**Documentation for Vivado HLS optimizations**

July 20 2015:

**Primitive Converter**

The reset is set as a 'control asynchronous' signal using “config\_rtl” directive in “solution settings”

The primitive converter module was initially translated using C-based approach. The C-design without any optimizations, had a latency of 144 clock cycles. However, the latency constraint was 1 clock cycle with the throughput being 1 output per clock cycle(from the Verilog implementation).

**Optimizations on params and th\_mem:**

The params and th\_mem are LUT's which are accessed in the primitive converter.

As the loop is unrolled multiple accesses are made to these LUT's

By default these LUTs are inferred as a dual port RAM which has greater latency and does not satisfy the needs as at least 4 simultaneous multiple accesses take place in order to maintain minimum latency.

Hence the array was completely partitioned into individual registers so that access can be multiple and simultaneous.

However this presented another problem. Now the data was not latching on the registers, rather it was changing on every clock cycle.

Multiple code structure combinations were tried and after much experimentation, a **for loop** was used to write into the LUT's. This worked but there was another problem. 'Params ' array which is of length '6' was working correctly. Only one thing to be kept in mind is that input should change exactly after the **interval specified** for the designed.

Another problem is that after 64 elements in the th\_mem array, the ports are inferred as inputs for unknown reason.

One more important yet unexplained requirement is that the input “**r\_in**” should be exactly of the same size as the word length of the array.

1. Dataflow directive
2. register port for 'r\_in' and 'addr'
3. array\_partition

July 24,2015:

The LUT's i.e. params and th\_mem were separately tested first to solve the problem of latching the data. When tested separately as a module, the LUT module performed exactly as required.

**if**(sel==0 && we==1)

params[addr]=r\_in;

**if**(sel==1 && we==1)

th\_mem1[addr]=r\_in;

**if**(sel==2 && we==1)

th\_mem2[addr]=r\_in;

The main directive which helped this to happen was the dataflow directive (#pragma HLS DATAFLOW). This essentially told the compiler that whatever data is being written in the LUTs will be immediately read and used elsewhere in the code. This forced the compiler to back each data up (the array was completely partitioned) into a register. These registers was later again assigned to the outputs in the RTL code.

The next step was to integrate the LUT module with the 'primitive converter' module.

1. I first started out with retaining the HLS DATAFLOW directive inside the mem.cpp file (LUTs)
2. The primitive converter module was pipelined with registered inputs and outputs.
3. A top module called “top\_prim\_conv.cpp” was created to integrate both modules.
4. The params and th\_mem arrays were initialized inside the 'top\_prim\_conv' function as local arrays.
5. The two function were initalized, and synthesized into RTL. The resulting RTL was surprisingly again not latching the data.
6. The solution was found after further investigation. The solution was to remove the HLS DATAFLOW directive from “mem.cpp” and use the INLINE directive to prevent inline of “mem.cpp”. Also the local arrays, params and th\_mem were promoted as “global arrays”.
7. This resulted in the RTL code meeting requirements.
8. But as mentioned previously, th\_mem was just limited to 64 elements due to a bug in the tool. Hence th\_mem was implemented as two arrays of 64 elements each. Corresponding changes were made to the “primitive converter” module.
9. This was also tested to function properly.
10. Now the design was working as per requirements. Now the latency and throughput had shot up to 4 clock cycles. The primitive converter module's inputs and outputs were registered and pipelined. This apparently was not pipelined.
11. To overcome this, the register interface from the inputs and outputs of the primitive converter module were removed and the top\_prim\_conv module's inputs and outputs were registered.

**if**(we==1)

{

mem( r\_in, we, addr, sel, params,th\_mem1,th\_mem2);

}

**else**

{

prim\_conv1(

quality,

wiregroup,

hstrip,

clctpat,

ph,

th,

vl,

phzvl,

me11a,

clctpat\_r,

ph\_hit,

th\_hit,

sel,

addr,

params,

th\_mem1,

th\_mem2,

r\_out

);

