Base ISA:

The SPAR ISA follows the encoding of a 32-bit MicroBlaze instruction. Table 1 shows the mapping of SPAR instructions to MicroBlaze instructions.

Table 1- SPAR-2 ISA Instructions Mapping to MicroBlaze.

Instr. Type	MicroBlaze				
	0 5	**6 10**	**11 15**	**16 20**	**21 31**
A-Type Arithmetic	Op_code	Reg_d	Reg_s1	Reg_s2	00000000000
A-Type Load/Store	Op_code	Reg_d	Address		00000000000
	SPAR-2				
	31 26	**25 21**	**20 16**	**15 11**	**10 0**
Execute	Op_code	Reg_d	Reg_s1	Reg_s2	00000000000
Read/Write	Op_code	Reg_d	Address		00000000000

1) Execute

execute (int opcode, int Reg_s1, int Reg_s2, int Reg_d)

Description: Performs the arithmetic operation specified by the opcode on register Reg_s1 and Reg_s2 of all the PEs of the processor array and stores the result in Reg_d.

Table 2- Bit-serial Instructions Description.

Instruction	Opcode	How it is called	What it does	Clock Count (n -bit data-width)
Addition	0	Execute (0, Rs_1, Rs_2, R_d)	R_d = Rs_1 + Rs_2 for all PEs	$2n$
Subtraction	1	Execute (1, Rs_1, Rs_2, R_d)	R_d = Rs_1 - Rs_2 for all PEs	$2n$
Multiplication	2	Execute (2, Rs_1, Rs_2, R_d)	R_d = Rs_1 * Rs_2 for all PEs	$n^2 + 2n$
Shift Right	5	Execute (5, Rs_1, 0, R_d)	moves R_s1 of all PEs to R_d of their right PE	n
Shift Left	6	Execute (6, Rs_1, 0, R_d)	moves R_s1 of all PEs to R_d of their left PE	n

Shift North	7	Execute (7, Rs_1, 0, R_d)	moves R_s1 of all PEs to R_d of their above PE	n
Shift South	8	Execute (8, Rs_1, 0, R_d)	moves R_s1 of all PEs to R_d of their below PE	n
Relu	9	Execute (9, Rs_1, 0, R_d)	if R_s1 > 0 R_d = R_s1 else R_d = 0	$2n$

2) Read/ Write

int Read (int Tile_i, int Tile_j, int Block_i, int Block _j, int addr)
void Write (int Tile_i, int Tile_j, int Block _i, int Block _j, int addr, int value)

Description: This function reads or writes a given value into a specific address of the specified BRAM using the built-in function of MicroBlaze for accessing BRAM. In this function, the Xil_in and Xil_out functions are called to read or write into the given address of the BRAM block that is specified by its Tile number (Tile_i, Tile_j) and its block number (Block _i, Block _j) within that Tile.

Software Macros:

The software macros implemented in C language on the MicroBlaze processor are divided into three groups: LSTM/MLP macros, CNN macros, and some general macros used in both. Hereafter, Reg_s1 means the register number for the first source operand, Reg_s2 means the register number for the second source operand, and Reg_d is the register number for destination operation. Array means the whole 2-D processor array.

Table 3- Software Macros Description.

Macro Name	How it is called	What it does	What Instructions are used in it	Where it is used
LSTM and MLP Macros				
Elementwise_Addition	$(in\ R_{s_1}, in\ R_{s_2}, in\ R\_d)$	Elementwise addition on two registers (R_{s_1}, R_{s_2}) of the whole array and storing the result in R_d	1 Addition	LSTM Gate MLP Node
Elementwise_Multiplication	$(in\ R_{s_1}, in\ R_{s_2}, in\ R\_d)$	Elementwise multiplication on two registers (R_{s_1}, R_{s_2}) of the whole array and storing the result in R_d	1 Multiplication	LSTM Gate MLP Node

