On-Chip Neural Chess Analyzer (ONe-ChAn)

Haihan Wu
Massachusetts Institute of Technology
Southern University of Science and Technology
Cambridge, MA
haihanwu@mit.edu

Muhammad Abdullah
Massachusetts Institute of Technology
Cambridge, MA
abd880@mit.edu

***Abstract*—We present a design for a Hardware based Chess Engine with uses a TPU.**

***Index Terms*—Digital Design, Chess Engine, Tensor Processing Unit, hardware accelerator, negamax algorithm, FPGA**

I. Introduction

The system consists of two parts, a Tree Traversal module, and a TPU. Both of these systems reside on different FPGAs and we use an SPI interface to communicate between them. The FPGA responsible for tree traversal also handles IO with the user by using switches to take input moves, buttons to run *undo* and *perform_move*, and the seven-segment display is used to show a row of the chess board under consideration. In the end, we plan to show the best move on the LED lights.

The user can scroll up and down using buttons to see different chess rows and make the move.

II. Tree Traversal

A. Stack and Board

The current game is represented by a 64 element 8-bit array. Each entry is a one-hot encoding of the piece and color at that position.The algorithm always assumes that it is the player white, but there are ways to get results from the perspective of black by making a series of moves with an odd length.

The module interacts with the player using switches. The player inputs a move and sends it to the board controller and starts the tree traversal module. The module goes through the entire algorithm and outputs the calculated next move using the led lights.

B. Step-Step-Spray Algorithm

This algorithm is used to step through a move graph, like Fig 1, perform the negamax algorithm for two player games, and generate packets for the TPU. It has three states of importance; *step_up, step_down, spray*. The traversal is done in the step stage where we do a depth first search of the move-tree by stepping up and down the graph. The stack state is shown for the traversal of Fig 1 in the table.

The module, in the stepping stage, interfaces with the move generator and uses the signal *step* to ask for a single move on the current board. It then adds the move to the stack or jumps up if there is no move left.

When ever the stack is full, in the example when the size is 3, we transition into the spray state. The spray state interfaces with the move generator using the signal *spray*. The move generator returns all possible moves on the current board. The traversal module then packages these moves in a packet and sends them to the TPU.

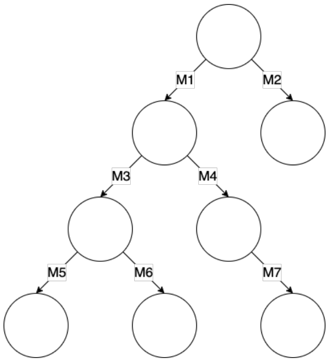


Fig. 1: An example move search tree where each node is a board position and the edges are all the moves from that position

Once the TPU returns, we take the evaluated value and save it on the stack. So the negmax algorithm is run on the stack as we go through it.

State	Current Stack
Down	ϕ
Down	M1
Down	M1M3
Down	M1M3M5
Up	M1M3
Down	M1M3M6
Up	M1M3
Up	M1
Down	M1M4
Down	M1M4M7
Up	M1M4
Up	M1
Up	ϕ
Down	M2
Up	ϕ

C. Move Generator

The move generator takes the current board position and tries to find a piece of the color that is supposed to play.

QB	N	QR	N	QB
N	KQB	KQR	KQB	N
QR	KQR	from	KQR	QR
N	KQB	KQR	KQB	N
QB	N	QR	N	QB

Fig. 2: The possible moves for each piece type from a starting position. The format is similar to Belle [2], an FPGA based chess engine from the 1990s

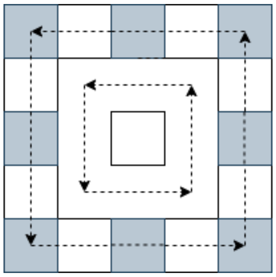


Fig. 3: The order of traversal of the moves, the shaded areas check for blocking in the inner ring

It interacts with the minmax algorithm using *step, spray* wires which cause it to search for a move from a starting coordinate..The starting coordinate is stored in *from* register, it is incremented until a valid piece is found. Then we generate 25 values of $\delta x, \delta y \in \{-2, -1, 0, 1, 2\}$ in the order shown in Fig 3. We reject the moves that are not valid for that particular piece using the map given in Fig 2. Furthermore, when we are traversing the outer ring, we additionally check for blocking friendly or enemy pieces in the inner ring.

