

The design caches the minimum data needed for unpadded convolution using a circular buffer architecture. Once the buffer fills with input data, convolution proceeds with the stored kernel. Read operations must complete before new writes begin, or may overlap during active reads (overlapping meaning simultaneous reading and writing at the same memory position). The CMOS sensor pixel clock determines input data rate. Kernel weights are loaded by the user and must be pre-flipped; the hardware performs no kernel flipping.

The clock rate used by the accelerator is determined by the clock rate of the overall processing pipeline shown in Figure 61. The entire design operates on a single synchronous clock domain, requiring no asynchronous clock crossing or clock domain conversion logic. All three subsystems (Memory Controller, Data Slicer/Unpacker, and Convolution Engine) operate on the same clock.

Input Data Formatting The hardware accepts two data streams with corresponding valid signals:

- **Frame Data:** 12-bit fractional values representing binned spectra, serialized from lowest to highest bin position for each spatial location. After 400 bins, the stream advances to the next spatial position.
- **Kernel Data:** 18-bit fractional weight values, serialized from lowest to highest kernel position for each acquisition frame, then repeating for subsequent acquisition positions.

Valid signals indicate data availability on each clock cycle, with no timing constraints between valid transitions.

Both input streams use pure fractional representation (no integer bits). Frame data and kernel weights are normalized to $[0, 1]$ in previous pipeline stages. While pure fractional representation technically cannot represent exactly 1.0 (the maximum representable value is $1 - 2^{-12}$ for frame data and $1 - 2^{-18}$ for kernel weights), this format provides maximum numeric resolution. Allocating even a single integer bit would reduce resolution by half, wasting precision since normalized values never exceed 1.0. The idea of this choice is to prioritize representation density over the ability to represent the exact value 1.0.

Table 4: Hardware Accelerator System Parameters

Parameter	Value
Input Dimensions ($X \times Y \times Z$)	$1100 \times 400 \times 5$
Kernel Dimensions ($X \times Y \times Z$)	$5 \times 400 \times 5$
Frame Data Bit Width	12 bits (fractional)
Kernel Weight Bit Width	18 bits (fractional)
Memory Row Width	72 bits
Frame Values per Row	6
Kernel Values per Row	4
Circular Buffer Depth	5 scanlines
Frame RAM Depth	73,700 addresses per Ram Block
Kernel RAM Depth	500 addresses per Ram Block

Output Data Formatting The hardware outputs 45-bit fixed-point values (15-bit integer, 30-bit fractional) representing convolution results from first to last position. Each circular buffer produces 1096 output values with accompanying valid signals.

The output bit-width is determined by the accumulation of 10,000 multiply-accumulate operations ($5 \times 400 \times 5$ kernel). Each multiplication produces a 30-bit result (12-bit input + 18-bit weight). The minimum bit-width to prevent overflow is:

$$b_{total} = b_x + b_w + \lceil \log_2(C_{in} \cdot K_h \cdot K_w) \rceil + 1 = 12 + 18 + 14 + 1 = 45 \text{ bits} \quad (9)$$

where the 14-bit term accounts for accumulation growth ($\lceil \log_2(10000) \rceil = 14$) and the additional bit provides sign representation. The 45-bit output format (15 integer, 30 fractional) allows arithmetic operations to occur with no overflow.

This single-layer implementation can be extended to multi-layer networks by modifying kernel dimensions and data sizes in the hardware design.

Memory Architecture

The accelerator processes hyperspectral data organized as a 3D datacube with dimensions 1100 (spatial X) $\times$ 400 (spectral bins Y) $\times$ 5 (acquisitions Z). The memory subsystem uses a circular buffer architecture implemented with Xilinx UltraRAM blocks [68] to store five acquisition frames. Processing begins once the buffer fills completely. As new frames arrive, the oldest frame is discarded and convolution proceeds continuously for every new frame after initial filling. Figure 63 shows the complete datacube structure.

The memory subsystem divides into two spaces, both using UltraRAM blocks:

- **Frame Memory:** Stores incoming hyperspectral data in a 5-frame circular buffer. Each frame holds 1100 spatial positions $\times$ 400 spectral bins of 12-bit data.
- **Kernel Memory:** Stores the $5 \times 400 \times 5$ pre-trained CNN kernel (18-bit weights). Loaded once at startup, but can be reloaded as needed.

The following subsections describe the memory organization and allocation scheme, the data retrieval and alignment mechanisms, and their hardware implementations.

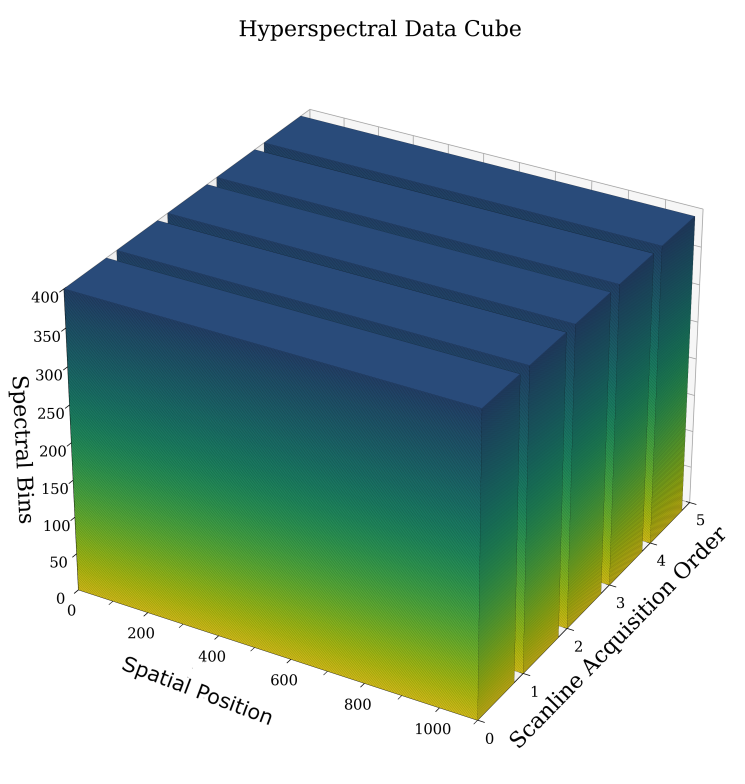


Figure 63: Example of the full data cube.

UltraRAM Organization and Memory Mapping

The memory subsystem uses Xilinx UltraRAM blocks available in the UltraScale+ architecture [68]. Each UltraRAM block is a single-clock dual-port memory with 288 Kb of storage capacity. Multiple blocks can be chained to increase memory depth. Each block is configured as a 72-bit width $\times$ 4K rows design, allowing each row to hold exactly 6 spectral bins of 12-bit depth (72 bits $\div$ 12 bits/bin = 6 bins/row), storing 24,000 spectral bins per block.

For one binned frame of 1100 spatial pixels $\times$ 400 spectral bins = 440,000 spectral bins total, the system requires $\lceil 440,000/24,000 \rceil = 19$ blocks. The XCZU15EG device provides 31.5 Mb of UltraRAM, translating to 109 blocks available. For frame storage, the system allocates 5 frames $\times$ 19 blocks/frame = 95 UltraRAM blocks, leaving 14 blocks for kernel weights. The kernel dimensions

of $5 \times 400 \times 5$ result in 10,000 weight values at 18 bits each, requiring 180,000 bits total. With 4 weights per 72-bit row, this requires 2,500 rows, fitting within a single UltraRAM block. Data is packed into 72-bit rows: six 12-bit spectral bin values for frame data, four 18-bit weight values for kernel data. This means that to access a specific weight value, the system must read the entire row it resides in and slice the bits according to the packing scheme.

The 3D datacube must map to UltraRAM's 1D address space. The standard 3D-to-1D mapping formula is:

$$\text{Address}_{1D} = z \times (d_y \times d_x) + y \times d_x + x \quad (10)$$

where d_x , d_y , and d_z are the maximum dimensions in X, Y, and Z. More generally, for N-dimensional arrays:

$$\text{Address}_{1D} = \sum_{k=1}^N i_k \cdot \prod_{j=1}^{k-1} d_j \quad (11)$$

However, this design uses a custom allocation scheme for the pipeline rather than the standard formula. Instead of dedicating each RAM block to a single frame or scanline, the datacube is distributed across multiple RAM block groups, allowing sequential access during convolution. Section 2 explains this optimization in detail. Figure 64 visualizes this custom mapping.

Figure 65 illustrates the custom allocation scheme across five RAM block groups, each holding one spatial position slice of the datacube. Each group maintains a fixed position relative to the cube. For example, RAM Block Group 0 (red) holds Spatial Position 0 and receives Spectral Bins 0-399 for each acquisition frame (Scanline Acquisitions 0-4). After completing all five acquisition positions for one spatial location, the system advances five spatial positions and repeats.

This scheme allows reading complete $5 \times 400 \times 5$ datacube slices sequentially for convolution with the $5 \times 400 \times 5$ kernel. The entire datacube streams out without calculating individual pixel addresses, simplifying control logic.

The frame index cycles from 0-4, wrapping back to 0, which determines memory placement offsets. Frame offsets in the UltraRAM address space are: Frame 0 (rows 0-67), Frame 1 (rows 67-134), Frame 2 (rows 134-201), Frame 3 (rows 201-268), Frame 4 (rows 268-335).

Hyperspectral RAM Layout Close Up

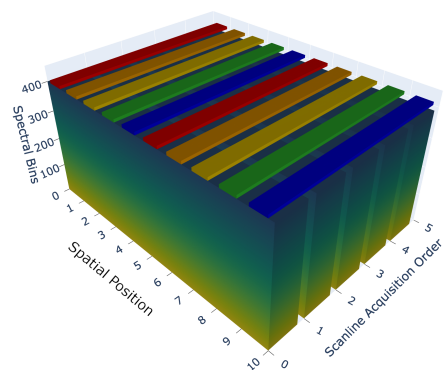


Figure 64: Datacube mapped to RAM block groups. Each color represents one RAM block group storing a spatial position slice.

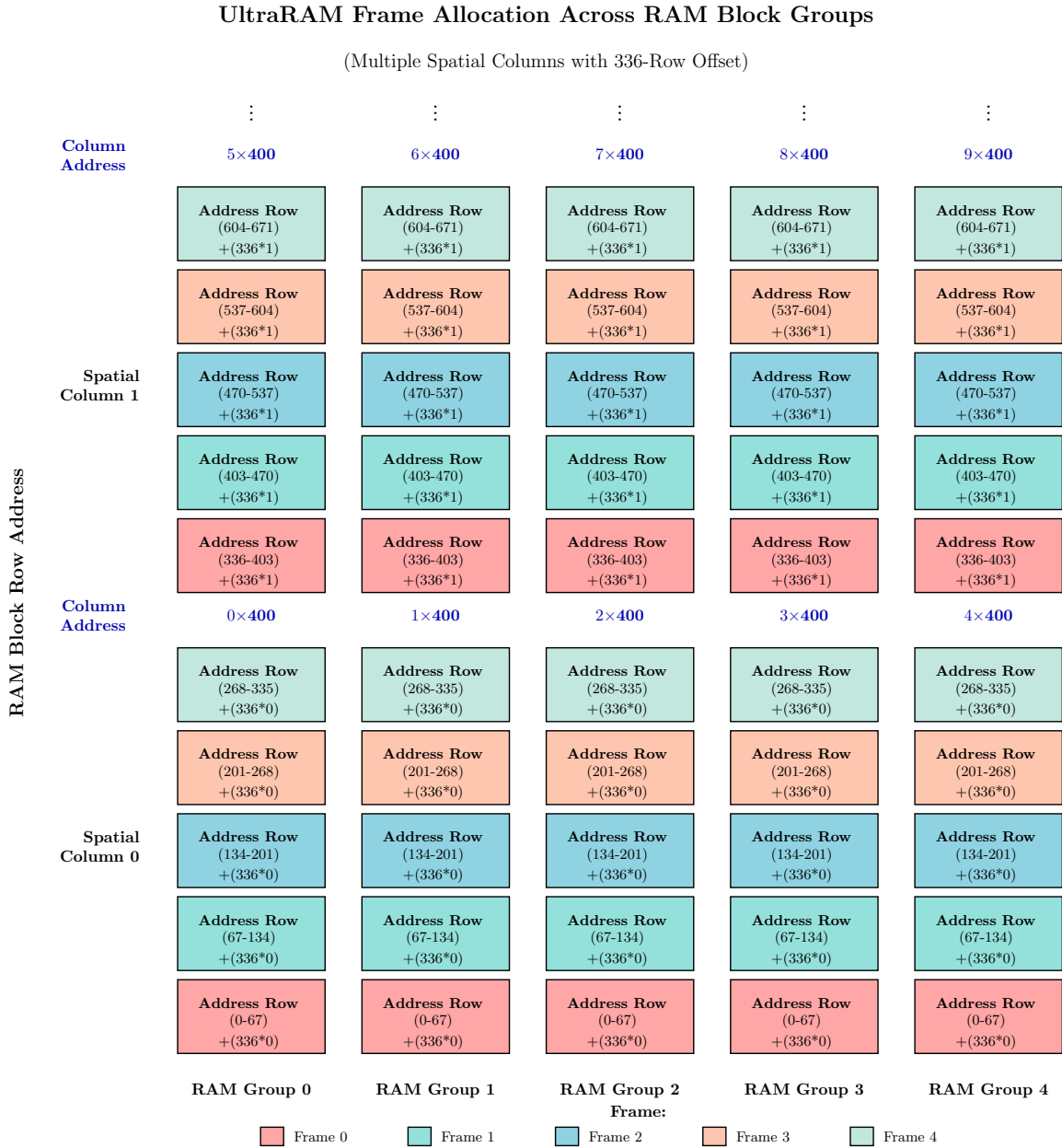


Figure 65: UltraRAM memory allocation scheme across RAM block groups showing spatial column progression and frame organization.

RAM Block Groups: Five independent memory groups (labeled RAM Group 0-4) are distributed horizontally. Each group stores data from one spatial position within a scanline, representing 1×400 spectral bins. The groups work together to hold the complete 5×400 spectral data for each spatial column.

Address Rows and Frame Organization: Within each RAM group, five color-coded address rows are stacked vertically, corresponding to the five acquisition frames (Frame 0-4 as shown in the legend). Each address row occupies 67 rows of UltraRAM memory. The color coding maintains consistent acquisition frame identification across all groups.

Column Address Sequencing: The blue labels above each group (0×400 , 1×400 , 2×400 , etc.) indicate the sequential write order of 1×400 spectral bin columns. For Spatial Column 0, columns 0×400 through 4×400 are written to RAM Groups 0-4. For Spatial Column 1, columns 5×400 through 9×400 are written to the same groups but at higher memory addresses. This pattern continues for all 1100 spatial columns.

Offset Calculation: The formula $+(336 \times n)$ shown in each address row indicates the memory offset applied based on the spatial column index n . For Spatial Column 0 ($n=0$), the offset is $+(336 \times 0) = 0$, resulting in base addresses (0-67, 67-134, etc.). For Spatial Column 1 ($n=1$), the offset is $+(336 \times 1) = 336$, shifting all addresses by 336 rows (336-403, 403-470, etc.). The 336-row offset accounts for 5 frames $\times$ 67 rows per frame plus a 1-row gap for separation.

Memory Access Pattern: This organization allows sequential access. During convolution operations, the system reads five consecutive address rows from each of the five RAM groups, producing a $5 \times 400 \times 5$ data window. The offset scheme allows the circular buffer to maintain acquisition continuity as new frames arrive and old frames are overwritten, with the pipeline requiring no explicit knowledge of the buffer state.

Frame Value Update: The frame value gets updated every time we write a new frame to the RAM blocks, and we simply return to 0 after completing the 5th frame in the sequence and repeat. This handles a portion of maintaining the circular buffer.

Read Operation: When reading data for convolution, the system reads addresses 0-73699

from each RAM block group sequentially. The pre-organized memory layout ensures that data emerges in the correct order for the pipeline to perform convolution operations without requiring complex address calculations during real-time processing.

Kernel Memory Organization: The writing to the kernel memory block is much simpler. We split the kernel along the Z dimension (acquisition dimension) into 5 slices of 400×5 . Then we write each of these slices to the kernel memory to their respective RAM block group. Figure 66 illustrates how the kernel is distributed across the five RAM groups.

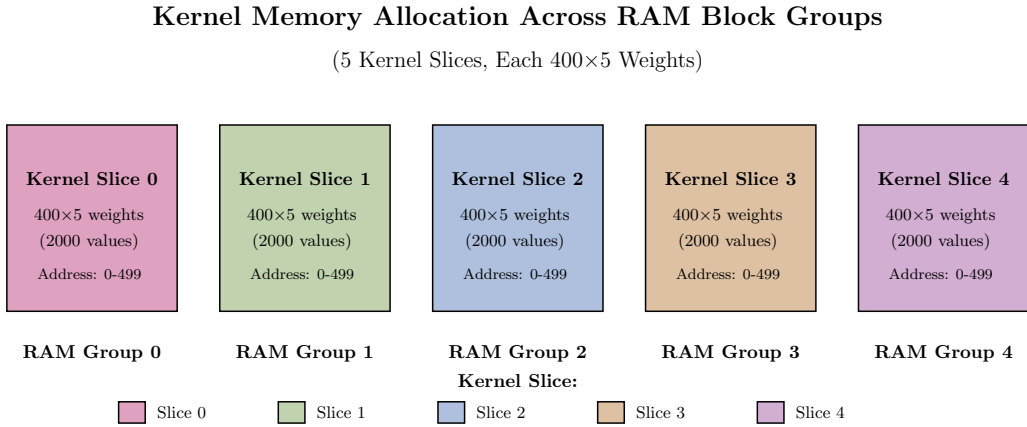


Figure 66: Kernel memory allocation across RAM block groups with Z-dimension slicing.

The diagram illustrates the kernel memory organization:

Z-Dimension Slicing: The $5 \times 400 \times 5$ kernel is partitioned along the Z (acquisition) dimension into five independent slices. Each slice contains $400 \times 5 = 2000$ weight values. The color coding (Slice 0-4) corresponds to the acquisition frame organization used in the frame memory, maintaining consistent ordering across the system.

RAM Group Distribution: Each of the five slices is stored in its corresponding RAM block group at addresses 0-499. Since each 72-bit memory row holds four 18-bit kernel weights, the 2000 weights per slice require 500 memory rows ($2000 \div 4 = 500$).

Memory Controller Implementation The Memory Controller implements the circular buffer architecture and memory allocation scheme described above. It manages data movement between

input streams and UltraRAM blocks through two independent parallel units: the Frame Memory Controller and Kernel Memory Controller. The Memory Manager Internal component (Figure 67) coordinates memory access patterns and generates control signals for both frame and kernel memory, managing the five UltraRAM blocks and controlling write/read access to each block.

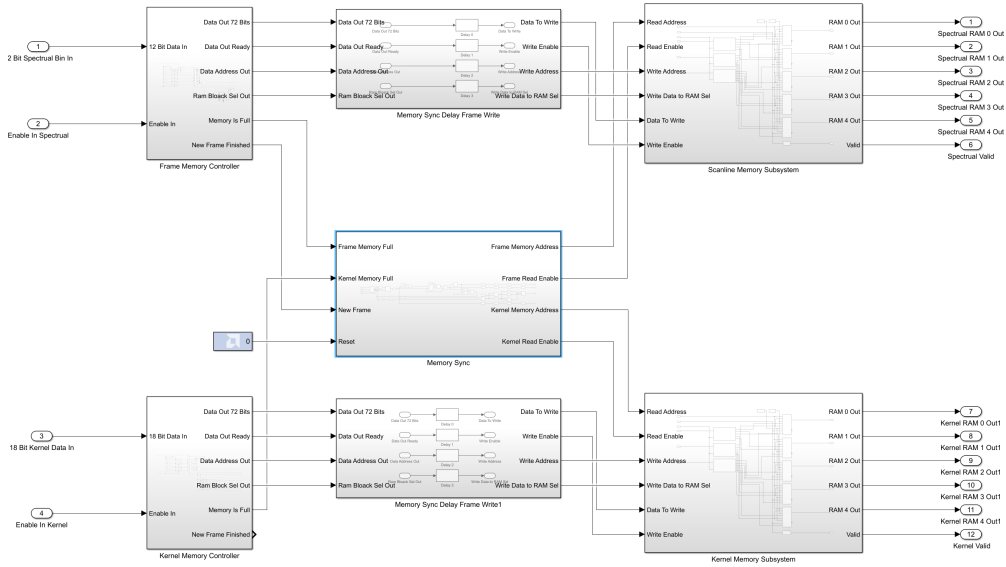


Figure 67: Memory Manager Internal implementation showing RAM selection and control logic.

The RAM Memory block itself is implemented using Xilinx Ultra RAM primitives, configured for the specific dimensions required by this application (Figure 68).

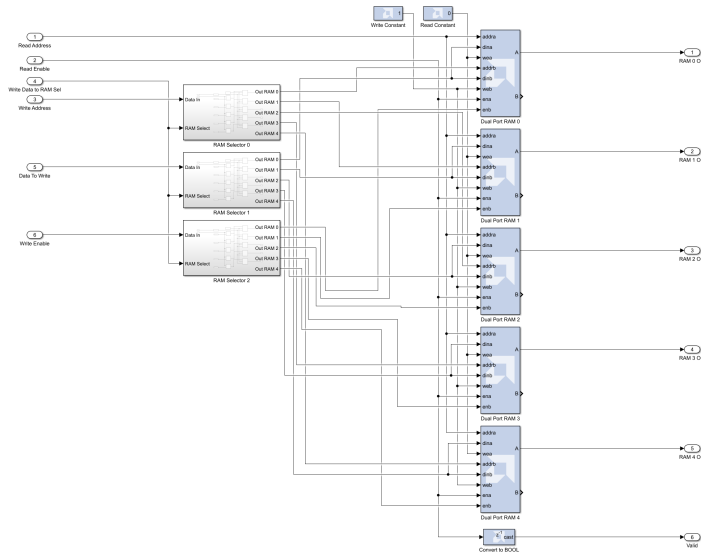


Figure 68: RAM Memory block configured as Ultra RAM primitive.

