Reconfigurable Weight and Output Stationary SIMD 2D Systolic Array AI Accelerator on Cyclone IV GX

Aastha Shah
ars031@ucsd.edu

Ali Alabiad
aalabiad@ucsd.edu

Mohamed Ibrahim
mwibrahim@ucsd.edu

Nathan Chao
nchao@ucsd.edu

Soham Karkhanis
skarkhanis@ucsd.edu

Venkateshwaran Sivaramakrishnan
vsivaramakrishnan@ucsd.edu
Electrical and Computer Engineering, University of California San Diego, La Jolla, USA

***Abstract*—This project develops and evaluates a reconfigurable AI accelerator capable of operating across multiple precision modes and dataflow schemes. All variants were validated against software-generated references. The results show correct functionality across modes and highlight the benefits of combining precision scaling with dataflow reconfigurability in a compact accelerator design.**

I. Introduction

A 2D systolic array is a parallel hardware architecture made up of Processing Elements (PEs) arranged in a two-dimensional grid, where each element performs a multiply-and-accumulate (MAC) operation and passes data to its neighboring PE. Instead of relying on frequent memory access, data flows through the array, where each PE is part of the datapath, allowing each element to reuse inputs as they move across rows and columns. This structure is highly efficient for general matrix matrix multiplication (GEMM) and other linear algebra operations. With appropriate modifications, a 2D systolic array can also support general matrix vector multiplication (GEMV). The regular layout, data reuse, and high degree of parallelism make 2D systolic arrays highly energy-efficient and well-suited for applications such as neural network accelerators, digital signal processing. In this project, we have implemented a 2D systolic array for VGG16.

A 4-bit weight stationary baseline was implemented and verified, then extended to support 2-bit and 4-bit activation modes through SIMD lane reconfiguration. A second extension enabled switching between weight stationary and output stationary operation within a unified processing element. All variants were validated against software-generated references.

II. Base Model

The top-level architecture of the core is shown in Fig. 1. The core consists of the following RTL modules:

- Weight and Activations SRAM
- Partial Sum (PSUM) SRAM
- Controller
- Corelet:
  - MAC Array
  - L0-FIFO (Level 0 - FIFO)
  - I-FIFO (Input FIFO)

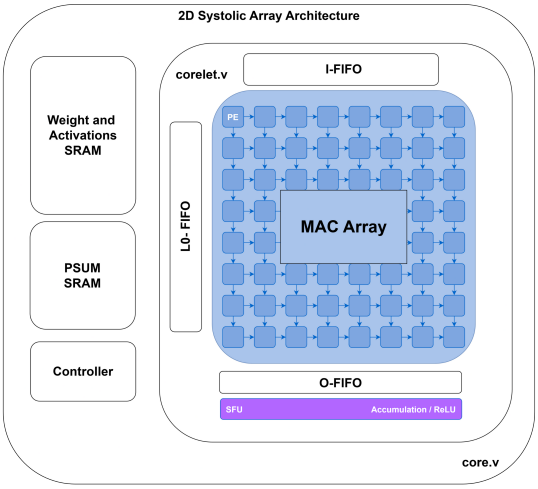

Fig. 1. Top-level Architecture

A. Part 1

A VGG16 based model for image classification was implemented and trained using 2 convolutional layers with ReLU activations followed by max-pooling. However, the last convolutional layer involves an 8-input, 8-output channel layer that was later used for hardware mapping. Training utilized Stochastic Gradient Descent and a step learning rate scheduler that decreased the learning after a fixed number of epochs. To help reduce memory complexity and computation, we trained the model on a 4-bit step-size quantization with trainable parameters allowing for better training.

The Vanilla design was synthesized and mapped to a Cyclone IV GX EP4CGX150DF31I7AD FPGA using Quartus Prime 20.1. At the slow corner of 1200 mV and 100°C, the design achieved a maximum operating frequency of 146.69 MHz with a measured total power of 338.69 mW assuming a 20 percent input activity factor.

