GeneSys Documentation

GeneSys Overview

The GeneSys platform uses a systolic array centric approach that forms the core convolution engine for implementing DNN algorithms, and can be customized to run a range of standard DNN topologies.

Architecture

The overall system view of the GeneSys DNN accelerator is shown in Figure 1. The DNN accelerator consists of two core components:

1) The *systolic array* (Figure 2) that performs the convolution and matrix multiplication operations, targeting convolution and fully-connected layers.
2) A *SIMD Vector Unit* that takes care of DNN layers other than convolution and fully-connected layers, namely pooling, activation, and batch normalization.

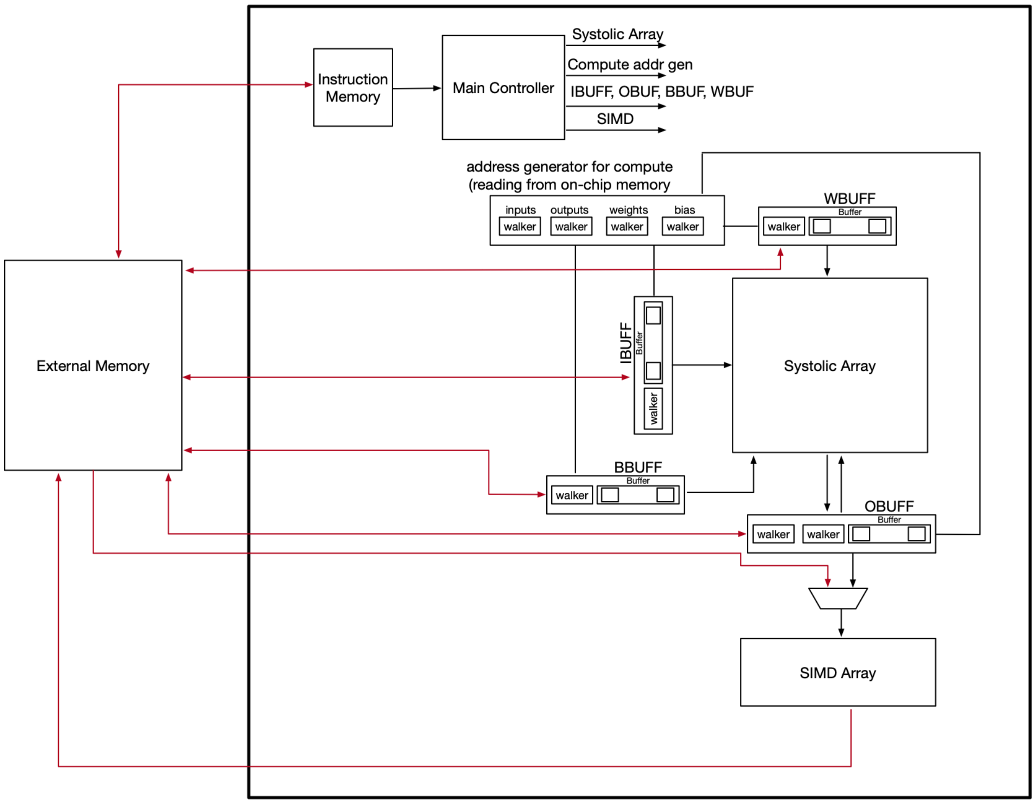

Figure 1: System organization of VeriGOOD-ML's GeneSys DNN accelerator

The SIMD Vector Unit also handles address generation for fetching the required datatypes of the accelerator (Input Activations, Weights, Partial Sums, and Output Activations) from the off-chip memory and writing to that. This reading and writing to off-chip memory happens through a programmable memory interface. The memory interface is also augmented with some second level on-chip buffers, namely, weight buffer (WBUFF), input buffer (IBUFF), output buffer (OBUFF), and bias buffer (BBUFF). The purpose of these buffers is to match the off-chip bandwidth with the consumption rate of the compute engines (Systolic Array and SIMD Vector Unit) to maximize both compute and off-chip bandwidth utilization. An address generator for on-chip accesses during systolic computation, the main controller and the instruction memory. The red lines in the figure denote the communication with external memory.

The main controller governs the execution of the DNN accelerator and orchestrates the communication between other components of the accelerator. At the bootup phase, the instructions will be written in the instruction memory. The main controller then starts executing the instructions. The initial body of instructions are dedicated to off-chip memory accesses for fetching a tile of inputs, weights and biases and writes them to their dedicated on-chip scratchpads.

