Catapult C Lab 5

Interactive and Iterative Analysis and Optimization Process

Lab Goals

Given a C++ fixed-point implementation of the DCT, the target is to find and generate an RTL architecture matching the requirements below:

Target Technology Xilinx VirtexE v2000efg680
- Clock Speed 50 MHz
- Latency 11 us or less
- Area 3700 LUTs or less
- I/O mapped to Single port RAMs

Source code for the DCT design can be found in:

```
<Installation Directory>/Mgc_home/shared/examples/catapult/dct
```

In order to achieve these results, the source code should not be modified and only loop and memory optimization options should be used.

You should converge to the above requirements by methodically studying the previous solution results using the built-in analysis tools (Gantt chart, mostly) and apply the next constraints.

Lab 5 – Solution

Adding the input file(s)

- Click on “Add Input Files” in the Design bar, or double click on the Input Files folder in the design browser.
- Select the “DCT.cpp” and click OK.
- Now you’ll see DCT.cpp in the input file list in the design browser.

Setup the Design

- Click on the Setup Design button in the design bar
- Open the technology list for Xilinx
- Select the Virtex-E technology
- Then select the v2000efg680 part from the part menu, speed grade -8
- A list of Compatible libraries is listed. This includes the base FPGA library and other IP blocks, such as RAMs. Select the Single Port RAMs
- Now set the clock frequency to 50 MHz.
- Click OK.

After you add the input file and set the technology constraints, the rest of the design flow becomes available.

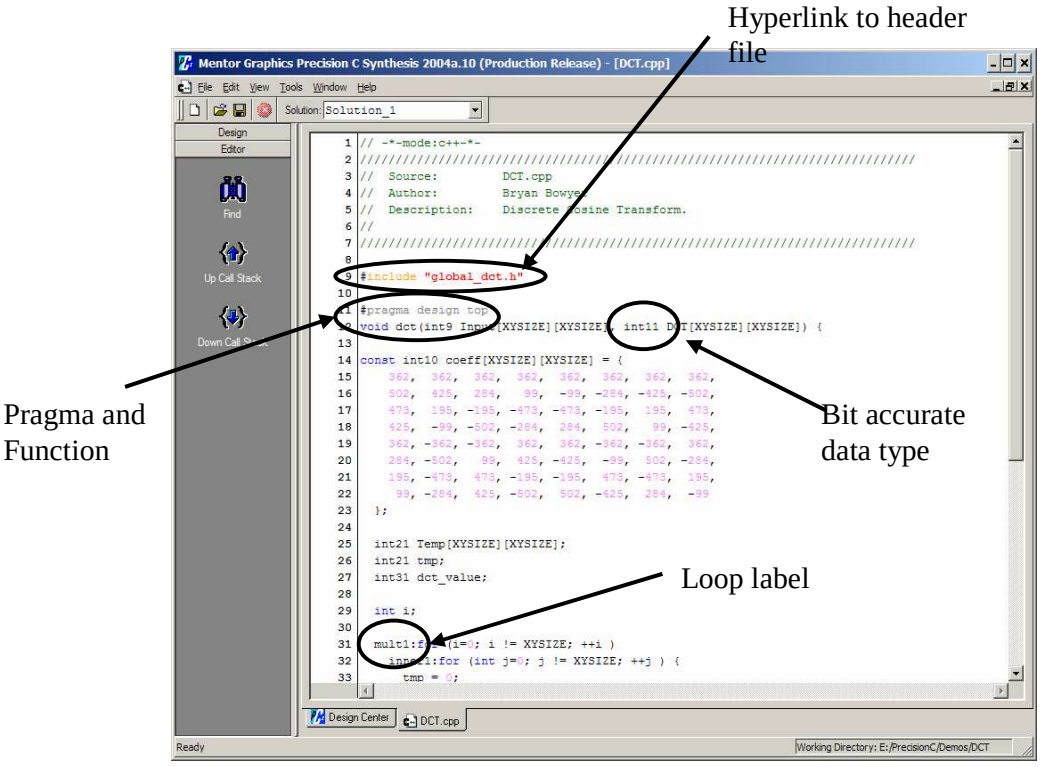
RTL generation and the Multi-threaded GUI

- Click on the Generate RTL button at the bottom of the Design Bar

Once a technology, clock speed and source file have been selected, the only thing you need to do to create hardware is to click on the Generate RTL button and Catapult C will use it’s defaults to generate the RTL.

