FLEXIBLE PARTIAL RECONFIGURATION BASED DESIGN ARCHITECTURE FOR DATAFLOW COMPUTATION

by

Mihir Shah

APPROVED BY SUPERVISORY COMMITTEE:

Benjamin Carrion Schaefer, Chair

Dinesh K. Bhatia

William P. Swartz, Jr.

To my Parents–Yogini and Rajesh

& my loving Sister–Aishu

FLEXIBLE PARTIAL RECONFIGURATION BASED DESIGN ARCHITECTURE FOR DATAFLOW COMPUTATION

by

MIHIR SHAH, B.Tech

THESIS

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN

ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT DALLAS

May 2018

ACKNOWLEDGMENTS

I would like to thank my thesis advisor Dr. Benjamin Carrion Schaefer for his unparalleled support, motivation and never-ending belief in me. It's been over a year since I first started working with professor Benjamin Carrion Schaefer as an independent research student. I am truly thankful to him for always providing me correct direction. I am highly inspired by his research aptitude and wonderful personality.

I would like to thank Dr. Dinesh Bhatia and Dr. William Swartz, Jr. for being on my committee and spending their quality time to review my work.

I am thankful to Jianqi Chen for his work with respect to benchmark and always helping me debug critical issues. Jianqi is a very kind hearted and generous person.

I would like to thank members of the DARClab: Farah, Siyuan, Pandy, Mahesh, Zhiqi and Zi Wang for always being by my side whenever I needed them. I would also like to thank Sushmitha Gogineni, Abhishek Krishnamurthy and Rohit Somwanshi for helping me realize how engaging and enlightening is the research track. I am thankful to my friends Sagar Patil, Aishwariya Bari and roomates Sagar Kansara, Siddharth Shah, Utsav Dholakia and Sagar Mehta for their moral support and encouragement. I am grateful to Vidhi Vora for always staying besides me through dark phases and providing me warmth.

I am also humbled and blessed by the Hardware Donations made to DARClab, which facilitated faster development. I am very thankful to Amanda, Josefine and Tamra from The Office of Graduate Studies for their patience and untiring efforts in evaluating my Thesis documents. I am grateful to the Department of Electrical and Computer Engineering at The University of Texas at Dallas for their support and research facilities that provided me ease throughout the journey.

January 2018

FLEXIBLE PARTIAL RECONFIGURATION BASED DESIGN ARCHITECTURE FOR DATAFLOW COMPUTATION

Mihir Shah, MSEE
The University of Texas at Dallas, 2018

Supervising Professor: Benjamin Carrion Schaefer, Chair

In this thesis research we proposed a generic semi-automatic partial reconfiguration based design methodology which takes inputs in the form of behavioral description files using C/C++/SystemC for a dataflow process and outputs partial binaries to deploy on the SoC FPGA. This methodology is coupled with a novel static design architectural framework utilizing internal block ram memory to store intermediate results. In order to prove the efficacy of the proposed methodology and architecture in terms of area and timing, we have implemented JPEG Encoder from S2CBench v.2.0 spatially and then with partial reconfiguration design methodologies. The proposed design method abbreviated as PR_{BRAM} where internal FPGA on-chip memory is used to store intermediate results when time multiplexing kernels and PR_{DDR} is a partial reconfiguration based design method utilizing external off-chip DDR memory. The reconfiguration time is a critical parameter determining the performance of DPR designs. Reconfiguration time depends on the area of Reconfigurable Partition (RP) and the generated partial bitstream. Thus, we study and prove experimentally considering equal area of RP for both PR_{BRAM} & PR_{DDR}, that the proposed former method is runtime and latency efficient compared to the latter. We also examine and study the effects of variations on reconfigurable partition area on running time, considering different number of reconfigurations required for the application on the proposed architecture PR_{BRAM}.

We prove that the implementation with the proposed Architecture PR_{BRAM} is area efficient compared to spatial implementation with LUT area savings upto 21.20 % and FF area savings up to 30.41 % for 1598.896 KB as partial bitstream size. These %'s are including the additional resources utilized by proposed static architecture. We also have seen an improvement in average hardware running of 0.529363s against PR_{DDR}.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF NOMENCLATURE

$A_{dynamic}$	Area utilization by the RP on PL
A_{static}	Area utilization by the static logic on PL
A_{total}	combined area utilization by A_{static} + $A_{dynamic}$
AMBA	Advanced Micro-controller Bus Architecture
AXI	Advanced eXtensible Interface
BDL	Behavioral Descriptive language
CWB	CyberWorkBench
DCP	Design Checkpoint
DCT	Discrete Cosine Transform
DFG	Directed Flow Graph
DPCM	Differential Pulse Code Modulation
DPR	Dynamic Partial Reconfiguration
HLS	High Level Synthesis
ICAP	Internal Access Control Port
IP	Intellectual Property
JPEG	Joint Photographics Expert Group
JTAG	Joint Test Action Group
MSE	Mean Square Error

N_{bin}	Number of partial bitstreams corresponding to total number of RMs
PCAP	Processor Access Control Port
PE	Processing Element or Node in DFG
PL	Programmable Logic
PR	Partial Reconfiguration
PR_{BRAM}	PR based design method using internal BRAM memory
PR_{DDR}	PR based design method using external off-chip DDR memory
PS	Processing System
PSNR	Peak Signal-to-Noise Ratio
RLC	RunLength on AC Components
RM	Reconfigurable Modules
RP	Reconfigurable Partition
$RP_{Bitsize}$	Partial Bitstream size in bytes corresponding to RP
RT	Reconfiguration Time
RT_{BRAM}	RT for PR_{BRAM} methodology
S2CBench	Synthesizable SystemC Benchmarks
SoC FPGAs	System-on-Chip Field Programmable Gate Arrays
SSIM	Structural Similarity Index
$Static_{blank}$.bit	Full Bitstream for the PL side - static logic + RP with buffered ports
T_{bin}	Time it requires to load partial bitstream of RM into RP

CHAPTER 1

INTRODUCTION

1.1 Thesis Motivation

When it comes to speeding up computationally intensive workloads, it's not only Graphic Processing Units (GPUs) but also FPGAs (Field Programmable Gate Arrays) that are gaining a lot of attention. Microsoft has been using Altera FPGAs in its servers to run neural networks for Bing Search Engine, Cortana speech recognition and Natural Language(NL) translation [1]. Baidu is also working on FPGAs for its data centers and Amazon Work Spaces (AWS) already offers Elastic Cloud Compute (EC2) F1 instances with Xilinx Virtex UltraScale+ FPGAs [2]. If one has an accelerated workload of a dataflow computation model which is common in multimedia, image processing and signal processing based applications then they have to pay a heavy price to buy more computing space and resources on these data servers, to run their large application. Such dataflow processes usually have their computing elements operate serially in a cascaded fashion and hence except for the current computing element, the remaining ones are in-active especially when the data to be processed and the time it takes to process is relatively large.

Reconfigurable architectures which integrate a hard processor core along with a reconfigurable fabric on a single device, allows faster computation by means of hardware accelerators implemented on the reconfigurable fabric [3]. A dataflow program can be represented as a DFG (Directed Flow Graph) where each node represents an execution kernel/accelerator/actor and each directed edge can be represented as a FIFO (First-In First-Out) queue or buffer [4]. There has been some work that attempted to model DRP (Dynamic Partial Reconfiguration) in a dataflow paradigm [5]. However, the work does not propose a strong foundation for generic dataflow paradigm in terms of architectural issues and neither supports the work with experimental validation. There has been a study recently on implementing partial

reconfiguration for data compression application [6]. This work validates the effects of partial reconfiguration on area, reconfiguration time and power consumption but does not discuss the issues associated with storing intermediate results when swapping different kernels which affects the running time performance. DPR is used extensively in SDR (Software Defined Radio) applications and there has been related work done in [7] but however the architecture proposed utilizes external off-chip memory.

The fundamental ideology behind the proposed design architecture is that in a dataflow computation process, each Processing Elements abbreviated as PE_0, PE_1, PE_2, ... PE_k , where **k** is the number of stages in a dataflow computation, require processed inputs from the previous stage except for terminal Processing Elements - $\mathbf{PE_0}$ **and** $\mathbf{PE_k}$. Thus, in a large dataflow computation process where the signal chaining is intensive for example in the case of SDR applications, several resources on the FPGA fabric area are utilised when the design is implemented spatially leaving not much room for any other compute intensive tasks to be run in parallel. In situations like these, additional FPGA ICs or a larger FPGA with additional resources are added to the PCB Board to balance any additional compute intensive tasks, thereby increasing the raw PCB hardware sizes and routing latency or propagation delay between FPGA ICs.

Partial Reconfiguration has been proposed long ago in order to solve the problem associated with reduced utilization and sharing of FPGA resources and Xilinx, Inc provides extensive support in terms of hardware and software to implement Partial Reconfiguration based designs [8]. With this solution, we can shrink on the FPGA area utilization without a doubt, but however the hardware running time and latencies are affected due to the time it requires to configure the bitstream partially through internal or external interfaces. Also, in a dataflow computation process where the computation is to be done for large amount of data, storing intermediate stage results in an off-chip memory adds additional latency to fetch and write back data. Thus, we propose a partial reconfiguration based design architecture and methodology for dataflow computation process using an internal FPGA BRAM memory.

1.2 Thesis Contribution

The following are the contributions to the Thesis:

• We have proposed a semi-automatic design methodology for a dataflow computation process with partial reconfigurability from SystemC behavior description to partial binaries for Reconfigurable Modules (RMs). This methodology follows a static design architecture to talk to the reconfigurable modules, which is also proposed and is novel.

• We have explored with the help of JPEG Encoder from S2CBench v.2.0 the design implementations and results obtained implementing the design spatially and with the proposed design methodology & architecture.

• We also explore the efficiencies achieved in runtime by changing the reconfigurable partition area in the proposed design architecture.

1.3 Thesis Organization

The organization of the remainder of this thesis is presented in this section. Chapter 2 explains High Level Synthesis design steps and RTL generation followed by discussion on Commercial HLS Tool - CyberWorkBench from NEC Corporation. Chapter 3 discusses SoC FPGA Trends, ZYNQ Architecture overview, PS-PL features in Zynq-7000 AP SoC device XC7Z020 & AXI interconnects as the first half of the chapter. The remainder chapter discusses Partial Reconfiguration (PR) ideology, benefits and methods to configure partial bitstreams for Reconfigurable Modules (RMs). Chapter 4 gives a high level explanation about dataflow computation process and then discusses about S2CBench v.2.0 JPEG Encoder in detail.

Chapter 5 describes the custom IP generation and validation design flow from SystemC to Xilinx SDK to ensure each computing element in a dataflow computation process is validated thoroughly for timing and functional requirements before integrating into more complex

design flows. Thus, a generic method is proposed for any dataflow computation process and is explained in great detail with a running example of lena.bmp test case image block running JPEG Encoder. Finally in Chapter 6, we explore the design implementations using spatial and partial reconfiguration based methods with JPEG Encoder and discuss the proposed design methodology for any dataflow computation process. Comparative results and analysis are done to prove the proposed architecture is efficient and flexible compared to the remaining implementations. Chapter 7 concludes with summary and possible future work originating from this thesis.

CHAPTER 2

HIGH-LEVEL SYNTHESIS (HLS)

2.1 Introduction

The VLSI design complexities have increased in recent decades which have caused design methodologies to raise the level of abstraction to facilitate faster time-to-market delivery. There has been an accelerated growth of automation tools for synthesis and verification process that has allowed designers to explore design space solutions efficiently. In the software domain, the assembly language was introduced in 1950s, before which machine code or binary sequences were used to program a computer [9]. In later years, high-level languages and their respective compilation techniques were developed to improve software productivity. These high-level languages hide details about the architecture, following rules of semantics, provide flexibility & portability as they are platform independent. Design Methodologies have evolved similarly in hardware domain as well [10, 11]. Simulation at the gate-level appeared in early 1970s and cycle based simulation became available by 1979 followed by which in 1980s techniques such as place and route, formal verification, schematic capture and static timing were introduced [9]. Hardware descriptive languages (HDLs) such as Verilog and VHDL have been widely adopted in simulation tools and are used as inputs to logic synthesis tools. During the 1990s, the first generation of HLS tools was available commercially [12, 13].

2.2 High-Level Synthesis Design Steps

Raising the level of abstraction for a hardware design is essential in order to evaluate architectural decisions such as resource utilization, power management, memory hierarchies, compiler & software support etc. HLS also provides re-usability of high-level specifications based on design constraints for any FPGA or ASIC technology. Thus, if an RTL is generated

for a specific target then it is easier to generate the same for another, just by altering library & constraints files.

Figure 2.1 shows the High-Level Synthesis Design Flow. Typically, a designer begins the specification of an application that has to be implemented such as a custom processor or any hardware unit with a high-level description using for instance Behavior Descriptive Language (BDL) C/C++/SystemC that captures desired functional behavior. This first step thus involves writing a functional specification which can also be called as 'an un-timed behavior description' in which a function collects all its input data, performs all computations and provides all its output data minimizing the execution time. The I/O operations can occur concurrently or sequentially depending upon the requirements.

Several optimizations such as false data-dependency elimination, constant folding and loop transformations are being done when the compilation occurs. The formal model generated after compilation involves control and data dependencies in the form of a dataflow graph (DFG) in which all the inter-dependencies in the behavior description is captured. Data dependencies can be easily represented with a DFG in which every node represents an operation and the arcs between the nodes signifies the input, output and temporary variables [14].

The following are the three main steps involved in High-Level Synthesis: Allocation, Binding and Scheduling.

Allocation

Allocation defines the type and the number of hardware resources, for instance: functional units, storage or connectivity components needed to satisfy the design constraints. Depending on the HLS tool, some components may be added during scheduling and binding tasks. For example, the connectivity components (such as buses or point-to-point connections among components) can be added before or after binding and scheduling tasks. The components are

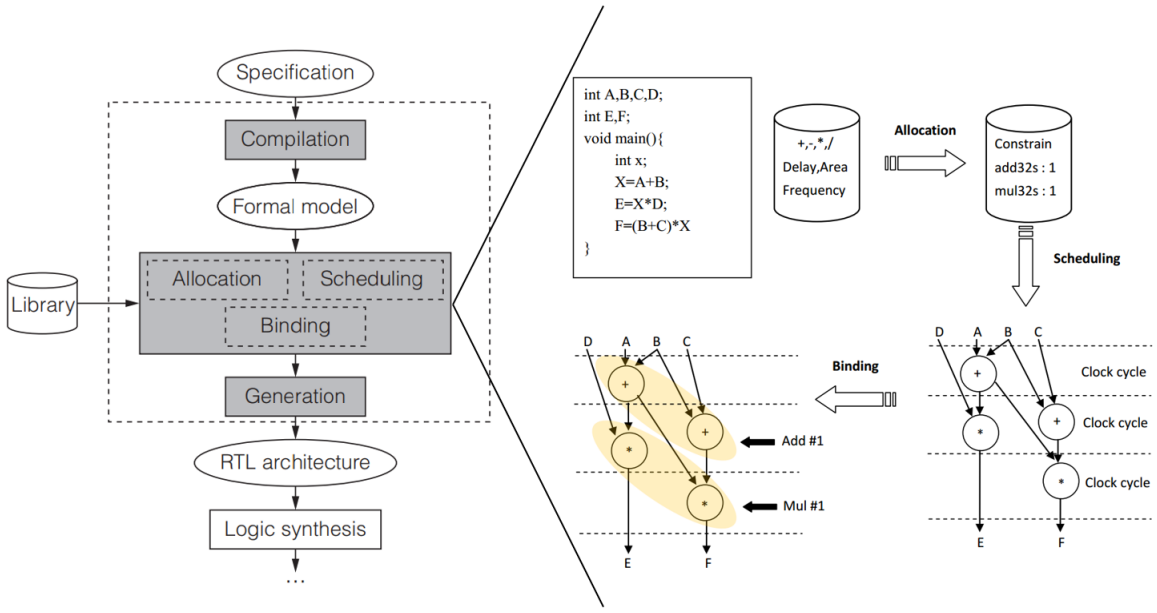


Figure 2.1: High Level Synthesis Design Steps

selected from the RTL component library which also contains component characteristics such as area, delay and power. By default the HLS tool will try to maximize the parallelism as much as possible in the scheduling stage and hence allocating large number functional units is always an advantage unless there is a design restriction.

Algorithm 1 Example Code to Explain Scheduling & Binding [15]

```
1: int foo (char x, char a, char b, char c) {
2: char y;
3: y=x*a+b+c;
4: return y;
5: }
```

Scheduling

All operations required in the specification model or the dependency graph must be scheduled into cycles in order to find the latency and overall compute time required for the entire process. Scheduling determines for each operation the exact clock step at which it will be

executed such that no precedence constraint is violated. For each operation such as a = b **op** c, variables 'b'and 'c' must be read from their sources and brought to the input of a functional unit **op** to perform execution and the result 'a' must be brought to its destination.

Consider an example of $y=x*a+b+c$ as shown in Algorithm 1 where each input variable is of datatype `char` which can be scheduled as shown in Figure 2.2 [15].

Depending on the functional component to which the operation is mapped, the operation can be scheduled within one clock cycle or scheduled over multiple cycles referencing the library files. Operations can also be chained where the output of an operation directly feeds as an input to another operation. Operations can be scheduled to execute in parallel provided that there are no data dependencies between them and there are sufficient resources available at the same time [9].

Many scheduling algorithms have been proposed in HLS [16]. There are two most basic scheduling algorithms, As Soon As Possible (ASAP) and As Late As Possible (ALAP). ASAP algorithm as the name suggests schedules operations in earliest possible time-step, as long as an operation is scheduled if and only if all its predecessors are scheduled in earlier control steps. In As-Late-As-Possible (ALAP) scheduling initially maximum number of time-steps that are allowed is determined, which is usually a constraint set by the tool or the user, after which the algorithm schedules each operation, one at a time into the latest possible time-step.

Binding

Every operation in the specification model must be bound to one of the functional units available which is capable of executing the operation. If there is a variable that carries values across cycles then it must be bound to a storage unit. Moreover, several variables with non-overlapping or mutually exclusive lifetimes can be bound to the same storage units [9].

Consider again the example of $y=x*a+b+c$ as shown in Algorithm 1 where each variable is of datatype `char`. Figure 2.2 [15] shows the binding phases -initial and final. Clock cycle

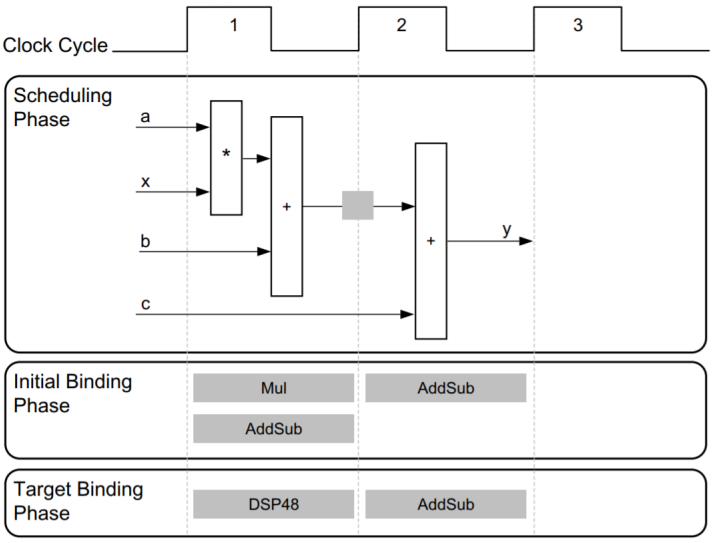


Figure 2.2: Scheduling and Binding Example

1 reads variables 'a, x, b' then does the multiplication and first addition. Clock cycle 2 reads the 'intermediate result(depicted by grey box in the figure), c' and does the second addition followed by generating output. In the initial binding phase, HLS implements the multiplier operation using a combinational multiplier denoted as 'Mul' and implements both add operations using a combinational adder subtractor denoted as 'AddSub'. In the target binding phase, HLS implements DSP48 resource (considering target FPGA is from Xilinx, Inc) instead of 'Mul'& 'AddSub' in Clock Cycle 1 for better improved performance. Also since every input variable is of datatype `char`, the input data ports of 8-bit width would be generated by the HLS tool. Since, the function return is integer datatype the output port would be a 32-bit width.

2.3 RTL Generation

After allocation, scheduling and binding, the next step in HLS process is to generate the RTL. The goal of the RTL architecture generation step is to apply all the design decisions made in previous stages and generate an RTL model of the synthesized design. The RTL

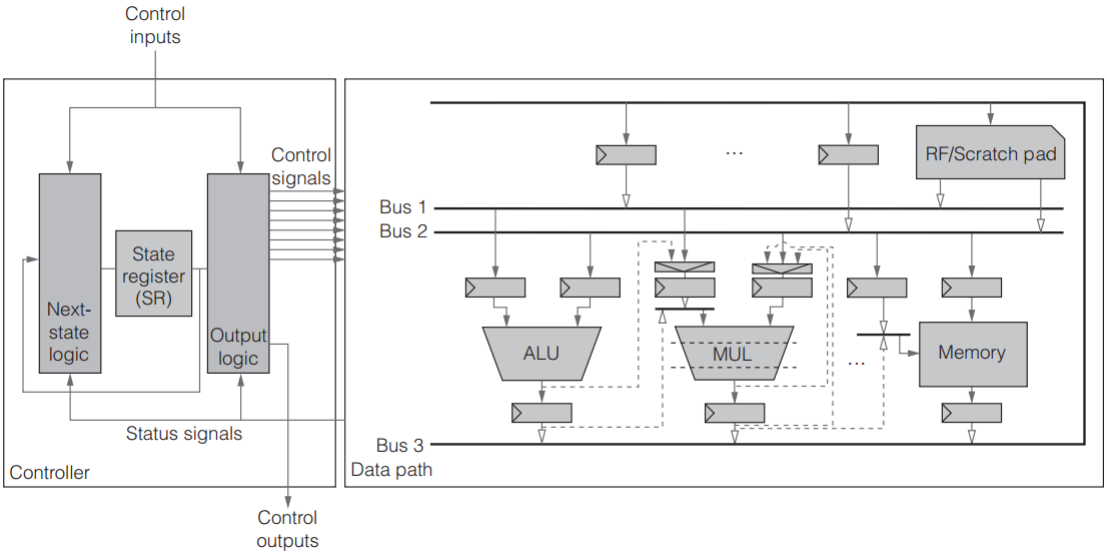


Figure 2.3: Typical Architecture

architecture consists of a controller and a data path usually as shown in Figure 2.3 [9]. A data path consists of a set of storage elements such as registers, register files & memories, a set of functional units such as ALUs, multipliers, shifters and other custom functions and interconnect elements such as tristate drivers, multiplexers and buses. The controller is a finite state machine that controls the flow of data in the data path by setting the values of control signals such as the select inputs of functional units, registers, and multiplexers. The inputs to the controller may come from primary inputs (control inputs) or from the data path components example comparators (status signals).

All these register-transfer components can be allocated in different quantities and types and connected arbitrarily through buses. Each component can take one or more clock cycles to execute, can be pipelined and can have input or output registers. In addition to this, the entire data path and controller can be pipelined in several stages. Data inputs and outputs are connected to the data path and similarly control inputs and outputs are connected to the controller.

2.4 Commercial HLS Tool

There are several commercially availble HLS Tools like Xilinx Vivado HLS, Cadence's Stratus, Mentor's Catapult, NEC's CyberWorkBench which take input in the form of C/C++/SystemC.

2.4.1 NEC's CyberWorkBench HLS Tool

For the purpose of our research study, we use NEC's CyberWorBench HLS Tool. NEC have been developing C-based behavioral synthesis called 'Cyber' since the late 80's and C-based verification tools such as formal verification and simulation around 'Cyber' during the last 20 years which have been integrated into an IDE known as CyberWorkBench (CWB). It is C-based High Level Synthesis and Verification "All-in-C" Tool and supports any ASIC Technology & Altera/Xilinx FPGAs.

Figure 2.4 shows the HLS Flow using CWB. Hardware is described at behavioral level using BDL languages and analysed using **bdlpars** to generated internal format files to be used as input for behavioral synthesis. Before executing the behavioral synthesis, resources needs to be allocated so that the scheduling stage can time the operations depending upon the dependency graph as explained in previous sections. Thus, constraint files are generated containing the functional units type & count appropriately for behavioral synthesis using non-use mode when behavioral description is analyzed in Cyber Behavioral synthesis system. There is also an option to set the scheduling mode - either manual or automatic. By default, the synthesis mode is set to **manual scheduling** mode where the execution timing for each operation can be explicitly specified by setting the clock cycle boundary "$" in the functional description. This mode is suitable when the timing are generally pre-determined. In our research study, we have used **automatic scheduling** mode to allow CWB to optimize latency. In automatic scheduling mode, circuits are initially synthesized to minimize the number of execution cycles within the scope of the specified constraints such as clock cycles,

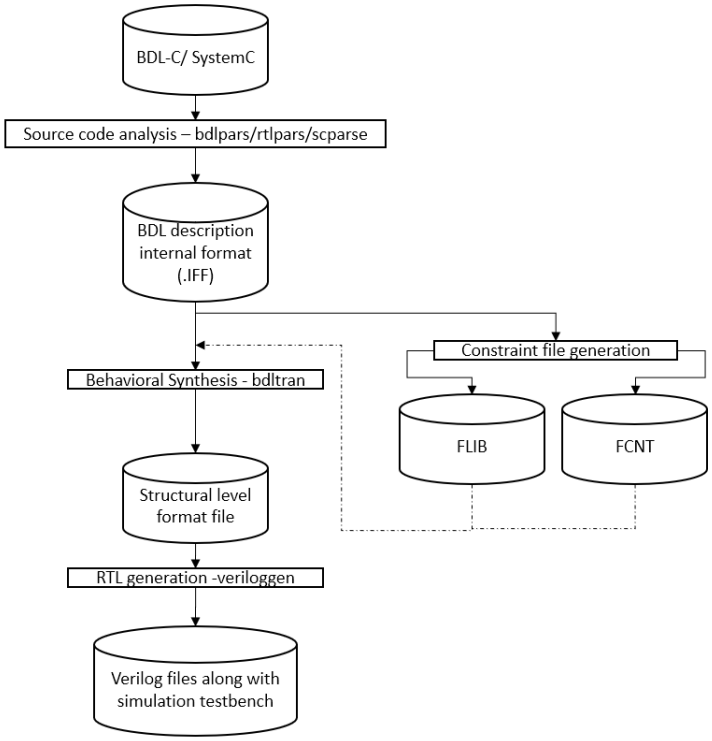


Figure 2.4: CWB HLS Design Flow

the limitations of functional units etc. Later, the circuits are configured to minimize the area requirement by sharing of registers, functional units and other resources. Using the functional and memory constraints file along with internal format file generated after bdlparse, the behavioral synthesis is executed using **bdltran** where scheduling and binding steps takes place. This step generates an internal structural file which is used to generate the RTL Architecture as shown in Figure 2.3 using **veriloggen**.

2.5 Summary

HLS tools transform an untimed high-level behavior specification into a fully timed implementation [17, 18]. One of the main advantage of HLS vs. traditional RTL design methods is that HLS allows the generation of different micro-architectures with unique area vs. performance trade-offs without having to modify the original behavioral description.

CHAPTER 3

PROGRAMMABLE SOC FPGA AND PARTIAL RECONFIGURATION DESIGN FLOW

3.1 Introduction

In this chapter, we will examine the potential benefits of having a Processor System along with Programmable Fabric in a single silicon chip. We will also discuss Partial Reconfiguration based design methodology which helps to time multiplex several kernels within a compact space.

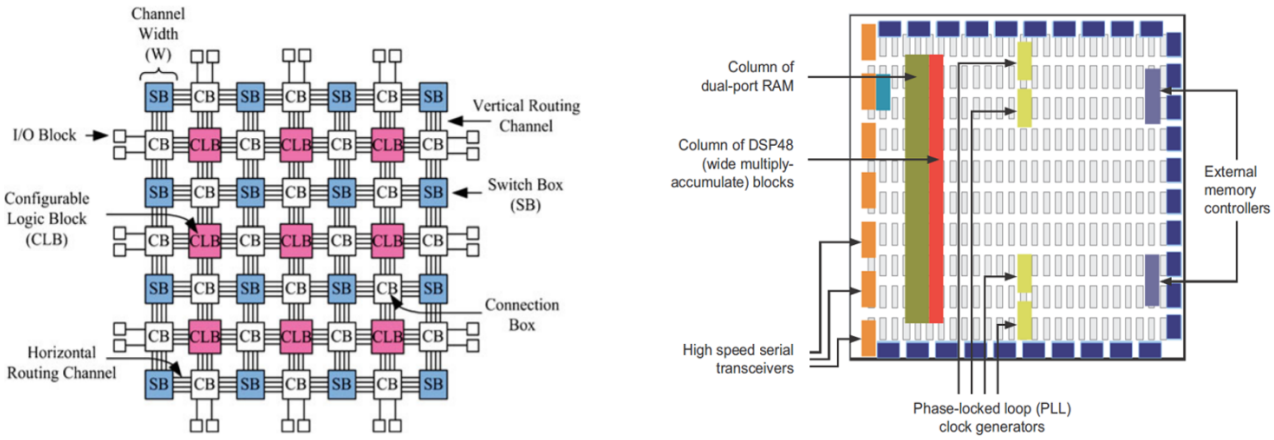


Figure 3.1: Traditional Island-Style FPGA (Left) and Modern FPGA (Right)

Figure 3.1(Left) shows a traditional island-style FPGA architecture which is also termed as mesh based FPGA architecture. It is called island-style architecture because in this architecture, Configurable Logic Blocks (CLBs) are arranged on a 2D-grid resemble islands in a sea of routing interconnect. The Input/Output (I/O) blocks on the periphery of FPGA chip are also connected to the programmable routing network. The routing network comprises of pre-fabricated wiring segments and programmable switches that are organized in horizontal and vertical routing channels [19].

