ECE4810J SoC Design

Fall 2024

Lab #3 PYNQ Overlays & Optimizing HLS Performance through Pipelining

Due: 11:59pm Nov. 4th, 2024

Logistics:

- This lab is a team exercise.
- Please use the discussion board on Piazza for Q&A.
- All reports and code (if available) MUST be submitted to the assignment of Canvas.
- Internet usage is allowed and encouraged.
- Late penalty (10% per day) will be applied for this lab.

Contents

1 Overview

In this lab, you will learn about PYNQ overlays and exercise the HLS flow on Zynq using Vitis HLS. The goals of this lab are to:

- Load overlay, use overlay and create overlays
- Understand the effect of the INLINE directive
- Improve performance using PIPELINE directive
- Distinguish between DATAFLOW directive and Configuration Command functionality

2 Optimize Performance through Pipelining

2.1 YUV model

The YUV model defines one luminance component (Y), meaning physical linear-space brightness, and two chrominance components called U (blue projection) and V (red projection), respectively. It can be used to convert to and from the RGB model, and with different color spaces. The closely related Y'UV model uses the luma component (Y') – nonlinear perceptual brightness, with the prime symbols (') denoting gamma correction [22]. Today, the term YUV is commonly used in the computer industry to describe colorspaces that are encoded using YCbCr, used for digital encoding of color information suited for video and image compression and transmission formats like MPEG & JPEG.

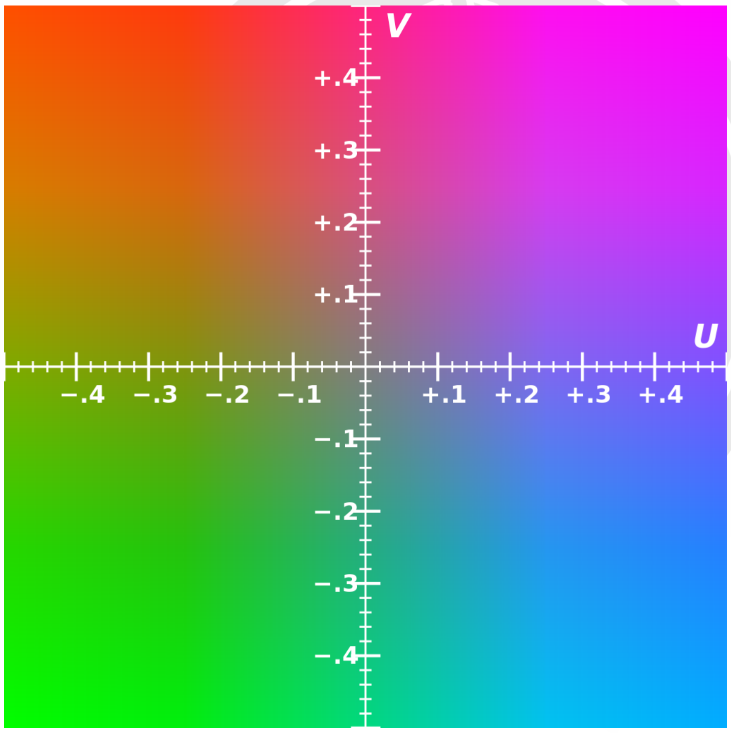


Figure 1: Example of U-V color plane, $Y' = 0.5$, represented within RGB color gamut [17].

Y'UV files can be encoded in 12, 16, or 24 bits per pixel. The common formats are:

Name	Size	Ratio		
		A:B:C	Horizontal	Vertical
Y'UV444/YUV444	3 bytes per pixel	4:4:4	Full	Full
Y'UV422/YUV422	4 bytes per 2 pixels	4:2:2	2:1	Full
Y'UV420p/YUV420	6 bytes per 4 pixels, reordered	4:2:0	2:1	2:1
YUV411	6 bytes per 4 pixels	4:1:1	4:1	Full

Table 1: YUV A:B:C (or 4:x:x) based on the NTSC (National Television System Committee) standard, a standard for analog television broadcasts.

RGB888 is R-8bit, G-8bit, B-8bit, i.e., RGB24.

For the conversion from RGB to Y'UV:

$$Y' = 0.299R + 0.587G + 0.114B \tag{1}$$
$$U = -0.147R - 0.289G + 0.436B \tag{2}$$
$$V = 0.615R - 0.515G - 0.100B \tag{3}$$

On older, non-SIMD architectures, floating point arithmetic is much slower than using fixed-point arithmetic, so most digital implementations of RGB →Y'UV used integer math, in particular fixed-point approximations. Approximation means that the precision of the used numbers (input data, output data, and constant values) is limited, and thus a precision loss of typically about the last binary digit is accepted by whoever makes use of that option in typically a trade-off to improved computation speeds [22]. Y' values are conventionally shifted and scaled to the range [16, 235] (referred to as studio swing or "TV levels") rather than using the full range of [0, 255] (referred to as full swing or "PC levels"). This practice was standardized in SMPTE-125M in order to accommodate signal overshoots ("ringing") due to filtering. The value 235 accommodates a maximal black-to-white overshoot of $255 - 235 = 20$, or $20/(235 - 16) = 9.1\%$, which is slightly larger than the theoretical maximal overshoot (Gibbs phenomenon) of about 8.9% of the maximal step. The toe-room is smaller, allowing only $16 / 219 = 7.3\%$ overshoot, which is less than the theoretical maximal overshoot of 8.9%. This is why 16 is added to Y' and why the Y' coefficients in the basic transform sum to 220 instead of 255 [10]. U and V values, which may be positive or negative, are summed with 128 to make them always positive, giving a studio range of 16–240 for U and V. (These ranges are important in video editing and production since using the wrong range will result either in an image with "clipped" blacks and whites, or a low-contrast image.)

For getting the traditional "studio-swing" 8-bit representation of YCbCr for SDTV/BT.601 the following operations can be used:

1. Basic transform from 8-bit RGB to 16-bit values (Y': unsigned, Cb/Cr: signed, matrix values got rounded so that the later-on desired Y' range of [16..235] and

Cb/Cr range of [16..240] is reached):

$$\begin{bmatrix} Y' \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 66 & 129 & 25 \\ -38 & -74 & 112 \\ 112 & -94 & -18 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \tag{4}$$

2. Scale down to 8 bits with rounding ("+128") (Y': unsigned, Cb/Cr: signed):

$$\begin{aligned} Y'_t &= (Y' + 128) \gg 8, \\ Cb_t &= (U + 128) \gg 8, \\ Cr_t &= (V + 128) \gg 8. \end{aligned} \tag{5}$$

3. Add an offset to the values to eliminate any negative values (all results are 8-bit unsigned):

$$\begin{aligned} Y'_u &= Y'_t + 16, \\ Cb_u &= Cb_t + 128, \\ Cr_u &= Cr_t + 128. \end{aligned} \tag{6}$$

An alternative formulation provided by Microsoft DirectDraw (a component of DirectX API) is [4]:

$$Y' = ((66R + 129G + 25B + 128) \gg 8) + 16 \tag{7}$$
$$U = ((-38R - 74G + 112B + 128) \gg 8) + 128 \tag{8}$$
$$V = ((112R - 94G - 18B + 128) \gg 8) + 128 \tag{9}$$

These formulas produce 8-bit results using coefficients that require no more than 8 bits of (unsigned) precision. Intermediate results require up to 16 bits of precision. For the conversion from Y'UV to RGB, using the coefficients c, d, and e, and noting that $[]_0^{255}$ denotes clamping a value to the 8-bit range of 0 to 255, the following formulae provide the conversion from Y'UV to RGB (NTSC version):

$$c = Y' - 16 \tag{10}$$
$$d = U - 128 \tag{11}$$
$$e = V - 128 \tag{12}$$

$$R = [(298c + 409e + 128) \gg 8]_0^{255} \tag{13}$$
$$G = [(298c - 100d - 208e + 128) \gg 8]_0^{255} \tag{14}$$
$$B = [(298c + 516d + 128) \gg 8]_0^{255} \tag{15}$$

In Microsoft documentation, the clamping is also denoted as a function `clip()`:

$$R = \text{clip}((298c + 409e + 128) \gg 8) \tag{16}$$
$$G = \text{clip}((298c - 100d - 208e + 128) \gg 8) \tag{17}$$
$$B = \text{clip}((298c + 516d + 128) \gg 8) \tag{18}$$

These formulas use some coefficients that require more than 8 bits of precision to produce each 8-bit result, and intermediate results require more than 16 bits of precision. Given the knowledge, you can fill your lab3_starter/yuv_filter.c now. Note that the macro `CLIP` has already been well defined in lab3_starter/yuv_filter.h.

Besides, you also need to prepare your input data `lab3_starter/yuv_filter/input.dat` from the standard test image `lab3_starter/lena_std.tif` [8]. One possible way is to use MATLAB:

```
lena_std = imread('lena_std.tif');
std_perm = permute(lena_std, [2 3 1]);
dlmwrite('input.dat', std_perm, 'delimiter', '\n')
```

The matrix is permuted in this way because we want the output dimension order conforms to the dimension order of the function `image_read` in `lab3_starter/image_aux.c`. **Remember to run `make clean` before `make` if you `make` multiple times.**

After you generate the `output.dat`, convert it back to a TIF file `lena_rec.tif` (other image formats are also OK). The sample MATLAB script is

```
output = dlmread('lab3/output.dat');
img_rec = reshape(output, 3, 512, 512);
img_rec_perm = permute(img_rec, [3 2 1]);
imwrite(uint8(img_rec_perm), 'lena_rec.tif');
```

2.2 Create a Vitis HLS project from command line

1. Open Vitis HLS 2022.1 Command Prompt.
2. Change directory to your lab directory. A self-checking program `yuv_filter_test.c` is provided. Using that, we can validate the design. A Makefile is also provided. Using the Makefile, the necessary source files can be compiled, and the compiled program can be executed. You can examine the contents of these files and the project directory.
3. Run `make` to compile and execute the program. If it fails, read Appendix B.1. You should see the output

   ```
   gcc -ggdb -w -I/f/Xilinx/Vitis_HLS/2022.1/include   -c -o
   ↪  yuv_filter.o yuv_filter.c
   gcc -ggdb -w -I/f/Xilinx/Vitis_HLS/2022.1/include   -c -o
   ↪  yuv_filter_test.o yuv_filter_test.c
   gcc -ggdb -w -I/f/Xilinx/Vitis_HLS/2022.1/include   -c -o
   ↪  image_aux.o image_aux.c
   gcc -lm   yuv_filter.o yuv_filter_test.o image_aux.o   -o yuv_filter
   ./yuv_filter
   Test passed!
   ```

   Note that the source files (yuv_filter.c, yuv_filter_test.c, and image_aux.c) were compiled, then the yuv_filter executable program was created, and then it was executed. Only Verilog RTL and VHDL RTL are generated. If you also want to generate SystemC RTL, change Line 13 of Makefile to:

```
USE_SYSTEMC                                           = true
```

A Vitis HLS Tcl script file pynq_yuv_filter.tcl is provided and can be used to create a Vitis HLS project.

4. Run `vitis_hls -f pynq_yuv_filter.tcl` in the Vitis HLS Command Prompt window to create the project targeting xc7z020clg400-1 part. The project will be created, and the vitis_hls.log file will be generated. You can see the output in vitis_hls.log.

5. Open the created project in GUI mode from the Vitis HLS Command Prompt window by running `vitis_hls -p yuv_filter.prj`. The Vitis HLS will open in GUI mode, and the project will be opened. Note that different old Vivado HLS, new Vitis HLS automatically enables loop pipelining, so if you keep default settings, you will see the following in vitis_hls.log and yuv_filter.prj/solution1/solution1.log:

```
INFO: [XFORM 203-510] Pipelining loop 'RGB2YUV_LOOP_Y'
 ↪  (yuv_filter.c:34) in function 'yuv_filter' automatically.
INFO: [XFORM 203-510] Pipelining loop 'YUV_SCALE_LOOP_Y'
 ↪  (yuv_filter.c:118) in function 'yuv_filter' automatically.
INFO: [XFORM 203-510] Pipelining loop 'YUV2RGB_LOOP_Y'
 ↪  (yuv_filter.c:72) in function 'yuv_filter' automatically.
```

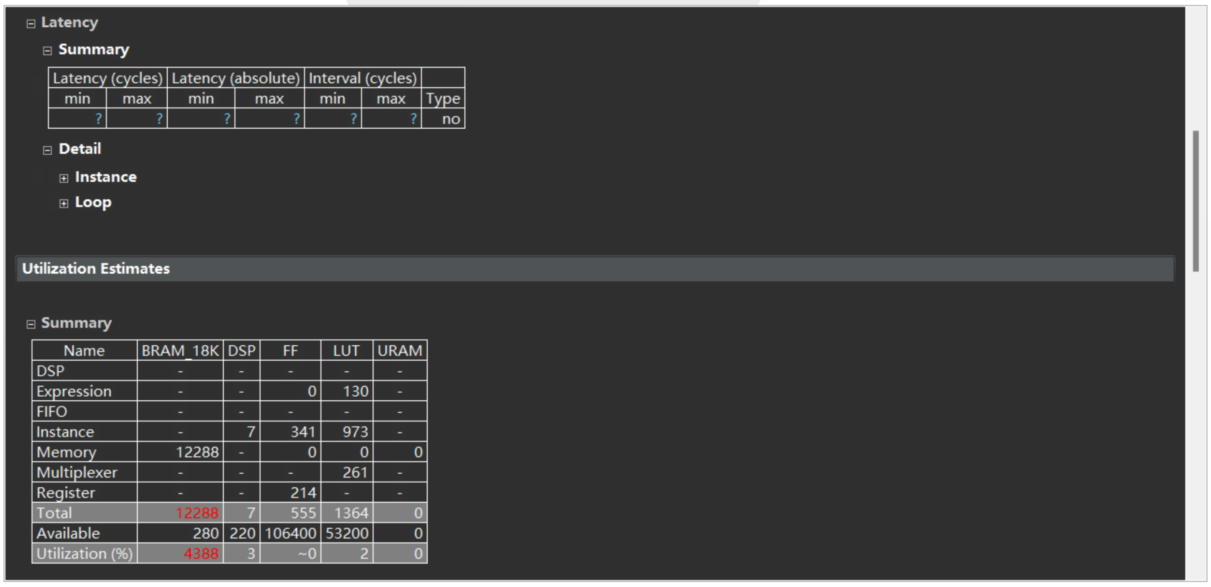
To prevent this, open tab yuv_filter.c. Menu Window->Show View->Directive, open Directive tab, for each of 'rgb2yuv', 'yuv2rgb', and 'yuv_scale', right-click it, Insert Directive... In the pop-up window Vitis HLS Directive Editor, Directive: PIPELINE, check Options: off (optional). After doing this, you will see a new directive: HLS PIPELINE off. Run C Synthesis->Active Solution again.

2.3 Analyze the created project and results

1. In Vitis HLS GUI, expand the source folder in the *Explorer* view and double click yuv_filter.c to view the content. The design is implemented in 3 functions: rgb2yuv, yuv_scale, and yuv2rgb. Each of these filter functions iterates over the entire source image (which has maximum dimensions specified in image_aux.h), requiring a single source pixel to produce a pixel in the resulting image. The scale function simply applies individual scale factors, supplied as top-level arguments to the Y'UV components. Notice that most of the variables are user-defined (typedef) and aggregate (e.g. structure, array) types. A viable workaround is conditionally compiled into the code, leveraging the `__SYNTHESIS__` macro. Vitis HLS automatically defines the `__SYNTHESIS__` macro when reading any code. This ensures the original `malloc()` code is used outside of synthesis, but Vitis HLS will use the workaround when synthesizing.

