Creating Avalon ST computational pipelines in SOPC Builder.

Last Modified: December 2009

Altera Tools Release: Quartus II 9.1

Introduction

In this paper we will discuss the concept of using the Avalon ST standard interfaces within SOPC Builder to construct a self paced computational data path which can be applied to various algorithms. Repetitive and iterative signal processing type algorithms can be adapted to this methodology quite easily.

We assume that the reader is already familiar with the characteristics and operation of the standard interfaces used within an SOPC Builder system. For more information on SOPC Builder and it's standard interfaces please refer to the "Quartus II Handbook Version 9.1 Volume 4: SOPC Builder" and the "Avalon Interface Specifications" literature from the Altera website. The examples in this paper are primarily driven by the Avalon ST streaming point to point protocol and they utilize the back pressure, forward enablement and channelization capabilities within the Avalon ST protocol, but they do not leverage any of the packet signaling capabilities of the protocol.

Overview

The example that follows is contrived by a simple signal processing filtering type of algorithm which consists of computing the double precision floating point dot product of an input vector multiplied across an input matrix. On the following pages you will find the MatLab equations that are used to define this matrix manipulation along with some example C program implementations to provide a context of the required operations within this algorithm. From this programmatic representation, we then extract a data flow diagram that illustrates the step by step data movement of the input operands flowing through each computational operation to produce the intermediate operands which are further processed to produce the final result. This whole data flow represents one iteration of the dot product operation required by the algorithm, so consecutive iterations which pass each vector of the matrix through this data flow will produce the final scalar result that is desired by the algorithm.

Once the data flow diagram has been established it should be fairly easy to see how this structure can be mapped into a simple multiplexed data path, and how the repeatedly used operations for multiplication and addition can be condensed into a single resource for each operation which is then reused by the data path in a time domain multiplexed fashion to compute all of the required operations in the algorithm. At this point the diagram gets a bit more complicated as we attempt to illustrate how this data flow requirement is mapped into the point to point multiplexed structure of the Avalon ST architecture, but it should become very apparent that this construct is nothing more than a data path which multiplexes operands into the shared multiplication and addition resources and then demultiplexes the results of these operations back into the data path for further processing. The multiplexing and demultiplexing of this data path is all based on the standard SOPC Builder components provided for this activity and the channelization which accumulates through the natural Avalon ST multiplexing activity is what we use to then demultiplex the data path into each consecutive operation.

A detailed explanation is provided for each stage of the Avalon ST data path diagram to bring insight into the more trivial details that are required by the implementation to deal with the real world requirements of deploying the data path. Things like data formatting translations and other aspects of the architecture will be explained so that it's clear how the double precision floating point operands are passed from the external system into the data path pipeline and then back out into the external system.

Points of Interest

As you read through this paper, it may seem like a very complex example to digest and you may not be intimately familiar with the Avalon ST concepts and constructs that it is built upon, but in the end this example is actually trying to illustrate some very simple concepts. So keep these thoughts in mind as you read through the details of the rest of this paper:

1 – The data path which is defined in this example is nothing more than a simple multiplexed and demultiplexed data stream.

2 – The double precision floating point cores that are deployed in this example are rather large IP blocks which can consume a lot of valuable FPGA real estate, however they also provide an enormous amount of computational bandwidth. So in order to make the most of this available computational bandwidth, this example is simply time sharing the multiplication and addition requirements of the algorithm through one instance of each of these resources.

3 – Since this architecture is put together using standard Avalon ST components within the SOPC Builder environment, the architecture can be easily reconfigured to modify the algorithm itself, or even add new algorithms to the existing data path to consume the available bandwidth of the double precision floating point elements to produce additional computational work output.

```
// MATLAB Equations
// y = [x(1:4);cos(x(5))*x(6);sin(x(5))*x(6)];
// a = data(:,1:6)*y;
// e = data(:,7)-a;
// F = e'*e;

double calculate_F_sw ( double x[6], double data[461][7] ) {

	int i;
	double y[6];
	double a[461];
	double e[461];
	double F;

	y[0] = x[0];
	y[1] = x[1];
	y[2] = x[2];
	y[3] = x[3];
	y[4] = cos(x[4]) * x[5];
	y[5] = sin(x[4]) * x[5];

	for( i = 0 ; i < 461 ; i++ ) {
		a[i] = data[i][0] * y[0];
		a[i] += data[i][1] * y[1];
		a[i] += data[i][2] * y[2];
		a[i] += data[i][3] * y[3];
		a[i] += data[i][4] * y[4];
		a[i] += data[i][5] * y[5];
	}

	for( i = 0 ; i < 461 ; i++ ) {
		e[i] = data[i][6] - a[i];
	}

	F = 0.0;
	for( i = 0 ; i < 461 ; i++ ) {
		F += e[i] * e[i];
	}

	return(F);
}
```

This is a simplistic and straight forward translation of the MatLab equations into a C subroutine.

Every stage of the MatLab equations is sequentially computed by this implementation. Intermediate results are stored temporarily to be processed by the next sequential stage.

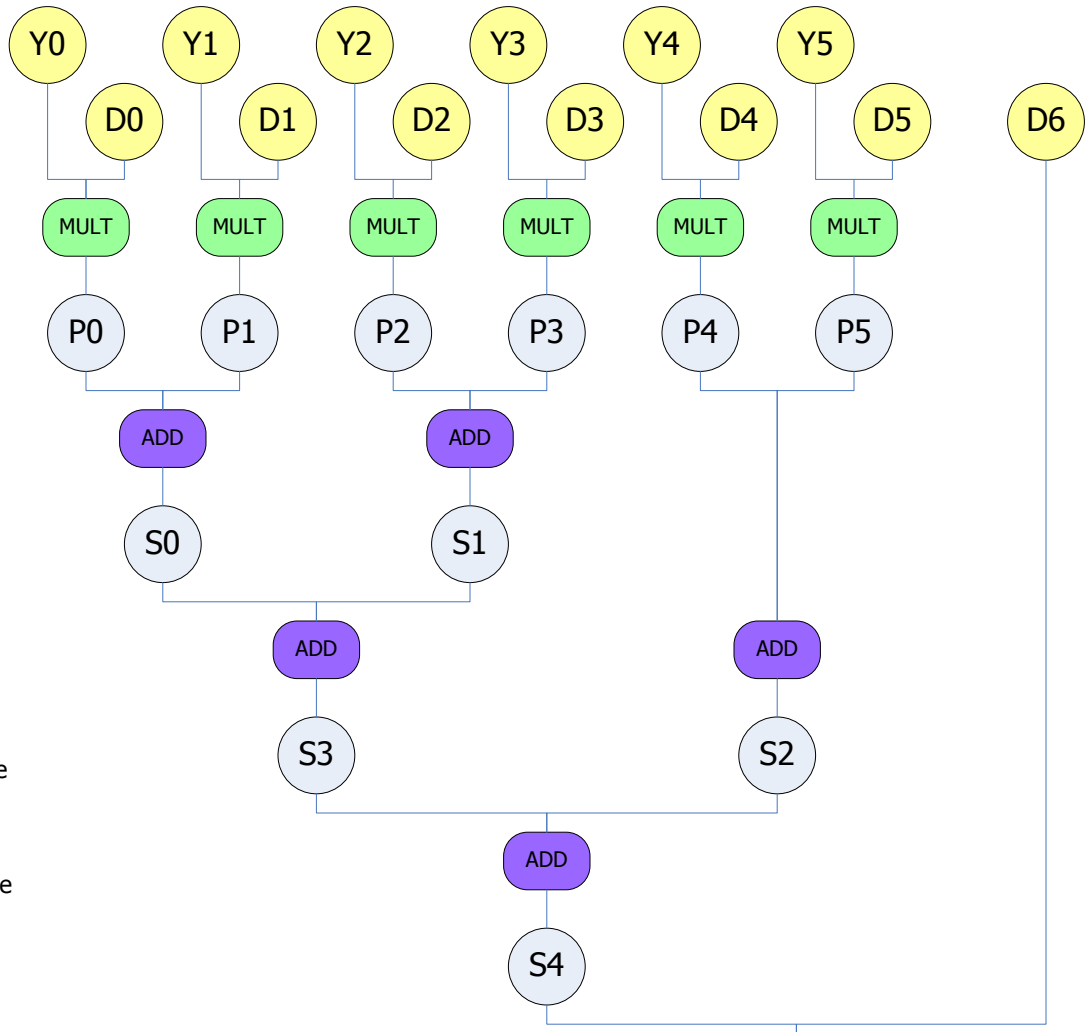
This is a "rolled up" implementation of the routine presented above. Here we eliminate the intermediate storage requirements and just process the intermediate results immediately.

```
double calculate_F_sw2 ( double x[6], double data[461][7] ) {

	int i;
	double y[6];
	double F = 0.0;
	double temp;

	y[0] = x[0];
	y[1] = x[1];
	y[2] = x[2];
	y[3] = x[3];
	y[4] = cos(x[4]) * x[5];
	y[5] = sin(x[4]) * x[5];

	for( i = 0 ; i < 461 ; i++ ) {
		temp  = data[i][0] * y[0];
		temp += data[i][1] * y[1];
		temp += data[i][2] * y[2];
		temp += data[i][3] * y[3];
		temp += data[i][4] * y[4];
		temp += data[i][5] * y[5];

		temp = data[i][6] - temp;

		F += temp * temp;
	}

	return(F);
}
```

