Mentor®
A Siemens Business

Catapult® Untimed C++ Training Labs

Software Version v10.3

September 2018

Table of Contents

Table of Contents

Introduction

Each one of the Catapult labs is designed to review the content covered in the training section leading up to the lab. The labs are designed so that each lab builds on the previous lab and progresses towards more complicated HLS topics. A Discrete Cosine Transform (DCT), a well know algorithm for image and video compression, is used as an example throughout the lab material, with the goal being to build a high-performance multi-block DCT in the final labs.

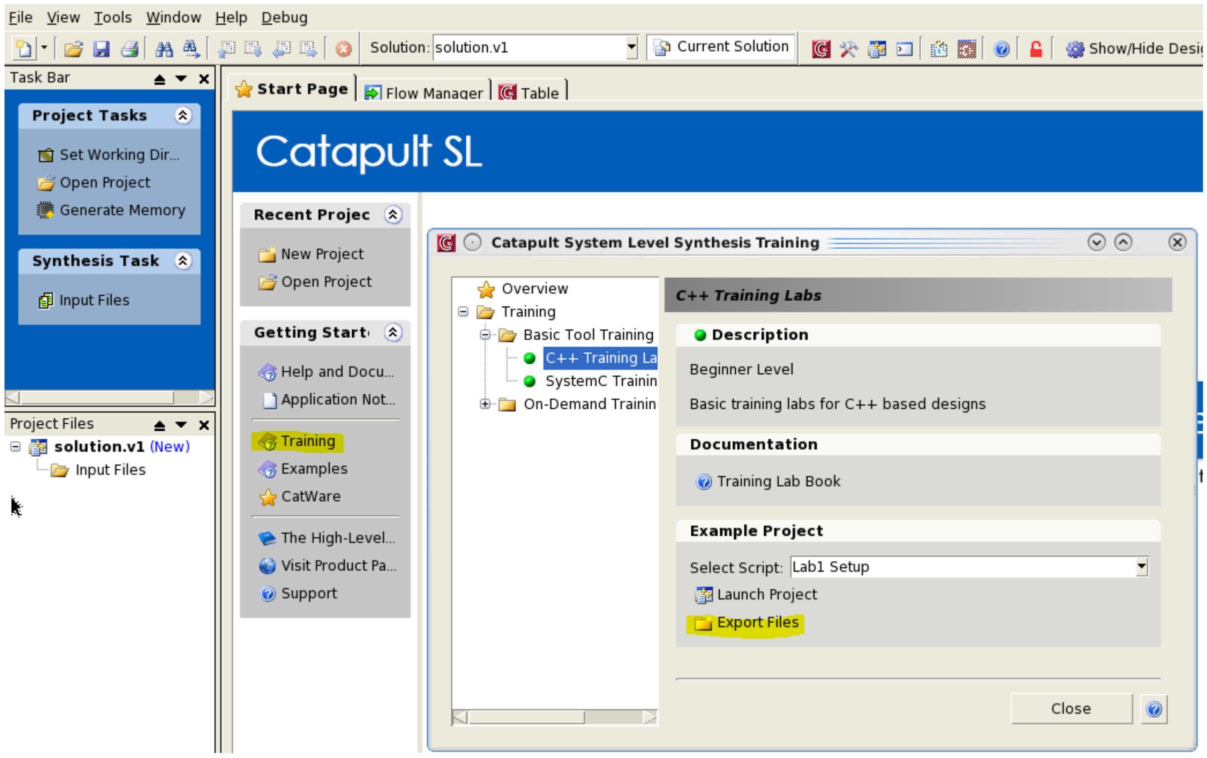
The Training Labs ship with the Catapult install. You will need to copy the labs into your local workspace. To do this:

1. Launch Catapult by typing the following command in a terminal window

   ```
   catapult
   ```

2. Go to the Catapult start page and click on the **Training** icon
3. Expand the **Basic Training** folder and select "**C++**"
4. Click on the **Export Files** folder and select a location to copy the labs.
5. Click **OK**

   This will create a Labs directory with Lab1 through Lab5 sub-directories
6. Change to the *Labs* directory where you exported the training files.



Lab 1 – Generating RTL

In this lab, you will learn the basic flow through Catapult by performing the following steps:

1. Invoke Catapult
2. Read the Design Files
3. Specify Hierarchy
4. Load the Technology Library
5. Specify Timing Constraints
6. Specify Architecture Constraints
7. Generate the RTL Netlist
8. Examine the Results, Reports, and RTL

Each of these steps is covered in the remaining subsections.

Invoke Catapult

1. Open a terminal window and cd to the Lab1 directory.

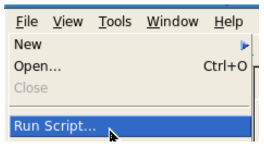
```
cd .../Labs/Lab1
```

2. Type *catapult* at the Linux command prompt.

```
catapult
```

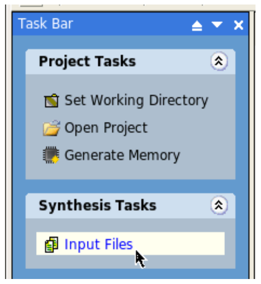
The Catapult GUI should appear.

3. Go to the File Menu and select "Run Script". Select the "setup.tcl" file and click "Open". This setup file disables all hardware verification synchronization signals since verification has not been discussed yet. This will be covered in the next lab.

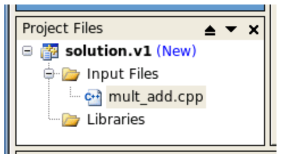


Read the Design Files

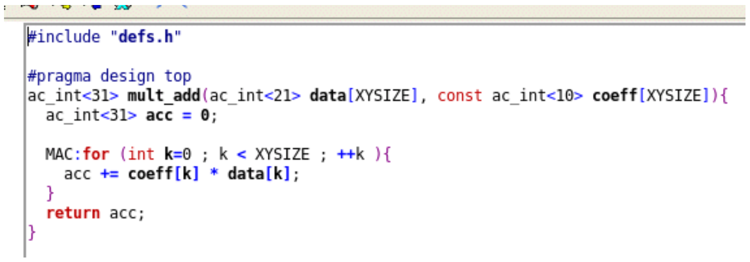
1. Click on **Input Files** in the Catapult task bar.



2. After the File Selection dialog opens, select the *mult_add.cpp* file and click **Open**.
3. Double-click on the *mult_add.cpp* file in the Project Files view to open the file.



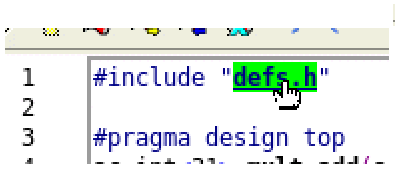
4. The *mult_add* function multiplies and accumulates all *data* and *coeff* array values and returns the result. The *mult_add* algorithm is a core feature to most DSP and video algorithms and as we will see later is also used in the DCT.



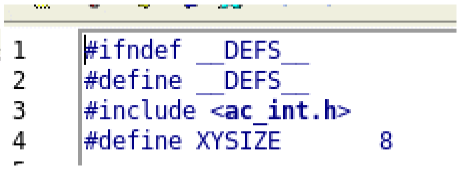
```
#include "defs.h"

#pragma design top
ac_int<31> mult_add(ac_int<21> data[XYSIZE], const ac_int<10> coeff[XYSIZE]){
  ac_int<31> acc = 0;

  MAC:for (int k=0 ; k < XYSIZE ; ++k ){
    acc += coeff[k] * data[k];
  }
  return acc;
}
```

Note: *The bit widths for data (21 bits signed), coeff (10 bits signed), and the function return type (31 bits signed).*

5. Double-click on *defs.h* at the top of the file to cross-probe to the header file. This contains a *#define* for *XYSIZE*. As you can see it is defined to be *8*.



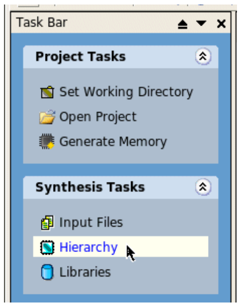
```
#include "defs.h"

#pragma design top
```



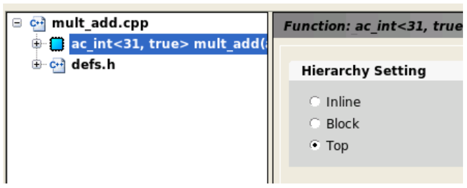
```
#ifndef __DEFS__
#define __DEFS__
#include <ac_int.h>
#define XYSIZE       8
```

Specify Hierarchy

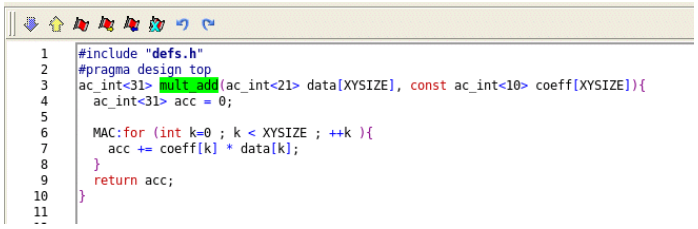
1. Click on **Hierarchy** in the Task Bar.



2. Click to select the *mult_add* function and note that its Hierarchy Setting is set to *Top*. This is also indicated by the blue chip icon with the black edges.



3. Double-click on the *mult_add* function to cross-probe back to the C++ source. You will see that there is a *#pragma design top* before the *mult_add* function. This is why Catapult defaulted to *mult_add* as the top-level design.

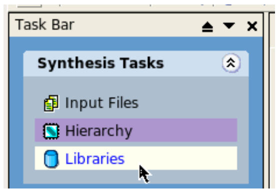


```
#include "defs.h"
#pragma design top
ac_int<31> mult_add(ac_int<21> data[XYSIZE], const ac_int<10> coeff[XYSIZE]){
  ac_int<31> acc = 0;

  MAC:for (int k=0 ; k < XYSIZE ; ++k ){
    acc += coeff[k] * data[k];
  }
  return acc;
}
```

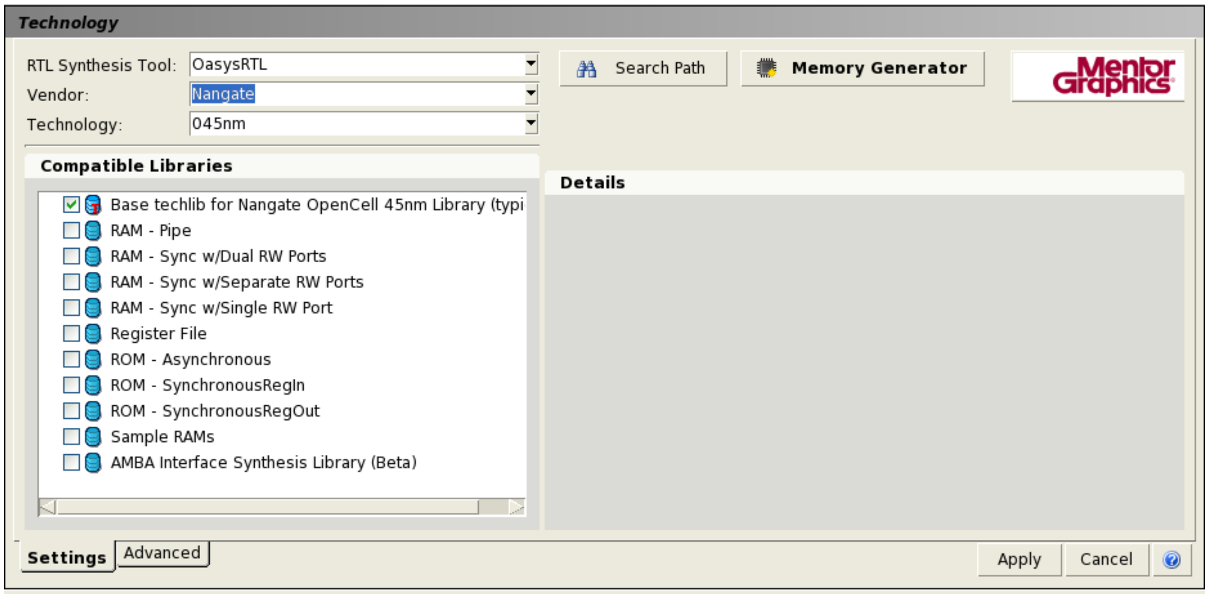
Alternatively this could have been specified in the GUI or as a design constraint.

Load the Technology Library

1. Click on **Libraries** in the task bar.



2. Select the open source *Nangate 45nm* technology by selecting the following values:
   - Set the RTL Synthesis tool to **OasysRTL**
   - Vendor to **Nangate**
   - Technology to **045nm**.



RTL Synthesis Tool: OasysRTL
Vendor: Nangate
Technology: 045nm

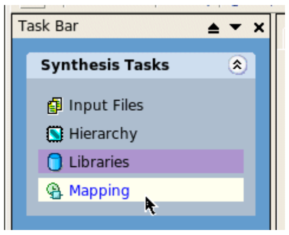
- [✓] Base techlib for Nangate OpenCell 45nm Library (typi
- [ ] RAM - Pipe
- [ ] RAM - Sync w/Dual RW Ports
- [ ] RAM - Sync w/Separate RW Ports
- [ ] RAM - Sync w/Single RW Port
- [ ] Register File
- [ ] ROM - Asynchronous
- [ ] ROM - SynchronousRegIn
- [ ] ROM - SynchronousRegOut
- [ ] Sample RAMs
- [ ] AMBA Interface Synthesis Library (Beta)

***NOTE**: These are sample libraries and should only be used for the labs. You must build your own libraries using Catapult Library Builder to target the technology that you're using. Otherwise there can be no guarantee of closing on timing or achieving good quality of results.*

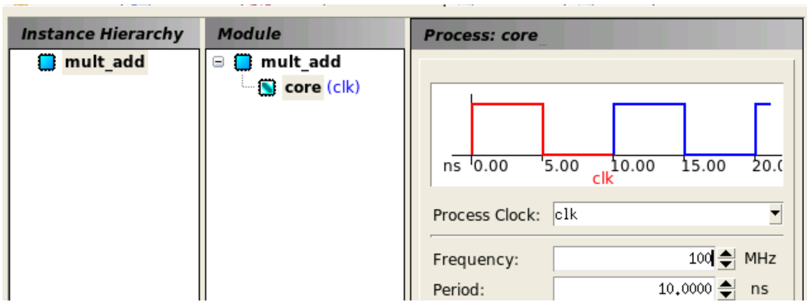
3. Click on the **Apply** button in the Technology view.

Specify Timing Constraints

1. Click on **Mapping** in the Task Bar.



2. Click on the "core" icon and set the **Clock Frequency** to *100 MHz* and click **Apply**.

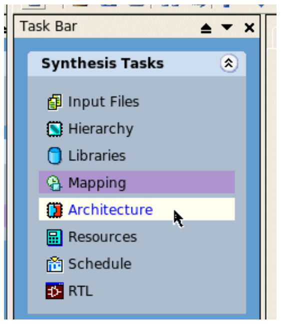


Process Clock: clk
Frequency: 100 MHz
Period: 10.0000 ns

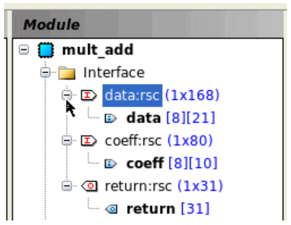
Specify Architecture Constraints

The Architectural Constraints view provides a view of the design interfaces as well as the ability to constraint various things such as interface protocol, loop constraints, optimization mode, etc. For now we are only interested in the display of the design interfaces, port direction, and input_delay constraints.

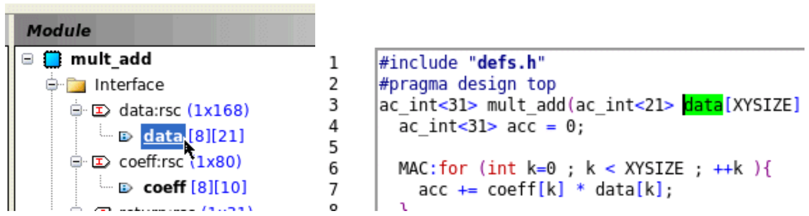
1. Click on **Architecture** in the Task Bar. This task will open the *Architectural Constraints* Editor.



2. Click on the "+" next to the interface folder. Also click on the "+" for each interface resource.

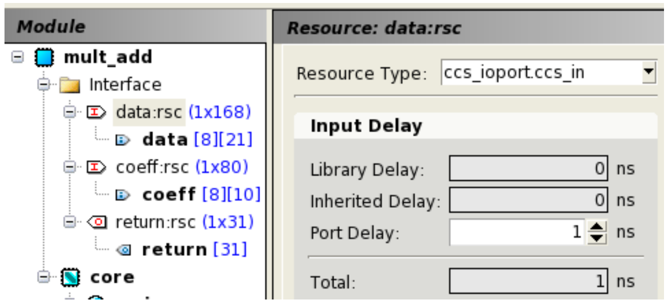


3. Underneath each resource is a variable that corresponds to the *mult_add* function interface variables. This view shows the port direction and bit width. The *data:rsc* resource is 168 bits wide (1x168) and it contains the *data* variable which is an array with 8 elements of 21 bits each ([8][21]). Note that *coeff* variable is also an input and the *return* statement is an output.
4. Double-click on the *data* variable underneath the *data:rsc* resource to cross-probe back to the C++. Examine the C++ code and you can see that *data* and *coeff* are only ever read in the C++ source so Catapult's analysis determines that they are inputs. The *return* value is always an output.



```
#include "defs.h"
#pragma design top
ac_int<31> mult_add(ac_int<21> data[XYSIZE]
  ac_int<31> acc = 0;

  MAC:for (int k=0 ; k < XYSIZE ; ++k ){
    acc += coeff[k] * data[k];
```

5. Go back to the *Architectural Constraints* Editor and select the *data:rsc* resource. Set an Input Delay constraint to 1 ns.



Resource: data:rsc

Resource Type: ccs_ioport.ccs_in

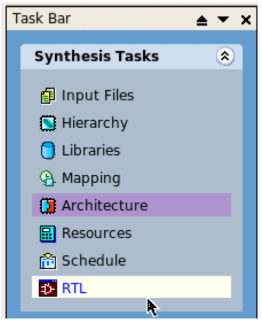
Input Delay

Library Delay: 0 ns
Inherited Delay: 0 ns
Port Delay: 1 ns
Total: 1 ns

6. Go to the **File** Menu and select **Run Script**. Select the *constraints.tcl* file and click **open**. This file is applying constraints for parallelism that we have not covered yet, but will cover in a later section. For now consider it as a constraint to tell Catapult to synthesize the design in a similar fashion to the way Verilog is synthesized so that the *mult_add* design is built in a fully parallel fashion.

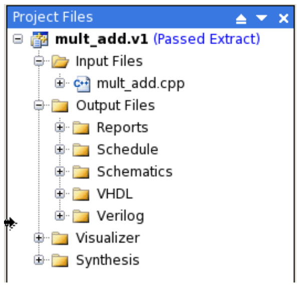
Generate the RTL Netlist

Click on **RTL** in the Task Bar.

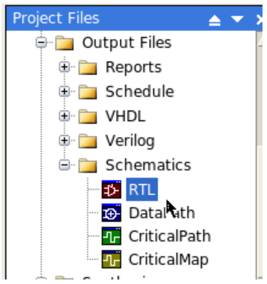


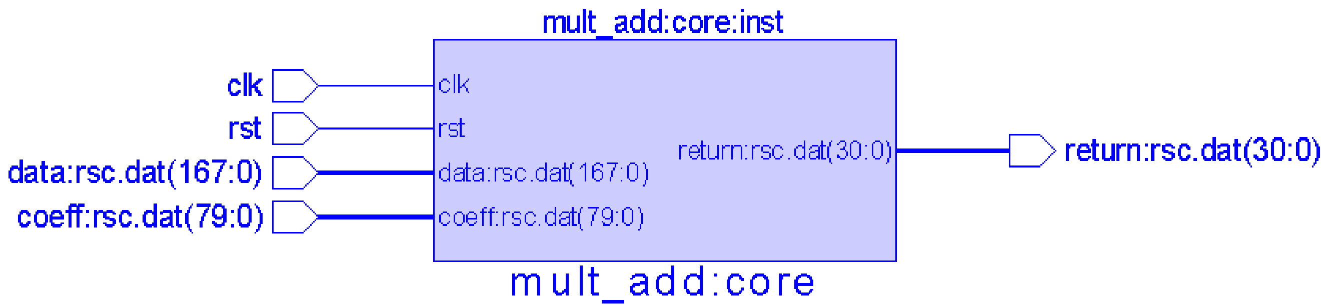
Examine the Results, Reports, and RTL

1. Expand the Output Files folder in the Project Files view by clicking on the "+". This is where Catapult writes all of its output including reports, constraints, schematics, etc.

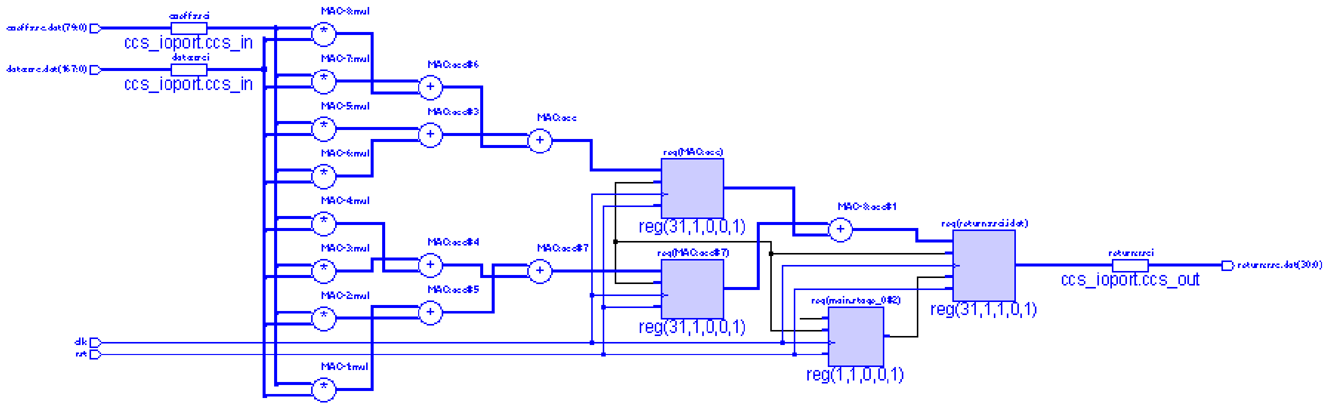


2. Expand the Schematics folder by clicking on the "+". This folder contains the RTL, Data Path, and critical path schematics.
3. Double-click on RTL in the Schematics Folder. You should see the top-level schematic that shows the top-level block and interfaces.

