Flexible Partial Reconfiguration based Design Architecture for Dataflow Computation

Mihir Shah

Advisor: Benjamin Carrion Schaefer

DARClab

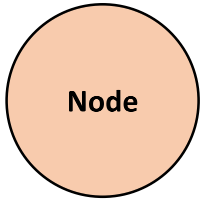
Design Automation and Reconfigurable Computing

Thesis Index

SR. NO	TITLE
1	Thesis Motivation
2	Thesis Contribution
5	Proposed Design Methodology
6	Design Implementations – Spatial & Partial Reconfiguration based
7	Comparative Study & Analysis
8	Conclusion & Future Works

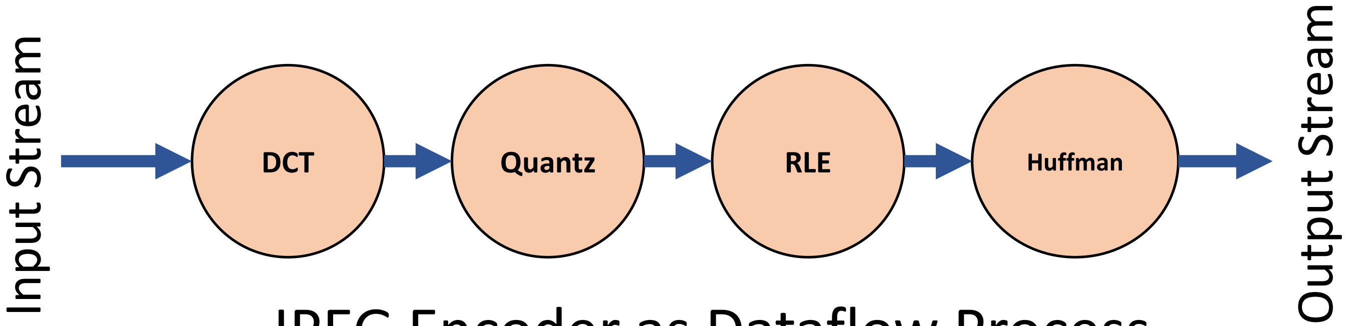
Thesis Motivation

- **Dataflow Computing (DC) specific model of computation**
  - Target application described as **Data Flow Graph (DFG)**
  - Used Extensively in High Frequency Trading, Image, Signal processing based applications



Node: Processing Element (PE), Execution kernel/accelerator/

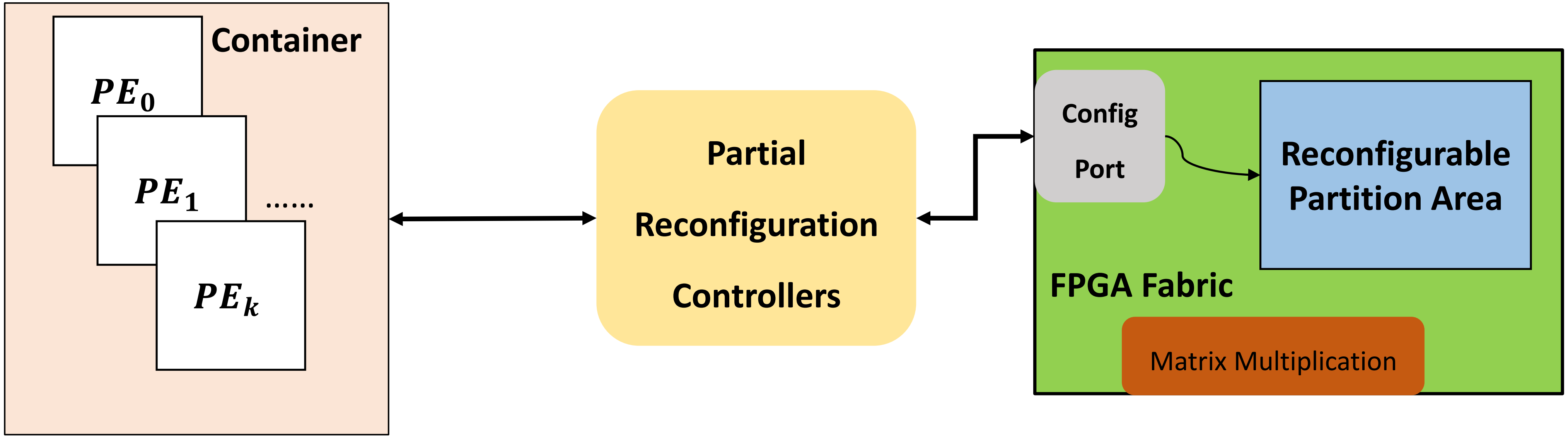
Links : FIFO (first-in first-out) queue or buffer



JPEG Encoder as Dataflow Process

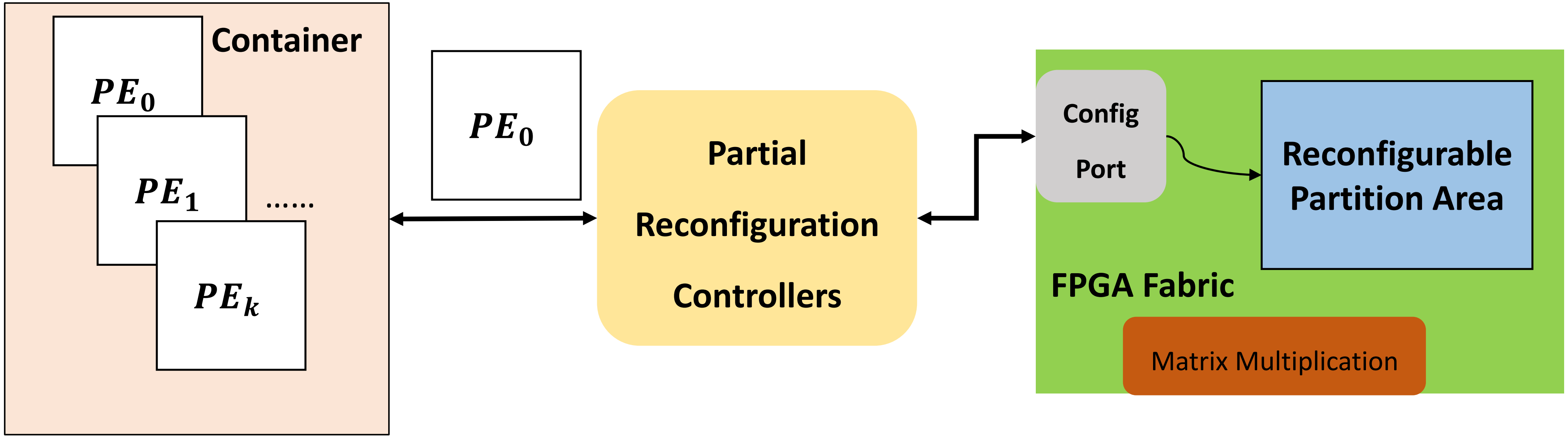
Thesis Motivation

- **Partial Reconfiguration**
  - **Allows modification of an operating FPGA** design by loading a partial configuration file, usually a partial BIT file
  - Time Multiplex several Processing Elements in a dataflow computation process



Thesis Motivation

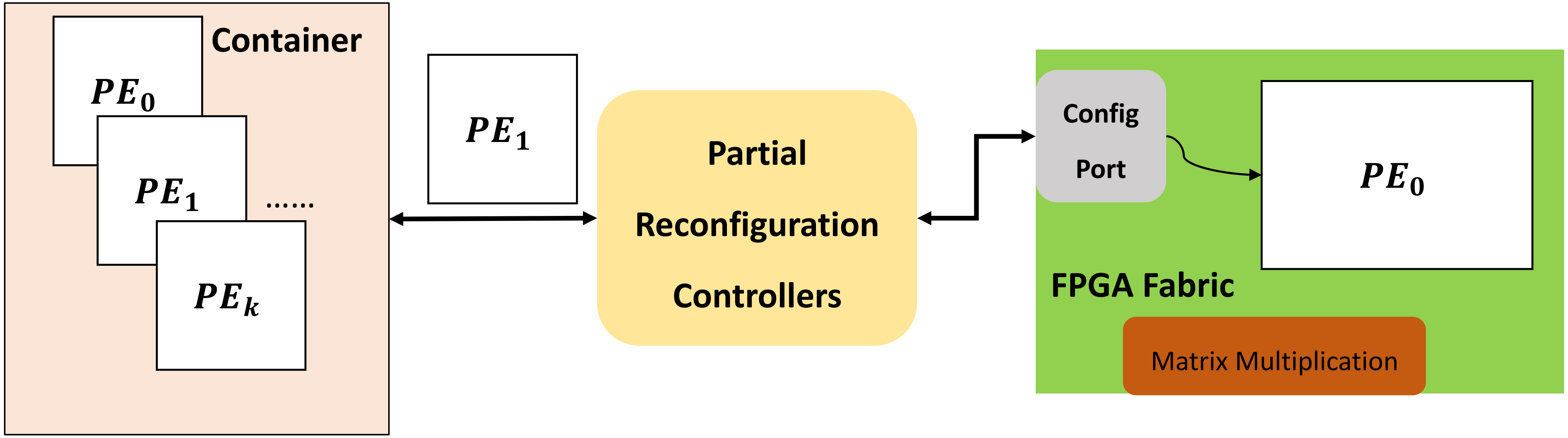
- **Partial Reconfiguration**
  - **Allows modification of an operating FPGA** design by loading a partial configuration file, usually a partial BIT file
  - Time Multiplex several Processing Elements in a dataflow computation process



Thesis Motivation

➢ Partial Reconfiguration

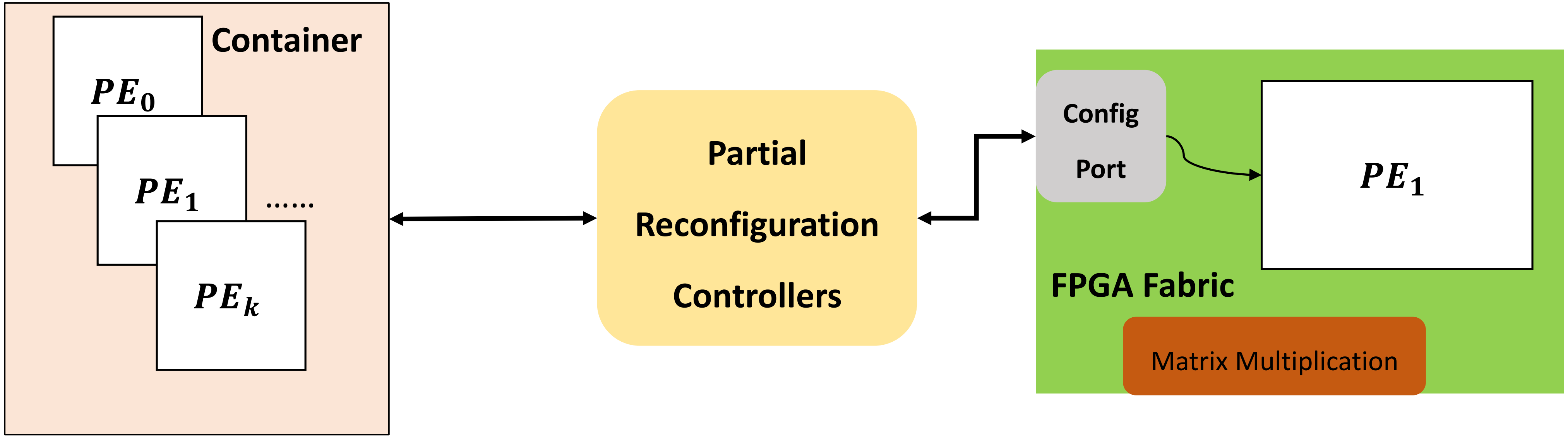
- **Allows modification of an operating FPGA** design by loading a partial configuration file, usually a partial BIT file
- Time Multiplex several Processing Elements in a dataflow computation process



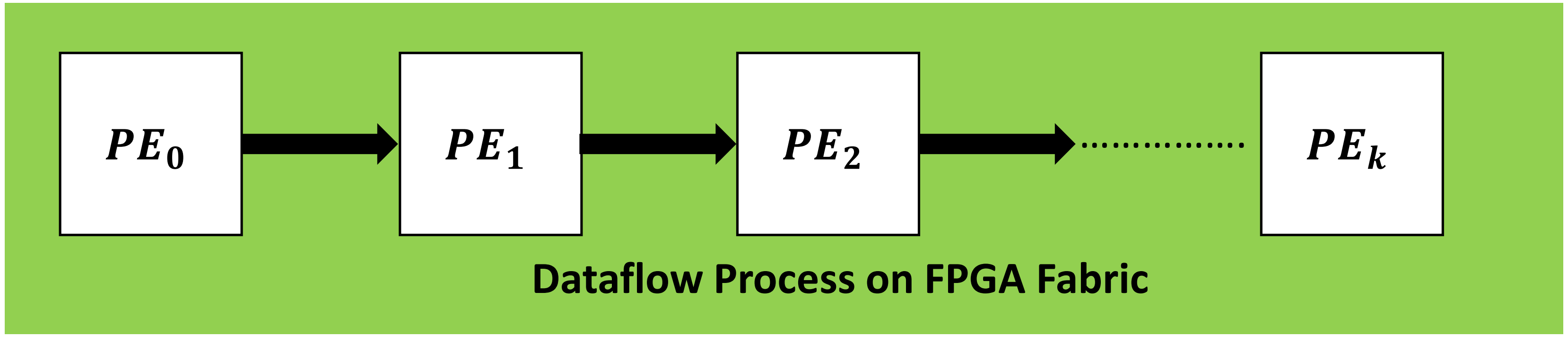
Thesis Motivation

Partial Reconfiguration

- **Allows modification of an operating FPGA** design by loading a partial configuration file, usually a partial BIT file
- Time Multiplex several Processing Elements in a dataflow computation process



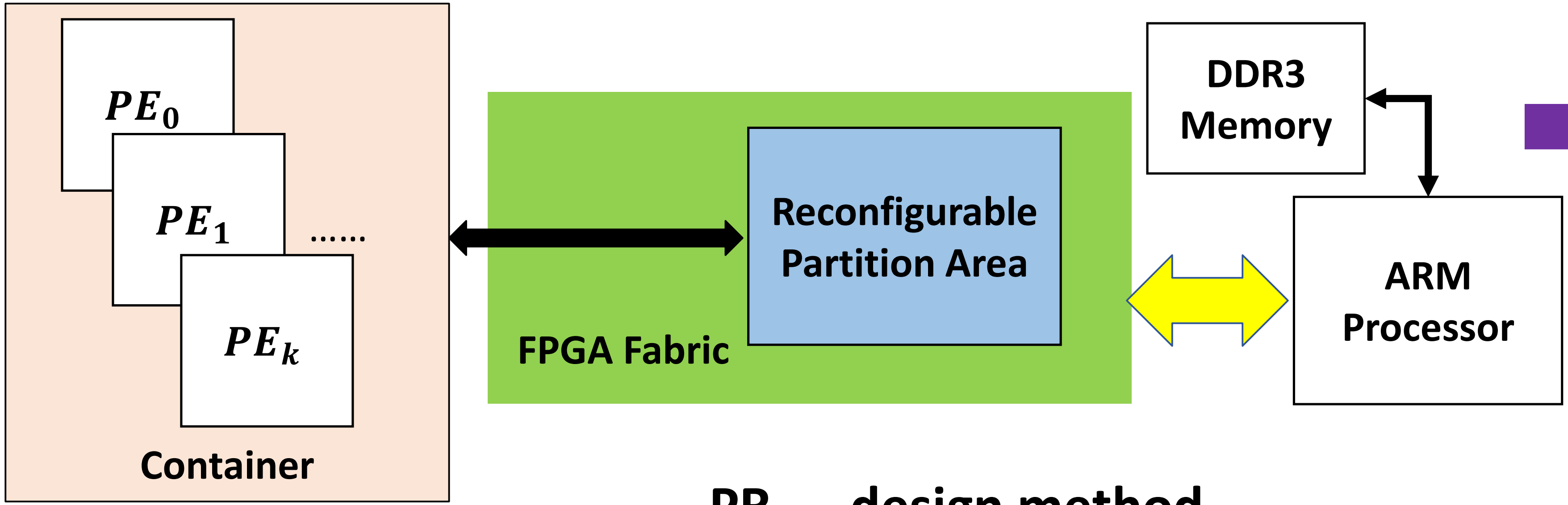
Thesis Motivation



Spatially placed FPGA Design

Problem ? Area Utilization High

Static design method



PR based design using External Off chip DDR memory as FIFO

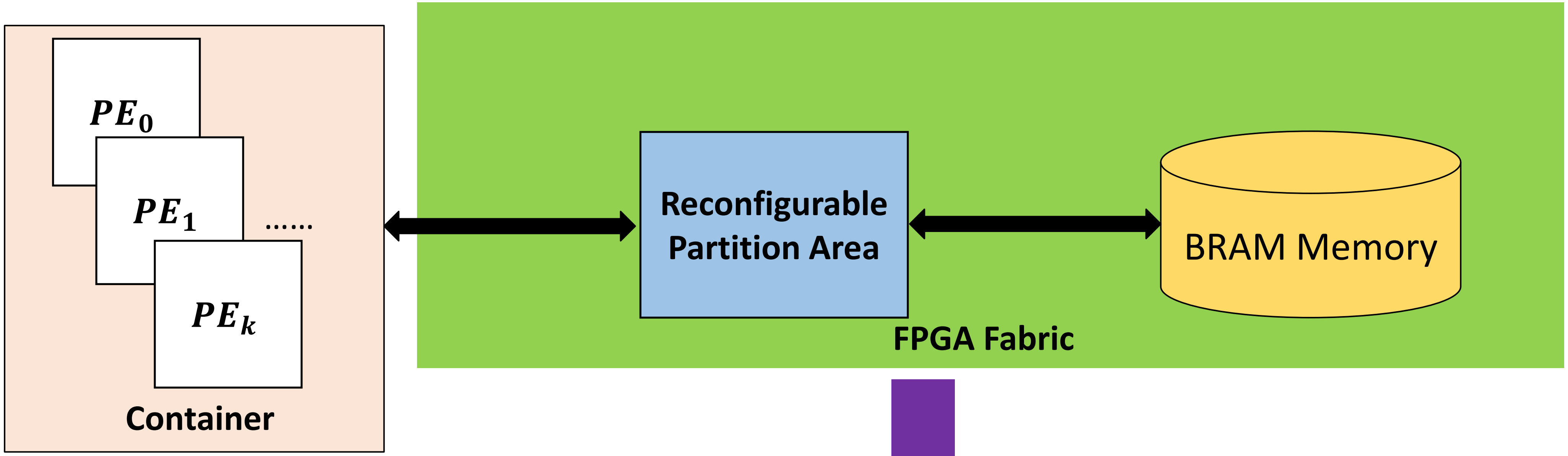
Solves: Area Utilization Issue

Problem ? Runtime & Latency Hit !!

PR_{DDR} design method

Thesis Motivation

PR_{BRAM} design method



THUS WE PROPOSE - PR based design using Internal On chip BRAM memory as FIFO for dataflow process

Improves Runtime & Latency compared to PR_{DDR} with reduced FPGA resource savings

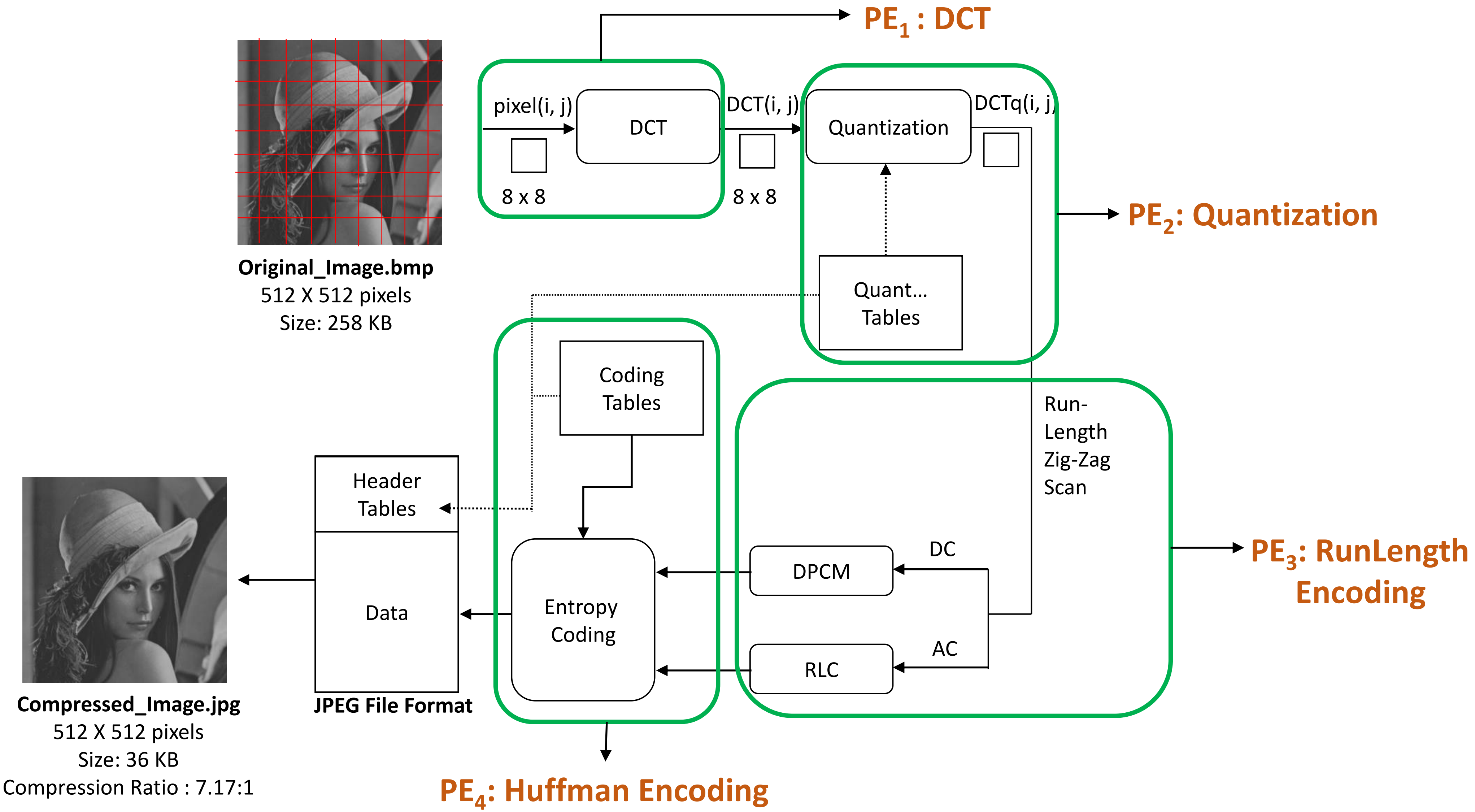
Thesis Contribution

1. **Semi-automatic design methodology for dataflow computation**
   - Fixed overlay Static architecture - PR_{BRAM}
   - Support for partial re-configurability
   - Input is behavior description language for HLS

2. **Prototyped on Xilinx Zynq FPGA**
   - JPEG Encoder given in SystemC
   - Three Testcase images

3. **Extensive experimental results**
   - Measure hardware running time vs. area characteristics of static and PR-based methods.
   - Comparative study between PR_{BRAM} and PR_{DDR} methods
   - Hardware Running Time vs. Varying Size of Pblock or Reconfigurable Partitions