Another important aspect of the system of move generation are the starting values of the *from* register and $\delta x, \delta y$. They allow stateful traversal of moves and are used to ensure that successive *step* activations cause new moves to be generated. The *from* register is initialized from the stack's top values and the $\delta x, \delta y$ are stored on the stack and initialized from there. The utilization of the entire move-genrator module is 2% of LUTs and no BRAM

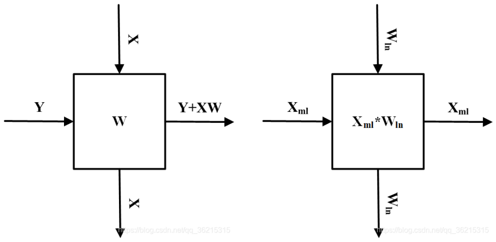
III. Tiny TPU

Deep neural networks (DNNs) are utilized for many artificial intelligence applications because it performs more accurate results on different AI tasks, however, the computation complexity is high. The main DNN computation consists of convolution, matrix multiplication, max-pooling and nonlinearity, and matrix multiplication and convolution consume a large amount of computation.

To boost DNN computations, the tiny TPU in Fig 5 incorporates design concepts and methods of hardware accelerators in Prof Sze's survey paper [1], and simplifies the processor design based on the needs of analyzing chess to achieve faster evaluation of the next move. The whole TPU design is uploaded on the Github.

A. Systolic Array and Working Principles

1) Systolic Array: The systolic array can efficiently implement matrix multiplication in hardware. Google Tensor Processing Unit(TPU) uses a systolic array architecture. A systolic array consists of processing elements (PEs), and the simplest processing element is divided into weight-stationary in Fig. 6(a) and output-stationary in Fig. 6(b):



(a) Weight stationary PE (b) Output Stationary PE

Fig. 6: Two typical types of the processing elements

A systolic array composed of weight-stationary PEs needs to store the weight matrix in each processing element in advance. The processing element will add the partial sum transmitted from the previous processing element to the product of the input and weight of this element, and then transfer the accumulated sum to the next processing element. Therefore, the weight-stationary systolic array will output the result matrix in the form of pulse data. The Google first generation of TPU utilized this category of PE. On the contrast, the output-stationary processing element accepts input and weight, and continues to accumulate the product of different input and weight, and the result is stored in the processing element.

In terms of convolution, several ways are mentioned in Professor Sze's survey paper about DNN accelerator designs [1]. One way is to flatten the weight kernel and transform the two-dimensional input matrix, and then multiply the two transformed matrices(Fig. 7(a)). However, it's less efficient especially when ifmap is very large. Another method is to use fast Fourier transform (Fig. 7(b)).

In this project, tiny-TPU uses a similar approach to fast Fourier transform: first, the weight kernel is placed in the weight-stationary pulse array, and then the number of rows of the same height as the weight kernel are continuously extracted from the input and transformed into systolic data, which is sent to the pulse array; after that, the elements of all rows of the output array are added, and after filtering, a row of output is obtained. Repeating the above operation can obtain the result of the convolution.

2) Working Principles: Instructions of TPU are stored at instruction block ROM in the program counter, layer information, weights and biases are stored in the main memory. The most critical part of TPU is to achieve convolution and matrix multiplication, and those two operation will be controlled by special instructions.