Frame Memory Controller: The Frame Memory Controller implements the frame memory allocation strategy shown in Figure 65, storing incoming hyperspectral data across five RAM block groups with the custom addressing scheme. Each incoming 12-bit spectral bin value is packed into 72-bit UltraRAM rows (6 values per row). The circular buffer uses 5 UltraRAM blocks, each storing one complete frame. Figures 69 and 70 show the implementation details.

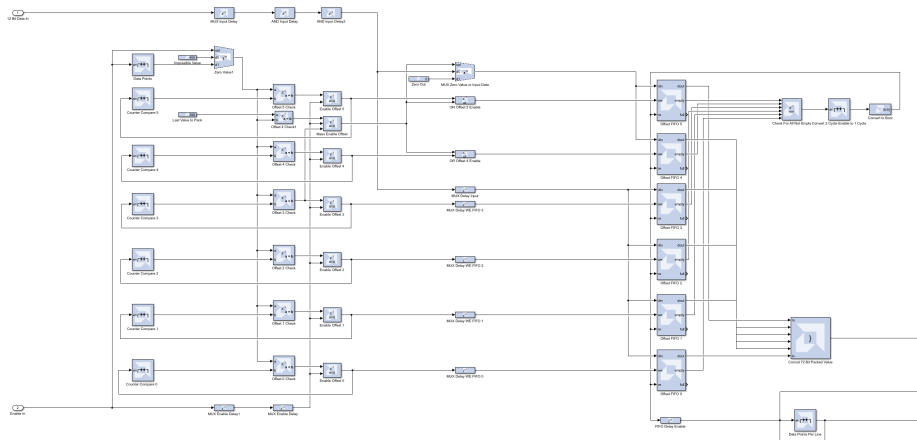


Figure 69: Frame Memory Controller Part 1: Input packing and write control logic.

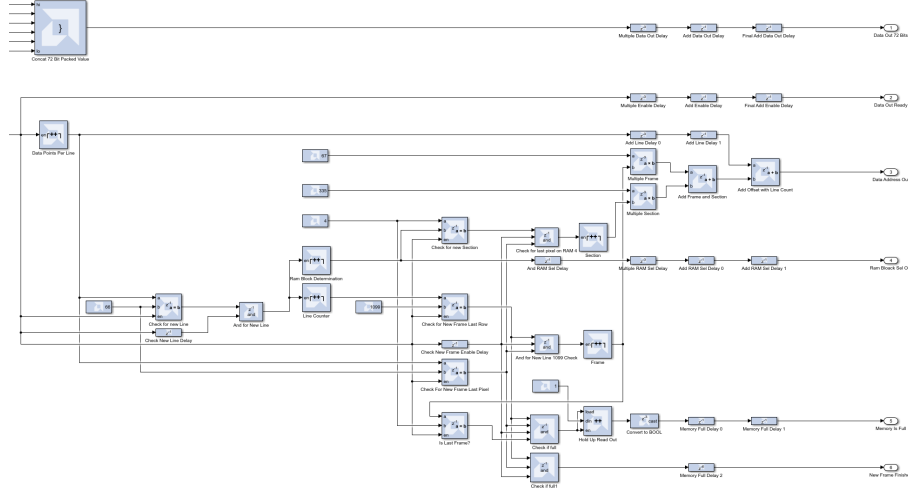


Figure 70: Frame Memory Controller Part 2: Read control and RAM selection logic.

Incoming 12-bit values pack into 72-bit rows using a shift-and-concatenate approach. A 72-bit packing register accumulates six values (Bits [71:60] through Bits [11:0]). When six values fill the register, the 72-bit row writes to the active UltraRAM block. When scanlines end with fewer than six remaining values, the partial row pads with zeros before writing.

Kernel Memory Controller: The Kernel Memory Controller implements the kernel memory allocation strategy illustrated in Figure 66. It stores CNN kernel weights distributed across the five RAM block groups. Kernel weights are serialized in $Y \rightarrow Z \rightarrow X$ order (spectral bins, then acquisition slices, then spatial positions) to match the data flow requirements. Four 18-bit values pack into each 72-bit row (Bits [71:54] through Bits [17:0]), requiring 2,500 memory rows total. Figures 71 and 72 show the implementation details.

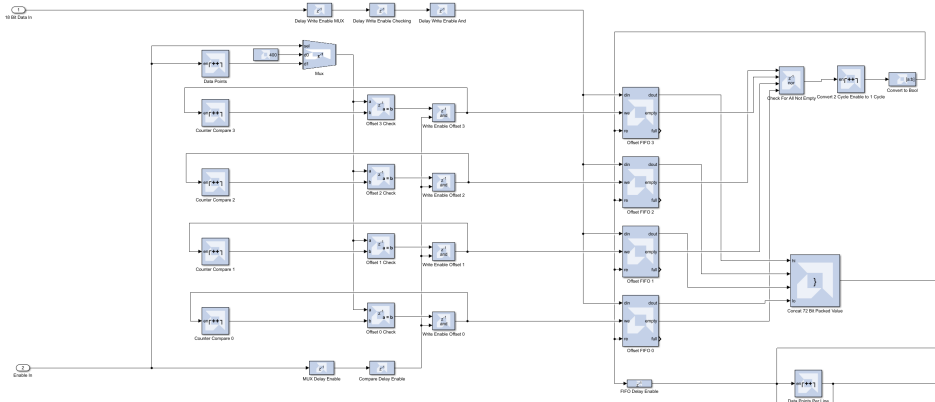


Figure 71: Kernel Memory Controller Part 1: Weight input and packing logic.

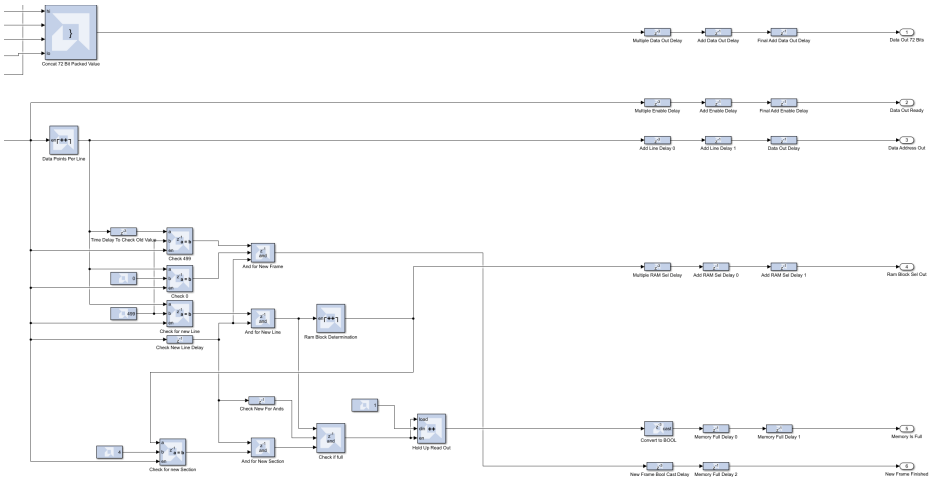


Figure 72: Kernel Memory Controller Part 2: Memory addressing and read control.

RAM Selector: The RAM Selector module coordinates access across the five UltraRAM blocks, managing write/read availability through a state machine that prevents read-write conflicts. Figure 73 shows the implementation, which uses a multiplexer tree that routes read and write ports to the appropriate Ultra RAM block based on the current buffer state.

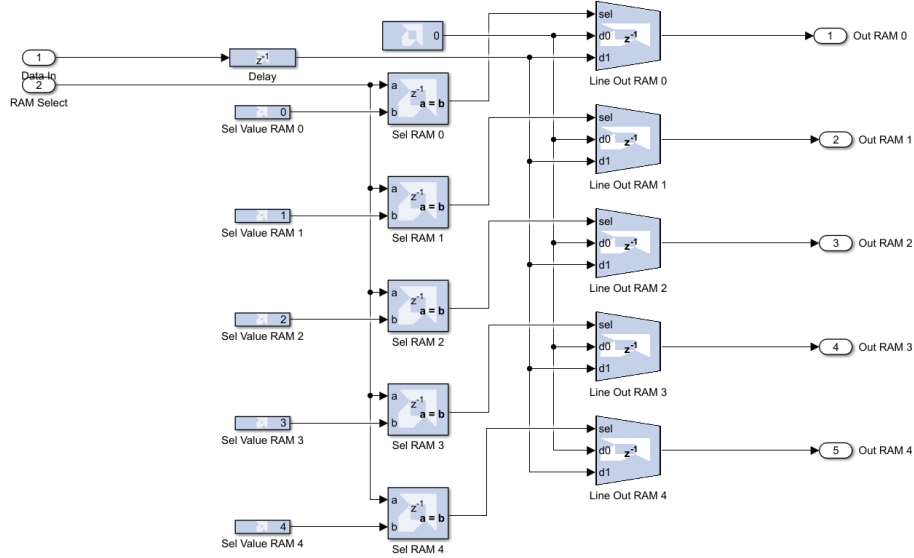


Figure 73: RAM Selector implementing multiplexing between five Ultra RAM blocks.

Data Retrieval and Alignment

With the data and kernel memory organized as described above, the system must retrieve and align data from the circular buffer. Alignment is achieved through careful address management and unpacking logic.

Circular Buffer Management: The system tracks the current oldest frame position in the circular buffer and applies an appropriate address offset when reading kernel weights. For example, when frame position 0 is the oldest frame, the system reads addresses 0-499 from each RAM group. When frame position 1 becomes the oldest frame, the system applies an offset of 400 rows, reading addresses 400-499 first, then wrapping to 0-399. This ensures the system always reads the correct kernel weights that correspond to the state of the circular buffer. The system only ever reads addresses 0-73699 from each RAM block group sequentially in the frame memory block, which is the complete address space of the frame memory block.

Row Alignment: Frame memory rows contain 6 spectral bins (12 bits each) while kernel memory rows contain 4 weights (18 bits each). To align these different packing densities, the system reads 3 kernel rows for every 2 frame rows, unpacks the values, and pairs each frame value with its

corresponding kernel weight. This 2:3 read ratio occurs per RAM group, delivering 120 values (60 12-bit frame values + 60 18-bit kernel weights) for each valid read output from the memory system and data slicer. Figure 74 illustrates this alignment mechanism.

The 2:3 read ratio requires a buffering stage to collect and align corresponding frame-kernel pairs before they enter the convolution pipeline. While the diagram implies that the 18-bit weights and 12-bit frame values are grouped into pairs, this is only the case for some of the convolution operations. This will be explained in more detail in Section 2.

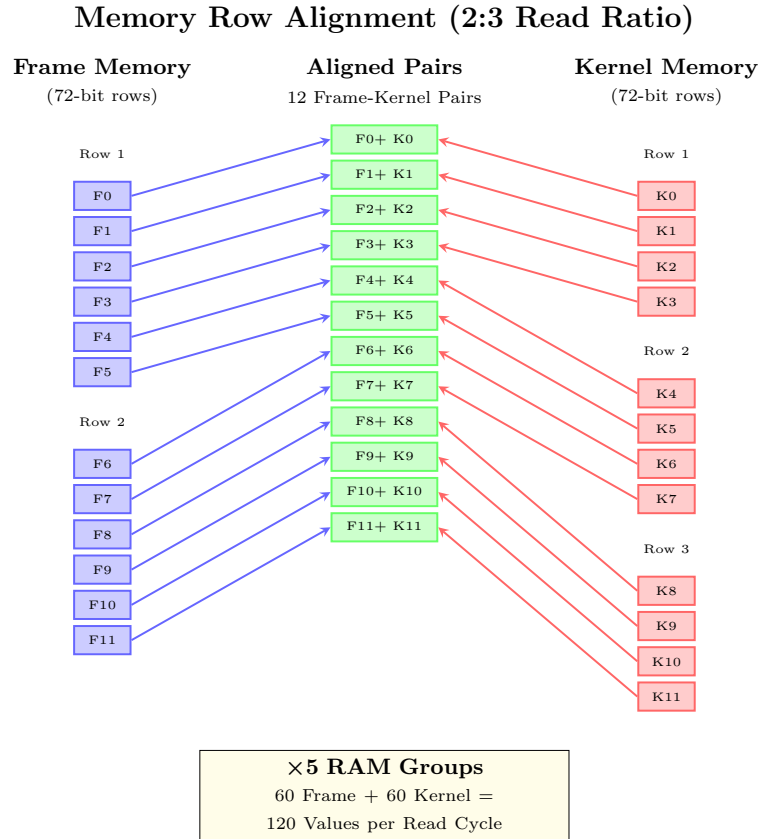


Figure 74: Memory row alignment showing 2:3 read ratio between frame and kernel memory.

Data Slicer Implementation The Data Slicer implements the bit-slicing operations required to unpack the 72-bit memory words into individual 12-bit frame values (six per word) or 18-bit

kernel weights (four per word) for the Convolution Engine. This unpacking is the inverse of the packing operations performed by the Frame and Kernel Memory Controllers. Figure 75 shows the top-level implementation.

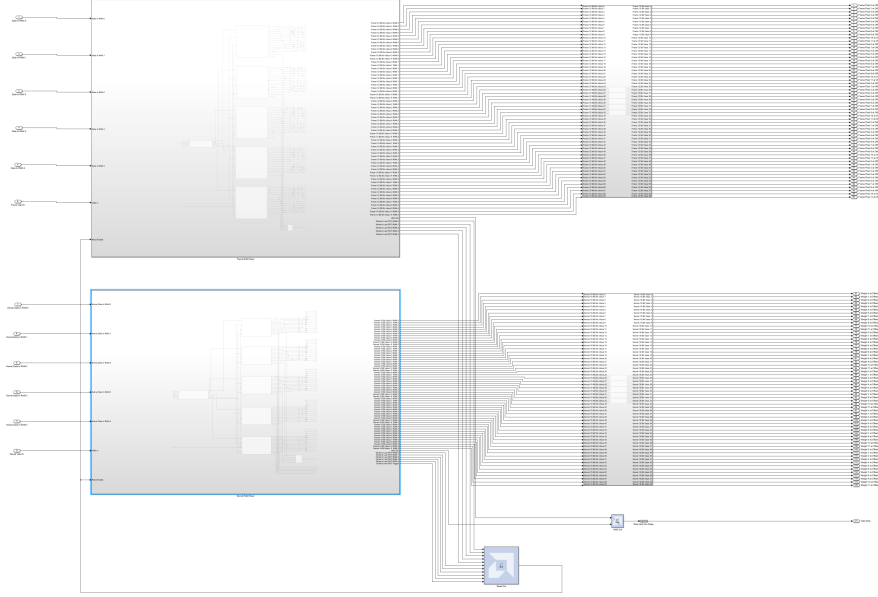


Figure 75: Data Slicer top-level implementation coordinating frame and kernel data unpacking.

The unpacking operation is purely combinational logic that extracts specific bit ranges from the 72-bit input word. For frame data, six separate slices are extracted, while for kernel data, four slices are extracted. This unpacking happens in parallel on every clock cycle where valid data is available. Figure 76 shows the implementation for extracting six 12-bit values.

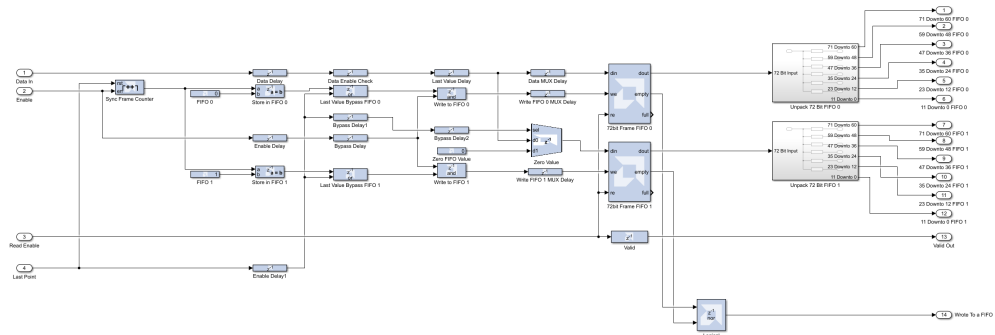


Figure 76: 72-Bit Unpacker for frame data showing extraction of six 12-bit values.

The unpacker connected to RAM output is shown in Figure 77, which illustrates how the unpacked values are synchronized with the RAM read timing.

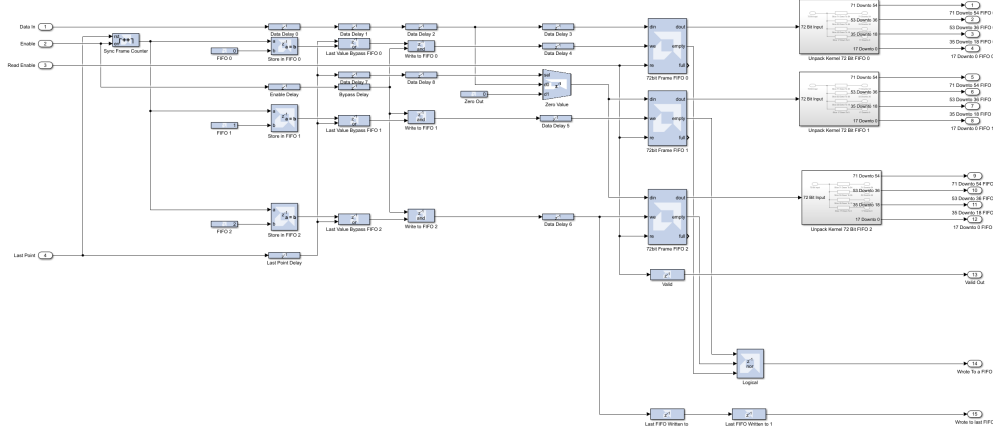


Figure 77: 72-Bit Unpacker connected to RAM output with timing control.

To coordinate the unpacked data values with the convolution engine, a Data Point Tracker module maintains counters and state information about which data values are currently being processed. This tracker generates control signals that indicate when a complete set of data is ready for multiplication and accumulation. Figure 78 shows the implementation.

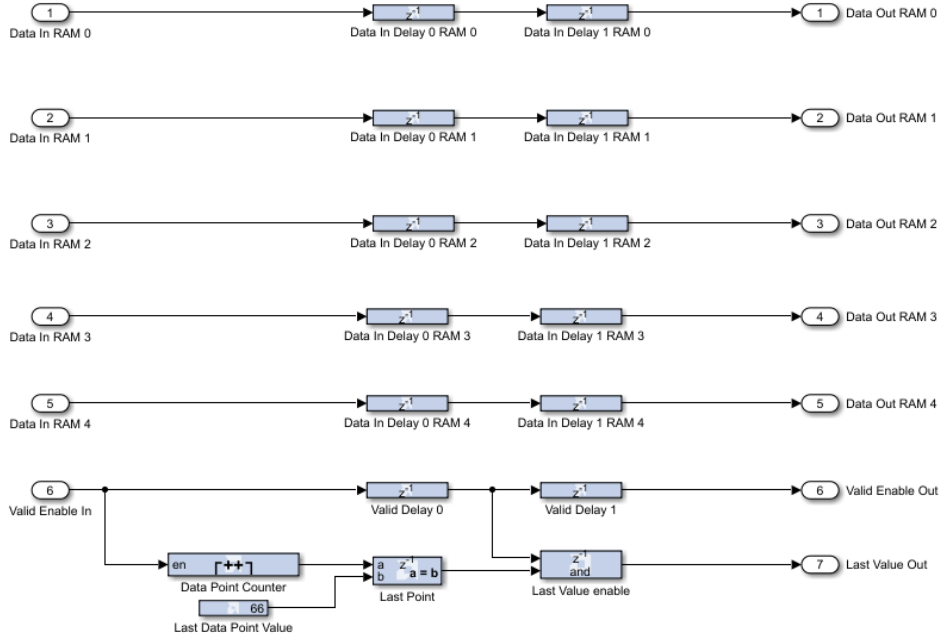


Figure 78: Data Point Tracker coordinating unpacked data with convolution timing.

Convolution Pipeline Architecture

This section describes the convolution pipeline, which performs the multiply-accumulate operations to compute the 3D convolution. The pipeline implements a stride-5 operation with partial accumulation and coordinates data flow from unpacking through computation to result output.

Value Alignment and Partial Accumulation Strategy

The convolution engine operates with a spatial stride of 5. The memory system reads $5 \times 400 \times 5$ datacube windows starting at spatial positions 0-4, then advances by a stride of 5 to positions 5-9, 10-14, and so on. While the memory system only reads these non-overlapping stride-5 windows, the convolution engine computes all intermediate overlapping windows (positions 1-5, 2-6, 3-7, 4-8) through partial accumulation.

When the system reads positions 0-4, it begins computing intermediate window 1-5 by processing slices 1-4 with appropriately shifted kernel weights (kernel weight 0 pairs with slice 1,

weight 1 with slice 2, etc.). These partial products accumulate in the MAC units. When the next stride reads positions 5-9, slice 5 arrives and completes the accumulation for window 1-5, allowing the MAC to output the final convolution result. This pattern repeats for all intermediate windows: partial results accumulate across stride boundaries, with each new stride providing the missing slice needed to complete multiple pending convolutions. Figure 79 illustrates this progression of partial accumulation across stride boundaries.

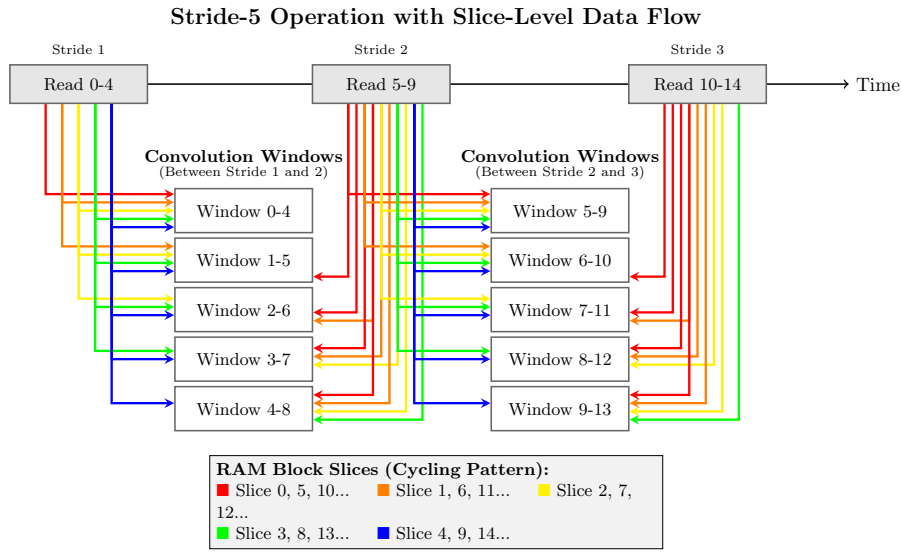


Figure 79: Stride-5 operation with partial accumulation showing how intermediate windows are computed across stride boundaries.

