pRISC*
Parallel Processing Accelerator for Video and AI Applications

Ypologist

Executive summary

pRISC is a parallel processing architecture designed to accelerate tensor-dominated workloads in video and transformer-based AI inference, with a focus on edge deployment under strict power and area constraints.

The architecture targets General Matrix Multiplication (GEMM) and related tensor operations that dominate modern transformer inference pipelines. Unlike conventional GPU-based solutions, pRISC is optimized for predictable dataflow and structured computation, enabling improved energy efficiency and silicon utilization in edge environments.

An FPGA-based prototype of a heterogeneous system (host CPU + pRISC accelerator) has been implemented and used to validate architectural behavior and measure cycle-level performance for representative workloads.

Problem

Edge AI deployment is constrained by:

- Power efficiency limits of GPU-based solutions
- Under utilization of off-the-shelf parallel architectures for structured tensor workloads
- Latency and bandwidth constraints in real-time inference scenarios
- Software complexity in mapping high-level models to heterogeneous hardware

While ASIC accelerators improve efficiency, they lack flexibility and require long development cycles, which is misaligned with the rapid evolution of AI models.

Solution

pRISC introduces a parallel architecture optimized for tensor computation with the following design principles:

- Deterministic dataflow execution replacing cache-driven execution models
- Buffer-based memory organization aligned with predictable access patterns in GEMM and transformer layers
- Linear compute topology enabling efficient mapping of tensor operations
- Native support for structured sparsity and search operations
- Tight coupling with host CPU, exposing the accelerator as a callable function layer

This approach reduces control overhead and improves compute utilization for inference workloads dominated by linear algebra operations.

*pRISC stands for "parallel RISC".

Technical Differentiation

Compared to GPUs and fixed-function ASICs:

- Versus GPUs (e.g., NVIDIA TU117):
  - Eliminates cache inefficiencies for predictable workloads
  - Reduces control divergence and scheduling overhead
  - Targets higher utilization on dense and structured-sparse GEMM
- Versus ASIC accelerators:
  - Retains programmability through a software abstraction layer
  - Avoids full redesign cycles as model architectures evolve

At the architectural level, pRISC focuses on:

- Dataflow-oriented execution rather than thread scheduling
- Explicit data movement instead of implicit cache coherence
- Mapping tensor operations onto a regular compute fabric

Prototype and Validation

A working FPGA prototype has been developed:

- Implements a full heterogeneous system (host + accelerator)
- Used for cycle-accurate measurement of GEMM and related kernels
- Enables architectural exploration and validation of execution model

pLreliminary results indicate significant improvements for pRISC (evaluated by simulation) compared to GPU (nVIDIA's TU117, for which we used data from the literature[1]) in terms of:

- **Energy efficiency**, expressed by ratio

$$\frac{E^{pRISC}_{1024x1024MM}}{E^{GPU}_{1024x1024MM}} = 4.34 \div 12.4$$

  evaluated for computing 1024×1024 GEMM (see Appendix B.3)

- **Silicon utilization** (area efficiency), expressed by ratio

$$\frac{A^{pRISC}_{1024x1024MM}}{A^{GPU}_{1024x1024MM}} = 3.1$$

  evaluated for computing 1024×1024 GEMM (see Appendix B.4)

- **Execution cycles for large GEMM workloads** expressed as the ratio of architectural alcceleration

$$\frac{\alpha^{1024x1024MM}_{pRISC(p)}}{\alpha^{1024x1024MM}_{GPU(p)}} \simeq 5$$

  evaluated for computing 1024×1024 GEMM by a p-cell pRISC and a p-cell GPU (see Appendix B.2)

These results are currently based on FPGA implementation and synthesis estimates and will require silicon validation.

[1] At: `https://arxiv.org/abs/2507.19723` published in 2025.

Software Model

pRISC is exposed as a hardware-accelerated function layer compatible with existing AI workflows:

- Integration path: ONNX / PyTorch → operator mapping → pRISC hardware library
- Initial focus:
  - GEMM
  - Transformer primitives (attention, projection layers)
  - Sparse operations

The objective is to minimize friction for developers by allowing existing models to target the accelerator without rewriting algorithms at the hardware level.

Target Market

Initial focus: **Edge AI inference systems**, including:

- Industrial inspection
- Embedded vision systems
- Telecom edge nodes
- Low-power AI appliances

These environments are characterized by:

- Tight power envelopes
- Predictable workloads
- Increasing use of transformer-based models

Go-To-Market Strategy

1. Phase 1 — SDK and Kernel Development
   - Deliver a minimal SDK supporting key ONNX operators
   - Demonstrate end-to-end inference on representative workloads
2. Phase 2 — Pilot Deployments
   - Partner with early adopters in edge AI domains
   - Validate performance under real deployment constraints
3. Phase 3 — Commercialization
   - Licensing of IP cores for integration into SoCs
   - Potential system-level offerings with partners

Business Model

- Primary: IP licensing to semiconductor and system companies
- Secondary (intermediate phase):
  - Evaluation platforms (FPGA-based)
  - SDK licensing and support

Funding Requirements

Initial funding will be used to:

- Develop a production-grade SDK
- Expand operator coverage for transformer workloads
- Improve toolchain (model mapping and optimization)
- Prepare for ASIC implementation and silicon validation

Key Risks and Mitigation

- Benchmark validation risk
  → Mitigation: standardized comparisons and real workload measurements
- Software adoption risk
  → Mitigation: ONNX compatibility and minimal integration friction
- Hardware execution risk (ASIC phase)
  → Mitigation: FPGA-based validation and staged development

Summary

pRISC targets a well-defined gap between general-purpose GPUs and rigid ASIC accelerators by focusing on:

- Efficient execution of tensor-dominated workloads
- Reduced energy and area cost for edge inference
- A software-accessible hardware acceleration model

The current stage demonstrates architectural feasibility; the next phase is focused on software maturity and real-world validation.

APPENDIXES

A General Presentation

Our *pRISC*[2] is the accelerator component of a heterogenous computing system (see Figure 1). It is designed to execute intense parts of an application (the most time, energy and area consuming) while the complex part is executed by the HOST computer part of the system. *pRISC* is a ***configurable & parameterizable*** many-cell computational system.

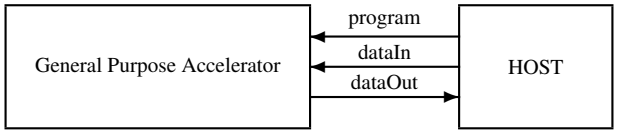

Figure 1: Heterogenous system.

Why such a solution? Many of the currently used applications require particularly intensive calculations in the sense of the simplicity of the description accompanied by prohibitive execution times and high consuming energy. A solution is required that accelerates the computation and minimizes the energy consumed. The solution proposed by the scientific community and accepted by the corporate space is the heterogeneous computing.

Where does our solution come in? We propose a general-purpose accelerator, *pRISC*, because the main weakness of the current parallel solution for intense computations are due to:

- the use of *ad-hoc* hardware structures designed more from geometrical considerations than from the consideration of computational aspects
- using specific accelerators to solve general-purpose computational issues (the oxymoronic example of GPGPU is typical).

The structural and architectural inadequacy of current solutions leads to high consumption of energy and silicon area (the manufacturing price of a chip increases super-linearly with the area).

pRISC **programming** is carried out at several levels as follows:

- level 1: in assembly language at the accelerator level for the development of function libraries using data structures limited by the hardware structure, called elementary libraries,
- level 2: in a high-level language, using elementary libraries to develop libraries of dedicated or user-defined functions,
- level 3: in a high-level language, using the libraries developed on level 2, for developing applications.

In Appendix 1 is shown a subset of functions developed at the level 1. The functions are used for writing program for matrix-matrix multiplication at the level 2 (see Appendix 2). Programming level 1 is accessible to internal developers, while levels 2 and 3 are also accessible to users.

[2] *pRISC* stands for *parallel RISC*. It was coined by Jim Peek who led the team of VLSI designers of the RISC I chip as a master's student of Professor David Patterson at the University of Berkeley (`https://www.linkedin.com/in/jim-peek-a233b8/` and `https://www.computerhistory.org/revolution/digital-logic/12/286/1593`).