Contemporary FPGA architectures incorporate the basic elements as discussed in island-style fpga along with additional computational and data storage blocks that increase the computational density and efficiency of the device. These additional elements include: Embedded memories for distributed data storage, Phase-locked loops (PLLs) for driving the FPGA fabric at different clock rates, High-speed serial transceivers, Off-chip memory controllers, Multiply-accumulate blocks etc as shown in Figure 3.1(Right).

3.2 SoC FPGAs

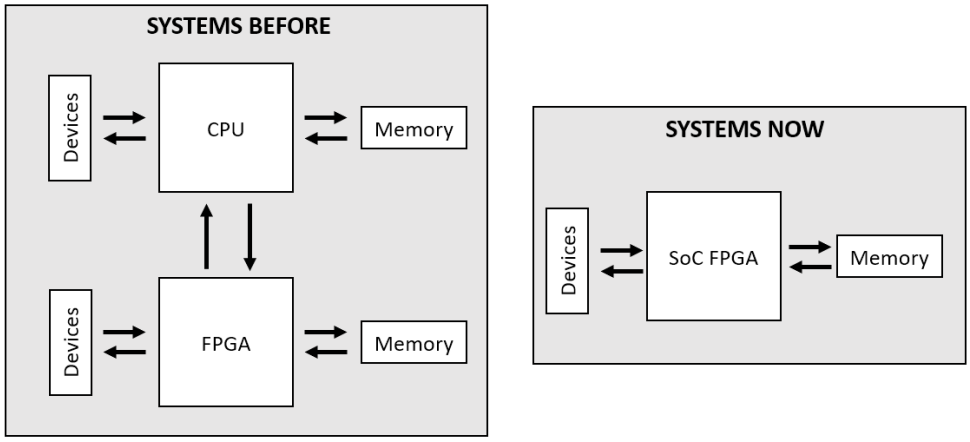


Figure 3.2: SoC FPGA Trend

System-on-Chip (SoC) FPGA devices integrate both processor and FPGA architectures into a single IC which provides lower power, smaller board size, better integration and higher bandwidth communication between the processor and FPGA [20]. SoC FPGAs helps integrate the best of both the worlds in processor domain for high-management of tasks and FPGAs for real time speedy data processing. Thus, if there are certain tasks in a computer program which are data-intensive, then they can be spin-off by the processor to the FPGA fabric to achieve maximum parallelism and faster throughput rate.

At present, there are three vendors of SoC FPGAs available in the market namely Altera SoC, Xilinx Zynq 7000 and Microsemi SmartFusion2. The processors in these devices are

fully dedicated, "hardened" processor subsystems unlike softcores implemented on FPGA fabric. If both the CPU and FPGA use separate external memories it may also be possible to consolidate both into one memory device to reduce cost as shown in Figure 3.2. As the signals between the processor and the FPGA now reside on the same silicon, communication between the two consumes substantially less power compared to using separate ICs. The integration of thousands of internal connections between the processor and the FPGA leads to substantially higher bandwidth and lower latency compared to a two-chip solution [20].

For the purpose of our study, we have selected Xilinx Zynq-7000 All Programmable (AP) SoC FPGA device part : XC7Z020, which is available with Zedboard from Digilent, Inc. Hence, for the purpose of description about the Zynq-7000 AP SoC in general, we would be referring to this part alone. It is All Programmable - meaning that not only can one add systems intelligence through software, but additional data processing and decisions can be executed in real time with programmable hardware [21]. All this intelligence can be added with reduced design cost and tremendous flexibility to change the design or upgrade on-site.

3.2.1 Altera SoC FPGAs

Although Altera has launched Startix & Arria 10 SoC FPGA devices, Cyclone & Arria V, based on 28-nm technology, still remain popular in the academia due to their lower cost and rich features. Figure 3.3 shows the Hard Processing Side (HPS) System and its interconnections to the FPGA fabric. The Processor Architecture features a dual core ARM Cortex-A9 MPCore with 32 KB instruction & data L1 cache and 512 KB L2 shared cache, where it has one AMBA AXI master port connected to Level3 (L3) interconnect and another connected to SDRAM Controller as seen in the figure.

Altera SoC FPGAs provide performance which cannot be matched with respect to a multi-chip solution of FPGA and processor because the former achieves superior performance and low latency with high-throughput compared to latter[22]. There are three HPS-FPGA

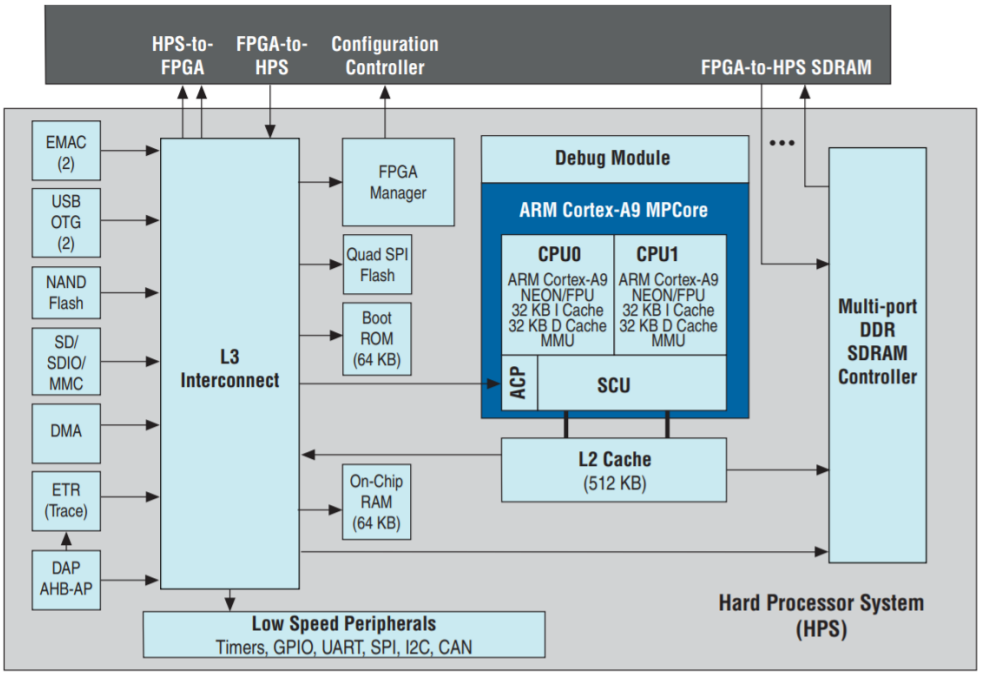


Figure 3.3: Altera SoC FPGA Architecture for Cyclone & Arria V

Bridges which support AXI AMBA bus protocol and allows peripherals on the opposite sides to be accessed on the respective sides:

• FPGA to HPS AXI Bridge: This is a high-performance bus supporting 32,64 and 128-bit data widths allowing FPGA fabric to be the master to slaves on HPS. This interface allows FPGA to have full visibility into the HPS address space.

• HPS to FPGA AXI Bridge: This is a high performance 32,64 and 128-bit data widths that allow HPS to be master to FPGA fabric slaves

• Lightweight (LW) HPS to FPGA Bridge: This is a low performance 32-bit width AXI bridge where HPS is the master.

3.2.2 Xilinx ZYNQ Architecture

Figure 3.4 [23] illustrates the functional blocks of the Zynq-7000 AP SoC. The PL and PS are on separate power domains, enabling power down on the PL-side if required for power management. The Zynq-7000 AP SoC is composed of the following major functional blocks:

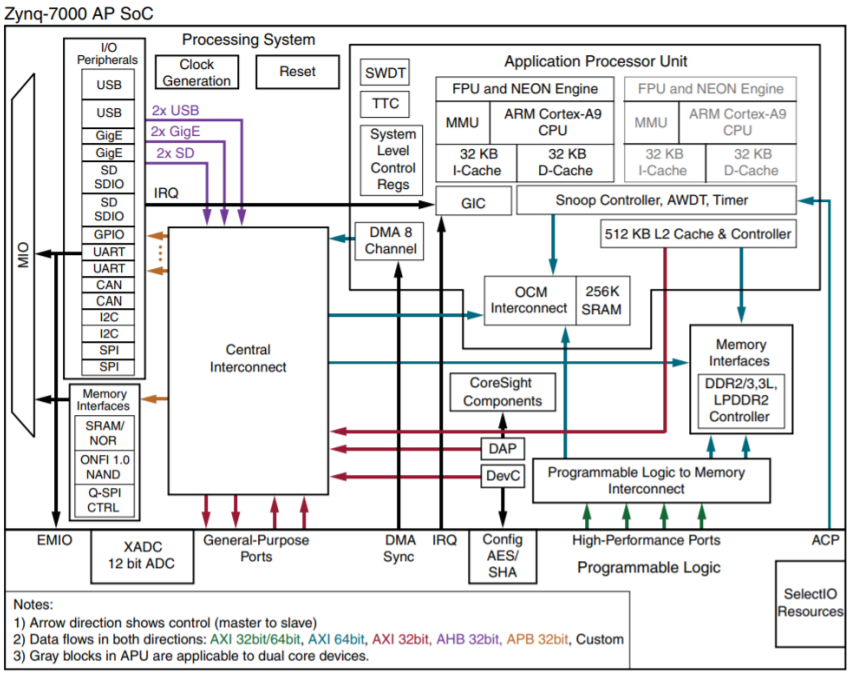


Figure 3.4: Zynq-7000 AP SoC

(a) Processing System (PS) - Application processor unit (APU), Memory interfaces, I/O peripherals & Interconnect and (b) Programmable Logic (PL).

Processing System

The APU consists of Dual ARM Cortex-A9 MPCore CPU with version 7 ARM ISA [23]. There is 32KB instruction and 32KB data L1 cache with parity per MPCore. There is also a shared L2 cache with parity of size 512 KB. The APU system feature highlight includes Snoop Control Unit to maintain L1 and L2 coherency, Accelerator Coherency Port (ACP) from PL to PS, 256 KB of On-Chip SRAM (OCM) to store user application, DMA and interrupt controller.

Zedboard has 512 MB (128M X 32) DDR3 and 256 Mb QSPI Flash external memory. Thus, the PS - side of the Zynq-7000 consists of memory controller interfaces for DDR3 and Quad SPI. The DDR3 controller supports 16b or 32b wide access, uses upto 73 dedicated PS pins and obeys AXI ordering rules. The DDR3 Controller Core does transaction scheduling

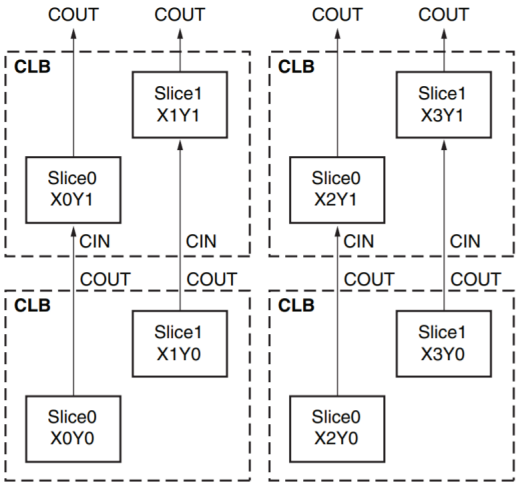


Figure 3.5: Row and Column Relationship between CLB and Slice in Xilinx 7-series FPGA

to optimize the data bandwidth and latency along with advance re-ordering engines to maximize memory access efficiency. Since the device density on Zedboard for the Flash Chip is > 128 Mb, the QSPI Controller supports linear mode. The maximum Quad-SPI clock at master mode is 100 MHz. The QSPI Controller supports 32-bit AXI linear address mapping interface for read operations. The I/O Peripherals are industry standard interfaces for external data communication which include GPIOs, Gigabit Ethernet Controllers, USB Controllers, SD/SDIO Controllers, SPI, UART, CAN & I2C Controllers.

Programmable Logic

The PL of XC7Z020 is based on Artix-7 FPGA logic [23]. The PL resources primarily include 13,300 Logic Slices, 53,200 LUTs, 1,088 LUTRAMs, 106,400 Flip-flops, 140 Block RAM and 220 DSP48E1.

A CLB element contains a pair of slices and each slice is composed of four 6-input Look Up Tables (LUTs) and eight storage elements [24]. The slices are defined as SLICE(0)- Slice at the bottom of the CLB and in left column whereas SLICE(1) - Slice at the top of the CLB and in the right column. Figure 3.5 shows the relationship between CLBs and Slices in terms of Rows & Columns. Each Slice, has an independent carry chain, is organised as a

column and do not have direct connection with each other.The LUTs have memory capability within them and between 25-50% of all the slices can use their LUTs as distributed 64-bit RAM or shift data with 32-bit registers. Slices that support these additional functions are called SLICEM; others are called SLICEL. SLICEM represent a superset of elements and connections found in all slices.

AXI Interconnect

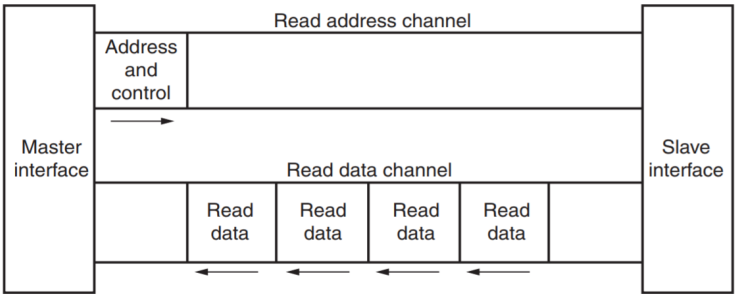
Xilinx adopted the Advanced eXtensible Interface (AXI) protocol for Intellectual Property (IP) cores for Zynq-7000 AP SoC devices with AMBA 4.0 [23]. AXI is part of the ARM AMBA, a family of micro-controller buses first introduced in 1996. There are three types of AXI4 interfaces viz; **AXI4**: for memory-mapped interfaces allowing high throughput bursts of upto 256 data transfer cycles with a single address phase, **AXI4-Lite**: for light-weight, single transaction memory mapped interface and **AXI4-Stream**: allows unlimited data burst size without need for address phase because of which they are not considered memory-mapped interface. Besides providing the right protocol for the application, the AXI4 also helps in improving productivity by making a single protocol for IP across developers as well as improves availability of more third party IPs output Vivado IP Catalog for design ease.

Figure 3.6 shows five AXI bus channels : Read Address Channel, Write Address Channel, Read Data Channel, Write Data Channel, Write Response Channel. Each transmission channel are single direction. Every transaction has unique address and control information which is used to describe the property of the data being transmitted.

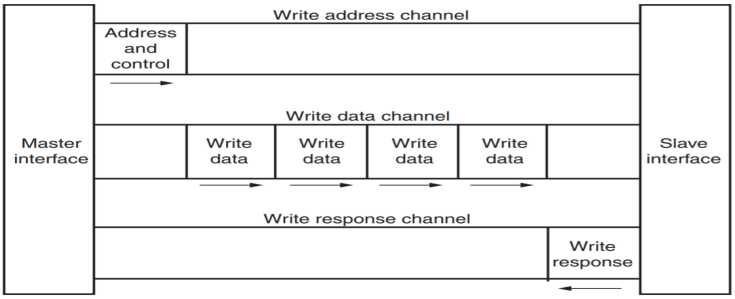
3.3 Partial Reconfiguration Design Process

3.3.1 Basic Ideology of Partial Reconfiguration

FPGA technology provides the flexibility of on-site re-programming without going through re-fabrication, unlike ASIC technology, if the design is altered. Partial Reconfiguration



(a)



(b)

Figure 3.6: (a) Chanel Architecture of Read (b) Chanel Architecture of Write

(PR) takes this flexibility one step further, allowing the modification of an operating FPGA design by loading a partial configuration file, usually a partial BIT file. After a full BIT file configures the FPGA, partial BIT files can be downloaded to modify reconfigurable regions in the FPGA without compromising the integrity of the applications running on those parts of the device that are not being reconfigured i.e the static region. As shown in Figure 3.7a, the function implemented in Reconfig Block A is modified by downloading one of several partial BIT files A1.bit, A2.bit, A3.bit or A4.bit. The logic in the FPGA design is divided into two different types, reconfigurable logic and static logic. The gray area of the FPGA block represents static logic and the block portion labeled Reconfig Block "A" represents reconfigurable logic. The static logic remains functioning and is unaffected by the loading of a partial BIT file. The reconfigurable logic is replaced by the contents of the partial BIT file. Figure 3.7b shows the Vivado Software design flow for PR Implementation [25].

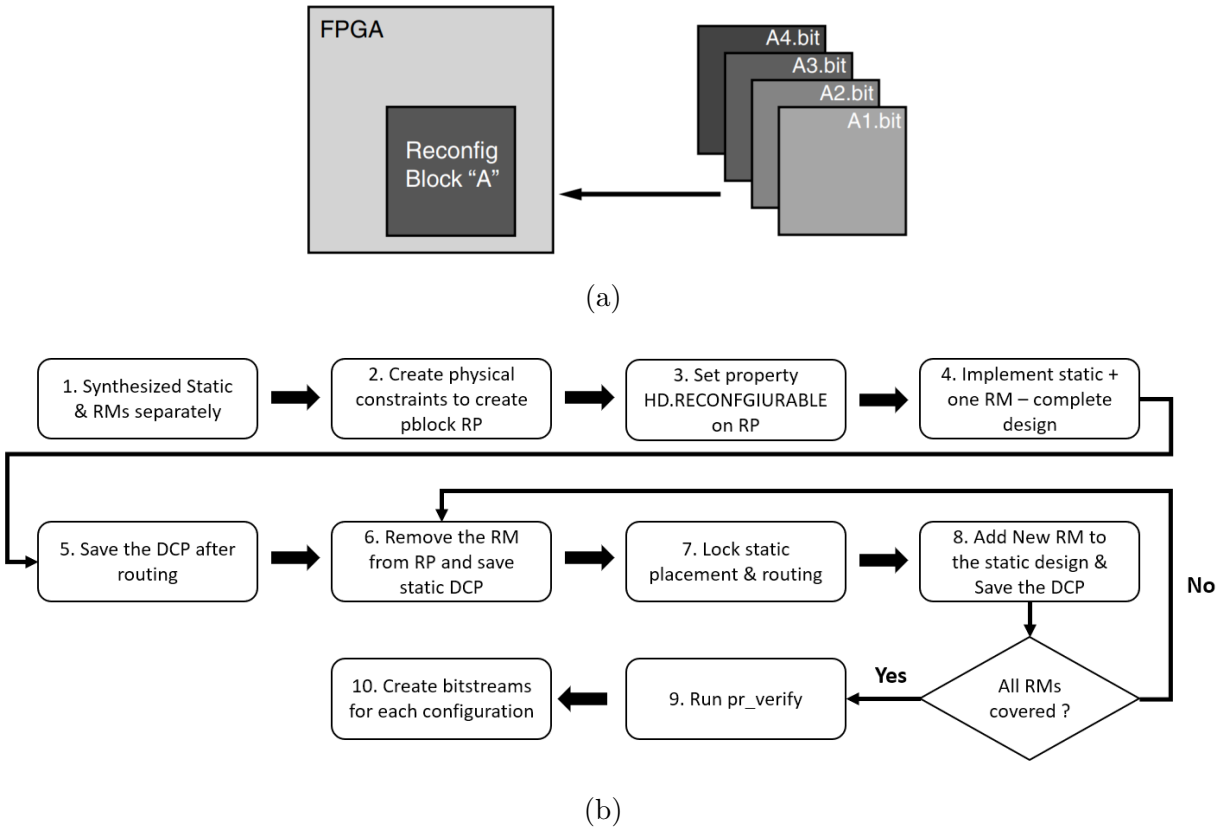


Figure 3.7: (a) Partial Reconfiguration Basic Ideology (b) PR Design Flow in Vivado Software

3.3.2 Configuring Bitstreams in PR Design

Full vs Partial Bitstream Configuration

Figure 3.8 shows the difference in contents of the partial and full bitstreams. A full bitstream consists of a header, configuration data and checksum. This is to verify the integrity of the bitstream and configure the complete FPGA fabric with the logic design. At the end of the configuration process, the FPGA asserts the 'DONE' signal and enters the user mode from the configuration mode where design starts functioning unless if the bitstream was corrupted. This ensures incorrect designs never start functioning on the FPGA [23, 25].

Partial bitstreams, on the other side, contain only configuration data and checksum. Contrast to full configuration method where FPGA is already in user mode with an operating

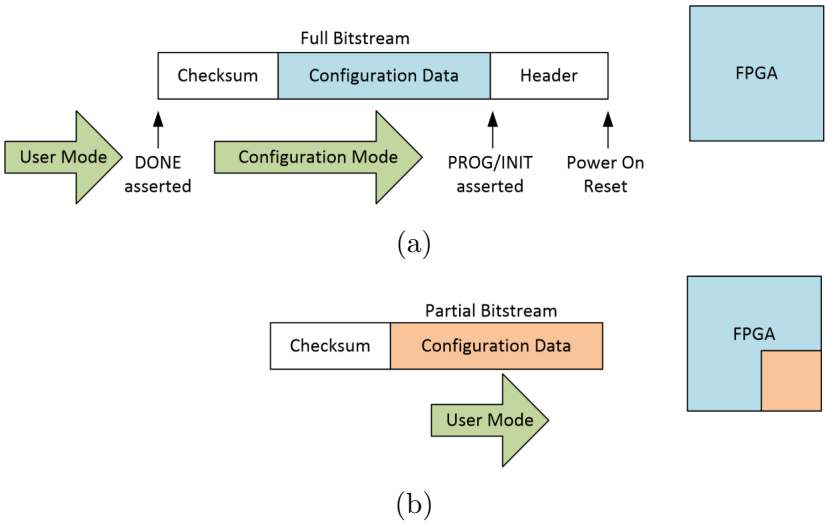


Figure 3.8: Difference between (a) Full Bitstream Configuration (b) Partial Bitstream Reconfiguration

design when the reconfiguration process takes place, asserting the DONE signal is not essential in here. The reconfiguration process lasts till the partial bitstream is entirely sent to the configuration port. Checksum failures indicate corrupt bitstreams and in case of partial bitstreams, they are a bigger concern because if the checksum failure is caused due to erroneous configuration data, the incorrect design is isolated within the reconfigurable partition. This can be corrected at the cost of loading another partial bitstream for the respective partition. However, if the the failure is caused due to erroneous frame address of the reconfigurable partition, then the static logic may be corrupted as a result of the reconfiguration [3]. To avoid such mishaps it is advisable that their integrity is checked before directing them to configuration ports [25].

Methods for Configuring Partial Bitstream

Configuration ports are responsible for configuring the FPGA fabric after verifying the integrity of the bitstreams. Generally, a processor or a state machine is used for fetching the partial bitstreams from non-volatile memory viz sd-card, qspi memory etc and directing

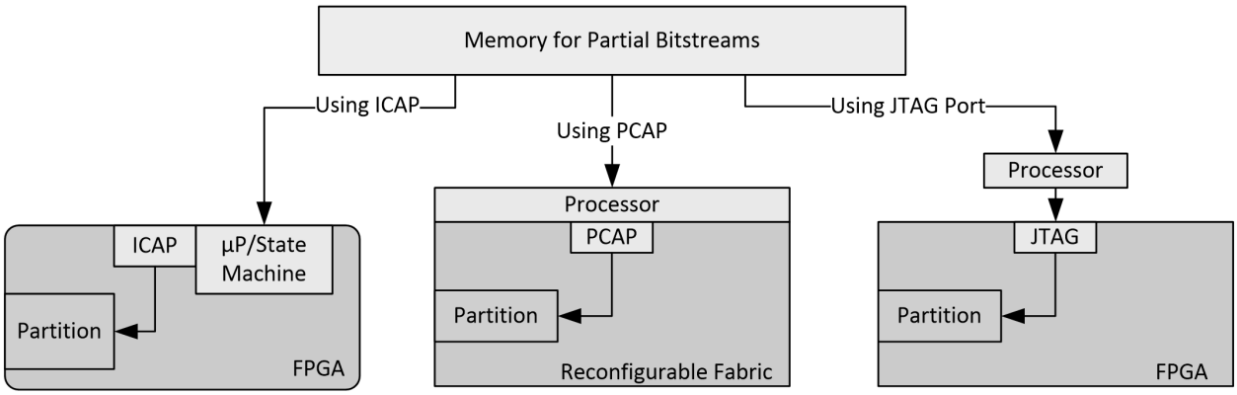


Figure 3.9: Configuration Methods for PR Design (a) ICAP (b) PCAP (c) JTAG

it to configuration ports for reconfiguration [25]. **Thus, having a Processing System (PS) on SoC FPGA can provide an advantage of transferring partial bitstream to configure the Reconfigurable Partition (RP), without the need to add an extra off-chip processor on hardware**. Different configuration ports can be used for performing partial reconfiguration of an FPGA.

The following are the three most commonly utilized methods:

• **Internal Configuration Access Port (ICAP):** This port enables partial reconfiguration within the FPGA, thus allowing self configuring FPGA designs. Self configuring FPGA designs with Xilinx Spartan III families have been presented in [26]. Designs for improving the fault tolerance of the ICAP has been presented in [27].

• **Processor Configuration Port (PCAP):** This configuration interface is used by the runtime reconfigurable architectures which integrate a hard processor core. The processor configures the reconfigurable fabric using this port [25, 23].

The ICAP and PCAP are internal configuration methods to load the partial bitstream.

• **JTAG Port:** This is an interface for quick testing of partial reconfiguration using external port method. Processors can be used for fetching the bitstreams from the memory and directing it towards the JTAG port. Different tools are available for this purpose [25]. The Figure 3.9 shows the usage of the three mentioned configuration ports.

3.3.3 Benefits of Partial Reconfiguration

PR based design implementations open new horizons of flexible usage in wide variety of applications. The benefits of using partial reconfiguration based designs are as follows:

• **Reduced Resource and Power Consumption:** With partial reconfiguration, it is possible to time multiplex several hardware modules. This reduces the total resource requirement for implementing any hardware design which directly translates into power and cost savings. For systems using multiple FPGAs, partial reconfiguration provides the possibility of integrating the design into a lower FPGA IC count. For such systems, power savings can be obtained not only from the reduced count of FPGAs but also from the reduction in off-chip communication [28].

• **Performance Improvements and Flexibility:** With partial reconfiguration, the computation capacity of the system can be adapted at run time. For instance in systems where different hardware kernels are used sequentially one after the other the additional resources can be used for speeding up the operation of the kernel or for creating more number of kernels to perform the operation in parallel [8, 3].

• **Improved Fault Tolerance and Dependability:** Fault tolerance is a highly important criterion for safety critical systems especially in aerospace & defense industries. System failures on account of hardware faults could be fatal in such systems. With partial reconfiguration, fault tolerance and dependability of the systems can be improved by means of techniques like module diversification, configuration scrubbing [29, 30]

• **Self Adapting Hardware Designs:** With partial reconfiguration hardware architectures can adapt themselves to changing operating and environmental conditions if required based on artificial intelligence and learning [31].

3.4 Summary

In this chapter, we discussed FPGA trends and more elaborately looked at Zynq SoC FPGA, which we would be using for our experimental studies. Also, in this chapter we have discussed partial reconfiguration ideology and benefits in brief.

CHAPTER 4

DATAFLOW COMPUTATION

4.1 Introduction

Dataflow Computing (DC) is a specific model of computation in which the target application is described as an appropriate data-flow graph (DFG) where nodes represent portions of computation or tasks and links represent the flow of intermediate results from input to output [32]. Data is streamed from memory onto the chip where operations are performed and data is forwarded from one functional unit to another as results are needed, without ever being written to the off-chip memory until the processing chain is terminated [33]. The dataflow models are intuitive and easy to understand especially in areas of digital signal processing [34]. Dataflow or Stream Computing have two types of parallelism; task and data parallelism. Task parallelism can be exploited by implementing each kernel or compute element as an independent processing unit and scheduling computation in a pipelined fashion. Data parallelism can be exploited at the kernel level because stream elements are independent.

FPGAs have several desirable properties for dataflow or stream processing applications. High-frequency trading is a good example, where high-rate data streams need to be processed in real time and microsecond latencies determine success or failure. I/O capabilities of FPGAs, allow for flexible integration, e.g. in the case of high-frequency trading, the FPGAs are inserted directly into the network, enabling most efficient processing of network traffic [35]. Furthermore, the reprogrammability of FPGAs makes it possible to quickly adapt to market changes.

Figure 4.1 shows the detailed steps required to perform encoding for JPEG. These steps can be divided to form each processing elements in the dataflow graph. Figure 4.2 shows JPEG Encoder as a dataflow process, which we will be analyzing for our research study, where nodes - DCT, Quantz, RLE and Huffman are tasks or computation elements and the interconnect between them represents the stream of data.

