Mentor®
A Siemens Business

FIR Filter Workthrough

Using C/C++

June 2017

Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777.
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: supportnet.mentor.com/

Send Feedback on Documentation: supportnet.mentor.com /doc_feedback_form

Table of Contents

Overview

This document takes you through synthesizing three fundamental memory architectures of a Finite Impulse Response Filter implementation using Catapult and the C++ language. A single simple testbench is also provided in C++ to validate and verify the algorithm and RTL.
The four architectures are:

1) Classic shift register and multiply-accumulate
2) Rotational shift
3) Circular Buffer Single Port RAM storage

By following through the steps in this document, you will gain a greater understanding of how what you code in your C++ affects what HLS can create, and how you can explore the implementation space for a given architecture. Information on the architectures themselves is contained in a separate document.
This is not intended to be training. We assume some familiarity with the basic operation of Catapult and will take you completely through the first architecture with notes on synthesizing the other two
The basic structure of the C++ source code is as follows:

- FIR_TESTBENCH.cpp
- FIR_SIMPLE_REFERENCE.cpp
- FIR_SIMPLE.cpp | FIR_ROTATE.cpp | FIR_CIRCULAR.cpp

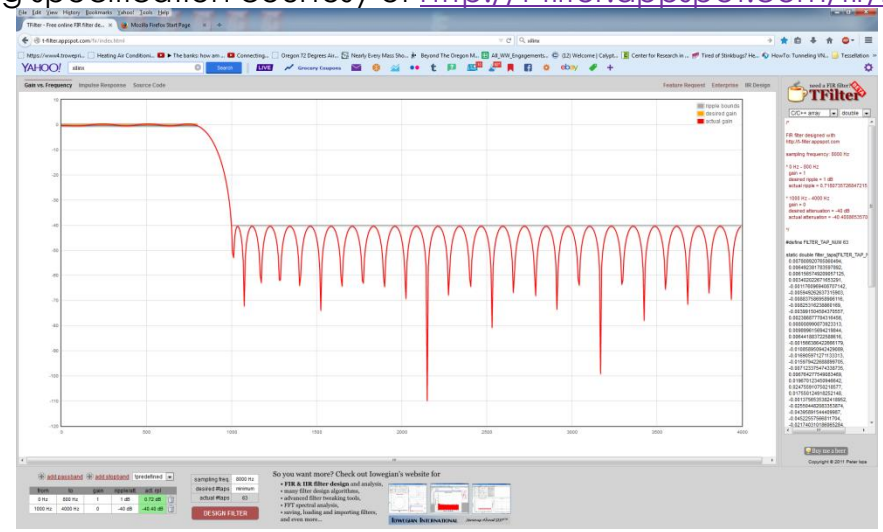
The same fundamental architectures are covered in SystemC and are essentially the same coding style with the functionality wrapped in a SC_CTHREAD
A header file called "FIR_TYPES.h" defines all the data types (and function prototype definitions):

```
#define N_TAPS 63
#define DATA_WIDTH 8
// 16-bit coefficients with rounding to +/- infinity are needed for +/- 0.5 LSB accuracy against
// floating point
// Feel free to change this to trade off area vs precision
#define COEFF_WIDTH 16
// This headroom determines the additional accuracy of the accumulator. Here we make it full
// precision for numerical safety
#define HEADROOM 6

typedef ac_fixed<DATA_WIDTH,DATA_WIDTH,true,AC_RND_INF,AC_SAT> d_type ;
typedef ac_fixed<COEFF_WIDTH,1,true,AC_RND_INF,AC_SAT> c_type ;
typedef ac_fixed<DATA_WIDTH+COEFF_WIDTH+HEADROOM,DATA_WIDTH+HEADROOM+1,true> a_type ;
```

Only three types are used. One for the sample data, one for the coefficient representation, and one for the accumulator precision. Note how the data and coefficient (d_type & c_type) leverage rounding and saturation, but the accumulator type (a_type) does not as it is "full precision".
Refer to the Algorithmic C Datatypes manual for more information about the ac_int and ac_fixed data types.

A header file called "COEFFS_DOUBLE.h" is provided that has filter designed with the following specification courtesy of http://t-filter.appspot.com/fir/index.html

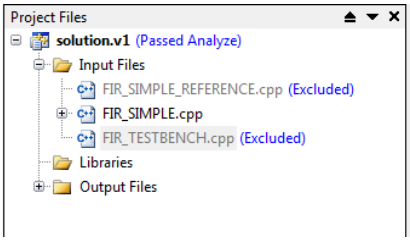


In this example we have a 63-tap low pass filter with a cutoff at $f_S/4$, 1dB of ripple in the passband and -40db rejection in the stop band. This web program gives us a C++ array of coefficients that we simply cut-and-paste into the "COEFFS_DOUBLE.h" file so we have double precision coefficients that can later be quantized to fixed point.

