

Customized Instruction on RISC-V for Winograd-Based Convolution Acceleration

Shihang Wang¹, Jianghan Zhu¹, Qi Wang¹, Can He¹ and Terry Tao Ye^{1,2*}

¹Department of Electrical and Electronic Engineering,

Southern University of Science and Technology, Shenzhen, China

²University Key Laboratory of Advanced Wireless Communications of Guangdong Province,

Southern University of Science and Technology, Shenzhen, China

Email: yet@sustech.edu.cn

Abstract—Convolution operation accounts for the major workload in convolutional neural networks (CNN). However, standard instruction set for RISC-V processor cannot efficiently perform the matrix convolution between kernel and input matrices. In this paper, we construct a custom instruction under the RISC-V ISA that can perform the $F(2 \times 2, 3 \times 3)$ convolution within one single execution. Particularly, optimized by the Winograd algorithm, the operation only needs 16 multiplications instead of 36 multiplications as needed by standard ISA. Benefit from this new instruction, $F(2 \times 2, 3 \times 3)$ can be calculated within 19 clock cycles, as compared to 140 cycles using standard instructions. The power consumed during convolution operation is also reduced significantly.

Index Terms—RISC-V, CNN, Winograd, Acceleration

I. INTRODUCTION

RISC-V architecture had demonstrated great potentials in recent years. Benefit from its open-source instruction set architecture (ISA), it enables the development of various Application Specific Instruction-Set Processors (ASIP) that can be tailor-made for different applications. RISC-V standard ISA only defines some basic instructions that are both aligned and compact, allowing for efficient decoding and processing. In the meantime, the ISA is also very flexible for new instruction extensions and additions [1]–[3].

In recent years, many artificial intelligence applications, such as image/audio/video rendering and processing tasks are implemented on edge computing platforms using CNN architecture [4], [5]. As RISC-V gains its popularity in edge computation platforms, CNN implementation on RISC-V processor had attracted many interests both from research institutes as well as industries [6], [7]. Convolution computation accounts for the majority, in many cases, over 90% of the workload in CNN. Under standard RISA-ISA pipelines, CNN calculation needs constant fetching and data loading from cache and memory, followed by ALU execution and writing-back. The process is inefficient and time consuming; it also consumes huge hardware overhead and power consumption.

In this paper, we have designed a custom instruction extended from RISC-V ISA that can effectively perform the convolution between kernel and input matrices. We also adopt the Winograd algorithm to optimize the convolution calculation [8], and implement the convolution operation, i.e., $F(2 \times 2, 3 \times 3)$ in Verilog and encapsulate the calculation process into one instruction. $F(2 \times 2, 3 \times 3)$ is the matrix operation that a kernel of the size 3×3 convolutes with an input matrix of

4×4 and generates an output matrix of 2×2 . With Winograd optimization, the calculation only needs 16 multiplications, as compared to 36 multiplications using traditional direct dot-product approaches. The instruction can be called directly from RISC-V CPU, and the total instruction can be executed within 19 clock cycles, as compared to 140 cycles if $F(2 \times 2, 3 \times 3)$ is implemented by standard RISC-V ISA.

We use an open-source RISC-V core, RI5CY, as the baseline CPU. The standard RISC-V ISA, as well as our new custom $F(2 \times 2, 3 \times 3)$ instruction, are all implemented in Verilog and synthesized into FPGA platform NEXYS A7. The execution latency, as well as power consumption, with and without the custom instruction set, is also measured. The experiment results demonstrate that our proposed new instruction not only can speed-up the convolution operation, it also saves power consumption in performing the CNN tasks.

The paper is organized as follows. Section II introduces the methods to add custom instructions in RISC-V ISA. Section III describes the Winograd algorithm that can be used to accelerate the convolution calculation. FPGA implementation and performance analysis will be in Section IV, followed by the conclusion in Section V.

II. ADDING CUSTOM INSTRUCTIONS

The RISC-V ISA supports multiple instruction subsets, and different instruction functions can be realized by configuring different instruction subsets. Different instruction subsets are suitable for different tasks. For example, the basic instruction set (I instruction subset) supports operations including addition, subtraction, shift, XOR and other logics. Integer multiplication and division instruction subsets (M instruction subset) can provide operations including multiplication and division operations. If a 32-bit RISC-V ISA supports a subset of I and M, it is conventionally named as RV32IM.