JPEG Encoder: A Dataflow Process for Comparative Study

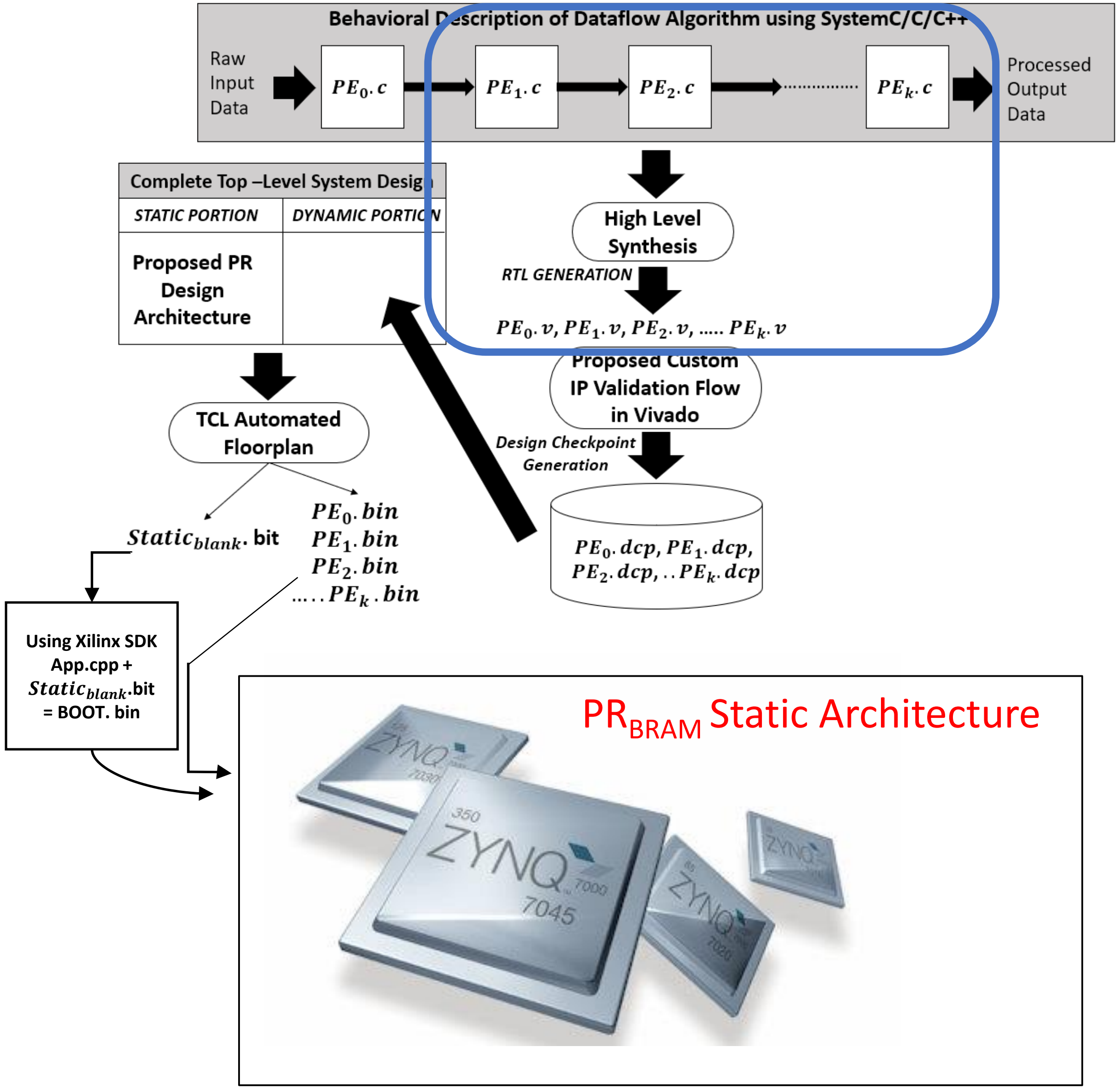


Original_Image.bmp
512 X 512 pixels
Size: 258 KB

Compressed_Image.jpg
512 X 512 pixels
Size: 36 KB
Compression Ratio : 7.17:1

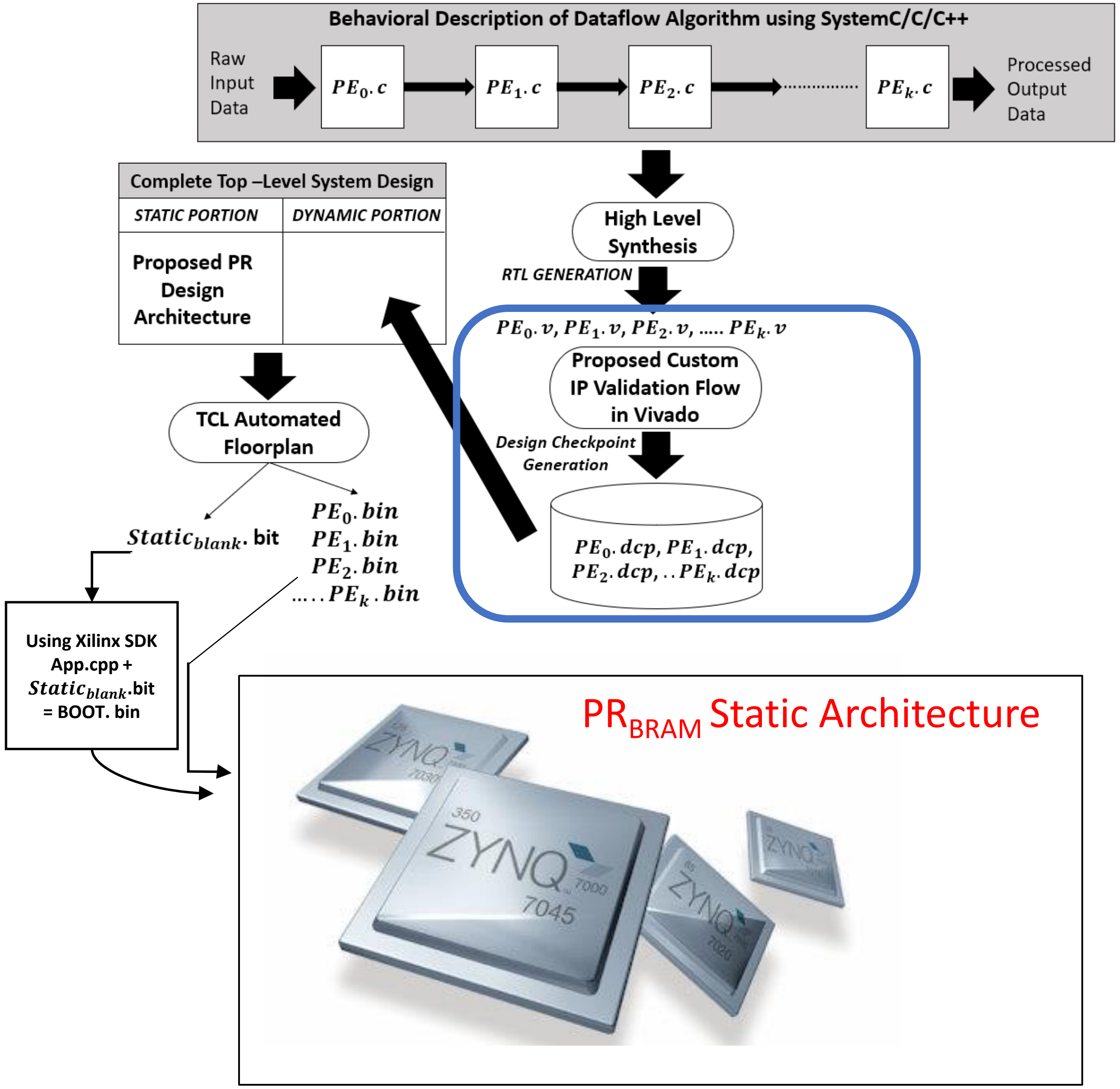
JPEG File Format

Overview of the Proposed Design Methodology


Behavioral Description of Dataflow Algorithm using SystemC/C/C++
Raw Input Data
$PE_0.c$
$PE_1.c$
$PE_2.c$
$PE_k.c$
Processed Output Data
High Level Synthesis
RTL GENERATION
$PE_0.v, PE_1.v, PE_2.v, ... PE_k.v$
Proposed Custom IP Validation Flow in Vivado
Design Checkpoint Generation
$PE_0.dcp, PE_1.dcp, PE_2.dcp, .. PE_k.dcp$
Complete Top –Level System Design
STATIC PORTION
DYNAMIC PORTION
Proposed PR Design Architecture
TCL Automated Floorplan
$Static_{blank}$. bit
$PE_0.bin$
$PE_1.bin$
$PE_2.bin$
$..... PE_k.bin$
Using Xilinx SDK App.cpp + $Static_{blank}$.bit = BOOT. bin
PR_{BRAM} Static Architecture
ZYNQ 7030
350 ZYNQ 7000 7045
ZYNQ 7020
ZYNQ

Stage 1: Behavioral Algorithm Description to RTL Generation

Overview of the Proposed Design Methodology



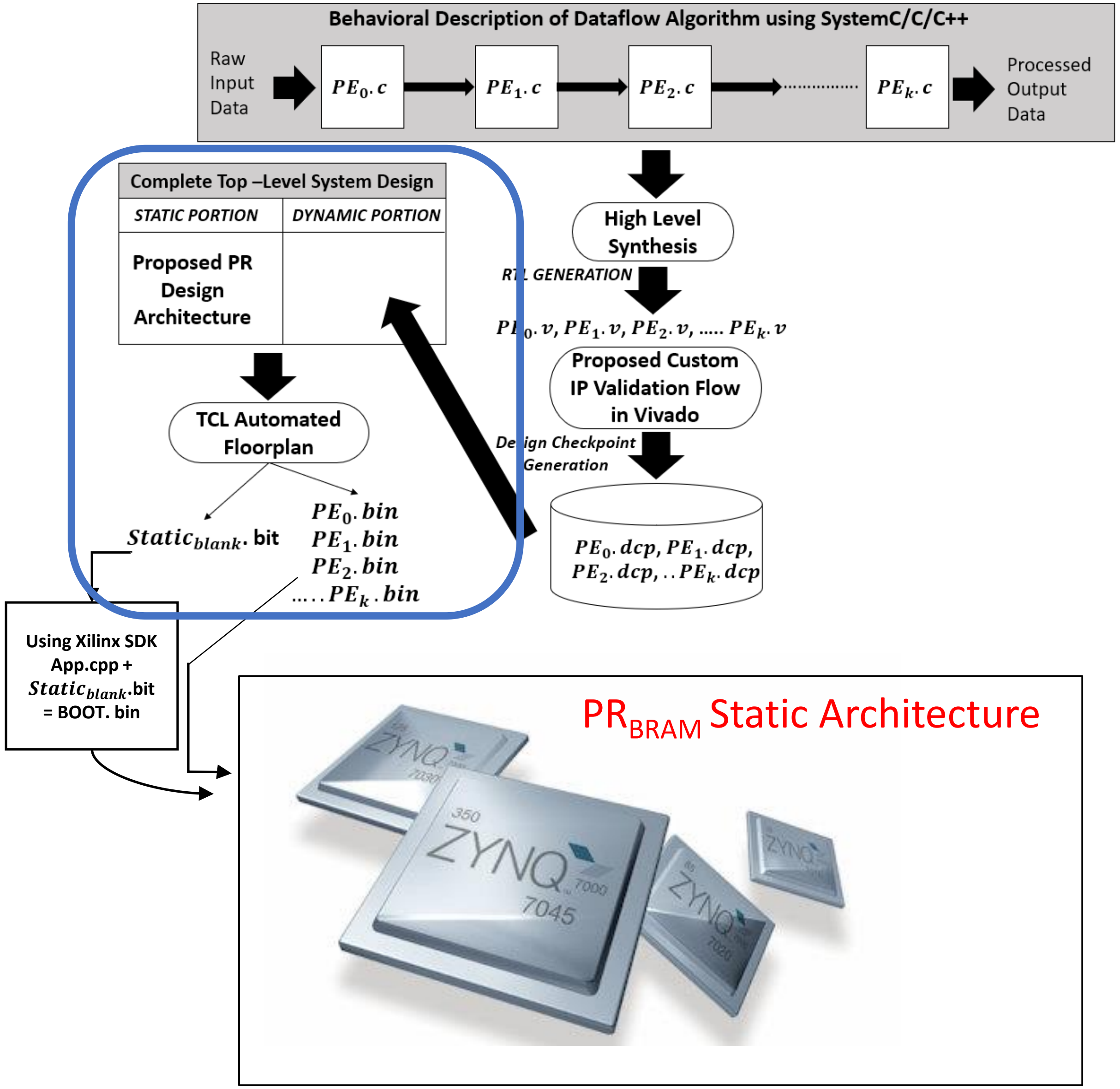
Behavioral Description of Dataflow Algorithm using SystemC/C/C++
Raw Input Data
$PE_0.c$
$PE_1.c$
$PE_2.c$
$PE_k.c$
Processed Output Data
High Level Synthesis
RTL GENERATION
$PE_0.v, PE_1.v, PE_2.v, ... PE_k.v$
Proposed Custom IP Validation Flow in Vivado
Design Checkpoint Generation
$PE_0.dcp, PE_1.dcp, PE_2.dcp, .. PE_k.dcp$
Complete Top –Level System Design
STATIC PORTION
DYNAMIC PORTION
Proposed PR Design Architecture
TCL Automated Floorplan
$Static_{blank}$.bit
$PE_0.bin$
$PE_1.bin$
$PE_2.bin$
$.....PE_k.bin$
Using Xilinx SDK App.cpp + $Static_{blank}$.bit = BOOT. bin
PR_{BRAM} Static Architecture
ZYNQ 7000 7045

Stage 1: Behavioral Algorithm Description to RTL Generation

⇩

Stage 2: Validation and Creation of Custom IPs

Overview of the Proposed Design Methodology



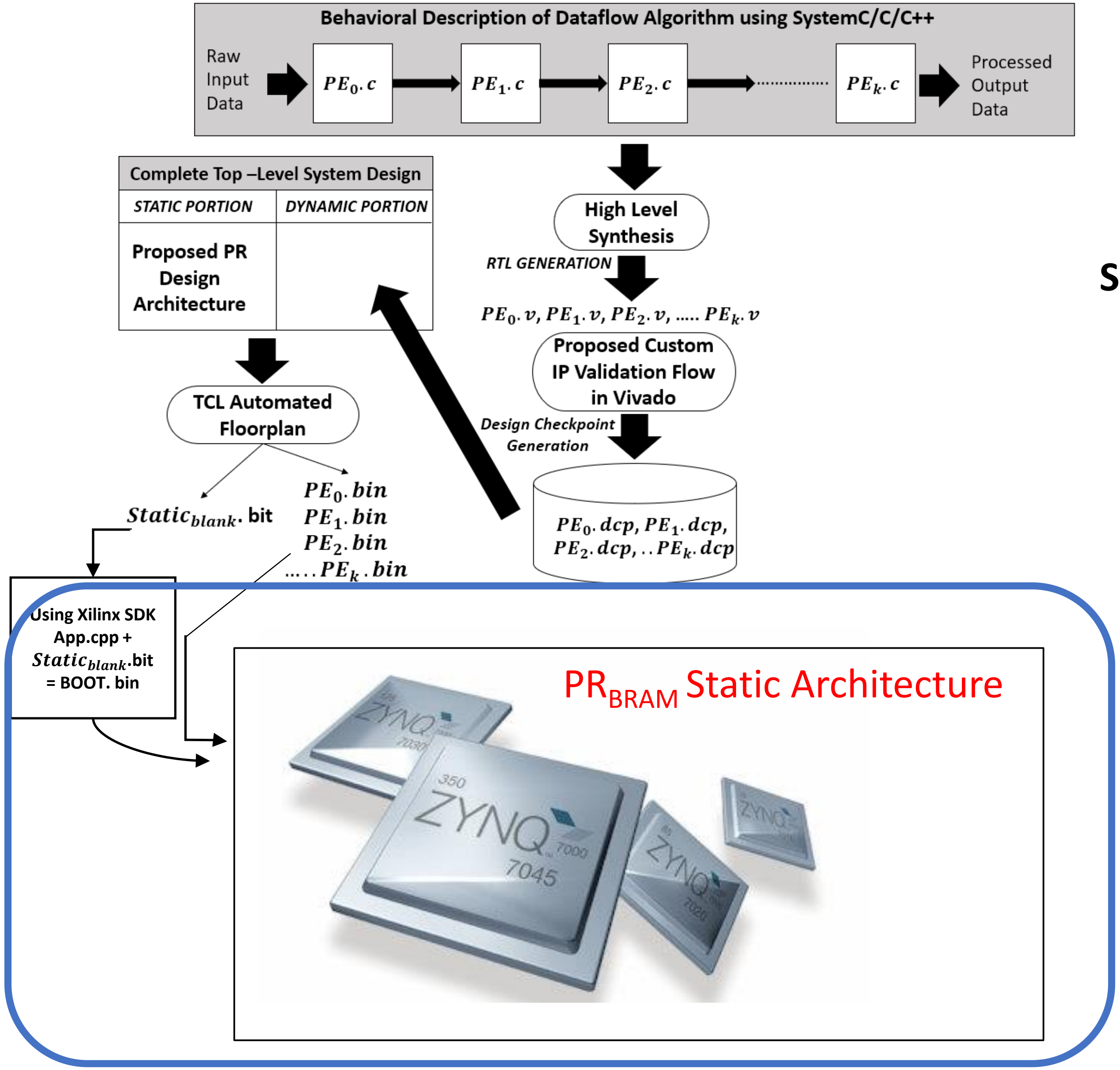
Behavioral Description of Dataflow Algorithm using SystemC/C/C++
Raw Input Data
$PE_0.c$
$PE_1.c$
$PE_2.c$
$PE_k.c$
Processed Output Data
High Level Synthesis
RTL GENERATION
$PE_0.v, PE_1.v, PE_2.v, ... PE_k.v$
Proposed Custom IP Validation Flow in Vivado
Design Checkpoint Generation
$PE_0.dcp, PE_1.dcp, PE_2.dcp, .. PE_k.dcp$
Complete Top –Level System Design
STATIC PORTION
DYNAMIC PORTION
Proposed PR Design Architecture
TCL Automated Floorplan
$Static_{blank}.$bit
$PE_0.bin$
$PE_1.bin$
$PE_2.bin$
$..... PE_k.bin$
Using Xilinx SDK App.cpp + $Static_{blank}$.bit = BOOT. bin
PR_{BRAM} Static Architecture

Stage 1: Behavioral Algorithm Description to RTL Generation

Stage 2: Validation and Creation of Custom IPs

Stage 3: TCL Automated Floorplan for PR Designs

Overview of the Proposed Design Methodology



Behavioral Description of Dataflow Algorithm using SystemC/C/C++

Raw Input Data → $PE_0.c$ → $PE_1.c$ → $PE_2.c$ → …………… $PE_k.c$ → Processed Output Data

High Level Synthesis

RTL GENERATION

$PE_0.v, PE_1.v, PE_2.v, \ldots\ldots PE_k.v$

Proposed Custom IP Validation Flow in Vivado

Design Checkpoint Generation

$PE_0.dcp, PE_1.dcp, PE_2.dcp, \ldots PE_k.dcp$

Complete Top –Level System Design

STATIC PORTION | *DYNAMIC PORTION*

Proposed PR Design Architecture

TCL Automated Floorplan

$Static_{blank}$. bit

$PE_0.bin$, $PE_1.bin$, $PE_2.bin$, $\ldots.PE_k.bin$

Using Xilinx SDK App.cpp + $Static_{blank}$.bit = BOOT. bin

PR_{BRAM} Static Architecture

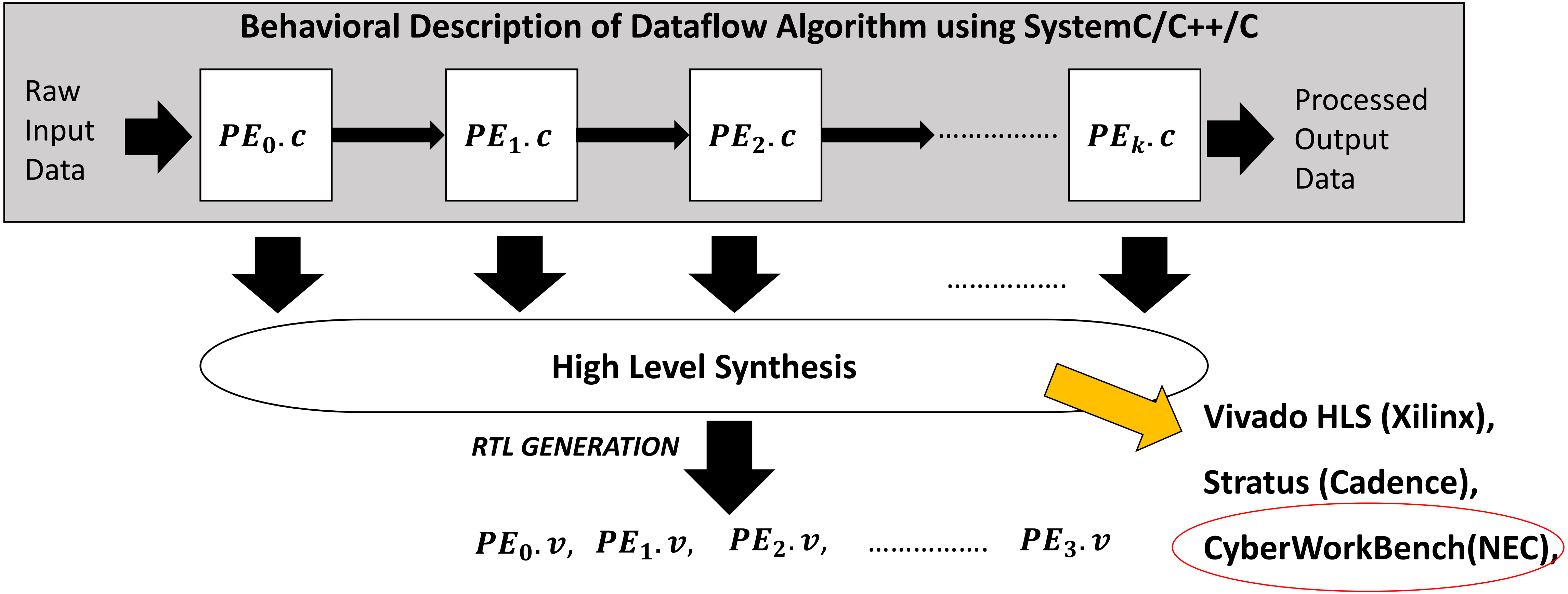
Stage 1: Behavioral Algorithm Description to RTL Generation

Stage 2: Validation and Creation of Custom IPs

Stage 3: TCL Automated Floorplan for PR Designs

Stage 4: Deploying the Binaries on Zynq-7000: PR_{BRAM} Static Architecture

Stage 1: Behavioral Description of Dataflow to RTL Generation



Behavioral Description of Dataflow Algorithm using SystemC/C++/C

Raw Input Data

$PE_0.c$ $PE_1.c$ $PE_2.c$ ……………. $PE_k.c$

Processed Output Data

…………….

High Level Synthesis

RTL GENERATION

$PE_0.v$, $PE_1.v$, $PE_2.v$, ……………. $PE_3.v$

Vivado HLS (Xilinx),

Stratus (Cadence),

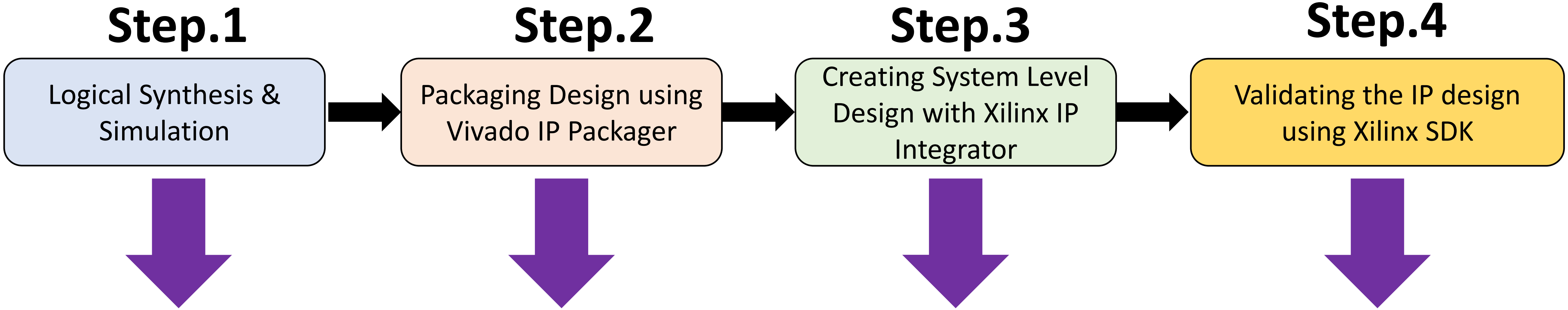
CyberWorkBench(NEC),

Catapult(Mentor)

➢ <u>Key-points when describing dataflow application using BDL:</u>

- Uniformity in the number, direction and data-widths of I/O all the PE’s
- Control interface signals - done, reset and start : Close Loop Feedback when Context Switching

Stage 2: Validation and Creation of Custom IPs



Step.1

Logical Synthesis & Simulation

- Structural RTL to Xilinx Primitives
- Writing Test Benches
- Ensure Target Platform Function & Timing Violations

Step.2

Packaging Design using Vivado IP Packager

- Design Re-usability
- Xilinx Supports AMBA AXI
- I/O instance port-map to internal slave registers

Step.3

Creating System Level Design with Xilinx IP Integrator

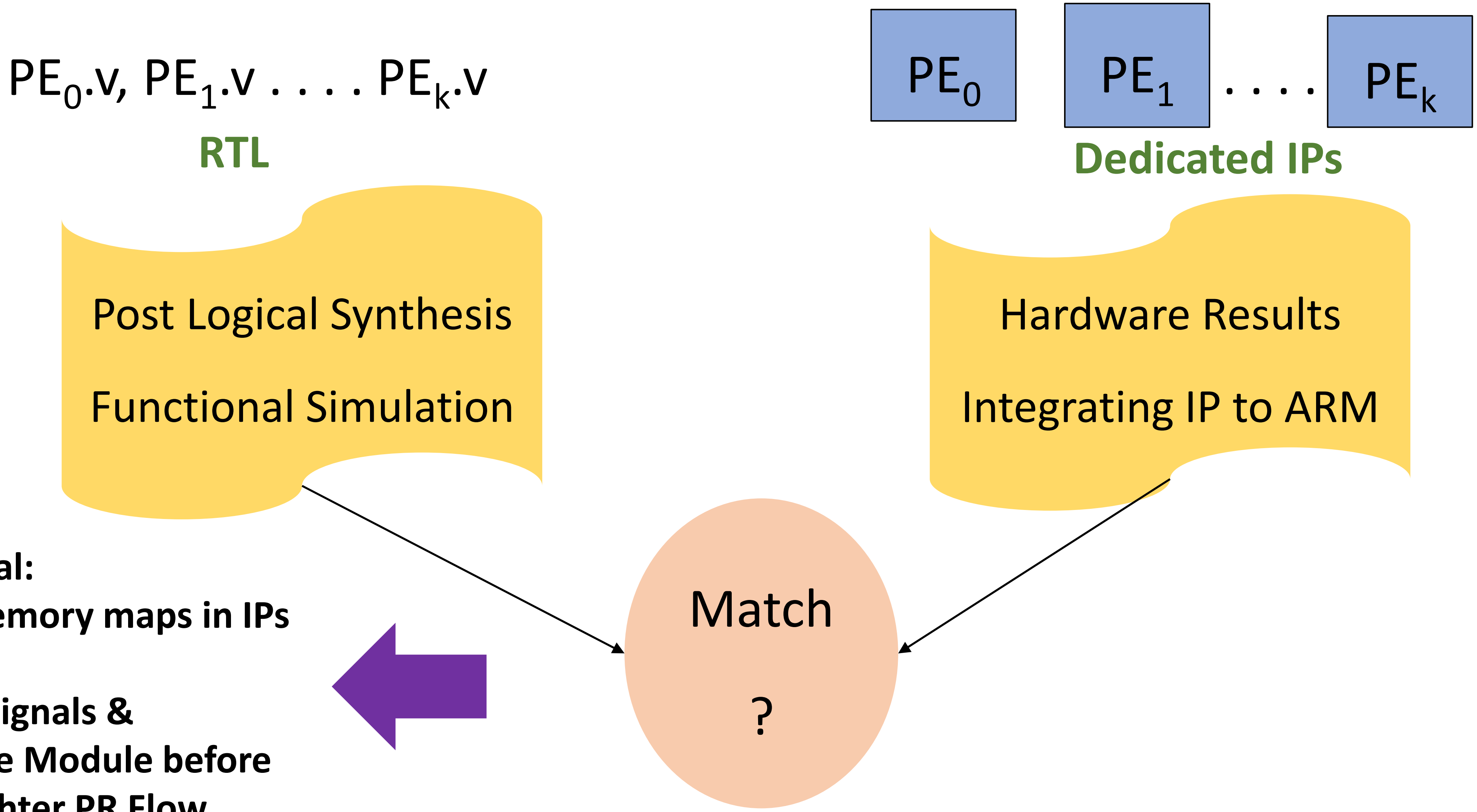
- Block Design with ZYNQ7 PS
- Create Top-Level Wrapper
- Export to SDK to create BSP & drivers

Step.4

Validating the IP design using Xilinx SDK

- Helps to Write Software Code for the IP
- Compare results with Simulation

Stage 2: Validation and Creation of Custom IPs



$PE_0.v, PE_1.v \dots PE_k.v$

RTL

Post Logical Synthesis

Functional Simulation

Dedicated IPs

Hardware Results

Integrating IP to ARM

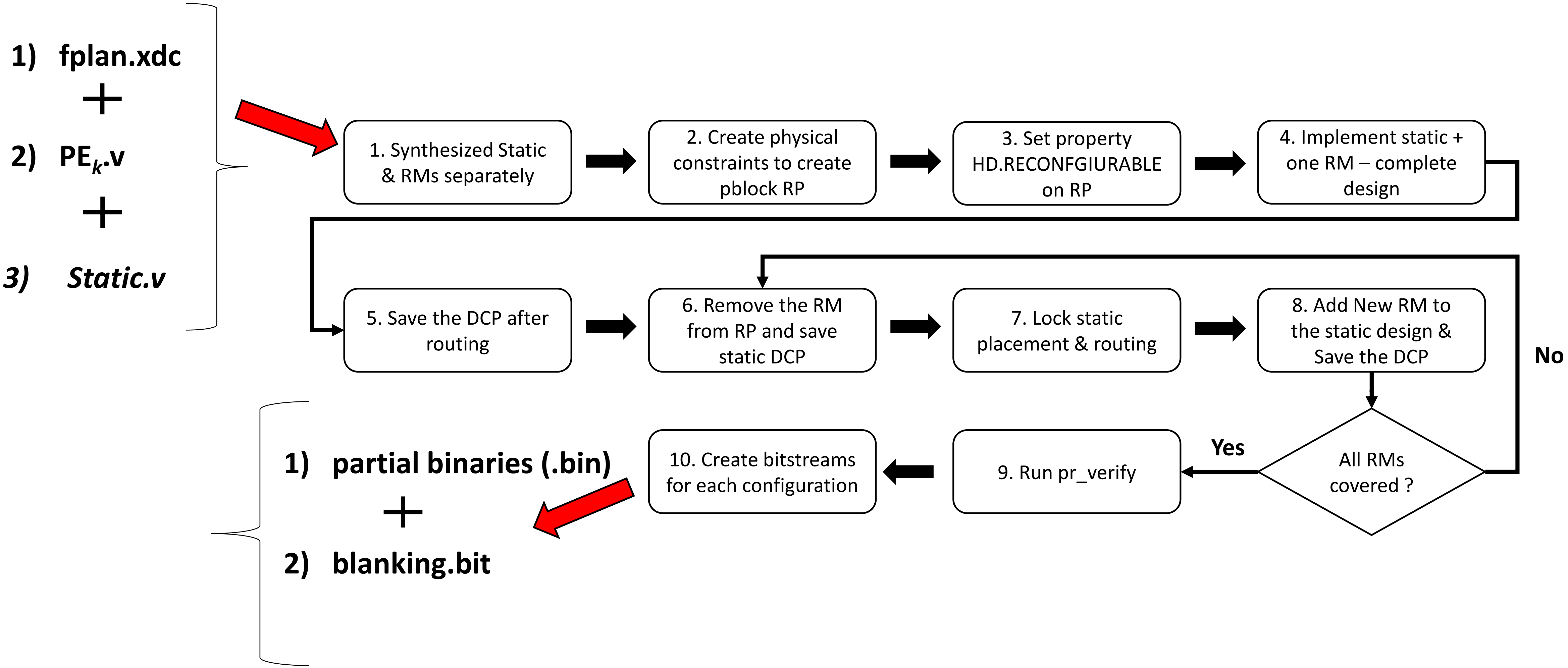
Match

?

