Designing a JPEG DCT in C/C++
using Catapult Synthesis

June 2017

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/

Improving Performance and Area via Design Partitioning

The previous Toolkits optimized the DCT for both performance and area using a combination of HLS constraints as well as architectural coding changes. This allowed us to achieve a throughput of 128 cycles and a large improvement in area via quantization using bit-accurate data types and architectural code changes (Folded filter). The use of a singleport RAM as well as the coding of sequential loops limits the performance on the order of 128 cycles. We saw from simulation that the ROW0 set of loops is idle when the COL1 set of loops is running and vice versa. The design loops must be partitioned into separate processes in order to achieve the final performance goal of 64 cycles. Catapult supports two type of design partitioning:

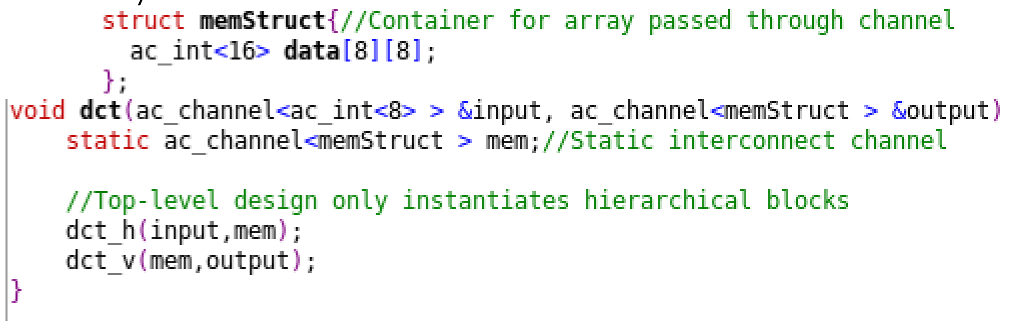
- Loosely coupled
    - Catapult "hierarchy" allows C++ functions to be synthesized as separate concurrently running processes with built-in synchronization.
- Tightly coupled
    - Catapult "CCORE" allows C++ functions to be synthesized as operators which can then be scheduled, shared, etc.

Compile the Design

- CD into the DCT_Hierarchy directory and launch Catapult by typing "catapult" at the command prompt (This step can be skipped if running the toolkit interactively).
- Go to File > Run Script and run the directives1.tcl file. If running interactively select the "Compile the design" script and click on "Launch Project" in the toolkit window.

Analyze the Code Changes for Catapult Hierarchy

- Open the "dct.cpp" file.
- Scroll through the design and look at how the code has been structured for Catapult "Hierarchy".
    - ac_channel on interface declared as a reference "&".
    - Arrays must be passed through channels via structs.
    - The top-level design only instantiates functions mapped to hierarchy ans static interconnect channels



```
    struct memStruct{//Container for array passed through channel
      ac_int<16> data[8][8];
    };
void dct(ac_channel<ac_int<8> > &input, ac_channel<memStruct > &output)
    static ac_channel<memStruct > mem;//Static interconnect channel

    //Top-level design only instantiates hierarchical blocks
    dct_h(input,mem);
    dct_v(mem,output);
}
```

- The ROW0 and COL1 set of loops have been moved into functions "dct_h" and "dct_v"
- The "dct_h" and "dct_v" functions have the hierarchy pragma "#pragma design" indicating that this function is mapped to hierarchy (This can also be done as a constraint).
- "dct_h" and "dct_v" both use the required coding style for memories, where the array mapped to memory must be in a struct (memStruct).
- "dct_h" and "dct_v" both define local structs of type "memStruct" to perform the memory reads/writes.

```
#pragma design
void dct_h(ac_channel<ac_int<8> > &input, ac_channel<memStruct > &mem)
    memStruct local_mem;//Local struct for temporary array operations
    ac_int<8> buf0[8];//Local storage for one row of input
#pragma design
void dct_v(ac_channel<memStruct > &mem, ac_channel<memStruct > &output)
    memStruct local_mem;//Local struct for temporary array operations
    memStruct local_output;
```

- "dct_h" and "dct_v" both use the recommended coding style for arrays mapped to memories in hierarchy. Using the recommended style guarantees that the local structs will be optimized away leaving only memory interfaces.

