SIEMENS EDA

Catapult Untimed C++ Training Labs

July 2020

SIEMENS

Contents

Introduction

Each one of the Catapult labs is designed to review the content covered in the training sections leading up to the lab. The labs are designed so that each lab builds on the previous labs and progresses towards more complicated HLS topics.

Setup:

- Open a terminal. CD to your working directory.
- Copy the labs into your local working directory:

```
cp –rf $MGC_HOME/shared/training/basic/cxx
```

Lab 1: Compiling, Debugging, and Executing Design Using AC Data-types

Overview

In this lab you will learn how to:

- Use the g++ that ships with Catapult
- Add include paths to the Catapult install tree
- Debug typical errors with algorithmic data types
- Understand left shift behavior of AC data-types

Steps

1. Open a terminal and cd into ‘lab1’ directory.
2. Open the test_chan_assert.cpp file for viewing. This is a simple design that uses ac_int and ac_fixed data types along with ac_channel to illustrate some common behaviors as well as errors.
3. At the terminal prompt type “make tb0”. This will compile and execute the tb_pod_err.cpp and test_chan_assert.cpp files
4. You get an error during compilation on line 36 of tb_pod_err.cpp on the ac_fixed<3,3,false,AC_RND,AC_SAT> variable

```
tb_pod_err.cpp: In function ‘int main()’:
tb_pod_err.cpp:36:47: error: cannot pass objects of non-trivially-copyable type ‘class ac_fixed<3, 3, false, (ac_q_mode)1u, (ac_o_mode)1u>’ through ‘...’
     printf("sat_behavior = %3d, ",sat_behavior);
```

5. Edit the tb_pod_err.cpp file and look at line 36.

```
printf("sat_behavior = %3d, ",sat_behavior);
```

You will see that the printf function is trying to directly print an ac_fixed data type. This will always generate an error since there is no implicit conversion from ac_fixed back to integer. Note that all the other ac data-type variables are using the “.to_uint()” helper function, but it is missing on the “sat_behavior” variable. This has been fixed in the tb.cpp file, line 27 which is used in the next example.

6. At the terminal prompt type “make tb1”. This will compile and execute the tb.cpp and test_chan_assert.cpp files.

7. The previous compilation error has been fixed but there is now a runtime error from reading an empty channel.

```
./tb

Read from empty channel
terminate called after throwing an instance of 'char const*'
make: *** [tb1] Aborted (core dumped)
```

8. If you have an IDE (Integrated Debugger) use it to find the channel that is asserting. Otherwise we will use the gdb debugger that ships with Catapult. NOTE: gdb is very primitive, Eclipse and Microsoft debuggers are much better.

9. At the terminal prompt type “make tb1_debug”. This will run the design in gdb

10. At the gdb prompt type “run” to execute the design.

```
/home/michaelf/.gdbinit:1: Error in sourced command file:
Python scripting is not supported in this copy of GDB.
Reading symbols from tb...done.
(gdb) run
```

11. After gdb aborts on the “Read from empty channel” type up at the gdb prompt to walk up the call stack. You will need to type up 7 times to get to the channel read that is causing the assert. Line 23 of test_chan_assert.cpp is where the empty channel is being read.

```
(gdb) run
Starting program: /var/tmp/tb
Assert in file /wv/hlsb/CATAPULT/10.5a/PRODUCTION/aol/Mgc_home/shared/include/ac_channel.h:250 Read from empty channel
tb: /wv/hlsb/CATAPULT/10.5a/PRODUCTION/aol/Mgc_home/shared/include/ac_channel.h:179: static void ac_channel<T>::ac_assert(bool, const char*, int, const ac_channel_exception::code&) [with T =
ac_int<4, false>]: Assertion `0' failed.

Program received signal SIGABRT, Aborted.
0x0000003113e32495 in raise () from /lib64/libc.so.6
(gdb) up
#1  0x0000003113e33c75 in abort () from /lib64/libc.so.6
(gdb) up
#2  0x0000003113e2b60e in __assert_fail_base () from /lib64/libc.so.6
(gdb) up
#3  0x0000003113e2b6d0 in __assert_fail () from /lib64/libc.so.6
(gdb) up
#4  0x0000000000401c84 in ac_channel<ac_int<4, false> >::ac_assert (condition=false, file=0x403288 "/wv/hlsb/CATAPULT/10.5a/PRODUCTION/aol/Mgc_home/shared/include/ac_channel.h", line=250,
    code=@0x7fffffffd21c: ac_channel_exception::read_from_empty_channel) at /wv/hlsb/CATAPULT/10.5a/PRODUCTION/aol/Mgc_home/shared/include/ac_channel.h:179
179             assert(0);
(gdb) up
#5  0x0000000000401880 in ac_channel<ac_int<4, false> >::fifo::read (this=0x7fffffffd2f0) at /wv/hlsb/CATAPULT/10.5a/PRODUCTION/aol/Mgc_home/shared/include/ac_channel.h:250
250             AC_CHANNEL_ASSERT(size(), ac_channel_exception::read_from_empty_channel);
(gdb) up
#6  0x00000000004014d8 in ac_channel<ac_int<4, false> >::read (this=0x7fffffffd2f0) at /wv/hlsb/CATAPULT/10.5a/PRODUCTION/aol/Mgc_home/shared/include/ac_channel.h:116
116         T read() { return chan.read(); }
(gdb) up
#7  0x0000000000402cf7 in test (data0=..., data1=..., chan_in=..., wrap_behavior=..., sat_behavior=..., shift_behavior=..., chan_out=...) at test_chan_assert.cpp:23
23          chan_out.write(chan_in.read());      // will assert if read of empty channel is attempted
(gdb)
```

12. Type ‘quit’ to exit gdb

13. Edit test_chan_assert.cpp and look at line 23. Note that the *.available(1) method is not being used to prevent the channel from being read when empty.

```
chan_out.write(chan_in.read());
```

14. Edit test_shift_loss.cpp. Look at the channel read and note that the *.available(1) method has been used to prevent reading an empty channel

```
if(chan_in.available(1))//Prevent reading empty channel
  chan_out.write(chan_in.read());
```

15. At the terminal prompt type “make tb2”. This will compile and execute the tb.cpp and test_shift_loss.cpp files. The design will now execute successfully and print to the terminal

```
Input =  0 wrap_behavior =  0, sat_behavior =  0, shift_behavior =  0
Input =  1 wrap_behavior =  1, sat_behavior =  1, shift_behavior =  2
Input =  2 wrap_behavior =  2, sat_behavior =  2, shift_behavior =  4
Input =  3 wrap_behavior =  3, sat_behavior =  3, shift_behavior =  6
Input =  4 wrap_behavior =  4, sat_behavior =  4, shift_behavior =  8
Input =  5 wrap_behavior =  5, sat_behavior =  5, shift_behavior = 10
Input =  6 wrap_behavior =  6, sat_behavior =  6, shift_behavior = 12
Input =  7 wrap_behavior =  7, sat_behavior =  7, shift_behavior = 14
Input =  8 wrap_behavior =  0, sat_behavior =  7, shift_behavior =  0
Input =  9 wrap_behavior =  1, sat_behavior =  7, shift_behavior =  2
Input = 10 wrap_behavior =  2, sat_behavior =  7, shift_behavior =  4
Input = 11 wrap_behavior =  3, sat_behavior =  7, shift_behavior =  6
Input = 12 wrap_behavior =  4, sat_behavior =  7, shift_behavior =  8
Input = 13 wrap_behavior =  5, sat_behavior =  7, shift_behavior = 10
Input = 14 wrap_behavior =  6, sat_behavior =  7, shift_behavior = 12
Input = 15 wrap_behavior =  7, sat_behavior =  7, shift_behavior = 14

Channel data =  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

```
void test(ac_int<4,false> data0,
          ac_fixed<5,5,false> data1,
          ac_channel<ac_int<4,false> > &chan_in,
          ac_int<3,false> &wrap_behavior,
          ac_fixed<3,3,false,AC_RND,AC_SAT> &sat_behavior,
          ac_int<5,false> &shift_behavior,
          ac_channel<ac_int<4,false> > &chan_out){


  wrap_behavior = data0;//will wrap for values > 7
  sat_behavior = data1;//will clamp to 7 for values > 7
  shift_behavior = data0 << 1;// will lose MSB after shift

  if(chan_in.available(1))//Prevent reading empty channel
    chan_out.write(chan_in.read());
}
```

16. Look at the testbench output.

- The input ‘data0’ is an ac_int<4,false> 4 bits unsigned and counts from 0 to 15

- ‘wrap_behavior’ is an ac_int<3,false> 3 bits unsigned and copies the input. You can see how it “wraps” at 7 back to 0. Just like a wire or register in Verilog

- ‘sat_behavior’ is ac_fixed<3,3,false,AC_RND,AC_SAT> 3 bits unsigned with saturation enabled. Note how it clamps the data at 7.

- ‘shift_behavior’ gets the input left shifted by 1, which is the same as multiplying by 2. However, note that it only goes to 14 and then back to 0. This is because data0 is an ac_int<4,false> 4 bit unsigned integer. Left shifting will cause the MSB to be lost.

17. At the terminal prompt type “make tb3”. This will compile and execute the tb.cpp and test.cpp files. Edit test.cpp and see that the left shift problem has been fixed by casting the data0 variable to a ac_int<5,false>.

```
shift_behavior = (ac_int<5,false>)data0 << 1;//cast to 5 bits to keep MSB

if(chan_in.available(1))//Prevent reading empty channel
  chan_out.write(chan_in.read());
```

DONE LAB1

Lab2: Catapult GUI, Constraints, and Analysis

Overview

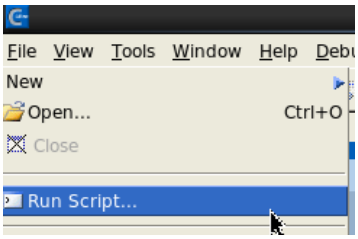
In this lab you will learn how to:

- Synthesize a design in Catapult using scripts and the Catapult GUI
- Map an ac_channel variable to a hardware interface component a.k.a “Resource”
- Analyze the design using the Catapult Gantt Chart and Catapult Design Analyzer to understand how the C++ has been synthesized to hardware
- Run automatic verification of the C++ and the RTL
- Enable pipeline flushing

Steps

1. From a terminal cd to the ‘lab2’ directory and launch Catapult by typing “catapult” at the terminal prompt.

2. Go to the Catapult File menu and select “Run script”



3. Select the ‘initial.tcl’ file and click Open.

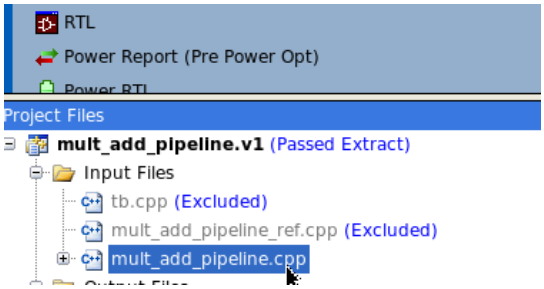
   This will add the input files, enable the verification flow, and synthesize the design. If you see the following error, enter the following command in Catapult: *’options set Input/TargetPlatform x86_64*’ after ‘*solution options defaults*‘ in the tcl directive.

This error shows up when the OS does not have 32 bit support.

```
cannot open source file "gnu/stubs-32.h"
Compilation aborted
```

4. There are 3 files for this design, tb.cpp and mult_add_pipeline_ref.cpp which are part of the testbench, and mult_add_pipeline.cpp which is the synthesizable design.

5. Double-click on the mult_add_pipeline.cpp file in the project files folder to open in the text editor.



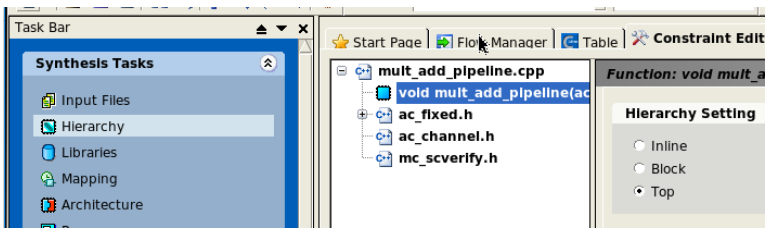
The design has ac_channel streaming interfaces for a,b, and c inputs and the result output. The design does a multiply-add of a, b, and c and multiplies by gain if gain_adjust is set to true.

```
#pragma hls_design top
void CCS_BLOCK(mult_add_pipeline)(ac_channel<ac_int<11,false> > &a,
                                  ac_channel<ac_int<14,false> > &b,
                                  ac_channel<ac_int<25,false> > &c,
                                  ac_fixed<10,2,true> gain, bool gain_adjust,
                                  ac_channel<ac_int<30,false> > &result){
  ac_int<25,false> product = a.read() * b.read();
  ac_int<26,false> sum = product + c.read();

  if(gain_adjust)
    sum = ((ac_fixed<26,26,false>)(sum*gain)).to_uint();
  result.write((ac_int<30,false> )sum<< 4);//NOTE: cast sum to ac_int<30,false
}
```

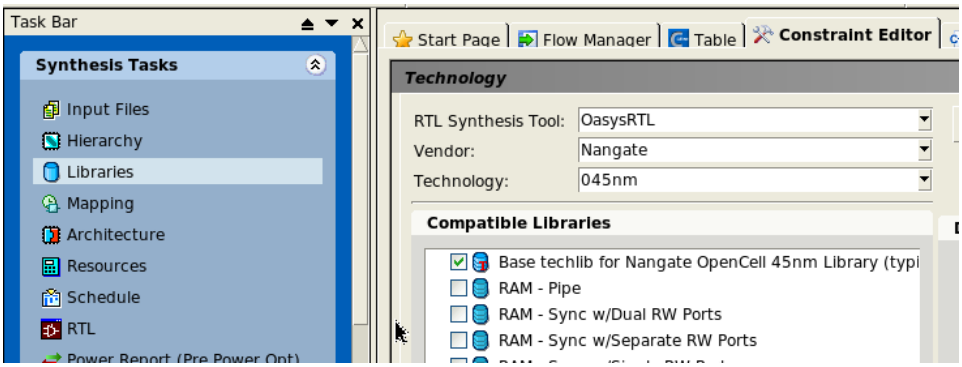
6. Click on the Hierarchy Button in the Task Bar.

Note that mult_add_pipeline has been set to the top-level design. Double-click on mult_add_pipeline to cross-probe to the C++.



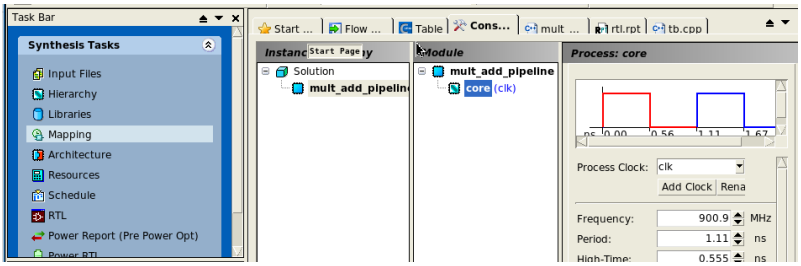
7. Click on Libraries in the Task Bar.

These labs use the Nangate 45nm sample libraries and the OasysRTL flow. In a production design this is where you would select your technology and synthesis flow.



8. Click on Mapping in the Task Bar.

   This is where the clock frequency constraints are set as well as other constraints covered later. Select the “core” icon and note that the clock frequency is set to 900MHz.



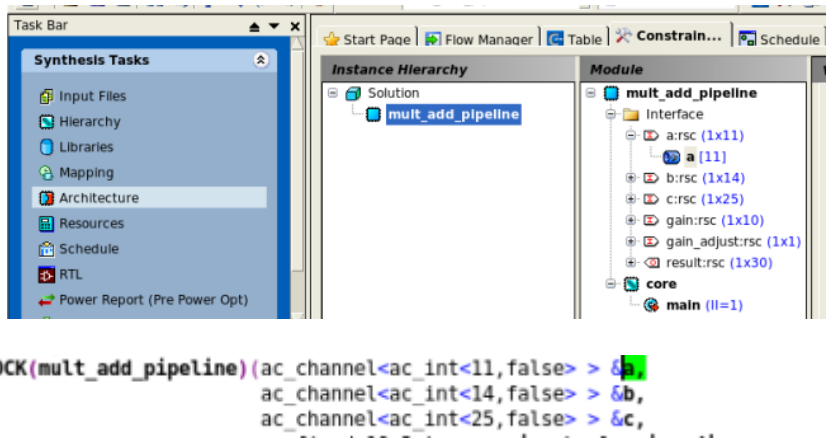
9. Click on Architecture in the Task Bar.

   This opens the Architectural constraints dialog. This is where design constraints are applied.

   a. Expand the Interface folder and note the interfaces that have been created for the C++ design.

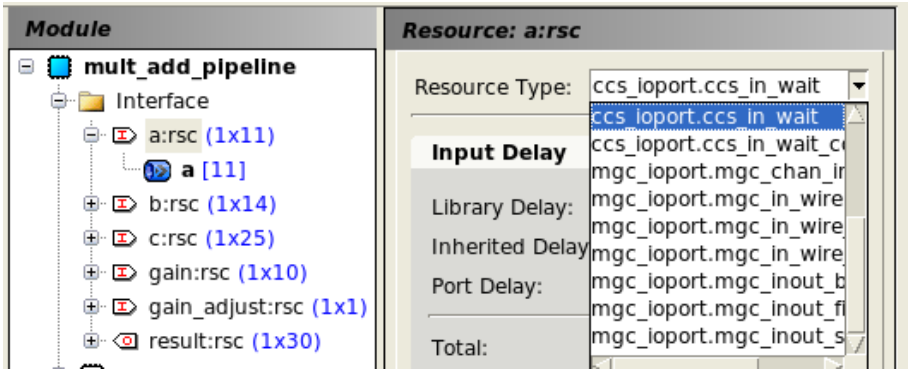
      Note that the bit-widths are shown along with the port direction.

   b. Expand the a:rsc interface and double-click on the “a” variable to go back to the source. Note how the interface is highlighted in the source editor.



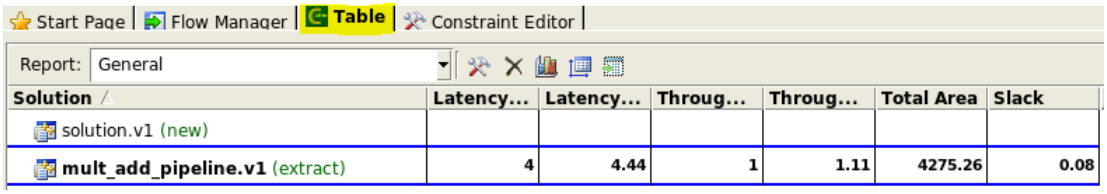
10. Select the *a:rsc* interface resource

    Note that it has been mapped to a *ccs_in_wait* interface which is the default for ac_channel interfaces. Other interfaces are selectable from the drop-down menu.



11. Click through the other resources under the interface folder. “a”, “b”, “c”, and “result” are all mapped to *_wait interfaces.

12. Click on the Table View Tab and note that Catapult reports a latency of 4 and a throughput of 1 for this design.

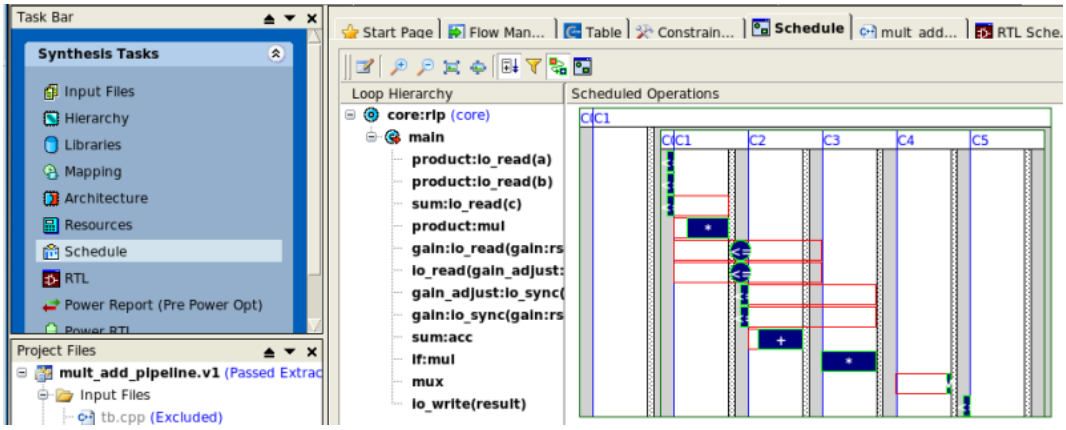


Solution	Latency...	Latency...	Throug...	Throug...	Total Area	Slack
solution.v1 (new)						
mult_add_pipeline.v1 (extract)	4	4.44	1	1.11	4275.26	0.08

13. Click on Schedule in the Task Bar. This will display the Catapult Gantt Chart which provides a data-flow view of the schedule design.

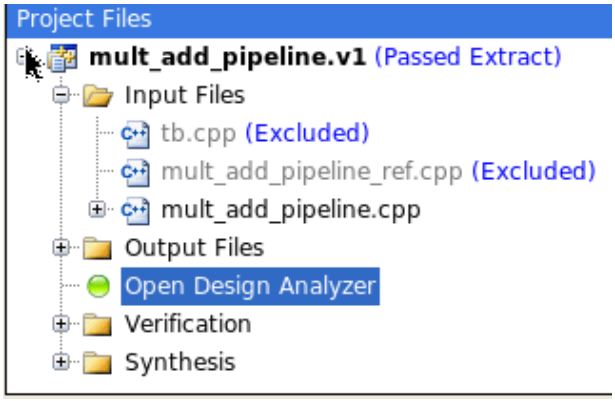
    a. Double-click on the operations in the Gantt chart to cross-probe back to the C++

b. Mouse over the operations to see the area and delay. Note that they are drawn to scale. The c-step (white column) generally represents one clock cycle, which in this example is 1.11ns



14. Expand the Design Analyzer Folder in the Project Files view and double-click on “Launch Design Analyzer”.

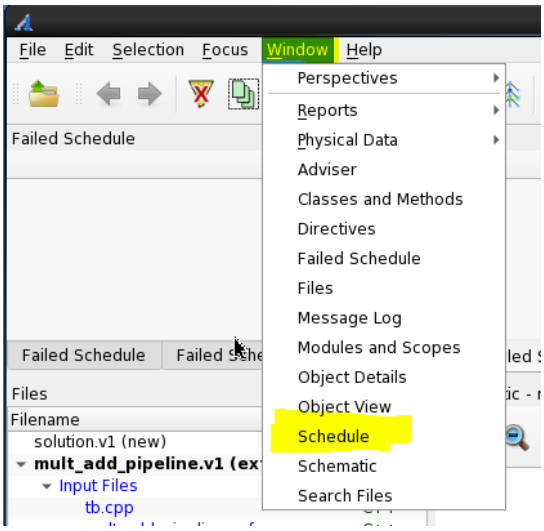
This will bring up the Catapult Design Analyzer tool which allows designers to see exactly how the hardware is created from the C++ code and design constraints.



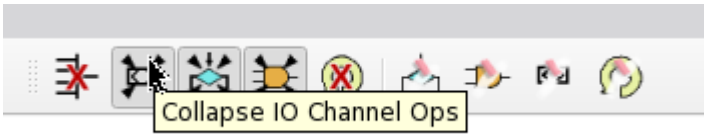
15. In Design Analyzer, click on the first multiplier in the Schedule view.

Note how the multiplier in the C++ code is highlighted. Note that the multiplier in the RTL schematic is also highlighted. Note also that the lines (data dependencies) that cross the c-step boundary (gray area) imply registers which can be seen in the RTL schematic.

- If the “Schedule” view does not open by default, you can open it from the “Window” menu.
- If the “Schematic” view does not open by default, select “Schematic” from “Window” menu.

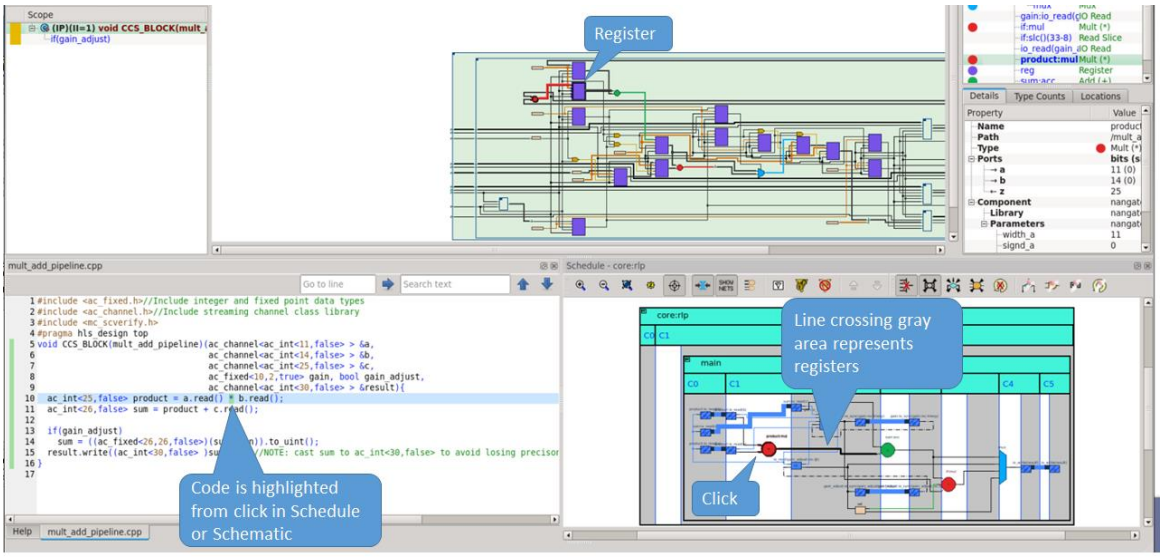


And zoom /re-center as needed. Toggle "Collapse IO Channel Ops" to view the multiplier.



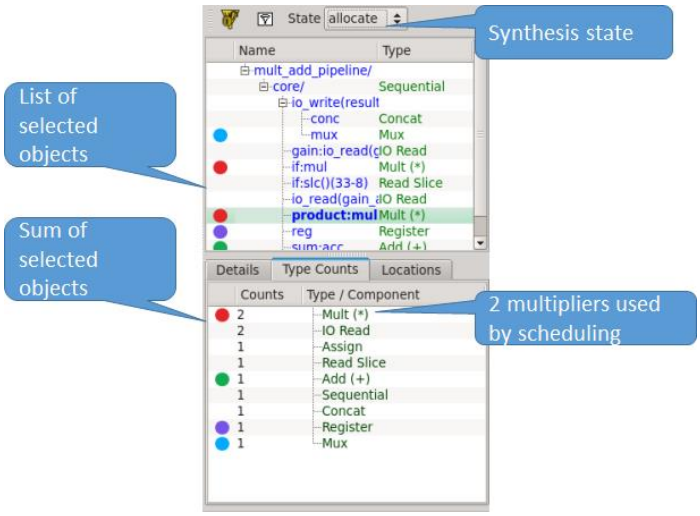
16. Click through the other operations in the schedule view and note how they map back to the C++ and to the RTL schematic.

17. Click on the "blue" objects in the Schedule view. These are the IO. Note how they map back to the C++.



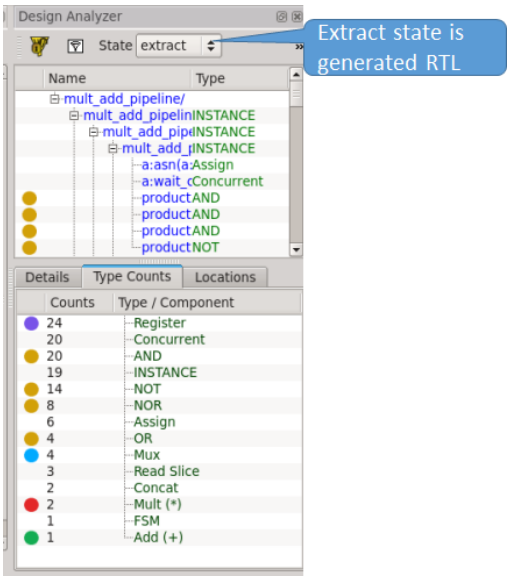
18. Click on the "Type Counts" Tab in the Design Analyzer window.

This gives a summary of all the objects for the current view and synthesis state.

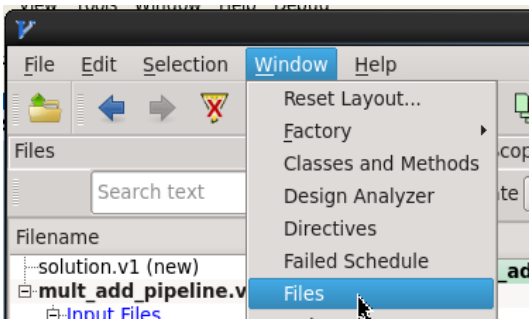


19. Change the State of the Design Analyzer to Extract

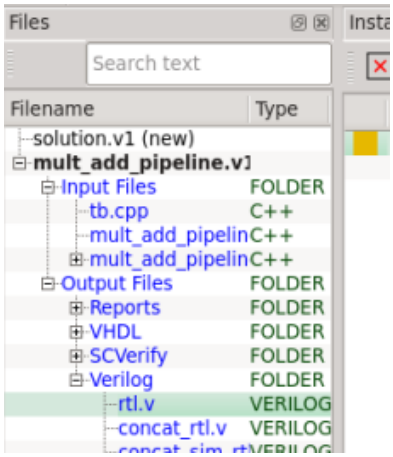
This allows you to see the hardware resources used for RTL generation. This includes registers and control logic added after scheduling.



20. Go to the Window menu and select Files.



21. Double-click on rtl.v in the Files window to open the RTL design.

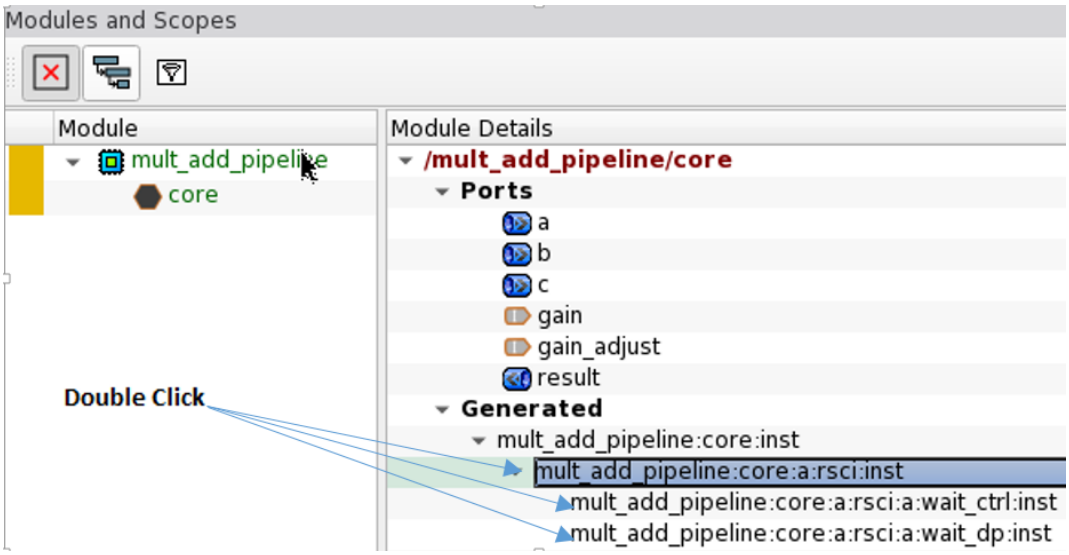


22. Click through the Schedule view and the RTL schematic and note how the RTL is highlighted. NOTE – not everything is cross-probable. There are some things in the schedule view that may not exist in the RTL.

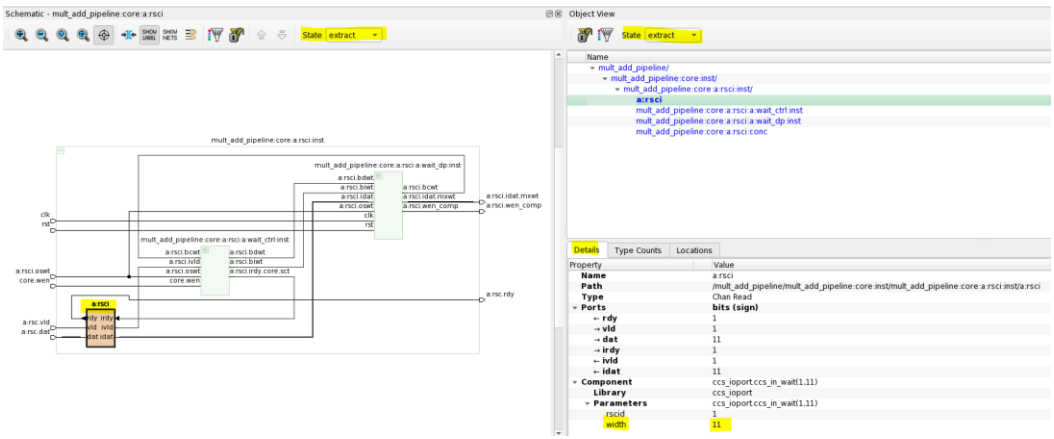
23. View “Modules and Scopes”.

    This will show all of the RTL instances added by synthesis such as IO, FSM, staller, etc.

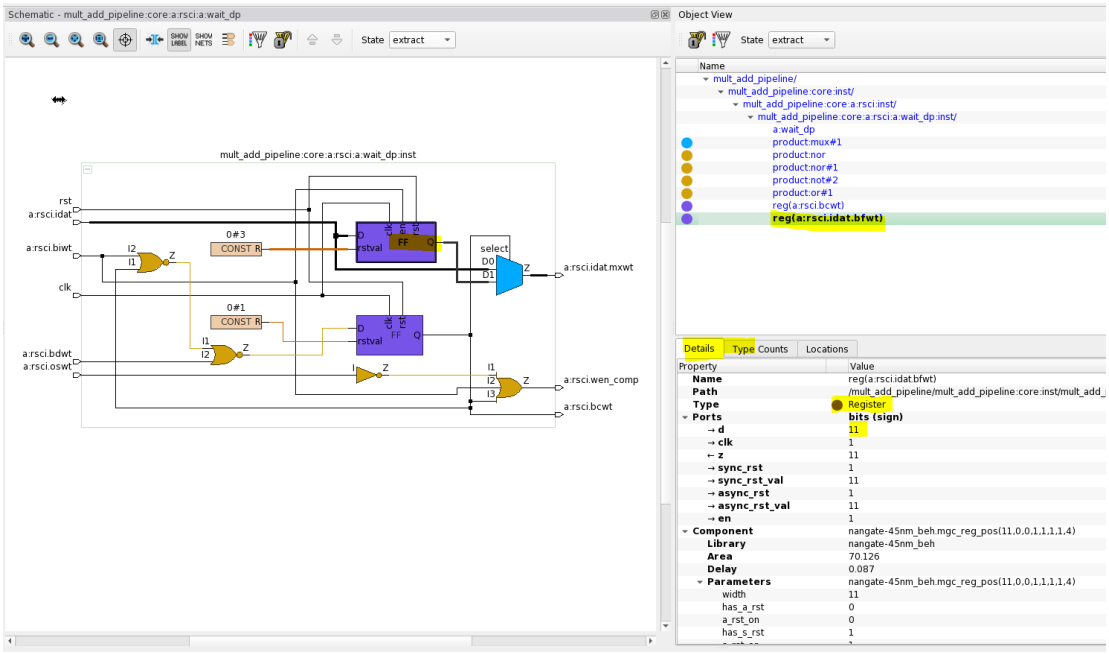
    a. Double-click on the a:rsci:inst block which is the interface for the “a” ac_channel.



b. Double-click on the two sub-blocks under a:rsci:inst which are the FIFO and wait controller.

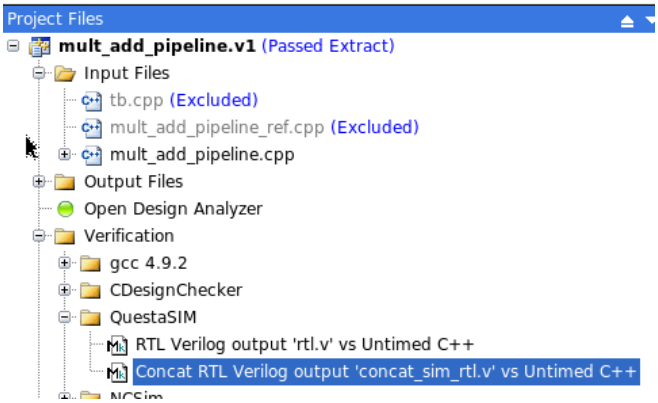


c. In ‘Schematic’ pane, double-click on “wait_dp:inst” block (1 element FIFO) to push down into the hierarchy and select the FFs and look at the details in “Object view”. This 11-bit register is the 1-element FIFO on the “a” input interface.



d. Close Design Analyzer

24. Expand the Verification Folder in the Project Files view and expand the folder for you RTL simulator. Then double-click on the “Concat RTL Verilog rtl vs Untimed C++” Makefile to launch the SCVerify verification flow.



25. Type “run –all” in the simulator command line.

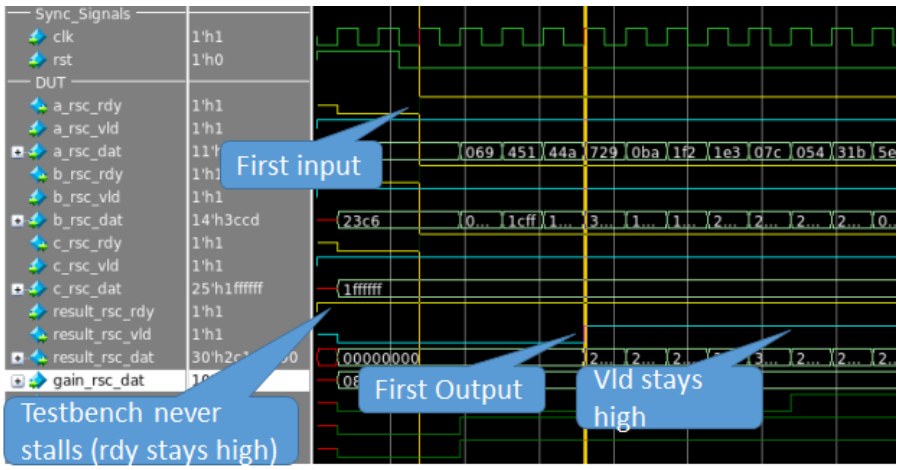
26. Look at the simulator transcript and you should see that the automatic verification has reported “PASSED” for 20 comparisons.

