Exact Mapping of Rewritten Linear Functions to Configurable Logic

Jonathan W. Greene, Microsemi, San Jose, CA, USA (jonathan.greene@microchip.com)

Abstract

We describe a method of mapping linear arithmetic functions to a targeted network of multipliers, adders, registers, muxes, and ROMs holding coefficients. The network may be fairly specific, or made more configurable by varying the primitives' control signals and adding routing muxes. Examples of possible functions include FIR and IIR filters, convolution, and FFTs. The method implicitly considers rewriting the function, such as by the associative and distributive properties of arithmetic, or methods such as Winograd filtering that benefit from combined computation of successive outputs. The method is intended to be used to implement IP for existing FPGA math blocks, evaluate new FPGA math block architectures, or potentially as a subroutine in high-level synthesis. Illustrative architectures using the PolarFire^{TM} FPGA math block are presented, including: a symmetric FIR in which the math block operates at twice the speed of the fabric; a general (asymmetric) FIR that produces one sample per clock cycle with fewer multipliers than taps; and a folded symmetric FIR using fewer memory blocks than multipliers. The method is related to modulo scheduling and relies on a SAT or ILP solver.

1 Introduction

The primary motivation for this work is the need to efficiently map linear functions to the math blocks in field-programmable gate arrays (FPGAs). Examples of linear functions are finite- and infinite-impulse-response (FIR and IIR) filters, 1D and 2D convolution, and fast Fourier transforms (FFTs). We focus on streaming computation, where the function is applied to an input stream of incoming data to produce an output stream of results.

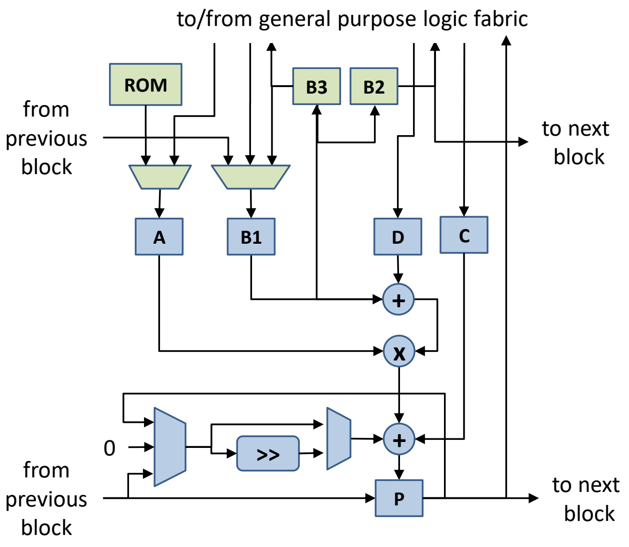

Figure 1: PolarFire^{TM} FPGA math block (simplified).

As an example, consider the simplified model of the PolarFire^{TM} FPGA math block [1] in **Figure 1**. The components include: a pre-adder; multiplier; adder; dynamically-controlled muxes; read-only memory for coefficient storage; shifter; and various registers. (For details, see Section 5 below.) Can we configure a cascade of N of these blocks, if necessary in conjunction with limited resources from the adjacent general-purpose FPGA fabric, to implement a $4N$-tap symmetric FIR filter with a sample interval of two clock cycles? How about a $(4N/3)$-tap general FIR with one cycle per sample? If yes, how should the various muxes, registers and coefficient ROMs be configured and controlled? If not, can we be sure we have not overlooked a solution? We present an algorithm to answer such questions. It considers equivalent ways of rewriting the desired function, beyond what typical commercial synthesis tools can do.

Numerous authors have studied related problems. The problem of scheduling a streaming data flow graph to accommodate the operator delays and resource limits is generally known as "modulo scheduling". Opperman *et al.* [2] use integer linear programming (ILP) to optimally solve the problem. Dai and Zhang [3] develop a more scalable but still exact algorithm. These approaches do not seek to rearrange the flow graph. Canis *et al.* [4] apply heuristics to transform the flow graph using the associative property of addition, but this is not an exact algorithm. The flow graph may also be "folded" to reduce resources at the expense of sample rate; see [5] for a recent example. Winograd [6] describes how multiplications can be saved at the expense of additions by joint computation of successive outputs. Gao *et al.* [7] give a method to rewrite arithmetic expressions using the associative and distributive properties to achieve pareto optimal resources and accuracy when using floating point. However this method does not (at least in its present form) consider optimizations across successive outputs.

The above mentioned work assumes unrestricted connectivity among the operators. In contrast, mapping programs, such as [8] and [9] and the prior work cited therein, assign specific routing resources needed to connect the operators in a targeted architecture. However these works do not support rewriting. In part this is due to the difficulty of precluding combinatorial loops in the routing.