The diagram below represents the fundamental data flow required to implement the matrix manipulations represented by the code on the left.

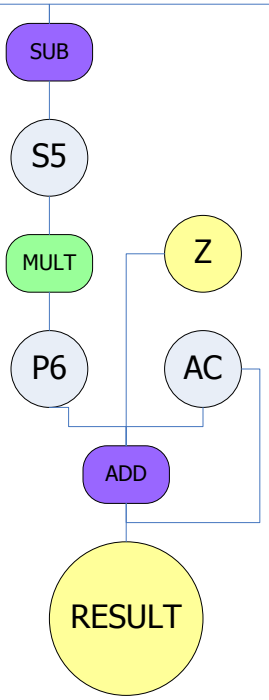
The processing pipeline is initialized by pushing a ZERO into the Z input operand. The pipeline is prioritized to take the Z input if it is present as one of the accumulator operands.

The processing pipeline is terminated by pushing an alternate D6 value into the pipeline, call this D6, but the picture to the right does not illustrate this data path. The diagram on the next page does illustrate the D6 input, and again the pipeline will prioritize the D6 input over the D6 input to ensure proper ordering of the operands.



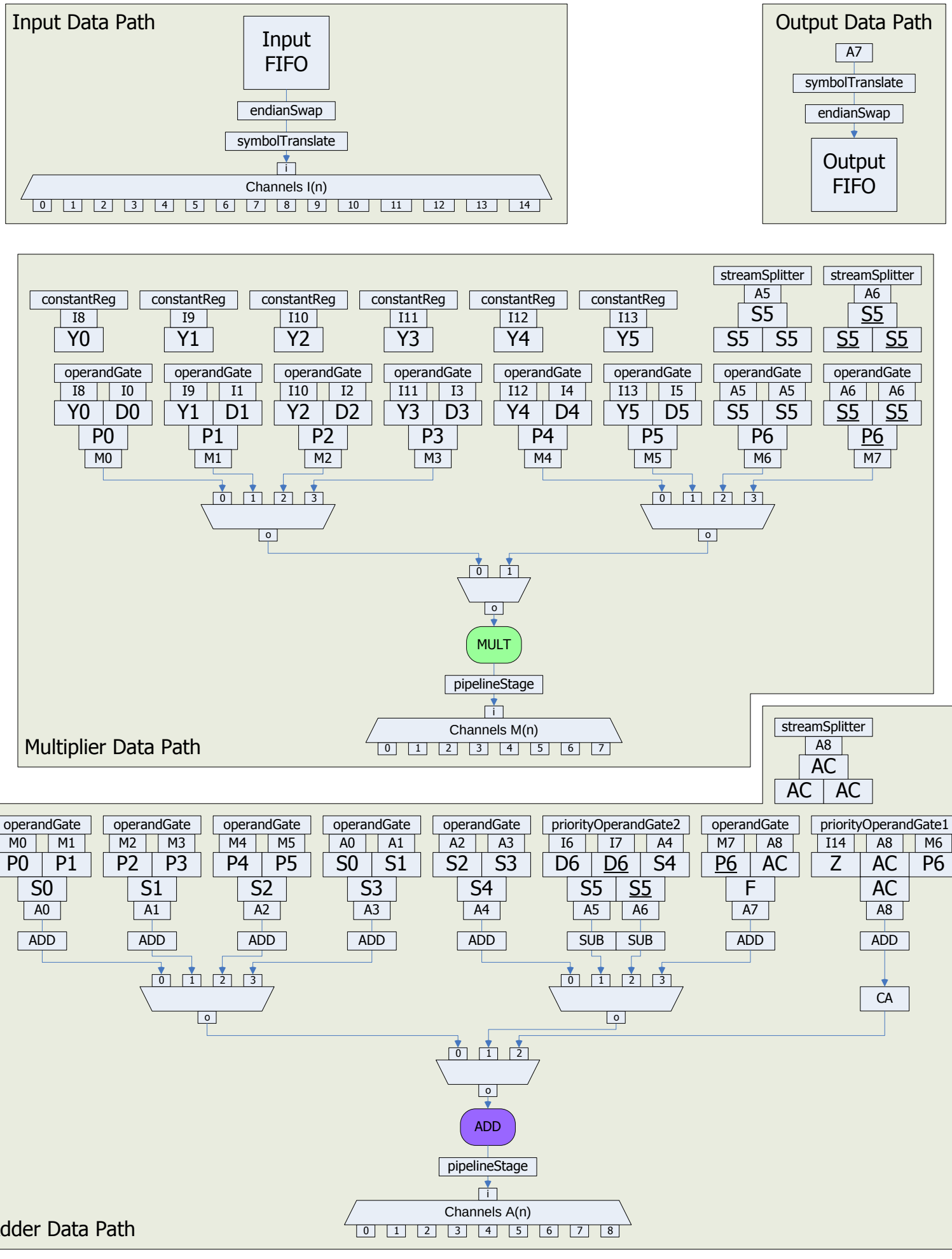
Inputs / Outputs to and from the processing pipeline.

Intermediate operands within the processing pipeline.