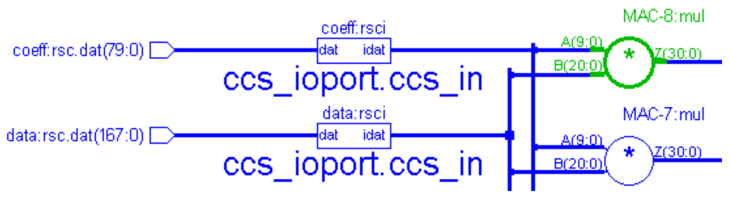


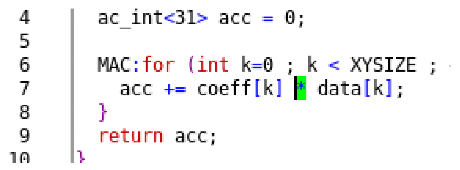


4. Double-click on the *mult_add:core:inst* block to push down into the schematic. You should see 8 multipliers followed by and adder tree. You can see that Catapult has built a fully parallel design that performs all 8 multiplies from the mult_add function in parallel and then sums the results followed by an output flop. This is comparable to what one would get with a similar Verilog implementation. However, as we will demonstrate in subsequent labs, High Level Synthesis provides the user the ability to tune the design architecture to meet different performance goals, all from the same design description.



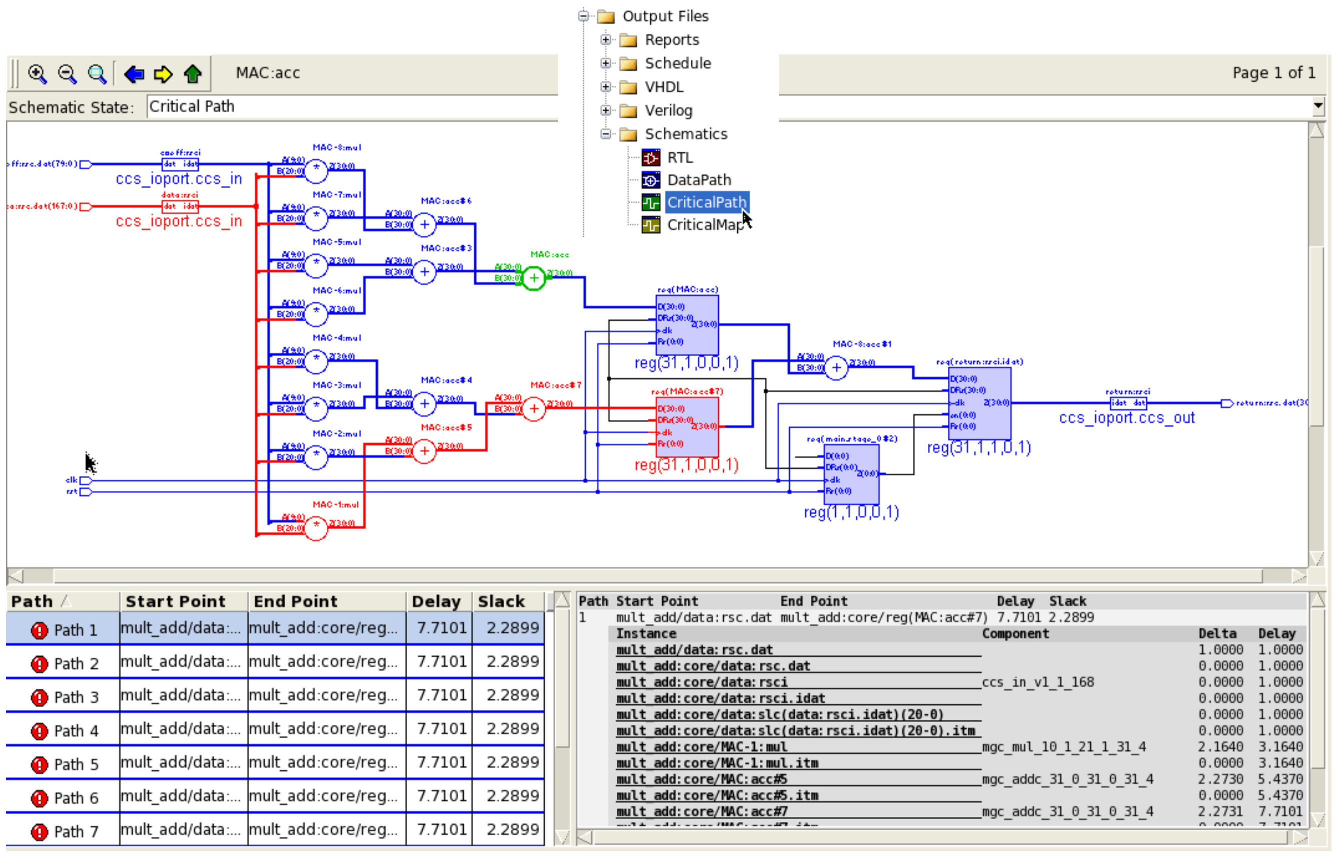
5. Double-click on one of the multipliers in the Schematic and it will cross probed back to the C++ source. Note how the "*" in the C++ has been highlighted.





```
ac_int<31> acc = 0;

MAC:for (int k=0 ; k < XYSIZE ;
  acc += coeff[k] * data[k];
}
return acc;
}
```

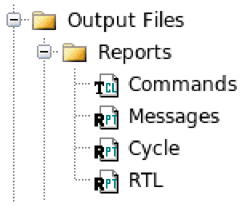
6. Double-click on **Critical Path** in the **Schematics** folder. This will bring up the critical path schematic and shows the 10 worst-case timing paths. You can click on the different paths and it will highlight that path in the schematic.



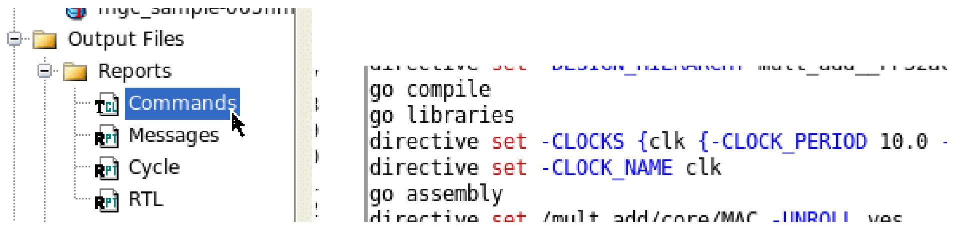
Path	Start Point	End Point	Delay	Slack
Path 1	mult_add/data:...	mult_add:core/reg...	7.7101	2.2899
Path 2	mult_add/data:...	mult_add:core/reg...	7.7101	2.2899
Path 3	mult_add/data:...	mult_add:core/reg...	7.7101	2.2899
Path 4	mult_add/data:...	mult_add:core/reg...	7.7101	2.2899
Path 5	mult_add/data:...	mult_add:core/reg...	7.7101	2.2899
Path 6	mult_add/data:...	mult_add:core/reg...	7.7101	2.2899
Path 7	mult_add/data:...	mult_add:core/reg...	7.7101	2.2899

```
Path Start Point          End Point                      Delay  Slack
1    mult_add/data:rsc.dat mult_add:core/reg(MAC:acc#7) 7.7101 2.2899
     Instance                                              Component                  Delta   Delay
     mult_add/data:rsc.dat                                                            1.0000  1.0000
     mult_add:core/data:rsc.dat                                                       0.0000  1.0000
     mult_add:core/data:rsci                               ccs_in_v1_1_168            0.0000  1.0000
     mult_add:core/data:rsci.idat                                                     0.0000  1.0000
     mult_add:core/data:slc(data:rsci.idat)(20-0)                                     0.0000  1.0000
     mult_add:core/data:slc(data:rsci.idat)(20-0).itm                                 0.0000  1.0000
     mult_add:core/MAC-1:mul                               mgc_mul_10_1_21_1_31_4     2.1640  3.1640
     mult_add:core/MAC-1:mul.itm                                                      0.0000  3.1640
     mult_add:core/MAC:acc#5                               mgc_addc_31_0_31_0_31_4    2.2730  5.4370
     mult_add:core/MAC:acc#5.itm                                                      0.0000  5.4370
     mult_add:core/MAC:acc#7                               mgc_addc_31_0_31_0_31_4    2.2731  7.7101
```

7. Expand the **Reports** folder by clicking on the "+". The reports folder contains the design constraint files, transcript messages, and RTL and cycle reports.

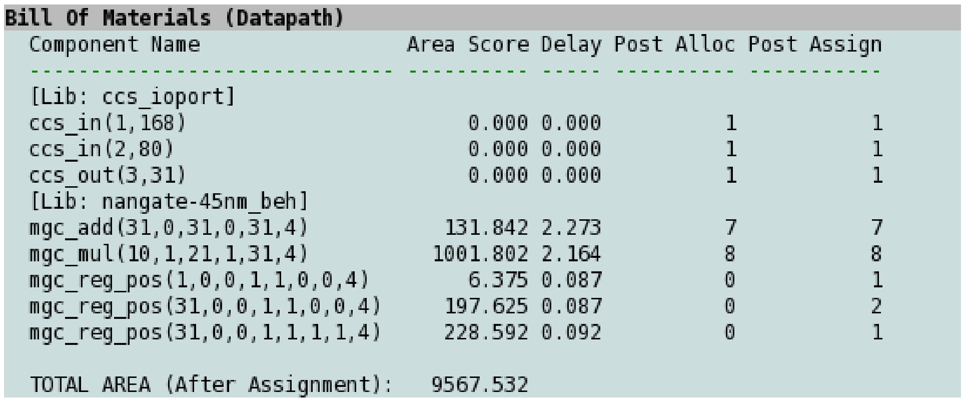


8. Double-click on **Commands** in the **Reports** folder. This file is called *directives.tcl* which contains all of the constraints that were applied to create the current solution. This file can be sourced in Catapult to exactly recreate the design. If you scroll through the file you can see the *CLOCKS* directive which sets the clock period to *10ns* which corresponds to the *100 MHz* clock frequency constraint that we set earlier.



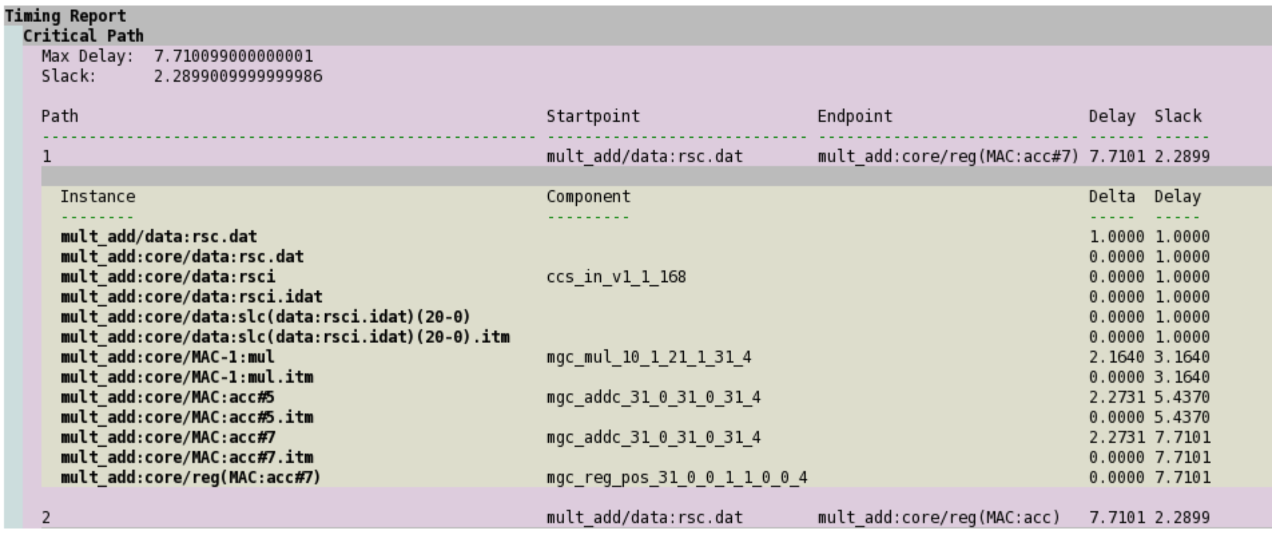
```
go compile
go libraries
directive set -CLOCKS {clk {-CLOCK_PERIOD 10.0 -
directive set -CLOCK_NAME clk
go assembly
```

9. Double-click on **RTL** in the **Reports** Folder. This is the *rtl.rpt* file for the current solution and contains things like *Bill of Materials* (BOM), area reports, timing, etc. Got to the *BOM* section and look at what's reported. You can see that the design required 8 multipliers and 7 adders by looking at the *Post Assign* column of the report. It also shows the area for each component as well as the total area.



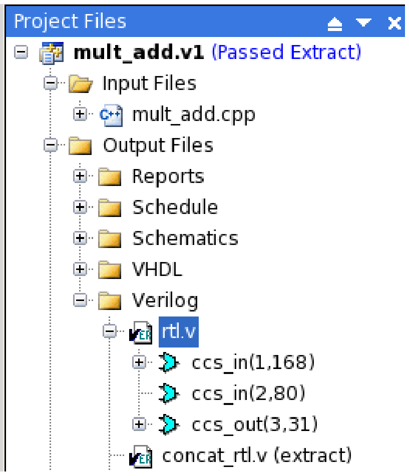
```
Bill Of Materials (Datapath)
  Component Name                 Area Score Delay Post Alloc Post Assign
  ------------------------------ ---------- ----- ---------- -----------
  [Lib: ccs_ioport]
  ccs_in(1,168)                       0.000 0.000          1           1
  ccs_in(2,80)                        0.000 0.000          1           1
  ccs_out(3,31)                       0.000 0.000          1           1
  [Lib: nangate-45nm_beh]
  mgc_add(31,0,31,0,31,4)           131.842 2.273          7           7
  mgc_mul(10,1,21,1,31,4)          1001.802 2.164          8           8
  mgc_reg_pos(1,0,0,1,1,0,0,4)        6.375 0.087          0           1
  mgc_reg_pos(31,0,0,1,1,0,0,4)     197.625 0.087          0           2
  mgc_reg_pos(31,0,0,1,1,1,1,4)     228.592 0.092          0           1

  TOTAL AREA (After Assignment):   9567.532
```

10. Scroll down in the *rtl.rpt* file to the *Timing Report* section. Catapult reports the critical path timing for the worst case paths in the design. Note that the design has met timing with positive slack. Also observe that the *1ns* Input Delay set on the *data:rsc* port has been accounted for in the timing reporting.

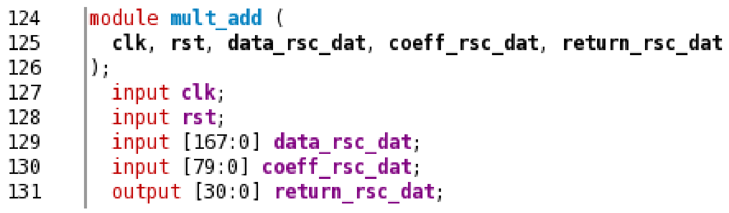


```
Timing Report
  Critical Path
    Max Delay:  7.710099000000001
    Slack:      2.2899009999999986

    Path                                                Startpoint                  Endpoint                    Delay  Slack
    --------------------------------------------------- --------------------------- --------------------------- ------ ------
    1                                                   mult_add/data:rsc.dat       mult_add:core/reg(MAC:acc#7) 7.7101 2.2899

      Instance                                          Component                                               Delta  Delay
      --------                                          ---------                                               -----  -----
      mult_add/data:rsc.dat                                                                                     1.0000 1.0000
      mult_add:core/data:rsc.dat                                                                                0.0000 1.0000
      mult_add:core/data:rsci                           ccs_in_v1_1_168                                         0.0000 1.0000
      mult_add:core/data:rsci.idat                                                                              0.0000 1.0000
      mult_add:core/data:slc(data:rsci.idat)(20-0)                                                              0.0000 1.0000
      mult_add:core/data:slc(data:rsci.idat)(20-0).itm                                                          0.0000 1.0000
      mult_add:core/MAC-1:mul                           mgc_mul_10_1_21_1_31_4                                  2.1640 3.1640
      mult_add:core/MAC-1:mul.itm                                                                               0.0000 3.1640
      mult_add:core/MAC:acc#5                           mgc_addc_31_0_31_0_31_4                                 2.2731 5.4370
      mult_add:core/MAC:acc#5.itm                                                                               0.0000 5.4370
      mult_add:core/MAC:acc#7                           mgc_addc_31_0_31_0_31_4                                 2.2731 7.7101
      mult_add:core/MAC:acc#7.itm                                                                               0.0000 7.7101
      mult_add:core/reg(MAC:acc#7)                      mgc_reg_pos_31_0_0_1_1_0_0_4                            0.0000 7.7101

    2                                                   mult_add/data:rsc.dat       mult_add:core/reg(MAC:acc)  7.7101 2.2899
```

11. Expand the **Verilog** folder in the **Output Files** folder. This contains the RTL netlist (*rtl.v)* of the design. It also contains files that concatenate all RTL netlist dependencies into a single file for both synthesis (*concat_rtl.v*).
12. Click on the "+" for *rtl.v* file. This shows the various netlist dependencies to other RTL files that were used in the design creation. This is usually I/O and memory modules. For this simple example, they are just basic wire interfaces from the builtin *ccs_ioport* library.

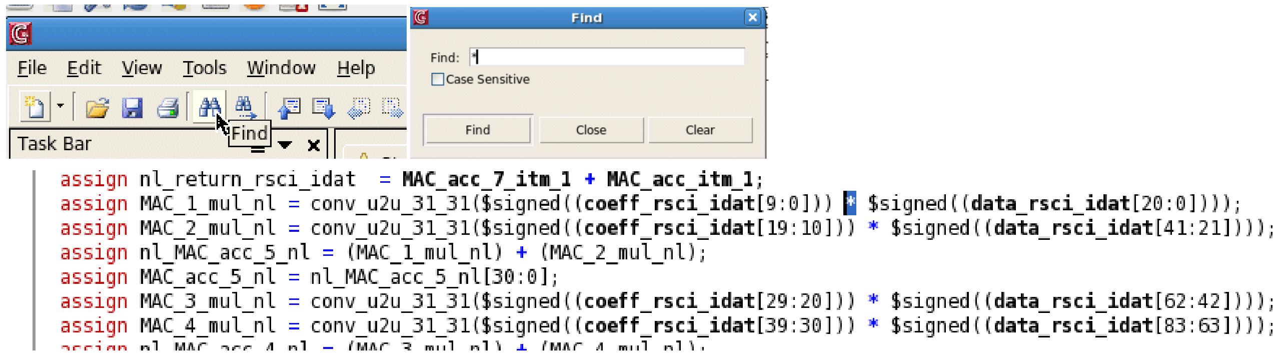


13. Double-click on the *rtl.v* file to open it. Scroll to the bottom of the file to find the *mult_add* module.



```
module mult_add (
  clk, rst, data_rsc_dat, coeff_rsc_dat, return_rsc_dat
);
  input clk;
  input rst;
  input [167:0] data_rsc_dat;
  input [79:0] coeff_rsc_dat;
  output [30:0] return_rsc_dat;
```

14. Move the cursor over the module name and port names and you will see the icon change into a hand icon. This indicates that they are cross-probable. Double-click on the *data_rsc_dat* port to cross-probe back to the C++. See how the *data* interface array is selected.
15. Go back to the *rtl.v* file and search for a multiply operation which is a "*". You can search by clicking on the *binocular* icon in the top-left tool bar. Once you find the "*" double-click on it and it should cross probe bask to the multiply in the C++ source.



```
assign nl_return_rsci_idat  = MAC_acc_7_itm_1 + MAC_acc_itm_1;
assign MAC_1_mul_nl = conv_u2u_31_31($signed((coeff_rsci_idat[9:0])) * $signed((data_rsci_idat[20:0])));
assign MAC_2_mul_nl = conv_u2u_31_31($signed((coeff_rsci_idat[19:10])) * $signed((data_rsci_idat[41:21])));
assign nl_MAC_acc_5_nl = (MAC_1_mul_nl) + (MAC_2_mul_nl);
assign MAC_acc_5_nl = nl_MAC_acc_5_nl[30:0];
assign MAC_3_mul_nl = conv_u2u_31_31($signed((coeff_rsci_idat[29:20])) * $signed((data_rsci_idat[62:42])));
assign MAC_4_mul_nl = conv_u2u_31_31($signed((coeff_rsci_idat[39:30])) * $signed((data_rsci_idat[83:63])));
assign nl_MAC_acc_4_nl = (MAC_3_mul_nl) + (MAC_4_mul_nl);
```

16. As a last step, open a terminal and go to the *Lab1* directory.
17. Change directories to the *Catapult* project directory and type *ls*. You should see two directories *mult_add.v1* and *SIF*. The *SIF* directory is the Catapult database files. The *mult_add.v1* directory is the solution directory for the design.

```
> cd Lab1
> ls
mult_add.v1 SIF
```

18. Change to the *mult_add.v1* directory. This directory contains all of the output files for the *mult_add.v1* solution, including *rtl.v*, *directives.tcl, rtl.rpt*, etc. You will also find a file called *concat_rtl.v* which contains all design netlists in a single file, including the I/O and memory netlists.

```
> cd mult_add.v1
> ls
adjust_char_library.tcl  cycle.rpt        res_sharing.tcl      rtl.v_order.txt

concat_rtl.v             cycle_set.tcl    rtl.rpt              rtl.v.or.sdc
concat_rtl.v.or          directives.tcl   rtl.v                schedule.gnt
concat_rtl.v.or.sdc      messages.txt     rtl.v.or             schematic.nlv
concat_sim_rtl.v         reg_sharing.tcl  rtl.v_order_sim.txt solIndex.xml
```

DONE Lab1

Lab2: Apply Interface Protocols and Verification

In this lab you will learn how to:

- Apply interface synthesis constraints to add a handshaking protocol to design inputs and outputs
- Automatically verify the generated RTL using the original C++ test bench
- Set the SCVerify transaction controls to inject stalls on an handshaking interface during verification

To complete this lab, follow these steps:

1. Invoke Catapult
2. Read the Design Files
3. Specify Hierarchy
4. Load the Technology Library
5. Specify the Timing Constraints
6. Specify Architecture Constraints (Interface Protocols)
7. Verify the Design

Invoke Catapult

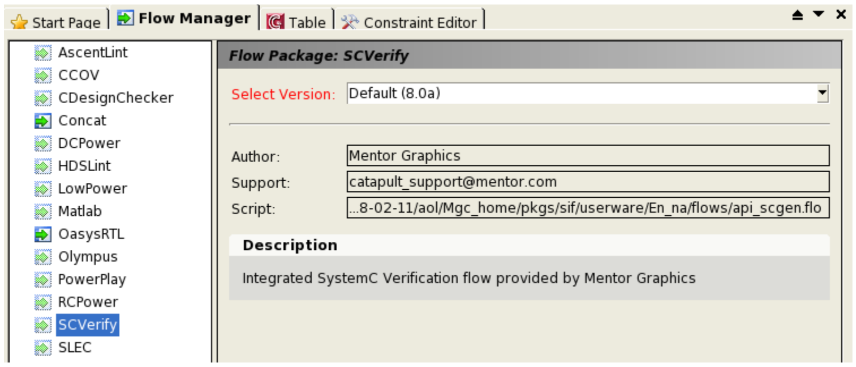
1. Open a terminal and cd into the Lab2 directory.

```
cd ../Lab2
```

2. Type catapult at the Linux command prompt.

```
catapult
```

3. Click on the **Flow Manager** tab, select **SCVerify**, and click the **enable** button to enable the verification flow. NOTE: Catapult Ultra enables SCVerify automatically.



