

# Catapult Basic Training 2011a

## Lab 6      Synthesizing our FIR filter in Catapult

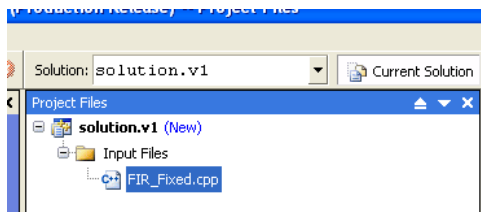
*NB: This is a long lab. Take your time, and try to really understand what is taking place*

This lab will take you through the basic setup of Catapult to synthesize to a given target technology. We will cover the following:

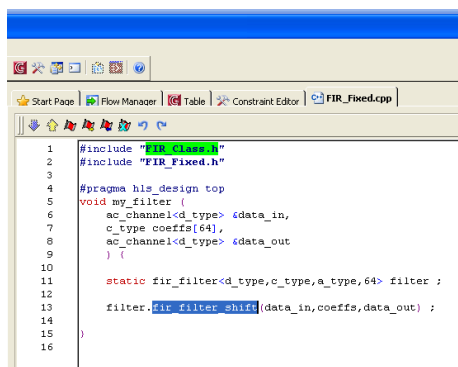
- Interface Synthesis including memories
- Memory width and interleaving optimizations
- Loop unrolling and pipelining
- Scheduler target constraints
- Resource constraints
- Report analysis

### Getting the first solution

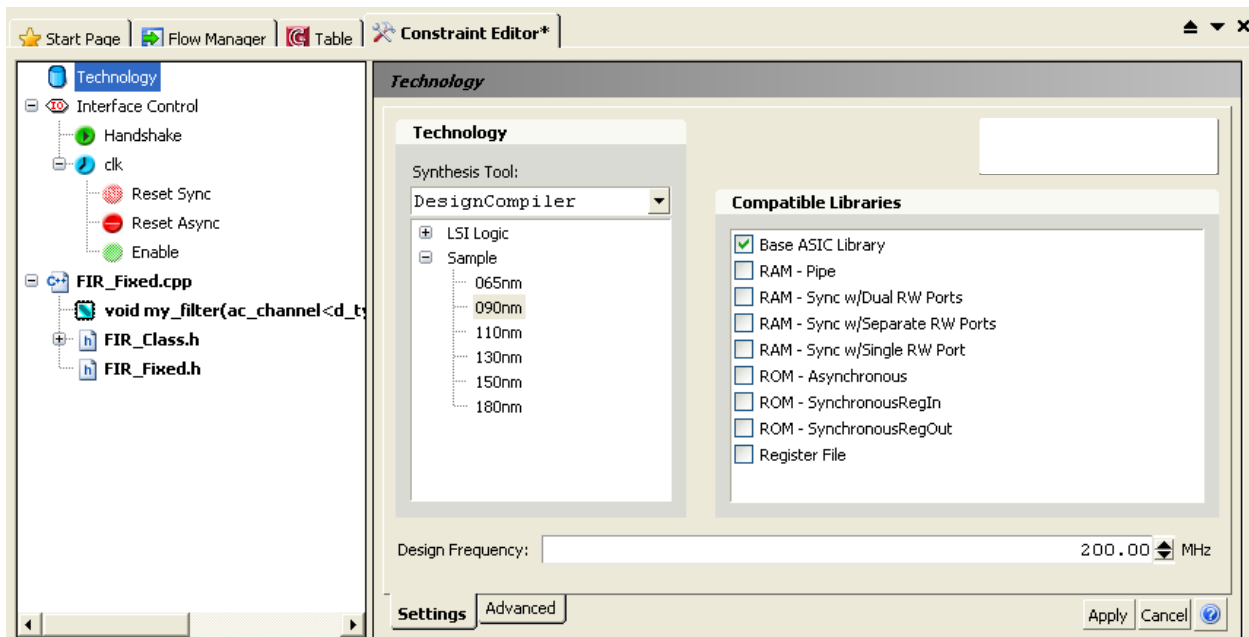
- 1      Invoke Catapult either in the Lab6 Directory or invoke Catapult and set the working directory to the Lab6 Directory
- 2      Add the following file using “Add Input Files”
  - a.   FIR\_Fixed.cpp
- 3      Double-click on the “FIR\_Fixed.cpp” file in the Input Files solution list



- 4      Confirm that the filter is using the “fir\_filter\_shift” member function
- 5      Double-click on the “FIR\_Class.h” #include line of code to open the include file



- 6 Verify that the same “fir\_filter\_shift” and “fir\_filter\_shift\_fold\_even” member functions are defined as you expect.
- 7 Close the “FIR\_Class.h” and “FIR\_Fixed.cpp” files
- 8 From the menu, pick “Tools => Set Options”
- 9 Select the “Output” pick and ensure that both the VHDL and Verilog outputs are checked.
- 10 If both VHDL and Verilog outputs are checked simply click on “OK”
- 11 If both VHDL and Verilog outputs are not checked, check them both and press “Apply & Save”, then click on “OK”
- 12 Click on “Libraries” in the Synthesis Tasks window pane.
- 13 Set the following in the Constraint Editor pane:
  - a. Synthesis Tool: Design Compiler
  - b. Sample, 65nm library (just the default Base ASIC library)
- 14 Click on “Libraries” again, then click on “Mapping”
- 15 Set the following in the Constraint Editor pane:
  - a. Under the “clk”:
    - i. Select a 400 MHz operating frequency
    - ii. In the Advanced tab, uncheck the “Enable Synchronous Reset”. The “Reset Async” should be automatically enabled
  - b. Under “my\_filter”:
    - i. Handshake: Check the “Transaction Done signal” box