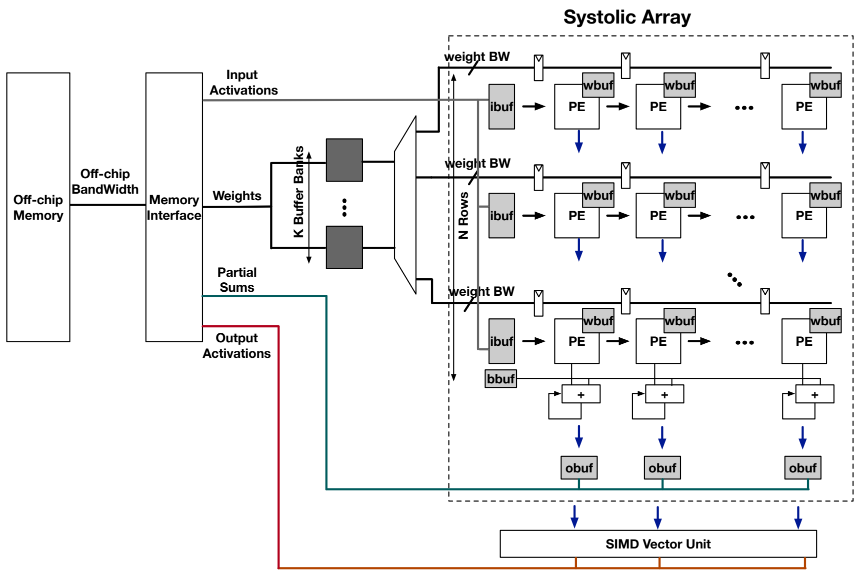

Figure 2: The block diagram of the overall system view of VeriGOOD-ML DNN accelerator and its systolic array

Systolic array

Architecture: As shown, the systolic array is a 2D array of PEs (N×M; N is the height and M is the width of the array), which are equipped with their dedicated on-chip weight buffers. The Input Buffer is multi-banked and each bank feeds a row of the systolic array. The output buffers are also multi-banked, each bank for each column of the systolic array, storing the partial sums and output activations.

Figure 3 depicts a more detailed diagram of the implementation of the Processing Engines in the systolic array. Each processing engine consists of (1) a weight buffer (scratchpad) that stores the weight values on-chip, (2) a multiply-accumulate unit that performs a multiplication between the inputs and weights and an accumulation of the partial results to perform the matrix-multiplication or convolution operation with the systolic array, and (3) a truncator to support lower bitwidth intermediate results, if the DNN can tolerate this level of accuracy. In addition to these components, each PE is equipped with four registers that aim to support the pipelined execution: a register for the output results, a register for the received input that will be forwarded to the adjacent PE in the systolic array, and two registers for handling the read accesses from the weight scratchpad (one register for the read request and one for the read address; the read request and read addresses for the weight scratchpads are shared across the 2D array of PEs).

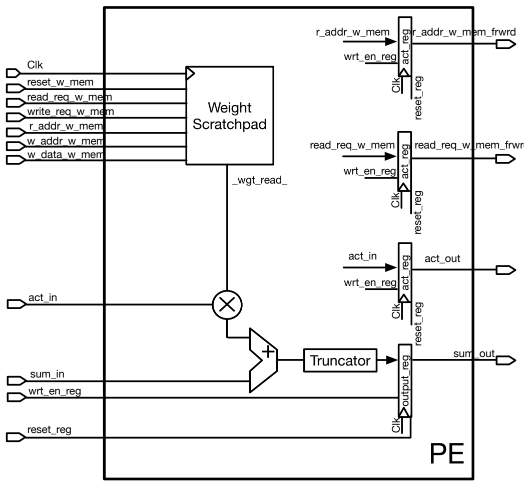

Figure 3: The detailed block diagram of a Processing Engine

We have developed a template Verilog implementation for the array and a Python-based generator that given a set of parameters list for the systolic array. We use these to generate a fully-functional systolic array as a synthesizable Verilog file. The systolic array is flexible in terms of the following key parameters and properties:

- The width of the array (M)
- The height of the systolic array (N)
- Precision of the datatypes used in the systolic array, such as input, weight, bias, partial sums, and the outputs
- On-chip storage for the systolic array consisting of four different types scratchpad buffers; weight buffer, input buffer, output buffer, and bias buffer.

To boost the maximum frequency of the computation, we enable fully pipelined datapaths for forwarding input activations along the rows and partial sums along the columns of the systolic array. We have also developed testbenches to verify the functionality of the systolic array.

ISA for Systolic Array: To enable a programmable systolic array that can fetch operand tensors (weights, inputs, biases) with any dimensions, perform the matrix multiplication/convolution, and store partial results or outputs with any dimensions, an Instruction Set Architecture (ISA) for the systolic array in the GeneSys chip has been devised, as overviewed in Table 1.