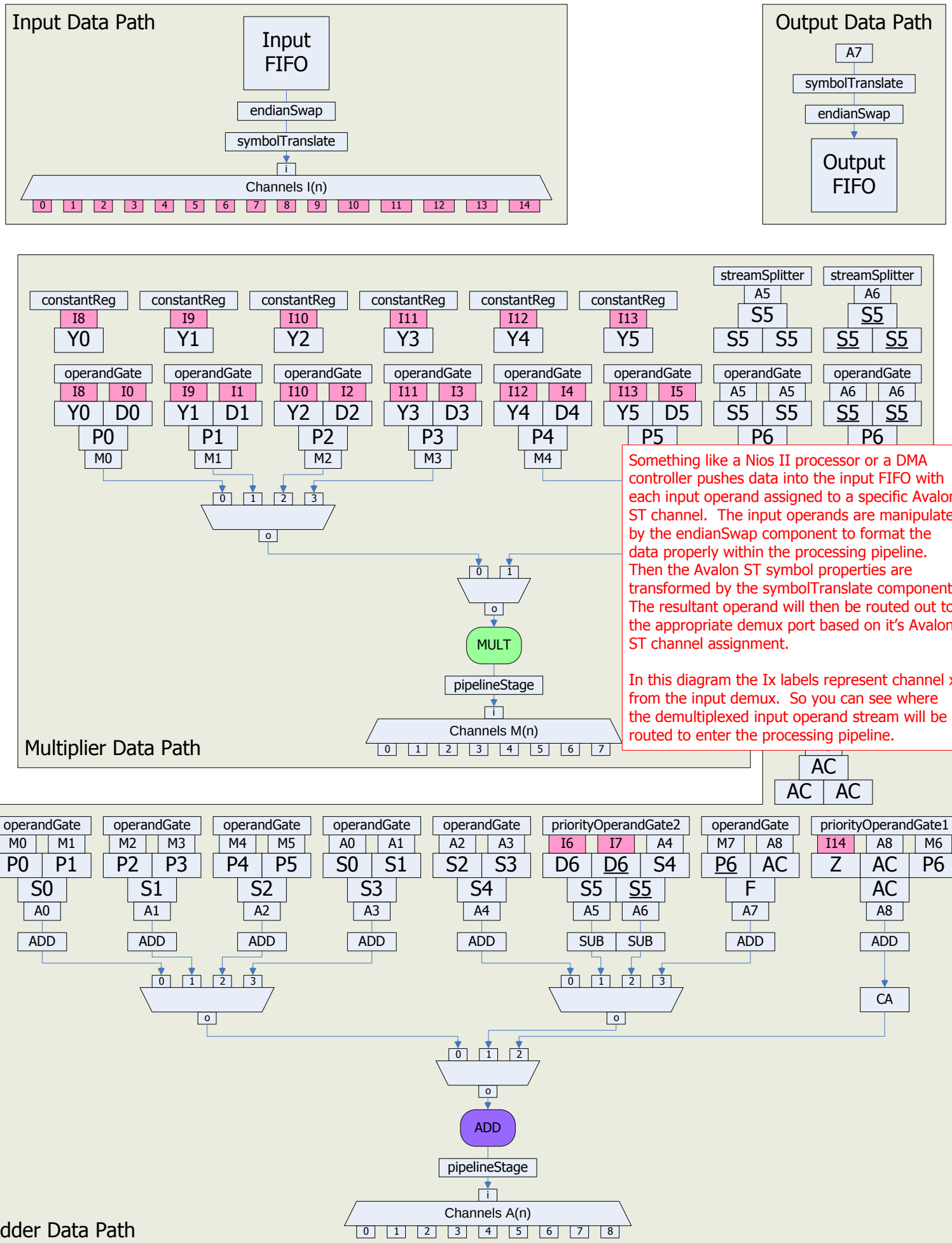


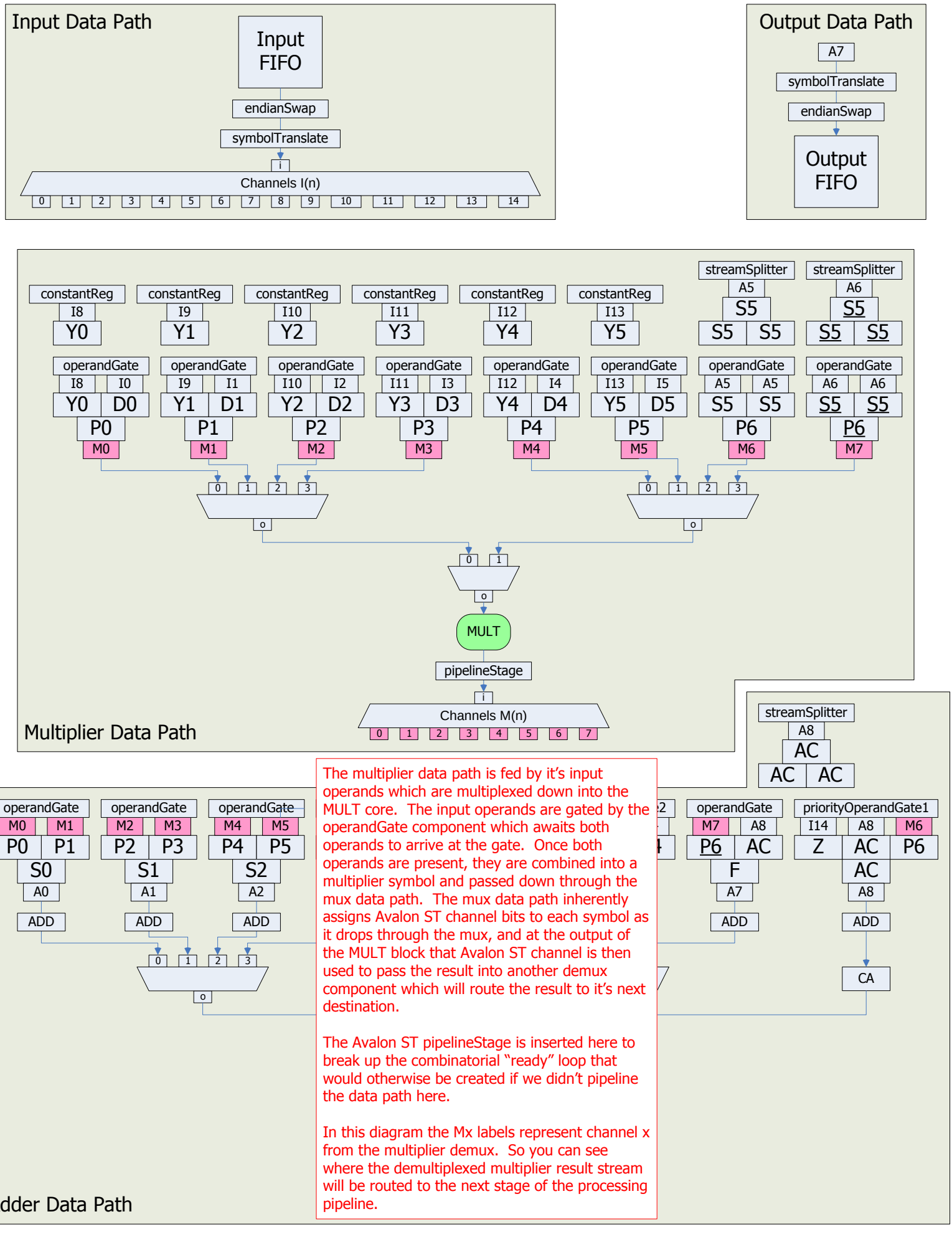

```
double calculate_F_sw2 ( double x[6], double data[461][7] ) {

        int i;
        double y[6];
        double F = 0.0;
        double temp;

        y[0] = x[0];
        y[1] = x[1];
        y[2] = x[2];
        y[3] = x[3];
        y[4] = cos(x[4]) * x[5];
        y[5] = sin(x[4]) * x[5];

        for( i = 0 ; i < 461 ; i++ ) {
                temp  = data[i][0] * y[0];
                temp += data[i][1] * y[1];
                temp += data[i][2] * y[2];
                temp += data[i][3] * y[3];
                temp += data[i][4] * y[4];
                temp += data[i][5] * y[5];

                temp = data[i][6] - temp;

                F += temp * temp;
        }

        return(F);
}
```

This is the Avalon ST mux / demux data path created to implement the matrix computation pipeline. See descriptions on the following pages.