The main contribution of this paper (Sections 3 and 4) is a conceptually straightforward algorithm to map a linear expression into targeted configurable logic. It implicitly considers folding, associative and distributive rewriting, as well as savings from joint computation of successive outputs in the manner of Winograd filtering.[†] If a solution exists, it will generate a complete description of the necessary routing configuration and control signal schedules. If a solution is not found, we can be confident none was overlooked (at least under the given constraints and for the targeted logic). The current implementation uses a SAT solver.

The algorithm can potentially be used by:

- FPGA manufacturers to propose and rapidly evaluate new math block hardware.
- IP developers to automatically generate DSP designs.
- End-users to implement specific desired functions in FPGAs.
- HLS algorithm developers as a subroutine.

Examples of novel mappings produced using the algorithm are given in Section 6. (These may be of interest in their own right.)

Other contributions include:

- A way to extend data flow graph mappers to support the limited rewriting possible using the associative property (Section 2).
- A more flexible way to prohibit combinatorial loops during mapping (Section 4).

2 A Way to Support Associative Rewriting

Before moving to our main results, we describe a possible trick we considered to add support for associative rewriting to conventional data flow graph (DFG) mappers such as [9].

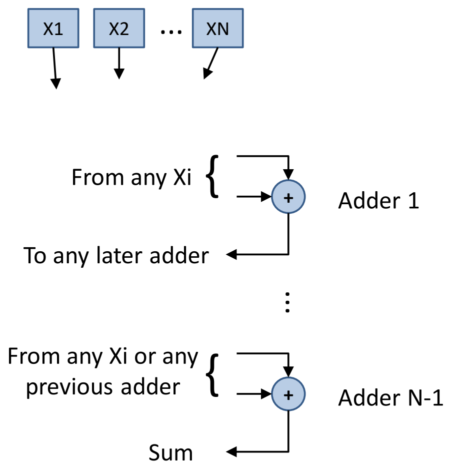

Figure 2: Intermediate operator order graph.

[†] We note that these kinds of rewriting can affect results when computing with finite precision, but they are nevertheless useful in many practical contexts.

Suppose the DFG calls for a sum of N values. Rather than specify an arbitrary order of summation in the DFG and map it directly to the "modulo routing resource graph" (MRRG) [9], we introduce an intermediate operator order graph (IOOG) as shown in **Figure 2**. Boolean variables are added to specify which data value or previous adder output drives each adder input in the IOOG. The usual constraints on fanout and fanin are added to ensure the result is a binary tree of N-1 adders. The IOOG can then be mapped to the MRRG in the usual way.

Unfortunately this approach cannot be extended to support more general types of rewriting.

3 Problem Statement

The setting of the problem is depicted in **Figure 3**. We have an input stream of data values, an output stream of results, and a targeted set of configurable logic, described by a network including primitives of these types:

- Input and Output
- Multiplier: One input is a sum of constants, the other a sum of data values.
- Adder, Subtractor
- Coefficient ROM: Outputs one or a specified sum of coefficients in response to an address.
- Register, with enable and clear.
- Mux, with dynamic select.
- Routing Mux, with static select.
- Delay line, with a static delay value.
- Addressable shift register (ASR), with a dynamic read address.

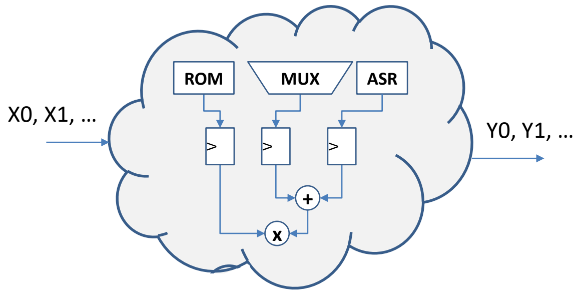

Figure 3: Problem setting.

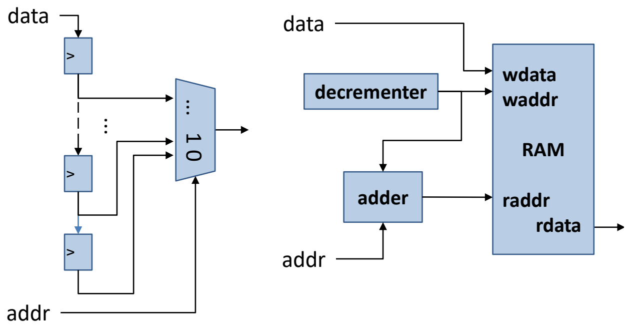

Figure 4: Two possible ASR implementations.