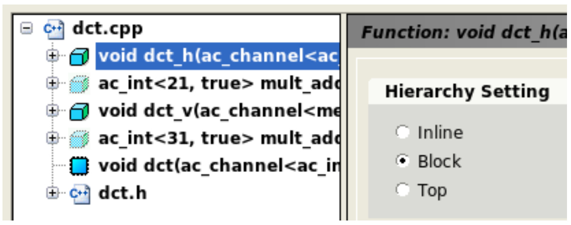
```
void dct_v(ac_channel<memStruct > &mem, ac_channel<memStruct > &output)
    memStruct local_mem;//Local struct for temporary array operations
    memStruct local_output;


    ac_int<16> buf1[8];//Local storage for one row of input
    ac_int<31> acc1;

    local_mem = mem.read();
    COL1:for (int j=0; j < 8; ++j ) {
        COPY_ROW:for(int p=0; p<8; p++)//Copy one row of mem into local
            buf1[p] = local_mem.data[j][p];
        ROW1:for (int i=0 ; i < 8; ++i ){
            acc1 = mult_add<ac_int<16>,ac_int<17>,ac_int<31> >(buf1,i);
            local_output.data[i][j] = acc1 >> 15 ;
        }
    }
    output.write(local_output);
```

Constrain and Synthesize the Design

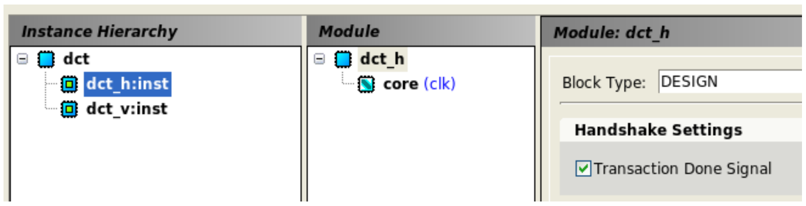
- Click on "Hierarchy" in the Task Bar.
- Note that the "dct_h" and "dct_v" Hierarchy Setting is set to "Block"


```
⊟ dct.cpp
  ⊞ void dct_h(ac_channel<ac
  ⊞ ac_int<21, true> mult_add
  ⊞ void dct_v(ac_channel<me
  ⊞ ac_int<31, true> mult_add
    void dct(ac_channel<ac_in
  ⊞ dct.h
```
Function: void dct_h(a

Hierarchy Setting

○ Inline
● Block
○ Top

- Click on Mapping
- Note that "dct_h" and "dct_v" are shown as Instance Hierarchy. Their Design Type is set to "DESIGN" which indicates that they will be synthesized as separate concurrent processes.

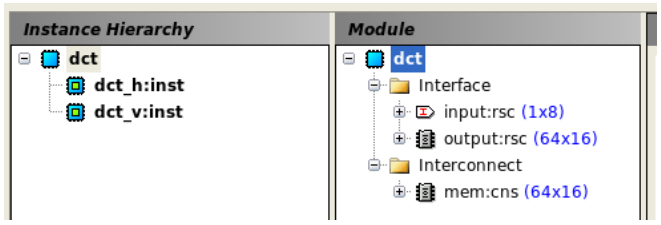

Instance Hierarchy
```
⊟ dct
    dct_h:inst
    dct_v:inst
```
Module
```
⊟ dct_h
    core (clk)
```
Module: dct_h

Block Type: DESIGN

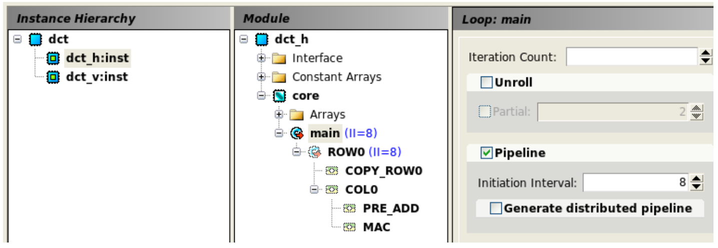
Handshake Settings

☑ Transaction Done Signal

- Go to Architectural Constraints and expand the Interface and Interconnect folders. You can see that the "output" channel and "mem" interconnect channel have been mapped to singleport RAM.


