Create Efficient FIR Filters Using Virtex and Spartan FPGAs

The Virtex and Spartan-II LUTs, configured as shift registers combined with Xilinx True Dual-Port™ RAM, give you a very compact, flexible, and area-efficient FIR filter design platform.

by Rotem Gazit
Design Engineer, MystiCom LTD.
rotemg@mysticom.com

A Finite Impulse Response (FIR) filter works by multiplying a vector of the most recent N data samples by a vector of coefficients and summing the elements of the resulting vector. In every cycle the filter receives a new sample of data and shifts out the oldest sample. FIR filters are very common in FPGA-based Digital Signal Processing applications.

The design concept described here is suitable for systems with relatively low input rates (0.5 to 8 MHz), which require a FIR filter implementation with hundreds of taps; this is common in modem and demodulation applications.

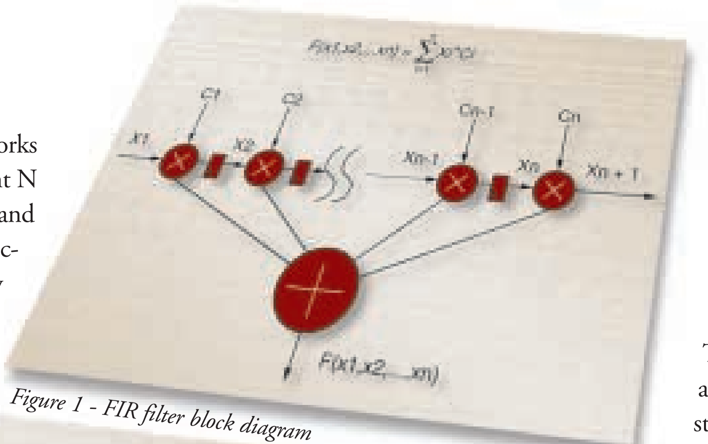

Figure 1 - FIR filter block diagram

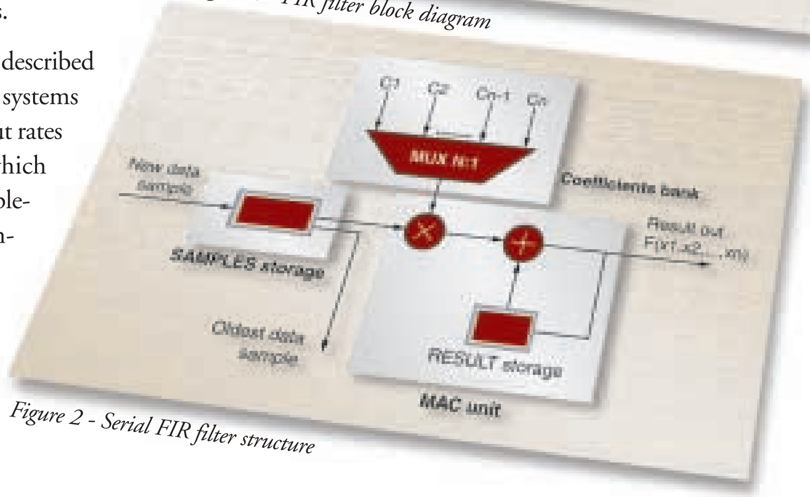

Figure 2 - Serial FIR filter structure

FIR Filter Design Concepts

By examining the FIR block diagram in Figure 1, you can see that if the filter is implemented in a straight forward manner, a multiplier will be required for every filter tap (N multipliers for an N-tap filter). In addition, an adder with N inputs will be needed to sum all multipliers outputs. However, if the data input rate is slower than the performance capability of the FPGA, the filter can be implemented much more efficiently.

Serial FIR Filters

Assuming that the performance capability of the FPGA is M times faster than the data input rate, we will examine the case where M is ≥ N (where N is the required number of filter taps).

To implement a serial N-tap filter uses only one multiplier, a 2-input adder, and storage for the partial results and the filter input samples. The input sample storage holds the last N input samples. For every new sample entering the filter, N multiply operations will be performed, each multiplying the filter coefficient by the respective input sample.

The result of each multiply operation is added to the partial result storage to produce a new partial result. This newly calculated partial result is then saved in the partial result storage by replacing the previous partial result. After N such multiply and add operations, the partial result storage content is driven out of the filter. The partial result storage content is then cleared to begin processing a new data sample. A block diagram of serial FIR filter structure is shown in Figure 2.

The hardware responsible for the combination of multiplying, adding, and storing is called a MAC (Multiply Accumulate) unit. Due to the serial nature of the filter, the MAC will operate on M taps of the filter. In the case where N is greater than M, several serial filters can be chained together. The oldest data sample leaving the first filter in the chain is used as the new data sample in the next filter, and so on. The results of all the chain filters must be added together.