For the baseline version of the design, the architecture shown in Fig. 1 was implemented to support the computation of a convolutional layer of the VGG16 network using a 2D systolic array configured in a weight-stationary dataflow. All the components in Fig. 1 were implemented for the baseline design, with the exception of the controller and the I-FIFO,

whose functionality was handled by the top-level testbench. Since this version served as the foundation for all later design iterations, significant care was taken in the design of each of the subcomponents, which were verified using separate component-level testbenches. The description of each subcomponent is as follows:

- **Systolic Array for MAC (Multiply-and-Accumulate):** A 2D systolic array was used to maximize computation throughput and enable efficient data movement. A diagram of the processing element (PE) micro-architecture operating in weight-stationary mode is shown in Fig. 2.
- **L0 FIFO:** This FIFO is used to load both weights and activations into the Weight+Activation SRAM. It consists of eight individual FIFOs, one per systolic array row, and is controlled by an L0 wrapper module through three main control ports: `rd`, `wr`, and `ld_mode`, which enable the required staggering of read and write operations during kernel and activation loading. While the FIFO was initially sized to a depth of 64 to support full activation storage, it was reduced to a depth of 16 in the baseline design, as discussed in Section IV.E.
- **OFIFO:** The output FIFO serves as intermediate storage for partial sums before they are written to the PSUM SRAM. It consists of eight FIFOs, one per systolic array column, each sized to a depth of 64 to support storage of 36 partial-sum entries over one iteration of the k_{ij} computation loop.
- **PSUM SRAM:** The baseline PSUM SRAM was re-parameterized to support a word length of 128 bits, allowing an entire row of systolic array outputs to be written in a single cycle. It was sized to store 512 words to accommodate all partial sums generated during the convolution computation.
- **Weight + Activation SRAM:** This SRAM was re-parameterized to support a word length of 32 bits, enabling the loading of an entire column of weights or activations per cycle. Given that a total of 100 weight and activation values needed to be stored, the SRAM was sized to 128 words.
- **SFU (Special Function Unit):** The SFU consists of an 8-element array of SFU units, one for each systolic array column, that performed either accumulation or RELU on an input PSUM. Each element is controlled by `relu` and `acc` ports, which set the SFU to the required calculation type.

The MAC array, L0 FIFO, SFU, and OFIFO were integrated into a corelet.v module, which was then integrated into a core.v module together with the PSUM and Weight+Activation SRAMs to form the top-level design. The top-level design exposes a 37-bit wide instruction port, which allows the testbench to control weight and activation loading as well as the execution of the convolution computation.

B. Part 2

We implemented and trained a model based on the VGG16 for image classification, using 2 convolutional layers with ReLU activations followed by max-pooling. However, the last convolutional layer involves an 16-input, 16-output channel layer that was later used for hardware mapping. Training utilized Stochastic Gradient Descent and a step learning rate scheduler that decreased the learning after a fixed number of epochs. Contrary to part 1, we utilized 2-bit step-size quantization on activations for training and 4-bit step-size quantization on the weights. The vanilla PE was modified to make it SIMD compatible for 2 bit activation. A single control bit was used to specify 2bit and 4bit activation modes. Two weight registers were used to store the 4 bit weights. Weight loading takes two cycles in 2 bit mode and one cycle in 4 bit mode. Two multipliers are used to generate the product for the MAC in 2 bit mode. These partial products are reused in the 4 bit mode to generate the product.

This model was verified using weight and activation data generated from the original 4 bit model, as well as the 2 bit model.

TABLE I
FPGA SYNTHESIS AND POWER RESULTS FOR VANILLA DESIGN

Metric	Measured Result
FPGA Device	Cyclone IV GX
Slow Corner Fmax (1200 mV, 100C)	146.69 MHz
Input Activity Factor	20%
Core Dynamic Power	43.27 mW
Core Static Power	119.59 mW
Total Thermal Power	338.69 mW
I/O Thermal Power	175.82 mW
Worst Case Slack	-5.817 ns
Total Logic Elements	17,416
Combinational ALUTs	12,391
Total Registers	12,276
Registers Using Clock Enable	11,817
Registers Using Sync Clear	1,269
Registers Using Sync Load	192
I/O Pins	427
Slow Corner Fmax (1200 mV, -40C)	168.21 MHz

C. Part 3

For part 3, the PE is made reconfigurable for weight stationary and output stationary mode. An I-FIFO is added to the north of the mac array to load weights in the output stationary mode. An extra control bit is added to specify operating mode. In the output stationary mode, there are two phases of operation. In the execution phase, weights flow from north to south and activations flow from west to east, in a diagonal manner. The psum register is used to hold the MAC output, and is fed back into the MAC for accumulation. In the flush stage, all PEs transfer psums in north to south fashion. Each psum represents an output index oij for a kernel.