RI5CY is a four-stage pipelined single-issue RISC-V ISA processor that supports the RV32IMC instruction set [9]. RISC-V ISA is very flexible in adding new instructions, the ISA reserves four opcodes for the users to define additional custom instructions in the opcode map, as shown in TABLE 1.

We use the “custom-1” reserved opcode in TABLE I to encode the proposed custom convolution instruction. We name the instruction “CONV23”, with the opcode 0x2B.

The main function of CONV23 is to perform a convolution calculation between a 4×4 input matrix and a 3×3 convolution

TABLE I
RISC-V OPCODE MAPPING TABLE ,(INST[1:0]=11)

Inst[6:5] \ Inst[4:2]	000	001	010	011	100	101	110	111 (>32b)
00	LOAD	LOAD-FP	custom-0	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	custom-1	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	reserved	custom-2/rv128	48b
11	BRANCH	JALR	reserved	JAL	SYSTEM	reserved	custom-3/rv128	≥ 80b

kernel matrix and generates a 2×2 output matrix. The input matrix is stored in the memory before the convolution operation, and the elements of the convolution kernel are stored in the convolution acceleration module.

There are six types of RISC-V instruction set encoding formats [10], as shown in Fig. 1. Each format is intended for different functions. For example, J-type format is for out-of-order instructions, R-type format is used to perform logical operations between values stored in two registers respectively, while I-type format is used to perform logical operations on immediate data and values stored in registers. Our proposed CONV23 performs convolution calculation, the elements of matrices are from memories and cache, not from immediate values, instead the instruction needs two source registers to store the size of the input matrix and the first address of memory that stores the input matrix data. So CONV23 uses the R-type instruction format.

	31	25	24	20	19	15	14	12	11	7	6	0
R-type	funct7	rs2	rs1	funct3	rd	opcode						
I-type	imm[11:0]	rs2	rs1	funct3	rd	opcode						
S-type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode						
B-type	imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:0]	imm[11]	opcode				
U-type	imm[31:12]				rd	opcode						
J-type	imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode						

Fig. 1. RISC-V instruction format.

The instruction format of CONV23 is shown in Fig. 2. The *rs1* register and *rs2* register are two source registers, and the processor will perform operations based on the values in these two registers, more specifically, the *rs1* register stores the size of the convolution input matrix, and the *rs2* register stores the value of the first address of the input matrix in memory. The *rd* register is the destination register; it can be any register in CONV23 because CONV23 will not write back any value to the *rd* register. The three bits in *funct3* represent *rd*, *rs1* and *rs2* from left to right, which indicate the usage of *rs1*, *rs2* and *rd*. If the values of *rs1*, *rs2* or *rd* is not used, the corresponding bit in *funct3* is set to 0, otherwise it is 1. *Funct7* is used to distinguish different functions of the same opcode. Since there are no other functions, this value is 0.

	funct7	rs2	rs1	funct3	rd	opcode
CONV23	0000000	address	size	011	*	0x2B

Fig. 2. CONV23 instruction format.

In order for RI5CY processor to recognize CONV23, the decoding unit needs to be modified. Once the decoder unit encounters the CONV23 instruction, the convolution acceleration module is executed to perform the convolution operation.

III. CONVOLUTION ACCELERATION MODULE

A. Winograd algorithm

Convolution acceleration module implements the CONV23 function once the instruction is called by the CPU. Winograd algorithm is used to accelerate the convolution calculation.

The Winograd minimal filtering algorithm is a multiplication-reduction method for optimizing convolution calculations [11]. It is more efficient when the kernel size is small. We will first use a simple example to explain the algorithm, and then derive its general formula.

For a convolution between a 4-element input vector $d = (d_0, d_1, d_2, d_3)$ and a 3-element filter vector $g = (g_0, g_1, g_2)$, it will generate a 2-elements output. This convolution can be denoted as $F(2, 3)$. The first number “2” indicates the size of the output, and the second number “3” indicates the size of the filter.

The 4-elements of the input can be folded into a two-dimensional matrix $\begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix}$. The convolution operation can then be transformed into the multiplication of two matrices, as shown in the Eq. 1.

$$F(2, 3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_0 + m_1 + m_2 \\ m_1 - m_2 - m_3 \end{bmatrix} \quad (1)$$

Where

$$m_0 = (d_0 - d_2)g_0 \quad m_1 = (d_1 + d_2) \frac{g_0 + g_1 + g_2}{2} \quad (2)$$

$$m_3 = (d_1 - d_3)g_2 \quad m_2 = (d_2 - d_1) \frac{g_0 - g_1 + g_2}{2} \quad (3)$$

The $\frac{g_0 + g_1 + g_2}{2}$ and $\frac{g_0 - g_1 + g_2}{2}$ can be calculated before convolution, so 4 multiplications and 4 additions needed to calculate the terms m_0, m_1, m_2 and m_3 . Additionally, 4 additions are needed to calculate the results from m_0, m_1, m_2 and m_3 . After the conversion, 4 multiplications and 8 additions are required.