Pipelined MAC Architecture

The convolution engine implements a hierarchical multiply-accumulate structure that resides within each of the five RAM block slices shown in Figure 79. The architecture consists of multiple pipeline stages, culminating in a final accumulator that produces the valid kernel result.

As illustrated in Figure 80, each slice processes 12 pixel-weight pairs per acquisition slice. The computation begins with 12 parallel multiplication operations, where each frame pixel value (p_i) is multiplied by its corresponding kernel weight (w_i). These products then flow through a hierarchical accumulation network consisting of three stages: the first stage combines the 12 products into 6

partial sums using pairwise addition, the second stage reduces these to 3 intermediate results, and the third stage produces a single combined value representing one complete slice calculation.

Since each convolution encompasses 5 acquisition slices across 170 spatial positions (spanning the full 400-pixel spectral dimension with stride-5 sampling), each slice accumulator must aggregate results from 170 accumulations. This per-slice accumulator holds and accumulates the slice results as they stream through the pipeline, collecting all 170 accumulations before asserting a completion flag.

The architecture employs 5 independent slice accumulators operating in parallel, one for each RAM block slice (0-4, 1-5, 2-6, 3-7, 4-8). Once all five slice accumulators have completed their 170 accumulations and raised their completion flags, the final accumulator stage collects these five partial results. This final accumulation stage sums the outputs from all slices, producing the complete convolution result for that kernel position. The valid kernel result is then output to the subsequent processing stages.

Pipelined MAC Accumulation Structure

Single Slice: 12 Products $\times$ 170 Accumulations

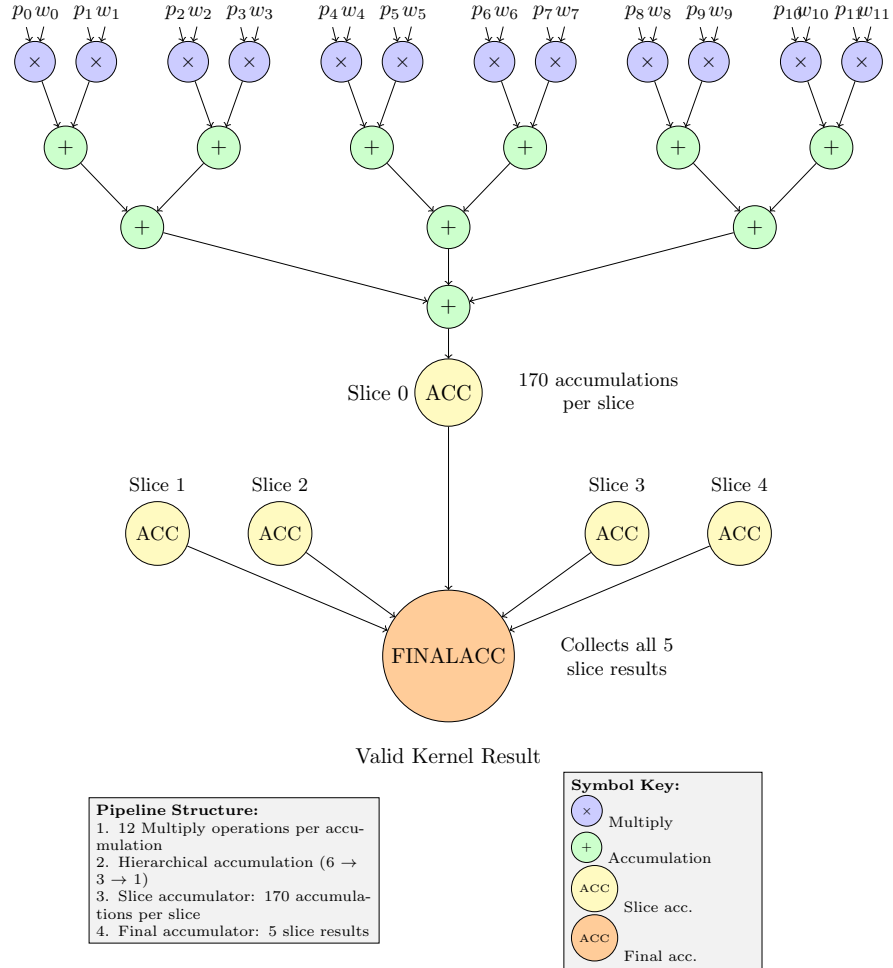


Figure 80: Pipelined MAC accumulation structure for a single slice, showing 12 multiply operations feeding into hierarchical accumulation stages, followed by a slice accumulator that collects 170 accumulations. The bottom shows all 5 slice accumulators feeding into the final accumulator that produces the valid kernel result.

Convolution Engine Implementation

The Convolution Engine implements the stride-5 operation with partial accumulation (Figure 79) and the pipelined MAC structure (Figure 80). It multiplies input data with kernel weights and accumulates the products. The engine uses the FPGA's DSP slices for multiplication operations.

The convolution operation for a single output position can be expressed as:

$$y[x_{out}, z_{out}] = \sum_{i=0}^4 \sum_{j=0}^{399} \sum_{k=0}^4 I[x_{out} + i, j, z_{out} + k] \cdot K[i, j, k] \quad (12)$$

where I is the input data, K is the kernel, and y is the output. This requires $5 \times 400 \times 5 = 10,000$ multiply-accumulate operations per output position.

The Convolution Engine Block, shown in Figure 81, is the top-level module that implements the Convolution Engine. This module coordinates all the computational components required for the convolution operation. A larger version is available in Appendix B.

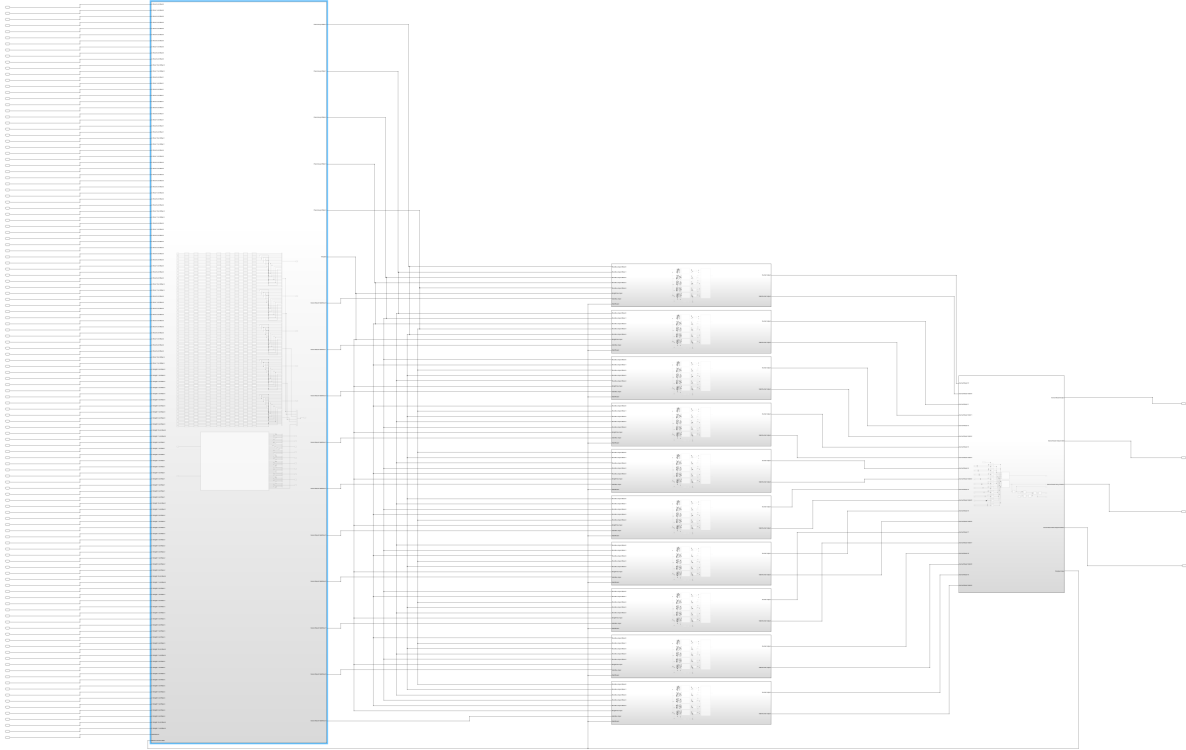


Figure 81: Math Block top-level implementation of the Convolution Engine.

The Math Block contains several interconnected components:

- **DSP Slices:** Core multiplication units using DSP48E2 primitives [68]
- **Adder Tree:** Hierarchical adder structure for parallel accumulation

- **Accumulate Results:** Two-stage accumulation for partial and final sums
- **Kernel Organizer:** Weight distribution and routing logic
- **Result Organizer:** Output formatting and valid signal generation

The Convolution Engine Block is configured to use the DSP48E2 primitive available in the UltraScale+ architecture [68]. This primitive provides dedicated hardware for multiplication with an optional pre-adder and accumulator, improving performance compared to using general-purpose CLB logic.

Adder Tree: To accumulate the products from the multiplication units, a Multiple Add tree is implemented. This tree performs parallel addition of multiple values in a hierarchical fashion, reducing the depth of the adder chain and improving timing. Figure 82 shows the top-level Multiple Add implementation, and Figure 83 shows the internal structure.

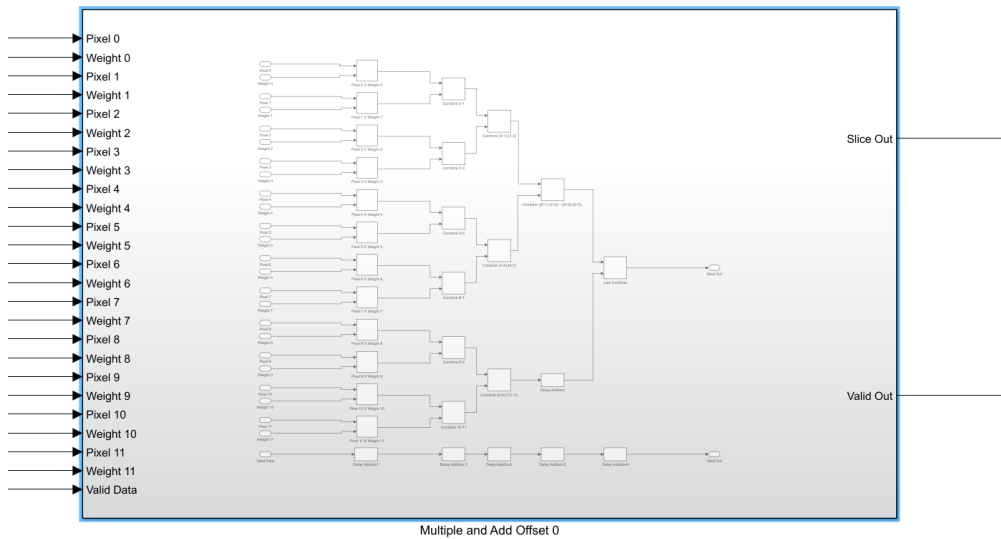


Figure 82: Multiple Add tree for parallel accumulation of products.

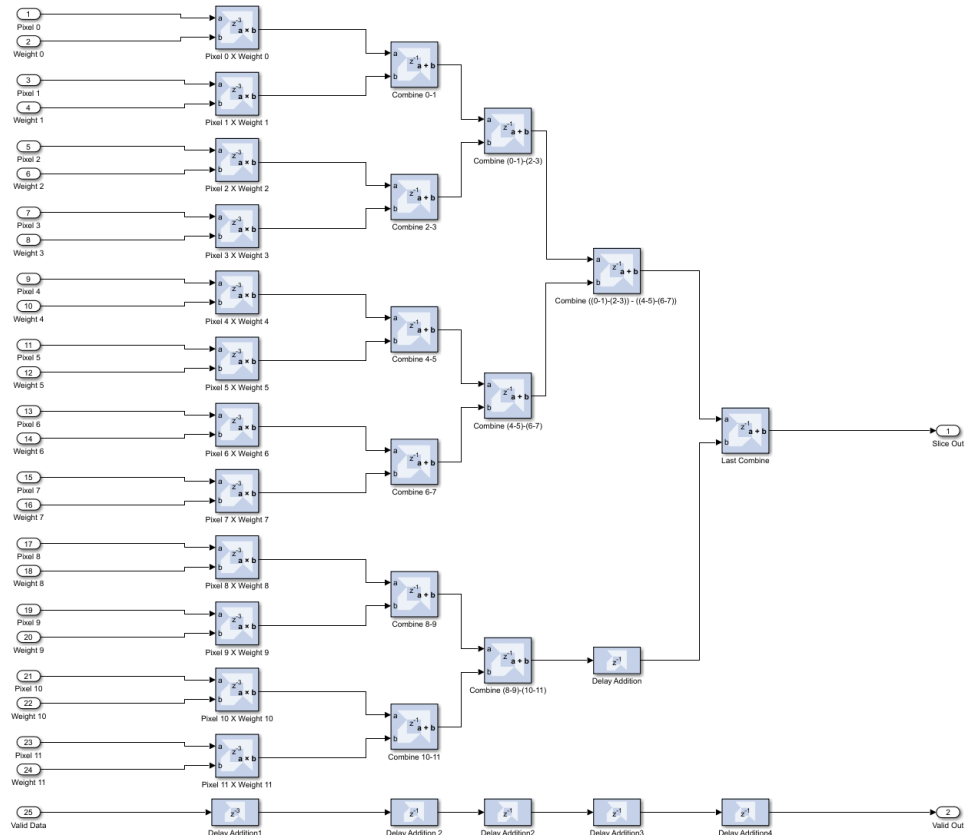


Figure 83: Multiple Add internal implementation showing adder tree structure.

Result Accumulation: The accumulation of results happens in two stages. First, partial sums are computed for each spectral bin across the kernel depth. Then, these partial sums are accumulated across spatial positions to produce the final output. The Accumulate Results module manages this two-stage accumulation process. The top-level implementation and internal accumulator structure are shown in Appendix B.

An Accumulator Offset module, shown in Figure 84, handles the addressing logic to ensure partial sums are accumulated to the correct output positions.

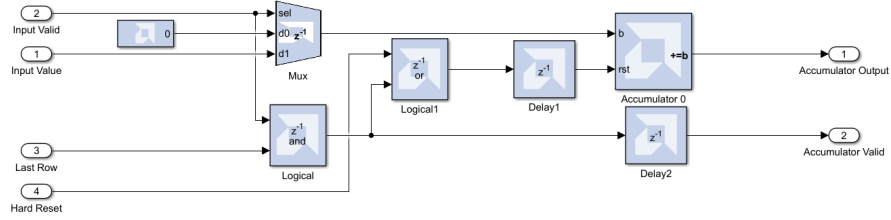


Figure 84: Accumulator Offset module for addressing partial sum storage.

Kernel Organization and Result Output: The Kernel Organizer module coordinates the delivery of kernel weights to the multiplication units in the correct order. This module reads kernel data from memory and distributes it to the appropriate DSP slices based on the current convolution position. The final stage is the Result Organizer, which formats the accumulated convolution results into the output stream, generates the output valid signals, and ensures that results are presented in the correct order. The detailed implementations are shown in Appendix B.

Performance Of Hardware

Computational Throughput

This section quantifies the data and operations for a single convolution and for processing a complete frame.

Data Processed per Convolution: Each convolution operation processes a $5 \times 400 \times 5$ datacube window against the kernel, consuming:

Input Data:

$$D_{input} = 5 \times 400 \times 5 \times 12 \text{ bits} = 120,000 \text{ bits} = 120 \text{ Kb} \quad (13)$$

Kernel Data:

$$D_{kernel} = 5 \times 400 \times 5 \times 18 \text{ bits} = 180,000 \text{ bits} = 180 \text{ Kb} \quad (14)$$

Total Data Movement:

$$D_{total} = 120 \text{ Kb} + 180 \text{ Kb} = 300 \text{ Kb per convolution} \quad (15)$$

The accelerator reads 300 Kb from UltraRAM per convolution output.

Arithmetic Operations Breakdown: The convolution consists of four arithmetic stages:

1. Multiplication Stage: The kernel dimensions of 5 (spatial) $\times$ 400 (spectral) $\times$ 5 (acquisition) determine the number of multiplications:

$$N_{mult} = 5 \times 400 \times 5 = 10,000 \text{ multiplications (12-bit} \times \text{18-bit} \rightarrow \text{30-bit products)} \quad (16)$$

Each input pixel value is multiplied by its corresponding kernel weight, producing 10,000 30-bit products.

2. Hierarchical Addition Trees: The 12 products per memory read cycle feed into a three-stage adder tree. Each stage combines pairs of products:

- Stage 1: $12/2 = 6$ additions (12 products $\rightarrow$ 6 partial sums)
- Stage 2: $6/2 = 3$ additions (6 partial sums $\rightarrow$ 3 partial sums)
- Stage 3: $3 - 1 = 2$ additions (3 partial sums $\rightarrow$ 2 $\rightarrow$ 1 slice result)

Each tree performs $6 + 3 + 2 = 11$ additions per group. The number of groups is determined by the memory read pattern: due to the 2:3 memory read ratio (2 frame rows to 3 kernel rows),

each memory read cycle delivers $6 \text{ values/row} \times 2 \text{ rows} = 12$ pixel-weight pairs. To process the 400 spectral bins requires $\lceil 400/12 \rceil = 34$ read cycles per acquisition slice. Across 5 acquisition slices, this yields $34 \times 5 = 170$ groups per RAM block slice. With 5 RAM block slices operating in parallel:

$$N_{tree} = 11 \text{ additions/group} \times 170 \text{ groups/slice} \times 5 \text{ slices} = 9,350 \text{ additions} \quad (17)$$

3. Slice Accumulations: Each slice accumulator maintains a running sum across all 170 groups ($34 \text{ read cycles} \times 5 \text{ acquisition slices}$). Each group result from the adder tree is added to the accumulator, requiring one addition per group:

$$N_{slice} = 170 \text{ accumulations/slice} \times 5 \text{ RAM block slices} = 850 \text{ accumulations} \quad (18)$$

4. Final Accumulation: The 5 independent slice accumulators (one per RAM block slice: 0-4, 1-5, 2-6, 3-7, 4-8) produce 5 partial results that must be combined. Summing 5 values requires $5 - 1 = 4$ additions:

$$N_{final} = 5 - 1 = 4 \text{ additions} \quad (19)$$

Total Arithmetic Operations: Summing all four stages:

$$N_{total} = N_{mult} + N_{tree} + N_{slice} + N_{final} \quad (20)$$

$$= 10,000 + 9,350 + 850 + 4 \quad (21)$$

$$= 20,204 \text{ operations} \quad (22)$$

Each convolution performs 20,204 fixed-point arithmetic operations: 10,000 multiplications and 10,204 additions ($9,350 + 850 + 4$).

Computational Intensity: The ratio of operations to data movement:

$$I_{compute} = \frac{20,204 \text{ operations}}{300 \text{ Kb}} = 67.3 \text{ operations per Kb} \quad (23)$$

The architecture performs 67.3 operations per kilobit of data read from memory.

Parallel Execution: The architecture processes 12 multiplications per slice across 5 slices, totaling 60 multiplications per clock cycle. The adder trees execute multiple stages simultaneously.

Full Frame Processing: The complete hyperspectral datacube stored in the circular buffer contains:

$$D_{frame} = 1100 \times 400 \times 5 \times 12 \text{ bits} = 26,400,000 \text{ bits} = 26.4 \text{ Mb} \quad (24)$$

To process this entire frame, the architecture performs 1,096 convolution operations (one per spatial position, where $1100 - 5 + 1 = 1096$ valid output positions exist for the 5-pixel kernel width):

Total Data Processed per Frame: The number of valid convolution positions is determined by the input width and kernel width: 1100 (input width) $- 5$ (kernel width) $+ 1 = 1,096$ positions.