In order to compile, link, and execute this code stand-alone you need to compile the FIR_TESTBENCH.cpp, FIR_SIMPLE_REFERENCE.cpp and one of the four example architectures listed above. You will need the include directory from the Catapult install tree as this example uses the AC data types and AC_channel. For each of the three architectures, the numerical accuracy and functionality externally is identical. It is only the architectures that are different and lead to different hardware implementations.

Starting with the Basic FIR filter

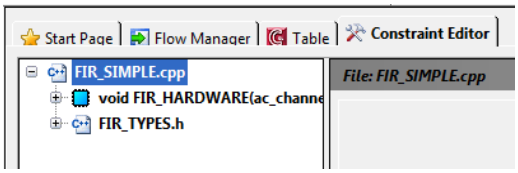
1) Invoke Catapult and add the following files:
   - FIR_SIMPLE.cpp
   - FIR_SIMPLE_REFERENCE.cpp (excluded)
   - FIR_TESTBENCH.cpp (excluded)



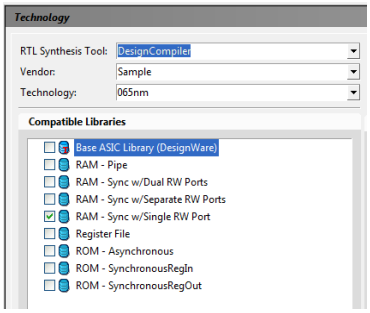
2) Enable the SCVerify Flow in the Flow Manager

   If your environment is not set up for SCVerify, refer to the "Setting Up the SCVerify Flow" section in the online help.

3) Click the "Hierarchy" icon and ensure that the top level is FIR_HARDWARE



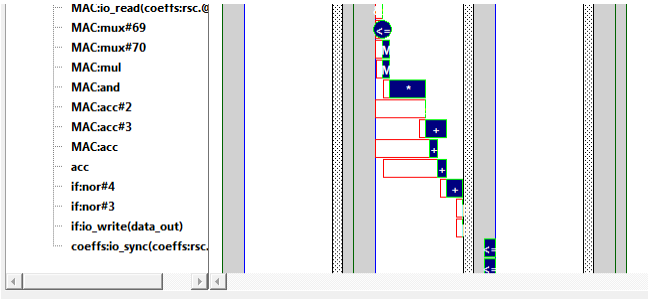
4) Click on "Libraries" and select a target technology that has a RAM. Any library will do, but for this walkthrough we will use the Design Compiler Sample 065nm library and single port RAM



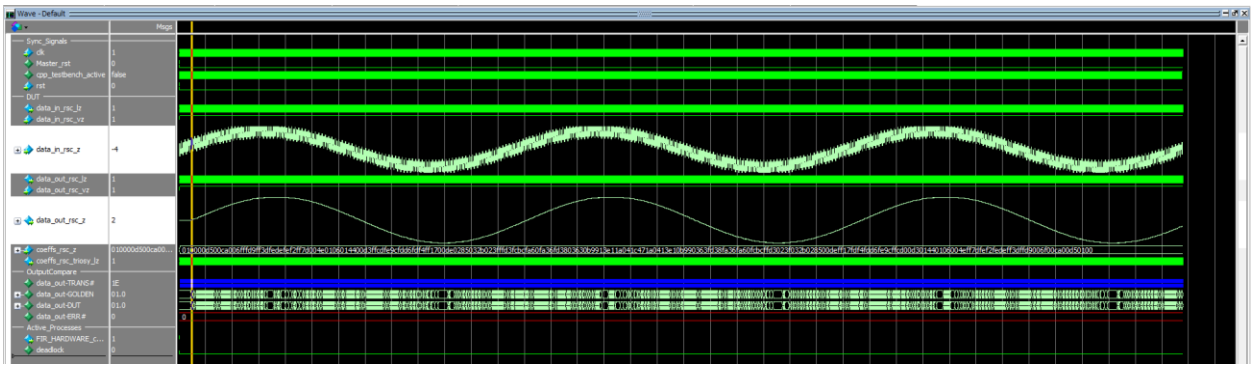
5) Go to the mapping stage and set a reasonable frequency target. In this case we will use 200MHz

6) Bring up architecture constraints and do the following
   a. Unroll the SHIFT loop
   b. Pipeline the main process loop with II=1
   c. Map the taps Array to Registers
   d. Map the coeffs resource to a wire with no handshaking