1. This resulted in latency of 2 and throughput (initiation interval) of 1 clock cycle. The reason for this was the functions mem and prim\_conv1 were scheduled on every initiation of the design resulting in 2 clock cycles.
2. To solve this, the code was modified as shown. “we” signal was used to as shown. This saved one extra clock cycle.
3. Finally after much investigation as some (not so obvious) work-arounds, the design is now fully pipelined with a pipeline depth of 2 and interval of 1 clock cycle.
4. Below is all the directives applied in the directives.tcl script

set\_directive\_array\_partition -type complete -dim 1 "top\_prim\_conv" params

set\_directive\_array\_partition -type complete -dim 1 "top\_prim\_conv" th\_mem1

set\_directive\_unroll "prim\_conv1/prim\_conv1\_label0"

set\_directive\_inline -off "mem"

set\_directive\_pipeline -II 1 "prim\_conv1"

set\_directive\_pipeline -II 1 "top\_prim\_conv"

set\_directive\_interface -register "top\_prim\_conv" quality

set\_directive\_interface -register "top\_prim\_conv" wiregroup

set\_directive\_interface -register "top\_prim\_conv" hstrip

set\_directive\_interface -register "top\_prim\_conv" clctpat

set\_directive\_interface -register "top\_prim\_conv" r\_in

set\_directive\_interface -register "top\_prim\_conv" we

set\_directive\_interface -register "top\_prim\_conv" ph

set\_directive\_interface -register "top\_prim\_conv" th

set\_directive\_interface -register "top\_prim\_conv" vl

set\_directive\_interface -register "top\_prim\_conv" phzvl

set\_directive\_interface -register "top\_prim\_conv" me11a

set\_directive\_interface -register "top\_prim\_conv" clctpat\_r

set\_directive\_interface -register "top\_prim\_conv" ph\_hit

set\_directive\_interface -register "top\_prim\_conv" th\_hit

set\_directive\_interface -register "top\_prim\_conv" sel

set\_directive\_interface -register "top\_prim\_conv" addr

set\_directive\_array\_partition -type complete -dim 1 "top\_prim\_conv" th\_mem2

September 12,2015:

**Integration of primitive converter module and replication of the the modules:**

The 'prim\_conv\_sector' is a module which implements 'primitive converter' module for the whole sector. This makes use of the multiple instances of the two submodules previously converted. These multiple instances have to be separate RTL implementations. For this purpose, I implemented a test design which has a function instantiated inside a nested for loop as shown below:

**#include** <ap\_int.h>

**void** **inst\_test**(

ap\_uint<8> a, //this function is a essentailly an adder which

ap\_uint<8> b, // implements c=a+b;

ap\_uint<8> \*c

);

**void** **test\_top**(

ap\_uint<8> a[4][4],

ap\_uint<8> b[4][4],

ap\_uint<8> c[4][4]

)

{

**int** i=0,j=0;

test\_top\_label0:**for**(i=0;i<4;i++){

test\_top\_label2:**for**(j=0;j<4;j++){

inst\_test(

a[i][j],

b[i][j],

&c[i][j]

);

}

}

}

The directive used are as follows:

**set\_directive\_inline -off "inst\_test"**

**set\_directive\_interface -mode ap\_none "test\_top" a**

**set\_directive\_interface -mode ap\_none "test\_top" b**

**set\_directive\_interface -mode ap\_none "test\_top" c**

**set\_directive\_unroll "test\_top/test\_top\_label0"**

**set\_directive\_unroll "test\_top/test\_top\_label2"**

From the above example it follows that the function that has to instantiated is made to be not “inlined” through the first directive. The for loops are completely unrolled and this will make “inst\_test” as separate blocks in the RTL implemetation.

However, this presents another problem. The a[][],b[][] and c[][] interfaces will be treated as a 'ap\_memory' interface. This makes these to be implemented as a memory. Implementing this as a memory will also add to the latency.

A work-around for this is to completely partition these arrays in all dimensions to make them behave as a separate ports using the below directives:

**set\_directive\_array\_partition -type complete -dim 0 "test\_top" a**

**set\_directive\_array\_partition -type complete -dim 0 "test\_top" b**

**set\_directive\_array\_partition -type complete -dim 0 "test\_top" c**

Note that the '-dim 0' will completely parttion the arrays in all dimenions.

Now we have separate modules for each instantiation.

**ZONE IMAGE FORMATION MODULE:**

The zone image formation module is a purely combinational logic. It OR's the cross-section images from all the chambers and forms the sector image. Ph\_zone is the array of flip-flops which is denotes the sector image.

The system verilog code implements indexed part select for easier manipulation of arrays. The same is done in the HLS code using range selection.

Example usage:

In system verilog :  **ph\_zone[1][3][ 2+: ph\_hit\_w10]**

In HLS : **ph\_zone[0][1] ( 2, 2+ph\_hit\_w10-1)**

This enabled us to use indexed part-select in HLS as implemented in System Verilog.