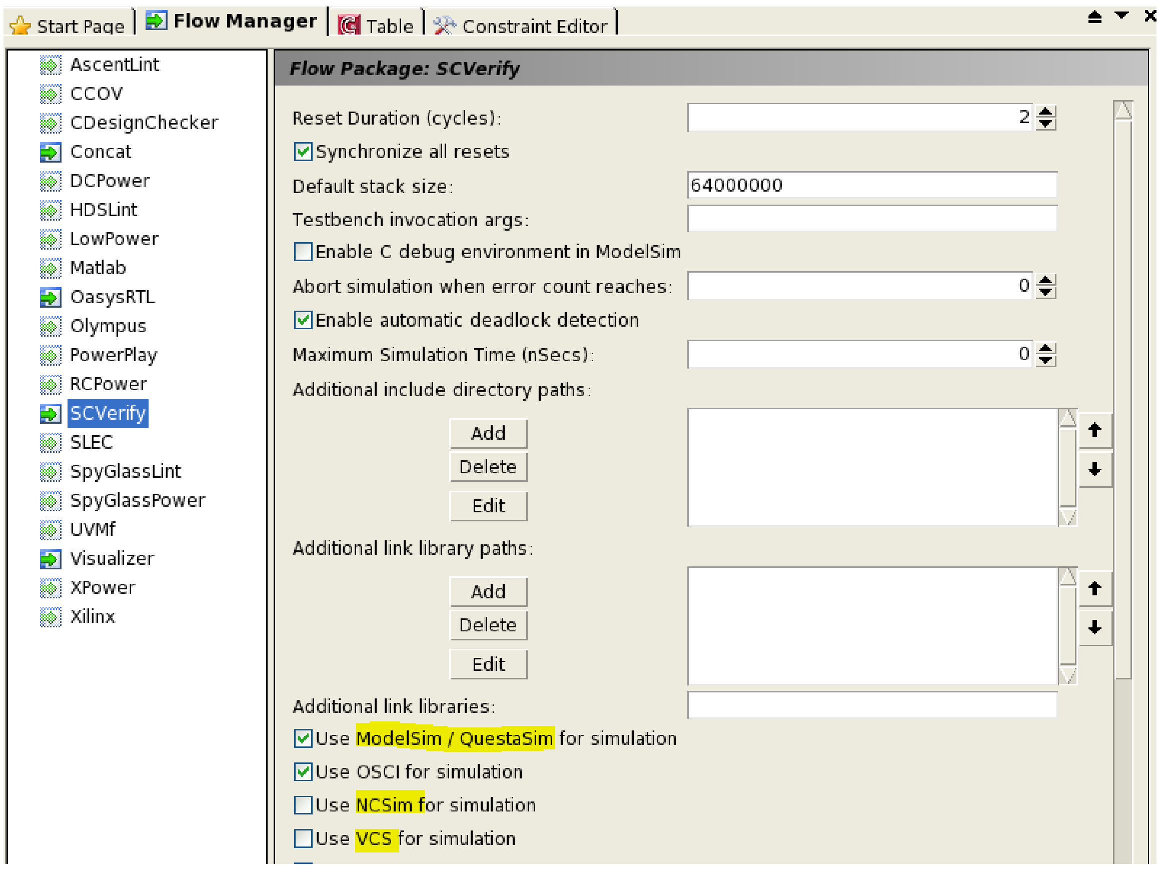
Flow Package: SCVerify

Select Version: Default (8.0a)

Author: Mentor Graphics
Support: catapult_support@mentor.com
Script: ...8-02-11/aol/Mgc_home/pkgs/sif/userware/En_na/flows/api_scgen.flo

Description
Integrated SystemC Verification flow provided by Mentor Graphics

4. Scroll down in the **Flow Manager SCVerify** view and enable the **RTL simulator** you use. The default is *Modelsim/Questa*.



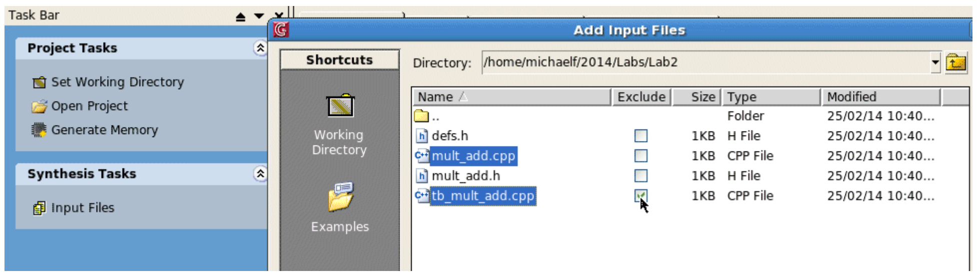
Flow Package: SCVerify

Reset Duration (cycles): 2
[✓] Synchronize all resets
Default stack size: 64000000
Testbench invocation args:
[ ] Enable C debug environment in ModelSim
Abort simulation when error count reaches: 0
[✓] Enable automatic deadlock detection
Maximum Simulation Time (nSecs): 0
Additional include directory paths:
Additional link library paths:
Additional link libraries:
[✓] Use ModelSim / QuestaSim for simulation
[✓] Use OSCI for simulation
[ ] Use NCSim for simulation
[ ] Use VCS for simulation

5. Click **Apply**

Read the Design Files

1. Click on **Input Files** in the Catapult Task Bar.
2. Add the *mult_add.cpp* file(this file is the same design we used in Lab1) and the *tb_mult_add.cpp* file. Make sure to exclude the *tb_mult_add.cpp* file from compilation since it is the C++ testbench and is not synthesizable.



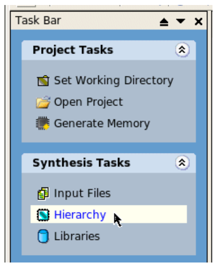
Directory: /home/michaelf/2014/Labs/Lab2

Name	Exclude	Size	Type	Modified
..			Folder	25/02/14 10:40...
defs.h	☐	1KB	H File	25/02/14 10:40...
mult_add.cpp	☐	1KB	CPP File	25/02/14 10:40...
mult_add.h	☐	1KB	H File	25/02/14 10:40...
tb_mult_add.cpp	✓	1KB	CPP File	25/02/14 10:40...

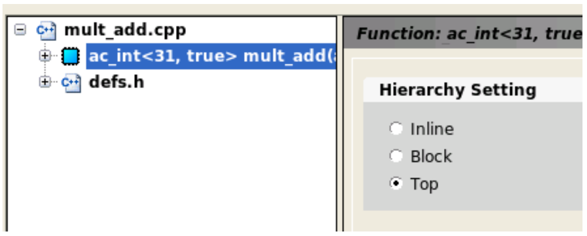
3. Click **Open**.

Specify Hierarchy

1. Click on **Hierarchy** in the Task Bar.

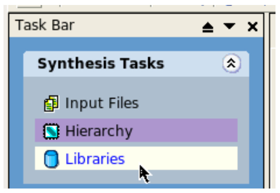


2. Click to select the "mult_add" function and note that its Hierarchy Setting is set to "Top". This is also indicated by the blue chip icon with the black edges.

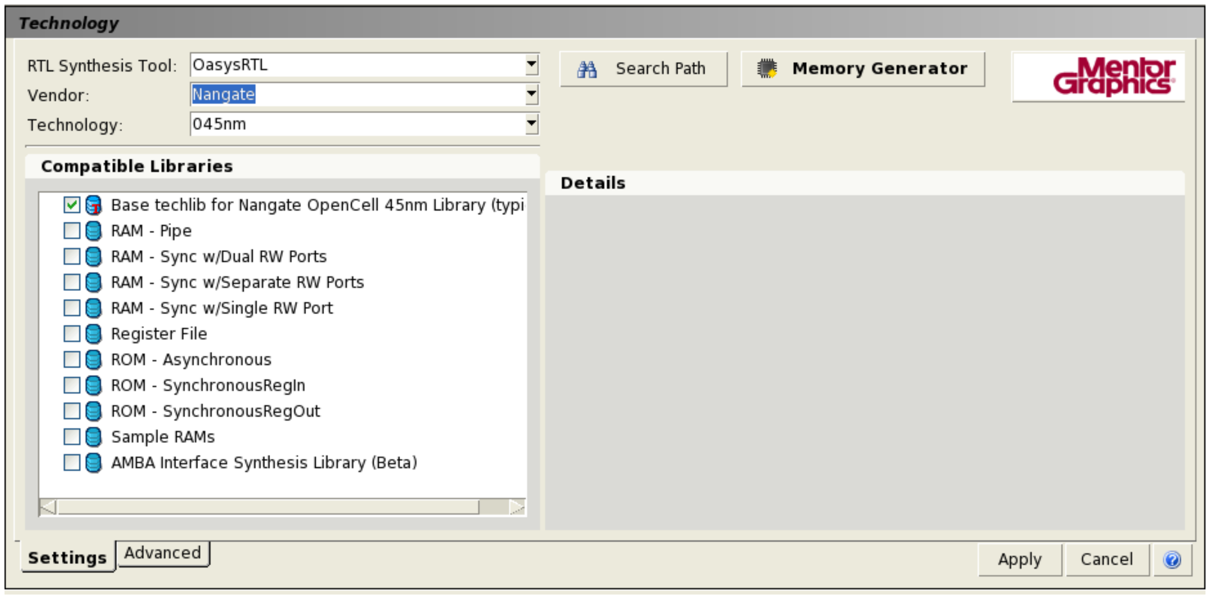


Load the Technology Library

1. Click on **Libraries** in the task bar.



2. Set the **RTL Synthesis tool** to *OasysRTL*, **Vendor** to *Nangate*, and **Technology** to *045nm*



Technology

RTL Synthesis Tool: OasysRTL
Vendor: Nangate
Technology: 045nm

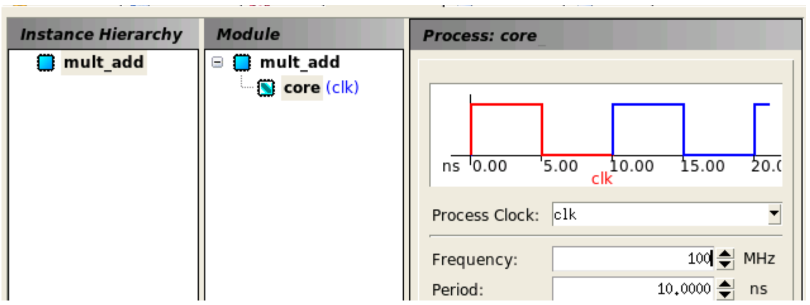
Compatible Libraries

- [x] Base techlib for Nangate OpenCell 45nm Library (typi
- [ ] RAM - Pipe
- [ ] RAM - Sync w/Dual RW Ports
- [ ] RAM - Sync w/Separate RW Ports
- [ ] RAM - Sync w/Single RW Port
- [ ] Register File
- [ ] ROM - Asynchronous
- [ ] ROM - SynchronousRegIn
- [ ] ROM - SynchronousRegOut
- [ ] Sample RAMs
- [ ] AMBA Interface Synthesis Library (Beta)

3. Click on the **Apply** button.

Specify the Timing Constraints

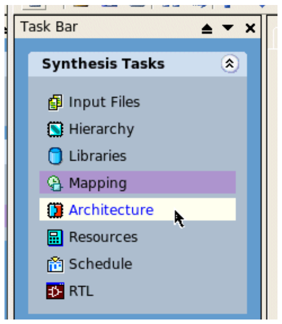
1. Click on **Mapping** in the Task Bar.
2. Set the **Clock Frequency** to *100 MHz* and click **Apply**.



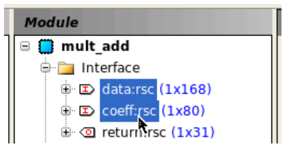
Process Clock: clk
Frequency: 100 MHz
Period: 10.0000 ns

Specify Architecture Constraints (Interface Protocols)

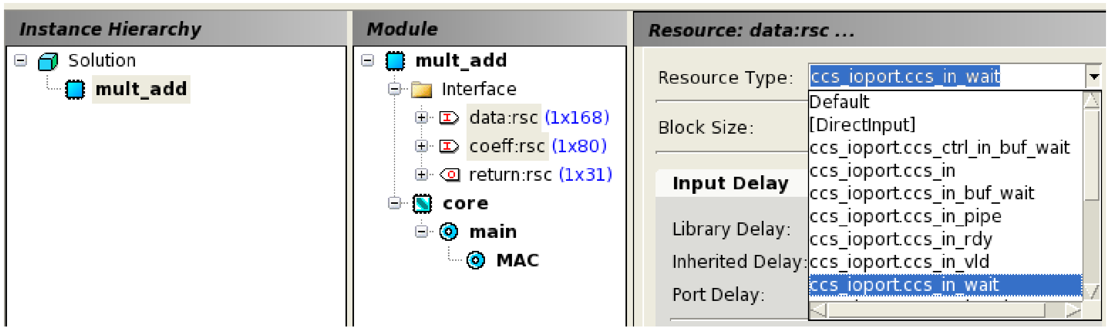
1. Click on **Architecture** in the Task Bar. This will bring up the *Architectural Constraints* Editor.



2. Click on the "+" next to the **Interface** folder.
3. Click on the *data:rsc* interface to select it and while holding the shift key down click on *coeff:rsc* to select it as well.



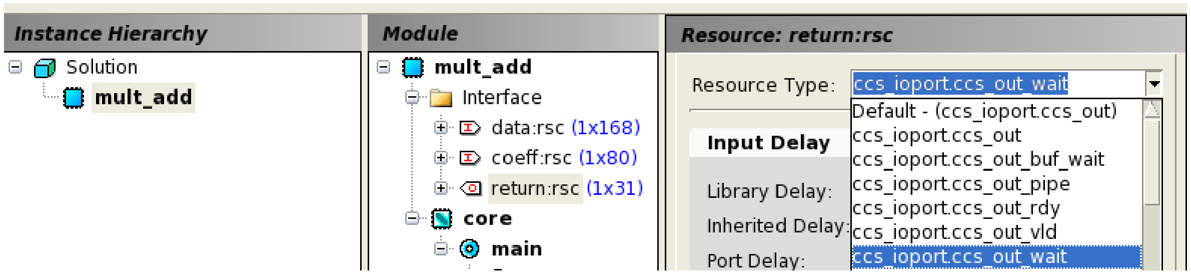
4. Click on the Resource Type drop-down list and select *ccs_ioport.ccs_in_wait*. This will map *data:rsc* and *coeff:rsc* to handshaking (data/ready/valid) input interfaces.



Resource Type: ccs_ioport.ccs_in_wait

- Default
- [DirectInput]
- ccs_ioport.ccs_ctrl_in_buf_wait
- ccs_ioport.ccs_in
- ccs_ioport.ccs_in_buf_wait
- ccs_ioport.ccs_in_pipe
- ccs_ioport.ccs_in_rdy
- ccs_ioport.ccs_in_vld
- ccs_ioport.ccs_in_wait

5. Click on *return:rsc* and map its resource type to *ccs_ioport.ccs_out_wait* to map it to a handshaking(data/ready/valid) output interface.



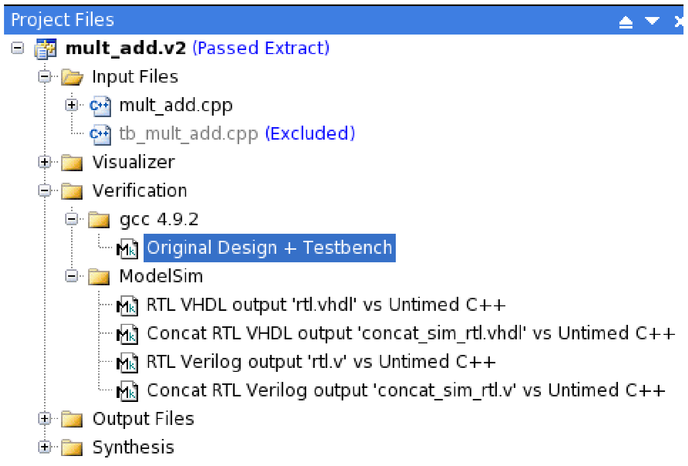
Resource Type: ccs_ioport.ccs_out_wait

- Default - (ccs_ioport.ccs_out)
- ccs_ioport.ccs_out
- ccs_ioport.ccs_out_buf_wait
- ccs_ioport.ccs_out_pipe
- ccs_ioport.ccs_out_rdy
- ccs_ioport.ccs_out_vld
- ccs_ioport.ccs_out_wait

6. Click the **Apply** button.
7. Go to the **File** Menu and select the **Run Script** item. Select the *constraints.tcl* file and click **open**. This file is applying constraints for parallelism that we have not covered yet, but will cover in a later section.
8. Click on **RTL** in the Catapult Task Bar

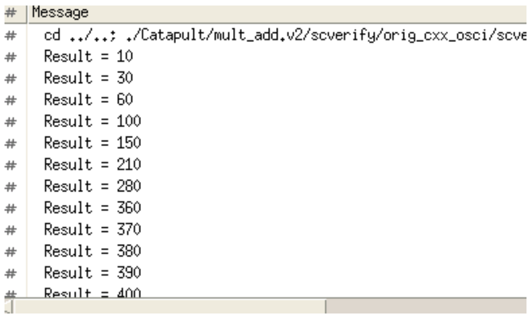
Verify the Design

1. In the Project Files view you will now see a Verification folder. Click on the "+" to expand the folder. Inside the verification folder you will see folders for gcc as well as the RTL simulator that was chosen in the SCVerify setup options. Each of these folders contains Makefiles for launching verification. The gcc flow will let you compile and execute the design and testbench inside of Catapult. Any standard IO will be printed to the Catapult transcript.



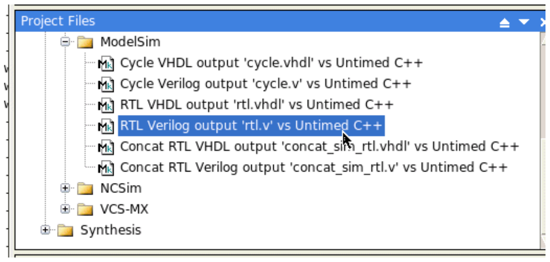
- mult_add.v2 (Passed Extract)
  - Input Files
    - mult_add.cpp
    - tb_mult_add.cpp (Excluded)
  - Visualizer
  - Verification
    - gcc 4.9.2
      - Original Design + Testbench
    - ModelSim
      - RTL VHDL output 'rtl.vhdl' vs Untimed C++
      - Concat RTL VHDL output 'concat_sim_rtl.vhdl' vs Untimed C++
      - RTL Verilog output 'rtl.v' vs Untimed C++
      - Concat RTL Verilog output 'concat_sim_rtl.v' vs Untimed C++
  - Output Files
  - Synthesis

2. Expand the **gcc** folder and double-click on "**Original Design** + **Testbench**" to compile and execute the design. Look in the Catapult transcript and you will see the testbench output being printed. You may have to scroll up in the transcript to see the first values printed.

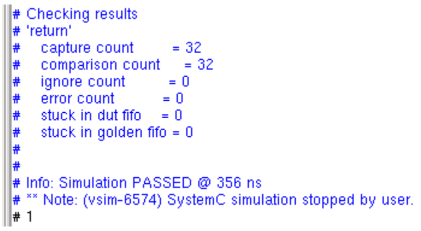


```
#  Message
#    cd ../..; ./Catapult/mult_add.v2/scverify/orig_cxx_osci/scve
#    Result = 10
#    Result = 30
#    Result = 60
#    Result = 100
#    Result = 150
#    Result = 210
#    Result = 280
#    Result = 360
#    Result = 370
#    Result = 380
#    Result = 390
#    Result = 400
```

3. Go to the **Input Files** folder and double-click on *tb_mult_add.cpp* fileto open it.
4. Look through the testbench file and you can see that the Catapult verification header file *mc_scverify.h* had been included. Also note that the verification macros *CCS_MAIN, CCS_DESIGN*, and *CCS_RETURN* have all been used. The testbench sets all of the *coeff* values equal to *10* and the *data* values equal to *zero*. The *data* array elements are then written one at a time for each call to *mult_add.* You will also see some code for controlling the stall behavior of the IO. We will cover that later.
5. In the **Verification** Folder expand the folder for the *RTL simulator* you are using and double-click on the Makefile called **RTL Verilog output vs. Untimed C++**. This will launch the RTL simulator in interactive mode. You can also right-click on the Makefile and run in batch mode if you don't wish to view the simulation waveforms.


Project Files
ModelSim
Cycle VHDL output 'cycle.vhdl' vs Untimed C++
Cycle Verilog output 'cycle.v' vs Untimed C++
RTL VHDL output 'rtl.vhdl' vs Untimed C++
RTL Verilog output 'rtl.v' vs Untimed C++
Concat RTL VHDL output 'concat_sim_rtl.vhdl' vs Untimed C++
Concat RTL Verilog output 'concat_sim_rtl.v' vs Untimed C++
NCSim
VCS-MX
Synthesis

6. Type "*run –all*" in the simulator command line.
7. View the simulator transcript and you should see a *Simulation Passed* message, indicating that the RTL has been compared against the C++ and there are no errors.

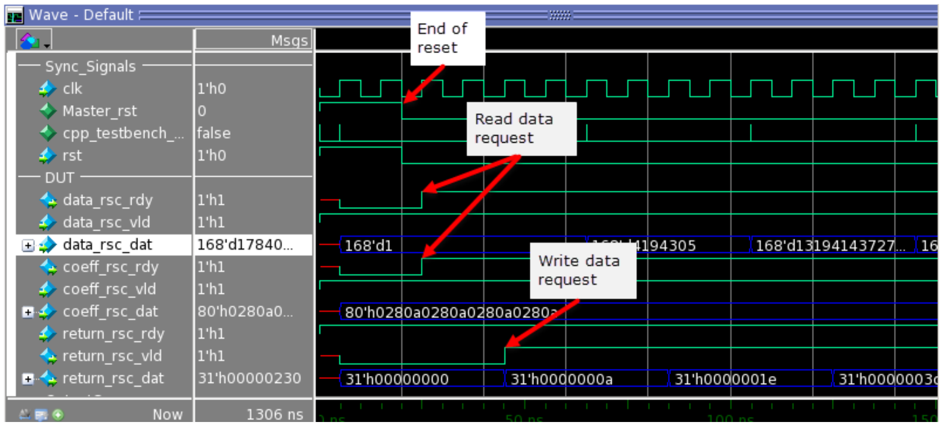


```
# Checking results
# 'return'
#    capture count        = 32
#    comparison count     = 32
#    ignore count         = 0
#    error count          = 0
#    stuck in dut fifo    = 0
#    stuck in golden fifo = 0
#
#
# Info: Simulation PASSED @ 356 ns
# ** Note: (vsim-6574) SystemC simulation stopped by user.
# 1
```

8. Open the waveform viewer of the RTL simulator and zoom all the way out.
9. Set the radix for *return_rsc_dat* to *unsigned*.
10. Look at the waveform view and you can see that once the design comes out of reset the *data* and *coeff* interfaces start requesting data every clock cycle by driving their "*_rdy*" (Ready for data) signals high. The test bench always supplies data in this case since the "*_vld*"(data available) signals for *data* and *coeff* are always driven high by the testbench. You will also see that one cycle after the input data is

read the "return" interface drives its *_vld* signal high indicating that the write data is available. The testbench in this case is always ready to receive the data since the *return_rsc_vld* signal is always driven high by the testbench.