Systolic Array ISA									
Opcode (4 bits)		Op Spec (6 bits)						LOOP ID (6 bits)	Immediate (16 bits)
1001	SA_LOOP	LOOP LEVEL						LOOP ID	Immediate (# iterations)
1010	INST_GROUP	0	SYSTOLIC ARRAY	0	START	GROUP #		LOOP ID	
		1	SIMD ARRAY	1	END				# instructions(start), used for SIMD
1011	BLOCK_END	X		X		X		X	[0] shows if it is the last block or not
1100	SET_LOOP_STRIDE	0	LOW Bits	X		00 LD	00 WBUF	LOOP ID	STRIDE
						01 ST	01 IBUF		
		1	HIGH Bits			10 RD	10 OBUF		
						11 WR	11 BBUF		
1101	SET_BASE_ADDR	0	LOW PART	0	W/I/O/B	X	00 WBUF	X	BASE ADDR
							01 IBUF		
		1	HIGH PART	1	IMEM		10 OBUF		
							11 BBUF		
1110	LD_ST	0	LD	0	W/I/O/B	X	00 WBUF	LOOP ID	REQ SIZE
							01 IBUF		
		1	ST	1	IMEM		10 OBUF		
							11 BBUF		

Table 1: ISA for the systolic array.

GeneSys follows a block-structured ISA. In this format, a group of layers in the DNN can be fused together and compiled to a block of instructions. This instruction block is fetched from the off-chip memory and decoded for execution. The fetching of instruction blocks happens in double-buffering mode to hide the latency of fetching instructions from off-chip memory. GeneSys executes a block of instructions in tile-based format. As such the execution of instructions is repeated for a number of tiles and the cost of fetch and decode for instructions gets amortized across multiple tiles of data. The systolic array ISA consists of the following opcodes:

1) *SA_LOOP*: To enable repeated execution of single operation due to vectorized nature of execution in convolution/matrix-multiplication, we define the SA-LOOP instruction. In this instruction, we assign a LOOP-ID to a specific loop and define the number of iterations for that loop and its LOOP-LEVEL, which specifies the order of this loop with respect to the other loops. This instruction is used for fetching data from the off-chip memory (LOAD), performing the computations on the systolic array (RD/WR to/from the on-chip buffers) and storing the partial results or final outputs to the off-chip memory.
2) *INST_GROUP*: This instruction specifies a body of systolic array and SIMD array instructions and specifies that the following instructions are either for systolic array or SIMD array in a block of instructions. An instruction block can comprise multiple systolic array or SIMD array instruction groups.
3) *BLOCK_END*: This instruction specifies the end of the instruction block. The top module and the decoder will know by this instruction that the fetching and decoding of the block has ended and it needs to start the execution.
4) *SET_LOOP_STRIDE*: As mentioned, the whole execution of the systolic array happens based on the loop instructions. In this format, the address generation requires a base address and a loop stride (discussed in the previous reports). With this instruction we set the stride for the systolic array loop operations. This instruction specifies the stride for the corresponding loop (LOOP-ID), its corresponding on-chip buffer, and its usage for LOAD/ STORE or READ/WRITE to/from off-chip/on-chip memories.
5) *SET_BASE_ADDRESS*: With this instruction we set the base address corresponding to each datatype in the off-chip memory. This base address is used to fetch the first tile of data and the corresponding memory walkers will update it for the consequent tiles based on the number of tiles and their sizes. This instruction specifies if it is used for the data memories (WBUF, IBUF, OBUF, BBUF) or if it is used for the instruction memory (loading a new block of instructions).

6) *LD_ST*: This instruction is used for either loading or storing a tensor tile from/to off-chip memory for each of the buffers. This instruction specifies if it is used for LD or ST, data memories or instruction memories, the corresponding on-chip buffer, the loop that is used for LD/ST, and the the request size (the number of data that are fetched/stored with one address).