To understand the total data movement, we must distinguish between data stored uniquely in memory versus data read multiple times:

Input Frame Data: The circular buffer stores the complete input frame: $1100 \times 400 \times 5 \times 12 \text{ bits} = 26.4 \text{ Mb}$. Each convolution reads a $5 \times 400 \times 5$ window (120 Kb), but adjacent convolutions have overlapping windows. Specifically:

- Convolution at position x reads: spatial positions $[x, x + 1, x + 2, x + 3, x + 4]$
- Convolution at position $x + 1$ reads: spatial positions $[x + 1, x + 2, x + 3, x + 4, x + 5]$
- Overlap: 4 out of 5 spatial positions (80% overlap)

For 1,096 convolutions, the naive approach would read $1,096 \times 120 \text{ Kb} = 131,520 \text{ Kb}$, but with 80% overlap, the actual unique data accessed is closer to the full frame size (26.4 Mb). However, the actual memory bandwidth depends on whether the architecture:

1. **Re-reads overlapping data from UltraRAM:** Total reads $= 1,096 \times 120 \text{ Kb} = 131.5 \text{ Mb}$
2. **Caches/reuses overlapping data in registers:** Total reads $= 26.4 \text{ Mb}$ (one-time read)

Kernel Data: The kernel (180 Kb) is stored in UltraRAM. The stride-5 operation processes 5 consecutive spatial positions using the same kernel before advancing. This means:

- Positions 0-4: read kernel once
- Positions 5-9: read kernel once
- Total kernel reads: $\lceil 1,096/5 \rceil = 220$ reads
- Total kernel bandwidth: $180 \text{ Kb} \times 220 = 39.6 \text{ Mb}$

Total Memory Bandwidth (assuming register reuse):

- Input data read: 26.4 Mb (read once with sliding window reuse)
- Kernel data read: 39.6 Mb (220 reads)
- Total memory reads: $26.4 \text{ Mb} + 39.6 \text{ Mb} = 66.0 \text{ Mb}$ per frame

Total Memory Bandwidth (without reuse):

- Input data read: 131.5 Mb (1,096 separate reads of 120 Kb each)
- Kernel data read: 39.6 Mb (220 reads)
- Total memory reads: $131.5 \text{ Mb} + 39.6 \text{ Mb} = 171.1 \text{ Mb}$ per frame

Total Arithmetic Operations per Frame: Processing one complete output frame requires computing all 1,096 convolution positions. Each position requires the full 20,204 operations:

$$N_{frame} = 20,204 \text{ operations/convolution} \times 1,096 \text{ convolutions} = 22,143,584 \text{ operations} \approx 22.1 \text{ million operations} \quad (25)$$

Breaking down by operation type (multiplying each stage's per-convolution count by 1,096):

- Multiplications: $10,000 \text{ mults/conv} \times 1,096 = 10,960,000$ multiplications
- Additions: $10,204 \text{ adds/conv} \times 1,096 = 11,183,584$ additions

Minimum Operations Per Second:

Each incoming frame triggers processing of the 5-frame circular buffer, requiring 1,096 convolutions to be completed before the next frame arrives. At the target frame rate of 181 fps, the minimum required operation rate is:

$$\text{Target frame rate} = 181 \text{ frames/sec} \quad (26)$$

$$\text{Convolutions per frame} = 1,096 \text{ convolutions} \quad (27)$$

$$\text{Operations per convolution} = 20,204 \text{ operations} \quad (28)$$

$$\text{Required rate} = 181 \times 1,096 \times 20,204 = 4,007,988,704 \text{ ops/sec} \approx 4.01 \text{ GOPS} \quad (29)$$

Breaking down by operation type:

- Multiplications: $181 \text{ fps} \times 1,096 \text{ conv/frame} \times 10,000 \text{ mults/conv} = 1,983,760,000 \text{ mults/sec} \approx 1.98 \text{ billion mults/sec}$
- Additions: $181 \text{ fps} \times 1,096 \text{ conv/frame} \times 10,204 \text{ adds/conv} = 2,024,228,704 \text{ adds/sec} \approx 2.02 \text{ billion adds/sec}$
- Total: $\approx 4.01 \text{ GOPS}$

This represents the minimum computational throughput required to keep up with incoming frames at 181 fps, assuming serial sequential processing of each convolution.

Converting to clock-based requirements at 100 MHz operation:

$$\text{Clock frequency} = 100 \text{ MHz} = 100,000,000 \text{ cycles/sec} \quad (30)$$

$$\text{Cycles available per frame} = \frac{100,000,000 \text{ cycles/sec}}{181 \text{ frames/sec}} = 552,486 \text{ cycles/frame} \quad (31)$$

$$\text{Required throughput} = \frac{22,143,584 \text{ operations}}{552,486 \text{ cycles}} = 40.1 \text{ operations per clock} \quad (32)$$

Convolution Engine Capability:

The parallel pipelined convolution engine with its 10 convolution blocks and 60 parallel multipliers provides the computational throughput necessary to meet these requirements. Each of the 10 blocks processes a complete convolution position concurrently, with the 60 multipliers (12 per slice $\times$ 5 slices) operating in parallel within each block. The hierarchical adder trees and pipelined accumulation stages allow continuous data flow through the computation pipeline.

However, the stride-5 operation with partial accumulation means only approximately 5 out of 10 convolution blocks are actively utilized at any given time due to data reuse patterns. This results in 50% effective block utilization, providing an effective 5 $\times$ throughput instead of a direct 10 $\times$ throughput from the 10 blocks.

The 5 effectively utilized blocks provide a 5 $\times$ speedup over serial processing. Given the required throughput of 40.1 ops/clock, the parallel architecture achieves:

$$\text{Parallel throughput} = 40.1 \text{ ops/clock} \times 5 = 200.5 \text{ ops/clock} \approx 200 \text{ ops/clock} \quad (33)$$

At 100 MHz operation, this yields:

$$\text{Engine capability} = 200 \text{ ops/clock} \times 100,000,000 \text{ cycles/sec} = 20,000,000,000 \text{ ops/sec} \approx 20 \text{ GOPS} \quad (34)$$

This computational capability (20 GOPS) exceeds the required throughput (4.01 GOPS) by approximately 5 $\times$, providing substantial margin for real-time operation. The actual achieved performance depends on the memory bandwidth and data delivery rates, which are analyzed in the performance estimation section below.

Output Data Generated per Frame:

$$D_{\text{output}} = 1,096 \text{ results} \times 45 \text{ bits} = 49,320 \text{ bits} = 49.3 \text{ Kb} \quad (35)$$

Frame-Level Computational Intensity:

With register reuse (best case):

$$I_{frame, reuse} = \frac{22,143,584 \text{ operations}}{66.0 \text{ Mb}} = 335.5 \text{ operations per Kb} \quad (36)$$

Without reuse (worst case):

$$I_{frame, no_reuse} = \frac{22,143,584 \text{ operations}}{171.1 \text{ Mb}} = 129.4 \text{ operations per Kb} \quad (37)$$

The actual intensity depends on the hardware implementation:

- **With register reuse:** 335.5 ops/Kb ($5\times$ higher than per-convolution due to overlapping window reuse and stride-5 kernel reuse)
- **Without reuse:** 129.4 ops/Kb ($1.9\times$ higher than per-convolution due to stride-5 kernel reuse only)
- **Per-convolution:** 67.3 ops/Kb (baseline, treating each convolution independently)

The difference between frame-level and per-convolution intensity quantifies the benefit of processing multiple convolutions together rather than individually. The per-convolution metric treats each convolution as completely independent, while the frame-level metrics account for data sharing across adjacent convolutions.

Performance Estimation at 100 MHz

Operating at a 100 MHz clock frequency, the accelerator's real-time performance is determined by comparing the parallel memory readout rate to the serial frame input rate.

Real-Time Constraint:

The accelerator must complete all convolution operations on the current 5-frame datacube before the next scanline finishes being captured into the circular buffer. New scanlines arrive serially from the processing pipeline (after binning and other preprocessing stages). Each scanline contains:

$$N_{scanline} = 400 \times 1100 = 440,000 \text{ values} \quad (38)$$

The input arrives at a rate of 1 value per clock. At 100 MHz, capturing a complete scanline serially requires:

$$T_{input} = 440,000 \text{ cycles} = 4.4 \text{ ms} \quad (39)$$

The accelerator must process all 1,096 convolutions on the 5-frame circular buffer before these 4.4 ms elapse.

Parallel Memory Architecture Advantage:

The key to meeting real-time performance lies in the memory packing and readout strategy. Frame data is packed 6 values per 72-bit row, and the 2:3 read ratio between frame and kernel memory means that every 3 clocks deliver 2 frame rows and 3 kernel rows, yielding 12 values total. This gives an effective readout rate of:

$$\text{Readout rate} = \frac{12 \text{ values}}{3 \text{ clocks}} = 4 \text{ values/clock} \quad (40)$$

Compared to the serial input rate of 1 value/clock, the accelerator reads data $4\times$ faster than it arrives.

Processing Throughput Analysis:

The accelerator must read through the circular buffer datacube to complete all 1,096 convolutions. Since the memory readout rate is $4\times$ faster than the serial input rate (4 values/clock vs 1 value/clock), processing the datacube takes:

$$T_{process} = \frac{T_{input}}{4} = \frac{440,000 \text{ cycles}}{4} = 110,000 \text{ cycles} = 1.1 \text{ ms} \quad (41)$$

This $4\times$ speedup comes directly from the memory packing strategy (6 values per row) and the 2:3 read ratio delivering 12 values every 3 clocks.

Performance Margin:

Comparing processing time to input capture time:

$$\text{Timing ratio} = \frac{T_{process}}{T_{input}} = \frac{110,000}{440,000} = 0.25 = 25\% \quad (42)$$

The accelerator completes all convolutions in 25% of the time required to capture the next scanline. This provides a timing margin of:

$$\text{Timing margin} = \frac{T_{input} - T_{process}}{T_{input}} \times 100\% = \frac{440,000 - 110,000}{440,000} \times 100\% = 75\% \quad (43)$$

Data Throughput:

The memory readout rate is determined by the packing strategy:

$$\text{Processing throughput} = 4 \text{ values/clock} \times 12 \text{ bits/value} = 48 \text{ bits/clock} \quad (44)$$

While the input captures one scanline serially:

$$\text{Input throughput} = 1 \text{ value/clock} \times 12 \text{ bits/value} = 12 \text{ bits/clock} \quad (45)$$

The memory packing strategy delivers a 4× throughput advantage (48 bits/clock vs 12 bits/clock), enabling the accelerator to process convolutions faster than new scanlines arrive.

Throughput Summary:

- **Serial input capture time:** 4.4 ms per scanline (440,000 cycles)
- **Parallel processing time:** 1.1 ms per frame (110,000 cycles)
- **Timing margin:** 75% (processing completes in 25% of input period)
- **Effective frame rate:** 227 fps (limited by 4.4 ms input capture period)
- **Target frame rate:** 181 fps
- **Processing throughput:** 48 bits/clock (4 values/clock via parallel memory readout)

- Input throughput: 12 bits/clock (1 value/clock serial capture)

The effective frame rate of 227 fps exceeds the target frame rate of 181 fps. The design is **input-bound**: the memory packing strategy (6 values per row) and 2:3 read ratio enable processing to complete in only 25% of the scanline capture period, ensuring robust real-time operation with a 75% timing margin. This validates that the three pipeline stages (Memory Controller, Data Slicer/Unpacker, and Convolution Engine) successfully meet real-time processing requirements.

Resource Estimation

The hardware accelerator’s FPGA resource utilization was estimated based on the Simulink design and VHDL generation reports. Table 5 summarizes the estimated resources required for the complete accelerator implementation on the target Zynq UltraScale+ MPSoC device.

Table 5: FPGA Resource Estimation for Hardware Accelerator on XCZU15EG

Resource Type	UltraRAM	Block RAM	DSPs	LUTs	Registers
Available (XCZU15EG)	109*	726	3,528	930,560	1,861,120
Used (Accelerator)	100**	30	602	26,061	61,394
Utilization (%)	91%	4%	17%	3%	3%

* Calculated from 31.5 Mb UltraRAM / 0.288 Mb per block (4K×72 bits)

** UltraRAM count manually calculated: 95 blocks for frame storage (5 frames × 19 blocks/frame) + 5 blocks for kernel weights.

The resource breakdown reflects the design’s emphasis on compute-intensive operations:

- **UltraRAM (100 blocks):** Stores the circular buffer frame data (95 blocks for 5 frames × 19 blocks/frame) and kernel weights (5 blocks), with each block configured as 4K×72 bits.
- **Block RAM (30):** Used for smaller buffers, control logic storage, and intermediate result staging.
- **DSP Slices (602):** Implements the 60 parallel multipliers (12 per slice × 5 slices) and associated adder trees, representing the core computational resources.
- **LUTs (26,061):** Provides the control logic, address generation, data unpacking, multiplexing, and pipeline management.

- **Registers (61,394):** Implements pipeline stages, accumulators, state machines, and data buffering throughout the accelerator.

These resource estimates are well within the capabilities of the XCZU15EG Zynq UltraScale+ MPSoC. Resource utilization as percentage of available XCZU15EG resources:

- UltraRAM: $28.8 \text{ Mb} / 31.5 \text{ Mb} = 91\%$
- Block RAM: $1.08 \text{ Mb} / 26.2 \text{ Mb} = 4\%$
- DSP Slices: $602 / 3,528 = 17\%$
- LUTs: $26,061 / 930,560 = 3\%$
- Registers: $61,394 / 1,861,120 = 3\%$

The design is usable on the Mercury+ XU1 as the target hardware platform, with UltraRAM being the primary resource constraint at 91% utilization.

Validation and Verification

This section describes the methods used to verify correct operation of the hardware accelerator. It covers the MATLAB simulation framework for functional testing, VHDL code generation and synthesis processes, and integration considerations with the Processing System.

MATLAB Simulation Framework

The primary validation approach uses MATLAB to generate test vectors, simulate the Simulink model, and compare results against a reference software implementation. The test framework, implemented in the `Kernel_Setup_Script.m` file, performs a validation workflow that verifies the hardware design produces correct results.

Test Vector Generation The test script begins by generating input data and kernel weights according to the specified system dimensions ($1100 \times 400 \times 36$ for input, $5 \times 400 \times 5$ for kernel). The script supports both random and structured test patterns:

- **Random Test Vectors:** Input values are generated using MATLAB's `randi` function, producing values in the range `[1, 4095]` for 12-bit input data and `[1, 10]` for 18-bit kernel weights. This tests the hardware's ability to handle arbitrary data patterns.
- **Structured Test Vectors:** For debugging purposes, the script can generate incrementing patterns where each value is computed as: `value = initial_value + (index * increment) * scalar`. This allows manual verification of intermediate results.

Listing 12 shows the test vector generation code from the `Kernel_Setup_Script.m` file:

Kernel_Setup_Script.m

```

274  if input_data_random == true
275      flattened_input_array = randi([1, (2^ScanlinePixelBitSizeController)-1],
          x_length_input * y_height_input * z_depth_input, 1);
276  elseif input_data_random == false
277      % Generate array with continuous incremental values that wrap at max
278      num_elements = x_length_input * y_height_input * z_depth_input;
279      continuous_values = input_value + (0:num_elements-1)' * input_increment *
          input_scalar;
280      % Wrap around max value (keeping range from input_value to input_max_value)
281      flattened_input_array = mod(continuous_values - input_value, input_max_value)
          + input_value;
282  end
283
284  %create a flat kernel array
285  if kernel_data_random == true
286      flattened_kernel_array = randi([1, 10], x_length_kernel * y_height_kernel *
          z_depth_kernel, 1);
287  elseif kernel_data_random == false
288      % Generate array with incremental values and scalar per z-slice
289      flattened_kernel_array = zeros(x_length_kernel * y_height_kernel *
          z_depth_kernel, 1);
290      slice_size = x_length_kernel * y_height_kernel;
291
292      for i = 1:z_depth_kernel

```

```

293         start_idx = (i-1) * slice_size + 1;
294         end_idx = i * slice_size;
295         slice_data = kernel_value + (0:slice_size-1)' * kernel_increment;
296         flattened_kernel_array(start_idx:end_idx) = slice_data * (kernel_scalar *
                i);
297     end
298 end

```

Code Listing 12: Test Vector Generation (Random and Structured)

The generated 3D arrays are then flattened into 1D arrays following the specific serialization order required by the hardware. For input data, values are serialized in Y→X→Z order (all spectral bins for a position, then all spatial positions, then depth slices). For kernel data, the serialization follows Y→Z→X order to align with the hardware’s data access patterns.

Timeseries Creation for Simulink Integration Once the test vectors are generated, the script creates MATLAB timeseries objects that interface with the Simulink model. This is accomplished through the following steps:

1. **Data Array Creation:** The flattened 1D arrays are converted into column vectors suitable for timeseries objects.
2. **Valid Signal Generation:** Boolean arrays of the same length are created, with all values set to `true` to indicate continuous valid data. This simulates the worst-case scenario where data arrives on every clock cycle.
3. **Timeseries Object Construction:** MATLAB’s `timeseries()` function creates objects from the data and valid arrays, as shown in Listing 13:
4. **Workspace Import:** These timeseries objects are stored in the MATLAB workspace where the Simulink model’s `From Workspace` blocks can access them during simulation.

Kernel_Setup_Script.m

```

300  %create a boolean array for both the input and kernel data so that simulink knows
      which data is valid
301  flattened_input_bool_array = true(x_length_input * y_height_input * z_depth_input,
      1);
302  flattened_kernel_bool_array = true(x_length_kernel * y_height_kernel *
      z_depth_kernel, 1);
303
304  %create a timeseries for both the input and kernel data so that simulink knows
      which data is valid and the valid lines
305  input_timeseries = timeseries(flattened_input_array);
306  input_valid_timeseries = timeseries(flattened_input_bool_array);
307  kernel_timeseries = timeseries(flattened_kernel_array);
308  kernel_valid_timeseries = timeseries(flattened_kernel_bool_array);

```

Code Listing 13: Timeseries Object Creation for Simulink

Simulink Simulation Execution The Simulink model is configured to run for a duration sufficient to process all input data. During simulation, the **From Workspace** blocks read the timeseries data and feed it into the hardware model at each time step. The model processes the data through the complete pipeline: memory packing, storage in circular buffers, unpacking, and convolution computation.

Output Capture Using Enabled Subsystem A critical component of the validation framework is the output capture mechanism, implemented using a Simulink Enabled Subsystem. This subsystem serves several purposes:

1. **Valid Output Detection:** The enabled subsystem is triggered only when the hardware's output valid signal is asserted, ensuring that only legitimate results are captured and not intermediate pipeline values or invalid data.
2. **Data Ordering:** As convolution results are produced, they are written to the subsystem's output port in the order they emerge from the hardware. This maintains the correct spatial

ordering (first position to last position) for comparison.

3. **MATLAB Workspace Export:** The captured output is logged to a To Workspace block, creating a variable (typically named `simout_kernel_values_out`) that can be accessed after simulation completes.

The enabled subsystem architecture ensures that the output array length matches the expected number of results (1096 for the complete output feature map) and that values appear in the correct sequence.

Reference Calculation Methods The test script computes reference convolution results using three independent methods to verify correctness:

- **Method 1 - Full Calculation:** Computes the complete convolution at each output position independently by extracting the appropriate $5 \times 400 \times 5$ region from the input, performing element-wise multiplication with the kernel, and summing all products. This serves as the primary reference implementation.
- **Method 2 - Sliding Window:** Uses incremental computation by tracking contributions from individual input X slices. This method maintains a 2D array (`slice_contributions_array`) that records how each input slice contributes to each output position, mimicking the hardware's sliding buffer approach and allowing verification of the circular buffer logic.
- **Method 3 - MATLAB `convn`:** Uses the built-in `convn()` function with 'valid' mode. Since the hardware expects pre-flipped kernels, the test kernel is flipped three times (along X, Y, and Z dimensions) before being passed to `convn()`, which will flip it back internally.

Listing 14 shows the Method 1 implementation:

Kernel_Setup_Script.m

```

431 % Method 1: Full calculation at each position
432 for z_pos = 1:num_positions_z
433     for x_pos = 1:num_positions_x
434         % Extract the region from input_3D_array that matches kernel size
435         input_region = input_3D_array(x_pos:x_pos+x_limit-1, :, z_pos:z_pos+
            z_limit-1);
436
437         % Element-wise multiplication
438         product = input_region .* kernel_3D_array_limited;
439
440         % Sum all products to get convolution result at this position
441         convolution_result_full(x_pos, z_pos) = sum(product(:));
442     end
443 end

```

Code Listing 14: Method 1: Full Convolution Calculation

All three methods produce identical results (within floating-point precision), confirming the correctness of the approach. Listing 15 shows the convn-based reference calculation and verification logic:

Kernel_Setup_Script.m

```

481 % Method 3: Using convn (kernel is pre-flipped, so flip it back)
482 % convn flips the kernel, but our kernel is already pre-flipped, so we need to
    flip it again
483 kernel_flipped = flip(flip(flip(kernel_3D_array_limited, 1), 2), 3);
484 convolution_result_convn = convn(input_3D_array, kernel_flipped, 'valid');
485
486 % Verify all methods agree
487 max_diff_full_sliding = max(abs(convolution_result_full(:) -
    convolution_result_sliding(:)));
488 max_diff_full_convn = max(abs(convolution_result_full(:) -
    convolution_result_convn(:)));
489 fprintf('Max difference between full and sliding: %e\n', max_diff_full_sliding);

```

```

490     fprintf('Max difference between full and convn: %e\n', max_diff_full_convn);
491
492     convolution_result_convn_flatten = convolution_result_convn(:);
493     % Use this to compare after running the simulation
494     % mismatch = find(out.simout_kernel_values_out ~= convolution_result_convn_flatten
        );

```

Code Listing 15: Reference Calculation using convn and Method Verification

Verification Results The hardware accelerator was validated through comprehensive testing using 36 frame scanlines of hyperspectral data, configured with the full system dimensions (1100 spatial positions $\times$ 400 spectral bins $\times$ 36 acquisition frames). The validation was performed 5 times with different randomly generated input data to ensure consistent operation across diverse test patterns. All MATLAB simulations were executed at a 100 MHz clock frequency. This test represents a complete end-to-end validation of the entire pipeline, including:

- Memory packing and storage in the circular buffer (95 UltraRAM blocks with custom addressing + 5 for kernel weights)
- Data unpacking and alignment (2:3 read ratio between frame and kernel memory)
- Convolution computation (pipelined MAC architecture with hierarchical adder trees)
- Stride-5 operation with partial accumulation
- Output result generation with valid signal timing

The hardware simulation results were compared against all three reference calculation methods (full calculation, sliding window, and MATLAB convn). **All 5 test runs passed with exact matches between the hardware output and reference values**, confirming that the accelerator operates correctly across different input data patterns. Specifically:

- All 1,096 convolution outputs per frame matched the expected results

- The circular buffer correctly maintained data across all 36 acquisition frames
- Memory addressing and data retrieval operated without errors
- The stride-5 accumulation logic produced correct partial and final sums
- Output valid signals asserted at the correct clock cycles

These results demonstrate that the hardware implementation correctly realizes the intended convolution operation and is ready for FPGA deployment. The verification framework provides confidence that the accelerator will process real hyperspectral data accurately in the target application.

SOFTWARE: EMBEDDED LINUX FOR SOC FPGAS

This chapter describes the implementation of the operating system and the build pipeline. It covers the creation and configuration of recipes, layers, and the overall build process. The primary goal is to provide an outline of how to build a customized Linux based operating system suitable for a System-on-Chip (SoC) Field-Programmable Gate Array (FPGA) or any general embedded device that can benefit from running a Linux based OS. This discussion covers the main concepts, tools, and processes offered by the Yocto Project, an open-source project that provides templates, tools, and methods to create custom Linux based systems for embedded products regardless of the hardware architecture. Additionally, this chapter describes the implementation of the operating system on the DE10-Nano board, which is based on Altera's Cyclone V SoC FPGA. The DE10-Nano was selected for this project due to the already existing audio mini project [56] which contains completed and functioning hardware, drivers, and software. This focuses the work on the yocto build pipeline. This involves using the Yocto Project to create a custom Linux distribution that creates a DE10-Nano image that boots the board and loads the audio mini drivers automatically.