- While the tool is running, double click on “DCT.cpp” in the input file list and look at the code

The GUI runs in a different thread than the core algorithms, so it’s possible to edit source files or to look at graphical reports while the core engine is running.



Catapult C has an integrated file editor to work on C code. The C code is just a standard C function call, only the "#pragma design top" needs to be added.

The labels on the loops and the names of variables are used throughout the tool, giving an easy way to see where something came from.

The "mc_bitvector.h" file includes our bit-accurate types. The include for this line can be found in the "global_dct.h" file. Bit accurate types can be seen on the design interface, these types simulate and synthesize bit accurate.

Result Analysis

Close the text window. You'll now see many files in the output file list in the design browser. These files are reports, RTL and command files for downstream tools. We'll cover these in detail after we find a design that meets our design goals.

- Click on the "XY-Plot" tab next to the "Project Files" tab, you'll now see an XY-Plot that shows the latency and area for the first solution we've created.
- Click on the "Bar Chart" tab to show the breakdown of the area.
- Click on the "Table" tab to show a summary of the various design metrics.

Obviously this solution is far from meeting latency requirements (54 us obtained vs 11 us required).

We now need to investigate how the architecture can be improved to narrow down on the requirements. And as new solutions will be generated, the graphical reports will be updated with the new solutions results.

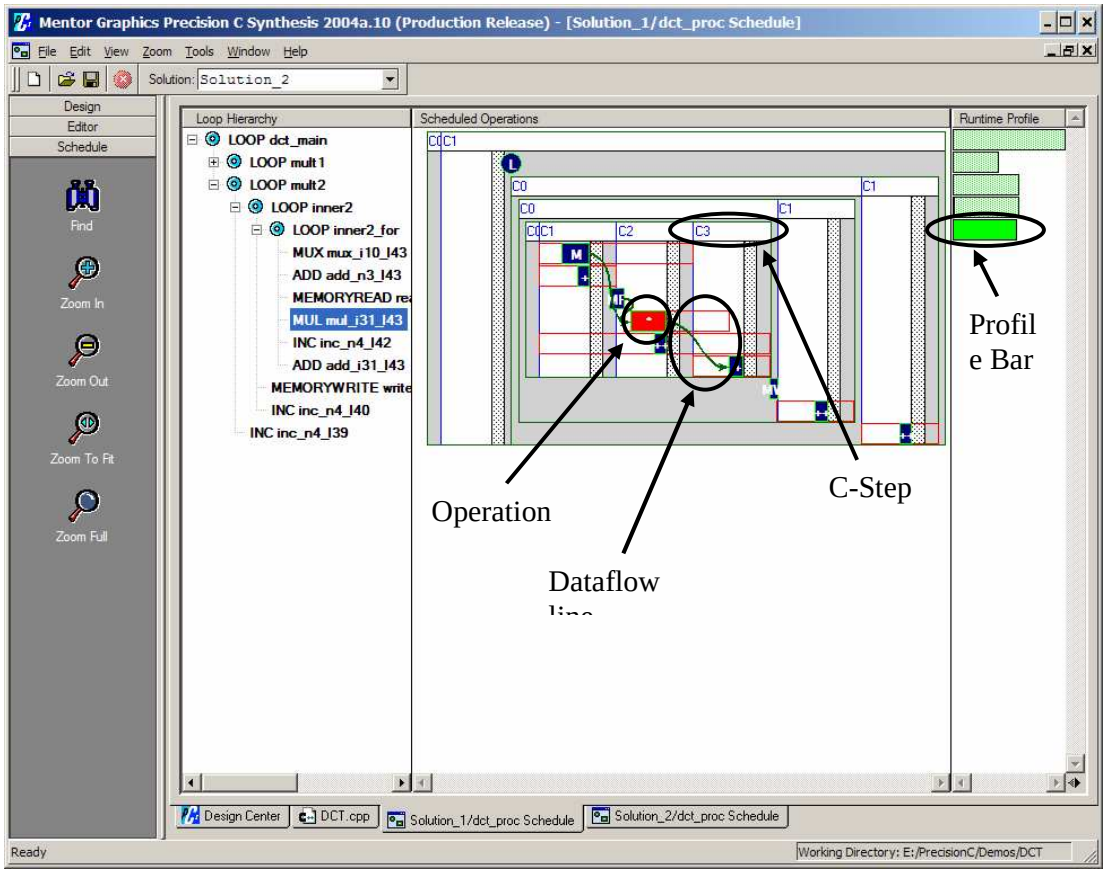
Design Analysis – The Gantt chart

The Gantt chart is the most powerful analysis tool in Catapult C. It helps to quickly find out design bottlenecks and appropriate optimization methods.