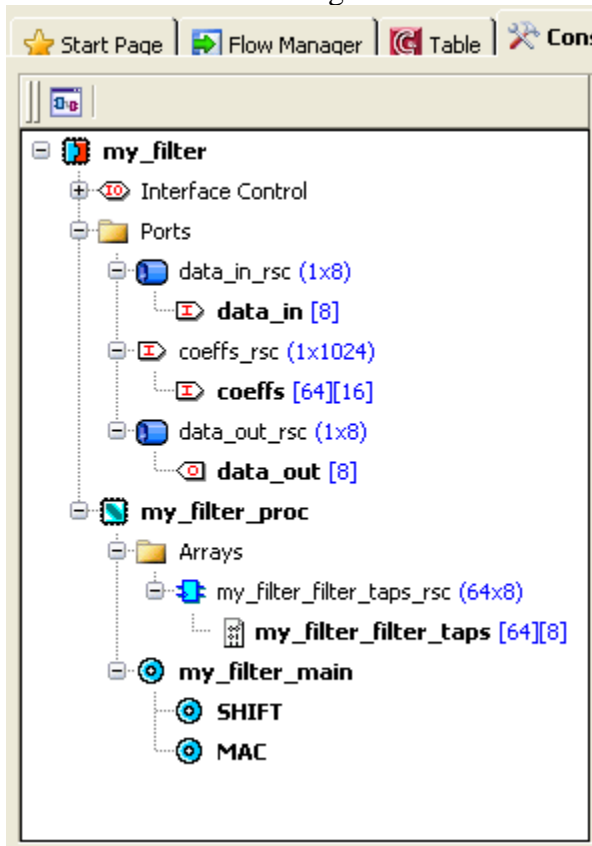


- 16 Once the above setup is done, click “Apply”

Note that in the Transcript window, all the Tcl script commands necessary will be printed to show you what has been set. You should also see all the other icons for “Architectural Constraints” through “Generate RTL” appear in the Task window pane.

- 17 Click on “Architecture”
- 18 Expand the “Interface” folder in the Constraint Editor Tab
- 19 Expand all the Port Resources in order to see the underlying variables
- 20 Verify that the width of the data input and output, as well as the depth of the coefficients are correct.
- 21 Make sure that the coeffs variable is mapped to mgc\_ioport.mgc\_in\_wire\_wait (i.e. a vector of wires)
- 22 Expand the “Arrays” folder and the taps\_rsc to see the underlying my\_filter\_filter\_taps variable. Make sure that it is mapped to [Register]

You should see something like this:

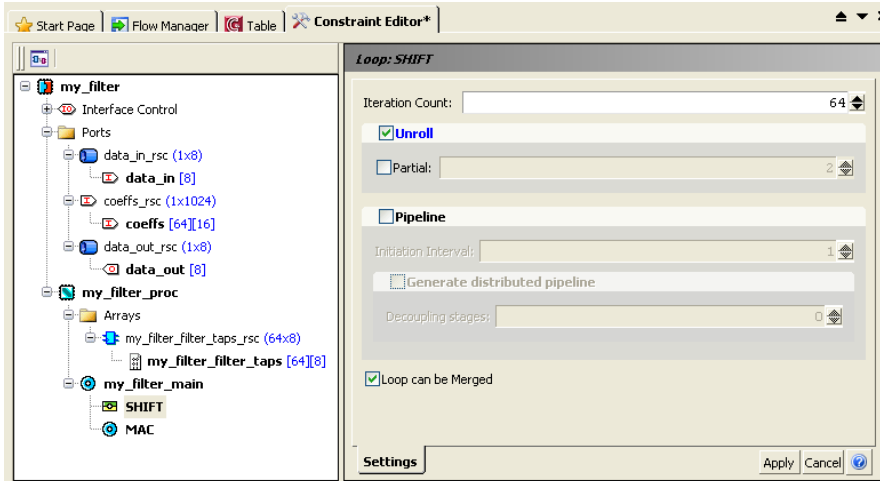


You may have noticed that some items are presented in “**BOLD**”. These objects can be cross-probed in the Constraint Editor by double-clicking

- 23 Double-click on the “SHIFT” loop
- 24 This should take you to the “FIR\_Class.h” file, and highlight the relevant for loop
- 25 Close the “FIR\_Class.h” file

Because we are describing a FIR filter with a conventional delay-line meant to be implemented in registers, where the shifting happens in parallel, we need to fully unroll the SHIFT loop no matter what other microarchitecture we choose to implement.

26 Select the “SHIFT” loop and set it to be fully unrolled:



27 Click “Apply”

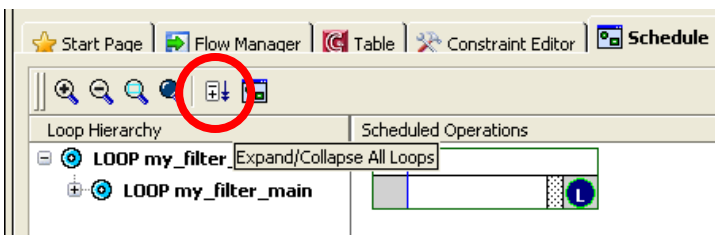
28 Click the “Schedule” Icon

After a few moments, a Schedule will appear on the screen.