Instance Hierarchy
```
⊟ dct
    dct_h:inst
    dct_v:inst
```
Module
```
⊟ dct
  ⊟ Interface
    ⊞ input:rsc (1x8)
    ⊞ output:rsc (64x16)
  ⊟ Interconnect
    ⊞ mem:cns (64x16)
```

- Click on "dct_h" and unroll all the loops under the "ROW0" loop. Then pipeline the "main" loops with II=8.
- Repeat previous step for "dct_v".


Instance Hierarchy
```
⊟ dct
    dct_h:inst
    dct_v:inst
```
Module
```
⊟ dct_h
  ⊞ Interface
  ⊞ Constant Arrays
  ⊟ core
    ⊞ Arrays
    ⊟ main (II=8)
      ⊟ ROW0 (II=8)
          COPY_ROW0
        ⊟ COL0
            PRE_ADD
            MAC
```
Loop: main

Iteration Count:

☐ Unroll

☐ Partial: 2

☑ Pipeline

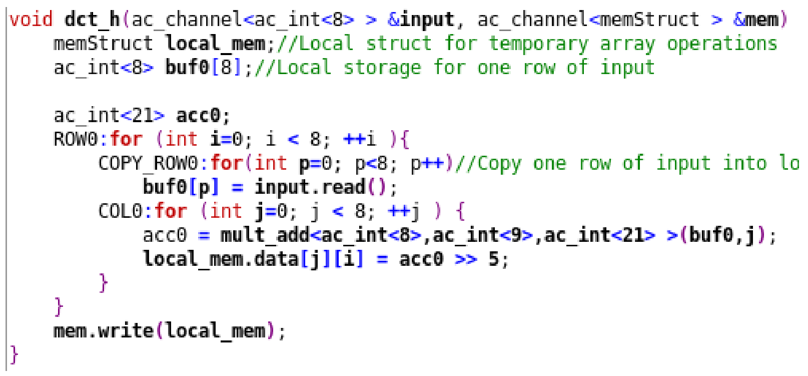
Initiation Interval: 8

☐ Generate distributed pipeline

- Generate RTL
- Look at the Table View, the throughput is reported as 64 cycles.
- Run SCVerify and verify the throughput equals 64 cycles. Observe that "input" is read continuously and "output" is written continuously.