How it works. Page 1, input operands. Please see the component definitions on the last page of this paper.





The multiplier data path is fed by it's input operands which are multiplexed down into the MULT core. The input operands are gated by the operandGate component which awaits both operands to arrive at the gate. Once both operands are present, they are combined into a multiplier symbol and passed down through the mux data path. The mux data path inherently assigns Avalon ST channel bits to each symbol as it drops through the mux, and at the output of the MULT block that Avalon ST channel is then used to pass the result into another demux component which will route the result to it's next destination.

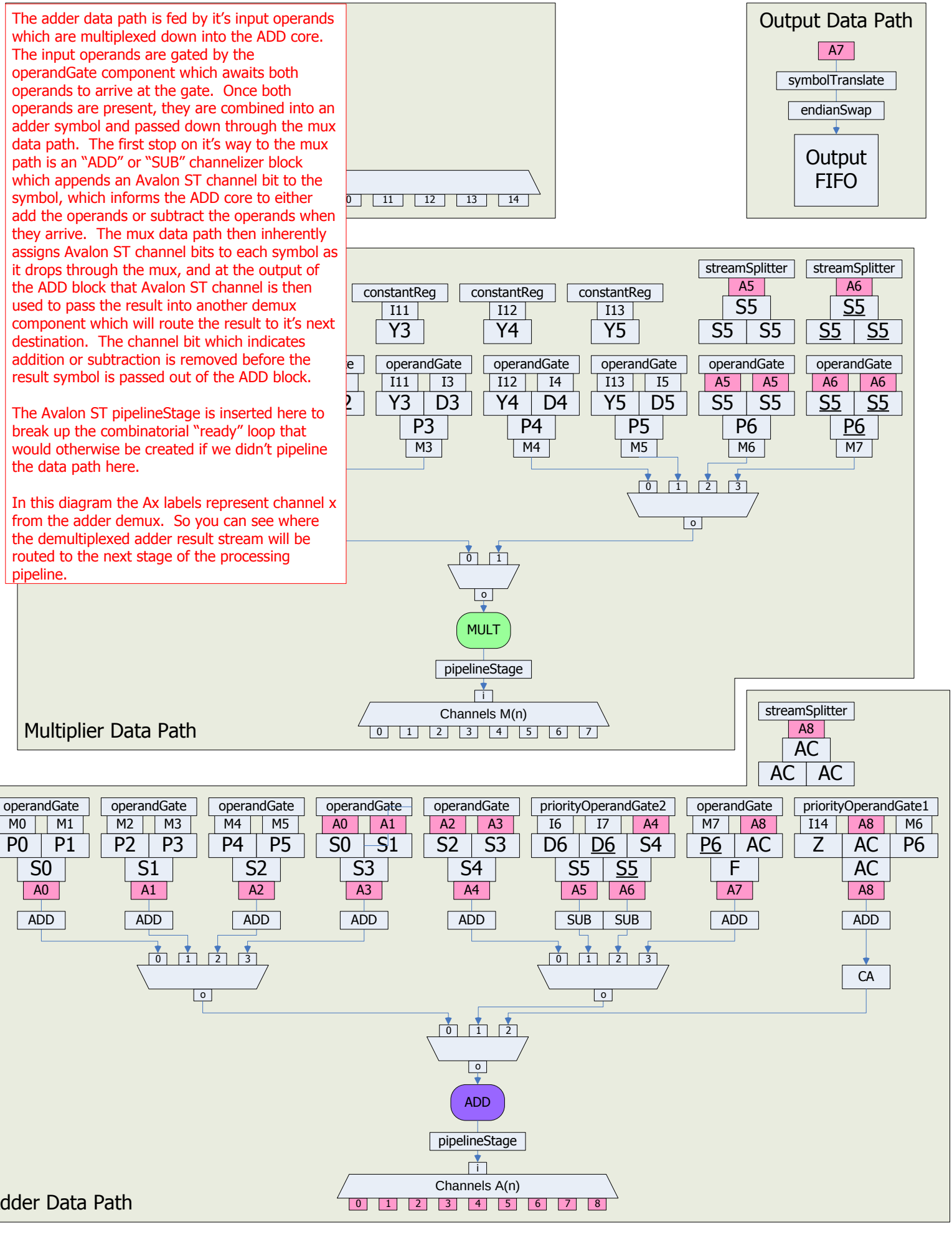
The Avalon ST pipelineStage is inserted here to break up the combinatorial "ready" loop that would otherwise be created if we didn't pipeline the data path here.