III. SOFTWARE ALPHAS

A. Hybrid Scheduler

Hybrid Scheduler consists of an increasing linear rate scheduler followed by Cosine Annealing. The increase in learning rate allows for rapid learning, and the Cosine Annealing helps

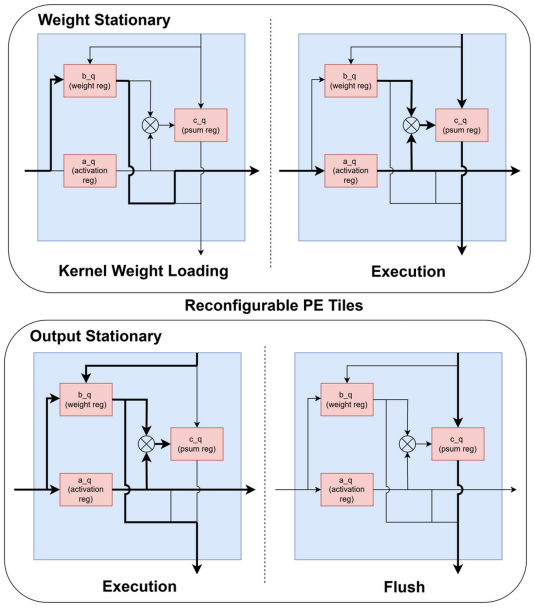

Fig. 2. Reconfigurable Processing Element

the model avoid local minimas, leading to better generalizations. When compared to the base scheduler, we note that it achieved a larger increase in accuracy.

B. 4-to-2 Bit Model Compression via Retraining

Retraining a model with 4-bit activations on 2-bit quantization provides a more accurate model than training the model from scratch. We compared the results of both methods and recorded the results in the table. We theorize that having the weights begin retraining at a good starting point allows it to better adapt to the 2-bit activations compared to starting with randomly initialized weights.

C. Orchid - Optimizer using Orthogonalization

For weight matrices of two or more dimensions, we orthogonalize the momentum matrix using Newton Schulz approximation, allowing for a more stable, better convergent training while decreasing the memory needed to update parameters. This also led to better performances during pruning.
Orchid draws inspiration from Keller Jordan's Muon optimizer which uses the Newton-Schulz which is motivated in scaling the gradient's "rarer directions" while also reducing the number of variables needed for calculation per matrix. The Newton-Schulz algorithm is as follows, given $a, b, c \in \mathbb{R}$:

$$X_t = aX_{t-1} + b(X_{t-1}X_{t-1}^T)X_{t-1} + c(X_{t-1}X_{t-1}^T)^2X_{t-1} \tag{1}$$

We note that this process is done for a fixed number of iterations (in our case, we did 5). We also note that through empirical tests done by Keller Jordan, we use $a = 3.4445, b = -4.7750, c = 2.0315$.

The parameter update is as follows, given Gradient matrix, G_t for parameters θ:

$$\begin{aligned} B_t &= G_t + \mu B_{t-1} \\ O_t &= \text{NewtonSchulz}(B_t) \\ \theta_t &= \theta_{t-1} - \eta O_t \end{aligned} \tag{2}$$

We note that Orchid only operates the convolutional parameters; scalar and vector parameters undergo regular stochastic gradient descent.

D. Mixed Pruning

We utilized a mixture of structured and unstructured pruning on select layers achieving up to 70% sparsity while preserving the accuracy. We skip the first two convolution layers and the last convolution layer before the linear layer since they are beneficial to capture features directly from the image and project the learned features to the classifier head. We hypothesized that pruning such layers might lead to decreases in model accuracy. We utilized structured pruning on layers whose number of output channels equals to that of the input and unstructured pruning on all other remaining layers. We note that without the Orchid optimizer, the average sparsity could not exceed 50% without performance degradation while Orchid allowed up to 70% which leads us to infer that the orthogonalization process of the optimizer is more suitable to sparsity.