This was the only issue in terms of conversion from System verilog to HLS.

The next **problem** was getting the required performance. As this is a purely combinational design (set wires cross-connecting), the ideal latency of the module should have been ‘0’.

But the latency was not one and behaved unpredictably. It was observed that the HLS was extracting state machine consisting of three states from the HLS code. This was not expected to happen.

After careful investigation, the reason behind this was determined; the same is illustrated below.

The following is a snippet of HLS code:

**ph\_zone[0][2] = 0;**

if (phzvl[2][0] & 0x1) **ph\_zone[0][2]**(1, 1+ph\_hit\_w20-1) = ph\_zone[0][2](1, 1+ph\_hit\_w20-1) | ph\_hit[2][0];

if(phzvl[2][1]&0x1)**ph\_zone[0][2]**(39,39+ph\_hit\_w20-1) =ph\_zone[0][2](39, 39+ph\_hit\_w20-1) | ph\_hit[2][1];

if(phzvl[2][2]&0x1)**ph\_zone[0][2]**(76,76+ph\_hit\_w20-1) =ph\_zone[0][2](76, 76+ph\_hit\_w20-1) | ph\_hit[2][2];

If we observe the above code, we can see that the array element **‘ph\_zone[0][2]’** is being manipulated for 3 times in 3 consecutive statements. The HLS infers it as separate operation every single time and hence used to infer a state machine which had 3 states. The number of states would change if there were additional manipulations on the array element. Such was the behavior of the HLS code.

**Solution:**

The problem described above existed because HLS treated each array element separately. To overcome this, the **array\_reshape directive** was used. This directive combines all the array elements (smaller arrays) into one larger array (big array). Thus now the HLS treats it as a single operation and we obtain the required performance i.e. latency=1 clock cycle.

**ZONE HIT EXTENDER MODULE:**

Please refer Polar CO-ordinate delay module. Same optimizations employed.

**Phi Pattern Detectors Module:**

The task of phi pattern detectors is to detect the track patterns in phi cross-section of the sector, and determine their quality. The quality is higher for the straighter tracks, and for tracks with hits in more layers.

The phi pattern detectors code in system verilog is converted in to HLS code.

The conversion is straight-forward as the code involves for loops and case statements. However, the first challenge was ‘reduction’ operators. In the system verilog code, ‘OR reduction’ is implemented as shown.

lyhits[2] = |st1[7:0];

lyhits[1] = st2;

lyhits[0] = (|st3[14:7]) | (|st4[14:7]);

This is implemented in the HLS code using the or\_reduce() function. An excerpt of the code is shown below:

a\_st1=st1(7,0);

a\_st3=st3(14,7);

a\_st4=st4(14,7);

lyhits[2] = a\_st1.or\_reduce();

lyhits[1] = st2;

lyhits[0] = ((a\_st3).or\_reduce()) | (a\_st4.or\_reduce());

Note here that the a\_st1, a\_st3 and a\_st4 have to be initialized first and then a\_st1, a\_st3 and a\_st4 have to used for ‘OR reduction’. If the st1(7,0) is directly used for the ‘OR reduction’ then it results in an error. Ex: st1(7,0).or\_reduce(); will result in a error.

Next challenge was concatenation. As seen in previous modules, concatenation is done as follows:

qcode\_p[mi]= (straightness[2], lyhits[2], straightness[1], lyhits[1], straightness[0], lyhits[0]);

The next challenge was the array bx[][]. Particularly the following piece of code

bx[mi][foldn] = (**int**(lyhits) == 0x0) ? 0x0 :\

(**int**(bx[ mi][foldn]) == 0x7) ? 0x7 :\

**int**(bx[mi][foldn]) + 0x1; // bx starts counting at any hit in the pattern, even single

‘bx’ is a two dimensional array. As always it was found that array is a major challenge and performance bottleneck. To optimize this the ARRAY\_PARTITION directive was used. The array was completely partitioned into its constituent elements in both 1-D and 2-D. But this presented in a major problem.