Implementing a Serial FIR Filter

You can implement Serial FIR filters very efficiently in Virtex and Spartan-II devices. The design can be divided into three separate units: the coefficients bank, the MAC unit, and the input sample storage.

Coefficients Bank

The Virtex block RAM can be used to hold the filter coefficients. No multiplexer is needed; all you need is a simple cyclic counter used as an address generator. In systems where a host DSP or an adaptation mechanism is present, the block RAM can be configured as a dual port RAM, enabling the coefficients to be dynamically changed during the normal filter operation.

MAC Unit

The MAC unit consists of an adder, a multiplier, and result storage. Careful design of the adder and multiplier is very important for area efficiency.

Theoretically, the result of a 2^x tap filter, which has 2^y bits on every input data and 2^z bits on every coefficient, will be $2^{(x+y+z)}$ bits wide. In real world applications however, the number of bits in the result is usually much smaller because the least significant bits of the result are usually ignored in the final result, after processing. It is very important to throw away those unnecessary bits as early as possible in the data processing (in the MAC multiplier and adder).

An example MAC implementation is shown in Figure 3.

Input Samples Storage Unit

The input data storage unit can be implemented very efficiently in Virtex devices using the LUTs as shift registers. Each MAC, operating on M taps of the filter, requires an input data storage of M-1 stage delay line. During the first M-1 cycles, the delay line output is driven both to the MAC and back to the delay line input.

In the Mth cycle, the delay line output is driven only to the MAC, and the new input data sample enters the delay line. If several filters are chained together, then the

Throwing away bits in the MAC can sometimes lead to different results than you get from throwing away the bits from the final result; a thorough discussion of the effect of such an operation on the filter performance is beyond the scope of this article.

```
///////////////////////////////////////////////////////////////////
// Name:mac
//————————————————————————-
// Target device:
//————————————————————————-
// Module description:
//————————————————————————-
// MAC of 16 bit coefficient by 5 bit input data_sample.
// the result is 22 bits wide
//
// Parent:
//————————————————————————-
// filter_top
//
// childrens:
//————————————————————————-
//mac_adder.v ,mac_multiplier.
///////////////////////////////////////////////////////////////////

module mac (coefficient,data_sample,rst,clk,enable,new_data,out);

input [15:0] coefficient;     //filter coefficient coming from coefficient storage
input [4:0] data_sample;  //filter data_samplescoming from samples storage
input clk,enable,rst;
input new_data;          //indicates a new data sample. new_data goes high for one cycle
                         //every 64 clocks, 3 clocks after the new data arrives
                         //Because of MAC pipeline.

output [21:0] out;       // MAC output.
reg [21:0]  out;         // MAC output changes whenever a new data is being processed.

wire [16:0] mul_out;   // mac_multiplier output.
wire [21:0] add_out;   // mac_adder output.

reg [21:0] add_out_d;    // sampled mac_adder output.
reg [16:0] mul_out_d;    // sampled mac_multiplier output.


mac_multiplier mac_multiplier(.coefficient(coefficient),.data_sample(data_sample),.mul_out(mul_out) );

always @(posedge clk or negedge rst) // sample the multiplier output
begin              // to improve timing
   if (!rst)
   mul_out_d <= #2 17'b0;
   else
   mul_out_d <= #2 mul_out;
end

mac_adder mac_adder(.adder_out(add_out),.adder_in_0(mul_out_d),.adder_in_1(add_out_d) );

always @(posedge clk or negedge rst) // sample the adder output
begin                                           // this is the "RESULT storege"
   if (!rst)
   add_out_d <= #2 22'b0;
   else
      if (new_data)   // clear accumulator for new data processing
         add_out_d <= #2 22'b0;
   else
         add_out_d <= #2 add_out;
end


always @(posedge clk or negedge rst) // MAC output changes only when a new data arrives
begin
   if (!rst)
   out <= #2 22'b0;
   else if (enable & new_data)
         out <= #2 add_out;
end

endmodule
```