```
# Checking results
# 'result'
#    capture count        = 20
#    comparison count     = 20
#    ignore count         = 0
#    error count          = 0
#    stuck in dut fifo    = 0
#    stuck in golden fifo = 0
#
# Info: scverify_top/user_tb: Simulation PASSED @ 29755 ps
# ** Note: (vsim-6574) SystemC simulation stopped by user.
```

27. Look at the simulator waveform view and verify that the latency (time from first input to first output) equals 4 and the throughput (time between function calls in hardware executing) equals 1.

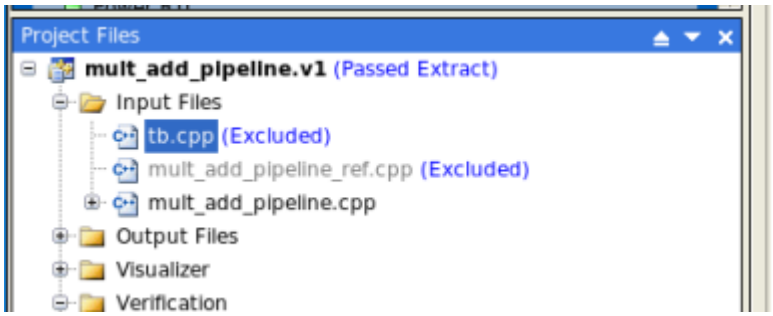
    You can see the throughput equals one because the result_rsc_vld signal goes high and stays high every clock cycle.

    Note also that the testbench never stalls the hardware. This is the default for SCVerify.



28. Close the simulator.

29. Go to the Catapult Project Files view and double click on tb.cpp.



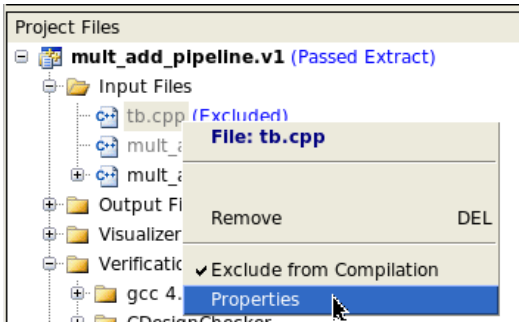
30. Look at the testbench and note that it has a #ifdef section of code to enable the stall controls for some of the IO. This code will stall the “a” input and “result” output IO.

CCS_SCVERIFY is defined automatically when SCVerify is run

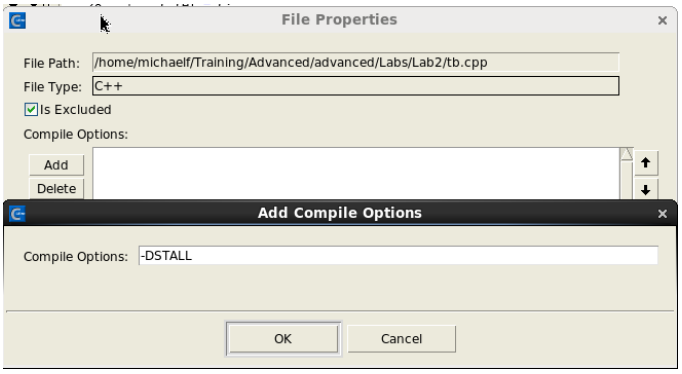
STALL must be set as a compiler define by the user.

```
29  #ifdef CCS_SCVERIFY
30  #ifdef STALL
31      if(i==3){
32        testbench::a_wait_ctrl.cycles = 2;
33      }
34      if(i==7){
35        testbench::result_wait_ctrl.cycles = 2;
36      }
37  #endif
38  #endif
```

31. Right-click on the testbench and select properties.



32. Click on the Add button and set a compiler define for stall. E.g. “-DSTALL”. Then click ok. This will branch a new solution.



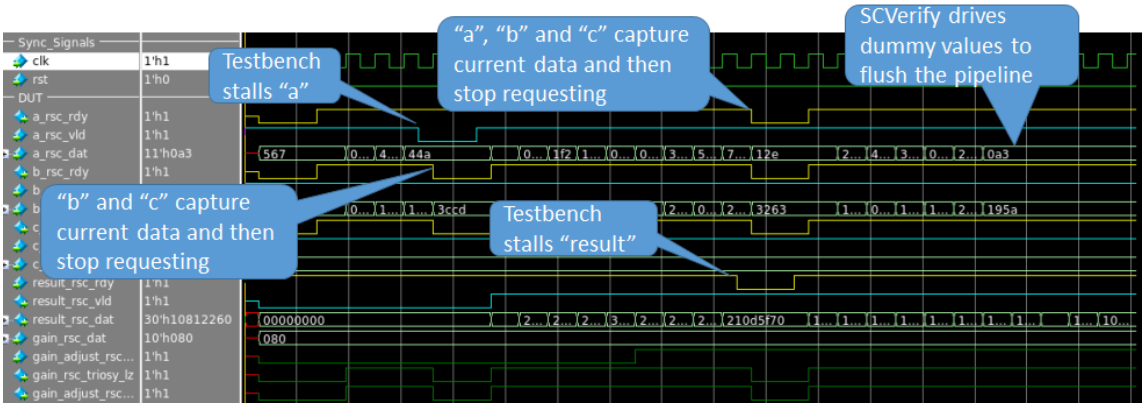
33. Regenerate the RTL design by clicking on Mapping in the Task bar and then Clicking on RTL in the Task.

34. Re-run the SCVerify verification flow and look at the waveform view

a. Note that a stall on any input or output IO will stall the entire design

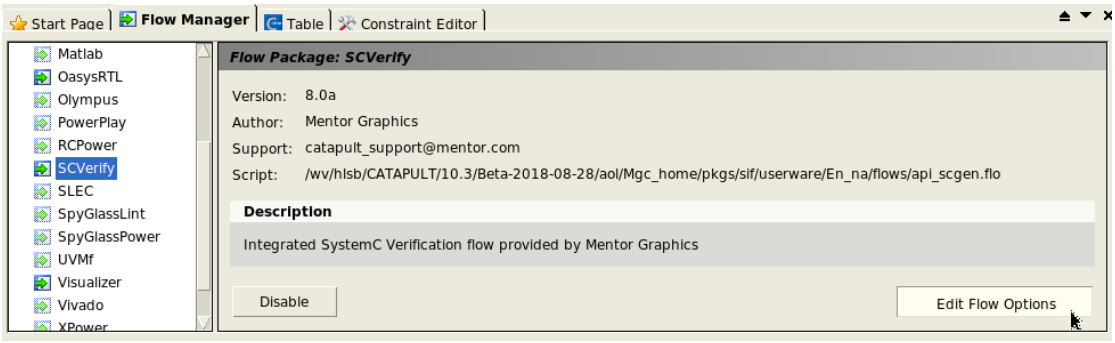
b. Note that the input IO that are not stalled will complete the current read and then stop requesting data (rdy goes low)

c. Note that SCverify continues to drive “dummy” inputs after all the testbench inputs have been read. This is done because SCVerify needs to compare all of the outputs captured in the C++ testbench. However, this means that the hardware pipeline will not flush by default if the inputs are removed.

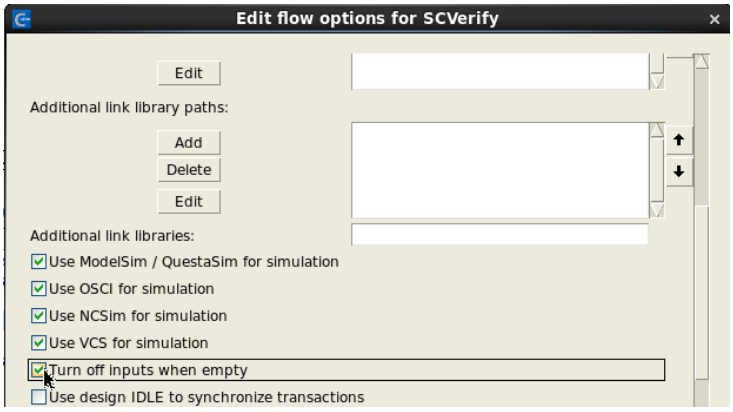


35. Close the RTL simulator

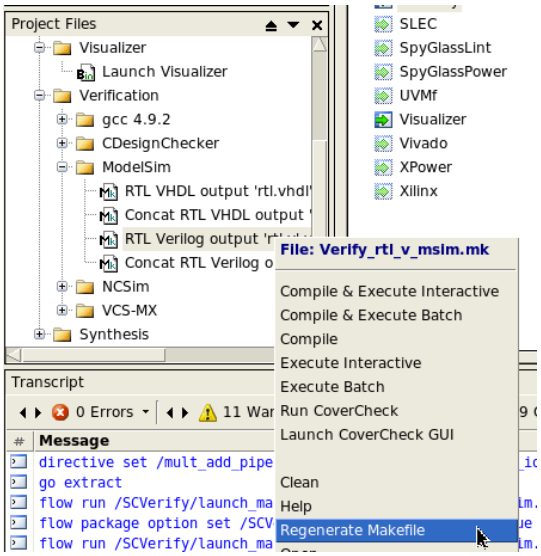
36. Click on the Flow Manager tab and edit the SCVerify flow options.



37. Check the box for “Turn off inputs when empty”. This will prevent SCVerify from driving dummy values to flush the pipeline.



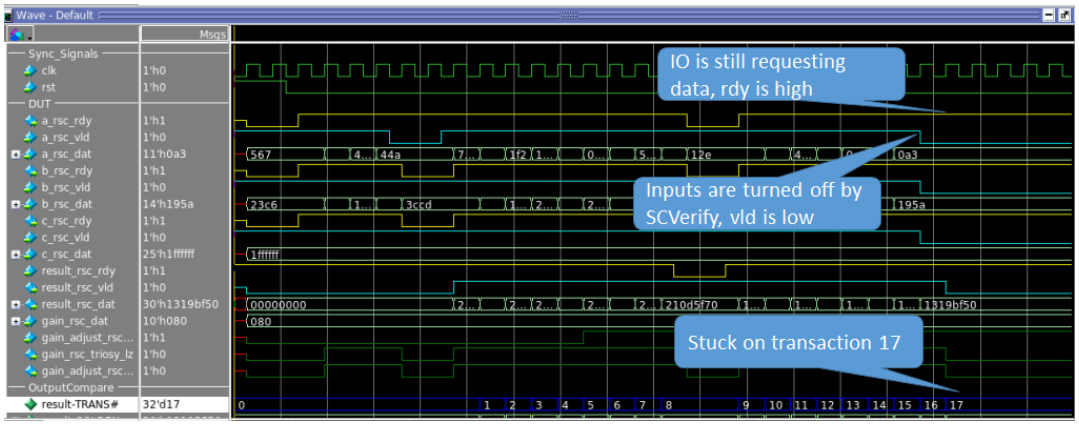
38. Right-click on the SCVerify Makefile and select “Regenerate Makefile”.



39. Launch SCVerify and run the verification for 100 ns.

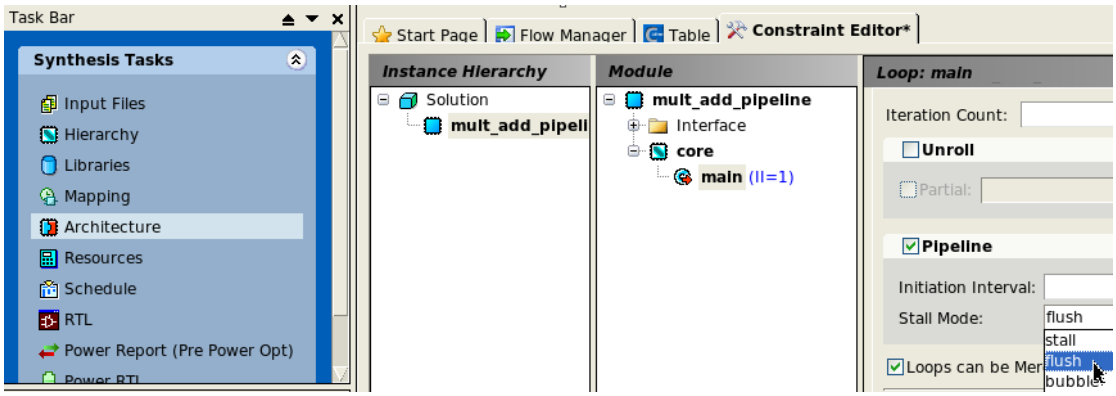
40. Look at the waveform view.

- Note that the inputs are now stalled by the testbench
- Also, note that the inputs are still requesting data
- Look at the tb.cpp file and note that the design is called 20 times in the for loop. This means there should be 20 outputs to be compared.
- Look at the output transactions in SCVerify and note that it is stuck on transaction number 17.
- Look at the transcript and note that there is no “PASSED” message issued.



41. Close the simulator.

42. Click on Architecture in the Task Bar, select the main loop, and set the stall mode to flush. This will enable hardware pipeline flushing.

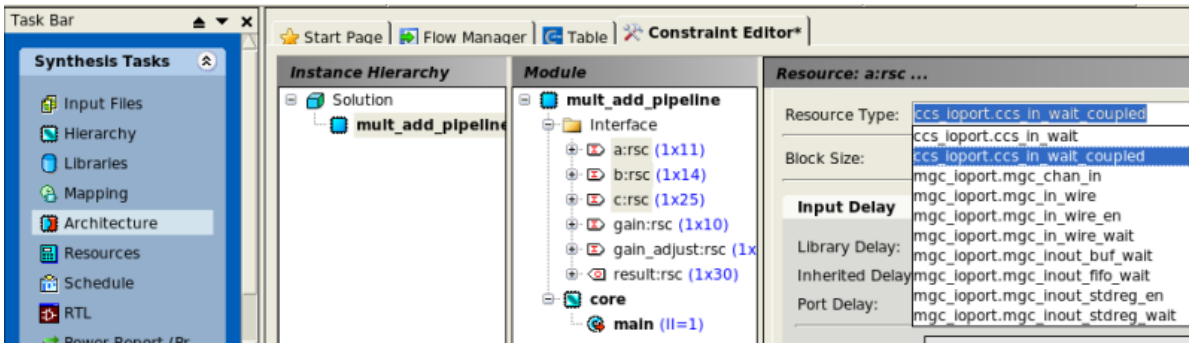


43. Regenerate the RTL

44. Launch SCVerify and run verification. Look at the waveform view and note that all 20 transactions are compared.

45. Look at the transcript and note the “PASSED” message.

46. Click on Architecture in the Task Bar and set the “a”, “b”, and “c” Resource Type to ccs_in_wait_coupled.



47. Regenerate the RTL.

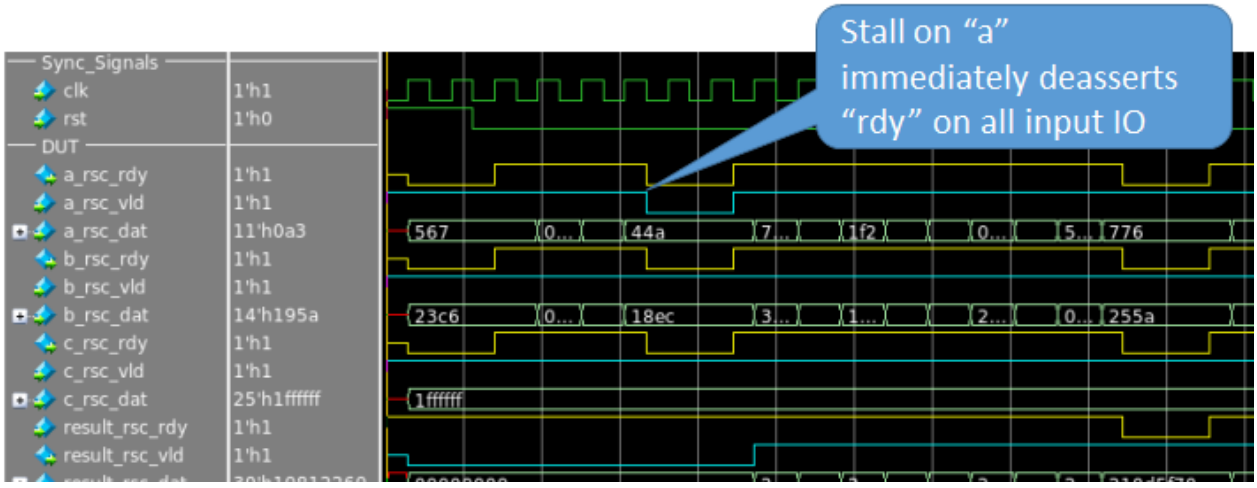
48. Look at the table view, switch to Area Score and note the register area and mux area has decreased.

Report: Area Score

Solution	Registers	MUX	Functional	Logic	Memory	FSM Reg	FSM Comb	FSM	Datapath	T
solution.v1 (new)										
mult_add_pipeline.v1 (extract)	1338.75	133.34	2751.63	42.14	0.00	10.00	1.00	11.00	4265.85	
mult_add_pipeline.v2 (extract)	1338.75	133.34	2751.63	42.14	0.00	10.00	1.00	11.00	4265.85	
mult_add_pipeline.v3 (extract)	**1000.88**	**45.62**	**2579.75**	**26.85**	**0.00**	**10.00**	**1.00**	**11.00**	**3653.09**	

49. Re-run SCVerify verification.

50. Note how the coupled IO all stop requesting data immediately on an IO stall



51. Close the Simulator

52. Open Design Analyzer and repeat step 23. You can see that the wait_dp block (1-element FIFO) has been removed from the IO.

53. Close Design Analyzer.

54. Close Catapult

DONE LAB2

Lab 3: Loop Unrolling and Pipelining

Overview

In this lab you will learn how to:

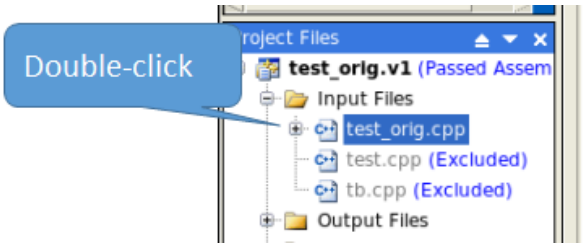
- Use loop unrolling and loop pipelining to improve design performance
- Analyze designs to see the effects of loop unrolling and bad coding style
- Recode the C++ to improve area and runtime

Steps Part 1

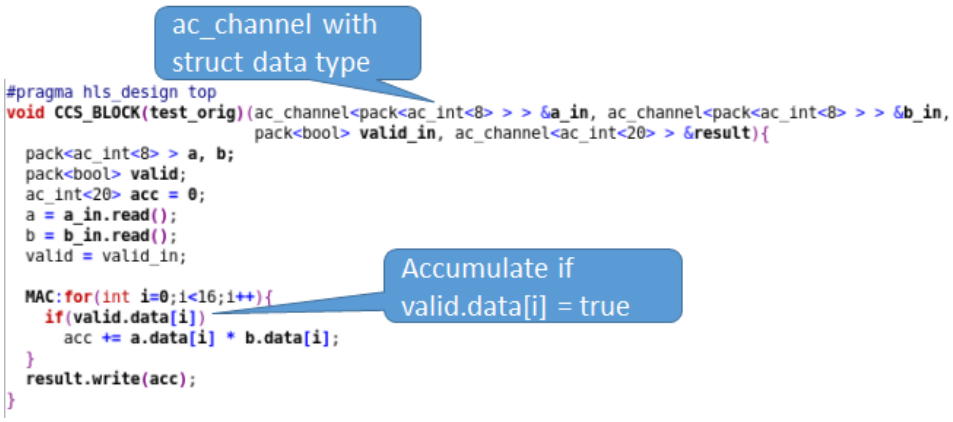
1. Generate the Initial Design
2. Pipeline the MAC Loop (II=1)
3. Pipeline the Entire Design
4. Unroll the MAC loop by 2
5. Fully unroll the MAC loop
6. Read the modified test design

Generate the Initial Design

1. CD to the 'lab3' directory
2. Launch catapult by typing "catapult" in the terminal
3. Go to the File menu and select "run script". Select the conditional_mac.tcl file and click open. This will:
    1. Add input files
    2. Set the synthesis tool to OasysRTL
    3. Set the clock frequency to 1000MHz
4. Double-click on the "test_orig.cpp" file in the Project Files folder.



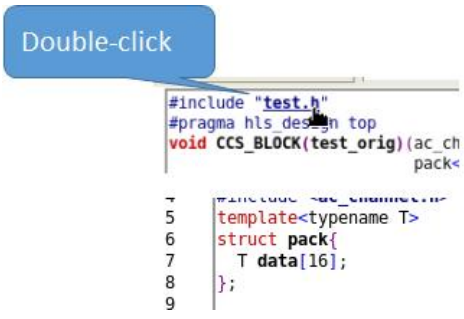
5. Examine the C++ design and note the following:
   - The design has ac_channels for a_in and b_in that have a struct data type called “pack”
   - An interface variable “valid_in” is used to conditionally multiply-accumulate 16 values from a_in and b_in



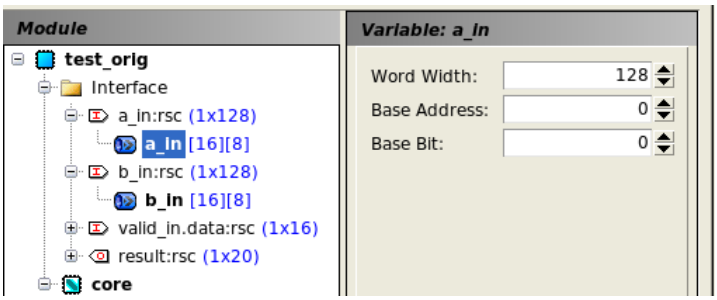