If you observe the above, bx[mi[foldn] is the array element to be updated. ‘mi’ is the iteration variable used in the for loop. ‘foldn’ is the input variable for the HLS function. This particular piece of code is part of a single always@ block. Hence on every clock edge this code should be executed taking into account the updated values of ‘foldn’ or any other variables. This always@ was mimicked using a ‘while(en==1)’ loop, which had previously worked for zone hit extender module. However the HLS converted RTL was behaving differently When the HLS code was converted into RTL, the RTL code has a state machine which read the value of ‘foldn’ in the first state of the state and then entered the while loop. The state machine did not visit the state where the ‘foldn’ input was read. Each iteration of the while loop used the same value of ‘foldn’ which it read in the first state. This is because, whenever you instantiate a function and pass augments using ‘pass by value’, then the function uses that same value till it returns, in this case till it exists the while loop and returns. Hence this option did not work out.

Next I tried calling the same function in the while loop in a top-level function. This also did not help as I was not explicitly changing the value of the argument for each invocation. Also ‘bx’ array should have held the values for each invocation. This was not the case the scope of the variable is only inside that invocation of the function. On the next invocation, the function uses new versions of the variables.

The solution for this was making ‘bx’ as a global array in the top-level function and passing this array every time the function for ph\_pattern is invoked. This will result in manipulating the same array every time. This is achieved now the next challenge was read of ‘foldn’. This was overcome by changing the code structure. The code was written as follows:

**if**(en==1){

ph\_pattern(st1, st2, st3, st4, drifttime, foldn, qcode,bx);

}

**else**

{

**for**(**int** j=0;j<9;j++){ **for**(**int** k=0;k<4;k++){ bx[j][k]=0;

}

}

}

This resulted in creation of separate process for ph\_pattern. The initialization of the ‘bx’ array to 0 is also to be noted as important because if this is not done then the bx elements are uninitialized and may result in corrupted results.

The next problem was now the latency was now ‘2’ clock cycles after unrolling all the iterations of the for loop. This was because of the Block-level I/O protocol which is inserted into the synthesized design by default. This protocol adds signals like ap\_start, ap\_idle and ap\_done to the RTL automatically. This meant that the initiation interval is now 2 clock cycles because in one clock cycle the computation is done and in the next clock cycle, the ap\_done signal is asserted. Because of these extra signals especially ap\_done, there were two states in the state machine, one computed the result and the other just asserted t h ap\_done signal and did not accept any input.

To overcome this the Block-level I/O protocol had to be removed. This was done by using the ‘ap\_ctrl\_none’ interface directive which did not insert any protocol signals to the synthesized design.

set\_directive\_interface -mode ap\_ctrl\_none "ph\_pattern"

set\_directive\_interface -mode ap\_ctrl\_none "ph\_pattern\_top"

The design now consisted of no extra signals to be asserted hence HLS compiler did not infer any state machine which resulted in initiation interval of 1 clock cycle and latency of 1 clock cycle. This was the target and it was achieved. The list of directives are as follows:

set\_directive\_array\_partition -type complete -dim 1 "ph\_pattern" qcode\_p

set\_directive\_unroll "ph\_pattern/ph\_for"

set\_directive\_array\_partition -type complete -dim 0 "ph\_pattern\_top" bx

set\_directive\_interface -mode ap\_ctrl\_none "ph\_pattern"

set\_directive\_interface -mode ap\_ctrl\_none "ph\_pattern\_top"

set\_directive\_pipeline -II 1 "ph\_pattern\_top"

NOTE: **HLS PIPELINE** directive is **needed** for ap\_ctrl\_none interface

AS OF JAN 11 2016:

So here I am again.

Ph\_pattern detectors have been the most eye opening and frustrating module. I think I have found out the solution. Fingers crossed.

Problem: The "bx" array in the ph\_pattern is used to keep tarck of the number of hits.

Making this as a global array did not work because the array had to retain its value fro all invocations.

I tried making this as a global array and also a static array. The problem was the HLS compiler inferred it as a single array which all the instances of ph\_pattern uses. Hence it was scheduling each invocation of the ph\_pattern function one after the other. Hence when I wanted to synthesize 4\*122 instances of ph\_pattern in the ph\_pattern\_sector function, it was taking almost an hour or so.

To overcome this problem, after many frustrating tries, I came up with a solution which uses C++ concepts of objects.

I created a class with a "bx" as variable inside the class. Now each object (hence each instance of the function) has its own copy of “bx”. Now I created a say 5 objects of the class I defined. Then I instantiated the ph\_pattern function using these objects (see sample code below) using a for loop. Then I have 5 instances of the function (module) ph\_pattern running in parallel.