The Winograd algorithm involves three steps. 1) Convert the input data and filter into the matrix forms, 2) Calculating the intermediate terms using Equation 2. and Equation 3. and 3) Adding and subtracting from the intermediate terms to generate the final results.

In general, Winograd convolution can be formulated into matrix operations, defined as follows:

$$Y = A^T[(Gg) \odot (B^T d)] \quad (4)$$

In this equation, d and g are the input vector and the filter vector respectively. A , B and G are the transformation matrix of d and g . A , B and G are defined as follows.

$$A = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}^T \quad B = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}^T \quad G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

For a one-dimensional convolution operation with m -elements output and r -elements filter, it can be denoted as $F(m, r)$ only $m + r - 1$ multiplications are needed.

We can derive the two-dimensional $F(2 \times 2, 3 \times 3)$ from the one-dimensional $F(2, 3)$, where the A, B, and G in equation 4 are changed into two-dimensional forms. As shown in Equation 6.

$$Y = A^T [(GgG^T) \odot (B^T dB)]A \quad (5)$$

Using Winograd algorithms, the convolution operation between a 4×4 input matrix and 3×3 filter matrix generating a 2×2 output matrix, i.e., denoted as $F(2 \times 2, 3 \times 3)$, requires a total of 16 multiplication operations and 85 addition operations.

In comparison, traditional convolution using direct dot-product operation requires 36 multiplication operations. Winograd algorithm can reduce the number of multiplication by a factor of 4/9, or 55.5%.

$F(2 \times 2, 3 \times 3)$ can be used as the basic convolution unit; convolution with larger input matrix (larger than 4×4) can be constructed from $F(2 \times 2, 3 \times 3)$. Winograd algorithm can also be applied to calculate convolutions with larger filter matrix and results matrix, i.e., larger m, r for $F(m, r)$, however, calculating larger m and r also requires more additions and subtractions. These overhead will compromise the savings from multiplication, and eventually makes Winograd on larger filter matrices less efficient as compared to smaller filter matrices. Therefore, $F(2 \times 2, 3 \times 3)$ is the commonly used in many CNN calculations.

B. Acceleration (CONV23) module design

The architecture of the acceleration module (CONV23) is shown in Fig. 3. When the module is called to perform convolution $F(2 \times 2, 3 \times 3)$, RI5CY processor retrieves the data from the memory by calling the LSU module and transfers the data of input matrix to the CONV23 module. The data of input matrix is temporarily stored in 16 32-bit registers. And then the CONV23 module performs the calculation and outputs 2×2 output matrix (4 elements). The execution of the CONV23 instruction only needs 16 load-data operations and 1 Winograd algorithm operation.

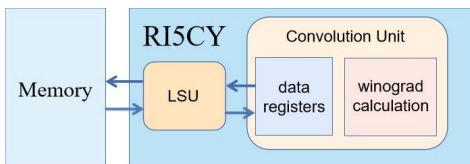


Fig. 3. architecture of acceleration system.

The operation of the CONV23 module is a 3-state state machine, namely, the *idle* state, the *get - data* state and the *calculation* state. In the *idle* state, if the RI5CY processor fetches the CONV23 instruction, the state machine changes from the *idle* state to the *get - data* state. In the *get - data* state, the CONV23 module reads one data from the memory each clock cycle according to the first address. After taking 16 input data, the state changes to the *calculation* state. In this state, the result is calculated by the combinational logics, the output can be generated after the logic delay performing the calculation.

In the CONV23 instruction, the clock cycles needed to perform the complete execution can be divided into four groups.

1) 1 cycle is needed to detect the enable signal of the CONV23.

2) 1 cycle is needed to assign the first address of input data in the memory to convolution acceleration module.

3) 16 cycles are needed to finish 16 load-data operations.

4) The last 1 cycle is needed to finish the calculation of Winograd.