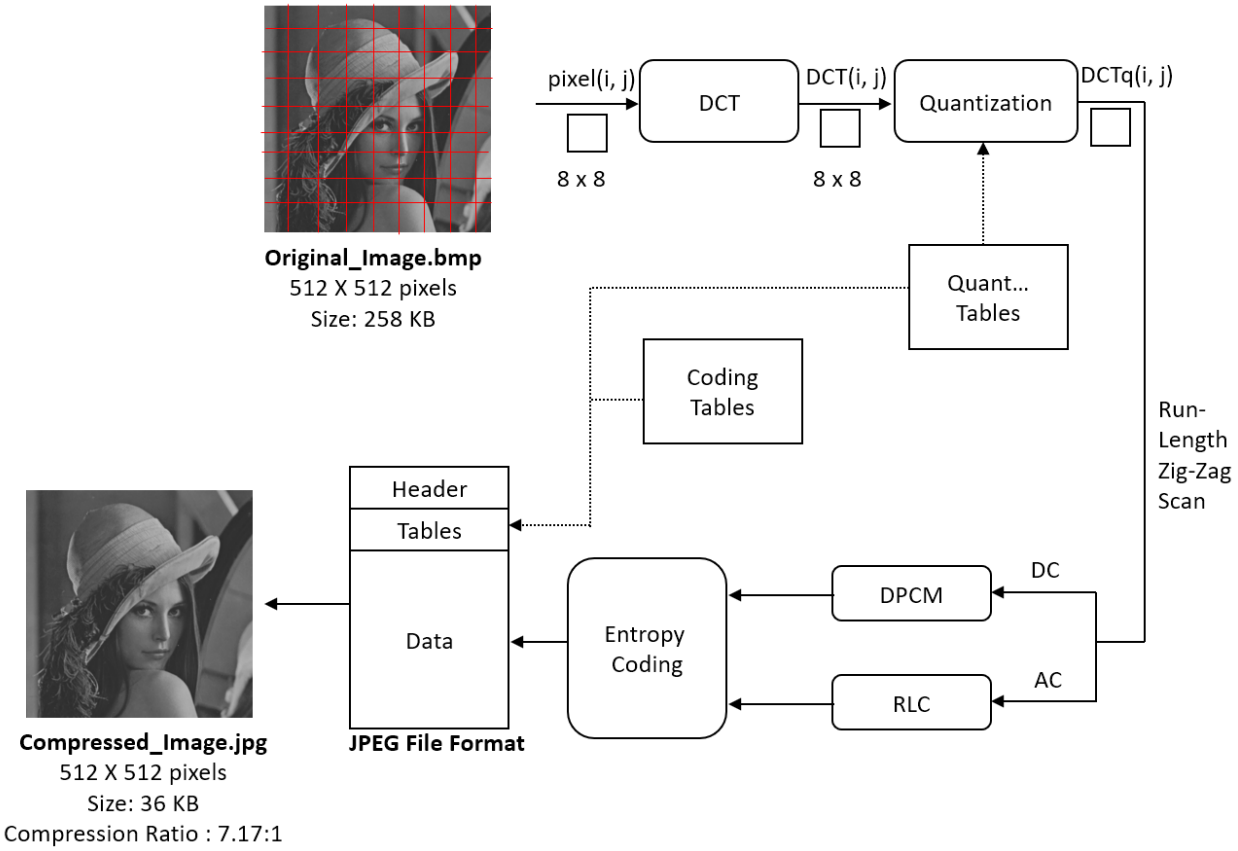


Figure 4.1: Detailed Block diagram of Diagram JPEG Encoder

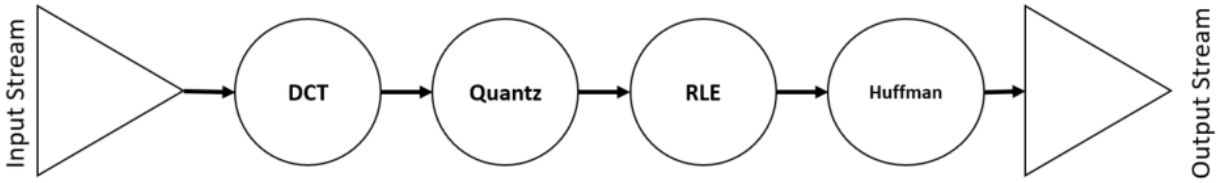


Figure 4.2: JPEG Encoder as Dataflow Computation Process

4.2 S2CBench v.2.0: JPEG Encoder

S2CBENCH stands for Synthesizable SystemC Benchmark suite. It is a open source SystemC benchmarks created to help designers evaluate the QoR of state of the art HLS Tools. S2CBench v.2.0 provides 18 programs written in synthesizable SystemC language. Each

benchmark is designed for specific domains such as multimedia, digital signal processing, security, image processing etc. JPEG Encoder was added to the second revision of the benchmark suite which we will be using in our research study as a dataflow computation process.

JPEG (Joint Photographic Experts Group) Encoder used for our comparative study is a lossy version of Encoder algorithm for images. A lossy Encoder scheme is a way to inexactly represent the data in the image, such that less memory is required yet the data appears to be very similar to the original data. This is why JPEG images will look almost the same as the original images from which they are derived most of the time, unless the quality is reduced significantly, in which case there will be visible differences. The JPEG algorithm takes advantage of the fact that humans cannot see colors at high frequencies. These high frequencies are the data points in the image that are eliminated during the Encoder.

4.2.1 Discrete Cosine Transform (DCT)

For an 8-bit image, in the original block each element falls in the range [0,255]. Data range that is centred around zero is produced after subtracting The mid-point of the range (the value 128) from each element in the original block, so that the modified range is shifted from[0,255] to [-128,127]. Images are separated into parts of different frequencies by the DCT. After dct transformation, the 'DC coefficient' is the element in the upper most left corresponding to (0,0) and the rest coefficients are called 'AC coefficients'. The 'DC coefficient' represent the average of the pixel values in the block whereas 'AC coefficients ' represent a measure of pixel variation. So, basically the Discrete Cosine Transform converts spatial domain to frequency domain and the DCT outputs are related to how much the pixel values changed as a function of their position in the block. A lot of variations in pixel values indicates an image with lot of fine detail whereas if there are small variations in pixel values then there is more uniformity & less fine details. DCT does not provide any Encoder however it rearranges the data into a

form that allows another coding technique to compress the data more effectively. Equation 4.1 shows the DCT equation used in programming the DCT module using SystemC.

$$DCT(i,j) = \frac{1}{4}C(i)C(j)\sum_{x=0}^{7}\sum_{y=0}^{7} pixel(x,y)\cos\frac{(2x+1)i\pi}{16}\cos\frac{(2y+1)j\pi}{16}$$
$$where, C(k) = \begin{cases} \frac{1}{\sqrt{2}}, & \text{if } k = 0 \\ 1, & \text{otherwise} \end{cases} \tag{4.1}$$

4.2.2 Quantization

We obtain the Quantization by dividing transformed image DCT matrix by the quantization matrix used . Values of the resultant matrix are then rounded off. Quantization aims at reducing most of the less important high frequency DCT coefficients to zero, the more zeros the better the image will compress. Lower frequencies are used to reconstruct the image because human eye is more sensitive to them and higher frequencies are discarded. If there are many high frequencies, then respective coefficients end up zeroed which leads to higher Encoder rates. One can use a uniform (each element has a constant value)or non-uniform (lower right have higher values to eliminate higher frequencies whereas upper left have lower values to account for lower frequencies) Quantization Matrix or Table. Equation 4.2 was implemented using SystemC where-in a Uniform Quantization Matrix was selected to have a constant value of each element to be 8.

$$DCT_Q(i,j) = Round\left[\frac{DCT(i,j)}{Q(i,j)}\right] \tag{4.2}$$

4.2.3 RunLength Encoding

The RLE module implemented in SystemC comprises of Zig-Zag Scanning, Differential Pulse Code Modulation for DC Components and Run Length on AC Components. These are discussed briefly in this subsection.

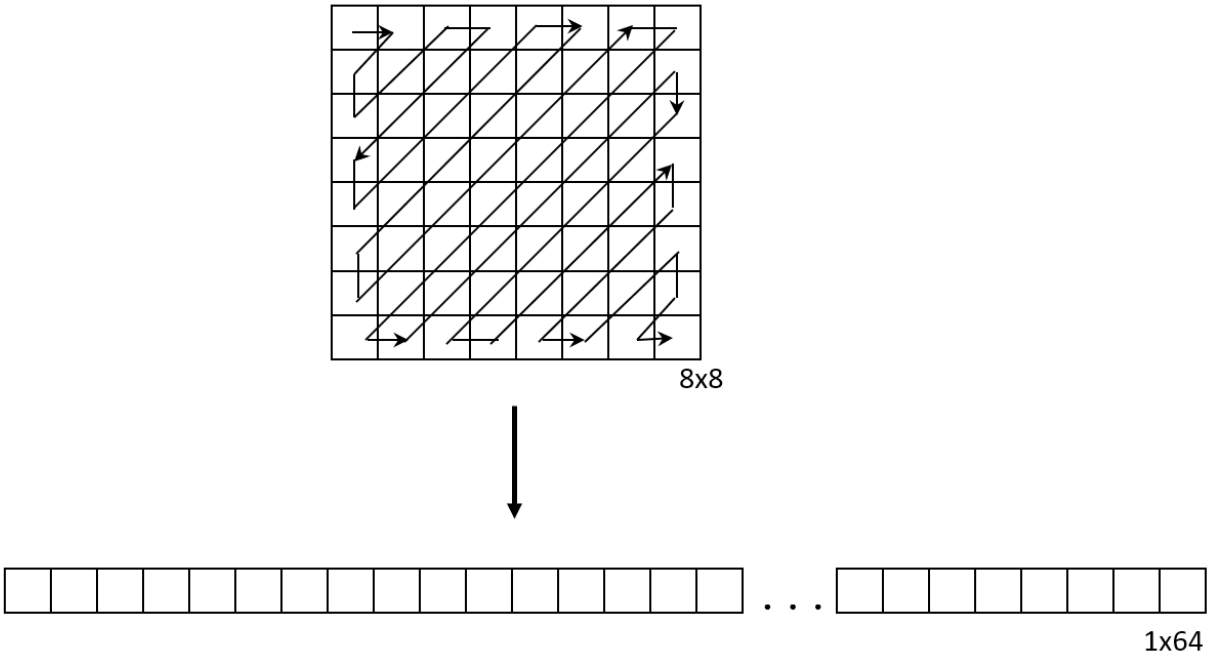


Figure 4.3: Zig-Zag Scan

Zig-Zag Scan

After zeroing out all the high frequency image data, in order to further compress the data of 8 X 8 block, grouping of low frequency coefficients in top of vector and high frequency coefficients at the bottom is done using Zig-Zag Scan. Thus, the 8 X 8 Matrix maps to a 1 X 64 vector. Figure 4.3 shows the Zig-Zag Scanning Method implemented.

DPCM on DC Components

The DC component value in every 8 X 8 block is large and varies across blocks but the difference between the DC component values between neighboring blocks is less. Meaning, the dc component values are often very close between current block and previous block in general. This fact is exploited here using the technique known as Differential Pulse Code Modulation(DPCM) where encoding the difference between current and the previous 8 X

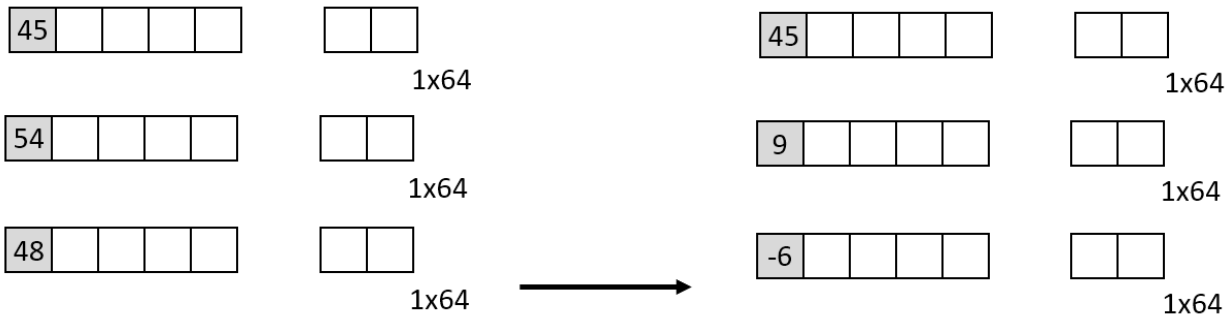


Figure 4.4: Example of DPCM

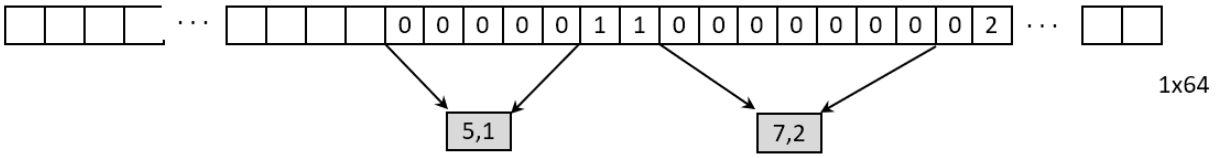


Figure 4.5: Example of RLC

8 block results in eventually requiring fewer bits to represent a smaller number. Figure 4.4 shows an example to illustrate the method of DPCM.

RunLength on AC Components

The 1 X 64 vector has lot of zeros in them after quantization and zig-zag scanning especially more towards the end of the vector which represents all the higher frequencies. This step encodes a series of zeros as a (skip, value) pair, where skip is the number of zeros and value is the next non-zero component. There is an end of the block sentinel value of (0,0) to indicate termination of block. Figure 4.5 shows an illustration of (skip, value) pairs encoded series.

4.2.4 Entropy Coding

The SystemC implementation has separate functions for Entropy coding of DC and AC Components.

DC Components

The DC Components are differentially coded as `(SIZE,VALUE)` where `VALUE` corresponds to the differential dc co-efficient component obtained from DPCM. The `SIZE` is obtained from a predefined Table consisting mappings of binary sizes of dc coefficients and their values, which we will abbreviate as size_and_value_table. The `SIZE` value read from this Table is referenced to **Huffman Table for DC Component** to obtain the Huffman Code for the `SIZE`.

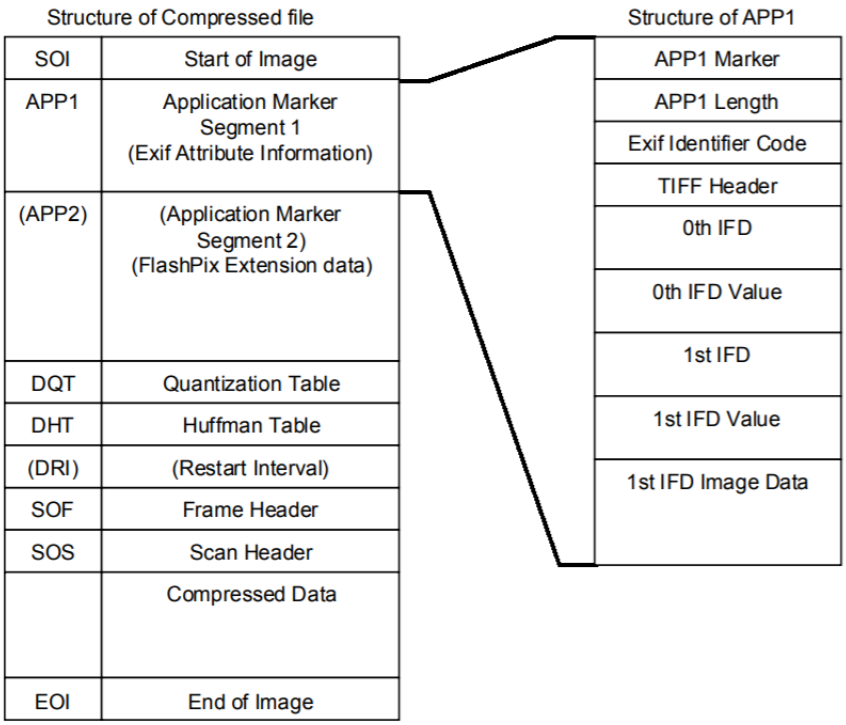


Figure 4.6: Basic Structure of JPEG Header File Format

AC Components

The AC Components are coded as `(S1,S2 pairs)`, where `S1` represents **(RunLength/SIZE)** and `S2` represents **VALUE**. The **RunLength** is the length of consecutive zero values, **SIZE** is the number of bits needed to code the next nonzero AC component value and **VALUE**

is the actual AC component. Referring to Huffman Table for AC Runs/Size Pairs and size_and_value_table, Huffman binary code is generated for each AC component.

The final huffman encoded bitstream of all the 8 X 8 blocks is entered as data into the JPEG Header file format along with Quantization table, AC and DC Huffman Tables to recreate the image with reduced file size. Figure 4.6 shows a basic structure of JPEG Header file format where the Compressed data Section is where the encoded Huffman binary data is entered.

4.3 Summary

In this chapter, we discussed about dataflow computation and its usage with resspect to FPGAs in general along with terminologies. We also explained fundamental blocks of JPEG Encoder, which will be implemented in forthcoming chapters.

CHAPTER 5

DEDICATED JPEG HARDWARE ACCELERATOR GENERATION AND VALIDATION

This chapter describes the results of SystemC to RTL generation and post-synthesis functional simulation of each processing element of JPEG Encoder. Also, a method to wrap each of the processing element as IP and verify its functionality with ARM Cortex A-9 is discussed. For the purpose of this study, we have used Zedboard, which is a low-cost development board for the Xilinx Zynq-7000 all programmable SoC (AP SoC) from Digilent, Inc. For brevity, we will take an example of 8 x 8 block of lena.bmp grayscale image with each pixel being 8-bit in size.

5.1 Validation Design Flow

Figure 5.1 shows the validation design flow steps which we will be discussing in this section.

5.1.1 High-Level Synthesis using NEC's CyberWorkBench

A shell script was written to automate the process of generation of RTL for each of the processing elements. In Section 2.4 we have discussed in detail the HLS process in CWB Tool. The SystemC code was first imported into NEC's CyberWorkBench tool and was parsed to compile for any errors in syntax. We need to set the target Basic Library (BLIB) and Standard Functional Library (FLIB) for Zynq devices during project creation as parameters. Clock period was always set to 50 MHz that is 20 ns. Automatic Scheduling was selected to enable CWB to automatically time the C description at the scheduling phase. A Functional Unit Constraints file needs to be generated which would provide the number and type of FUs allocated for performing HLS. CWB provides FUs of different bit width sizes such as Large (<=8 bits), middle (<=4 bits) and small (>4 bits). HLS tool will try to maximize

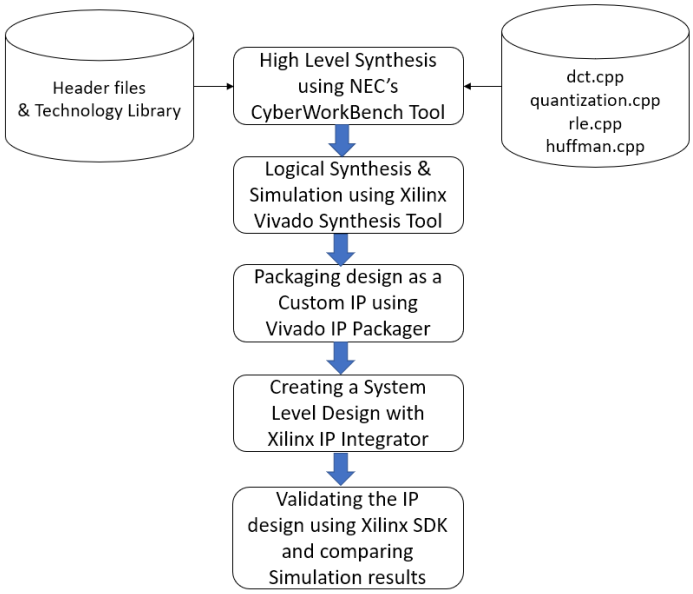


Figure 5.1: SystemC to Custom IP Package Design Validation Flow

parallelism by referring to the generated .FCNT constraints file and hence it is always a good practice to provide as many FUs as possible for each category unless there is a restriction. Finally, the design was synthesized and Verilog RTL was generated. A report file – Quality of Results (QoR) was generated automatically which contained all the synthesis information such as Latency Index, FPGA resources used, Target FPGA device, Critical Path etc. Below is a shell script example for performing HLS on dct.cpp file:

- `scpars -EE -info_base_name scpars "./dct.cpp"`
- `bdltran -EE -c2000 -s -Zresource_fcnt=GENERATE -Zresource_mcnt=GENERATE -Zresource_mcnt=GENERATE -Zdup_reset=YES -tcio -EE -lb /eda/cwb/cyber_561/LINUX/packages/zynq-1.BLIB -lfl /eda/cwb/cyber_561/LINUX/packages/zynq-1.FLIB dct.IFF`
- `veriloggen -EE dct_E.IFF`

5.1.2 Logical Synthesis and Simulation using Xilinx Vivado Suite

The Structural RTL code generated by CWB for each processing element was logically synthesized using Xilinx Vivado Design Suite. The Vivado Integrated Design Environment (IDE) synthesis is timing-driven and optimized for memory usage and performance. This tool performs logical optimization of gate-level design and maps the netlist to Xilinx primitives. Post Synthesis Utilization Report shows the resources such as memory, DSP Slice, LUTs, IO, clocking etc. used by the design.

To verify the functional correctness of the design of each processing element of JPEG Encoder, testbench were written in Verilog to perform Post Synthesis Functional Simulation using Vivado. The waveform results were observed with the true values and correct behavior of control signals. The .dcp file, which is the design checkpoint was generated for each processing element. These checkpoints are used to manage design progress and analysis in a Non-Project Flow Methodology.

5.1.3 Creating and Packaging Custom IP using Vivado IP Packager

The Vivado IP packager tool provides design reusability feature based upon IP-XACT standard [36]. The Vivado IP catalog contains Xilinx IP, third party IP or custom-developed IP thus giving a unified IP repository that provides framework for IP-centric design flow. Xilinx has adopted to the standard of Advanced eXtensible Interface (AXI) protocol for its IP cores and hence using the same in Custom IP would give the flexibility to connect with another IP in the Vivado IP Catalog [36]. AXI is part of the ARM AMBA, a family of micro-controller buses and Vivado supports AMBA 4.0 which is released in 2010. The Custom IPs can be edited and re-packaged anytime using the IP packager.

The JPEG Encoder Processing Element Custom IPs were created using the 'Create and Package IP' wizard available in the Vivado Integrated Design Environment (IDE) using the RTL source files and AXI4-Lite Interface - which is a light-weight single transaction

memory-mapped interface. The interface allows data-widths up to 32-bits and 512 number of registers. We first add the interface requirements and then use the Edit IP option in the 'Create and Package IP' wizard to open the top-level files generated to add an instance of the required RTL source file. The input and output signals of each Processing element instance needs to be port mapped and wired with the internal slave registers respectively. The clock signal also needs to be port mapped with system clock '`S_AXI_ACLK`'.

5.1.4 Creating a System Level Design and Validating the IP using Xilinx SDK

To Validate that the Custom IP packaged works as expected, block design was created with ZYNQ7 Processing System using the Vivado IP Integrator. Running Connection Automation generates AXI Interconnect Block to connect Slave Processing Element Custom IP with Master AXI port of ARM Processor and Processor System Reset Block connects asynchronous reset signals with AXI Interconnect Block and peripheral Custom IP. Design Rule Check (DRC) was performed on each block design to find out missing interconnects, clock frequency not set, IP catalog not updated etc. After the block design was completed and DRC check was cleared, output products were generated using the global synthesis option. These include source files and appropriate constraints for all the IP, which is made available in the source pane [37]. A top-level HDL file for the IP Integrator sub-system was generated by creating HDL Wrapper. This top-level file was then ready for elaboration, synthesis and implementation.

After running the implementation stage, the generated hardware was exported to SDK i.e. the necessary XML files needed for SDK to understand the IPs used in the design and the memory mapping from the processor's perspective. SDK creates necessary drivers and board support package for the target hardware. An application project was created for each PE individually to validate the functional correctness of the design and memory mappings. Also, the results were compared with the post synthesis functional simulation results.

5.2 Validating JPEG Encoder Processing Elements

In June 2017, S2CBench (Synthesizable SystemC Benchmark Suite) v.2.0 introduced JPEG Encoder Algorithm as one of the benchmarks for hardware acceleration. The Encoder Algorithm comprises of processing elements: Discrete Cosine Transform, Quantization, Run-Length Encoding and Huffman Encoding. The input data stream requires to be computed by each processing element in a serial cascaded fashion and the output stream generated can be stored back. We would be using this particular benchmark for our study.

In the existing benchmark, all the processing elements of JPEG Encoder are called serially in the top module of the SystemC file to perform computation on the input stream. In order to implement Partial Reconfiguration as one of the design methodologies, it was important to perform high level synthesis on each of the processing element individually and verify its functionality. Moreover, it was required to add few additional control signals to initiate and terminate the computing of each processing element in order to have a close loop control. Thus, after performing high level synthesis using NEC's CyberWorkBench[1] tool, the generated RTL code was logically synthesized using Xilinx Vivado and tested.

In the following remaining sub-sections of this chapter,Custom IP of JPEG Encoder Processing Element is created using the method described previous section and is validated with an example of one 8 X 8 block out of a 512 X 512 grayscale lena image.

5.2.1 Discrete Cosine Transform IP Block

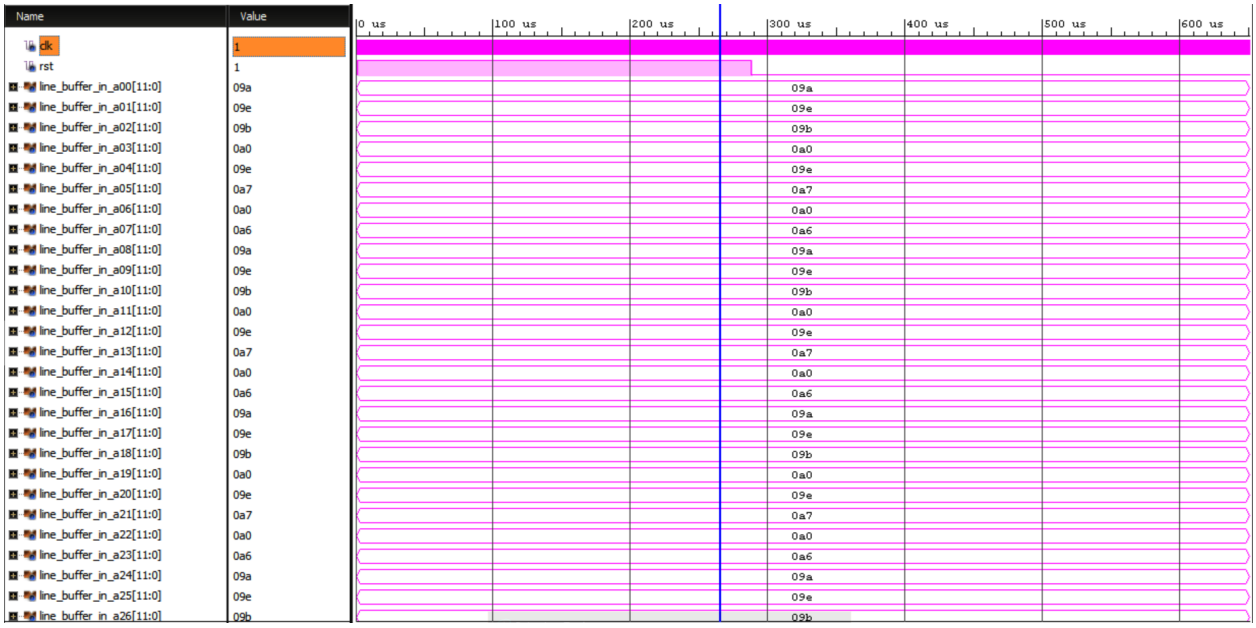
The Discrete Cosine Transform (DCT) Block IP receives the raw image data as input and produces 64 12-bit signed integer output values along with toggling a valid signal high. The valid signal high indicates that the computation is completed and the results are ready to sample. The reset signal is active low and a transition from low to high initiates the computation. The `DCT_Co-efficient Matrix` used is as follows:

$$\begin{bmatrix} 1.00000 & 0.98079 & 0.92388 & 0.83147 & 0.70711 & 0.55557 & 0.38268 & 0.19509 \\ 1.00000 & 0.83147 & 0.38268 & -0.19509 & -0.70711 & -0.98079 & -0.92388 & -0.55557 \\ 1.00000 & 0.55557 & -0.38268 & -0.98079 & -0.70711 & 0.19509 & 0.92388 & 0.83147 \\ 1.00000 & 0.19509 & -0.92388 & -0.55557 & 0.70711 & 0.83147 & -0.38268 & -0.98079 \\ 1.00000 & -0.19509 & -0.92388 & 0.55557 & 0.70711 & -0.83147 & -0.38268 & 0.98079 \\ 1.00000 & -0.55557 & -0.38268 & 0.98079 & -0.70711 & -0.19509 & 0.92388 & -0.83147 \\ 1.00000 & -0.83147 & 0.38268 & 0.19509 & -0.70711 & 0.98079 & -0.92388 & 0.55557 \\ 1.00000 & -0.98079 & 0.92388 & -0.83147 & 0.70711 & -0.55557 & 0.38268 & -0.19509 \end{bmatrix}$$

