

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

Register-to-Variable Mappings					
Register	Size(bits)	Gated	Register	CG Opt	Done Variables
acc#1.sva	32	Y		Y	acc#1.sva
inner1:mul.itm#1	32	Y			inner1:mul.itm#1
acc#2.sva#2	31	Y		Y	acc#2.sva#2
inner2.reg	31	Y			inner2:asn#1.itm
					inner2:mul.itm

- Double-click on “acc#1.sva” which is a 32-bit register. This cross-probes to the “acc” variable in dct.cpp.

```
int mem[8][8];
int buf[8]; //Local storage for one row of input
int acc;
```

- Look at the Catapult Table View and write down the design area. This number will be compared with the design after quantization using bit-accurate data types.

Report: General						
Solution /	Latency C...	Latency T...	Throughp...	Throughp...	Total Area	Slack
dct.v1 (extract)	2765	9207.45	2770	9224.10	12367.39	0.21

- Go to the next Toolkit example.

Quantizing the Design Using Bit-Accurate Data Types

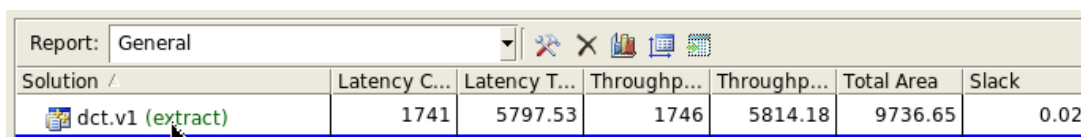
This Toolkit converts the DCT to use bit-accurate data types on the interface and internally to reduce area. The bit-accurate design is then optimized using HLS constraints to achieve the first performance requirement of less than 1200 cycles. The 128 cycle target is attempted which results in a scheduling failure, after which analysis is used to pinpoint the cause of the failure.

Synthesize the Design

- CD into the DCT_Quantization 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 "Synthesize the design unconstrained" script and click on "Launch Project" in the toolkit window. This sets the technology to Sample 65nm, adds single-port RAM libraries, sets the clock to 300 MHz, and synthesizes the unconstrained design.

Code Change Using Bit-accurate Data Types

- Open the dct.cpp file and observe how most of the internal and interface data types have been converted to use bit accurate data types. The bit widths for "mem", "buf" and "acc" have all been reduced to the minimum number of bits without changing the functionality of the design.
- Go to the Table View and look at the area. The area has been reduced by over 20% by constraining the bit-widths using bit-accurate data types.



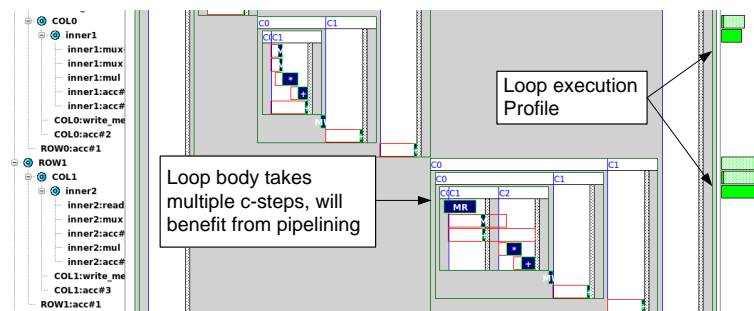
The screenshot shows the Catapult Table View with the 'Report' set to 'General'. The table displays performance metrics for the solution 'dct.v1 (extract)'. The metrics are: Latency Cycles (1741), Latency Time (5797.53), Throughput Cycles (1746), Throughput Time (5814.18), Total Area (9736.65), and Slack (0.02).

Solution /	Latency C...	Latency T...	Throughp...	Throughp...	Total Area	Slack
dct.v1 (extract)	1741	5797.53	1746	5814.18	9736.65	0.02

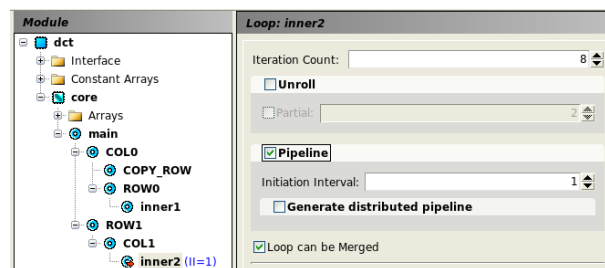
Performance Analysis and Optimization

Now that the design has been constrained to be bit-accurate we can begin applying HLS constraints to achieve the performance goal of 1200 cycles of throughput to perform the DCT. The current throughput is 1746 cycles. There are two ways to improve the throughput, one is to add parallelism via **loop unrolling**, and the second is to use **loop pipelining** to allow the execution of loop iterations to overlap. Loop pipelining is generally a cheaper constraint in terms of resource area so it is always a good idea to use loop pipelining constraints before loop unrolling constraints (See chapter 4 of the HLS Bluebook for a detailed discussion of loop pipelining and loop unrolling). Some preliminary analysis to identify where to optimize the design should be done using the Gantt chart before applying loop pipelining constraints.