Figure 4 shows how an ASR can be implemented in a simple way using a shift register and a mux, or in a more complicated but power-efficient way using a decrementer, adder, and two-port RAM. In some cases a single-port RAM may be used if it supports simultaneous write and read (in either order) using the same address.

All control signals (enables, clears, selects, addresses) are periodic functions with a period P, and are independent of the data values.

We are given:

- The targeted configurable logic network.
- A linear function of the form
$$F(X_0,\ X_1,\ \dots\ X_{N-1}) = \sum_{i=0}^{N-1} C_i X_i$$
where X_i are a portion of the incoming data stream and C_i are symbolic (arbitrary) constants.
- The period P (analogous to the initiation interval in modulo scheduling).
- The phases (in clock cycles or time steps) during the period when output samples must be produced.
- A range of allowable latencies.
- Any additional optional constraints, e.g. regarding symmetries among the control signals.

We wish to find:

- A specific latency.
- A schedule of all control signals.
- A symbolic simulation trace confirming proper computation of the function.
- … or a demonstration that no solution exists.

4 Algorithm Sketch

Conceptually, we formulate the problem as a mixed integer linear program. At a time step t, the output of a primitive p takes the value

$$\sum_i K_{itp} C_i + \sum_j K_{jtp} X_j + \sum_{i,j} K_{ijtp} C_i X_j$$

where the various K are real-valued variables. Alternatively, if a Boolean variable I_{tp} is true, the output is considered invalid. We also define a Boolean variable to indicate the state of each control signal at each time step.

For each primitive in the targeted architecture, we add constraints on the variables for its inputs, outputs and control signals that ensure the proper relationship among them. For example, consider a 2-input mux and the following variables for a particular time step:

- S: indicating the state of the select input
- K_{in0}: one of the real-valued K variables for input 0
- K_{out}: the corresponding K variable for the mux output
- I_{in0}: the Boolean variable indicating invalid data at input 0
- I_{out}: the Boolean variable indicating invalid data at the output

We impose the constraint that if S is false (selecting input 0) then $K_{\text{out}} = K_{\text{in0}}$ and $I_{\text{out}} = I_{\text{in0}}$. As another example, for a two-input adder we have $K_{\text{out}} = K_{\text{in0}}+K_{\text{in1}}$ and $I_{\text{out}} = \text{OR}(I_{\text{in0}}, I_{\text{in1}})$. With some care, it is straightforward to define appropriate constraints in a similar way for all the primitives.

We also add constraints imposing that the desired terms (and only the desired terms) are present at the desired time(s) on the output. But what range of times must we consider?

Claim: If we can show that the output assumes the proper values during one period, then this will also be true of all prior and subsequent periods.

Proof: By induction. We rely on the fact that all control signals and data arrival/departure times are periodic and independent of the data values.

The claim allows us to limit consideration to one period. It also allows us to restrict consideration to only the X_i values that must appear at the output during that period. Any prior or subsequent X_i values can just be treated as invalid, without further distinction.

The variables K are created selectively by propagating from the appearance of the required output values backward in space and time. Care must be taken to provide variables for all terms that might be required to support rewriting involving multiple outputs.

One additional complication must be handled: the need to avoid combinatorial loops (which if present might allow a value to appear out of nowhere). These can be avoided by first decomposing the architecture into strongly-connected components of combinational primitives, and then adding constraints among the variables specifying the state of the select inputs of the muxes in each such component, for instance using the techniques described in [10].

Our initial implementation uses an ordinary SAT solver (specifically MiniSAT) rather than ILP. This makes it inconvenient to support many possible choices for each K. However, SAT can easily support three choices of K (specifically -1, 0 and 1), using two Boolean variables. This suffices for the results we demonstrate below. (We discuss in Section 7 how this can be generalized.)

5 PolarFire™ FPGA Math Block

To exemplify the use of the algorithm, we will target the normal mode of the PolarFire™ FPGA math block, shown in **Figure 1**. It includes an 18-bit pre-adder (with 19-bit output) and an 18x19 bit multiplier. The final adder is 48 bits, and includes a carry input and overflow output (not shown). The adder and pre-adder also support subtraction. An 18-bit, 16-word read-only memory is provided for coefficient storage. A shifter allows multiple blocks to be combined into wider multipliers. The registers support independent enable and (synchronous and asynchronous) clear inputs. Each register may also be bypassed (made transparent). For further details see [1].

The math block also supports two reduced precision modes (not shown):

- A two-element, 9-bit dot-product, including use of the pre-adder and final adder.
- Dual independent 9x9 bit multipliers.