- In the Design Bar, click on the "Schedule" button. This will open the Gantt chart for analysis

On the left side of the Gantt chart, the design structure and loop hierarchy is represented. Notice how you can cross-probe from the Gantt chart to the source code by double clicking on a loop name.

- Expand the "mult2" loop until you reach the inner most loop.



On the right side of the Gantt chart is the "Runtime Profile". It gives a quick way to find the critical parts of your design. The length of the bar is the percentage of the time your design spends inside a loop. The light green bars show that a sub loop is where the time is spent and a dark green bar means that you've found the right loop.

- Move the Mouse over the dark green bar corresponding to the “inner2_for” loop. This will give you additional information about the loop execution profile.

In the middle and main section of the Gantt chart is the actual representation of the design schedule. The columns represent “C-Steps”. C-Steps correspond to one clock cycle and are equivalent to states in a state machine. This allows seeing when (in which C-Step) each operation occurs. The length of each operation is to scale, so the multiplier takes longer than the adder operation.

- Each operation can be clicked on to show the dataflow through that operation. Click on the multiply operation to show the dataflow
- You can also cross probe from the operations to the original source code. Double click on the multiplier name (on the left) and notice where this operation comes from.

Now try to answer these simple questions about the “inner2_for” loop:

- *How many C-Steps (clock cycles) are required to execute one iteration of the loop?*
- *How many multipliers are needed for one loop iteration?*
- *How many times does the loop iterate?*
- *What is the total latency (number of clock cycles) required to fully execute the loop?*
- *What is the total latency (number of clock cycles) required to fully execute the “mult2” loop?*

Now try to answer the same questions for the “inner1_for” loop under the “mult1” loop hierarchy.

The answers to these questions should hint you that this loop is a good candidate for unrolling because it has a small number of operations, it has a small number of iterations, and the dataflow is all from left to right, so there is no data feedback in the loop.

Loop Unrolling

We’ll now create a new solution with the loops unrolled to see what the results are. To unroll the loop we analyzed, simply open the “Architectural Constraints” window and set the unroll constraints as explained below.

- Select “Architectural Constraints” from the Design Bar
- Select the “inner1_for” and “inner2_for” loops and check the unroll flag. You should see the icons change for both loops
- Click OK

The GUI issues a directive command to the core in order to unroll the loops. This causes Catapult C to automatically duplicate the solution. All of the constraints from Solution 1 are copied to Solution 2 and then the new directive is applied.

- Click on the “Generate RTL” button in the Design Bar

Unrolling loops causes the biggest design changes of any of the architectural constraints. This can be seen in the XY-Plot.

Let’s now look at the Gantt chart to understand the reasons for such a change in the architecture.

- Open the Gantt chart by clicking the “Schedule” button in the Design Bar
- Expand the “mult2” loop hierarchy. The “inner2_for” loop has disappeared. Indeed, when fully unrolled, the loop body is copied as many times as the loop iterates and the original loop is dissolved. Here, the entire sequence of operations appearing under the “inner2” loop corresponds to the full execution of the “inner2_for” loop.

Using the Gantt chart, try to find the answer the following questions, and compare them to the first solution:

- *How many cycles are required to execute the unrolled “inner2_for” loop?*
- *How many cycles were needed to execute the rolled “inner2_for” loop?*
- *How many cycles are required to execute the “mult2” loop with “inner2_for” unrolled?*
- *How many cycles were required to execute the “mult2” loop with “inner2_for” rolled?*

When loops are unrolled, Catapult C can extract parallelism out of the design. When unrolled there is no more “time boundary” between one iteration and next one. Independent operations of different iterations can now happen in the same cycle. As a result, the loop will run much faster.

Unrolling loops and speeding a design through parallelization can sometimes be done at the expense of area. Let’s now look at the area penalty of unrolling the loops.

- Click on the “Bar Chart” tab to show the breakdown of the area.