Yocto and OpenEmbedded

OpenEmbedded is the core portion of the Yocto pipeline. It is responsible for defining the entire build pipeline that Yocto utilizes. This pipeline can be seen in Figure 85. While OpenEmbedded is intertwined with Yocto, they are two independently managed projects. Yocto is reliant on OpenEmbedded to function and cannot function without it, while OpenEmbedded can function without Yocto.

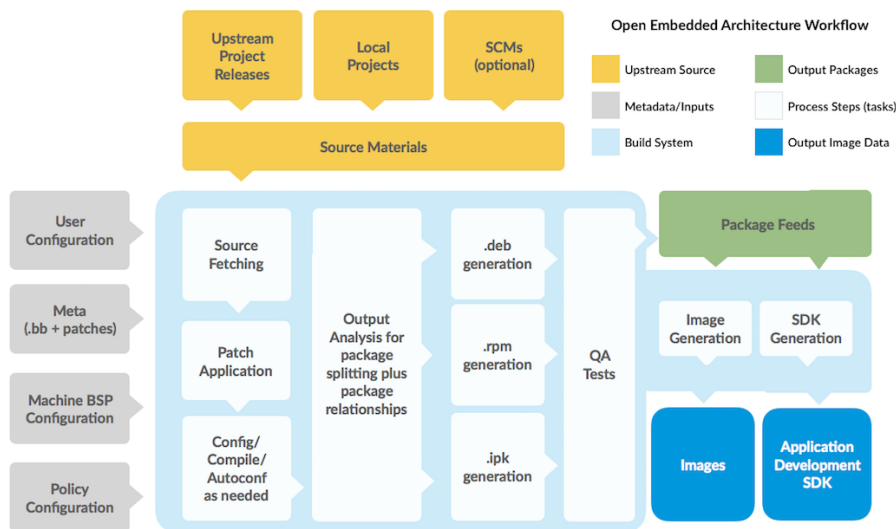


Figure 85: Yocto Project How It Works [45].

Yocto, OpenEmbedded, and Bitbake Concepts

OpenEmbedded is also responsible for the main tool BitBake. BitBake is a Python-based generic task executor. This tool is responsible for parsing and acting upon the final metadata to create the desired output, which could be simple compilation of a software package or the creation of a bootable image. BitBake defines the layering concept of Yocto by utilizing the recipes, configuration files, classes, layers, and append files to create the desired output. Its primary functions are cross-compilation, dependency resolution, and source code management while being as operating-system agnostic as possible [44].

Layers and Directory Structure

Layers are the organizational framework of the Yocto Project. Each layer is a directory containing related metadata—recipes, classes, configuration files, and machine definitions. Layers allow modular development by separating hardware specific, distribution specific, and application specific code into distinct, reusable components.

A layer's name conventionally begins with the prefix `meta-`, known as a "meta-layer." This

naming convention identifies the directory as a Yocto layer and distinguishes it from regular directories. For example, `meta-poky` provides the Poky reference distribution, `meta-openembedded` offers additional recipes, and custom layers like `meta-msu-de10-nano` contain project specific customizations for the DE10-Nano board.

Layers stack on top of each other, with higher priority layers overriding lower priority ones. This stacking mechanism allows customization without modifying base layers. When BitBake searches for recipes or configuration files, it traverses layers according to their priority, using the first match found. A layer's priority is defined in its `conf/layer.conf` file using the `BBFILE.PRIORITY` variable (line 5 in Listing 28).

Common layer types include base layers (providing core functionality), BSP layers (board support packages for specific hardware), and distribution layers (defining system policies). Each layer contains a standardized directory structure: `recipes-*` directories for recipes, `classes/` for shared functionality, and `conf/` for configuration files. This structure allows BitBake to discover and integrate metadata across layers.

meta-msu-de10-nano Layer Structure The `meta-msu-de10-nano` layer, available on GitHub [9], shows how to create a complete BSP layer structure for the DE10-Nano board for a specific project. The layer organizes hardware specific metadata, kernel configurations, bootloader customizations, device drivers, and custom images. The complete directory structure is shown below in Listing 16 and in the Appendix A.

meta-msu-de10-nano Layer

```
meta-msu-de10-nano/
|-- conf/
|   |-- layer.conf                # Layer configuration and priority
|   +-- machine/
|       +-- de10-nano-audio-mini.conf # Machine-specific settings
|-- classes/
|   +-- audio-mini-image.bbclass    # Reusable image class
```

```

|-- recipes-audio-mini/
|   |-- Audio-Mini-Drivers/
|   |   +-- audiomini-drivers.bb      # Systemd service recipe
|   |-- AD1939-Audio-Codec-KernelModule/
|   |   +-- ad1939-audiomini.bb      # Audio codec driver
|   +-- TPA6130A2-Headphone-Amplifier-KernelModule/
|       +-- tpa6130a2-audiomini.bb    # Headphone amp driver
|-- recipes-bsp/
|   |-- audio-mini-bitstream/
|   |   |-- audio-mini-bitstream.bb  # FPGA bitstream recipe
|   |   +-- files/
|   |       +-- soc_system.rbf        # FPGA configuration
|   |-- device-tree/
|   |   |-- de10-nano-audio-mini-devicetree.bb
|   |   +-- files/
|   |       |-- de10-nano-audio-mini-base.dts
|   |       +-- de10-nano-audio-mini-base.dtsi
|   +-- u-boot/
|       |-- u-boot-socfpga_%.bbappend # U-Boot customization
|       +-- files/
|           |-- de10_nano_audio_mini_sd_defconfig
|           +-- de10_nano_audio_mini_tftp_nfs_defconfig
|-- recipes-core/
|   +-- images/
|       +-- audio-mini-passthrough.bb # Custom image recipe
|-- recipes-kernel/
|   +-- linux/
|       |-- linux-socfpga-lts_%.bbappend
|       +-- files/
|           +-- de10nano-fragment.cfg # Kernel configuration
|-- wic/
|   |-- de10-nano-sd-build.wks        # SD card image layout
|   +-- de10-nano-tftp-nfs-build.wks  # Network boot layout
+-- README.md                        # Layer documentation

```

Code Listing 16: meta-msu-de10-nano Complete Layer Structure

The layer follows Yocto's standardized naming conventions: `recipes-audio-mini/` groups

application-specific drivers, `recipes-bsp/` contains board support packages (bootloaders, device trees, FPGA bitstreams), `recipes-core/` holds system images, and `recipes-kernel/` manages kernel modifications. The `classes/` directory provides reusable functionality across recipes, while `conf/` establishes layer and machine configurations. The `wic/` directory contains WIC kickstart files that define partitioning schemes for bootable media.

Each recipe subdirectory contains a `.bb` file defining the build process and optional `files/` subdirectories for source code, patches, or configuration files. The `%.bbappend` files modify existing recipes from other layers without duplicating code. The layer also includes a nested sub-layer, `meta-my-audiomini-combfilter/`, with its own `conf/layer.conf`, showing how projects can organize related but distinct functionality hierarchically. The following subsections examine key components of this layer.

Recipes

Bitbake's highest level of metadata is the recipe file. This file is the highest level of metadata that bitbake will act upon. This file is denoted by the `.bb` extension. It also has a `.bbappend` extension which is used to append to the recipe file or layer on top of metadata that already exists in the recipe file. These recipes describe the build process for a range of software packages from system elements like the Linux kernel and bootloaders to user-space libraries, applications, or custom device drivers. Recipes are the building blocks of the Yocto Project, as they encapsulate the logic needed to produce binary packages that will be integrated into the final operating system image. Recipes are written in the BitBake language, a domain specific language that extends Python syntax with task-oriented constructs serving as the main set of instructions for fetching source code, building it, and installing the resulting artifacts. This layering mechanism is useful for customizing builds for specific devices, applications, or hardware variants while maintaining upstream compatibility and ease of updates.

All code examples in this chapter are drawn from the `meta-msu-de10-nano` layer, an open-source Board Support Package (BSP) layer developed for the DE10-Nano board with Audio Mini hardware [56]. This layer is publicly available on GitHub [9]. The examples are used to build

bootable Linux images for embedded audio processing applications.

The following is a `.bb` recipe file used to build and install device drivers for the DE10-Nano board, specifically to support an Audio Mini extension board. This recipe, named `audiomini-drivers.bb`, demonstrates how Yocto handles the integration of custom drivers with system services like `systemd`. The full recipe can be found in Appendix A. This recipe sets up a `systemd` service that automatically loads the audio mini drivers on the DE10-Nano board during boot. `Systemd` is an init system and service manager in modern Linux distributions, responsible for bootstrapping the user space and managing system processes (daemons) after the kernel has initialized [16]. By integrating with `systemd`, this recipe loads the necessary audio drivers at system startup without manual intervention.

The recipe's structure is broken down section by section:

Metadata and License Information These lines define metadata using variables such as `SUMMARY` (line 1), `DESCRIPTION` (line 2), `HOMEPAGE` (line 3), `BUGTRACKER` (line 4), `SECTION` (line 5), `LICENSE` (line 6), and `LIC_FILES_CHKSUM` (line 7), as shown in Listing 17. This information documents the recipe, categorizes it within the build system (e.g., under the "kernel" section via `SECTION = "kernel"`), and ensures compliance with licensing requirements. The `LIC_FILES_CHKSUM` variable points to a file whose checksum verifies the license integrity. This metadata also aids in package management and dependency resolution during the build.

```
audiomini-drivers.bb: 1-7

1  SUMMARY = "Linux I2C Device Driver for the TPA6130A2 Headphone Amplifier"
2  DESCRIPTION = "Linux I2C Device Driver for the TPA6130A2 Headphone Amplifier"
3  HOMEPAGE = "https://github.com/ADSD-SoC-FPGA"
4  BUGTRACKER = "https://github.com/ADSD-SoC-FPGA/Code/issues"
5  SECTION = "kernel"
6  LICENSE = "GPL-3.0-only"
7  LIC_FILES_CHKSUM = "file://${S}/audio-mini-drivers.service;md5=
    b07de196e86f9e826054228a95139d4c"
```

Code Listing 17: Metadata and License Information

Inheriting Classes Listing 18 shows the `inherit systemd` directive (line 9), which incorporates the `systemd.bbclass` file that provides predefined functions and variables for handling systemd services. Classes in Yocto are reusable modules of code stored in files with a `.bbclass` extension. They act like libraries or mixins, allowing recipes to inherit common functionality without duplication. For instance, the `systemd` class automates the installation and configuration of service files for managing daemons. A class file might define default variables, add build tasks (e.g., `do_install_append`), or export helper functions that can be used across multiple recipes. This inheritance mechanism provides code reuse, modularity, and consistency in the build process.

```
audiomini-drivers.bb: 9
```

```
9   inherit systemd
```

Code Listing 18: Inheriting Classes

Source Code Fetching The source code fetching configuration shown in Listing 19 uses the `SRC_URI` variable (line 11) to specify the location from which to fetch the source code—in this case, a Git repository on GitHub with a specific branch. The `protocol=https` determines https based fetching, while `SRCREV = "${AUTOREV}"` (line 12) automatically uses the latest commit from that repository. The `S` variable (line 14) sets the source directory to a subdirectory within the fetched Git repository, defining the working directory for subsequent build tasks. This configuration handles version control integration and allows reproducible builds by pinning to specific revisions if needed. You can also use a specific commit hash instead of the `AUTOREV` variable or use different packaging other than Git, such as a tarball or a zip file.

```
audiomini-drivers.bb: 11-14
```

```
11   SRC_URI = "git://github.com/night1rider/ADSD-SoC-FPGA-Code.git;protocol=https;
           branch=yocto-audio-mini-fixes"
12   SRCREV = "${AUTOREV}"
```

```

13
14  S = "${WORKDIR}/git/examples/passthrough/linux/systemd"

```

Code Listing 19: Source Code Fetching

Build Dependencies Listing 20 shows the `DEPENDS` variable (line 17), which lists packages required during the build process, such as `systemd` and `bash`. These dependencies ensure that the necessary tools and libraries are available when compiling or installing the recipe's components, preventing build failures due to missing prerequisites.

```

audiomini-drivers.bb: 17
17  DEPENDS += " systemd bash "

```

Code Listing 20: Build Dependencies

Runtime Dependencies Listing 21 demonstrates the `RDEPENDS:${PN}` variable (line 21), which specifies packages needed at runtime, including the specific audio drivers (`ad1939-audiomini` and `tpa6130a2-audiomini`), along with `systemd` and `bash`. This ensures that when the `audiomini-drivers` package is installed on the target system, all required components are present. The system will automatically install them when the package is installed.

```

audiomini-drivers.bb: 21
21  RDEPENDS:${PN} += " ad1939-audiomini tpa6130a2-audiomini systemd bash "

```

Code Listing 21: Runtime Dependencies

Provides Statement Listing 22 shows the `PROVIDES` variable (line 23), which indicates what virtual packages or names this recipe fulfills, so other recipes can depend on `audiomini-drivers` abstractly without specifying the exact package name. This means the `audiomini-drivers` package

can be used in other recipes without specifying the exact package name.

```
audiomini-drivers.bb: 23
23  PROVIDES = "audiomini-drivers"
```

Code Listing 22: Provides Statement

Installation Task Listing 23 shows the `do_install()` function (lines 25-34), which overrides the default installation behavior to copy specific files (the systemd service file and a shell script) to appropriate directories on the target filesystem. It uses `install` commands (lines 27-32) to create directories and set permissions, allowing the files to be placed correctly for systemd to recognize and use.

```
audiomini-drivers.bb: 25-34
25  do_install() {
26      # Install the systemd service file
27      install -d ${D}/etc/systemd/system
28      install -d ${D}/usr/local/bin
29
30      # Install the systemd service file and the script to load the drivers via
        insmod
31      install -m 0644 ${S}/audio-mini-drivers.service ${D}/etc/systemd/system/
32      install -m 0755 ${S}/load-audio-mini-drivers.sh ${D}/usr/local/bin/
33
34  }
```

Code Listing 23: Installation Task

Installed Files Tracking Listing 24 demonstrates the `FILES:${PN}` variable (line 37), which explicitly lists the files installed by this recipe, so Yocto can package them correctly and track them for dependency management and image generation.

```
audiomini-drivers.bb: 37

37  FILES:${PN} += "/etc/systemd/system/audio-mini-drivers.service /usr/local/bin/load
    -audio-mini-drivers.sh"
```

Code Listing 24: Installed Files Tracking

Systemd Configuration Listing 25 shows how to configure the systemd service by specifying the service name via `SYSTEMD_SERVICE:${PN}` (line 40) and enabling it automatically on boot via `SYSTEMD_AUTO_ENABLE` (line 41). This uses the inherited `systemd` class to handle service management without additional custom code so that the service is automatically started on boot. This is useful because it ensures that we can set a service using a guided method instead of having to write custom code to start the service and potentially cause issues with the build or other services.

```
audiomini-drivers.bb: 40-41

40  SYSTEMD_SERVICE:${PN} += " audio-mini-drivers.service"
41  SYSTEMD_AUTO_ENABLE:${PN} = "enable"
```

Code Listing 25: Systemd Configuration

To layer on top of functional elements of this recipe, add a `.bbappend` file. This file would be named `audiomini-drivers.bbappend` and would be placed in a directory that is in Yocto's layer structure. This overrides the original recipe without modifying the original recipe file, adding additional features or changing the behavior of the recipe without modifying the original recipe file.

The `meta-msu-de10-nano` layer includes `.bbappend` files that demonstrate this layering mechanism. Listing 26 shows how the `linux-socfpga-%.bbappend` file extends the kernel recipe to add a custom kernel configuration fragment and implement TFTP deployment:

```
linux-socfpga_%.bbappend: 1-6

1  # Get Directory Path
2  BBDIR := "${@os.path.dirname(d.getVar('FILE', True))}"
3
4  FILESEXPATHS:prepend := "${BBDIR}/files:"
5  SRC_URI += "file://de10nano-fragment.cfg"
```

Code Listing 26: Example: Kernel Configuration bbappend

This append file uses Python inline code to dynamically determine the directory path via BBDIR (line 2), extends the FILESEXPATHS (line 4) to include the layer's `files/` directory, and adds a custom configuration fragment to SRC_URI (line 5). The `%` wildcard in the filename means this append applies to any version of the `linux-socfpga` recipe, ensuring compatibility across kernel updates.

Listing 27 provides another example from the nested `meta-my-audiomini-combfilter` sub-layer, demonstrating how to override device tree sources and append to configuration tasks:

```
de10-nano-audio-mini-devicetree.bbappend: 1-8

1  BBDIR_APP := "${@os.path.dirname(d.getVar('FILE', True))}"
2
3  FILESEXPATHS:prepend := "${BBDIR_APP}/files:"
4
5  SRC_URI += "file://de10nano-audiomini-combfilter.dts"
6
7  #override the default DT_FILES variable
8  DT_FILES = " de10nano-audiomini-combfilter.dts"
```

Code Listing 27: Example: Device Tree bbappend

This example extends FILESEXPATHS (line 3), adds device tree sources to SRC_URI (line 5), overrides the DT_FILES variable (line 8), and appends to an existing task via `do_configure:append()` (line 10). These `.bbappend` files only need to contain the changes to the original recipe, showing the layering concept where higher priority layers can selectively modify lower priority recipes

without code duplication. This pattern separates base board support and application specific customizations, as seen with the nested sub-layer structure.

Configuration Files

Configuration files control the behavior of the build system by setting global variables that affect how BitBake parses recipes, resolves dependencies, and generates images. These files use the `.conf` extension and reside in `conf/` directories within layers. Unlike recipes that define individual software packages, configuration files establish build policies such as target architecture, toolchain settings, and distro features.

The primary configuration file is `bblayers.conf`, located in the build directory's `conf/` subdirectory. It lists all layers included in the build via the `BBLAYERS` variable, defining the search path for recipes and classes. BitBake processes layers in the order listed, with priority determined by each layer's `BBFILE_PRIORITY` value.

Another file is `local.conf`, also in the build directory. It specifies machine specific settings through the `MACHINE` variable (e.g., `MACHINE = "de10-nano-audio-mini"`), distribution features via `DISTRO_FEATURES`, and package formats using `PACKAGE_CLASSES`. Developers use `local.conf` to customize builds without modifying layer metadata for different deployment scenarios.

Layer Configuration Layer specific configuration occurs in `conf/layer.conf` within each meta-layer. This file defines the layer's priority, recipe paths using `BBFILES`, and any layer dependencies via `LAYERDEPENDS`. Listing 28 shows the `meta-msu-de10-nano` layer configuration:

layer.conf: 1-7

```

1  BBPATH .= ":${LAYERDIR}"
2  BBFILES += "${LAYERDIR}/recipes-*/*/*.bb ${LAYERDIR}/recipes-*/*/*.bbappend"
3  BBFILE_COLLECTIONS += "msu-de10-nano"
4  BBFILE_PATTERN_msu-de10-nano = "^${LAYERDIR}/"
5  BBFILE_PRIORITY_msu-de10-nano = "6"
6  LAYERDEPENDS_msu-de10-nano = "core meta-intel-fpga"
```

```
7  LAYERSERIES_COMPAT_msu-de10-nano = "scarthgap"
```

Code Listing 28: Layer Configuration: layer.conf

This configuration file extends the BitBake search path (BBPATH, line 1) to include the layer directory, defines recipe search patterns (BBFILES, line 2) to discover all `.bb` and `.bbappend` files in `recipes-*` subdirectories, and creates a unique collection identifier (BBFILE_COLLECTIONS, line 3). The priority value of 6 (BBFILE_PRIORITY_msu-de10-nano = "6", line 5) ensures this layer's recipes override those from lower-priority layers. Layer dependencies (LAYERDEPENDS, line 6) specify that this layer requires the `core` and `meta-intel-fpga` layers, establishing a clear dependency chain. The `LAYERSERIES_COMPAT` variable (line 7) declares compatibility with the Scarthgap release series, which enforces version alignment across the build environment. That means this can only be used with the Scarthgap release of Yocto.