Goal:

- **Ensures correct memory maps in IPs**
- **Validates Control Signals & Functionality of the Module before integrating into tighter PR Flow**

Stage 3: TCL Automated Floorplan for PR Designs



1) **fplan.xdc**

+

2) **PE$_k$.v**

+

3) ***Static.v***

1. Synthesized Static & RMs separately
2. Create physical constraints to create pblock RP
3. Set property HD.RECONFGIURABLE on RP
4. Implement static + one RM – complete design
5. Save the DCP after routing
6. Remove the RM from RP and save static DCP
7. Lock static placement & routing
8. Add New RM to the static design & Save the DCP

All RMs covered ?

No

Yes

9. Run pr_verify
10. Create bitstreams for each configuration

1) **partial binaries (.bin)**

+

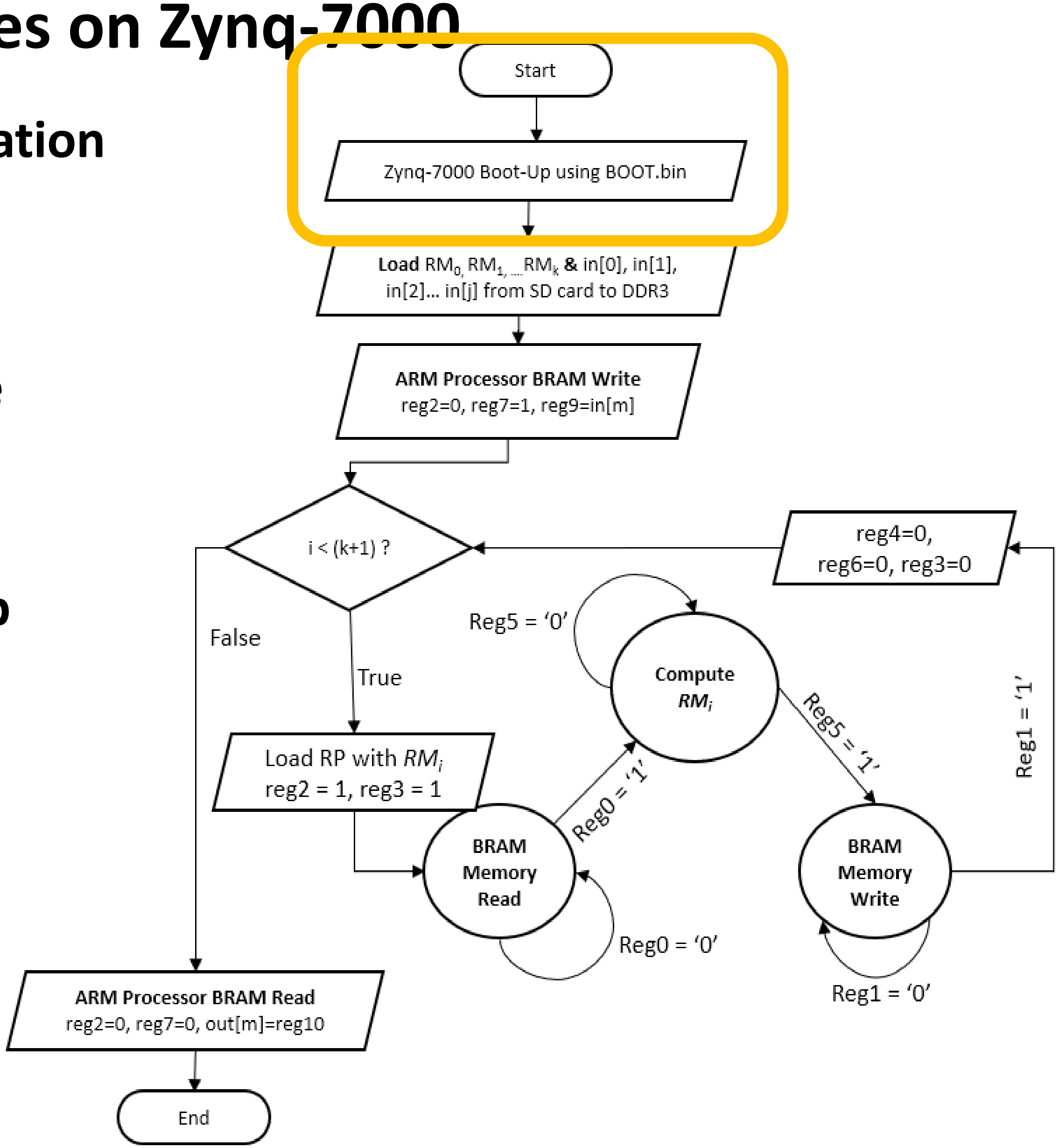
2) **blanking.bit**

****Processing Element (PE) will be referred as, Reconfigurable Module(RM) in PR Designs**

Stage 4: Deploying the Binaries on Zynq-7000

- **BOOT.bin = first stage image for PL side + User application software** ⟶ **SD Card**

  - After power-on reset, the **Boot ROM determines the boot mode (SD flash memory)** and the encryption status (non-secure)
    - **Load First Stage Boot Loader (FSBL) into on-chip RAM (OCM).**
    - **Releases CPU control to the FSBL which in turn configures the PL** with the full Static$_{blank}$.bit via **PCAP (Processor Control Access Port)**

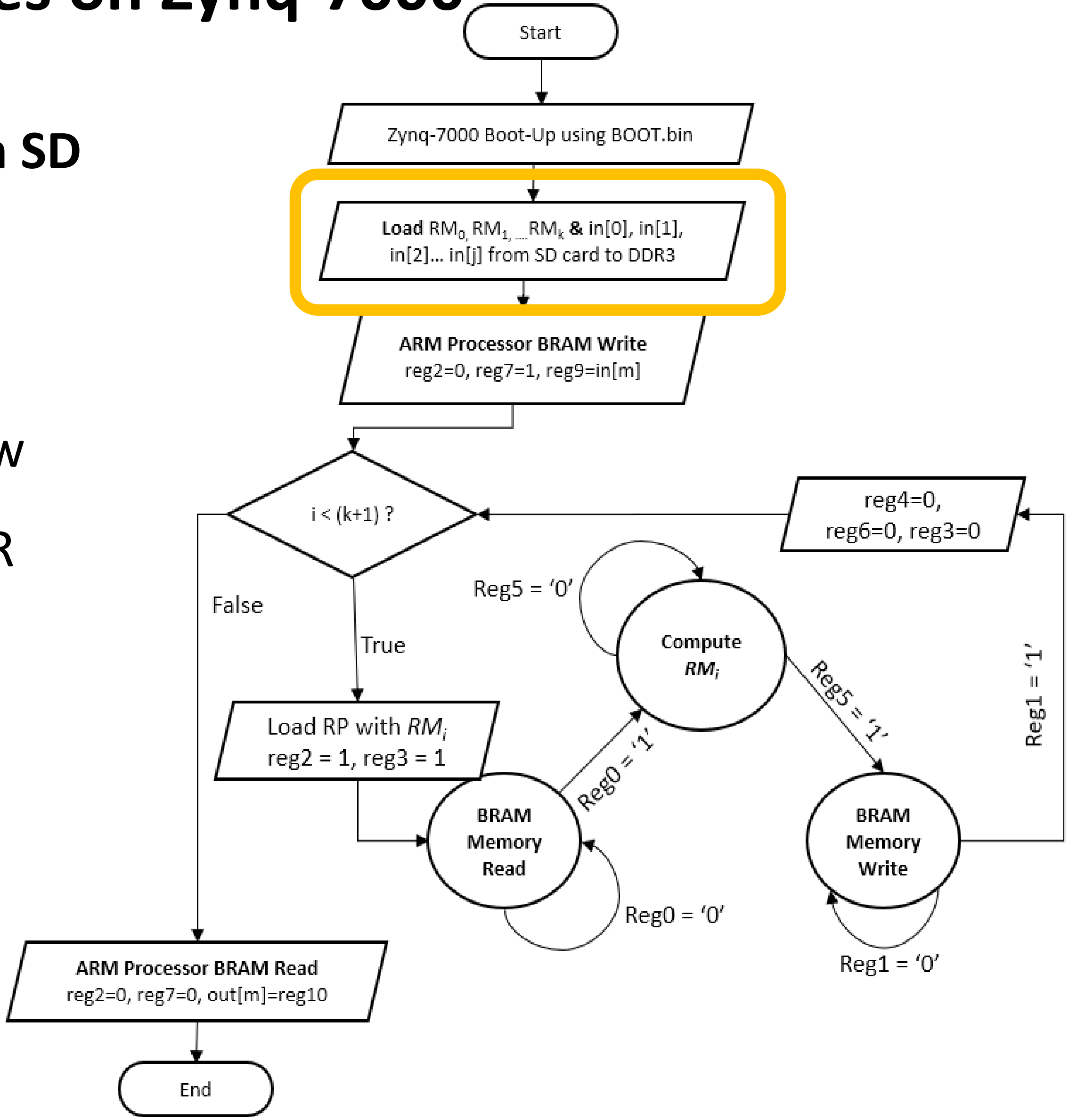


Stage 4: Deploying the Binaries on Zynq-7000

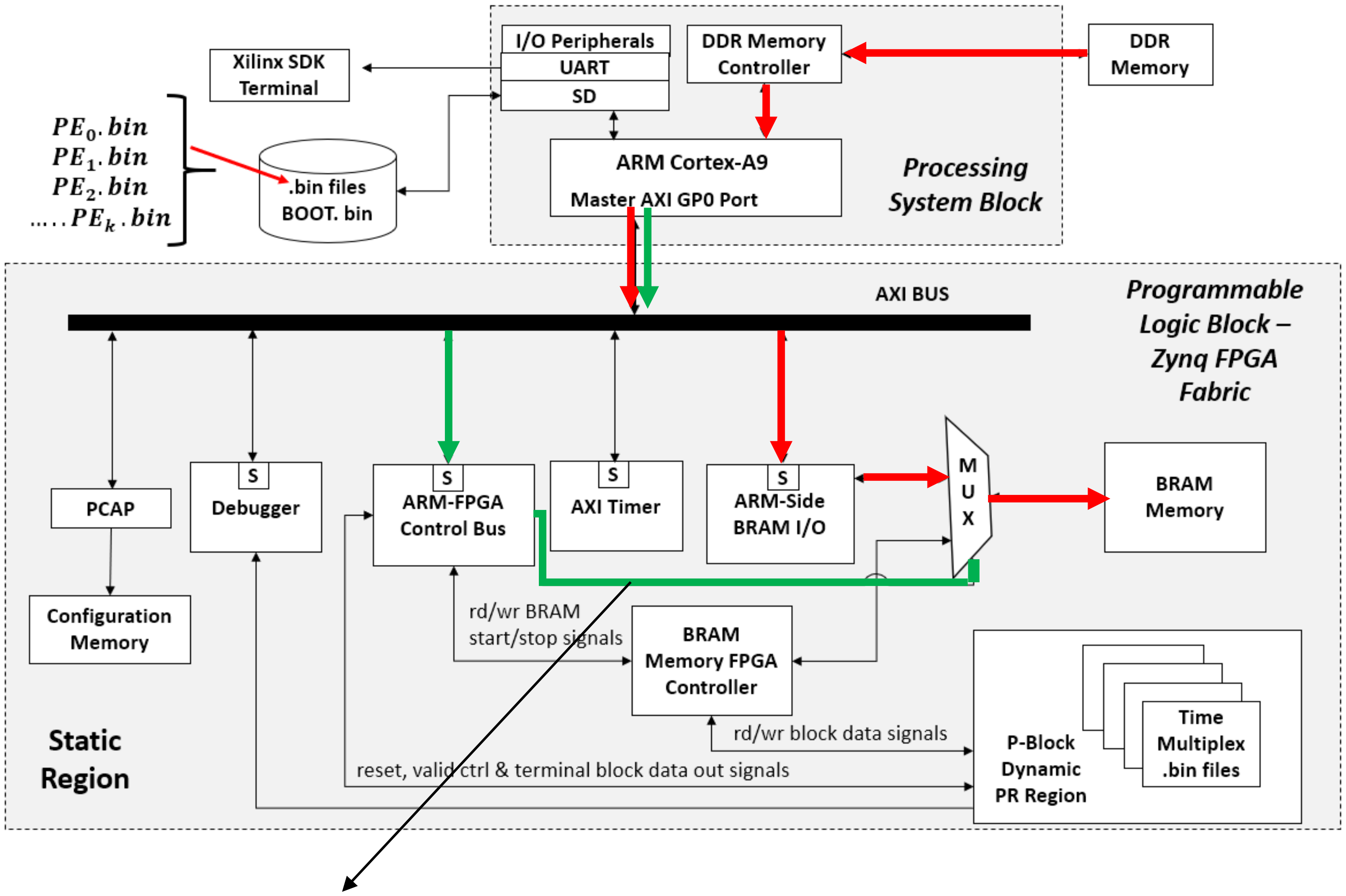
- Partial **bitstreams are loaded into DDR memory from SD card** to **maximize throughput** during configuration

- The Raw Image data which is the input to the dataflow process in JPEG Encoder is also transferred to the DDR Memory

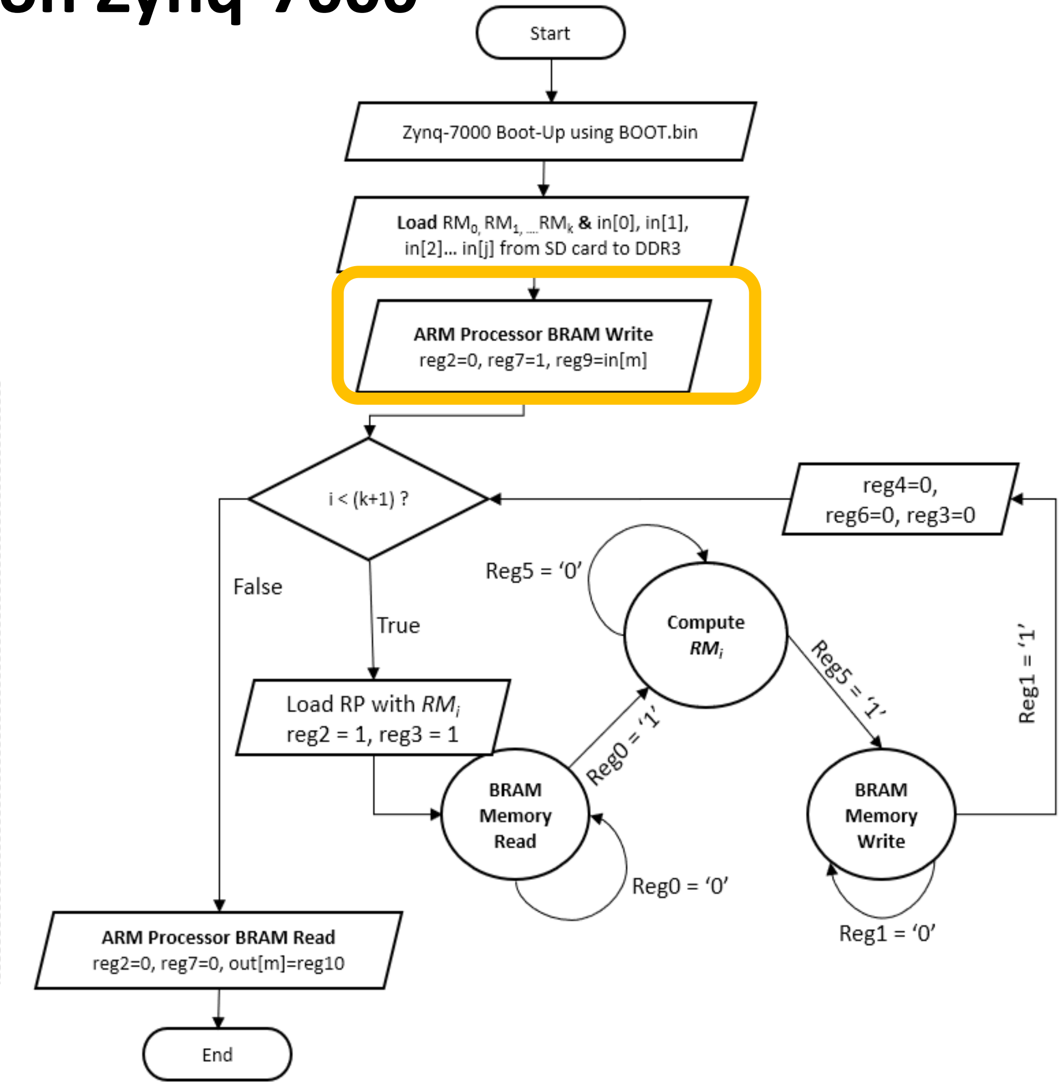
- After this step, the Reconfigurable Modules can be loaded into the Reconfigurable Partition to start computation.



Stage 4: Deploying the Binaries on Zynq-7000

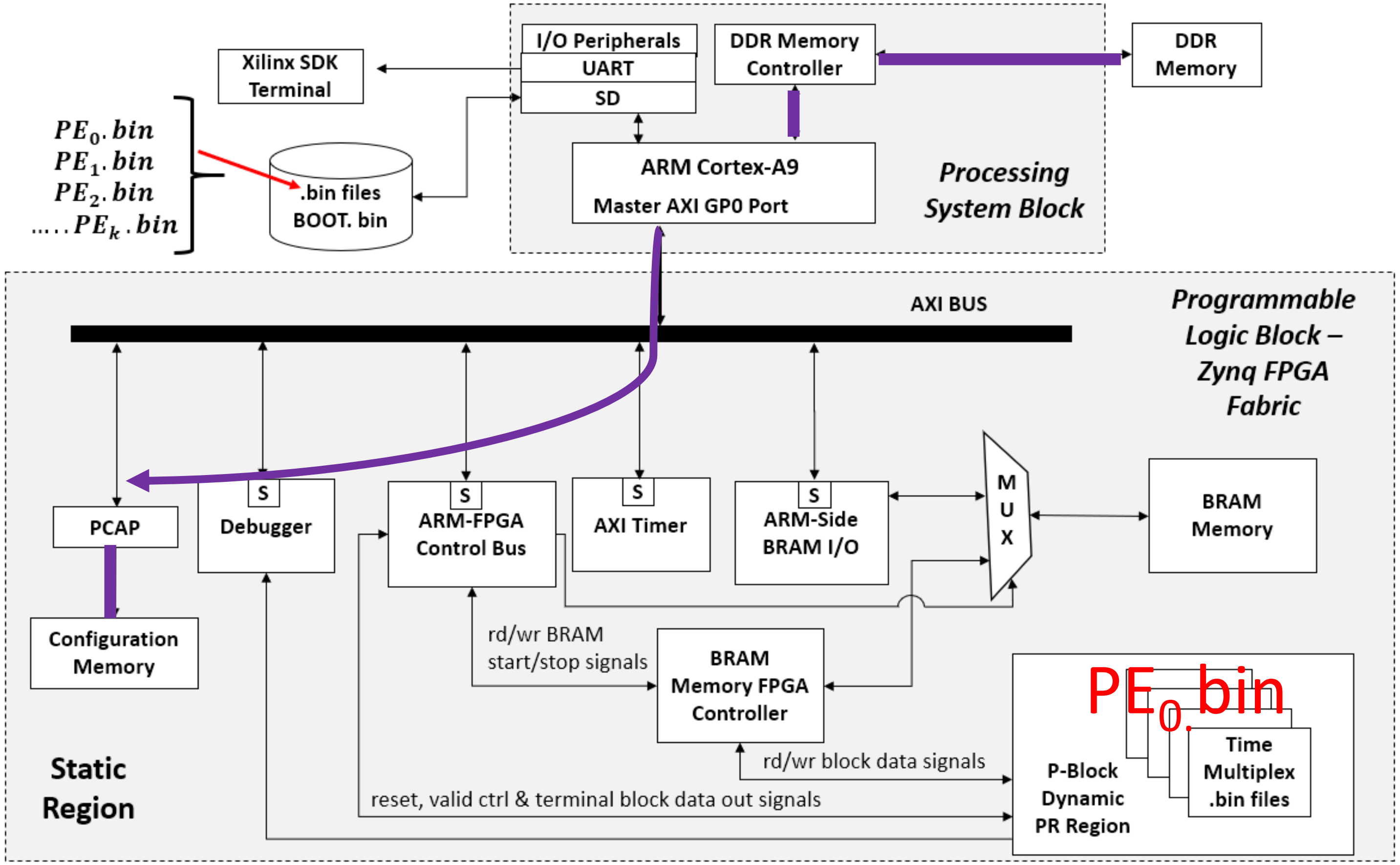


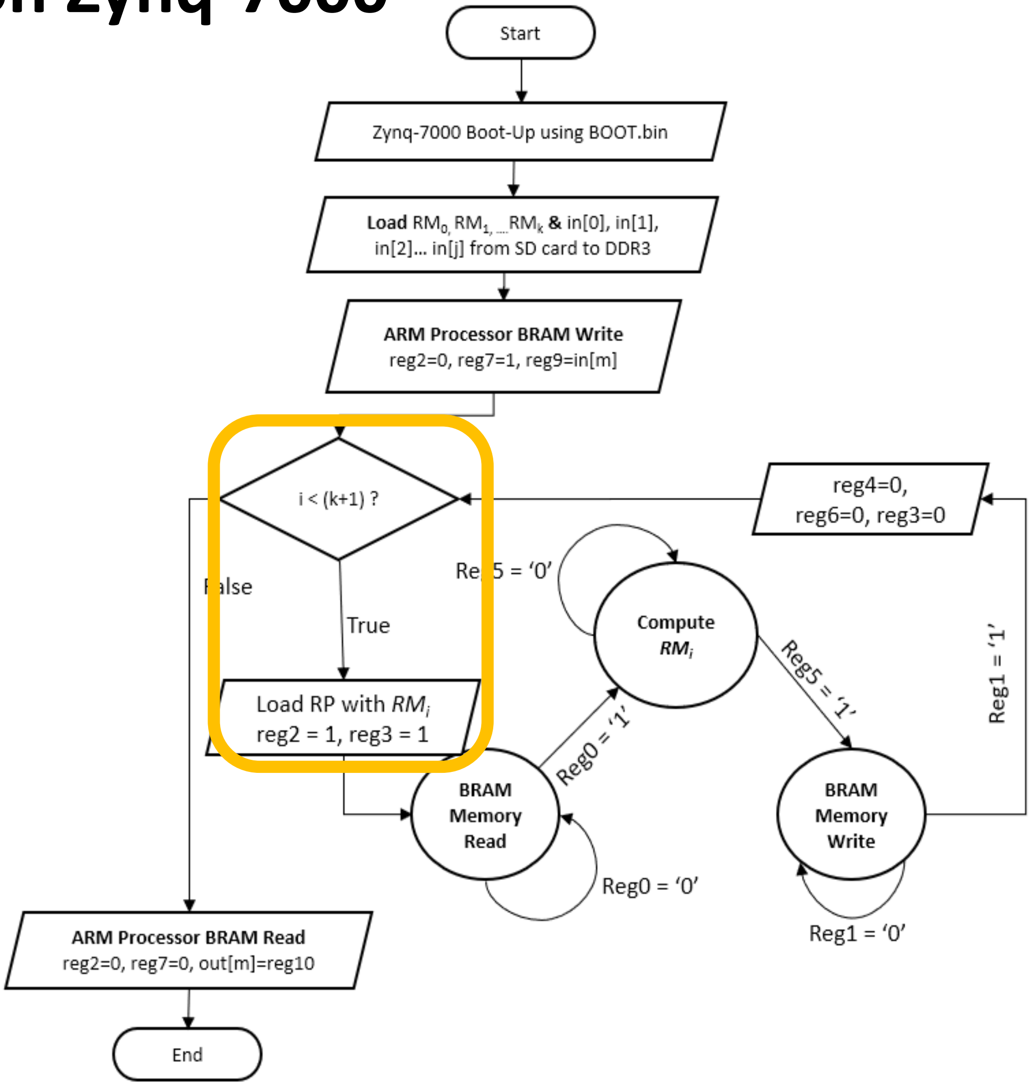
Mux Sel=0



- **Load the BRAM Memory with Raw Data from DDR**

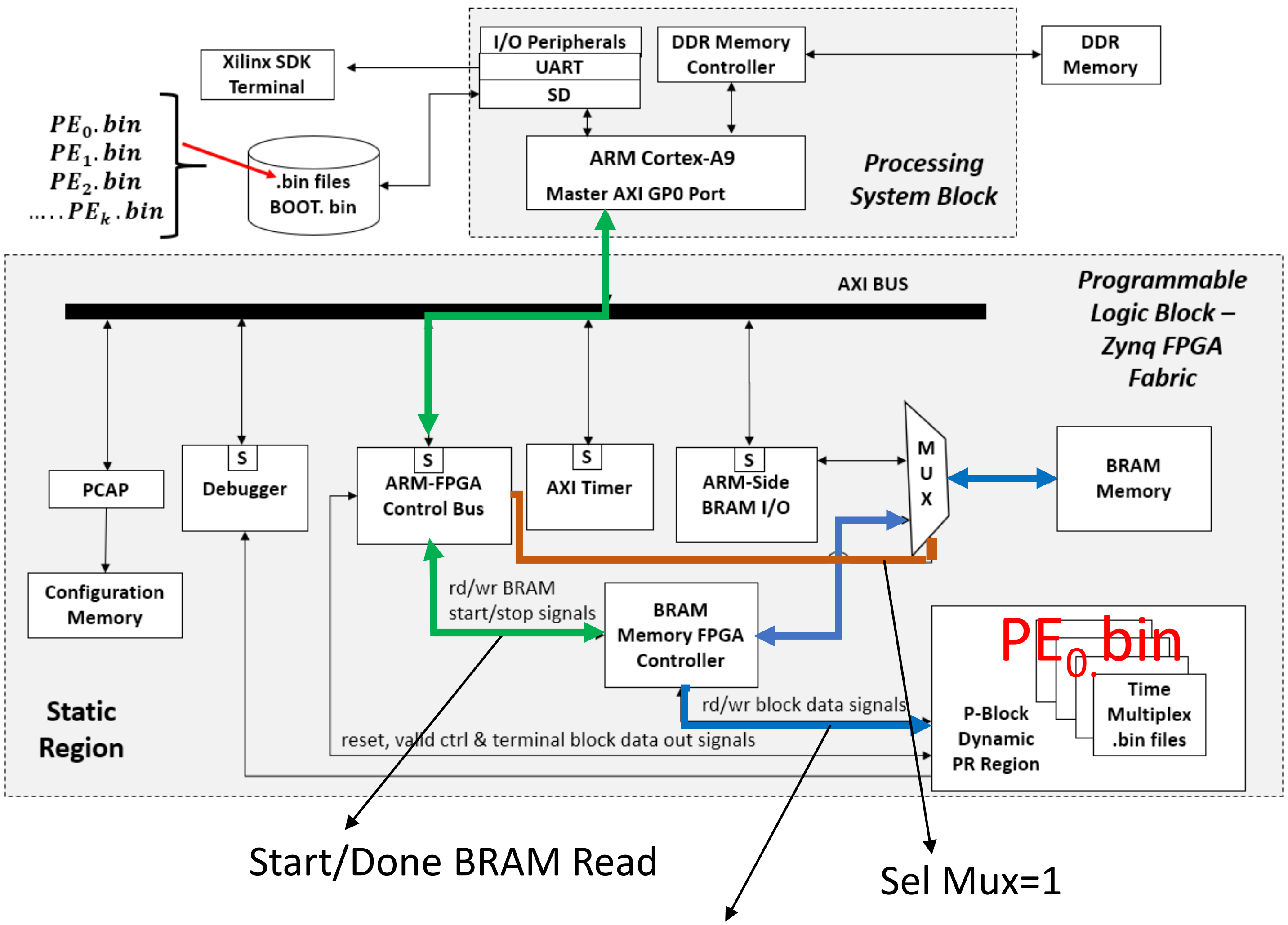
Stage 4: Deploying the Binaries on Zynq-7000