Optimize Area and Runtime using CCOREs

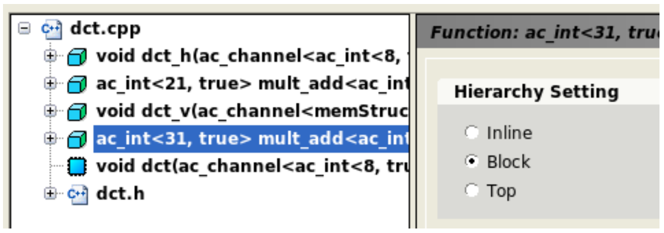
You may have noticed that the Catapult runtime slowed down as we added more parallelism into the single block design. Catapult's runtime is proportional to the number of operations in the design. Fully unrolling the inner loops and pipelining with II=8 creates lots of parallel operations, some of which Catapult can re-share because the II=8. However, doing this creates an "irregular" design, and Catapult will not be efficient when sharing resources (Note: you can usually see this irregularity in the Gantt chart). Catapult shares resources based on area cost. The bigger the resource the more likely it will get shared because Catapult has to weigh the area cost of adding input multiplexers to share a resource against the resource area itself. Unrolling, in this example, creates lots of smaller resources and Catapult is less efficient in sharing. Analysis of the C++ and constraints reveals that the fundamental computational unit for both "dct_h" and "dct_v" is the "mult_add" function. Although COL0 is unrolled 8 times, creating 8 copies of "mult_add", "main" is pipelined with II=8, which should allow "mult_add" to be re-shared down to a single copy. In order for this to happen we need a way to tell Catapult to preserve "mult_add" as a single object so the entire object is easy to share. This can be accomplished using the Catapult CCORE flow.



```
void dct_h(ac_channel<ac_int<8> > &input, ac_channel<memStruct > &mem)
    memStruct local_mem;//Local struct for temporary array operations
    ac_int<8> buf0[8];//Local storage for one row of input

    ac_int<21> acc0;
    ROW0:for (int i=0; i < 8; ++i ){
        COPY_ROW0:for(int p=0; p<8; p++)//Copy one row of input into lo
            buf0[p] = input.read();
        COL0:for (int j=0; j < 8; ++j ) {
            acc0 = mult_add<ac_int<8>,ac_int<9>,ac_int<21> >(buf0,j);
            local_mem.data[j][i] = acc0 >> 5;
        }
    }
    mem.write(local_mem);
}
```

- Click on "Hierarchy" in the Task Bar.
- Set the Hierarchy Setting to "Block" for each instance of the "mult_add" function.



```
dct.cpp
    void dct_h(ac_channel<ac_int<8,
    ac_int<21, true> mult_add<ac_int
    void dct_v(ac_channel<memStruc
    ac_int<31, true> mult_add<ac_int
    void dct(ac_channel<ac_int<8, tru
    dct.h
```

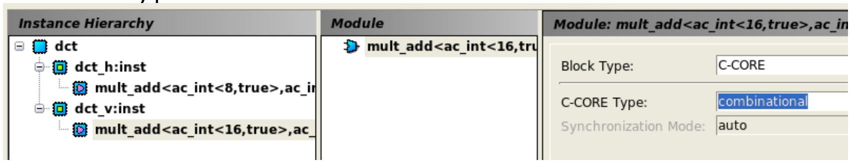
Function: ac_int<31, tru

Hierarchy Setting

Inline [✓ Block] Top

- Click on "Mapping" in the Task Bar.

- Select each instance of "mult_add" and set the "Block Type" to CCORE and the "CCORE Type" to "Combinational".



```
Instance Hierarchy
⊟ dct
  ⊟ dct_h:inst
      mult_add<ac_int<8,true>,ac_i
  ⊟ dct_v:inst
      mult_add<ac_int<16,true>,ac_
```

Module: mult_add<ac_int<16,tru

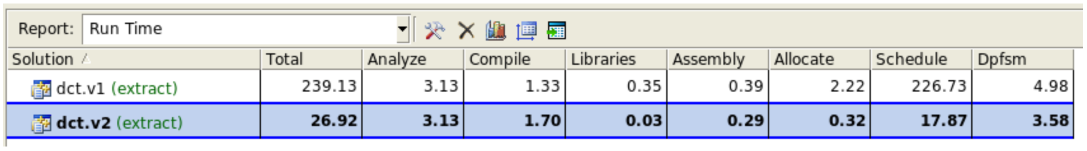
Module: mult_add<ac_int<16,true>,ac_in

Block Type: C-CORE

C-CORE Type: combinational

Synchronization Mode: auto

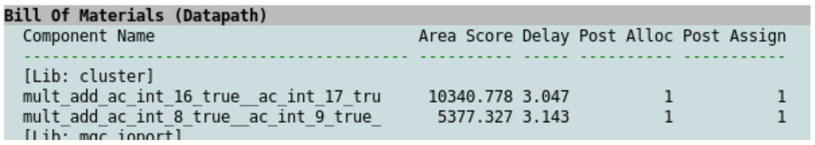
- Schedule the design.
- Open the Gant chart and view it by component utilization.
- The schedule consists almost entirely of CCORE, memory, and IO operations. Note how regular the design "looks".
- Generate RTL.
- Go to the Table View and look at how much area has been reduced.
- Set the Table View Report type to "Run Time". Look at how much faster the design synthesized when using CCOREs. This is because most of the optimizations only have to deal with a single CCORE object as opposed to hundreds of individual operations.