SIMD Array

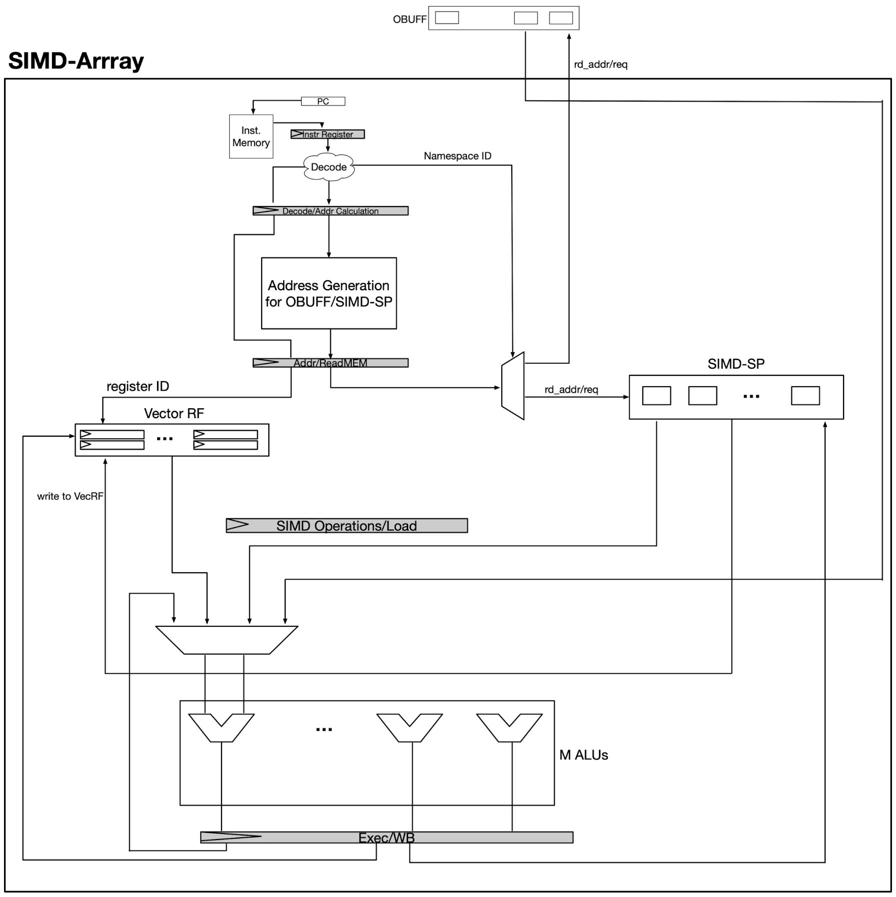

Figure 4: The high-level architecture of the SIMD array.

Architecture: The SIMD array is an important component of the GeneSys that targets the execution of other layers than convolution and matrix multiplication. This is, in a sense, a customized general-purpose processor that is capable of executing a more diverse series of operations that are required by DNN benchmarks with a SIMD vectorized execution patterns that is common across these operations.

The SIMD array can either access the data from external memory or OBUFF and write the results back to external memory. Figure 4 illustrates the high-level design of this component. As shown, this component follows a regular multi-stage pipelined processor. The first stage will be fetching the instruction from the instruction memory. Second, the instruction will be decoded. This decode stage mainly decides which ALU functionality needs to be executed in that instruction and from which namespace the data needs to be accessed. The third stage will be accessing the data from the namespaces. The SIMD array can access the data from three on-chip namespaces. The OBUFF, Vector Register File and the SIMD-SP. The SIMD array accesses the data from OBUFF when it performs the consecutive layers and operations after the systolic array. The SIMD array is also equipped with a dedicated scratchpad, SIMD-SP, to store the intermediate results of its computations there and read them back again for the rest of operations. Along with these two namespaces, the SIMD array can also utilize a Vector Register File for storing the datatypes that are required for the layers such as the parameters of batch normalization layer, and the scalar values for quantization. After accessing the data from the proper namespaces, an array of ALUs performs the required operations. The ALUs are programmable with a set of operations that are required such as add, subtract, multiply, max, min, etc. and work in parallel in a SIMD mode on the data. After finishing the computations and storing the data on the SIMD-SP to either be used by the SIMD array or sent to off-chip memory for the rest of the operations.