```
#pragma hls_design top
void CCS_BLOCK(test_orig)(ac_channel<pack<ac_int<8> > > &a_in, ac_channel<pack<ac_int<8> > > &b_in,
                          pack<bool> valid_in, ac_channel<ac_int<20> > &result){
  pack<ac_int<8> > a, b;
  pack<bool> valid;
  ac_int<20> acc = 0;
  a = a_in.read();
  b = b_in.read();
  valid = valid_in;

  MAC:for(int i=0;i<16;i++){
    if(valid.data[i])
      acc += a.data[i] * b.data[i];
  }
  result.write(acc);
}
```

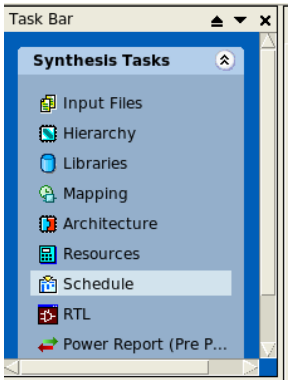
6. Double-click on the #include “test.h” at the top of the “test_orig.cpp” file.
7. Note that the “pack” data type is defined here and contains an array of 16 words.



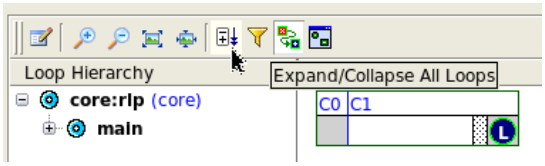
8. Click on Architecture in the Task Bar and expand the Interface Folder.
9. Note that “a_in” and “b_in” interfaces are 16x8 or 128 bits wide. All 16 values are available in parallel.



10. Click on Schedule in the Task Bar

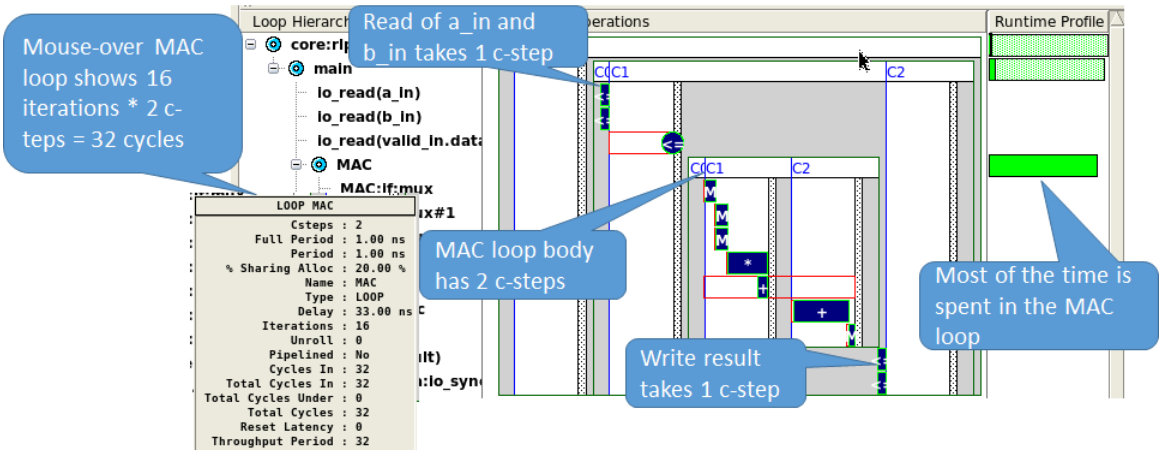


11. Look at the table view and note that the latency is 33 and throughput is 34.
12. Open the Catapult Gantt chart and expand all loops by clicking on the icon with the down arrow in the tool bar.



Look at the Gantt chart and note the following:

- The read of a_in and b_in is taking one c-step (cycle)
- The write of result takes 1 c-step
- The MAC loop body has 2 c-steps
- Most of the runtime (Big green bar) is spent executing the MAC loop
- Hover the mouse over the MAC loop and see that it takes 32-cycles to execute. This is because the unpipelined loop has 16 iterations with each iteration taking 2 c-steps. 16x2 = 32.

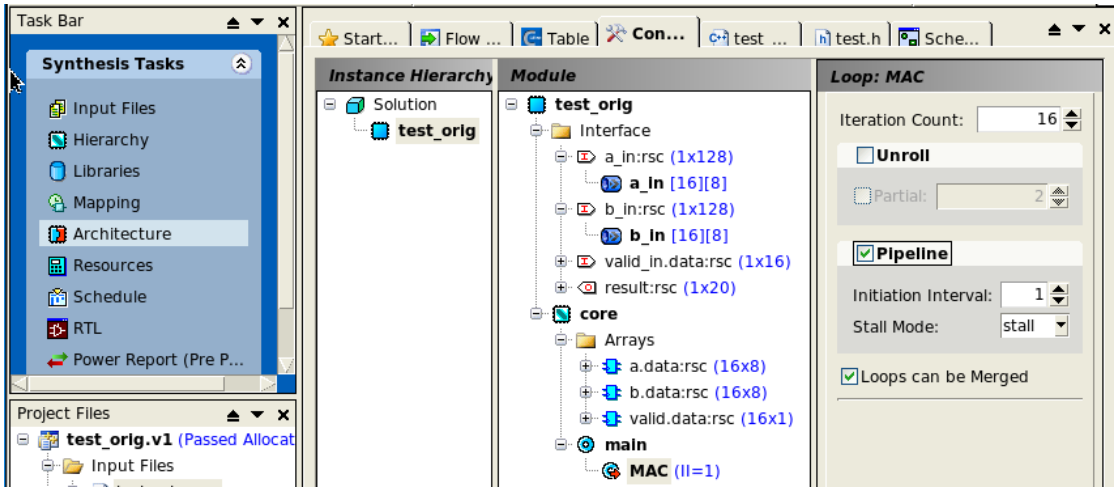


13. Click on RTL in the Task Bar to generate RTL.

Pipeline the MAC Loop (II=1)

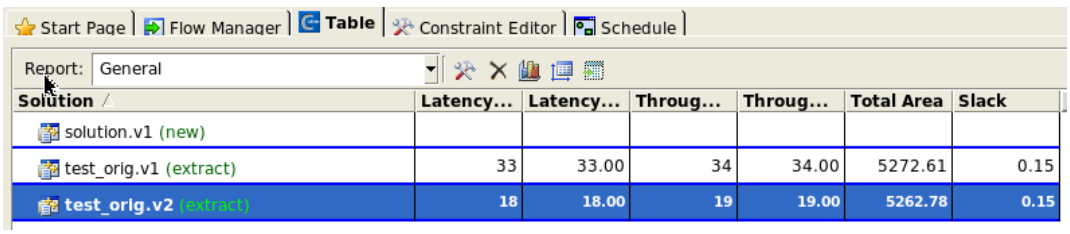
1. Click on Architecture in the Task Bar.

2. Select the MAC loop and set the Pipeline Initiation Interval to II=1.



3. Click on RTL in the Task Bar.

   Look at the Table View and note that the latency has been reduced to 18 and the throughput to 19. Also, note that the area has gotten slightly smaller. Pipelining can often improve performance with less of an impact on area.

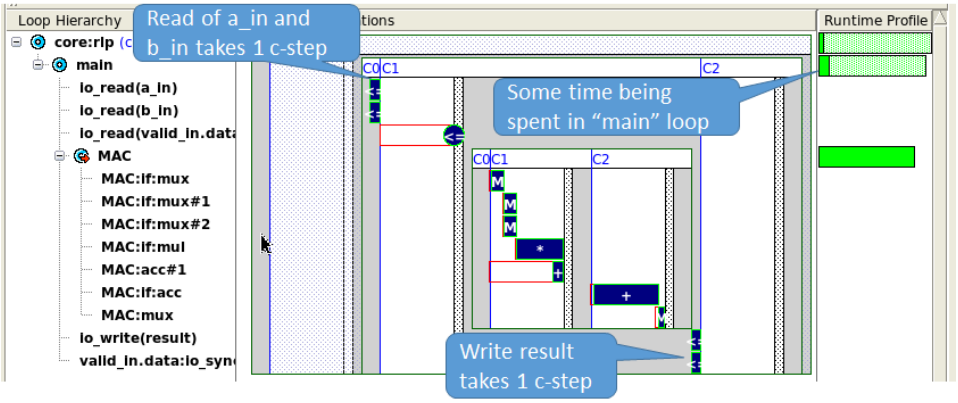


Solution	Latency...	Latency...	Throug...	Throug...	Total Area	Slack
solution.v1 (new)						
test_orig.v1 (extract)	33	33.00	34	34.00	5272.61	0.15
test_orig.v2 (extract)	**18**	**18.00**	**19**	**19.00**	**5262.78**	**0.15**

4. Click on Schedule in the Task Bar and open the Gantt Chart.

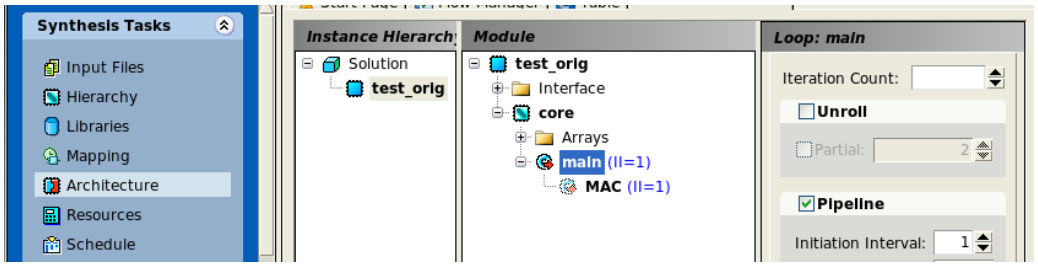
5. Expand all loops in the Gantt chart

   Note that most of the time is still spent in the MAC loop (big green bar) but 2 c-steps are reading and writing the IO in the “main” loop (Little green bar).



Pipeline the Entire Design

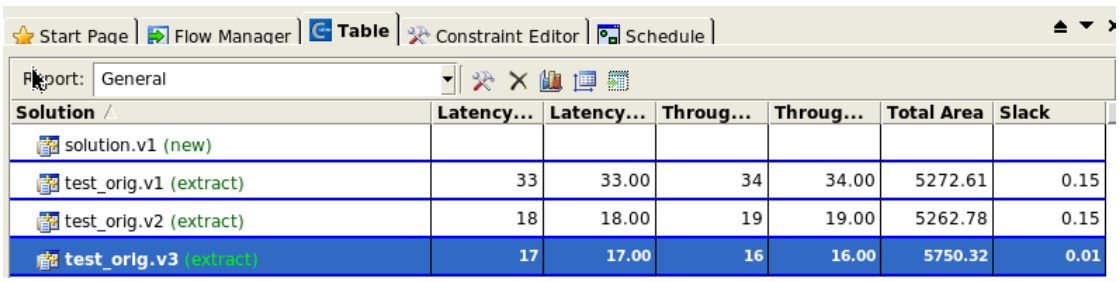
1. Click on Architecture in the Task Bar and pipeline the “main” loop with II=1.



2. Click on RTL in the Task Bar

3. Look at the Table View and Note:

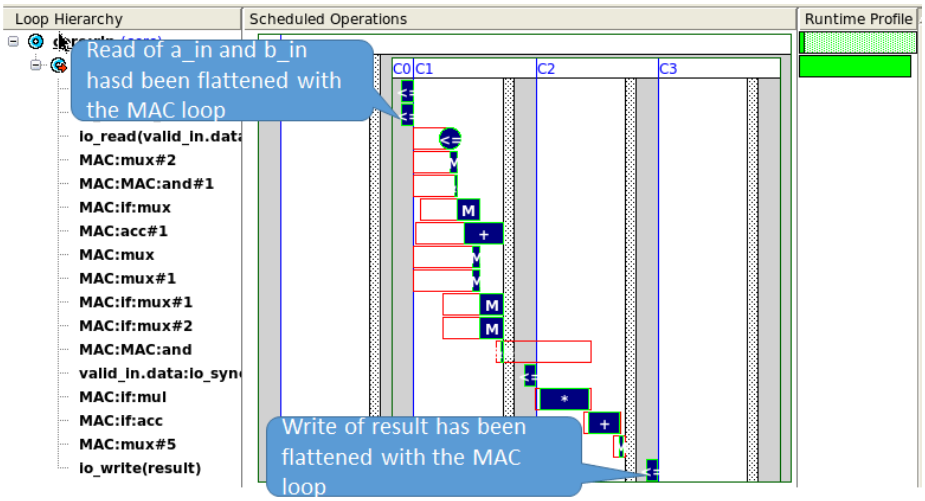
   - The latency is 17 and the throughput is 16
   - The area has increased slightly.



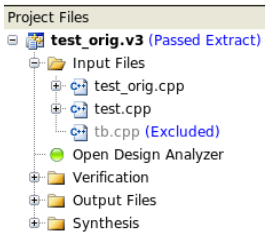
Solution	Latency...	Latency...	Throug...	Throug...	Total Area	Slack
solution.v1 (new)						
test_orig.v1 (extract)	33	33.00	34	34.00	5272.61	0.15
test_orig.v2 (extract)	18	18.00	19	19.00	5262.78	0.15
test_orig.v3 (extract)	**17**	**17.00**	**16**	**16.00**	**5750.32**	**0.01**

4. Click on Schedule in the Task Bar and open the Gantt chart.

   Note that the “main” loop and the MAC loop have been flattened together.

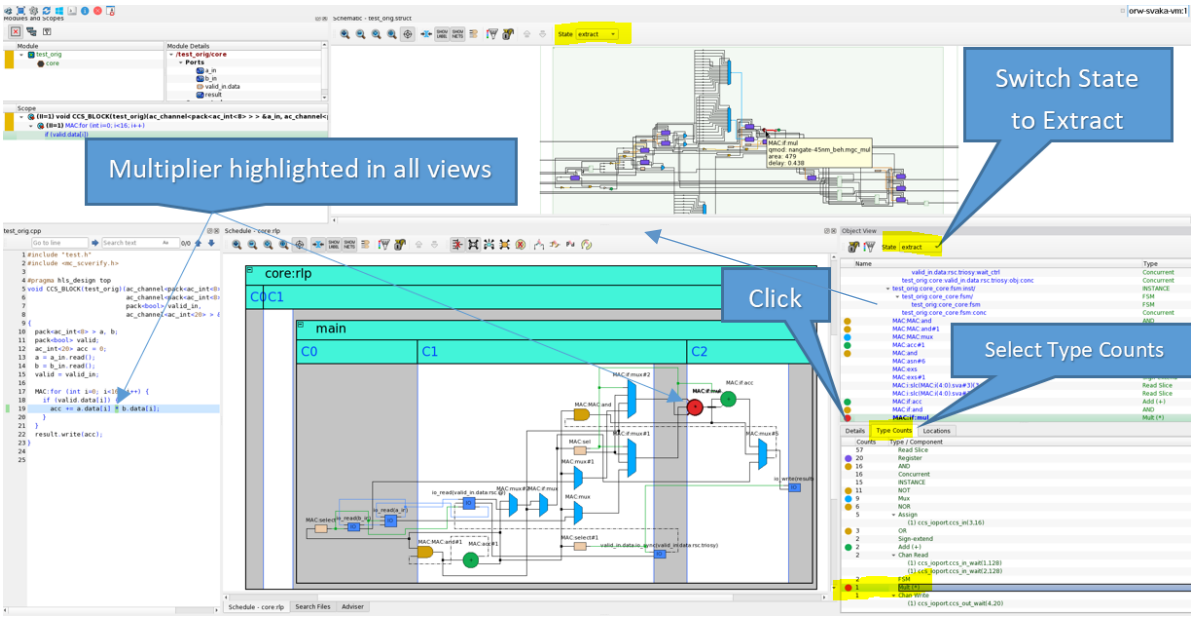


5. Go to the Project Files and double-click Open Design Analyzer to launch the Design Analyzer.



6. Switch the Design Analyzer view state to extract and click on the type counts tab. This gives a summary of all hardware resources. Note that a single hardware multiplier is used for this design.

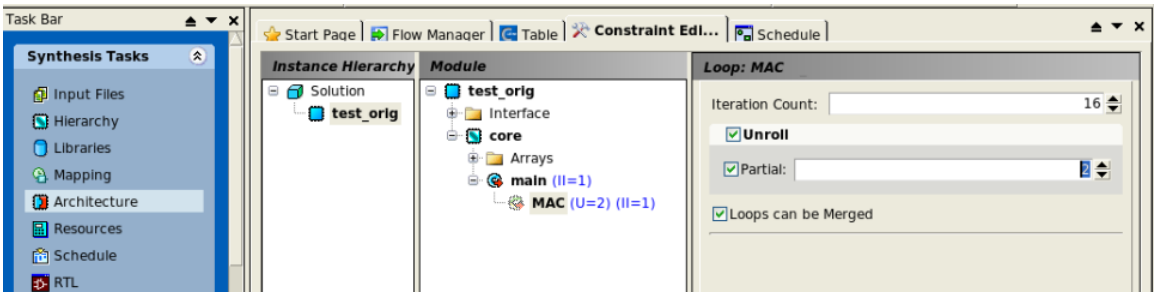
7. Click to select the multiplier in the ‘Object View’ list and note how it is highlighted in all other views.



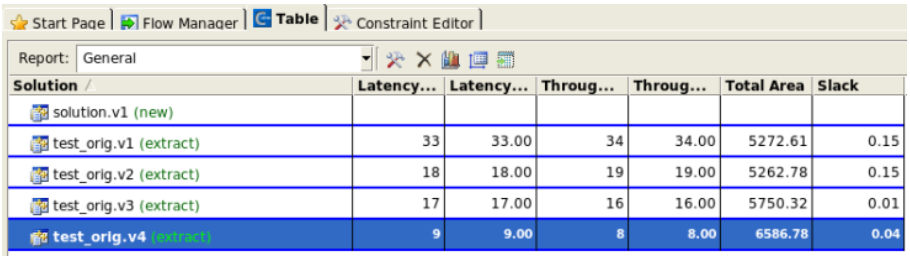
14. Close Design Analyzer.

Unroll the MAC loop by 2

1. In Catapult, click on Architecture in the Task Bar.
2. Select the MAC loop and unroll partially by 2.



3. Generate RTL.
4. Look at the table view and note that the throughput has been reduced to 8 cycles, with an increase in area.



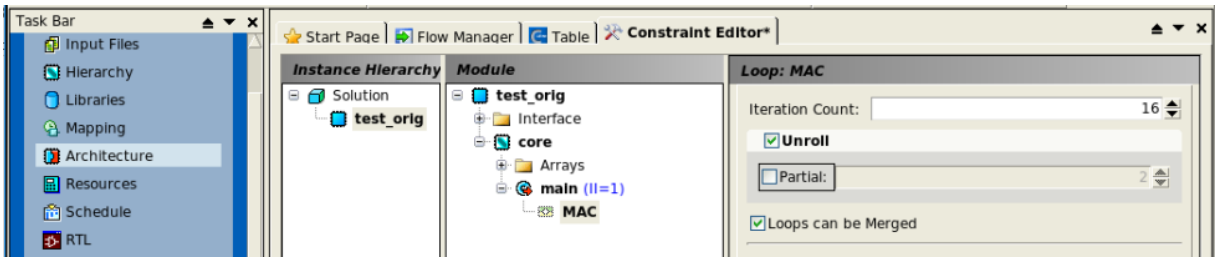
Solution	Latency...	Latency...	Throug...	Throug...	Total Area	Slack
solution.v1 (new)						
test_orig.v1 (extract)	33	33.00	34	34.00	5272.61	0.15
test_orig.v2 (extract)	18	18.00	19	19.00	5262.78	0.15
test_orig.v3 (extract)	17	17.00	16	16.00	5750.32	0.01
test_orig.v4 (extract)	9	9.00	8	8.00	6586.78	0.04

5. Use the Design Analyzer to verify that the design is now implemented using 2 hardware multipliers. Loop unrolling the MAC loop by 2x has doubled the number of hardware resources.

Fully unroll the MAC loop

1. In Catapult click on Architecture in the Task Bar.

2. Select the MAC loop and fully unroll the loop.

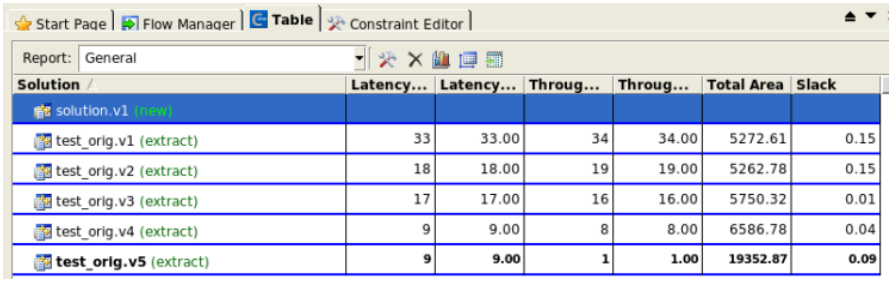


3. Click on RTL in the Task Bar.

4. Look at the Table View and note:

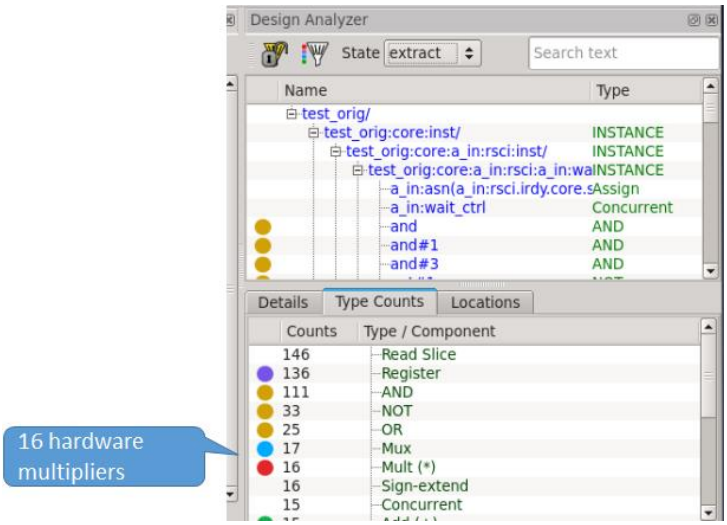
   - The throughput is now equal to one but the latency is still 9

   - The area has increased substantially.

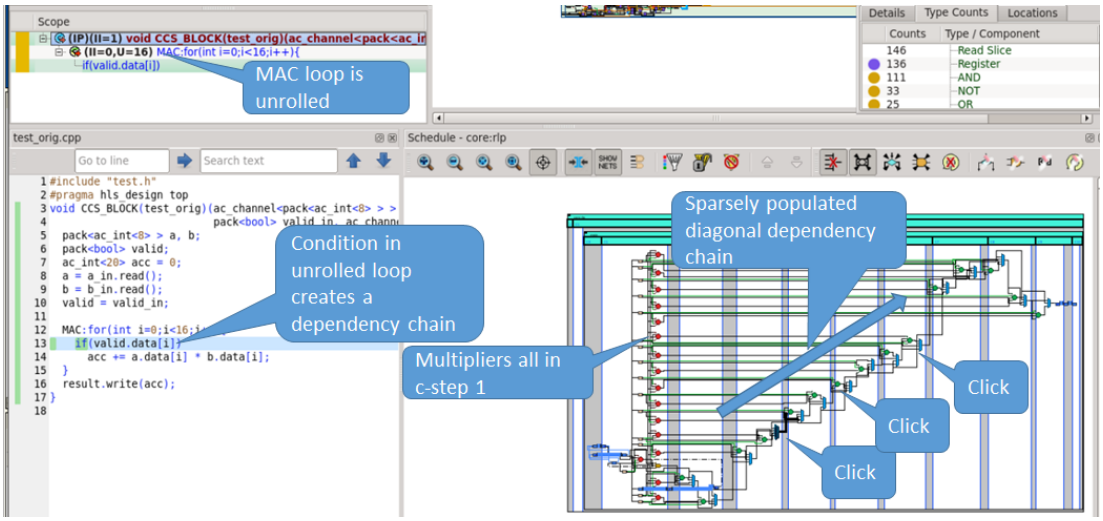


Solution	Latency...	Latency...	Throug...	Throug...	Total Area	Slack
solution.v1 (new)						
test_orig.v1 (extract)	33	33.00	34	34.00	5272.61	0.15
test_orig.v2 (extract)	18	18.00	19	19.00	5262.78	0.15
test_orig.v3 (extract)	17	17.00	16	16.00	5750.32	0.01
test_orig.v4 (extract)	9	9.00	8	8.00	6586.78	0.04
test_orig.v5 (extract)	**9**	**9.00**	**1**	**1.00**	**19352.87**	**0.09**

5. Use the Design Analyzer to verify that the design is now implemented using 16 hardware multipliers.



6. Look at the Design Analyzer Schedule view and note:
   - All the multipliers are scheduled in c-step 1
   - There is a diagonal chain of operations (add and mux) scheduled in c-step 2 to c-step 9.
7. Click through the chain of adders and muxes in the Schedule view and see how the “if” condition and accumulate operation are highlighted in the C++.



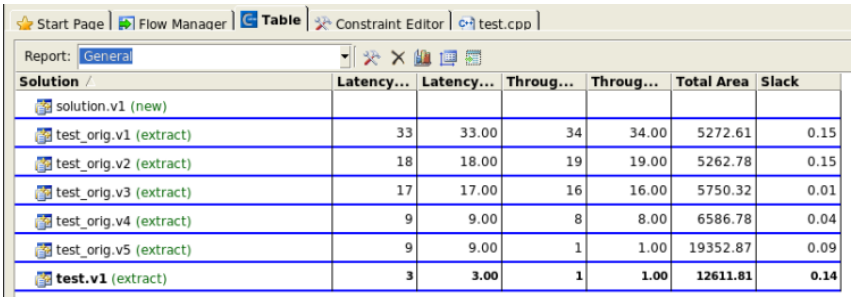
8. Unrolling loops with conditions, conditional breaks, or complex control may result in priority encoded dependency chain logic.
9. Close Design Analyzer.

Read the modified test design

1. Go to the File Menu and select “Run Script”.

2. Select the “coding_style_cond.tcl” script and click Open. This will make the “test.cpp” the top-level design, re-constraint the design, and generate RTL.

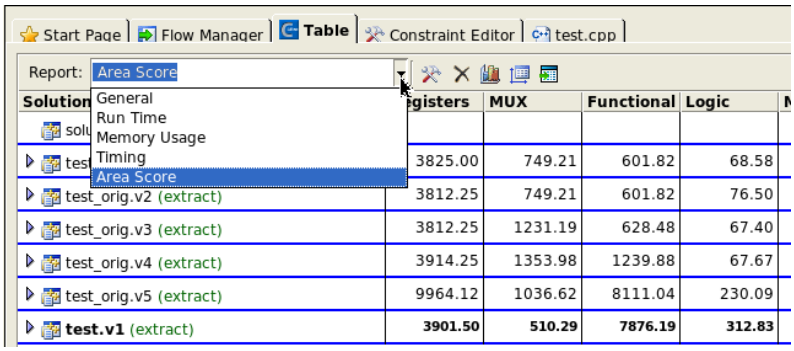
3. Look at the table view and Note that the latency is now 3, throughput is 1, and the area has gone down significantly.



Solution	Latency...	Latency...	Throug...	Throug...	Total Area	Slack
solution.v1 (new)						
test_orig.v1 (extract)	33	33.00	34	34.00	5272.61	0.15
test_orig.v2 (extract)	18	18.00	19	19.00	5262.78	0.15
test_orig.v3 (extract)	17	17.00	16	16.00	5750.32	0.01
test_orig.v4 (extract)	9	9.00	8	8.00	6586.78	0.04
test_orig.v5 (extract)	9	9.00	1	1.00	19352.87	0.09
test.v1 (extract)	**3**	**3.00**	**1**	**1.00**	**12611.81**	**0.14**

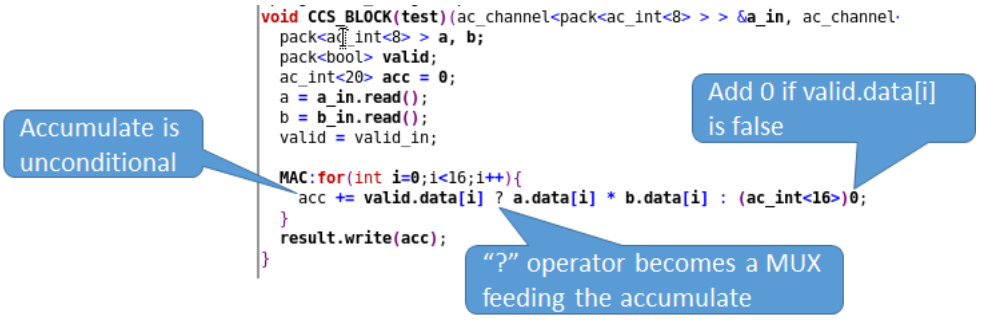
4. Change the Table View report to Area Score

   Note that the registers area has decreased by a large amount. This should make sense because the design latency went from 9 cycles to 3 cycles so there are fewer pipeline registers.



5. Go to the Input Files Folder and double-click on “test.cpp”

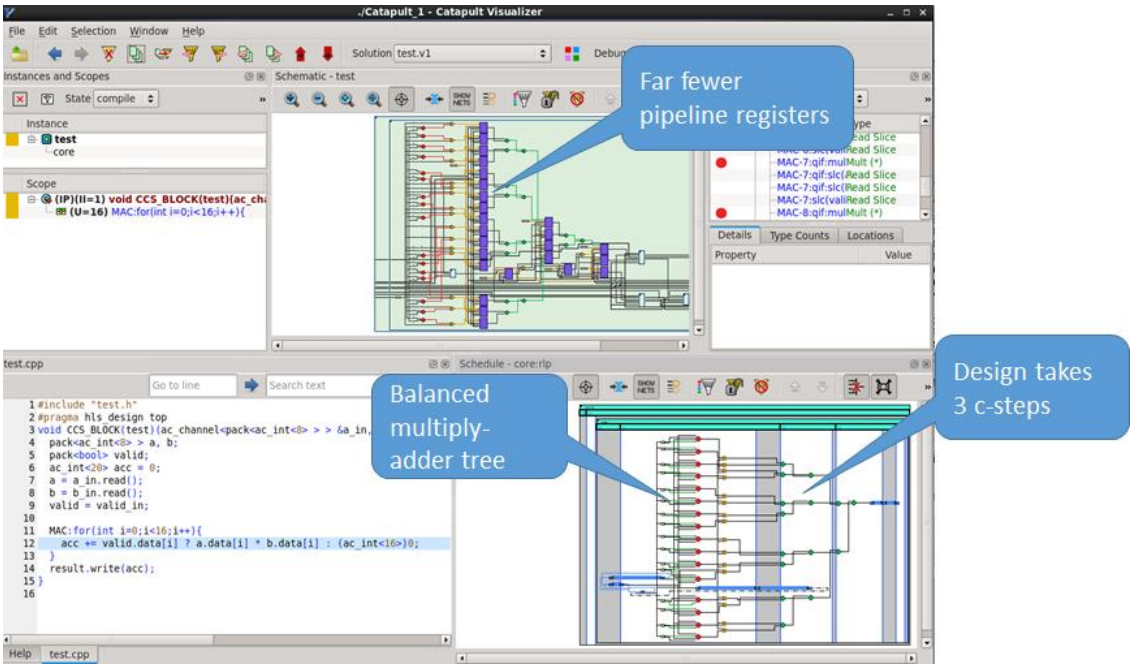
   Note how the code has been rewritten to that there is no longer a conditional accumulate. The “?” operator is used to always add either the result from the multiplication or zero.