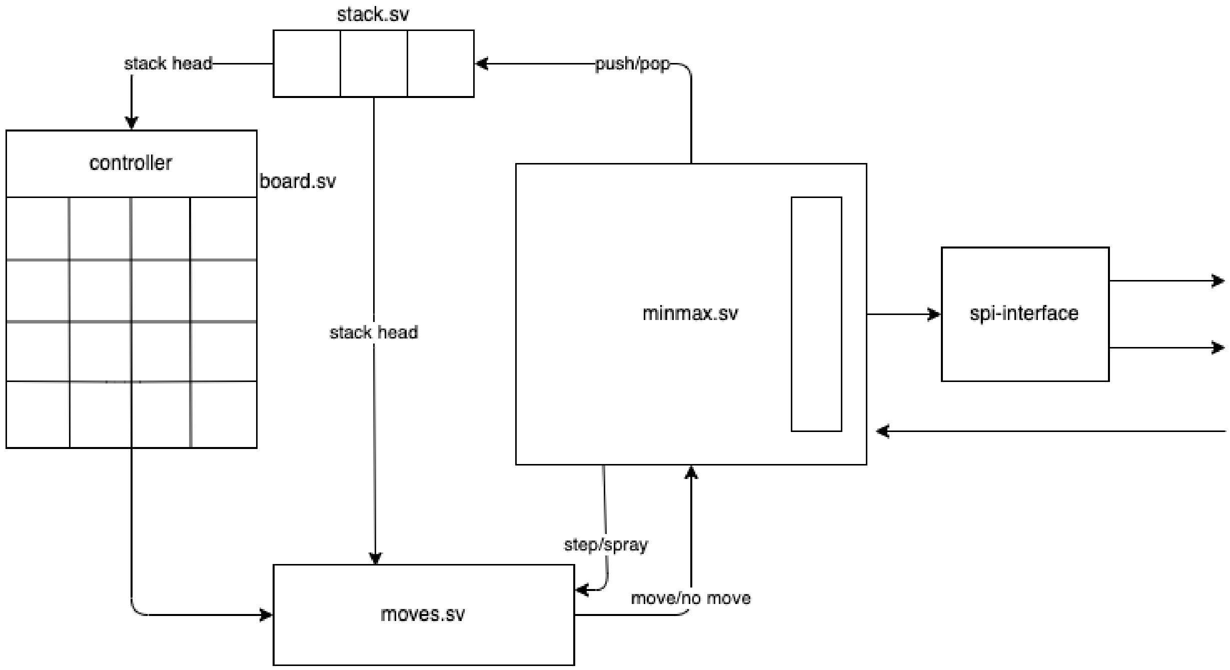


Fig. 4: Overview Block Diagram for the Tree Traversal module

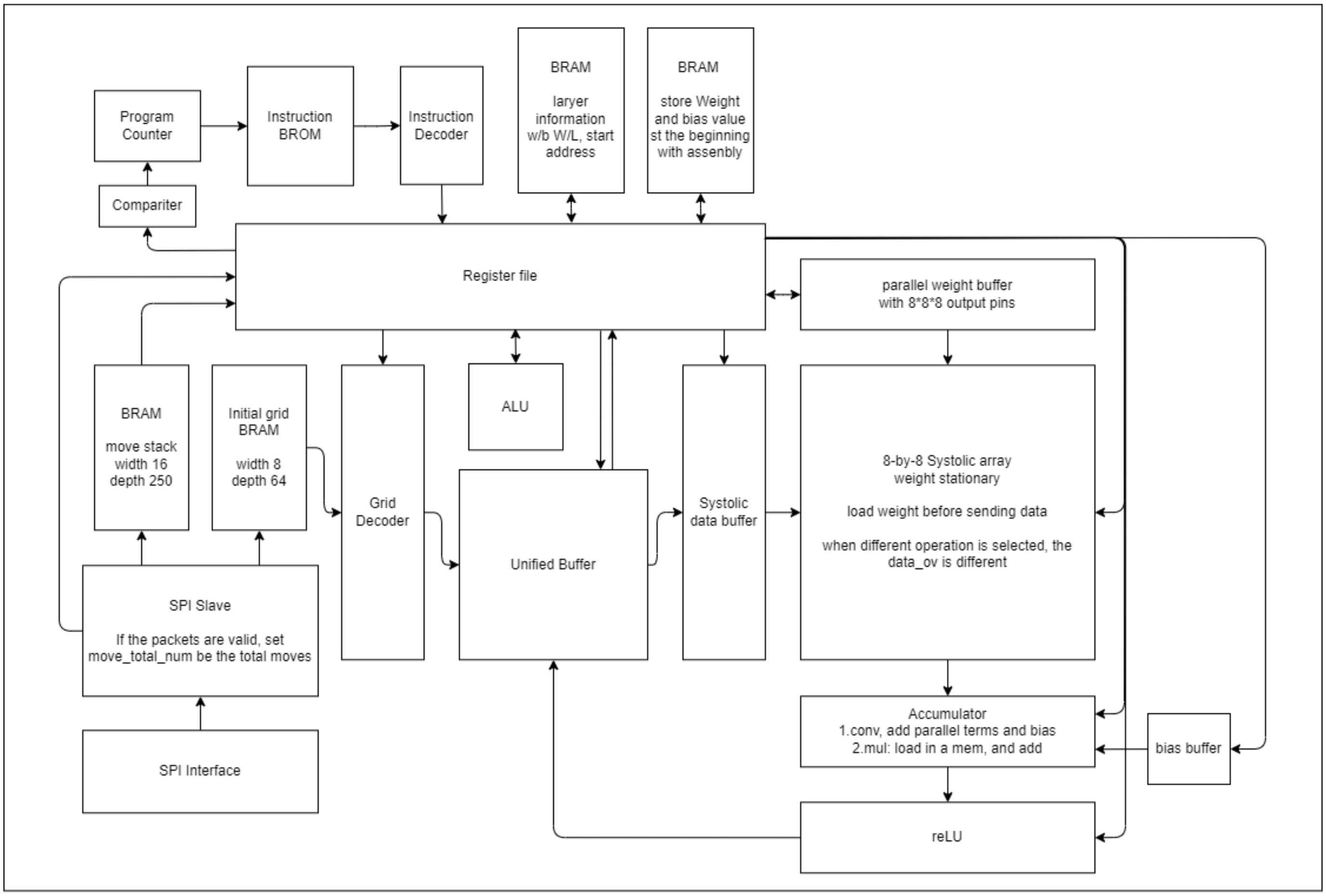
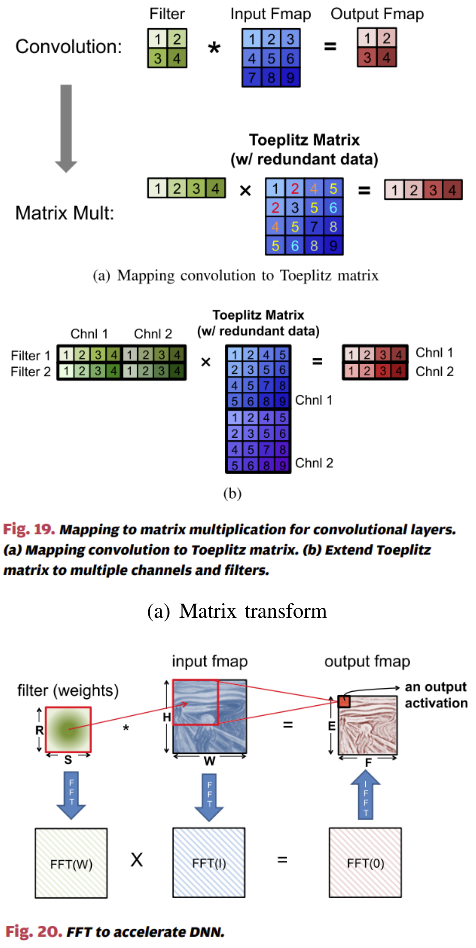

Program Counter
Instruction BROM
Instruction Decoder
BRAM
laryer information w/b W/L, start address
BRAM
store Weight and bias value st the beginning with assenbly
Compariter
Register file
parallel weight buffer with 8*8*8 output pins
BRAM
move stack width 16 depth 250
Initial grid BRAM
width 8 depth 64
Grid Decoder
ALU
Unified Buffer
Systolic data buffer
8-by-8 Systolic array weight stationary
load weight before sending data
when different operation is selected, the data_ov is different
SPI Slave
If the packets are valid, set move_total_num be the total moves
SPI Interface
Accumulator
1.conv, add parallel terms and bias
2.mul: load in a mem, and add
bias buffer
reLU