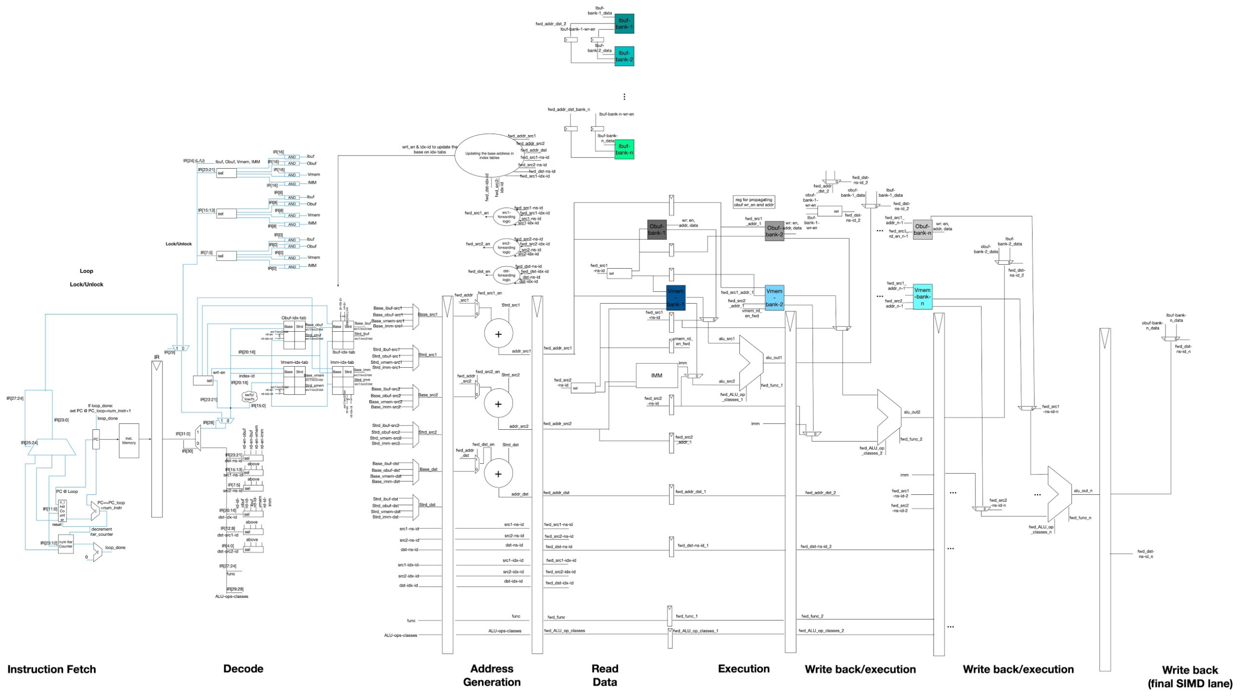

Figure 5: Architecture of the SIMD unit of GeneSys.

Figure 5 shows the pipeline stages of this SIMD processor, which is generally similar to a MIPS processor with a major difference: There is no register file in this design and since the memory access patterns are regular in DNNs, we eliminated the register file in this architecture to save Load/Store instructions. With this design, we directly read from the on-chip namespaces that store the data, execute the operations, and then write it back to the destination namespace. The stages in the pipeline are described below:

1) **Instruction Fetch:** In this stage the program counter reads the instructions from the instruction buffer one by one and sends them to the second stage for the decode. There is a peripheral

circuitry for handling the loop instructions. There is a counter that counts the iterations of the loop and another counter that counts the instructions in the body of the loop. When the body of the loop is being executed the program counter moves forward over the instruction buffer and the instruction counter counts the executed instructions. Whenever this counter reaches the value that corresponds to the (program_counter+number of instructions in the body of the loop), then the counter of the loop iterations decrements by one and the program counter will be set again at the start of the loop. This continues until all the iterations of the loop finishes.

2) **Instruction Decode:** The next stage is the decode. There are two sets of instructions for the SIMD array that will be discussed later: setup and execution. The blue lines illustrate the setup instructions that are executed in another path than the execution instructions. In the decode stage, it will be figured out which set of instructions are being executed. Moreover, all the control signals are generated in this stage. Another important event in this stage is the address generation. There are a set of index tables in this stage, each of which is associated with one on-chip memory namespace that the SIMD array deals with. These tables store the offset value and stride for each of the loop indices. To generate the addresses, based on the loop index ID and the namespace ID, the offset and stride value for each of the operands will be selected in this stage and sent to the next stage.
3) **Address Generation:** This stage harbors three adders to calculate the addresses for the two (or one) operands and the destination. As such three simple additions happen and will be sent to the next stage.
4) **Read Data:** In this stage multiple events happen. Based on the calculated addresses and the namespaces, the read addresses and requests will be sent to the pertinent namespaces. Also , the addresses will be sent to the index tables to update the offset values. Moreover, address forwarding will also happen in this stage, since the next instruction might require the current address as the offset value. There are multiple namespaces in this stage that the data can be read from: (a) VMEM (Vector Memory) which is a banked scratchpad that the SIMD array can use for storing intermediate and final results of the computations. (b) OBUFF (Output Buffer) which the SIMD array will read from it if it requires to execute the operations right after the systolic array. (c)IMM (Immediate) which stores the required immediate values.
5) **Execution:** The next stage is the execution stage that the ALU operations happen. Due to the pipelined nature of the systolic array and its corresponding buffers, mainly OBUFF and IBUFF, we interleave the ALU stage with the write-back stage. As such, when an instruction is being executed, the first SIMD lane will work on its execution, while in the next cycles the rest of the SIMD lanes will execute the instruction in a pipelined fashion. The first SIMD lane after finishing its execution will write back the result to its corresponding memory bank and will start executing the next instruction, while the rest of the SIMD lanes are working on the previous instructions. This ensures that the SIMD lanes are not stalled and the pipeline fashion both across SIMD array processor stages and the SIMD lanes are fully occupied.
6) **Write Back:** In this stage, the result of the instruction will be written to the target namespace, still in the aforementioned interleaved fashion.

The non-linear operations such as some of the activation functions (tanh, sigmoid, etc.), exponential function, logarithm etc. cannot be synthesized and implemented in the hardware and hence we approximate these operations using piecewise linear approximation or linear interpolation techniques. As an example, we describe tanh in more detail below.