```
void CCS_BLOCK(test)(ac_channel<pack<ac_int<8> > > &a_in, ac_channel
  pack<ac_int<8> > a, b;
  pack<bool> valid;
  ac_int<20> acc = 0;
  a = a_in.read();
  b = b_in.read();
  valid = valid_in;

  MAC:for(int i=0;i<16;i++){
    acc += valid.data[i] ? a.data[i] * b.data[i] : (ac_int<16>)0;
  }
  result.write(acc);
}
```

6. Open Catapult Design Analyzer and look at the hardware structure.



7. Close Design Analyzer

8. Close catapult

Steps Part 2

Part one of the lab showed how to understand the effects of loop unrolling and pipelining as well as how to analyze the effects of "bad" coding style for HLS when combined with loop unrolling. It showed how unrolling loops with operations such as conditional accumulates can create long dependency chains of logic. This lab will look at the effects of dynamically indexing an array mapped to registers inside a loop that is unrolled.

1. CD to the Labs/Lab3 directory

2. Launch Catapult at the command prompt by typing "catapult"

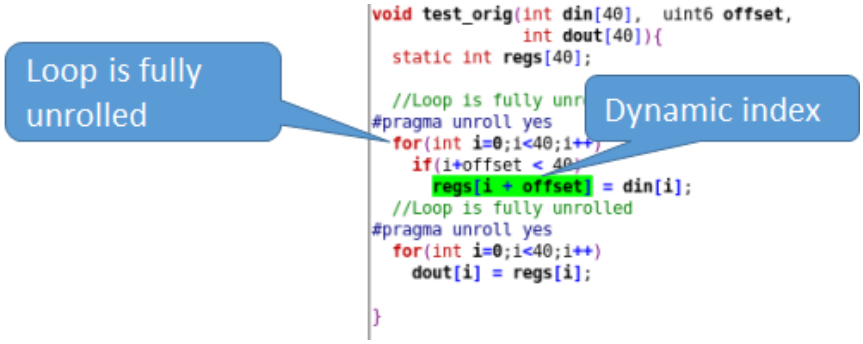
3. Go to File > Run Script and select the ***dynamic_offset.tcl*** file and click OK. This will synthesize the design which will take 3 or 4 minutes.

4. Open the design “test_orig.cpp” and look at the C++ code.

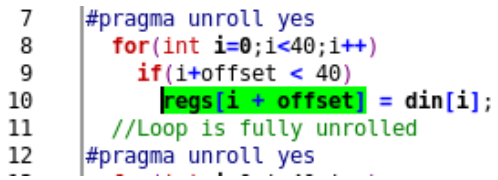
   - This design copies an input array “din[40]” with 40 elements to an internal array “regs[40]” which is mapped to registers. The index to write “regs” contains a variable “offset” plus the loop index “i” to select where to begin writing.

   - The loop, which has 40 iterations is fully unrolled. By unrolling the dynamic index “regs[i+offset]” we are creating 40 dynamic indices into “regs”.

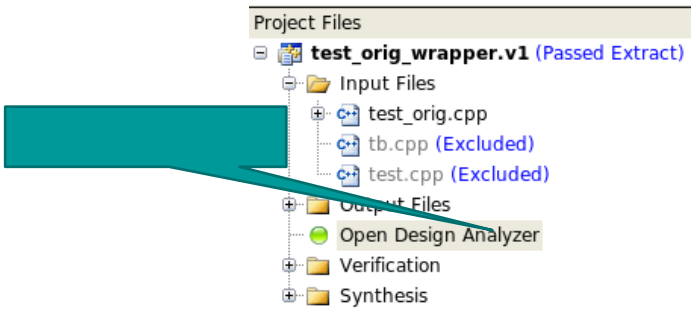
   - Catapult will split an array mapped to registers with a dynamic index into many variables. If the array is inside an unrolled loop it will be split many times. This will lead to long runtime and larger area.



5. Find the warnings about “dynamic index” and double-click on them to cross to the C++. The warnings are all cross-probing back to the line that indexes “regs[i+offset”.



6. Launch Catapult Design Analyzer. Note it may take several seconds to load.



7. Click on the “Locations” tab in the Design Analyzer window.

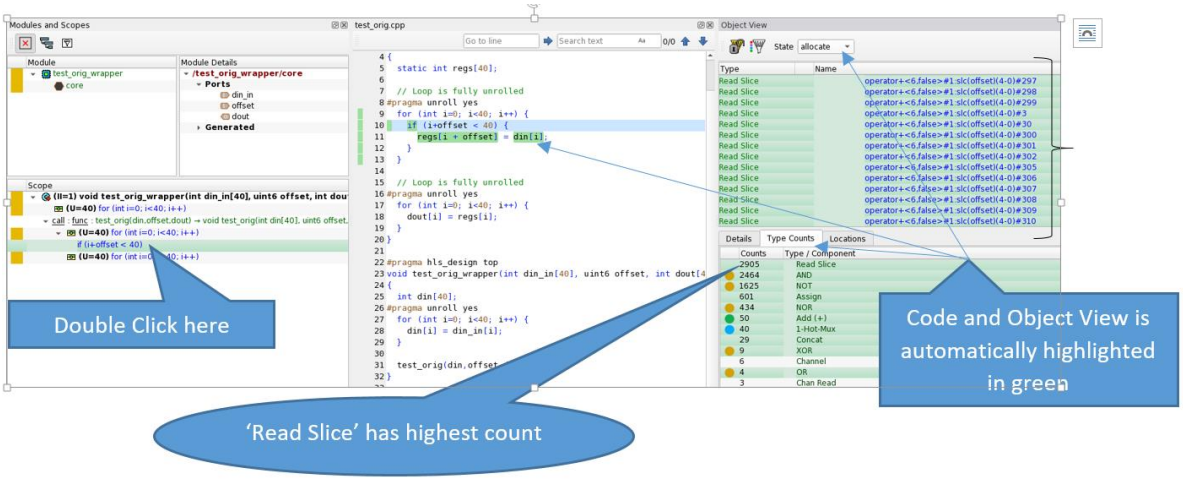
   This provides an ordered (largest to smallest) list of the number of operations per line of C++. A very large number of operations associated to a single line of C++ is usually an indication that an optimization has “blow-up” the design.

   a. Note that there are thousands of operations (and, readslice, etc) all coming from a single line of C++.

   b. Click on line in the Locations tab and it will highlight the C++ source as well as the scope where the source comes from.

   c. Note that the source contains a dynamic index.

   d. Note that the highlighted scope in the scope view is inside of a loop that is unrolled 40 times.



8. Change the Design Analyzer state to “Memories” and reanalyze the design by checking the Locations tabs.

   The “memories” state is after memory optimizations including register splitting so analyzing here still provides all of the necessary detail. This is very useful since more complicated design may get stuck in allocate.

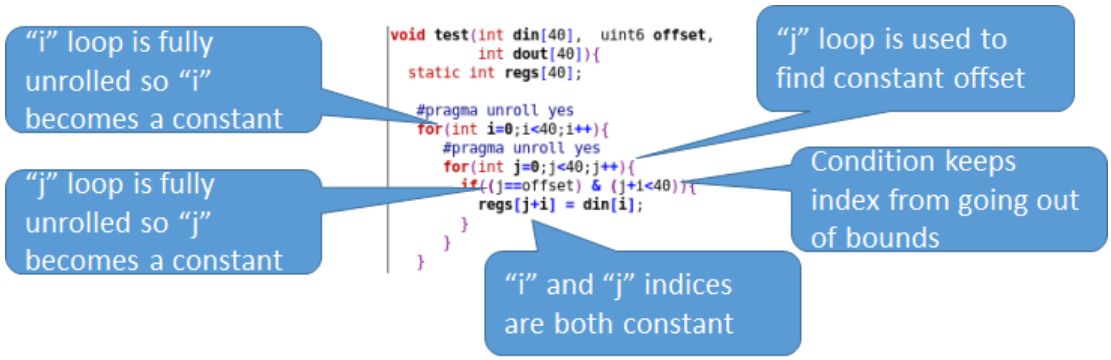
9. Close Design Analyzer

10. Go to File > Run Script and select the "***coding_style_dynamic.tcl***" file and click Open. This will make "test.cpp" the top-level design and will exclude "test_orig.cpp" from synthesis but still use it to compare against in the testbench. The script will also synthesize the design.

11. Open "test.cpp" and note the following:

    - An additional loop with index "j" has been added to compute the offset. Because the "j" loop is fully unrolled the loop iterator "j" will become a constant allowing Catapult to optimize the design.

    - "regs" is indexed by "i+j" which resolves to constants because both loops are fully unrolled.



12. Look at the Table View and note the area improvement.

Report: General

Solution	Latency...	Latency...	Throug...	Throug...	Total Area	Slack
test_orig_wrapper.v1 (extract)	1	2.00	1	2.00	35921.47	1.20
test_wrapper.v1 (extract)	**1**	**2.00**	**1**	**2.00**	**31306.21**	**1.73**

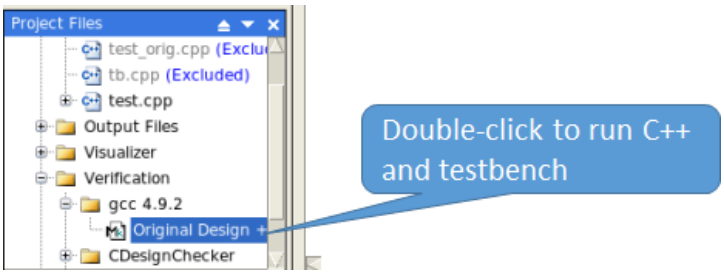
13. Change the Table View to show Runtime and note the runtime improvement.

Report: Run Time

Solution	Total	Analyze	Compile	Libraries	Assembly	Loops	Memories	Cluster	Architect	Allocate	Schedule	Dpfsm	Instance	Extract
test_orig_wrapper.v1 (extract)	184.98	0.82	0.77	0.33	0.06	0.44	36.51	0.37	28.95	16.61	44.06	35.21	6.48	14.37
test_wrapper.v1 (extract)	**23.10**	**0.86**	**0.85**	**0.24**	**0.06**	**3.04**	**1.02**	**0.09**	**0.77**	**2.21**	**2.76**	**3.98**	**1.94**	**5.28**

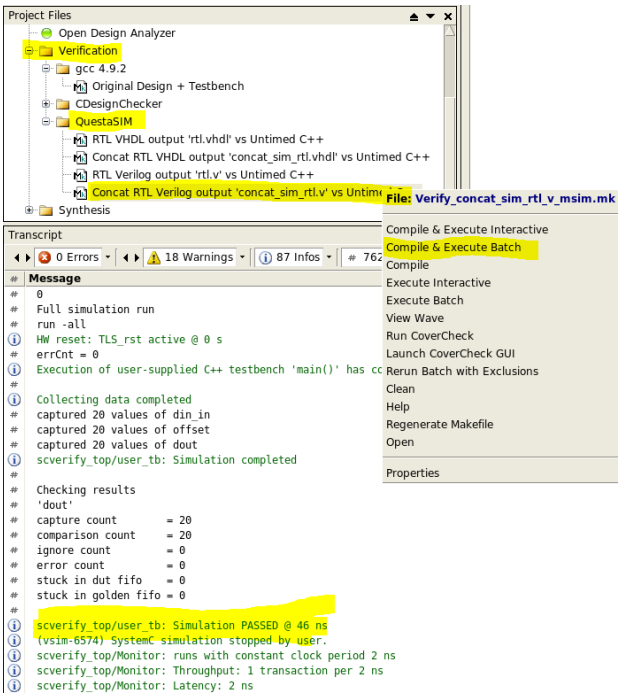
14. Go the Project Files > Verification folder and run the C++ design and testbench in g++.

    The testbench compares the original design against the modified code.



Look at the transcript and you should see the testbench reports 0 errors.

15. Run SCVerify in batch mode to verify that C and RTL match.



16. Close Catapult.

DONE LAB 3

Lab 4: Working with Memories

Overview

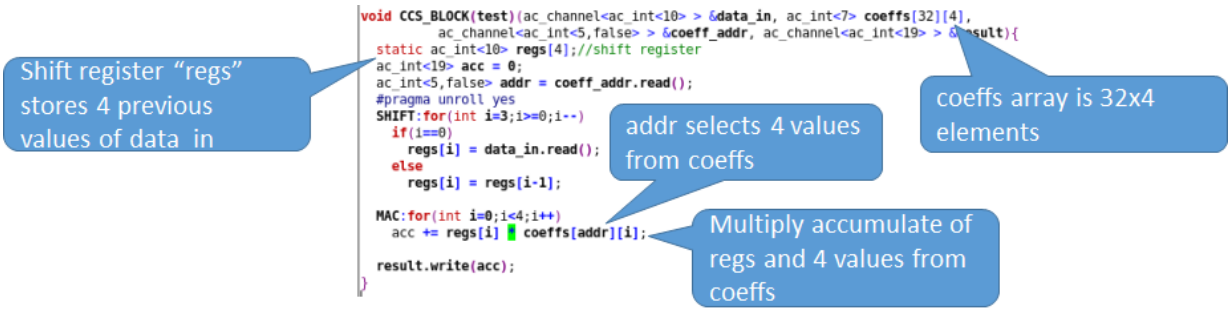
In this lab you will learn how to:

- Map C++ arrays to memories
- Analyze and debug pipeline failures due to memory resource conflicts
- Use memory word-width, interleave, and block_size constraints to improve performance

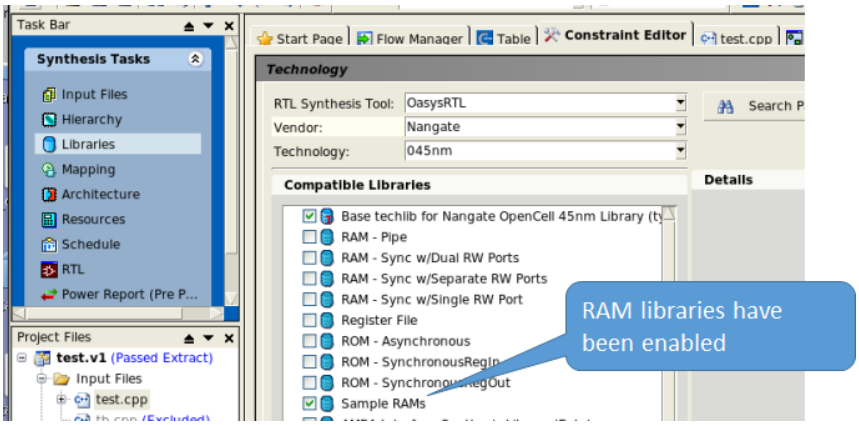
Steps

1. CD to the ‘lab4’ directory.
2. Launch Catapult by typing “catapult” at the terminal prompt.
3. Go to File > Run Script and select “***directives_mem_merge.tcl***” file and click open.

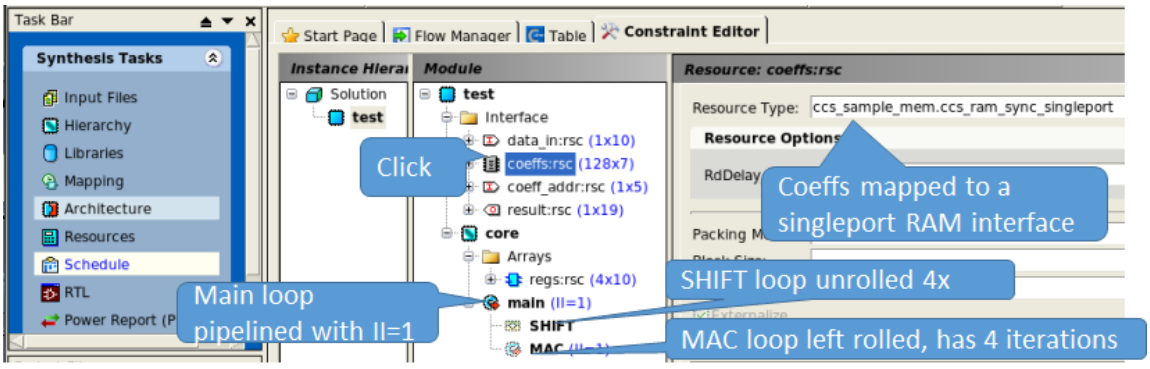
   This will load input files and compile and synthesize a design that uses Catapult memory libraries to map arrays to memories.
4. In Project Files > Input Files folder double-click on *test.cpp*. Note the following:
   - Data “data_in” is streamed from an ac_channel and shifted into 4-element shift register
   - “coeffs” is an array on the design interface with 32x4 elements.
   - The MAC loop has 4 iterations and multiplies the shift register against 4 values from coeffs selected by “addr”



5. Click on Libraries in the Task Bar and note that the RAM libraries have been selected.



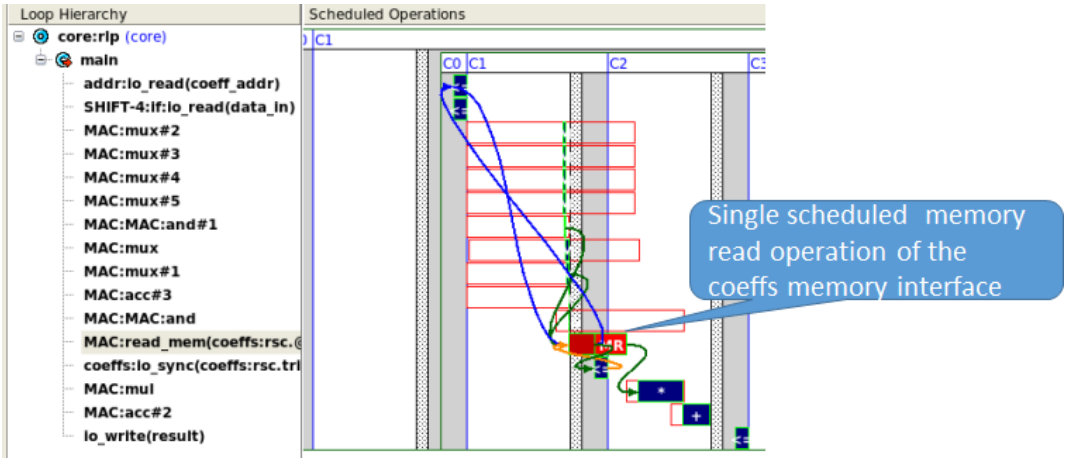
6. Click on Architecture in the Task Bar and note:
   - The coeffs array has been mapped to a singleport RAM interface
   - The main loop is pipelined with II=1.
   - The SHIFT loop is fully unrolled
   - The MAC loop is left rolled and has 4 iterations.



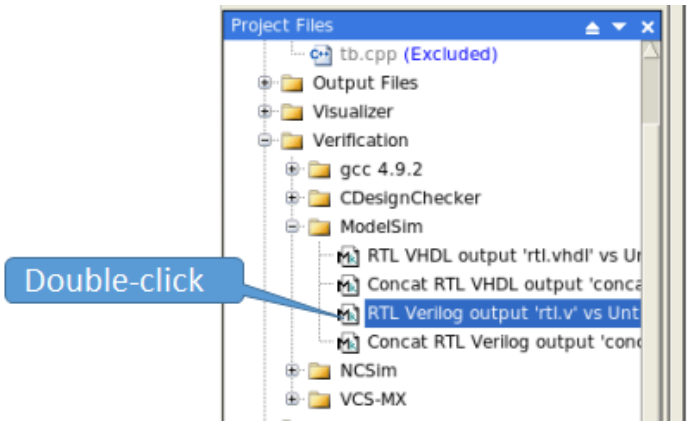
7. Look at the Table View and note that the design has a latency of 5 and a throughput of 4.

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

   Note that there is a single memory read operation of the “coeffs” singleport memory



9. Go to Project Files > Verification and launch SCVerify for you RTL simulator

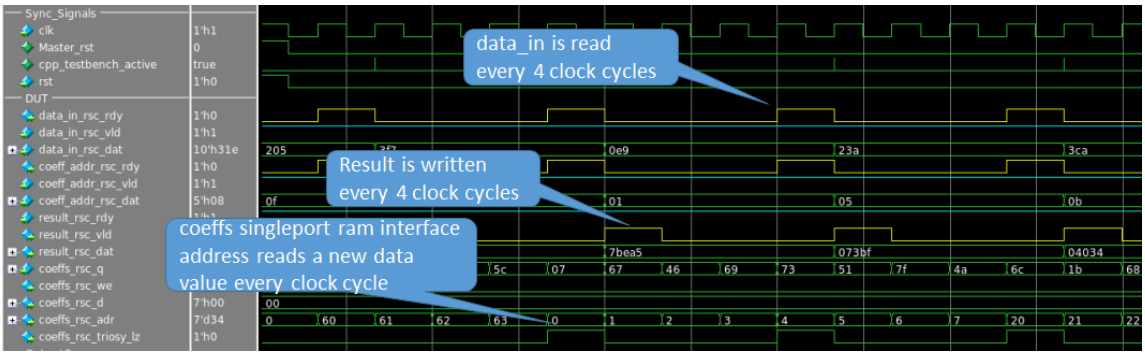


10. Enter run –all at the RTL simulator command prompt.

11. Look at the waveform view and note the following:

    - The input data_in is only read every 4 clock cycles since the MAC loop takes 4 iterations to complete.

    - The coeffs memory interface is read every clock cycle. You can see the address toggling every clock cycle.

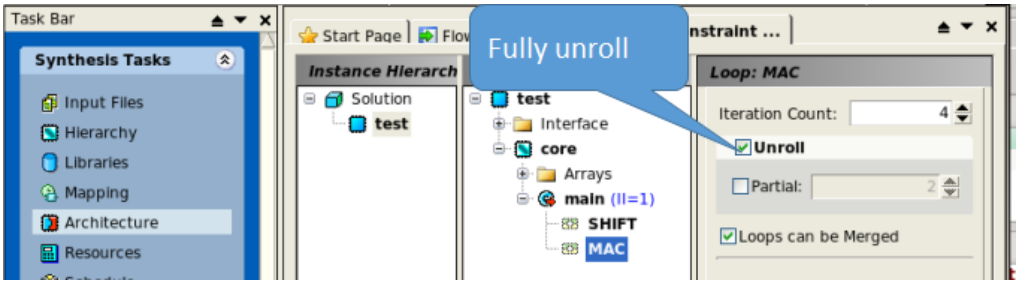


12. Close the RTL simulator.

Parallelism must be added to the design in order to read “data_in” every clock cycle. In this case loop unrolling should be used since the design is already fully pipelined.

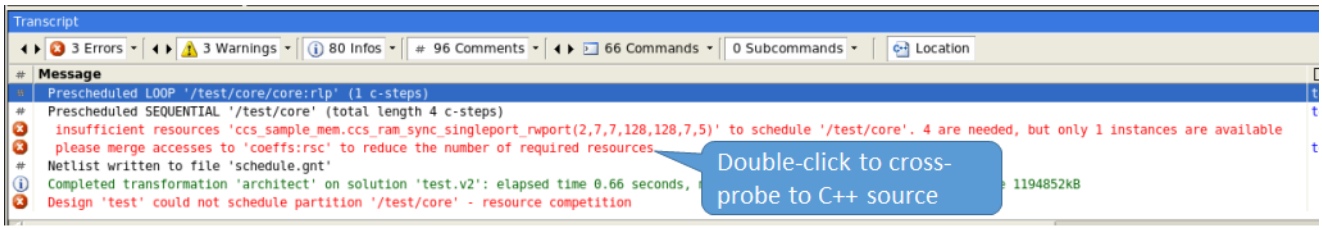
Overconstrain the memory resource

1. Click on Architecture in the Task Bar.
2. Click on the MAC loop and set loop unrolling to fully unroll the loop.



3. Click on RTL in the Task Bar
4. Look at the Catapult Transcript and note:
   - The tool has generated a resource competition error. This indicates that the pipeline initiation interval (II) has been set so that the design schedule is attempting to read or write to a memory or IO resource more times in a clock cycle than there are ports on the resource.
   - The error indicates that it needs 4 singleport RAMS, but only 1 is available.
   - The error indicates that the “coeffs_ram” resource is the problem and that the accesses to this memory need to be merged in order to pipeline the design with the current II.

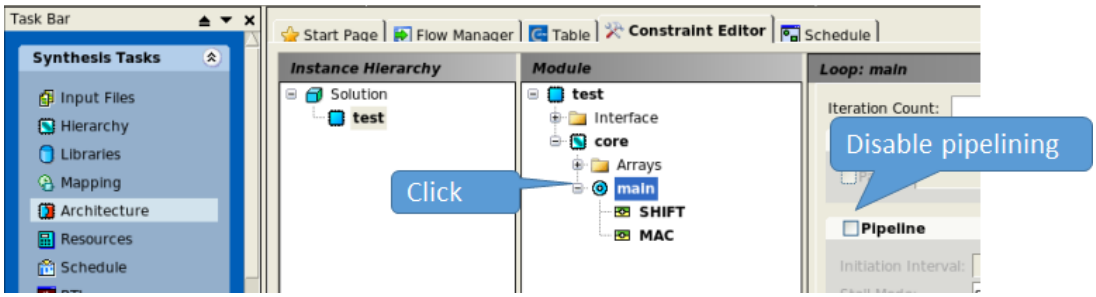
- Double-click on “please merge accesses to coeffs…” to cross probe to the source and note that the “coeffs” array on the top-level function interface is selected.