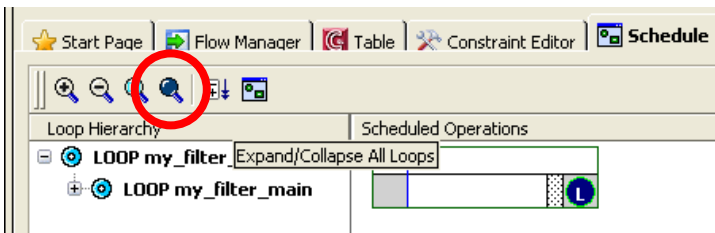
29 Click the “Expand button” for the Schedule window to maximize the window. It looks a bit like a VCR “Eject” symbol:



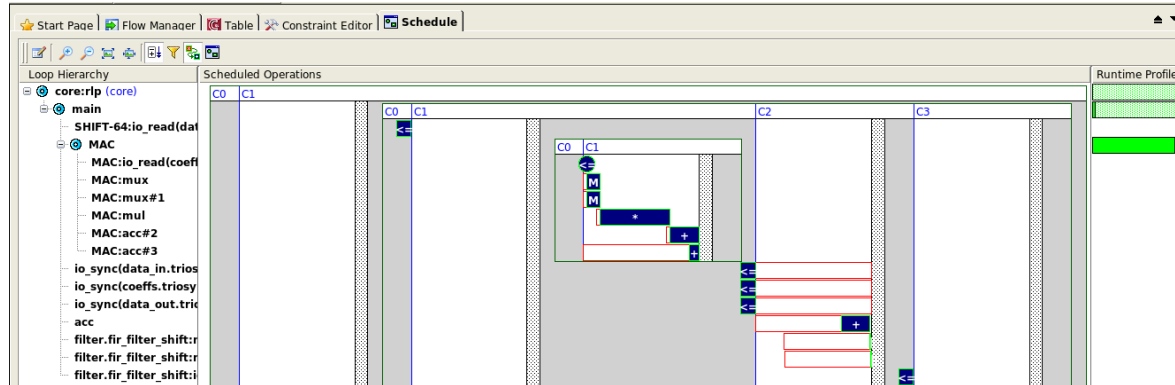
30 Click on the “Expand/Collapse All Loops” Icon:



31 Then click on the “Zoom Full” Icon



You should now see the entire schedule with the “main” and MAC loops:



Note that the Runtime profile shows the vast majority of activity to be in the loop MAC, but there is some time spent in the my\_filter\_main. This can be seen by some small amount of solid green in the Runtime Profile.

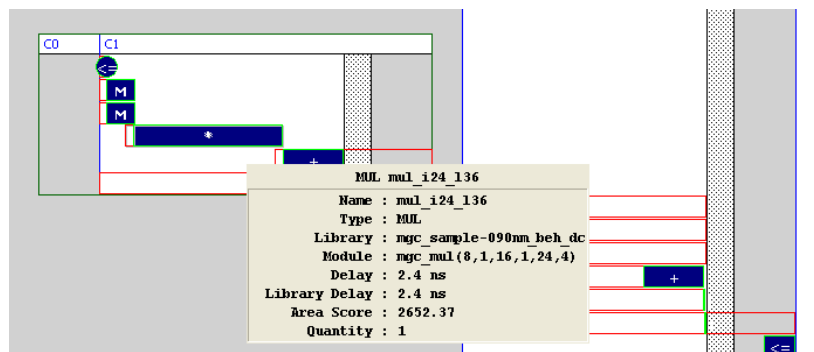
Because the MAC loop has not been unrolled at all, you can see only one multiplier and an accumulator in the loop. You should also see two multiplexer elements. This is because one is used to select the taps, the other to select the coefficients. Interface synthesis has mapped the coefficients to wires, so when using only one multiplier, it is necessary to select the correct index for the loop iteration.

There is also a smaller adder with lots of slack (red bar lines) which is the iterator for the loop iterations.

The adder and NOR gate logic outside the MAC loop at the end of the schedule is the adder and associated logic needed for rounding and saturation. This was derived from the data type we used in our C++ code, rather than explicitly needing to code the behavior.

32 Hover your mouse pointer over the multiplier

A pop-up box should appear:



Note that this pop-up contains the name of the component, the module type, the associated delay and area score derived from the library data.

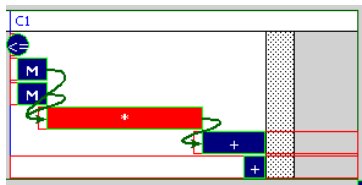
In this case, the `mgc_mul(8,1,16,1,24,4)` indicates that this is a 8-bit signed multiplier by a 16-bit signed with a 24-bit result being needed, and a speed-grade '4' being selected. Speed grades in Characterization for ASIC's typically run from 1 (the fastest) to 4 (the slowest).

### 33 Double Click on the Multiplier

Catapult should cross-probe to the `FIR_Class.h` file and highlights the '\*' that created this multiplier (but it may not, if the information was lost through the synthesis flow due to some optimization). Cross-probing from the Schedule can be very useful when debugging the architecture of C++ code.

### 34 Close the `FIR_Class.h` file

The multiplier in the Schedule should be Red. If it is not, single click on it to select it. You should see that Catapult has drawn dependency lines showing operators that drive, and are driven by, the selected operator:



We can clearly see two operators (they are muxes) feeding into the multiplier, which feeds an arithmetic operator (the accumulator). If you Select the accumulator, you will see a green feedback path from the output to the input.