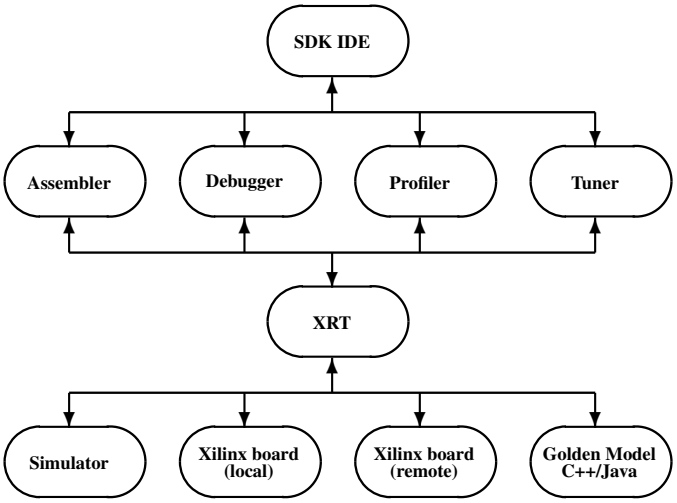

Figure 2: Software architecture: component view

The software components built on top of *pRISC* are represented in Figure 2[3]. While the flow view of the software architecture is represented in Figure 3[4].

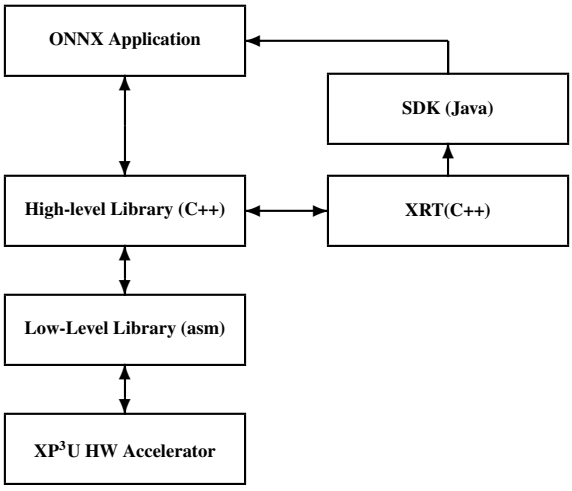

Figure 3: Software architecture: flow view

pRISC **Structure** (see Figure 4) consists of:

[3]SDK: Software Development Kit; IDE: Integrated Development Environment; XRT: *pRISC* Run-Time environment.

[4]ONNX: Open Neural Network Exchange, is an open format built to represent machine learning models. ONNX defines a common set of operators - the building blocks of machine learning and deep learning models - and a common file format to enable AI developers to use models with a variety of frameworks, tools, runtimes, and compilers.

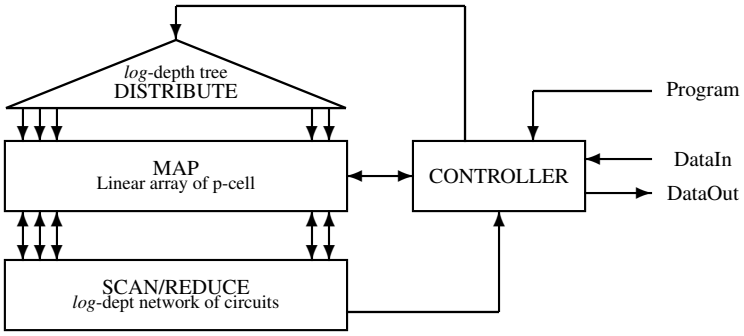

Figure 4: *pRISC*.

- MAP: a linear array of p cells, each with its data memory and an execution unit
- CONTROLLER: which sends an instruction in each clock cycle to be executed in the active cells of the MAP
- a distribution pipeline network of logarithmic depth
- a REDUCE network of logarithmic depth that takes a vector from the MAP and sends a scalar to the CONTROLLER (sum, min, max, ...)
- a logarithmic depth SCAN network that takes a vector from the MAP and sends a vector back to the MAP (permutation, prefix, ...)

CONTROL receives the program from HOST and exchanges data with the HOST's memory.

The entire structure is parameterizable (the size of the word with which the MAP execution units operate, the size of the local memories in the MAP cell, the number of cells in the MAP) and configurable (the SCAN and REDUCE functions, floating point operation, ...). Our solution does not require special technologies to achieve the performance we claim. The advantages of our solution come from organizational and architectural improvements only.

History: several versions of *pRISC* have been implemented in silicon in the first decade of the century in a Silicon Valley startup (see `http://users.dcae.pub.ro/~gstefan/2ndLevel/connex.html`). The 90 nm version is represented in Figure 5. The last one, a 65nm version was used to implement the frame rate conversion function for dual HD video stream. At the current stage we have an FPGA implementation that is programmed in assembly language (working prototype, on PYNQ-Z2 development board, for p = 128).

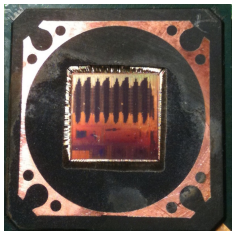

Figure 5: The 90 nm version of CA1024, a many-core of 1024-cell *pRISC* (16-bit execution unit, 0.5KB local memory per cell).

August 20–22, 2006: the BA1024 chip was presented in Memorial Auditorium of Stanford University in *Hot Chips: A Symposium on High Performance Chips*.

July 2007: the first real & complex application (HDTV postprocessing for frame-rate conversion) running on BA1024 was presented in Department of Devices, Circuits and Architectures of Politehnica University of Bucharest (the software team coordinated by Bogdan Mitu, Marius Stoian, and Radu Weiss)

Architectural perfromances of our proposal were made by investigating the calculation of computationally intensive functions involved in current applications such as: AI, automotive, bio-computing, DSP, etc.

The architectural speedup of a parallel computing structure, $\alpha^{function}_{parallelStructure}$, is the ratio between the number of clock cycles in which a certain function is executed on a single-core structure and the number of clock cycles in which the same function is executed on the evaluated parallel machine. The architectural acceleration for the following functions was determined through simulation for pRISC:

- GEMM: $\alpha^{GEMM}_{pRISC} =\sim 3.62p$, obtained by running the program for GEMM on our system and compared with measurements reported in [1]
- FFT: $\alpha^{FFT}_{pRISC} =\sim 1.5p$
- convolution: $\alpha^{convolution}_{pRISC} =\sim 2.1p$

where p is the number of processing cores in pRISC.

Where do these improvements come from? It can be partially explained by:

- the avoidance of using caches due to the predictability of the information flow
- the sufficiently large local memory distributed in each of the p cells of the MAP structure
- the additional parallelism introduced by the concurrent operation of the CONTROLler unit, the MAP network and the REDUCE network.

Stage & future work is summarized in Figure 6.

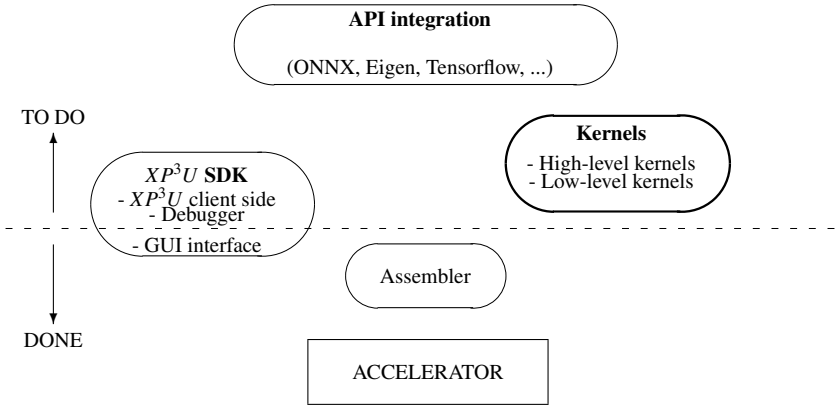

Figure 6: Stage & future work.

B Simulation for matrix multiplication on *pRISC*

B.1 Assembly code

Assembly code for dense matrix multiplication on *pRISC* represents an example of code written for *pRISC* that illustrates how to program in assembler for a frequently used program: matrix multiplication.

The assembly code for multiplying square matrices is listed below. It includes the transfer of the two matrices from the host processor, the transpose of the second matrix, and the transfer back of the result into the host's memory. The total execution time and the execution time per the element of the result matrix are listed in Table B.1, where:

- $T_{transfer}(p)$: the total transfer time for two $p \times p$ matrix loads and one $p \times p$ matrix store
- $T_{transpose}(p)$: the transpose time for a $p \times p$ matrix
- $T_{multiply}(p)$: the multiplication time for multiplying the first loaded matrix transposed with the second matrix

$\mathbf{T_{function}(p)}$	#Cycles	#Cycles per element of result
$T_{multiply}(16)$	525	2.050
$T_{multiply}(256)$	70,669	1.078
$\mathbf{T_{multiply}(1024)}$	1,071,107	**1.021**
$T_{transpose}(16)$	528	2.062
$T_{transpose}(256)$	38,808	0.592
$\mathbf{T_{transpose}(1024)}$	548,366	**0.522**
$T_{transfer}(16)$	860	3.359
$T_{transfer}(256)$	106,168	1.620
$\mathbf{T_{transfer}(1024)}$	1,590,278	**1.516**
$T_{transfe+transpose+multiply}(16)$	1,913	7.472
$T_{transfe+transpose+multiply}(256)$	212,139	3.290
$\mathbf{T_{transfe+transpose+multiply}(1024)}$	3,209,751	**3.061**

Table 1: Simulation results.