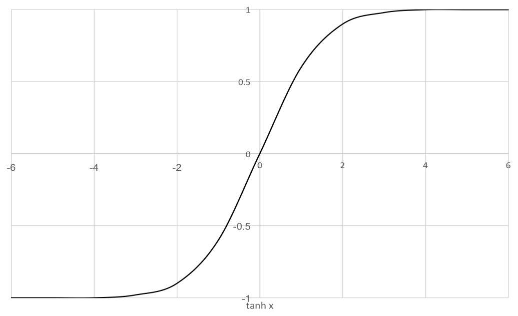

Figure 6: Input-to-output mapping for the tanh function.

As seen in the Figure 6, tanh has a steep slope in the domain [-1,1] while it saturates to 1 for positive x values and –1 for negative x. Therefore, we approximate tanh using the equation

$$y = \begin{bmatrix} x - \varepsilon & 0 \leq x \leq 1 \\ 1 & x > 1 \\ -1 & x < -1 \end{bmatrix}$$

Where ε is a delta term which is the difference between the actual nearest tanh output and x. Subtracting the ε term, which is fetched from a small (8 entry) LUT, results in adequate output resolution.

Instruction Set Architecture (ISA) for the SIMD array: To program this architecture, we have designed a custom ISA. Table 2 lists the instructions of the SIMD array. There are two classes of instructions in this ISA: Execution instructions, which are ALU, CALCULUS, COMPARISON, and DATATYPE CAST and setup instructions, which are DATATYPE CONFIG, LOCK NAMESPACE, ITERATOR CONFIG, and LOOP. Specifically,

- The instructions generally specify the destination namespace, the destination namespace ID, the operands namespace IDs, and their indices ID.
- The ALU operations require two operands for more simpler operations such as ADD, SUB, MUL, and etc. in addition to the MOVE instructions which are for moving the data between namespaces.
- The CALCULUS operations require one operand and are for performing more complex operations such as activation functions, exponential, logarithm, and division.
- The DATATYPE CAST, also requires one operand and it is for casting the data between various datatype representations (FXP to FP and vice a versa.)
- The LOCK NAMESPACE instructions are for locking namespaces and making sure that the rest of the components of the system will not access to that namespace while its data is not finalized.
- The ITERATOR CONFIG and LOOP instructions are for setting the loop counters and the base and strides for index iterators in the index tables.

Opcode (4 bits)		function (4 bits)		dest ns id (3 bits)	dest index id (5 bits)	src1 ns id (3 bits)	src1 index id (5 bits)	src2 ns id (3 bits)	src2 index id (5 bits)
0000	ALU	0000	ADD	Destination Namespace id	Destination Index id	Source1 Namespace id	Source1 Index id	Source2 Namespace id	Source2 Index id
		0001	SUB						
		0010	MUL						
		0011	MACC						
		0100	DIV						
		0101	MAX						
		0110	MIN						
		0111	RSHIFT						
		1000	LSHIFT						
		1001	MOVE						
		1010	COND MOVE						
0001	CALCULUS	0000	RELU	Destination Namespace id	Destination Index id	Source1 Namespace id	Source1 Index id	X	X
		0001	LEAKY RELU						
		0010	SIGMOID						
		0011	TANH						
		0100	EXP						
		0101	LN						
		0110	SQRT						
		0111	1/SQRT						
		1000	LOG2						
0010	COMPARISON	0000	EQUAL	Destination Namespace id	Destination Index id	Source1 Namespace id	Source1 Index id	Source2 Namespace id	Source2 Index id
		0001	NEQ						
		0010	GT						
		0011	GTE						
		0100	LT						
		0101	LTE						
0011	DATATYPE CAST	0000	32 FXP -> 16 FXP	Destination Namespace id	Destination Index id	Source1 Namespace id	Source1 Index id	X (IMM-ns-id)	Immediate namespace Index id
		0001	32 FXP -> 8 FXP						
		0010	16 FXP -> 32 FXP						
		0011	8 FXP -> 32 FXP						
		0100	32 FP -> 16 FP						
		0101	16 FP -> 32 FP						
0100	DATATYPE CONFIG	0000	32 FXP	X	X	X	X	X	Immediate namespace Index id
		0001	16 FXP						
		0010	8 FXP						
		0011	32 FP						
		0100	16 FP						
0101	LOCK NAMESPACE	0000	LOCK	namespace id	XXXX0 - DON'T APPLY XXXX1 - APPLY	namespace id	XXXX0 - DON'T APPLY XXXX1 - APPLY	namespace id	XXXX0 - DON'T APPLY XXXX1 - APPLY
		0001	UNLOCK						
0110	ITERATOR CONFIG	0000	BASE SIGNEXT	namespace id	namespace Index id	Immediate			
		0001	BASE LOW						
		0010	BASE HIGH						
		0011	BASE ZEROFILL						
		0100	STRIDE SIGNEXT						
		0101	STRIDE LOW						
		0110	STRIDE HIGH						
		0111	STRIDE ZEROFILL						
0111	LOOP	0000	SHORT LOOP	Number of Instructions (12 bits)			Number of Iterations (12 bits)		
		0001	LONG LOOP INST	Number of Instructions					
		0010	LONG LOOP ITER	Number of Iterations					

Table 2: An overview of the ISA for the SIMD array.