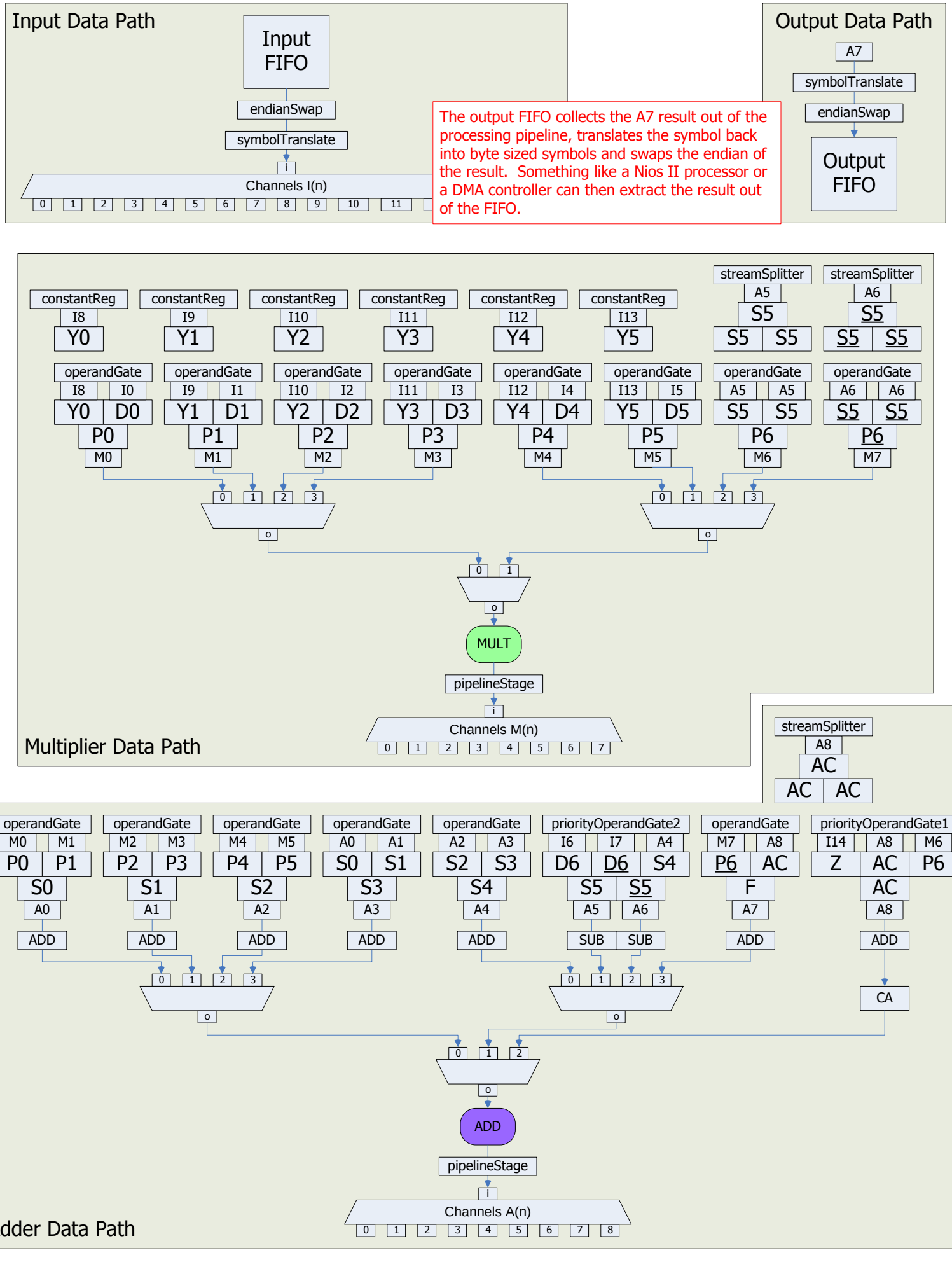
In this diagram the Mx labels represent channel x from the multiplier demux. So you can see where the demultiplexed multiplier result stream will be routed to the next stage of the processing pipeline.

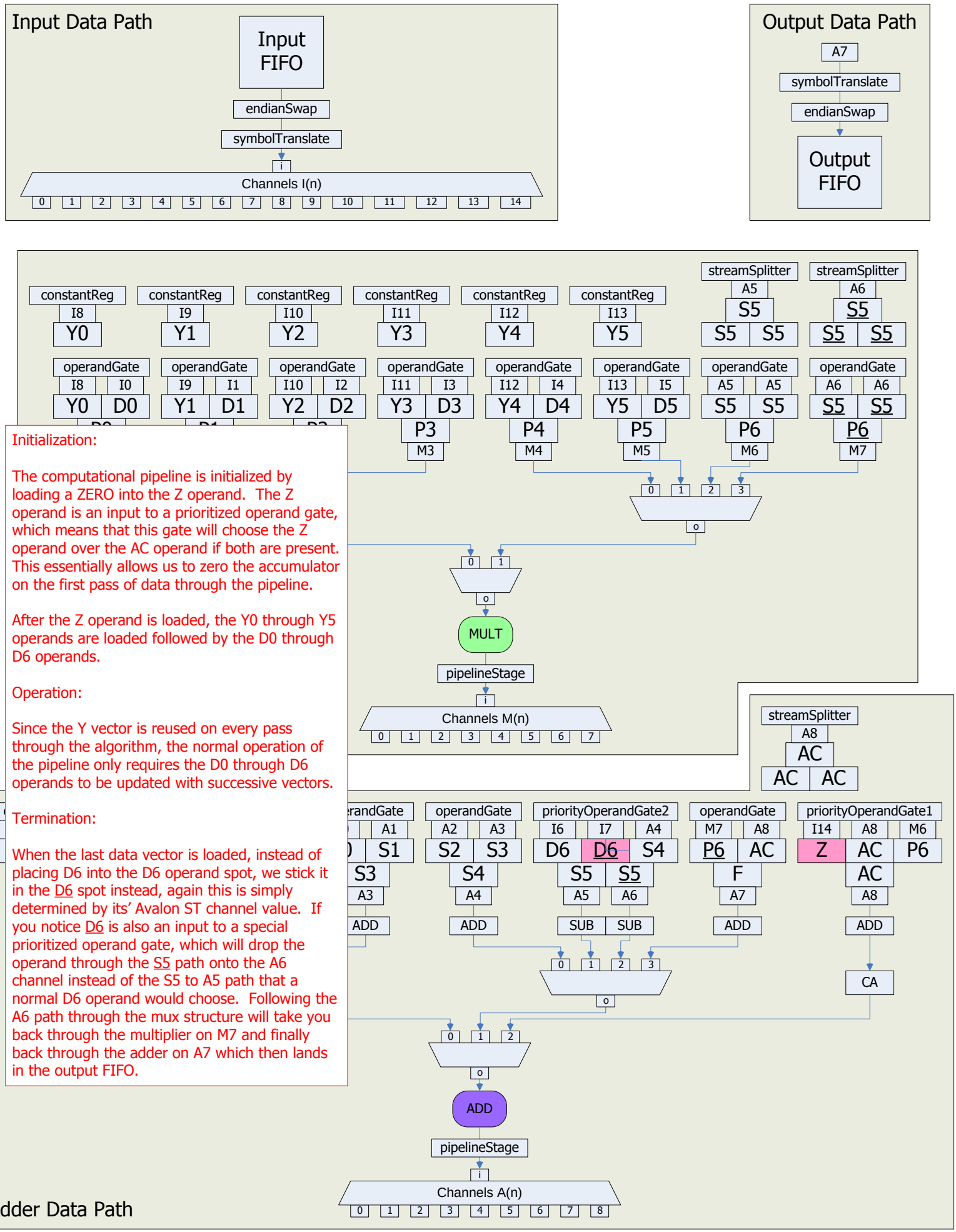
The adder data path is fed by it's input operands which are multiplexed down into the ADD core. The input operands are gated by the operandGate component which awaits both operands to arrive at the gate. Once both operands are present, they are combined into an adder symbol and passed down through the mux data path. The first stop on it's way to the mux path is an "ADD" or "SUB" channelizer block which appends an Avalon ST channel bit to the symbol, which informs the ADD core to either add the operands or subtract the operands when they arrive. The mux data path then inherently assigns Avalon ST channel bits to each symbol as it drops through the mux, and at the output of the ADD block that Avalon ST channel is then used to pass the result into another demux component which will route the result to it's next destination. The channel bit which indicates addition or subtraction is removed before the result symbol is passed out of the ADD block.

The Avalon ST pipelineStage is inserted here to break up the combinatorial "ready" loop that would otherwise be created if we didn't pipeline the data path here.

In this diagram the Ax labels represent channel x from the adder demux. So you can see where the demultiplexed adder result stream will be routed to the next stage of the processing pipeline.