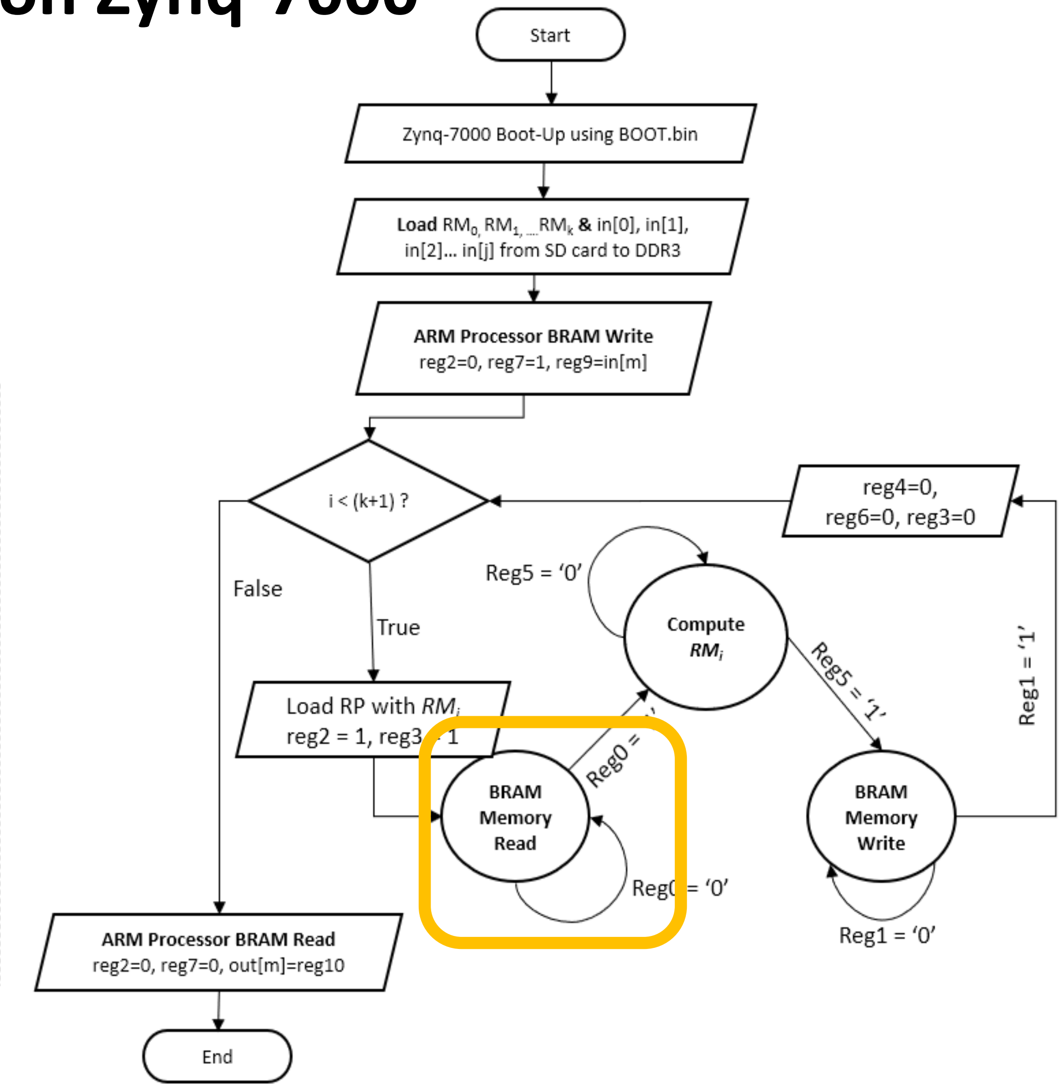
- PCAP fetches partial bitfile (PE_0.bin) from ddr3 memory to load into configuration port

Stage 4: Deploying the Binaries on Zynq-7000



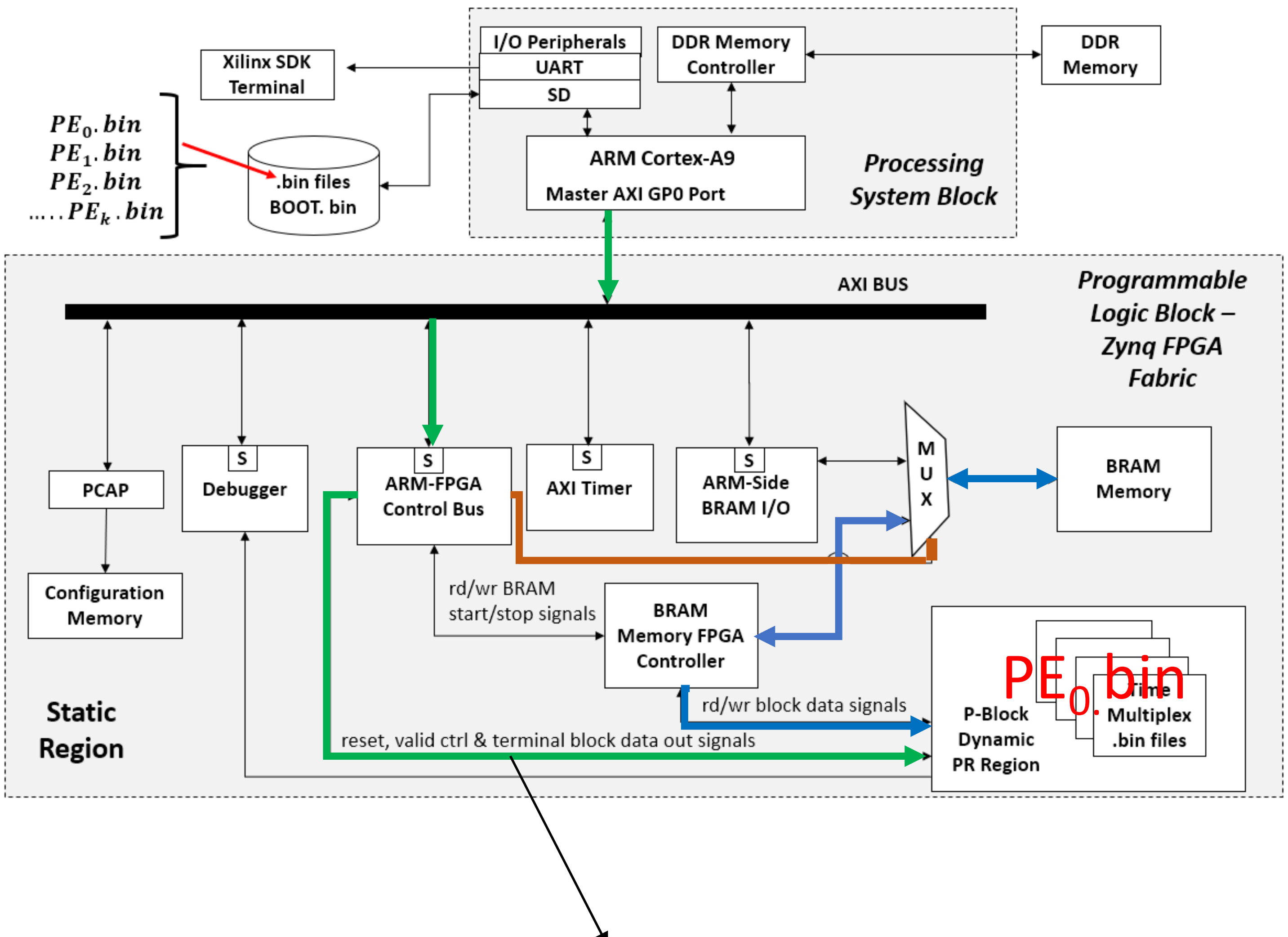
Start/Done BRAM Read

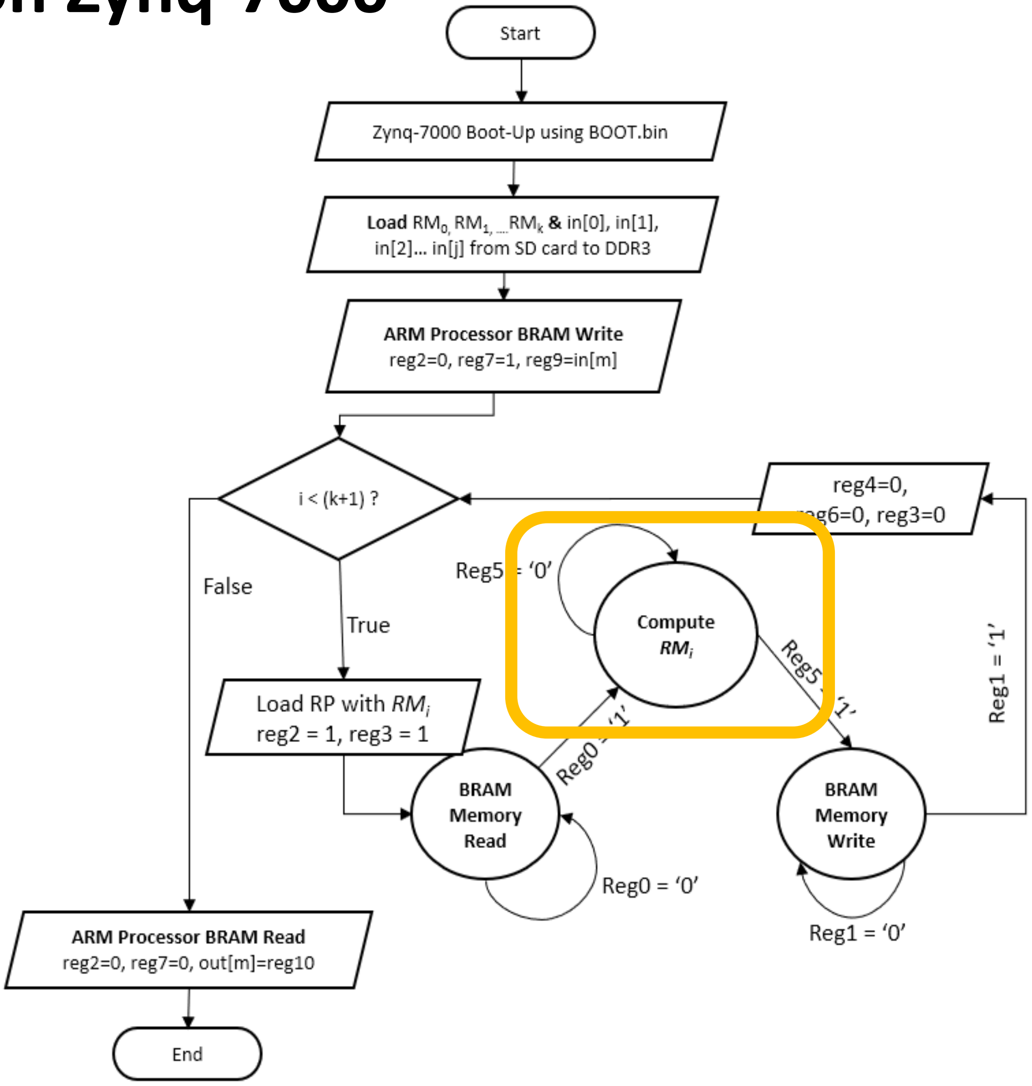
Sel Mux=1



- ➢ **Read the Raw Image data from BRAM Memory and Input it to PE_0.bin**

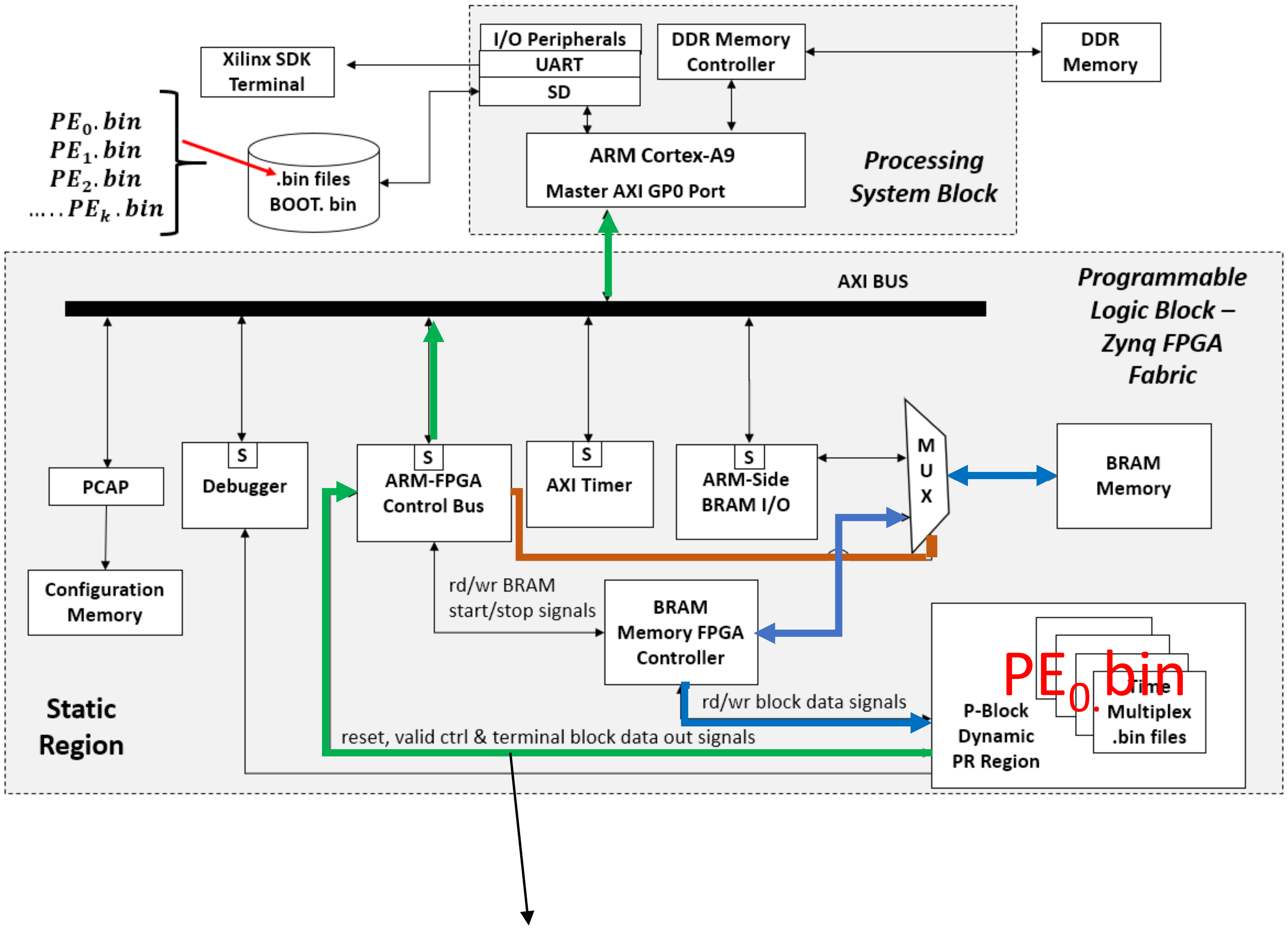
Stage 4: Deploying the Binaries on Zynq-7000

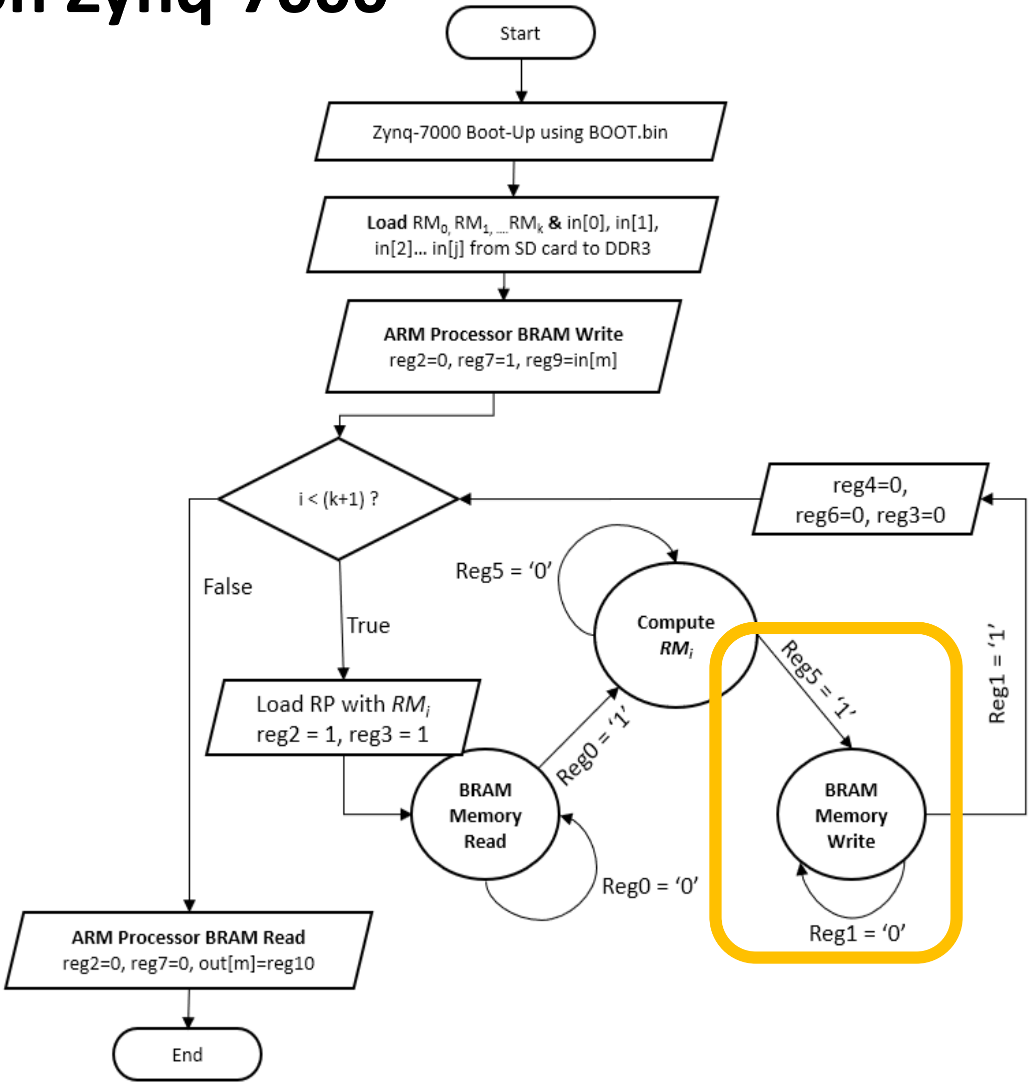

$PE_0.bin$
$PE_1.bin$
$PE_2.bin$
$.....PE_k.bin$
Xilinx SDK Terminal
.bin files
BOOT. bin
I/O Peripherals
UART
SD
DDR Memory Controller
DDR Memory
ARM Cortex-A9
Master AXI GP0 Port
Processing System Block
AXI BUS
Programmable Logic Block – Zynq FPGA Fabric
PCAP
S
Debugger
S
ARM-FPGA Control Bus
S
AXI Timer
S
ARM-Side BRAM I/O
M U X
BRAM Memory
Configuration Memory
rd/wr BRAM start/stop signals
BRAM Memory FPGA Controller
rd/wr block data signals
reset, valid ctrl & terminal block data out signals
P-Block Dynamic PR Region
Time Multiplex .bin files
$PE_0.bin$
Static Region


Start
Zynq-7000 Boot-Up using BOOT.bin
Load RM_0, RM_1, ... RM_k & in[0], in[1], in[2]... in[j] from SD card to DDR3
ARM Processor BRAM Write
reg2=0, reg7=1, reg9=in[m]
i < (k+1) ?
reg4=0, reg6=0, reg3=0
False
True
Load RP with RM_i
reg2 = 1, reg3 = 1
Reg5 = '0'
Compute RM_i
Reg5 = '1'
Reg0 = '1'
Reg1 = '1'
BRAM Memory Read
BRAM Memory Write
Reg0 = '0'
Reg1 = '0'
ARM Processor BRAM Read
reg2=0, reg7=0, out[m]=reg10
End

- ➢ **Enable 'start computation' signal from ARM for PE_0.bin**

Stage 4: Deploying the Binaries on Zynq-7000

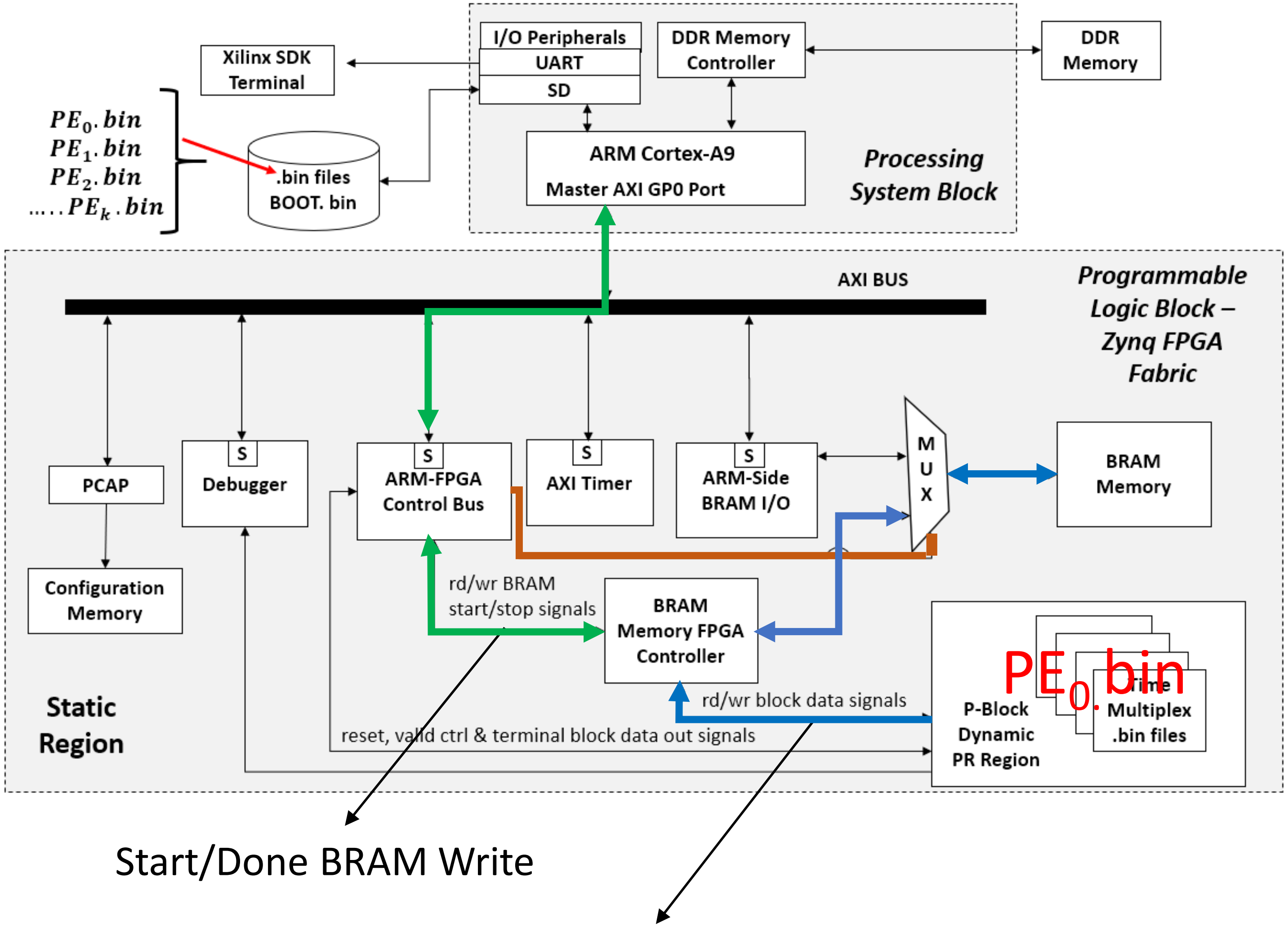




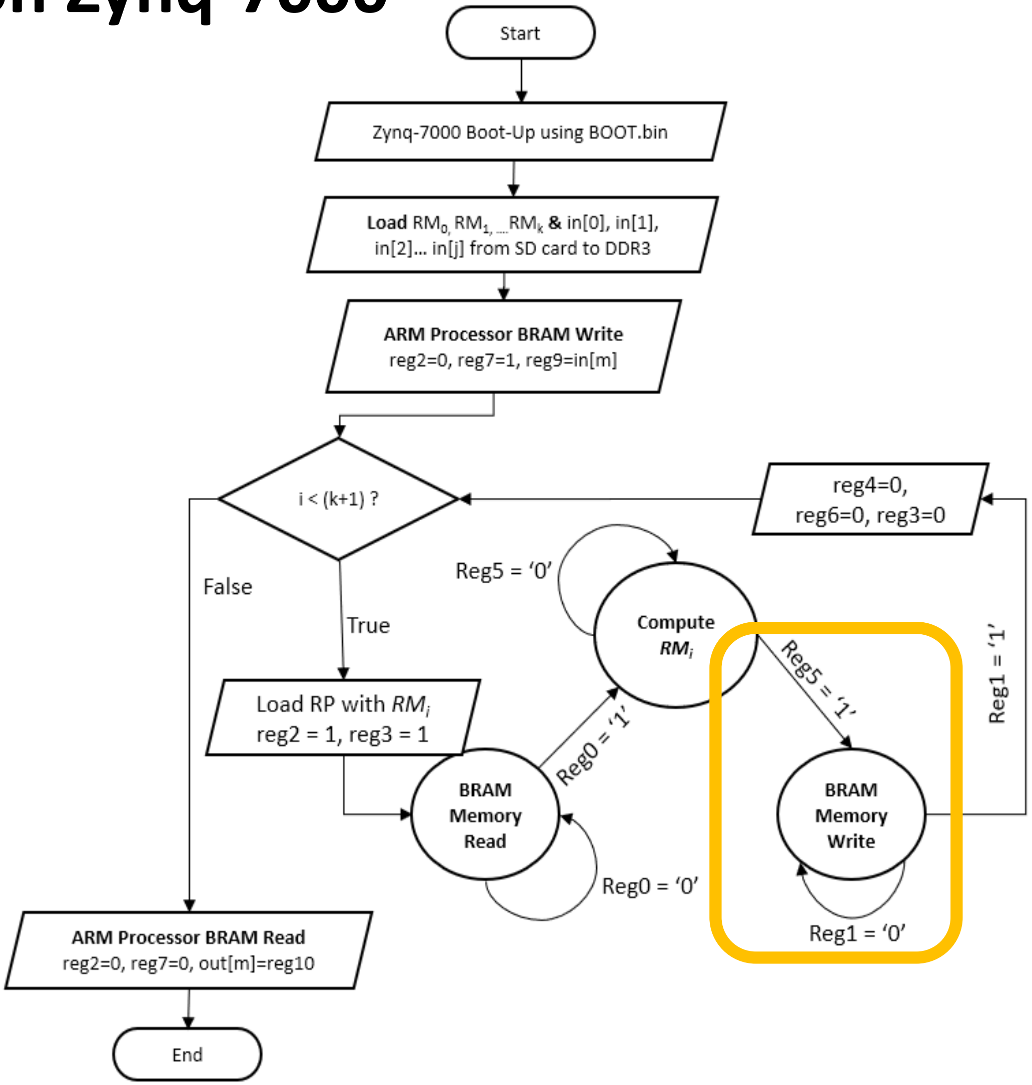
Start
Zynq-7000 Boot-Up using BOOT.bin
Load RM_0, RM_1, ... RM_k & in[0], in[1], in[2]... in[j] from SD card to DDR3
ARM Processor BRAM Write reg2=0, reg7=1, reg9=in[m]
i < (k+1) ?
False
True
reg4=0, reg6=0, reg3=0
Load RP with RM_i reg2 = 1, reg3 = 1
Compute RM_i
Reg5 = ‘0’
Reg5 = ‘1’
Reg0 = ‘1’
BRAM Memory Read
Reg0 = ‘0’
BRAM Memory Write
Reg1 = ‘0’
Reg1 = ‘1’
ARM Processor BRAM Read reg2=0, reg7=0, out[m]=reg10
End