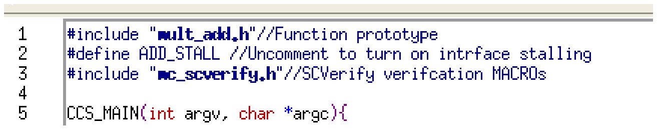
11. Look at the data values for *return_rsc_dat* and note that they match the data that was printed out in the Catapult transcript when the C++ and testbench was compiled and executed.



12. Close the RTL simulator

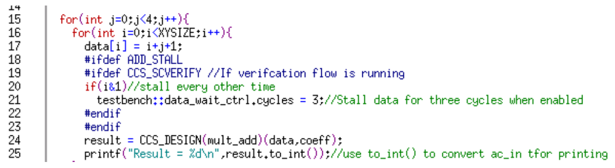
13. One thing you may have noticed about the simulation is that the testbench never stalls the IO, even though we have mapped all of the interfaces to handshaking interfaces. Go back to the **Input Files** view and double-click on the *tb_mult_add.cpp* file.

14. Uncomment the line at the top that defines "*ADD_STALL*". This define is used to enable the transaction controls for stalling the *data* input to *mult_add*.



```
#include "mult_add.h"//Function prototype
#define ADD_STALL //Uncomment to turn on intrface stalling
#include "mc_scverify.h"//SCVerify verifcation MACROs

CCS_MAIN(int argv, char *argc){
```

15. Scroll down in the file and you can see that with "ADD_STALL" defined the testbench will stall "data" for three cycles for every other iteration of the loop variable "i".

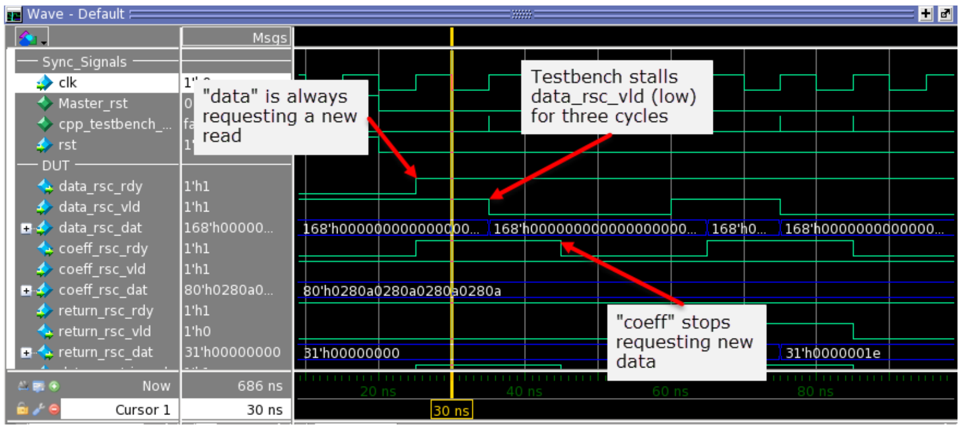


```
  for(int j=0;j<4;j++){
    for(int i=0;i<XYSIZE;i++){
      data[i] = i+j+1;
      #ifdef ADD_STALL
      #ifdef CCS_SCVERIFY //If verifcation flow is running
      if(i&1)//stall every other time
        testbench::data_wait_ctrl.cycles = 3;//Stall data for three cycles when enabled
      #endif
      #endif
      result = CCS_DESIGN(mult_add)(data,coeff);
      printf("Result = %d\n",result.to_int());//use to_int() to convert ac_in tfor printing
```

16. Go back to the **Verification** folder in the **Project Files** Pane and re-run **RTL Verilog output vs. Untimed C++** by double-clicking on the Makefile.

17. Type *run –all* in the simulator command line.

18. Open the waveform viewer of the RTL simulator and zoom all the way out.



19. You can see in the waveform view how the testbench stalls *data* for three cycles, then runs for two. Also notice that the *coeff* interface stops requesting new input when the stall occurs since it has acquired the current *coeff* values that get multiplied against *data*.

DONE Lab2

Lab 3: Micro-architecture Exploration and Analysis

In this lab you will learn how to:

- Use Loop Unrolling and Loop Pipelining to control parallelism and throughput
- Use the Gantt Chart to analyze synthesis results
- Cross-probe from Gantt Chart to C++ source and from source to Gantt Chart

To complete this lab, you will perform these steps:

1. Perform Initial Synthesis
2. Pipeline the Main Loop
3. Partially Unroll the MAC Loop
4. Fully Unroll the MAC Loop
5. Increase the Clock Frequency

Perform Initial Synthesis

1. Open a terminal and cd into the Lab3 directory.

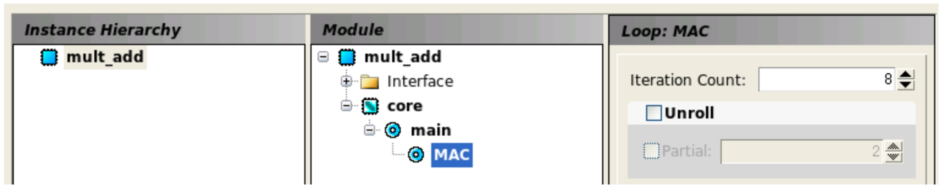
```
cd lab3
```

2. Type catapult at the command prompt.

```
catapult
```

3. Go to the **File** Menu and select **Run Script**. Select the *setup.tcl* and click **Open**. This will add in the *mult_add* design you used in the previous lab along with the testbench. The script also enables SCVerify, sets the technology to Nangate 45nm, clock to 100MHz, and maps all the interfaces to handshaking interfaces just like you did in the previous Lab2.
4. Click on **Architecture** in the Catapult Task Bar.
5. Click to select the *MAC* loop.

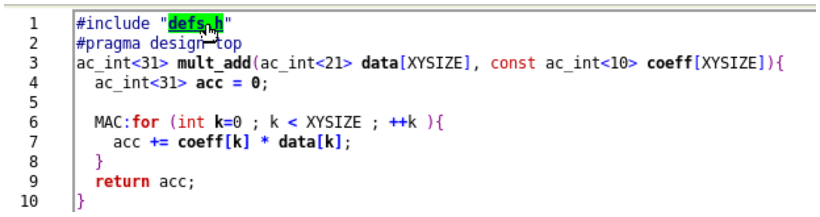
   In the previous labs the constraint file you were sourcing was fully unrolling this loop to give similar behavior to what one normally sees in Verilog synthesis, where the micro-architecture is always the most parallel version possible. It was also setting loop pipelining *Initiation Interval (II)* = 1 on the main loop. In other words synthesis of an RTL Verilog implementation always fully unrolls all loops, and a verilog clocked process has a throughput equal to 1. However one of the most powerful features of HLS is that it allows the designer to control the amount of loop unrolling (parallelism) and Initiation Interval (performance) to meet the design specification.



Iteration Count: 8
Unroll
Partial: 2

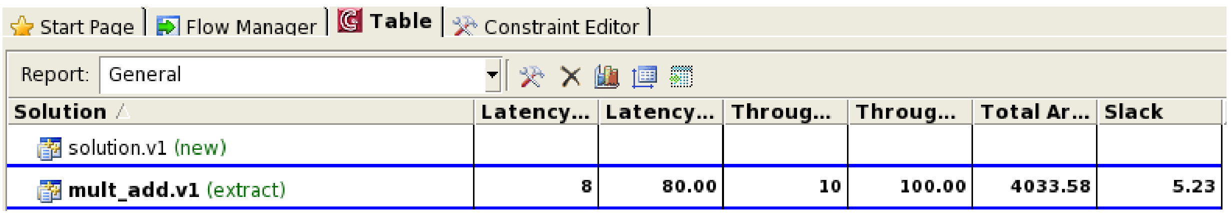
6. After selecting the *MAC* loop you should see that Catapult is reporting 8 iterations. You can also tell that Catapult *knows* the iteration count, otherwise it would display a "?" on the MAC loop icon.

7. Double-click on the *MAC* loop to cross-probe back to the C++ code. You can see why it's a good idea to label the loops in a design since it makes it obvious where things come from in the C++ code. Note that the loop bound is *XYSIZE*. Double-click on the "defs.h" file at the top and note that *XYSIZE* is defined equal to 8.



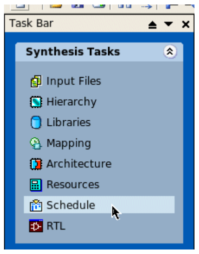
```
#include "defs.h"
#pragma design top
ac_int<31> mult_add(ac_int<21> data[XYSIZE], const ac_int<10> coeff[XYSIZE]){
  ac_int<31> acc = 0;

  MAC:for (int k=0 ; k < XYSIZE ; ++k ){
    acc += coeff[k] * data[k];
  }
  return acc;
}
```

8. Click on **RTL** in the Catapult **Task Bar.**
9. Look at the Catapult Table View and note the throughput equals 10 cycles.

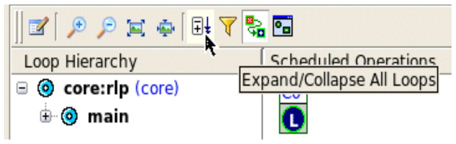


Solution	Latency...	Latency...	Throug...	Throug...	Total Ar...	Slack
solution.v1 (new)						
mult_add.v1 (extract)	8	80.00	10	100.00	4033.58	5.23

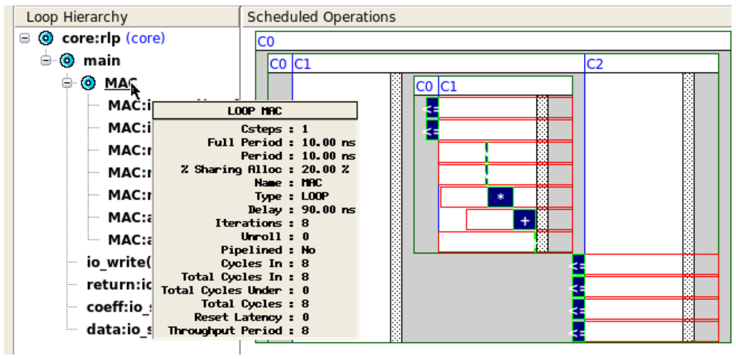
10. Click on **Schedule** in the Catapult Task Bar.



11. Expand all the loops in the Gantt chart by clicking on the Loop Expansion Icon in the tool bar.



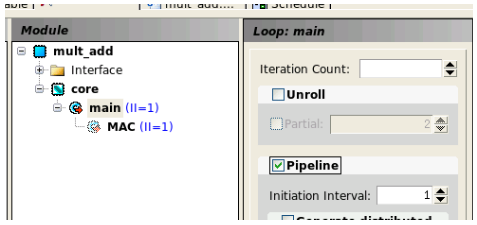
12. Hover over the *MAC* loop and you'll see that it takes 8 cycles to complete (8 Iterations x 1 c-step). Also note that the "main" loop takes two c-steps C1 and C2 plus the MAC loop (2 + 8 = 10 cycles throughput).



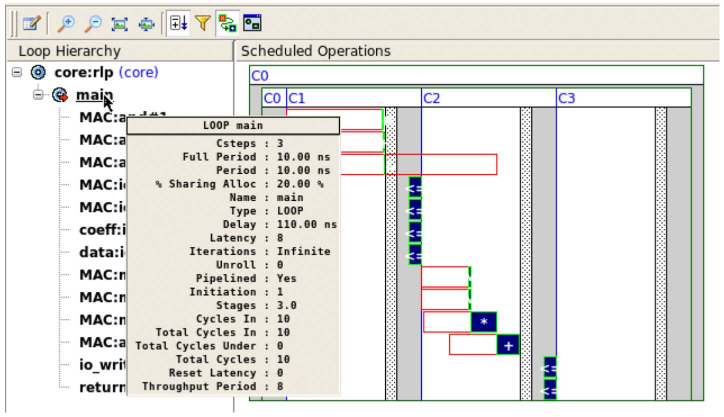
```
LOOP MAC
              Csteps : 1
         Full Period : 10.00 ns
              Period : 10.00 ns
    % Sharing Alloc : 20.00 %
                Name : MAC
                Type : LOOP
               Delay : 90.00 ns
          Iterations : 8
              Unroll : 0
           Pipelined : No
           Cycles In : 8
     Total Cycles In : 8
  Total Cycles Under : 0
        Total Cycles : 8
       Reset Latency : 0
   Throughput Period : 8
```

Pipeline the Main Loop

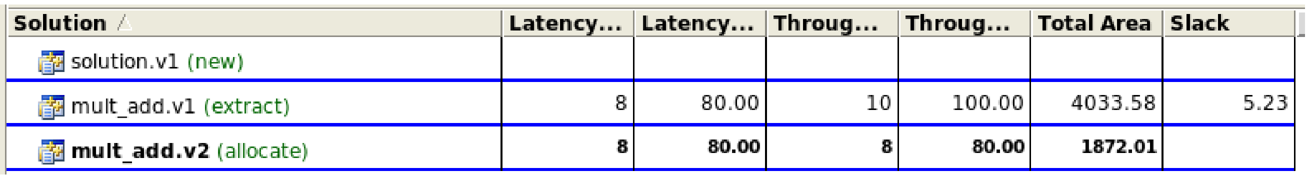
1. Click on **Architecture** in the Catapult **Task Bar**. We can use Loop Pipelining to improve performance by removing the effect of the 2 extra c-steps in the *main* loop.
2. Click on the *main* loop in the **Module** section of the **Architectural Constraints Editor** and set the **Initiation Interval** equal to *1*. This has the effect of running the MAC loop with *II=1* and starting a new *main* loop iteration every time the MAC loop finishes.



3. Click on **Schedule** in the Catapult **Task Bar.** Expand all loops and you'll see that the *MAC* loop has been *flattened* into the *main* loop, which now has a throughput of 8. Look at the Table view to see the throughput.

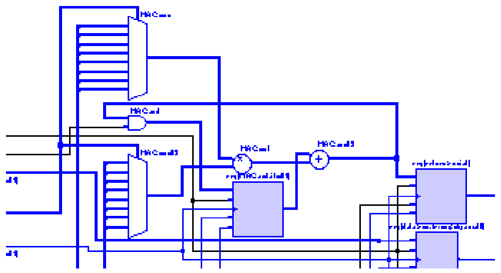


```
          LOOP main
              Csteps : 3
         Full Period : 10.00 ns
              Period : 10.00 ns
    % Sharing Alloc : 20.00 %
                Name : main
                Type : LOOP
               Delay : 110.00 ns
             Latency : 8
          Iterations : Infinite
              Unroll : 0
           Pipelined : Yes
          Initiation : 1
              Stages : 3.0
           Cycles In : 10
     Total Cycles In : 10
  Total Cycles Under : 0
        Total Cycles : 10
       Reset Latency : 0
   Throughput Period : 8
```

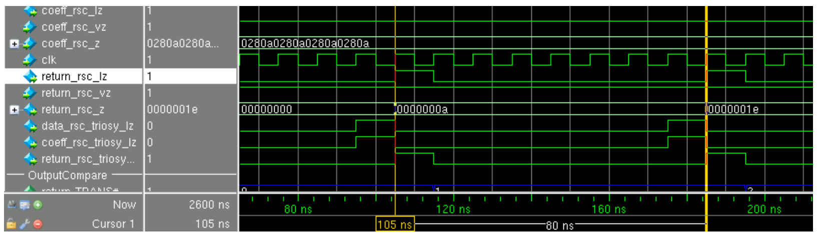


Solution	Latency...	Latency...	Throug...	Throug...	Total Area	Slack
solution.v1 (new)						
mult_add.v1 (extract)	8	80.00	10	100.00	4033.58	5.23
mult_add.v2 (allocate)	**8**	**80.00**	**8**	**80.00**	**1872.01**	

4. Double-click on the multiplier in the Gantt chart to cross-probe back to the C++ code.
5. Click on **RTL** in the **Task Bar**
6. Open the RTL Schematic and double-click on the *mult_add:core:inst* block to push down into the schematic. You'll see that the hardware has been implemented using a single multiplier and adder.



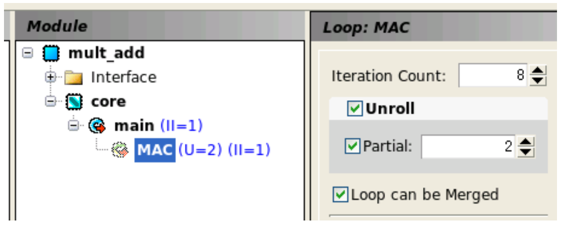
7. Double-click on the multiplier in the RTL schematic to cross-probe back to the C++.
8. Launch *SCVerify* verification in interactive mode on the Verilog RTL, run the simulation, and view it in the waveform viewer of the simulator. Verify that the throughput equals 8 clock cycles by checking the time between *return_rsc_vld* signal pulses. The *return_rsc_vld* port is the handshake signal that indicates the *return* output is being written.



9. If we want to improve the performance of the design to better than computing one output every 8 clock cycles we need to add parallelism to compute multiple values of *coeff[k]*data[k]* at the same time. We'll do this by using *Loop Unrolling*.

Partially Unroll the MAC Loop

1. Go to **Architectural** constraints, select the *MAC* loop and unroll *partially* by *2*. Do this by "checking" the **Partial** check box and set the **Unroll** amount equal to *2*. Note that the GUI shows the partially unrolled loop with a yellow and green circle.

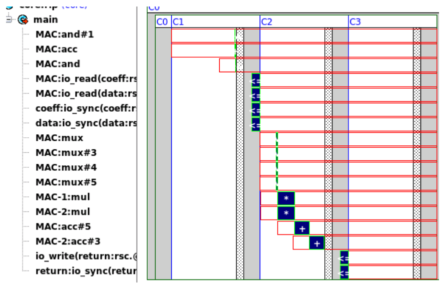


Module
- mult_add
  - Interface
  - core
    - main (II=1)
      - MAC (U=2) (II=1)

Loop: MAC

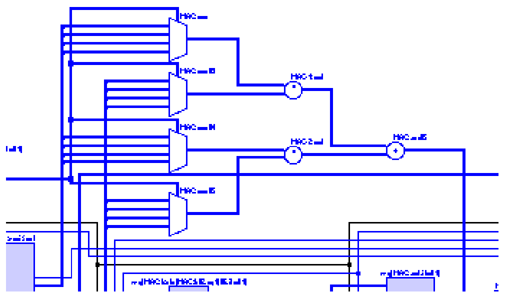
Iteration Count: 8
[✓] Unroll
[✓] Partial: 2
[✓] Loop can be Merged

2. Go to the **Schedule** view and note that the number of multipliers has doubled. Look at the Table view and note that the throughput now equals 4.



3. Double-click on the multipliers to cross-probe back to the C++.

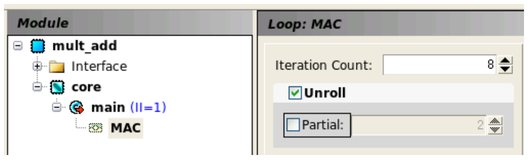
4. Generate **RTL**, open the *RTL schematic*, push down into the *mult_add:core:inst* block and note that there are two multipliers and two adders.



5. Run *SCVerify RTL verification* and verify the throughput of the design equals 4 in the waveform view.

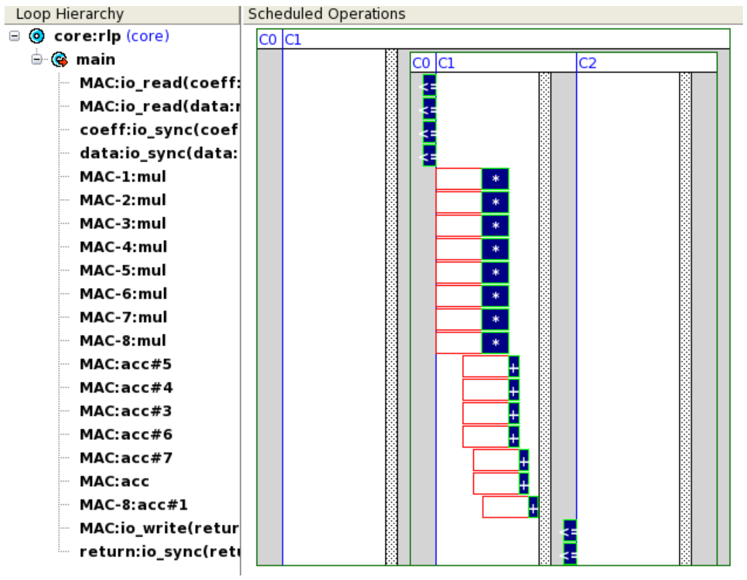
Fully Unroll the MAC Loop

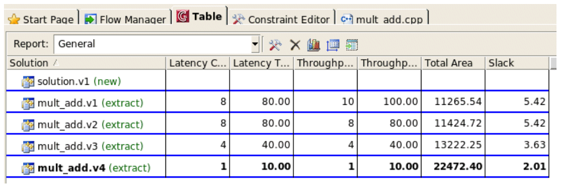
1. Go back to **Architecture** constraints and fully unroll the *MAC* loop. Catapult will now try to schedule all 8 iterations in parallel. Note that the GUI shows the fully unrolled loop with a yellow and green square icon.



Iteration Count: 8
✓ Unroll
Partial: 2

2. Go to the **Schedule** view and note that all 8 multiplies are scheduled in the same c-step along with the adder tree. Check the Table view and note that the throughput equals 1.



