Mentor®
A Siemens Business

Modeling Hierarchy in C++
With Catapult Synthesis

June 2017

© 2015-17 Mentor Graphics Corporation
All rights reserved.

This document contains information that is proprietary to Mentor Graphics Corporation. The original recipient of this document may duplicate this document in whole or in part for internal business purposes only, provided that this entire notice appears in all copies. In duplicating any part of this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use and distribution of the proprietary information.

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

U.S. GOVERNMENT LICENSE RIGHTS: The software and documentation were developed entirely at private expense and are commercial computer software and commercial computer software documentation within the meaning of the applicable acquisition regulations. Accordingly, pursuant to FAR 48 CFR 12.212 and DFARS 48 CFR 227.7202, use, duplication and disclosure by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in the license agreement provided with the software, except for provisions which are contrary to applicable mandatory federal laws.

TRADEMARKS: The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the owner of the Mark, as applicable. The use herein of a third- party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics' trademarks may be viewed at: www.mentor.com/trademarks.

The registered trademark Linux® is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

End-User License Agreement: You can print a copy of the End-User License Agreement from: www.mentor.com/eula.

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

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

Table of Contents

Introduction

The two labs in this lab book are designed to illustrate how design hierarchy can be modeled in C++. The two main concerns when it comes to design hierarchy is the modeling of hierarchical connectivity as well as the representation of concurrency in a purely sequential design language such as C++.

Lab1 – Decimating Average with Bit-accurate Data Types

In this lab example a simple decimating average filter is designed as a C++ function with ac_channel interfaces to add two consecutive values together, producing an output every other time the function is called. This block illustrates the basic concepts of a leaf level block in a hierarchical design and concurrent C++ model execution. In addition, the example illustrates general mechanics of modeling for synthesis and pure C++ simulation (compilation).

Topics covered in this lab

- Including Catapult C++ libraries
- Defining the top-level design
- ac_channel function interfaces
- Bit-accurate data types
- Making code simulateable
- Compiler include paths
- Using ac_channel ".available()"
- Synthesis and SCVerify

Review the C++

1. CD into the *Lab1* directory.
2. Open the *avg2.cpp* file.
3. Look at the top of the file. You will see includes for the *ac_int* and *ac_channel* header files whose data types are used in the design.

```
//Include ac data types and ac_channel libraries
#include <ac_int.h>
#include <ac_channel.h>
```

4. Look at the function definition. You will see that there is a *hls_design* compiler pragma indicating it is the top-level design

   The function formals are defined as *ac_channel<uint4 >*. *uint4* is a *typedef* defined for *ac_int* and is equivalent to *ac_int<4,false>*. These *typedefs* are defined for bit widths up to 64 bits in the *ac_int.h* header file.

```
#pragma hls_design top
void avg2(ac_channel<uint4 > &din, ac_channel<uint4 > &dout){
```

   The algorithm has an ACC loop that reads two values from *din* and sums them together in the variable *sum*. Note that *sum* must be initialized, otherwise the simulation behavior is unpredictable (***U****ninitialized* ***M****emory* ***R****ead* or *UMR*) based on the compiler and platform and the synthesis tool may optimize away any UMRs it detects.

Inside of the *ACC* loop a *#ifndef __SYNTHESIS__* compiler define is used to guard the *printf* statement from synthesis. This statement will print to the output each time the channel is read during C++ simulation.

After the *ACC* loop the value of *sum* is written into the *dout* channel. There is also a *printf* statement, guarded from synthesis, that indicates the output channel is being written.

```
uint5 sum = 0;//Initialize variables to avoid Unitialized Memory Reads (UMR)
ACC:for(int i=0;i<2;i++){
  #ifndef __SYNTHESIS__
  printf("Reading data input channel\n");
  #endif

  sum += din.read();
}

#ifndef __SYNTHESIS__
printf("Writing data output channel\n");
#endif

dout.write(sum);
```

5. Open the test bench file "tb.cpp".
   The testbench defines the *din* and *dout* ac_channels which are used to connect to the *avg2* Device Under Test (DUT).

```
int main(int argv, char *argc){
  ac_channel<uint4 > din;
  ac_channel<uint4 > dout;
```

   Note, ac_channels need to be used in hierarchical designs to implement a point-to-point connection between two instances. They may only be passed by references and local channels must be declared static because they may need to retain state.