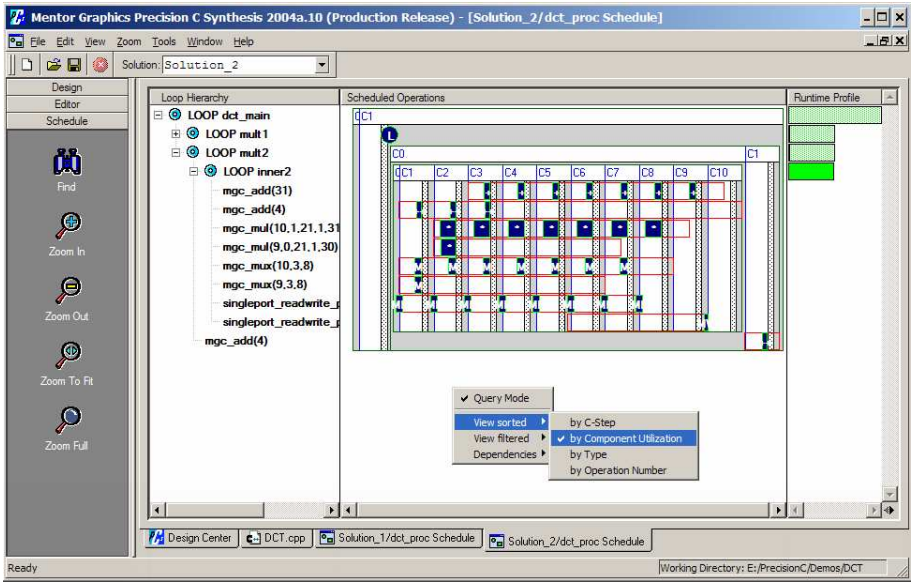
While the latency decreased by more than 60%, the area has doubled. But as the Bar Chart reports, most of this area increase is due to muxes. When loops are unrolled, Catapult C will try to share arithmetic components so that many different C operations can be computed by the same resource, but at different points in time (C-steps). This keeps the number of arithmetic resources down, but causes additional mux logic to feed the right operands on the components.

We’ll now look in more details at how Catapult C dealt with the multipliers when the loops where unrolled.

Component Utilization

We will now use the Gantt chart in a different mode, called “Component Utilization” view.

- Right click in the Gantt chart and select “View Sorted -> By Component Utilization”.



The previous view – called "C-Step view" – showed when each of the operations defined in the C code were executed. In contrast, the "Component Utilization view" shows when each of the allocated hardware resource is being used.

This is an important difference as many operations of the same kind in the C code can be mapped to the same hardware component, which will get shared in a time multiplexed fashion.

- *From the "Component Utilization view", find out how many multipliers are needed to execute the unrolled "inner2_for" loop.*
- *Compare that to the number of multipliers required to execute the rolled "inner2_for" loop in the previous solution.*

As our design doesn't meet the latency constraint yet, we are continuing to search for areas were the architecture can be optimized.

Looking at the Gantt chart for the current solution (with unrolled loops), we can see in the "Component Utilization" view that our design is bound by the memory accesses to the "Temp" local memory.

Indeed, the Gantt chart shows that the "inner2" loop is accessing the single port memory on the first 8 cycles of each iteration. We could assume that if the design could retrieve more than one sample per cycle, then some of the processing could be scheduled sooner, therefore shortening the loop latency.

Here, we are physically constrained to one access per cycle since we have mapped the "Temp" array to a single port memory.

Memory Mapping

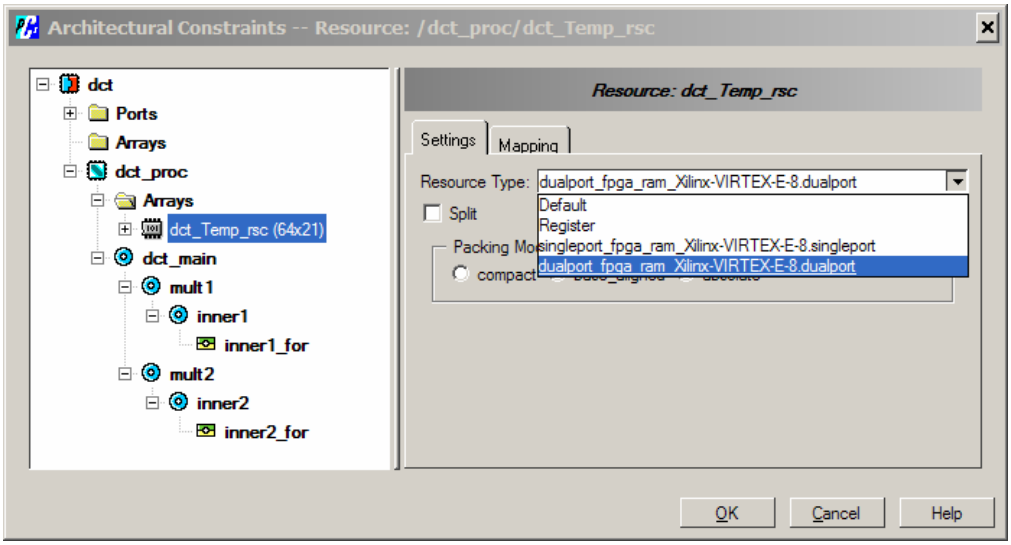
Since we want to further optimize the "inner2" loop, we will map the "Temp" array to a dual port memory in order to read up to two samples per cycle.