Input Data Path
Input FIFO
endianSwap
symbolTranslate
i
Channels I(n)
0 1 2 3 4 5 6 7 8 9 10 11
Output Data Path
A7
symbolTranslate
endianSwap
Output FIFO
The output FIFO collects the A7 result out of the processing pipeline, translates the symbol back into byte sized symbols and swaps the endian of the result. Something like a Nios II processor or a DMA controller can then extract the result out of the FIFO.
constantReg I8 Y0
constantReg I9 Y1
constantReg I10 Y2
constantReg I11 Y3
constantReg I12 Y4
constantReg I13 Y5
streamSplitter A5 S5 S5 S5
streamSplitter A6 S5 S5 S5
operandGate I8 I0 Y0 D0 P0 M0
operandGate I9 I1 Y1 D1 P1 M1
operandGate I10 I2 Y2 D2 P2 M2
operandGate I11 I3 Y3 D3 P3 M3
operandGate I12 I4 Y4 D4 P4 M4
operandGate I13 I5 Y5 D5 P5 M5
operandGate A5 A5 S5 S5 P6 M6
operandGate A6 A6 S5 S5 P6 M7
0 1 2 3 o
0 1 o
MULT
pipelineStage
i
Channels M(n)
0 1 2 3 4 5 6 7
Multiplier Data Path
streamSplitter A8 AC AC AC
operandGate M0 M1 P0 P1 S0 A0 ADD
operandGate M2 M3 P2 P3 S1 A1 ADD
operandGate M4 M5 P4 P5 S2 A2 ADD
operandGate A0 A1 S0 S1 S3 A3 ADD
operandGate A2 A3 S2 S3 S4 A4 ADD
priorityOperandGate2 I6 I7 A4 D6 D6 S4 S5 S5 A5 A6 SUB SUB
operandGate M7 A8 P6 AC F A7 ADD
priorityOperandGate1 I14 A8 M6 Z AC P6 AC A8 ADD CA
0 1 2 3 o
0 1 2 o
ADD
pipelineStage
i
Channels A(n)
0 1 2 3 4 5 6 7 8
Adder Data Path



Initialization:

The computational pipeline is initialized by loading a ZERO into the Z operand. The Z operand is an input to a prioritized operand gate, which means that this gate will choose the Z operand over the AC operand if both are present. This essentially allows us to zero the accumulator on the first pass of data through the pipeline.

After the Z operand is loaded, the Y0 through Y5 operands are loaded followed by the D0 through D6 operands.

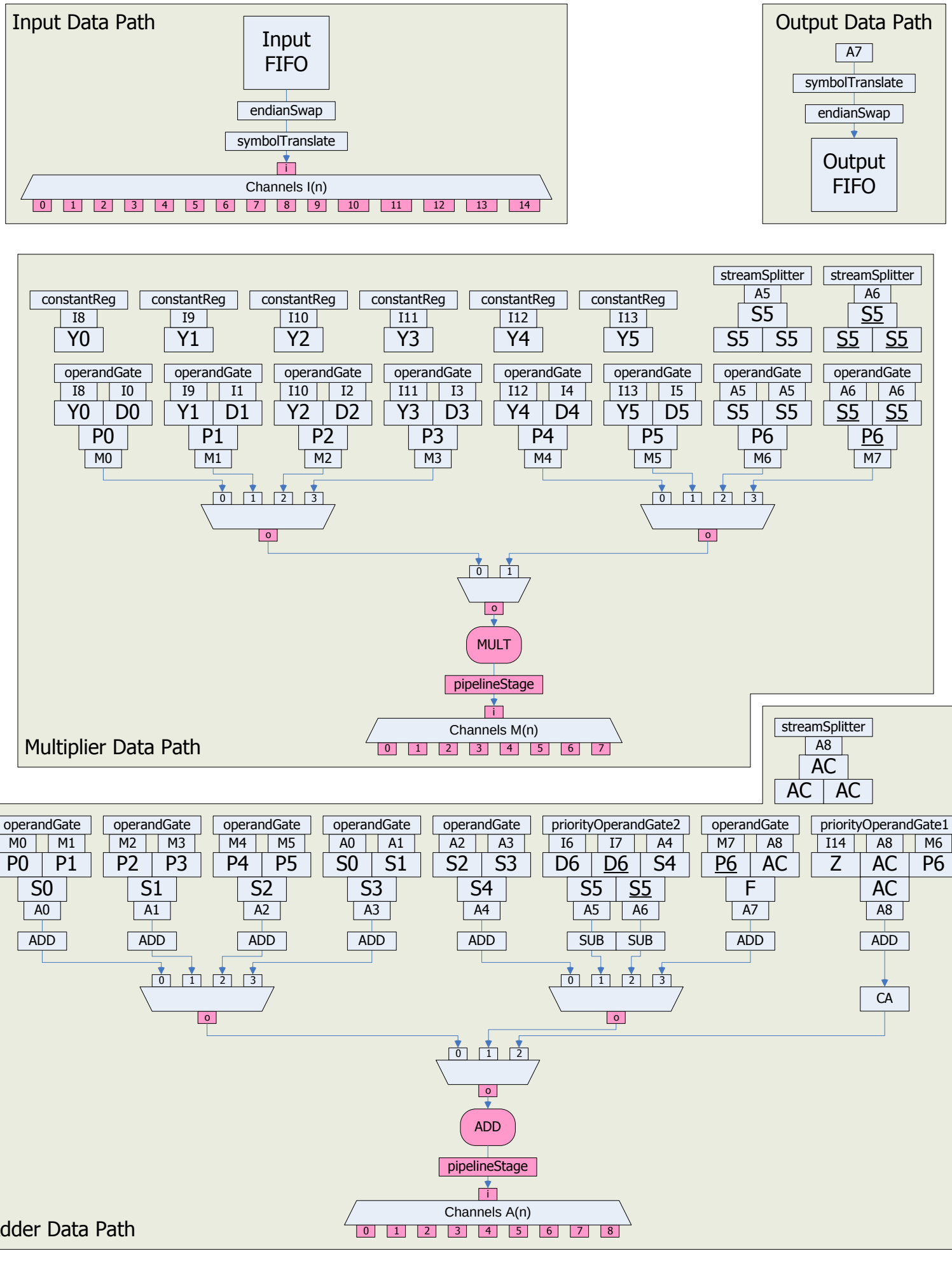
Operation:

Since the Y vector is reused on every pass through the algorithm, the normal operation of the pipeline only requires the D0 through D6 operands to be updated with successive vectors.

Termination:

When the last data vector is loaded, instead of placing D6 into the D6 operand spot, we stick it in the D6 spot instead, again this is simply determined by its' Avalon ST channel value. If you notice D6 is also an input to a special prioritized operand gate, which will drop the operand through the S5 path onto the A6 channel instead of the S5 to A5 path that a normal D6 operand would choose. Following the A6 path through the mux structure will take you back through the multiplier on M7 and finally back through the adder on A7 which then lands in the output FIFO.

The registers contained in the mux / demux blocks become the holding registers required by the various stages of the computational pipeline.