- Read ‘done computation’ signal to ARM for PE_0.bin

Stage 4: Deploying the Binaries on Zynq-7000

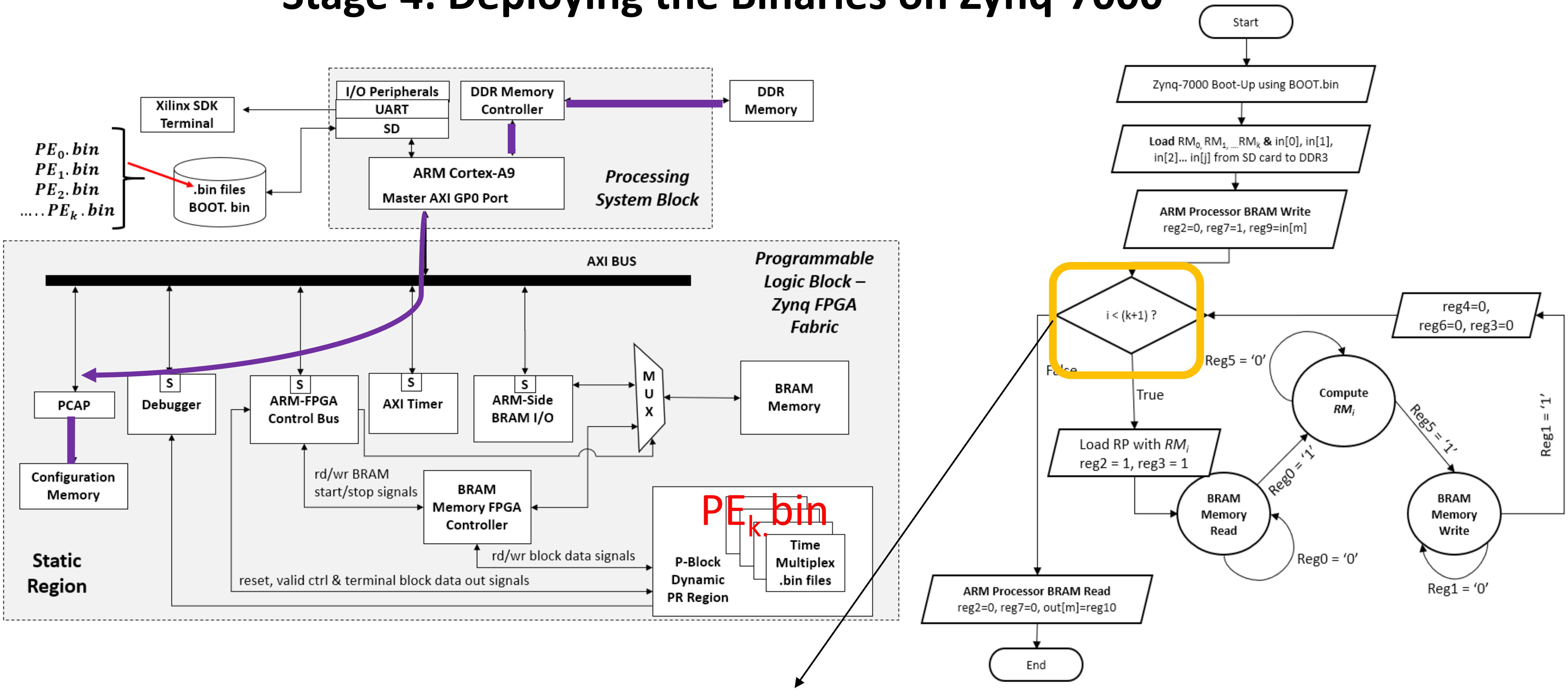


Start/Done BRAM Write



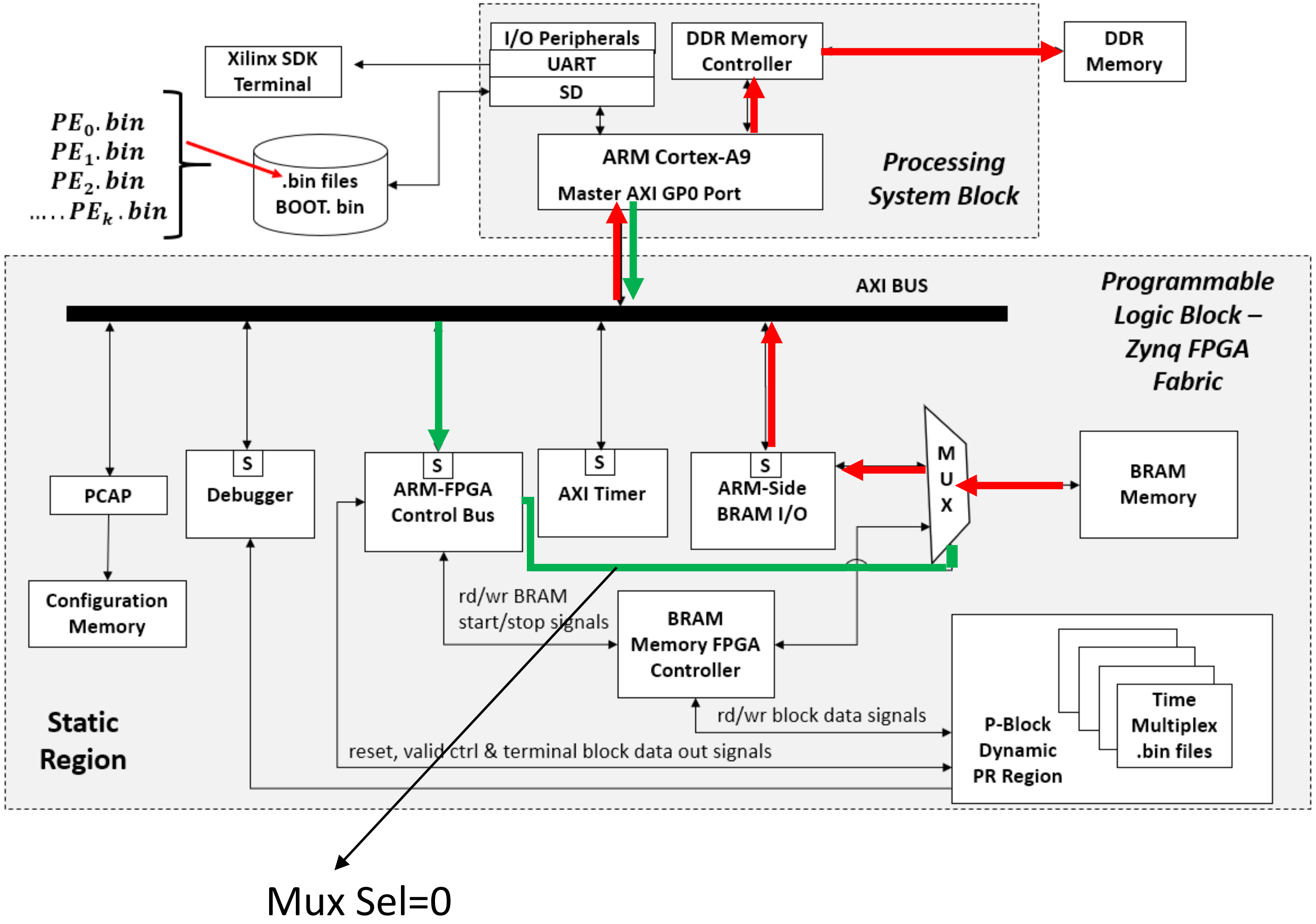
- **Write the output generated by PE_0.bin to BRAM memory**

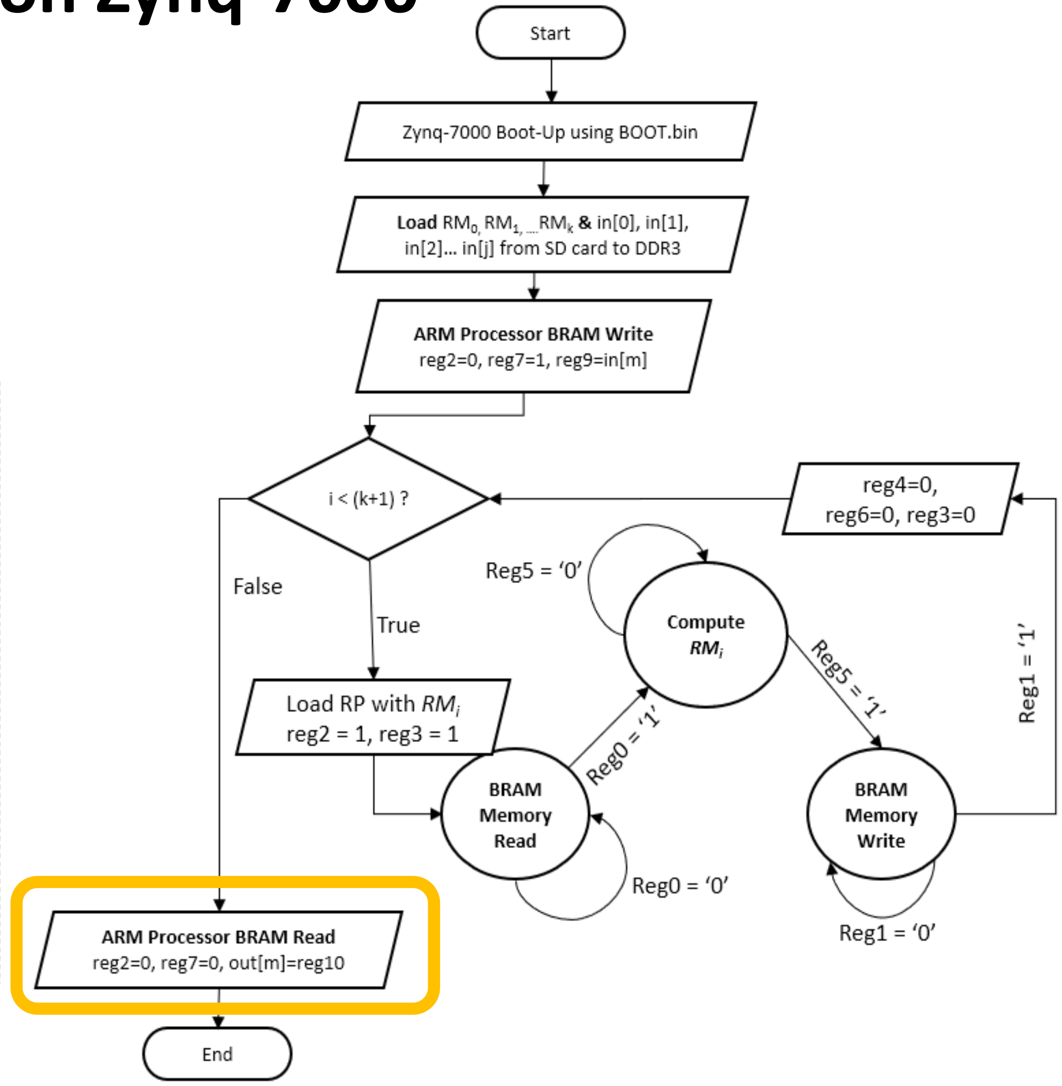
Stage 4: Deploying the Binaries on Zynq-7000



- The process to Load RM, Read BRAM, Compute & Write BRAM continues till the terminal PE …
- Finally, PCAP fetches partial bitfile (PE_k.bin) from ddr3 memory to load into configuration port

Stage 4: Deploying the Binaries on Zynq-7000



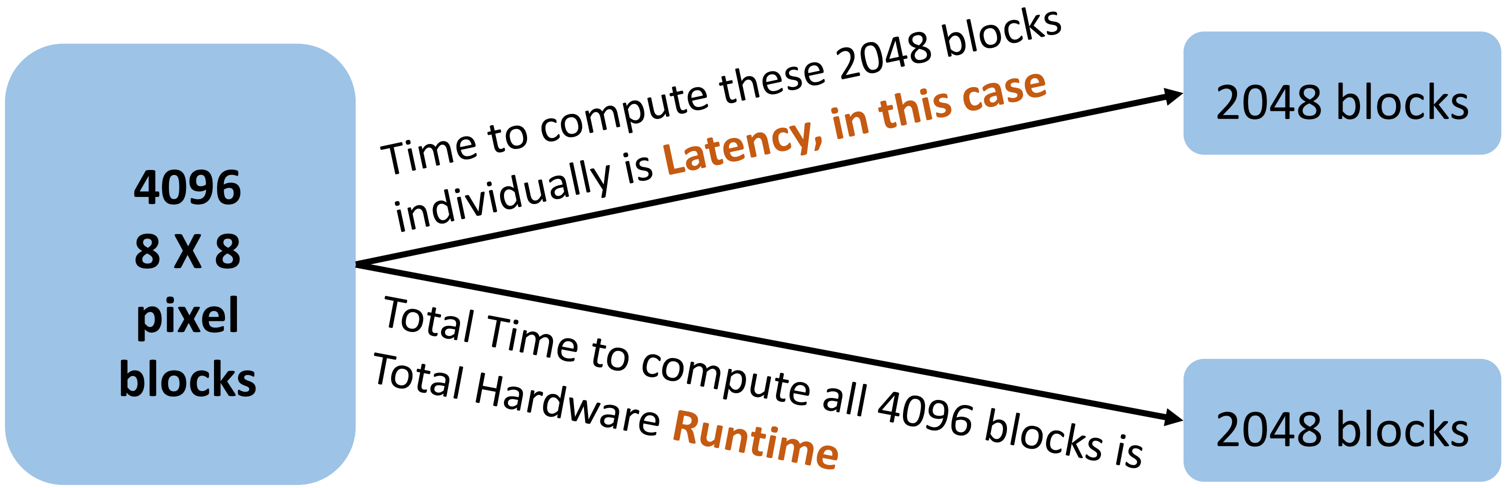


- ➢ **Load the Results generated by PE_k.bin to SD Card from BRAM Memory**

Design Automation and Reconfigurable Computing

JPEG Encoder PR_{BRAM} Design Implementation

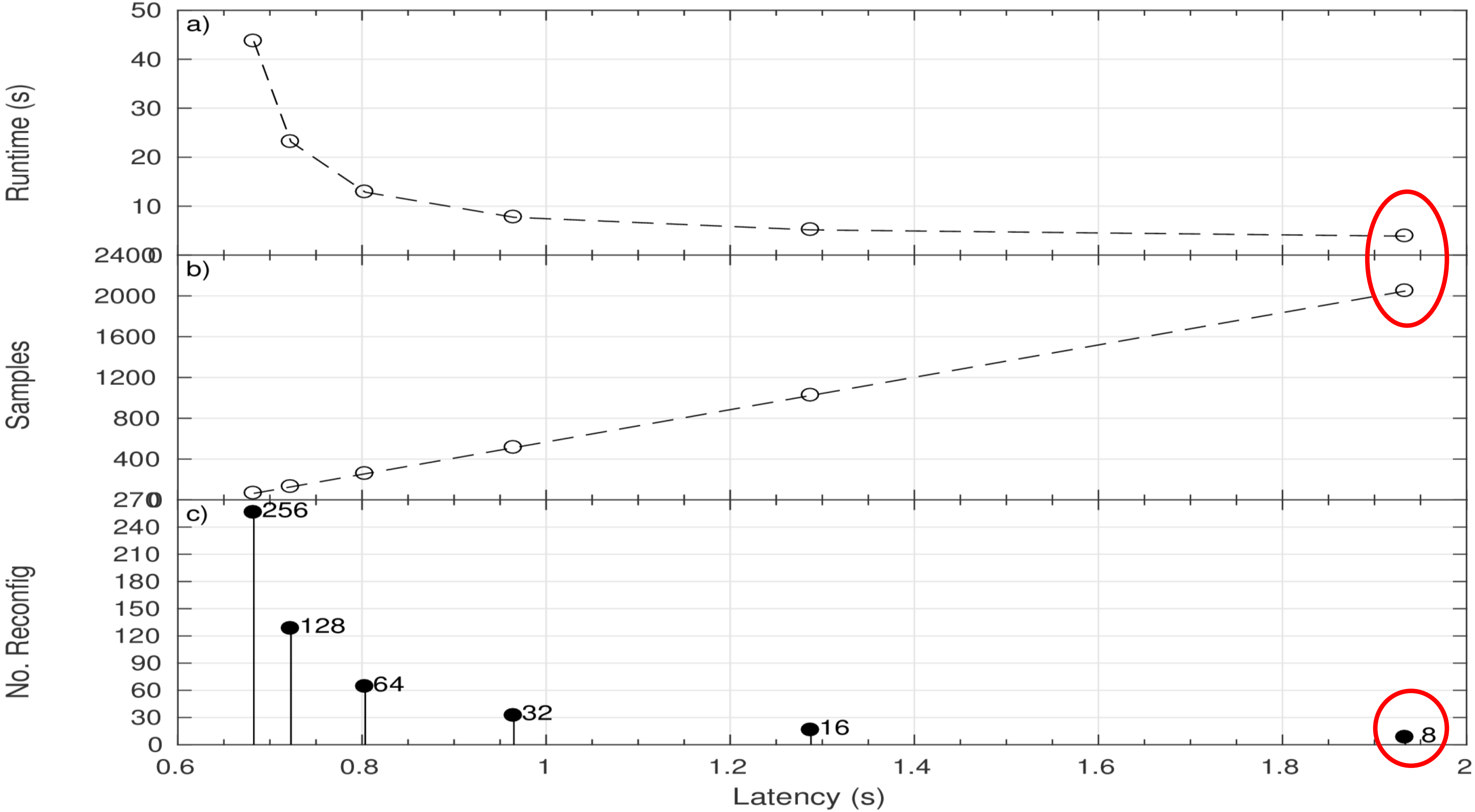
- Utilizing 91.42 % of BRAM memory to store intermediate results
- *Total memory requirements (1048.576 Kbytes)* **>** Available BRAM memory (140 blocks X 36Kb = 630 Kbytes)
  - Minimum number of partial reconfigurations = 8 (4 (Reconfigurable Modules) * 2(Dividing factor))



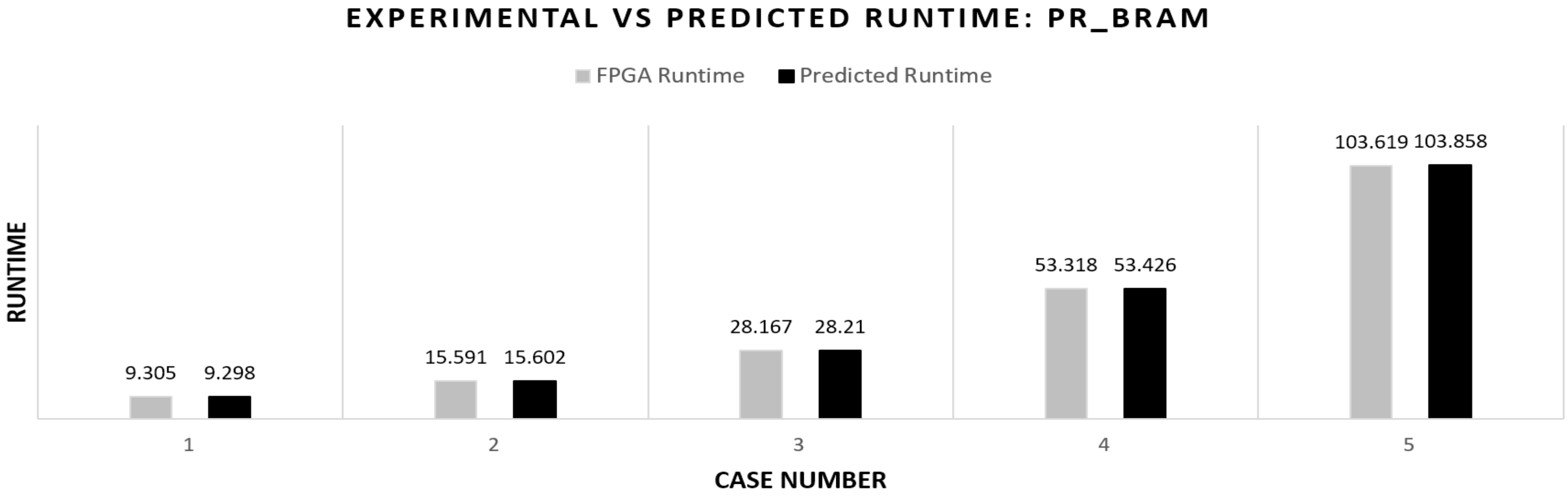
4096 8 X 8 pixel blocks
Time to compute these 2048 blocks individually is **Latency, in this case**
2048 blocks
Total Time to compute all 4096 blocks is Total Hardware **Runtime**
2048 blocks

Divided the image dataset into 2 & Reload BRAM

PR_{BRAM} Design Implementation – Experimental Result



PR_{BRAM} Design Implementation – Experimental Results II



Cases of no. of reconfigurations (1) 32 (2) 64 (3) 128 (4) 256 (5) 512

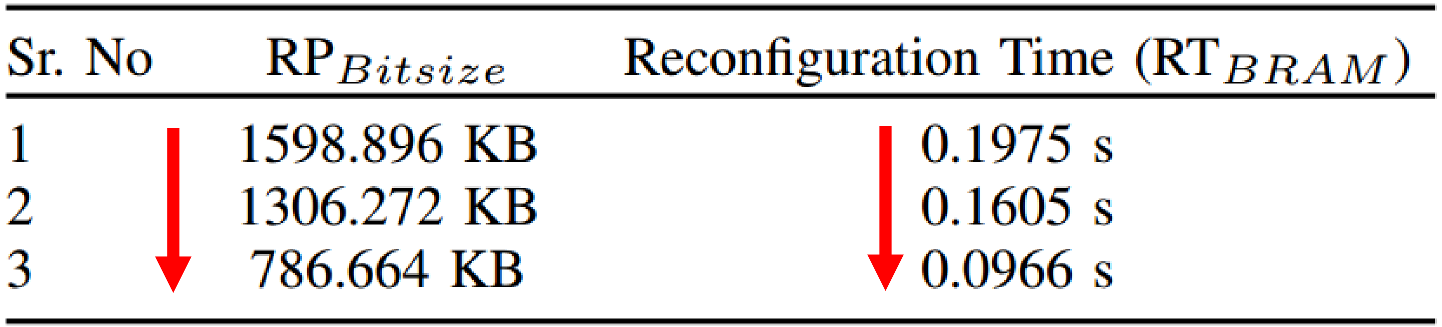
$$\boldsymbol{T_{runtime} = \{T_{jpeg-computing} + T_{overhead} + T_{bin} * N_{bin}\}}$$

- $T_{jpeg\text{-}computing}$: actual computing time it takes for processing all the inputs of each reconfigurable module
- T_{bin} : time it takes to partially configure the bitstream
- N_{bin} : number of times reconfiguration occurs
- $T_{overhead}$: 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, $T_{jpeg\text{-}computing}$ = 2.994 s and $T_{overhead}$ = 1.675 s are obtained experimentally

PR_{BRAM} Design Implementation – Experimental Results III

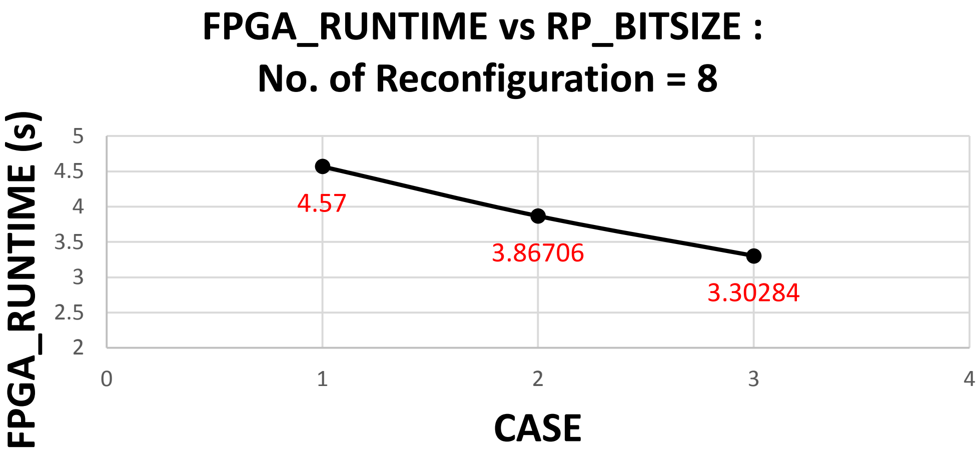
RT_{BRAM} values for varying $RP_{Bitsize}$

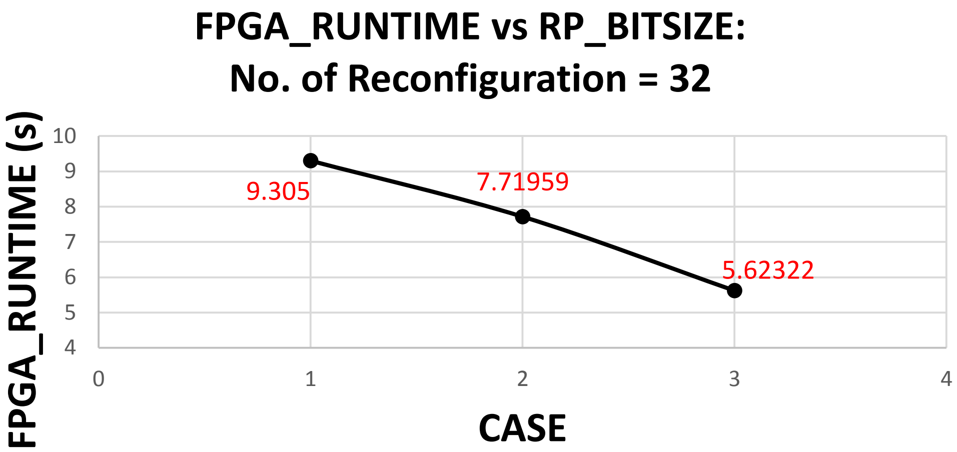


Sr. No	$RP_{Bitsize}$	Reconfiguration Time (RT_{BRAM})
1	1598.896 KB	0.1975 s
2	1306.272 KB	0.1605 s
3	786.664 KB	0.0966 s

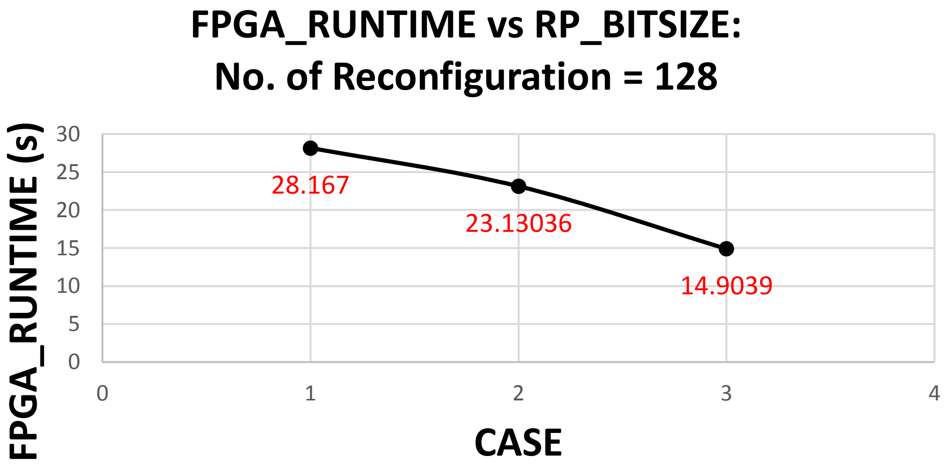
- The size of the pblock or reconfigurable partition affects T_{bin}
  - There is a linear relationship
- The Hardware running time for reconfigurable architectures is significantly impacted by this.