Fig. 5: Block diagram of tiny TPU



(a) Matrix transform

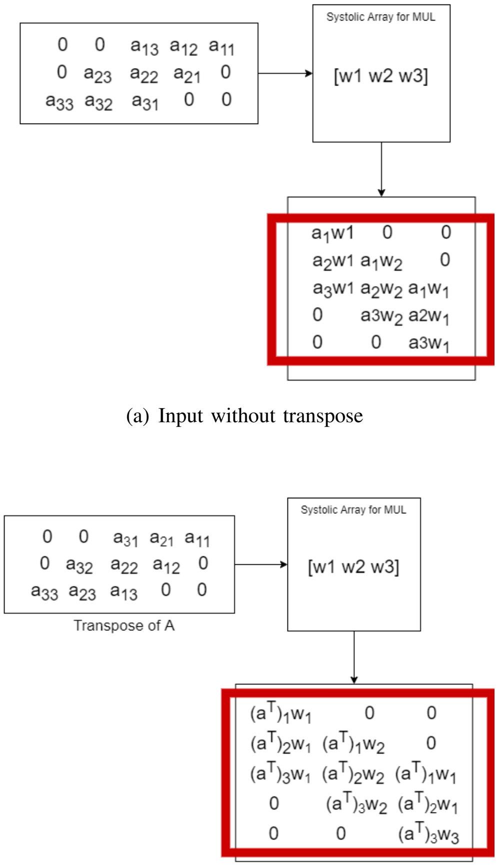
(b) Fast Fourier transform

Fig. 7: Two approaches mentioned in the survey paper written by Prof. Sze [1]

The register file module has 3 groups of 64 registers: The first group of registers stores the number of moves and data relevant to deep learning model, such as the length, height, and corresponding starting address of weights and bias, the type of operation between layers, and whether non-linearity is used. The second and the third group of registers are loaded with weights and bias respectively.

Fig 8 illustrates the dataflow of matrix multiplication. The input is a systolic data of 3 rows generated from input matrix A. The weights W are pre-stored in systolic array and output are at the bottom. All the output elements are represented by the column vector inner products as $a_i{\cdot}w_j$. The outputs are valid inside the red rectangles. For multiplication, all of the systolic output are valid and will be further processed into a matrix. If we want to compute AW, one thing should be noticed that each output element is inner product of row of A and column of W in Fig 8(b) (If the input is not transposed, then the outputs are A^TW). So the input matrix should be transposed and then converted to systolic data form.

As to Convolution, if weight matrix is 3-by-3, then the



(a) Input without transpose

(b) Input with transpose

Fig. 8: Tiny TPU matrix multiplication

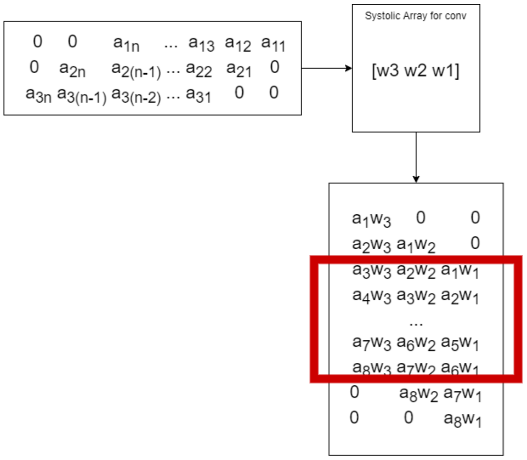


Fig. 9: Tiny TPU convolution

buffer sends 3 rows of the input in systolic data form each time. On Fig. 9, one interesting thing is that the weight matrix stored is horizontally flipped. The reason is that the valid output of convolution is (a_1w_1+a_2w_2+a_3w_3), (a_2w_1+a_3w_2+a_4w_3) ... ($a_{n-2}w_1$+$a_{n-1}w_2$+a_nw_3), and the output on each column is the inner products of the input column and the corresponding weight column, e.g., the outputs of the first column are dot-products of input columns and $\mathbf{w}_1$. To get the correct sum of those inner products, the weight matrix should be horizontally flipped before loading into the systolic array. And the output of cnovolution will be further processed in the accumulator module.