E. Quantization using Rotation Matrices

We utilize Hadamard Matrices on the activations to preserve outliers in the data. This is done by projecting the activations to a different linear space before quantization then reverting the quantized matrix back with an inverse of that transformation. A Hadamard matrix is a square matrix $\mathbb{R}^n \times \mathbb{R}^n$, where n is 1,2, or an order of 4 with mutually orthogonal columns. One key feature of the Hadamard matrix, H is that $\frac{1}{\sqrt{n}}H^{-1} = \sqrt{n}H^T$ thus ensuring an invertible matrix without the need for learning the rotation matrices. Therefore, the quantization process is as follows; given an activation matrix,X, Hadamard matrix, H and a clipping factor,α, the quantized activations, X_q:

$$X_q = \text{UQ}_\alpha(\frac{1}{\sqrt{n}}XH)\sqrt{n}H^T \tag{3}$$

Where UQ is the unit-step quantization function parametrized by clipping parameter α. We note however that, for ease of computation, we constructed the Hadamard matrix as a block-diagonal of smaller Hadamard matrices (orders of 2 or 4).

TABLE II
MODEL SPARSITY AND TEST ACCURACY

Model	Avg. Sparsity (%)	Test Acc. (%)
4-bit (Base)	23.36	92.31
4-bit (Base w/out hybrid sched)	35.4	86.58
4-bit (w/Orchid and hybrid sched)	42.35	94.15
4-bit (Pruned) (w/Orchid and hybrid sched)	70.01	90.81
4-bit (Pruned) (w/SGD and hybrid sched)	50.00	91.76
2-bit (Model Compression)	26.50	89.92
2-bit (Pretrain)	38.73	84.92
2-bit (Pretrain w/Orchid)	31.77	89.81
4-bit (w/Rotation Matrices)	21.67	88.90

IV. HARDWARE ALPHAS

A. Weight-Stationary Controller

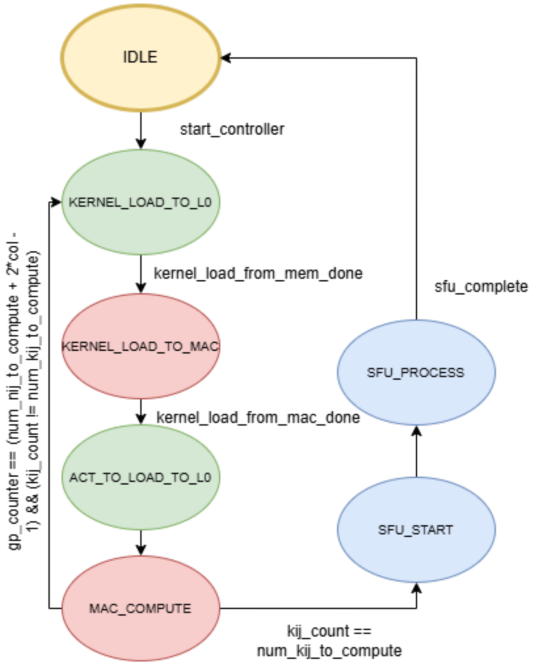

Fig. 3. WS Controller State Transition Diagram

For this alpha version, the baseline design was adapted to be controlled by a dedicated controller module integrated within the top-level core, as shown in Fig. 1. The goal of this enhancement was to enable the core to execute the complete weight-stationary convolution computation with minimal external control signaling. Accordingly, the controller-enhanced design is intended to be operated as follows:

1) Load all weights and activations into the Weight+Activation SRAM.
2) Pulse the `start_controller` signal.
3) Wait for computation to complete, as indicated by the `core_busy` signal deasserting. Upon completion, the controller writes the output values to the first $oijo_{ij}oij$ addresses of the PSUM SRAM.
4) Read the output values from the PSUM SRAM for verification.

To support this control model, the top-level core interface was simplified by reducing the instruction bus width from 37 bits to 13 bits. This includes:

- 9 bits for SRAM addressing
- 2 bits for SRAM read/write control
- 1 bit for the `start_controller` signal
- 1 bit for a debug mode used during early controller development

The controller design is based on the FSM shown in Fig. 3. The controller automatically performs the following sequence of operations:

1) Load weights from the Weight+Activation SRAM into the L0 FIFO.
2) Load weights from the L0 FIFO into the 2D systolic array.
3) Load activations from the Weight+Activation SRAM into the L0 FIFO.
4) Load activations from the L0 FIFO into the 2D systolic array and enable execution.
5) Write intermediate partial sums from the OFIFO into the PSUM SRAM in parallel.
6) Repeat steps 1–5 for all k_{ij} iterations.
7) Enable the SFU sub-controller, which:
   - Computes the required PSUM indices for each output and reads the corresponding PSUM entries.
   - Accumulates the required PSUMs and applies the ReLU operation.
   - Performs these operations in a pipelined manner to reduce SFU latency.