For an enough big p, almost all the time the programs for transfer, transpose, and for multiplication (see Appendix D) run on the one-instruction cycle.

Because $T_{Multiply} + T_{transpose} \simeq T_{Transfer}$, for computing a *stream* of matrix multiplications, $T_{Transfer}$ is not added to the total computation time, because all transfers can be performed in parallel with the computation. Thus, we can claim that for each component of the result matrix the computation time is ~ 1.543 clock cycle, so doubling the architectural acceleration.

The program on HOST for multiplication is:

```
/*************************************************************************
File name: 0_hProgram.sv
                    HOST PROGRAM
*************************************************************************/
    hVALUE(0,0);      // rf[0] <= 0
    hPSEND(0,0);      // send program from 0 and run in array from 0
// the host program
    hSQGENX(8);                 // to array: generate matrix X
    hMAIN(`p+8,2);              // to array: generate diagonal matrix
    hSTART;                     // start clock cycles counter
    hSQMMULT(2*`p+8,`p+8,8);    // to array: multiply matrices
    hSTOP;                      // stop clock cycles counter
```

```
// end program
    hHALT;
```

The program on HOST for transpose is:

```
/*************************************************************************
File name: 0_hProgram.sv
                    HOST PROGRAM
*************************************************************************/
    hVALUE(0,0);      // rf[0] <= 0
    hPSEND(0,0);      // send program from 0 and run in array from 0
// the host program
    hSQGENN(8); // generate matrix N
    hSTART;
    hTRANS(`p+8,8);
    hSTOP;
// end program
    hHALT;
```

The full program associated to a multiplication must consider two matrix loads, a transpose, the multiplication and a matrix store. For simulation are added before the simulation the generation of the two matrixes in array the load in the host's memory, and ,at the and of simulation, the load of the result back in the array to display the result. The all program is:

```
/*************************************************************************
File name: 0_hProgram.sv
                    HOST PROGRAM
*************************************************************************/
    hVALUE(0,0);      // rf[0] <= 0
    hPSEND(0,0);      // send program from 0 and run in array from 0
// the host program
        // Generaate two matricesin the host's memory
        // send at 0 2 diagonal to host
            hMAIN(0,2);       // to array: generate matrix 2 diagonal
            hMSEND(0, 16);    // to array: send matrix
            hVALUE(1,127);    // end address in host
            hVALUE(0,0);      // start address in host
            hDGET(0,0,1);     // get data in host

        // send at 128 IX matrix to host
            hSQGENX(0);       // to array: generate matrix IX
            hMSEND(0, 16);    // to array: send matrix
            hVALUE(1,255);    // end address in host
            hVALUE(0,128);    // start address in host
            hDGET(0,0,1);     // get data in host

            hSTART;

    // Load at 0 the second matrix form the host's memory
            hMGET(0, 16);     // to array: get matrix at 16
```

```
            hVALUE(1,127);  // end address in host
            hVALUE(0,0);    // start address in host
            hDSEND(0,0,1);      // send data in host

    // Transpose at 16 the second matrix
            hTRANS(`p,0);

    // Load at 0 the first matrix from host
            hMGET(0, 16);   // to array: get matrix at 16
            hVALUE(1,255);  // end address in host
            hVALUE(0,128);  // start address in host
            hDSEND(0,0,1);      // send data in host

    // Multiply
            hSQMMULT(2*`p,`p,0);    // to array: multiply matrices

    // Send the result to host
            hMSEND(2*`p, 16);   // to array: send matrix
            hVALUE(1,127);      // end address in host
            hVALUE(0,0);        // start address in host
            hDGET(0,0,1);       // get data in host

            hSTOP;

    // Get back the result from host's memory
            hMGET(0, 16);   // to array: get matrix at 16
            hVALUE(1,127);  // end address in host
            hVALUE(0,0);    // start address in host
            hDSEND(0,0,1);  // send data in host
// end program
    hHALT;
```

B.2 Architectural acceleration

Architectural acceleration for a certain function is estimated considering the number of clock cycles used by accelerator to compute the selected function, and the number of clock cycles used by a mono-core processor to compute the same function. For our estimation we consider data provided by our simulation for pRISC and the results reported in [1] for matrix multiplication. Using data from Table B.1 and from TABLE 1 published [1], results:

$$\alpha_{pRISC(p)}^{1024x1024MM} = \mathbf{3.62p}$$

$$\alpha_{GPU(p)}^{1024x1024MM} = \mathbf{0.71p}$$

with:

$$\frac{\alpha_{pRISC(p)}^{1024x1024MM}}{\alpha_{GPU(p)}^{1024x1024MM}} \simeq \mathbf{5}$$

where:

pRISC : a $p = 1024$-cell accelerator produced in $12nm$, on $A_{pRISC} = 100mm^2$, consuming $P_{pRISC}(f_{pRISC}) \simeq 25Watt$, running at $f_{pRISC} = 1.4GHz$ (according to the simulation done using Synopsis environment).

GPU : the $p = 896$-cell GPU accelerator TU117, produced in $12nm$, on $A_{GPU} = 200mm^2$, consuming $P_{GPU}(f_{GPU}) \simeq 0.8 \times TDP = 60Watt$ (TGP is provided by the producer), running at $f_{GPU} = 1.4GHz$ (according to: `https://www.techpowerup.com/gpu-specs/nvidia-tu117.g881`)

CPU : AMD Ryzen 75800H, running at 3.2GHz.

Super-linear acceleration is achieved, in addition to the parallelism due to the cellular structure, through the parallelism between the multiplication performed in the linear array, the summation obtained in the reduction network, and the process control performed in the Controller.

B.3 Energy efficiency

We will approximate how both GPU and pRISC use power to perform a given task. We will consider that matrix multiplication consumes an average level of energy in the digital circuits of the two accelerators. Using the data provided by the simulations, for pRISC, and evaluations extracted from [1], for GPU TU117, we will calculate the energy consumed by running GEMM of 1024×1024.

$$E^{pRISC}_{1024x1024MM} = 16.12mJ$$

$$E^{GPU}_{1024x1024MM} = 70 \div 200mJ$$

$$\frac{E^{pRISC}_{1024x1024MM}}{E^{GPU}_{1024x1024MM}} = \mathbf{4.34 \div 12.4}$$

The ratio obtained represents an approximate assessment because the assessment of the power consumed in running a certain task cannot be done as rigorously as determining the architectural performance. The value of the power consumed was made under the assumption of an average use of the circuit structure, both in the case of pRISC and in the case of GPU.

B.4 Area use

We determine the silicon area utilization rate in relation to the same operation, by computing how many multiplication of 1024×1024 matrices, MM, are computed per second and per mm^2. For the two solutions we have:

$$A^{GPU}_{1024x1024MM} = 1000/13.35ms/200mm^2 = 0.375MM/sec/mm^2$$

$$A^{pRISC}_{1024x1024MM} = 1000/2.29ms/376mm^2 = 1.16MM/sec/mm^2$$

The following ratio of silicon area utilization rates results:

$$\frac{A^{pRISC}_{1024x1024MM}}{A^{GPU}_{1024x1024MM}} = \mathbf{3.1}$$

C How pRISC is used in a heterogeneous system

C.1 Subset of the library for linear algebra

A subset of functions belonging to the library of functions `theKernel16.v` is listed below. These functions are used by the host computer as part of a hardware implemented library of functions. All these functions are implemented in assembler (see in Appendix B an example of assembly code).

START : start cycles counter

STOP : stop cycles counter

INTRQ : send interrupt and cycles counter

MM_EWO(destination, source1, source2, linesNr, operation, waitMatricesNo) : element-wise operation on matrix

SM_MULT(destination, scalar, source, linesNr, waitMatricesNo) : scalar-matrix multiplication

MM_MULT(destination, source1, source2, linesNr, waitMatricesNo) : matrix-matrix multiplication

MM_MAC(destination, source1, source2, linesNr, waitMatricesNo) : matrix-matrix multiplication & accumulate

SEND_MATRIX_ARRAY(addr, nr_lines, nr_columns) : Data Transfer Engine command that performs the transfer of a data matrix in the data memory of the Array. This command needs to be followed by three parameters: the internal data memory address where the store will start, the number of lines, and the number of columns. If the number of columns is smaller than the number of cells in the Array, each data line will be padded with zeros.