Matrix-Vector Multiplication	$(in\ R_{s_1}, in\ R_{s_2}, in\ R_d, in\ matrix\_sises)$	Matrix Multiplication on Matrix stored in $R_{s_1}$and the vector stored in R_{s_2}. Storing the result vector in R_d register (assuming matrix size $m * n$ and vector size $n * 1$)	1 Multiplicaiton $\log(n)$ Addition $n-1$ Shift	LSTM Gate MLP Node
Column to Row	$(in\ R_{s_1}, in\ R_d, in\ AF)$	Moves values in R_{s_1} from the last column of PEs of the array to the Sigmoid or Tanh activation functions (determined by AF), and storing the result into the first row of the PEs in R_d register	1 Shift East 1 Shift South	After Matrix Multiplication when performing activation functions
Row to Row	$(in\ R_{s_1}, in\ R_d, in\ AF)$	Moves values in R_{s_1} from the first row of the array to the Tanh activation functions (determined by AF), and storing the result into the first row of the PEs in R_d register	1 Shift North 1 Shift South	In LSTM Cell when applying Tanh on a vector
Gate	$(in\ W, in\ X, in\ U, , in\ H, in\ matrix\_sises)$	Calls macros needed to implement any of four (i, g, o, c) LSTM gates (σ(WX+UH+b)).	2 Matrix Mult 2 Elmnt_Add 1 Col_to_Row	LSTM Cell
CNN Macros				
Convolution	$(in\ R_{s_1}, in\ R_{s_2}, in\ R\_d)$	Performs convolution algorithm between (R_{s_1}, R_{s_2}) and storing the result in R_d (assuming output kernel size $n * n$)	1 Multiplicaiton $n-1$ Shift $n-1$ Add	CNN layer
Max Pooling	$(in\ R_{s_1}, in\ R_d)$	Max pooling on R_{s_1} and storing the result in R_d	$n-1$ Max $n-1$ Shift $n-1$ Add	CNN pooling layer
Relu	$(in\ R_{s_1}, in\ R_d)$	Applying Relu on R_{s_1} and storing the result in R_d	$n * n$ Relu	CNN activation function
General Macros				
Write Matrix	$(in\ M, in\ M_{size}, in\ mode, in\ R\_d)$	Writes the given matrix or vector M into R_d, based on the provided mode (either as a 2-D matrix into the whole array, or into last column or into the first row of the array) (assuming M has n elements)	n BRAM Write	For writing network params into BRAMs

LSTM Macros:

1) Elementwise_Addition

void Elementwise_Addition (int Reg_s1, int Reg_s2, int Reg_d)

Description: Performs elementwise bit-serial addition between vector elements stored in Reg_s1 and Reg_s2 and stores the result in Reg_d. This function calls the instruction:

execute(0, Reg_s1, Reg_s2, Reg_d)

This operation is applied to all PE's in the processor array.

2) Elementwise_Multiplication

void Elementwise_Multiplication (int Reg_s1, int Reg_s2, int Reg_d)

Description: This function is used when two vectors are multiplied. It includes a single bit-serial multiplication between the equivalent elements of the input vectors that are stored in Reg_s1 and Reg_s2registers. This function calls the instruction:

execute(2, Reg_s1, Reg_s2, Reg_d)

This operation is applied to the whole processor array PEs.

3) Matrix_Vector_Multiplication_Optimized

Matrix_Vector_Multiplication (int Reg_s1, int Reg_s2, int Reg_d, int col_s1, int arr_size)

Description: This function performs a matrix-vector multiplication on given registers that are stored throughout the processor array PEs. How to write the values into the right PEs will be more explained in the last section. The first operation for performing a matrix multiplication is a parallel multiplication between the elements of the matrix and the equivalent vector elements. After this, partial products should be added. This is performed using a binary tree reduction that shifts (moves) the partial products to the right and adds them together. This operation is conducted as the number of columns of the matrix. In this step, if the array size is larger than the matrix column, the result would be in a middle column of the array, so it is shifted right to reach the last column of the processor array. This means the result of a matrix-vector multiplication would always be in the PEs of last column of the processor array.

Figure. 1 shows the execution of a typical matrix-vector multiplication (MVM) operation as the main operation of LSTM/GRU/MLP networks within SPAR-2. For example purposes, we show a small $3 * 4$ matrix W multiplied by vector X resulting vector Y $(4 * 1)$. In this Fig. 1(a) shows how the weight matrix W and is partitioned into distributed BRAMs of the 2-D array and how elements of vector X are mapped and replicated into the array. One SIMD parallel multiplication is performed (Fig. 1(b)) on all PEs, generating all partial products in one instruction. The addition of partial products is then followed using shift_and_add operations (Fig. 1(b)). For this step, the addition of partial products uses the binary reduction tree shown in Fig. 1(b). The final output from the multiply-accumulate step is then sent to the output buffer (Fig. 1(c)). The larger matrix multiplications that could not be fit into the 2-D processor array are implemented using partitioning so that each partition is stored in

a different register and merged after matrix multiplication to form the output (refer to Matrix_Vector_Multiplication_Large section).