7) Generate a Schedule to see this single multiplier implementation:

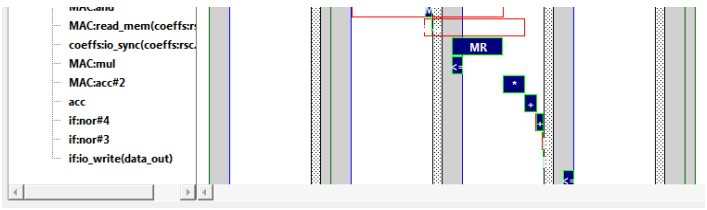


8) Generate RTL
9) If your environment is correctly set up for SCVerify, you can run interactively to validate that the RTL functions correctly
    a. You should be able to see that samples go in and out of the filter every 63 clock cycles
    b. If you plot the waveforms assigned analog values, you will see something like this for the input and output values that are also written to file I/O in the testbench:



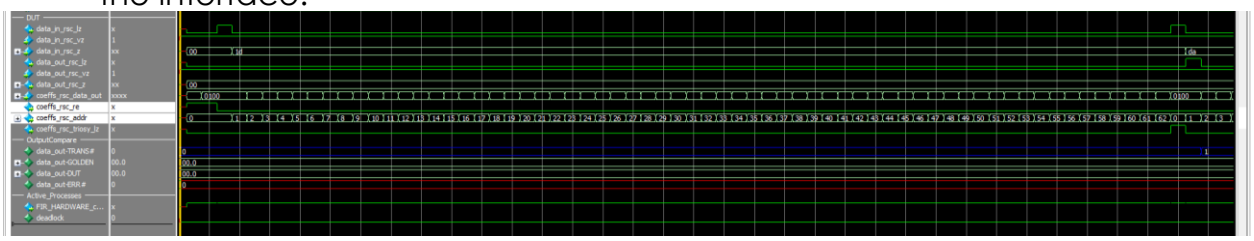
    c. Note that the coefficients are effectively constant wires into the hardware
10) Close the simulator and return to Architecture constraints
11) Make the following changes:
    Map the coefficient interface to a single port RAM – this will limit bandwidth and introduce a synchronous access with a clk2q delay.

12) Create a new Schedule

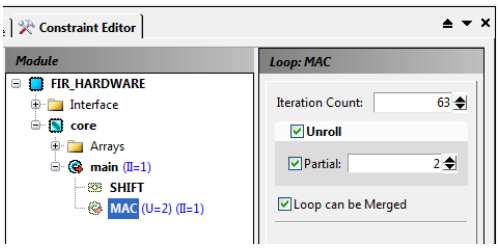


13) Generate new RTL and Run SCVerify

    You should be able to see that the interface for the RAM is now generating address values and getting individual coefficient values from the interface:

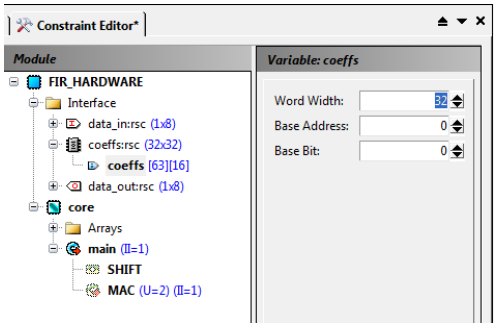


14) Go back to Architecture constraints and partially unroll the MAC loop by 2

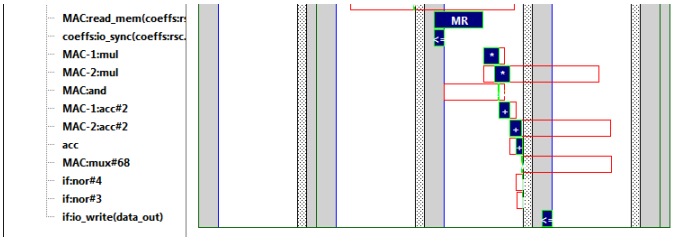


15) Generate a Schedule
The schedule should fail because of the bandwidth limitation on the coeffs RAM.