GET_MATRIX_ARRAY(addr, nr_lines, nr_columns, wait_result) : Data Transfer Engine command that performs the transfer of a data matrix from the data memory of the Array to the Data Output FIFO. This command must be followed by three parameters: the internal data memory address where the read will start, the number of lines, and the number of columns. If the number of columns is smaller than the number of cells in the array, the transfer of a data line will be considered finished after shifting out only the needed data. If wait_result is 1, the transfer will wait for a result to be marked as ready by the Controller.

SEND_MATRIX_CTRL(addr, nr_lines, nr_columns) : Data Transfer Engine command that performs the transfer of a data matrix in the data memory of the Controller. This command needs to be followed by three parameters: the internal data memory address where the store will start, the number of lines, and the number of columns. If the number of columns is smaller than the number of cells in the Array, each data line will be padded with zeros.

GET_MATRIX_CTRL(addr, nr_lines, nr_columns, wait_result) : Data Transfer Engine command that performs the transfer of a data matrix from the data memory of the Controller to the Data Output FIFO. This command must be followed by three parameters: the internal data memory address where the read will start, the number of lines, and the number of columns. If the number of columns is smaller than the number of cells in the array, the transfer of a data line will be considered finished after shifting out only the needed data. wait_result is 1, the transfer will wait for a result to be marked as ready by the Controller.

cWAITMATW(scalar) : Wait for Matrices to be Written: Instruction for the Controller that tells it to perform no operation until the Data Transfer Engine confirms that a number of scalar matrices needed for the processing sequence are transferred into the Array's internal data memory.

cRESREADY : Result Ready: Instruction for the Controller to acknowledge the Data Transfer Engine that a processing sequence is finished and the result is ready to be read.

C.2 Code on HOST

Using mainly functions exemplified in Appendix C.1, the program for dense matrix multiplication defined for large matrices is listed below. The size of matrices is MAT_DIM, a multiple of p, the number of cells of our accelerator.

```
/*****************************************************************************
THE PROGRAM: matrix multiplication
*****************************************************************************/
// THE PROGRAM USED TO INITIALIZE THE ACCELERATOR
            pload   activate
            nop     nop
            nop     rednop
            `include "theKernel16.v"
            prun 0
// THE EXECUTION
 START;                             // start the cycle counter
 for(index2 = 0; index2 < (MAT_DIM/NR_CELLS)**2; index2 = index2 + 1) begin
     SEND_MATRIX(16, 16, 16);       // write dest=16,  #Lines=16, #Columns=16
     SEND_MATRIX(144, 16, 16);      // write dest=144, #Lines=16, #Columns=16
     SEND_MATRIX(512, 16, 16);      // write dest=512, #Lines=16, #Columns=16
     SEND_MATRIX(640, 16, 16);      // write dest=640, #Lines=16, #Columns=16
     MM_MULT(272, 16, 144, 16, 2); // dest=272, left=16, right=144, #Lines=16, #Mat=2
     MM_MAC(272, 512, 640, 16, 2); // dest=272, left=512, right=640, #Lines=16, #Mat=2
     if(MAT_DIM/NR_CELLS - 1 > 1) begin
         for(index = 0; index < (MAT_DIM/NR_CELLS)/2 - 1; index = index + 1) begin
             WAIT_RESULT_READY();           // wait for result to be ready
             SEND_MATRIX(16, 16, 16);       // write dest=16,  #Lines=16, #Columns=16
             SEND_MATRIX(144, 16, 16);      // write dest=144, #Lines=16, #Columns=16
             WAIT_RESULT_READY();           // wait for result to be ready
             SEND_MATRIX(512, 16, 16);      // write dest=512, #Lines=16, #Columns=16
             SEND_MATRIX(640, 16, 16);      // write dest=640, #Lines=16, #Columns=16
             MM_MAC(272, 16, 144, 16, 2);   // dest=272, left=16,  right=144, #Lines=16,
                                            // #Mat=2
             MM_MAC(272, 512, 640, 16, 2); // dest=272, left=512, right=640, #Lines=16,
                                            // #Mat=2
         end
     end
     WAIT_RESULT_READY();
     WAIT_RESULT_READY();
     GET_MATRIX(272, 16, 16); // read from dest=272, #Lines=16, #Columns=16
     INTRQ;                   // send the interrupt
 end
 STOP;                        // stop the cycle counter
```

Because the controller is implemented as two entities, one for controlling the calculation and another for controlling the data transfer, in the previous program, in the case of a matrix multiplication sequence, most of the transfers are performed in parallel with the computation.