The reduced precision modes are useful for video and especially neural network applications [11].

The primitives shown in blue are implemented by hardened circuitry for optimum speed, power and area. The green primitives provide additional flexibility. They are implemented by LUTs and registers typical of those in the fabric except that they are located adjacent to the math block and can be connected to it and each other via direct routes. For this reason routing delays among the blue and green primitives are still fast and predictable.

6 Examples

In this section we give a few example applications that were developed with the help of the algorithm. We don't claim that these were discovered purely by applying the algorithm in one shot. Some human insight and trial and error was required. But the algorithm was indispensable to prove that the applications can be mapped as shown.

6.1 Double Data Rate Symmetric FIR

Figure 5 shows how a symmetric 4*N*-tap FIR filter can be implemented in *N* PolarFire^{TM} math blocks. The period is two clock cycles, with one sample per period at both the input and output.

The logic includes an input, output, adders, multipliers, muxes (with dynamic selects), ROMs for the coefficients, and registers (each with an enable signal). There are also a routing mux and optional registers, shown with dotted borders. Recall that we have fast and predictable routing delays among the blue primitives (implemented in hard circuitry) and green primitives (implemented using directly-connected adjacent LUTs and flip-flops in the fabric). In contrast, the primitives shown here in red are implemented in the remainder of the fabric, and routing delays among them are less predictable. So we add a constraint that the red sequential elements can be enabled (simultaneously) only on alternate clock cycles. This allows the math block to be clocked at twice the rate that can be supported by general fabric routing.

Determining manually whether and how all the control signals and coefficient addresses can be set properly would be difficult, especially considering the various possible remainders when the number of taps is divided by four, and cases of odd and even symmetry. No user, or even IP developer, would relish working through this.

Figure 6 shows the schedule of control signals and execution trace reported by the algorithm for the case of 11 taps. Observe that the B2 and R registers are enabled in synchrony and only on alternate clock cycles, as required for double data rate operation.

6.2 Winograd FIR Filter

Consider two successive outputs from a 2-tap FIR filter, normally computed using four multiplication operations:

$$Y_0 = C_0X_0 + C_1X_1$$
$$Y_1 = C_0X_1 + C_1X_2$$

As Winograd observed in [6], these can be rewritten as follows using only three multiplications:

$$D = C_0(X_0 - X_1)$$
$$E = X_1(C_0 + C_1)$$
$$F = -C_1(X_1 - X_2)$$
$$Y_0 = D + E$$
$$Y_1 = E + F$$

Using our algorithm, we have found an efficient way to leverage this trick to implement a general *T*-tap FIR filter in $N = (3/4)T$ math blocks, shown in **Figure 7**. The period is 2 clock cycles, but a sample is produced every clock cycle. The implementation requires a fixed delay of 1 or 4 clock cyles depending on the remainder *T* mod 4. This is represented by the indicated delay line. The solution uses the 3rd addend input C provided by the PolarFire^{TM} math block to reduce the total latency. The schedule of control signals and trace is shown in **Figure 8**.

6.3 Folded Symmetric FIR with Few RAMs

Many forms of folded FIRs are known in the literature. Here we show that the algorithm can generate a particularly intricate one that can support folding while still benefiting from symmetrical coefficients, and without requiring a separate RAM block for each multiplier. It relies for its proper operation on the folding factor *F* and number of multipliers *N* being relatively prime (i.e., having no common factors other than one). The structure requires one RAM for the forward ASR and $\min\{F,N\}$ RAMs for the reverse ASRs. The filter is depicted in **Figure 9** and the schedule and trace in **Figure 10**.

7 Limitations and Future Work

As mentioned above, our initial implementation used a SAT solver, and for convenience we limited the values of the variables *K* to -1, 0 or 1. To handle more complex Winograd filter schemes, we would need to support other values of *K*. A larger but still finite selection of values could be supported with SAT by using more Boolean variables to encode the value. Using ILP or SAT modulo theory (SMT) solvers would support all real values.

This work assumed that any combinational path through the target architecture would have a sufficiently small delay so as not to exceed the target clock frequency (which is reasonable for FPGA math blocks). Additional constraints on the mux select variables could be added to ensure this, as is done in modulo scheduling.

Any time SAT or ILP solvers are used, run time may be an issue. Typical FPGA DSP structures are organized as a chain of identical units. We have found that the addition of a few corresponding symmetry constraints on control variables can significantly reduce run time. It may also be possible to further leverage the symmetry by considering only the first unit, last unit, and a representative unit in the middle of the chain to reduce the problem size. The approach of [3], combining SAT with a linear constraint solver to achieve scalability, might also be investigated.