The simulation waveform of the execution is shown in Fig. 4. The position of first line in Fig. 4 is the start time of the CONV23 instruction. After 1 clock cycle, the state machine changes from IDLE to GET_DATA. The position of the second line and third line in Fig. 4 is the start and end time of the 16 load-data operation. The output of 4 elements are calculated in the position of the last line.

IV. IMPLEMENTATION AND PERFORMANCE ANALYSIS

A. Timing and latency comparison

Using the standard ISA in a generic RI5CY processors, calculating the convolution between a 3×3 input and a 3×3 kernel needs 9 load-data operations of input data, 9 load-data operations of kernel, 9 multiplication operations and 8 addition operations. And the above process needs to repeat four times to calculate $F(2 \times 2, 3 \times 3)$, where the input matrix is 4×4 . Therefore, using standard RISC-V ISA, $F(2 \times 2, 3 \times 3)$ needs 72 load-data instructions, 36 multiplication instructions and 32 addition instructions. For the RI5CY processor, executing an instruction requires 1 clock cycle, a total of 140 clock cycles are required to perform $F(2 \times 2, 3 \times 3)$ at RI5CY.

In comparison, CONV23 instruction only takes 19 clock cycles to perform the $F(2 \times 2, 3 \times 3)$ operation, achieving a $7.37 \times$ of speedup.

B. Comparison of hardware resource overhead

The original generic RI5CY processor, as well as the extended CONV23 instruction, is synthesized by Vivado and implemented on the NEXYS A7 FPGA platform. In order to estimate the hardware resources needed by the CONV23 module, the number of resources such as LUT and FF, with and without CONV23 module, is shown in Fig. 5.

From the table, we can estimate that CONV23 uses 1253 LUTs, 54 FFs, and 10 BUFGs. Conversely, the overhead created by CONV23 is 10.9% of LUT and 0.8% of FF.

C. Power consumption comparison

The power consumption to perform the $F(2 \times 2, 3 \times 3)$ function, with and without the CONV23 instruction, is compared from the Vivado hardware resource report, as shown in Fig. 6.

For the generic RI5CY processor, before $F(2 \times 2, 3 \times 3)$ operation is performed, the power consumption (baseline power consumption) is 0.262W, during the calculation of $F(2 \times 2, 3 \times 3)$, the power consumption rises up to 0.310W, i.e., the operation costs a power overhead of 0.048W. In comparison, with the CONV23 instruction added to the core, the baseline power consumption increased to 0.283W, because extra hardware overheads (more LUTs and FFs) are added. However, when $F(2 \times 2, 3 \times 3)$ is executed with the CONV23 instruction, the power consumption only increases to 0.302W, i.e., CONV23 only costs 0.019W of extra power

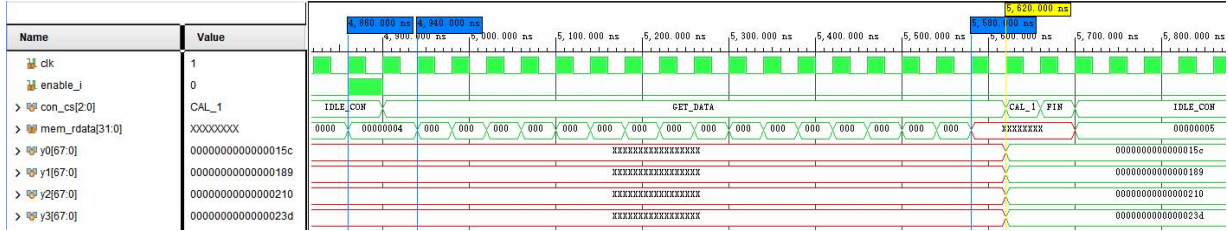


Fig. 4. Simulation waveform of CONV23 instruction.

Resource	Utilization	Available	Utilization %
LUT	11502	63400	18.14
FF	6378	126800	5.03
DSP	6	240	2.50
IO	32	210	15.24
BUFG	4	32	12.50
PLL	1	6	16.67

(a) Without CONV23 module

Resource	Utilization	Available	Utilization %
LUT	12755	63400	20.12
FF	6432	126800	5.07
DSP	6	240	2.50
IO	32	210	15.24
BUFG	14	32	43.75
PLL	1	6	16.67

(b) With CONV23 module

Fig. 5. Resource of CONV23.