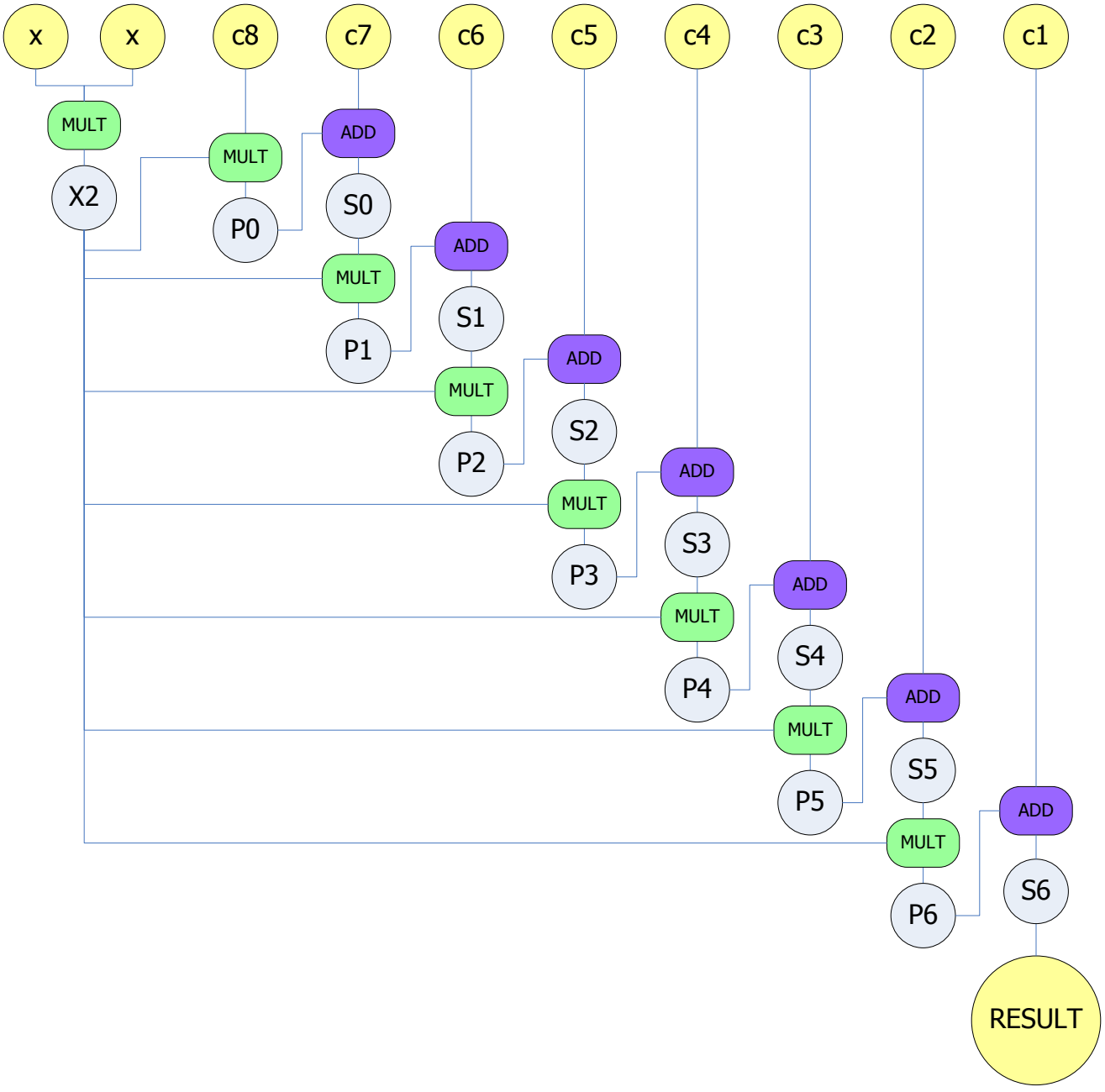


You may have noticed the sin() and cos() requirements in the initial vector setup from the matrix manipulation source code. These trigonometric operations are fairly computationally challenging in and of themselves. But if you analyze the structure used by common approximation algorithms, you find that the processing structure of the approximation is not very different from the previous matrix manipulations that we just looked at.

Below we show what one such approximation could look like in the same format that we used before.

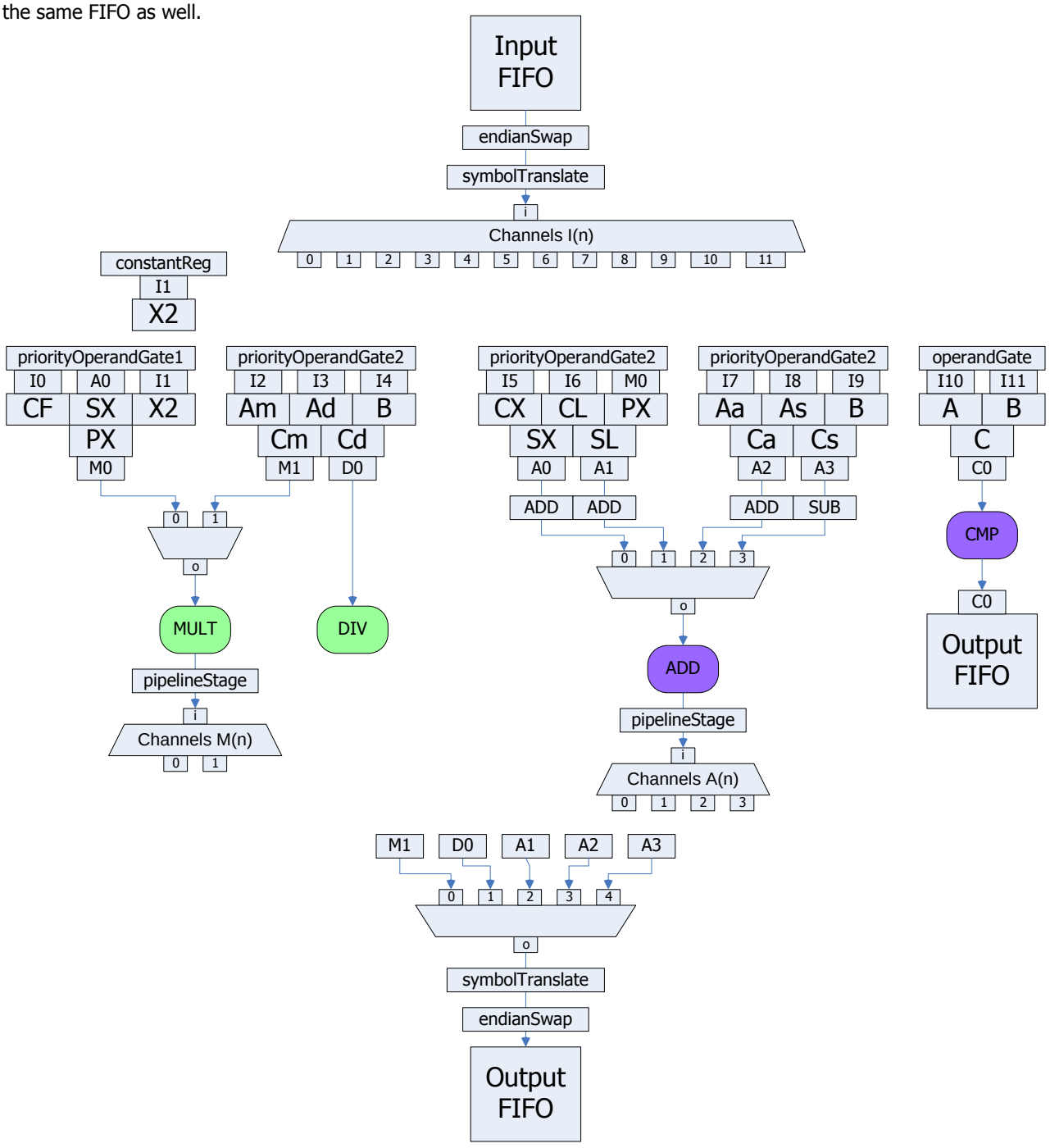
Cosine Approximation from http://www.ganssle.com/approx/approx.pdf

X2 = X*X;
cos(x) = c1 + X2 * (c2 + X2 * (c3 + X2 * (c4 + X2 * (c5 + X2 * (c6 + X2 * (c7 + X2 * c8))))));



While the heart of the cosine approximation relies on the multiply accumlate operations, the preparation for the main loop of the approximation can make use of multiplication, division, addition, subtraction and various compare operations. The Avalon ST mux / demux data path shown below can give access to an ALT_FPMULT, ALT_FPDIV, ALT_FPADDSUB and ALT_FPCOMP LPM blocks to allow the full cosine approximation algorithm to leverage acceleration for all of its' floating point operations.