2. Expand the solution1 > syn > report folder in the *Explorer* view and double-click

yuv_filter_csynh.rpt entry to open the synthesis report. Each of the loops in this design has variable bounds – the width and height are defined by members of input type image_t. When variable bounds are present on loops, the total latency of the loops cannot be determined: this impacts the ability to perform analysis using reports. Hence, "?" is reported for various latencies.


Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
?	?	?	?	?	?	no

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	130	-
FIFO	-	-	-	-	-
Instance	-	7	341	973	-
Memory	12288	-	0	0	0
Multiplexer	-	-	-	261	-
Register	-	-	214	-	-
Total	12288	7	555	1364	0
Available	280	220	106400	53200	0
Utilization (%)	4388	3	~0	2	0

Figure 2: Latency computation.

You can scroll down to the bottom of the report and click the "Export Dialog" to export the report to an HTML format file. Then, you can convert HTML to PDF.

2.4 Apply `TRIPCOUNT` pragma

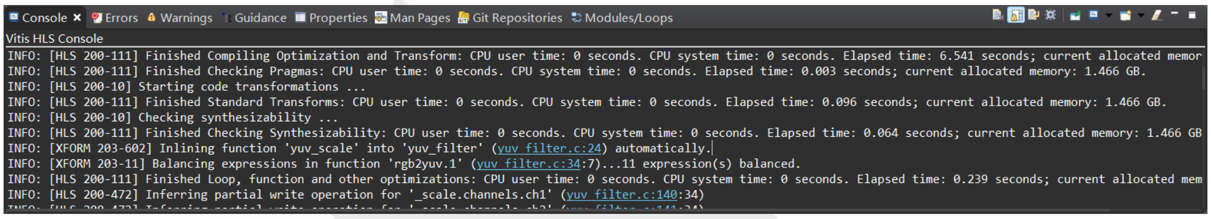
Open the source file and uncomment pragma lines, re-synthesize, and observe the resources used as well as estimated latencies. Answer the questions listed in the detailed section of this step.

1. To assist in providing loop-latency estimates, Vitis HLS provides a `TRIPCOUNT` directive, which allows limits on the variables' bounds to be specified by the user. In this design, such directives have been embedded in the source code, in the form of `#pragma` statements.

2. Uncomment the `#pragma` lines (50, 53, 90, 93, 130, 133) to define the loop bounds and save the file.

3. Synthesize the design by selecting Solution > Run C Synthesis > Active Solution. View the synthesis report when the process is completed.

4. Scroll the *Console* window and note that the yuv_scale function is automatically inlined into the yuv_filter function.

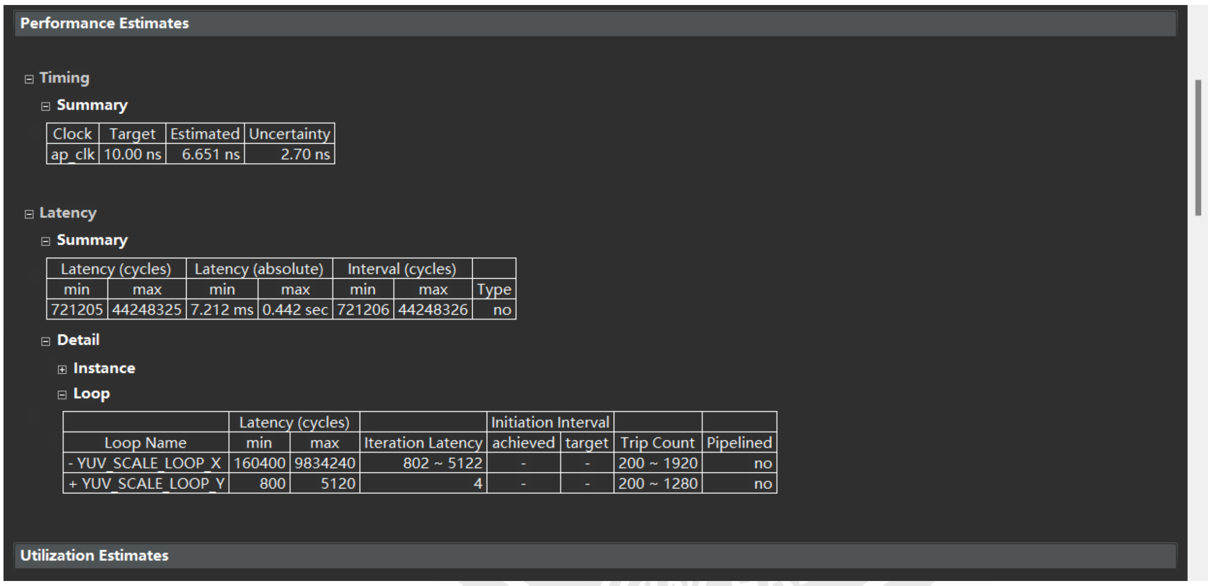


```
INFO: [HLS 200-111] Finished Compiling Optimization and Transform: CPU user time: 0 seconds. CPU system time: 0 seconds. Elapsed time: 6.541 seconds; current allocated memor
INFO: [HLS 200-111] Finished Checking Pragmas: CPU user time: 0 seconds. CPU system time: 0 seconds. Elapsed time: 0.003 seconds; current allocated memory: 1.466 GB.
INFO: [HLS 200-10] Starting code transformations ...
INFO: [HLS 200-111] Finished Standard Transforms: CPU user time: 0 seconds. CPU system time: 0 seconds. Elapsed time: 0.096 seconds; current allocated memory: 1.466 GB.
INFO: [HLS 200-10] Checking synthesizability ...
INFO: [HLS 200-111] Finished Checking Synthesizability: CPU user time: 0 seconds. CPU system time: 0 seconds. Elapsed time: 0.064 seconds; current allocated memory: 1.466 GB
INFO: [XFORM 203-602] Inlining function 'yuv_scale' into 'yuv_filter' (yuv_filter.c:24) automatically.
INFO: [XFORM 203-11] Balancing expressions in function 'rgb2yuv.1' (yuv_filter.c:34:7)...11 expression(s) balanced.
INFO: [HLS 200-111] Finished Loop, function and other optimizations: CPU user time: 0 seconds. CPU system time: 0 seconds. Elapsed time: 0.239 seconds; current allocated mem
INFO: [HLS 200-472] Inferring partial write operation for '_scale.channels.ch1' (yuv_filter.c:140:34)
```

Figure 3: Vitis HLS automatically inlining function.

5. Observe that there are three entries – rgb2yuv.rpt, yuv_filter.rpt, and yuv2rgb.rpt under the syn report folder in the Explorer view. There is no entry for yuv_scale.rpt since the function was inlined into the yuv_filter function. You can access the lower-level module's report by either traversing down in the top-level report under components (under Utilization Estimates > Details > Component) or from the reports container in the project explorer.

6. Expand the Summary of loop latency and note the latency and trip count numbers for the yuv_scale function. Note that the YUV_SCALE_LOOP_Y loop latency is 4x the specified TRIPCOUNT, implying that four cycles are used for each iteration of the loop.



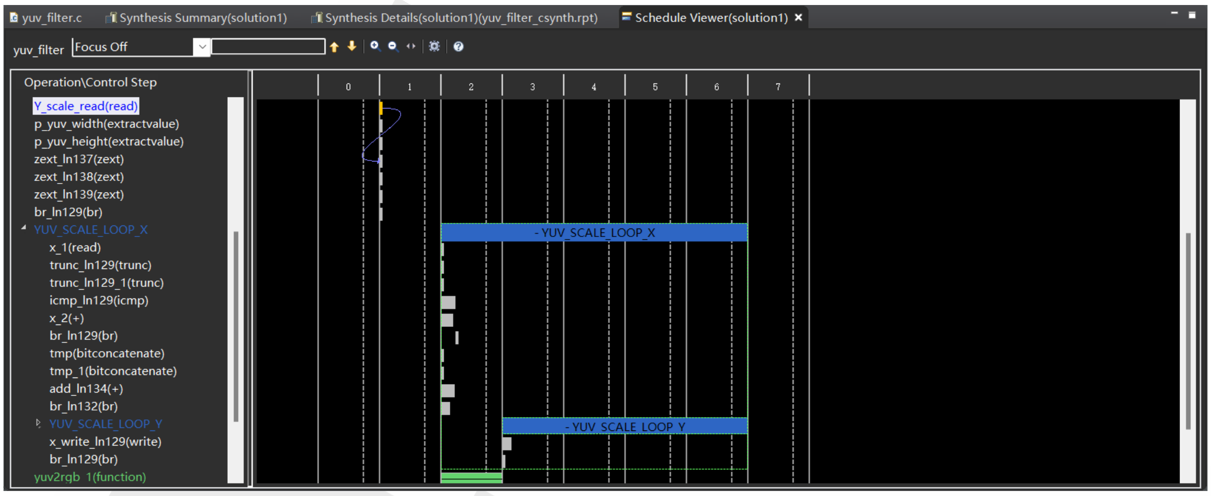
Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	6.651 ns	2.70 ns

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
721205	44248325	7.212 ms	0.442 sec	721206	44248326	no

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- YUV_SCALE_LOOP_X	160400	9834240	802 ~ 5122	-	-	200 ~ 1920	no
+ YUV_SCALE_LOOP_Y	800	5120	4	-	-	200 ~ 1280	no

Figure 4: Loop latency.

7. You can verify this by opening an analysis perspective view in the Solution menu->Open Schedule Viewer, expanding the YUV_SCALE_LOOP_X entry, and then expanding the YUV_SCALE_LOOP_Y entry.


yuv_filter.c
Synthesis Summary(solution1)
Synthesis Details(solution1)(yuv_filter_csynth.rpt)
Schedule Viewer(solution1)
yuv_filter
Focus Off
Operation\Control Step
0
1
2
3
4
5
6
7
- YUV_SCALE_LOOP_X
- YUV_SCALE_LOOP_Y

Y_scale_read(read)
p_yuv_width(extractvalue)
p_yuv_height(extractvalue)
zext_ln137(zext)
zext_ln138(zext)
zext_ln139(zext)
br_ln129(br)
YUV_SCALE_LOOP_X
x_1(read)
trunc_ln129(trunc)
trunc_ln129_1(trunc)
icmp_ln129(icmp)
x_2(+)
br_ln129(br)
tmp(bitconcatenate)
tmp_1(bitconcatenate)
add_ln134(+)
br_ln132(br)
YUV_SCALE_LOOP_Y
x_write_ln129(write)
br_ln129(br)
yuv2rgb_1(function)

Figure 5: Design analysis view of the YUV_SCALE_LOOP_Y loop.

8. In the Synthesis Details, expand Detail > Instance section of the Utilization Estimates and click on the grp_rgb2yuv_fu_251 (rgb2yuv_1) entry to open the report Synthesis Details(solution1)(rgb2yuv_1_csynth.rpt).

9. Similarly, click on the grp_yuv2rgb_fu_271 entry to open the report Synthesis Details(solution1)(yuv2rgb_1_csynth.rpt).

10. Report the following metrics of yuv_filter, rgb2yuv, and yuv2rgb functions:

    (a) Estimated clock period

    (b) Worst case latency

    (c) Number of DSP48E used

    (d) Number of BRAMs used

    (e) Number of FFs used

    (f) Number of LUTs used

2.5 Turn off `INLINE` and apply `PIPELINE` directive

Create a new solution by copying the previous solution settings. Prevent the automatic `INLINE` and apply `PIPELINE` directive. Generate the solution and understand the output.

1. Select Project menu > New Solution...; in the pop-up Solution Wizard, check "Copy directives and constraints from solution:" and select solution1 in the Options frame.

2. Make sure that the yuv_filter.c source is opened and visible, and click on the **Directive** tab.

3. Select function yuv_scale in the directives pane, right-click on it and select Insert Directive...

4. Click on the drop-down button of the Directive field in the pop-up Vitis HLS Directive Editor showing up listing various directives. Select INLINE directive.

5. Click on the off (optional) option to turn off the automatic inlining. Make sure that the Directive File is selected as the destination. Click OK. When an object (function or loop) is pipelined, all the loops below it, down through the hierarchy, will be automatically unrolled. In order for a loop to be unrolled, it must have fixed bounds: all the loops in this design have variable bounds, defined by an input argument variable to the top-level function. Note that the `TRIPCOUNT` directive on the loops only influences reporting, it does not set bounds for synthesis. Neither the top-level function nor any of the sub-functions are pipelined in this example. The pipeline directive must be applied to the inner-most loop in each function – the innermost loops have no variable-bounded loops inside which are required to be unrolled, and the outer loop will simply keep the inner loop fed with data.

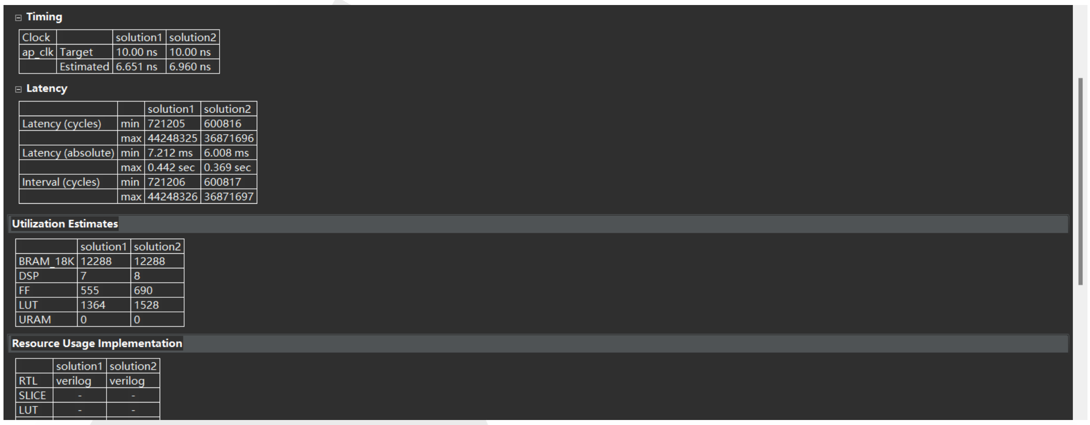
6. Expand the yuv_scale in the Directives tab, right click on `HLS PIPELINE off` directive; select "Remove Directive". Similarly, remove the `HLS PIPELINE off` directive of rgb2yuv and yuv2rgb.