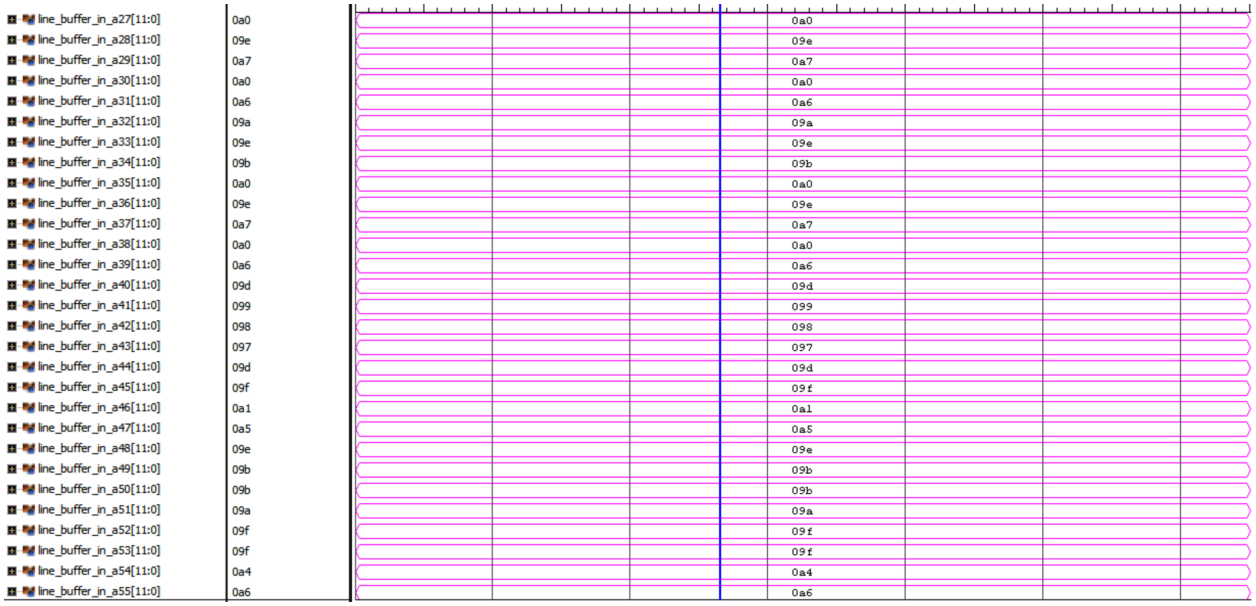
`Raw_Image_Matrix=`

$$\begin{bmatrix} 0x9a & 0x9a & 0x9a & 0x9a & 0x9a & 0x9d & 0x9e & 0x9c \\ 0x9e & 0x9e & 0x9e & 0x9e & 0x9e & 0x99 & 0x9b & 0x9d \\ 0x9b & 0x9b & 0x9b & 0x9b & 0x9b & 0x98 & 0x9b & 0x98 \\ 0xa0 & 0xa0 & 0xa0 & 0xa0 & 0xa0 & 0x97 & 0x9a & 0x99 \\ 0x9e & 0x9e & 0x9e & 0x9e & 0x9e & 0x9d & 0x9f & 0x9d \\ 0xa7 & 0xa7 & 0xa7 & 0xa7 & 0xa7 & 0x9f & 0x9f & 0x9e \\ 0xa0 & 0xa0 & 0xa0 & 0xa0 & 0xa0 & 0xa1 & 0xa4 & 0xa4 \\ 0xa6 & 0xa6 & 0xa6 & 0xa6 & 0xa6 & 0xa5 & 0xa6 & 0xa2 \end{bmatrix}$$

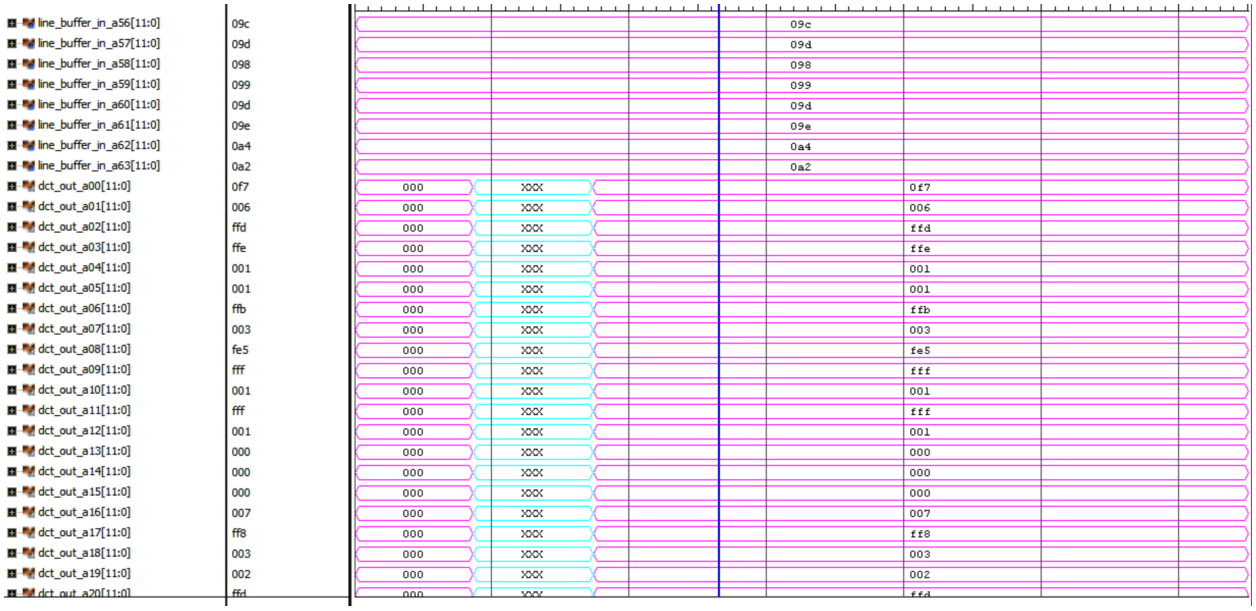
Figure 5.2a - 5.2e shows post logical synthesis functional simulation waveform results which are analogous to the sdk terminal results obtained in Figure 5.3, confirming that the memory mapping of the interface signals and functional equivalent of the Custom IP Block is accurate. DCT Standalone system-level design can be seen in Figure 5.4.



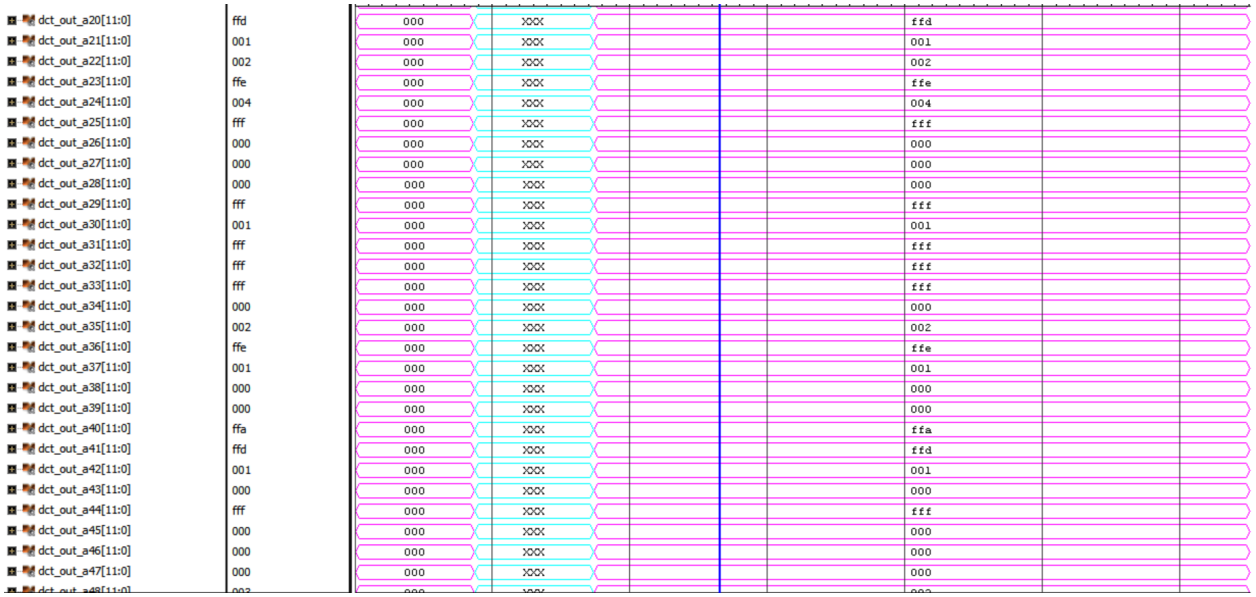
(a) Raw_Image_Matrix values [0 to 26] for inputs to DCT PE along with clock and reset signals



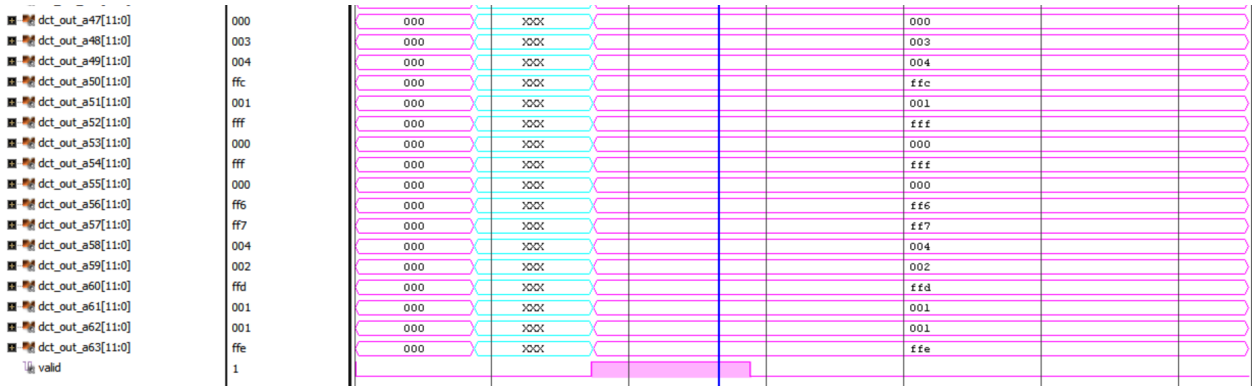
(b) Raw_Image_Matrix values [27 to 55] for inputs to DCT PE



(c) Raw_Image_Matrix values [56 to 63] for inputs to DCT PE resulting in DCT_Output_Matrix output values [0 to 20]



(d) DCT_Output_Matrix output values [21 to 47]



(e) DCT_Output_Matrix output values [48 to 63] along with valid signal toggling

Figure 5.2: (a)-(e)Post Logical Synthesis Functional Simulation Results for DCT RTL

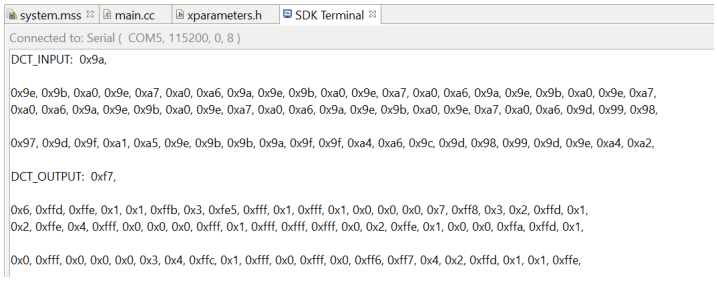


Figure 5.3: Xilinx SDK Terminal displaying DCT Input and Output values

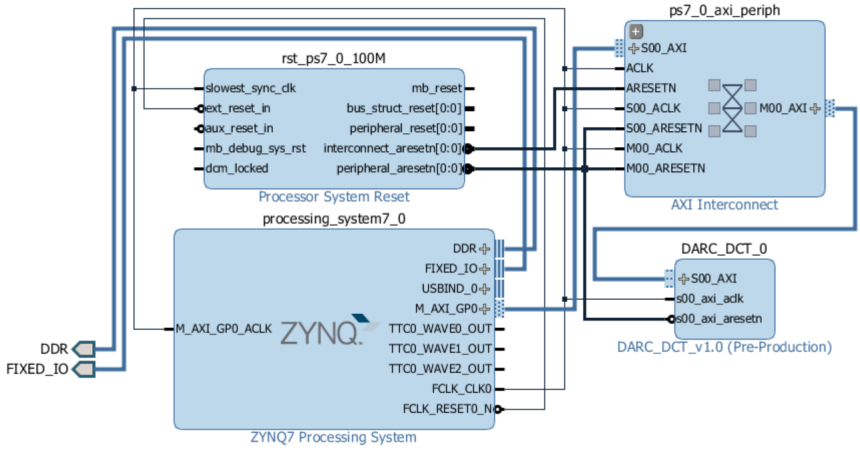


Figure 5.4: System Design and Validation of DCT IP using Vivado IP Integrator

DCT_Output_Matrix=

$$\begin{bmatrix} 0x0f7 & 0x006 & 0xffd & 0xffe & 0x001 & 0x001 & 0xffb & 0x003 \\ 0xfe5 & 0xfff & 0x001 & 0xfff & 0x001 & 0x000 & 0x000 & 0x000 \\ 0x007 & 0xff8 & 0x003 & 0x002 & 0xffd & 0x001 & 0x002 & 0xffe \\ 0x004 & 0xfff & 0x000 & 0x000 & 0x000 & 0xfff & 0x001 & 0xfff \\ 0xfff & 0xfff & 0x000 & 0x002 & 0xffe & 0x001 & 0x000 & 0x000 \\ 0xffa & 0xffd & 0x001 & 0x000 & 0xfff & 0x000 & 0x000 & 0x000 \\ 0x003 & 0x004 & 0xffc & 0x001 & 0xfff & 0x000 & 0xfff & 0x000 \\ 0xff6 & 0xff7 & 0x004 & 0x002 & 0xffd & 0x001 & 0x001 & 0xffe \end{bmatrix}$$

5.2.2 Quantization IP Block

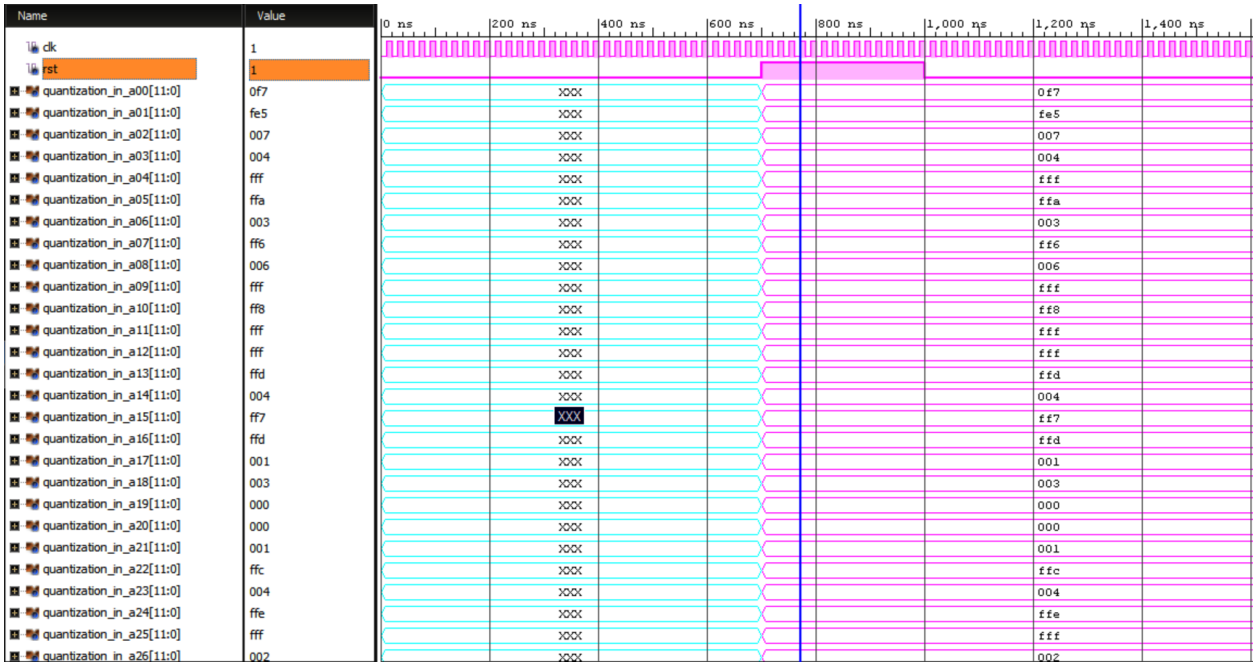
The Quantization IP Block receives the dct output data as input and produces 64 12-bit signed integer output values along with toggling a valid signal high. The valid signal high indicates that the computation is completed and the results are ready to sample. The reset signal is active low and a transition from low to high initiates the computation. Quantization is an important stage in which data Encoder happens significantly because all the image data of high frequency to which humans are insensitive are zeroed out with the help of `Quantization_Matrix`. The `DCT_Output_Matrix` is divided by `Quantization_Matrix` and the result is rounded off to the nearest signed integer. In our case, we have used a Uniform `Quantization_Matrix` as follows:

$$\begin{bmatrix} 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 \\ 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 \\ 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 \\ 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 \\ 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 \\ 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 \\ 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 \\ 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x8 & 0x20 \end{bmatrix}$$

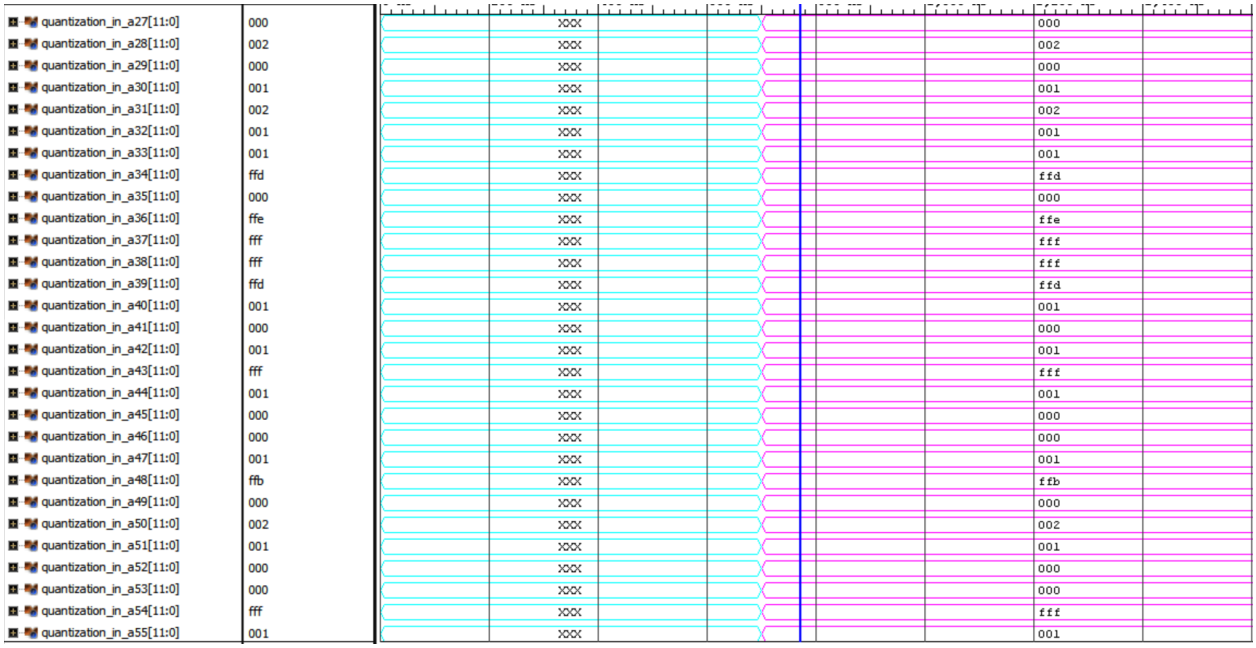
Figure 5.5a - 5.5e shows post logical synthesis functional simulation waveform results which are analogous to the sdk terminal results obtained in Figure 5.6, confirming that the memory mapping of the interface signals and functional equivalent of the Custom IP Block is accurate. Quantization Standalone system-level design can be seen in Figure 5.7.

`Quantization_Output_Matrix=`

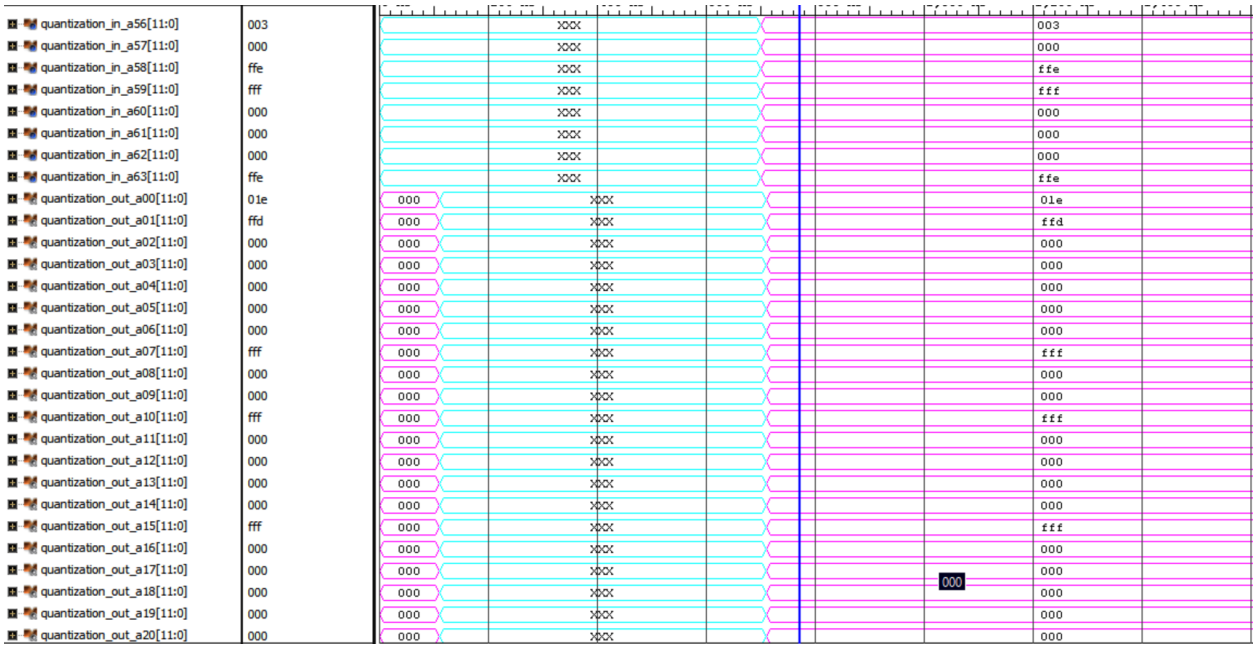
$$\begin{bmatrix} 0x1e & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 \\ 0xffd & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 \\ 0x0 & 0xfff & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 \\ 0x0 & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 \\ 0x0 & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 \\ 0x0 & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 \\ 0x0 & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 \\ 0xfff & 0xfff & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 & 0x0 \end{bmatrix}$$



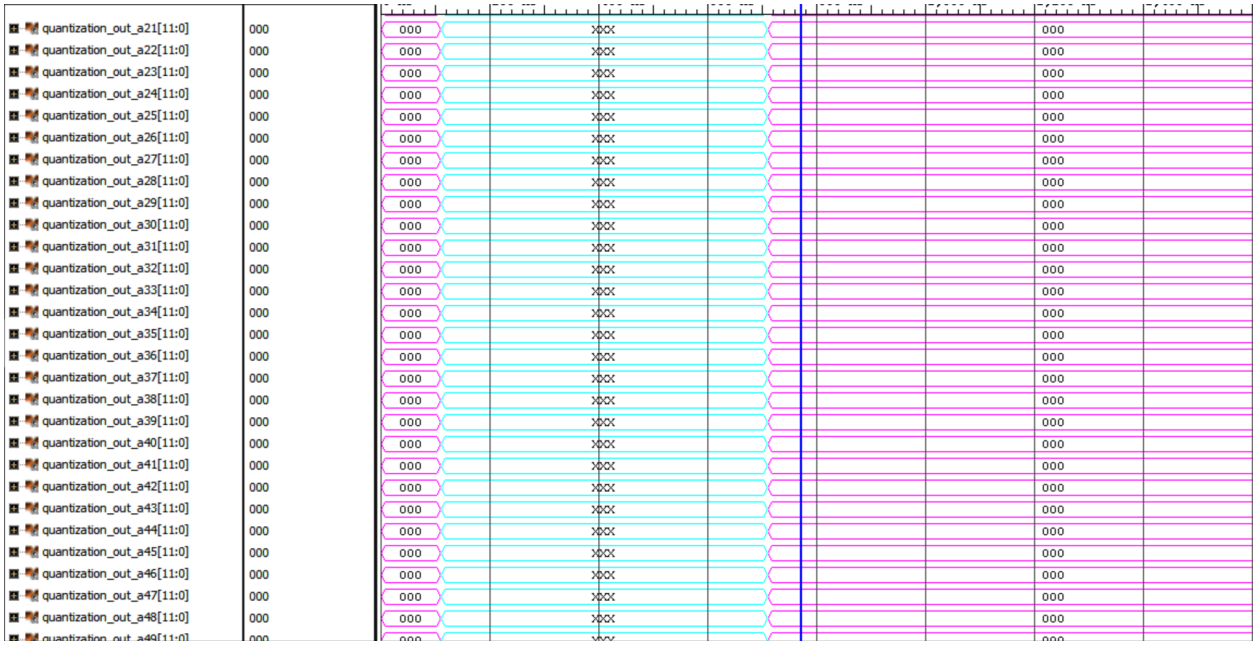
(a) DCT_Output_Matrix values[0 to 26] as inputs to Quantization PE along with clock & reset signals



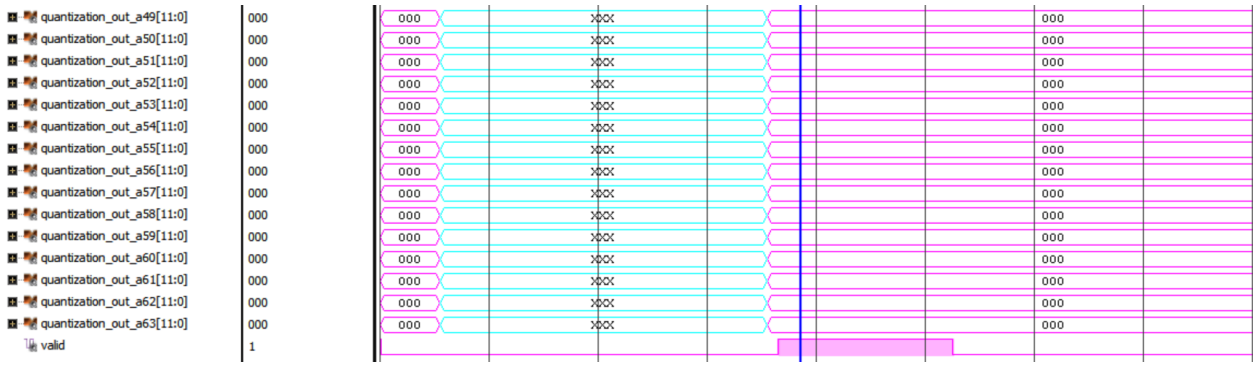
(b) DCT_Output_Matrix values[27 to 55] as inputs to Quantization PE



(c) DCT_Output_Matrix values[56 to 63] as inputs to Quantization PE resulting in Quantization_Output_Matrix output values [0 to 20]



(d) Quantization_Output_Matrix output values [21 to 48]



(e) Quantization_Output_Matrix output values [49 to 63] along with valid signal toggling

Figure 5.5: (a)-(e)Post Logical Synthesis Functional Simulation Results for Quantization RTL

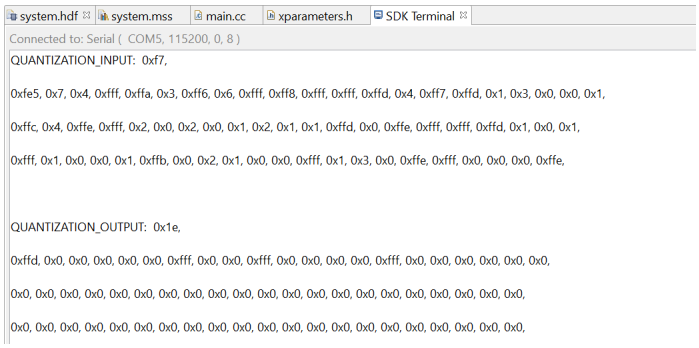


Figure 5.6: Xilinx SDK Terminal displaying Quantization Input and Output values

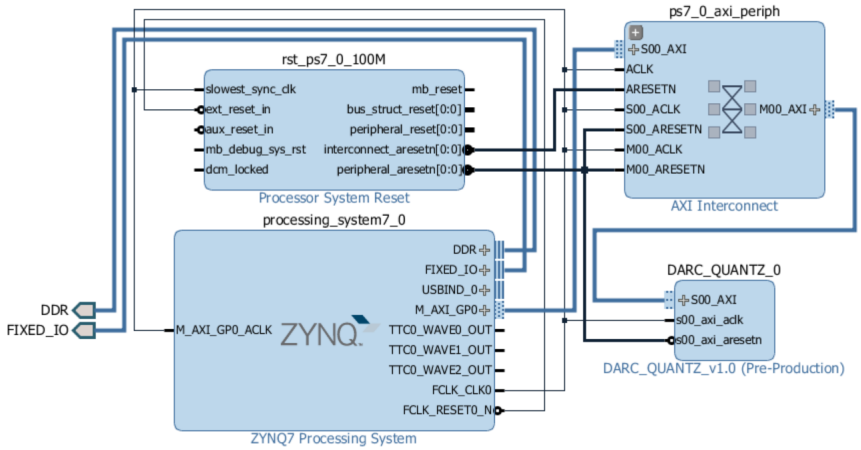


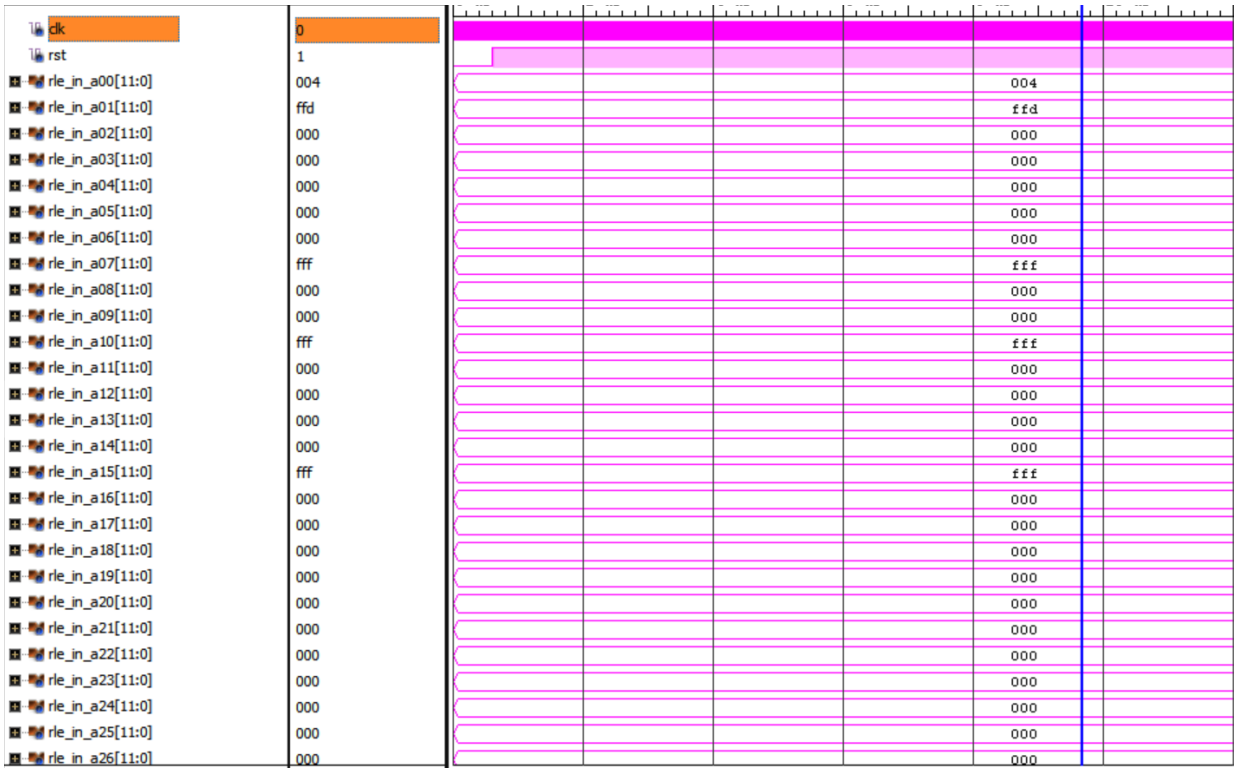
Figure 5.7: System Design and Validation of Quantization IP using Vivado IP Integrator