First we need to make dual port memories available:

- Click on “Setup Design” in the Design Bar.
- In the Setup Design window, check the check box next to the Dualport RAM
- Click OK. This will enable dualport RAMs and branch a 3rd solution.

Then we need to map the array defined by the code to a dual port RAM:

- Click on “Architectural Constraints” in the Design Bar. This will bring up the constraints window and allow you to set the memory constraints on the design.
- In the Architectural Constraints window, open the Arrays folder inside of “dct_proc”.
- Select “dct_Temp_rsc” and change the Resource Type to ram_dualport_fpga_ram…



- Click OK to apply the changes.
- Click on the “Generate RTL” button in the Design Bar

The internal array results in RTL code that will map to a block RAM through Precision RTL. Once the RTL code is generated, look at the XYPlot and compare the various solutions.

Using the Gantt chart in the “Component Utilization” view, try to find the answers to the following questions:

- *Latency: how many cycles are needed to execute an iteration of “inner2”? How does this compare with the previous solution?*
- *Area: how many multipliers are needed to implement “inner2”? How does this compare with previous solution?*
- *Memory Accesses: how many C-steps require accesses to the dual port RAM? How does this compare with previous solution?*

Now look at the execution profile of loop “mult1/inner1”. Similarly to “inner2” before using a dual port RAM, “inner1” is bound by memory accesses.

Here also we could map the “Input” array to a dual port memory in order to retrieve two samples per cycle (instead of one) and shorten the overall loop latency. But this would violate one of our initial requirements: the design should communicate through single port memories only... So we’ll have to leave this memory as single port and use other optimization methods to get the “mult1” loop latency down.

We are continuing to search for ways to optimize the architecture to reach our design goals.

Looking at the Gantt chart we can see that the “inner1” and “inner2” loops involve a lot of operations. This means they are not good candidates for unrolling and pipelining now becomes an option.

Switching to the Gantt chart component utilization view, we can see when the different components are used. This helps highlighting the potential resource bottlenecks and allows for analysis of how much to pipeline a loop.

The component view shows that in the “inner2” loop, the two ports on the RAM are used for four cycles. This means that we can pipeline with an initiation interval of four and get a good performance improvement without adding any more RAM ports or using more multipliers.

Loop Pipelining

Pipelining is a way to improve the design performance by forcing to start a loop iteration before the previous iteration finishes. Manually pipelining a design can be a very complicated task, taking a large percentage of the design time. In Catapult C, it requires a few clicks only, and the tool takes care of all the details, building correct and optimized hardware.

- Click on the Architectural Constraints in the Design Bar
- Click on the “mult2” loop and check the “Pipeline” box. This will enable pipelining.
- Set the initiation interval to 4 to match what we saw in the Gantt Chart.
- Click OK
- Click on Generate RTL in the design bar.

- Using the various reporting tools (XY Plot, Bar Chart, Table view) compare and contrast the changes in performance and area

- *Based on the Gantt chart component utilization view, what is the minimum Initiation Interval possible for pipelining “mult1/inner1”?*
- *Set the pipeline constraints accordingly and generate a new RTL*
- *Are we meeting design goals yet?*

Although we are converging to the required latency time, we still need further optimization to meet our goals. The Gantt chart profiler clearly indicates that the "mult1" loop is where most time is spent in the design. This is where we will have probably most margin for improvement.

As discussed in Part 3, the "mult1" loop is I/O bound. Each iteration requires 8 input samples to be read from a single port memory – therefore the design needs to access the memory on the first 8 cycles of each iteration, preventing efficient pipelining.