7. Run C Synthesis for solution2. You can observe the output:

```
INFO: [SCHED 204-61] Pipelining loop
↪  'RGB2YUV_LOOP_X_RGB2YUV_LOOP_Y'.
INFO: [HLS 200-1470] Pipelining result : Target II = NA, Final II =
↪  1, Depth = 9, loop 'RGB2YUV_LOOP_X_RGB2YUV_LOOP_Y'
INFO: [SCHED 204-61] Pipelining loop
↪  'YUV_SCALE_LOOP_X_YUV_SCALE_LOOP_Y'.
INFO: [HLS 200-1470] Pipelining result : Target II = NA, Final II =
↪  1, Depth = 7, loop 'YUV_SCALE_LOOP_X_YUV_SCALE_LOOP_Y'
INFO: [SCHED 204-61] Pipelining loop
↪  'YUV2RGB_LOOP_X_YUV2RGB_LOOP_Y'.
INFO: [HLS 200-1470] Pipelining result : Target II = NA, Final II =
↪  1, Depth = 11, loop 'YUV2RGB_LOOP_X_YUV2RGB_LOOP_Y'
```

8. When the synthesis is completed, select Project > Compare Reports... to compare the two solutions. Select Solution1 and Solution2 from the Available Reports in the Solution Selection Dialog, and click on the Add» button.

9. Observe the latency reduced in the Vitis HLS Report Comparison.



Timing

Clock		solution1	solution2
ap_clk	Target	10.00 ns	10.00 ns
	Estimated	6.651 ns	6.960 ns

Latency

		solution1	solution2
Latency (cycles)	min	721205	600816
	max	44248325	36871696
Latency (absolute)	min	7.212 ms	6.008 ms
	max	0.442 sec	0.369 sec
Interval (cycles)	min	721206	600817
	max	44248326	36871697

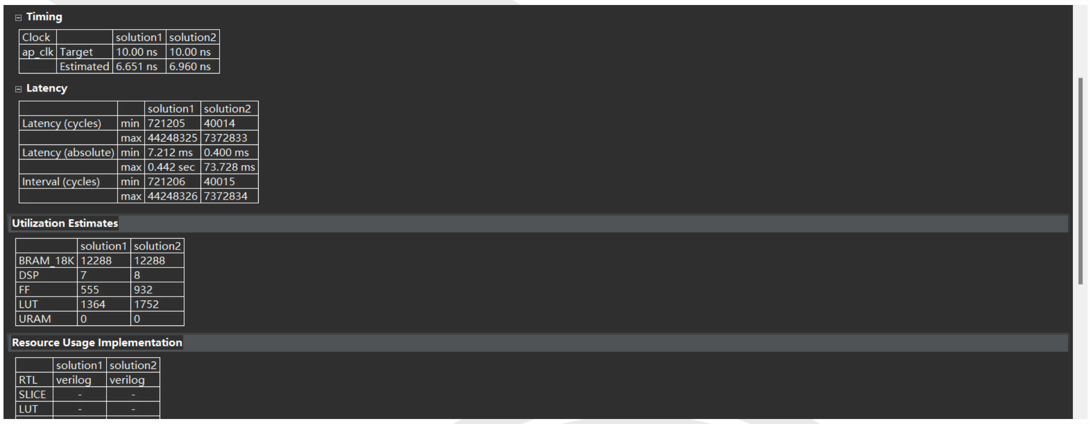
Utilization Estimates

	solution1	solution2
BRAM_18K	12288	12288
DSP	7	8
FF	555	690
LUT	1364	1528
URAM	0	0

Resource Usage Implementation

	solution1	solution2
RTL	verilog	verilog
SLICE	-	-
LUT	-	-

Figure 6: Performance comparison after pipelining yuv_scale only.



Timing

Clock		solution1	solution2
ap_clk	Target	10.00 ns	10.00 ns
	Estimated	6.651 ns	6.960 ns

Latency

		solution1	solution2
Latency (cycles)	min	721205	40014
	max	44248325	7372833
Latency (absolute)	min	7.212 ms	0.400 ms
	max	0.442 sec	73.728 ms
Interval (cycles)	min	721206	40015
	max	44248326	7372834

Utilization Estimates

	solution1	solution2
BRAM_18K	12288	12288
DSP	7	8
FF	555	932
LUT	1364	1752
URAM	0	0

Resource Usage Implementation

	solution1	solution2
RTL	verilog	verilog
SLICE	-	-
LUT	-	-

Figure 7: Performance comparison after pipelining.

In Solution1, the total loop latency of the inner-most loop was loop_body_latency x loop iteration count, whereas in Solution2 the new total loop latency of the inner-most loop is loop_body_latency + loop iteration count. Scroll down in the comparison report to view the resource utilization. Observe that the FFs, LUTs, and DSP48E utilization increased, whereas BRAM remained the same. Take a screenshot of the Performance Estimates and Utilization Estimates of the Vitis HLS Report Comparison.

2.6 Apply `DATAFLOW` Directive and Configuration Command

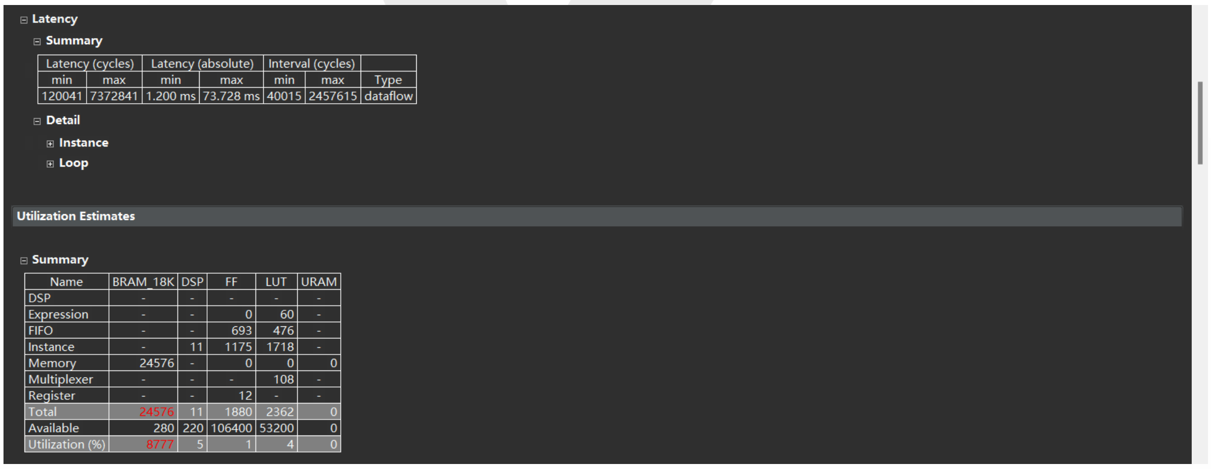
Create a new solution by copying the previous solution (Solution2) settings. Apply `DATAFLOW` directive. Generate the solution and understand the output.

1. Create a new solution by copying the previous solution (Solution2) settings.
2. Close all inactive solution windows by selecting Project menu > Close Inactive Solution Tabs.
3. Make sure that the yuv_filter.c source is opened in the information pane and select the *Directive* tab.

4. Insert the `DATAFLOW` directive for yuv_filter function.

5. Run C Synthesis. When the synthesis is completed, the synthesis report is automatically opened. Open the Synthesis Details(solution3)(yuv_filter_csynth.rpt) report from Explorer.

6. Observe additional information, Dataflow Type, in the *Performance Estimates->*Latency section is mentioned.



Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
120041	7372841	1.200 ms	73.728 ms	40015	2457615	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	60	-
FIFO	-	-	693	476	-
Instance	-	11	1175	1718	-
Memory	24576	-	0	0	0
Multiplexer	-	-	-	108	-
Register	-	-	12	-	-
Total	24576	11	1880	2362	0
Available	280	220	106400	53200	0
Utilization (%)	8777	5	1	4	0

Figure 8: Performance and Resource estimate after `DATAFLOW` directive applied.

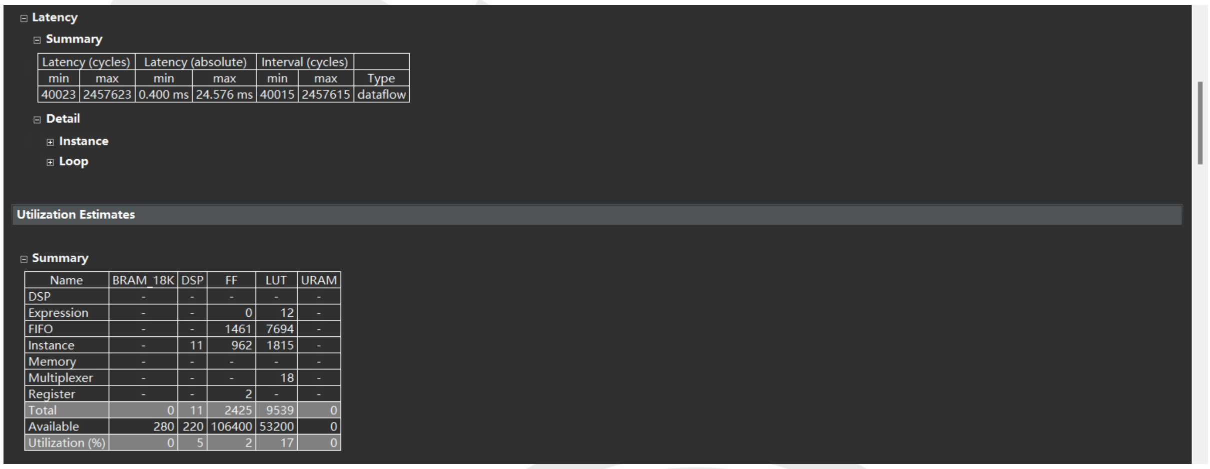
The Dataflow pipeline throughput indicates the number of clocks cycles between each set of input reads. If this throughput value is less than the design latency, it indicates the design can start processing new inputs before the current input data are output. While the overall latencies haven't changed significantly, the dataflow throughput shows that the design can achieve close to the theoretical limit ($1920 \times 1280 = 2457600$) of processing one pixel every clock cycle. Take a screenshot of the Performance Estimates and Utilization Estimates of the Synthesis Details Report after `DATAFLOW` directive is applied.

7. Scrolling down into the *Utilization Estimates* section, observe that the number of BRAMs required has doubled. This is due to the default ping-pong buffering in the data flow. When DATAFLOW optimization is performed, memory buffers are automatically inserted between the functions to ensure the next function can begin operation before the previous function has finished. The default memory buffers are ping-pong buffers sized to accommodate the largest producer or consumer array fully. Vitis HLS allows the memory buffers to be the default ping-pong buffers or FIFOs. Since this design has data accesses that are fully sequential, FIFOs can be used. Another advantage to using FIFOs is that the size of the FIFOs can be directly controlled (not possible in ping-pong buffers where random accesses are allowed).

8. The memory buffer type can be selected using the Vitis HLS Configuration command.

Apply the Dataflow configuration command, generate the solution, and observe the improved resource utilization.

1. Select Solution > Solution Settings... to access the configuration command settings.
2. In the Solution Settings (solution3) dialog, General->Configuration Settings->Commands. Expand config_dataflow, change default_channel to "fifo" with fifo_depth = 2, scalar_fifo_depth = 0, start_fifo_depth = 0, strict_mode is warning.
3. Run C Synthesis. If your resource estimate is still the same, see Appendix B.2. Note that the performance parameter has not changed; however, resource estimates show that the design is not using any BRAM, and other resources (FF, LUT) usage has also been reduced:



Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
40023	2457623	0.400 ms	24.576 ms	40015	2457615	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	12	-
FIFO	-	-	1461	7694	-
Instance	-	11	962	1815	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	18	-
Register	-	-	2	-	-
Total	0	11	2425	9539	0
Available	280	220	106400	53200	0
Utilization (%)	0	5	2	17	0

Figure 9: Resource estimation after Dataflow configuration command.

Take a screenshot of the Performance Estimates and Utilization Estimates of the Synthesis Details Report after configuring Dataflow commands.

2.7 (Optional) Export and implement the design in Vitis HLS

In Vitis HLS, export the design, select Verilog as a language, and run the implementation by selecting Evaluate option.

1. In Vitis HLS, select Solution > Export RTL to open the Export RTL dialog so the desired implementation can be run.
2. Go to Solution Settings (solution3)->General->Configuration Settings->Commands->config_export->rtl, choose either verilog or vhdl. Then, you have to go to Flow Navigator->IMPLEMENTATION->Run Implementation [24], in the pop-up window Run Implementation, select RTL Synthesis, Place & Route, select RTL as Verilog or VHDL, and choose an IP location. These arguments are the same in solution settings. If you do not run the implementation first, Export RTL will automatically do the implementation, but it specifies RTL Synthesis only by default. If you get an error:

```
ERROR: [IMPL 213-28] Failed to generate IP.
↪  yuv_filter.prj:solution3
```

see Appendix B.3. If your anti-virus software warns you that C:/Windows/SysWOW64/cscript.exe is executing target yuv_filter.prj/solution3/impl/verilog/project.runs/bd_0_hls_inst_0_synth_1/rundef.js, ignore it. After the implementation finishes [1], export the RTL. Take a screenshot of your implementation report.

2.8 Conclusion

In this part, you learned that even though this design could not be pipelined at the top level, a strategy of pipelining the individual loops and then using dataflow optimization to make the functions operate in parallel was able to achieve the same high throughput, processing one pixel per clock. When the `DATAFLOW` directive is applied, the default memory buffers (of ping-pong type) are automatically inserted between the functions. Using the fact that the design used only sequential (streaming) data accesses allowed the costly memory buffers associated with dataflow optimization to be replaced with simple 2-element FIFOs using the Dataflow command configuration.

3 Implement and Optimize Discrete Cosine Transform

3.1 Discrete cosine transform

A discrete cosine transform (DCT) expresses a finite sequence of data points in terms of a sum of cosine functions oscillating at different frequencies[20]. In signal processing and data compression, the most important variants are what's called DCT-II through DCT-IV. What's commonly called "the" DCT is the DCT-II; this is the transform used in, among other things, JPEG, MPEG-1 and MPEG-2. The DCT-III is the inverse of the DCT-II (up to scaling, depending on how the two are normalized; there are different conventions in use). Thus it's often called "the" IDCT (inverse DCT). The DCT-IV is its own inverse and forms the basis for the MDCT (modified DCT), a lapped transform that's at the heart of most popular perceptual audio codecs (MP3, AAC, Vorbis, and Opus, among others) [9]. DCT helps separate the image into parts (or spectral sub-bands) of differing importance (with respect to the image's visual quality). DCT is similar to the discrete Fourier transform: it transforms a signal or image from the spatial domain to the frequency domain. To an image f, f(x,y) represents the pixel value at (x,y), the equation of the DCT transformation is [3]:

$$F(u,v) = \alpha(u)\alpha(v)\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)\cos\frac{(2x+1)u\pi}{2M}\cos\frac{(2y+1)v\pi}{2N}, \tag{19}$$

where $u = 0, 1 \ldots . M-1, v = 0, 1 \ldots N-1$. $\alpha(u) = \begin{cases} \frac{1}{\sqrt{M}} & u = 0 \\ \sqrt{\frac{2}{M}} & u \neq 0 \end{cases}$ $\alpha(v) = \begin{cases} \frac{1}{\sqrt{N}} & v = 0 \\ \sqrt{\frac{2}{N}} & v \neq 0 \end{cases}$.