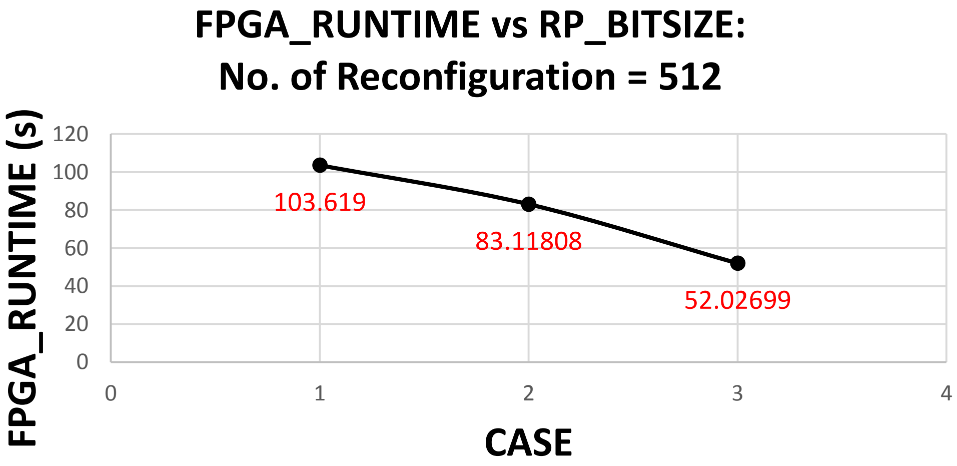
PR_{BRAM} Design Implementation – Experimental Results IV





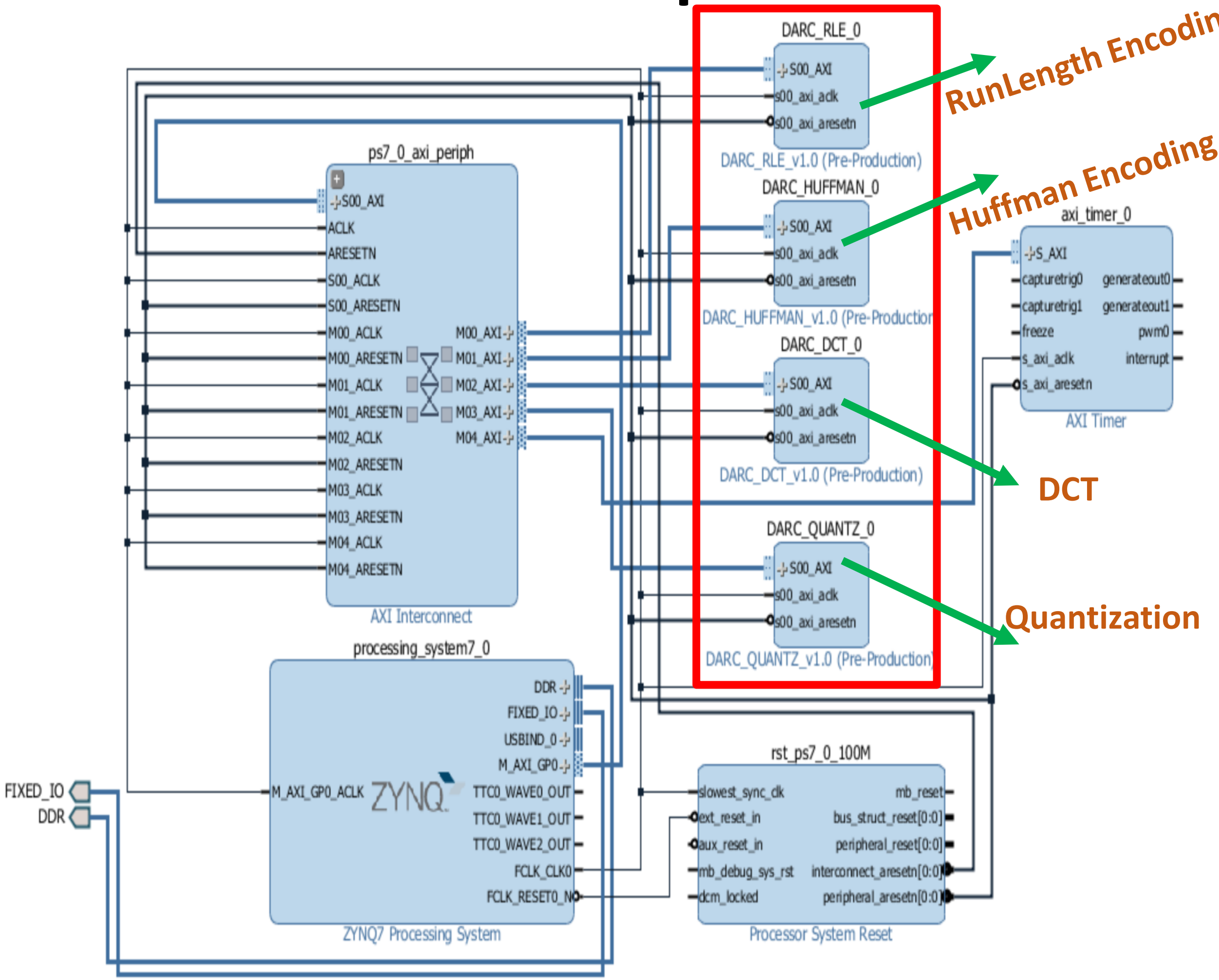
[Case 1: $RP_{Bitsize}$ = 1598.896 KB, Case 2: $RP_{Bitsize}$ = 1306.272 KB, Case 3: $RP_{Bitsize}$ = 786.664 KB]





DARClab Design Automation and Reconfigurable Computing

JPEG Encoder Spatial Method



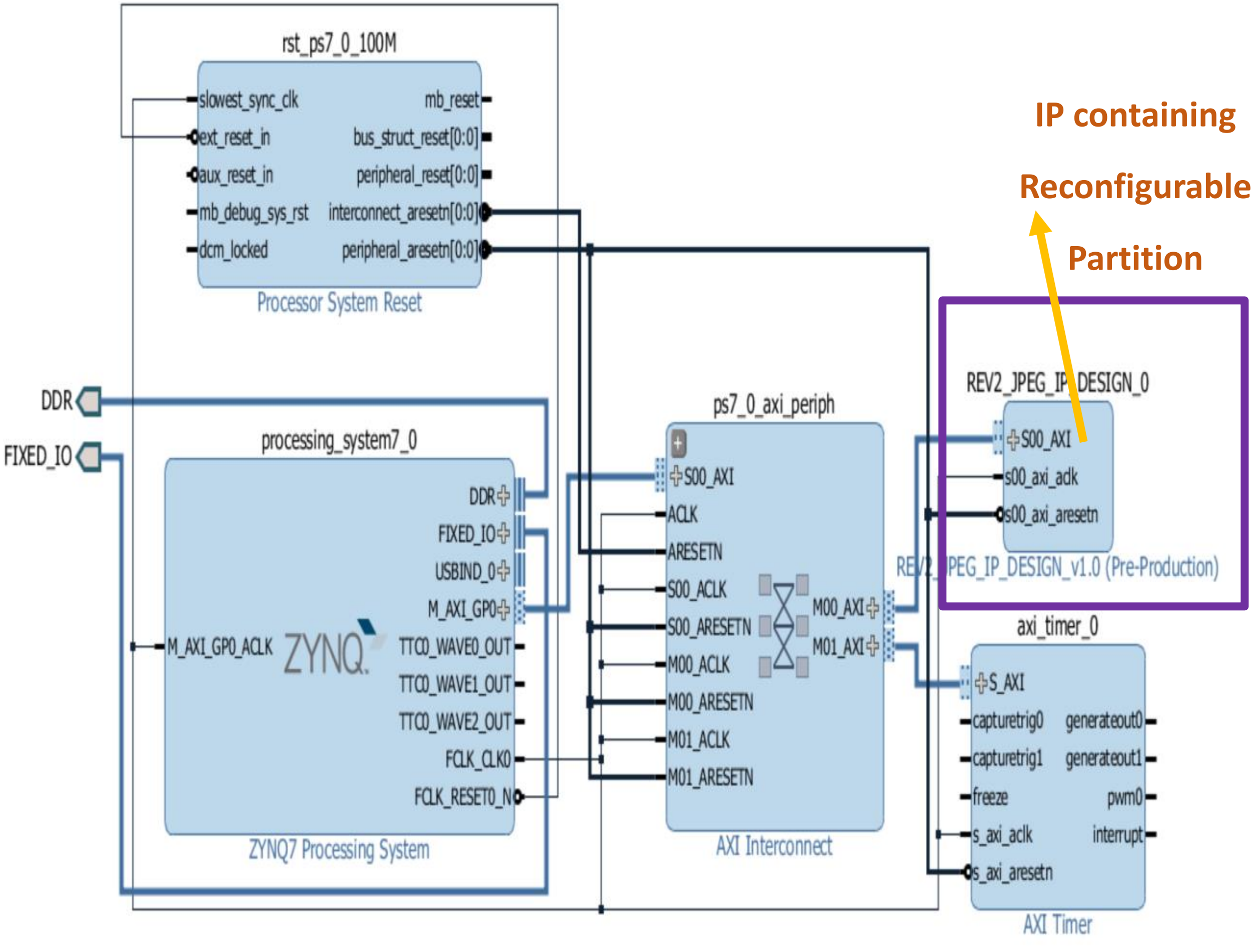
RunLength Encoding

Huffman Encoding

DCT

Quantization

JPEG Encoder PR_{DDR} Method



IP containing

Reconfigurable

Partition

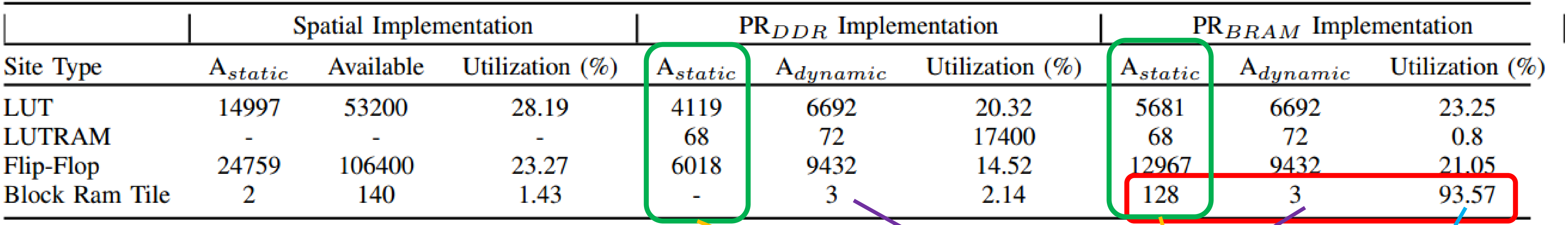
Objective:

- ➢ Prove **area utilization efficacy** of PR_{BRAM}

Objective:

- ➢ Prove PR_{BRAM} is **runtime and latency efficient** compared to PR_{DDR}

Calculating Area Utilization



	Spatial Implementation			PR_{DDR} Implementation			PR_{BRAM} Implementation		
Site Type	A_{static}	Available	Utilization (%)	A_{static}	$A_{dynamic}$	Utilization (%)	A_{static}	$A_{dynamic}$	Utilization (%)
LUT	14997	53200	28.19	4119	6692	20.32	5681	6692	23.25
LUTRAM	-	-	-	68	72	17400	68	72	0.8
Flip-Flop	24759	106400	23.27	6018	9432	14.52	12967	9432	21.05
Block Ram Tile	2	140	1.43	-	3	2.14	128	3	93.57

RunLength Encoding PE

Max. BRAM Utilization

A_{static} of PR_{BRAM} is high due to additional IPs in the overlay architecture

Static Designs

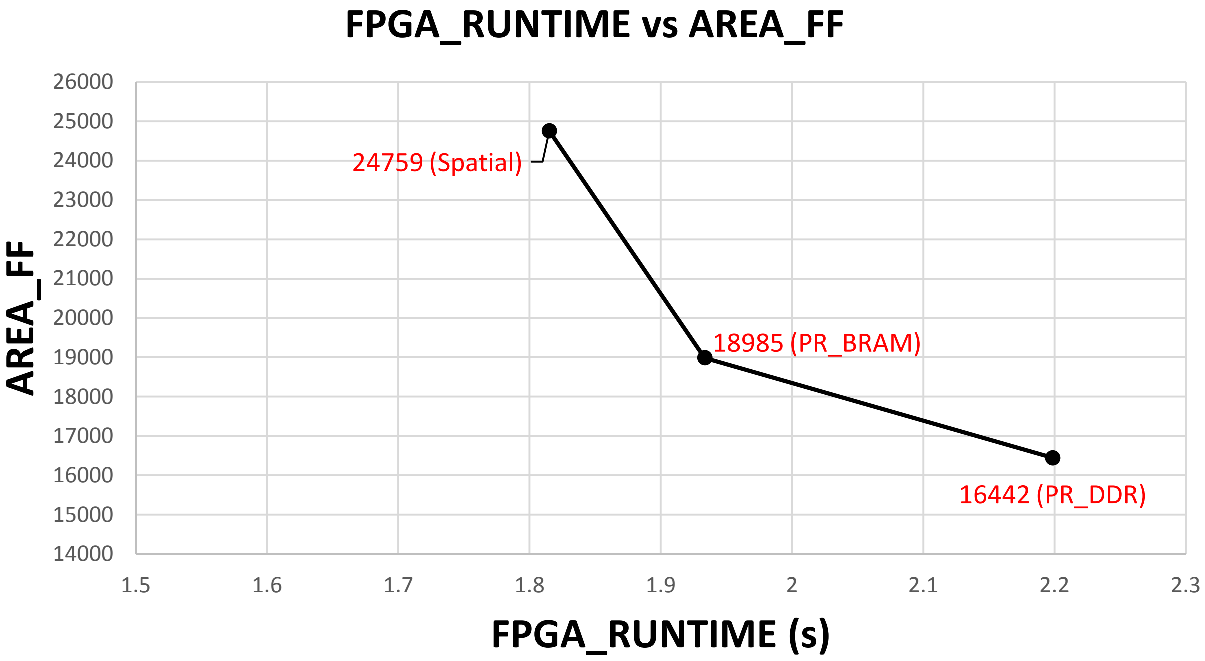
$$A_{total} = \sum (PE_0, PE_1, PE_2, \ldots . PE_k)$$

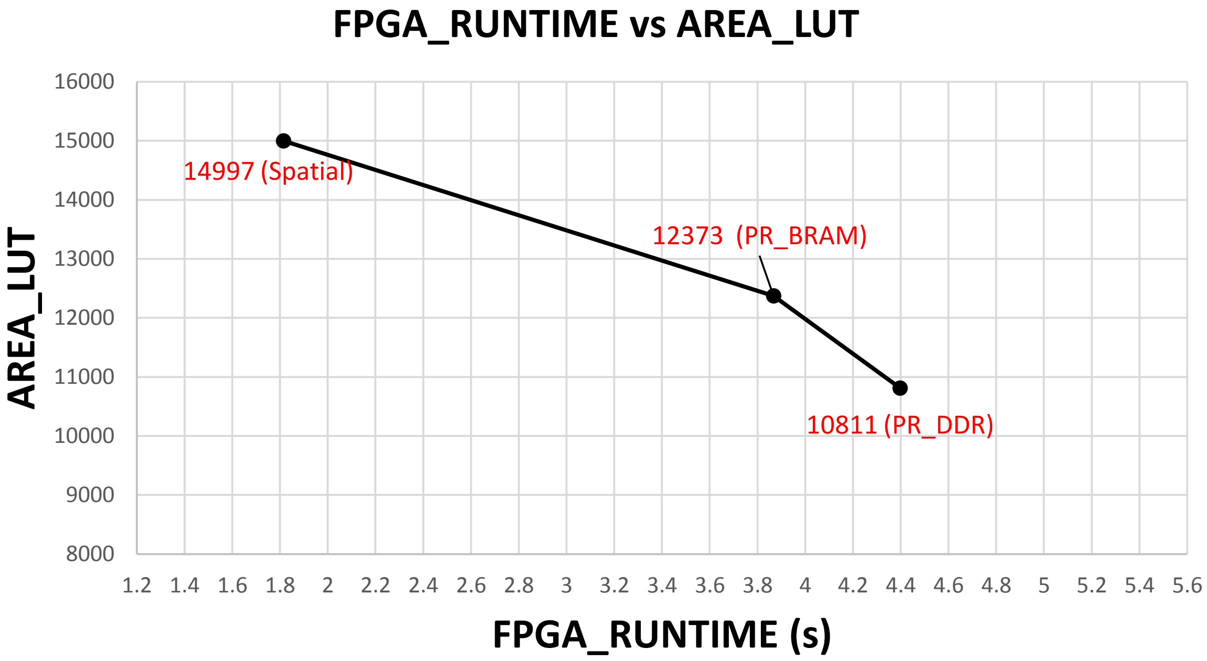
PR Based Designs

$$A_{total} = \sum \left\{A_{static} + A_{dynamic}\{max(PE_0, PE_1, PE_2, \ldots . PE_k)\}\right\}$$

*PE: Process Element

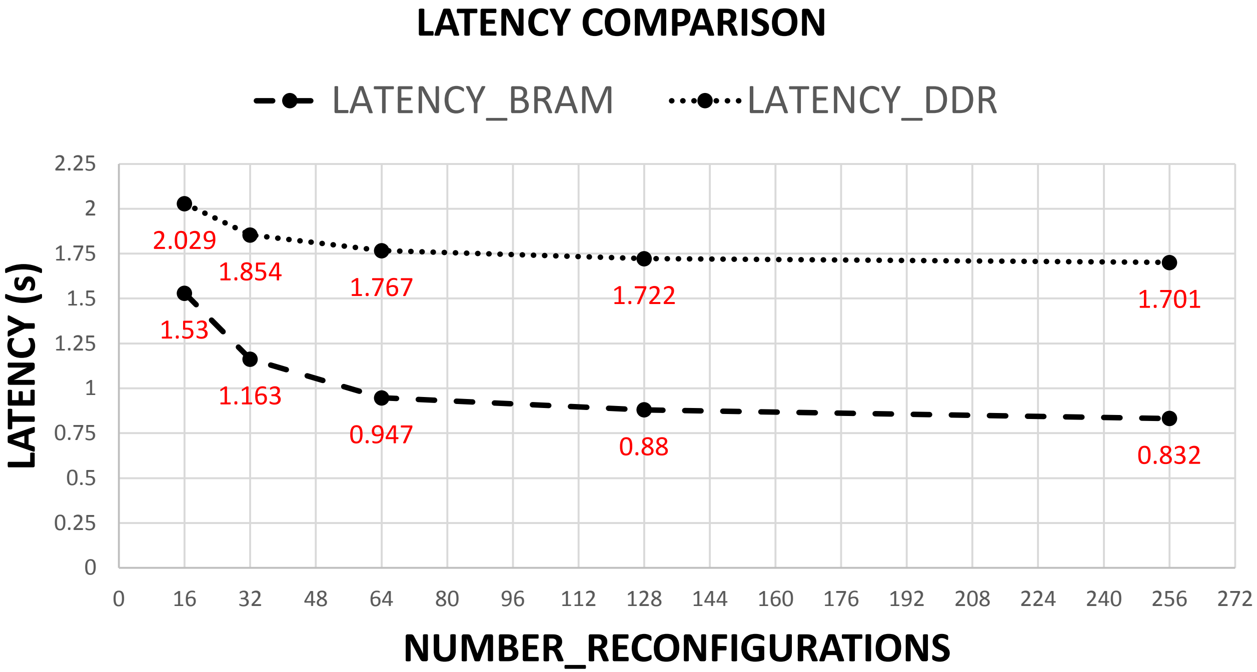
I. COMPARATIVE STUDY – AREA UTILIZATION vs FPGA RUNTIME



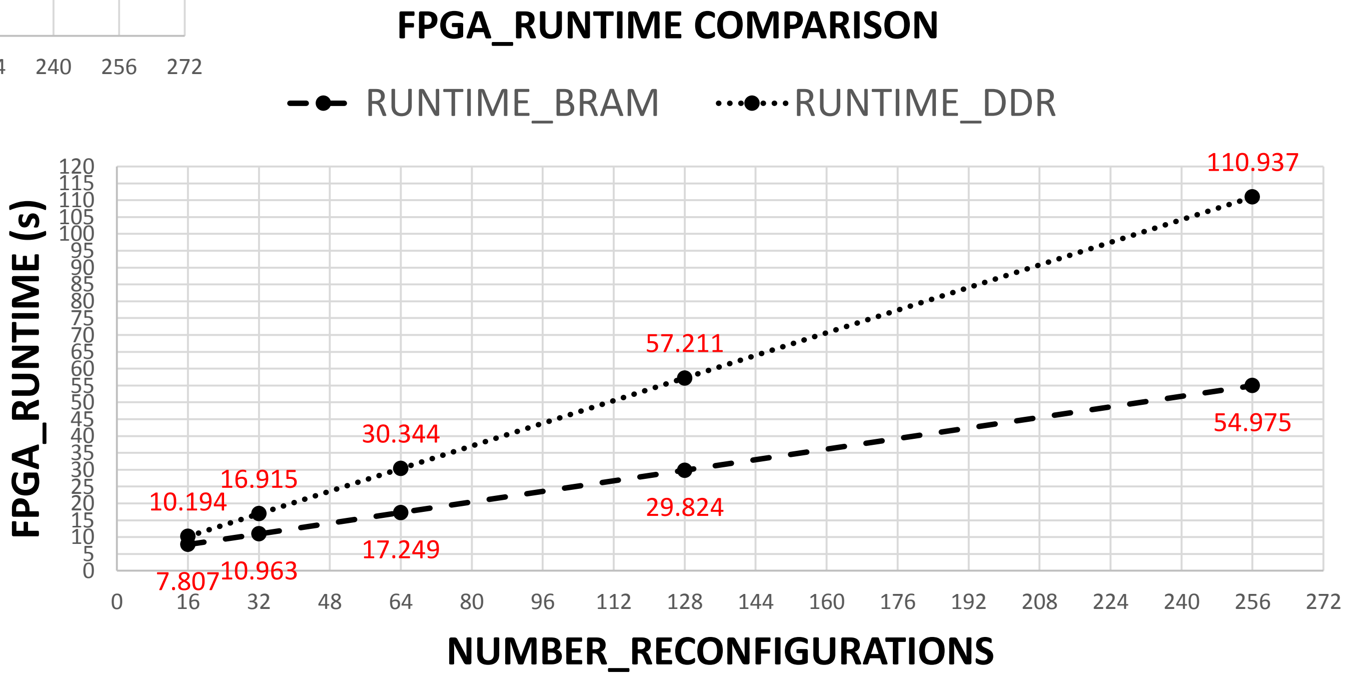


- PR_{BRAM} design method requires slightly more area compared to PR_{DDR} **due additional IPs - ARM-FPGA Control Bus, ARM-Side BRAM Control, MUX and Block RAM Memory modules**.
- Comparing spatial design implementation, the utilization of PR_{BRAM} is **significantly low**, which is as expected.