```
3  void CCS_BLOCK(test)(ac_channel<ac_int<10> > &data_in, ac_int<7> coeffs[32][4],
4            ac_channel<ac_int<5,false> > &coeff_addr, ac_channel<ac_int<19> > &result){
5    static ac_int<10> regs[4];//shift register
6    ac_int<19> acc = 0;
7    ac_int<5,false> addr = coeff_addr.read();
```

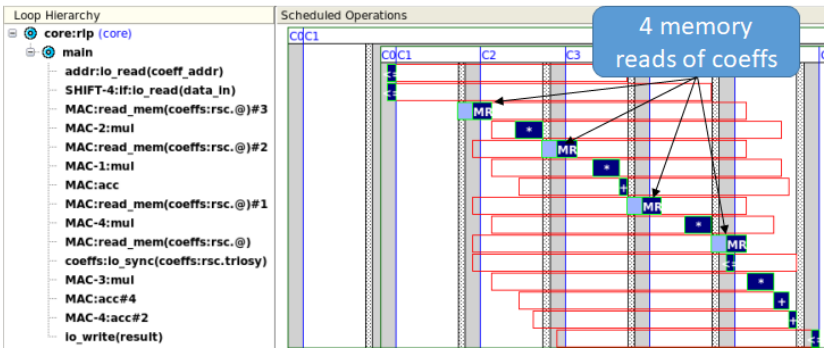
One of the best ways to debug these types of errors is to disable pipelining and reschedule the design so that the Gantt chart can be used to analyze the resource bottleneck.

5. Go to architectural constraints and turn off pipelining.



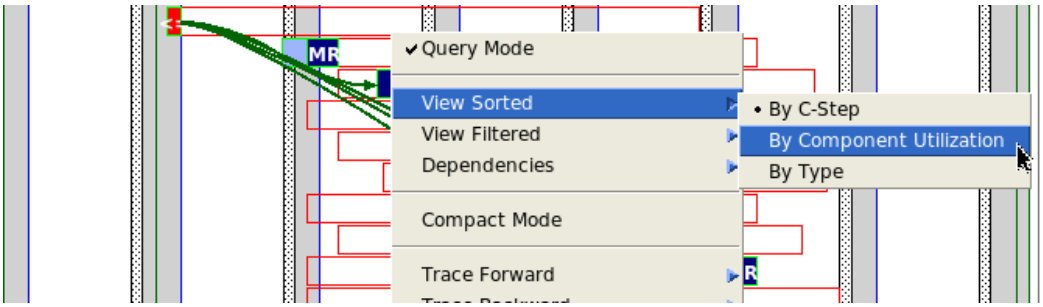
6. Click on Schedule in the Task Bar and expand the Gantt chart.

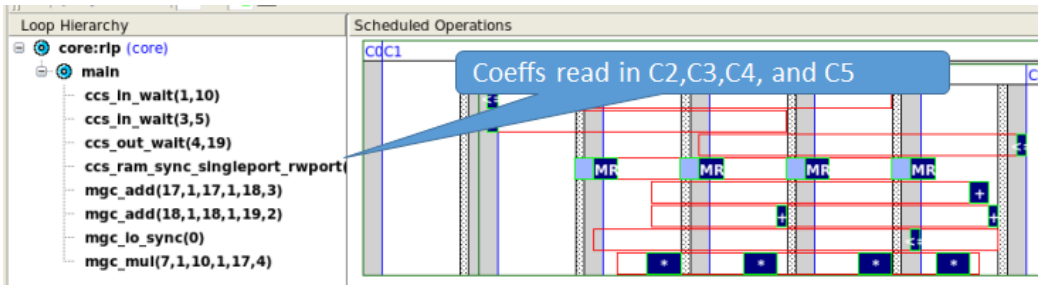
   Note that there are 4 memory read operations of “coeffs”.



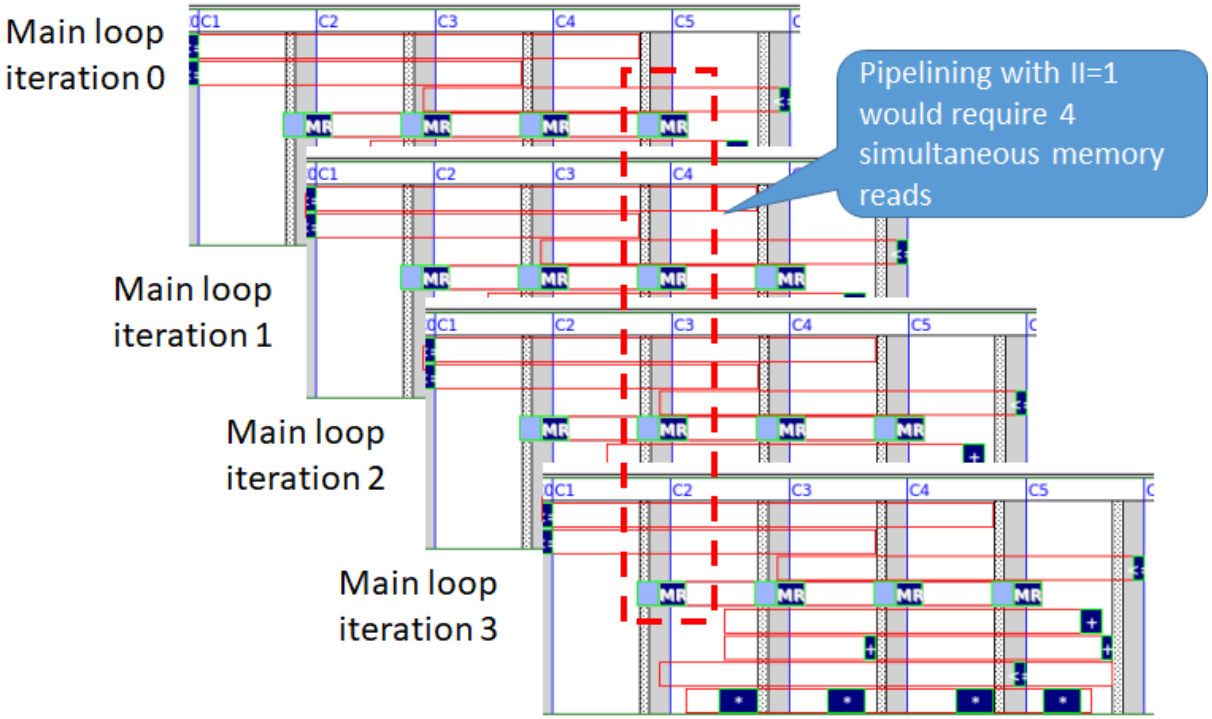
7. Right-click in the Gantt chart and select “View Sorted > By Component Utilization.

This will display the schedule so that reads/writes a resource are displayed in the same row. You can see that “coeffs” is read in C2, C3, C4, and C5. Unrolling the MAC look that has the read to coefs[addr][i] has replicated the read 4x.



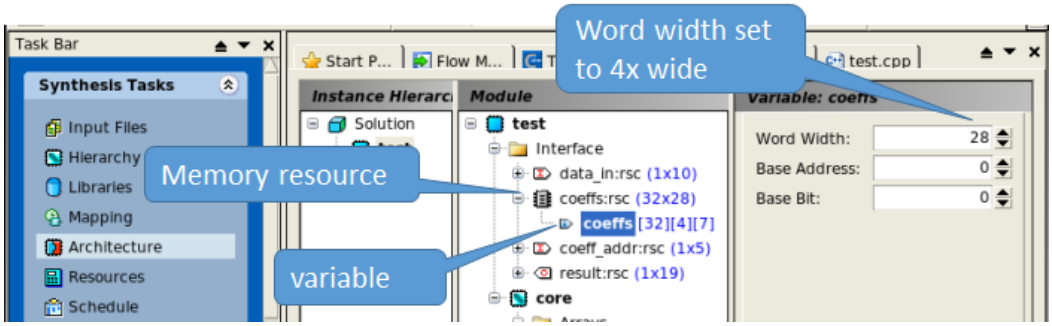


Pipelining the main loop with II=1 means that a new iteration of the main loop must start every clock cycle. This can be visualized by imagining the schedule of the main loop being overlapped so that a new iteration starts every clock cycle. This is illustrated below. It clearly shows that II=1 would require reading the “coeffs” mem 4 times in a c-step or clock cycle.



8. Go to architectural constraints and do the following:

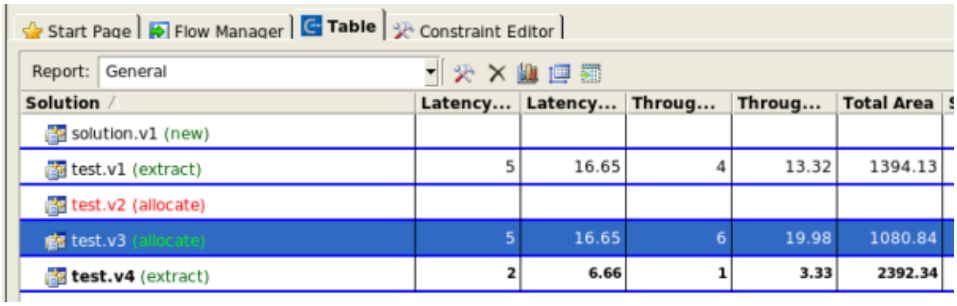
   - Select the coeffs:rsc interface and click on the “+”
   - Select the “coeffs” variable under the resource and set the word width to 28. This is 4x wider than the data type width of 7-bits. This should allow Catapult to read all 4 values of coeffs used in the MAC loop in parallel from a 4x wide memory.



Pipeline the Main loop with II=1.

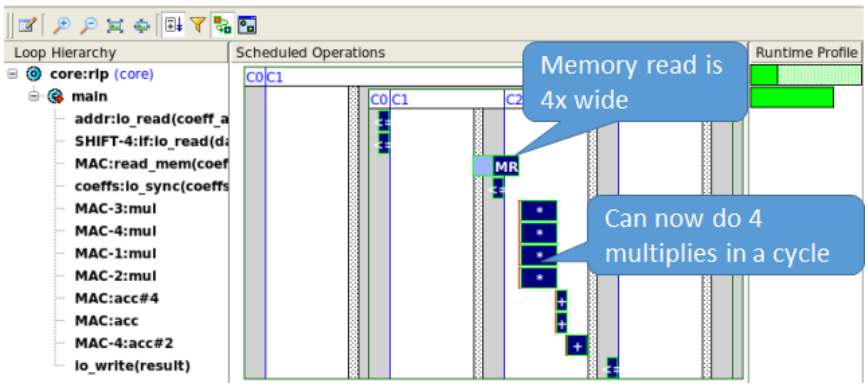
9. Click on RTL in the Task Bar.

   In the Table view, note that the throughput is now equal to 1.



10. Open the Gantt chart

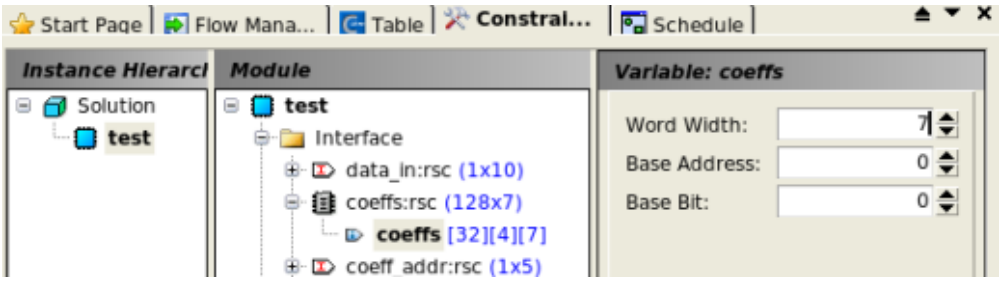
    Observe that the schedule now has a single memory operation. The 4 memory reads have been merged into a single wide memory read.



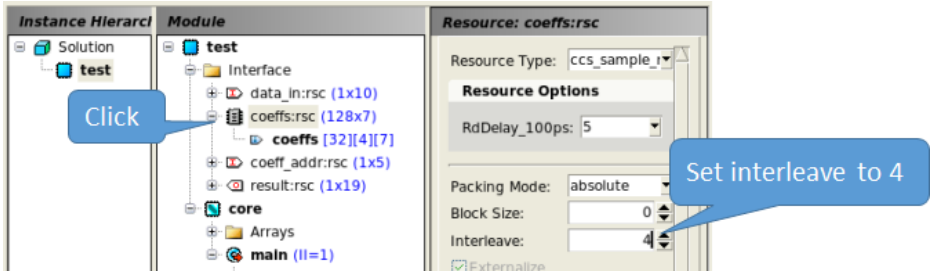
Instead of making the memory wider we could have also taken of advantage of the coeffs array accesses. The read of coeffs[addr][i] in the unrolled mac loop means that we are reading coeffs[addr][0], coeffs[addr][1], coeffs[addr][2], and coeffs[addr][3] in parallel. Alternatively we could use interleaving constraints to split the coeffs memory into 4 separate memories.

11. Go to architectural constraints and do the following:

a. Select the coeffs variable under the coeffs resource and set the word-width back to 7-bits, which is the original width.

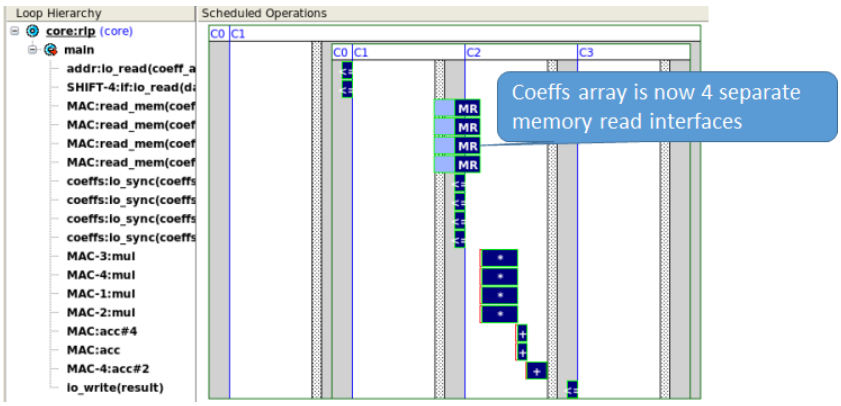


b. Click on the coeffs:rsc resource and set interleaving to 4.



12. Click on Schedule in the Task Bar and expand the Gantt chart.

The coeffs array has now been “interleaved” into 4 separate memory interfaces.

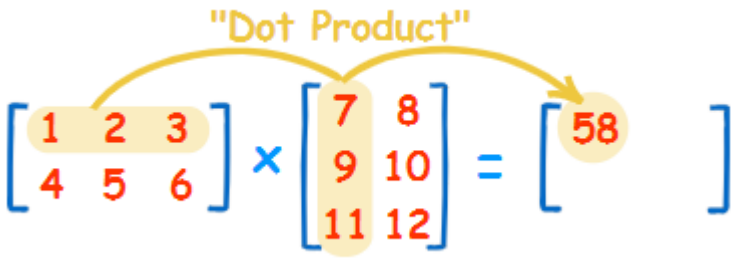


13. Close Catapult

DONE LAB 4

Lab5: Multi-Block Design

During matrix multiplication of matrices A X B, the first row of the matrix A is multiplied with the corresponding elements of the first column of matrix B and then added.



Matrix Multiplication

Matrices are usually arranged by raster order in memory. This means that the matrix elements are arranged sequentially row by row. Consider A and B matrices that are streamed in raster order. This can cause a bottleneck in performance due to non-sequential accesses to the matrix B column data. The best performance can be achieved by transposing the matrix B and then performing row-by-row multiplication of the matrices.

Overview

In this lab you will learn how to:

- Analyze performance issues for sequential loops
- Build a multi-block matrix multiplication design to improve performance
- Learn how to code for using write-mask memories

Part 1 - Single block Matrix Multiplier

This design performs a matrix multiplication of two 8x8 matrices using signed data. The first implementation we will look at does the entire design as a single block design. The best-case throughput achievable is 128 cycles.

1. CD to the ‘lab5’ directory

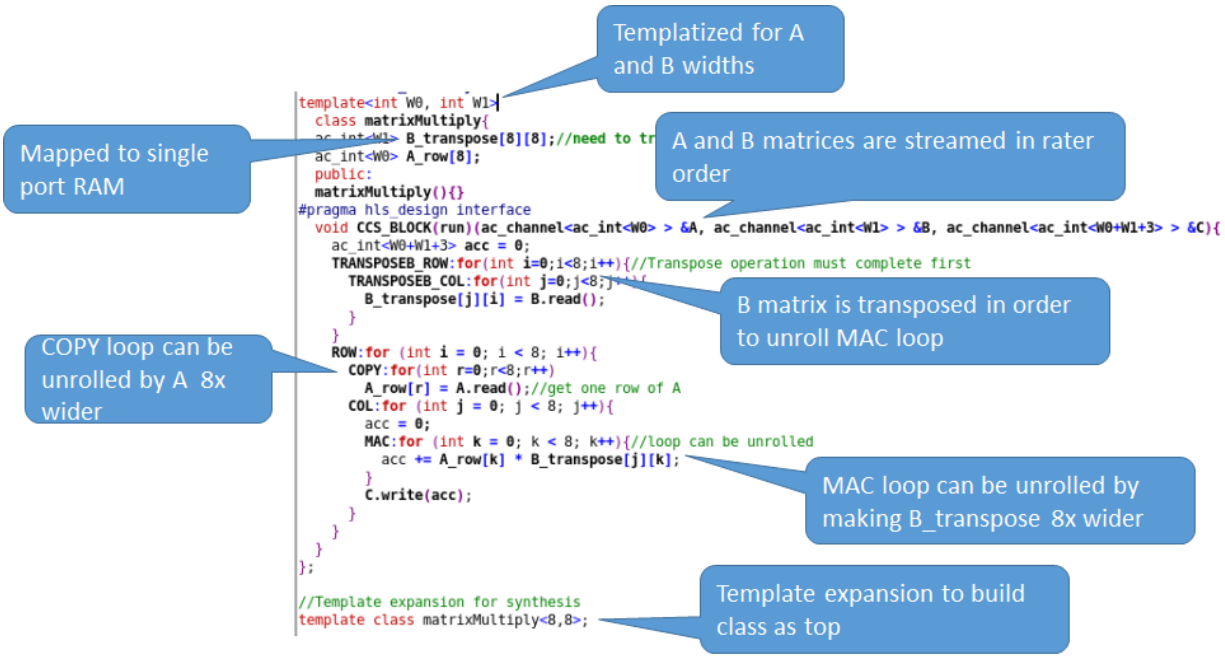
2. Type “catapult” at the terminal prompt

3. Go to the File Menu > Run Scripts and select “run_single_block.tcl”.

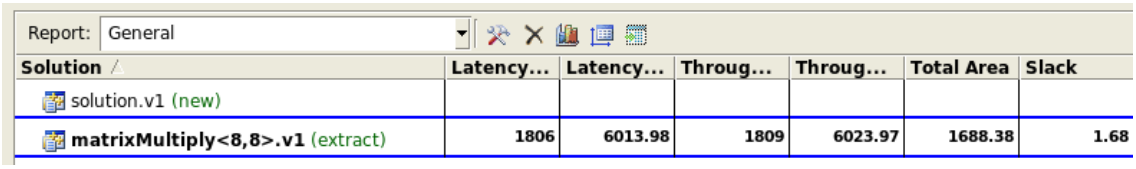
   This will add the input files and testbench and generate RTL on the unconstrained design.

4. Go to the Project Files View and open the “test_single_block.h” file and note the following:

   - The design is a class-based design that is templatized to specify the bit-widths of A and B matrices
   - The A and B matrices are streamed in raster order.
   - There is an internal array B_transpose that is used to transpose the B matrix so that loop unrolling can be used on the MAC loop.
   - The COPY and MAC loops can be unrolled if the A channel interface and B_transpose memory are made 8x wider.



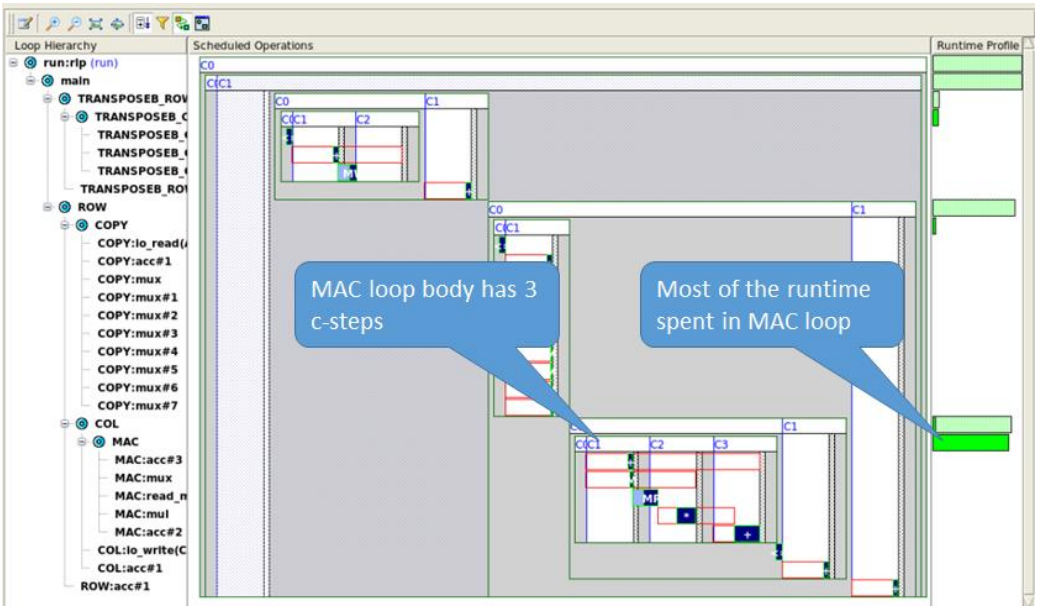
5. Go to the table view and note the performance of the unconstrained design. The throughput is 1809 cycles.



Solution	Latency...	Latency...	Throug...	Throug...	Total Area	Slack
solution.v1 (new)						
matrixMultiply<8,8>.v1 (extract)	1806	6013.98	1809	6023.97	1688.38	1.68

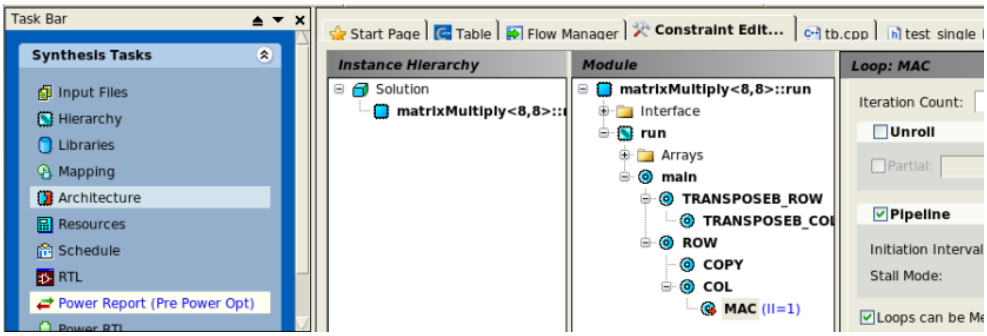
6. Click on Schedule in the Task Bar

7. Expand the Gantt Chart and note that most of the runtime (Dark green bar) is spent in the MAC loop. Also, note that the MAC loop body has three c-steps.



Pipeline the MAC loop in new solution

8. Click on Architecture in the Task Bar and pipeline the MAC loop with II=1.



9. Click on RTL in the Task Bar.

10. Go to the Table View and look at the throughput. Note that the throughput has improved a lot, and there has also been a slight reduction in area. This will sometimes happen when pipelining a design.

Report: General

Solution	Latency...	Latency...	Throug...	Throug...	Total Area	Slack
solution.v1 (new)						
matrixMultiply<8,8>.v1 (extract)	1806	6013.98	1809	6023.97	1688.38	1.68
matrixMultiply<8,8>.v2 (extract)	**846**	**2817.18**	**849**	**2827.17**	**1622.33**	**0.30**

11. Click on Schedule in the Task Bar and expand the Gantt Chart.

    Note that most of the time (Big dark green bar) is still being spent in the MAC loop. The MAC loop body reads the B_transpose memory and multiplies against a row from A. The MAC loop has 8 iterations. So if we want to do all 8 multiplies in parallel then we need to make the B_transpose memory 8x wider.

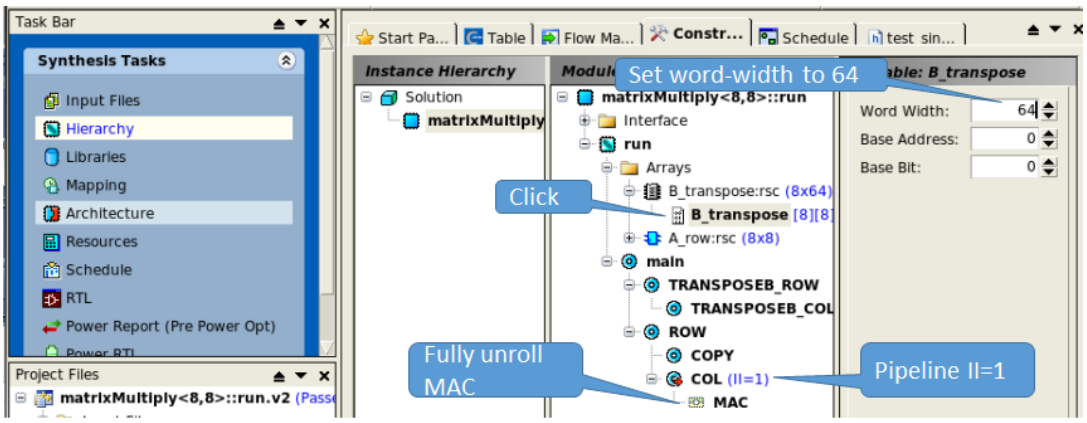
Adjust loops and memories in new solution

12. Click on Architecture and:

    a. Fully unroll the MAC loop

    b. Pipeline the COL loop with II=1

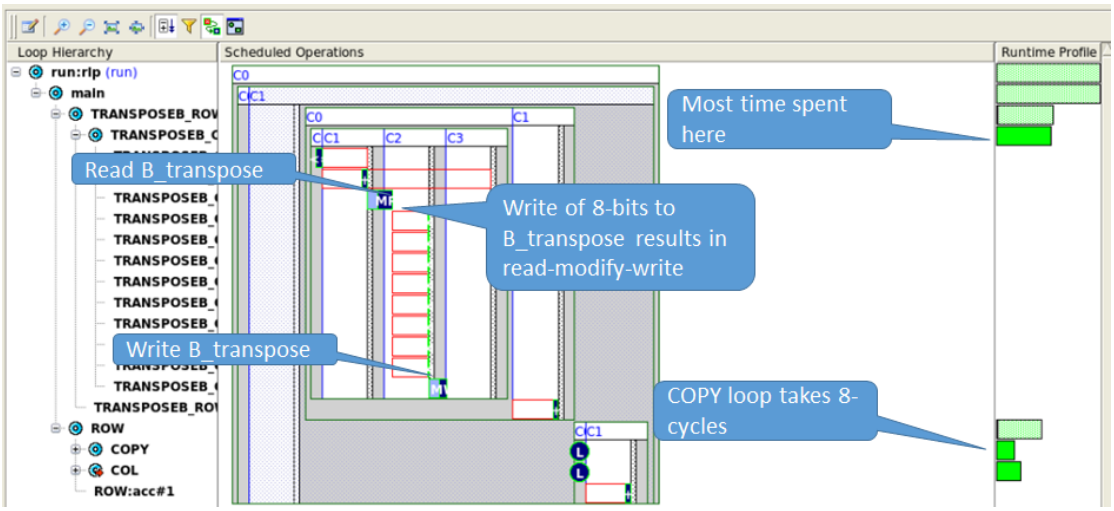
    c. Make the word-width of the B_transpose memory 64 bits.



13. Click on Schedule in the Task Bar and expand the Gantt Chart and note:

    - Most of the time is now spent in the TRANSPOSE loops.
    - A read-modify-write is required because the transpose loop writes 8-bits into a 64-bit wide single port memory. The memory is 64-bits wide because the word-width was widened to fully unroll the MAC loop. This means that the transpose loop can only be pipelined with II=1 because it is a singleport memory.

- The COPY loop is also spending time copying 8 values (1 row) of A.



Adjust loops and memories in new solution

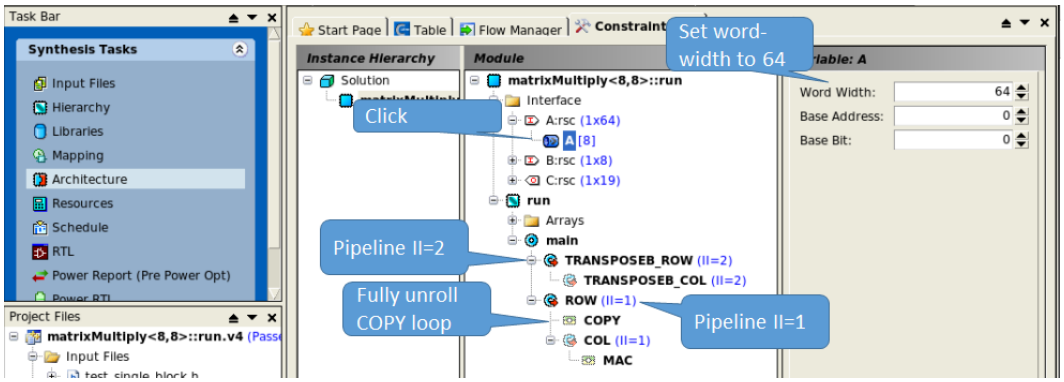
14. Click on Architecture and do the following:

a. Pipeline the TRANSPOSEB_ROW loop with II=2.

b. Fully unroll the COPY loop.

c. Pipeline the ROW loop with II=1.

d. Set the word-width of the A interface variable to 64 bits.



15. Click on Schedule in the Task Bar and then expand the Gantt Chart.

Note that most of the time is now spent in the TRANSPOSE loops. The read-modify-write is costing performance.

16. Look at the table view and note the throughput is 196 cycles.

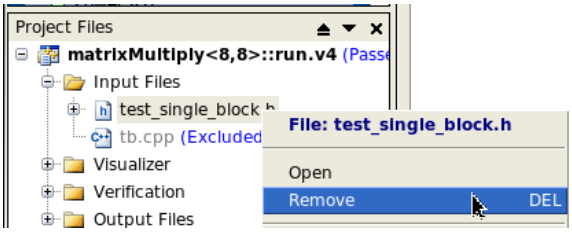