One solution for speeding this loop is to retrieve more samples per clock cycle. But as already mentioned, using a dual port memory is not an option. In this case, other "Interface Synthesis" features can help us achieve our goals.

Interface Synthesis

Interface Synthesis lets the user define the protocol for each I/O port.

- Open the "Architectural Constraints" from the Design Bar.
- In the Architectural Constraints window, open the Ports folder

Clock and Reset are automatically added to the design. Enable, Start and Done are optional control and synchronization signals (off by default).

The "Input" variable was identified as an array of 64 (8x8) 9-bit elements. The direction was identified as an input and therefore the "Input" variable was mapped to an input resource called "Input_rsc". By default the "type" of the "Input_rsc" input resource is set to single port RAM.

To change the protocol associated to a port, select the desired "Resource Type" in right pane of the "Architectural Constraints" window.

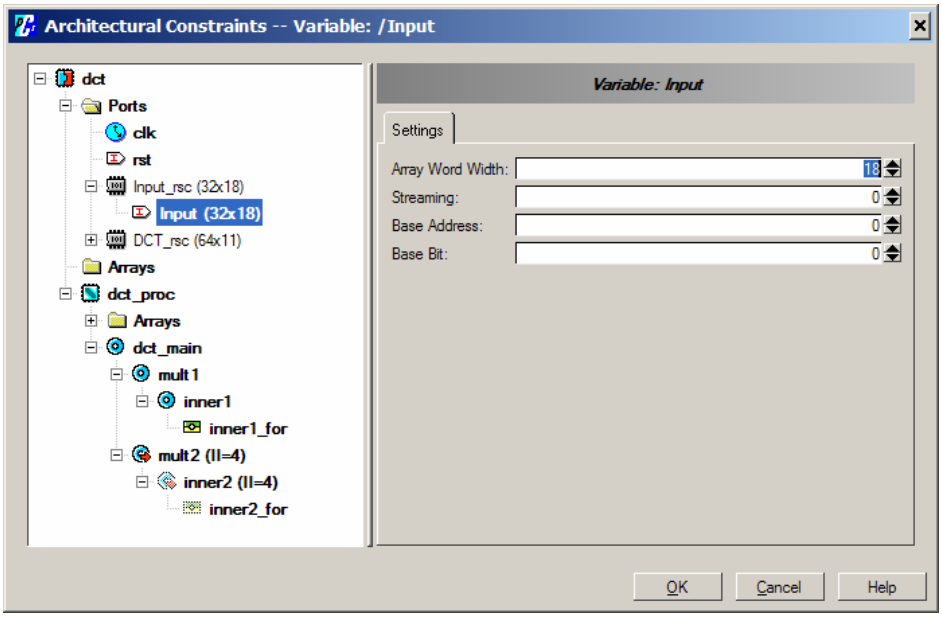
For instance, mapping "Input_rsc" to a wire, would mean that all 64 samples would be available at once, over a 64*9=576 bit wide input port.

But in our case, we need both I/O ports to be mapped to single port RAMs.

Memory Word Width

A possibility offered by Catapult C when synthesizing C++ Arrays is to define the word width of the memory. By default, the generated memory will have a word width equivalent to the bit width of the stored data type. In this case, the memory is 9 bit wide, and can store up to 64 (8*8) elements.

- Open the "Architectural Constraints" from the Design Bar.
- In the Architectural Constraints window, open the Ports folder
- Select the "Input" variable folded under the "Input_rsc" resource it is mapped to
- By default, the array word width is set the 9 bits.
- Increase the word width to 18 bits
- Click OK
- Click "Generate RTL" from the Design Bar



By doubling the array word width (from 9 to 18 bits) you allow the design to retrieve two valid samples (2*9 bits=18) per memory access. For instance, a read at address 0, will return samples index 0 and 1 grouped in an 18 bits data word.

Catapult C can now merge consecutive memory accesses therefore requiring much less memory access cycles (potentially twice as less).

In the Gantt chart component utilization view, look at the “mult1” loop:

- *How many memory access cycles does it require?*
- *How many memory access cycles did it require before?*
- *Can pipelining initiation interval be improved?*
- *Try to pipeline the “mult1” loop with the minimum possible initiation interval.*
- *Do we now meet the latency constraint?*

- Click in the Project Files tab to show the list of result files for Catapult C.