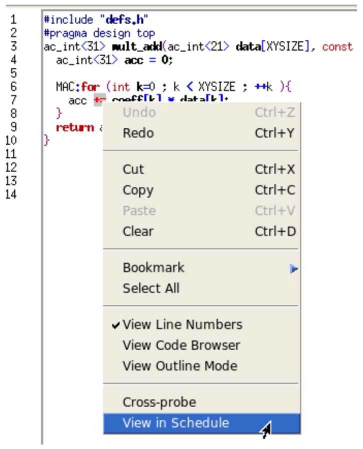


Report: General

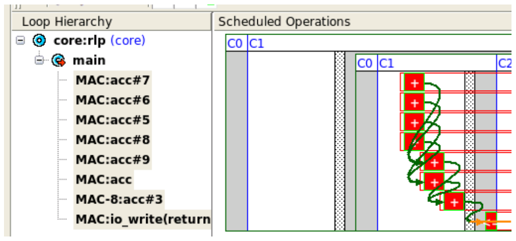
Solution	Latency C...	Latency T...	Throughp...	Throughp...	Total Area	Slack
solution.v1 (new)						
mult_add.v1 (extract)	8	80.00	10	100.00	11265.54	5.42
mult_add.v2 (extract)	8	80.00	8	80.00	11424.72	5.42
mult_add.v3 (extract)	4	40.00	4	40.00	13222.25	3.63
mult_add.v4 (extract)	**1**	**10.00**	**1**	**10.00**	**22472.40**	**2.01**

3. Open the *mult_add.cpp* file and use the mouse to click-drag-select the "+" operator in the *MAC* loop.

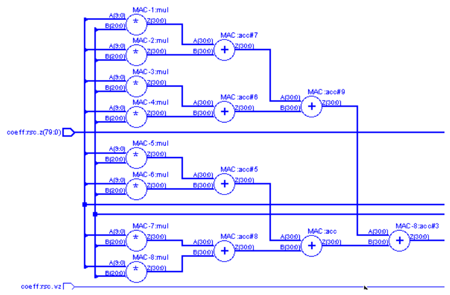
Right-click on the selected operator and choose **View in schedule** which will show you all adders that resulted from Loop Unrolling.



```
#include "defs.h"
#pragma design top
ac_int<31> mult_add(ac_int<21> data[XYSIZE], const
  ac_int<31> acc = 0;

  MAC:for (int k=0 ; k < XYSIZE ; ++k ){
    acc += coeff[k] * data[k];
  }
  return a
}
```

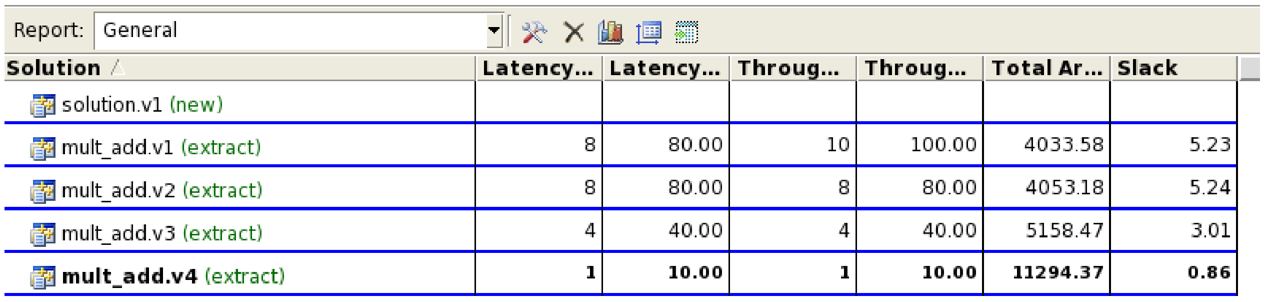


4. Generate RTL and view the RTL schematic. You can see that all 8 multiplies are done in parallel.



5. Run SCVerify verification to verify that the throughput equals one. You should see that the *return_rsc_vld* port goes high and stays high every clock cycle indicating that it is writing data every clock.

6. Go to the Catapult Table View and see how the area increases for each solution as the performance improves.

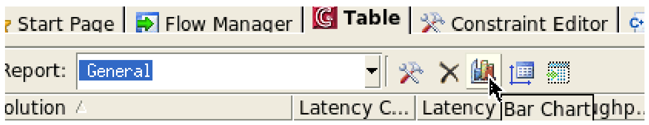


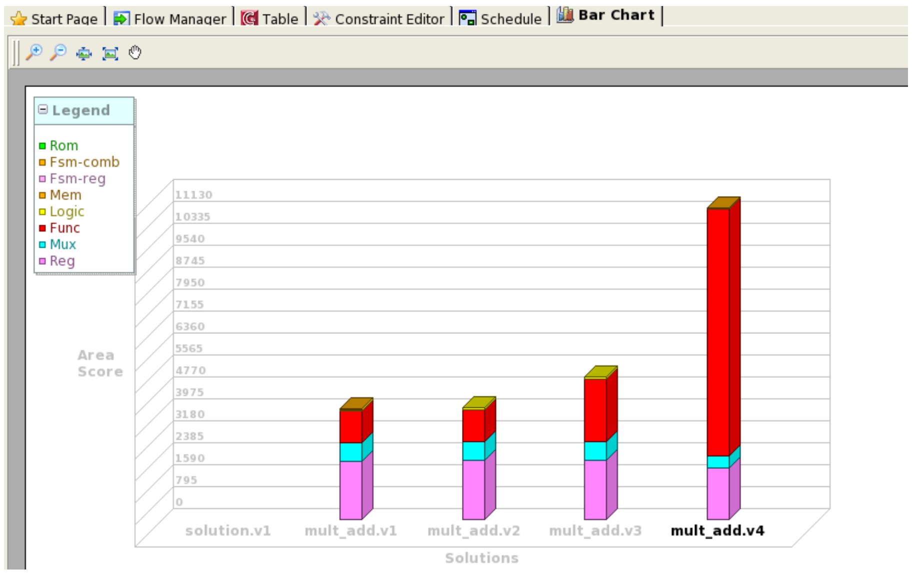
Report: General

Solution	Latency...	Latency...	Throug...	Throug...	Total Ar...	Slack
solution.v1 (new)						
mult_add.v1 (extract)	8	80.00	10	100.00	4033.58	5.23
mult_add.v2 (extract)	8	80.00	8	80.00	4053.18	5.24
mult_add.v3 (extract)	4	40.00	4	40.00	5158.47	3.01
mult_add.v4 (extract)	**1**	**10.00**	**1**	**10.00**	**11294.37**	**0.86**

7. Open the Bar Chart view and you can graphically see the area breakdown for the different

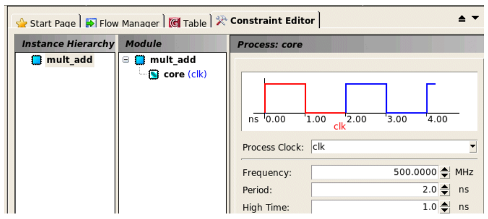
solutions/micro-architectures.





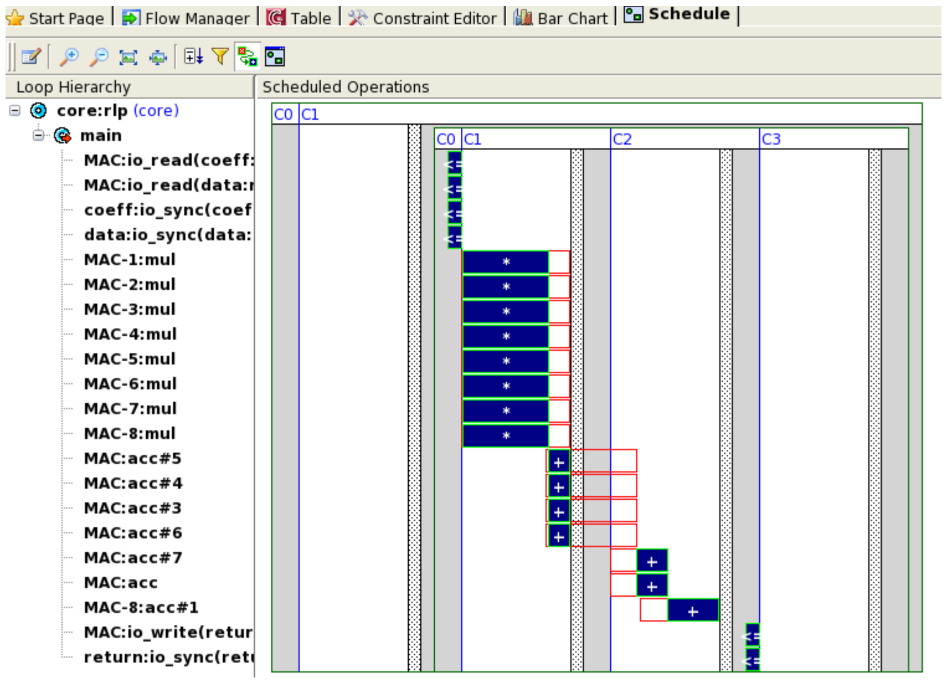
Increase the Clock Frequency

1. As a last step to illustrate the power of High Level Synthesis, go to the mapping view and set the clock frequency to *500MHz*.

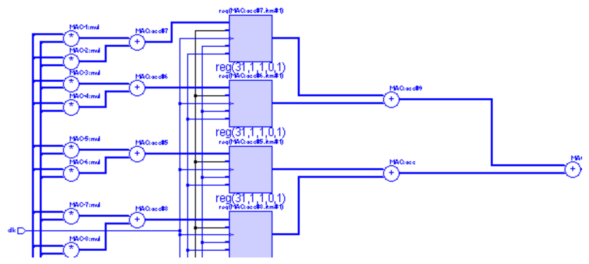


Process Clock: clk
Frequency: 500.0000 MHz
Period: 2.0 ns
High Time: 1.0 ns

2. Click on **Schedule** to re-schedule the design and you can see in the Gantt chart that Catapult scheduled the design with more c-steps by moving some of the additions into later cycles . In other words Catapult has added additional pipeline stages in order to meet the 500MHz clock frequency.



3. Look at the **Table view** and note that although the latency has increased to *2*, the throughput remains equal to *1* since the design is pipelined with *II=1* at the top-level.
4. Generate RTL, open the RTL schematic and you'll see that Catapult has inserted registers into the adder tree.



DONE Lab3

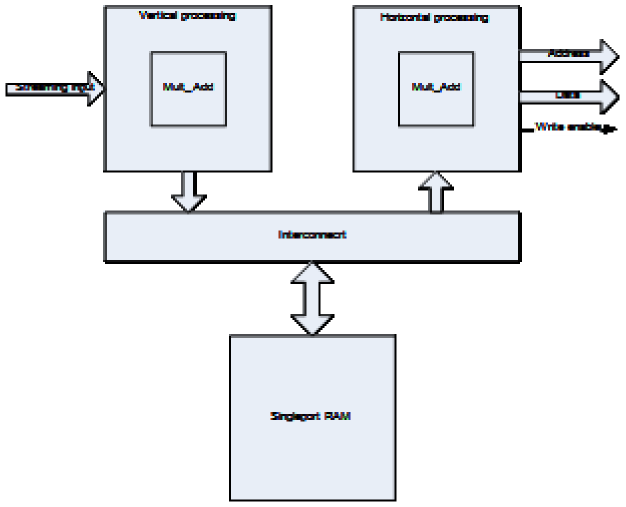
Lab 4 – Building a Discrete Cosine Transform (DCT) with an internal memory

In this lab you will learn how to:

- Use ac_channel to model a streaming input interface
- Add memory libraries to a design
- Map arrays to memories
- Map constant arrays to ROMs or MUX with constant inputs
- Use loop unrolling and pipelining to improve throughput
- Make an architectural code change to improve area

DCT overview and architecture

The DCT in this lab processes an 8x8 grid of pixels. DCT's are used in image and video compression algorithms and convert a 2-dimensional array of pixels from the spatial domain to the 2-d frequency domain. DCT's usually process the array of pixels vertically and then horizontally, with the intermediate results stored in a memory. For lower performance designs a single memory can be used for the vertical and horizontal processing. The core algorithm for both horizontal and vertical processing is a multiply-add structure. The general hardware architecture of the design for this lab is shown below. The single port memory is shared between the vertical and horizontal processing algorithms.



Design Goals

- Meet 300MHz timing
- Achieve a throughput of less than 1200 clock cycles to compute the DCT.
- Achieve a throughput of 128 clock cycles to compute the DCT.

- Reduce design area by changing the C++ code architecture to be more efficient.

Perform the Initial Synthesis

6. Open a terminal and cd into the Lab4 directory.

```
cd lab4
```

7. Type "catapult" at the command prompt.

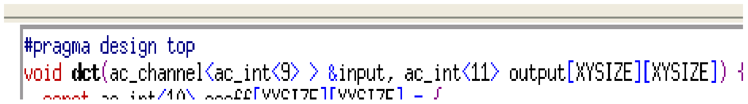
```
catapult
```

8. Go to the **File** Menu and select "**Run Script**". Select the *setup.tcl* and click **Open**.

   This will add in the *dct_stream.cpp*, *dct_ref.cpp*, and *tb_dct_stream.cpp* design files The script also enables SCVerify and sets the technology to Nangate 45nm.

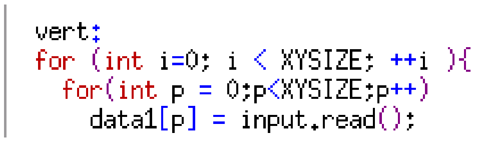
9. Open the *dct_stream.cpp* file and take a minute to see the general features of the design.

   - On the interface you will see that the input is a streaming interface modeled using the ac_channel class. The output is an array that will be mapped to a memory interface. Open the *defs.h* file and you'll see that *XYSIZE == 8*. So the *output* array is 8x8.



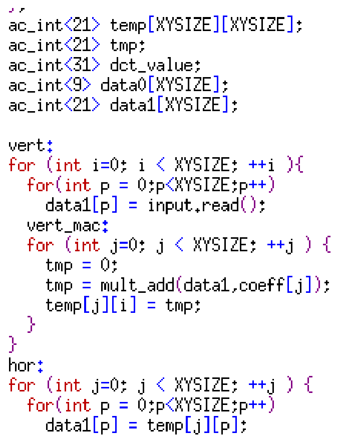
```
#pragma design top
void dct(ac_channel<ac_int<9> > &input, ac_int<11> output[XYSIZE][XYSIZE]) {
```

   - In the *vert* loop you'll see that the input is read using the ac_channel *read()* method. This essentially models a streaming interface behavior where pixel data is read sequentially over time.



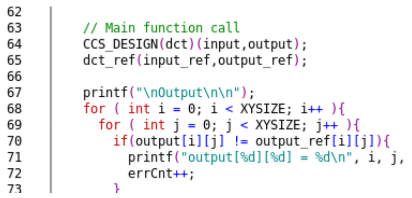
```
vert:
for (int i=0; i < XYSIZE; ++i ){
  for(int p = 0;p<XYSIZE;p++)
    data1[p] = input.read();
```

   - The core algorithm processes the input and writes it to the *temp* array in column order for the *vert* set of loops, and then the *horz* set of loops reads the data from the *temp* array in row order. Note that both *horz* and *vert* set of loops call a *mult_add* function that we have used in previous labs.



```
ac_int<21> temp[XYSIZE][XYSIZE];
ac_int<21> tmp;
ac_int<31> dct_value;
ac_int<9> data0[XYSIZE];
ac_int<21> data1[XYSIZE];

vert:
for (int i=0; i < XYSIZE; ++i ){
  for(int p = 0;p<XYSIZE;p++)
    data1[p] = input.read();
  vert_mac:
  for (int j=0; j < XYSIZE; ++j ) {
    tmp = 0;
    tmp = mult_add(data1,coeff[j]);
    temp[j][i] = tmp;
  }
}
hor:
for (int j=0; j < XYSIZE; ++j ) {
  for(int p = 0;p<XYSIZE;p++)
    data1[p] = temp[j][p];
```

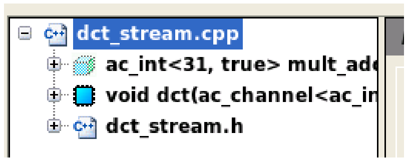
   - Open the *tb_dct_stream.cpp* file. You will see that this testbench is set up to run SCVerify. Also note that the testbench calls both the *dct* function and a reference design *dct_ref*. The *dct_ref* design is used to compare the synthesizable DCT model against a *golden* reference. This is useful for making code changes on the synthesis model and verifying that you haven't broken functionality.



```

    // Main function call
    CCS_DESIGN(dct)(input,output);
    dct_ref(input_ref,output_ref);

    printf("\nOutput\n\n");
    for ( int i = 0; i < XYSIZE; i++ ){
      for ( int j = 0; j < XYSIZE; j++ ){
        if(output[i][j] != output_ref[i][j]){
          printf("output[%d][%d] = %d\n", i, j,
          errCnt++;
        }
```

10. Click on **Hierarchy** in the Catapult **Task Bar.**

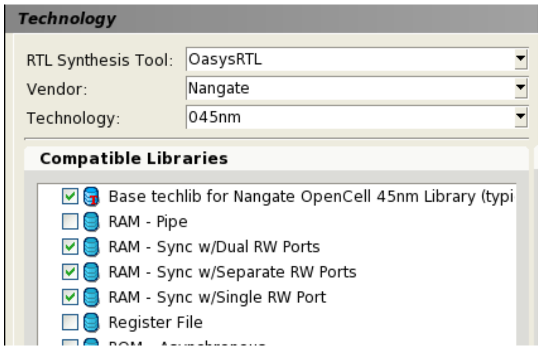
    You will see that the *dct* function is set as the top-level design.



```
⊟ dct_stream.cpp
   ⊞ ac_int<31, true> mult_ad
   ⊞ void dct(ac_channel<ac_i
   ⊞ dct_stream.h
```

11. Click on **Libraries** in the Catapult **Task Bar.**

    Using the OasysRTL > Nangate > 45nm technology, select the *Base ASIC Library* and the *Dual port, Separate RW,* and *Single port memory* libraries and click **Apply**.



Technology

RTL Synthesis Tool: OasysRTL
Vendor: Nangate
Technology: 045nm

Compatible Libraries

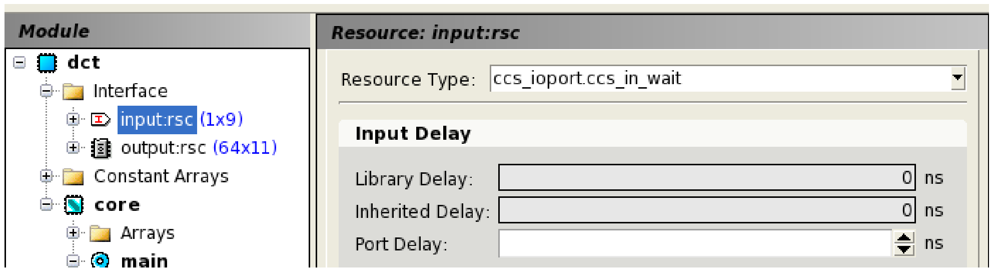
- [✓] Base techlib for Nangate OpenCell 45nm Library (typi
- [ ] RAM - Pipe
- [✓] RAM - Sync w/Dual RW Ports
- [✓] RAM - Sync w/Separate RW Ports
- [✓] RAM - Sync w/Single RW Port
- [ ] Register File

12. Click on **Mapping** in the **Task Bar**.

    Set the **clock frequency** to *300 MHz* and click **Apply**.

13. Click on **Architecture** in the Catapult **Task Bar**.

    Expand the interface folder and note that the interface variable *ac_channel<ac_int<9> &input* has been auto-mapped to an *ccs_in_wait* interface. This is the default interface mapping for ac_channel variables on the top-level interface.



Module

```
⊟ dct
   ⊟ Interface
      ⊞ input:rsc (1x9)
      ⊞ output:rsc (64x11)
   ⊞ Constant Arrays
   ⊟ core
      ⊞ Arrays
      ⊟ main
```

Resource: input:rsc

Resource Type: ccs_ioport.ccs_in_wait

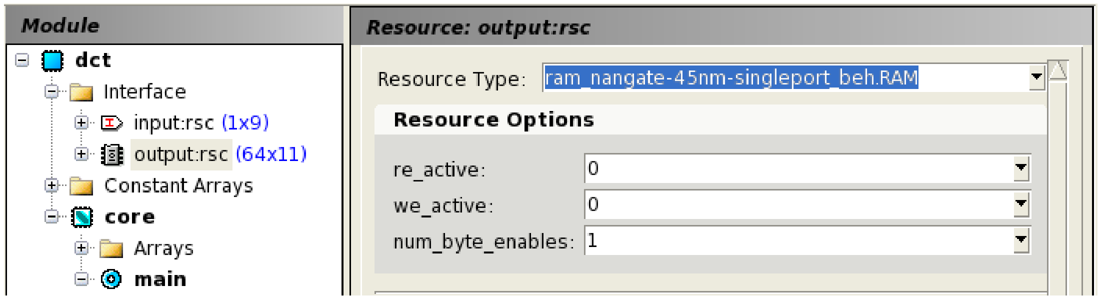
Input Delay

Library Delay: 0 ns
Inherited Delay: 0 ns
Port Delay: ns

14. Click on the *output:rsc* interface

    Note that it has been auto-mapped to a memory. Catapult will auto-map arrays to memories based on the number of array elements and the **Automatic Memory Mapping Threshold** set under **Tools > Set Options > Architectural**. The default threshold is *32*.

15. Set the **Resource Type** for *output:rsc* to **Singleport RAM**.



```
Module
⊟ dct
  ⊟ Interface
    ⊞ input:rsc (1x9)
    ⊞ output:rsc (64x11)
  ⊞ Constant Arrays
  ⊟ core
    ⊞ Arrays
    ⊟ main
```

Resource: output:rsc

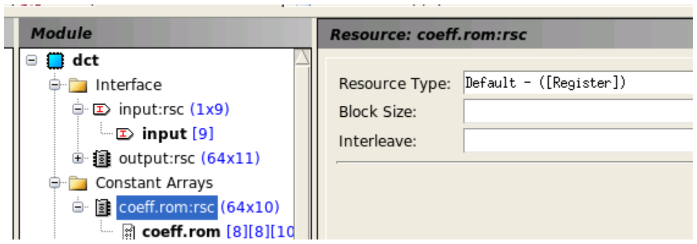
Resource Type: ram_nangate-45nm-singleport_beh.RAM

Resource Options

re_active: 0
we_active: 0
num_byte_enables: 1

16. Expand the **Constant Arrays** folder.

This folder contains any constant arrays that are defined in the design. Constant arrays can either be synthesized as a MUX with constant inputs, or as a ROM/LUT if a ROM library is available. Auto-mapping of constant arrays is controlled by the memory mapping threshold. In this example the constant array defaults to ([Register]) which results in a MUX with constant inputs.



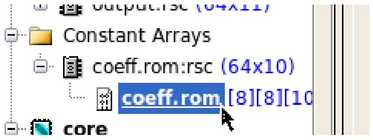
```
Module
⊟ dct
  ⊟ Interface
    ⊟ input:rsc (1x9)
      └ input [9]
    ⊞ output:rsc (64x11)
  ⊟ Constant Arrays
    ⊟ coeff.rom:rsc (64x10)
      └ coeff.rom [8][8][10
```

Resource: coeff.rom:rsc

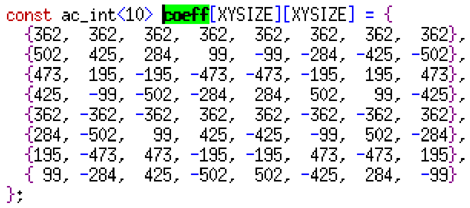
Resource Type: Default - ([Register])
Block Size:
Interleave:

17. Double-click on the *coeff* variable to cross-probe back to the C++ source.

You'll see that the DCT has an 8x8 array of constants.