In order to get to 128 cycles we need to be able to pipeline the entire design with II=1. This is not currently possible because of the read-modify-write in the transpose loop. To get to II=1 we need a memory with write masking that will allow us to avoid a read-modify-write.

Report: General

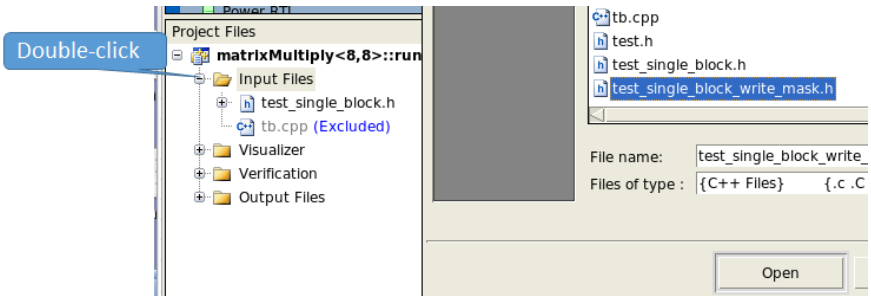
Solution	Latency...	Latency...	Throug...	Throug...	Total Area
solution.v1 (new)					
matrixMultiply<8,8>.v1 (extract)	1806	6013.98	1809	6023.97	1688.38
matrixMultiply<8,8>.v2 (extract)	846	2817.18	849	2827.17	1622.33
matrixMultiply<8,8>.v3 (allocate)	358	1192.14	361	1202.13	4600.89
matrixMultiply<8,8>.v4 (allocate)	**194**	**646.02**	**196**	**652.68**	**5106.34**

Use different memory component in new solution

17. Select the test_single_block.h file in the Input Files Folder and select remove.

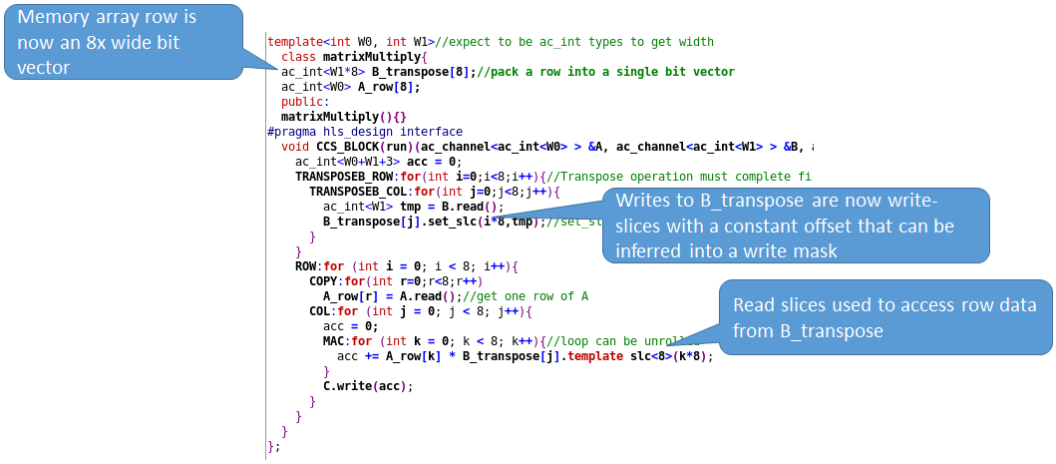


18. Double-click on the Input Files Folder and add the test_single_block_write_mask.h file.

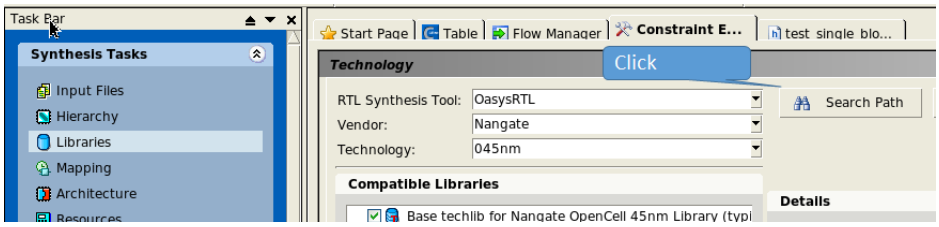


19. Open “test_single_block_write_mask.h” and note the following:

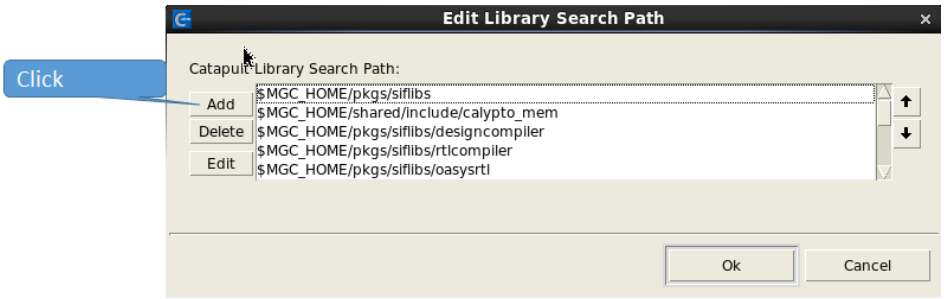
- The B_transpose array has been recoded to be a single dimension array with 8x wide bit-width. So an entire row of B_transpose is packed into a bit-vector.
- The writes to B_transpose use the “set_slc” method using a constant offset. This will allow catapult to infer this into the correct write mask for a write mask memory.



20. Click on Libraries in the Task Bar and click on the “Search Path” button.



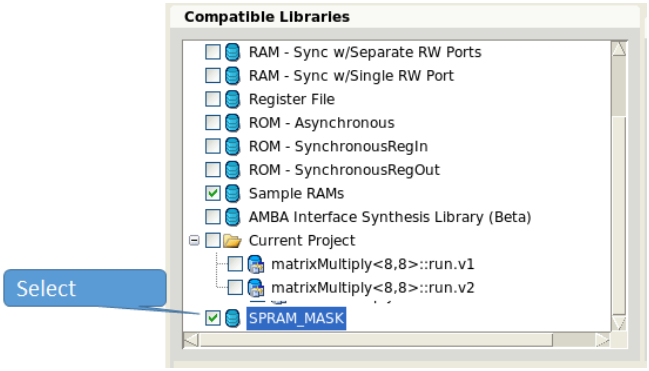
21. Click on “Add” in the popup dialog.



22. Click on OK in the “Select Directory” pop-up dialog.

This will set a library search path to the Lab5 directory. There is a SPRAM_MASK.lib Catapult memory library in this directory that is a write mask memory. There is also SPRAM_MASK.v which is the Verilog simulation model for the memory.

23. In the Compatible Libraries View select and apply the SPRAM_MASK library.

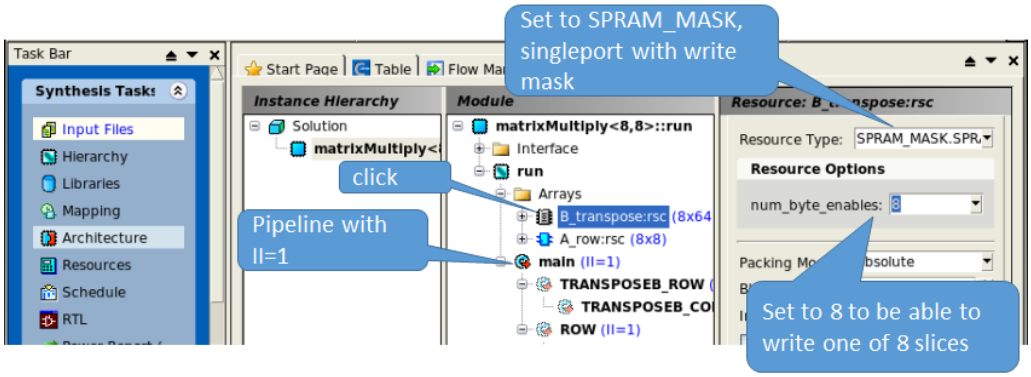


24. Click on Architecture and do the following:

a. Click on B_transpose:rsc and set the Resource Type to SPRAM_MASK. This is a singleport memory with write masking.

b. Set the num_byte_enables to 8. “num_byte_enables” is the number of fixed width word lanes that can be masked on the memory. So on a 64-bit memory this will give 8-lanes of 8-bits each.

c. Pipeline the entire design with II=1



25. Click on Schedule in the Task Bar.

26. Look at the Table View and note that the Throughput is now 128.

This is the best performance that can be achieved with this single block implementation. The performance is limited because the TRANSPOSE_ROW/TRANSPOSE_COL loops must run to completion before the ROW/COL/MAC loops can run. This is because these are sequential loops that cannot be automatically merged by Catapult. The design must be re-written to be a multi-block design to allow the TRANSPOSE_ROW/TRANSPOSE_COL and ROW/COL/MAC loops to run concurrently.

Report: General

Solution	Latency...	Latency...	Throug...	Throug...	Total Area
solution.v1 (new)					
matrixMultiply<8,8>.v1 (extract)	1806	6013.98	1809	6023.97	1688.38
matrixMultiply<8,8>.v2 (extract)	846	2817.18	849	2827.17	1622.33
matrixMultiply<8,8>.v3 (allocate)	358	1192.14	361	1202.13	4600.89
matrixMultiply<8,8>.v4 (allocate)	194	646.02	196	652.68	5106.34
matrixMultiply<8,8>.v5 (allocate)	**130**	**432.90**	**128**	**426.24**	**4570.08**

Part 2 – Multi-block Matrix Multiplier

1. Close any current sessions of Catapult and invoke Catapult to start a new project.

2. Go to the File Menu > Run Script and select “run_multi_block.tcl”. This will create a new project and add the file “test_multi_block.h” and testbench “tb.cpp”, set the design constraints, and synthesize the design. This design uses class-based hierarchy to improve the performance of the matrix multiply design.

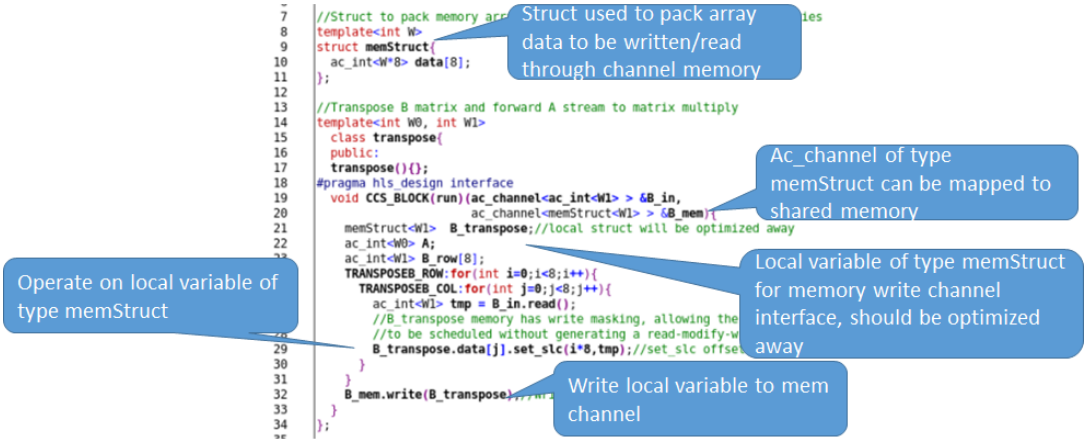
3. Open the test_multi_block.h file in the Catapult editor and note the following:

   a. The design has been implemented using three classes mapped to design blocks, transpose class, matrixMultiply class, and matrixMultiplyTop class.

   b. The transpose class transposes the B matrix.

      It uses the required coding style for class-based hierarchical design where a struct is used to pack the memory array to be written through an ac_channel.

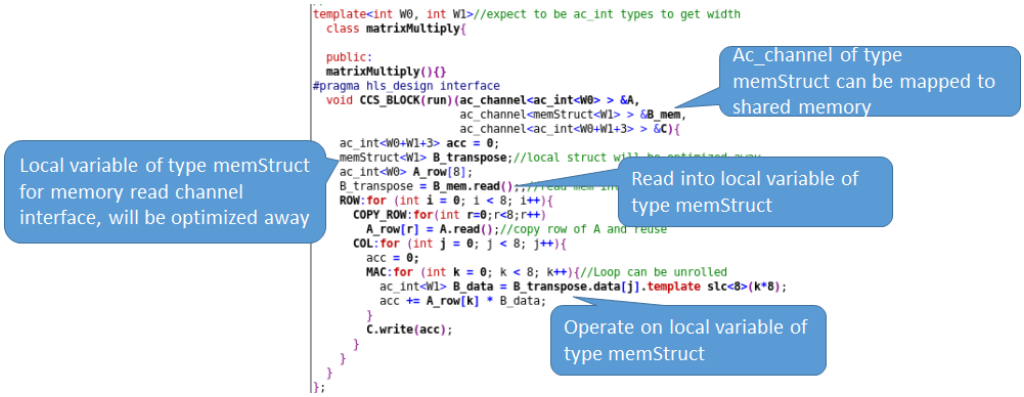
      There is a local variable of type memStruct that is written to. This variable is then written into the ac_channel. This style will allow Catapult to optimize the local variable away.



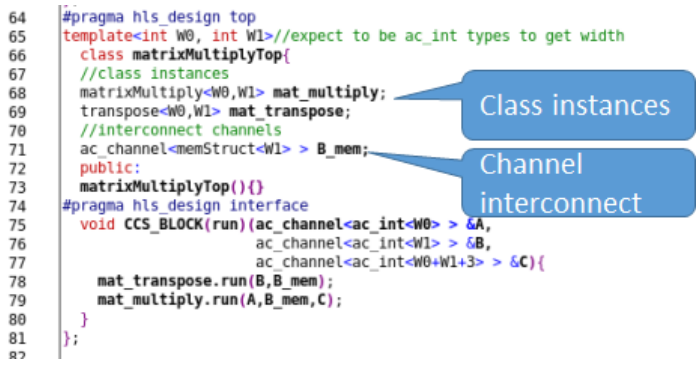
c. The matrixMultiply class does the multiple-accumulate of the rows of the transpose matrix and the A channel.

   It uses the required coding style for class-based hierarchical design where a struct is used to pack the memory array to be read from an ac_channel.

   There is a local variable of type memStruct used to read form the ac_channel. This variable is then used locally. This style will allow Catapult to optimize the local variable away.

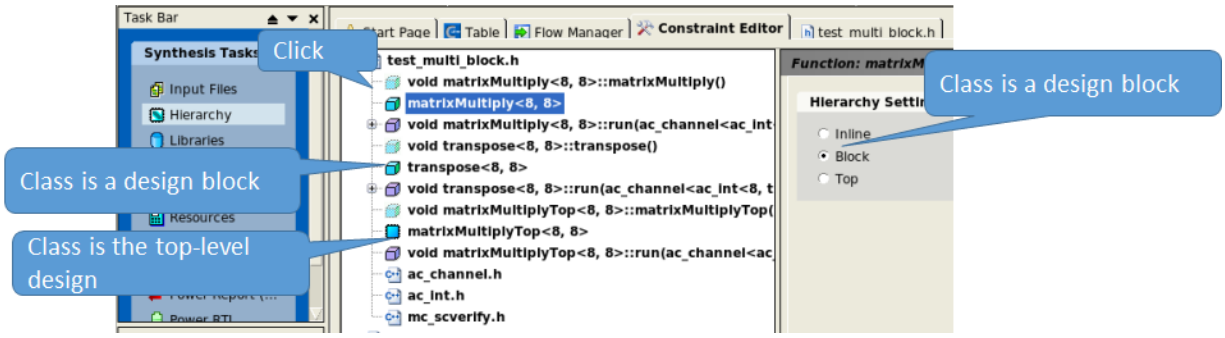


The matrixMultiplyTop class is used to stitch the transpose and matrixMultiply classes together.



4. Click on Hierarchy in the Task Bar.

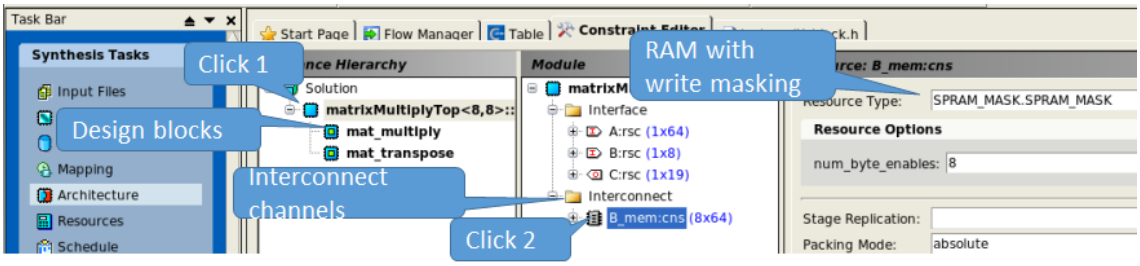
   Note that the transpose class and matrixMultiply classes are constrained as design blocks. The matrixMultiplyTop class is set as the top-level design.



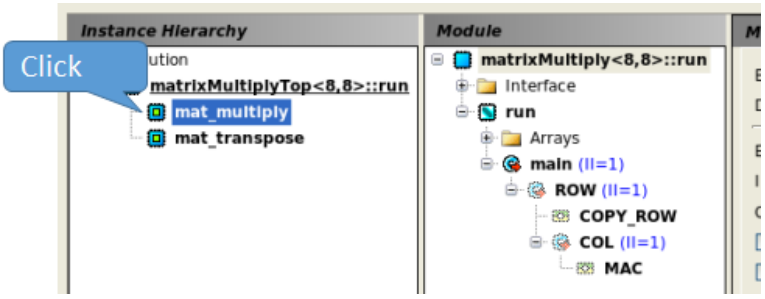
5. Click on Architecture in the Task Bar and note:

   - The transpose and matrixMultiply class instances are shown as separate design blocks.

   - The interconnect channel for B_mem has been mapped to a singleport RAM with masking.



   - Click on the mat_multiply and mat_transpose design blocks and note that they each have their own set of loops that can be constrained separately.

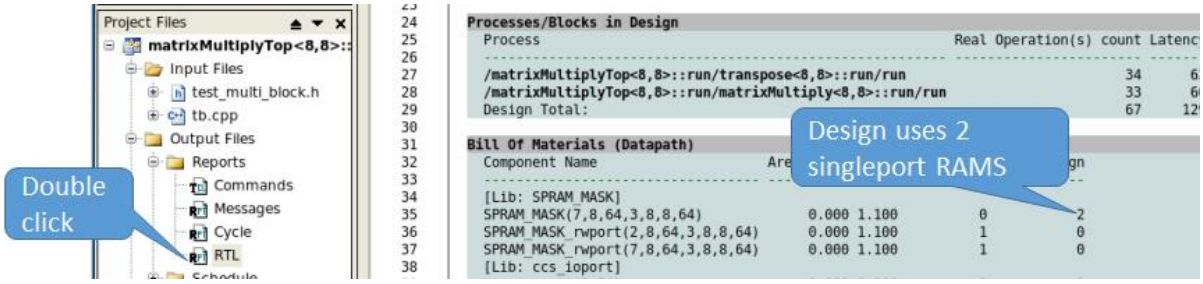


6. Look at the Table View and note that the throughput is now 64 cycles. This is because the transpose and matrixMultiply blocks can run concurrently.

Report: General

Solution	Latency...	Latency...	Throug...	Throug...	Total Area	Slack
matrixMultiplyTop<8,8>.v1 (compile)						
matrixMultiplyTop<8,8>.v2 (extract)	**129**	**429.57**	**64**	**213.12**	**6166.69**	**0.38**

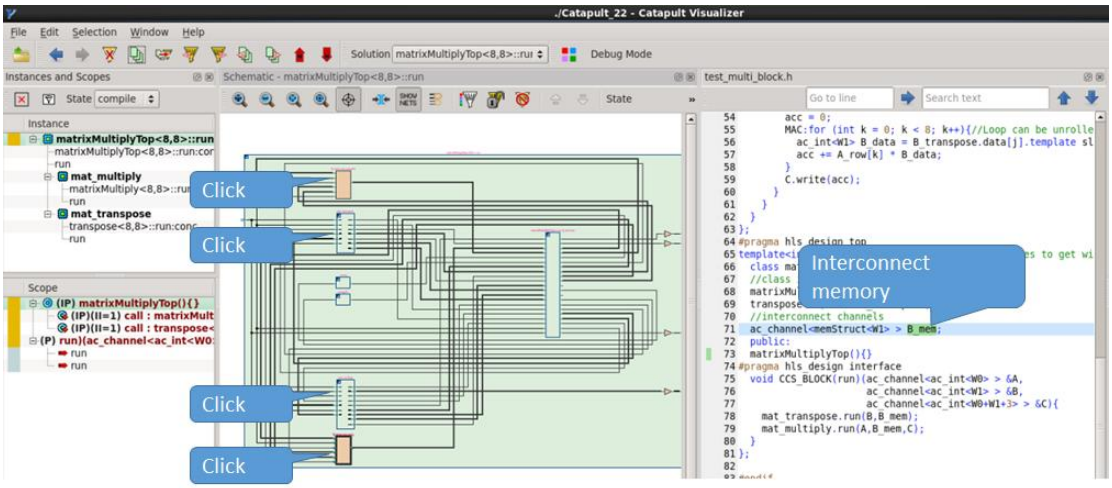
7. Double-click on the RTL.rpt file in the Output Files Folder. Note that the design is using 2 singleport rams for the shared memory. This is to allow them to operate in ping-pong fashion to achieve a throughput of 64 cycles.



8. Launch the Catapult Design Analyzer Tool and note the following:

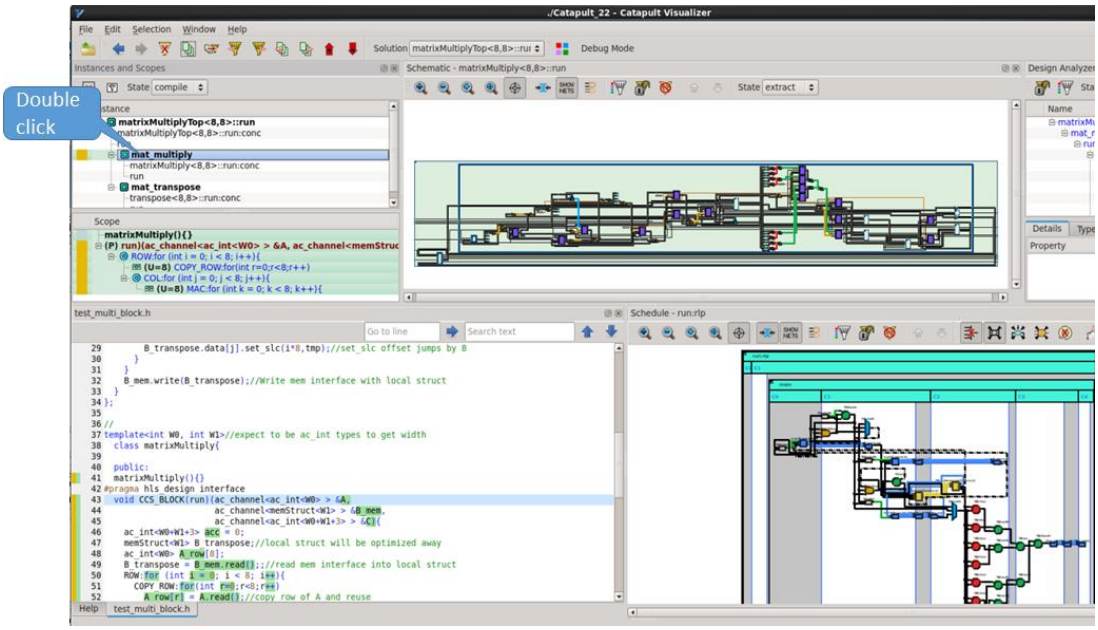
   - The top-level schematic has separate design blocks for the transpose and matrixMultiply instances. Clicking on them will highlight them in the C++ source.

   - There are two memory blocks. Click on them to see them in the source.



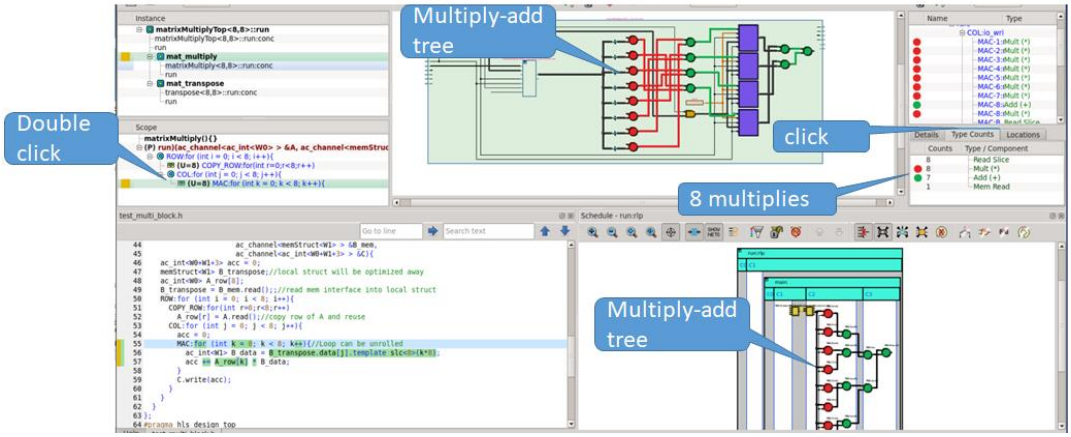
   a. Double-click on the mat_muliply instance in the ‘Module’ view.

      This will filter to the entire design block. You can see the Schematic, C++ editor, and Schedule views update.



b. Double-click on the MAC loop in the Scope view.

This will filter to just what is under the MAC loop. Note that this filters to 8-multipliers and an adder tree in the views.



c. Close Design Analyzer

DONE LAB 5

Lab 6: Shared Memories

Overview

In the previous lab (Lab 5) we saw how to code a multi-block design with a shared memory between blocks. This style allows Catapult to infer a ping-pong memory structure between blocks for best performance. However, it is limited in that the memory accessed are unidirectional. One block can write the memory and one block can read the memory. If read-write access is needed by both blocks you either must combine the blocks into a single block with an internal memory, or else you need to code a standalone memory block that provides read-write access.

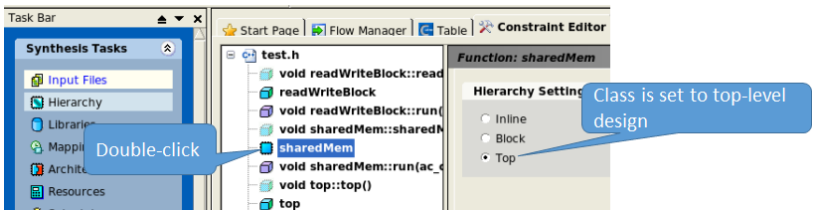
In this lab you will:

- Use non-blocking reads to code a shared memory block
- Use cycle constraints or ac_wait to manually schedule independent address and data channels to allow read transactions of the shared memory block to be pipelined.

Steps

1. CD to the ‘lab6’ directory
2. Type “catapult” at the terminal prompt
3. Go to the File Menu > Run Scripts and select “run_shared_mem_block.tcl”.

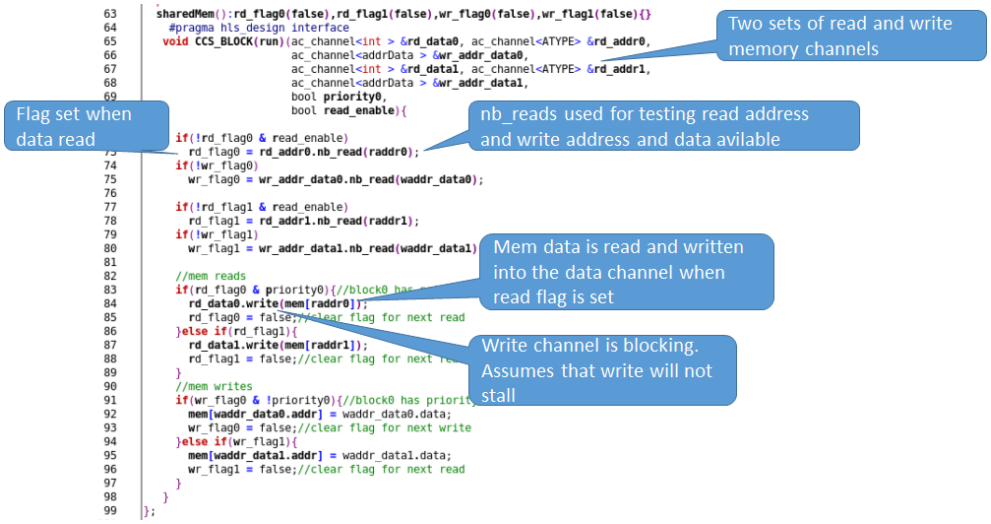