   The test bench has a loop that iterates 10 times. Each iteration writes data into the input channel *din* and calls the *avg2* function. This loop also prints to the output that the test bench is writing data into the channel.

```
for(int i=0;i<10;i++){
  uint4 tmp = i +1;
  printf("Testbench writing data%d = %d into channel\n\n",i, tmp);
  din.write(tmp);//Write one value into the channel
  avg2(din,dout);//Call top-level design
}
```

After the test bench loop has completed, all the values that are stored in the *dout* output channel are printed to the output. Remember that *ac_channel* behaves like an infinite FIFO in C++ simulation, so the number of element stored in the channel is not necessarily known at compile time. The *.avaiable()* member function from the *ac_channel* class is used to read data from the channel until it's empty.

```
while(dout.available(1)){//While data is in the channel, print data
  printf("Testbench dout%d = %d\n", cnt, dout.read().to_int());
  cnt++;
}
```

Compile the Design

1. Open the *run* script and edit the *CCS_HOME* variable to point to the *Mgc_home* directory of your Catapult install.
2. Look at the compile line that calls *g++*. Catapult ships with g++ in its *bin* directory. Also note the include line that is required since we are using the *ac_int.h* and *ac_channel.h* header files and the compiler needs to know where to find them, and they are located in *…/Mgc_home/shared/include*.

```
#!/bin/csh
#path to Catapult Mgc_home dir
setenv CCS_HOME /wv/hlsb/CATAPULT/8.0/CURRENT/ixl/Mgc_home

clear
$CCS_HOME/bin/g++ -g -O0 -I$CCS_HOME/shared/include avg2.cpp tb.cpp -o tb.exe

./tb.exe
```

3. Execute the script by typing "*./run*" at the command line. Note, you may have to change the permissions of the command to allow execution of the script.

```
chmod 744 run*
```

You will get a compilation error with the following message:

```
tb.cpp: In function 'int main(int, char*)':
tb.cpp:10: warning: cannot pass objects of non-POD type 'class ac_intN::uint4' through '...'; call
will abort at runtime
Illegal instruction (core_dumped)
```

4. Open the *tb.cpp* file and look at line 10.
   Line 10 is trying to *printf* the *tmp* variable which is of type *uint4*. The *printf* function only works with native C++ data types so we need to convert *tmp* into an integer.

```
 8   for(int i=0;i<10;i++){
 9     uint4 tmp = i +1;
10     printf("Testbench writing data%d = %d into channel\n\n",i, tmp);
11     din.write(tmp);//Write one value into the channel
12     avg2(din,dout);//Call top-level design
```

Fix the "ac_int" Print Error

1. Open the *tb_fix.cpp* file.
2. Look at line 10 and you'll see that it has be changed so that *tmp* is using the *to_uint()* member function to convert it to an unsigned integer.

```
 8   for(int i=0;i<10;i++){
 9     uint4 tmp = i +1;
10     //Added ac_int member function to convert to unsigned for printing
11     printf("Testbench writing data%d = %d into channel\n\n",i, tmp.to_uint());
12     din.write(tmp);//Write one value into the channel
13     avg2(din,dout);//Call top-level design
```

3. Execute the *run_fix1* compile script by typing "./run_fix1" at the command line. This script uses the *tb_fix.cpp* file as the test bench.