D Example of Kernel Library

The library of functions on pRISC accelerator used by the programs in Section B is:

```
/*****************************************************************
File name: 00_theKernel.v
Description: LINEAR ALGEBRA KERNEL LIBRARY FUNCTIONS
*****************************************************************/
            cHALT;      NOP;
            cJMP(1);    NOP; // hSTART
            cJMP(2);    NOP; // hSTOP
            cJMP(3);    NOP; // hINTRQ
            cJMP(4);    NOP; // hSQGENX(d)
            cJMP(5);    NOP; // hSQGENN
            cJMP(6);    NOP; // hMSEND(addr-1,size)
            cJMP(7);    NOP; // hMGET(addr-1,size)
            cJMP(8);    NOP; // hMAIN(addr, value)
            cJMP(9);    NOP; // hSQADD(dest,left,right)
            cJMP(10);   NOP; // hVGENX(addr)
            cJMP(11);   NOP; // hVGENN(addr,value)
            cJMP(12);   NOP; // hSQMVMULT(matrix,vectror,dest)
            cJMP(13);   NOP; // hSQMMULT(dest,left,right)
            cJMP(14);   NOP; // hSQMMAC(dest,left,right)
            cJMP(15);   NOP; // hTRANS(dest,left)
            cJMP(16);   NOP; // hPRANDOM(dest)
            cJMP(64);   NOP; // hFILTER4x4(dest,s0,s1,...,sF)
            cJMP(65);   NOP; // hCONV4x4(dest,filter,matrix)
//********** START COUNTER ***************************************
    LB(1);  cSTART;         VLOAD(22);
            cJMP(32);       NOP;
//********** STOP COUNTER ****************************************
    LB(2);  cSTOP;          VLOAD(33);
            cJMP(32);       NOP;
/*
START/STOP: 6 cycles
*/
//********** INTERRUPT REQEST ************************************
    LB(3);  cSETINT;        VLOAD(44);
            cJMP(32);       NOP;
//********** SQUARE MATRIX X GENERATE ****************************
    LB(4);  cPARAM;         NOP;
            cNOP;           CLOAD;
            cVLOAD(`p-1);   VSUB(1);
            cNOP;           ADDRLD;
            cNOP;           IXLOAD;
    LB(17); cNOP;           RISTORE(1);
            cBRNZDEC(17);   VADD(1);
            cJMP(32);       NOP;
//********** SQUARE MATRIX N GENERATE ****************************
    LB(5);  cPARAM;         NOP;
            cNOP;           CLOAD;
            cVLOAD(`p-1);   VSUB(1);
            cNOP;           ADDRLD;
            cNOP;           VLOAD(0);
    LB(18); cNOP;           RISTORE(1);
```

```
                cBRNZDEC(18);     VADD(1);
                cJMP(32);         NOP;
//********** SEND MATRIX ****************************************
    LB(6);      cPARAM;           NOP;
                cPARAM;           CLOAD;
                cNOP;             ADDRLD;
                cNOP;             NOP;
                cNOP;             RISENDIO(0);
    LB(19);     cBRZDEC(32);      NOP;
                cNOP;             NOP;    // p=64
                cNOP;             NOP;
                cNOP;             NOP;
                cNOP;             NOP;
                cNOP;             NOP;
                cDATAEXT('p/2);   NOP;
                cJMP(19);         RISENDIO(1);
//********** GET MATRIX *****************************************
    LB(7);      cPARAM;           GETIO('m-1);
                cVSUB(1);         NOP;
                cPARAM;           CLOAD;
                cVSUB(1);         ADDRLD;
                cNOP;             NOP;
                cNOP;             NOP;
                cNOP;             NOP;
    LB(20);     cDATAINS('p/2);   NOP;
                cNOP;             NOP;
                cNOP;             NOP;
                cNOP;             RIGETIO(1);
                cBRZDEC(32);      NOP;
                cNOP;             NOP;
                cNOP;             NOP;
                cNOP;             NOP;
                cNOP;             NOP;
                cJMP(20);         NOP;
//********** MAIN MATRIX GENERATE *************************************
    LB(8);      cPARAM;           NOP;
                cNOP;             CLOAD;
                cPARAM;           ADDRLD;
                cNOP;             IXLOAD;
                cNOP;             WHEREZERO;
                cNOP;             CLOAD;
                cNOP;             ELSEWHERE;
                cNOP;             VLOAD(0);
                cNOP;             ENDWHERE;
                cVLOAD('p-2);     SENDSR;
                cGRSHIFT;         RSTORE(0);
    LB(22);     cNOP;             GETSR;
                cGRSHIFT;         RISTORE(1);
                cBRNZDEC(22);     NOP;
                cJMP(32);         NOP;
//********** ADD SQUARE MATRICES **************************************
    LB(9);      cPARAM;           NOP;
                cSTORE(3);        NOP;      // dest at mem[3]
                cPARAM;           NOP;
                cSTORE(4);        NOP;      // left at mem[4]
```

```
            cPARAM;             NOP;
            cSTORE(5);          NOP;        // right at mem[5]
            cSUB(4);            NOP;
            cSTORE(0);          NOP;        // right-left at mem[0]
            cLOAD(3);           NOP;
            cSUB(5);            NOP;
            cSTORE(1);          NOP;        // dest-right at mem[1]
            cLOAD(4);           NOP;
            cSUB(3);            NOP;
            cVADD(1);           NOP;
            cSTORE(2);          NOP;        // left-dest+1 at mem[2]
            cVLOAD('p);         NOP;
            cSTORE(6);          NOP;
    LB(23); cLOAD(4);           NOP;
            cADD(0);            CALOAD;
            cADD(1);            CAADD;
            cADD(2);            CSTORE;
            cSTORE(4);          NOP;
            cLOAD(6);           NOP;
            cVSUB(1);           NOP;
            cSTORE(6);          NOP;
            cBRNZ(23);          NOP;
            cJMP(32);           NOP;
//************ INDEX VECTOR GENERATE *********************************
    LB(10); cPARAM;             IXLOAD;
            cJMP(32);           CSTORE;
//************ N VECTOR GENERATE *************************************
    LB(11); cPARAM;             NOP;
            cPARAM;             CLOAD;
            cJMP(32);           CSTORE;
//************ MATRIX-VECTOR MULTIPLY ********************************
    LB(12); cPARAM;                     REDADD;     // dest address
            cSTORE(1);                  VLOAD('p);  // mem[1] dest address
            cPARAM;                     NOP;        // matrix address
            cNOP;                       CADD;       // end matrix
            cPARAM;                     ADDRLD;
            cVLOAD('p-1);               CALOAD;

    // MAIN LOOP
    LB(24); cRINSBR(24);                REDMULT(-1);
    // END MAIN LOOP

            cVLOAD($clog2('p)-1);   NOP;

    // LATENCY LOOP
    LB(27); cBRNZDEC(27);               NOP;
    // END LATENCY LOOP

            cLOAD(1);                   GETSR;
            cJMP(32);                   CSTORE;
//********** MULTIPLY SQUARE MATRICES ********************************
    LB(13); cPARAM;                     REDADD;
            cVSUB(1);                   NOP;
            cSTORE(3);                  NOP;    // dest => 3
            cPARAM;                     NOP;
```

```
                cVADD(`p-1);            NOP;
                cSTORE(0);              NOP;    // left => 0
                cPARAM;                 CLOAD;
                cSTORE(2);              ADDRLD; // right => 2
                cVLOAD(`p);             NOP;
                cSTORE(1);              NOP;    // counter
                cLOAD(2);               NOP;
                cNOP;                   CALOAD;
    LB(66);     cVLOAD(`p-2);           NOP;
                cLREDINS;               REDMULT(0); // RILOAD(0);

    // MAIN LOOP
    LB(25);     cRINSBR(25);            REDMULT(-1);
    // END MAIN LOOP

                cVLOAD($clog2(`p)-2);   NOP;

    // LATENCY LOOP
    LB(28);     cBRNZDEC(28);           NOP;
    // END LATENCY LOOP

                cLOAD(3);               NOP;
                cVADD(1);               GETSR;
                cSTORE(3);              CSTORE;
                cLOAD(0);               NOP;
                cLOAD(2);               CLOAD;
                cVADD(1);               ADDRLD;
                cSTORE(2);              CALOAD;
                cLOAD(1);               NOP;
                cVSUB(1);               NOP;
                cSTORE(1);              NOP;
                cBRNZ(66);              NOP;
                cJMP(32);               NOP;
/*
    #cycles(p) = p^2 + plog_2(p) + 12p + 13
    #cycles(16)  = 525  => 2.05 #cycles/resultMatrixElement
    #cycles(256) = 70669 => 1.078 #cycles/resultMatrixElement
    #cycles(1024) = 1,071,107 => 1.021 #cycles/resultMatrixElement
*/
//********** MULTIPLY & ACCUMULATE SQUARE MATRICES ******************
    LB(14);     cPARAM;                 REDADD;
                cVSUB(1);               NOP;
                cSTORE(3);              NOP;    // dest => 3
                cPARAM;                 NOP;
                cVADD(`p-1);            NOP;
                cSTORE(0);              NOP;    // left => 0
                cPARAM;                 CLOAD;
                cSTORE(2);              ADDRLD; // right => 2
                cVLOAD(`p);             NOP;
                cSTORE(1);              NOP;    // counter
                cLOAD(2);               NOP;
                cNOP;                   CALOAD;
    LB(67);     cVLOAD(`p-2);           NOP;
                cLREDINS;               REDMULT(0); // RILOAD(0);
```

```
    // MAIN LOOP
    LB(26); cRINSBR(26);            REDMULT(-1);
    // END MAIN LOOP

            cVLOAD($clog2(`p)-2);   NOP;

    // LATENCY LOOP
    LB(29); cBRNZDEC(29);           NOP;
    // END LATENCY LOOP

            cLOAD(3);               NOP;
            cVADD(1);               GETSR;
            cSTORE(3);              CAADD;
            cLOAD(0);               CSTORE;
            cLOAD(2);               CLOAD;
            cVADD(1);               ADDRLD;
            cSTORE(2);              CALOAD;
            cLOAD(1);               NOP;
            cVSUB(1);               NOP;
            cSTORE(1);              NOP;
            cBRNZ(67);              NOP;
            cJMP(32);               NOP;
// ********** TRANSPOSE ***********************************************
    LB(15); cPARAM;                 IXLOAD;
            cSTORE(0);              SENDSR; // mem[0]=dest
            cPARAM;                 NOP;
            cSTORE(1);              NOP;    // mem[1]= source
            cVADD(2*`p);            NOP;
            cSTORE(2);              NOP;    // mem[2]=temp
            cVLOAD(`p+1);           NOP;    // counter
            cSTORE(3);              NOP;
// READ ALL
    LB(33); cLOAD(1);               GETSR;
            cGRROTATE;              CADD;
            cLOAD(3);               ADDRLD;
            cVSUB(1);               NOP;
            cBRZ(34);               NOP;
            cSTORE(3);              NOP;
            cLOAD(2);               RLOAD(0);
            cVADD(1);               CSTORE;
            cSTORE(2);              NOP;
            cJMP(33);               NOP;
// ROTATE ALL
    LB(34); cVLOAD(`p);             VLOAD(3*`p);
            cSTORE(3);              ADDRLD;
            cNOP;                   NOP;
    LB(35); cVSUB(1);               RILOAD(-1);
            cBRZ(37);               SENDSR;
            cSTORE(3);              NOP;

    LB(36); cLROTBR(36);            GETSR;

            cLOAD(3);               NOP;
            cJMP(35);               RSTORE(0);
// STORE ALL
```

```
    LB(37); cVLOAD('p-1);           IXLOAD;
            cSTORE(3);              SENDSR;
            cGRROTATE;              NOP;
    LB(38); cLOAD(0);               GETSR;
            cLOAD(3);               CADD;
            cVADD(2*'p);            ADDRLD;
            cGRROTATE;              CALOAD;
            cLOAD(3);               RSTORE(0);
            cBRZDEC(32);            NOP;
            cSTORE(3);              NOP;
            cJMP(38);               NOP;
/*