5.2.3 Run-Length Encoding IP Block

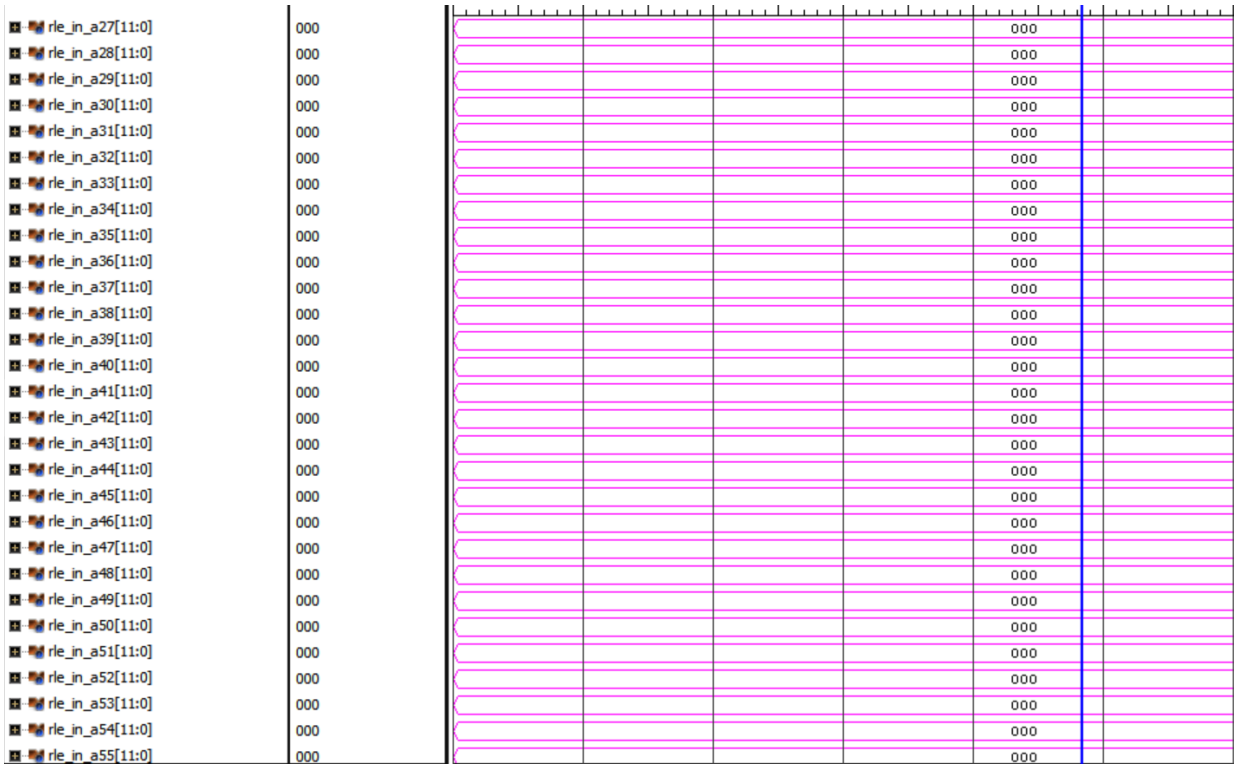
The RLE IP Block receives the quantization output data as input and produces 12-bit signed integer output of maximum vector length 128. The output vector has variable length depending upon `Quantization_Matrix_Output`. The valid signal high indicates that the computation is completed and the results are ready to sample. The reset signal is active low and a transition from low to high initiates the computation. The RLE output stream always terminates with two zeros indicating end of block. For the running case, the `RLE_Length` is 13 and `RLE_Output_Vector` is as follows

$$\left[0x4, 0x0, 0xffd, 0x5, 0xfff, 0xf, 0x0, 0x4, 0xfff, 0xd, 0xfff, 0x0, 0x0\right]$$

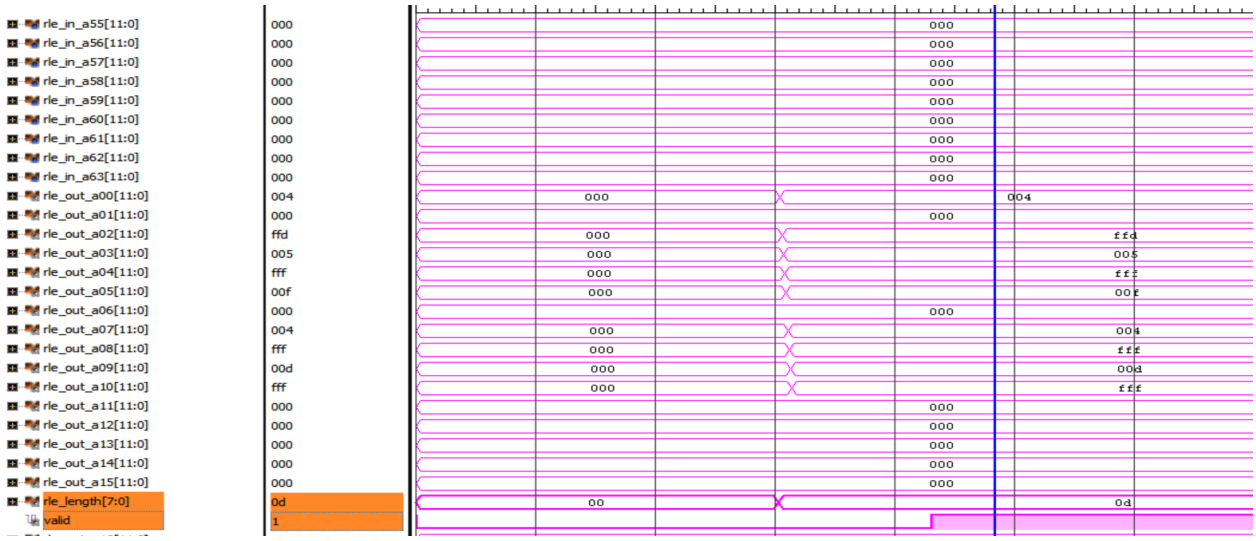
The dc component of the RLE Vector, which is the first element is the difference in the quantization value between the current block and the previous block. This is done to avaoid saturation of the image. Figure 5.8a - 5.8c shows post logical synthesis functional simulation waveform results which are analogous to the sdk terminal results obtained in Figure 5.9, confirming that the memory mapping of the interface signals and functional equivalent of the Custom IP Block is accurate. RLE Standalone system-level design can be seen in Figure 5.10.



(a) Quantization_Output_Matrix output values [0 to 26] as inputs to RLE PE along with clock & reset signals



(b) Quantization_Output_Matrix output values [27 to 55] as inputs to RLE PE



(c) Quantization_Output_Matrix output values [56 to 63] as inputs to RLE PE resulting in RLE_Output_Vector of length 13 along with valid signal toggling

Figure 5.8: (a)-(c)Post Logical Synthesis Functional Simulation Results for RLE RTL

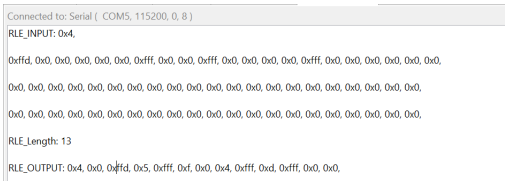


Figure 5.9: Xilinx SDK Terminal displaying RLE Input and Output values

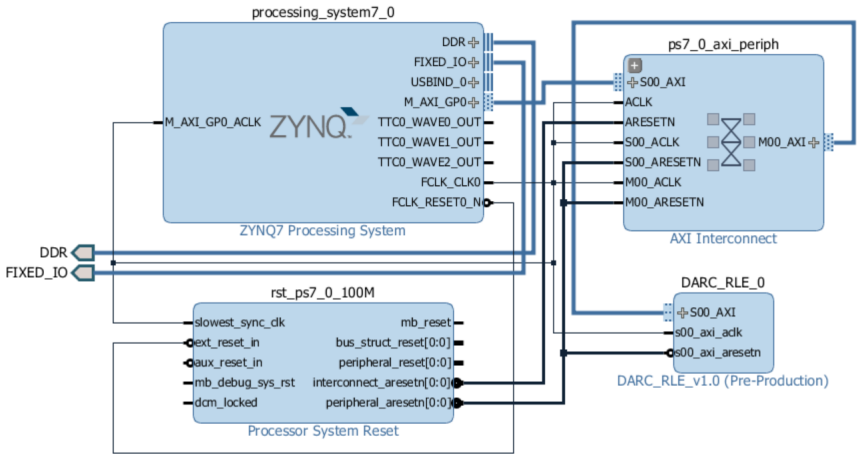


Figure 5.10: System Design and Validation of RLE IP using Vivado IP Integrator

5.2.4 Huffman Encoding IP Block

The `RLE_Output_Vector` and `RLE_Length` are inputs to the Huffman Encoding IP Block. The SystemC file has the main function which calls `jpeg_DCcode` and `jpeg_ACcode` functions which create the Huffman encoded binary data depending upon the inputs and Lookup tables of code-lengths, code and power-tables. In order to maximize the throughput of output data, instead of declaring 512 boolean output ports, we have created 8 64-bit output ports in the top-level module.

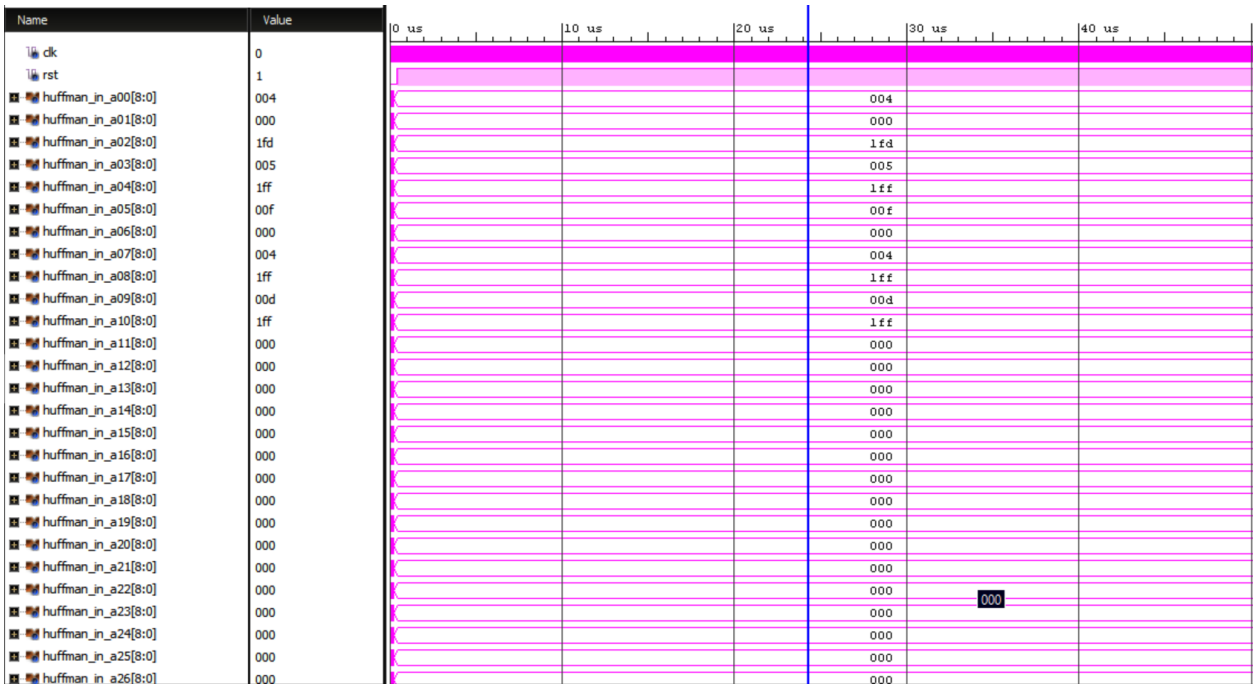
However, since the AXI4 has a limitation on the bit width of 32-bits, we have split the MSB and LSB for each output port of top module and hence there are 16 memory maps for output ports between ARM Cortex Master AXI4-lite and Slave AXI4-lite Huffman Encoding IP Block. In the software application, `huffman_binary()` is a function written to create the Huffman binary data from 16 output values obtained. The IP also outputs the bit-length of the output data For the running case, the `Huffman_bitlength` is 52 and `Huffman_Output_Vector` is as follows

$$\left[0xfdfe5e44, 0x52ff6, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,\right]$$

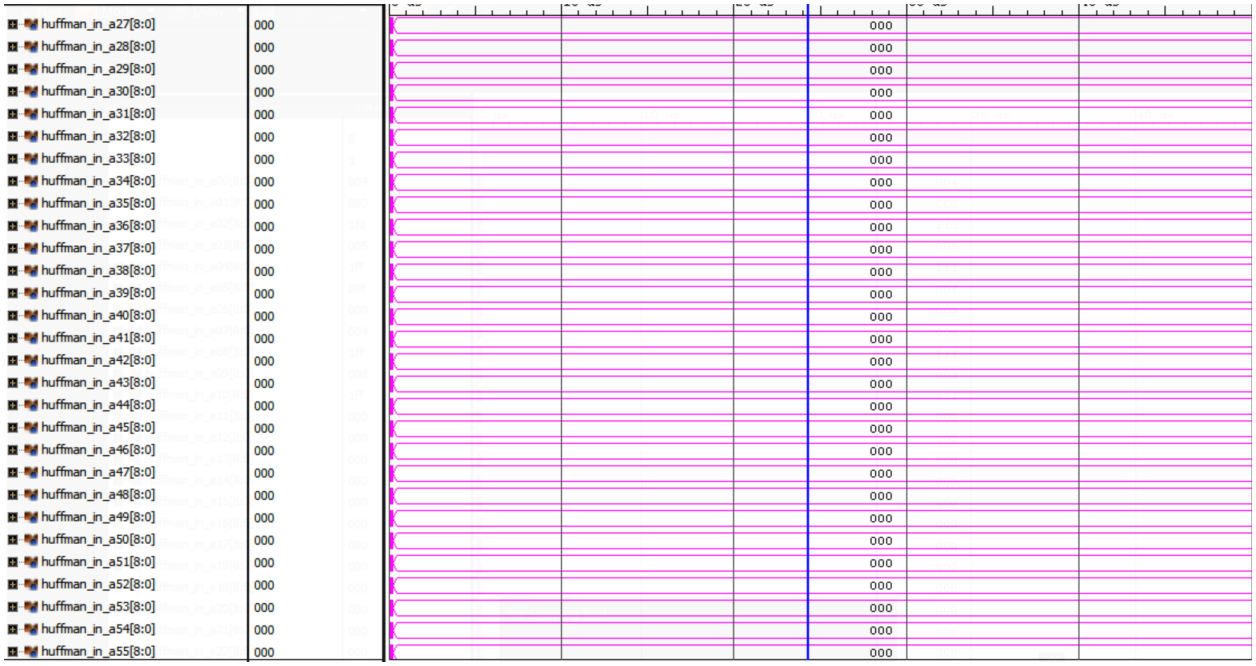
`Huffman_Encoded_BitStream=`

0010001001111010011111111011111101101111111101001010

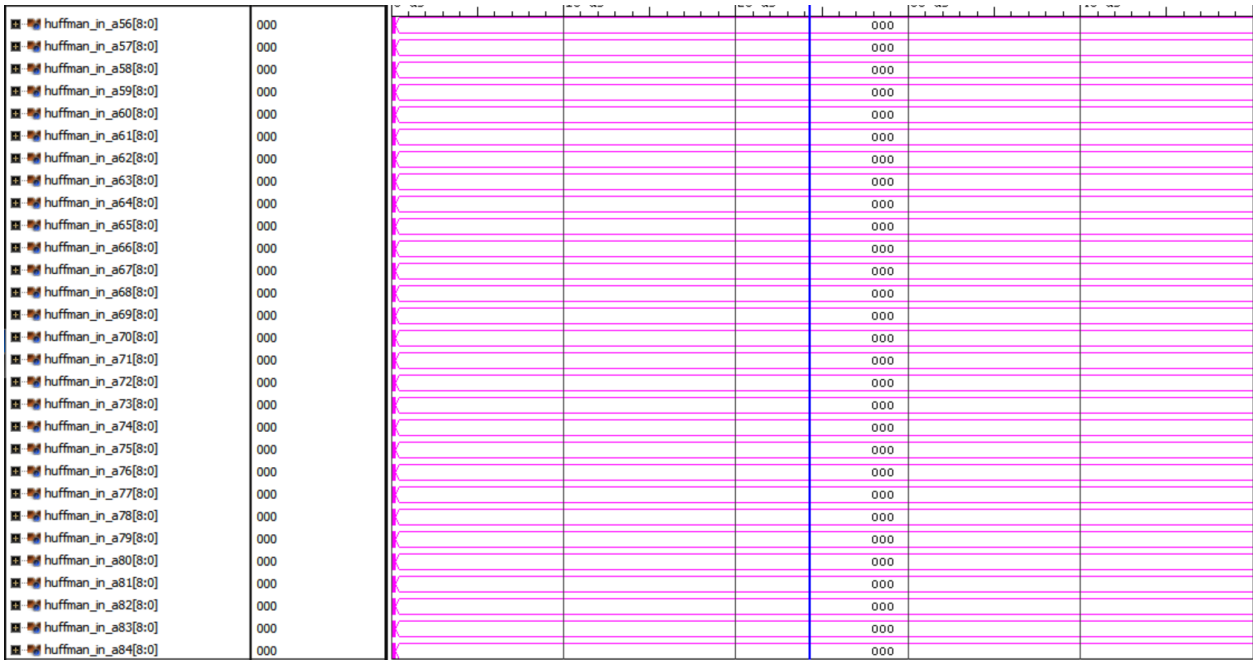
The last 4-digits in `Huffman_Encoded_BitStream` should always be `1010`, which indicates termination of the block. Figure 5.11a - 5.11e shows post logical synthesis functional simulation waveform results which are analogous to the sdk terminal results obtained in Figure 5.12, confirming that the memory mapping of the interface signals and functional equivalent of the Custom IP Block is accurate.Huffman Encoding Standalone system-level design can be seen in Figure 5.13.



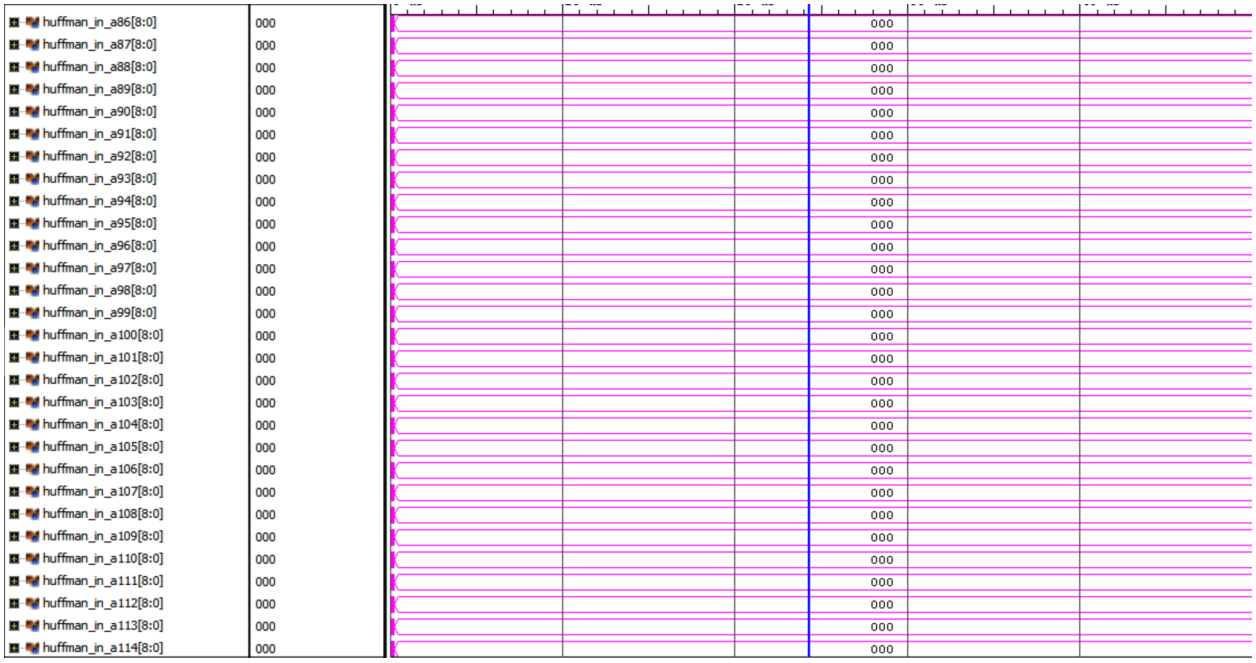
(a) RLE_Output_Vector with length 13 as input to the Huffman PE along with clock & reset signals



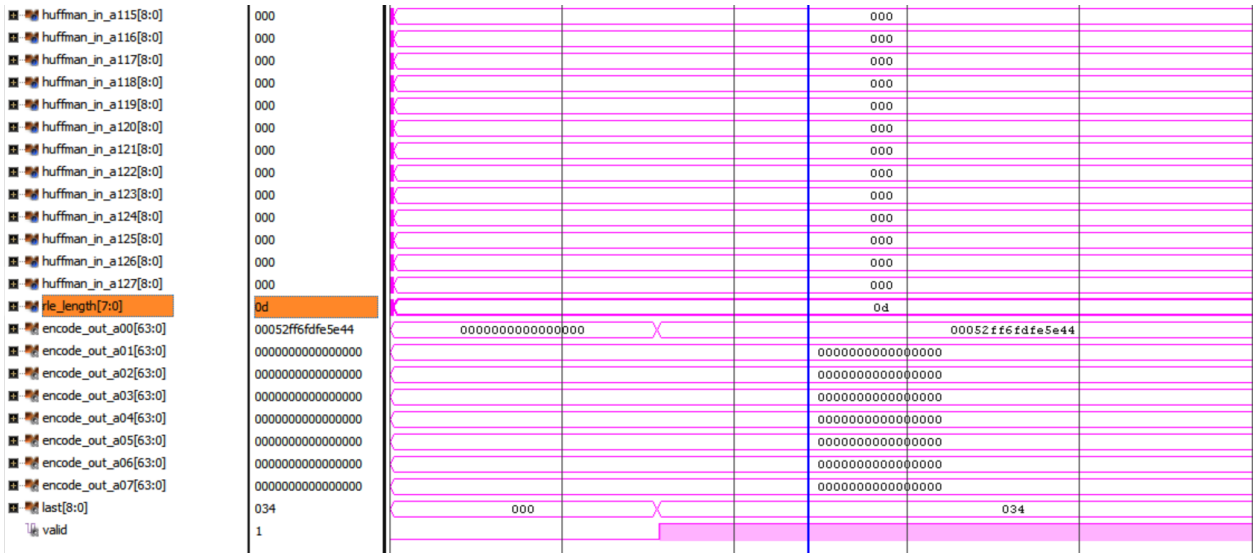
(b) The Huffman PE block has 128 inputs considering the maximum length of RLE_Output_Vector and treating the remaining values outside the vector length as zero



(c) Since the maximum length is 13 in the running example case, inputs [55 to 84] are zero



(d) Since the maximum length is 13 in the running example case, inputs [85 to 114] are zero



(e) Huffman Encoded data is obtained along with valid signal toggline

Figure 5.11: (a)-(e)Post Logical Synthesis Functional Simulation Results for Huffman Encoding RTL

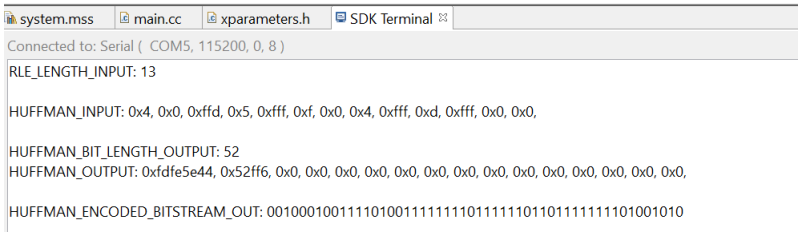


Figure 5.12: Xilinx SDK Terminal displaying Huffman Input and Output values

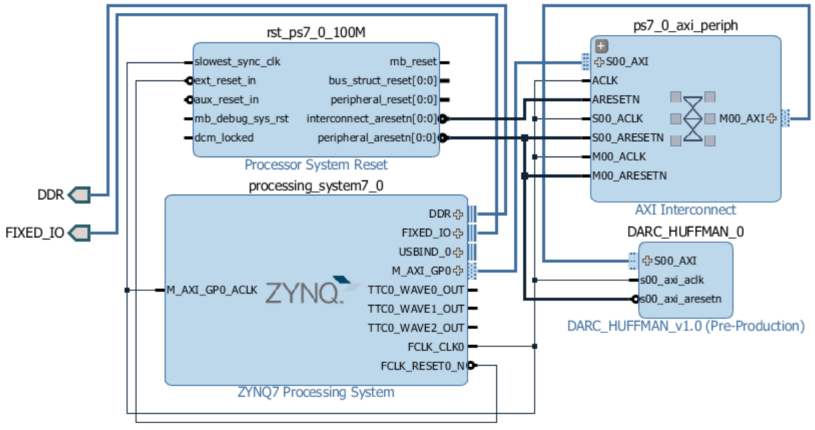


Figure 5.13: System Design and Validation of Huffman IP using Vivado IP Integrator

5.3 Summary

In this chapter, we have explained using a running example of 8 X 8 block, the S2CBench v.2.0 - JPEG Encoder Hardware Accelerators. The Validation Design Flow was also discussed which helps to identify early bugs in terms of functional and timing issues before integrating these accelerators into much more complex system designs.

CHAPTER 6

DESIGN METHODOLOGIES FOR DATAFLOW COMPUTATION

In this Chapter, we propose partial reconfiguration based design methodology for dataflow computation which consists of a novel static architecture using on-chip internal BRAM memory for storing intermediate data results when time multiplexing several kernels of dataflow abbreviated as $\mathbf{PR}_{BRAM}$. We also claim that the proposed architecture is area efficient compared to implementing the design spatially on FPGA as fewer resources are required. Also, this implementation is runtime and latency efficient compared to implementing partial reconfiguration design for data flow computation using external off-chip DDR memory to store intermediate data results abbreviated as $\mathbf{PR}_{DDR}$.

Thus we would be exploring the spatial and partial reconfiguration based design methodologies - PR_{BRAM} & PR_{DDR} for dataflow computation processes and discuss the results obtained when implementing on FPGAs. The dataflow computation process in our study as previously discussed is JPEG Encoder and we would be using the custom IPs that were validated using the design flow discussed in previous chapter for all the design explorations discussed in this chapter. We would be analyzing our study with the help of three testcase grayscale images of 512 X 512 pixels: lena.bmp, peppers.bmp and goldhill.bmp. We will also explain how the proposed architecture can be scaled for any dataflow computation process in general.

6.1 Overview of the Proposed Design Methodology

Figure 6.1 shows the overall design flow using proposed architecture and as it can be observed there are 4 main Stages: SystemC/BDL Description of the algorithm to RTL Generation, Validation and Creation of Custom IPs, TCL based automated floorplanning using Xilinx Vivado and finally deploying the binaries on Zynq 7000 AP SoC Device Part: XC7Z020.