   You will see the test bench executing and almost immediately abort. The test bench output indicates that one data value is written into the channel followed by the DUT indicating that it is reading twice, after which you get a message "Read from empty channel" followed by program termination. Normally for more complex designs and testbenches a C++ debugger or IDE should be used to debug these types of error. However in this simple example it is obvious what is happening based on the *printf* statements in the test bench and DUT. The test bench has performed one channel write into *din* and the DUT has tried to read twice from *din*. The second read is attempting to read an empty channel. We need to add *.available()* to the DUT to prevent reading an empty channel.

```
Testbench writing data0 = 1 into channel

Reading data input channel
Reading data input channel
Read from empty channel
terminate called after throwing an instance of 'char*'
Abort (core dumped)
```

Using ".available()"

1. Open the *avg2.cpp* file.
   You can see that the ACC loop always runs and reads two values from *din* regardless if data is available.

```
ACC:for(int i=0;i<2;i++){
  #ifndef __SYNTHESIS__
  printf("Reading data input channel\n");
  #endif

  sum += din.read();
}
```

2. Open the *avg2_fix.cpp* file.
   The *.available(2)* member function for *din* has been added to prevent the function body from executing unless there are two elements in the channel.

```
if(din.available(2)){
  ACC:for(int i=0;i<2;i++){
    #ifndef __SYNTHESIS__
    printf("Reading data input channel\n");
    #endif

    sum += din.read();
  }

  #ifndef __SYNTHESIS__
  printf("Writing data output channel\n");
  #endif

  dout.write(sum);
}
```

   The *.available* member function allows to model concurrency and mult-rate execution in a very simplistic manner. This modeling style doesn't require thread switching or any other simulation overhead which enables fast execution.
3. Execute the *run_fix2* script by typing "./run_fix2" at the command line. This script uses the *avg2_fix.cpp* file.
4. Look at the output and you'll see the test bench writing twice before the DUT reads twice and writes once.
5. Look at the output and note the averaged values printed out by the test bench.

```
Testbench writing data8 = 9 into channel

Testbench writing data9 = 10 into channel

Reading data input channel
Reading data input channel
Writing data output channel

Testbench dout0 = 3
Testbench dout1 = 7
Testbench dout2 = 11
Testbench dout3 = 15
Testbench dout4 = 3
```

The last value of *dout* equals 3. This mistake is commonly encountered in designs using bit-accurate datatypes. The output corresponds to adding the input values 9 and 10.

```
9+10 = 19 = 0xb10011
```

But *dout* is a *uint4* type so it only uses the 4 LSBs:

```
(uint4) dout = 0xb10011 = 0xb0011 = 3
```

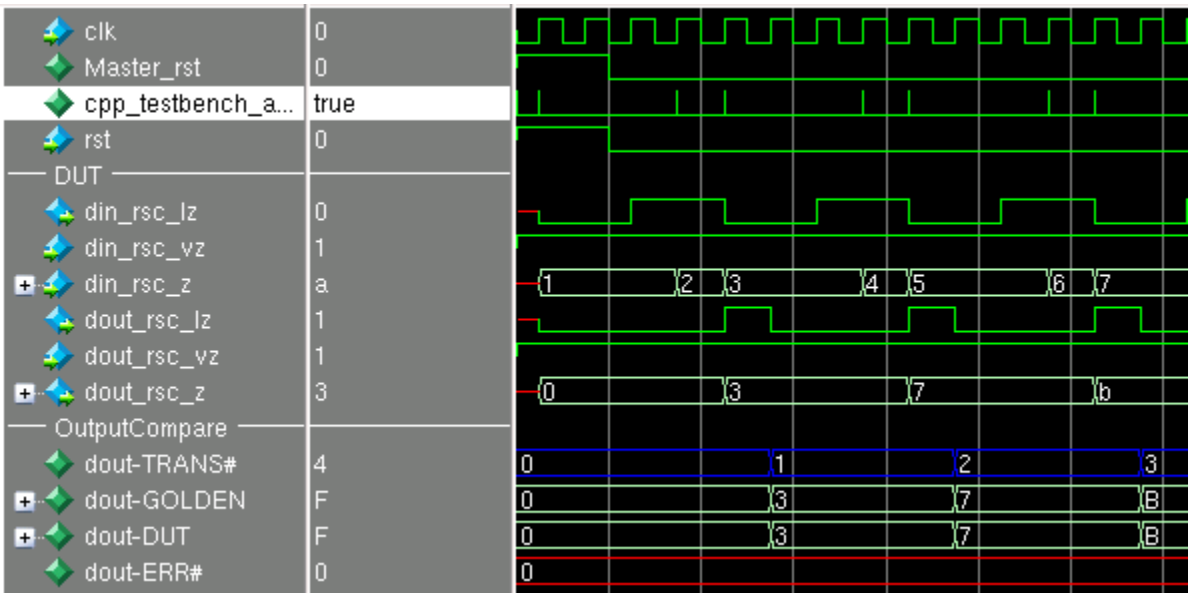
The *ac_int* data types behave just like bit-vectors and will roll-over when assigned values outside of their dynamic range.