These operations are coordinated using a General-Purpose (GP) counter and output-state logic, as illustrated in Fig. 4.

The primary advantages of this alpha version are:

- A significant simplification of the usage model for the weight-stationary systolic array.
- A reduction in total computation latency from 1229 cycles in the baseline design to 941 cycles in the controller-enhanced version.

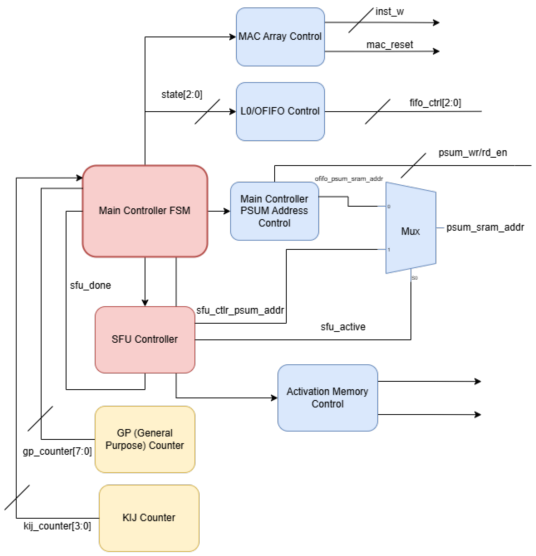

Fig. 4. WS Controller Block Diagram

B. Enhanced Verification

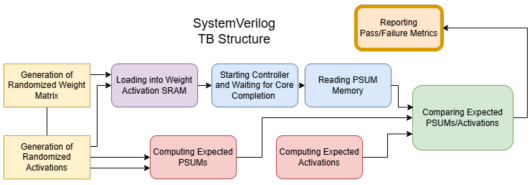

Fig. 5. Enhanced Verification Block Diagram

To increase confidence in both the controller-enhanced design and the original corelet, comprehensive end-to-end verification was performed. A SystemVerilog testbench was developed to automate the following verification tasks:

- Randomized generation of 4-bit weights and activations matching the dimensions of the convolution layer input.
- Automated computation of the expected outputs based on the generated weights and activations.
- Automated verification of correct kernel loading during the weight-loading phase.
- Comparison of the expected and actual outputs, followed by generation of an aggregate error metric.

The overall verification flow is illustrated in Fig. 5.

C. PCIe IP Integration

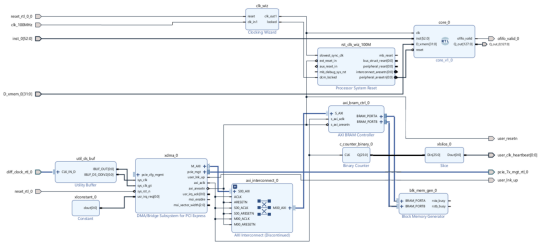

Fig. 6. PCIE IP Integration Postmapping

The PCIe interconnect IP is integrated with the Corelet of the reconfigurable systolic array. The design envisions a multi-core architecture in which users or servers can configure multiple Corelets and communicate efficiently via the PCIe hub/interface. In this implementation, the input activations are directly connected to the Corelet, while the weights are loaded by the user or server onto two FPGA SRAM modules (Activation/Weight SRAM and PSUM SRAM) instantiated externally as hard macros. The PCIe DMA engine interfaces with a subordinate AXI interface on its Master port, which relays the data to the corresponding SRAMs with the appropriate control signals. Similarly, the AXI interface can also read data from the PSUM SRAM when required.