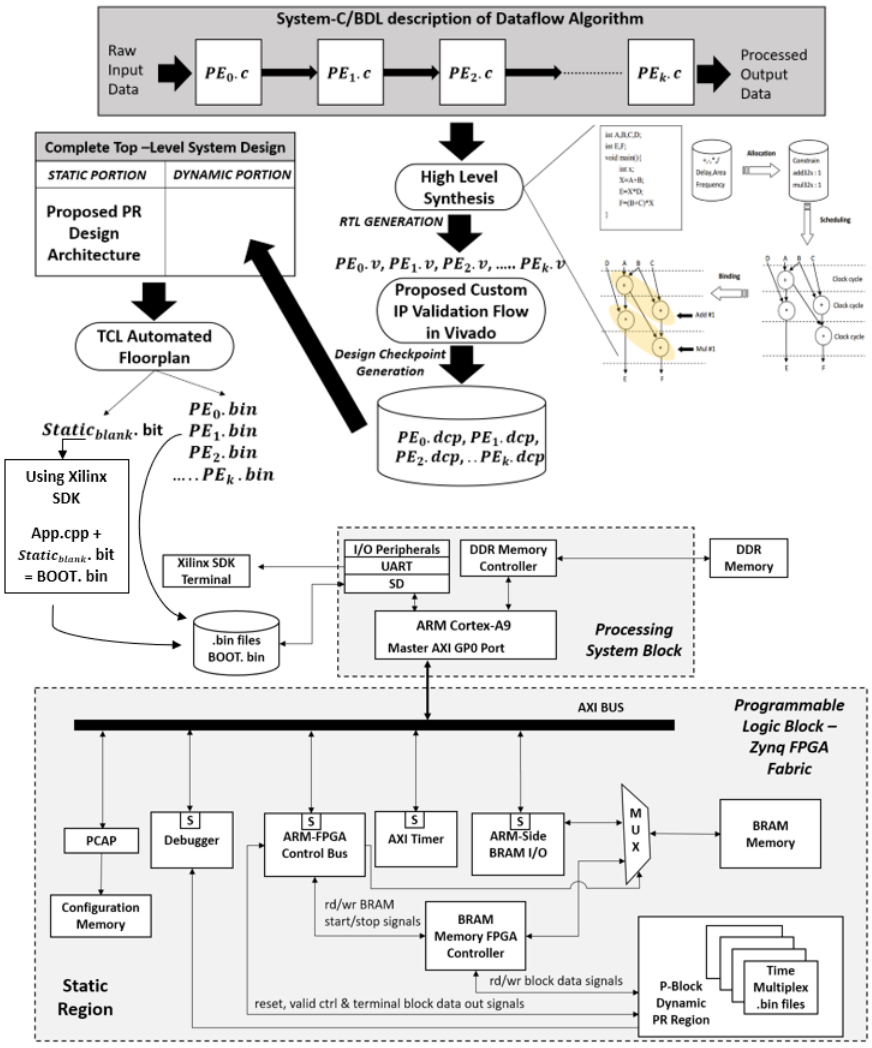


Figure 6.1: Proposed Design Methodology

6.1.1 Stage 1: SystemC/BDL Algorithm Description to RTL Generation

Using any commercial HLS Tool a behavioral description of a dataflow computation process in C/C++/SystemC can be converted to RTL files. When using a partial reconfiguration based design methodology, it is important to take into account the uniformity in the number and data-widths of inputs & outputs across all the processing elements. This is because when reading checkpoints of reconfigurable modules into the static design's reconfigurable partition pblock, the netlist processing would throw floating I/O error. Also, each of the dataflow processing elements are required to have control interface signals such as done, reset and start in order to have close feedback loop on the

ARM Processor Side (PS), when context switching the reconfigurable modules. Thus, given any dataflow compute system ensuring these keypoints is critical to the performance of the proposed system.

6.1.2 Stage 2: Validation and Creation of Custom IPs

Validation and Creation of custom IPs using the design flow described in Figure 5.1 in Section 5.1 helps to write the software code for the IP and ensure correct functional behavior beforehand integrating into complex design flow. Also, post synthesis implementation on target specific board can ensure any setup and hold time violations which would require a revision from the HLS Side or a reduction in the overall frequency of the programmable logic system. For the purpose of our studies in this research, we have used NEC's CyberWorkBench HLS Tool and Xilinx Vivado.

6.1.3 Stage 3: TCL Automated Floorplan for PR Designs

After ensuring the functional correctness of individual processing element blocks, the synthesized design checkpoints along with the proposed design architecture's static logic design checkpoint (as seen in the bottom half of the Figure 6.1) are used to generate the partial and full bitstreams using the Non-Project TCL Flow in Vivado. The approach used in partial reconfiguration based design flow is also known as bottom up synthesis where-in the RMs are synthesized separately and when running synthesis on the overall design, these are treated as black boxes and is not optimized for any further logical reduction in FPGA resources. Since, there could be any number of stages in a dataflow process thus increasing the count of RMs, hence automated TCL script is written as shown in Algorithm 2 to handle the back-end FPGA design flow of placement optimization, routing and generation of binaries. The Algorithm 2 requires user inputs of pblock region or RP dimensions which are stored as fplan.xdc file along with design checkpoint files for all the RMs abbreviated as 'PE$_k$.dcp' and static logic design checkpoint abbreviated as 'Static$_{BRAM}$.dcp'. The very first RM needs to be loaded into RP (lines 1 to 5). The RP is originally a black box whose area needs to be assigned using fplan.xdc (line 3). Now that the pblock is defined we can actually run a loop for all the remaining RMs. After place and route is done for each RMs with the static logic using the loop

Algorithm 2 TCL Automated Floorplan for Proposed Design Methodology: Pseudo Code

Input: Static$_{bram}$.dcp, PE$_0$.dcp, PE$_1$.dcp, PE$_2$.dcp, ...PE$_k$.dcp
Output: Static$_{bram}$.bit, Static$_{blank}$.bit, PE$_0$.bin, PE$_1$.bin, PE$_2$.bin, ...PE$_k$.bin
Begin:
1: open_checkpoint Static$_{bram}$.dcp
2: read_checkpoint -cell blackbox PE$_1$.dcp
3: set_properties & select pblock region -fplan.xdc
4: opt_design; place_design; route_design;
5: write_checkpoint PE$_0$.dcp
Loop Process
6: **for** $i = 1$ to $k + 1$ **do**
7: update_design to black_box
8: lock_design & write_checkpoint Static$_{bram}$.dcp
9: read_checkpoint -cell blackbox PE$_i$.dcp
10: opt_design; place_design; route_design;
11: write_checkpoint PE$_i$.dcp
12: **end for**
13: open_checkpoint Static$_{bram}$.dcp
14: update_design -buffer_ports -cell black_box
15: opt_design; place_design; route_design;
16: write_checkpoint Static$_{blank}$.dcp
17: pr_verify -initial PE$_0$.dcp -additional { PE$_1$.dcp ... PE$_k$.dcp Static$_{blank}$.dcp }
18: write_bitsream Static$_{blank}$.bit
Loop Process
19: **for** $j = 0$ to $k + 1$ **do**
20: open_checkpoint PE$_j$.dcp
21: write_bitsream PE$_j$.bit & write_cfgmem -format -BIN PE$_j$.bin
22: **end for**
End:

(line 6), the partial binaries are are generated as .bin files (line 19) and a blanking configuration also called as static$_{blank}$.bit which is a full bitstream to be loaded upon device bootup is generated (line 18). This configuration has LUTs tied to constants in order to ensure the outputs of the RP are not floating. Before generating binaries, it is important to verify that the static implementation, including interfaces with RP is consistent across all the configurations or RMs and hence (line 17) verification for all the generated DCPs post implementation stage needs to be done.

Table 6.1: Register Description in Custom IPs

ARM-FPGA Control Bus IP		
Abbreviated	**IP Adress Map**	**Description**
reg0	Base_Address + 1	FPGA BRAM Controller Read DONE
reg1	Base_Address + 2	FPGA BRAM Controller Write DONE
reg2	Base_Address + 3	BRAM MUX Select: 0-ARM, 1-FPGA
reg3	Base_Address + 4	FPGA BRAM Controller Read START
reg4	Base_Address + 5	FPGA BRAM Controller Write START
reg5	Base_Address + 6	Reconfigurable Partition Compute DONE
reg6	Base_Address + 7	Reconfigurable Partition Compute START
ARM Side BRAM Controller IP		
reg7	Base_Address + 0	Write Enable
reg8	Base_Address + 1	Address
reg9	Base_Address + 2	Data In
reg10	Base_Address + 3	Data Out

6.1.4 Stage 4: Deploying the Binaries on Zynq-7000

After generating the partial binaries for each reconfigurable module, a BOOT.bin is created using Xilinx SDK which contains the first stage image for Programmable Logic (PL) side as well as the user application C/C++ software to talk to the hardware on PL Side and is stored in SD card along with all the .bin files. The Zynq-7000 FPGA follows a redefined boot up process upon powering the device. After power-on reset, the Boot ROM determines the external memory interface or boot mode (SD flash memory) and the encryption status (non-secure). The Boot ROM uses the DevC's DMA to load the First Stage Boot Loader (FSBL) into on-chip RAM (OCM).The Boot ROM shuts down and releases CPU control to the FSBL which in turn configures the PL with the full Static_{blank}.bit via the Processor Configuration Access Port (PCAP). The device is now fully configured and operational for the standalone user software application. The partial bitstreams are loaded into DDR memory from SD card to maximize throughput during configuration. At this point, the application can use the partial bitstreams at any time to modify the pre-defined PL regions while the rest of the FPGA remains fully active and uninterrupted.

Table 6.1 shows the register mappings for ARM-FPGA Control Bus and ARM Side BRAM Controller IPs with respect to the Base Address for each of them. There is also a seperated

Algorithm 3 User Application Software Flow for Proposed Architecture: Pseudo Code

Input: Raw Data-in[0],in[1],in[2],..in[j], RM_0.bin, RM_1.bin,...RM_k.bin, BOOT.bin
Output: Final Output-out[0],out[1],out[2].... out[j]

Begin:

1: Load RM_0, RM_1,...RM_k & in[0],in[1],in[2],..in[j] from SD card to DDR3
2: **for** $m = 0$ to j **do**
3: reg2=0; reg7=1; reg8=m; reg9=in[m]; {ARM BRAM Write with Raw Inputs}
4: **end for**
5: **for** $i = 0$ to $k + 1$ **do**
6: Configure RP with RM_i through PCAP from DDR3
7: reg2=1; reg3=1;
8: **while** (!(reg0==1)) **do**
9: {Wait till BRAM Memory is being Read by FPGA BRAM Controller}
10: **end while**
11: reg6=1;
12: **while** (!(reg5==1)) **do**
13: {Wait till the Computation is completed by the RM_i}
14: **end while**
15: reg4=1;
16: **while** (!(reg1==1)) **do**
17: {Wait till BRAM Memory is being Written by FPGA BRAM Controller}
18: **end while**
19: reg4=0; reg6=0; reg3=0;
20: **end for**
21: **for** $m = 0$ to j **do**
22: reg2=0; reg7 =0; reg8=m; out[m]=reg10; {ARM BRAM Read for Processed Outputs}
23: **end for**

End:

column with abbreviated names for each register mappings in order to make the Algorithm 3 more understandable. This pseudo-code explains the entire flow of the user application that is suggested to work with proposed architecture as discussed. In Algorithm 3, line 2 represents a loop to load the BRAM with raw data from DDR3 memory using ARM Processor, line 5 represents an outer loop whose count depends upon the number of RMs in a dataflow process and line 21 represents a loop to read BRAM results into DDR3 memory using ARM Processor. The outer main loop (in line 5) consists of internal three while loops (lines 8, 12, 16) which waits for the respective flags to

be toggled high before proceeding to next statement. Before these loops run, it is important to load the RP with RM (line 6).

6.2 Spatial Design Implementation : JPEG Encoder

In order to prove the area utilization efficacy of the proposed design architecture, the dataflow computation process was implemented spatially first on FPGA and its post placement utilization area on FPGA as well as hardware running time was experimentally obtained. When using JPEG Encoder as dataflow computation process, it is important to measure the quality of the compressed image results and thus Structural Similarity Index (SSIM Index) was used as an image quality metric to validate the filtered image.

6.2.1 System Implementation and Setup

Using the Vivado IP Integrator, we connected the custom IPs of JPEG Encoder using AXI4 interconnect to the Processing System (PS) side of the Zynq FPGA. Thus, with this design approach all the processing elements are spatially allocated on the FPGA fabric increasing the overall area utilization. However, since all the processing elements are available without the need to reconfigure, there should be an improvement in runtime performance because the time required to transfer and load the partial bitstream is eliminated. There is also an `AXI Timer IP` from Xilinx which helps to calculate the number of clock cycles utilized and hence estimate the computation time, which was also added to system to measure time performances. The clock frequency for the FPGA Fabric was selected to be 50 MHz. We also utilized SD Card and DDR memory as additional hardware resources connected to the external interfaces on the Processing Side (PS) of the Zynq FPGA for storing data. Figure 6.2 shows the overall system using Vivado IP Integrator.

The system design is validated for any missing interconnects and output products are generated using global approach as well as top level wrapper is created. The Post-Placement utilization report as shown in Table 6.2 was obtained after synthesizing and implementing the design in Vivado. This utilization report was used to compare with the utilization reports of partial reconfiguration designs.

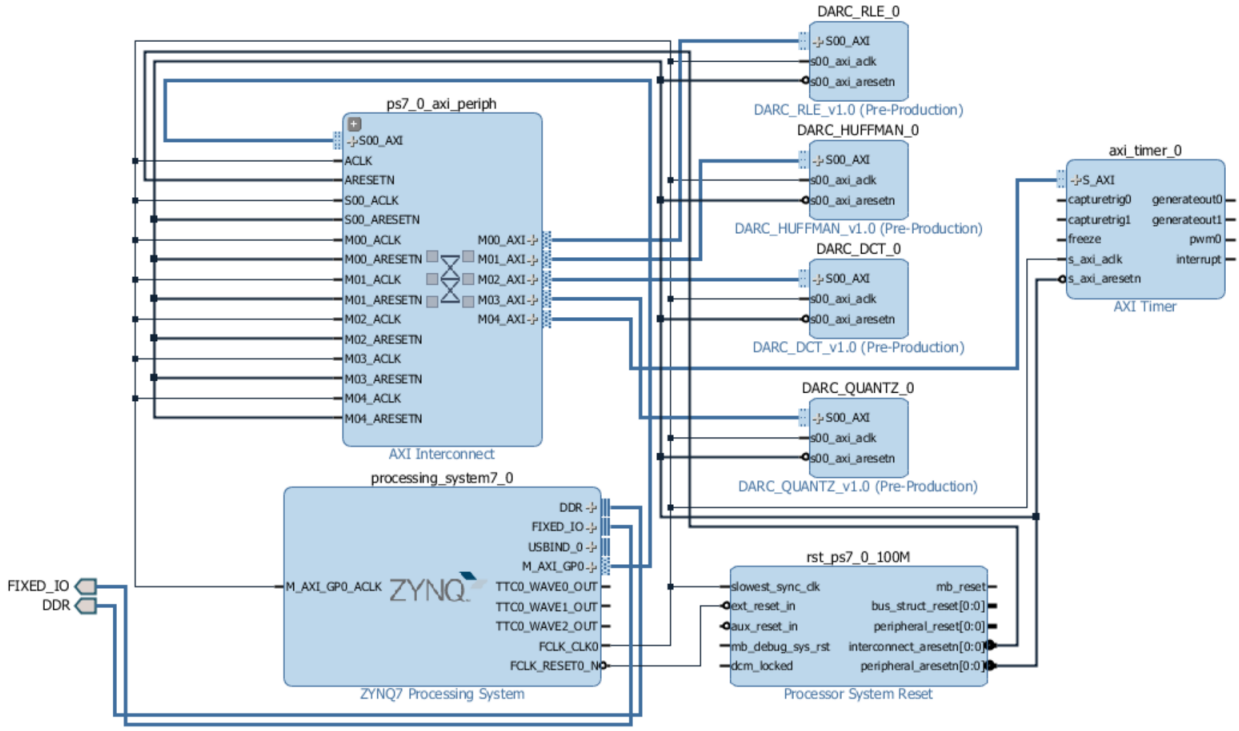
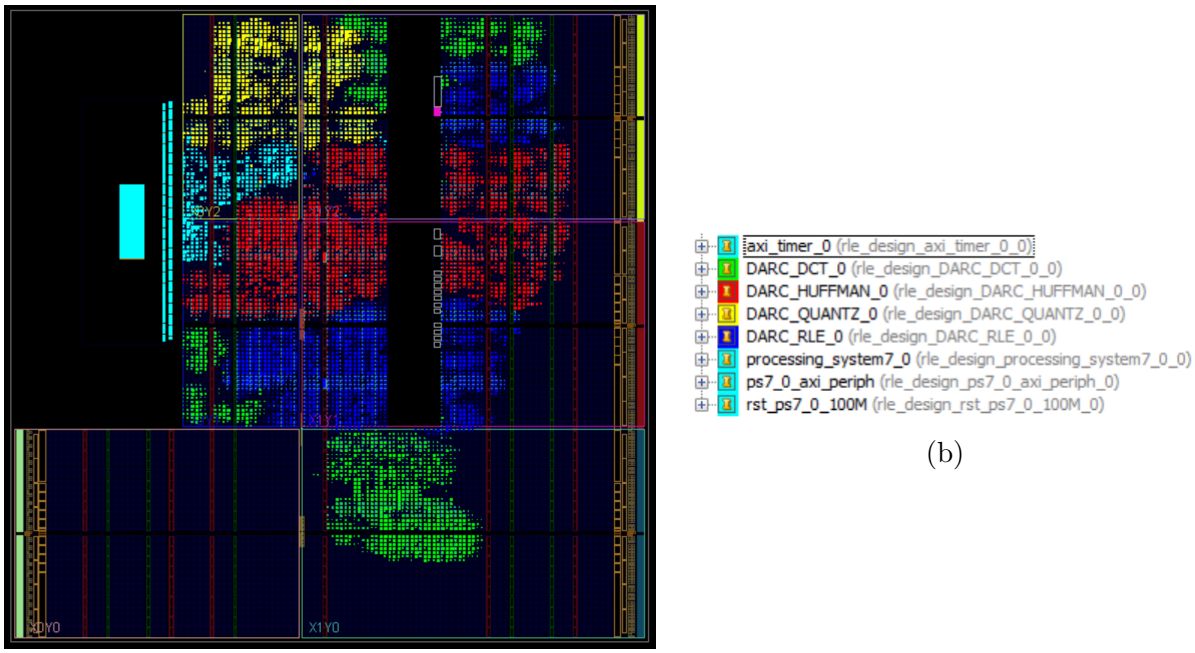


Figure 6.2: Spatial System Design view in Vivado IP Integrator

Table 6.2: Post-Placement Utilization Report

Sr.No	SiteType	Used	Available	Utilization
1	Slice LUTs	14997	53200	28.19
2	Slice Registers	24759	106400	23.27
3	F7 Muxes	2276	26600	8.56
4	F8 Muxes	1024	13300	7.70
5	Block RAM Tile	2	140	1.43
6	RAMB18	4	280	1.43



(a)

(b)

Figure 6.3: (a)-(b)Floorplan View of Spatial Implementation of JPEG Encoder on Zynq XC7Z020

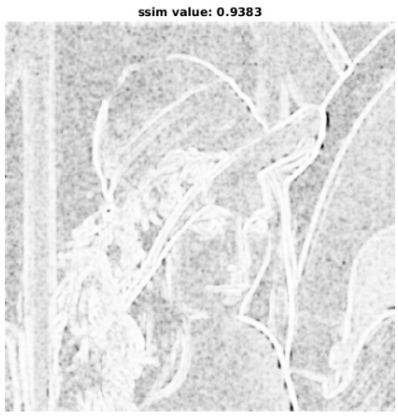
Figure 6.3 shows the floorplan view of spatial design implementation post place and route stage (Implementation Stage in Vivado).

6.2.2 Experimental Results

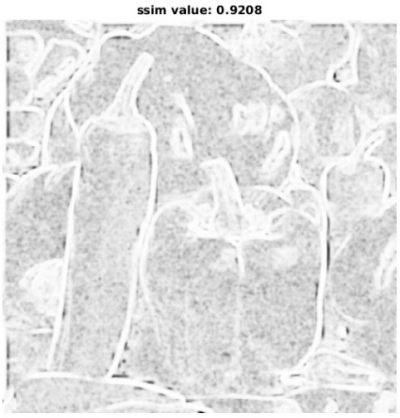
In order to test that all the corner cases are covered by the custom IPs generated through SystemC, we have tested the spatial design implementation with three grayscale images of 512 X 512 pixels: lena.bmp, peppers.bmp and goldhill.bmp. Table 6.3 shows the spatial design implementation results of compression ratios achieved, SSIM Values, FPGA Runtime and Huffman Bitlength for each of the testcase images. A uniform Quantization matrix was used in each case as discussed in previous chapter section 5.2.2. The FPGA runtime varies linearly with Huffman bitlength as expected which is observed in the table.

Table 6.3: JPEG Encoder Results with Spatial Design Implementation

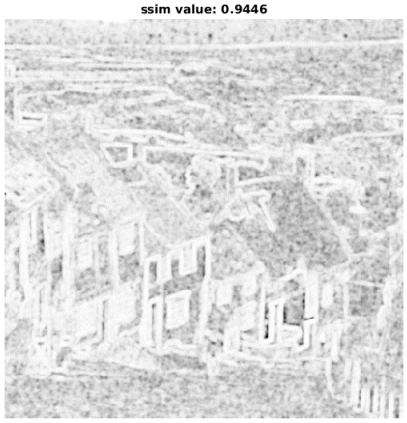
Sr.No	Filename	Original Size	Compressed Size	Encoder Ratio	FPGA Exe Time	SSIM Value	Huffman bitlength
1	Lena.bmp	258 KB	36 KB	7.17:1	1.815 sec	0.9383	283268
2	Peppers.bmp	258 KB	46 KB	5.60:1	1.842 sec	0.9208	357491
3	Goldhill.bmp	258 KB	54 KB	4.78:1	1.877 sec	0.9446	427483



(a)



(b)



(c)

Figure 6.4: SSIM values and graphs of (a) lena (b) peppers (c) goldhill

Figure 6.4 shows SSIM maps and values for the three test case images considered for study. The SSIM metric combines local image structure, luminance, and contrast into a single local quality score. In this metric, structures are patterns of pixel intensities, especially among neighboring

pixels, after normalizing for luminance and contrast. Because the human visual system is good at perceiving structure, the SSIM quality metric agrees more closely with the subjective quality score. Because structural similarity is computed locally, 'ssim' can generate a map of quality over the image.

The SSIM metric value closer to 1 shows highest metric of image quality between the original image and distorted (compressed) image. The difference with respect to other techniques such as MSE or PSNR is that these approaches estimate absolute errors whereas SSIM is a perception-based model that considers image degradation as perceived change in structural information, while also incorporating both luminance masking and contrast masking terms. Luminance masking is a phenomenon whereby image distortions (in this context) tend to be less visible in bright regions, while contrast masking is a phenomenon whereby distortions become less visible where there is significant activity or "texture" in the image.

6.3 Implementation using DDR3 Memory [$\mathbf{PR}_{DDR}$]: JPEG Encoder

In order to prove that the proposed design architecture is runtime and latency efficient compared to traditional approach of using DDR memory for storing large intermediate data results in a partial reconfiguration based design methodology for dataflow computation process, we implemented the JPEG Encoder, as dataflow computation process, on FPGA using this traditional approach, which we would like to abbreviate as **$\mathbf{PR}_{DDR}$ Design Methodology**.

6.3.1 System Implementation and Setup

The ZedBoard, which is used as a hardware platform for our research study, includes two Micron DDR3 128 Megabit x 16 memory components creating a 32-bit interface, totaling 512 MB. The DDR3 is connected to the hard memory controller in the Processor Subsystem (PS). The DDR memory controller is configured for 32-bit wide accesses to a 512 MB address space. The DDR3 uses 1.5V SSTL-compatible inputs and termination is utilized on the ZedBoard. The Zynq-7000 AP SoC and DDR3 ICs have been placed close together keeping traces short and matched. In the

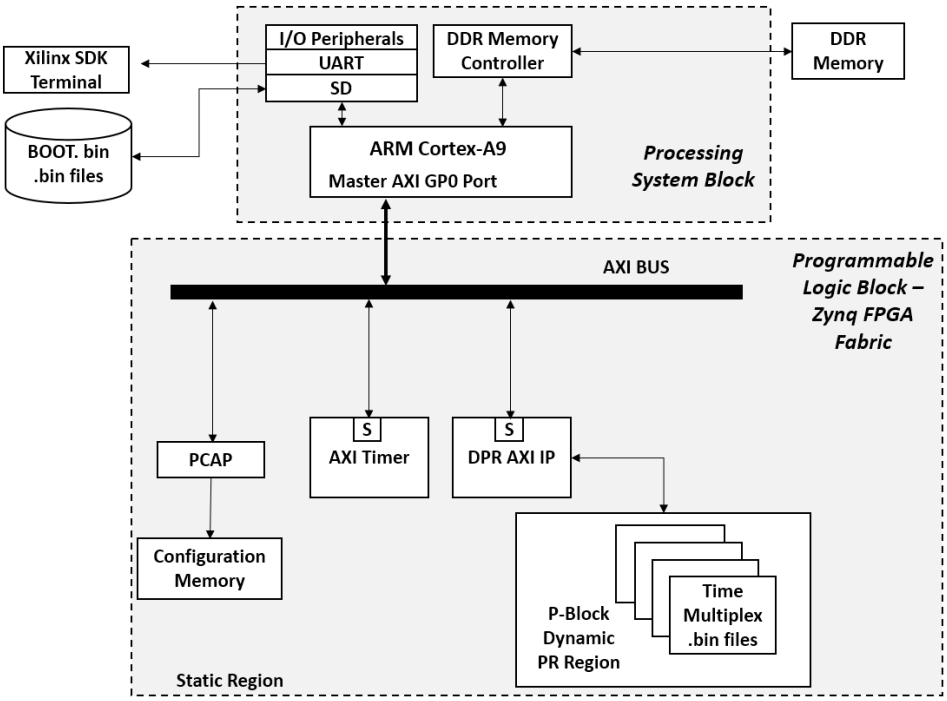


Figure 6.5: PR_{DDR} design methodology Block Diagram for dataflow computation

user application software, this DDR3 memory is referenced using pointers to the address mappings provided in the systems.hdf file generated when the hardware is exported from Xilinx Vivado to SDK.

Figure 6.5 shows the PR_{DDR} design methodology block diagram and the hardware resources required to implement. The Programmable Logic Block - Zynq FPGA Fabric shows the `DPR AXI IP` which is **D**ynamic **P**artial **R**econfiguration custom IP developed containing only the input-output instance of the Reconfigurable Partition (RP) with AXI4-Lite interconnect written in verilog and packaged using IP Packager in Vivado. The input-output instance depends upon the maximum number and data-widths amongst processing elements of dataflow computation process as studied in previous section of this chapter. This basically synthesizes as a black box since there is no actual logic. This black box is the cell which we would be referring henceforth as **pblock Reconfiguration Partition or pblock RP**, which would be reconfigured partially at runtime and would be used in TCL based automated floorplanning to allocate area on fabric and generate partial binary files of each processing elements. We have also used `AXI Timer IP` provided in the Xilinx IP catalog,

Table 6.4: Post Placement Utilization Report: PR_{DDR} Design Methodology

Sr.No	SiteType	A_{static}	$A_{dynamic}$	Available	Utilization (A_{total})
1	LUT	4119	6692	53200	20.32 %
2	LUTRAM	68	72	17400	0.80 %
3	FF	6018	9432	106400	14.52 %
4	Block RAM Tile	-	3	140	2.14 %

which would be used for measuring time performances. AXI-PCAP (Processor Access Control Port) Bridge is used to configure partial and full bitstreams, from the ARM Cortex Side during device bootup stage and runtime, to program the FPGA fabric. The user application loads the partial bitstreams into DDR memory upon start-up. This was done to maximize the configuration throughput over the PCAP interface and hence speed up the configuration time and take advantage of caching. The full, partial bitstreams along with user application to run on ARM Processor are stored in a non-volatile SD card memory. The BOOT.bin contains the user software application and first stage Programmanble Logic (PL) side full bitstream.