#cycles(16)   = 528    => 2.065 #cycles/resultMatrixElement
#cycles(32)   = 1288   => 1.1257 #cycles/resultMatrixElement
#cycles(64)   = 3576   => 0.873 #cycles/resultMatrixElement
#cycles(128)  = 11224  => 0.685 #cycles/resultMatrixElement
#cycles(256)  = 38808  => 0.592 #cycles/resultMatrixElement
#cycles(512)  = 143128  => 0.545 #cycles/resultMatrixElement
#cycles(1024) = 548366  => 0.522 #cycles/resultMatrixElement
*/
//********** PSEUDO-RANDOM MATRIX **********************************
    LB(16); cPARAM;                 ACTIVATE;
            cNOP;                   CLOAD;
            cNOP;                   ADDRLD;
            cVLOAD('p-1);           IXLOAD;
    LB(33); cNOP;                   VADD(29);
            cNOP;                   VMULT(98765);
            cNOP;                   SHR;
            cNOP;                   SHR;
            cNOP;                   SHR;
            cNOP;                   SHR;
            cNOP;                   SHR;
            cNOP;                   VAND(31);
            cNOP;                   CRSTORE;
            cBRNZDEC(33);           NOP;
            cJMP(32);               NOP;

//*************CONVOLUTION*****************************************

//********** 4x4 FILTER LOAD ***************************************
    LB(64); cPARAM;                 NOP;
            cSTORE(0);              VAND(3);
            cPARAM;                 WHEREZERO;
            cNOP;                   CLOAD;
            cPARAM;                 SELSHIFT;
            cNOP;                   CLOAD;
            cPARAM;                 SELSHIFT;
            cNOP;                   CLOAD;
            cPARAM;                 SELSHIFT;
            cNOP;                   CLOAD;
            cLOAD(0);               ACTIVATE;
            cVADD(1);               CSTORE;

            cSTORE(0);              IXLOAD;
            cNOP;                   VAND(3);
```

```
            cPARAM;                 WHEREZERO;
            cNOP;                   CLOAD;
            cPARAM;                 SELSHIFT;
            cNOP;                   CLOAD;
            cPARAM;                 SELSHIFT;
            cNOP;                   CLOAD;
            cPARAM;                 SELSHIFT;
            cNOP;                   CLOAD;
            cLOAD(0);               ACTIVATE;
            cVADD(1);               CSTORE;

            cSTORE(0);              IXLOAD;
            cNOP;                   VAND(3);
            cPARAM;                 WHEREZERO;
            cNOP;                   CLOAD;
            cPARAM;                 SELSHIFT;
            cNOP;                   CLOAD;
            cPARAM;                 SELSHIFT;
            cNOP;                   CLOAD;
            cPARAM;                 SELSHIFT;
            cNOP;                   CLOAD;
            cLOAD(0);               ACTIVATE;
            cVADD(1);               CSTORE;

            cSTORE(0);              IXLOAD;
            cNOP;                   VAND(3);
            cPARAM;                 WHEREZERO;
            cNOP;                   CLOAD;
            cPARAM;                 SELSHIFT;
            cNOP;                   CLOAD;
            cPARAM;                 SELSHIFT;
            cNOP;                   CLOAD;
            cPARAM;                 SELSHIFT;
            cNOP;                   CLOAD;
            cLOAD(0);               ACTIVATE;
            cNOP;                   CSTORE;

            cJMP(32);               NOP;
/*
53 cycles
*/
//********** 4x4 CONVOLUTION ****************************************
    // initialization
    LB(65); cVLOAD('p);             NOP;
            cSTORE(3);              NOP;    // mem[3}=line out of p
            cVLOAD(0);              NOP;
            cSTORE(4);              NOP;    // mem(4)=position out of 4
            cPARAM;                 NOP;
            cSTORE(0);              NOP;    // mem(0)=dest
            cPARAM;                 NOP;
            cSTORE(2);              CLOAD;  // mem(2)=filter
            cPARAM;                 ADDRLD;
            cSTORE(1);              NOP;    // mem(1)=matrix

    // position filter in the shift space at 2p+4
```

```
LB(41);  cNOP;                       RLOAD(0);
         cNOP;                       RSTORE(`p+4);
         cNOP;                       RILOAD(1);
         cNOP;                       RSTORE(`p+4);
         cNOP;                       RILOAD(1);
         cNOP;                       RSTORE(`p+4);
         cNOP;                       RILOAD(1);
         cNOP;                       RSTORE(`p+4);
// mult & add lines & filter shift
LB(39);  cLOAD(2);                   NOP;
         cVADD(`p+3);                NOP;
         cNOP;                       CLOAD;
         cLOAD(1);                   ADDRLD;
         cNOP;                       NOP;

         cNOP;                       RLOAD(1);
         cNOP;                       SENDSR;
         cGRSHIFT;                   CAMULT;
         cVADD(1);                   STORE(2*(`p+4));
         cNOP;                       GETSR;
         cNOP;                       RISTORE(1);
         cNOP;                       NOP;

         cNOP;                       RLOAD(1);
         cNOP;                       SENDSR;
         cGRSHIFT;                   CAMULT;
         cVADD(1);                   ADD(2*(`p+4));
         cNOP;                       STORE(2*(`p+4));
         cNOP;                       GETSR;
         cNOP;                       RISTORE(1);
         cNOP;                       NOP;

         cNOP;                       RLOAD(1);
         cNOP;                       SENDSR;
         cGRSHIFT;                   CAMULT;
         cVADD(1);                   ADD(2*(`p+4));
         cNOP;                       STORE(2*(`p+4));
         cNOP;                       GETSR;
         cNOP;                       RISTORE(1);
         cNOP;                       NOP;

         cNOP;                       RLOAD(1);
         cNOP;                       SENDSR;
         cGRSHIFT;                   CAMULT;
         cVADD(1);                   ADD(2*(`p+4));
         cNOP;                       STORE(2*(`p+4));
         cNOP;                       GETSR;
         cNOP;                       RISTORE(1);

         cNOP;                       LOAD(2*(`p+4));

// add on line
         cNOP;                       SENDSR;
         cGLSHIFT;                   NOP;
         cGLSHIFT;                   SRADD;
```

```
            cGLSHIFT;               SRADD;
            cNOP;                   SRADD;
            cNOP;                   STORE(2*('p+4));
    // store at destination
            cLOAD(4);               IXLOAD;
            cNOP;                   VAND(3);
            cVADD(1);               SEARCH;
            cSTORE(4);              NOP;
            cLOAD(0);               LOAD(2*('p+4));
            cLOAD(4);               CSTORE;
            cVSUB(4);               ACTIVATE;
    // test endline
            cBRZ(40);               NOP;

            cJMP(39);               NOP;

    LB(40); cLOAD(0);               NOP;
            cVADD(1);               NOP;
            cSTORE(0);              NOP;

            cLOAD(1);               NOP;
            cVADD(1);               NOP;
            cSTORE(1);              NOP;

            cVLOAD(0);              NOP;
            cSTORE(4);              NOP;

            cLOAD(1);               NOP;
            cVSUB(13);              NOP;
            cBRZ(32);               NOP;
            cLOAD(2);               NOP;
            cNOP;                   CLOAD;
            cNOP;                   ADDRLD;

            cJMP(41);               NOP;

/*
2948cycles; 17.44 cycles/element; ACC=14.22 for p=16
*/

//********** 4x4 CONVOLUTION ****************************************
    // initialization
    LB(65); cVLOAD('p);             NOP;
            cSTORE(3);              NOP;    // mem[3}=line out of p
            cVLOAD(0);              NOP;
            cSTORE(4);              NOP;    // mem(4)=position out of 4
            cPARAM;                 NOP;
            cSTORE(0);              NOP;    // mem(0)=dest
            cPARAM;                 NOP;
            cSTORE(1);              NOP;    // mem(1)=matrix
            cPARAM;                 NOP;
            cSTORE(2);              CLOAD;  // mem(2)=filter
            cNOP;                   ADDRLD;
            cNOP;                   NOP;
    // position filter in the shift space at 2p+4
```

```
LB(41); cNOP;                   RLOAD(0);
        cNOP;                   RSTORE(‘p+4);
        cNOP;                   RILOAD(1);
        cNOP;                   RSTORE(‘p+4);
        cNOP;                   RILOAD(1);
        cNOP;                   RSTORE(‘p+4);
        cNOP;                   RILOAD(1);
        cNOP;                   RSTORE(‘p+4);
// mult & add lines
LB(39); cLOAD(2);               NOP;
        cVADD(‘p+3);            NOP;
        cNOP;                   CLOAD;
        cLOAD(1);               ADDRLD;

        cVADD(1);               CALOAD;
        cNOP;                   RIMULT(1);
        cNOP;                   STORE(2*(‘p+4));

        cVADD(1);               CALOAD;
        cNOP;                   RIMULT(1);
        cNOP;                   ADD(2*(‘p+4));
        cNOP;                   STORE(2*(‘p+4));

        cVADD(1);               CALOAD;
        cNOP;                   RIMULT(1);
        cNOP;                   ADD(2*(‘p+4));
        cNOP;                   STORE(2*(‘p+4));

        cNOP;                   CALOAD;
        cNOP;                   RIMULT(1);
        cNOP;                   ADD(2*(‘p+4));
        cNOP;                   STORE(2*(‘p+4));
// add on line
        cNOP;                   SENDSR;
        cGLSHIFT;               NOP;
        cGLSHIFT;               SRADD;
        cGLSHIFT;               SRADD;
        cNOP;                   SRADD;
        cNOP;                   STORE(2*(‘p+4));
// store at destination
        cLOAD(4);               IXLOAD;
        cNOP;                   VAND(3);
        cVADD(1);               SEARCH;
        cSTORE(4);              NOP;
        cLOAD(0);               LOAD(2*(‘p+4));
        cNOP;                   CSTORE;
        cNOP;                   ACTIVATE;
// test endline
        cLOAD(4);               NOP;
        cVSUB(4);               NOP;
        cBRZ(40);               NOP;
// filter shift
        cLOAD(2);               NOP;
        cVADD(‘p+4);            NOP;
        cNOP;                   CALOAD;
```

```
            cNOP;                   SENDSR;
            cGRSHIFT;               NOP;
            cNOP;                   NOP;
            cNOP;                   GETSR;
            cVADD(1);               CSTORE;

            cNOP;                   CALOAD;
            cNOP;                   SENDSR;
            cGRSHIFT;               NOP;
            cNOP;                   GETSR;
            cVADD(1);               CSTORE;

            cNOP;                   CALOAD;
            cNOP;                   SENDSR;
            cGRSHIFT;               NOP;
            cNOP;                   GETSR;
            cVADD(1);               CSTORE;

            cNOP;                   CALOAD;
            cNOP;                   SENDSR;
            cGRSHIFT;               NOP;
            cNOP;                   GETSR;
            cNOP;                   CSTORE;

            cJMP(39);               NOP;

    LB(40); cLOAD(0);               NOP;
            cVADD(1);               NOP;
            cSTORE(0);              NOP;

            cLOAD(1);               NOP;
            cVADD(1);               NOP;
            cSTORE(1);              NOP;

            cVLOAD(0);              NOP;
            cSTORE(4);              NOP;

            cLOAD(1);               NOP;
            cVSUB(13);              NOP;
            cBRZ(32);               NOP;
            cLOAD(2);               NOP;
            cNOP;                   CLOAD;
            cNOP;                   ADDRLD;
            cJMP(41);               NOP;
```