What's more, the tiny TPU is created with basic functions of general-purpose processor such as arithmetic, branch, jump and load function. When the TPU receives packet from the move generator through SPI interface, it will check the packet first. If the received data complies with the encoding, then the TPU stores the initial grid in distributive ram and all the possible moves in move stack. After that, the nonzero total move number is directly loaded in the register file module. The first line of the program is

while(move_total_num==0) {}

so the DNN computation program begins, otherwise it will halt the program counter until receives a valid packet and change the total move number.

The tiny TPU design below will follow this principle and realize DNN computation.

B. Computation Architecture

The more detailed computing architecture is shown in the following figure.

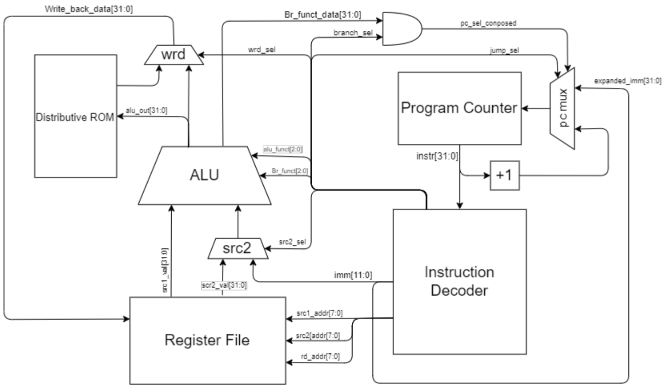
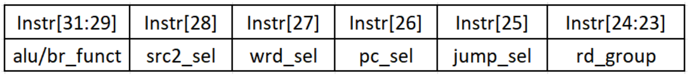


Fig. 10: TPU single-cycle computation architecture

One good thing of this processor is that it can transmit data in parallel. For example, when the second groups of the registers updates the weights from the main memory, all 64, 8-by-8 weight matrix can be loaded in the systolic array module simultaneously. Compared to a general-purpose processor, this processor greatly improves the efficiency of data transmission.

This processor use single-cycle pattern instead of pipelines, and there are several reasons for this: For one thing, most importantly, is that some of the instructions take more than tens or even hundreds of clock cycles to complete, and those instruction execution dominates the performance. The fig. 11 proves this. The 3 signal, **load_weight**, **load_bias** and **send_systolic_data** are decoded from **instr[31:0]**. Apparently, those three multi-cycle instructions take most of the time

The arithmetic, branch, jump and load functions are realized with this single-cycle structure. The selector signals are decoded from instruction. **src2_sel** determines whether uses the immediate. **wrd_sel** selects write back data from ALU or main memory. And the branch MUX uses **pc_sel** or **jump_sel**. **alu_funct**(or **br_funct**) chooses the operation in ALU. They are encoded in instruction in the following manner below. And the detailed instruction set architecture is provided in Fig 13.



Instr[31:29]	Instr[28]	Instr[27]	Instr[26]	Instr[25]	Instr[24:23]
alu/br_funct	src2_sel	wrd_sel	pc_sel	jump_sel	rd_group

Fig. 12: TPU instruction encoding

C. Layer Information Encoding

There are two types of layer information in total: layer number information and layer information. Layer number information is a 32-bit integer storing total layer number, the height and width of the input layer. It's encoded in a following manner in table I. After layer information is loaded from the main memory, the TPU can decode it into input layer width, input layer height and total layer number at instruction **decode_layer**.

[31:28]	[37:24]	[23:20]	[19:0]
input_height	**input_width**	**total_layer_num**	empty

TABLE I: Layer number information Encoding

32-bit layer information are relevant to layer computation (Table II). The number of layer information integers is determined by total layer number. It contains the shape of the weight and bias kernels, and mapping starting address in the main memory. If instruction **load_weight** is received by the register map module, the TPU will load weights from the base address to the sum of base address and the product of height and width. The special instruction **load_bias** works in a similar manner. Special operation such as flatten, rectified leaky unity useage is also recorded in the layer information.