Machine Configuration Machine definitions reside in `conf/machine/` directories within layers. These files detail hardware specific settings: bootloader type, kernel version, device tree configuration, serial console parameters, and boot methods. The `de10-nano-audio-mini.conf` file defines a machine configuration for the DE10-Nano board with Audio Mini hardware [56]. The full configuration can be found in Appendix A. Key sections include:

Listing 29 shows the machine type and includes that define the foundation by inheriting SoC FPGA base settings and filesystem support:

```
de10-nano-audio-mini.conf: 1-7
```

```
1  #@TYPE: Machine
2  #@NAME: de10-nano-audio-mini
3  #@DESCRIPTION: Machine configuration for the DE10 Nano with Audio Mini SD Card
    Boot"
4
5  # Include base SOCFPGA configuration and UBIFS filesystem support
```

```

6  require conf/machine/include/socfpga.inc
7  require conf/machine/include/ubifs.inc

```

Code Listing 29: Machine Type and Includes

Listing 30 shows deployment and network configuration that establishes and production or development boot options (SD card or TFTP/NFS network boot) and network parameters for development:

```
de10-nano-audio-mini.conf: 9-24
```

```

9  # Setup Configuration
10 # Default deployment method - can be "sd" for SD card or "tftp-nfs" for network
    boot
11 DE10_NANO_DEPLOY_CONFIG ??= "sd"
12 # Combine machine name with deployment config to create unique configuration
    identifier
13 DE10_NANO_CONFIG ??= "${MACHINE}-${DE10_NANO_DEPLOY_CONFIG}"
14
15 DE10_NANO_STATIC_IP  ?= "192.168.2.20"
16 DE10_NANO_GATEWAY    ?= "0.0.0.0"
17 DE10_NANO_MASK       ?= "255.255.255.0"
18 DE10_NANO_ETH_ADAPTER ?= "eth0"
19 DE10_NANO_DHCP_STATUS ?= "off"
20
21 DE10_NANO_NFS_IP     ?= "192.168.2.10"
22 DE10_NANO_NFS_PORT   ?= "2049"
23 DE10_NANO_NFS_DIR    ?= "/srv/nfs/shared/de10nano"
24 DE10_NANO_TFTP_DIR   ?= "/srv/tftp"

```

Code Listing 30: Deployment and Network Configuration

Listing 31 shows U-Boot and bootloader configuration that maps deployment methods to U-Boot defconfig files, enabling multiple boot strategies from a single machine definition:

```
de10-nano-audio-mini.conf: 45-52
```

```

45  # U-Boot Configuration
46  # Set the U-Boot configuration based on the deployment method
47  UBOOT_CONFIG ??= "${DE10_NANO_CONFIG}"
48  # U-Boot defconfig for SD card boot method
49  UBOOT_CONFIG[de10-nano-audio-mini-sd] = "de10_nano_audio_mini_sd_defconfig"
50  # U-Boot defconfig for TFTP/NFS network boot method
51  UBOOT_CONFIG[de10-nano-audio-mini-tftp-nfs] = "
    de10_nano_audio_mini_tftp_nfs_defconfig"
52  UBOOT_CONFIG[de10-nano-audio-mini-custom] = "${CUSTOM_UBOOT_CONFIG}"

```

Code Listing 31: U-Boot Configuration

Listing 32 illustrates kernel and provider preferences that specify which kernel version and bootloader to use, ensuring reproducible builds:

```
de10-nano-audio-mini.conf: 55-67
```

```

55  # Kernel and bootloader preferences
56  # Use the SOCFPGA-specific long-term support kernel
57  PREFERRED_PROVIDER_virtual/kernel = "linux-socfpga-lts"
58  # Prefer kernel version 6.6.x (% allows any patch version)
59  PREFERRED_VERSION_linux-socfpga-lts = "6.6%"
60  # Use U-Boot as the bootloader
61  PREFERRED_PROVIDER_virtual/bootloader = "u-boot-socfpga"
62  PREFERRED_PROVIDER_virtual/dtb = "de10-nano-audio-mini-devicetree"
63  PREFERRED_PROVIDER_virtual/dtbs = "de10-nano-audio-mini-devicetree"
64
65  # System configuration
66  # Use systemd as the init system (alternative would be SysVinit)
67  INIT_MANAGER ??= "systemd"

```

Code Listing 32: Kernel and Bootloader Preferences

Listing 33 details serial console and ExtLinux configuration that defines boot menu entries and console parameters so that we will correctly boot and push logging out to the expected serial terminal of the device.

```
de10-nano-audio-mini.conf: 77-95
```

```

77  # Serial Console Configuration
78  # Configure serial console at 115200 baud rate on ttyS0 (UART0)
79  SERIAL_CONSOLES ?= "115200;ttyS0"
80
81  # U-Boot ExtLinux Configuration
82  # Enable U-Boot ExtLinux support for boot menu
83  UBOOT_EXTLINUX ?= "1"
84  # Define available boot labels (menu entries)
85  UBOOT_EXTLINUX_LABELS ?= "default"
86  # Set the default boot label name
87  UBOOT_EXTLINUX_DEFAULT_LABEL ?= "DE10 Nano Audio Mini ${DE10_NANO_DEPLOY_CONFIG}
    Build"
88
89  # Console configuration for kernel command line
90  UBOOT_EXTLINUX_CONSOLE ?= "console=ttyS0,115200n8"
91  # Root filesystem location (partition 2 on SD card)
92  UBOOT_EXTLINUX_ROOT ?= "${@'root=/dev/nfs nfsroot=' + d.getVar('DE10_NANO_NFS_IP')
    + ':' + d.getVar('DE10_NANO_NFS_DIR') + ',port=' + d.getVar('
    DE10_NANO_NFS_PORT') + ',nfsvers=3,tcp' if d.getVar('DE10_NANO_DEPLOY_CONFIG')
    == 'tftp-nfs' else 'root=/dev/mmcblk0p2'}"
93
94  # Human-readable description for the boot menu
95  UBOOT_EXTLINUX_MENU_DESCRIPTION:default ?= "DE10 Nano Audio Mini ${
    DE10_NANO_DEPLOY_CONFIG} Build"

```

Code Listing 33: Serial Console and ExtLinux Configuration

Listing 34 shows boot partition and image generation settings that specify which files to include in the SD card boot partition and how to structure the final bootable image:

```
de10-nano-audio-mini.conf: 111-124
```

```

111  # Boot Partition Files Configuration
112  # Define files to include in the boot partition of the SD card image
113  IMAGE_BOOT_FILES ?= "\

```

```

114     devicetree/${DE10_NANO_CUSTOM_DTB};intel/socfpga/${DE10_NANO_CUSTOM_DTB} \
115     ${KERNEL_IMAGETYPE} \
116     extlinux.conf;extlinux/extlinux.conf \
117     soc_system.rbf;bitstream/soc_system.rbf \
118     "
119
120     # Image Generation Configuration
121     # Use the Cyclone5/Arria5 WIC kickstart file for SD card image layout
122     WKS_FILE ?= "de10-nano-${DE10_NANO_DEPLOY_CONFIG}-build.wks"
123     # Add WIC (OpenEmbedded Image Creator) format to generated image types
124     IMAGE_FSTYPES:append = " wic"

```

Code Listing 34: Boot Partition and Image Generation

This machine configuration shows how Yocto separates hardware specific concerns from application logic, allowing the same recipes and classes to target different hardware platforms by simply changing the `MACHINE` variable in `local.conf`. The use of conditional expressions (e.g., `${@...}`) allows dynamic configuration based on deployment method, supporting both development (TFTP/NFS) and production (SD card) workflows without duplicate configuration files.

Configuration files also manage distribution policies in `conf/distro/` directories. A distro configuration like `poky.conf` sets system-wide defaults for package managers, init systems (e.g., `systemd`), and feature sets, making images consistent when it comes to certain aspects like the init system.

Classes

In the Yocto Project, `.bbclass` files serve as reusable modules that provide common functionality, tasks, variables, and logic to streamline the build process across multiple recipes or images. These files are used to provide modularity and pipeline reuse in embedded Linux development. By defining shared behaviors such as default variable settings, task overrides, or new task additions classes it gives a developer the ability to avoid duplicating pipelines across numerous recipes, leading to more maintainable and scalable build configurations. This is particularly valuable in complex projects involving custom hardware, where common patterns (e.g., for kernel

configuration, package installation, or image formatting) can be centralized in classes and inherited as needed.

Classes are incorporated into recipes or other classes using the `inherit` directive, which pulls in the class's definitions during the BitBake parsing phase. For example, a class might define default values for variables like `IMAGE_FSTYPES` (specifying output filesystem types) or override core tasks such as `do_compile` and `do_install`. Additionally, classes can introduce new tasks via the `addtask` directive, extending the build pipeline with custom steps tailored to specific needs, such as hardware specific deployments or post-build validations. Core classes like `core-image.bbclass` provide basic support for generating bootable images, handling aspects like root filesystem population, package installation, and filesystem formatting (e.g., ext4, FAT32, or NTFS).

The `audio-mini-image.bbclass` is a custom class designed for generating minimal focused images on the DE10-Nano board utilizing the Audio Mini Add-on. This class extends standard image building by incorporating package management (e.g., APT/DPKG for Debian-style feeds to allow runtime package installation), NFS deployment for network-based root filesystems (allowing the board to boot over the network for development), and TFTP integration for bootloader and kernel transfers common in embedded development workflows to allow quick iteration without repeatedly flashing hardware, thus speeding up debugging and testing cycles. By inheriting `core-image`, it leverages Yocto's standard image tasks (such as `do_rootfs` for filesystem creation and `do_image` for image generation) while adding project specific features like deployment scripts. This shows how classes can be customized for niche applications, ensuring that images that utilize the class have basic development tools.

Inherit Directive Listing 35 shows the `inherit core-image` statement (line 6), which pulls in the `core-image.bbclass` that defines standard tasks for image generation, such as creating the root filesystem (`do_rootfs`), generating image files (`do_image`), and handling post-processing. By inheriting this, the custom class builds upon an already established class, allowing us to add specialized features without reinventing basic image building logic.

```
audio-mini-image-annotated.bbclass: 6
```

```
6    inherit core-image
```

Code Listing 35: Inherit Directive

Output Image Formats Listing 36 shows `IMAGE_FSTYPES = "tar.gz ext4 wic"` (line 25), which specifies the formats for the generated image: a compressed tarball for easy extraction, an ext4 filesystem for direct mounting, and a WIC Image file that contains bootable media with partitions.

```
audio-mini-image-annotated.bbclass: 25
```

```
25    IMAGE_FSTYPES = "tar.gz ext4 wic"
```

Code Listing 36: Output Image Formats

NFS Deployment Variables Listing 37 defines paths for the root filesystem tarball (`ROOTFS_TARBALL`, line 28) and an NFS deployment script (`NFS_SERVER_SCRIPT`, line 29). Using variables like `${DEPLOY_DIR_IMAGE}` (a standard Yocto directory for output artifacts) ensures the class integrates with the build environment, making deployments reproducible across different machines.

```
audio-mini-image-annotated.bbclass: 28-29
```

```
28    ROOTFS_TARBALL = "${DEPLOY_DIR_IMAGE}/${IMAGE_BASENAME}-${MACHINE}.rootfs.tar.gz"
```

```
29    NFS_SERVER_SCRIPT = "${DEPLOY_DIR_IMAGE}/deploy_nfs.sh"
```

Code Listing 37: NFS Deployment Variables

Custom Task: do_deploy_nfs Listing 38 shows the custom `do_deploy_nfs` task (lines 32-51), which creates a shell script for deploying the root filesystem to an NFS server. It first checks for the tarball's existence (lines 33-36), removes any existing script (lines 38-40), copies the latest WIC

image for convenience (line 43), and then generates the script with commands (lines 45-47) to clear the NFS directory and extract the tarball while preserving ownership flags. The `chmod +x` (line 48) makes it executable, and `bbwarn` (line 50) logs a warning for visibility. This task automates NFS setup, which is useful for development on boards like the DE10-Nano, where network booting eliminates repeated flashing.

audio-mini-image-annotated.bbclass: 31-51

```

31  # Custom task to deploy the rootfs tarball via script
32  do_deploy_nfs() {
33      if [ ! -f "${ROOTFS_TARBALL}" ]; then
34          bberror "Rootfs tarball not found at ${ROOTFS_TARBALL}"
35          exit 1
36      fi
37
38      if [ -e "${NFS_SERVER_SCRIPT}" ]; then
39          rm -f ${NFS_SERVER_SCRIPT}
40      fi
41
42      #Also make a latest .wic copy for easy access
43      cp -Lf ${DEPLOY_DIR_IMAGE}/${IMAGE_BASENAME}-${MACHINE}.rootfs.wic ${
          DEPLOY_DIR_IMAGE}/${IMAGE_BASENAME}-${MACHINE}.latest.wic
44
45      echo "#!/bin/bash" > ${NFS_SERVER_SCRIPT}
46      echo "sudo rm -rf ${DE10_NANO_NFS_DIR}/*" >> ${NFS_SERVER_SCRIPT}
47      echo "sudo tar --same-owner -xzf ${ROOTFS_TARBALL} -C ${DE10_NANO_NFS_DIR}" >>
          ${NFS_SERVER_SCRIPT}
48      chmod +x ${NFS_SERVER_SCRIPT}
49
50      bbwarn "NFS server script: ${NFS_SERVER_SCRIPT}"
51  }
```

Code Listing 38: Custom Task: do_deploy_nfs

Task Management Listing 39 illustrates task management, where `addtask deploy_nfs` after `do_image_complete` before `do_build` (line 54) inserts the custom task into the build

pipeline, ensuring it runs after image completion but before the full build finishes. Flags like `[recrdeps]` (line 55, recursive dependencies), `[depends]` (line 56, task dependencies), and `[nostamp]` (line 57, no timestamp checking to force re-execution) edit execution. Additional dependencies (lines 60-62) on TFTP deployment tasks from other recipes (e.g., kernel, bitstream, device tree) ensure all components are deployed together.

```
audio-mini-image-annotated.bbclass: 54-62

54  addtask deploy_nfs after do_image_complete before do_build
55  do_build[recrdeps] += "do_deploy_nfs"
56  do_deploy_nfs[depends] += "${PN}:do_image_complete"
57  do_deploy_nfs[nostamp] = "1"
58
59  # Force TFTP deployment tasks from individual components to run
60  do_deploy_nfs[depends] += "linux-socfpga-lts:do_tftp_deploy"
61  do_deploy_nfs[depends] += "audio-mini-bitstream:do_tftp_deploy"
62  do_deploy_nfs[depends] += "de10-nano-audio-mini-devicetree:do_tftp_deploy"
```

Code Listing 39: Task Management

By expanding on these elements, we can see how Bitbake classes like `audio-mini-image.bbclass` can allow customizable image generation tailored to specific hardware, applications, and stages of development.

Images

Images represent the defining feature of the Yocto Project, serving as the final output of the build process. They consist of bootable filesystems tailored for the target device, specifying the installed packages, filesystem structure, and device configurations. Essentially, an image acts as a master recipe that aggregates all necessary dependencies for the target hardware. To create an image, developers primarily need to specify the target architecture (via the `MACHINE` variable), along with any custom layers or configurations. Assuming the image recipe is correctly defined, it provides reproducibility as any user with a compatible Yocto environment and the required layers can regenerate an identical build, ensuring consistency across development teams

or production pipelines. Images can range from extremely simple to highly complex, depending on the inheritance, appended packages, and variables used. In Yocto, images are defined in `.bb` files under `recipes-core/images` or similar directories within layers. They typically inherit from base classes like `core-image.bbclass` to use standard tasks for root filesystem generation, package installation, and image formatting. Key variables include:

IMAGE_INSTALL: Lists packages to include in the root filesystem, often appended for custom additions. **IMAGE_FSTYPES:** Specifies output formats (e.g., `tar.gz` for archives, `ext4` for filesystems, `wic` for partitioned disk images). **IMAGE_FEATURES:** Enables high-level features like debugging tools or read-only rootfs. Inheritance via **inherit:** Allows extending functionality from classes, promoting modularity and reuse.

This modular approach allows developers to build upon core images (e.g., `core-image-minimal` for lightweight systems) or custom classes, adding layers of complexity as needed without redundant pipelines.

The following examples illustrate how image recipes leverage the `audio-mini-image.bbclass` discussed previously. Listing 40 demonstrates a basic image recipe that inherits from the custom class, pulling in predefined functionality for a minimal setup:

audio-mini-passthrough.bb

```
1  # meta-msu-de10-nano/recipes-core/images/audio-mini-passthrough.bb
2  SUMMARY = "Custom image based on core-image-minimal for de10-nano-audio-mini"
3  LICENSE = "MIT"
4
5  # Inherit core-image for standard image tasks
6  inherit audio-mini-image
```

Code Listing 40: Yocto Project Image: `audio-mini-passthrough.bb`

Listing 41 shows a more complex example that builds upon the same class but extends it by appending additional packages via `IMAGE_INSTALL:append`, showcasing how images can layer custom components while maintaining reusability:

```

audio-mini-combfilter.bb

1  # meta-msu-de10-nano/recipes-core/images/audio-mini-combfilter.bb
2  SUMMARY = "Custom image with CombFilter support for DE10-Nano"
3  LICENSE = "MIT"
4
5  # Inherit the common Audio Mini image functionality
6  inherit audio-mini-image
7
8  # Add CombFilter-specific packages
9  IMAGE_INSTALL:append = " audiomini-combfilter-controller"

```

Code Listing 41: Yocto Project Image: audio-mini-combfilter.bb

Both examples use inheritance to avoid duplication, showing off how utilizing Yocto's emphasis on modular build concepts can make a final image simple to produce. This inheritance ties back to the class discussed previously, allowing for scalable image definitions across projects.

Board Support Package Components

Beyond recipes, classes, and images, a complete Yocto BSP layer includes critical hardware-specific components that enable the system to boot and interact with specialized peripherals. The `meta-msu-de10-nano` layer contains several key BSP elements organized under `recipes-bsp/` and `recipes-kernel/`, each playing a vital role in the embedded Linux system. These components include bootloader configurations, device trees, FPGA bitstreams, and kernel customizations. The following subsections examine each component, with contextual directory structures showing the relevant files within the layer.

U-Boot Bootloader Configuration U-Boot (Universal Boot Loader) is the first software that runs on the DE10-Nano's ARM processor after power-on. It initializes memory, loads the Linux kernel, and passes control to the operating system. The `meta-msu-de10-nano` layer customizes U-Boot for the Audio Mini expansion board through `.bbappend` files that extend the base U-Boot recipes from the `meta-intel-fpga` layer.

The U-Boot customization directory structure within `recipes-bsp/u-boot/` is shown in Listing 42:

```
recipes-bsp/u-boot/
|   +-- u-boot/
|       |-- u-boot-socfpga_%.bbappend  # U-Boot customization
|       +-- files/
|           |-- de10_nano_audio_mini_sd_defconfig
|           +-- de10_nano_audio_mini_tftp_nfs_defconfig
```

Code Listing 42: `recipes-bsp/u-boot/` Structure

The `.bbappend` file adds custom defconfig files for different boot scenarios: SD card boot and network boot via TFTP/NFS. These defconfig files specify which U-Boot features to enable, memory addresses for loading the kernel and device tree, and boot command sequences. The TFTP/NFS configuration enables rapid development by booting the kernel and root filesystem over the network, eliminating the need to reflash SD cards during testing. The `qts/` subdirectory contains Quartus-generated hardware configuration headers for SDRAM, clock, and pin multiplexing, ensuring U-Boot properly initializes the FPGA's Hard Processor System (HPS) before kernel execution.

Device Tree Specifications Device trees are data structures that describe hardware topology to the Linux kernel. They specify peripherals, memory regions, interrupts, and connections, allowing a single kernel binary to support multiple hardware configurations. For SoC-FPGA systems like the DE10-Nano, device trees are particularly important as they bridge the ARM processor with FPGA-implemented hardware.

The device tree directory structure within `recipes-bsp/device-tree/` is shown in Listing 43:

```
recipes-bsp/device-tree/
```

```
| |-- device-tree/
| | |-- de10-nano-audio-mini-devicetree.bb
| | +-- files/
| |     |-- de10-nano-audio-mini-base.dts
| |     +-- de10-nano-audio-mini-base.dtsi
```

Code Listing 43: recipes-bsp/device-tree/ Structure

The `.dts` (Device Tree Source) file defines the complete hardware description for the Audio Mini board, including the AD1939 audio codec, the TPA6130A2 headphone amplifier, and memory-mapped FPGA components. The `.dtsi` (Device Tree Source Include) file contains shared definitions that can be included by multiple device trees, promoting reuse. These files compile into a `.dtb` (Device Tree Blob) binary that U-Boot loads alongside the kernel. The nested `meta-my-audiomini-combfilter` sub-layer demonstrates device tree extension through `.bbappend` files, adding entries for custom FPGA IP cores without modifying the base device tree.

FPGA Bitstream Deployment For SoC-FPGA platforms, the FPGA fabric must be configured with a bitstream before FPGA-based peripherals become accessible. The `audio-mini-bitstream` recipe within `recipes-bsp/audio-mini-bitstream/` manages this step, as shown in Listing 44:

```
recipes-bsp/audio-mini-bitstream/
```

```
| |-- audio-mini-bitstream/
| | |-- audio-mini-bitstream.bb      # FPGA bitstream recipe
| | +-- files/
| |     +-- soc_system.rbf          # FPGA configuration
```

Code Listing 44: recipes-bsp/audio-mini-bitstream/ Structure

The `.rbf` (Raw Binary File) is generated by Intel Quartus from the FPGA design and contains the configuration data for the Cyclone V FPGA. The recipe packages this bitstream so it's available in the root filesystem, under `/lib/firmware/`. Early-boot mechanisms or systemd services can

then load this bitstream into the FPGA fabric during system initialization. The nested sub-layer's `.bbappend` file demonstrates how to override the base bitstream with a modified design, such as one including additional signal processing IP cores, without modifying the base layer.

Linux Kernel Configuration The Linux kernel provides the foundation for the embedded system, managing hardware resources, device drivers, and system services. Kernel customization within `recipes-kernel/linux/` is achieved through configuration fragments and `.bbappend` files, as shown in Listing 45:

```
recipes-kernel/linux/

|-- recipes-kernel/
|   |-- linux/
|       |-- linux-socfpga-lts_%.bbappend
|       |-- files/
|           |-- de10nano-fragment.cfg # Kernel configuration
```

Code Listing 45: `recipes-kernel/linux/` Structure

The `linux-socfpga` recipes in the `meta-intel-fpga` layer provide the baseline kernel for Altera/Intel SoC-FPGA devices. The `.bbappend` file extends this recipe by appending the `de10nano-fragment.cfg` configuration fragment. Configuration fragments are preferred over full `.config` files because they specify only the required changes, making them portable across kernel versions and reducing maintenance burden. The `.bbappend` file also implements custom deployment tasks for TFTP, automatically copying the compiled kernel image to a TFTP server directory when the `DE10_NANO_DEPLOY_CONFIG` variable is set, streamlining network-based development workflows.

Build Utility Files

Beyond the core BSP components, the `meta-msu-de10-nano` layer includes utility files and scripts that facilitate the build process and deployment workflows. These files automate common tasks, define disk partitioning schemes, and configure development environments. The following subsections examine these utility components.

WIC Kickstart Files Yocto’s tool for creating bootable disk images with proper partitioning schemes is the OpenEmbedded Kickstart `.wks`, and this is what is used to produce `.wic` files. The `wic/` directory at the layer root contains WIC kickstart files (`.wks`) that define partition layouts, filesystems, and bootloader installation, as shown in Listing 46:

```
wic/
|-- wic/
|   |-- de10-nano-sd-build.wks      # SD card image layout
|   +-- de10-nano-tftp-nfs-build.wks # Network boot layout
```

Code Listing 46: `wic/` Structure

The `de10-nano-sd-build.wks` file creates a complete SD card image with a FAT32 boot partition containing U-Boot, the kernel, and device tree, plus an ext4 root filesystem partition. The `de10-nano-tftp-nfs-build.wks` creates a minimal boot partition for network booting scenarios, where the kernel and root filesystem load over the network. WIC processes these kickstart files during the `do_image_wic` task, directly incorporating outputs from U-Boot, kernel, and image recipes. The resulting `.wic` files can be written directly to SD cards with `dd` or similar tools, producing immediately bootable media for the needed boot method.