E Microarchitecture

```
/*************************************************************
File name: 0_DEFINES.vh
*************************************************************/
`define n (32)      // internal word
```

```
`define p (16)       // number of cells
`define m (1024)     // memory size in cells
`define M (4096)       // memory size in host
`define c (0)          // origin of matrix in convolution

                      ////////
                      // HOST //
                      ////////

/*************************************************************
Name: Host's instruction set architecture

instr={opCode[5:0],dest[4:0],left[4:0],right[4:0],nu[10:0]} |
      {opCode[5:0],dest[4:0],left[4:0],val[15:0]}
reg [31:0]                rf[0:31]
reg [$clog2(`m)-1:0]      pc
reg                       cr
*************************************************************/
// CONTROL INSTRUCTIONS
`define hnop      6'b000000 // pc <= pc+1
`define hjmp      6'b000001 // pc <= pc+val
`define hjmpz     6'b000010 // pc <= (rf[l]==0) ? pc+val:pc+1
`define hjmpnz    6'b000011 // pc <= (rf[l]==0) ? pc+1:pc+val
`define hpsend    6'b100000 // pc <= (host2int[31:24]==
                            // {`prun,`ctl}) ? pc + 1 : pc
                            // h2iPwrite = 1
                            // rf[l] <= rf[l]+!i2hPfull
`define hdget     6'b100010 // pc <= (leftOut==rightOut) ?
                            // pc + 1 : pc
                            // i2hDread = 1
                            // rf[l] <= rf[l]+!i2hDfull
`define hdsend    6'b100001 // pc <= (leftOut==rightOut) ?
                            // pc + 1 : pc
                            // h2iDwrite = 1
                            // rf[l] <= rf[l]+!i2hDfull
`define hintwait  6'b000111 // pc <= (int) ? pc+1 : pc
`define hajmp     6'b001000 // pc <= val
`define hhalt     6'b001001 // pc <= pc
`define hsttcc    6'b001010 // start host cycle counter
`define hstpcc    6'b001011 // stop host cycle counter

// FUNCTIONAL INSTRUCTIONS
`define hadd      6'b010000 // {cr,rf[d]} <= rf[l]+rf[r]
`define haddcr    6'b010001 // {cr,rf[d]} <= rf[l]+rf[r]+cr
`define hsub      6'b010010 // {cr,rf[d]} <= rf[l]-rf[r]
`define hsubcr    6'b010011 // {cr,rf[d]} <= rf[l]-rf[r]-cr
`define hmult     6'b010100 // {cr,rf[d]} <= {cr, rf[l]*rf[r]}
`define hbwand    6'b010101 // {cr,rf[d]} <= {cr, rf[l] & rf[r]}
`define hbwor     6'b010110 // {cr,rf[d]} <= {cr, rf[l] | rf[r]}
`define hbwxor    6'b010111 // {cr,rf[d]} <= {cr, rf[l] ^ rf[r]}
`define haddv     6'b011000 // {cr,rf[d]} <= rf[l]+{{16{val[15]}},val}
`define haddcrv   6'b011001 // {cr,rf[d]} <= rf[l]+{{16{val[15]}},val}+cr
`define hsubv     6'b011010 // {cr,rf[d]} <= rf[l]-{{16{val[15]}},val}
`define hsubcrv   6'b011011 // {cr,rf[d]} <= rf[l]-{{16{val[15]}},val}-cr
`define hmultv    6'b011100 // {cr,rf[d]} <= rf[l]*{{16{val[15]}},val}
```

```
`define hbwandv  6'b011101 // {cr,rf[d]} <= {cr,rf[l]&{{16{val[15]}},val}}
`define hbworv   6'b011110 // {cr,rf[d]} <= {cr,rf[l]|{{16{val[15]}},val}}
`define hbwxorv  6'b011111 // {cr,rf[d]} <= {cr,rf[l]^{{16{val[15]}},val}}

// DATA TRANSFER INSTRUCTIONS
`define hfsend   6'b100011 // function send {code[5:0], `prun, `ctl, initAddress}
`define hssend   6'b100100 // scalar send {code[5:0], scalar}
`define hvalue   6'b101000 // {cr,rf[d]} <= {cr, {{16{val[15]}},val}}
`define hinsval  6'b101001 // {cr,rf[d]} <= {cr, {rf[l][15:0],val}
`define hload    6'b101010 // {cr,rf[d]} <= {cr, hDmem[rf[l]}
`define hstore   6'b111011 // hDmem[rf[l] <= rf[r]

                //////////////////////////
                // ACCELERATOR'S CONTROLLER //
                //////////////////////////

/*************************************************************
 Accelerator's controller state:
*************************************************************/
`define idle    2'b00
`define loading 2'b01
`define running 2'b10

/*************************************************************
Name: Controller's instruction set architecture

instr = {cOpcodeBinary[4:0] , // operation code for binary operations
         cMode[2:0]         , // second operand selection mode
         value[23:0]        } // signed integer value
         |
        {8'b11111_000       ,
         cOpcodeUnary[4:0]  , // operation code for unary operations
         value[18:0]        } // integer to parameterize operations

reg [$clog2(`m)-1:0] pc          // program counter
reg [31:0]           acc         // controller's accumuulator
reg                  cr          // controller's carry
reg[$clog2(`m)-1:0]  addrReg     // controller's address reg.
reg[31:0]            mem[0:m-1] // controller's data memory

val = {{8{value[23]}},value}     // used as operand
addr = value[$clog2(`m)-1:0]     // used as address

coOp = redOut
*************************************************************/

/*************************************************************
 Addressing modes defining the right operand:
*************************************************************/
`define imm 3'b000 // op: val
`define dir 3'b001 // op: mem[addr]
`define rel     3'b010 // op: mem[addrReg+addr]
`define rei 3'b011 // op: mem[addrReg+addr]; addrReg = addrReg+addr
`define cim 3'b100 // op: coOp
`define cdr 3'b101 // op: mem[coOp]
```

```
`define crl 3'b110 // op: mem[addrReg+coOp]
`define cri     3'b111 // op: mem[addrReg+coOp]; addrReg = addrReg+coOp

// FUNCTIONAL INSTRUCTIONS
`define cadd    5'b00000 // {cr,acc} <= acc + op
`define caddcr  5'b00001 // {cr,acc} <= acc + op + cr
`define csub    5'b00010 // {cr,acc} <= acc - op
`define csubcr  5'b00011 // {cr,acc} <= acc - op - cr
`define cmult   5'b00100 // {cr,acc} <= {cr, acc * op}
`define cbwand  5'b00101 // {cr,acc} <= {cr, acc & op}
`define cbwor   5'b00110 // {cr,acc} <= {cr, acc | op}
`define cbwxor  5'b00111 // {cr,acc} <= {cr, acc ^ op}
// DATA MOVE INSTRUCTIONS
`define cload   5'b01000 // {cr,acc} <= {cr, op}
`define cstore  5'b01001 // cMem <= acc
`define cinsval 5'b01010  // {cr,acc} <= {cr, (acc[23:0],val[7:0]}

// SELECTS NO-OP INSTRUCTIONS
`define contr   5'b11111 // instruction: instr[23:19]

// CONTROL & GLOBAL INSTRUCTIONS for instr[31:24]= 11111_000
`define nop     5'b00000 // pc <= pc+1
`define jmp     5'b00001 // pc <= pc+addr
`define brz     5'b00010 // pc <= (acc = 0) ? pc+addr : pc+1
`define brnz    5'b00011 // pc <= (acc = 0) ? pc+1 : pc+addr
`define brzdec  5'b00100 // pc <= (acc = 0) ? pc+addr : pc+1
`define brnzdec 5'b00101 // pc <= (acc = 0) ? pc+1 : pc+addr
`define ajmp    5'b00110 // pc <= addr
`define halt    5'b00111 // pc <= pc

`define setint  5'b01000 // set interrupt

`define datains 5'b01001 // insert data in array
`define dataext 5'b01010 // extract data from arrray

`define param   5'b01011 // load parameter
`define pload   5'b01100 // load program starting to val
`define prun    5'b01101 // run the program at val

`define start   5'b01110 // start cycle counter
`define stop    5'b01111 // stop cycle coounter

// SPECIAL UNCONVENTIONAL INSTRUCTIONS
`define lrotbr  5'b11011 // left rotate; brnzdec !!!!!
`define rinsbr  5'b11100 // left instert redOut; brnzdec !!!!!

`define gshift  5'b11101 // global_shift(val[2:0])

`define crela   5'b11110 // cAddrReg <= acc; load address
`define cshift  5'b11111 // local_shift(val[2:0])


                    //////////////////////
                    // ACCELERATOR'S ARRAY //
                    //////////////////////
```

```
/*************************************************************
Name: Array's instruction set architecture

instr = {opcodeBinary[4:0]   , // operation code for binary operations
         mode[2:0]           , // second operand selection mode
         value[23:0]         } // signed integer value
         |
        {8'b11111_000        ,
         opcodeUnary[4:0]    , // operation code for unary operations
         value[18:0]         } // integer to parameterize operations

reg [31:0]           acc[0:p-1]       // array's accumuulator
reg                  cr[0:p-1]        // array's carry
reg[$clog2(m)-1:0]   addrReg[0:p-1]   // array's address reg.
reg[n-1:0]           mem[0:m]         // array's data memory
reg[$clog2(p)-1]     act[0:p-1]       // activate counter
reg[n-1:0]           sr[0:p-1]        // serial register

val = {{8{value[23]}},value}     // used as operand
addr = value[$clog2(`m)-1:0]     // used as address

b[i] = (act == 0) ? 1 : 0

coOp = acc
*************************************************************/

/*************************************************************
 Addressing modes defining the right operand:
*************************************************************/
`define imm 3'b000 // op: val
`define dir 3'b001 // op: mem[addr]
`define rel     3'b010 // op: mem[addrReg+addr]
`define rei 3'b011 // op: mem[addrReg+addr]; addrReg = addrReg+addr
`define cim 3'b100 // op: coOp
`define cdr 3'b101 // op: mem[coOp]
`define crl 3'b110 // op: mem[addrReg+coOp]
`define cri     3'b111 // op: mem[addrReg+coOp]; addrReg = addrReg+coOp

// FUNCTIONAL INSTRUCTIONS
`define add     5'b00000 // {cr[i],acc[i]} <=  acc[i] + op
`define addcr   5'b00001 // {cr[i],acc[i]} <=  acc[i] + op + cr
`define sub     5'b00010 // {cr[i],acc[i]} <=  acc[i] - op
`define subcr   5'b00011 // {cr[i],acc[i]} <=  acc[i] - op - cr
`define mult    5'b00100 // {cr[i],acc[i]} <= {cr[i], acc[i] * op}
`define bwand   5'b00101 // {cr[i],acc[i]} <= {cr[i], acc[i] & op}
`define bwor    5'b00110 // {cr[i],acc[i]} <= {cr[i], acc[i] | op}
`define bwxor   5'b00111 // {cr[i],acc[i]} <= {cr[i], acc[i] ^ op}
// DATA MOVE INSTRUCTIONS
`define load    5'b01000 // {cr[i],acc[i]} <= {cr[i], op}
`define store   5'b01001 // aMem[address[i]] <= aAcc
`define insval  5'b01010 // acc[i] <= {acc[i][23:0], val[7:0]}

`define sendio  5'b01011 // ioReg[i] <= aMem[address]; ioSet
`define getio   5'b01100 // aMem[address[i]] <= ioReg[i]; ioRst
```

```
// SEARCH INSTRUCTIONS
`define search  5'b01101 // act[i] <= (b[i] & acc[i]=op) ? act[i] : act[i]+1
`define csearch 5'b01110 // act[i] <= (b[i-1] & acc[i]=op) ? 0 : act[i]+1
`define insert  5'b01111 // insert in the first active position in sr

// FLOAT INSTRUCTIONS
`define fpmult  5'b10000 // FP32 multiplication
//`define hfpmul  5'b10010 // FP16 multiplication


// SPECIAL UNCONVENTIONAL INSTRUCTION
`define redmult 5'b10001 // reductionInput <= rimult(...); acc[i] <= acc[i]  !!!!!
`define redfpm  5'b10010 // redmult with fp multiplication

// SELECTS NO-OP INSTRUCTIONS
`define contr   5'b11111 // instruction: instr[23:19]

// CONTROL & GLOBAL INSTRUCTIONS for instr[31:24]= 11111_000
`define allact  5'b00000 // act[i] <= 0
`define where   5'b00001 // act[i] <= (b[i]&cond) ? act[i] : act[i]+1
`define elsew   5'b00010 // act[i] <= (act[i]=0)?1:(act[i]=1)?0:act[i]
`define back    5'b00011 // act[i] <= (act[i]=0)? 0 : act[i]-1
`define selsh   5'b00100 // selection global shift right

`define delete  5'b00110 // delete in first position in sr
`define getsr   5'b00111 // acc[i] <= sr[i]
`define sendsr  5'b01000 // sr[i] <= acc[i]

`define ixload  5'b01001 // acc[i] <= i

`define setred  5'b01010 // set reduce's function

`define sradd   5'b01011 // acc[i] <= acc[i] + sr[i]          !!!!!
`define srmin   5'b01100 // acc[i] <= min(acc[i], sr[i])      !!!!!
`define srmax   5'b01101 // acc[i] <= max(acc[i], sr[i])      !!!!!

`define arela   5'b01110 // aAddrReg <= aAcc
`define shift   5'b01111 // local_shift(val[2:0])

// FLOAT INSTRUCTIONS for instr[31:24]= 11111_000
//`define i2hf  5'b10000 // 32-bit integer 2 FP16
//`define i2bf  5'b10001 // 32-bit integer 2 BF16
`define i2f             5'b10010 // 32-bit integer 2 FP32
```