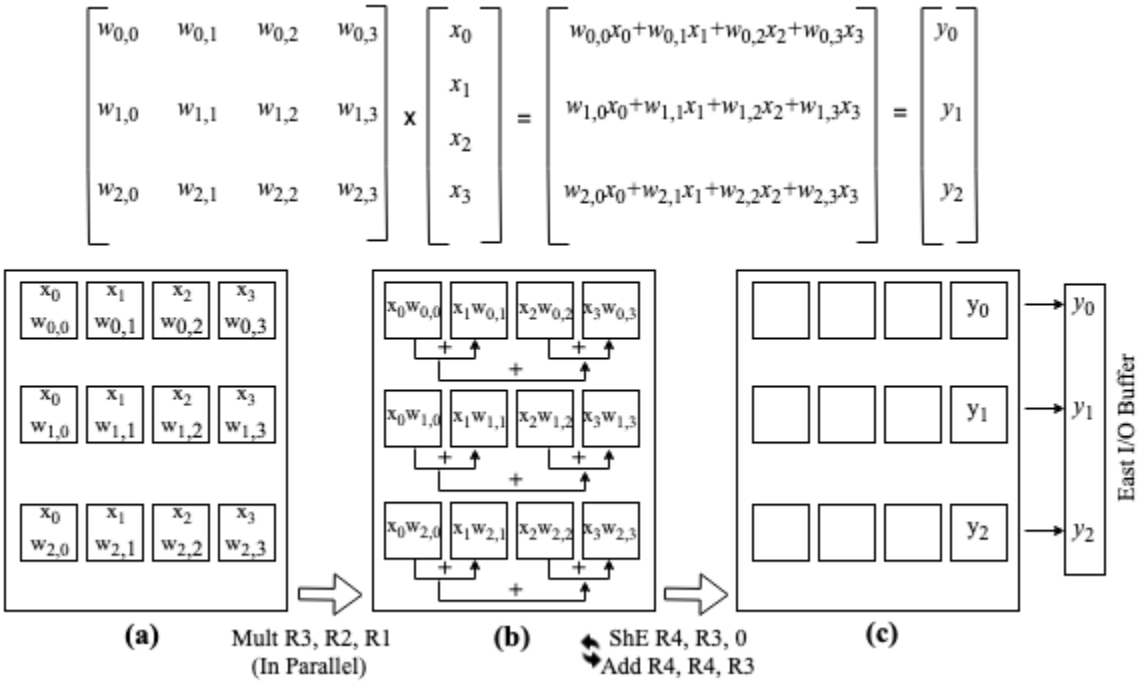

Figure 1- Mapping Matrix Multiplication to the Processor Array.

4) Matrix_Vector_Multiplication

This function operates the same as the number (3), except in this function the shift and add operation is conducted using regular single-hop method, not binary tree.

5) Matrix_Vector_Multiplication_Large

Matrix_Multiplication_Large (int Reg_s1, int Reg_s2, int Reg_d, int col_s1, int arr_size, int sub_size_height, int sub_size_width)

Description: This function is used when multiplying a matrix that its dimension is larger than the processor array. In this case, the matrix is divided into smaller sub-matrixes, and then the results are merged.

Shown in the Fig. 2, assuming the processor array size is 20 * 20, the matrix is 60 * 40, and the vector is 40 * 1. They are divided into sub-matrixes of 20 * 20. The Matrix_Vector_Multiplication_Optimized function discussed in (3) is called on R1 and R7, and then on R2 and R8. The result of this matrix multiplication is stored in two different registers, and they are added and stored in the result register. This method is performed to the rest of the sub-matrixes, and the result would be in three different registers in this case. Therefore, in case of large matrix and vectors, we utilized more than one register to store the inputs and outputs. The sub_size_height and sub_size_width inputs to the function determined the dimension of the sub-matrixes (in this example: 20 and 20). In the current implemented function, the size of sub-matrixes should be the same for all sub-matrixes, but they do not have to be squares.