D. Instruction and Main Memory Generation

All the instructions are generated by the python scripts if all the weights, biases and neural network information is known. Instruction memory has a width of 12, which is sufficient for deep networks with hundreds of layers. Main memory has the same depth as instruction memory. Layer-relevant information, weights and biases are stored in 0x000-0x0FF, 0x100-0x1FF and 0x200-0x2FF respectively.

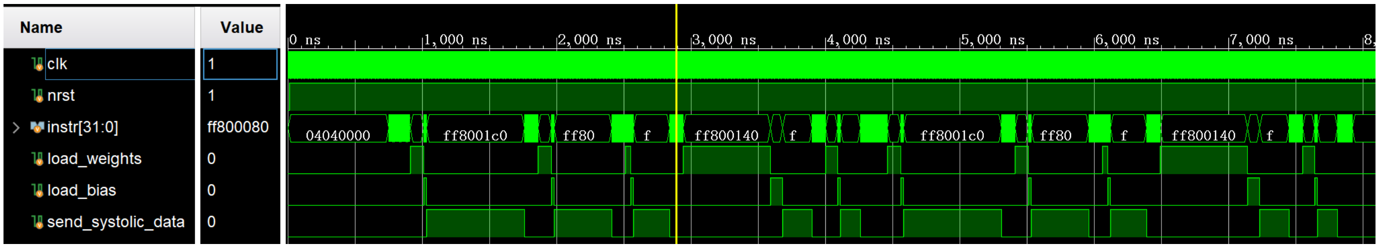


Fig. 11: TPU single-cycle simulation

Instruction type	Instruction	Instr[31:0]					
ARITHMETIC	ADD	000000000	src1_addr[4:0]	src2_addr[4:0]	0000000		rd[5:0]
	MUL	001000000	src1_addr[4:0]	src2_addr[4:0]	0000000		rd[5:0]
	SHL	010000000	src1_addr[4:0]	src2_addr[4:0]	0000000		rd[5:0]
	SHR_A	011000000	src1_addr[4:0]	src2_addr[4:0]	0000000		rd[5:0]
	BAND	100000000	src1_addr[4:0]	src2_addr[4:0]	0000000		rd[5:0]
	BXOR	101000000	src1_addr[4:0]	src2_addr[4:0]	0000000		rd[5:0]
	BOR	110000000	src1_addr[4:0]	src2_addr[4:0]	0000000		rd[5:0]
	ADDI	000100000	src1_addr[4:0]	imm[11:0]			rd[5:0]
	MULI	001100000	src1_addr[4:0]	imm[11:0]			rd[5:0]
	SHLI	010100000	src1_addr[4:0]	imm[11:0]			rd[5:0]
	SHRA_I	011100000	src1_addr[4:0]	imm[11:0]			rd[5:0]
	BANDi	100100000	src1_addr[4:0]	imm[11:0]			rd[5:0]
	BXORI	101100000	src1_addr[4:0]	imm[11:0]			rd[5:0]
	BORI	110100000	src1_addr[4:0]	imm[11:0]			rd[5:0]
BRANCH	EQ	000001000	src1_addr[4:0]	src2_addr[4:0]	0	label[11:0]	
	NE	001001000	src1_addr[4:0]	src2_addr[4:0]	0	label[11:0]	
	GE	010001000	src1_addr[4:0]	src2_addr[4:0]	0	label[11:0]	
	LE	011001000	src1_addr[4:0]	src2_addr[4:0]	0	label[11:0]	
	GT	100001000	src1_addr[4:0]	src2_addr[4:0]	0	label[11:0]	
	LT	101001000	src1_addr[4:0]	src2_addr[4:0]	0	label[11:0]	
	NEG	110001000	src1_addr[4:0]	src2_addr[4:0]	0	label[11:0]	
JUMP	JUMP	111000111				label[11:0]	
LOAD	LW_0	000110000	src1_addr[4:0]	offset[11:0]			rd[5:0]
	LW_weight	000110001	src1_addr[4:0]	offset[11:0]			rd[5:0]
	LW_bias	000110010	src1_addr[4:0]	offset[11:0]			rd[5:0]
SPECIAL	decode_layer	111111111				0000	
	compute_grid	111111111				0001	
	decode_layer_info	111111111				0010	
	compute_ifmap	111111111				0011	
	send_layer_info	111111111				0100	
	load_weight	111111111				0101	
	load_bias	111111111				0110	
	send_systolic_data	111111111				0111	
	set_ifmap_o	111111111				1000	
	send_optimal_move	111111111				1001	