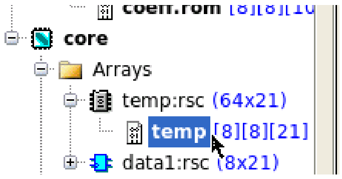
```
  ⊟ Constant Arrays
    ⊟ coeff.rom:rsc (64x10)
      └ coeff.rom [8][8][10
  ⊟ core
```



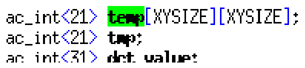
```
const ac_int<10> coeff[XYSIZE][XYSIZE] = {
  {362,  362,  362,  362,  362,  362,  362,  362},
  {502,  425,  284,   99,  -99, -284, -425, -502},
  {473,  195, -195, -473, -473, -195,  195,  473},
  {425,  -99, -502, -284,  284,  502,   99, -425},
  {362, -362, -362,  362,  362, -362, -362,  362},
  {284, -502,   99,  425, -425,  -99,  502, -284},
  {195, -473,  473, -195, -195,  473, -473,  195},
  { 99, -284,  425, -502,  502, -425,  284,  -99}
};
```

18. Expand the internal **Arrays** folder which will display all arrays that are internal to the C++ design.

Note that the *temp:rsc* resource has been auto-mapped to a RAM. Double-click on the *temp* variable under the *temp:rsc* resource and note that this is the 8x8 *temp* array that is used to store data between the horizontal and vertical processing.

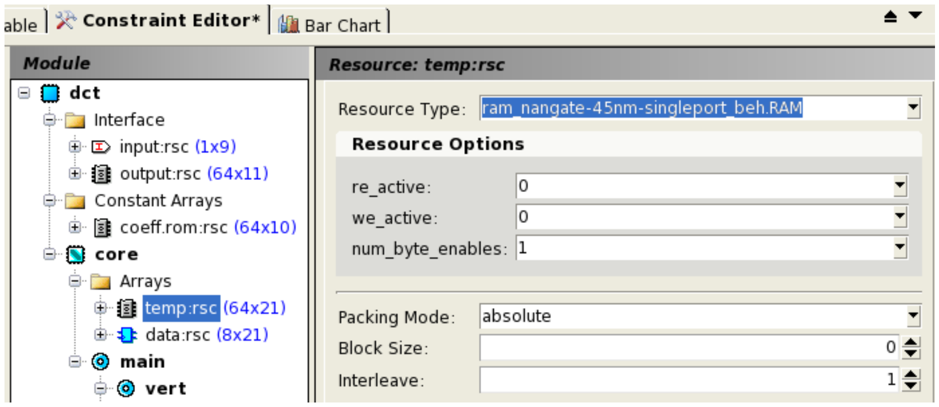


```
      └ coeff.rom [8][8][10
⊟ core
  ⊟ Arrays
    ⊟ temp:rsc (64x21)
      └ temp [8][8][21]
    ⊞ data1:rsc (8x21)
```



```
ac_int<21> temp[XYSIZE][XYSIZE];
ac_int<21> tmp;
ac_int<31> dct_value;
```

19. Select the *temp:rsc* resource and set the **Resource Type** to a *single port RAM*.



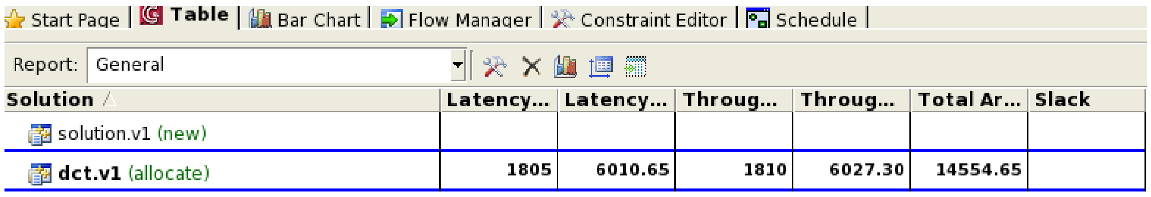
Resource Type: ram_nangate-45nm-singleport_beh.RAM
re_active: 0
we_active: 0
num_byte_enables: 1
Packing Mode: absolute
Block Size: 0
Interleave: 1

20. Click on **Schedule** in the Catapult **Task Bar**.

    There is no need to generate RTL before achieving the performance goals. RTL generation usually accounts for 50% of the design runtime. By scheduling the design we can measure the latency and throughput. The Gantt chart schedule view provides a way to analyze the performance bottlenecks and identify loops in the design which are potential candidates for optimizations such as pipelining and loop unrolling.

21. Go to the **Table View.**

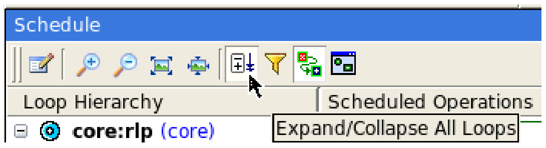
    Note that the initial throughput equals *1810* clock cycles



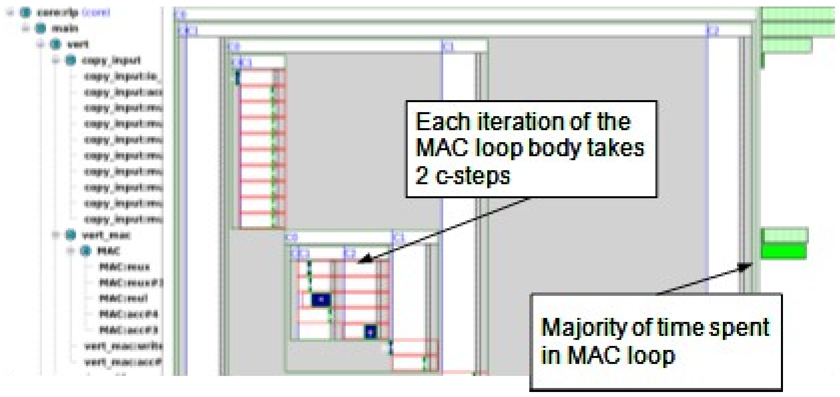
Solution	Latency...	Latency...	Throug...	Throug...	Total Ar...	Slack
solution.v1 (new)						
dct.v1 (allocate)	1805	6010.65	1810	6027.30	14554.65	

22. Go to the Gantt Chart view.

    Expand all loops in the design by clicking on the Down Arrow Icon in the tool bar.

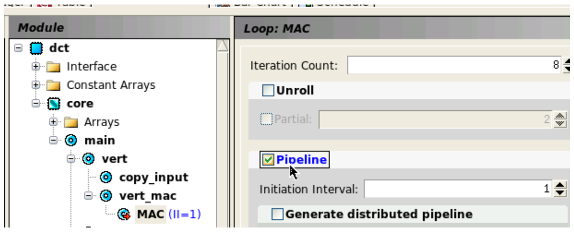


The green bars on the right side of the Gantt Chart show the runtime profile of the loops in the design. A large dark green bar for a loop indicates that more time is spent executing that loop. These loops should be considered as candidates for optimization if the desired performance goals have not been reached. The Gantt chart shows that the *MAC* loop, which is a sub-loop of the *vert* loop, is where the majority of time is spent executing the algorithm. Each iteration of the *MAC* loop body takes 2 c-steps to execute. Scroll down in the Gantt Chart and you'll also see that the *MAC* loop which is under the *horz* loop also takes a lot of time, but not as much time as the other once since it's loop body only has one c-step.


Each iteration of the MAC loop body takes 2 c-steps
Majority of time spent in MAC loop

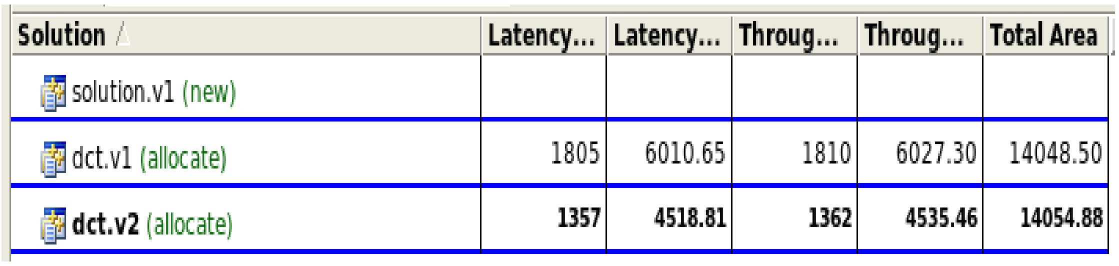
Pipeline the *vert_mac* loop

1. Click on **Architecture** in the Catapult **Task bar**. When optimizing a design for performance in Catapult it is usually best to start with pipelining constraints since they tend to cost less in terms of increased area.
2. Select the *MAC* loop under the *vert* loop and pipeline with II=1.



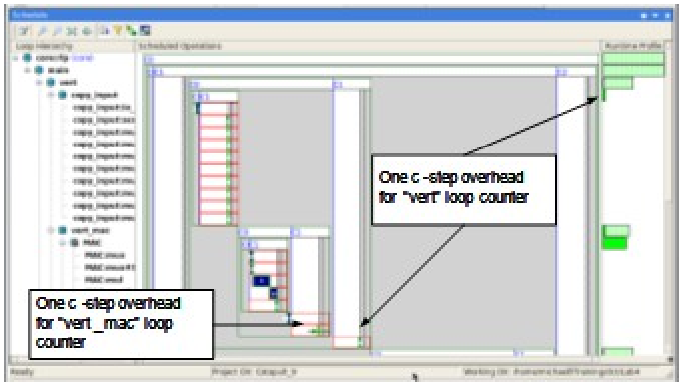
Iteration Count: 8
Pipeline: ✓
Initiation Interval: 1

3. Click on **Schedule** in the Catapult **Task Bar.**
4. Look at the table view and note the improvement in performance. The performance has improved a lot but still doesn't meet the initial goal of less than *1200* cycles.

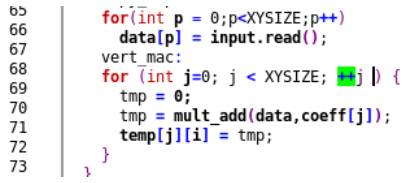


Solution	Latency...	Latency...	Throug...	Throug...	Total Area
solution.v1 (new)					
dct.v1 (allocate)	1805	6010.65	1810	6027.30	14048.50
dct.v2 (allocate)	**1357**	**4518.81**	**1362**	**4535.46**	**14054.88**

5. Look at the Gantt Chart and you can see that there is an extra c-step in both the *vert* and *vert_mac* loop that only performs an add operation "+". These adders are for computing the loop counters for the loop iterators.


One c-step overhead for "vert" loop counter
One c-step overhead for "vert_mac" loop counter

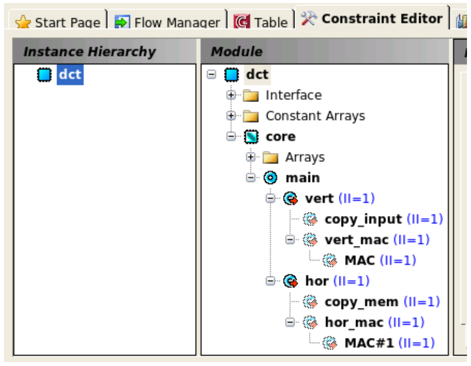
6. Double-click on the adder in C1 of the *vert_mac* loop to cross-probe back to the C++ and you will see that the loop index variable for the *vert_mac* loop is highlighted.



```
    for(int p = 0;p<XYSIZE;p++)
      data[p] = input.read();
    vert_mac:
    for (int j=0; j < XYSIZE; ++j ) {
      tmp = 0;
      tmp = mult_add(data,coeff[j]);
      temp[j][i] = tmp;
    }
  }
```

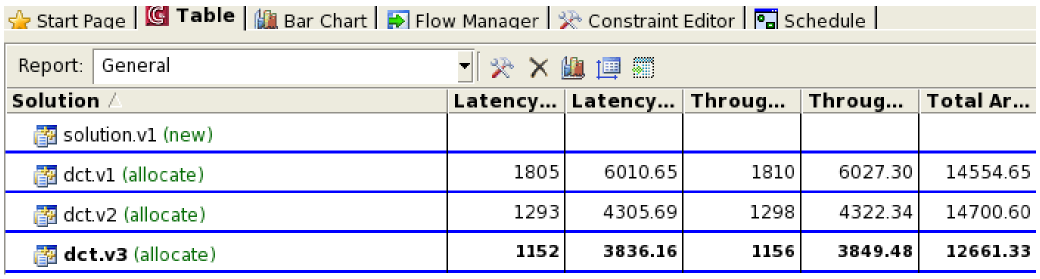
Pipeline the *vert* and *hor* loops

1. Click on **Architecture** in the Catapult **Task bar.**
2. Pipeline the *vert* and *hor* loops with II=1.



```
Start Page | Flow Manager | Table | Constraint Editor
Instance Hierarchy    Module
dct                   dct
                        Interface
                        Constant Arrays
                        core
                          Arrays
                          main
                            vert (II=1)
                              copy_input (II=1)
                              vert_mac (II=1)
                                MAC (II=1)
                            hor (II=1)
                              copy_mem (II=1)
                              hor_mac (II=1)
                                MAC#1 (II=1)
```

3. Click **Schedule** in the **Task bar** to schedule the design. You should see a throughput of *1156* cycles in the table view. The first performance target has been met.



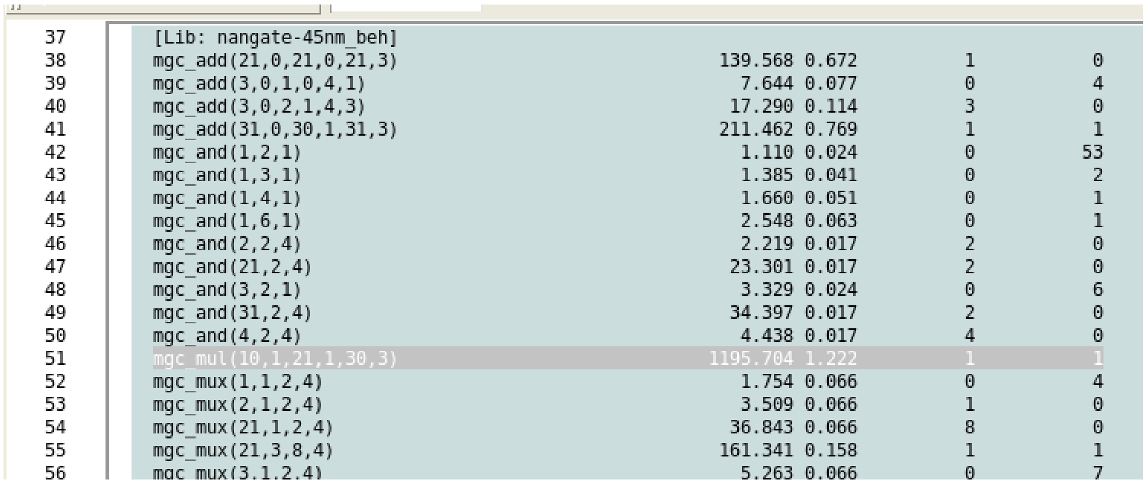
Start Page | **Table** | Bar Chart | Flow Manager | Constraint Editor | Schedule

Report: General

Solution	Latency...	Latency...	Throug...	Throug...	Total Ar...
solution.v1 (new)					
dct.v1 (allocate)	1805	6010.65	1810	6027.30	14554.65
dct.v2 (allocate)	1293	4305.69	1298	4322.34	14700.60
dct.v3 (allocate)	**1152**	**3836.16**	**1156**	**3849.48**	**12661.33**

4. Click **RTL** in the Task Bar to generate the design RTL.

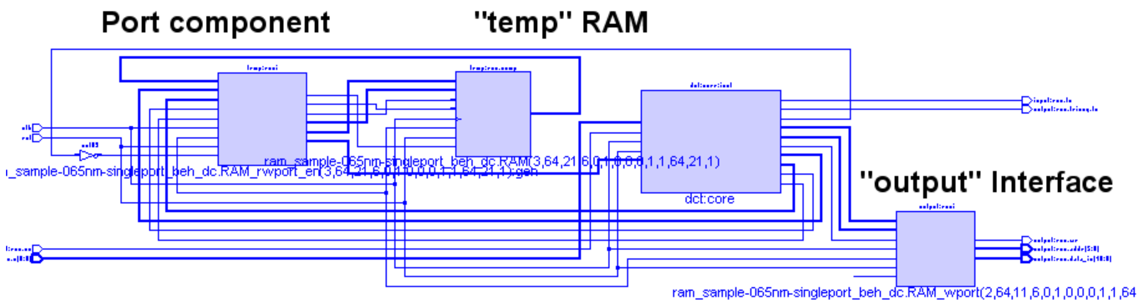
5. Open the RTL report in the Output Files folder and go to the Bill of Materials section. This will show that Catapult built the DCT using a single multiplier, which is indicated in the "Post Assign" column of the report. The **Post Alloc** column shows the number of resources that were used during scheduling and the **Post Assign** column shows the number of resources used after RTL generation.



```
[Lib: nangate-45nm_beh]
mgc_add(21,0,21,0,21,3)              139.568 0.672        1        0
mgc_add(3,0,1,0,4,1)                   7.644 0.077        0        4
mgc_add(3,0,2,1,4,3)                  17.290 0.114        3        0
mgc_add(31,0,30,1,31,3)              211.462 0.769        1        1
mgc_and(1,2,1)                         1.110 0.024        0       53
mgc_and(1,3,1)                         1.385 0.041        0        2
mgc_and(1,4,1)                         1.660 0.051        0        1
mgc_and(1,6,1)                         2.548 0.063        0        1
mgc_and(2,2,4)                         2.219 0.017        2        0
mgc_and(21,2,4)                       23.301 0.017        2        0
mgc_and(3,2,1)                         3.329 0.024        0        6
mgc_and(31,2,4)                       34.397 0.017        2        0
mgc_and(4,2,4)                         4.438 0.017        4        0
mgc_mul(10,1,21,1,30,3)             1195.704 1.222        1        1
mgc_mux(1,1,2,4)                       1.754 0.066        0        4
mgc_mux(2,1,2,4)                       3.509 0.066        1        0
mgc_mux(21,1,2,4)                     36.843 0.066        8        0
mgc_mux(21,3,8,4)                    161.341 0.158        1        1
mgc_mux(3,1,2,4)                       5.263 0.066        0        7
```

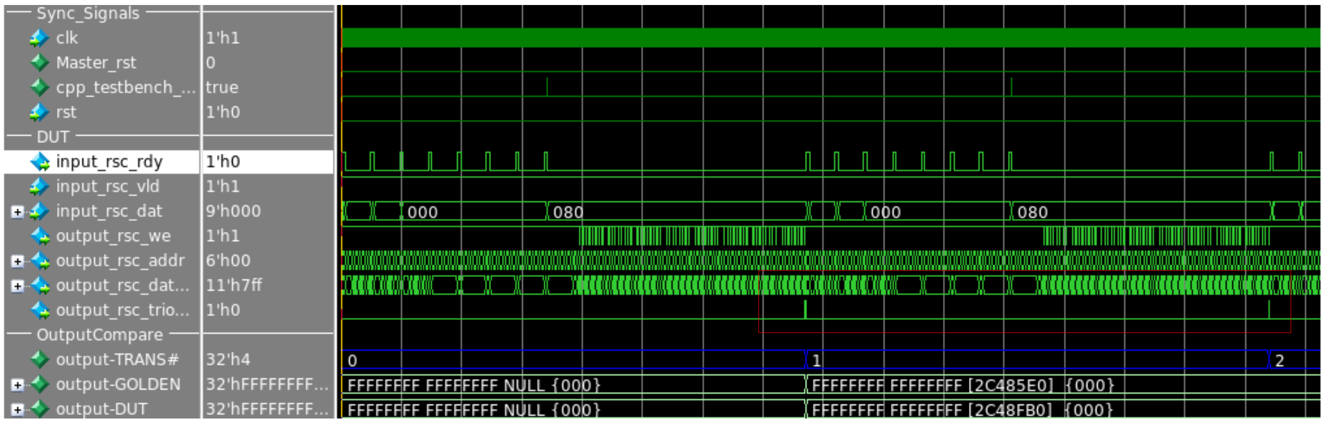
6. Open the RTL schematic.

   You will see that Catapult has synthesized an internal memory and port component for the *temp* array has also created a memory interface/port component for the *output* array interface. The *Port components* are simply interconnect modules that only contain wire interconnect.


Port component
"temp" RAM
"output" Interface
dct:core
ram_sample-065nm-singleport_beh_dc.RAM(3,64,21,6,0,1,0,0,0,1,1,64,21,1)
_sample-065nm-singleport_beh_dc.RAM_rwport_en(3,64,21,6,0,1,0,0,0,1,1,64,21,1)gen
ram_sample-065nm-singleport_beh_dc.RAM_wport(2,64,11,6,0,1,0,0,0,1,1,64

7. Go to the **Verification** Folder and launch SCVerify on the generated RTL. Run the simulation and verify the throughput.

   You can do this by measuring the time between "Transaction Done" signal pulses for the output memory write (*output_triosy_lz*). "Transaction Done" is a hardware verification signal added by Catapult to help synchronize the automated verification comparisons when an interface does not have an explicit synchronization signal. You should see the time between "Transaction Done"signals is 3852.938ns / 3.333ns/clock = 1156 clocks. Also observe that the reading of the input data is very intermittent. This can be seen by the amount of time that the *input_rsc_rdy* (read request) signal is high.

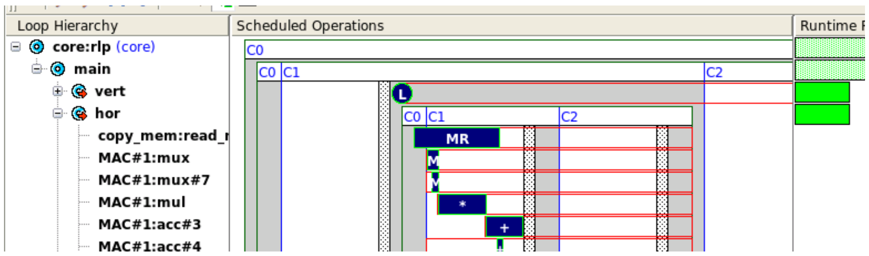


```
Sync_Signals
clk                 1'h1
Master_rst          0
cpp_testbench_...   true
rst                 1'h0
DUT
input_rsc_rdy       1'h0
input_rsc_vld       1'h1
input_rsc_dat       9'h000
output_rsc_we       1'h1
output_rsc_addr     6'h00
output_rsc_dat...   11'h7ff
output_rsc_trio...  1'h0
OutputCompare
output-TRANS#       32'h4
output-GOLDEN       32'hFFFFFFFF...
output-DUT          32'hFFFFFFFF...
```

000
080
000
080
0
1
2
FFFFFFFF FFFFFFFF NULL {000}
FFFFFFFF FFFFFFFF [2C485E0] {000}
FFFFFFFF FFFFFFFF NULL {000}
FFFFFFFF FFFFFFFF [2C48FB0] {000}

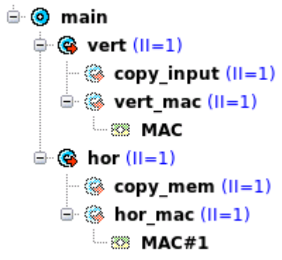
8. Click **Schedule** in the Task Bar and expand the Gantt chart.