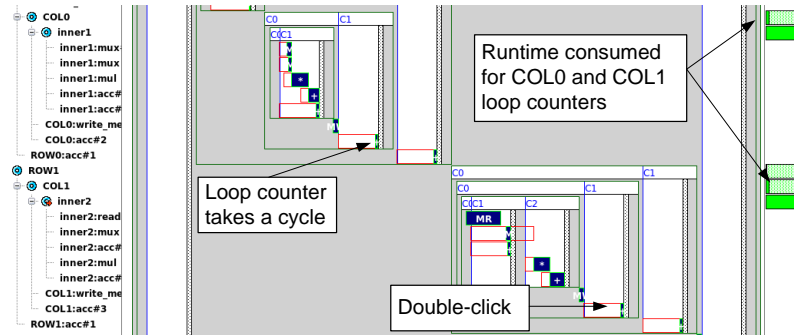
- Click on “Schedule” in the Task.
- Expand all loops in the Gantt Chart.
- Look at the green bars on the right side of the Gantt chart. These are the loop runtime profiles and indicate how much time is spent executing in a given loop. The bigger the green bar the more time is spent in the loop, making the loop a good candidate for optimization. Note that the most time is spent in the “inner2” loop. Also note that the loop body for “inner2” takes two c-steps. Loop pipelining allows us to overlap the execution of the loop body so that the next loop iteration can start before the current one finishes.



- Click on “Architecture” in the task bar.
- Pipeline “inner2” with II=1.



- Click on “Schedule” in the task bar.
- Look at the Table View. The throughput is now 1298, which is close to the first performance goal.
- Go to the Gantt chart and expand all loops. Note that although most of the runtime is still consumed by the “inner1” and “inner2” loops, there is a single adder in c-step “C1” of the “COL0” and “COL1” loops.

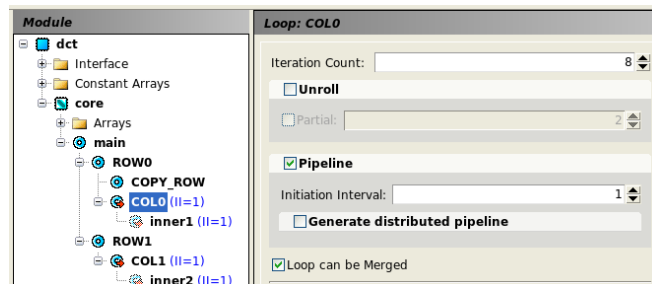


- Double-click on the adder in “C1” of the “COL1” loop. This will cross-probe back to the C++ source. Note that the adder is from the “COL1” loop counter.

```
ROW1:for (int i=0 ; i < 8; ++i )
COL1:for (int j=0; j < 8; ++j ) {
    acc1 = 0;
    inner2:for (int k=0 ; k < 8 ; ++k )
        acc1 += coeff[i][k] * mem[j][k];
    output[i][j] = acc1 >> 15 ;
}
```

Each iteration of the COL0 and COL1 loops is spending an extra cycle just to compute the loop counter. There are 8 iterations for each of these loops and they are each run 8 times inside the ROW0 and ROW1 loops respectively. So $8 \times 8 + 8 \times 8 = 128$ cycles that are spent just computing the loop counters. What this shows us is that the loop bodies of un-pipelined nested loops do not overlap. Pipelining the nested loops will “flatten” them into a single structure allowing the loop bodies to be overlapped. This is known as “Loop Flattening” (See Nested Loop Pipelining and Loop Flattening in chapter 4 of the HLS Bluebook).

- Go to Architectural Constraints and pipeline the COL0 and COL1 loop with II=1.



- Schedule the design and look at the Table View. The performance goal of 1200 cycles throughput has been met.
- Click on “RTL” in the Task Bar to generate the RTL.
- Look at the Table View. You can see that although the area has increased slightly there is a significant improvement in performance. Loop Pipelining should usually be tried first to improve performance since it tends to have less of an impact on area.

Report: General						
Solution /	Latency C...	Latency T...	Throughp...	Throughp...	Slack	Total Area
dct.v1 (extract)	1741	5797.53	1746	5814.18	-0.03	9712.12
dct.v2 (allocate)	1293	4305.69	1298	4322.34		7759.34
dct.v3 (extract)	1125	3746.25	1130	3762.90	0.13	10103.15