   This will add the input files and testbench and generate RTL on the unconstrained design. It will also make the “sharedMem” class the top-level design.
4. Click on Hierarchy in the Task Bar and note that the sharedMem class is set as the top-level design.



5. Double-click on the sharedMem icon to go to the C++ source.

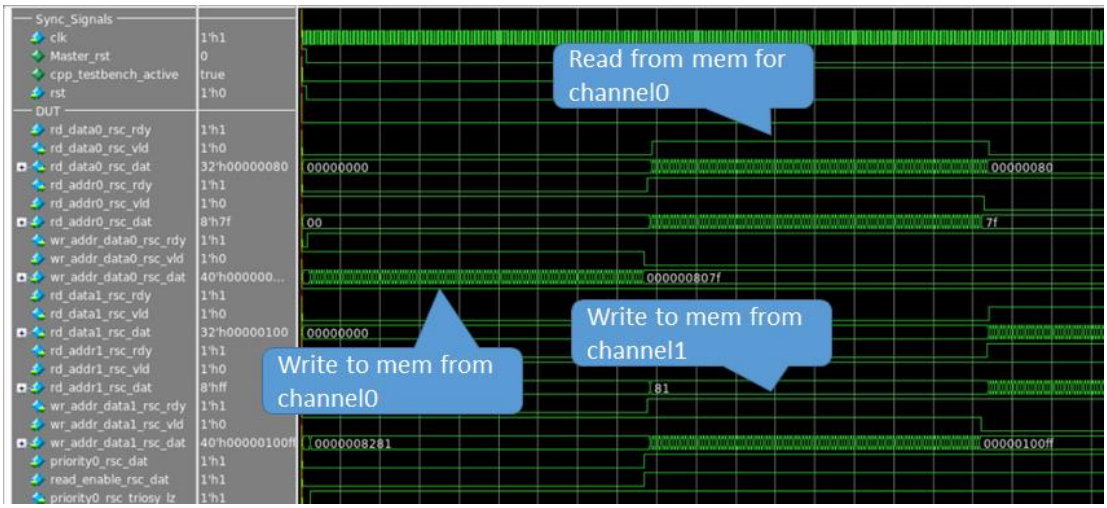
   Note the following about the sharedMem class:

   - This design block has two sets of read and write channels for reading/writing the internal memory “mem”.
   - The design uses nb_reads on all read channels to avoid stalling. The priority0 variable gives priority to one of the sets of channels over the other.
   - The channel writes are blocking and it is assumed that the write data will always be accepted. If this is not the case then nb_writes would be required which are covered in Lab7.



6. Go to the Project Files > Verification folder and launch SCVerify using your RTL simulator.

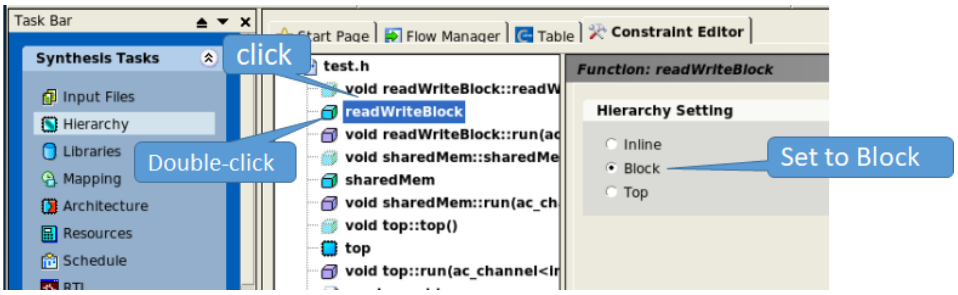
7. Run the verification and check the waveform view to verify that the memory interface channels can run independently.



8. Close the RTL simulator

9. In Catapult go to File >Run Script and select run_multi_block.tcl and click Open. This will synthesize the top-level design which has an instance of the sharedMem class and two read/write blocks to read and write the memory in sharedMem.

10. Click on Hierarchy in the Task Bar and note that readWriteBlock and sharedMem are set as type “Block” and top is set to “top”



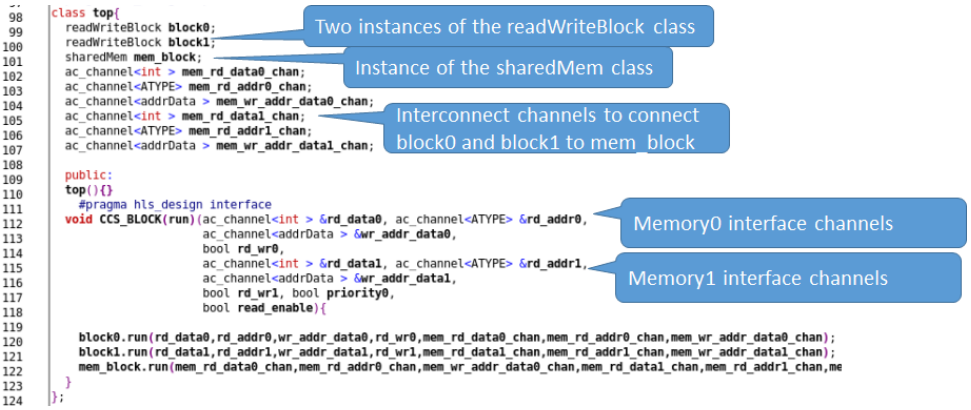
11. Double-click on the readWriteBlock class to cross-probe to the C++.

Note the following about the readWrite block.

- The block gets read or write requests from the top-level design interface and sends them to the sharedMem block.
- The reads and writes are mutually exclusive.
- The design uses blocking read() and write() and assumes that the hardware will not stall.
- On a read, the read address is sent to the sharedMem block and then the read data is read back from the sharedMem block.

12. Scroll down in the editor and look at the top-level design class “top” and note:

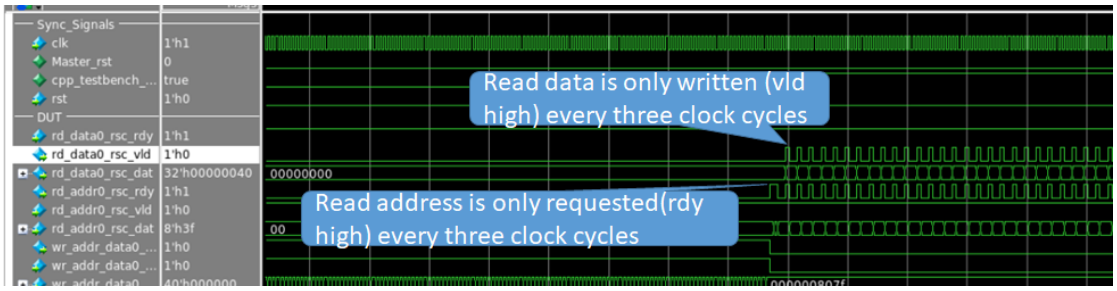
- The top-level class has two instances of the readWriteBlock class, block0 and block1 and one instance of the sharedMem class mem_block.
- There are two sets of interconnect channels to connect block0 and block1 to mem_block.
- The top-level design has two sets of memory interface channels that connect to block0 and block1.



13. Launch SCVerify Verification and run the simulation and note the following:

- The simulation passes.
- The performance is not 1 read/write per clock cycle
- The “rd_data_vld” signal is only going high every three clock cycles.

- The “rd_address_rdy” signal is only going high every three clock cycles. Something is preventing the read addresses from being pipelined.

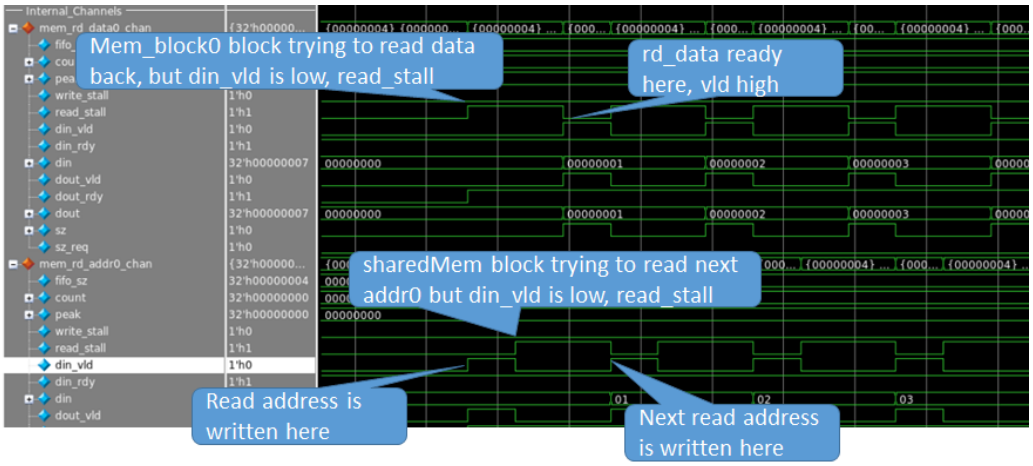


14. Scroll down in the waveform view and look at the Internal Channels.

This shows the interconnect channels between blocks. Expand the internal channel waveforms by clickin the “+”. Note the following:

- A single read address from mem_rd_addr0_chan is written into the channel (vld high) that connects to the sharedMem block which is ready to read the address (rdy high). The sharedMem block is immediately ready to read the next address but none is available so the “read_stall” flag goes high.

- The mem_rd_data0_chan attempts to read the data back from the sharedMem block at the same time the mem_rd_addr0_chan is written. There is no data available so the read_stall flag goes high. rd_data is available two cycles later (vld high) after which the next read address is written. So the next read address has to wait for the read data to be returned from the sharedMem block.

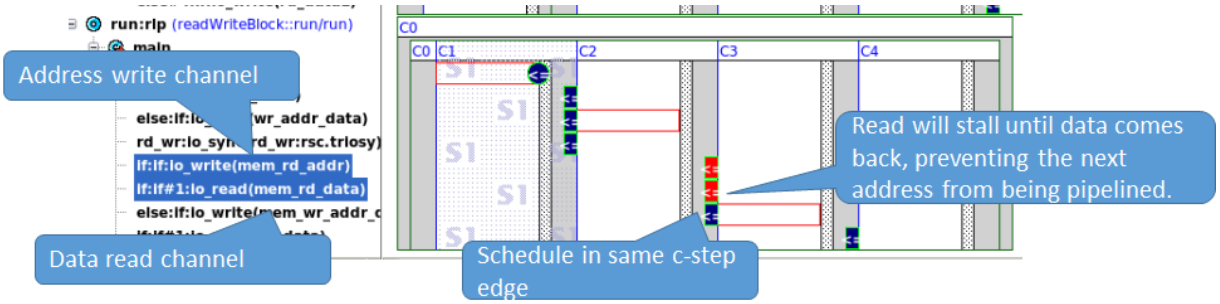


15. Close the RTL simulator.

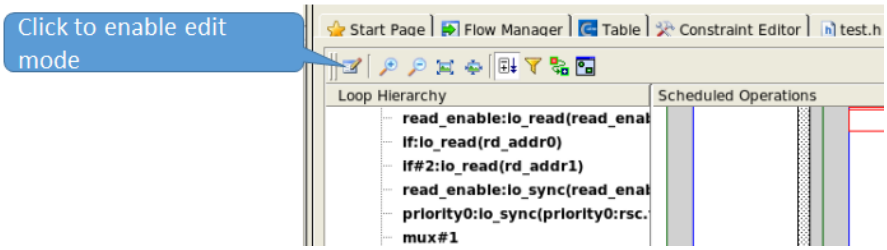
16. In Catapult click on Schedule in the Task Bar.

17. Look at the schedule for the readWriteBlock class and note:

- The rd_addr channel write and the rd_data channel read are scheduled on the same c-step edge.
- This means that the address write and data read will happen at the same time. The data read must wait till the address has been processed by the shareMemBlock class instance mem_block causing the hardware to stall.

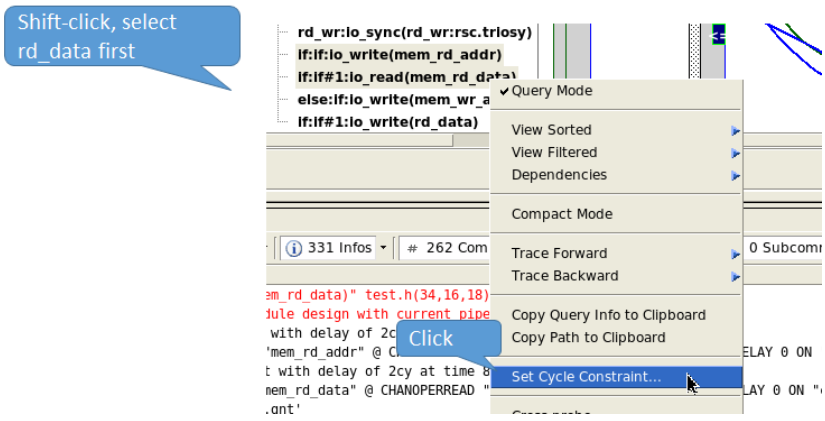


18. Click on the Edit button in the Gantt chart.



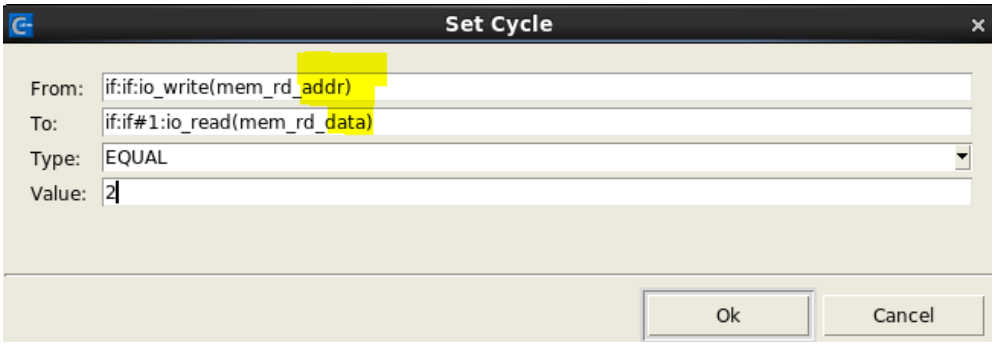
a. Click on the mem_rd_data operation and then holding down the shift key click on mem_rd_addr.

b. Right click on the selected items and select Set Cycle Constraint

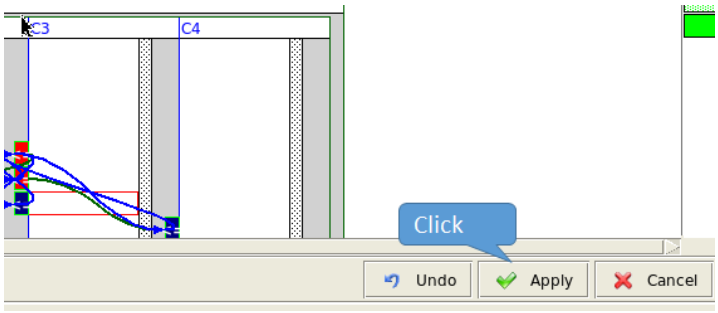


c. Set the number of cycles to 2 and click OK.

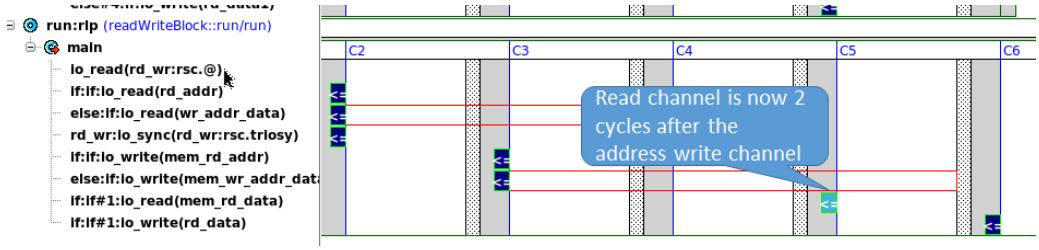
This will force 2 c-steps between the address writes and the data read. (Alternatively you could edit the C++ and place 2 ac_wait() function calls between the write and the read)



d. Click the Apply Button in the Gantt chart.



Look at the new Gantt chart and note that the mem_rd_data channel read is now scheduled 2 cycles (c-steps) after the mem_rd_addr channel write. This will allow two address to be issued in pipelined fashion before the read channel is read.



19. Click on RTL in the Task Bar.

20. Re-run SCVerify verification and verify that the design is now able to run with a continuous throughput on the rd_addr and rd_data channels.

21. Close the RTL simulator

22. Close Catapult

DONE LAB 6

Lab 7: Shared Memory with Independent Read/Write

Designing a block to implement a memory that is shared between multiple processes can be challenging in untimed C++ and HLS. Using blocking read and write coding style and interfaces is relatively easy to do, but there are performance limitations where a stall on one interface will stall the entire design.

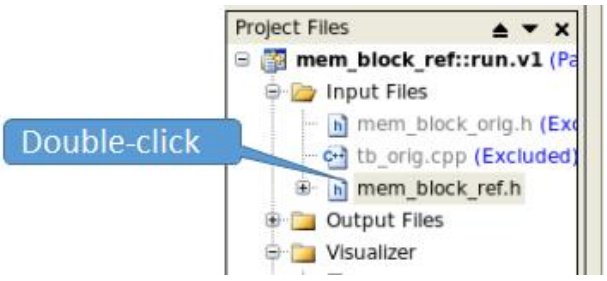
Overview

In this lab you will learn how to:

- Use non-blocking reads and writes to build a shared memory block
- Analyze and debug pipeline feedback failures
- Recode the C++ to fix feedback failures

Steps

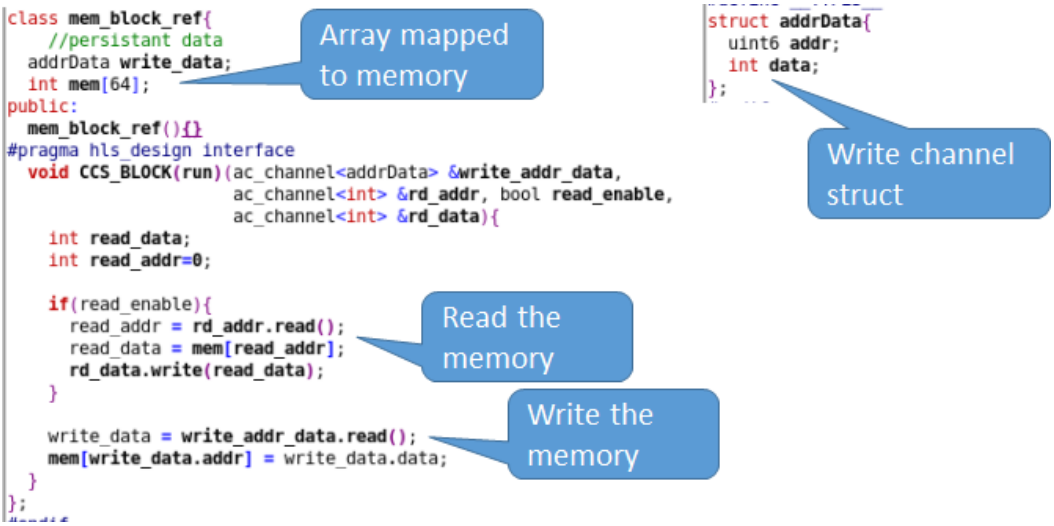
1. CD to the 'lab7' directory
2. Launch Catapult by typing "catapult" at the terminal prompt
3. Go to File > Run Script and select the "run_orig.tcl" file and click Open. This will add inputs files including testbench and synthesize the design.
4. Go to project files open the mem_block_ref.h design file.



5. Look at the C++ and note the following:
   - This design can be used to share a memory between two processes, where one process writes the memory and another process reads

the memory.

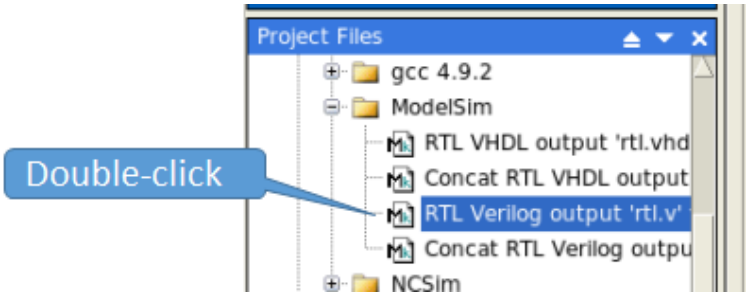
- The design has a write input channel “write_adrr_data” that provides a write address and data in a struct “addrData” that is written into the internal array “mem”
- There is a data output channel “rd_data” that reads data from the internal array “mem” based on the “rd_addr” channel
- All reads/writes are blocking and are mapped to *_wait interfaces
- The memory “mem” can support simultaneous reads and writes



6. Click on Architecture

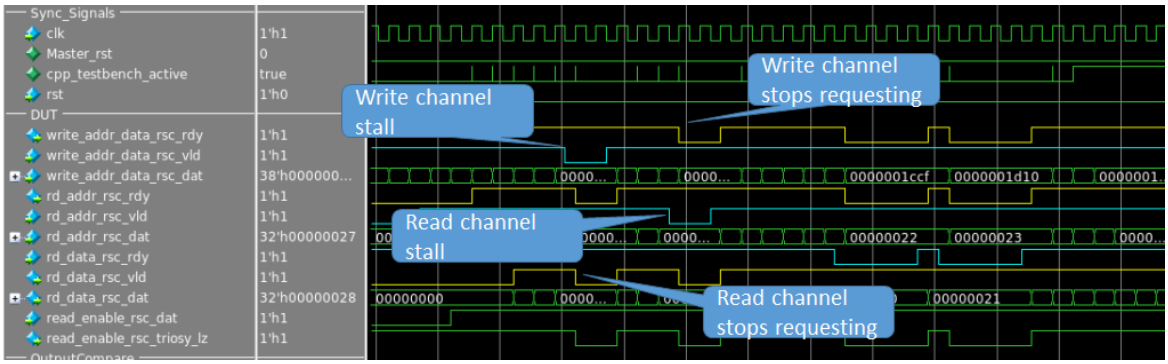
   note that all channel interfaces are mapped to *_wait interfaces, the internal array is mapped to a RAM with 1 read and 1 write port, and the design is pipelined with II=1.

7. Go to Project Files > Verification and launch SCVerify using your RTL simulator.



8. Type “run –all” at the simulator command line.

Look at the waveform and not that when the read channel stalls, the write channel stops requesting data. Likewise when the write channel stalls the read channel stops sending data. So the reads and writes cannot operate independently.



9. Close the RTL Simulator

10. In Catapult go to File > Run Script and select the run_non_blocking.tcl.

    This will make the “mem_block_orig.h” file the top-level design and will use “mem_block_ref.h” as part of the testbench.

11. Double-click on “mem_block_orig.h” in the Project Files folder to open in the editor and note the following:

    The design has been recoded to use non-blocking reads and non-blocking writes to allow the reads and writes to the memory to operate independently.

    The read flag “read_flag0” to read the next address is cleared when the non-blocking write completes.