35 What is the delay of the multiplexers? \_\_\_\_\_

36 What is the delay of the Multiplier? \_\_\_\_\_

37 What is the delay of the accumulator? \_\_\_\_\_

38 What is the total delay of the operators in this C-Step? \_\_\_\_\_

As we are running at 400MHz (2.5 nS) and the "Percent Sharing Allocation" is the default 20%, Catapult has scheduled to ensure that no data path has a delay more than 2ns. 20% of the clock is reserved for FSM control that will be built when the RTL is created.

You should observe that the LOOP `my_filter_main` has three c-steps (C0 is an imaginary c-step) and that the LOOP MAC executes in the C1 step of `my_filter_main`. The I/O write can be seen at the end of C-Step 2, and C-Step 3 is empty.

Therefore, the behavior of this schedule would be that the FSM enters C1 of `my_filter_main`, executes the 64 iterations of the MAC loop, then does the rounding and saturation and writes the

result out at the end of C2, with the FSM then entering C3 to “pause” before being able to return to the start again.

Therefore, there are 3+64 cycles of throughput, but the latency is 2+64

- 39 Close the Schedule
- 40 Select the Table Tab

You should see a single solution reporting latency and throughput as well as area. Note that the area is just the datapath area, and not the total area estimate. We need to generate RTL to get final numbers:

- 41 Click on the “RTL” icon in the Task window pane

The Area will change, and a slack will be reported.

- 42 In the Project files list, expand the Output Files => Reports and double click on the RTL report
- 43 Click the + to expand the Bill of Materials section
- 44 Scroll down and find the “mgc\_mul(8,1,16,1,24,4)” element

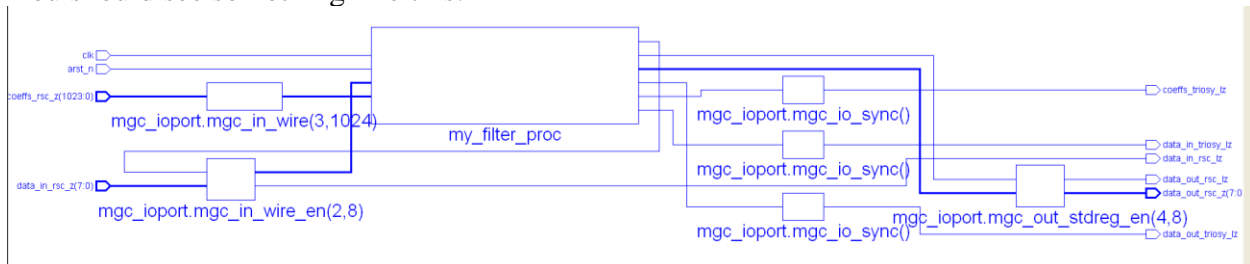
You will see that only one is used both in Post Allocation (Scheduling) and Post Assign (RTL Generation)

- 45 Scroll down to the Timing Report
- 46 Expand the Timing Report and Critical Path report.

Here you see the estimated worst-case critical path that Catapult has computed. Because of the way Catapult builds the data path, this is usually positive slack, giving us a high confidence of meeting timing in RTL synthesis and Place & Route.

- 47 Close the RTL report
- 48 Open the Schematics folder
- 49 Double Click on the RTL schematic to open it.
- 50 As with the Schedule, maximize the window and click on “Zoom to Fit” icon in the top left hand corner

You should see something like this:



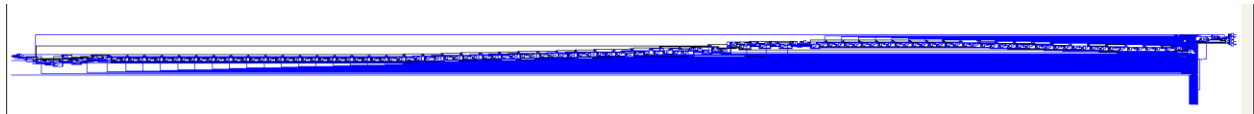
Here we can see the coefficients as a large vector (64\*16 = 1024 bits) along with the 8-bit data\_in, clock, and reset. Outputs are the transaction done flags \*\_lz and the data\_out.

- 51 Double-click on the “my\_filter\_proc” block

This will open up a schematic that shows the RTL implementation of the filter. There will be a lot of multiplexers and registers. The multiplexers and registers form the delay line that deals with the samples. We can see the schematic on one page:

- 52 Using the Right Hand Mouse Button, click on the schematic to bring up a context-sensitive menu
- 53 Uncheck the Multipage Schematics Option

You should see a schematic like this.



This shows the long 64-tap shift register and the computational logic at the far right.

- 54 Using the Left Mouse Button, select the small area on the right to zoom in:



You should now be able to see the muxes, multiplier, adder and rounding/saturation logic along with the output registers:

- 55 Close the Schematic
- 56 Open the VHDL (or Verilog) output folders
- 57 Double click either the “rtl.vhdl” or “rtl.v” file to open the design
- 58 Move all the way to the end of the file and then scroll upwards to find the top level entity or module.

This is machine-generated code, and just like a machine-generated Gate level netlist (VHDL, Verilog, EDIF) you are never expected to edit this file by hand.