   Pipelining the *vert* and *hor* loops has caused the sub-loops underneath them to be *flattened*. Although you can't see them, these loops are still *rolled*, and will take multiple cycles to execute based on the number of loop iterations. The *MAC* loop still takes 8 iterations to execute since it is *rolled*, which is why you still only see a single multiplier in the Gantt Chart. You can see from the dark green runtime profile bars that all the execution time of the algorithm is now spent in the *vert* and *hor* loops. In order to improve performance we need to start adding parallelism into the design. We can do this by loop unrolling.



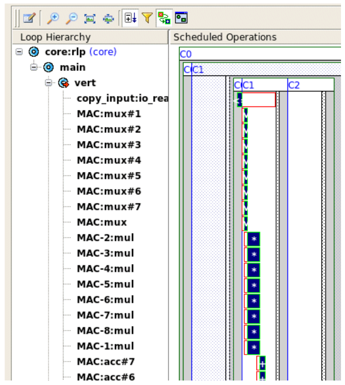
Unroll the MAC Loops

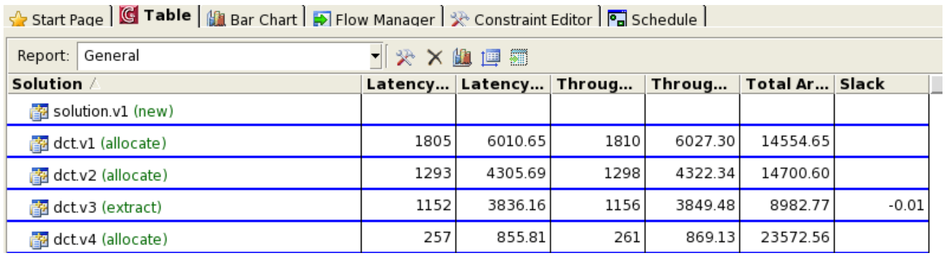
1. Go to **Architectural** Constraints and unroll both *MAC* loops fully.



```
main
  vert (II=1)
    copy_input (II=1)
    vert_mac (II=1)
      MAC
  hor (II=1)
    copy_mem (II=1)
    hor_mac (II=1)
      MAC#1
```

2. Re-schedule the design and look at the Gantt Chart. You will now see 8 multipliers scheduled in the same c-step for both the *vert* and *hor* loops. Look at the Table View. While the performance has improved greatly, it is still not close to the 128 cycle performance target. The reason for this is because the *copy_input* and *copy_mem* loops have to read all 8 input samples before the processing can begin. What is needed to achieve performance is that the reading of the data needs to be overlapped with the processing.

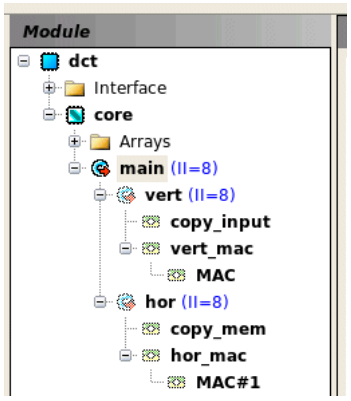




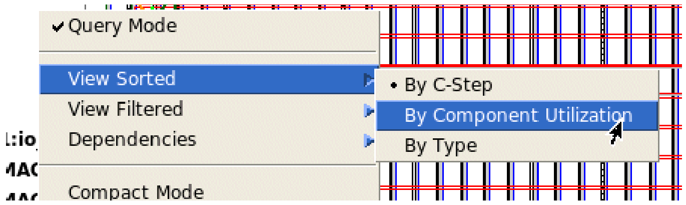
Solution	Latency...	Latency...	Throug...	Throug...	Total Ar...	Slack
solution.v1 (new)						
dct.v1 (allocate)	1805	6010.65	1810	6027.30	14554.65	
dct.v2 (allocate)	1293	4305.69	1298	4322.34	14700.60	
dct.v3 (extract)	1152	3836.16	1156	3849.48	8982.77	-0.01
dct.v4 (allocate)	257	855.81	261	869.13	23572.56	

Change the Iteration Interval to 8 on the Main Loop

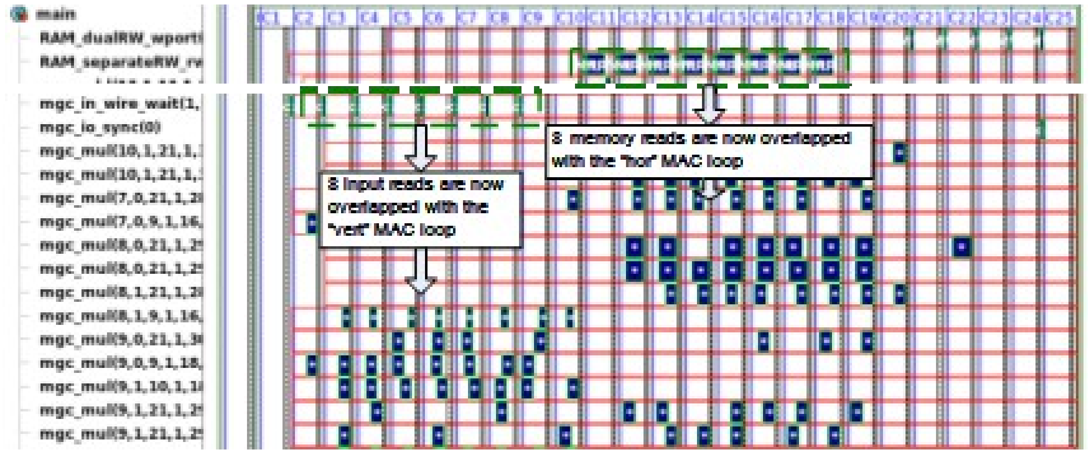
1. Go to **architectural** constraints and unroll all of the loops under the *vert* and *hor* loops, but leave the *vert* and *hor* loops "rolled".
2. Pipeline the *main* loop with II=8. The reason for II=8 is because the *copy_input* and *copy_mem* loops each read 8 samples from the input and memory respectively. Since these loops are now fully unrolled that will take 8 c-steps. Thus it is only possible to restart the *vert* and *hor* loops every 8 (II=8) clock cycles. We will be able to see this in the Gantt chart.



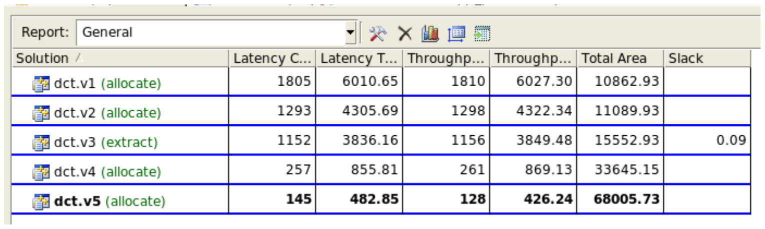
3. Reschedule the design and open the Gantt chart.
4. Run SCVerify and verify that the *128* cycle throughput is correct. You can do this by measuring the time between "Transaction Done" signal pulses for the output memory write (*output_triosy_lz*).The next lab will cover how we can increase performance beyond what we have achieved so far.
5. Right-click in the Gantt Chart and change the view to the Component Utilization view.



6. The *vert* and *hor* loops have been *flattened* into the *main* loop. However the *vert* and *hor* loops have been left rolled so they each still require 8 iterations to execute. Since they are sequential loops the *vert* loop must run till completion and then the *hor* loop can execute. Only one of the loops is ever active at the same time. Note: You will need to scroll down in the Gantt chart to see the multipliers.


8 memory reads are now overlapped with the "hor" MAC loop
8 input reads are now overlapped with the "vert" MAC loop

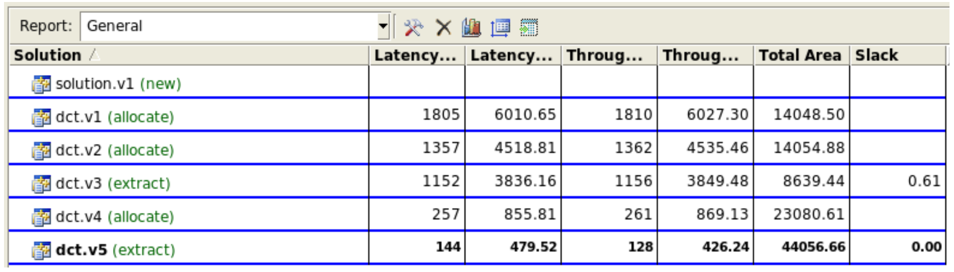
7. Look at the Table view and you should see that the throughput is now *128* cycles.



Report: General

Solution	Latency C...	Latency T...	Throughp...	Throughp...	Total Area	Slack
dct.v1 (allocate)	1805	6010.65	1810	6027.30	10862.93	
dct.v2 (allocate)	1293	4305.69	1298	4322.34	11089.93	
dct.v3 (extract)	1152	3836.16	1156	3849.48	15552.93	0.09
dct.v4 (allocate)	257	855.81	261	869.13	33645.15	
dct.v5 (allocate)	**145**	**482.85**	**128**	**426.24**	**68005.73**	

8. Generate **RTL**.
9. Look at the table view and compare the area between the two solutions which met the *1200* and *128* cycle performance targets. Usually increased performance costs more area.



Report: General

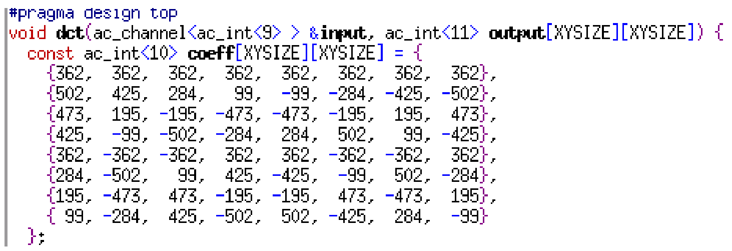
Solution	Latency...	Latency...	Throug...	Throug...	Total Area	Slack
solution.v1 (new)						
dct.v1 (allocate)	1805	6010.65	1810	6027.30	14048.50	
dct.v2 (allocate)	1357	4518.81	1362	4535.46	14054.88	
dct.v3 (extract)	1152	3836.16	1156	3849.48	8639.44	0.61
dct.v4 (allocate)	257	855.81	261	869.13	23080.61	
dct.v5 (extract)	**144**	**479.52**	**128**	**426.24**	**44056.66**	**0.00**

10. Open the RTL report and look at the BOM for number of multipliers. The increase in area is largely due to the increase in number of multipliers.
11. Run SCVerify verification and verify that the throughput equals 128 cycles

Optimize for Area

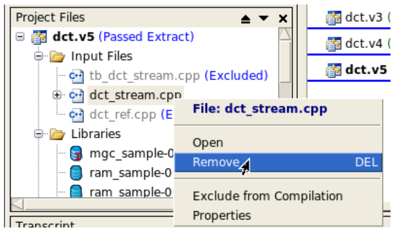
Once the performance goals of a design are met the next step is usually to see if area can be reduced. Once of the first things one should do in optimizing area is to analyze the architecture of the C++ design and determine if it is coded in an efficient manner. HLS is hardware design using C++ as the input language. It is still the designer's responsibility to ensure that the optimal architecture has been described in the C++.

1. Open the *dct_stream.cpp* file and look at the constant coefficient array. You can see that the coefficients are even symmetrical with a change in sign every other row. This symmetry can be exploited to reduce the number of multipliers used in the *mult_add* function by *folding* the algorithm.

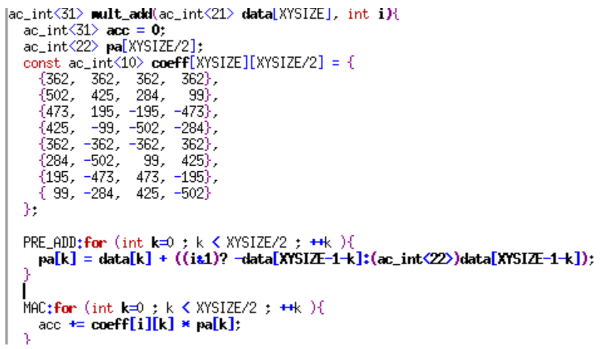


```
#pragma design top
void dct(ac_channel<ac_int<9> > &input, ac_int<11> output[XYSIZE][XYSIZE]) {
  const ac_int<10> coeff[XYSIZE][XYSIZE] = {
    {362,  362,  362,  362,  362,  362,  362,  362},
    {502,  425,  284,   99,  -99, -284, -425, -502},
    {473,  195, -195, -473, -473, -195,  195,  473},
    {425,  -99, -502, -284,  284,  502,   99, -425},
    {362, -362, -362,  362,  362, -362, -362,  362},
    {284, -502,   99,  425, -425,  -99,  502, -284},
    {195, -473,  473, -195, -195,  473, -473,  195},
    { 99, -284,  425, -502,  502, -425,  284,  -99}
  };
```

2. Right-click on the *dct_stream.cpp* file in the **Input Files** folder and select **Remove**.

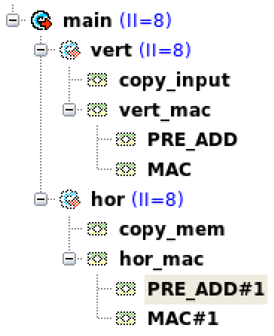


3. Double-click on the **Input Files** folder and add the file *dct_stream_sym.cpp* file.
4. Open the *dct_stream_sym.cpp* file and look at the *mult_add* function. You can see that it has been rewritten to take advantage of the coefficient symmetry by pre-adding/subtracting the data before the MAC. This reduces the number of MAC loop iterations by half, and thus half as many multipliers are required.



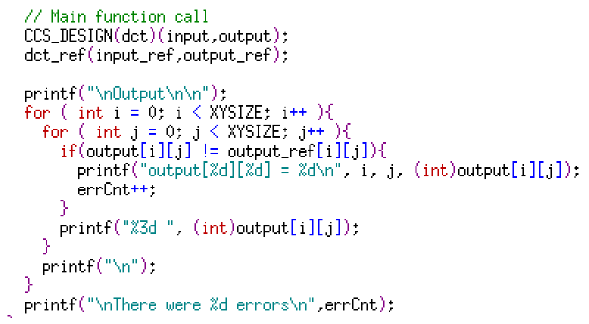
```
ac_int<31> mult_add(ac_int<21> data[XYSIZE], int i){
  ac_int<31> acc = 0;
  ac_int<22> pa[XYSIZE/2];
  const ac_int<10> coeff[XYSIZE][XYSIZE/2] = {
    {362,  362,  362,  362},
    {502,  425,  284,   99},
    {473,  195, -195, -473},
    {425,  -99, -502, -284},
    {362, -362, -362,  362},
    {284, -502,   99,  425},
    {195, -473,  473, -195},
    { 99, -284,  425, -502}
  };

  PRE_ADD:for (int k=0 ; k < XYSIZE/2 ; ++k ){
    pa[k] = data[k] + ((i&1)? -data[XYSIZE-1-k]:(ac_int<22>)data[XYSIZE-1-k]);
  }

  MAC:for (int k=0 ; k < XYSIZE/2 ; ++k ){
    acc += coeff[i][k] * pa[k];
  }
```

5. Go to **Architectural** Constraints and unroll all loops underneath the *vert* and *hor* loops (leave the *vert* and *hor* loops "rolled") and pipeline the *main* loop with *II=8*.
6. Click **Apply**.



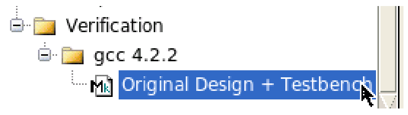
```
main (II=8)
  vert (II=8)
    copy_input
    vert_mac
      PRE_ADD
      MAC
  hor (II=8)
    copy_mem
    hor_mac
      PRE_ADD#1
      MAC#1
```

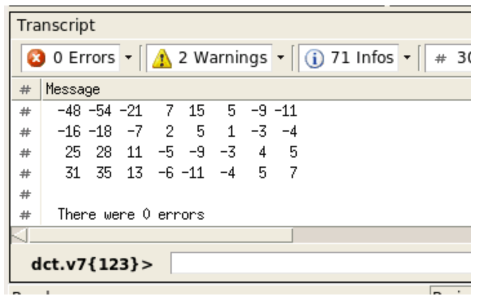
7. Before generating RTL it would be good to know that the architectural change that was made to the design hasn't change the functionality. The C++ testbench has been built to check the synthesizable DCT against a reference model. Open the *tb_dct_stream.cpp* file and you'll see that the outputs of the synthesizable design are always compared against the reference.



```
// Main function call
CCS_DESIGN(dct)(input,output);
dct_ref(input_ref,output_ref);

printf("\nOutput\n\n");
for ( int i = 0; i < XYSIZE; i++ ){
  for ( int j = 0; j < XYSIZE; j++ ){
    if(output[i][j] != output_ref[i][j]){
      printf("output[%d][%d] = %d\n", i, j, (int)output[i][j]);
      errCnt++;
    }
    printf("%3d ", (int)output[i][j]);
  }
  printf("\n");
}
printf("\nThere were %d errors\n",errCnt);
```

8. Go to the Verification Folder and execute the C++ design and testbench by double-clicking on the *Original Design + Testbench* Makefile. Look in the Catapult transcript and you will see the testbench output that indicates that the design is still functionally correct.



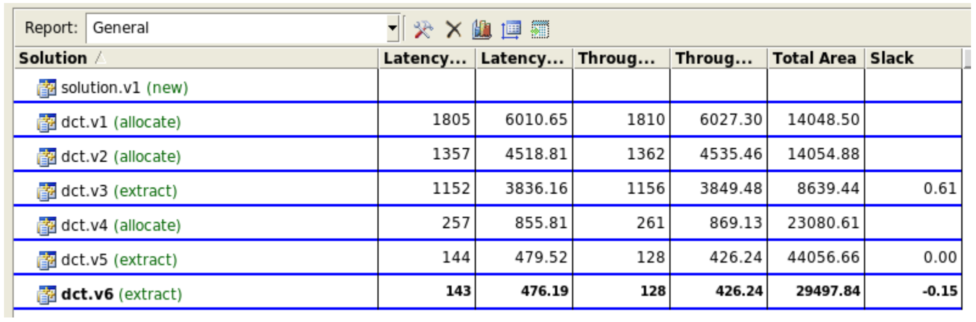


```
#    -48 -54 -21   7  15   5  -9 -11
#    -16 -18  -7   2   5   1  -3  -4
#     25  28  11  -5  -9  -3   4   5
#     31  35  13  -6 -11  -4   5   7
#
#    There were 0 errors
dct.v7{123}>
```

9. Generate **RTL**

10. Open the RTL report file and go to the BOM. How many multiplier are used?

11. Go to the Table view and note the large decrease in area.



Report: General

Solution	Latency...	Latency...	Throug...	Throug...	Total Area	Slack
solution.v1 (new)						
dct.v1 (allocate)	1805	6010.65	1810	6027.30	14048.50	
dct.v2 (allocate)	1357	4518.81	1362	4535.46	14054.88	
dct.v3 (extract)	1152	3836.16	1156	3849.48	8639.44	0.61
dct.v4 (allocate)	257	855.81	261	869.13	23080.61	
dct.v5 (extract)	144	479.52	128	426.24	44056.66	0.00
dct.v6 (extract)	**143**	**476.19**	**128**	**426.24**	**29497.84**	**-0.15**

DONE Lab4

Lab5 – Building a Multi-Block Hierarchical DCT for Increased Performance

In this lab you will learn how to:

- Build a multi-block hierarchical class-based design in Catapult
- Code the C++ for a shared memory architecture
- Use CCOREs to improve area and runtime
- Use the Catapult Bottom-up design flow

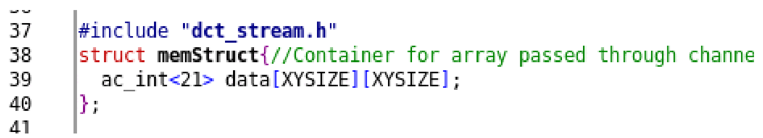
In Lab4 we were able to optimize the performance of the single-block DCT to achieve a best case throughput of 128 cycles. The performance bottleneck for that design architecture was caused by the "vert" and "hor" loops executing sequentially with a singleport RAM shared between the loops. In this lab Catapult "Hierarchy" will be used to synthesize the "vert" and "hor" loops as separate concurrent processes, allowing the resulting hardware blocks to run concurrently.

Design Goals

- Achieve a throughput of 64 clock cycles to compute the DCT.
- Reduce design area by using CCOREs to increase sharing

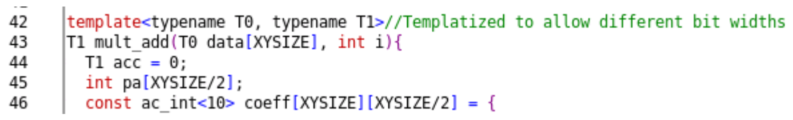
Optimize for Performance

1. Open a terminal and cd into the Lab5 directory.
2. Type "catapult" at the command prompt.
3. Go to the File Menu and select "Run Script". Select the "setup.tcl and click Open. This will add in the dct_stream.h, dct_ref.cpp, and tb_dct_stream.cpp design files The script also enables SCVerify, sets the **RTL Synthesis tool** to *OasysRTL*, **Vendor** to *Nangate*, and **Technology** to *045nm*, and enables singleport RAM, and sets the clock to 300MHz.
4. Open the dct_stream.h file and take a minute to see the general features of the design.
   - At the top of the design file you can see a struct, "memStruct", containing an array has been defined to be used for the required coding style for shared memories between blocks.



```
#include "dct_stream.h"
struct memStruct{//Container for array passed through channe
  ac_int<21> data[XYSIZE][XYSIZE];
};
```

   - The "mult_add" function has been templatized to allow instances with different bit widths. (If you are not familiar with C++ templates, they act like VHDL generics or Verilog parameters).



```
template<typename T0, typename T1>//Templatized to allow different bit widths
T1 mult_add(T0 data[XYSIZE], int i){
  T1 acc = 0;
  int pa[XYSIZE/2];
  const ac_int<10> coeff[XYSIZE][XYSIZE/2] = {
```

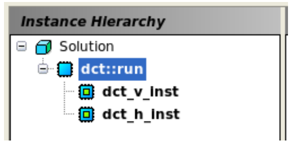
- The vertical and horizontal processing loops have been made into standalone classes "dct_v" and "dct_h". Note the #pragma hls_design interface indicating that the class should be synthesized as a design block.

```
class dct_h{
  public:
  dct_h(){}
#pragma hls_design interface
  void run(ac_channel<memStruct > &mem, ac_int<11> output[8][8]) {
```

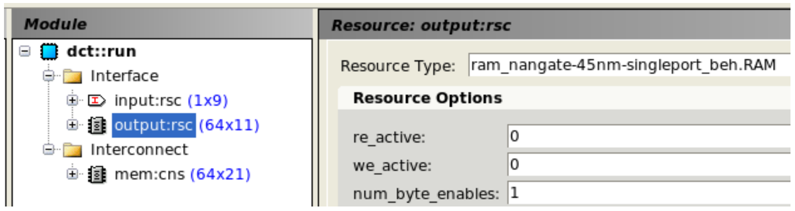
- The top-level class instantiates the "dct_v" and dct_h" classes and connects them using an ac_channel data member of type "memStruct".

```
#pragma hls_design top
class dct{
  ac_channel<memStruct > mem;//Static interconnect channel
  dct_v dct_v_inst;
  dct_h dct_h_inst;

  public: dct(){}//Required constructor
  //Pragma defines top-level interface
#pragma hls_design interface
  void CCS_BLOCK(run)(ac_channel<ac_int<9> > &input, ac_int<11> output[8][8]) {
    dct_v_inst.run(input,mem);
    dct_h_inst.run(mem,output);
  }
};
```

5. Go to Architectural Constraint. You should notice that the view is slightly different than the previous single block design views. You should now see the different design blocks in the Instance Hierarchy pane.