Sample test case:

#include <ap\_int.h>

class test{

public:

ap\_uint<3> bx[7];//bx array.

public:

void test\_func(ap\_uint<3> a,ap\_uint<3> b,ap\_uint<3> index,ap\_uint<3> \*c);

test();

};

test::test(void){

//initialize bx array

#pragma HLS ARRAY\_PARTITION variable=test::bx complete dim=1

label1:for(int i=0;i<7;i++){

bx[i]=0;

}

std::cout<<"inside object creation"<<std::endl;

}

void test::test\_func(ap\_uint<3> a,ap\_uint<3> b,ap\_uint<3> index,ap\_uint<3> \*c)

{

#pragma HLS INLINE off

test::bx[index]=test::bx[index]+1;

\*c=a+b+test::bx[index];

std::cout<<"c is "<<\*c<<std::endl;

}

void multiple\_inst(ap\_uint<3> a[5],ap\_uint<3> b[5],ap\_uint<3> index[5],ap\_uint<3> c[5],ap\_uint<1> en)

{

#pragma HLS INTERFACE ap\_ctrl\_none port=return

#pragma HLS PIPELINE II=1

#pragma HLS ARRAY\_PARTITION variable=a complete dim=1

#pragma HLS ARRAY\_PARTITION variable=b complete dim=1

#pragma HLS ARRAY\_PARTITION variable=c complete dim=1

#pragma HLS ARRAY\_PARTITION variable=index complete dim=1

static test inst[5];

#pragma HLS ARRAY\_PARTITION variable=inst complete dim=1

//create 5 objects

// unroll loop to have 5 instances running parallely

multiple\_inst\_label1:for(int i=0;i<5;i++){

#pragma HLS UNROLL

if(en==1)

inst[i].test\_func(a[i],b[i],index[i],&c[i]);

}

}

There are a few important things to do if the above code has to schedule 5 instances the “test\_func” parallely.

1. It has to partition the array of objects (test\_inst in this case) completely.
2. The directive to partition bx array has to be placed in the constructor so that each object has a copy of “partitoned” bx array.
3. UNROLL the loop which calls test\_func.
4. **MOST IMPORTANT guideline**: When you want the array (or any other variable) to be persistent over several calls to the function; in other words, if you want a variable to retain its value over several function calls then we have to define the objects created to be a static array of objects; **static test\_inst[5];**

Defining the array or any other variable as static will not work because this creates a dependency which prevents parallel execution. A static array is an array which is used for all invocations of that function. Hence making the objects as static tells the HLS compiler to make all member variables to be persistent while not creating any dependencies.

Using these techniques and some other explained in the previous modules, Ph\_pattern detector was successfully implemented and verifed.

**NOTE**: I suspect that using the concept of C++ objects makes the code more understandable to the HLS compiler and hence we get a surprising yet welcome result. The resource usage of the HLS generated RTL code is 13% of LUTs whereas the existing verilog module has a resource usage of 16% LUTs. This is a very good breakthrough for us.

**SORTER MODULE:**

We now move on to the sorter module. The essence of the sorter module is to pick out the best 3 quality codes from an array of quality codes. Its basically picking out largest 3 numbers from an array.

The existing verilog code takes 3 clock cycles to select the best 3 out of the array (one in each clock cycle). My aim was to minimize the latency of the sorter module to at least 2 clock cycle if not 1clcok cycle.

Even the best sorting algorithm take O(nlogn) time which is not feasible for us. The best selection algorithm takes O(n) time so O(3n) for 3 such selections. Again not feasible. Bear in mind that the complexity does not present any information about the latency. Also Vivado HLS does not support recursion (variable latency). So the sorting appraoch was abandoned.

My first step was to start and create a comparison tree with less levels so that I have more number of comparisons. So instead of comparing two numbers at a time, I started comparing 4 elements at once. Now the number of levels decreased from log2 N to log4 N. My objective was to fit in at least two iterations of this comparison tree into one clock cycle.

When I applied the directives and generated the RTL, the pipeline had 5 stages, hence 5 clock cycle latency.

Solution:

1. Inline the function sort function into zone\_best3 function. This gets rid of 1 pipeline stage each hence we now get a latency of 1 clock cycle.
2. But the problem is that HLS compiler tries to fit in everything into one clock cycle which results in timing violation. The critical path delay was reported to be 45.15 ns whereas the clock period is 25ns. Clearly no acceptable.
3. The solution to this problem is explicitly tell the compiler to create a pipeline. There is no directive to tell the compiler that but I changed the code structure to something like this:

sort( a,winner0,&winid[0],ret\_a);

sort( ret\_a,winner1,&winid[1],ret\_a1);

sort( ret\_a1,winner2,&winid[2],ret\_a2);

“ret\_a” is same as the array “a” but excluding the First maximum element. This is fed into the next sorter module. “ret\_a1” is same as the array “a” but excluding the First maximum and the second maximum element. This is fed into the third sorter module. This forces the HLS compiler to generate a pipeline.