Despite these limitations, we have already found the current implementation to be very helpful in our own work generating IP for existing math blocks and evaluating architectures for new ones.

8 References

[1] Microsemi: PolarFire FPGA Product Overview (PO0137), www.microsemi.com, 2018.

[2] Oppermann, J., Koch, A., Reuter-Oppermann, M., Sinnen, O.: ILP-based Modulo Scheduling for High-level Synthesis. CASES '16, 2016.

[3] Dai, S. Liu, G., Zhang, Z.: A Scalable Approach to exact Resource-Constrained Scheduling based on a Joint SDC and SAT Formulation. Int'l Symp. On Field-Programmable Gate Arrays, 2018.

[4] Canis, A., Brown, S., Anderson, J.: Modulo SDC Scheduling with Recurrence Minimization in High-Level Synthesis. Int'l Conf. On Field Programmable Logic, 2014.

[5] Denk, T., Parhi, K.: Synthesis of Folded Pipelined Architectures for Multirate DSP Algorithms. IEEE Trans. VLSI Systems, Dec. 1998.

[6] Winograd, S.: Arithmetic Complexity of Computations. Soc. Indus. & Applied Math., 1980.

[7] Gao, X., Baylis, S., Constantinides, G.: SOAP: Structural Optimization of Arithmetic Expressions for High-Level Synthesis. Int'l Conf. Field-Programmable Tech., 2013.

[8] Fan, K., Park, H., Kudlur, M., Mahlke, S.: Modulo scheduling for highly customized datapaths to increase hardware reusability. Proc. IEEE/ACM Intl'l Symp. On Code Generation and Optimization., 2008.

[9] Chin, S., Anderson, J.: An Architecture-Agnostic Integer Linear Programming Approach to CGRA Mapping. Design Automation Conf., 2018.

[10] Gebser, M., Janhunen, T., Rintanen, J.: Declarative encodings of acyclicity properties. Journal of Logic and Computation, 2015.

[11] Microsemi: Efficient INT8 Dot Product using Microsemi Math Block (WP0216), www.microsemi.com, 2018

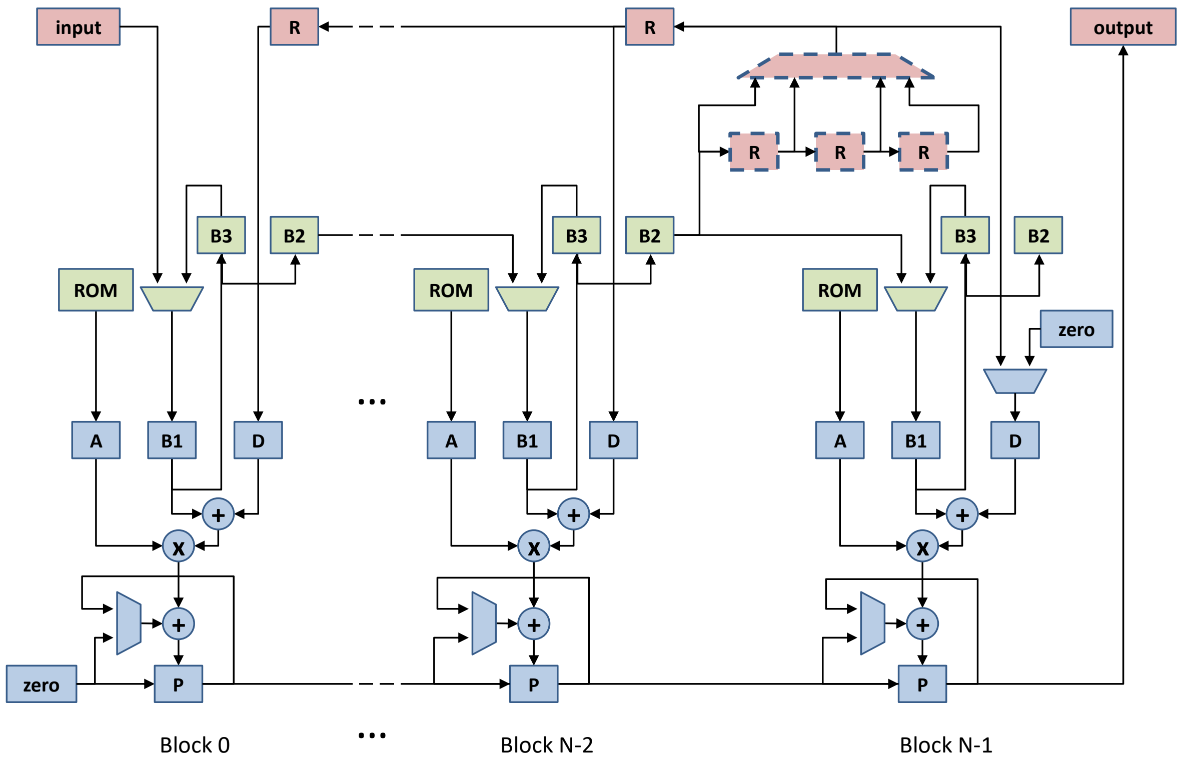

Figure 5: Structure of double data rate FIR.