```
class mem_block{
    //persistant data
  bool write_flag;
  addrData write_data;
  bool read_flag0;
  int mem[64];
  int read_addr=0;
public:
  mem_block():write_flag(0),read_flag0(0){}
#pragma hls_design interface
  void run(ac_channel<addrData> &write_addr_data,
           ac_channel<int> &rd_addr,
           ac_channel<int> &rd_data){
    int read_data;

    if(!read_flag0){
      read_flag0 = rd_addr.nb_read(read_addr);
    }
    if(read_flag0){
      read_data = mem[read_addr];
      read_flag0 = !rd_data.nb_write(read_data);
    }

    write_flag = write_addr_data.nb_read(write_data);

    if(write_flag)
      mem[write_data.addr] = write_data.data;
  }
};
```

Non-blocking read

Read flag for next address cleared when write completes

Non-blocking write

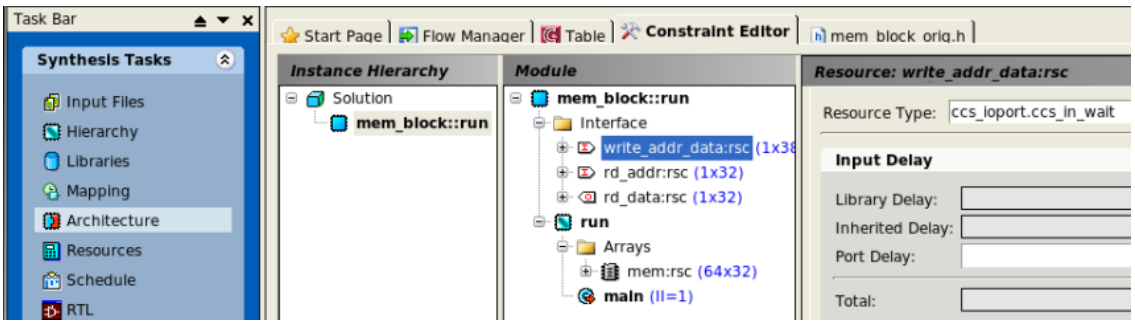
Non-blocking read

12. Go to the Verification Folder > gcc and execute the design and testbench.

    Look at the tb_orig.cpp file and see that it compare the non-blocking and blocking versions of the design to see if they are the same.

13. Go to the Architectural Constraints view

    note that the constraints are the same, all *_wait interfaces and pipelined with II=1.

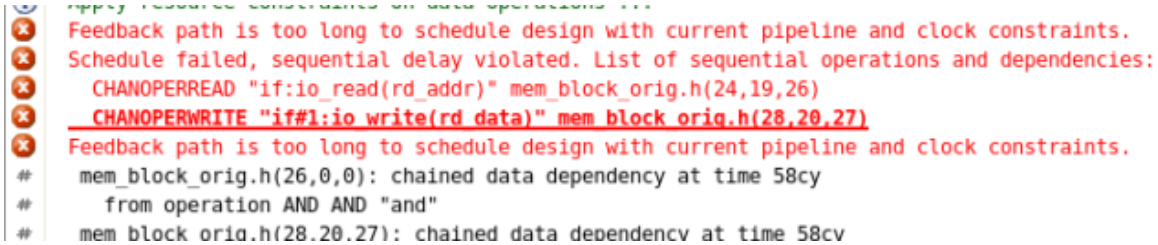


14. Click on Schedule in the Task Bar to schedule the design.

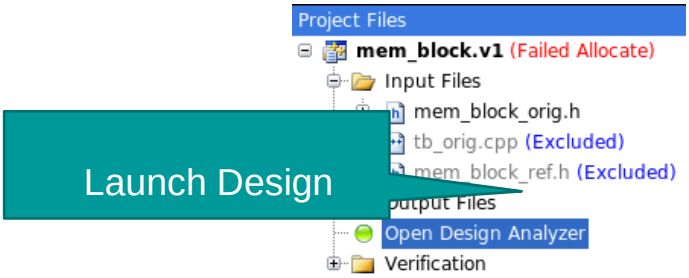
    Look at the Catapult transcript and you will see that an error has been issued.

```
#     from operation ASSIGN ASSIGN "asn"
#   mem_block_orig.h(26,0,0): chained feedback data dependency at time 58cy
#     from operation ASSIGN ASSIGN "asn#4"
#     with output variable "read_flag0.sva"
#   Netlist written to file 'schedule.gnt'
    Completed transformation 'architect' on solution 'mem_block::run.v2': elapsed time 0.74 seconds, memory usage 1258356kB, peak memory
    Design 'mem_block::run' could not schedule partition '/mem_block::run/run' - could not schedule even with unlimited resources
```

Scroll up in the transcript and you will see that the schedule failed due to feedback between a channel write and a channel read operation.

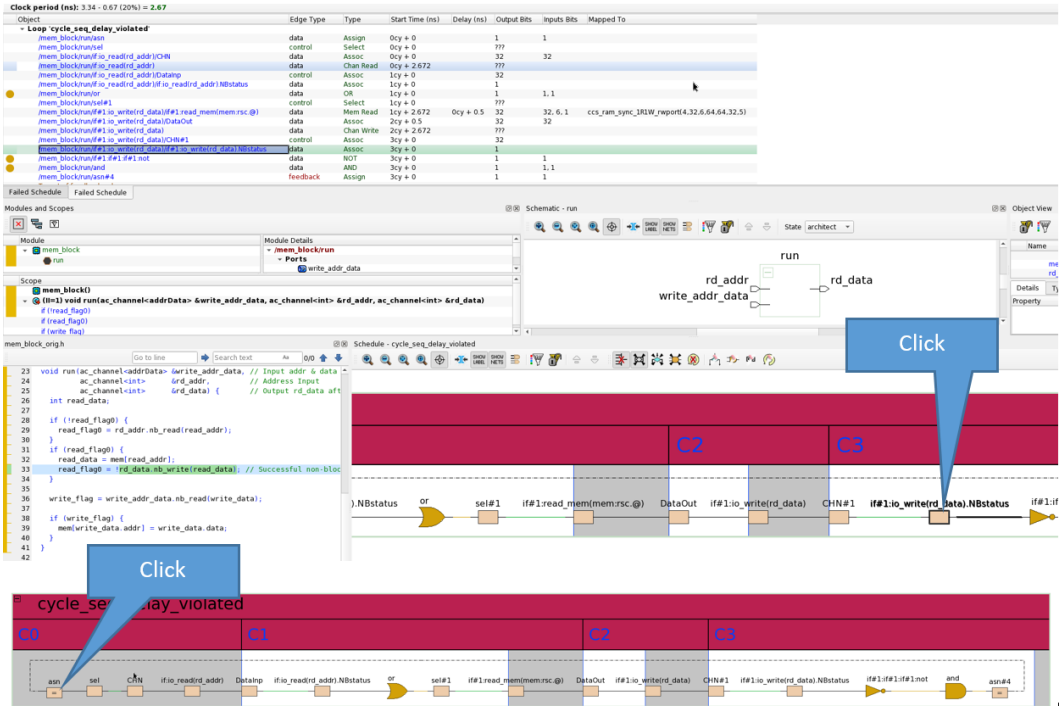


15. Go to the Project Files Folder > Design Analyzer and launch the Catapult Design Analyzer.



16. Click through the operations in the failed schedule view and look at where in the C++ they come from.

The feedback is from the rd_flag0 = rd_data_nb_write(..) back to the !if(rd_flag0). The feedback path is crossing 3 c-steps. Start time column is 0cy+0 to 3c+0



This highlights the “if” condition from the feedback path

The feedback path is crossing three c-step (3 clock cycles) boundaries.

When the nb_write fails there is active data in c1, c2 and c3. Because the nb_write does not stall the pipeline this data would be lost if pipelining with II=1. This is why Catapult cannot schedule the design.

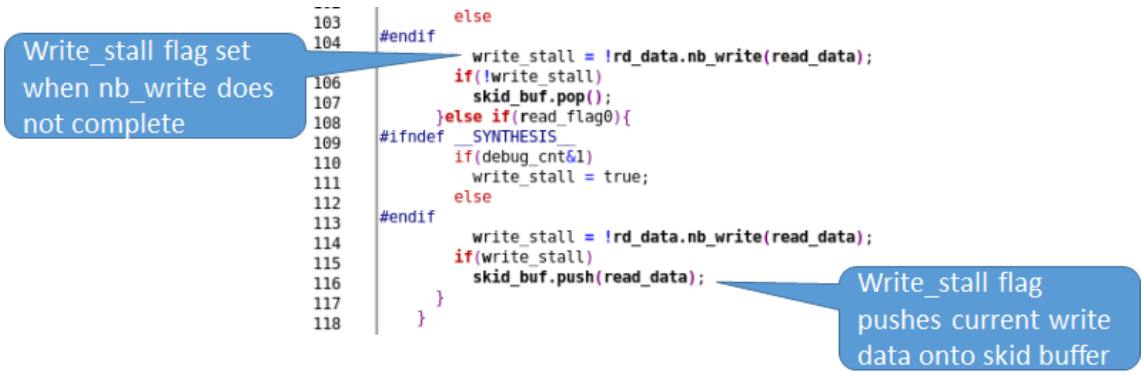
17. Close Design Analyzer

18. In Catapult go to File > Run Script and select “run_nb_skid_buffer.tcl”.

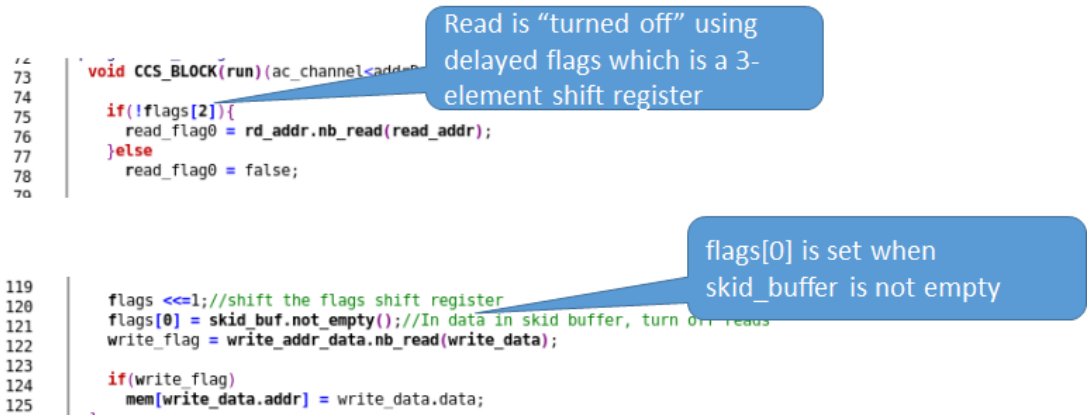
    This will remove the “mem_block_orig.h” design and add in “mem_block_shifted.h”.

19. Open “mem_block_shifted.h” in the Catapult editor and note the following:

    This design has enhanced the previous design to add a “skid buffer” to capture the data “in-flight” in the pipeline when the nb_write does not complete.



The gating off of the nb_read uses a delayed (shift-register) version of the skid-buffer status (not empty).

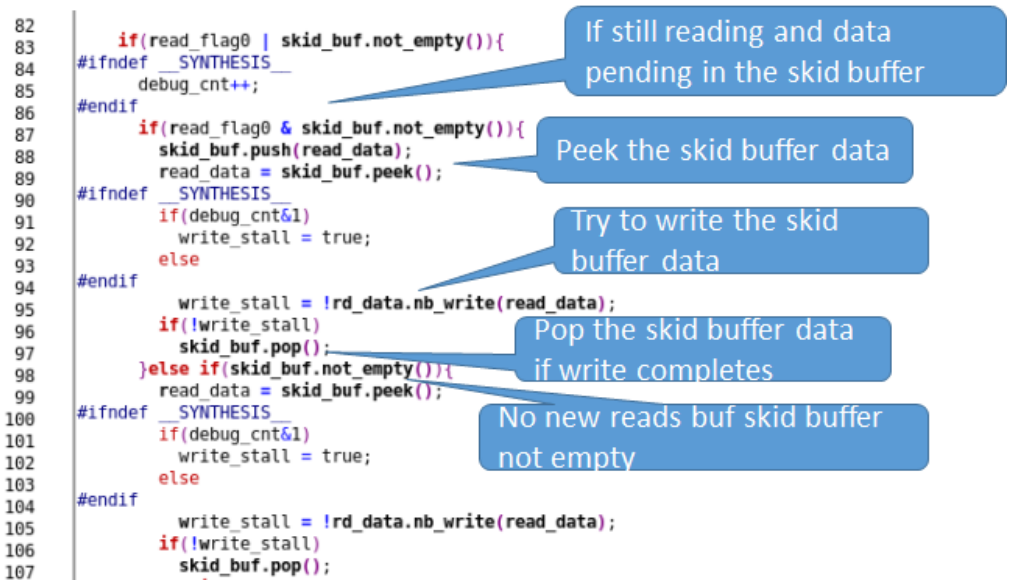



```
void CCS_BLOCK(run)(ac_channel<addr
  if(!flags[2]){
    read_flag0 = rd_addr.nb_read(read_addr);
  }else
    read_flag0 = false;

  flags <<=1;//shift the flags shift register
  flags[0] = skid_buf.not_empty();//In data in skid buffer, turn off reads
  write_flag = write_addr_data.nb_read(write_data);

  if(write_flag)
    mem[write_data.addr] = write_data.data;
```

The skid buffer stores the reads that are in-flight until the reads are gated off. The skid buffer is then emptied. Both the skid buffer and the shift register have to be sized based on the number of cycles in the feedback path. Sizing them too small will result in either a feedback failure (shift register to small) or a simulation failure (skid buffer too small)




```
    if(read_flag0 | skid_buf.not_empty()){
#ifndef __SYNTHESIS__
      debug_cnt++;
#endif
      if(read_flag0 & skid_buf.not_empty()){
        skid_buf.push(read_data);
        read_data = skid_buf.peek();
#ifndef __SYNTHESIS__
        if(debug_cnt&1)
          write_stall = true;
        else
#endif
          write_stall = !rd_data.nb_write(read_data);
        if(!write_stall)
          skid_buf.pop();
      }else if(skid_buf.not_empty()){
        read_data = skid_buf.peek();
#ifndef __SYNTHESIS__
        if(debug_cnt&1)
          write_stall = true;
        else
#endif
          write_stall = !rd_data.nb_write(read_data);
        if(!write_stall)
          skid_buf.pop();
```

Scheduler may throw the following error depending on the memory model ‘RDWRRESOLUTION’

If there are errors in the Catapult transcript as shown below, scroll up in the transcript to the errors. The feedback is from a memory read to a memory write.

```
Apply resource constraints on data operations ...
Feedback path is too long to schedule design with current pipeline and clock constraints.
Schedule failed, sequential delay violated. List of sequential operations and dependencies:
  MEMORYWRITE "if#3:write_mem(mem:rsc.@)" mem_block_shifted.h(124,6,20)
  MEMORYREAD "if#1:read_mem(mem:rsc.@)" mem_block_shifted.h(80,18,14)
Feedback path is too long to schedule design with current pipeline and clock constraints.
 mem_block_shifted.h(74,0,0): chained data dependency at time 61cy
   from operation AND AND "and"
```

20. Double-click on the memory read and memory write errors to cross-probe to the C++.

    These are just the read and write to the “mem” array. This is a false failure. Catapult cannot prove that there is no possibility of address contention and issues this failure. However, as the designer you know that the read address and the write address will never be the same in the same clock cycle. This error can be “waived”

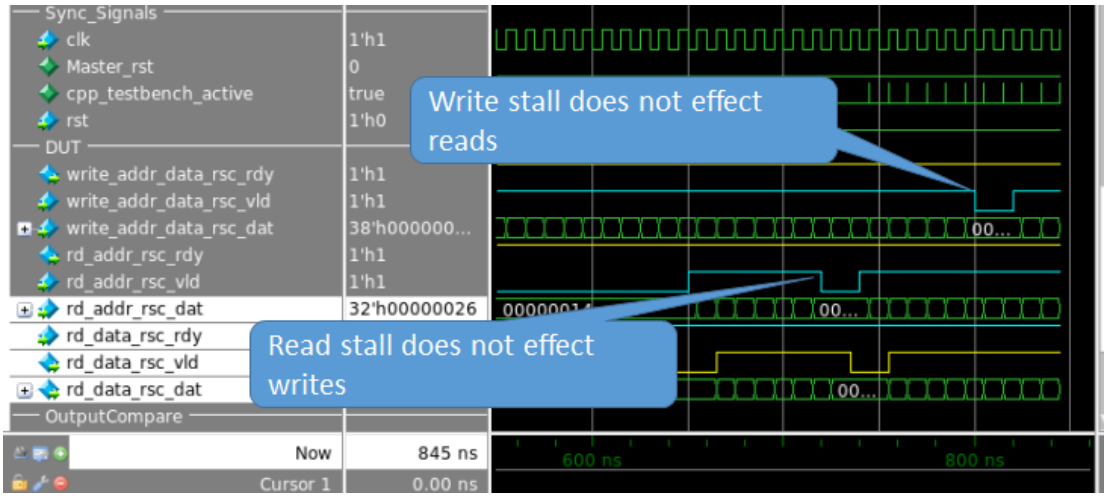
21. Go to File > Run Scrip and select “ignore_mem_err.tcl” this will use the “ignore_memory_precedence” constraint to waive the error.

22. Click on Schedule in the Task Bar. The design should now schedule.

23. Click on RTL in the task bar.

24. Launch SCVerify and run the verification.

    You should be able to see that the memory read and memory writes can now operate independently with stalling the design.



25. Close the RTL simulator

26. Close Catapult

DONE LAB 7