1. Now when I generated the RTL code, the latency was 2 clock cycle. A significant result. However, for some reason, the HLS compiler was including an extra register to the winner0 output. Hence winner0 output was getting delayed whereas the winner1 and winner2 were not.

After inspection, I realized this was due to the fact that the compiler scheduled the winner0 module in the first clock cycle. So after the first clock cycle, winner0 output was obtained. But it has to wait for winner1 and winner2 which is to be calculated in the second clock cycle. At this juncture, HLS should insert a register in order to delay the winer0 output by one clock cycle. But for some reason, it inserted an extra register.

1. I suspected that the extra register problem was because I was inlining the sorter module into the best3 function. This meant that HLS can optimize the sorter module as it likes. I have no control over it now.
2. The solution was to create a new function called “sort\_1”. This is exactly same as the “sort” function but has a different name and it is inlined.
3. Now I called the sorter module in the following way:

sort( a,winner0,&winid[0],ret\_a);

sort\_1( ret\_a,winner1,&winid[1],ret\_a1);

sort\_1( ret\_a1,winner2,&winid[2],ret\_a2);

Now, I used a directive to force the “sort” function not be inlined (make it as a separate function in the hierarchy).

1. Because my problem was with winner0, I used “sort” function which is not inlined to calculate winner0. And I used the “sort\_1” function which is inlined to calculate the winner1 and winner2.
2. The HLS compiler proceeds in the same way as it did for winner1 and winner2, but now it does not try to optimize the “sort” function because it is a separate function in the hierarchy.
3. This way the extra register is removed and only one register is included in the path of winner0.
4. All the outputs are available after exactly 2 clock cycles. A massive improvement!

**CO-ORDINATE DELAY module;**

This module basically is a delay module which delays the outputs of the primitive converter for future use. In this module, the inputs are delayed by 5 clock cycles (this can be changed to get required delay).

The sample test case below demonstrates this.

**void** **test**(ap\_uint<4> a, ap\_uint<4> c[3],ap\_uint<4> d[3])

{

**volatile** ap\_uint<4> temp[5];

**int** i,j,k;

i=0;j=0;k=0;

temp[4]=a;

test\_label8:**for**(i=4;i>0;i--){

**#pragma** HLS unroll

temp[i-1]=temp[i];

}

c[2]=temp[0];

c[1]=temp[1];

c[0]=temp[2];

}

In the sample test case, I have created a shift register (the for loop) by explicitly telling the HLS compiler what I want to achieve. But the HLS compiler does not acknowledge this and optimizes the Shift register away. As observed, the output is an array. The 3 elements of the output array is assigned delayed inputs from 3 consecutive clock cycles.

This was the requirement but the HLS optimized this and did not assign the values to the output array on the required clock cycles. In other words, the outputs were all delayed by 5 clock cycles.

SOLUTION: The problem basically boils down to “CONTROL”. The HLS has control and optimizes the code as it pleases. The whole point of creating a shift register explicitly is to have control. The solution hence lies in seizing control from HLS.

CONTROL in hardware means FSM, I had to explicitly tell the HLS to generate a FSM.

Notice the “volatile” keyword before the declaration of the “temp” array.

This tells the compiler to avoid any optimizations whenever temp is involved in an operation. By doing this the FSM is generated for every operation. The operations are not delayed or done beforehand. This enables us to assign the outputs at an exact clock cycle.

**MATCH\_PH\_SEGMENT MODULE:**

This module was straightforward to convert considering the experience and optimization techniques gained during the conversion process of the previous modules.

The main optimization technique used here was the C++ approach. Each and every function call in the HLS code was made using objects. The object approach helps us to optimize resources and operations.

Earlier in the some of the previous modules, the “en” signal was used to help generate a FSM with more than one state. This was done because many processes inside the generated RTL code had CURRENT\_STATE on its sensitivity list. If there was only one state,, then CURRENT\_STATE would never change and hence these processes will not be executed.

To overcome this, make the top function (of the sector) as a member function in the defined class and write a wrapper function to instantiate this top function. There is now no need of the “en” signal.

**NOTE TO SELF**:This technique has to applied to all the previous modules