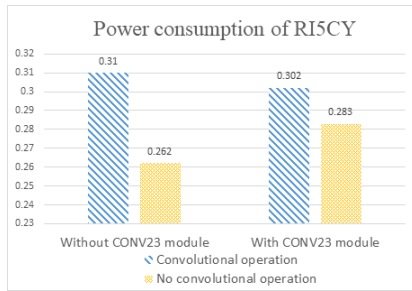


Fig. 6. Power consumption of RISCY.

consumption. The total power consumption by the RISCY core with CONV23 is around 2.6% less than that without CONV23. The power saving is coming from the reduction of multiplication operation counts. This analysis indicates that while the CONV23 instruction achieves the 7.37x speedup, it still maintains a similar, or even lower power consumption.

We further compare the energy consumed by the operation $F(2 \times 2, 3 \times 3)$ alone on the RISCY core, with or without the CONV23 instructions. RISCY core is running at 100MHz clock frequency on the FPGA platform. Without CONV23, $F(2 \times 2, 3 \times 3)$ needs 140 clock cycles, or 1400ns, the energy consumed is $0.048W \times 1400ns = 67.2nJ$. With CONV23, $F(2 \times 2, 3 \times 3)$ needs 19 clock cycles, the energy consumption for this instruction is $0.019W \times 190ns = 3.61nJ$.

V. CONCLUSION

In this paper, we constructed a customized instruction “CONV23” based on RISC-V ISA. The instruction can perform a basic convolution operation between a 4×4 input matrix and a 3×3 kernel matrix and generates a 2×2 output

matrix. We also exploit Winograd algorithm to optimize the convolution calculation, only 16 multiplications is needed to perform the convolution as compared to 36 multiplications needed from direct dot-product calculation. The execution of CONV23 only needs 19 clock cycles, as compared to 140 clock cycles if $F(2 \times 2, 3 \times 3)$ is performed by standard RISC-V ISA. The proposed CONV23 instruction is implemented on an open-source RISC-V CPU RI5CY core and synthesized into FPGA platform. The source code is placed on GitHub <https://github.com/QmppmQ/riscv>. Benefit from CONV23, convolution operation can speed up by a factor of 7.37x with 2.6% power consumption saving.

ACKNOWLEDGMENT

This work was supported by Key-Area Research and Development Program of Guangdong Province (Grant No. 2020B0101030002), High-level University Fund G02236002.

REFERENCES

- [1] V. Jain, A. Sharma, and E. A. Bezerra, “Implementation and extension of bit manipulation instruction on RISC-V architecture using FPGA,” in *2020 IEEE 9th International Conference on Communication Systems and Network Technologies (CSNT)*, 2020, pp. 167–172.
- [2] A. Garofalo, G. Tagliavini, F. Conti, D. Rossi, and L. Benini, “Xpulpnn: Accelerating quantized neural networks on RISC-V processors through ISA extensions,” in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 186–191.
- [3] C. A. R. A. Melo and E. Barros, “Oolong: A baseband processor extension to the RISC-V ISA,” in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2016, pp. 241–242.
- [4] G. Ottavi, A. Garofalo, G. Tagliavini, F. Conti, L. Benini, and D. Rossi, “A mixed-precision RISC-V processor for extreme-edge DNN inference,” in *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2020, pp. 512–517.
- [5] M. Alizadeh and M. Sharifkhani, “Extending RISC-V ISA for accelerating the h.265/HEVC deblocking filter,” in *2018 8th International Conference on Computer and Knowledge Engineering (ICCCKE)*, 2018, pp. 126–129.
- [6] Z. Li, W. Hu, and S. Chen, “Design and implementation of CNN custom processor based on RISC-V architecture,” in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2019, pp. 1945–1950.
- [7] N. Wu, T. Jiang, L. Zhang, F. Zhou, and F. Ge, “A reconfigurable convolutional neural network-accelerated coprocessor based on RISC-V instruction set,” *Electronics*, vol. 9, no. 6, 2020. [Online]. Available: <https://www.mdpi.com/2079-9292/9/6/1005>
- [8] D.-Z. Li, H.-R. Gong, and Y.-C. Chang, “Implementing RISC-V system-on-chip for acceleration of convolution operation and activation function based on FPGA,” in *2018 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*. IEEE, 2018, pp. 1–3.
- [9] A. Traber, M. Gautschi, and P. Schiavone, “RI5CY: User manual,” 2017.
- [10] A. Waterman and K. Asanovic, “The RISC-V instruction set manual, volume I: User-level ISA, document version 20190608-base-ratified,” *RISC-V Foundation*, 2019.
- [11] A. Lavin and S. Gray, “Fast algorithms for convolutional neural networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4013–4021.