Building and Deploying the DE10-Nano Image

This section describes building a bootable Linux image for the DE10-Nano board using the Yocto Project and the `meta-msu-de10-nano` layer. It covers environment setup, build configuration, compilation, and deployment methods. The build process integrates components discussed in previous sections: recipes, classes, configuration files, and BSP elements.

Environment Setup and Layer Configuration

The build process requires cloning the Yocto Project repositories, adding layers, and configuring the build environment for the DE10-Nano target.

Repository Setup Clone the Poky reference distribution and the Intel FPGA meta-layer using the Scarthgap LTS release:

```
export DE10_WORKSPACE=$(pwd)
git clone git@github.com:yoctoproject/poky.git \
    --branch=scarthgap $DE10_WORKSPACE/yocto

mkdir -p $DE10_WORKSPACE/meta-layers
git clone https://github.com/night1rider/meta-msu-de10-nano.git \
    $DE10_WORKSPACE/meta-layers/meta-msu-de10-nano
git clone https://git.yoctoproject.org/meta-intel-fpga \
    --branch=scarthgap \
    $DE10_WORKSPACE/meta-layers/meta-intel-fpga
```

This creates the directory structure and fetches the Yocto components. The `meta-intel-fpga` layer provides Altera/Intel SoC-FPGA support with board definitions and hardware abstraction layers. The `meta-msu-de10-nano` layer adds Audio Mini specific customizations.

Build Directory Initialization Source the Yocto setup script to configure environment variables and create the build directory:

```
mkdir -p $DE10_WORKSPACE/de-10-nano_minimal
cd $DE10_WORKSPACE/de-10-nano_minimal
source $DE10_WORKSPACE/yocto/oe-init-build-env
```

This creates a `build/` directory containing `conf/bblayers.conf` and `conf/local.conf`.

Layer Registration Register layers in `conf/bblayers.conf`. Modify the file to include the Intel FPGA layer and the DE10-Nano layer:

```

BBLAYERS ?= " \
/path/to/yocto/meta \
/path/to/yocto/meta-poky \
/path/to/meta-layers/meta-intel-fpga \
/path/to/meta-layers/meta-msu-de10-nano \
"

```

Layer order in `bblayers.conf` and priority values determine override precedence. The `meta-msu-de10-nano` layer defines `BBFILE_PRIORITY = "6"` in its `layer.conf`, overriding layers with lower priority values when recipe conflicts occur.

Machine and Deployment Configuration

The `conf/local.conf` file specifies the target machine and deployment method.

Target Machine Selection The `MACHINE` variable identifies the target hardware:

```
MACHINE = "de10-nano-audio-mini"
```

This references the machine configuration file discussed in Section 27, which defines bootloader settings, kernel versions, device tree configurations, and serial console parameters specific to the DE10-Nano with Audio Mini hardware.

Deployment Method Configuration The `DE10_NANO_DEPLOY_CONFIG` variable selects between SD card and network boot configurations:

```
# For SD card deployment
```

```
DE10_NANO_DEPLOY_CONFIG = "sd"
```

```
# For TFTP/NFS network deployment
```

```
DE10_NANO_DEPLOY_CONFIG = "tftp-nfs"
```

SD card deployment produces a bootable image for production. TFTP/NFS deployment loads the kernel and root filesystem over the network, eliminating repeated SD card writes during testing.

Image Build Process

BitBake compiles all components and generates the image. The build system resolves dependencies, downloads sources, compiles packages, and assembles the root filesystem.

Building the Image Execute the build command:

```
bitbake audio-mini-passthrough
```

BitBake parses recipes and configuration files in the registered layers, building a dependency graph. It executes tasks in dependency order: fetching source code, unpacking archives, patching sources, configuring build systems, compiling code, installing artifacts, and packaging results. Initial builds require several hours as BitBake downloads and compiles the Linux ecosystem from source.

The `audio-mini-passthrough` image inherits from `audio-mini-image.bbclass`, which includes package management tools, Audio Mini drivers, and deployment automation. The class's `do_deploy_nfs` task (Section 34) generates deployment scripts when network boot is configured.

Build Output Artifacts Builds produce artifacts in the deployment directory:

```
$DE10_WORKSPACE/de-10-nano_minimal/build/tmp/deploy/images/cyclone5/
```

Outputs include:

- `core-image-minimal-cyclone5.rootfs.wic`: Bootable disk image with partitions
- `core-image-minimal-cyclone5.rootfs.tar.gz`: Root filesystem archive for NFS deployment
- `core-image-minimal-cyclone5.rootfs.ext4`: Root filesystem in ext4 format
- `zImage`: Linux kernel binary

- `*.dtb`: Device tree blob(s)
- `u-boot-with-spl.sfp`: U-Boot bootloader with Secondary Program Loader

For SD card deployment, only the `.wic` file is needed. For network deployment, the kernel, device tree, and root filesystem tarball are used separately.

Deployment Methods

The `meta-msu-de10-nano` layer supports two deployment methods determined by the `DE10_NANO_DEPLOY_CONFIG` variable.

SD Card Deployment SD card deployment creates a bootable image for production. The `.wic` file contains a disk layout with boot and root partitions, created according to the WIC kickstart file specifications (Section 45).

To write the image to an SD card:

```
sudo dd if=core-image-minimal-cyclone5.rootfs.wic \
of=/dev/sdX bs=4M status=progress && sync
```

The `dd` command copies the image to the SD card device (`/dev/sdX`). The `bs=4M` parameter sets the block size to 4 megabytes, and `sync` ensures data is written before returning. The SD card boots the custom Linux system when inserted into the DE10-Nano.

TFTP/NFS Network Deployment Network deployment separates boot components to enable development without reflashing hardware. This method requires a development host running TFTP and NFS servers. The `meta-msu-de10-nano` layer includes a `setup_servers.sh` script for server configuration [9].

The deployment uses three components:

TFTP Server for Bootloader and Kernel: U-Boot fetches the kernel (`zImage`) and device tree (`.dtb`) from the TFTP server during boot. The `do_deploy_tftp` tasks copy these files to the TFTP directory when `DE10_NANO_DEPLOY_CONFIG = "tftp-nfs"`.

NFS Server for Root Filesystem: The root filesystem mounts over NFS. The `do_deploy_nfs` task generates a deployment script that extracts the root filesystem tarball to the NFS export directory.

U-Boot Network Boot Configuration: The U-Boot configuration sets environment variables for server IP addresses, TFTP paths, and NFS mount parameters. These settings are defined in `UBOOT_EXTLINUX_KERNEL_ARGS` (Section 27).

The SD card must contain the U-Boot bootloader for network boot. Initial setup requires flashing the SD card with the bootloader and network boot configuration. The bootloader loads the kernel and device tree from the TFTP server and starts the kernel. The kernel then mounts the root filesystem over NFS. Subsequent changes to the kernel, device tree, or root filesystem require only network file updates. Bootloader changes require reflashing the SD card.

Network boot enables testing driver or kernel module changes with only a board reset.

Build System Integration

The build system demonstrates Yocto’s layering model for modular embedded Linux development. Recipes, classes, configuration files, and BSP elements combine to produce reproducible images.

The `meta-msu-de10-nano` layer extends vendor-provided layers (`meta-intel-fpga`) with application-specific customizations while maintaining compatibility with upstream updates. The layer’s structure, with nested sub-layers for derivative projects, supports scalability.

Build documentation, including troubleshooting guides and network server setup procedures, is available in the layer’s GitHub repository [9].

CONCLUSION AND FUTURE WORK

Hardware Acceleration

This work successfully implemented a custom pipelined CNN convolution accelerator for real-time hyperspectral image processing on FPGA hardware (Chapter 2). The accelerator achieves its real-time processing goals through a complete hardware pipeline with three integrated subsystems: a memory controller using Xilinx UltraRAM blocks with circular buffering, a data slicer for unpacking 72-bit memory words, and a convolution engine with hierarchical multiply-accumulate structures.

Performance analysis (Section 2) demonstrates that the accelerator achieves 227 frames per second at 100 MHz, exceeding the target of 181 fps. The memory packing strategy (6 values per row) and 2:3 read ratio enable parallel readout at 4 values/clock (48 bits/clock), 4× faster than the serial input rate of 1 value/clock (12 bits/clock). Processing 1,096 convolutions completes in 1.1 ms (110,000 cycles), while scanline capture requires 4.4 ms (440,000 cycles), providing a 75% timing margin. The design is input-bound: processing completes in 25% of the scanline capture period. The pipelined MAC architecture processes $5 \times 400 \times 5$ kernel operations on $400 \times 1100 \times 5$ datacubes, performing 20,204 arithmetic operations per convolution (10,000 multiplications and 10,204 additions across four pipeline stages: multiplication, hierarchical adder trees, slice accumulation, and final accumulation). The stride-5 operation with partial accumulation computes all 1,096 overlapping windows per frame. Processing one complete output frame requires $1,096 \text{ convolutions} \times 20,204 \text{ operations} = 22.1 \text{ million arithmetic operations}$. The architecture processes 60 multiplications per clock cycle (12 per slice across 5 slices) using the FPGA's DSP48E2 primitives for fixed-point arithmetic. Verification testing confirmed correct operation across 5 test runs with randomly generated 36-frame datasets at 100 MHz.

The custom memory allocation scheme distributes data across five RAM block groups, allowing sequential access without complex address calculations during convolution. This organization, combined with the 2:3 memory read ratio between frame and kernel data, allows

the pipeline to stream complete datacube slices continuously. The 45-bit fixed-point output format (15 integer, 30 fractional bits) prevents overflow across the 10,000 multiply-accumulate operations per output position.

Implementation using MATLAB Simulink and Vitis Model Composer generates synthesizable VHDL while providing clear hardware visualization for future development. Comprehensive validation through 5 test runs with randomly generated 36-frame datasets at 100 MHz confirmed correct operation—all test cases passed with exact matches between hardware output and reference calculations using three independent verification methods. The accelerator utilizes modest FPGA resources: 100 UltraRAM blocks (95 for frame circular buffer, 5 for kernel weights), 30 Block RAMs, 602 DSP slices, 26,061 LUTs, and 61,394 registers. These resource estimates are well within the capabilities of the XCZU15EG Zynq UltraScale+ MPSoC (31.5 Mb UltraRAM, 26.2 Mb Block RAM, 3,528 DSP slices), confirming the design’s feasibility for the target hardware platform.

The successful verification and resource analysis demonstrate that the pipelined CNN convolution architecture is ready for FPGA deployment and real-time hyperspectral data processing in the Smart Optical Sensor application.

Software Solution

This work demonstrated embedded Linux development for SoC-FPGA platforms using the Yocto Project. The `meta-msu-de10-nano` custom BSP layer implements a complete build system for the DE10-Nano board with Audio Mini hardware, integrating U-Boot bootloader configuration, Linux kernel customization, device tree specifications, and FPGA bitstream deployment. The layer structure separates hardware-specific BSP components (`recipes-bsp/`), kernel modifications (`recipes-kernel/`), and system images (`recipes-core/`) using Yocto’s modular architecture. Custom classes extend standard image building with package management, NFS deployment, and TFTP integration for development workflows. The build system supports two deployment methods: SD card images for production and network boot (TFTP/NFS) for development, eliminating repeated reflashing during testing. Machine configuration files use conditional expressions to support

both deployment modes from a single definition. BitBake resolves dependencies and compiles the Linux ecosystem from source, producing bootable WIC images, kernel binaries, device trees, and root filesystems. The implementation extends the vendor-provided `meta-intel-fpga` layer while maintaining upstream compatibility, demonstrating how application-specific customizations layer onto base BSP functionality. The open-source layer serves as a reference for SoC-FPGA Board Support Package development.

Future Work

To continue this project, two paths can be explored. The first is to begin programming and testing the initial layer implementation in the hardware accelerator. The second is a parallel path to develop a custom BSP layer for the target hardware needed for the Smart Camera. This involves creating a BSP layer for the Xilinx Zynq UltraScale+ MPSoC similar to what was done for the DE10-Nano. Crucially, future work must also focus on bridging these two domains by developing custom Linux kernel drivers that utilize Direct Memory Access (DMA) engines and AXI interconnects to efficiently transfer data between the FPGA accelerator and the software application layer. These drivers will also be responsible for configuring runtime parameters, such as loading kernel weights, and handling hardware interrupts to signal the completion of convolution operations. Successful implementation of these drivers will complete the system architecture, allowing the needed hardware-software coordination required for real-time hyperspectral analysis.

REFERENCES CITED

- [1] AMD. *Vitis Model Composer User Guide (UG1483)*, 2021. URL <https://docs.amd.com/r/2021.2-English/ug1483-model-composer-sys-gen-user-guide>. Accessed: 2025-09-24.
- [2] AMD. *Kria SOMs for Vision AI Applications White Paper (WP538)*, 2023. URL <https://docs.amd.com/v/u/en-US/wp529-som-benchmarks>. Accessed: 2025-09-24.
- [3] AMD. *Kria K26 System-on-Module Data Sheet (DS987)*, 2024. URL <https://docs.amd.com/r/en-US/ds987-k26-som>. Accessed: 2025-09-24.
- [4] AMD. *Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393)*, 2025. URL <https://docs.amd.com/v/u/en-US/ug1393-vitis-application-acceleration>. Accessed: 2025-09-24.
- [5] AMD. Zynq ultrascale+ mpsoc, n.d. URL <https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-ultrascale-plus-mpsoc.html>. Accessed: 2025-09-24.
- [6] AMD (Xilinx). *7 Series FPGAs Configurable Logic Block User Guide (UG474)*, 2018. URL https://docs.amd.com/r/en-US/ug474_7Series_CLB. Accessed: 2025-09-24.
- [7] AMD (Xilinx). *Zynq-7000 SoC Technical Reference Manual (UG585)*, 2021. URL <https://docs.amd.com/r/en-US/ug585-zynq-7000-SoC-TRM>. Accessed: 2025-09-24.
- [8] ArXiv Preprint. Hyperspectral imaging applications, 2024. URL <https://arxiv.org/html/2411.16107v1>. Accessed: 2025-09-23, Note: Check authorship.
- [9] Zackery Backman. meta-msu-de10-nano: Yocto bsp layer for de10-nano with audio mini, 2024. URL <https://github.com/ADSD-SoC-FPGA/meta-msu-de10-nano>. Accessed: 2025-11-07.
- [10] Zackery Backman. Hyperspectral processing. <https://github.com/rksnider/HyperspectralProcessing>, 2025. Accessed: 2025-11-21.
- [11] C&T Solution. What is fpga (field programmable gate array) and how does it work?, n.d. URL https://www.candtsolution.com/jp/news_events-detail/what-is-fpga-field-programmable-gate-array-and-how-does-it-work/. Accessed: 2025-09-24.
- [12] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2018. URL <https://arxiv.org/abs/1603.07285>.
- [13] Enclustra. Mercury+ xu1 soc module user manual, 2023. URL <https://www.enclustra.com/en/products/system-on-chip-modules/mercury-xu1/>. Accessed: 2025-10-04.
- [14] Enclustra. Mercury st1 base board user manual, 2024. URL <https://www.enclustra.com/en/products/base-boards/mercury-st1/>. Accessed: 2025-10-04.
- [15] Michael Falkowski, Jacquelyn Shuman, Justin Boland, Teresa Kauffman, Barry Lefer, Melissa Meying Martin, and Haris Riris. The nasa firesense project: Responding to stakeholder needs across the fire life cycle, 2024. ISSN 2153-7003.

- [16] freedesktop.org. systemd system and service manager, n.d. URL <https://www.freedesktop.org/wiki/Software/systemd/>. Accessed: 2025-11-07.
- [17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL <http://www.deeplearningbook.org>.
- [18] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2 edition, 2009. URL <https://hastie.su.domains/ElemStatLearn/>.
- [19] The Deep Hub. Convolutional neural networks: A comprehensive guide, 2021. URL <https://medium.com/thedeephub/convolutional-neural-networks-a-comprehensive-guide-5cc0b5eae175>. Accessed: 2025-10-04.
- [20] IBM. Convolutional neural networks, 2021. URL <https://www.ibm.com/think/topics/convolutional-neural-networks>. Accessed: 2025-10-04.
- [21] ISPRS Archives. Hyperspectral remote sensing for fire applications. *ISPRS Archives*, XLIII-B3-2021:215–221, 2021. URL <https://isprs-archives.copernicus.org/articles/XLIII-B3-2021/215/2021/isprs-archives-XLIII-B3-2021-215-2021.pdf>. Accessed: 2025-09-23.
- [22] Ricardo Jasinski. *Effective Coding with VHDL: Principles and Best Practice*. The MIT Press, 2016. ISBN 0262034220.
- [23] Remote Sensing Journal. Hyperspectral imaging for wildfire detection and management: A review. 10(12):456, 2020. doi: 10.3390/rs10120456.
- [24] Linux Kernel. About the linux kernel, n.d. URL <https://www.kernel.org/category/about.html>. Accessed: 2025-10-27.
- [25] Qiang et al. Lan. High performance implementation of 3d convolutional neural networks on a gpu. 2017. URL <https://www.hindawi.com/journals/cin/2017/8348671/>. Accessed: 2025-10-04.
- [26] Yann LeCun. Gradient-based learning applied to document recognition, 1998. URL http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf. Accessed: 2025-10-04.
- [27] Yann LeCun. Lenet-5, convolutional neural networks, 2017. URL <https://web.archive.org/web/20171228091645/http://deeplearning.net/tutorial/lenet.html>. Accessed: 2025-10-04.
- [28] S. A. Lewis. Fire and fuels management research applications, 2004. URL https://www.fs.usda.gov/rm/pubs_other/rmrs_2004_lewis_s001.pdf. Accessed: 2025-09-23.
- [29] MathWorks. *Simulink User’s Guide*, 2023. URL <https://www.mathworks.com/help/simulink/>. Accessed: 2025-11-07.

- [30] MDPI Remote Sensing. Hyperspectral remote sensing applications. *Remote Sensing*, 13(5): 897, 2021. URL <https://www.mdpi.com/2072-4292/13/5/897>. Accessed: 2025-09-23.
- [31] Michigan State University College of Agriculture and Natural Resources. Wui policies, n.d. URL https://www.canr.msu.edu/fccp/Uploads/Files/4h.%20WUI%20Policies_FINAL.pdf. Accessed: 2025-09-23.
- [32] N/A. Smart fires: Sensors, machine learning, and artificial intelligence in real time fire science, 2023. URL <https://www.mtnsfepscor.org/projects/smart-fires>. Accessed: 2025-09-23.
- [33] NASA. Tech today: Remote sensing technology fights forest fires, n.d. URL <https://www.nasa.gov/general/tech-today-remote-sensing-technology-fights-forest-fires/>. Accessed: 2025-01-22.
- [34] NASA Earth Science Division. Using hyperspectral imaging to monitor vegetation and wildfire dynamics. n.d. URL <https://www.nasa.gov>. Accessed: 2025-01-22.
- [35] NASA Earth Science Technology Office. New nasa firesense technology program will help first responders combat wildfires, n.d. URL <https://esto.nasa.gov/new-nasa-firesense-technology-program-will-help-first-responders-combat-wildfires/>. Accessed: 2025-01-22.
- [36] National Park Service. Fire ecology teacher guide, n.d.. URL <https://www.nps.gov/romo/learn/education/upload/Fire-Ecology-Teacher-Guide-Final-2.pdf>. Accessed: 2025-09-23.
- [37] National Park Service. Wildland fire facts: There must be all three, n.d.. URL <https://www.nps.gov/articles/wildlandfire-facts-fuel-heat-oxygen.htm>. Accessed: 2025-09-23.
- [38] National Park Service. Wildland fire behavior, n.d.. URL <https://www.nps.gov/articles/wildland-fire-behavior.htm>. Accessed: 2025-09-23.
- [39] NV5 Geospatial. Leveraging hyperspectral data for wildfire mapping and risk mitigation. n.d.. URL <https://www.nv5.com>. Accessed: 2025-01-22.
- [40] NV5 Geospatial. Leveraging remote sensing for wildfire management, n.d.. URL https://www.nv5geospatialsoftware.com/Portals/0/pdfs/9-21_L3HG-WildfireMngmt_Whitepaper_Web.pdf. Accessed: 2025-01-22.
- [41] International Journal of Wildland Fire. Applications of hyperspectral imagery in wildfire risk modeling. 2018. URL <https://www.publish.csiro.au/wf>. Accessed: 2025-01-22.
- [42] PMC - National Center for Biotechnology Information. Post-fire vegetation recovery assessment using hyperspectral data. 2021. URL <https://pmc.ncbi.nlm.nih.gov/articles/PMC8230056/>. Accessed: 2025-09-23.
- [43] OpenEmbedded Project. Openembedded project documentation, n.d.. URL https://www.openembedded.org/wiki/Main_Page. Accessed: 2025-10-27.