You should find the my\_filter interface:

```
ENTITY my_filter IS
  PORT (
    data_in_rsc_z : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    data_in_rsc_lz : OUT STD_LOGIC;
    coeffs_rsc_z : IN STD_LOGIC_VECTOR (1023 DOWNTO 0);
    data_out_rsc_z : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
    data_out_rsc_lz : OUT STD_LOGIC;
    clk : IN STD_LOGIC;
    arst_n : IN STD_LOGIC;
```



```

    data_in_triosy_lz : OUT STD_LOGIC;
    coeffs_triosy_lz : OUT STD_LOGIC;
    data_out_triosy_lz : OUT STD_LOGIC
);
END my_filter;

```

or

```

module my_filter (
    data_in_rsc_z, data_in_rsc_lz, coeffs_rsc_z, data_out_rsc_z,
    data_out_rsc_lz, clk,
    arst_n, data_in_triosy_lz, coeffs_triosy_lz, data_out_triosy_lz
);

```

This looks a little like the interface of the function in C++, just with the clock and reset added, and some flags for system integration.

56 Close the VHDL or Verilog file

## Changing The Clock Frequency

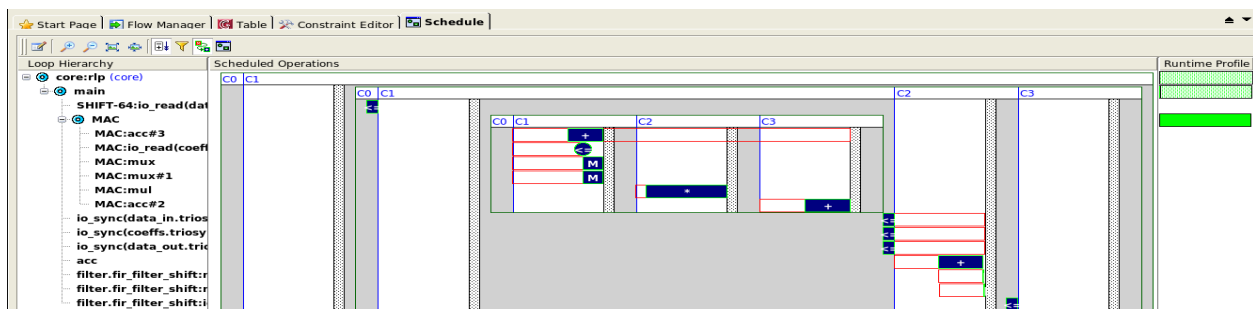
We have synthesized an implementation at 200MHz, but now let's double the frequency to 400 MHz:

- 1 Click on “Setup Design” icon
- 2 Select the “clk” in “Interface Control”
- 3 Change the Frequency to 600MHz
- 4 Note that the Period and High Time should also change
- 5 Click “Apply”

Because we have changed constraints, Catapult will build a new solution

- 6 Click on the “Schedule” icon to create a new schedule
- 7 Expand the Schedule as before (maximize, expand, zoom full)

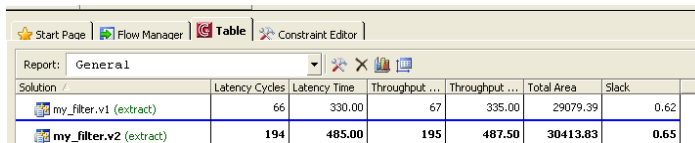
Now we should see a different Schedule from before:



The Scheduler is operating in “Area” mode by default, so the higher frequency requires that the data path be spread over three cycles in order to avoid using faster (and larger) components. So now you start to see the value of technology and timing-aware synthesis

- 8 Close the Schedule
- 9 Click on the “RTL” icon.
- 10 Once the tool stops running, select the Table Tab

You should now be seeing two solutions:



Solution	Latency Cycles	Latency Time	Throughput ...	Throughput ...	Total Area	Slack
my_filter.v1 (extract)	66	330.00	67	335.00	29079.39	0.62
my_filter.v2 (extract)	194	485.00	195	487.50	30413.83	0.65

Note how the area has not increased much, but because we have no pipelining on any of the loops, we have a huge difference in Latency and throughput. This is because the new schedule inner MAC loop has three c-steps instead of one, so instead of taking 64 steps, it now takes 192 steps. Pipelining the loop can fix this.

## Pipelining the design

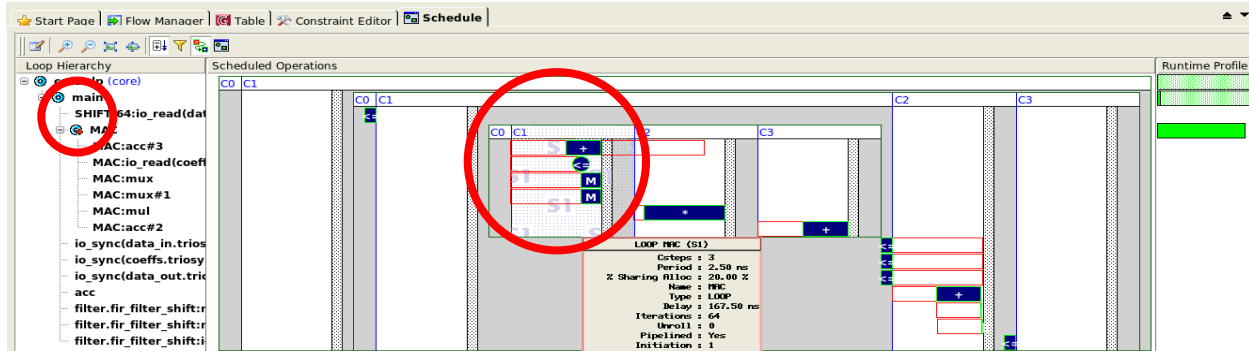
- 1 Click on the “Architecture Constraints” icon
- 2 Select the MAC loop
- 3 Set it to pipeline with an II=1 by checking the Pipeline check box
- 4 Click “Apply” – A new solution will be created
- 5 Click on the “Schedule” icon to create a new schedule
- 6 When the schedule appears, select the Table tab to see the new latency and throughput numbers

You should see that the difference is now just two cycles when compared with our first solution. These two cycles are caused by the two extra cycles in the MAC loop, but no matter how many iterations of the loop we do (thanks to pipelining) the performance hit is small.