Figure 3 - An example MAC implementation

```
////////////////////////////////////////////////////////////
// Name:delay_line
//————————————————————-
// Target device:
//————————————————————-
// Module description:
//————————————————————-
// delay line of 63 delays x 5 bit.
// the oldest sample is delayed for 64-clock cycle before driven to the next
// delay line in the chain
//
// Parent:
//————————————————————-
// filter_top
//
// Childrens:
//————————————————————-
//shift5x63.v shift63.v
////////////////////////////////////////////////////////////
module delay_line (new_data_sample,clk,rst,enable, new_data, mac_data,next_mac_data);

input [4:0] new_data_sample;          // new_data sample
input clk,enable,rst;
input new_data;                       // new_data is active every 64 cycle for one cycle ->
                                      // SR mux control (input from it's output OR new_data_sample)

output [4:0] mac_data;                // data for MAC
output [4:0] next_mac_data;           // data for next MAC in chain
reg  [4:0] next_mac_data;             // Hold next_mac_data back for one MAC cycle
                                      // (64 clock cycles)

wire [4:0] mac_data;

shift5x63 shift5x63(.din(new_data ? new_data_sample : mac_data) ,
          .clk(clk),.enable(enable),
          .dout(mac_data)
         );

// Hold next_mac_data back for one MAC cycle (64 clock cycles)
always @(posedge clk or negedge rst)
begin
   if (!rst)
   next_mac_data <= 5'b0;
   else if (new_data & enable)
   next_mac_data <= mac_data;
end

endmodule

module shift5x63 (din, clk,enable, dout);
input [4:0] din;
input clk,enable;
output [4:0] dout;
shift63 bit0(.din(din[0]), .clk(clk),.enable(enable), .dout(dout[0]));
shift63 bit1(.din(din[1]), .clk(clk),.enable(enable), .dout(dout[1]));
shift63 bit2(.din(din[2]), .clk(clk),.enable(enable), .dout(dout[2]));
shift63 bit3(.din(din[3]), .clk(clk),.enable(enable), .dout(dout[3]));
shift63 bit4(.din(din[4]), .clk(clk),.enable(enable), .dout(dout[4]));
endmodule

module shift63 (din, clk,enable, dout);
input din, clk,enable;
output dout;                                 //Synplify automatically infers
                                             //SRL16 for shift register with no reset

reg [62:0] shifter;
always @(posedge clk)
begin
   if (enable)
   begin
   shifter[62:0] <= {shifter[61:0],din} ;
   end
end
assign dout = shifter[62] ;
endmodule
```

Figure 4 - An example of a LUT SRL16-based delay line implementation

delay line output needs to be held for M cycles before it is driven as an input to the next filter in the chain. Sometimes (depending on the available resources inside the device) it is better to implement the delay line using block RAM configured is a simple FIFO.

An example of a LUT SRL16-based delay line implementation is shown in Figure 4. A diagram of the complete serial FIR filter is shown in Figure 5.

Conclusion

FIR filters with many hundreds of taps can be implemented easily even in the smallest members of the Virtex and Spartan-II FPGA families. By taking advantage of the Virtex and Spartan-II architecture, you can implement FIR filters very efficiently.

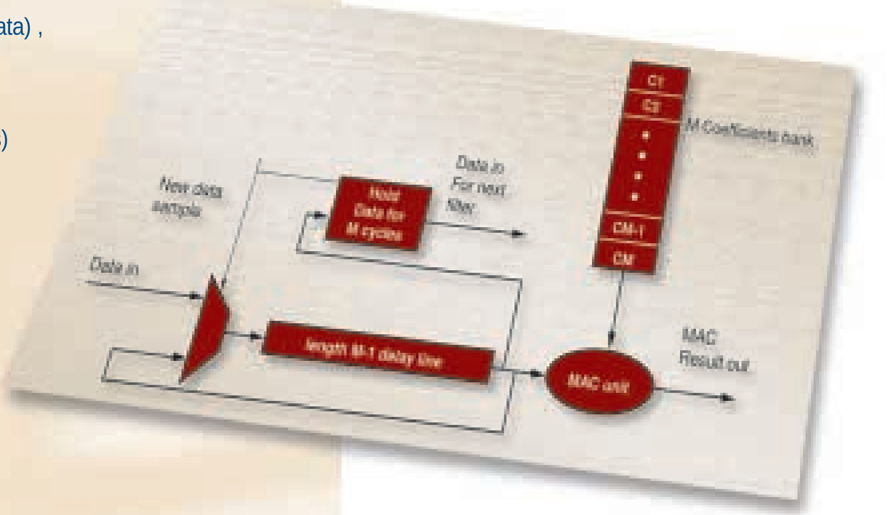

Figure 5 - Serial FIR filter implementation

About MystiCom

Founded in 1997, MystiCom is dedicated to providing DSP and mixed-signal VLSI cores for high-speed communications. The company's first product line implements the physical layer (PHY) for Local Area Networks (LANs) using Fast Ethernet and Gigabit Ethernet protocols. MystiCom is headquartered in Netanya, Israel, and has marketing and customer support offices in Mountain View, Calif. Additional information can be found at www.mysticom.com.