For most images, much of the signal energy lies at low frequencies; these appear in the upper left corner of the DCT. To reverse the image, the equation of the inverse DCT (iDCT) transformation is:

$$f(x,y) = \sum_{u=0}^{M-1}\sum_{v=0}^{N-1} \alpha(u)\alpha(v)F(u,v)\cos\frac{(2x+1)u\pi}{2M}\cos\frac{(2y+1)v\pi}{2N}, \tag{20}$$

where $x = 0, 1 \ldots . M-1, y = 0, 1 \ldots N-1$. $\alpha(u) = \begin{cases} \frac{1}{\sqrt{M}} & u = 0 \\ \sqrt{\frac{2}{M}} & u \neq 0 \end{cases}$ $\alpha(v) = \begin{cases} \frac{1}{\sqrt{N}} & v = 0 \\ \sqrt{\frac{2}{N}} & v \neq 0 \end{cases}$.

If the input image is square, another way is to use DCT (basis) (transform) matrix, which is much faster. The M-by-M transform matrix T is given by

$$T_{pq} = \begin{cases} \frac{1}{\sqrt{M}} & p = 0, \quad 0 \leq q \leq M-1 \\ \sqrt{\frac{2}{M}}\cos\frac{\pi(2q+1)p}{2M} & 1 \leq p \leq M-1, \quad 0 \leq q \leq M-1 \end{cases} \tag{21}$$

For an M-by-M matrix A, T*A is an M-by-M matrix whose columns contain the one-dimensional DCT of the columns of A. The two-dimensional DCT of A can be computed as B=T*A*T'. Since T is a real orthonormal matrix, its inverse is the same as its transpose. Therefore, the inverse two-dimensional DCT of B is given by T'*B*T [16].

The sample MATLAB code is:

```
image = double(imread('lena.jpg'));
% image must be double rather than uint8
[H, W] = size(image);
if H == W
    dct_basis = dctmtx(512);
    image_dct = dct_basis * image * dct_basis';
    image_recover = dct_basis' * image_dct * dct_basis;
else
    image_dct = dct2(image);
    image_recover = idct2(image_dct);
end
imwrite(uint8(image_dct), 'lena_dct_test.jpg');
imwrite(uint8(rescale(image_recover,0,255)), 'lena_recover_test.jpg');
```

The Fourier DCT matrix can also be applied. The rows of the Fourier DCT matrix are basis sequences of the DCT. The DCT matrix of type 2 has an inverse of type 3. Each entry F_{rs} of an $n \times n$ DCT matrix of type 2 is computed as [23]:

$$\frac{1}{\sqrt{n}} \cos\left(\frac{\pi}{n}\left(r-\frac{1}{2}\right)(s-1)\right) \tag{22}$$

Each entry F_{rs} of the DCT matrix of type 3 is computed as:

$$\frac{1}{\sqrt{n}} \begin{cases} 1 & r=1 \\ 2\cos\left(\frac{\pi}{n}(r-1)\left(s-\frac{1}{2}\right)\right) & r=2,\ldots,n \end{cases} \tag{23}$$

The sample Wolfram Mathematica code is:

```
(* Content-type: application/vnd.wolfram.mathematica *)

(*** Wolfram Notebook File ***)
(* http://www.wolfram.com/nb *)

(* CreatedBy='Mathematica 13.0' *)

(*CacheID: 234*)
(* Internal cache information:
NotebookFileLineBreakTest
NotebookFileLineBreakTest
NotebookDataPosition[       158,          7]
NotebookDataLength[      2757,         80]
NotebookOptionsPosition[      2097,         62]
NotebookOutlinePosition[      2530,         79]
CellTagsIndexPosition[      2487,         76]
WindowFrame->Normal*)

(* Beginning of Notebook Content *)
Notebook[{
Cell[BoxData[
 RowBox[{"image", "=",
  RowBox[{
  "Import", "[",
   "\"\<F:/Documents/GitHub/ECE4810J_FA2022_TA/Lab3/lena.jpg\>\"",
   "]"}]}]], "Input",
 CellChangeTimes->{{3.8738612862074704`*^9, 3.8738612971875153`*^9}},
 CellLabel->"In[1]:=",ExpressionUUID->"5773606b-bd30-47ff-a9d6-89
   cb74e1edf1"],

Cell[BoxData[{
 RowBox[{
  RowBox[{"DCT", "[",
   RowBox[{"x_", "?", "MatrixQ"}], "]"}], " ", ":=", " ", "\n", " ",
  RowBox[{
   RowBox[{"Transpose", "[",
    RowBox[{"FourierDCTMatrix", "[", "512", "]"}], "]"}], " ", ".", " "
   , "x",
   " ", ".", " ",
   RowBox[{"FourierDCTMatrix", "[", "512", "]"}]}]}]}], "\n",
 RowBox[{
  RowBox[{"IDCT", "[",
   RowBox[{"x_", "?", "MatrixQ"}], "]"}], " ", ":=", " ", "\n", " ",
  RowBox[{
   RowBox[{"Transpose", "[",
    RowBox[{"FourierDCTMatrix", "[",
     RowBox[{"512", ",", "3"}], "]"}], "]"}], " ", ".", " ", "x", " ",
   ".",
   " ",
   RowBox[{"FourierDCTMatrix", "[",
    RowBox[{"512", ",", "3"}], "]"}]}]}]}]}], "Input",
 CellLabel->"In[2]:=",ExpressionUUID->"5155629a-f464-41e0-9081-8
   cedb345e4c8"],

Cell[BoxData[{
 RowBox[{"dctimage", " ", "=", " ",
  RowBox[{"DCT", "[",
   RowBox[{"ImageData", "[", "image", "]"}], "]"}]}], "\[
   IndentingNewLine]",
 RowBox[{"recoverimage", "=", " ",
  RowBox[{"IDCT", "[", "dctimage", "]"}]}]}], "Input",
 CellLabel->"In[4]:=",ExpressionUUID->"cb93d60d-a814-4e1d-bc03-
   e4716248839c"],

Cell[BoxData[
 RowBox[{"Image", "[", "recoverimage", "]"}]], "Input",
 CellLabel->"In[6]:=",ExpressionUUID->"f3f2720f-4dda-4c8e-8f38-6
   b0e75d89f47"]
},
WindowSize->{571.8, 524.4},
WindowMargins->{{278.4, Automatic}, {6.600000000000023, Automatic}},
FrontEndVersion->"13.0 for Microsoft Windows (64-bit) (2022\:5e742
   \:67084\
\:65e5)",
StyleDefinitions->"Default.nb",
ExpressionUUID->"89d0fd93-2c80-49a0-b556-d6105e235fba"
]
(* End of Notebook Content *)
(* Internal cache information *)
(*CellTagsOutline
CellTagsIndex->{}
*)
(*CellTagsIndex
CellTagsIndex->{}
*)
(*NotebookFileOutline
Notebook[{
Cell[558, 20, 304, 7, 62, "Input",ExpressionUUID->"5773606b-bd30-47ff-
    a9d6-89cb74e1edf1"],
Cell[865, 29, 754, 19, 149, "Input",ExpressionUUID->"5155629a-f464-41e0
    -9081-8cedb345e4c8"],
Cell[1622, 50, 321, 6, 64, "Input",ExpressionUUID->"cb93d60d-a814-4e1d-
    bc03-e4716248839c"],
Cell[1946, 58, 147, 2, 43, "Input",ExpressionUUID->"f3f2720f-4dda-4c8e
    -8f38-6b0e75d89f47"]
}
]
*)
```

The sample Python code is:

```python
import numpy as np

def DCT_basis(M):
    dct_basis = np.zeros((M, M))
    dct_basis[0,:] = 1 / np.sqrt(M)
    for p in range(1,M):
        for q in range(M):
            dct_basis[p, q] = np.sqrt(2 / M) * np.cos(np.pi * (2 * q + 1)
             ↪  * p / (2 * M))

    return dct_basis


def DCT_transform(img):
    H,W = img.shape
    if H != W:
        img_dct = np.zeros((H, W))
        for u in range(H):
            for v in range(W):
                img_dct[u, v] = 0
                for x in range(H):
                    for y in range in range(W):
                        if u == 0 and v == 0:
                            img_dct[u, v] += 1 / np.sqrt(H * W) * img[x,
                             ↪  y]
                        elif u == 0:
                            img_dct[u, v] += np.sqrt(2 / H * W) * img[x, y] * np.cos(((2 * y + 1) * v * np.pi) / (2 * W))
                        elif v == 0:
                            img_dct[u, v] += np.sqrt(2 / H * W) * img[x, y] * np.cos(((2 * x + 1) * u * np.pi) / (2 * H))
                        else:
                            img_dct[u, v] += 2 / np.sqrt(H * W) * img[x, y] * np.cos(((2 * x + 1) * u * np.pi) / (2 * H)) * np.cos(((2 * y + 1) * v * np.pi) / (2 * W))
    else:
        dct_basis = DCT_basis(W)
        img_dct = dct_basis @ img @ np.transpose(dct_basis)

    return img_dct


def iDCT_transform(img_dct):
    H,W = img_dct.shape
    if H != W:
        img_recover = np.zeros((H, W))
        for x in range(H):
            for y in range(W):
                img_recover[x, y] = 0
                for u in range(H):
                    for v in range in range(W):
                        if u == 0 and v == 0:
                            img_recover[x, y] += 1 / np.sqrt(H * W) * img_dct[u, v]
                        elif u == 0:
                            img_recover[x, y] += np.sqrt(2 / H * W) * img_dct[u, v] * np.cos(((2 * y + 1) * v * np.pi) / (2 * W))
                        elif v == 0:
                            img_recover[x, y] += np.sqrt(2 / H * W) * img_dct[u, v] * np.cos(((2 * x + 1) * u * np.pi) / (2 * H))
                        else:
                            img_recover[x, y] += 2 / np.sqrt(H * W) * img_dct[u, v] * np.cos(((2 * x + 1) * u * np.pi) / (2 * H)) * np.cos(((2 * y + 1) * v * np.pi) / (2 * W))
    else:
        dct_basis = DCT_basis(W)
        img_recover = np.transpose(dct_basis) @ img_dct @ dct_basis

    return img_recover
```

One of the properties of the 2-D DCT is that it is separable, meaning that it can be separated into a pair of 1-D DCTs. To obtain the 2-D DCT of a block, a 1-D DCT is first performed on the rows of the block then a 1-D DCT is performed on the columns of the resulting block [5].

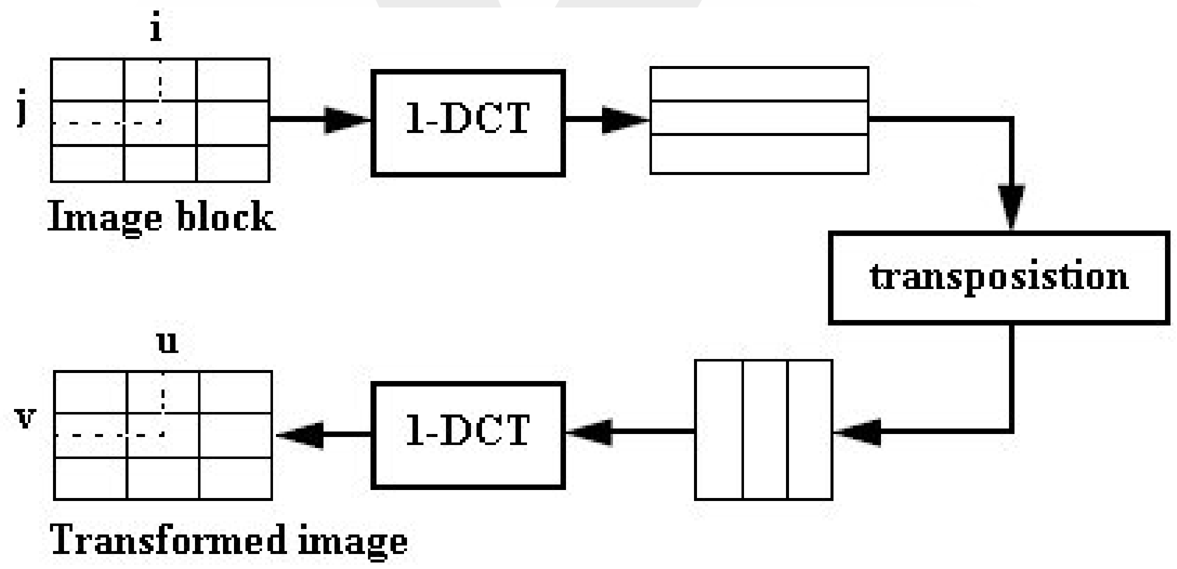


Figure 10: Computation of 2-D DCT using separability property [12].

The same applies to the IDCT. A 2-D DCT can be separated by applying 1-D DCT (vertically) to columns and then applying 1-D DCT (horizontally) to resultant vertical DCT, or equivalently, applying 1-D DCT (horizontally) to rows and then applying 1-D DCT (vertically) to resultant horizontal DCT. A simple MATLAB script can be used for verification:

```
image = imread('lab3_starter/lena_gray.bmp');
block = double(image(1:8,1:8));
dct_res = zeros(8,8);
dctint8 = round(dctmtx(8) * 2^14.5);
for i = 1:8
    % dct_res(i,:) = dct(block(i,:));
    dct_res(i,:) = dctint8 * block(i,:)';
    dct_res(i,:) = bitshift(int32(dct_res(i,:)) + bitshift(1, 12), -13);
end
for i = 1:8
    % dct_res(:,i) = dct(dct_res(:,i));
    dct_res(:,i) = dctint8 * dct_res(:,i);
    dct_res(:,i) = bitshift(int32(dct_res(:,i)) + bitshift(1, 12), -13);
end
```

```
dct_expected = dct2(block);
idct_res = zeros(8,8);
for i = 1:8
    % idct_res(i,:) = idct(dct_expected(i,:));
    idct_res(i,:) = dctint8' * dct_res(i,:)';
    idct_res(i,:) = bitshift(int32(idct_res(i,:)), -16);
end
for i = 1:8
    % idct_res(:,i) = idct(idct_res(:,i));
    idct_res(:,i) = dctint8' * idct_res(:,i);
    idct_res(:,i) = bitshift(int32(idct_res(:,i)), -16);
end
% isequal(uint8(idct_res), uint8(block))
```

3.2 Validate your design from Vitis HLS command line

1. Prepare your `input.dat` from the top-left corner 8×8 block of `lab3_starter/lena_gray.bmp`. You can also use `lab3_starter/lena_gray.raw`, which is the binary of all the points of the bitmap (BMP) file.

2. Denote the DCT basis matrix as T. Prepare your `dct_coeff_table.txt` as a matrix C by

$$C_{i,j} = \mathrm{round}(2^{\mathrm{CONST\_BITS}+1.5} T_{i,j}) \tag{24}$$

3. Run `make` by `lab3_starter/dct/Makefile`.

3.3 Create a new project

1. Create a new Vitis HLS project named **dct** with the top function `dct` and the xc7z020clg400-1 part.

2. Add `lab3_starter/dct.c` as the Source and add Add `lab3_starter/dct_test.c` as the Test Bench.

3. The top-level function `dct`, is defined at line 78. It implements a 2D DCT algorithm by first processing each row of the input array via a 1D DCT then processing the columns of the resulting array through the same 1D DCT. It calls `read_data`, `dct_2d`, and `write_data` functions.
   The `read_data` function is defined at line 54 and consists of two loops – RD_Loop_Row and RD_Loop_Col. The `write_data` function is defined at line 66 and consists of two loops to perform writing the result. The `dct_2d` function, defined at line 23, calls `dct_1d` function and performs transpose.
   Finally, `dct_1d` function, defined at line 4, uses `dct_coeff_table` and performs the required function by implementing a basic iterative form of the 1D Type-II DCT algorithm. The following figure shows the function hierarchy on the left-hand side, the loops in the order they are executed, and the flow of data on the right-hand side.