- 7 Return to the Schedule and maximize it so you can see the implementation

The implementation is not much different. There are only two differences.

- a) The MAC loop now has an arrow showing pipelining
- b) If you hold the mouse over the white-space of a c-step in the MAC loop, you will see a pipeline stage shading appear (S1, S2, S3 are the pipeline *stages*).



- 8 Close the Schedule
- 9 Click on “RTL” to get a final design
- 10 Check the Table and see the marginal increase in area due to pipelining

As you can see, pipelining can be very effective in improving performance.

As smart engineers, we know that a FIR filter with 64 taps using one multiplier should be able to continuously process one sample every 64 cycles. To improve the throughput, we need to pipeline the design at the top level.

- 11 Click on “Architecture Constraints”
- 12 Select the my\_filter\_main loop and set it to pipeline with II=1
- 13 Click “Apply”
- 14 Generate a new Schedule
- 15 Check the Table Tab
- 16 What is the new Throughput Reported? \_\_\_\_\_ cycles
- 17 Click on the “RTL” icon to create new RTL

You should see that while the slack is now less positive (due to flattening the main loop), the area saved by doing this and having a continuously running data path is significant.

Now let’s see about making our design have faster Latency and Performance

## Improving Latency and Performance

- 1 Click on the “Architecture Constraints” icon
- 2 Select the “core” block
- 3 Change the “Design Goal” to Latency:
- 4 Click “Apply”
- 5 Select the Table Tab
- 6 Click the “RTL” icon and watch the flow through the table view

You should see the latency improve, but the area will get larger (because RTL synthesis will need larger components) and the slack may become more challenging as more operators are forced into fewer clock cycles.

While we have improved latency, we have not improved the throughput as there is still only one multiplier being used. Area optimization for throughput-oriented designs is therefore desirable, as a few cycles of initial latency does not often mean much.

- 7 Open the RTL report and check the bill of materials
- 8 What speed multiplier is being used? \_\_\_\_\_
- 9 Close the RTL report

In order to get more throughput, we need more parallelism. This can be achieved by partially or fully unrolling the MAC loop.

- 10 Click on Architectural constraints
- 11 Select the MAC loop
- 12 Check the “Unroll” check box
- 13 Check the “Partial” check box
- 14 Re-check the “Pipeline” check box
- 15 Click “Apply”
- 16 Click on the “Schedule” icon
- 17 When the schedule appears, maximize and expand it as before, and use the scroll bar at the right hand side to move all the way to the bottom of the Schedule

You should see two multipliers that feed to an adder before going to the accumulator and later rounding and saturation.

Doubling the number of multipliers means that the individual multiplier results must be added together before accumulating them, so even though we are operating in latency mode, this extra adder in the data path means that the technology can now no longer implement the filter in one clock cycle, so another cycle is needed. This means catapult can still optimize the area by picking slower multipliers to take most of one clock cycle, and have the adder, accumulator, and rounding logic in another C-step.

This is more area efficient than trying to squeeze everything into the first C-Step and then having the accumulator and rounding in the second C-Step.

High Level Synthesis can often make smart choices globally that often surpass writing the design in RTL.

- 18 Close the Schedule and generate the RTL
- 19 Look at the Table tab and compare with the previous solution
- 20 Is the area larger or smaller? \_\_\_\_\_

So now, by a strange coincidence of micro-architecture, frequency and target ASIC technology capability, we have a system with double the performance, but less area!

- 21 Click on the “Architecture Constraints” icon
- 22 Select “core” block

- 23 Change the design goal to “area”
- 24 Click “Apply”
- 25 Click on the “RTL” icon to generate new RTL
- 26 Check the Table view

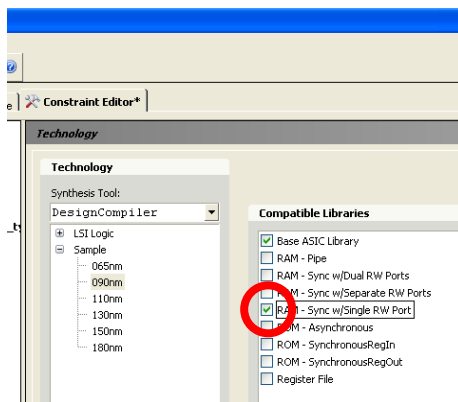
Throughput remains unchanged, but latency increases due to the pipeline being spread across more cycles. Adding more c-steps increases registers and the FSM control logic (which increases area) but relaxes the need for higher speed (higher area) operators, so with smart scheduling, Catapult can reduce the area by making the tradeoff of more registers against smaller operators.

Again, something that is hard to do when writing RTL manually.

## Introducing RAM’s to the design

Thus far, we have only used wires for our inputs and registers for our outputs. Let’s see how we can use a RAM interface for our 16-bit coefficients rather than 1024 wires.