Scratchpads and memory interface: Each scratchpad is equipped with a memory-walker component along to its SRAM buffers to generate the addresses for off-chip accesses. The only exception is OBUFF which requires two memory-walker modules for both generating addresses to write to the off-chip memory and to generate addresses if some partial intermediate results have been transferred to external memory and need to be fetched from off-chip to perform the rest of the operations.

These memory-walker units are specially very useful in systolic execution for convolution/matrix-multiplication, since the memory accesses enjoy regular patterns and the memory allocation can be statically decided in the compile time. To use the memory-walkers, they first need to be programmed with a base-address that indicates the location of the first data in the memory, a stride parameter that indicates with what steps the data needs to accessed, and finally the number of iterations that address generation is required. With these parameters then the address generation can be realized easily with this notation: addr = base_addr + stride * i, where "i" denotes the current iteration. Once the memory-walkers are programmed with these parameters, they can easily generate the addresses for the required number of iterations. The main controller programs the walkers.

The compute-address-generator is responsible to generate the addresses for accessing the data stored on the on-chip scratchpads. This component encompasses four memory walkers for each of the scratchpads (WBUFF, IBUFF, BBUFF, OBUFF) which are programmed via the main controller. Once they are programmed the compute-address-generator starts the execution of the systolic array and generates the addresses to read weights, inputs, partial sums (stored in OBUFF), and biases to perform the computation and store the partial sums/outputs on the OBUFF. Then, the SIMD array will start executing the other required operations in DNN algorithms and perform them in a SIMD manner, including but not limited to quantization, ReLU, pooling, batch normalization, etc.

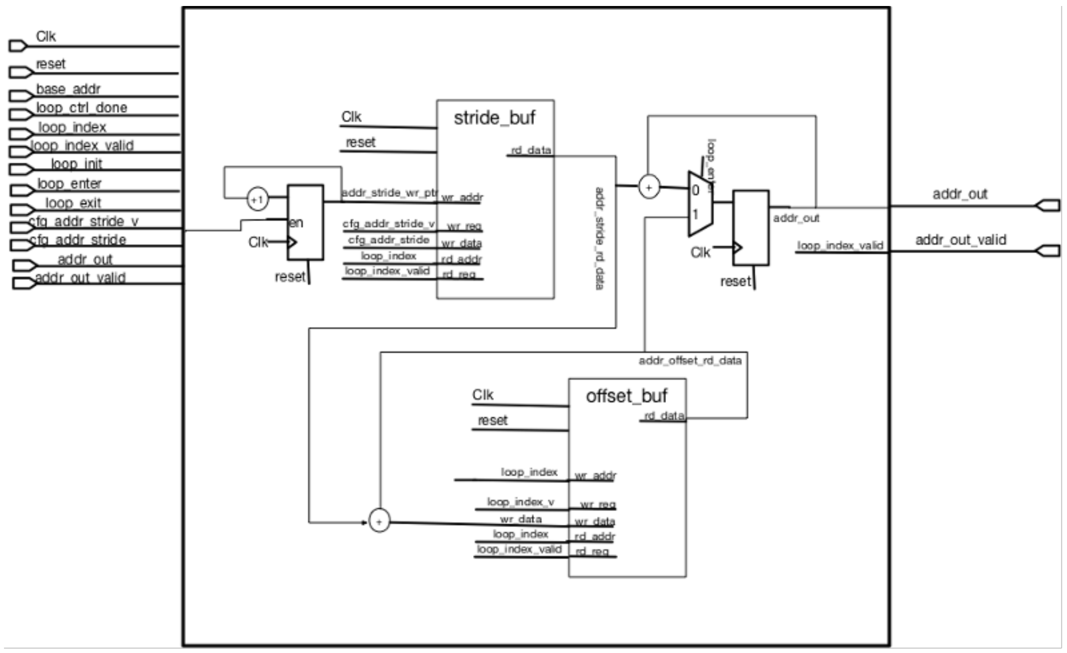

Figure 7: Memory walker-stride module.