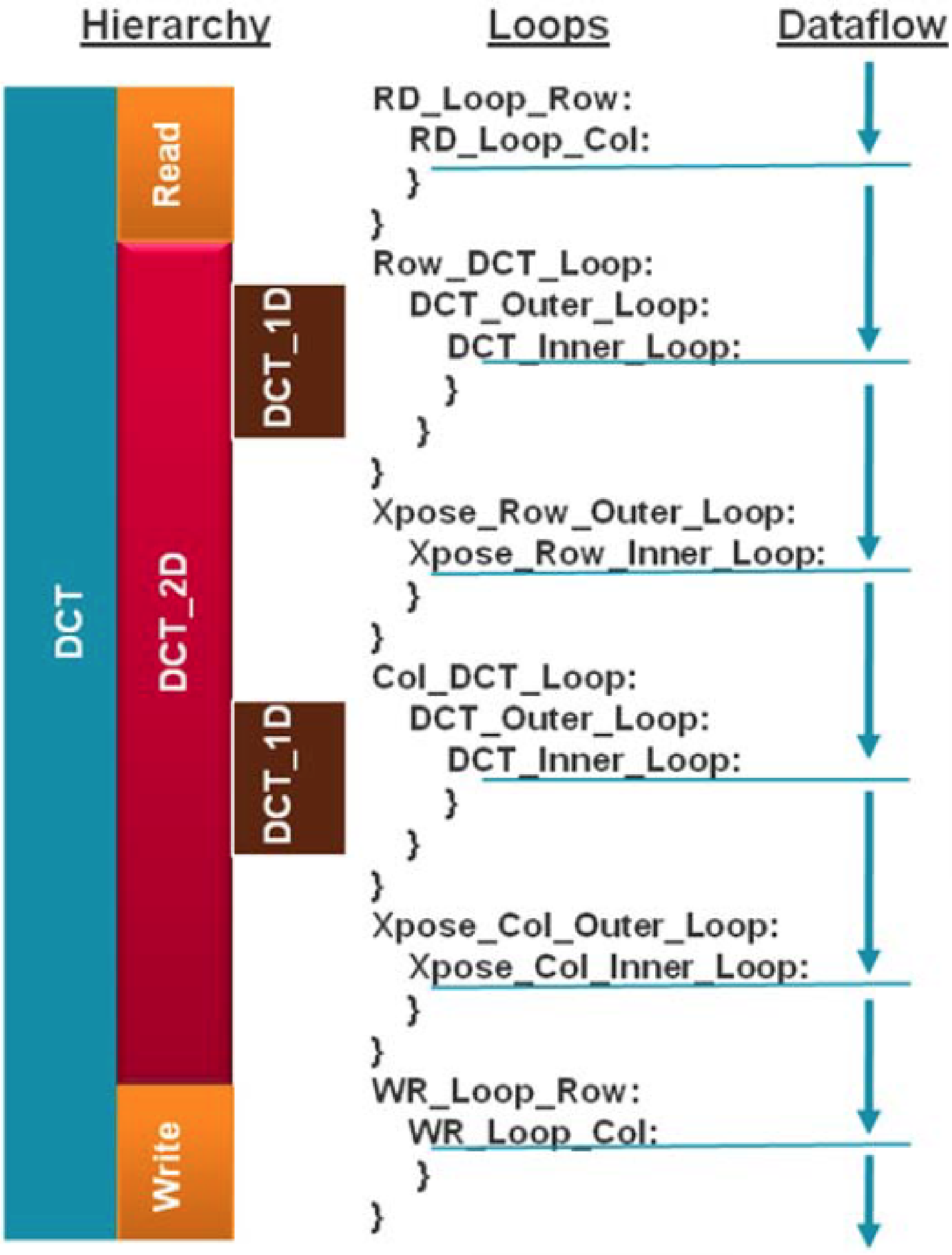
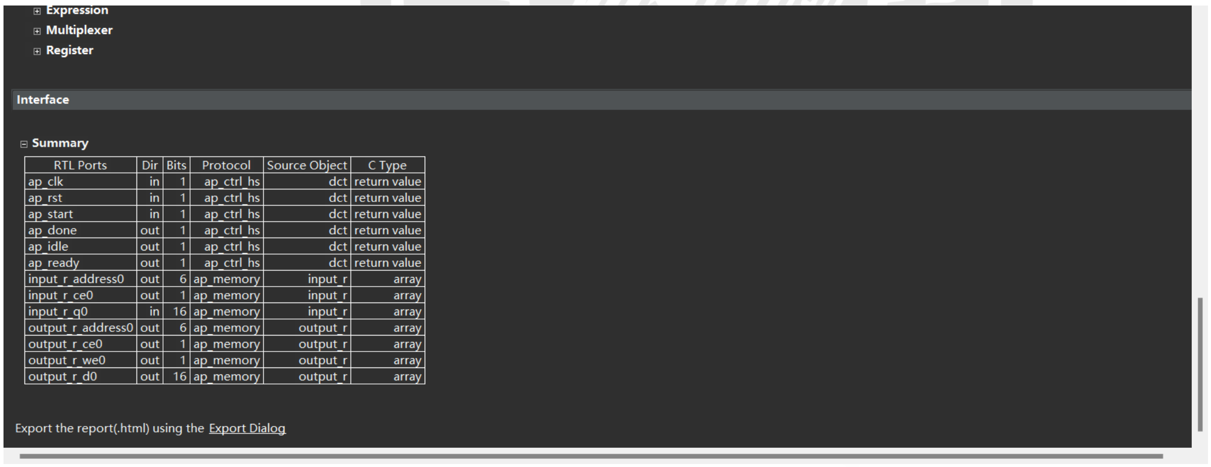


Figure 11: Design hierarchy and dataflow.

3.4 Apply `PIPELINE` Directive

Synthesize the design with the defaults. View the synthesis results and answer the question listed in the detailed section of this step. Run the Co-simulation, selecting Verilog. Verify that the simulation passes. Apply the PIPELINE directive to DCT_Inner_Loop, Xpose_Row_Inner_Loop, Xpose_Col_Inner_Loop, RD_Loop_Col, and WR_Loop_Col. Generate the solution and analyze the output. Create a new solution by copying the previous solution settings.

1. Explicitly add the HLS PIPELINE directive for dct_1d->DCT_Outer_Loop->DCT_Inner_Loop, because Vitis HLS will automatically pipeline loop RD_Loop_Col, WR_Loop_Col, Xpose_Row_Inner_Loop, Xpose_Col_Inner_Loop, and DCT_Outer_Loop, but we want to pipeline DCT_Inner_Loop rather than DCT_Outer_Loop.
2. Run C Synthesis.
3. When synthesis is completed, several report files will become accessible, and the Synthesis Results will be displayed in the information pane.
4. The Synthesis Report shows the performance and resource estimates as well as estimated latency in the design. Report the following metrics of the `dct` function:
   (a) Estimated clock period
   (b) Worst case latency
   (c) Number of DSP48E used
   (d) Number of BRAMs used
   (e) Number of FFs used
   (f) Number of LUTs used
5. The report also shows the top-level interface signals generated by the tools.



⊞ Expression
⊞ Multiplexer
⊞ Register

Interface

⊟ **Summary**

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	dct	return value
ap_rst	in	1	ap_ctrl_hs	dct	return value
ap_start	in	1	ap_ctrl_hs	dct	return value
ap_done	out	1	ap_ctrl_hs	dct	return value
ap_idle	out	1	ap_ctrl_hs	dct	return value
ap_ready	out	1	ap_ctrl_hs	dct	return value
input_r_address0	out	6	ap_memory	input_r	array
input_r_ce0	out	1	ap_memory	input_r	array
input_r_q0	in	16	ap_memory	input_r	array
output_r_address0	out	6	ap_memory	output_r	array
output_r_ce0	out	1	ap_memory	output_r	array
output_r_we0	out	1	ap_memory	output_r	array
output_r_d0	out	16	ap_memory	output_r	array

Export the report(.html) using the Export Dialog

Figure 12: Generated interface signals.

You can see ap_clk, ap_rst are automatically added. The ap_start, ap_done, ap_idle, and ap_ready are top-level signals used as handshaking signals to indicate when the design is able to accept next computation command (ap_idle), when the next computation is started (ap_start), and when the computation is completed (ap_done). The top-level function has input and output arrays; hence an ap_memory interface is generated for each of them.

6. Run C Cosimulation. You can now inspect your generated `output.dat`.

7. In the Module Hierarchy pane, select the dct entry and observe the RD_Loop_Row_RD_Loop_Col and WR_Loop_Row_WR_Loop_Col entries in the Schedule Viewer. These are two nested loops flattened and given the new names formed by appending the inner loop name to the outer loop name. You can also verify this by looking in the Console view message. If it is

```
INFO: [XFORM 203-541] Flattening a loop nest 'RD_Loop_Row'
↪  (../lab3_starter/dct/dct.c:56:8) in function 'dct'.
WARNING: [HLS 200-960] Cannot flatten loop 'DCT_Outer_Loop'
↪  (../lab3_starter/dct/dct.c:6:17) in function 'dct' the outer
↪  loop is not a perfect loop because there is nontrivial logic in
↪  the loop latch.
Resolution: For help on HLS 200-960 see www.xilinx.com/cgi-bin/docs
↪  /rdoc?v=2022.1;t=hls+guidance;d=200-960.html
INFO: [XFORM 203-541] Flattening a loop nest 'Row_DCT_Loop'
↪  (../lab3_starter/dct/dct.c:28:13) in function 'dct'.
INFO: [XFORM 203-541] Flattening a loop nest 'Xpose_Row_Outer_Loop'
↪  (../lab3_starter/dct/dct.c:28:16) in function 'dct'.
WARNING: [HLS 200-960] Cannot flatten loop 'DCT_Outer_Loop'
↪  (../lab3_starter/dct/dct.c:6:17) in function 'dct' the outer
↪  loop is not a perfect loop because there is nontrivial logic in
↪  the loop latch.
Resolution: For help on HLS 200-960 see www.xilinx.com/cgi-bin/docs
↪  /rdoc?v=2022.1;t=hls+guidance;d=200-960.html
INFO: [XFORM 203-541] Flattening a loop nest 'Col_DCT_Loop'
↪  (../lab3_starter/dct/dct.c:28:13) in function 'dct'.
INFO: [XFORM 203-541] Flattening a loop nest 'Xpose_Col_Outer_Loop'
↪  (../lab3_starter/dct/dct.c:28:16) in function 'dct'.
INFO: [XFORM 203-541] Flattening a loop nest 'WR_Loop_Row'
↪  (../lab3_starter/dct/dct.c:68:8) in function 'dct'.
```

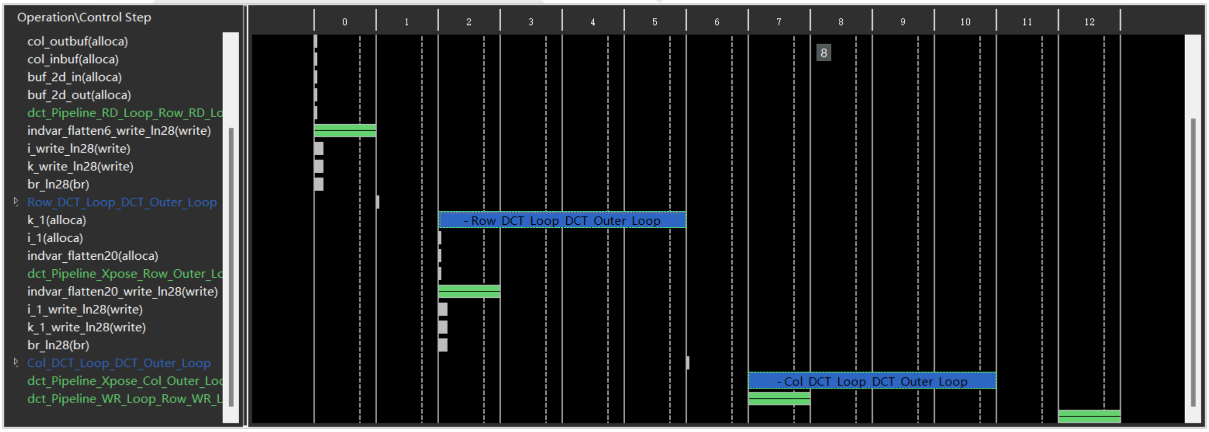
then you are correct. If it is

```
INFO: [XFORM 203-541] Flattening a loop nest 'RD_Loop_Row'
↪  (../lab3_starter/dct/dct.c:56:8) in function 'dct'.
INFO: [XFORM 203-541] Flattening a loop nest 'Row_DCT_Loop'
↪  (../lab3_starter/dct/dct.c:28:13) in function 'dct'.
INFO: [XFORM 203-541] Flattening a loop nest 'Xpose_Row_Outer_Loop'
↪  (../lab3_starter/dct/dct.c:28:16) in function 'dct'.
```

```
INFO: [XFORM 203-541] Flattening a loop nest 'Col_DCT_Loop' (../lab3_starter/dct/dct.c:28:13) in function 'dct'.
INFO: [XFORM 203-541] Flattening a loop nest 'Xpose_Col_Outer_Loop' (../lab3_starter/dct/dct.c:28:16) in function 'dct'.
INFO: [XFORM 203-541] Flattening a loop nest 'WR_Loop_Row' (../lab3_starter/dct/dct.c:68:8) in function 'dct'.
```

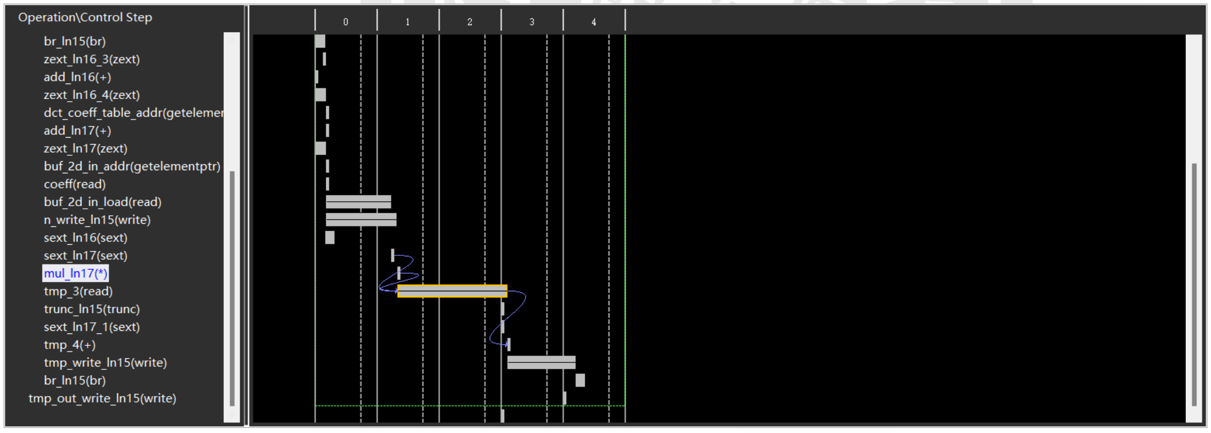
then you are incorrect.