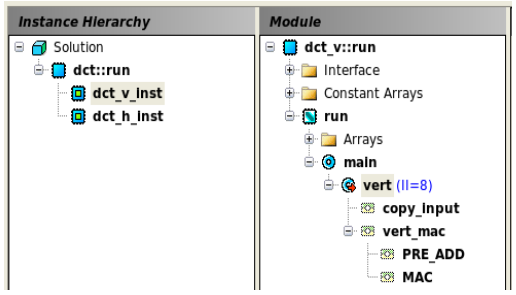


6. Click on the "dct" in the Instance Hierarchy view. Set the "output:rsc" resource type to singleport RAM. Expand the Interconnect folder and set the shared interconnect resource to singleport RAM and set the "Stage Replication" to 2. This will generate a ping-pong shared memory architecture between "dct_v" and "dct_h".

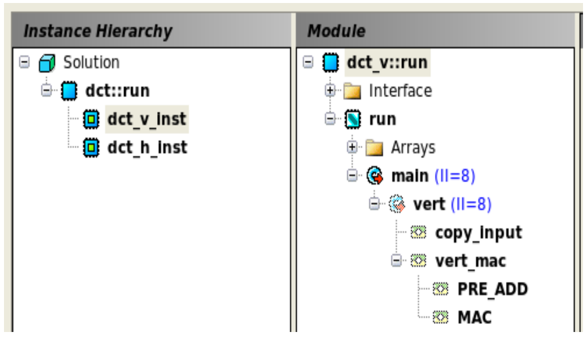


Resource Type: ram_nangate-45nm-singleport_beh.RAM
re_active: 0
we_active: 0
num_byte_enables: 1

7. Click on dct_v_inst in the Instance Hierarchy pane, unroll all the loops under the "vert" loop but leave the “vert” loop “rolled”, and pipeline the "main: loop with II=8.



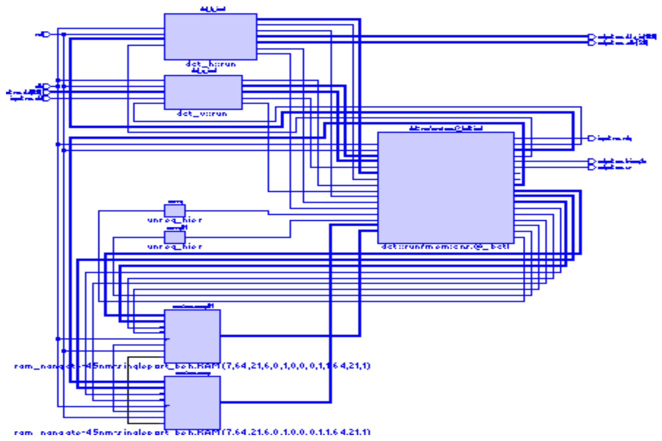
8. Click on dct_h_inst in the Instance Hierarchy pane, unroll all the loops under the "hor" loop but leave the “hor” loop “rolled”, and pipeline the "main: loop with II=8.



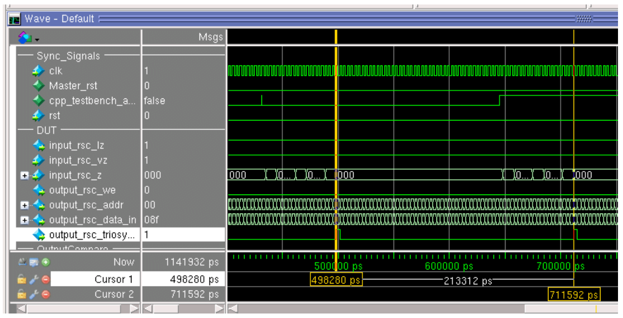
9. Generate RTL

10. Look at the Table view and you’ll see that the throughput is now 64 cycles. This is because the "dct_v" and "dct_h" functions can now run concurrently. The ping-pong shared memory between them provides enough elasticity so that neither block has to wait.

11. Open the RTL schematic, Right-click and un-check "Multipage Schematics" and you’ll see hierarchy blocks for "dct_v_inst" and "dct_h_inst" plus two RAMs that implement the ping-pong.


dct_h::run
dct_v::run
unreg_hier
unreg_hier
dct::run:mem:cns:@_bctl
ram_nangate-45nm-singleport_beh.RAM(7,64,21,6,0,1,0,0,0,1,1,6,4,21,1)
ram_nangate-45nm-singleport_beh.RAM(7,64,21,6,0,1,0,0,0,1,1,6,4,21,1)

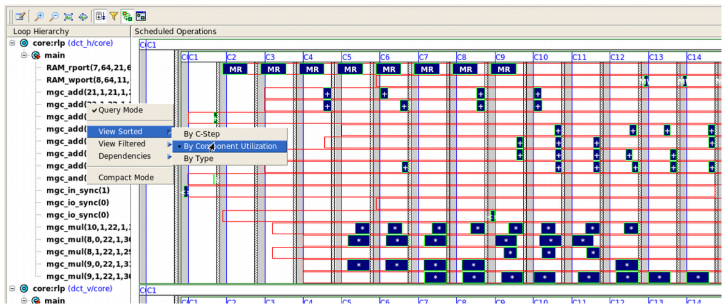
12. Launch SCVerify and verify that the throughput equals 64 cycles. You can do this by measuring the time between "Transaction Done" signal pulses for the output memory write (*output_triosy_lz*). Note that the input request for data (*input_rsc_vld*) goes high and stays high for the simulation. In other words the hardware never stops reading the input.



Optimize for Area Using CCOREs

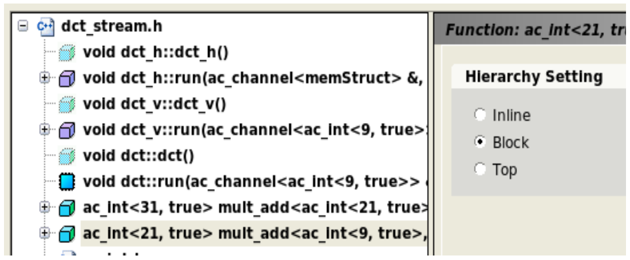
1. Now that the performance is met, it would be nice to see if the area can be optimized further. Go to the

Gantt chart schedule view and expand all the loops and view by component utilization.



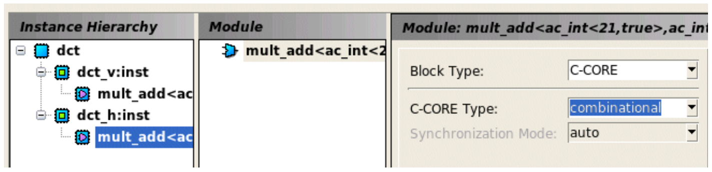
The schedule view shows that the design mostly consists of multipliers and adders and the schedule looks somewhat irregular. The reason for this is that by unrolling all of the loops inside the "vert" and "hor" loops we essentially created 64 multipliers and a whole bunch of adders. Then by pipelining with II=8 we allowed Catapult to re-share those multipliers and adders. The unrolling was required in order to get the IO reads to overlap the computation, but it introduced some irregularity in the design that makes it hard to re-share into an optimal hardware architecture. What is needed is a way to preserve the mult_add function so that it is easier for Catapult to share optimally. This can be done using the Catapult CCORE flow.

2. Click on "Hierarchy" in the Catapult Task Bar.
3. Set the Hierarchy Setting to "Block" for the two "mult_add" functions.


dct_stream.h
void dct_h::dct_h()
void dct_h::run(ac_channel<memStruct> &,
void dct_v::dct_v()
void dct_v::run(ac_channel<ac_int<9, true>
void dct::dct()
void dct::run(ac_channel<ac_int<9, true>>
ac_int<31, true> mult_add<ac_int<21, true>
ac_int<21, true> mult_add<ac_int<9, true>,
Function: ac_int<21, tr
Hierarchy Setting

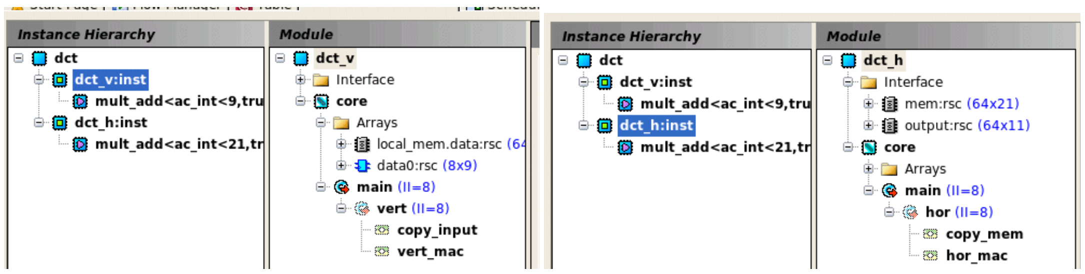
Hierarchy Setting: Inline [✓ Block] Top

4. Click on "Mapping" in the Catapult Task bar.
5. Select each "mult_add" in the Instance Hierarchy pane and set the Block Type to "CCORE" and set the C-CORE type to combinational.



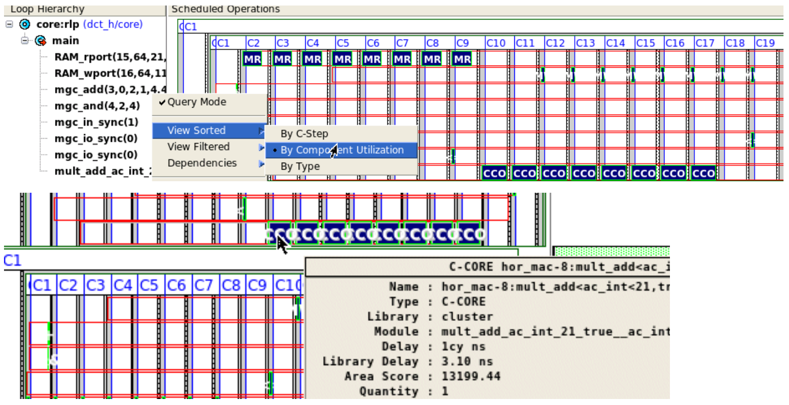
Block Type: C-CORE
C-CORE Type: combinational
Synchronization Mode: auto

6. Click on Architecture select dct_v instance hierarchy and unroll all the loops under the "vert" loop. Pipeline the main loop with II=8. Click on the dct_h instance hierarchy and repeat these steps.



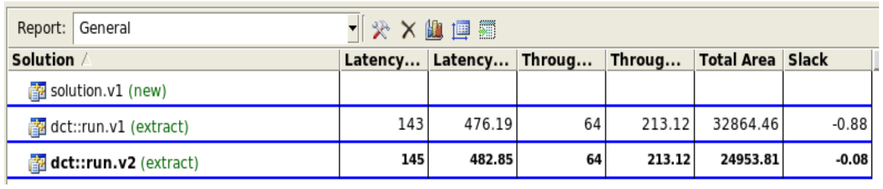
7. Click on "Schedule" in the Catapult Task bar.

8. Open the Gantt chart and view by Component Utilization. Note that the mult_add CCORE has been perfectly shared in both the dct_v and dct_h functions. You can see this visually because all the mult_add CCORE operations are shown in the same row. You can also hover over the mult_add CCORE and it will show the number of instances required by the scheduler.



```
C-CORE hor_mac-8:mult_add<ac_i
         Name : hor_mac-8:mult_add<ac_int<21,tr
         Type : C-CORE
      Library : cluster
       Module : mult_add_ac_int_21_true__ac_int
        Delay : 1cy ns
Library Delay : 3.10 ns
   Area Score : 13199.44
     Quantity : 1
```

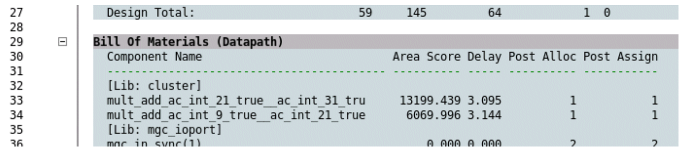
9. Click on RTL in the Catapult Task bar to generate the RTL for the design.
10. Look at the table view and note the big reduction in area. Building the "mult_add" into CCOREs preserved the regularity in the design and allowed better sharing of larger resources.



Report: General

Solution	Latency...	Latency...	Throug...	Throug...	Total Area	Slack
solution.v1 (new)						
dct::run.v1 (extract)	143	476.19	64	213.12	32864.46	-0.88
dct::run.v2 (extract)	**145**	**482.85**	**64**	**213.12**	**24953.81**	**-0.08**

11. Go to the Output Files folder and open the rtl.rpt file. Look at the BOM and find the MAC CCOREs. You should see that the entire design was implemented using two of them.

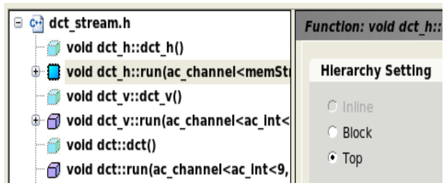


```
  Design Total:                                 59     145         64           1  0

Bill Of Materials (Datapath)
  Component Name                              Area Score Delay Post Alloc Post Assign
  ------------------------------------------ --------- ----- ---------- -----------
  [Lib: cluster]
  mult_add_ac_int_21_true__ac_int_31_tru     13199.439 3.095          1           1
  mult_add_ac_int_9_true__ac_int_21_true      6069.996 3.144          1           1
  [Lib: mgc_ioport]
  mgc_in_sync(1)                                 0.000 0.000          2           2
```

Using the Bottom-up Design Flow

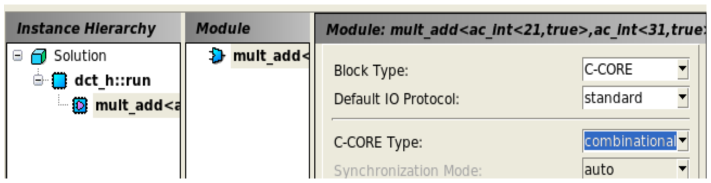
Up until this point all the lab designs have been built in a top-down fashion where the entire design is re-synthesized any time a code or constraint change is made. This is true for multi-block hierarchical designs like the DCT from the previous section even if one of the blocks remains unchanged. Bottom-up design allows designer(s) to synthesize the design blocks individually and assemble the top-level design as a final step.

1. Click on the Hierarchy button in the Task Bar and set "dct_h::run" to be the top-level design.



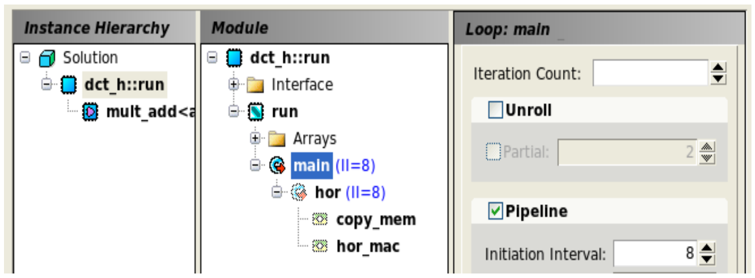
Inline Block [✓ Top]

2. Click on Mapping in the Task Bar, select the mult_add design block, and set the Block Type to C-CORE and the C-CORE type to combinational.



Instance Hierarchy: Solution / dct_h::run / mult_add<a
Module: mult_add<
Module: mult_add<ac_int<21,true>,ac_int<31,true:
Block Type: C-CORE
Default IO Protocol: standard
C-CORE Type: combinational
Synchronization Mode: auto

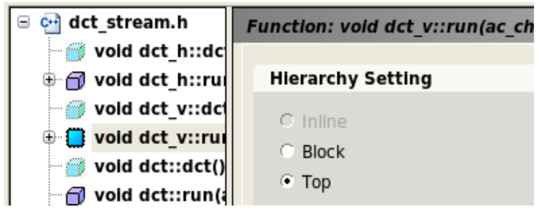
3. Go to Architectural constraints and unroll all loops under the "hor" loop and pipeline the Main loop with II=8.



Instance Hierarchy: Solution / dct_h::run / mult_add<a
Module: dct_h::run / Interface / run / Arrays / main (II=8) / hor (II=8) / copy_mem / hor_mac
Loop: main
Iteration Count:
Unroll
Partial: 2
Pipeline ✓
Initiation Interval: 8

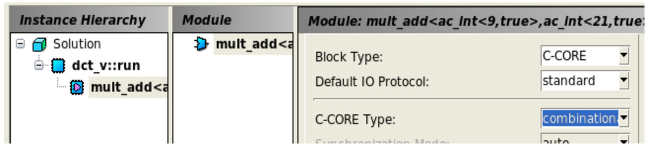
4. Generate RTL

5. Click on Hierarchy in the Task Bar and set "dct_v::run" as the top-level design.



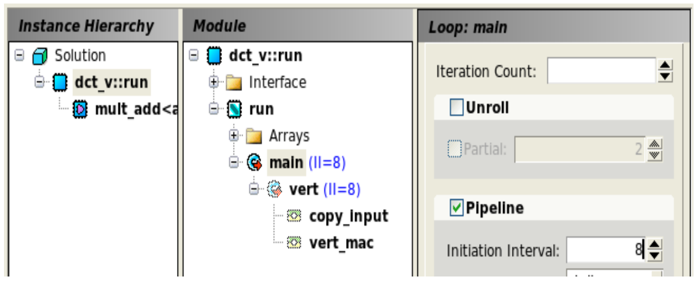
dct_stream.h
void dct_h::dc
void dct_h::ru
void dct_v::dc
void dct_v::ru
void dct::dct()
void dct::run(a
Function: void dct_v::run(ac_ch
Hierarchy Setting
Inline
Block
[✓ Top]

6. Click on Mapping in the Task Bar select the mult_add design block, and set the Block Type to C-CORE and the C-CORE type to combinational.



Instance Hierarchy: Solution / dct_v::run / mult_add<a
Module: mult_add<a
Module: mult_add<ac_int<9,true>,ac_int<21,true:
Block Type: C-CORE
Default IO Protocol: standard
C-CORE Type: combination
Synchronization Mode: auto

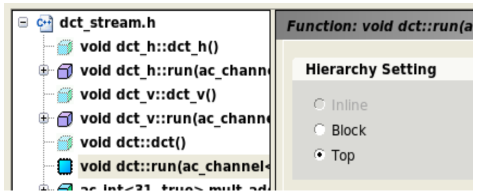
7. Click on Architecture in the Task Bar and unroll all loops under the "hor" loop and pipeline the Main loop with II=8.



Instance Hierarchy: Solution / dct_v::run / mult_add<a
Module: dct_v::run / Interface / run / Arrays / main (II=8) / vert (II=8) / copy_input / vert_mac
Loop: main
Iteration Count:
Unroll
Partial: 2
Pipeline ✓
Initiation Interval: 8

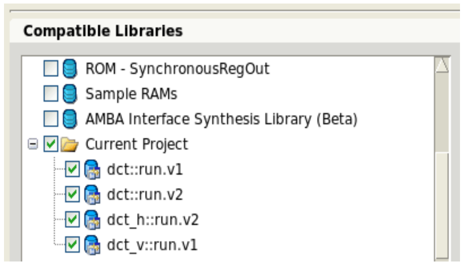
8. Generate RTL

9. Click on Hierarchy in the Task Bar and set "dct::run" as the top-level design.

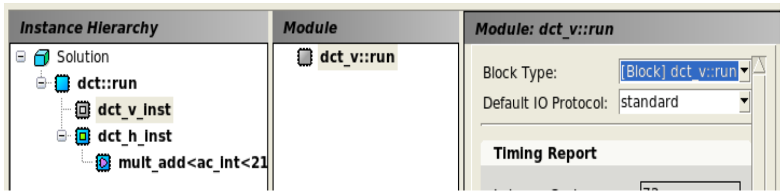

dct_stream.h
void dct_h::dct_h()
void dct_h::run(ac_chann
void dct_v::dct_v()
void dct_v::run(ac_chann
void dct::dct()
void dct::run(ac_channel<

Function: void dct::run(a
Hierarchy Setting
Inline
Block
[✓ Top]

10. Click on Libraries in the Task Bar and scroll to the bottom of the Compatible Libraries pane. You will see that Catapult has created and selected library components for all of the design blocks that have been synthesized so far.


Compatible Libraries
☐ ROM - SynchronousRegOut
☐ Sample RAMs
☐ AMBA Interface Synthesis Library (Beta)
✓ Current Project
✓ dct::run.v1
✓ dct::run.v2
✓ dct_h::run.v2
✓ dct_v::run.v1

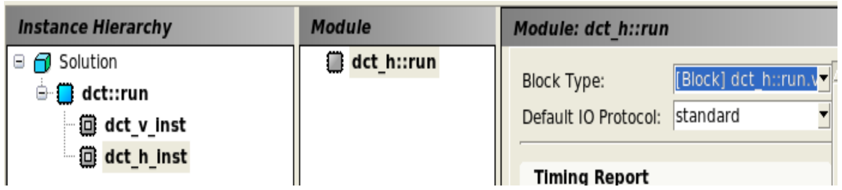
11. Click on Mapping in the Task Bar
12. Select the "dct_v_inst" block in the Instance Hierarchy pane and set the Block type to [Block] dct_v::run.v1. This will map the "dct_v::run" library component that we synthesized previously. This is indicated by a gray chip icon.


Instance Hierarchy
Solution
dct::run
dct_v_inst
dct_h_inst
mult_add<ac_int<21

Module
dct_v::run

Module: dct_v::run
Block Type: [Block] dct_v::run
Default IO Protocol: standard
Timing Report

13. Select the "dct_h_inst" block in the Instance Hierarchy pane and set the Block type to [Block] dct_h::run.v1. This will map the "dct_h" library component that we synthesized previously.


Instance Hierarchy
Solution
dct::run
dct_v_inst
dct_h_inst

Module
dct_h::run

Module: dct_h::run
Block Type: [Block] dct_h::run.v
Default IO Protocol: standard
Timing Report

14. Click on Architecture in the Task bar and select the "dct_v" block in the Instance Hierarchy pane. Note that there are no loops to constrain since this is block has already been synthesized.
15. Generate RTL. This will stitch in all of the bottom-up blocks and generate the interconnect memory architecture.
16. Open the RTL schematic. Note that the "dct_v" and "dct_h" blocks are grayed out indicating that these are "bottom-up" blocks.

DONE Lab5