It should be apparent as well, that if we wanted to combine the picture below, with the matrix manipulation picture from the previous pages, this could easily be done by merging the mux / demux paths from the similar resources in this picture, with those from the previous picture. The most obvious data paths which might be worth merging in order to share the resources would be for the adder and multiplier. The input FIFO may be a good candidate as well, unless you wish to maintain a parallel path to inject the cosine approximation operands simultaneously with the matrix operands. The output FIFO is probably best to leave separate, unless you don't mind merging the cosine results and the matrix results out the same FIFO as well.



Cosine Approximation Variables
CF = first coefficient for approximation
CX = intermediate coefficients for approximation
CL = last coefficient for approximation
X2 = x-squared term used in approximation
SX = intermediate sum in approximation
SL = last sum in approximation
PX = intermediate product in approximation

Multiplier / Divider variables
Am = operand A driven into multiplier
Ad = operand A driven into divider
B = operand B driven into multiplier and divider

Addition / Subtraction variables
Aa = operand A driven into adder
As = operand A driven into subtractor
B = operand B driven into adder and subtractor

Conclusion

Hopefully this example has sufficiently illustrated how we can take very high performance and area consuming IP cores like the double precision floating point LPM cores and leverage the simple data path multiplexing and demultiplexing nature of the Avalon ST interface topology to create a very efficient data processing pipeline within an SOPC Builder system which effectively reuses these IP cores to accomplish the desired algorithmic outcome. It should also be apparent that the flexible topology described by this example can be easily modified or augmented to change the characteristics of the underlying algorithm or combine additional algorithms which can simultaneously leverage the existing processing pipeline that has been established.

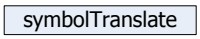
Some subtle points that may not be clearly stated in the diagrams for this example are that the ALT_FPxx LPM cores are totally pipelined, so when they are used to create SOPC Builder Avalon ST based cores, they can fully leverage the back pressure and forward enablement policies that the Avalon ST interface defines. So when the pipeline is stalled due to congestion, these cores can be stalled and back pressure the multiplex tree that feeds them with data as well. This pipelined implementation of the ALT_FPxx LPM cores also means that they are capable of accepting one new operation on every clock cycle in the pipeline, so they are not bottlenecks in the data path pipeline by themselves, and this provides an enormous amount of computational density within the pipeline.

The fact is that in a typical application the processing performance of this data path pipeline would likely be limited by the memory bandwidth of the external system that is feeding the data into this pipeline. Once the data enters the pipeline, the natural flow control of the Avalon ST interfaces will pace the scheduling of operands flowing through the pipeline. So there is no explicit requirement for a state machine to control or orchestrate the processing pipeline, the natural arrival times of the operands is all that is required to trigger the proper sequencing of the pipeline.

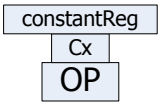
The details of the custom components that were defined in this example are documented on the final page of this paper.

Descriptions of the various components that were created to facilitate this example.

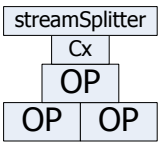
The endian swap peripheral is used to swap the data bytes of the input data and the output data from the memory mapped environment that interacts with this streaming data path through the input and output FIFOs. Since the Avalon MM space is defined as little endian and the Avalon ST space is defined as big endian, we swap the byte symbols around once they ingress into the streaming data path so that they represent properly formatted IEEE-754 numbers. We then swap the result data on egress from the streaming data path as well.



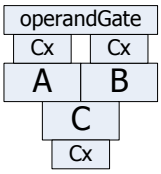
The symbol translation peripheral is used to convert the 8 x 8-bit symbols that flow through the input and output FIFOs into 1 x 64-bit symbols, which are used throughout the streaming data path.



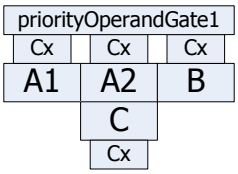
The constant register peripheral is used to pass the value from an upstream mux down to the next Avalon ST sink interface. There is no logic in this peripheral other than the wires that pass the data from the sink to the source and signals that constantly assert ready to the upstream source and valid to the downstream sink. The operands that pass through this block are constantly available to the data path.



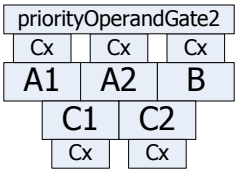
The stream splitter peripheral is used to pass one Avalon ST source down to two different Avalon ST sinks. A ready assertion from either sink interface will propagate up to the source to acknowledge the next data word. This block allows one operand to appear ready at two different sink interfaces.



The operand gate peripheral is used to synchronize two 64-bit operands and release them downstream as a single 128-bit symbol. When operand A and operand B are both ready, this peripheral combines them into operand C and signals valid downstream to the next sink interface. The assumption is that these two input operands will flow downstream towards some arithmetic operation that will operate on them and produce a result.

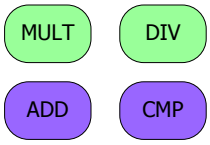


The priority operand gate 1 peripheral is used to synchronize two of three input operands much like the operand gate peripheral above. This gate chooses A1 over A2 if both are ready, then combines that operand with B to forward downstream.



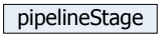
The priority operand gate 2 peripheral is used to synchronize two of three input operands much like the operand gate 1 peripheral above. This gate chooses A1 over A2 if both are ready, then combines that operand with B to forward downstream. When A1 is selected, the resultant symbol is forwarded out the source data path C1 and when A2 is selected, the resultant symbol is forwarded out the source data path C2.

The assign ADD channel peripheral and assign SUB channel peripherals are used to append a channel bit to indicate whether the operands contained in the symbol should be added or subtracted. When the operands arrive at the ALT_FPADDSUB LPM block this channel bit is used to select the addition or subtraction operation for the operands.



The arithmetic peripherals are created by parameterizing the standard ALT_FPMULT, ALT_FPADDSUB, ALT_FPDIV and ALT_FPCOMP LPM functions for double precision floating point mode, and then an AvalonST wrapper is tied around these to allow them to be inserted into an SOPC builder system and connected to the other streaming interfaces. Each of these accept their operands as 1 128-bit symbol and all but the compare LPM will produce a result as a 64-bit symbol. The ADDSUB module requires a single channel bit to designate an operation as addition or subtraction. All but the comparison module will pass through any channel bits presented at the input sink interface.

Aside from the standard AvalonST mux and demux peripherals that this example uses, it also leverages a standard channel adapter to insert channel bits where a mux is not required.



There is also a pipeline stage peripheral inserted at key locations in order to break the combinatorial path that is created by the ready signal that loops through the mux and demux chains.