8. Click the "Open Viewer" button in the top bar. It will open the Schedule Viewer by default.

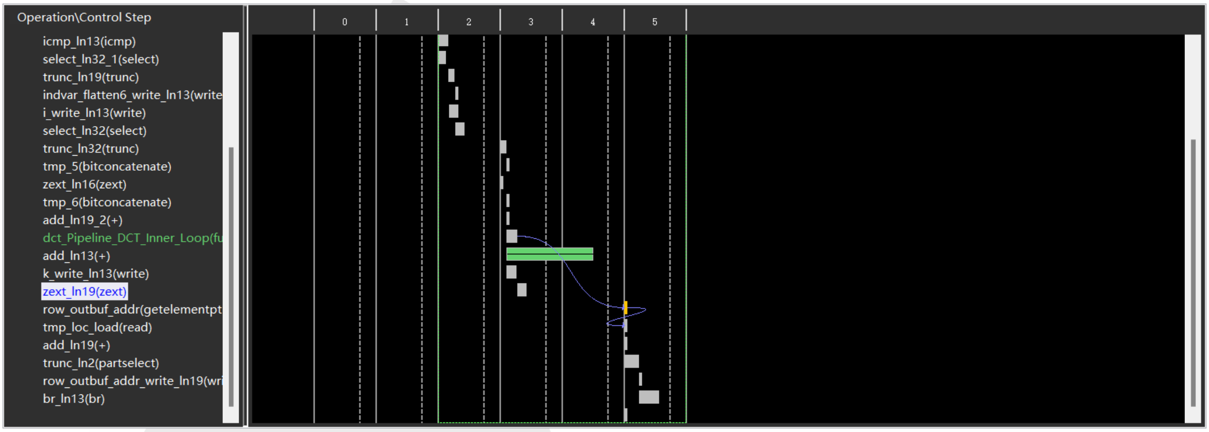


Operation\Control Step
col_outbuf(alloca)
col_inbuf(alloca)
buf_2d_in(alloca)
buf_2d_out(alloca)
dct_Pipeline_RD_Loop_Row_RD_Lo
indvar_flatten6_write_ln28(write)
i_write_ln28(write)
k_write_ln28(write)
br_ln28(br)
Row_DCT_Loop_DCT_Outer_Loop
k_1(alloca)
i_1(alloca)
indvar_flatten20(alloca)
dct_Pipeline_Xpose_Row_Outer_Lo
indvar_flatten20_write_ln28(write)
i_1_write_ln28(write)
k_1_write_ln28(write)
br_ln28(br)
Col_DCT_Loop_DCT_Outer_Loop
dct_Pipeline_Xpose_Col_Outer_Loo
dct_Pipeline_WR_Loop_Row_WR_L
0 1 2 3 4 5 6 7 8 9 10 11 12
8
- Row_DCT_Loop_DCT_Outer_Loop
- Col_DCT_Loop_DCT_Outer_Loop

You can see that most cycle time is cost by Row_DCT_Loop_DCT_Outer_Loop and Col_DCT_Loop_DCT_Outer_Loop, which are mainly cost by dct_Pipeline_DCT_Inner_Loop and dct_Pipeline_DCT_Inner_Loop1 respectively. You can furthermore expand the DCT_Inner_Loop of them and observe that most cycle time is cost by coeff(read), buf_2d_in_load(read), mul_ln17(*), and tmp_4(+). You can right-click on the Operation/Control Step and click "Goto Source", then a "C Source" tab will show in the bottom perspective. Use up and down key to browse the operations/control steps with their source.



Line 19 is not operated by dct_Pipeline_DCT_Inner_Loop or dct_Pipeline_DCT_Inner_Loop1. Line 19 is scheduled after they finish.


Operation\Control Step
icmp_ln13(icmp)
select_ln32_1(select)
trunc_ln19(trunc)
indvar_flatten6_write_ln13(write
i_write_ln13(write)
select_ln32(select)
trunc_ln32(trunc)
tmp_5(bitconcatenate)
zext_ln16(zext)
tmp_6(bitconcatenate)
add_ln19_2(+)
dct_Pipeline_DCT_Inner_Loop(fu
add_ln13(+)
k_write_ln13(write)
zext_ln19(zext)
row_outbuf_addr(getelementpt
tmp_loc_load(read)
add_ln19(+)
trunc_ln2(partselect)
row_outbuf_addr_write_ln19(wr
br_ln13(br)

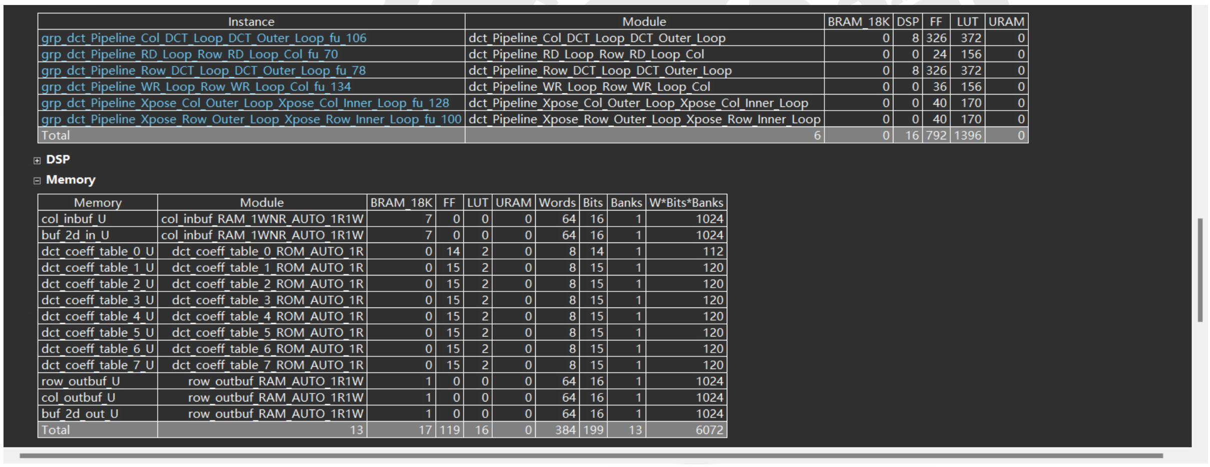
9. Create a new solution solution2 from solution1. Remove the HLS PIPELINE directive of DCT_Inner_Loop to allow auto-pipelining of DCT_Outer_Loop.

10. Run C Synthesis for solution2.

11. Generate the Vitis HLS Report Comparison between solution1 and solution2. Take a screenshot of the Performance Estimates and the Utilization Estimates.

12. Note that different from solution1, in the Utilization Estimates section of the Synthesis Details(solution2)(dct_csynth.rpt), expand Detail->Memory, the coefficient table was automatically partitioned, resulting in 8 separate ROMs: this helped reduce the latency by keeping the unrolled computation loop fed; however, the input arrays to the `dct_1d` function were not automatically partitioned, and the `dct_1d` function utilize more resources. In the Console view, you can also see one line:

```
INFO: [XFORM 203-102] Partitioning array 'dct_coeff_table' in
    ↪ dimension 2 automatically.
```



Instance	Module	BRAM_18K	DSP	FF	LUT	URAM
grp_dct_Pipeline_Col_DCT_Loop_DCT_Outer_Loop_fu_106	dct_Pipeline_Col_DCT_Loop_DCT_Outer_Loop	0	8	326	372	0
grp_dct_Pipeline_RD_Loop_Row_RD_Loop_Col_fu_70	dct_Pipeline_RD_Loop_Row_RD_Loop_Col	0	0	24	156	0
grp_dct_Pipeline_Row_DCT_Loop_DCT_Outer_Loop_fu_78	dct_Pipeline_Row_DCT_Loop_DCT_Outer_Loop	0	8	326	372	0
grp_dct_Pipeline_WR_Loop_Row_WR_Loop_Col_fu_134	dct_Pipeline_WR_Loop_Row_WR_Loop_Col	0	0	36	156	0
grp_dct_Pipeline_Xpose_Col_Outer_Loop_Xpose_Col_Inner_Loop_fu_128	dct_Pipeline_Xpose_Col_Outer_Loop_Xpose_Col_Inner_Loop	0	0	40	170	0
grp_dct_Pipeline_Xpose_Row_Outer_Loop_Xpose_Row_Inner_Loop_fu_100	dct_Pipeline_Xpose_Row_Outer_Loop_Xpose_Row_Inner_Loop	0	0	40	170	0
Total	6	0	16	792	1396	0

⊞ **DSP**

⊟ **Memory**

Memory	Module	BRAM_18K	FF	LUT	URAM	Words	Bits	Banks	W*Bits*Banks
col_inbuf_U	col_inbuf_RAM_1WNR_AUTO_1R1W	7	0	0	0	64	16	1	1024
buf_2d_in_U	col_inbuf_RAM_1WNR_AUTO_1R1W	7	0	0	0	64	16	1	1024
dct_coeff_table_0_U	dct_coeff_table_0_ROM_AUTO_1R	0	14	2	0	8	14	1	112
dct_coeff_table_1_U	dct_coeff_table_1_ROM_AUTO_1R	0	15	2	0	8	15	1	120
dct_coeff_table_2_U	dct_coeff_table_2_ROM_AUTO_1R	0	15	2	0	8	15	1	120
dct_coeff_table_3_U	dct_coeff_table_3_ROM_AUTO_1R	0	15	2	0	8	15	1	120
dct_coeff_table_4_U	dct_coeff_table_4_ROM_AUTO_1R	0	15	2	0	8	15	1	120
dct_coeff_table_5_U	dct_coeff_table_5_ROM_AUTO_1R	0	15	2	0	8	15	1	120
dct_coeff_table_6_U	dct_coeff_table_6_ROM_AUTO_1R	0	15	2	0	8	15	1	120
dct_coeff_table_7_U	dct_coeff_table_7_ROM_AUTO_1R	0	15	2	0	8	15	1	120
row_outbuf_U	row_outbuf_RAM_AUTO_1R1W	1	0	0	0	64	16	1	1024
col_outbuf_U	row_outbuf_RAM_AUTO_1R1W	1	0	0	0	64	16	1	1024
buf_2d_out_U	row_outbuf_RAM_AUTO_1R1W	1	0	0	0	64	16	1	1024
Total	13	17	119	16	0	384	199	13	6072

Figure 13: Automatic partitioning of `dct_coeff_table`.

In the Schedule Viewer, we can see the partitioned `dct_coeff_table` pipelining read and getelement.

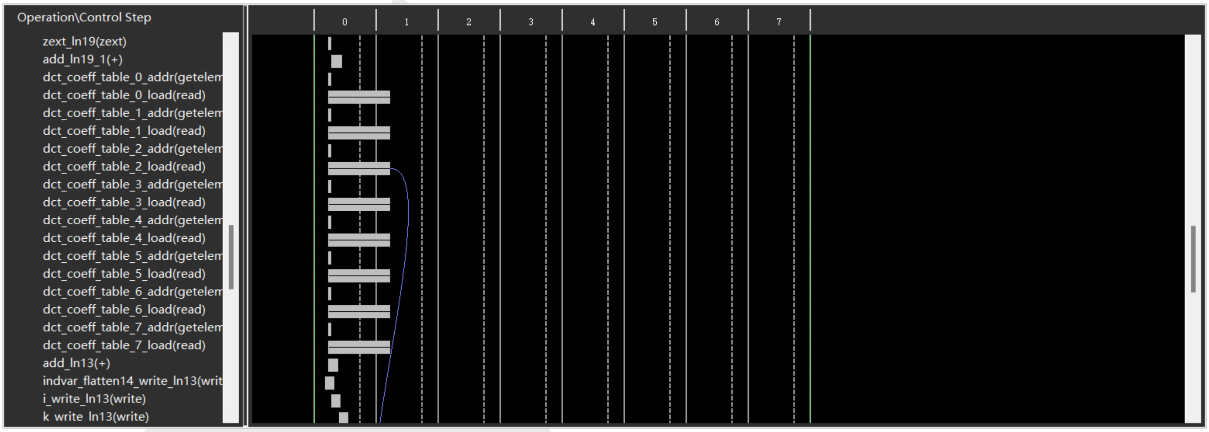
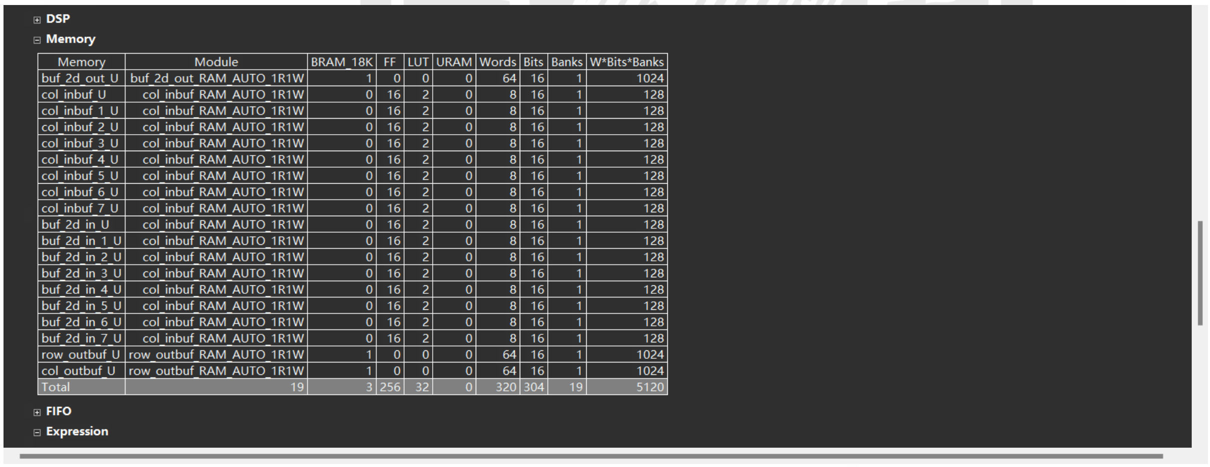

Operation\Control Step
zext_ln19(zext)
add_ln19_1(+)
dct_coeff_table_0_addr(getelem
dct_coeff_table_0_load(read)
dct_coeff_table_1_addr(getelem
dct_coeff_table_1_load(read)
dct_coeff_table_2_addr(getelem
dct_coeff_table_2_load(read)
dct_coeff_table_3_addr(getelem
dct_coeff_table_3_load(read)
dct_coeff_table_4_addr(getelem
dct_coeff_table_4_load(read)
dct_coeff_table_5_addr(getelem
dct_coeff_table_5_load(read)
dct_coeff_table_6_addr(getelem
dct_coeff_table_6_load(read)
dct_coeff_table_7_addr(getelem
dct_coeff_table_7_load(read)
add_ln13(+)
indvar_flatten14_write_ln13(writ
i_write_ln13(write)
k_write_ln13(write)
0
1
2
3
4
5
6
7

Figure 14: The Performance view of the (Row_DCT_Loop_)DCT_Outer_Loop function.