- [44] Yocto Project. Bitbake concepts, n.d.. URL <https://docs.yoctoproject.org/bitbake/2.12/bitbake-user-manual/bitbake-user-manual-intro.html#concepts>. Accessed: 2025-10-27.
- [45] Yocto Project. Yocto project documentation, n.d.. URL <https://docs.yoctoproject.org/ref-manual/index.html>. Accessed: 2025-10-27.
- [46] Yocto Project. Yocto project technical overview, n.d.. URL <https://www.yoctoproject.org/development/technical-overview/>. Accessed: 2025-10-27.
- [47] Rohit Raja. *Data mining and machine learning applications*. Scrivener Publishing LLC, Beverly, MA, 2022. ISBN 9781119791782.
- [48] ResearchGate. Assembly of uv-ozone reactor to combat of coronavirus and other pathogenic microorganisms, 2021. URL https://www.researchgate.net/publication/356262147_Assembly_of_UV-Ozone_Reactor_to_Combat_of_Coronavirus_and_Other_Pathogenic_Microorganisms. Accessed: 2025-09-23.
- [49] ResearchGate. The use of hyperspectral remote sensing for mineral exploration: A review, 2022. URL https://www.researchgate.net/publication/363465665_The_use_of_hyperspectral_remote_sensing_for_mineral_exploration_a_review. Accessed: 2025-09-23.
- [50] ResearchGate. Application of hyperion hyperspectral remote sensing data for wildfire fuel mapping, n.d. URL https://www.researchgate.net/publication/263641755_Application_of_Hyperion_Hyperspectral_Remote_Sensing_Data_for_Wildfire_Fuel_Mapping. Accessed: 2025-09-23.
- [51] Resources for the Future. Wildfires in the united states 101: Context and consequences, n.d. URL <https://www.rff.org/publications/explainers/wildfires-in-the-united-states-101-context-and-consequences/>. Accessed: 2025-01-22.
- [52] Haris Riris, Teresa Kauffman, Michael Falkowski, Jacquelyn Shuman, Justin Boland, Melissa Meying Martin, Barry Leffer, and Michael Seablom. Nasa’s technology development program for wildfire science, management, and disaster mitigation, 2024. ISSN 2153-7003.
- [53] D. A. Roberts. Hyperspectral remote sensing for wildland fire applications, 1999. URL https://content.csbs.utah.edu/~pdennison/pubs/roberts_jfsc1999.pdf. Accessed: 2025-09-23.
- [54] D. A. Roberts and P. E. Dennison. Hyperspectral remote sensing for fire fuel mapping, 2003. URL https://content.csbs.utah.edu/~pdennison/pubs/roberts_ghent2003.pdf. Accessed: 2025-09-23.
- [55] ScienceDirect. Hyperspectral remote sensing for burn severity assessment. *Remote Sensing of Environment*, 2007. URL <https://www.sciencedirect.com/science/article/abs/pii/S0034425706005025?via%3Dihub>. Accessed: 2025-09-23.

- [56] Ross K. Snider. *Advanced digital system design using SoC FPGAs : an integrated hardware/software approach*. Springer, Cham, Switzerland, 1st ed. 2023. edition, 2023. ISBN 9783031154164.
- [57] SPIE Digital Library. Hyperspectral remote sensing in lithological mapping, mineral exploration and environmental applications. *Journal of Applied Remote Sensing*, 15(3):031501, 2021. URL <https://www.spiedigitallibrary.org/journals/journal-of-applied-remote-sensing/volume-15/issue-3/031501/Hyperspectral-remote-sensing-in-lithological-mapping-mineral-exploration-and-environmental> 10.1117/1.JRS.15.031501.full. Accessed: 2025-09-23.
- [58] Ljubisa Stankovic and Danilo Mandic. Convolutional neural networks demystified: A matched filtering perspective basis tutorial, 2022. URL <https://arxiv.org/abs/2108.11663>.
- [59] Universal Lab. Non-destructive testing technology based on hyperspectral imaging, n.d. URL https://www.universallab.org/blog/blog/non_destructive_testing_technology_based_on_hyperspectral_imaging/. Accessed: 2025-01-22.
- [60] University of Alaska Fairbanks Geophysical Institute. Agu research spotlight: Hyperspectral imaging forests can aid wildfire prevention, n.d. URL <https://www.gi.alaska.edu/news/agu-research-spotlight-hyperspectral-imaging-forests-can-aid-wildfire-prevention>. Accessed: 2025-09-23.
- [61] U.S. Environmental Protection Agency. Climate change indicators: Wildfires, n.d. URL <https://www.epa.gov/climate-indicators/climate-change-indicators-wildfires>. Accessed: 2025-09-23.
- [62] U.S. Fire Administration. Wui issues and resolutions report, n.d. URL <https://www.usfa.fema.gov/downloads/pdf/publications/wui-issues-resolutions-report.pdf>. Accessed: 2025-09-23.
- [63] U.S. Forest Service. Managing land: Fire, n.d. URL <https://www.fs.usda.gov/managing-land/fire>. Accessed: 2025-09-23.
- [64] U.S. Geological Survey. Hyperspectral remote sensing applications for wildfire risk assessment. n.d. URL <https://www.usgs.gov>. Accessed: 2025-01-22.
- [65] F. D. Van der Meer. Multi and hyperspectral geologic remote sensing: A review, 2012. URL https://ris.utwente.nl/ws/portalfiles/portal/183729759/Van_der_meer_2012_Multi_and_hyperspectral_geologic_r.pdf. Accessed: 2025-09-23.
- [66] Analytics Vidhya. Types of convolutional neural networks: Lenet, alexnet, vgg-16, net, resnet and inception net, 2021. URL <https://medium.com/analytics-vidhya/types-of-convolutional-neural-networks-lenet-alexnet-vgg-16-net-resnet-and-inception-net-> Accessed: 2025-10-04.
- [67] Vrije Universiteit Amsterdam. Hyperspectral remote sensing of fire: State of the art and future perspectives, n.d. URL https://research.vu.nl/ws/portalfiles/portal/249559292/Hyperspectral_remote_sensing_of_fire_State_of_the_art_and_future_perspectives.pdf. Accessed: 2025-09-23.

- [68] Xilinx Wiki. Zynq ultrascale+ mpsoc, 2021. URL <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/444006775/Zynq+UltraScale+MPSoC>. Accessed: 2025-11-07.

APPENDICES

APPENDIX A

COMPLETE SOURCE CODE

VHDL Source Files

and-gate.vhd

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  entity AND_GATE is
4      port( A, B : in std_logic;
5            Y : out std_logic);
6  end AND_GATE;
7  architecture DATAFLOW of AND_GATE is
8  begin
9      Y <= A and B;
10 end DATAFLOW;
```

d-flip-flop.vhd

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  entity D_FLIP_FLOP is
4      port( CLK : in std_logic;
5            D : in std_logic;
6            Q : out std_logic);
7  end D_FLIP_FLOP;
8  architecture BEHAVIORAL of D_FLIP_FLOP is
9  begin
10     process(CLK)
11     begin
12         if rising_edge(CLK) then
13             Q <= D;
14         end if;
15     end process;
16 end BEHAVIORAL;
```

async-reset.vhd

```
1  process(CLK, RST)
2  begin
3      if RST = '1' then
4          Q <= '0';
5      elsif rising_edge(CLK) then
6          Q <= D;
7      end if;
8  end process;
```

sync-reset.vhd

```
1  process(CLK)
2  begin
3      if rising_edge(CLK) then
4          if RST = '1' then
```

```

5         Q <= '0';
6     else
7         Q <= D;
8     end if;
9 end if;
10 end process;

```

and-gate-registered.vhd

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity AND_GATE_REGISTERED is
5      port( CLK : in std_logic;
6            A   : in std_logic;
7            B   : in std_logic;
8            Y   : out std_logic);
9  end AND_GATE_REGISTERED;
10
11  architecture BEHAVIORAL of AND_GATE_REGISTERED is
12      signal and_result : std_logic;
13  begin
14      -- Combinational AND logic
15      and_result <= A and B;
16
17      -- Register the output
18      process(CLK)
19      begin
20          if rising_edge(CLK) then
21              Y <= and_result;
22          end if;
23      end process;
24  end BEHAVIORAL;

```

half-adder.vhd

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  entity HALF_ADDER is
4      port( A, B : in std_logic;
5            SUM  : out std_logic;
6            CARRY: out std_logic);
7  end HALF_ADDER;
8  architecture DATAFLOW of HALF_ADDER is
9  begin
10     SUM <= A xor B;
11     CARRY <= A and B;
12 end DATAFLOW;

```

ripple-carry-adder-4bit.vhd

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  entity RIPPLE_CARRY_ADDER_4BIT is
4      port( A, B : in std_logic_vector(3 downto 0);
5            Cin : in std_logic;
6            SUM : out std_logic_vector(3 downto 0);
7            Cout : out std_logic);
8  end RIPPLE_CARRY_ADDER_4BIT;
9  architecture STRUCTURAL of RIPPLE_CARRY_ADDER_4BIT is
10     component FULL_ADDER is
11         port( A, B, Cin : in std_logic;
12              SUM, Cout : out std_logic);
13     end component;
14     signal C : std_logic_vector(4 downto 0);
15  begin
16     C(0) <= Cin;
17     FA0 : FULL_ADDER port map(A=>A(0), B=>B(0), Cin=>C(0), SUM=>SUM(0), Cout=>C(1)
18     );
19     FA1 : FULL_ADDER port map(A=>A(1), B=>B(1), Cin=>C(1), SUM=>SUM(1), Cout=>C(2)
20     );
21     FA2 : FULL_ADDER port map(A=>A(2), B=>B(2), Cin=>C(2), SUM=>SUM(2), Cout=>C(3)
22     );
23     FA3 : FULL_ADDER port map(A=>A(3), B=>B(3), Cin=>C(3), SUM=>SUM(3), Cout=>C(4)
24     );
25     Cout <= C(4);
26  end STRUCTURAL;

```

signed-multiplier.vhd

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4  entity SIGNED_MULTIPLIER is
5      port( A, B : in std_logic_vector(15 downto 0);
6            PRODUCT : out std_logic_vector(31 downto 0));
7  end SIGNED_MULTIPLIER;
8  architecture BEHAVIORAL of SIGNED_MULTIPLIER is
9  begin
10     PRODUCT <= std_logic_vector(signed(A) * signed(B));
11  end BEHAVIORAL;

```

counter-4bit.vhd

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4
5  entity COUNTER_4BIT is
6      port( CLK : in std_logic;
7            RST : in std_logic;
8            EN : in std_logic;
9            COUNT : out std_logic_vector(3 downto 0));
10  end COUNTER_4BIT;
11

```

```

12  architecture BEHAVIORAL of COUNTER_4BIT is
13      signal count_int : unsigned(3 downto 0) := (others => '0');
14  begin
15      process(CLK, RST)
16      begin
17          if RST = '1' then
18              count_int <= (others => '0');
19          elsif rising_edge(CLK) then
20              if EN = '1' then
21                  count_int <= count_int + 1;
22              end if;
23          end if;
24      end process;
25
26      COUNT <= std_logic_vector(count_int);
27  end BEHAVIORAL;

```

fifo-sync.vhd

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4
5  -- Synchronous FIFO with single clock domain (2 words deep, 2-bit wide)
6  entity FIFO_SYNC is
7      port(
8          CLK : in std_logic;
9          RST : in std_logic;
10         WR_EN : in std_logic;
11         RD_EN : in std_logic;
12         DIN : in std_logic_vector(1 downto 0);
13         DOUT : out std_logic_vector(1 downto 0);
14         FULL : out std_logic;
15         EMPTY : out std_logic
16     );
17  end FIFO_SYNC;
18
19  architecture BEHAVIORAL of FIFO_SYNC is
20      type memory_type is array (0 to 1) of std_logic_vector(1 downto 0);
21      signal memory : memory_type := (others => (others => '0'));
22      signal wr_ptr : unsigned(0 downto 0) := (others => '0');
23      signal rd_ptr : unsigned(0 downto 0) := (others => '0');
24      signal count : unsigned(1 downto 0) := (others => '0');
25  begin
26      process(CLK)
27      begin
28          if rising_edge(CLK) then
29              if RST = '1' then
30                  wr_ptr <= (others => '0');
31                  rd_ptr <= (others => '0');
32                  count <= (others => '0');
33              else
34                  -- Write operation
35                  if WR_EN = '1' and count < 2 then
36                      memory(to_integer(wr_ptr)) <= DIN;
37                      wr_ptr <= wr_ptr + 1;
38                  end if;
39
40                  -- Read operation

```

```

41         if RD_EN = '1' and count > 0 then
42             rd_ptr <= rd_ptr + 1;
43         end if;
44
45         -- Update count
46         if WR_EN = '1' and RD_EN = '0' and count < 2 then
47             count <= count + 1;
48         elsif WR_EN = '0' and RD_EN = '1' and count > 0 then
49             count <= count - 1;
50         end if;
51     end if;
52 end if;
53 end process;
54
55 DOUT <= memory(to_integer(rd_ptr));
56 FULL <= '1' when count = 2 else '0';
57 EMPTY <= '1' when count = 0 else '0';
58
59 end BEHAVIORAL;

```

fifo-async.vhd

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4
5  -- Asynchronous FIFO with dual clock domains using Gray code (2 words deep, 2-bit
6  -- wide)
7  entity FIFO_ASYNC is
8      port(
9          WR_CLK : in std_logic;
10         RD_CLK : in std_logic;
11         RST : in std_logic;
12         WR_EN : in std_logic;
13         RD_EN : in std_logic;
14         DIN : in std_logic_vector(1 downto 0);
15         DOUT : out std_logic_vector(1 downto 0);
16         FULL : out std_logic;
17         EMPTY : out std_logic
18     );
19 end FIFO_ASYNC;
20
21 architecture BEHAVIORAL of FIFO_ASYNC is
22     type memory_type is array (0 to 1) of std_logic_vector(1 downto 0);
23     signal memory : memory_type := (others => (others => '0'));
24
25     signal wr_ptr_bin : unsigned(1 downto 0) := (others => '0');
26     signal rd_ptr_bin : unsigned(1 downto 0) := (others => '0');
27     signal wr_ptr_gray : unsigned(1 downto 0) := (others => '0');
28     signal rd_ptr_gray : unsigned(1 downto 0) := (others => '0');
29
30     signal wr_ptr_gray_sync1 : unsigned(1 downto 0) := (others => '0');
31     signal wr_ptr_gray_sync2 : unsigned(1 downto 0) := (others => '0');
32     signal rd_ptr_gray_sync1 : unsigned(1 downto 0) := (others => '0');
33     signal rd_ptr_gray_sync2 : unsigned(1 downto 0) := (others => '0');
34
35     function bin_to_gray(bin : unsigned) return unsigned is
36         variable gray : unsigned(bin'range);

```

```

37         gray := bin xor shift_right(bin, 1);
38         return gray;
39     end function;
40
41 begin
42     -- Write clock domain
43     process(WR_CLK)
44     begin
45         if rising_edge(WR_CLK) then
46             if RST = '1' then
47                 wr_ptr_bin <= (others => '0');
48                 wr_ptr_gray <= (others => '0');
49             else
50                 if WR_EN = '1' and FULL = '0' then
51                     memory(to_integer(wr_ptr_bin(0 downto 0))) <= DIN;
52                     wr_ptr_bin <= wr_ptr_bin + 1;
53                     wr_ptr_gray <= bin_to_gray(wr_ptr_bin + 1);
54                 end if;
55             end if;
56
57             -- Synchronize read pointer to write clock domain
58             rd_ptr_gray_sync1 <= rd_ptr_gray;
59             rd_ptr_gray_sync2 <= rd_ptr_gray_sync1;
60         end if;
61     end process;
62
63     -- Read clock domain
64     process(RD_CLK)
65     begin
66         if rising_edge(RD_CLK) then
67             if RST = '1' then
68                 rd_ptr_bin <= (others => '0');
69                 rd_ptr_gray <= (others => '0');
70             else
71                 if RD_EN = '1' and EMPTY = '0' then
72                     rd_ptr_bin <= rd_ptr_bin + 1;
73                     rd_ptr_gray <= bin_to_gray(rd_ptr_bin + 1);
74                 end if;
75             end if;
76
77             -- Synchronize write pointer to read clock domain
78             wr_ptr_gray_sync1 <= wr_ptr_gray;
79             wr_ptr_gray_sync2 <= wr_ptr_gray_sync1;
80         end if;
81     end process;
82
83     DOUT <= memory(to_integer(rd_ptr_bin(0 downto 0)));
84     FULL <= '1' when wr_ptr_gray = (rd_ptr_gray_sync2 xor "10") else '0';
85     EMPTY <= '1' when rd_ptr_gray = wr_ptr_gray_sync2 else '0';
86
87 end BEHAVIORAL;

```

Yocto Recipe Files**audiomini-drivers.bb**

```

1  SUMMARY = "Linux I2C Device Driver for the TPA6130A2 Headphone Amplifier"
2  DESCRIPTION = "Linux I2C Device Driver for the TPA6130A2 Headphone Amplifier"
3  HOMEPAGE = "https://github.com/ADSD-SoC-FPGA"
4  BUGTRACKER = "https://github.com/ADSD-SoC-FPGA/Code/issues"
5  SECTION = "kernel"
6  LICENSE = "GPL-3.0-only"
7  LIC_FILES_CHKSUM = "file://${S}/audio-mini-drivers.service;md5=
    b07de196e86f9e826054228a95139d4c"
8
9  inherit systemd
10
11 SRC_URI = "git://github.com/night1rider/ADSD-SoC-FPGA-Code.git;protocol=https;
    branch=yocto-audio-mini-fixes"
12 SRCREV = "${AUTOREV}"
13
14 S = "${WORKDIR}/git/examples/passthrough/linux/systemd"
15
16 # kernel modules to be installed that need to be built first
17 DEPENDS += " systemd bash"
18
19
20 # Runtime dependencies - ensures these packages are installed when audiomini-
    drivers is installed
21 RDEPENDS:${PN} += " ad1939-audiomini tpa6130a2-audiomini systemd bash"
22
23 PROVIDES = "audiomini-drivers"
24
25 do_install() {
26     # Install the systemd service file
27     install -d ${D}/etc/systemd/system
28     install -d ${D}/usr/local/bin
29
30     # Install the systemd service file and the script to load the drivers via
    insmod
31     install -m 0644 ${S}/audio-mini-drivers.service ${D}/etc/systemd/system/
32     install -m 0755 ${S}/load-audio-mini-drivers.sh ${D}/usr/local/bin/
33
34 }
35
36 # Must specify the files that are installed by the recipe for yocto to track them
    correctly
37 FILES:${PN} += "/etc/systemd/system/audio-mini-drivers.service /usr/local/bin/load
    -audio-mini-drivers.sh"
38
39 # Systemd service configuration
40 SYSTEMD_SERVICE:${PN} += " audio-mini-drivers.service"
41 SYSTEMD_AUTO_ENABLE:${PN} = "enable"

```

audio-mini-image.bbclass

```

1  # audio-mini-image.bbclass
2  # Reusable class for Audio Mini images on DE10-Nano
3  # Provides package management, NFS deployment, and TFTP integration
4

```

```

5  # Inherit core-image for standard image tasks
6  inherit core-image
7
8  # Dependencies
9  DEPENDS += "core-image-minimal virtual/fakeroot-native"
10
11 # Package Management Configuration
12 # Add APT and DPKG package managers to the root filesystem
13 IMAGE_INSTALL:append = " apt dpkg audiomini-drivers"
14 # Add Debian package feed URI for additional packages
15 PACKAGE_FEED_URIS:append = " http://mirror.0x.sg/debian/"
16 # Set base path for package feeds (RPM format)
17 PACKAGE_FEED_BASE_PATHS:append = " rpm"
18 # Define supported architectures for package feeds
19 PACKAGE_FEED_ARCHS:append = " all armhf"
20
21 # Use Debian package format (.deb) for package management
22 PACKAGE_CLASSES ?= "package_deb"
23
24 # Specify output image formats
25 IMAGE_FSTYPES = "tar.gz ext4 wic"
26
27 # NFS deployment variables
28 ROOTFS_TARBALL = "${DEPLOY_DIR_IMAGE}/${IMAGE_BASENAME}-${MACHINE}.rootfs.tar.gz"
29 NFS_SERVER_SCRIPT = "${DEPLOY_DIR_IMAGE}/deploy_nfs.sh"
30
31 # Custom task to deploy the rootfs tarball via script
32 do_deploy_nfs() {
33     if [ ! -f "${ROOTFS_TARBALL}" ]; then
34         bberro "Rootfs tarball not found at ${ROOTFS_TARBALL}"
35         exit 1
36     fi
37
38     if [ -e "${NFS_SERVER_SCRIPT}" ]; then
39         rm -f ${NFS_SERVER_SCRIPT}
40     fi
41
42     #Also make a latest .wic copy for easy access
43     cp -Lf ${DEPLOY_DIR_IMAGE}/${IMAGE_BASENAME}-${MACHINE}.rootfs.wic ${
        DEPLOY_DIR_IMAGE}/${IMAGE_BASENAME}-${MACHINE}.latest.wic
44
45     echo "#!/bin/bash" > ${NFS_SERVER_SCRIPT}
46     echo "sudo rm -rf ${DE10_NANO_NFS_DIR}/*" >> ${NFS_SERVER_SCRIPT}
47     echo "sudo tar --same-owner -xzf ${ROOTFS_TARBALL} -C ${DE10_NANO_NFS_DIR}" >>
        ${NFS_SERVER_SCRIPT}
48     chmod +x ${NFS_SERVER_SCRIPT}
49
50     bbwarn "NFS server script: ${NFS_SERVER_SCRIPT}"
51 }
52
53 # Run after do_image_complete
54 addtask deploy_nfs after do_image_complete before do_build
55 do_build[recrdeps] += "do_deploy_nfs"
56 do_deploy_nfs[depends] += "${PN}:do_image_complete"
57 do_deploy_nfs[nostamp] = "1"
58
59 # Force TFTP deployment tasks from individual components to run
60 do_deploy_nfs[depends] += "linux-socfpga-lts:do_tftp_deploy"
61 do_deploy_nfs[depends] += "audio-mini-bitstream:do_tftp_deploy"
62 do_deploy_nfs[depends] += "de10-nano-audio-mini-devicetree:do_tftp_deploy"

```

audio-mini-passthrough.bb

```
1  # meta-msu-de10-nano/recipes-core/images/audio-mini-passthrough.bb
2  SUMMARY = "Custom image based on core-image-minimal for de10-nano-audio-mini"
3  LICENSE = "MIT"
4
5  # Inherit core-image for standard image tasks
6  inherit audio-mini-image
```

audio-mini-combfilter.bb

```
1  # meta-msu-de10-nano/recipes-core/images/audio-mini-combfilter.bb
2  SUMMARY = "Custom image with CombFilter support for DE10-Nano"
3  LICENSE = "MIT"
4
5  # Inherit the common Audio Mini image functionality
6  inherit audio-mini-image
7
8  # Add CombFilter-specific packages
9  IMAGE_INSTALL:append = " audiomini-combfilter-controller"
```

APPENDIX B

MATLAB SIMULINK HARDWARE DIAGRAMS

This appendix contains full-page versions of all MATLAB Simulink block diagrams used in the hardware accelerator design. Each figure is presented in landscape orientation for improved readability.