- 1 Click on the “Setup Design” icon
- 2 Enable Single R/W Port RAMs



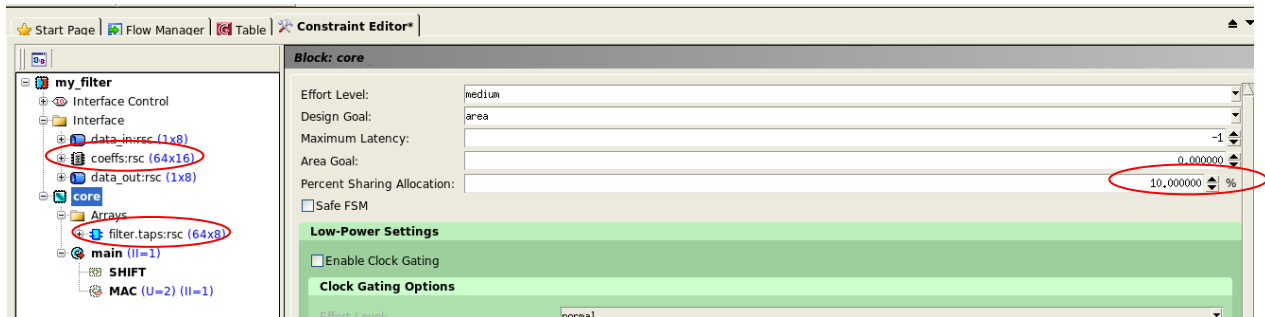
- 3 Click “Apply” – a new solution will be created
- 4 Click on the “Architecture Constraints” icon
- 5 Open the Ports Folder and select the “Coeffs” resource

Note that the resource has now defaulted to a RAM.

- 6 Set the resource type to a “ram\_sample-065nm-singleport\_beh\_dc.RAM”

This will explicitly constrain the I/O to a RAM interface

- 7 Select “core” and change the “Percentage Sharing Allocation” to “10”.
- 8 Reset the clock frequency to 400MHz.



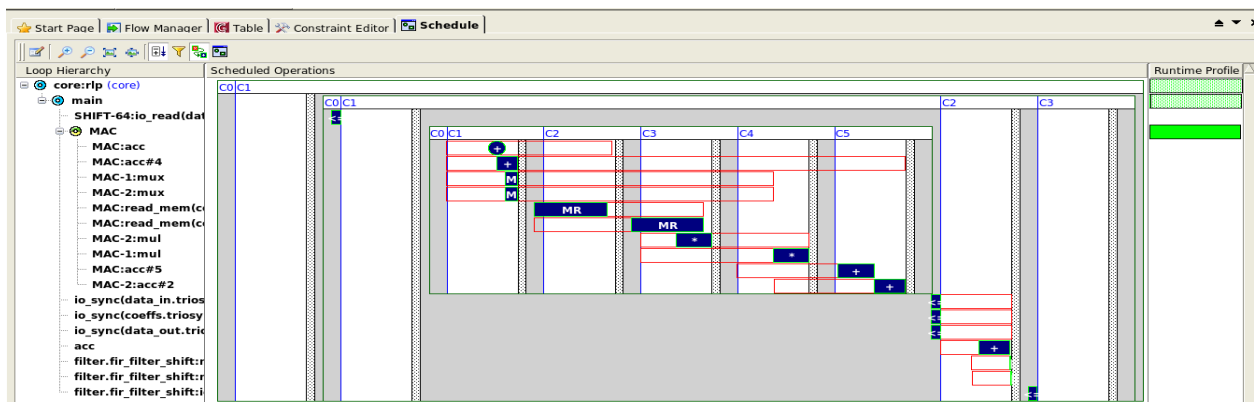
- 9 Click “Apply”
- 10 Click the “Schedule” icon

Scheduling will fail, giving a message in the transcript that there are “insufficient resources” relating to a RAM, where 2 are needed, but only 1 is available.

The reason for this failure is that while we have partially unrolled the MAC loop by two, to give us two multipliers in parallel, pipelined with  $II=1$ , that would require being able to read two coefficients at the same time. Which with one 16-bit wide single port RAM is impossible. In order to debug scheduling problems, it is usual to turn off all pipelining associated with the design.

- 11 Click on Architectural constraints and
  - a. Remove pipelining from the main loop
  - b. Remove pipelining from the MAC loop if necessary
- 12 Schedule the design
- 13 Maximize and expand the schedule to examine the MAC loop

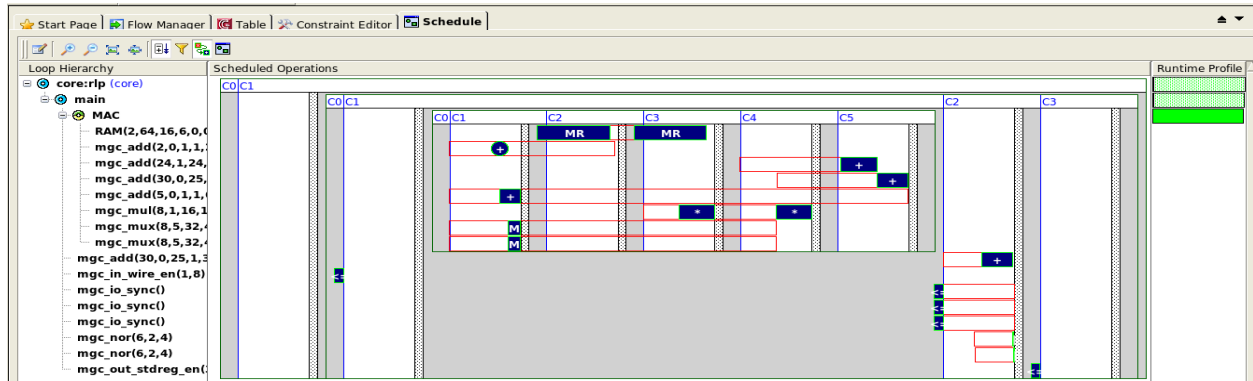
You should see two memory reads being displayed with their associated component  $clk2q$  delay:



We can see that components in the schedule show a diagonal flow of data, indicating a bandwidth limitation.