II. COMPARATIVE STUDY – RUNNING TIME vs LATENCY

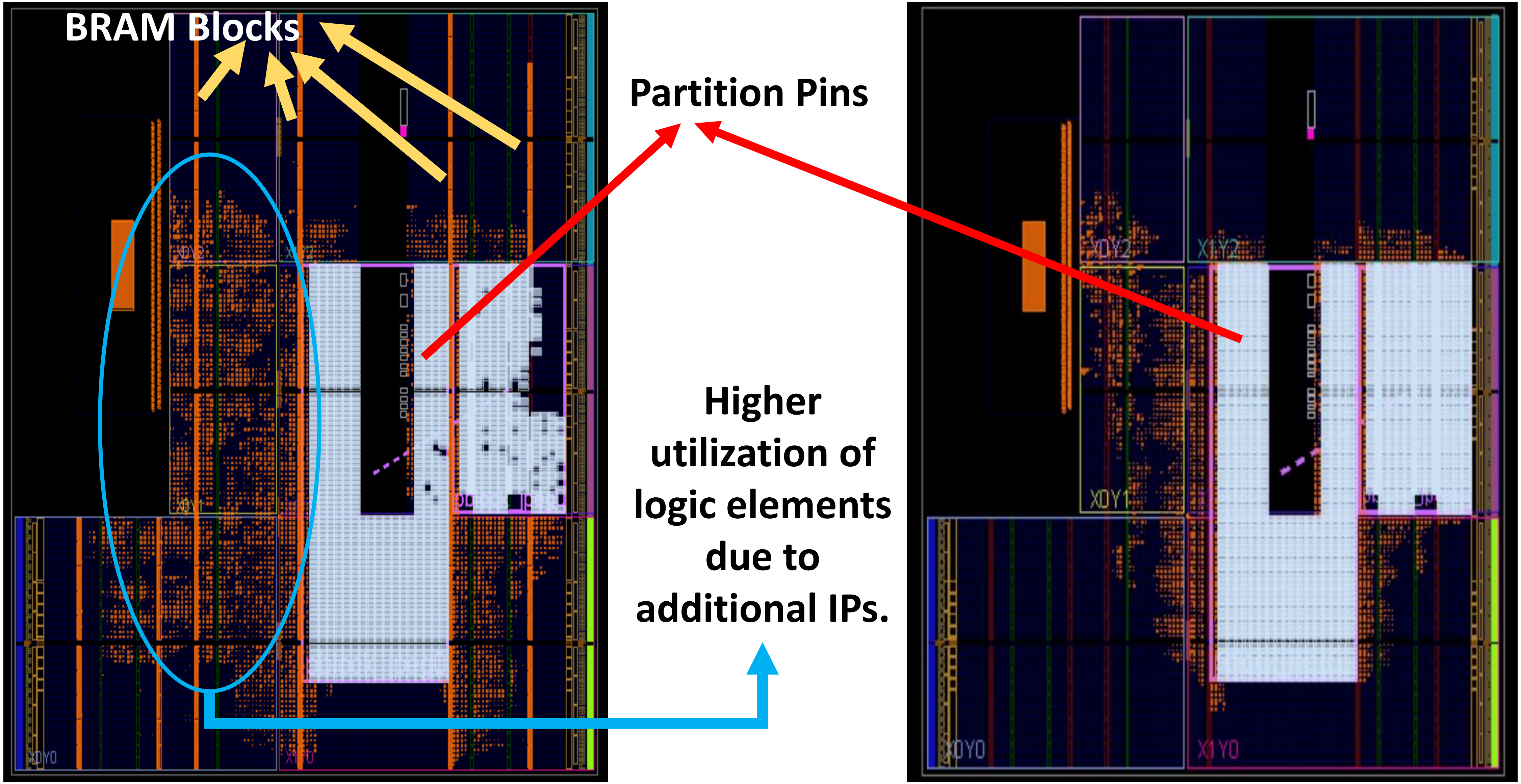


- **Experiment with unequal $RP_{Bitsize}$**
  - $RP_{Bitsize}$ = 3416.088 KB for PR_{DDR}
  - $RP_{Bitsize}$ = 1598.896 KB for PR_{BRAM}
- Non-Linear Relationship in Runtime between the graphs due to **Tbin * Nbin not constant**



Design Automation and Reconfigurable Computing

PR_{BRAM} & PR_{DDR} Static.dcp Floorplan View – Equal Pblock Sizes

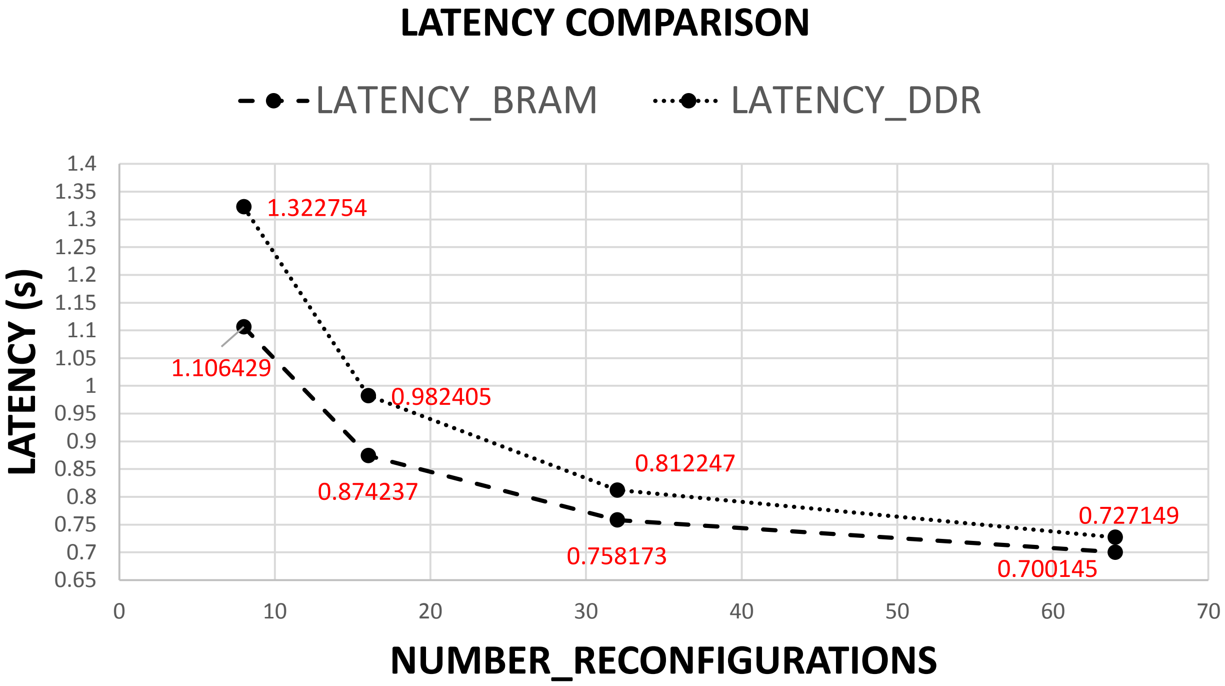


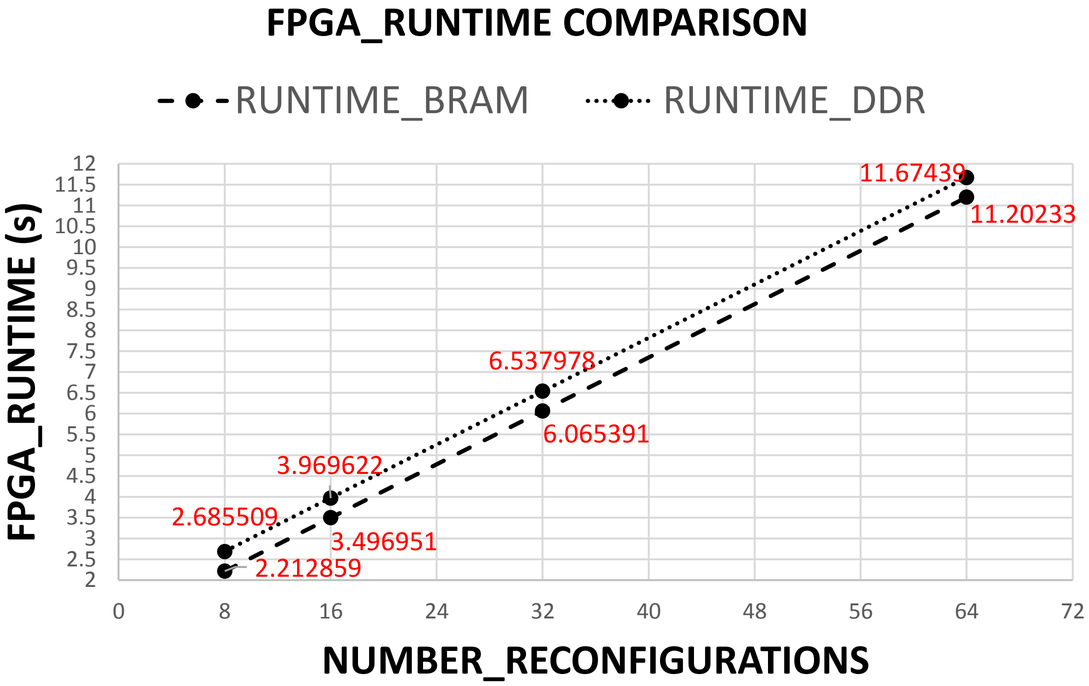
BRAM Blocks

Partition Pins

Higher utilization of logic elements due to additional IPs.

II. COMPARATIVE STUDY – RUNNING TIME vs LATENCY





- **Experiment with equal RP$_{\textit{Bitsize}}$ = 1306.272 KB** for both PR implementations.
  - **Average improvement in runtime is 0.529s**
- **Runtime varies linearly** because **N$_{\textit{bin}}$ * T$_{\textit{bin}}$** ***is constant***

CONCLUSION

- **Novel design methodology** for dataflow computation with proposed PR_{BRAM} overlay static architecture
  - Including TCL based automated floorplanning + User software application algorithms

- Implemented **JPEG Encoder on Zynq -7000**
  - For three testcase images-Lena, Peppers and Goldhill
  - Using Spatial, PR_{DDR} & PR_{BRAM} for Comparative Study & Analysis

- Implementation with the proposed Architecture PR_{BRAM} is area efficient compared to spatial implementation with
  - **LUT area savings up to 21.20 % & FF area savings up to 30.41 %** for 1306.272 KB
  - These %'s are including the additional resources utilized by proposed static architecture

- Improvement in average hardware running (PR_{BRAM}) of 0.529s vs. PR_{DDR}

FUTURE WORKS

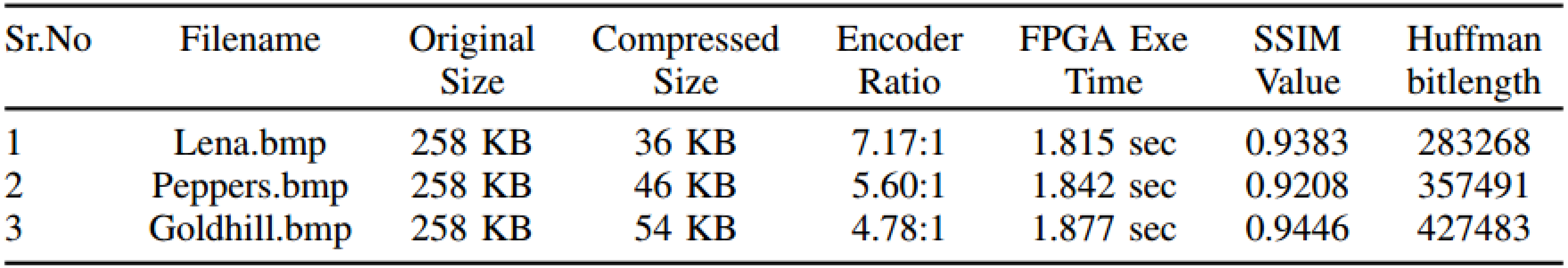
- Sophisticated Partial Reconfiguration Controllers
  - **Minimize the time required for reconfiguring**
- **Enhanced parallelism of operations** in hardware accelerators/Processing Elements due to saved resources in reconfigurable architecture.
- In extremely data-intensive applications, exploring performance impact on PR_{BRAM}
  - **BRAM + Distributed RAM** to deal with limitations of on-chip memory

THANK YOU

APPENDIX

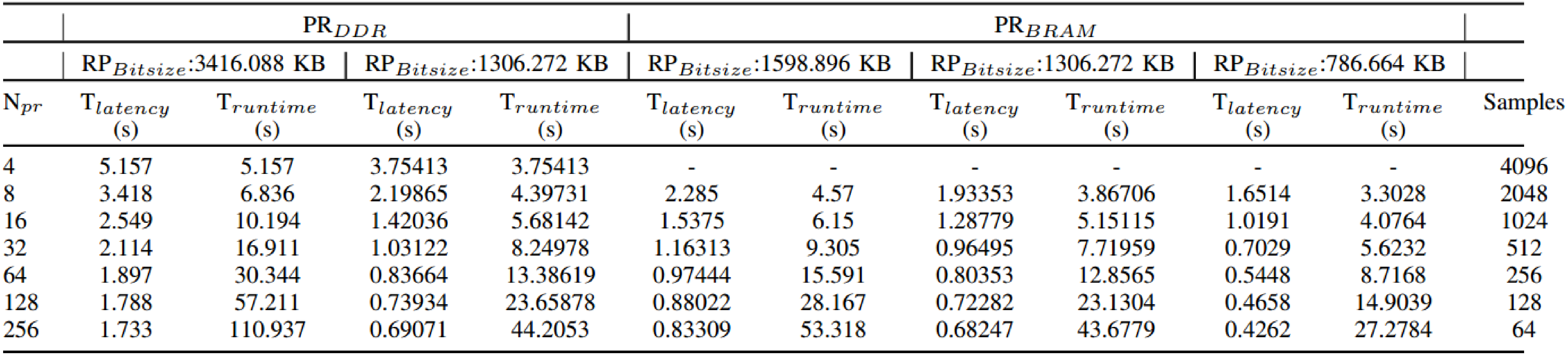
Experimental Results – Spatial, PR_{BRAM} & PR_{DDR}

Table III: JPEG Encoder Results with Spatial Design Implementation



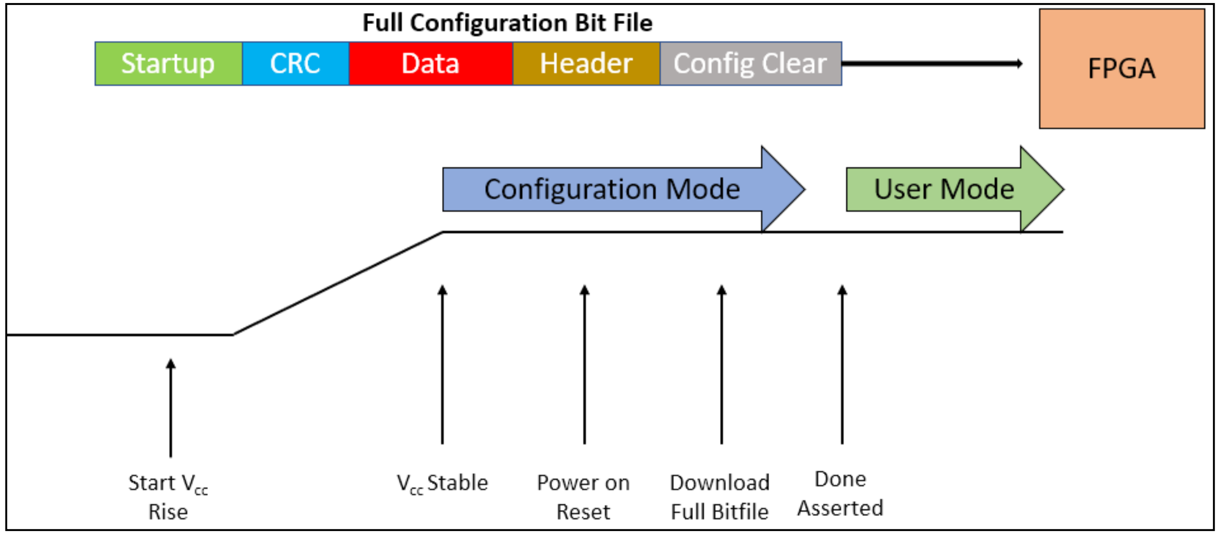
Sr.No	Filename	Original Size	Compressed Size	Encoder Ratio	FPGA Exe Time	SSIM Value	Huffman bitlength
1	Lena.bmp	258 KB	36 KB	7.17:1	1.815 sec	0.9383	283268
2	Peppers.bmp	258 KB	46 KB	5.60:1	1.842 sec	0.9208	357491
3	Goldhill.bmp	258 KB	54 KB	4.78:1	1.877 sec	0.9446	427483

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



	PR_{DDR}				PR_{BRAM}						
	$RP_{Bitsize}$:3416.088 KB		$RP_{Bitsize}$:1306.272 KB		$RP_{Bitsize}$: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)	$T_{latency}$ (s)	$T_{runtime}$ (s)	$T_{latency}$ (s)	$T_{runtime}$ (s)	Samples
4	5.157	5.157	3.75413	3.75413	-	-	-	-	-	-	4096
8	3.418	6.836	2.19865	4.39731	2.285	4.57	1.93353	3.86706	1.6514	3.3028	2048
16	2.549	10.194	1.42036	5.68142	1.5375	6.15	1.28779	5.15115	1.0191	4.0764	1024
32	2.114	16.911	1.03122	8.24978	1.16313	9.305	0.96495	7.71959	0.7029	5.6232	512
64	1.897	30.344	0.83664	13.38619	0.97444	15.591	0.80353	12.8565	0.5448	8.7168	256
128	1.788	57.211	0.73934	23.65878	0.88022	28.167	0.72282	23.1304	0.4658	14.9039	128
256	1.733	110.937	0.69071	44.2053	0.83309	53.318	0.68247	43.6779	0.4262	27.2784	64

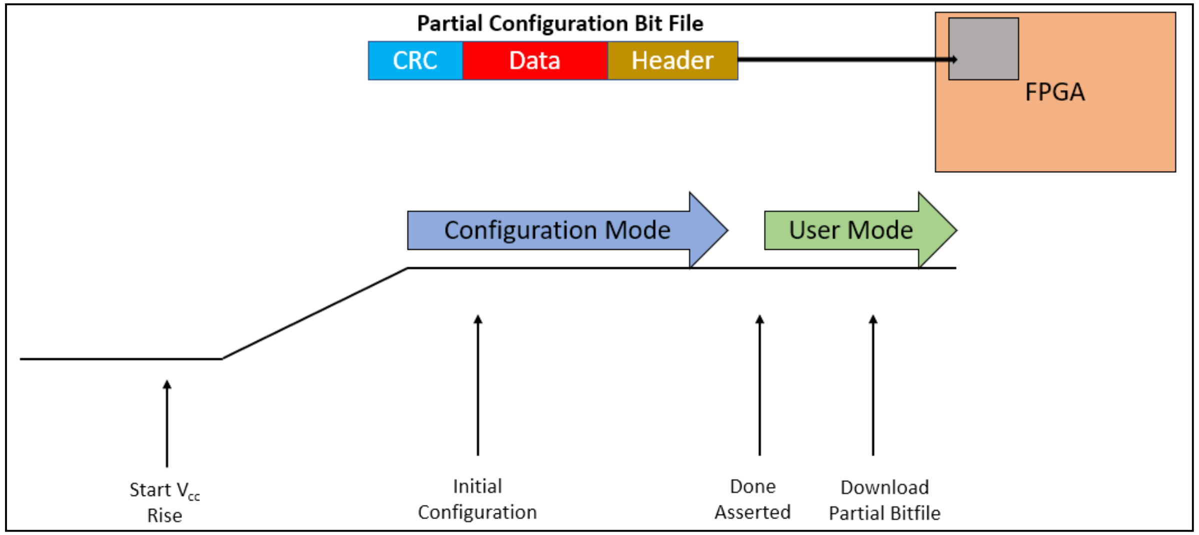
Partial Reconfiguration – Full vs Partial Bitstream



(a)

Figure.2 Configuration Process & Contents of (a) Full and (b) Partial bitstreams

(b)



Partial Reconfiguration – Method to configure Partial Bitstreams

- **Internal Configuration Access Port (ICAP) :**
  - User configuration solutions
  - Requires ICAP controller + Logic to drive the ICAP interface

- **JTAG Port :**
  - Quick Testing or Debug
  - Driven using iMPACT or ChipScope Analyzer

- **Processor Configuration Port (PCAP) :**
  - Configuration mechanism for all Zynq-7000 designs.

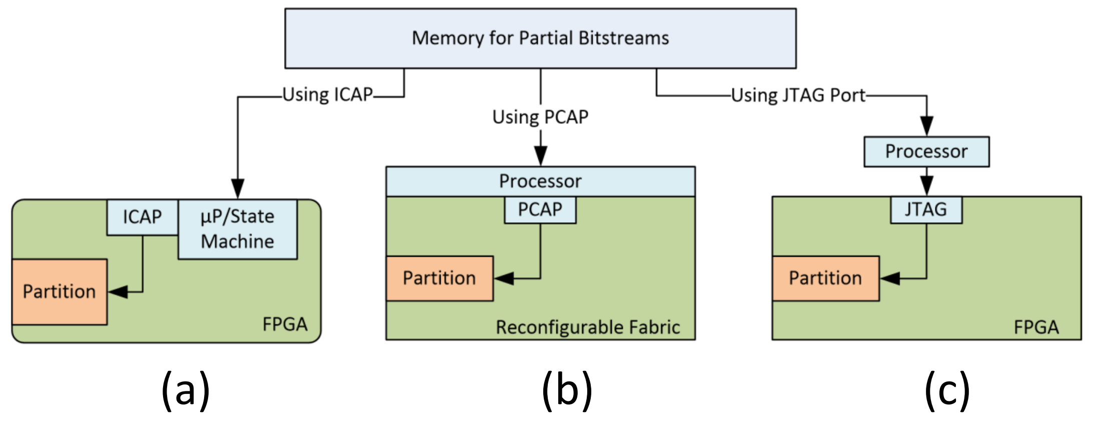


Figure.3 (a) ICAP (b) PCAP (c) JTAG

JPEG Encoder Hardware Accelerators : DCT & Quantization

➢ Discrete Cosine Transform (DCT):

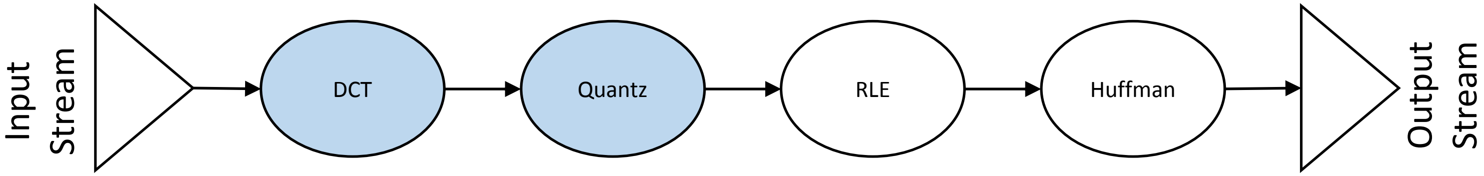
- Converts spatial domain to frequency domain

$$DCT(i,j) = \frac{1}{4}C(i)C(j)\sum_{x=0}^{7}\sum_{y=0}^{7} pixel(x,y)\cos\left(\frac{(2x+1)i\Pi}{16}\right)\cos\left(\frac{(2y+1)j\Pi}{16}\right) \quad (1)\text{ , Where, } C(k) = \frac{1}{\sqrt{2}}\ if\ k = 0\ \&\ C(k) = 1\ otherwise$$

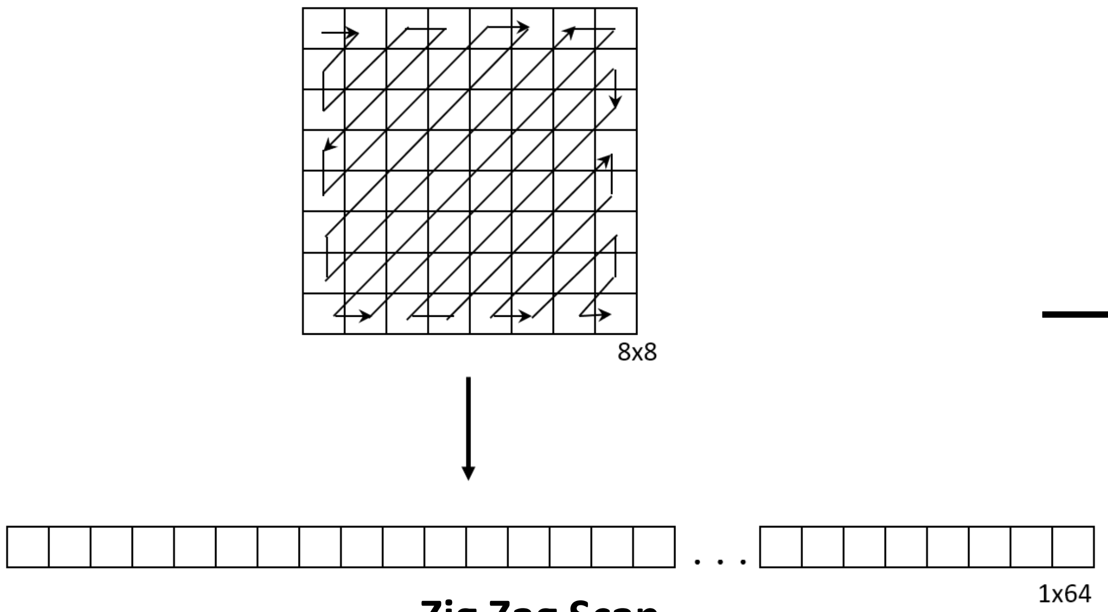
➢ Quantization:

- Dividing transformed image DCT matrix by quantization matrix used and rounding off
- Aims at reducing most of the less important high frequency DCT coefficients to zero

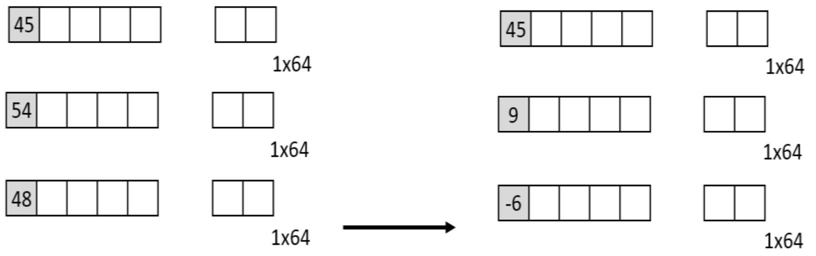
$$DCT_Q(i,j) = Round\left(\frac{DCT(i,j)}{Q(i,j)}\right) \quad (2)$$



JPEG Encoder Hardware Accelerators : RunLength Encoding

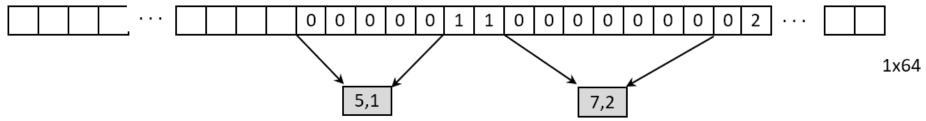


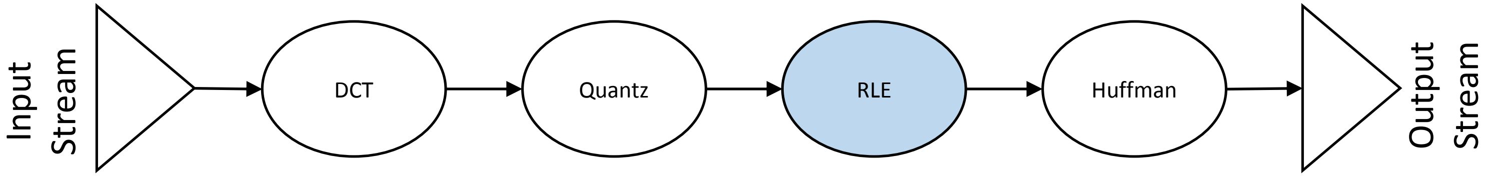
Zig Zag Scan



Differential Pulse Code Modulation (DPCM)

Encoding the difference between current and the previous 8 X 8 block

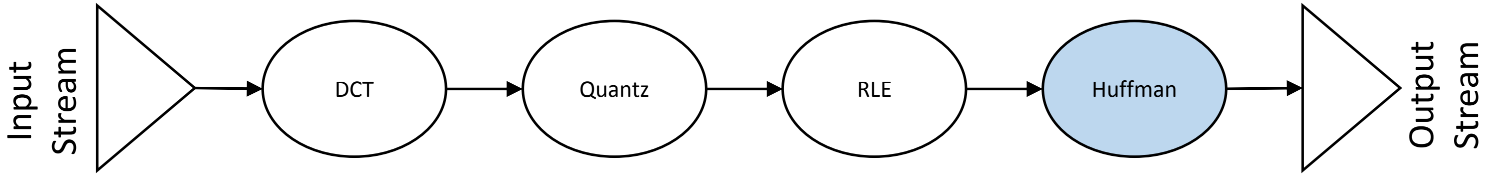




RunLength on AC Components (RLC)

Encodes a series of zeros as a (skip, value) pair

JPEG Encoder Hardware Accelerators : Entropy Coding



DC Components

- DC components are differentially coded as (**SIZE**, **Value**)
- Code for a **Value** is derived from the **Size_and_Value Table (Table.1)**
- Code for a **SIZE** is derived from **Table.2**

Example: If a DC component is 40 and the previous DC component is 48. The difference is -8. Huffman coded as: 1010111

- 0111: The value for representing –8 (Size_and_Value table)
- 101: The size from the same table reads 4, which corresponds to 101 from Table.2

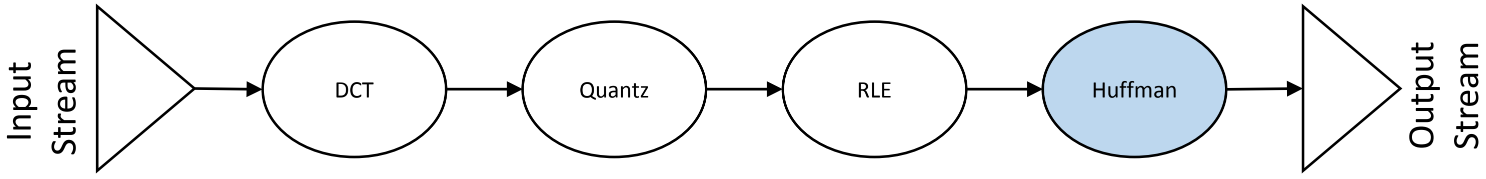
Table.1 Size_and_Value

SIZE	Value	Code
0	0	---
1	-1,1	0,1
2	-3, -2, 2,3	00,01,10,11
3	-7,…, -4, 4,…, 7	000,…, 011, 100,…111
4	-15,…, -8, 8,…, 15	0000,…, 0111, 1000,…, 1111
.		.
.		.
11	-2047,…, -1024, 1024,… 2047	…