Report: Run Time

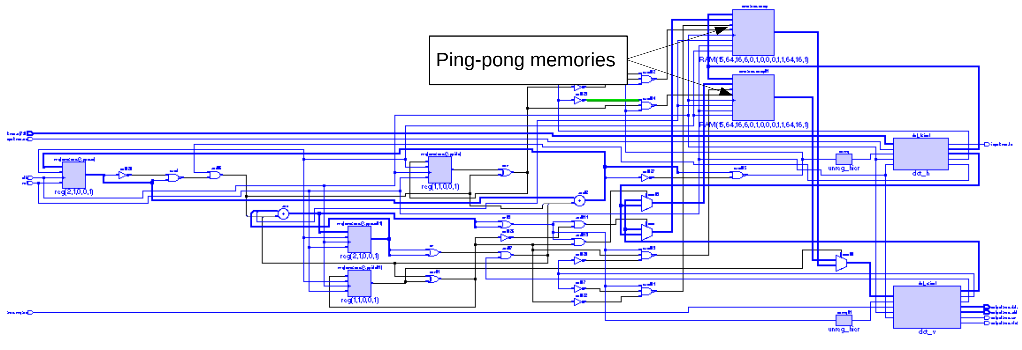
Solution	Total	Analyze	Compile	Libraries	Assembly	Allocate	Schedule	Dpfsm
dct.v1 (extract)	239.13	3.13	1.33	0.35	0.39	2.22	226.73	4.98
dct.v2 (extract)	**26.92**	**3.13**	**1.70**	**0.03**	**0.29**	**0.32**	**17.87**	**3.58**

- Open the RTL report and look at the BOM. The entire design requires only 2 "mult_add" CCOREs, one for each "dct_h" and "dct_v".



```
Bill Of Materials (Datapath)
  Component Name                            Area Score Delay Post Alloc Post Assign
  ----------------------------------------- ---------- ----- ---------- -----------
  [Lib: cluster]
  mult_add_ac_int_16_true__ac_int_17_tru     10340.778 3.047          1           1
  mult_add_ac_int_8_true__ac_int_9_true_      5377.327 3.143          1           1
  [Lib: mgc_ioport]
```

- Open the RTL schematic.
- Catapult has automatically created a ping-pong memory for the shared interconnect memory "mem". This allows both blocks, dct_h and dct_v, to run simultaneously.

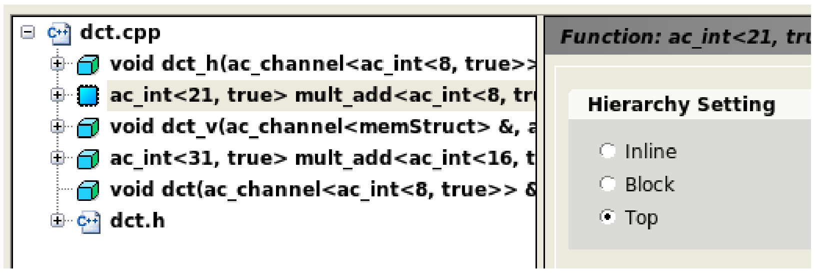


Bottom-up Design Flow

Using Catapult Hierarchy and CCOREs we were able to greatly improve design performance, area, and runtime. However, the design was being synthesized in a top-down fashion, which means that any C++ code change would force a re-synthesis of the entire design. This is bad for both runtime as well as impacting the back-end flows. Because of this Catapult allows designs to be synthesized in a bottom-up fashion at the DESIGN or CCORE level.