Node	Variable	phase=0	phase=1
A[0-2]	enable	1	1
B1[0-2]	enable	1	1
B2[0-1]	enable	0	1
B2[2]	enable	0	0
B3[0-2]	enable	1	0
Bmux[0]	select	B3[0]	Input
Bmux[1]	select	B3[1]	B2[0]
Bmux[2]	select	B3[2]	B2[1]
D[0-2]	enable	1	1
Dmux	select	Zero	Rmux
Input	valid	0	1
P[0-2]	enable	1	1
Pmux[0]	select	Zero	P[0]
Pmux[1]	select	P[0]	P[1]
Pmux[2]	select	P[1]	P[2]
R[0-4]	enable	0	1
Rmux	route	R[3]	R[3]
Rom[0]	coeff	C1	C0
Rom[1]	coeff	C3	C2
Rom[2]	coeff	C5	C4

Time	Phase	Input	B1[0]	D[0]	P[0]	B1[1]	D[1]	P[1]	B1[2]	D[2]	P[2]
1	1	X0								0	
2	0		X0								
3	1	X1								0	
4	0		X1								
5	1	X2	X0							0	
6	0		X2								
7	1	X3	X1							0	
8	0		X3			X0					
9	1	X4	X2							0	
10	0		X4			X1					
11	1	X5	X3			X0				0	
12	0		X5			X2					
13	1	X6	X4			X1				0	
14	0		X6			X3			X0		
15	1	X7	X5			X2				0	
16	0		X7			X4			X1		
17	1	X8	X6			X3			X0	0	
18	0		X8			X5			X2	X0	
19	1	X9	X7			X4	X0		X1	0	
20	0		X9			X6	X0		X3		
21	1	X10	X8	X0		X5	X1		X2	0	
22	0		X10	X0		X7			X4		
23	1		X9	X1	C0X0+C0X10	X6			X3	0	
24	0			X1	C0X0+C1X1 +C1X9+C0X10	X8	X2		X5		
25	1					X7	X3	C0X0+C1X1+C2X2 +C2X8+C1X9+C0X10	X4	0	
26	0					X9	X3	C0X0+C1X1+C2X2 +C3X3+C3X7+C2X8 +C1X9+C0X10	X6	X4	
27	1								X5	0	C0X0+C1X1+C2X2+C3X3 +C4X4+C4X6+C3X7+C2X8 +C1X9+C0X10
28	0										C0X0+C1X1+C2X2+C3X3 +C4X4+C5X5+C4X6+C3X7 +C2X8+C1X9+C0X10

Figure 6: Schedule and trace for double data rate 11-tap symmetric FIR in 3 math blocks.