Figure 6.6 shows the PR_{DDR} design methodology systems-view in Xilinx Vivado IP Integrator for JPEG Encoder dataflow process. Since, in the JPEG Encoder the maximum number of input & outputs are contained in the RLE Processing element block and maximum datawidths of input & output is contained in the Huffman Encoding Processing element block, we have designed a specific IP: `REV2_JPEG_IP_DESIGN_0` to account for the pblock RP as seen in the figure. The system was synthesized using Vivado Synthesis Tool and $Static_{ddr}$.dcp was obtained. All the Processing Elements of JPEG Encoder were synthesized seperately and their .dcp files were generated. The processing elements of JPEG Encoder would be referred as Reconfigurable Modules (RM) in the partial reconfiguration based design methodology. The $Static_{ddr}$.dcp contains a blackbox instance to load Reconfigurable Modules and perform place & route the design to generate partial and full bitstreams using the TCL Automated Floorplan as discussed in Algorithm 2.

$$A_{total} = \sum \{A_{static} + A_{dynamic}\{max(RM_0, RM_1, RM_2, ...RM_k)\}\} \tag{6.1}$$

Table 6.4 shows the utilization area report and contains fpga resources utilized by static and dynamic portion in seperate columns. The total fpga area utilization was calculated using the

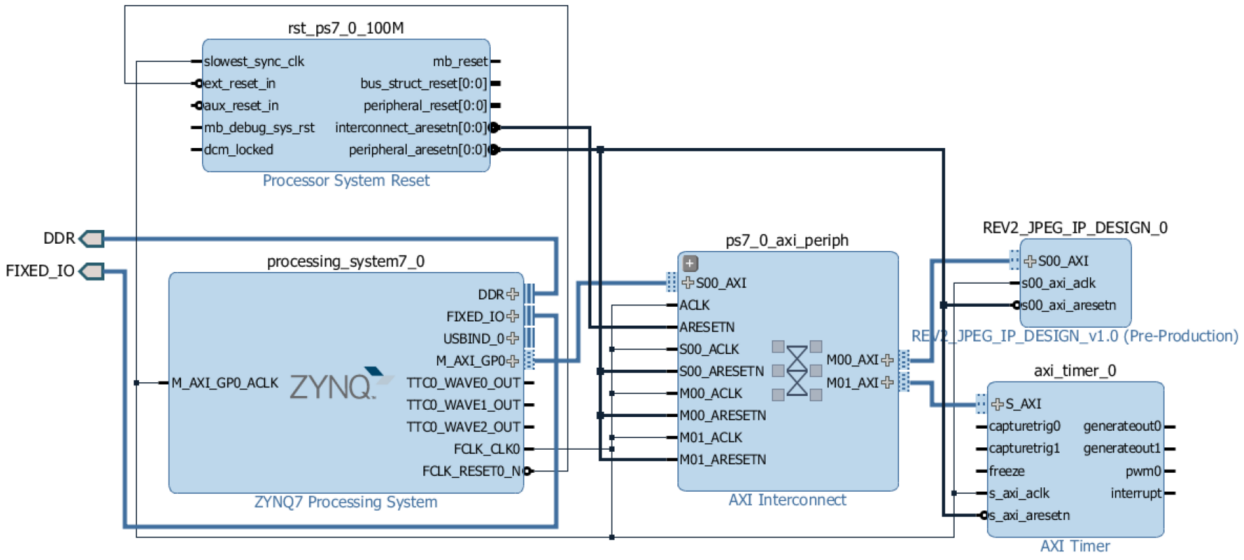
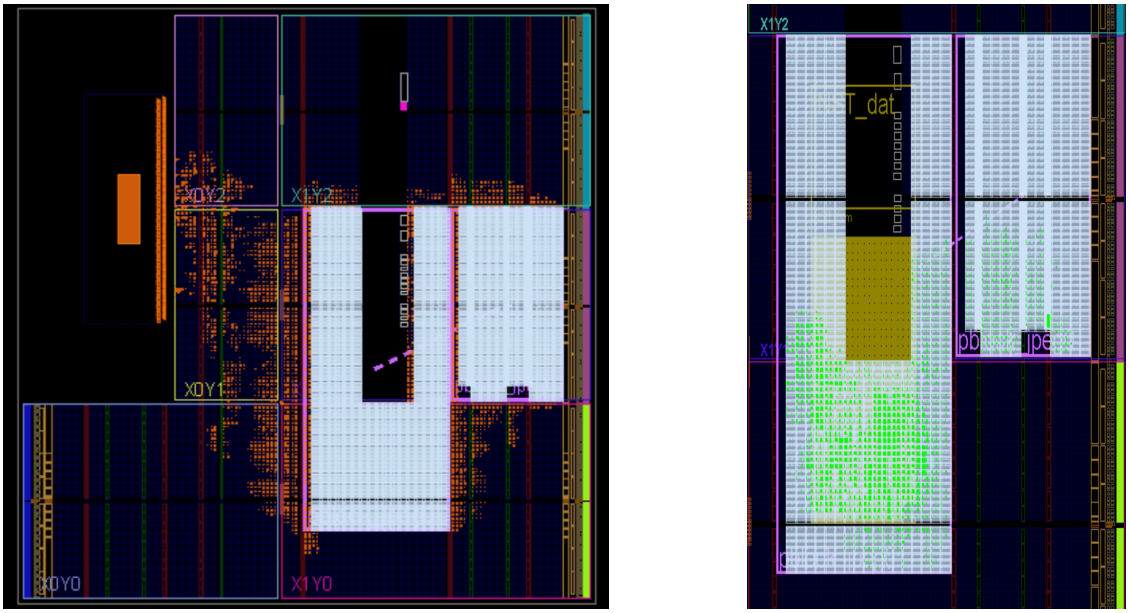


Figure 6.6: PR_{DDR} system view in Vivado IP Integrator

equation 6.1 and is shown in % form in the last column in Table 6.4. The value of **k** is 4 in the case of JPEG Encoder and Reconfigurable Modules (RM) are DCT, Quantization, Run-Length Encoding and Huffman Encoding. Thus, the maximum count of each fpga resource required by RMs is accounted in the table.

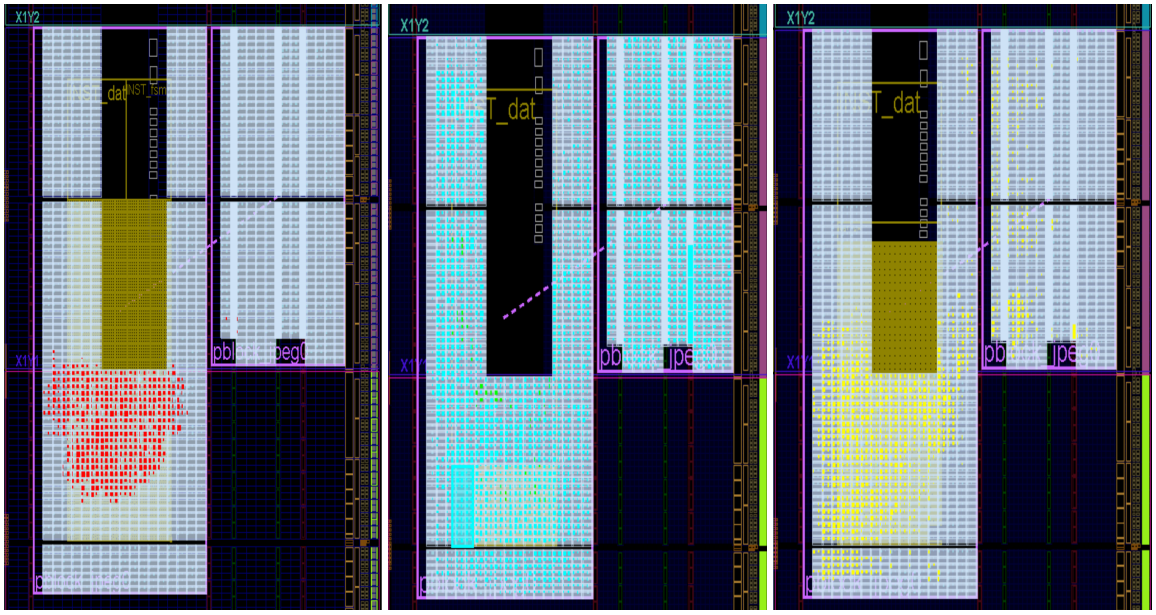
6.3.2 Experimental Results

Figures 6.7 and 6.8 show the floorplan view post placement and routing before generating partial and full bitstreams. After generating the .bin files for each reconfigurable modules of JPEG Encoder and the $Static_{blank}$.bit, the SD Card BOOT.bin image was created using Xilinx SDK. The clock on FPGA Programmable Logic fabric was selected to be 50 MHz. The user application software had a controlled loop to partially reconfigure each reconfigurable module after the previous results were stored in DDR memory. The testcase images were same as previously used in the spatial design experimentation viz; lena.bmp, peppers.bmp and goldhill.bmp. Since these are 512 X 512 pixel grayscale images and the reconfigurable modules process on each 8 X 8 pixels every iteration time, there are 4096 blocks of 8 X 8 pixels in the entire image. Experiments were performed by varying the number of times partial reconfiguration occurs over these 4096 blocks thereby generating varying



(a) (b)

Figure 6.7: Design Checkpoints after performing Place & Route (a) Static_{ddr}.dcp (b) DCT_{ddr}.dcp for $\text{RP}_{Bitsize}$ = 1306.272 KB case



(a) (b) (c)

Figure 6.8: Design Checkpoints after performing Place & Route (a) $\text{Quantization}_{ddr}$.dcp (b) RLE_{ddr}.dcp (c) Huffman_{ddr}.dcp for $\text{RP}_{Bitsize}$ = 1306.272 KB case

numbers in runtime and latency. Also, additional experiments were performed by varying the area of pblock and observing how the runtime is affected as shown in Tables 6.5 , 6.6 and 6.7.

Table 6.5: JPEG Encoder Results for lena.bmp with PR_{DDR} Design Implementation

		$RP_{Bitsize}$ = 3416.088 KB		$RP_{Bitsize}$ = 1306.272 KB		
Sr.No	N_{pr}	$T_{latency}$ (s)	$T_{runtime}$ (s)	$T_{latency}$ (s)	$T_{runtime}$ (s)	Samples
1	4	5.157	5.157	3.75413	3.75413	4096
2	8	3.418	6.836	2.19865	4.39731	2048
3	16	2.549	10.194	1.42036	5.68142	1024
4	32	2.114	16.911	1.03122	8.24978	512
5	64	1.897	30.344	0.83664	13.38619	256
6	128	1.788	57.211	0.73934	23.65878	128
7	256	1.733	110.937	0.69071	44.2053	64

Table 6.6: JPEG Encoder Results for goldhill.bmp with PR_{DDR} Design Implementation

		$RP_{Bitsize}$ = 3416.088 KB		$RP_{Bitsize}$ = 1306.272 KB		
Sr.No	N_{pr}	$T_{latency}$ (s)	$T_{runtime}$ (s)	$T_{latency}$ (s)	$T_{runtime}$ (s)	Samples
1	4	5.221	5.221	3.82025	3.82025	4096
2	8	3.45	6.9	2.23123	4.46246	2048
3	16	2.56475	10.259	1.43666	5.74662	1024
4	32	2.12188	16.975	1.03937	8.31492	512
5	64	1.90056	30.409	0.84071	13.4514	256
6	128	1.78984	57.275	0.74137	23.72387	128
7	256	1.73448	111.007	0.69173	44.27053	64

Table 6.7: JPEG Encoder Results for peppers.bmp with PR_{DDR} Design Implementation

		$RP_{Bitsize}$ = 3416.088 KB		$RP_{Bitsize}$ = 1306.272 KB		
Sr.No	N_{pr}	$T_{latency}$ (s)	$T_{runtime}$ (s)	$T_{latency}$ (s)	$T_{runtime}$ (s)	Samples
1	4	5.185	5.185	3.78413	3.78413	4096
2	8	3.4325	6.865	2.21312	4.42623	2048
3	16	2.55575	10.223	1.42758	5.71033	1024
4	32	2.1175	16.94	1.03484	8.27872	512
5	64	1.89831	30.373	0.83845	13.41515	256
6	128	1.78875	57.24	0.74024	23.68778	128
7	256	1.73397	110.974	0.69116	44.23426	64

It can be observed looking at Figures 6.9 and 6.10 that runtime and latency have inverse relationship, latency and throughput have linear relationship, when number of times partial reconfiguration occurs is varied in each case. Thus, if a certain application demands greater throughput, then latency is higher but the overall running time is less due to fewer times the reconfiguration modules are configured to the reconfigurable partition for computation. On the other hand if latency required for an application is to be the least, then every iteration time units there are some samples available at the output but however the running time takes a hit because now smaller portions of data blocks are being processed and hence more number of dynamic partial reconfiguration is required. Hence, there is a trade-off curve depending upon the requirements. However, fewer fpga resources are required to implement PR_{DDR} design methodology compared to spatial design implementation as can be observed by comparing Tables 6.4 and 6.2.

We also noted that the size of the partial bitstream of the reconfiguration modules is a critical parameter which affects the running time performance. It is intuitive to think about it that larger the size of binary, greater is the time required to load the partial bitstream through PCAP port to Configuration Memory on PL-Side. The size of partial bitstream corresponding to reconfigurable modules depends upon the area of pblock. Hence, we carried out experiments to observe the variation in latencies and running time by altering the pblock area.

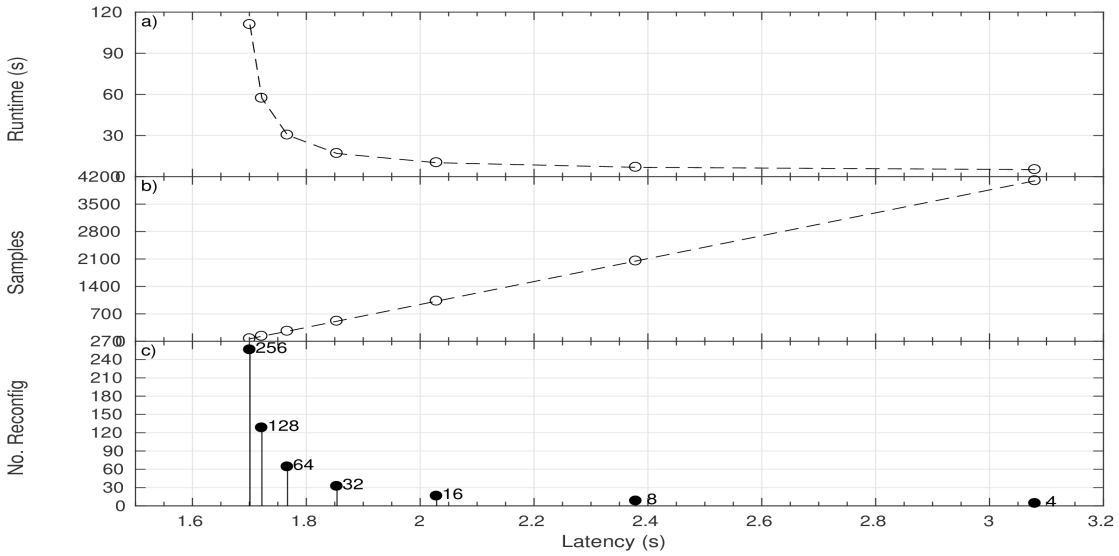


Figure 6.9: PR_{DDR} results for lena.bmp testcase with $\mathrm{RP}_{Bitsize}$ = 3416.088 KBytes

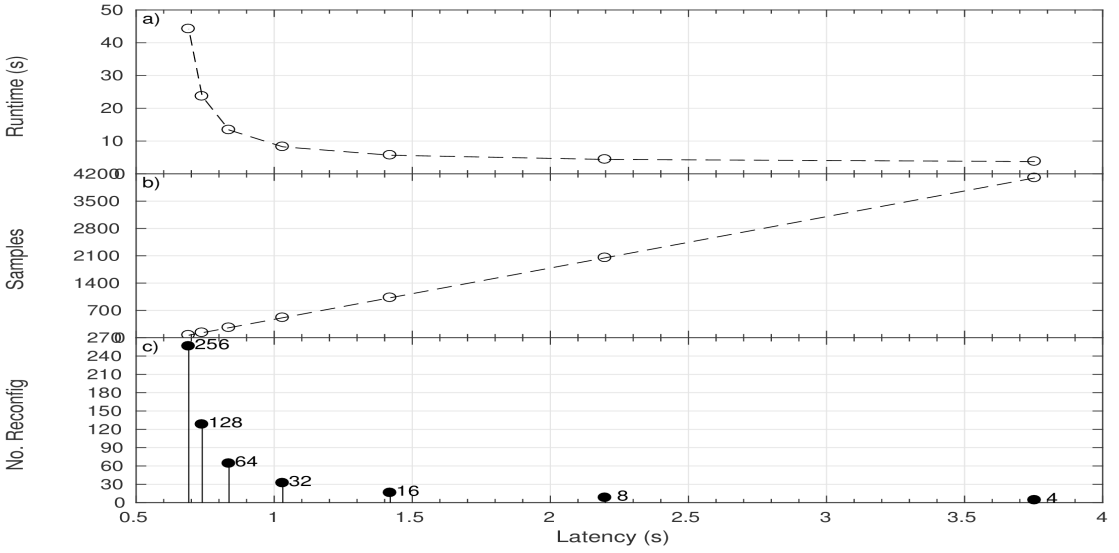


Figure 6.10: PR_{DDR} results for lena.bmp testcase with $\mathrm{RP}_{Bitsize}$ = 1306.272 KBytes

6.4 Proposed Architecture PR_{BRAM} Implementation: JPEG Encoder

Finally in this section, we will be implementing the running dataflow computation process - JPEG Encoder, using the Proposed Design Architecture abbreviated as PR_{BRAM} which is described in detail in Section 6.1 and discuss the experimental results obtained. As mentioned in Section 6.3.2, the area of pblock impacts the partial bitstream size, which in turn affects the overall runnning time and this analysis will be covered in this section with the help of three different pblock areas. We also propose in this section a mathematical prediction model which helps to estimate the running time given any dataflow computation process parameters without the need to compute experimentally.

6.4.1 System Implementation and Setup

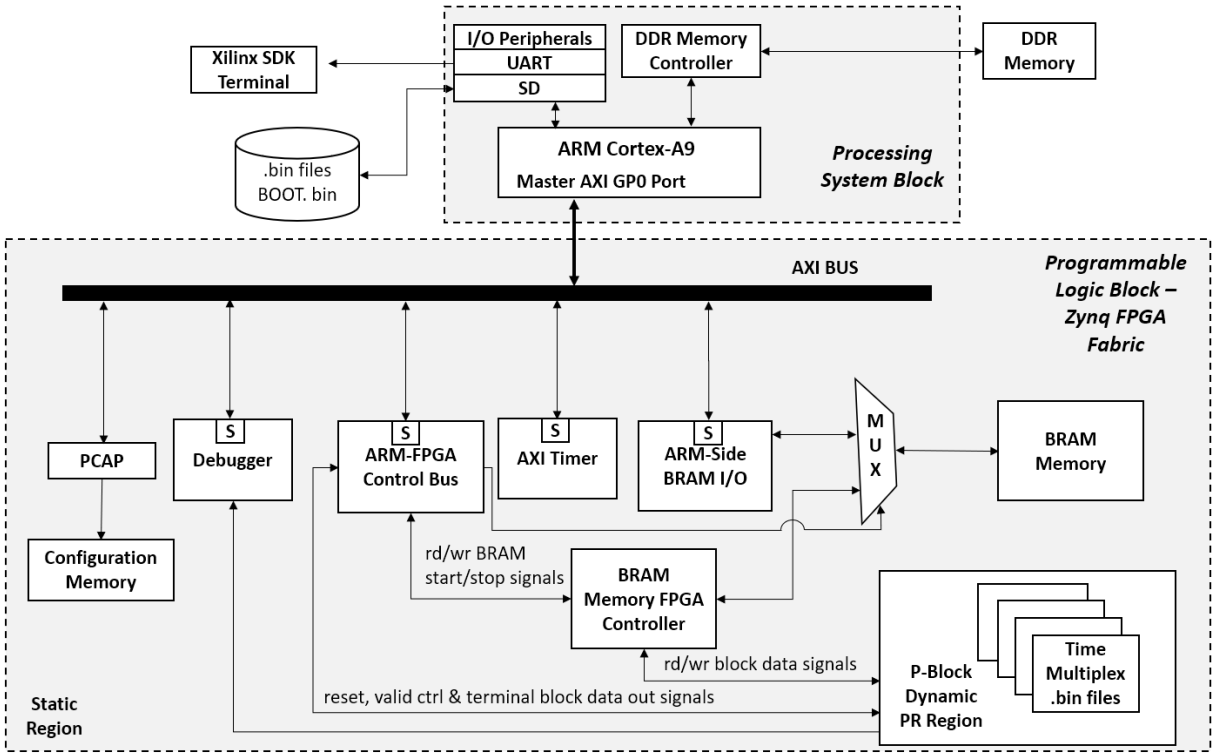


Figure 6.11: Proposed System Model

Figure 6.11 shows the overall proposed PR_{BRAM} design architecture which is discussed in great detail in section 6.1. As we have discussed earlier that, we have used Zedboard for performing all

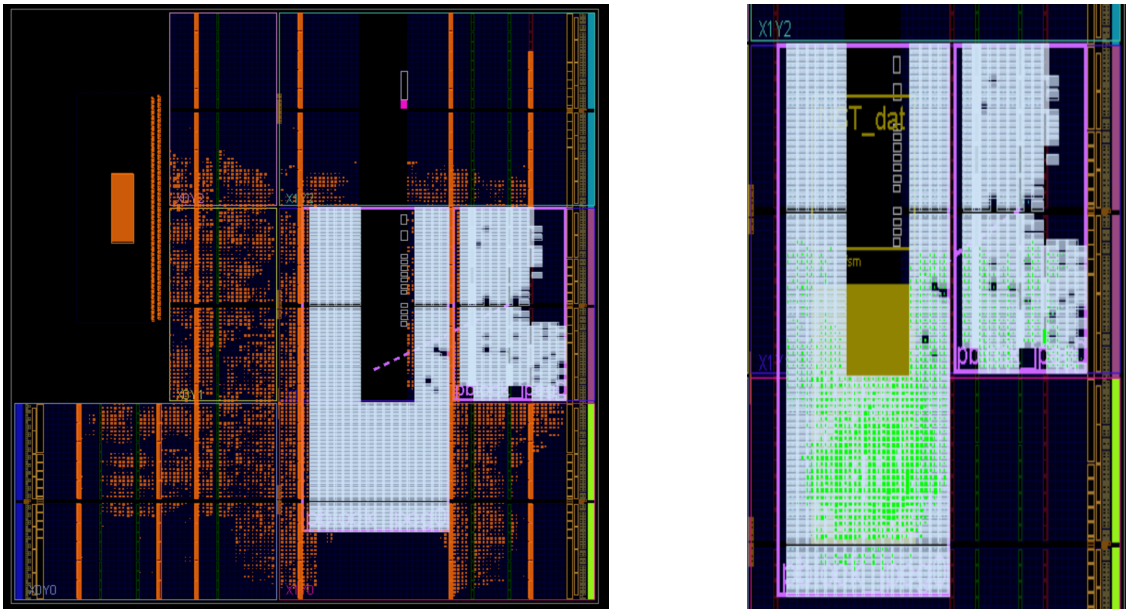
Table 6.8: Post Placement Utilization Report: PR_{BRAM} Design Methodology

Sr.No	SiteType	A_{static}	$A_{dynamic}$	Available	Utilization (A_{total})
1	LUT	5681	6692	53200	23.25 %
2	LUTRAM	68	72	17400	0.80 %
3	FF	12967	9432	106400	21.05 %
4	Block RAM Tile	128	3	140	93.57 %

the experiments, the board has Zynq -7000 AP SoC FPGA from Xilinx with device part number : XC7Z020. This device contains 140 blocks of Block RAM memory where each block comprises of 36Kb in size and the total sums upto 4.9 Mb. We are utilizing 91.42 % of Block RAM memory for the purpose of storing intermediate data results which would be the inputs to the next reconfigurable module. The total memory requirements for implementing JPEG Encoder is 1048.576 KBytes, which is more than the available Block RAM. Thus, we divided the image dataset of 4096 8 X 8 pixel blocks into two sets and reload the BRAM memory after the computation is done for the first half of image. Thus, we are restricted to the minimum number of partial reconfigurations in this case to 8. The XC7Z020 device part on Zedboard has the maximum Block RAM available considering Artix-7 based Zynq 7000 AP SoC Chip. In Kintex-7 based Zynq 7000, there are huge amounts of Block RAM available ranging from 265 -755 Blocks of 36 Kb each.

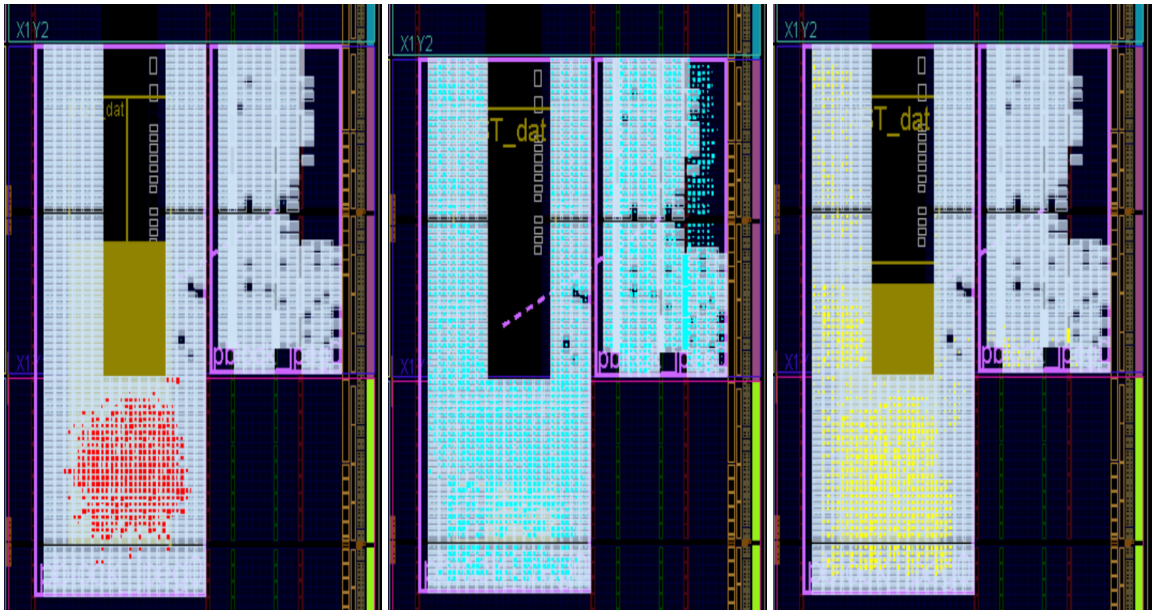
6.4.2 Experimental Results

Figures 6.12 and 6.13 shows the post place & route design checkpoints for $RP_{Bitsize}$ = 1306.272 KB & Figures 6.14 and 6.15 shows the same for $RP_{Bitsize}$ = 786.664 KB. Table 6.8 shows the utilization report for $RP_{Bitsize}$ = 1306.272 KB case. The .bin partial files for reconfiguration modules and $static_{blank}$.bit full bitstream are generated using TCL Automated Floorplan as discussed in Algorithm 2. BOOT.bin was created to start automatic device configuration for both PS & PL after device power-up. The working of the proposed architecture is explained in great detail in Section 6.1.4. The ARM Processor requires to write the test image data twice as described earlier in this section due to limitations of BRAM memory available in XC7Z020 however the overall user software architecture remains intact as explained in Algorithm 3 .