3.5 Improve memory bandwidth

Create a new solution by copying the previous solution (solution2) settings. Apply the `ARRAY_PARTITION` directive to `buf_2d_in` of `dct` (since the bottleneck was on src port of the `dct_1d` function, which was passed via `in_block` of the `dct_2d` function, which in turn was passed via `buf_2d_in` of the `dct` function) and `col_inbuf` of `dct_2d`. Generate the solution.

1. Create a new solution solution3 from solution2.
2. Add the `ARRAY_PARTITION` directive to `buf_2d_in` of `dct`. Specify dim = 2, type is complete. Similarly, apply the `ARRAY_PARTITION` directive with dimension of 2 to the `col_inbuf` array.
3. Run C Synthesis.
4. When the synthesis is completed, select Project > Compare Reports... to compare the two solutions. Observe that the latency is reduced. Scroll down in the comparison report to view the resource utilization. Observe the increase in FF resource utilization.



⊞ DSP

⊟ Memory

Memory	Module	BRAM_18K	FF	LUT	URAM	Words	Bits	Banks	W*Bits*Banks
buf_2d_out_U	buf_2d_out_RAM_AUTO_1R1W	1	0	0	0	64	16	1	1024
col_inbuf_U	col_inbuf_RAM_AUTO_1R1W	0	16	2	0	8	16	1	128
col_inbuf_1_U	col_inbuf_RAM_AUTO_1R1W	0	16	2	0	8	16	1	128
col_inbuf_2_U	col_inbuf_RAM_AUTO_1R1W	0	16	2	0	8	16	1	128
col_inbuf_3_U	col_inbuf_RAM_AUTO_1R1W	0	16	2	0	8	16	1	128
col_inbuf_4_U	col_inbuf_RAM_AUTO_1R1W	0	16	2	0	8	16	1	128
col_inbuf_5_U	col_inbuf_RAM_AUTO_1R1W	0	16	2	0	8	16	1	128
col_inbuf_6_U	col_inbuf_RAM_AUTO_1R1W	0	16	2	0	8	16	1	128
col_inbuf_7_U	col_inbuf_RAM_AUTO_1R1W	0	16	2	0	8	16	1	128
buf_2d_in_U	col_inbuf_RAM_AUTO_1R1W	0	16	2	0	8	16	1	128
buf_2d_in_1_U	col_inbuf_RAM_AUTO_1R1W	0	16	2	0	8	16	1	128
buf_2d_in_2_U	col_inbuf_RAM_AUTO_1R1W	0	16	2	0	8	16	1	128
buf_2d_in_3_U	col_inbuf_RAM_AUTO_1R1W	0	16	2	0	8	16	1	128
buf_2d_in_4_U	col_inbuf_RAM_AUTO_1R1W	0	16	2	0	8	16	1	128
buf_2d_in_5_U	col_inbuf_RAM_AUTO_1R1W	0	16	2	0	8	16	1	128
buf_2d_in_6_U	col_inbuf_RAM_AUTO_1R1W	0	16	2	0	8	16	1	128
buf_2d_in_7_U	col_inbuf_RAM_AUTO_1R1W	0	16	2	0	8	16	1	128
row_outbuf_U	row_outbuf_RAM_AUTO_1R1W	1	0	0	0	64	16	1	1024
col_outbuf_U	row_outbuf_RAM_AUTO_1R1W	1	0	0	0	64	16	1	1024
Total	19	3	256	32	0	320	304	19	5120

⊞ FIFO

⊟ Expression

Figure 15: Resource profile after partitioning buffers.

5. In the Utilization Estimates section of the Synthesis Details(solution3)(dct_csynth.rpt), expand Detail->Memory, and observe that most of the resources are consumed by instances. The `buf_2d_in` array is partitioned into multiple memories, and most of the operations are done in addition and comparison. Take a screenshot of the utilization estimates detail of memory.

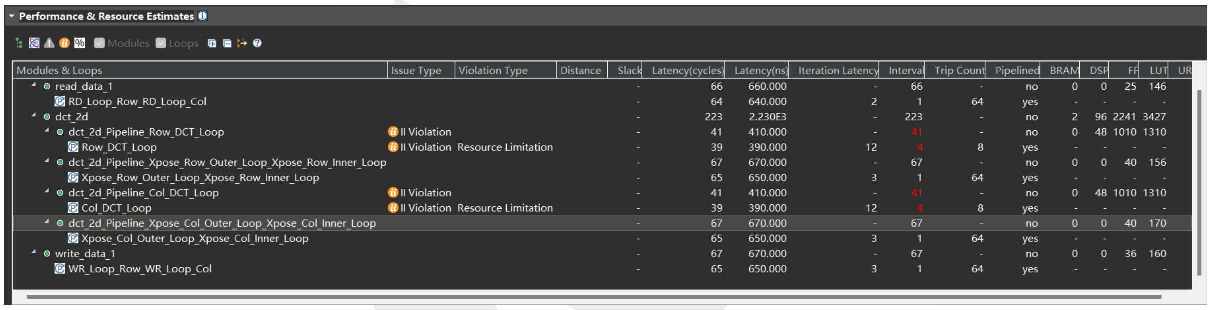
3.6 Apply `DATAFLOW` Directive

Create a new solution solution4 by copying the previous solution (solution3) settings. Apply the `DATAFLOW` directive to the `dct` function to improve the throughput. Run C Synthesis. Generate the solution and analyze the output.

In the Synthesis Details, the Dataflow pipeline throughput indicates the number of clock cycles between each set of input reads (interval parameter). If this value is less than the design latency, it indicates the design can start processing new inputs before the current input data are output. Note that the dataflow is only supported for the functions and loops at the top level, not those which are down through the design hierarchy. Only loops and functions exposed at the top level of the design will get benefit from dataflow optimization.

Compare solution3 and solution4. Observe that the latency is reduced. Scroll down into the Area Estimates, and observe that the number of BRAM_18K required at the top level remained at 3. Observe that most of the latency and interval (throughput) is caused by the `dct_2d` function. The interval of the top-level function `dct`, is less than the sum of the intervals of the `read_data`, `dct_2d`, and `write_data` functions indicating that they operate in parallel and `dct_2d` is the limiting factor.

```
INFO: [XFORM 203-712] Applying dataflow to function 'dct'
↪  (../lab3_starter/dct/dct.c:9:1), detected/extracted 3 process
↪  function(s):
         'read_data.1'
         'dct_2d'
         'write_data.1'.
INFO: [XFORM 203-11] Balancing expressions in function 'dct_2d'
↪  (../lab3_starter/dct/dct.c:17:7)...124 expression(s) balanced.
INFO: [HLS 200-111] Finished Loop, function and other optimizations: CPU
↪  user time: 0 seconds. CPU system time: 0 seconds. Elapsed time: 0.383
↪  seconds; current allocated memory: 1.421 GB.
INFO: [XFORM 203-541] Flattening a loop nest 'WR_Loop_Row'
↪  (../lab3_starter/dct/dct.c:68:8) in function 'write_data.1'.
INFO: [XFORM 203-541] Flattening a loop nest 'RD_Loop_Row'
↪  (../lab3_starter/dct/dct.c:56:8) in function 'read_data.1'.
INFO: [XFORM 203-541] Flattening a loop nest 'Xpose_Row_Outer_Loop'
↪  (../lab3_starter/dct/dct.c:28:16) in function 'dct_2d'.
INFO: [XFORM 203-541] Flattening a loop nest 'Xpose_Col_Outer_Loop'
↪  (../lab3_starter/dct/dct.c:28:16) in function 'dct_2d'.
```



Performance & Resource Estimates

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	UR
read_data_1				-	66	660.000	-	66	-	no	0	0	25	146	
RD_Loop_Row_RD_Loop_Col				-	64	640.000	2	1	64	yes	-	-	-	-	
dct_2d				-	223	2.230E3	-	223	-	no	2	96	2241	3427	
dct_2d_Pipeline_Row_DCT_Loop	II Violation			-	41	410.000	-	41	-	no	0	48	1010	1310	
Row_DCT_Loop	II Violation	Resource Limitation		-	39	390.000	12	4	8	yes	-	-	-	-	
dct_2d_Pipeline_Xpose_Row_Outer_Loop_Xpose_Row_Inner_Loop				-	67	670.000	-	67	-	no	0	0	40	156	
Xpose_Row_Outer_Loop_Xpose_Row_Inner_Loop				-	65	650.000	3	1	64	yes	-	-	-	-	
dct_2d_Pipeline_Col_DCT_Loop	II Violation			-	41	410.000	-	41	-	no	0	48	1010	1310	
Col_DCT_Loop	II Violation	Resource Limitation		-	39	390.000	12	4	8	yes	-	-	-	-	
dct_2d_Pipeline_Xpose_Col_Outer_Loop_Xpose_Col_Inner_Loop				-	67	670.000	-	67	-	no	0	0	40	170	
Xpose_Col_Outer_Loop_Xpose_Col_Inner_Loop				-	65	650.000	3	1	64	yes	-	-	-	-	
write_data_1				-	67	670.000	-	67	-	no	0	0	36	160	
WR_Loop_Row_WR_Loop_Col				-	65	650.000	3	1	64	yes	-	-	-	-	

Figure 16: Performance analysis after the `DATAFLOW` directive.

One of the limitations of dataflow optimization is that it only works on top-level loops and functions. One way to have the blocks in `dct_2d` operate in parallel would be to pipeline the entire function. This, however, would unroll all the loops and can sometimes lead to a large area increase. An alternative is to raise these loops up to the top level of the hierarchy, where dataflow optimization can be applied, by removing the `dct_2d` hierarchy, i.e. inline the `dct_2d` function.

3.7 Apply `INLINE` Directive

Create a new solution solution5 by copying the previous solution (solution4) settings. Apply `INLINE` directive to `dct_2d`. Run C Synthesis. Generate the solution and analyze the output.

In the Console view, observe that

```
INFO: [XFORM 203-712] Applying dataflow to function 'dct'
↪  (../lab3_starter/dct/dct.c:9:1), detected/extracted 6 process
↪  function(s):
        'read_data.1'
        'Loop_Row_DCT_Loop_proc'
        'Loop_Xpose_Row_Outer_Loop_proc'
        'Loop_Col_DCT_Loop_proc'
        'Loop_Xpose_Col_Outer_Loop_proc'
        'write_data.1'.
```

Compare solution4 and solution5. Observe that the latency is reduced. Switch to the Analysis perspective, expand the Module Hierarchy entries, and select the `dct` entry. Observe that the `dct_2d` entry is now replaced with Loop_Row_DCT_Loop_proc, Loop_Xpose_Row_Outer_Loop_proc, Loop_Col_DCT_Loop_proc, and Loop_Xpose_Col_Outer_Loop_proc since the `dct_2d` function is inlined. BRAM usage has increased once again (from 3 to 5), due to ping-pong buffering between more dataflow processes.

Compare all five solutions and take a screenshot of the Performance Estimates and the Utilization Estimates.

3.8 Conclusion

In this lab, you learned various techniques to improve performance and balance resource utilization. The PIPELINE directive that is applied to the outer loop will automatically cause the inner loop to unroll. When a loop is unrolled, resource utilization increases as operations are done concurrently. Partitioning memory may improve performance but will increase BRAM utilization. When the INLINE directive is applied to a function, the lower-level hierarchy is automatically dissolved. When the DATAFLOW directive is applied, the default memory buffers (of ping-pong type) are automatically inserted between the top-level functions and loops. The Analysis perspective and console logs can provide insight into what is going on.

3.9 Inverse discrete cosine transform (IDCT)

1. Write three new functions based on Section 3.1:
   - `void idct_1d(dct_data_t src[DCT_SIZE], dct_data_t dst[DCT_SIZE])`
   - `void idct_2d(dct_data_t in_block[DCT_SIZE][DCT_SIZE], dct_data_t out_block[DCT_SIZE][DCT_SIZE])`
   - `void idct(short input[N], short output[N])`
2. In `dct_test.c`, uncomment Line 20 `idct(b,c)` and print the array `short c[N]` to `output.dat`.
3. Run `make`.
4. (Optional) Re-generate `input.dat` of the full `lena_gray.bmp` block-by-block. Process DCT and IDCT on each 8×8 block. Compose the transformed block into a recovered image.

4 (Optional) JPEG and DCT Further On

4.1 Introduction to JPEG

JPEG (Joint Photographic Experts Group) is a commonly used method of lossy compression for digital images, particularly for those images produced by digital photography. JPEG compression is used in a number of image file formats. JPEG/Exif is the most common image format used by digital cameras and other photographic image capture devices; along with JPEG/JFIF, it is the most common format for storing and transmitting photographic images on the World Wide Web. These format variations are often not distinguished and are simply called JPEG [21].

A JPEG file can be encoded in various ways. A simplified example encoding process consists of several steps:

- Color space transformation and downsampling: The representation of the colors in the image is converted from RGB to Y'CBCR, consisting of one luma component (Y'), representing brightness, and two chroma components, (CB and CR), representing color. This is what we have done in Section 2.

- Block splitting and discrete cosine transform: The image is split into blocks of 8×8 pixels, and for each block, each of the Y, CB, and CR data undergoes the DCT that is similar to a Fourier transform in the sense that it produces a kind of spatial frequency spectrum. This is what we have done in Section 3. The top-left corner entry of the transformed matrix (G) has a rather large magnitude. This is the DC coefficient (also called the constant component), which defines the basic hue for the entire block. The remaining 63 coefficients are AC coefficients (also called alternating components).

- Quantization and entropy coding: The amplitudes of the frequency components are quantized. Human vision is much more sensitive to small variations in color or brightness over large areas than to the strength of high-frequency brightness variations. Therefore, the magnitudes of the high-frequency components are stored with lower accuracy than the low-frequency components. There are different versions of quantization matrices Q, for example, Adobe Photoshop uses its own quantization matrix. The quality setting of the encoder (for example, 50 or 95 on a scale of 0–100 in the Independent JPEG Group's library) affects to what extent the resolution of each frequency component is reduced. If an excessively low-quality setting is used, the high-frequency components are discarded altogether. The quantized DCT coefficients are computed with

$$B_{j,k} = \text{round}\left(\frac{G_{j,k}}{Q_{j,k}}\right) \quad \text{for } j = 0, 1, 2, \ldots, 7; k = 0, 1, 2, \ldots, 7 \tag{25}$$

  The resulting data for all 8×8 blocks is further compressed by the coding of Shannon (simpler) and Fano, or the coding of Huffman [18]. The JPEG standard uses entropy coding, a variant of the Huffman encoding, which is a special form of lossless data compression. It involves arranging the image components in a "zigzag" order employing a run-length encoding (RLE) algorithm that groups similar frequencies together, inserting length coding zeros, and then using Huffman coding on what is left.

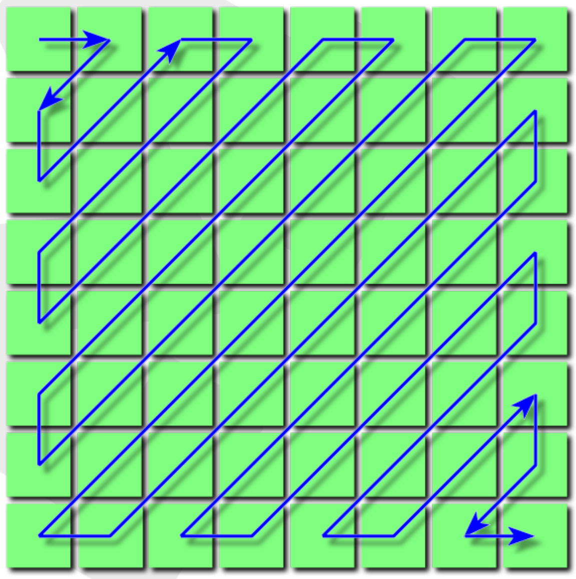


Figure 17: Zigzag ordering of JPEG image components.