- 14 Using the Right Hand Mouse button, select “View Sorted => By Component Utilization”

The Schedule will change to show which components are used in which cycle:



We can now see the memory bandwidth limitation. Because of this bandwidth limitation, Catapult will not use two multipliers, because to do so would be wasteful. We could pipeline here with an  $II=2$ , but that would be inefficient as it would still only give us a throughput of 64, and would have larger area due to the sharing of the RAM and multipliers, caused by additional C-Steps in the schedule.

There are a number of solutions to improving the bandwidth at the interface:

- a) Increase the word width of the RAM if accesses addresses are linear and contiguous
- b) Use a Dual Port RAM if the accesses are truly random and two values are required. DPRAM is expensive in ASIC's, but in FPGA technologies the trade-off may be nothing except a little address generation logic.
- c) Use Blocking, or Interleaving, depending on the access patterns in order to create two RAM interfaces of half the size.

Let's try the simplest approach first, and use doubling the word width

- 15 Close the Schedule
- 16 Click on "Architectural Constraints"
- 17 Expand the Ports folder and select the coeffs variable under the coeffs\_rsc
- 18 Set the word width to 32 – you can either type 32 in the box, or push the up arrow to the side of it once.
- 19 Click "Apply"
- 20 Click "Schedule"
- 21 Examine the Schedule. There should now only be one memory access.
- 22 Examine the new schedule

You should see a single memory access along with two multipliers now in parallel:





- 33      Generate new RTL
- 34      Check the Table

The latency and throughput should be similar to the previous solution

- 35      Check the Schematic

You will now see that there are two 16-bit data bus ports for the two interleaved memories on the interface

- 36      Close the Schematic

If you want to, take a look at the VHDL or Verilog RTL files to see the interfacing for the coefficient interleaved resources at the top level “my\_filter” design.

## Getting even more performance

We can extend what we have learned to create a solution with 4,8,16,32, or even 64 multipliers, however at some point, you will want (or need) to switch back to wires for the coefficients, rather than RAM's. Let's look at creating a 64-multiplier implementation:

- 1      Set the following Architectural Constraints:
  - a.    Set coeffs\_rsc interleaving back to '1'
  - b.    Set the coeffs\_rsc resource type to mgc\_ioport.mgc\_in\_wire
- 2      Make sure that the Design Goal is Latency.
- 3      Click “Apply”
- 4      Create a new Schedule (run time will be a bit longer than before)
- 5      Examine the Schedule

You should see a very obvious set of parallel multipliers, followed by a number of c-steps implementing the adder tree for accumulation (assuming you disabled CSA optimization at the start of the lab as directed)

- 6      For completeness, generate RTL
- 7      Note from the Table that the area is much larger, but the throughput is now about 1 sample every clock cycle.

## Changing to the folded filter

In Lab 5, there was the opportunity to create a folded filter architecture. Let's use that to see what happens:

- 1 Edit the FIR\_Fixed.cpp file to call the “fir\_filter\_shift\_fold\_even” member function instead of the “fir\_filter\_shift”
- 2 Save the file
- 3 Catapult will recognize that the file has changed, and ask you if you want to branch a new solution. Click “Yes”
- 4 Click on “Architecture Constraints”

You should now see that some constraints carry over from the previous implementation, but there is a new LOOP called “FOLD” that has 32 iterations

Note also that the MAC loop only has 32 iterations

- 5 Do the following in the Architecture Constraints
  - a. Set the Coefficients Resource to use a single port RAM
  - b. Set the fold resource in the arrays folder to be registers
  - c. Remove the unrolling on the MAC loop
- 6 Click “Apply”

Now that the MAC loop is left rolled, and the coefficients are mapped to RAM again, we should only have one multiplier. However, there is a microarchitectural choice to be made.

- a) Leave the FOLD loop rolled and have it merged into the MAC loop such that one folding adder is shared as part of the iterative multiply-accumulate process.
- b) Unroll the FOLD loop such that there are 32 adders that are then selected using a single multiplexer to feed the single multiply-accumulate looping

The “right” answer depends on technology, frequency, and latency targets. So for our 400MHz, let’s try both:

- 7 Click “Schedule”

You may miss the message in the transcript:

**# Loop 'MAC' is merged and folded into Loop 'FOLD' (LOOP-9)**

If you examine the schedule, you will see that the multiplier is fed by a memory read (coefficients) and a single Adder. This shows that the folding adder was shared.

- 8 Generate RTL to get an area score
- 9 Go to Architectural constraints and fully unroll the FOLD loop
- 10 Apply the constraints
- 11 Generate a new Schedule

You should see a nice parallel set of many Adders as the folding is now done in parallel

- 12 Generate new RTL
- 13 Examine the Table view to compare the last two solutions

You should see that while they both have the same Latency and Throughput, the area is rather different as the last solution (with the folding adders unrolled) requires more logic.

This concludes this lab. As you can see, as single, simple, piece of C++ can generate many different micro-architectural and interface solutions, and with architectural constraints and scheduling, you can find an optimal solution for implementing your particular algorithm.

ENDS