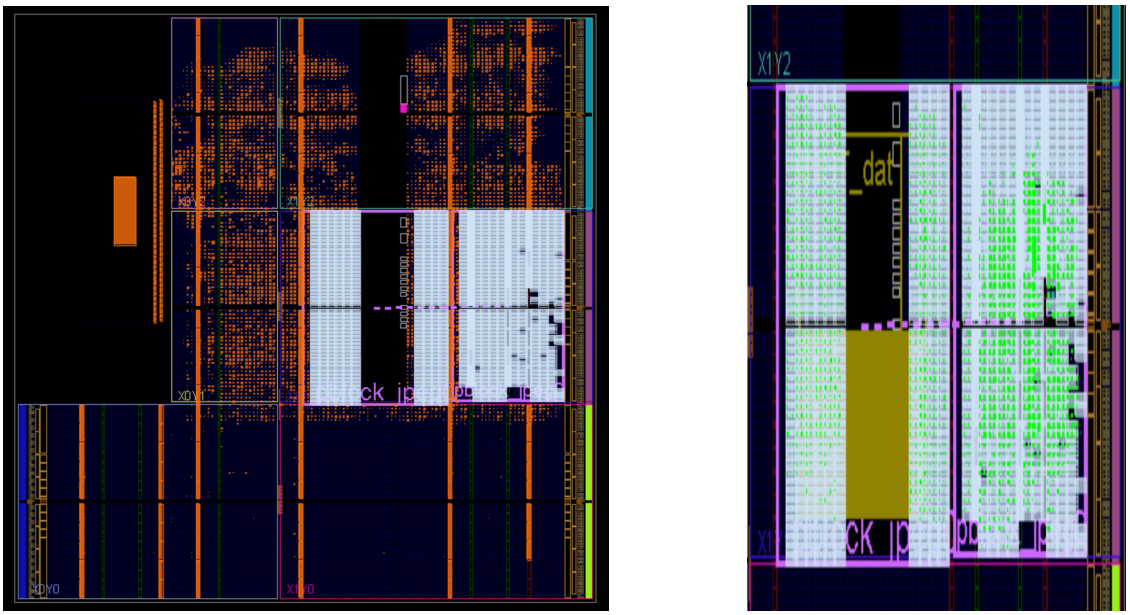
(a) (b)

Figure 6.12: Design Checkpoints after performing Place & Route (a) Static$_{bram}$.dcp (b) DCT$_{bram}$.dcp for RP$_{Bitsize}$ = 1306.272 KB case



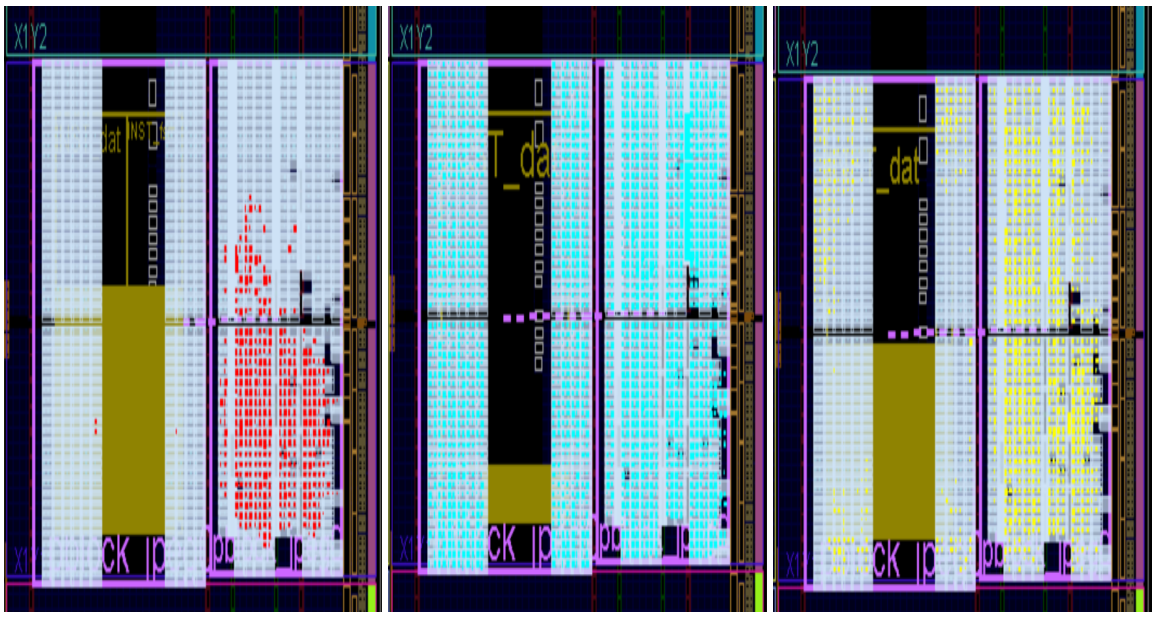
(a) (b) (c)

Figure 6.13: Design Checkpoints after performing Place & Route (a) Quantization$_{bram}$.dcp (b) RLE$_{bram}$.dcp (c) Huffman$_{bram}$.dcp for RP$_{Bitsize}$ = 1306.272 KB case



(a) (b)

Figure 6.14: Design Checkpoints after performing Place & Route (a) Static$_{bram}$.dcp (b) DCT$_{bram}$.dcp for RP$_{Bitsize}$ = 786.664 KB case



(a) (b) (c)

Figure 6.15: Design Checkpoints after performing Place & Route (a) Quantization$_{bram}$.dcp (b) RLE$_{bram}$.dcp (c) Huffman$_{bram}$.dcp for RP$_{Bitsize}$ = 786.664 KB case

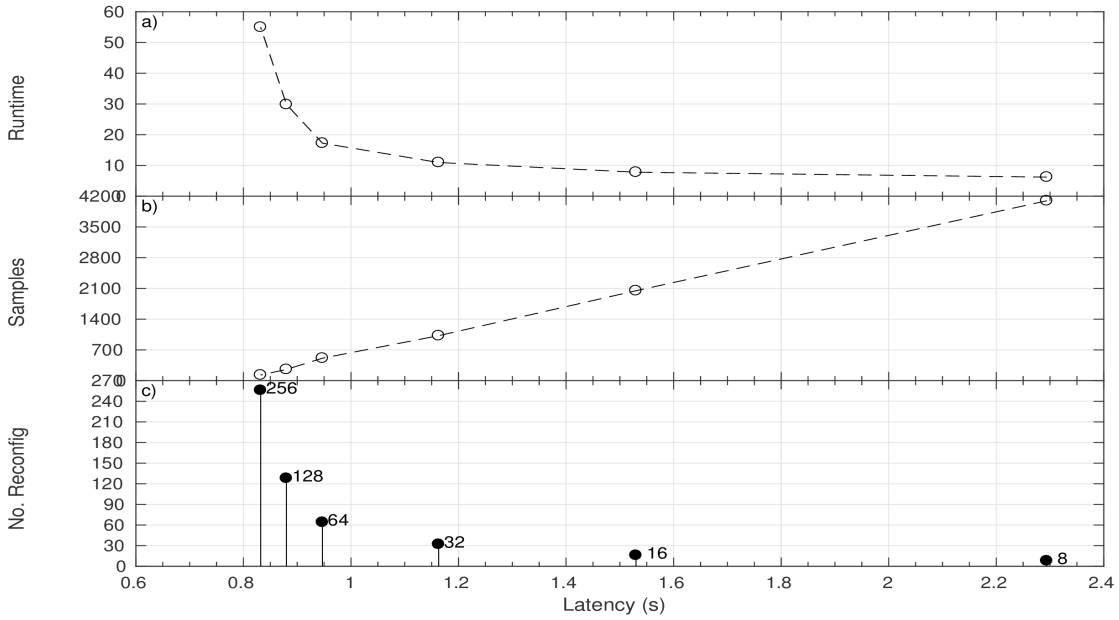


Figure 6.16: PR_{BRAM} results for lena.bmp testcase with $\mathrm{RP}_{Bitsize}$ = 1598.896 KBytes

Experimental results of testcase lena image are tabulated in Table 6.9 for three different pblock sizes and graph is plotted in Figure 6.16 for $\mathrm{RP}_{Bitsize}$ = 1598.896 KBytes to understand the relationship between latency and runtime as well as latency and throughput for varying number of partial reconfigurations. A similar observation can be made looking at the graph as in the case of PR_{DDR} implementation, that the runtime and latency are inversely related & latency and throughput are linearly related. Table 6.10 shows additional experimental results with $\mathrm{RP}_{Bitsize}$ = 1598.896 KBytes for testimages goldhill and peppers.

It can be observed from Table 6.9 that reducing to the most optimized size of pblock improves the runtime performance. Figure 6.17 shows the improvement in running time when the size of partial bitstream, $\mathrm{RP}_{Bitsize}$ is 786.664 KB. It can also be observed that the running time improves significantly in the case when number of reconfigurations is high for instance in Figure 6.17d, which is 512. Table 6.11 shows the reconfiguration time required for each RMs in each of the three cases of different partial bitsize. FPGA hardware running time can be estimated using equation 6.2.

Table 6.9: JPEG Encoder Results for lena.bmp with PR_{DDR} Design Implementation

	$RP_{Bitsize}$ = 1598.896 KB		$RP_{Bitsize}$ = 1306.272 KB		$RP_{Bitsize}$ = 786.664 KB		
N_{pr}	$T_{latency}$ (s)	$T_{runtime}$ (s)	$T_{latency}$ (s)	$T_{runtime}$ (s)	$T_{latency}$ (s)	$T_{runtime}$ (s)	Samples
8	2.285	4.57	1.93353	3.86706	1.65142	3.30284	2048
16	1.5375	6.15	1.28779	5.15115	1.0191	4.07638	1024
32	1.16313	9.305	0.96495	7.71959	0.7029	5.62322	512
64	0.97444	15.591	0.80353	12.85653	0.5448	8.71677	256
128	0.88022	28.167	0.72282	23.13036	0.46575	14.9039	128
256	0.83309	53.318	0.68247	43.67791	0.42622	27.27837	64
512	0.80952	103.619	0.64936	83.11808	0.40646	52.02699	32

Table 6.10: JPEG Encoder Results for goldhill.bmp and peppers.bmp with PR_{BRAM} Design Implementation

$RP_{Bitsize}$ = 1598.896 KB							
		goldhill.bmp		peppers.bmp			
Sr.No	N_{pr}	$T_{latency}$ (sec)	$T_{fpgaexetime}$ (sec)	$T_{latency}$ (sec)	$T_{fpgaexetime}$ (sec)	$T_{overhead}$ (sec)	Samples
1	8	2.3155	4.631	2.305	4.610	1.639	2048
2	16	1.551	6.204	1.545	6.183	1.657	1024
3	32	1.168	9.347	1.165	9.325	1.658	512
4	64	0.977	15.634	0.975	15.613	1.658	256
5	128	0.881	28.210	0.880	28.190	1.657	128
6	256	0.834	53.361	0.833	53.340	1.657	64
7	512	0.809	103.662	0.809	103.640	1.657	32

Table 6.11: RT_{BRAM} values for varying RP_{BRAM}

Sr. No	$RP_{Bitsize}$	Reconfiguration Time (RT_{BRAM})
1	1598.896 KB	0.1975 s
2	1306.272 KB	0.1605 s
3	786.664 KB	0.0966 s

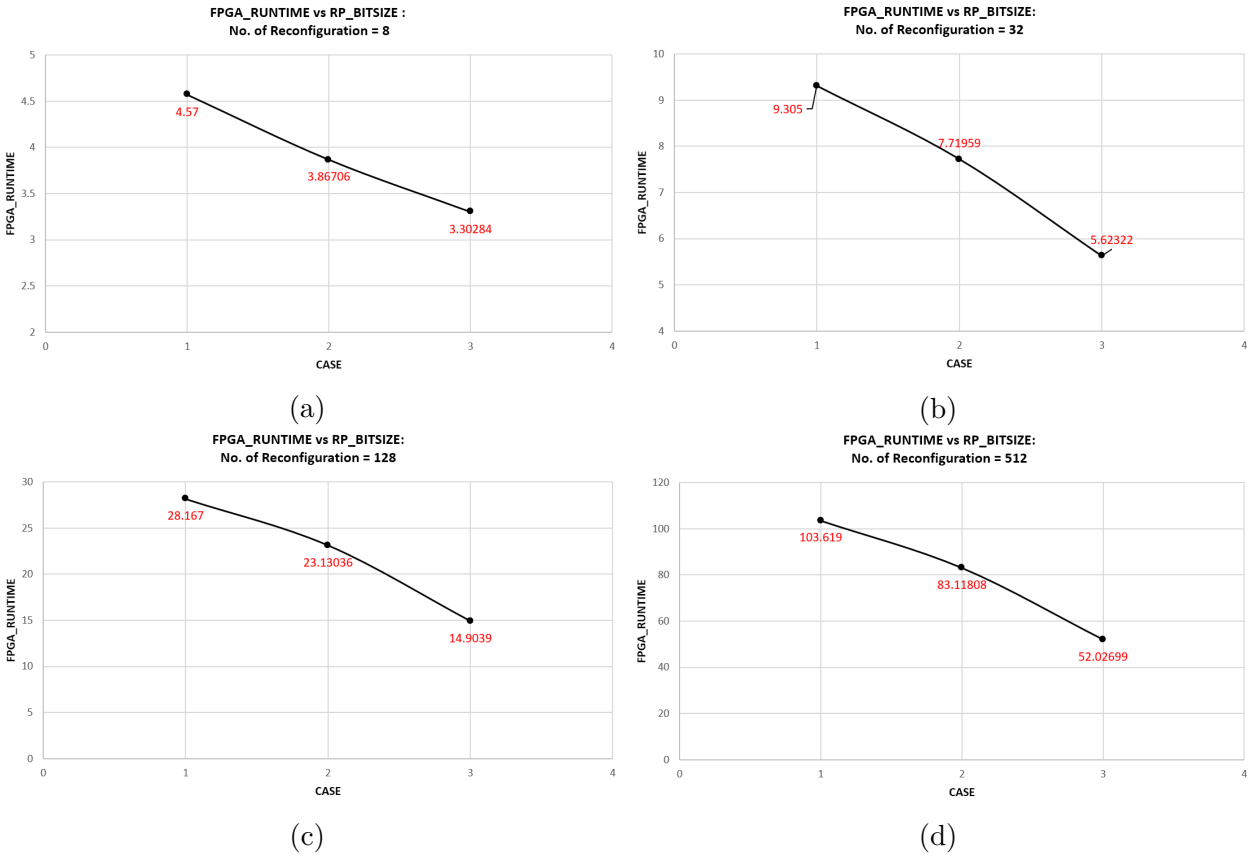


Figure 6.17: Graph Plots of FPGA Running Time vs Reconfigurable Partition Bitstream sizes of lena test image for varying number of partial reconfigurations (a) 8 (b) 32 (c) 128 (d) 512 [Case 1:$RP_{Bitsize}$ = 1598.896 KB, Case 2: $RP_{Bitsize}$ = 1306.272 KB, Case 3: $RP_{Bitsize}$ = 786.664 KB]

$$T_{runtime} = \{T_{jpeg-computing} + T_{overhead} + T_{bin} * N_{bin}\} \tag{6.2}$$

Referring to Table 6.9 and Equation 6.2, we can obtain the values of $T_{jpeg-computing}$, which is the actual computing time it takes for for processing all the inputs of each reconfigurable module; T_{bin} is the time it takes to partially configure the bitstream from DDR memory through PCAP port to configure programmable logic in reconfigurable partition; $T_{overhead}$ is the time it takes to load the partial binaries and raw image data from sd card to DDR memory.

The values T_{bin} = 0.1975 s, $T_{jpeg-computing}$ = 2.994 s and $T_{overhead}$ = 1.675 s are obtained experimentally. Using equation 6.2, we predicted the values of runtime as seen in Figure 6.18.

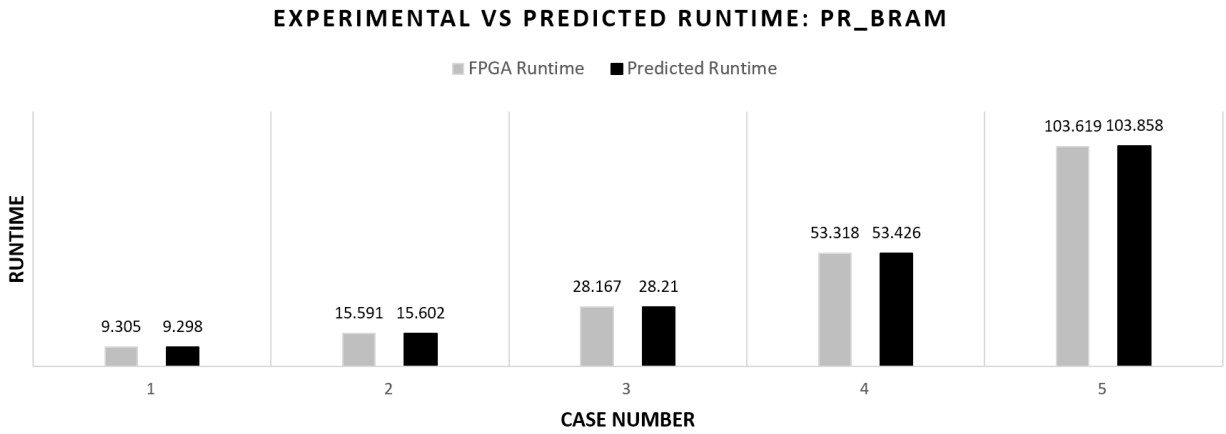


Figure 6.18: Experimental vs Predicted Results for $\mathrm{RP}_{Bitsize}$ = 1598.896 KB for varying cases of number of reconfigurations (1) 32 (2) 64 (3) 128 (4) 256 (5) 512

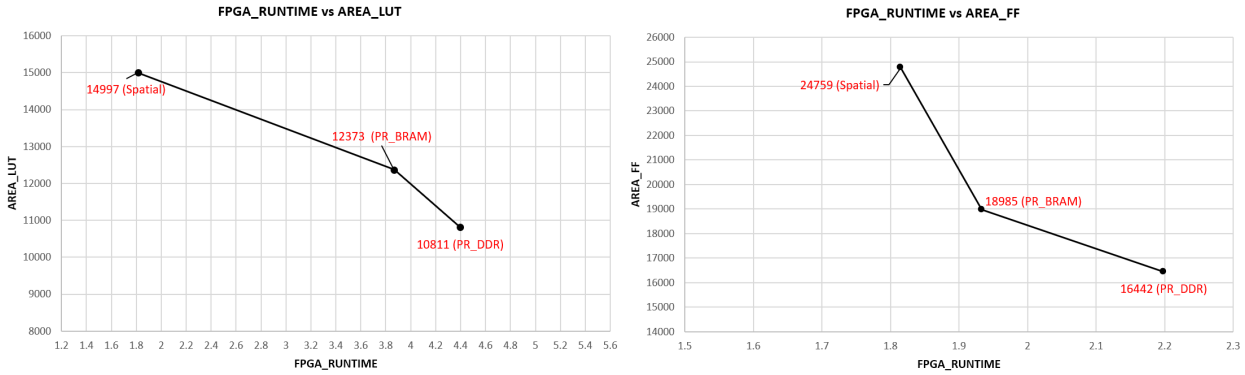
6.5 Results and Analysis of Design Implementations : Comparative Study

6.5.1 Area vs Runtime Comparison

Figure 6.19 is obtained from data as seen in Tables 6.2, 6.4 and 6.8. It can be observed that PR_{BRAM} design method requires slightly more area compared to PR_{DDR} due addition of ARM-FPGA Control Bus, ARM-Side BRAM Control, MUX and Block RAM Memory modules but however with respect to spatial design implementation, the utilization is significantly low, which is as expected.

6.5.2 Runtime and Latency Comparison

In order to prove improvement in runtime and latency with PR_{BRAM} compared to PR_{DDR} approach, we ran an experiment with equal $\mathrm{RP}_{Bitsize}$ = 1306.272 KB for both implementations. Thus, this ensures that the time it takes to configure the partial bitstream is constant in both cases unlike results shown in Figure 6.20. The average improvement in runtime is 0.529363 s over varying range of reconfigurations. There is also improvement in latency especially when the number of reconfigurations is less. It can be observed that the runtime and latencies vary linearly in Figure 6.21 compared to 6.20, which is because the N_{bin} * T_{bin} in Equation 6.2 is constant in former case.



(a) (b)

Figure 6.19: Area vs Runtime Plots

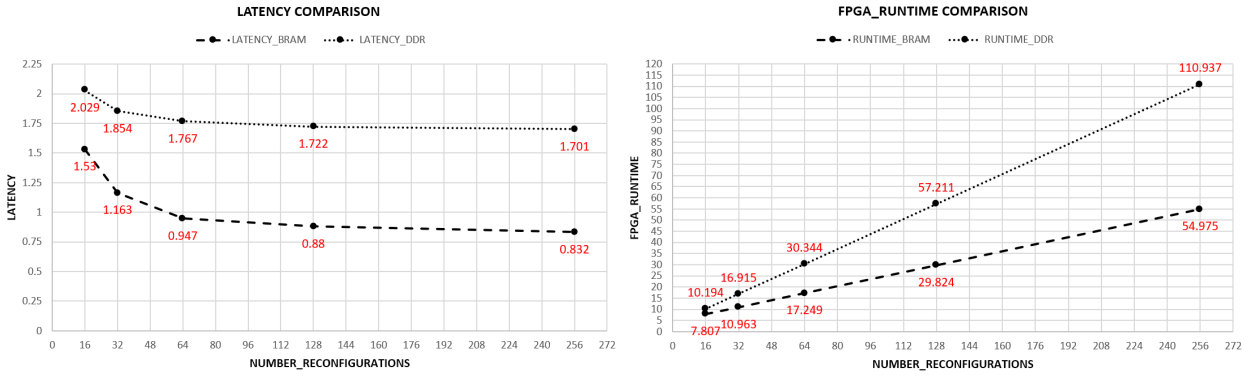
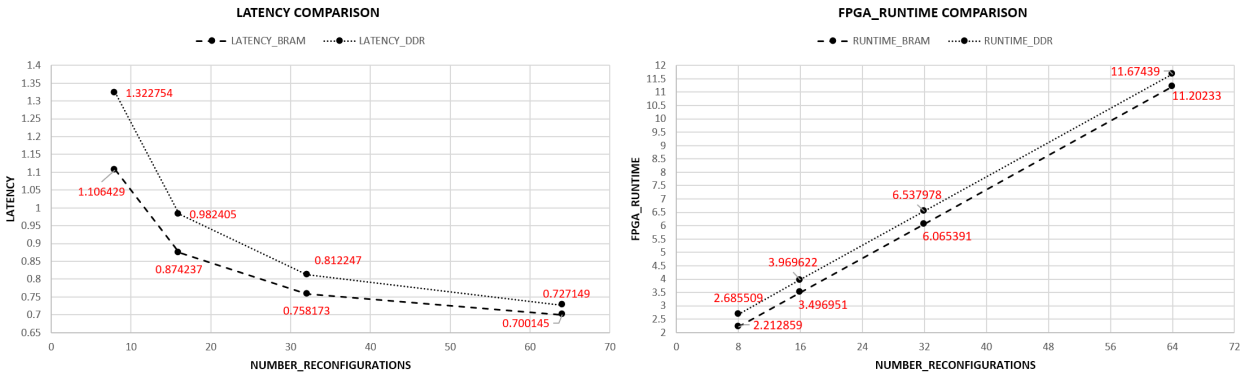


Figure 6.20: Runtime and Latency Plots with $\text{RP}_{Bitsize}$ = 3416.088 KB for PR_{DDR} and 1598.896 KB for PR_{BRAM}



(a) (b)

Figure 6.21: Runtime and Latency Plots with $\text{RP}_{Bitsize}$ = 1306.272 KB for both PR_{DDR} and PR_{BRAM}

6.6 Summary

In this chapter, we discussed in great detail about the proposed design methodology along with design implementation results with spatial & partial reconfiguration based architectures. We also examined the improvements achieved in runtime and latency with the proposed architecture.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

In this thesis we have proposed and implemented the novel design methodology using partial reconfiguration as described in Section 6.1 with the help of JPEG Encoder, a S2CBenchmark in v.2.0. We have also described TCL based automated floorplaning and user software application psuedo code as explained in Algorithms 2 & 3 respectively. We have also verified with three testcase images-lena,peppers and goldhill for all the spatial and partial reconfiguration based implementations discussed in Chapter 6.

We have seen that the implementation with the proposed Architecture PR_{BRAM} is area efficient compared to spatial implementation with LUT area savings upto 21.20 % and FF area savings upto 30.41 % for 1598.896 KB as partial bitstream size. These %'s are including the additional resources utilized by proposed static architecture. We also have seen an improvement in average hardware running of 0.529363s against PR_{DDR}.

7.2 Future work

• Developing sophisticated Partial Reconfiguration Controllers to minimize the time required for reconfiguring.

• Exploring enhanced parallelism in hardware accelerators utilizing saved area due to PR Implementation against spatial implementation of those hardware accelerators.

• In extremely data-intensive applications, exploring performance impact on proposed architecture when BRAM along with distributed RAM is utilized to cope with limitations of on-chip memory in FPGA fabric.

REFERENCES

[1] B. Darrow, "Why microsoft is putting these chips at the center of its cloud," http://fortune.com/2016/10/17/microsoft-fpga-chips-azure/, Tech. Rep., Oct. 2017.

[2] J. Morris, "Intel pushes fpgas into the data center," http://www.zdnet.com/article/intel-pushes-fpgas-into-the-data-center/, Tech. Rep., Oct. 2017.

[3] N. G. Nayak, "Accelerated computation using runtime partial reconfiguration," Master's thesis, University of Stuttgart, 2013.

[4] S. Casale-Brunet, E. Bezati, and M. Mattavelli, "Design space exploration of dataflow-based smith-waterman fpga implementations," in *2017 IEEE International Workshop on Signal Processing Systems (SiPS)*, Oct 2017, pp. 1–6.

[5] J. Piat and J. Crenne, "Modeling dynamic partial reconfiguration in the dataflow paradigm," in *Signal Processing Systems (SiPS), 2014 IEEE Workshop on.* IEEE, 2014, pp. 1–6.

[6] E. S. C. de Comer, V. A. M. Coronel, Y. L. K. B. Macayana, L. M. F. Mañalac, A. C. C. Torreno, L. P. Alarcón, M. T. G. D. Leon, C. V. J. Densing, M. D. Rosales, and R. J. M. Maestro, "A study on partial reconfiguration with compression via modularizing secondary processes of a general purpose processor," in *TENCON 2017 - 2017 IEEE Region 10 Conference*, Nov 2017, pp. 443–448.

[7] A. Kamaleldin, S. Hosny, K. Mohamed, M. Gamal, A. Hussien, E. Elnader, A. Shalash, A. M. Obeid, Y. Ismail, and H. Mostafa, "A reconfigurable hardware platform implementation for software defined radio using dynamic partial reconfiguration on xilinx zynq fpga," in *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug 2017, pp. 1540–1543.

[8] C. Kao, "Benefits of partial reconfiguration," *Xcell journal*, vol. 55, pp. 65–67, 2005.

[9] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 8–17, 2009.

[10] D. MacMillen, R. Camposano, D. Hill, and T. W. Williams, "An industrial view of electronic design automation," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 19, no. 12, pp. 1428–1448, 2000.

[11] A. Hemani, "Charting the eda roadmap," *IEEE Circuits and Devices Magazine*, vol. 20, no. 6, pp. 5–10, 2004.

[12] D. W. Knapp, *Behavioral synthesis: digital system design using the synopsys behavioral compiler.* Prentice Hall PTR, 1996.

[13] J. P. Elliott, *Understanding behavioral synthesis: a practical guide to high-level design.* Springer Science & Business Media, 2012.

[14] D. D. Gajski, N. Dutt, A. Wu, and S. Lin, "High level synthesis, introduction to chip and system design, chapter 1," 1992.

[15] *Vivado Design Suite User Guide High Level Synthesis - UG902*, v2017.3 ed., Xilinx, Oct. 2017.

[16] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A formal approach to the scheduling problem in high level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 4, pp. 464–475, 1991.

[17] P. Coussy and A. Morawiec, *High-level synthesis: from algorithm to digital circuit.* Springer Science & Business Media, 2008.

[18] D. C. Ku and G. De Mitcheli, "Relative scheduling under timing constraints: Algorithms for high-level synthesis of digital circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 6, pp. 696–718, 1992.

[19] U. Farooq, Z. Marrakchi, and H. Mehrez, "Fpga architectures: An overview," *Tree-based Heterogeneous FPGA Architectures*, pp. 7–48, 2012.

[20] Altera, "Architceture brief- what is an soc fpga?" Tech. Rep., 2014.

[21] "Zynq-7000 generation ahead backgrounder," Xilinx, Tech. Rep., 2014.

[22] Altera, "Soc fpga product overview advance information brief," Tech. Rep., Feb. 2012.

[23] *Zynq-7000 All Programmable SoC Technical Reference Manual - UG585*, Xilinx, Dec. 2017.

[24] *7 Series FPGAs Configurable Logic Block User Guide - UG474*, Xilinx, Sep. 2016.

[25] *Vivado Design Suite User Guide Partial Reconfiguration - UG909*, Xilinx, Apr. 2017.

[26] K. Paulsson, M. Hubner, G. Auer, M. Dreschmann, L. Chen, and J. Becker, "Implementation of a virtual internal configuration access port (jcap) for enabling partial self-reconfiguration on xilinx spartan iii fpgas," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on.* IEEE, 2007, pp. 351–356.

[27] A. Ebrahim, K. Benkrid, X. Iturbe, and C. Hong, "A novel high-performance fault-tolerant icap controller," in *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on.* IEEE, 2012, pp. 259–263.

[28] D. Koch, *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications.* Springer Science & Business Media, 2012, vol. 153.

[29] J. Heiner, B. Sellers, M. Wirthlin, and J. Kalb, "Fpga partial reconfiguration via configuration scrubbing," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on.* IEEE, 2009, pp. 99–104.

[30] H. Zhang, L. Bauer, M. A. Kochte, E. Schneider, C. Braun, M. E. Imhof, H.-J. Wunderlich, and J. Henkel, "Module diversification: Fault tolerance and aging mitigation for runtime reconfigurable architectures," in *Test Conference (ITC), 2013 IEEE International*. IEEE, 2013, pp. 1–10.

[31] H. M. Hussain, K. Benkrid, and H. Seker, "An adaptive implementation of a dynamically reconfigurable k-nearest neighbour classifier on fpga," in *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on*. IEEE, 2012, pp. 205–212.

[32] R. Cattaneo, X. Niu, C. Pilato, T. Becker, W. Luk, and M. D. Santambrogio, "A framework for effective exploitation of partial reconfiguration in dataflow computing," in *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on*. IEEE, 2013, pp. 1–8.

[33] O. Pell, O. Mencer, K. H. Tsoi, and W. Luk, "Maximum performance computing with dataflow engines," in *High-Performance Computing Using FPGAs*. Springer, 2013, pp. 747–774.

[34] J. Sérot and F. Berry, "High-level dataflow programming for reconfigurable computing," in *Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2014 International Symposium on*. IEEE, 2014, pp. 72–77.

[35] F. Plavec, "Stream computing on fpgas," Ph.D. dissertation, 2010.

[36] *Creating and Packaging Custom IP - UG1118*, Xilinx, 2015.

[37] *Designing IP Subsystems Using IP Integrator - UG994*, Xilinx, Apr. 2015.

BIOGRAPHICAL SKETCH

Mihir Shah was born in Mumbai, India on 10^{th} May, 1992. He finished his high school in 2008 from St. Joseph's High School, Mumbai and college in 2010 from G.N Khalsa, Mumbai. After that, he completed his undergraduate degree (B.Tech) in Electronics and Computer Engineering with distinction from SRM Institute of Science and Technology, Chennai in 2014. He worked as a Research Associate at IIIT Hyderabad after completing his bachelors where he published his research work at IROS 2015. He joined The University of Texas at Dallas for his Master of Science in Electrical Engineering in August 2015. He has been working with DARClab (Design Automation and Reconfigurable Computing lab) since December 2016.

CURRICULUM VITAE

Mihir Shah

January 12^{th}, 2018

Contact Information:

Department of Electrical Engineering
The University of Texas at Dallas
800 W. Campbell Rd.
Richardson, TX 75080-3021, U.S.A.

Voice: (972) 408-6307
Email: mihir.shah2@utdallas.edu

Educational History:

M.S.E.E, The University of Texas at Dallas, 2018
MSEE Thesis: *Flexible Partial Reconfiguration based Design Architecture for Dataflow Computing*
Thesis Advisor: Dr. Benjamin Carrion-Schaefer

B.Tech., Electronics & Computer Engineering, SRM Institute of Science and Technology, India, 2014

Work Experience:

FPGA Design & Validation Intern, *Signal Laboratories, Inc*-Menlo Park, CA (May'17-Aug'17)
Hardware Design Intern, *DEKA Research & Development Corp*-Boston, MA (May'16 - Dec'16)
Research Associate, *IIIT Hyderabad*-India (June'14- July'15)