Synthesis

A closer look at the testbench (*tb_fix.cpp*) shows that the example is setup to run with SCVerify (CCS_MAIN, CCS_DESIGN, CCS_RETURN)

You can start up Catapult and type "source directives.tcl" which will synthesize the design and create the necessary Makefiles to run SCVerify which it will try to start automatically for the g++ simulation and Modelsim.

You will see the same output generated as through the straight C++ simulation in the Transcript window of Catapult and Modelsim. However, the waveform will in addition show the 4 cycle throughput.



DONE Lab1

Lab2 – Matrix Transposition and Moving Average Filter

This lab covers basic coding style for hierarchical designs and memory architecture.

Topics covered in this lab

- "ac_channel" streaming interfaces
- Multi-block hierarchical design style
- Arrays pass through "ac_channel"
- Passing configuration data between blocks
- "ac_channel" shared memory coding style
- "windowing" memory architecture"

Design Architecture

The design architecture for this lab consists of a configurable two-block design that transposes a 2-D matrix of data and then performs a simple moving average filter on the rows of data of the transposed matrix. The data is streamed into and out of the design using "ac_channel" interfaces.

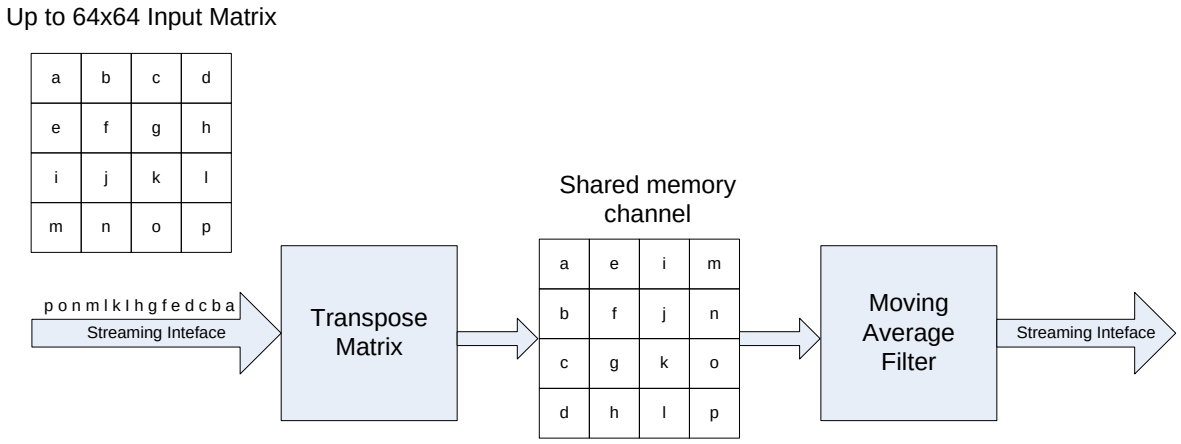


Figure 1 Multi-block Architecture

Review the design C++

1. CD in to the *Lab2* directory.
2. Open the *transpose_filter.cpp* file.
3. Scroll to the bottom of the file and look at the *transpose_and_filter* top-level function definition.