Table.2 Huffman Table for DC component SIZE field

SIZE	Code Length	Huffman Code
0	2	00
1	3	010
2	3	011
3	3	100
4	3	101
5	3	110
6	4	1110
7	5	11110
8	6	111110
9	7	1111110
10	8	11111110
11	9	111111110

JPEG Encoder Hardware Accelerators : Entropy Coding



40	12	0	0	0	0	0	0
10	-7	-4	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure.9 Example of 8 X 8 block after quantization

- **AC Components:** Coded as (S1,S2 pairs)
  - **S1:** (**RunLength/SIZE**) , where **RunLength** : length of the consecutive zero values [0..15] & **SIZE** : No. of bits needed to code the *next* nonzero AC component's value
  - **S2: (Value),** where **Value** is the value of the AC component from Table.1

- Zig-Zag order -> 12,10, 1, -7 2 0s, -4, 56 zeros
  - **12**: read as zero 0s,12: (0/4)12 → 10111100

1011: The code for (0/4 from Table.3)
1100: The code for 12 from the Table.1

  - **56 0s**: (0,0) → 1010 (Rest of the components are zeros therefore we simply put the EOB to signify this fact)

Table.3 Huffman Table for AC component SIZE field

Run/ SIZE	Code Length	Code	Run/ SIZE	Code Length	Code
0/0	4	1010	1/1	4	1100
0/1	2	00	1/2	5	11011
0/2	2	01	1/3	7	1111001
0/3	3	100	1/4	9	111110110
0/4	4	1011	1/5	11	11111110110
0/5	5	11010	1/6	16	1111111110000100
0/6	7	1111000	1/7	16	1111111110000101
0/7	8	11111000	1/8	16	1111111110000110
0/8	10	1111110110	1/9	16	1111111110000111
0/9	16	1111111110000010	1/A	16	1111111110001000
0/A	16	1111111110000011	… 15/A	More	Such rows

PR$_{DDR}$ Design Implementation – Floorplan View Post P & R

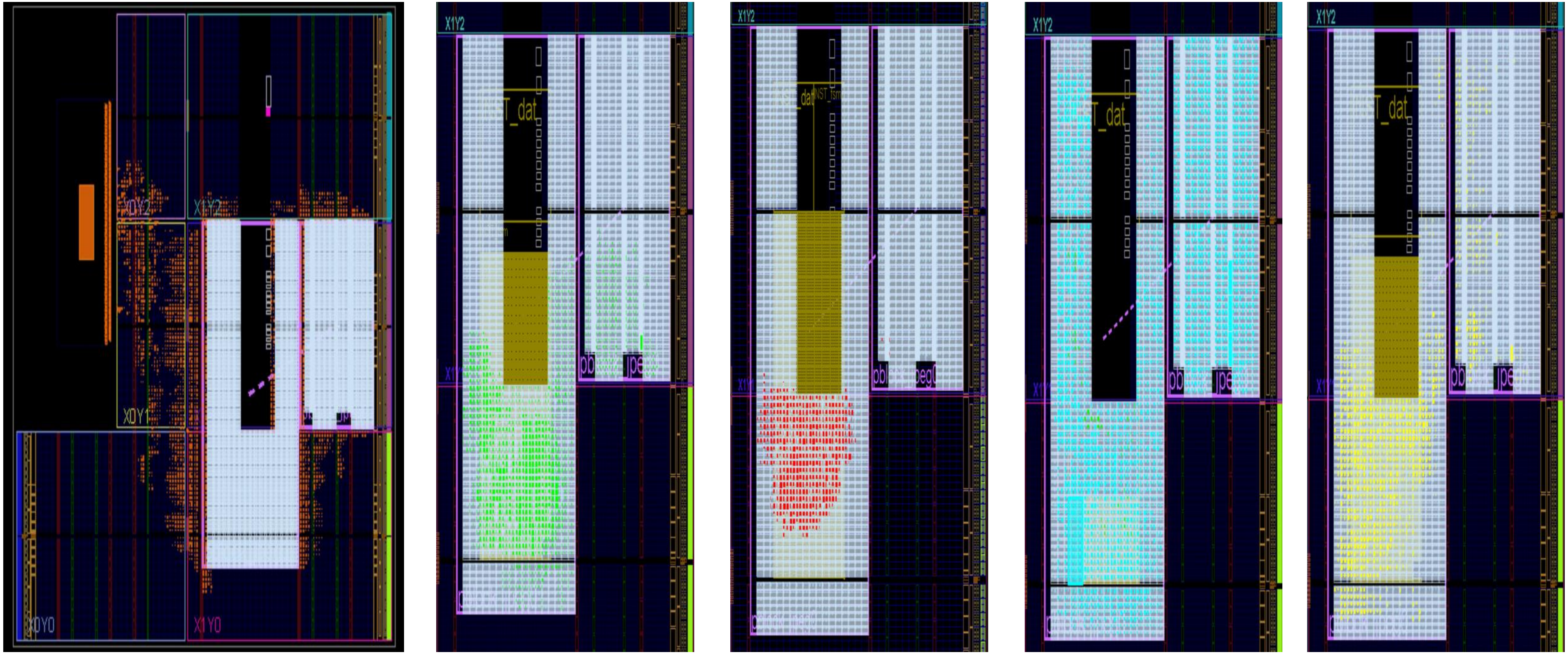
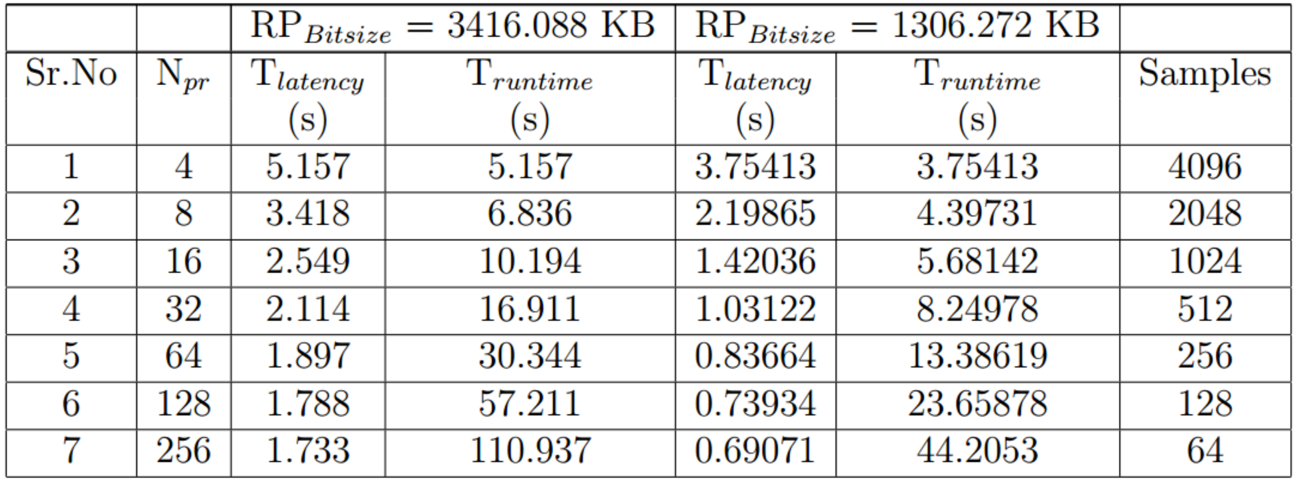


Figure.24 Design Checkpoints after performing Place & Route (a) Static$_{ddr}$.dcp (b) DCT$_{ddr}$.dcp (c) Quantization$_{ddr}$.dcp (d) RLE$_{ddr}$.dcp (e) Huffman$_{ddr}$.dcp ***for RP$_{Bitsize}$ = 1306.272 KB case***

PR_{DDR} Design Implementation – Experimental Results II

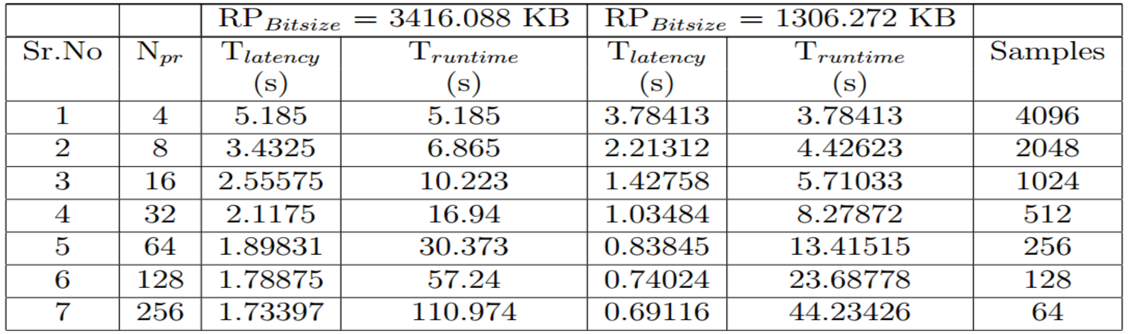
Table.13 JPEG Encoder Results for lena.bmp with PR_{DDR} Design Implementation



		$RP_{Bitsize} = 3416.088$ 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

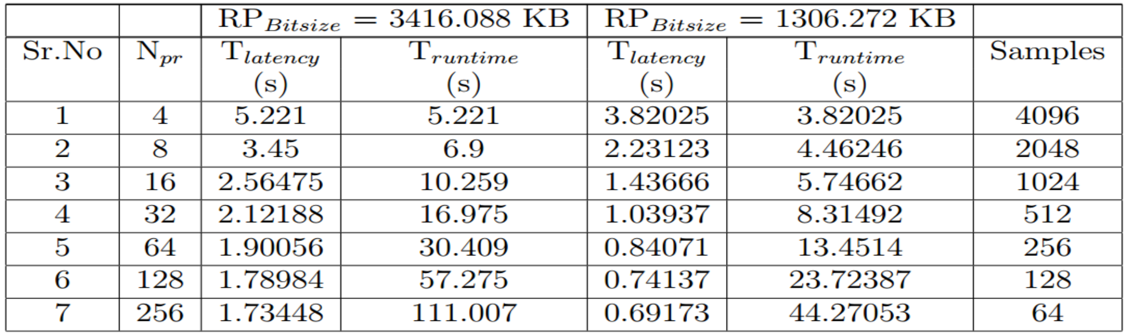
- **Additional experiment** results were tabulated testcase images of peppers.bmp & goldhill.bmp:

Table.14 peppers.bmp



		$RP_{Bitsize} = 3416.088$ 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

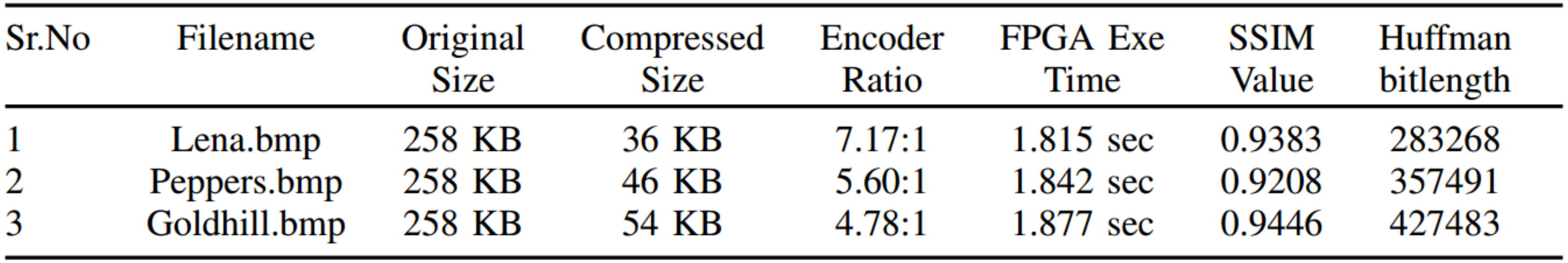
Table.15 goldhill.bmp



		$RP_{Bitsize} = 3416.088$ 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

SPATIAL Design Implementation – Experimental Results

Table.11 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

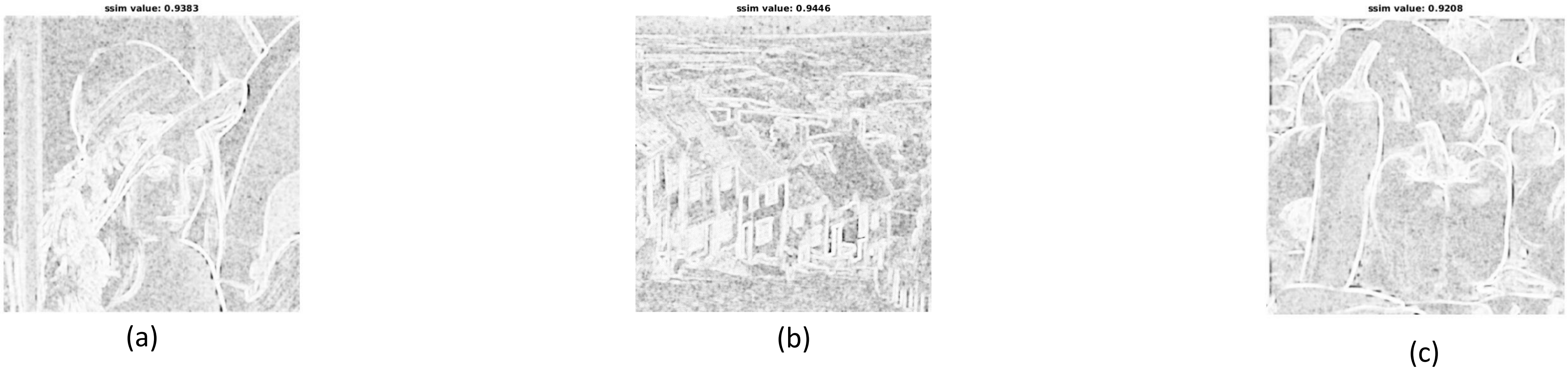
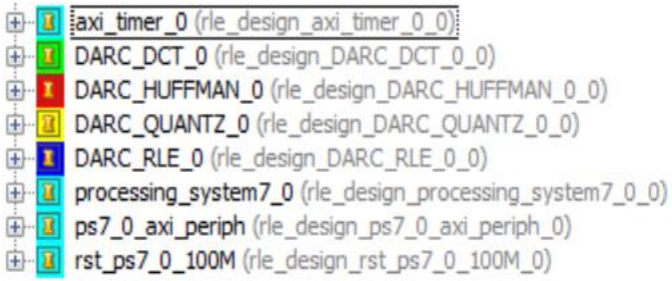


Figure.22 SSIM Maps (a) Lena (b) Goldhill (c) Peppers

SPATIAL Design Implementation - System Implementation and Setup

Table.10 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



axi_timer_0 (rle_design_axi_timer_0_0)
🟩 DARC_DCT_0 (rle_design_DARC_DCT_0_0)
🟥 DARC_HUFFMAN_0 (rle_design_DARC_HUFFMAN_0_0)
🟨 DARC_QUANTZ_0 (rle_design_DARC_QUANTZ_0_0)
🟦 DARC_RLE_0 (rle_design_DARC_RLE_0_0)
processing_system7_0 (rle_design_processing_system7_0_0)
ps7_0_axi_periph (rle_design_ps7_0_axi_periph_0)
rst_ps7_0_100M (rle_design_rst_ps7_0_100M_0)

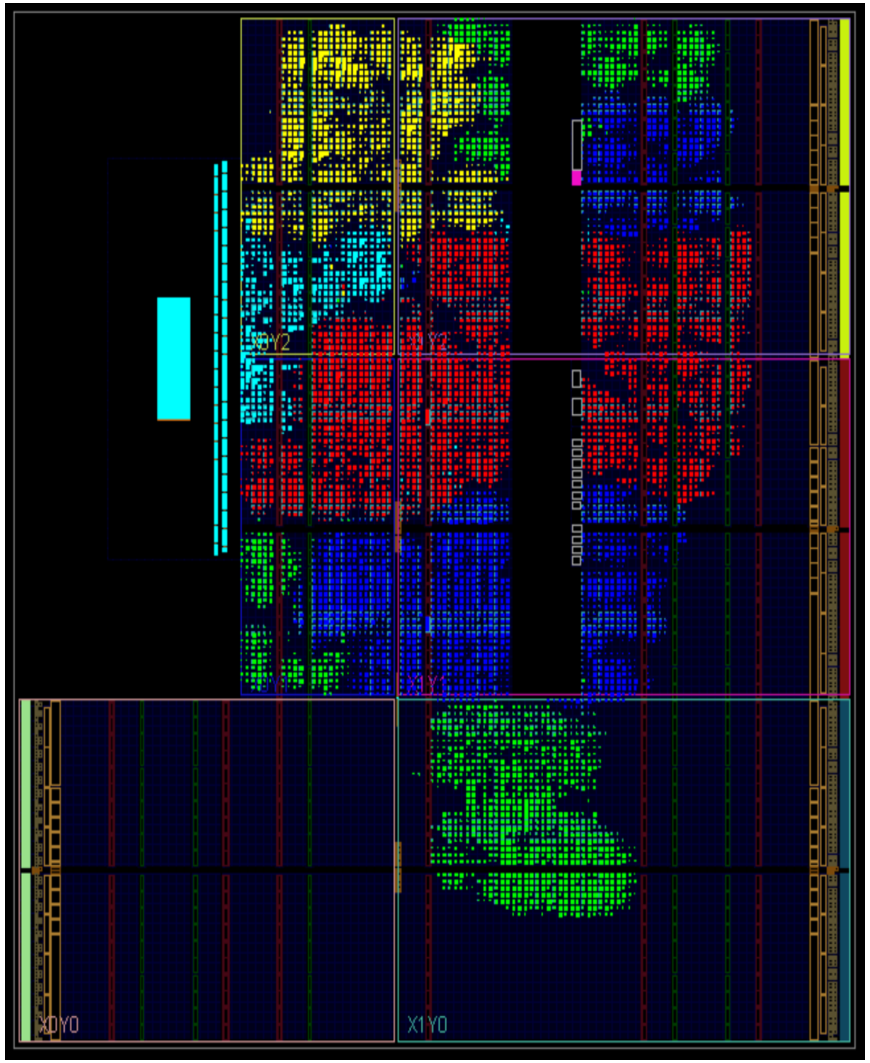


Figure.21 Floorplan View

- Using the Vivado IP Integrator, custom IPs of JPEG Encoder are connected using AXI4 spatially.
- Clock frequency : 50 MHz
- Additional Hardware Resources :
  - SD Card and DDR3 connected to the external interfaces on the Processing Side (PS) of the Zynq FPGA for storing data

PR_{DDR} Design Implementation - System Implementation and Setup

- **2 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).
- DDR3 memory is referenced using **pointers** in user software application
  - **address mappings** provided in the **systems.hdf** file
- In JPEG Encoder Implemented,
  - Max. No of I/O: **RLE RM block** &
  - Max. data-widths of I/O : **Huffman Encoding RM block**

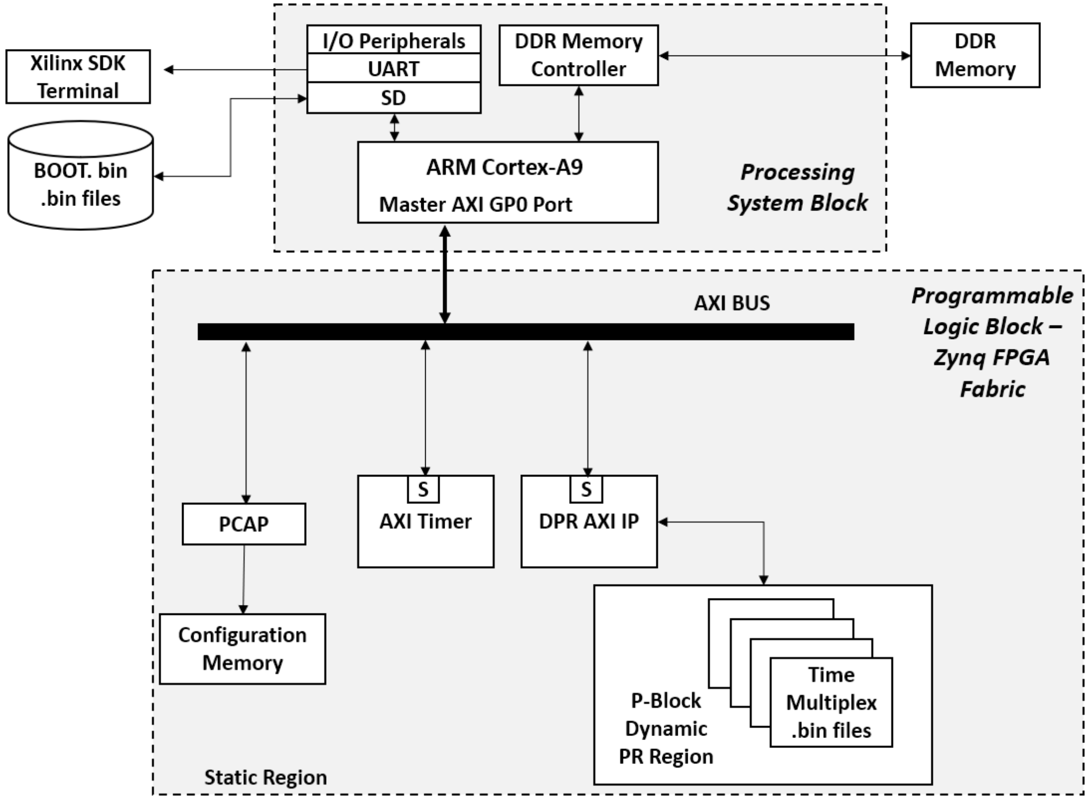


Figure.23 PR_{DDR} design methodology Block Diagram for dataflow computation

Table.12 Utilization Report Post P & R

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 %

Partial Reconfiguration - Ideology & Benefits

- **Reduced Resource and Power Consumption:**
  - Integrating the design into a lower FPGA IC count.
  - Power savings due to Reduction in off-chip communication.
- **Performance Improvements and Flexibility:**
  - Computation capacity of the system adapted at run time
  - Additional resources for speeding up the operation of the kernel
  - More number of kernels to perform the operation in parallel.
- **Improved Fault Tolerance and Dependability:**
  - Safety critical systems - aerospace & defense industries.
- **Self Adapting Hardware Designs:**
  - Adapt to changing operating and environmental conditions based on AI & learning.

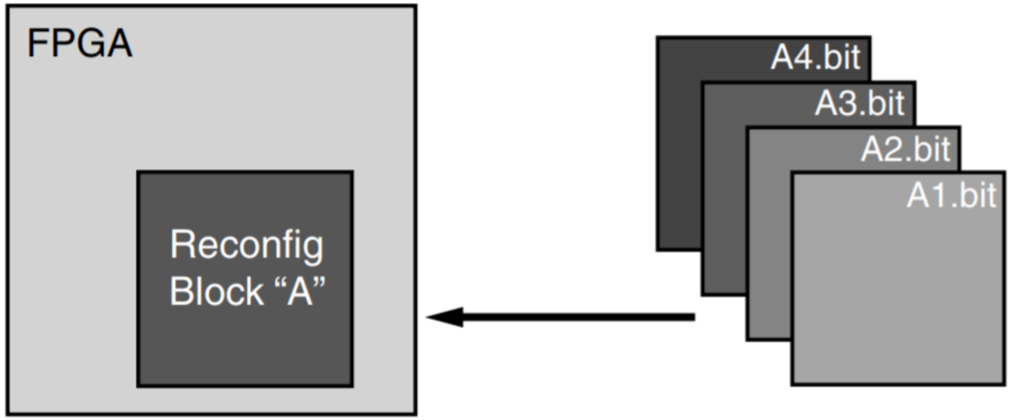


Figure.1 Basic Ideology of PR

❖ ***Partial Reconfiguration Definition–*** Allows the modification of an operating FPGA by downloading Bitfile

PR_{BRAM} Design Implementation – Experimental Results I

- **Runtime and Latency have inverse relationship**
- **Latency and Throughput have linear relationship**

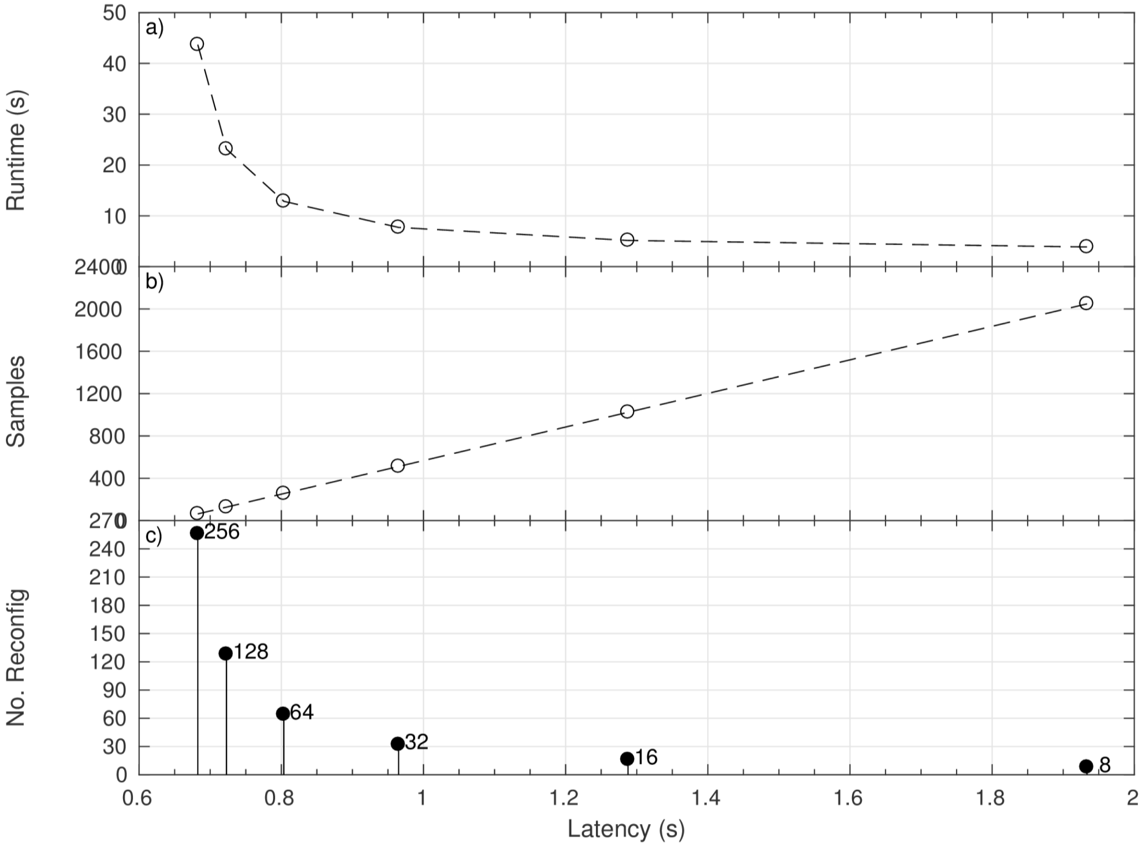


Figure.16 PR_{BRAM} results for lena.bmp testcase with $RP_{Bitsize}$ = 1306.272 KB

Table.7 JPEG Encoder Results for lena.bmp with PR_{BRAM} 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.8 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