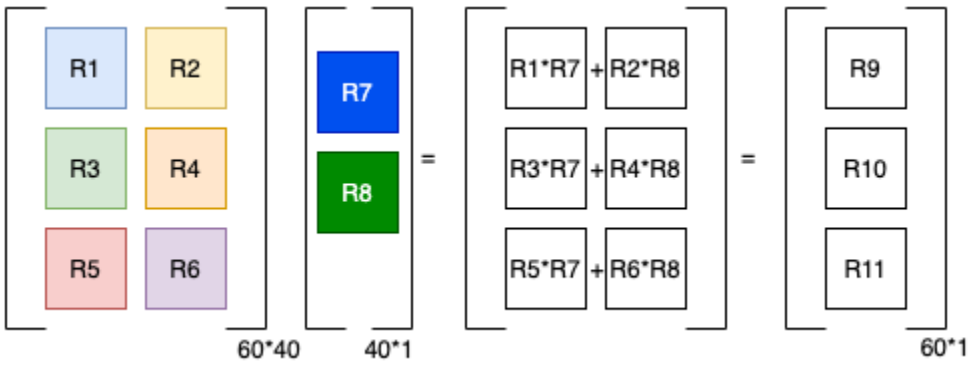

Figure 2- Mapping Large Matrix Multiplication to the Processor Array.

6) Column_to_Row

Column_to_Row (int Reg_s1, int Reg_d, int AF)

Description: In this function, the values of a given register (Reg_s1) are moved from the last column of PEs into the first row of PEs. While the values are moved, the activation functions can also be called to perform the Sigmoid or Tanh on the values of the last column and store the outputs into the first row. This function is used in the LSTM Gates (3 Sigmoid and 1 Tanh gates). This is conducted using a shift east operation that moves the values of the last column into the Parallel_Serial_Convertor modules. The parallel output of these modules is the input to the activation function modules. Then, the parallel output of the activation functions is fed to the north Parallel_Serial_Convertor module to convert it from parallel to serial and store the result into the first row of the PEs using a shift south operation. So, this function includes a shift east, performing the activation functions, and then a shift south operation. The AF input to this function selects between Sigmoid and Tanh modules.

7) Row_to_Row

Row_to_Row (int Reg_s1, int Reg_d, int AF)

Description: In this function, the values of the first row go through the activation functions and the result is stored back in the first row. This function is used in the LSTM cells when performing Tanh on a vector. This is similar to what is explained in (6). The difference is that in this function, the values before and after the activation function are stored in the first row.

8) Gate

Gate (int X_col, int X_row, int W_col, int W_row, int U_col, int U_row, int H_col, int H_row, int b_col, int b_row, int Reg_X, int Reg_W, int Reg_U, int Reg_H, int Reg_b, int Reg_Result, int array_size, int AF)

Description: This function implements the four gates of an LSTM cell. It includes two matrix-vector multiplication (WX and UH) and addition with bias (WX + UH + b). This function is called four times for i, f, g, o gates. In this function, the matrix W is multiplied by vector X, and U is multiplied by H. This is done by calling the Matrix_Vector_Multiplication function discussed in (3). The result of this matrix multiplication would be in the last column of processor array in two different registers. Then, the b vector, which is also stored in the last column, would be added to the result for WX + UH. At the end, by calling the Col_to_Row function, the values of WX + UH + b will be moved from the last column into the first row of PEs, to be ready as the input to the next time step. The activation function modules which are implemented outside the processor array can be called after reading the values from the last column and before storing them back into the first row. The AF input to the Gate function selects between Sigmoid and Tanh to be applied on the WX + UH + b before moving it to the first row.

9) **Gate_Large**

This function works similarly to the Gate function (8), except inside this function instead of calling Matrix_Vector_Multiplication, Matrix_Multiplication_Large function is celled.

CNN Macros:

The operations are needed in the CNN networks are coded without using separate macros as they depend to the benchmark. Here we provide a summary of the operations. The first operation is a convolution algorithm. Fig. 3 shows how a simple small CNN network is mapped and executed on SPAR-2. Shown in this figure, the input feature maps of three nodes are mapped into the PEs of the processor array. Assuming a 6 * 6 feature map that produces a 4 * 4 output feature map (kernel_size = 3, padding = 0, stride = 1). The three 6 * 6 feature maps are mapped into 12 * 12 PEs (small squares in Fig. 3 are PEs). The convolution algorithm is applied using a single multiplication on all PE inputs stored in R1 and weights in R2. Partial products are stored in R3. In the following step, the partial products are added using shift_and_add operations with results stored in R4, R5, R6 for each feature map. This is shown in Fig. 3(a) for only the first feature map. The other two follow the same operations and results saved in R5, R6. It should be noted that no additional read and write is required to move the highlighted values into the top-left of the processor array as the final results are mapped and saved in the correct position. Finally, by adding R4, R5, R6 (Fig. 3(b)), the output feature map is stored in R4 (Fig. 3(c)). The bias parameter is added in the next step and the Relu function is applied. The pooling (if any) is then applied to the output feature map (Fig. 3(c)) (not represented in this simple example). In a SIMD architecture, the convolution algorithm is conducted in parallel over the number of nodes for each CNN layer. For larger CNNs where feature maps do not fit into available processor array, multiple registers are used to store the input feature maps. This reduces the transfer latency swapping feature maps between DRAM and BRAM. For example, assuming feature maps of 6 * 6 on an array of 12 * 12 PEs, if the number of input nodes is 5, the first 4 feature maps can be stored in R1 (same as the prior example) and their associated weights stored in R2. However, the last feature map cannot be stored in R1 as the array is fully utilized. In this case, the last feature map can be stored in a different register (e.g. R10) on the top-left position of the processor array. The convolution algorithm is first applied on registers

R1, R2 (for nodes 1-4) with the inputs in R1 and their associated weights stored in R2. A second convolution can then be applied on R10, R2 for node 5. In essence, a virtual array larger than the physical array can be defined, with each physical PE operating as multiple virtual PEs. The max pooling and the Relu functions are then applied on the red area in the below figure which is the output of the convolution layer.

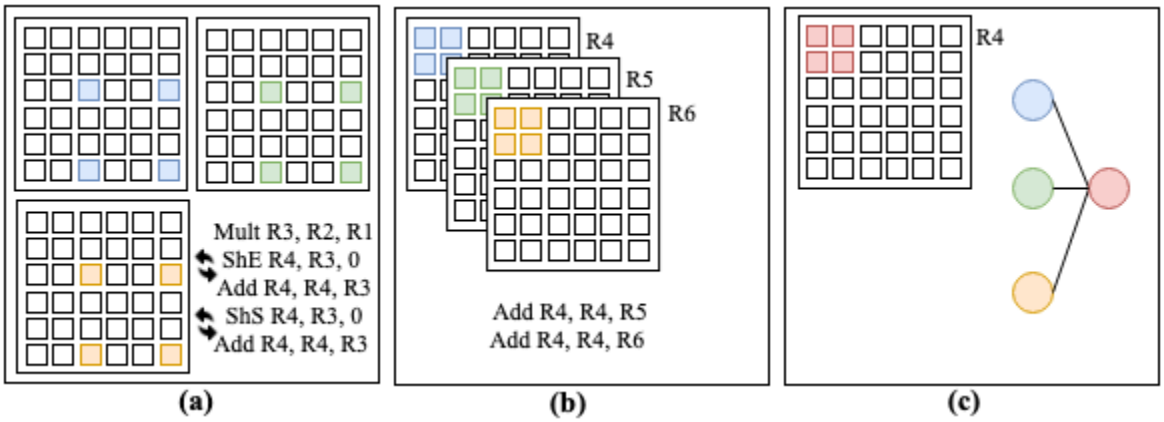

Figure 3- Mapping CNN to the Processor Array.

General Macros:

1) Write_Matrix

Write_Matrix (int row, int col, int M[][col], int reg, int mode)

Description: This function writes the values of a matrix or a vector into the right PEs and registers. There are four options inside this function and can be set as an input. Based on the value of the mode input, this function writes the matrix M in one of the four methods. The first method is writing it as a 2-D array throughout the processor array PEs. This option is used when, for example, writing the U matrix into the array. The next one writes a vector into the last column of the PEs and is used when writing a bias vector (b) vector in the LSTM networks. Another option is writing a vector into the first row of the PEs and copying it into the below PEs. This is used when writing the X or H values of an LSTM gate. In the next option, the vector is written into the first column of the processor array that is not currently used. In all these options, based on the row number and column number of the matrix's elements, the correct Tile, Block, and PE number is computed and then the function Write/Read (in Table. 1) is called to write that specific element into the right position.

2) Write_Matrix_Large

This function operated similar to (1), except since the matrixes and vectors, in this case, are larger than the processor array, they are divided into sub-matrixes, and so each time this function is called, the values of different smaller parts of the large matrix are written into the proper register of the processor array.