   The *#pragma hls_design top* line indicates this is the top-level design.

   The function formals are two *ac_channel* of type *uint8*, which are 8-bits unsigned.

The top-level function body uses the required coding style for hierarchical designs where the top-level design consists only of static interconnect channels and function instances. This coding style is also required for any intermediate block which is intended to represent hierarchy.

The interconnect channels are used to connect the transpose function to the filter function.

```
#pragma hls_design top
void transpose_and_filter(ac_channel<uint8 > &din, ac_channel<uint8 > &dout){
  //Static channels at top-level used to interconnect hierarchical blocks
  static ac_channel<memStruct > mem;//Shared memory object
  static ac_channel<uint7 > height;//Control channels
  static ac_channel<uint7 > width;

  transpose(din,mem,height,width);
  filter(mem,dout,height,width);
}
```

Please note, the ac_channels are declared static to ensure that the channels are able to keep state between calls to the transpose_and_filter function.

There is a shared memory interconnect channel "mem" with the data type "memStruct".

4. Open the *transposed_filter.h* file.
   The *memStruct* struct has a 2-d array as a data member. Hierarchy coding style requires that arrays mapped to memories, or any array for that matter, must be packed in a *struct* and written through an *ac_channel*.

```
struct memStruct{//Struct used to pack array data for ac_channel reads/writes
  uint8 data[64][64];
};
```

5. Scroll to the top of the "transpose_filter.cpp" file and look at the "transpose" function definition.
   - The "#pragma hls_design" indicates that this is a design block.
   - The input "din" is a streaming channel that gets the input matrix data one element at a time.
   - The output channel "dout" is a shared memory channel of type "memStruct"
   - "height_out" and "width_out" are control channels that pass the matrix height and width to the "filter" block.
   - An instant "mem" of type "memStruct" is defined internally to be used as temporary storage for the shared memory channel, which is the required coding style. Note the temporary storage should

always be read at the top of the function if the memory is read and written at the bottom of the function if the memory is written.

```
#pragma hls_design //Design hierarchy pragma
void transpose(ac_channel<uint8 > &din, ac_channel<memStruct > &dout,
               ac_channel<uint7 > &height_out, ac_channel<uint7 > &width_out){
  ac_int<7,false> height,width;
  memStruct mem;//temporary storage for shared memory
```

6. Look at the "transpose" function body.
   - The first thing the function does is two read two consecutive values from the "din" input channel. These are the matrix width and height and can be anywhere up to 64. Reading the same channel twice like this will prevent pipelining with II=1 at the top level.
   - After reading the configuration data it is written out into the "width_out" and "height_out" configuration channels which has the width and height to the "filter" block.
   - The ROW and COL loops read the matrix data from "din" and write it into the temporary storage "mem" in transposed order.
   - The ROW and COL loops are bounded with an upper bound of 64, but will conditionally exit based on "width" and "height", which is the recommended coding style.
   - After the loops exit, the temporary struct "mem" is written into the shared memory channel. The temporary memory shouldn't be accessed after this write to ensure good quality of results for synthesis.

```
//Sequential channel reads from "din" will limit pipelining at the top-level
//First two values read from "din" are configuration data for height and width
width = din.read();//Make an internal copy of configuration data to limit channel reads
height = din.read();
width_out.write(width);//Copy configuration data to next block
height_out.write(height);

ROW:for(int r=0; r<64; r++){//ROW can be pipelined with II=1
  COL:for(int c=0; c<64; c++){
    mem.data[c][r] = din.read();//Single channel read
    if(c == width -1)
      break;
  }
  if(r == height -1)
    break;
}
dout.write(mem);
```

7. Look at the function definition for the "filter" function in "transpose_filter.cpp" and note the following:
   - "#pragma hls_design" to indicate this is a design block.
   - A memory input channel "din" of type "memStruct".
   - A streaming output channel "dout".