Figure 7 depicts the block diagram of this module (memory-walker-stride) and its controller FSM. As the figure shows, this module constitutes a buffer that stores the stride for various loops, and another buffer to store the offset values. In each iteration, based on the current loop index (indicates the current loop that is being executed), the stride will be selected and then get added with the offset value. In the first iteration, the offset is the base_address and after each iteration it is being replaced with the current address value. As such, there is no need to perform multiplications, as we are updating the offset value every cycle. The memory walker module is also being controlled by a controller_fsm module, shown in Figure 8, a finite state machine that controls the current state (loop initialization, entering loop, exiting loop, and being in the inner loop). The controller_fsm module also has a buffer that stores the information about the loops and nesting, as well as the number of iterations for each loop.

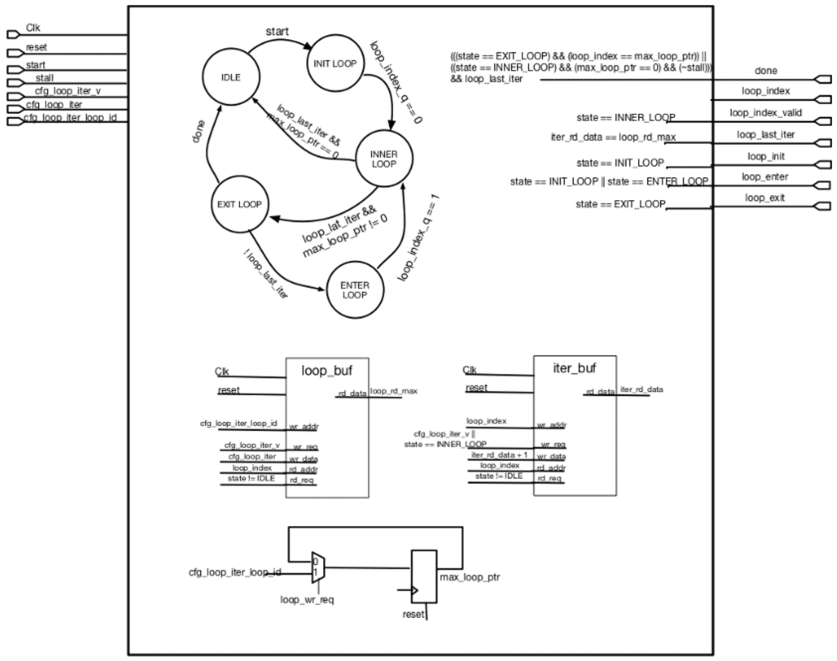

Figure 8: Finite state machine for the controller.

With these two components, based on the loop parameters, the addresses can be automatically generated for the systolic array and there is no need to specify them in the instructions. Each on-chip buffer (input buffer, weight buffer, etc.) is equipped with a walker to generate the read addresses for on-chip calculations, the output buffer is equipped with two walkers for generating both read/write addresses.

GeneSys Top Module

To sequence the execution of the GeneSys, we started the design and implementation of the top module for our DNN accelerator. This module encapsulates all the previously discussed modules, such as systolic array, SIMD array, on-chip buffers, address generators, etc. Following discusses the steps that have been taken so far:

1) *Instruction Loading*: As previously mentioned, GeneSys follows a block-structured ISA and execution semantics. An instruction memory is accounted for storing a block of instructions. This instruction memory is also coupled with a an AXI master interface that handles instruction fetching from the off-chip memory. In the beginning of each block, the instructions for fetching the next block are resident (a SET_BASE_ADDR instruction with a LD_ST instruction). The base address and the load information for the first block are communicated through an AXI Lite interface. Loading the instruction blocks happen in double-buffering mode to hide the latency of fetching the next block.

2) *GeneSys Instruction Decoder*: After finishing loading one block, the decoding phase starts. The instruction decode of a block happens all at once, in contrast to conventional processors that each instruction is fetched, decoded, and executed. During the decode, the instructions of the systolic array are all decoded and the corresponding memory walkers (address generators) are all configured. Different memory walkers are used for loading/storing and on-chip address generation during computations. The instructions of the SIMD array are all dispatched to the SIMD array and are stored on the SIMD array local instruction buffer. This is due to the different nature of execution in the SIMD array and systolic array. The SIMD array is mostly similar to a general-purpose processor that executes variegated instructions and fetch, decode, and execution happens instruction by instructions, as opposed to the systolic array which only performs regular matrix multiplications for multiple cycles. After decoding the next step is to design the off-chip interface that takes care of fetching data tiles for the corresponding block.

High-level testbench for GeneSys Systolic Array

To ensure that the integration has been correctly carried out and the systolic array is fully functional, we developed a high-level testbench that randomly generates a set of matrices, maps them to a set of nested loops, programs the memory walkers of the associated on-chip buffers and then tests the execution of the systolic array by validating the outputs of the systolic array with the result of the matrix multiplication done by the software.