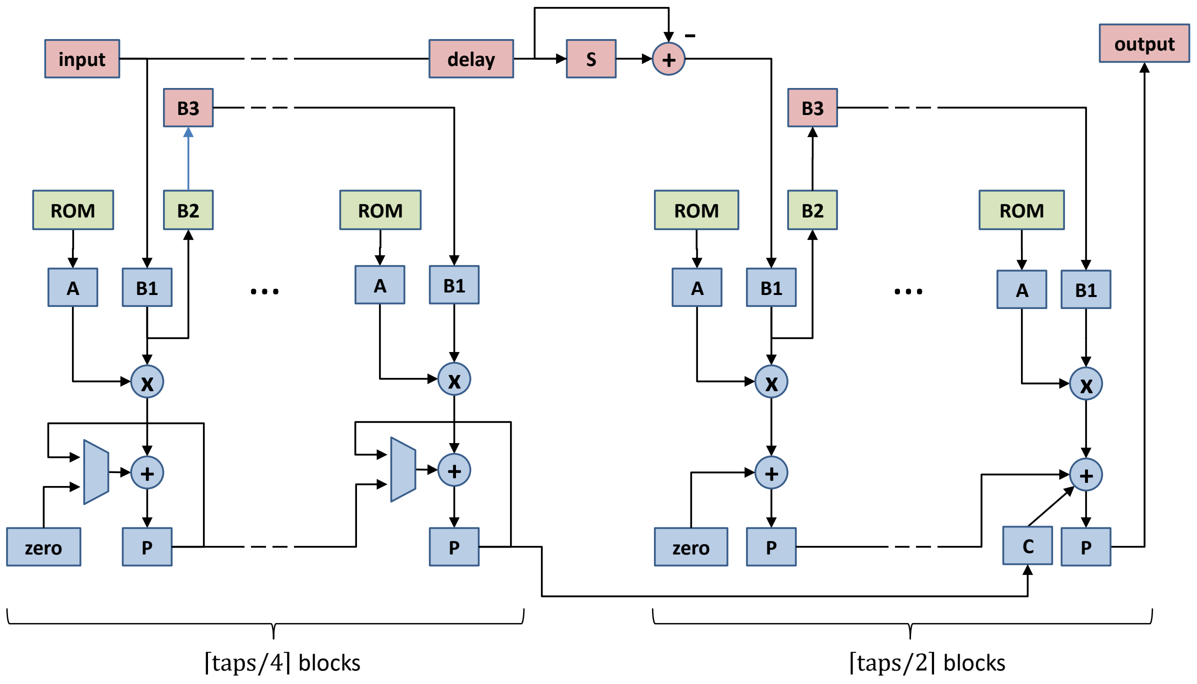

Figure 7: Structure of Winograd FIR filter.

Node	Variable	phase=0	phase=1
A[0-5]	enable	1	1
B1[0]	enable	0	1
B1[1]	enable	0	1
B1[2]	enable	1	1
B1[3]	enable	1	1
B1[4]	enable	1	1
B1[5]	enable	1	1
B2[0]	enable	0	1
B2[2]	enable	1	1
B2[3]	enable	1	1
B2[4]	enable	1	1
B3[0]	enable	0	1
B3[2]	enable	1	1
B3[3]	enable	1	1
B3[4]	enable	1	1
C	enable	0	1
Delay	enable	1	1
Delay	addr	0	0
Input	valid	1	0
P[0-5]	enable	1	1
Pmux[0]	select	P[0]	Zero
Pmux[1]	select	P[1]	P[0]
Rom[0]	coeff	C4+C5	C6+C7
Rom[1]	coeff	C0+C1	C2+C3
Rom[2]	coeff	C6	-C7
Rom[3]	coeff	-C5	C4
Rom[4]	coeff	C2	-C3
Rom[5]	coeff	-C1	C0
S	enable	1	1

Time	Phase	Input	B1[0]	P[0]	B1[1]	P[1]	C	B1[2]	P[2]	B1[3]	P[3]	B1[4]	P[4]	B1[5]	P[5]
0	0	X0													
1	1	X1													
2	0	X2	X1												
3	1	X3	X1					X0-X1							
4	0	X4	X3					X1-X2							
5	1	X5	X3					X2-X3							
6	0	X6	X5					X3-X4		X0-X1					
7	1	X7	X5					X4-X5		X1-X2					
8	0	X8	X7	C4X5+C5X5	X1			X5-X6		X2-X3					
9	1			C4X5+C5X5 +C6X7+C7X7	X1			X6-X7		X3-X4		X0-X1			
10	0				X3	C0X1+C1X1 +C4X5+C5X5 +C6X7+C7X7		X7-X8	C6X6 - C6X7	X4-X5		X1-X2			
11	1				X3	C0X1+C1X1 +C2X3+C3X3 +C4X5+C5X5 +C6X7+C7X7			-C7X7 +C7X8	X5-X6	C4X4-C4X5 +C6X6-C6X7	X2-X3			
12	0						C0X1+C1X1 +C2X3+C3X3 +C4X5+C5X5 +C6X7+C7X7				-C5X5+C5X6 -C7X7+C7X8	X3-X4	C2X2-C2X3 +C4X4-C4X5 +C6X6-C6X7	X0-X1	
13	1						C0X1+C1X1 +C2X3+C3X3 +C4X5+C5X5 +C6X7+C7X7						-C3X3+C3X4 -C5X5+C5X6 -C7X7+C7X8	X1-X2	C0X0+C1X1 +C2X2+C3X3 +C4X4+C5X5 +C6X6+C7X7
14	0														C0X1+C1X2 +C2X3+C3X4 +C4X5+C5X6 +C6X7+C7X8

Figure 8: Schedule and trace for 8-tap general (asymmetric) Winograd FIR filter in 6 math blocks.