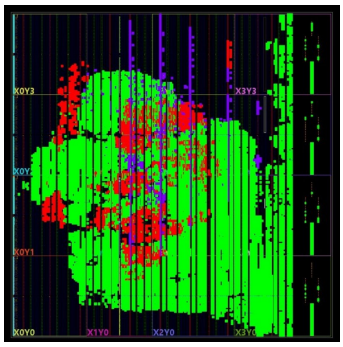

Fig. 7. PCIE IP Place and Route

D. Power Saving - Clock and Data Gating

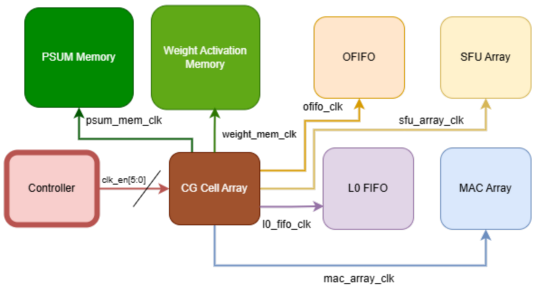

Fig. 8. Clock Gating Block Diagram

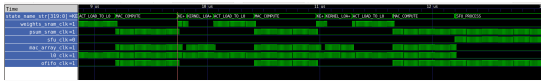

Fig. 9. Clock Gating Verification

Data gating was introduced to exploit activation sparsity resulting from mixed pruning. A 1 bit control register, dgate, was integrated into each mac_tile. The signal was asserted by default and deasserted when the mac_tile input was zero. The dgate signal masked the input operand a through a bitwise AND operation, thereby suppressing unnecessary computation for zero activations.

Clock gating was used to reduce dynamic power by disabling clocks to idle blocks, as shown in Fig. 9. An array

of clock gating cells, shown in Fig. 8, was instantiated to implement this functionality. For example, the SFU clock was enabled only during the SFU_PROCESS state, thereby reducing the switching activity factor and overall dynamic power consumption.

E. Optimized FIFO Length and Loading Scheme

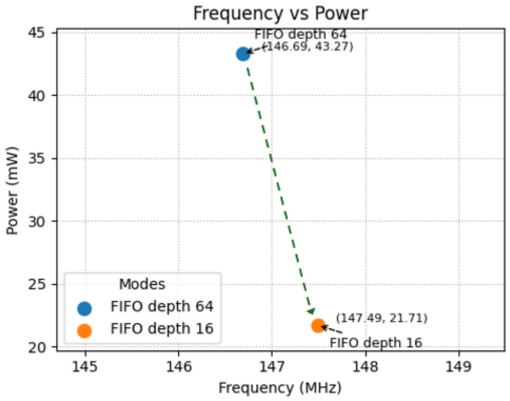

Fig. 10. Reduced FIFO Length in Vanilla Version

In the original dataflow, operands were first loaded into the FIFO and subsequently transferred to the 2D systolic MAC array. By modifying the testbench to enable parallel loading of the FIFO and MAC array, the FIFO depth was reduced from 64 to 16. As shown in Fig. 10, a FIFO depth of 64 yielded a baseline operating frequency of 146.69 MHz with a power consumption of 43.27 mW. With parallel data loading, the operating frequency increased to 147.49 MHz, while power consumption decreased to 21.71 mW. Although the FIFO depth could theoretically be reduced further to 8, a depth of 16 was selected as the nearest power of two to provide robustness against timing uncertainties arising from asynchronous events. This optimization was incorporated as a +Alpha enhancement to the Vanilla version.

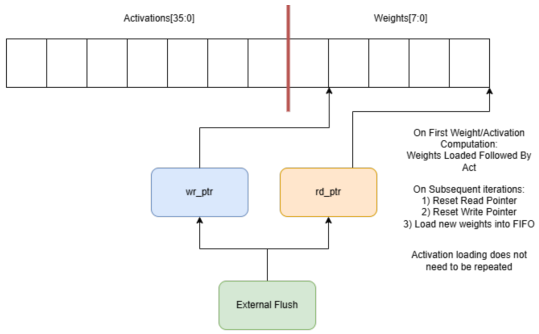

Fig. 11. Optimized FIFO Length and Modified Parallel Loading

The FIFO length was optimized to reduce architectural latency. As illustrated in Fig. 11, the FIFO was partitioned to store activations and weights concurrently. During the initial phase, when the FIFO is empty, weights are loaded first, followed by activations. In subsequent iterations, corresponding activation and weight pairs (e.g., (X0, W0)) are loaded in parallel using the read and write pointers. After each iteration, the pointers are reset and new weights are loaded. This approach enables activations to be loaded only once, eliminating repeated activation loading required in the baseline design. As a result, the estimated latency was reduced from 1160 cycles to 941 cycles.