16) Change the width of the coeffs variable for the RAM resource to 32



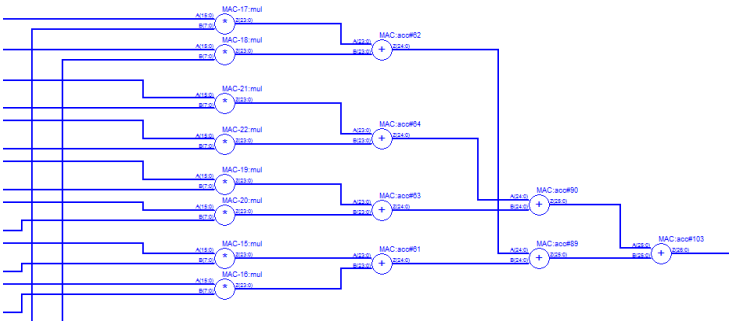
17) Generate a new Schedule
The throughput should now be 32, and the schedule should have two multipliers instead of one:



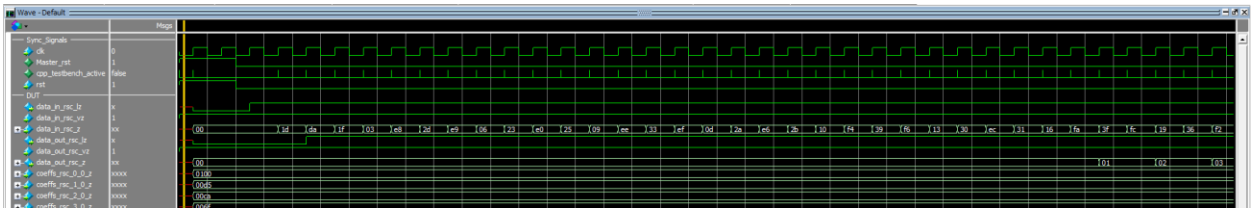
18) Generate RTL and verify that the RTL simulation still functions correctly in SCVerify
You should observe that the throughput is now 32 and the width of the coefficient memory is doubled in order to improve bandwidth

19) Return to Architecture Constraints and do the following:
   a. Fully unroll the MAC loop
   b. Map the coeffs to an in_wire interface again
   c. Change the word width back to 16 for the coeffs variable

20) Generate a new schedule. You should see a classic parallel adders and adder tree implementation that likely spans a few cycles.
21) Generate RTL
22) Take a look at the schematic view
    a. Observe that the coeffs ports have been split into individual 16-bit ports for each index of the input array. This was caused by the word width being 16 instead of 1008
    b. Pushing down into the top level and viewing the schematic as a single page, you should be able to easily see the long shift register and portions of the MAC tree:



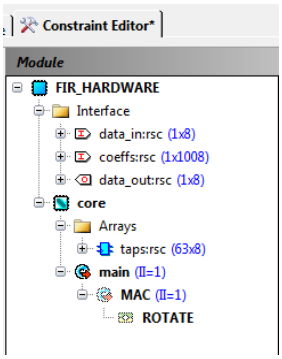
23) Run SCVerify on this implementation to verify that samples enter and exit every clock cycle in the simulation:



That concludes the walkthrough for the simple FIR implementation. We will now look at the Rotational shift architecture

The Rotational Shift FIR filter

1) Use FIR_ROTATE.cpp instead of FIR_SIMPLE.cpp
2) Go through Hierarchy, Libraries and Mapping stages
3) In Architecture Constraints, set the following:
    a. MAC loop should not be unrolled (note it has 64 iterations)
    b. ROTATE loop should be fully unrolled
    c. Top level main loop should be pipelined with II=1
    d. coeffs input should be mapped to wire inputs
    e. taps should be mapped to registers

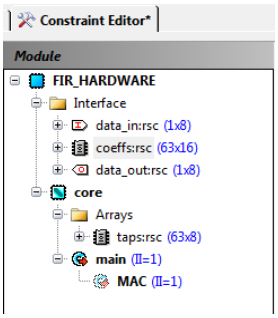


The rotational shift architecture is seldom seen in real usage due to the high power consumption inherent in shifting the entire tap array for every MAC loop iteration.
For filters with large numbers of taps, and/or lower overall data rates, a circular buffer implemented using RAM is often desirable. With appropriate coding, we can leverage a single port RAM for storing the taps, and avoid the area cost and power of the shift register.

The Circular Buffer FIR

Using a RAM will require that our target technology library have RAMs available. If you are using the example libraries that ship with Catapult there is nothing to do. If you are using your own characterized library, you will need to build your own RAM library. See the online help on "Memory Generator" for more information.

1) Use FIR_CIRCULAR.cpp instead of FIR_ROTATE.cpp
2) Work through Hierarchy, Libraries and mapping as usual
3) Now set architecture constraints as follows
    a. MAC loop is not unrolled (note that there is no SHIFT loop)
    b. taps must be mapped to a single port RAM
    c. coeffs can be mapped to a single port RAM interface or wires



That will give an efficient implementation.
If you need more assistance, ask your local FAE for help, or contact support_net@mentor.com