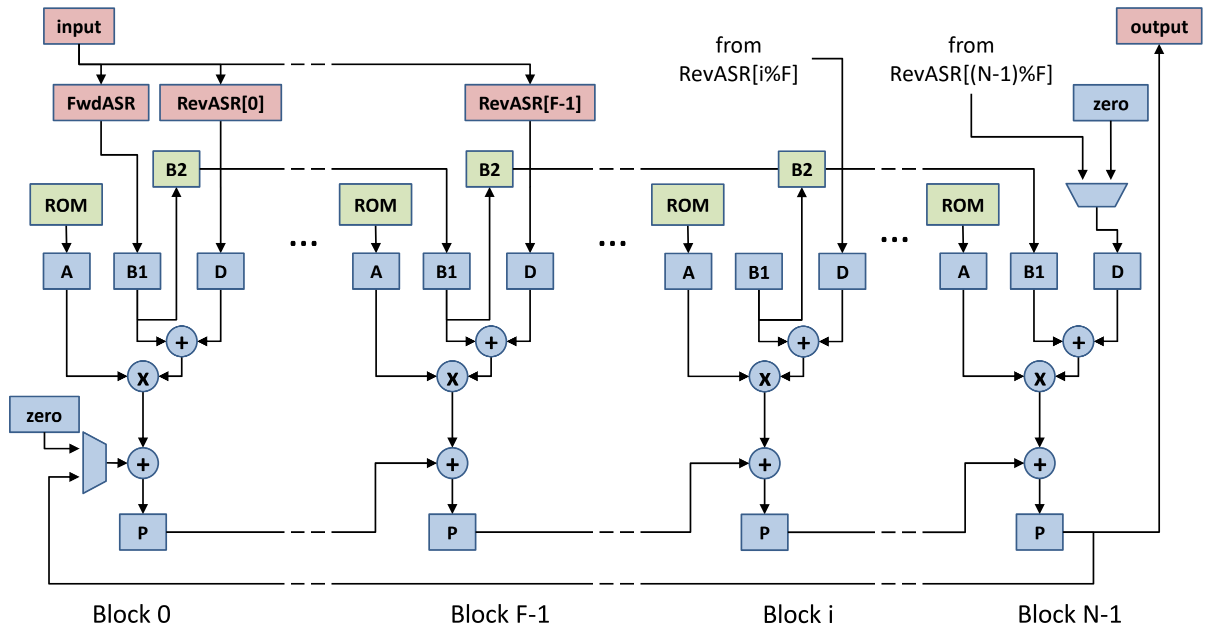

Figure 9: Structure of symmetric folded FIR with fewer RAMs than math blocks.

Node	Variable	phase=0	phase=1
A[0-2]	enable	1	1
B1[0-2]	enable	1	1
B2[0-1]	enable	1	1
D[0-2]	enable	1	1
Dmux	select	RevASR[0]	RevASR[0]
FwdASR	enable	1	0
FwdASR	addr	3	0
Input	valid	1	0
P[0-2]	enable	1	1
Pmux	select	Zero	P[2]
RevASR[0]	enable	1	0
RevASR[0]	addr	8	9
RevASR[1]	enable	1	0
RevASR[1]	addr	5	12
Rom[0]	coeff	C3	C1
Rom[1]	coeff	C5	C0
Rom[2]	coeff	C4	C2

Time	Phase	Input	B1[0]	D[0]	P[0]	B1[1]	D[1]	P[1]	B1[2]	D[2]	P[2]
6	0	X0									
7	1										
8	0	X1									
9	1										
10	0	X2	X1								
11	1										
12	0	X3	X2			X1					
13	1										
14	0	X4									
15	1		X0								
16	0	X5									
17	1		X1			X0					
18	0	X6	X5								
19	1		X2			X1	X0				
20	0	X7	X6			X5					
21	1		X3			X2	X1				
22	0	X8	X7			X6					
23	1		X4			X3	X2				
24	0	X9				X7					
25	1		X5	X0		X4	X3		X3	X0	
26	0	X10	X9						X7	X0	
27	1		X6			X5			X4		
28	0	X11	X10	X1		X9					
29	1		X7	X2	C1X1+C1X10	X6	X5		X5	X2	
30	0		X11	X2		X10		C1X1+C5X5 +C5X6+C1X10	X9	X2	
31	1		X8	X3		X7	X6				C1X1+C2X2+C5X5 +C5X6+C2X9+C1X10
32	0				C1X1+C2X2+C3X3 +C5X5+C5X6+C3X8 +C2X9+C1X10	X11	X0		X10	X3	
33	1							C0X0+C1X1 +C2X2+C3X3 +C5X5+C5X6	X7	X4	
34	0										C0X0+C1X1+C2X2 +C3X3+C4X4+C5X5 +C5X6+C4X7+C3X8 +C2X9+C1X10+C0X11

Figure 10: Schedule and trace for 12-tap symmetric FIR folded by 2, in 3 math blocks and 3 RAMs.