Synthesize the CCOREs

- Click on "Hierarchy in the Task Bar.
- Set the "ac_int<21,true> mult_add" function to be the top-level.

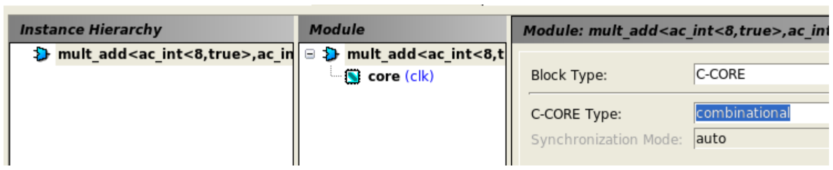


dct.cpp
void dct_h(ac_channel<ac_int<8, true>>
ac_int<21, true> mult_add<ac_int<8, tr
void dct_v(ac_channel<memStruct> &, a
ac_int<31, true> mult_add<ac_int<16, t
void dct(ac_channel<ac_int<8, true>> &
dct.h

Function: ac_int<21, tr

Hierarchy Setting
Inline
Block
[✓ Top]

- GO to "Mapping" and set the Design Type to CCORE and the CCORE Type to Combinational.



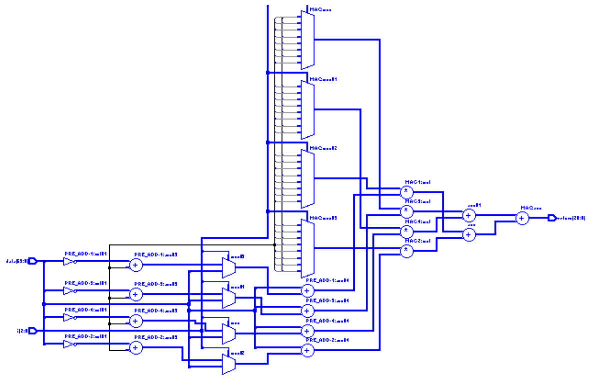
Instance Hierarchy
mult_add<ac_int<8,true>,ac_in

Module
mult_add<ac_int<8,t
core (clk)

Module: mult_add<ac_int<8,true>,ac_int

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

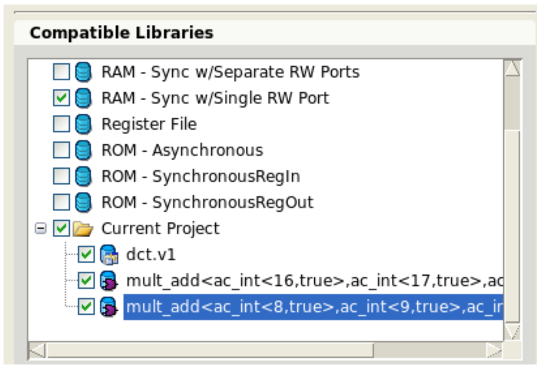
- Generate RTL.
- Open the RTL Schematic. Note the regularity of the mult_add function hardware.



- GO to mapping and repeat the previous steps on the "ac_int<31,true>mult_add" function.

Synthesize the Design Blocks

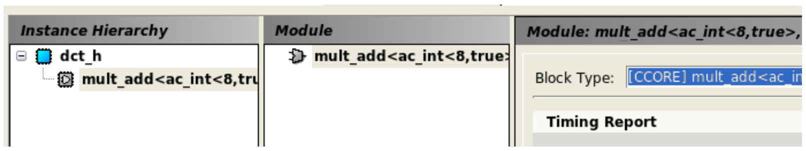
- Go to "Hierarchy" and set the "dct_h" function as the top-level design.
- Go to "Libraries" and select the two "mult_add" library components.