Fig. 13: TPU Instruction Set Architecture

[31:29]	[28:26]	[25:18]	[17:15]	[14:12]	[11:4]	[3:1]	[0]
weight_height-1	**weight_width-1**	**weight_start_addr**	**bias_height-1**	**bias_width-1**	**bias_start_addr**	**{reLU_sel, op_sel, flatten}**	empty

TABLE II: Layer number information Encoding

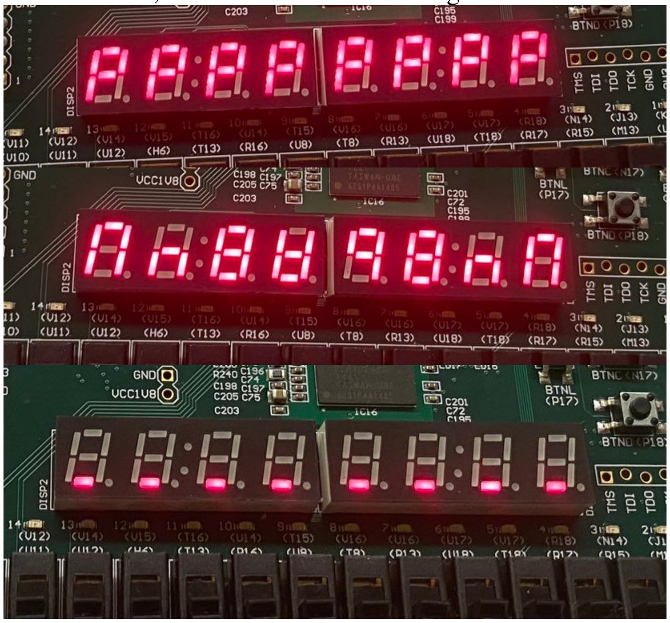
E. *Evaluation from RTL simulation*

To evaluate the performance based on the current utilized DNN, a valid packet consisting of a initial grid and five moves is sent to the TPU through the SPI interface. And it takes 19 μs to finish all the computation. The results is acceptable compared with the one generated by the python script. The output of the TPU is 0x57(87), which is very closed to the 83.3 computed by python.

IV. Realization

A. *Grid visualization with 7-segment displayer*

The grid is visualized on the 8-bit 7-segment displayer. One number shows one chess piece. BTNU and BTND are used to scroll through different rows. The following figure shows the 1st, 2nd and 3rd rows initial grid with the button.



B. *Progress*

The TPU and the move generator both pass the RTL simulation, however, there area some communication error between these two section which still need more time to figure out. Although we are not able to should how this detects the optimal move, the algorithm and TPU can function separately, especially the TPU is capable to compute different types of the neural network with a relatively high accuracy.

In the future, the author will focus on the compiler, which can generate the instruction in machine language with a program. Secondly, the TPU is not capable of dealing with convolution with multiple channel, so the 3-D convolution will be further investigated and implemented.

As for the move generator, we are able to generate correct moves in hardware and display them to leds. The tree traversal module works in simulation but there seem to be some timing issues which hinder it to be combined with the move generator.

References

[1] V. Sze, Y. -H. Chen, T. -J. Yang and J. S. Emer, ”Efficient Processing of Deep Neural Networks: A Tutorial and Survey,” in Proceedings of the IEEE, vol. 105, no. 12, pp. 2295-2329, Dec. 2017, doi: 10.1109/JPROC.2017.2761740.

[2] https://en.wikipedia.org/wiki/Belle_(chess_machine)

[3] https://github.mit.edu/abd880/one-chan/tree/master/mov-gen

[4] https://github.mit.edu/abd880/one-chan/tree/master/whh_tpu_v1