References

[1] M. Q. Ansari, M. Q. Ansari (2025) Accelerating Matrix Multiplication: A Performance Comparison Between Multi-Core CPU and GPU, At: https://arxiv.org/abs/2507.19723

[2] Mihaela Malița, Gheorghe Ștefan, Dominique Thiébaut (2007) Not Multi-, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation, *ACM SIGARCH Computer Architecture News*, **35**(5)32:38, Special issue: ALPS '07 - Advanced low power systems; communication at International Workshop on Advanced Low Power Systems held in conjunction with 21st International Conference on Supercomputing June 17, 2007 Seattle, WA, USA.

[3] Gheorghe Ştefan (2006) The CA1024: A Massively Parallel Processor for Cost-Effective HDTV, *SPRING PROCESSOR FORUM JAPAN*, June 8-9, 2006, Tokyo.

[4] Gheorghe Ştefan, Anand Sheel, Bogdan Mîţu, Tom Thomson, Dan Tomescu (2006) The CA1024: A Fully Programmable System-On-Chip for Cost-Effective HDTV Media Processing, *Hot Chips: A Symposium on High Performance Chips*, Memorial Auditorium, Stanford University, August 20 to 22, 2006.

[5] Gheorghe M. Ștefan, Mihaela Malița (2014) Can One-Chip Parallel Computing Be Liberated From Ad Hoc Solutions? A Computation Model Based Approach and Its Implementation, *18th International Conference on Circuits, Systems, Communications and Computers* , Santorini Island, Greece, pp 582-597. http://www.inase.org/library/2014/santorini/bypaper/COMPUTERS/COMPUTERS2-42.pdf

[6] User's Guide — NVIDIA Docs (2023) Matrix Multiplication Background, https://docs.nvidia.com/deeplearning/performance/pdf/Matrix-Multiplication-Background-User-Guide.pdf

[7] NVIDIA GA100 (2023) NVIDIA GA100, https://www.techpowerup.com/gpu-specs/nvidia-ga100.g931