Compatible Libraries

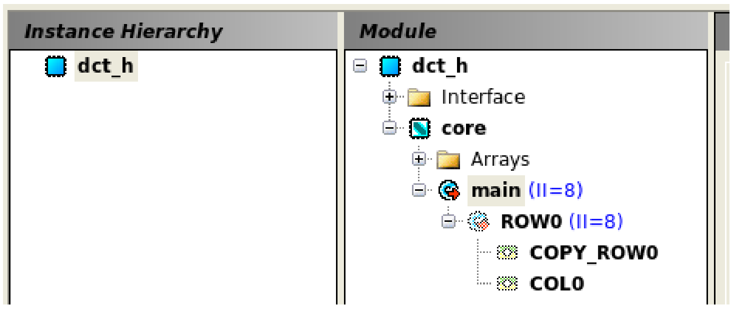
- [ ] RAM - Sync w/Separate RW Ports
- [✓] RAM - Sync w/Single RW Port
- [ ] Register File
- [ ] ROM - Asynchronous
- [ ] ROM - SynchronousRegIn
- [ ] ROM - SynchronousRegOut
- [✓] Current Project
  - [✓] dct.v1
  - [✓] mult_add<ac_int<16,true>,ac_int<17,true>,ac
  - [✓] mult_add<ac_int<8,true>,ac_int<9,true>,ac_ir

- Go to "Mapping", click on the "mult_add" instance, and set to Block Type to the pre-synthesized mult_add library component.



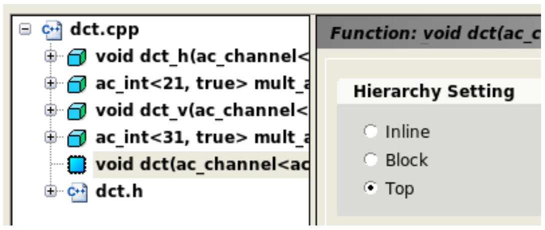
Instance Hierarchy	Module	Module: mult_add<ac_int<8,true>,
dct_h	mult_add<ac_int<8,true:	Block Type: [CCORE] mult_add<ac_ir
mult_add<ac_int<8,tru		Timing Report

- Go to Architectural Constraints, unroll the COPY_ROW0 and COL0 loops and pipeline "main" with II=8.


Instance Hierarchy
dct_h

Module
dct_h
Interface
core
Arrays
main (II=8)
ROW0 (II=8)
COPY_ROW0
COL0

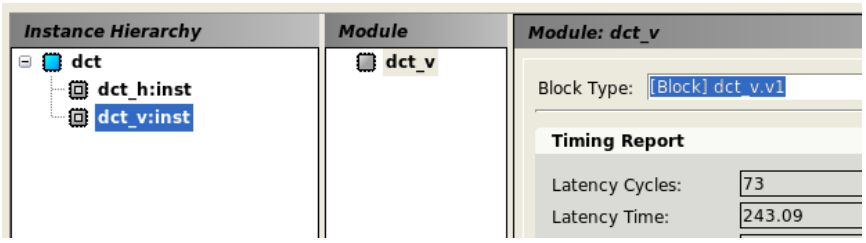
- Generate RTL.
- Go to "Mapping" and make "dct_v" the top-level design.
- Repeat the previous steps for "dct_h" on "dct_v".
- Go to "Mapping" and make "dct" the top-level design.


dct.cpp
void dct_h(ac_channel<
ac_int<21, true> mult_
void dct_v(ac_channel<
ac_int<31, true> mult_
void dct(ac_channel<ac
dct.h

Function: void dct(ac_c

Hierarchy Setting
Inline
Block
[✓ Top]

- Go to "Libraries" and note that "dct_h" and "dct_v" library componets have been added to the list of libraries.
- GO to "Mapping" and map the "dct_h" and "dct_v" instances to their respective library component.


Instance Hierarchy
dct
dct_h:inst
dct_v:inst

Module
dct_v

Module: dct_v

Block Type: [Block] dct_v.v1

Timing Report

Latency Cycles: 73
Latency Time: 243.09

- Generate RTL. Note that this step is very fast since Catapult only needs to synthesize the interconnect and simply stiches the RTL already created for "dct_h" and "dct_v".

DONE Toolkit