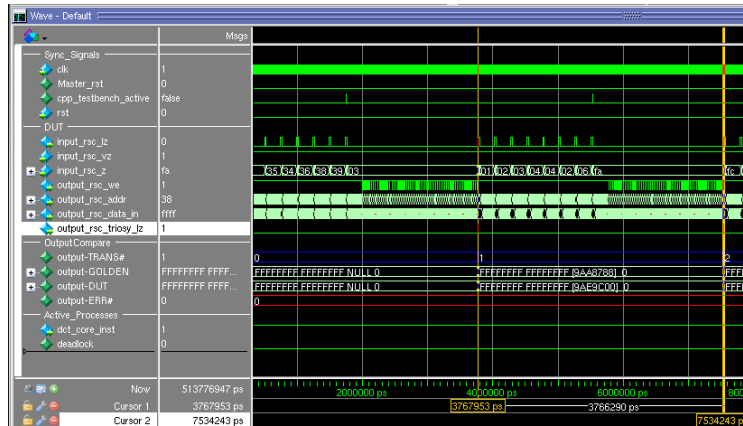
RTL Reporting

- Go to the Output Files folder and open the RTL report. Go to the Bill of Materials section and look for the multipliers used in the design. You will see two columns listed on the right side of the BOM, "Post Alloc" and "Post Assign". "Post Alloc" is the number of multipliers needed by the scheduler and "Post Assign" is the number after resource sharing and RTL generation. You can see that the scheduler needed one 8x10 multiplier (for the "inner1" loop) and one 10x16 multiplier (for inner2 loop). These multipliers can be shared because the "inner1" and "inner2" loops are never running at the same time. Thus the "Post Assign" number of multipliers is one 10x16 multiplier.

Bill Of Materials (Datapath)						
Component Name	Area	Score	Delay	Post Alloc	Post Assign	
[Lib: mgc_ioport]						
mgc_in_wire_wait(1,8)	0.000	0.000		1	1	
mgc_io_sync(0)	0.000	0.000		1	1	
[Lib: mgc_sample-065nm-dw_beh_dc]						
mgc_add(21,0,17,1,21,3)	292.160	0.689		1	0	
mgc_add(3,0,1,0,4,4)	20.175	0.166		0	2	
mgc_add(3,0,2,1,4,4)	27.273	0.178		2	0	
mgc_add(31,0,25,1,31,2)	476.554	0.578		0	1	
mgc_add(31,0,25,1,31,3)	426.423	0.990		1	0	
mgc_and(1,2,4)	2.400	0.030		0	24	
mgc_and(1,3,4)	2.800	0.049		0	4	
mgc_and(21,2,4)	50.400	0.030		1	0	
mgc_and(21,3,4)	58.800	0.049		0	1	
mgc_and(3,2,4)	7.200	0.030		0	6	
mgc_and(3,3,4)	8.400	0.049		0	1	
mgc_and(31,2,4)	74.400	0.030		1	1	
mgc_and(4,2,4)	9.600	0.030		1	0	
mgc_mul(10,1,16,1,25,3)	1432.352	1.099		1	0	
mgc_mul(10,1,16,1,25,4)	1278.510	1.692		0	1	
mgc_mul(8,1,10,1,17,4)	672.671	1.140		1	0	
mgc_mux(1,1,2,3)	3.997	0.081		0	1	

Automated Verification

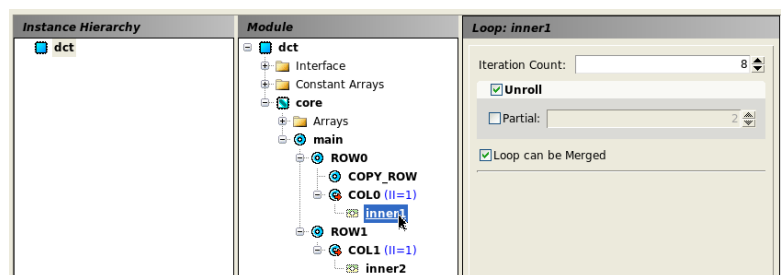
- Go to the Verification Folder and launch SCVerify on the generated RTL. Run the simulation and verify the throughput. You can do this by measuring the time between "Transaction Done" signal pulses for the output memory write "output_triosy_lz". "Transaction Done" is a hardware verification signal added by Catapult to help synchronize the automated verification comparisons when an interface does not have an explicit synchronization signal. For a clock of 300MHz, 3.33 ns, the time should be less than $1200 \times 3.33 = 3996$ ns. Also observe that the reading of the input data is very intermittent. This can be seen by the amount of time that the input_rsc_lz (read request) signal is high. In other words, this is a very low performance DCT.



Failed Scheduling and Analysis

The performance goal of a throughput less than or equal to 1200 cycles was satisfied by using Loop Pipelining to improve the design throughput. We will need to improve the design throughput by an order of magnitude in order to achieve the second performance goal of 128 cycles. Such a large increase in performance usually implies that parallelism must be added to the design. Loop Unrolling is the primary mechanism for controlling parallelism in HLS. The Gantt chart analysis in the previous steps showed that the main amount of processing time is spent in the “inner1” and “inner2” loops. These loops are both performing 8 multiplies and additions. The previous performance requirements were slow enough that these loops were left “rolled”, allowing the design to be implemented using a single hardware multiplier. For much higher performance these multiplies should be made to run in parallel.

- Go to Architectural Constraints and fully unroll the “inner1” and “inner2” loops.



- Schedule the design
- Scheduling will fail and Catapult will print an error in the transcript indicating that there are not enough memory resources needed to schedule the design. It says it needs 8 but only 1 is available. What this means is that the schedule was trying to read or write the memory 8 times in the same clock cycle. This is impossible to do for a single-port memory.