- A variable "mem" of type "memStruct" to act as temporary storage for the shared memory.
- A variable "shift_reg" of type "shift_class<uint8,3>" which acts as an 8-bit wide, 3-tap shift register. The shift register class is defined in "shift.h" and is templatized to allow specification of the data type and the number of taps. It implements the shift operator "<<" and the array operator "[ ]" for writing and reading data. You can review this file if you wish.

```
#pragma hls_design
void filter(ac_channel<memStruct > &din, ac_channel<uint8 > &dout,
            ac_channel<uint7 > &height_in, ac_channel<uint7 > &width_in){
  uint7 height,width;
  memStruct mem;//Temporary storage for shared memory

  shift_class<uint8, 3> shift_reg;
  uint8 window[3];
  uint8 mac;
  uint8 din_tmp;
```

8. Look at the "filter" function body and note the following:
   - Configuration input channels for the height and width of the matrix are read into temporary variables so that the channels are only read once.
   - Shared memory interface channel "din" is read and stored in the temporary struct "mem".
   - ROW and COL loops are bounded and run to 64(ROW) and 65 (COL) iterations. They will conditionally break based on "width" and "height".
   - COL loop runs for 1 extra iteration to allow for the startup time of the shift register (sliding window). NOTE: There will be a "dead" cycle between each row processed from the memory since no output is written for the first iteration of COL. This may be acceptable. If the "dead" cycle is not acceptable the code will need to be re-written so that the writing of the output is continuous.
   - The reading of "mem[r][c] is guarded to prevent an out of bounds access. This is due to the extra iteration in the COL loop.
   - Each value read from "mem" is shifted into the shift register using the "<<" operator.
   - The shift register output taps are processed in "clip_window" to handle the boundary conditions.

```
height = height_in.read();//Make local copy of control signals sc
width = width_in.read();

mem = din.read();//Read memory data into temporary struct

ROW:for(int r=0; r<64; r++){
  COL:for(int c=0; c<64 + 1; c++){
    if(c<width)//prevent over-read of din
      din_tmp = mem.data[r][c];
    shift_reg << din_tmp;//Sliding window storage of din
    clip_window(shift_reg,c,width,window);//Boundary processing
    mac = window[0]/4 + window[1]/2 + window[2]/4;
    if(c>=1)//startup
      dout.write(mac);
    if(c == width)
      break;
  }
  if(r == height)
    break;
}
```

9. Look at the "clip_window" function definition in "transposed_filter.cpp" and note the following:
    - The "#pragma map_to_operator [CCORE] tells the synthesis tool that this function should be synthesized as a standalone object to preserve the desired MUXing structure.

```
#pragma map_to_operator [CCORE]//Force muxing to be a single object
void clip_window( shift_class<uint8, 3> shift_reg,
                  uint6 i, uint7 width, uint8 window[3]){

  window[0] = (i==1) ? shift_reg[1]:shift_reg[2];
  window[1] = shift_reg[1];
  window[2] = (i==width) ? shift_reg[1]:shift_reg[0];
}
```

Compile the design

1. Edit the "run" file and set "CCS_HOME" to point to the "Mgc_home" directory of you Catapult install.
2. Compile the design by typing "./run" at the command line. Note, you may have to change the permissions of the command to allow execution of the script.

```
chmod 744 run*
```

Synthesis and SCVerify

You can start up Catapult and type "source directives.tcl" which will synthesize the design and create the necessary Makefiles to run SCVerify which it will try to start automatically for the C++ simulation and Modelsim.

You will see the same output generated as through the straight C++ simulation in the Transcript window of Catapult and Modelsim. But Modelsim will allow to actually investigate the cycle behavior.

DONE Lab2