The previous quantized DC coefficient is used to predict the current quantized DC coefficient. The difference between the two is encoded rather than the actual value. The encoding of the 63 quantized AC coefficients does not use such prediction differencing.

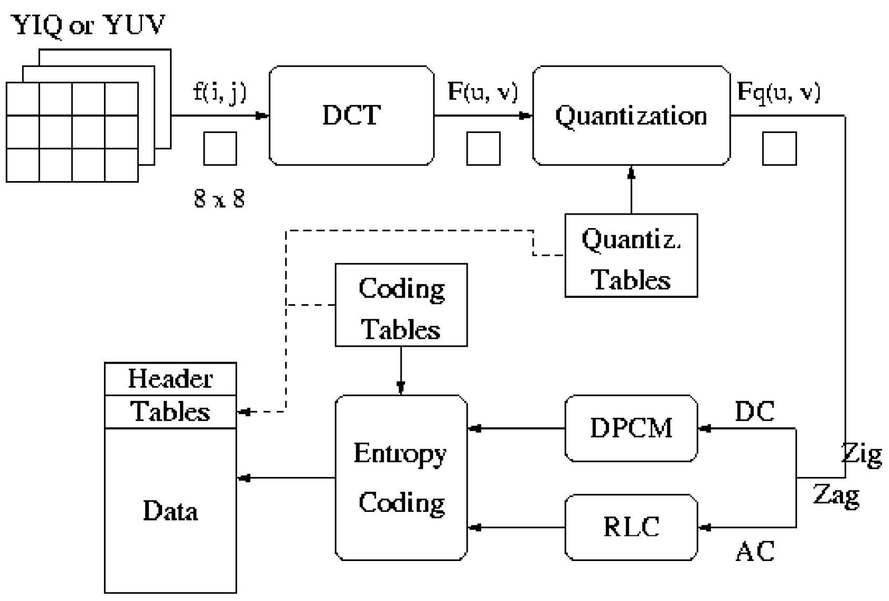


Figure 18: JPEG encoding overview [7].

The decoding is basically just reversing the three steps. There are two popular implementations of the JPEG standard: libjpeg in C and libjpeg-turbo in C and assembly. For example, you can see how they implement integer DCT and IDCT in libjpeg/jfdctint.c and libjpeg/jidctint.c, and you will see how are macros like `CONST_BITS` and `DESCALE` defined. JPEG or MPEG standards did not exactly specify which IDCT was to be used by decoders or which coding to be used for compression.

It is exciting that you have already done 2/3 of the work of JPEG encoding. However, the last part is not very simple. If you are familiar with video encoding algorithms, you will find that the three steps can also be seen in the encoding of video formats like H.264/AVC or HEVC. You may choose to complete a simple JPEG encoder/decoder or a simple video encoder/decoder in your final project.

4.2 Butterfly algorithm

Previously, you may notice that libjpeg implements DCT and IDCT in different ways. This is because it uses an algorithm called the butterfly algorithm, which is also used in structured linear maps like the discrete Fourier transform (DFT), discrete sine transform (DST), etc [6]. It is not simple to implement the entropy coding now, but you can optionally implement the butterfly algorithm for your DCT.

A good integer DCT has the following requirements:

- Bit-exact implementation strictly required.
- Must be separable, i.e. the 2D DCT factors into 1D DCT steps.
- Must be a close approximation to the DCT.
- Basis vectors must be fully orthogonal (or very close).

Before exploiting the algorithm, you should learn a diagram called the butterfly diagram. In the context of fast Fourier transform algorithms, a butterfly is a portion of the computation that combines the results of smaller discrete Fourier transforms (DFTs) into a larger DFT, or vice versa (breaking a larger DFT up into sub-transforms). Generally, A butterfly is the transform $(a, b) \mapsto (a + b, a - b)$. A butterfly is also its own inverse, up to a scale factor of two, since $(a + b) + (a - b) = 2a$ and likewise $(a + b) - (a - b) = 2b$ [9]. The name "butterfly" comes from the shape of the data-flow diagram in the radix-2 case.

$$y_0 = x_0 + x_1 \tag{26}$$

$$y_1 = x_0 - x_1 \tag{27}$$

If one draws the data-flow diagram for this pair of operations, the (x0, x1) to (y0, y1) lines cross and resemble the wings of a butterfly, hence the name [19].

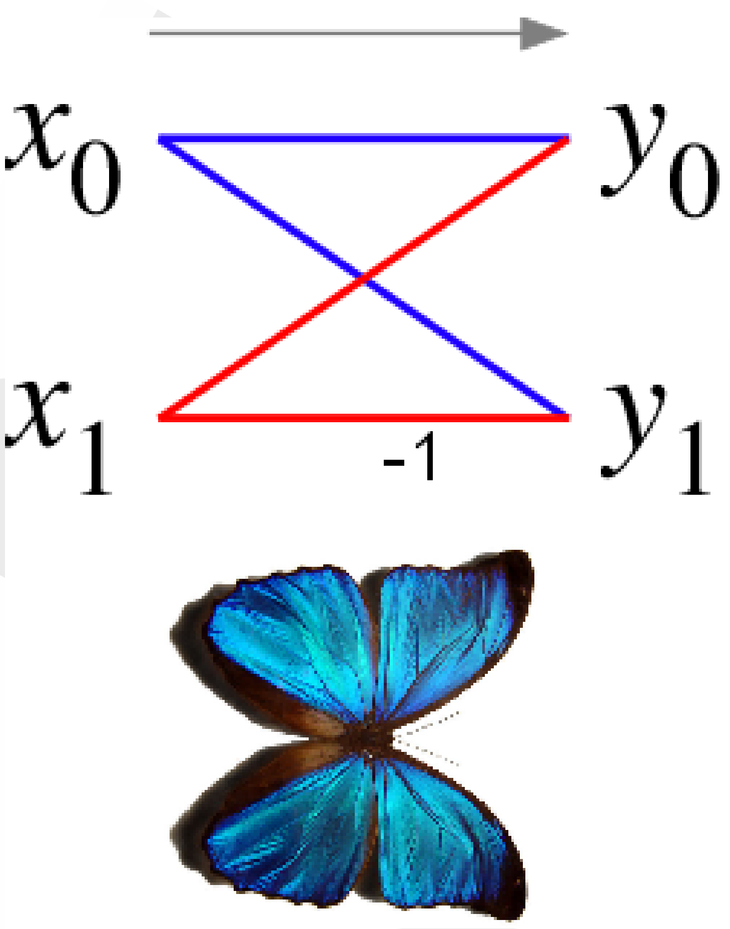


Figure 19: Signal-flow graph connecting the inputs x (left) to the outputs y that depend on them (right) for a "butterfly" step. This diagram resembles a butterfly (as in the morpho butterfly shown for comparison), hence the name, although in some countries it is also called the hourglass diagram [11].

Consider the 4 × 4 DCT basis matrix:

$$\mathbf{A} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{\cos\left(\frac{\pi}{8}\right)}{\sqrt{2}} & \frac{\sin\left(\frac{\pi}{8}\right)}{\sqrt{2}} & -\frac{\sin\left(\frac{\pi}{8}\right)}{\sqrt{2}} & -\frac{\cos\left(\frac{\pi}{8}\right)}{\sqrt{2}} \\ \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ \frac{\sin\left(\frac{\pi}{8}\right)}{\sqrt{2}} & -\frac{\cos\left(\frac{\pi}{8}\right)}{\sqrt{2}} & \frac{cos\left(\frac{\pi}{8}\right)}{\sqrt{2}} & -\frac{\sin\left(\frac{\pi}{8}\right)}{\sqrt{2}} \end{pmatrix} \tag{28}$$

Let

$$\begin{aligned} a &= \frac{1}{\sqrt{2}} \\ b &= \frac{\cos\left(\frac{\pi}{8}\right)}{\sqrt{2}} \\ c &= \frac{\sin\left(\frac{\pi}{8}\right)}{\sqrt{2}} \end{aligned} \tag{29}$$

Then

$$\mathbf{A} = \begin{pmatrix} a & a & a & a \\ b & c & -c & b \\ a & -a & -a & a \\ c & -b & b & -c \end{pmatrix} \tag{30}$$

Remember the DCT and IDCT are calculated by

$$\mathbf{Y} = \mathbf{A}\mathbf{X}\mathbf{A}^T \tag{31}$$

$$\mathbf{X} = \mathbf{A}^T\mathbf{Y}\mathbf{A} \tag{32}$$

respectively, where $\mathbf{X}$ is the original image. Then, Equation 31 is equivalent to

$$\mathbf{Y} = (\mathbf{C}\mathbf{X}\mathbf{C}^T)\cdot\mathbf{E} = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & d & -d & -1 \\ 1 & -1 & -1 & 1 \\ d & -1 & 1 & -d \end{bmatrix} \mathbf{X} \begin{bmatrix} 1 & 1 & 1 & d \\ 1 & d & -1 & -1 \\ 1 & -d & -1 & 1 \\ 1 & -1 & 1 & -d \end{bmatrix}\right) \cdot \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix} \tag{33}$$

where $d = \frac{c}{b} = \tan\frac{\pi}{8} = \sqrt{2} - 1$, $\cdot$ is the element-wise matrix multiplication. To simplify, take $d = 0.5$. To keep the orthogonality of the transform, modify $b = \sqrt{\frac{2}{5}}$. Multiply the second and the fourth row of $\mathbf{C}$ and modify $\mathbf{E}$:

$$\mathbf{Y} = (\mathbf{C}_f\mathbf{X}\mathbf{C}_f^T)\cdot\mathbf{E}_f = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \mathbf{X} \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{bmatrix}\right) \cdot \begin{bmatrix} a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \\ a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \end{bmatrix} \tag{34}$$

The element-wise matrix multiplication $(\cdot\mathbf{E}_f)$ do multiplication on each matrix element only once and will be reduced to the quantization process. Thus, the equation only contains integer addition, subtraction, and shifting (multiply by 2) [2].

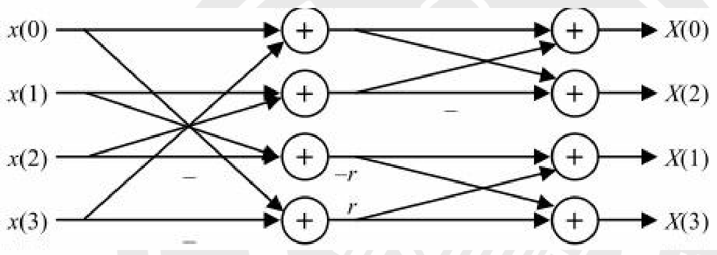
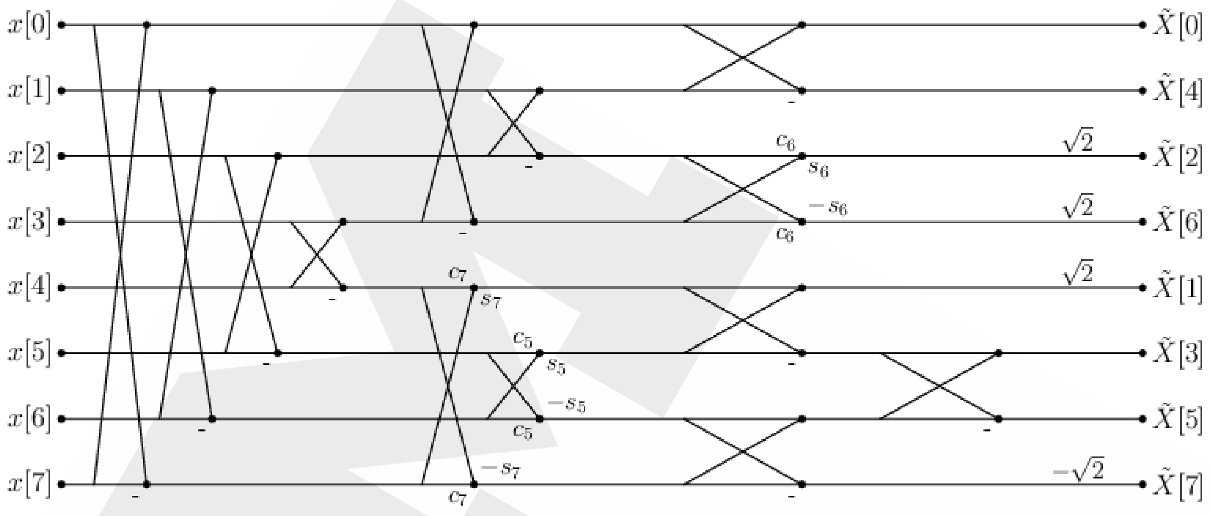


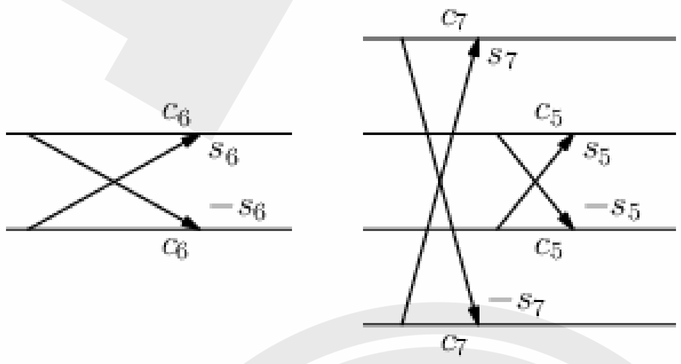
Figure 20: 1-D fast transform algorithm ($r = 2$ for integer DCT, $r = 1$ for Hadamard transform) [2].

Consider one column of $\mathbf{X}$,

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \begin{bmatrix} x_{01} \\ x_{11} \\ x_{21} \\ x_{31} \end{bmatrix} = \begin{bmatrix} (x_{01} + x_{31}) + (x_{11} + x_{21}) \\ 2(x_{01} - x_{31}) + (x_{11} - x_{21}) \\ (x_{01} + x_{31}) - (x_{11} + x_{21}) \\ (x_{01} - x_{31}) - 2(x_{11} - x_{21}) \end{bmatrix} \tag{35}$$

We found that there are four terms that are to be calculated twice, so can be optimized by the butterfly algorithm. For 8×8 DCT, the butterfly diagram is [9]





The sample MATLAB provided by [9] is

```
function out=bink_dct_B2(in)
  % extract rows
  i0 = in(1,:);
  i1 = in(2,:);
  i2 = in(3,:);
  i3 = in(4,:);
  i4 = in(5,:);
  i5 = in(6,:);
  i6 = in(7,:);
  i7 = in(8,:);

  % stage 1 - 8A
  a0 = i0 + i7;
  a1 = i1 + i6;
  a2 = i2 + i5;
  a3 = i3 + i4;
  a4 = i0 - i7;
  a5 = i1 - i6;
  a6 = i2 - i5;
  a7 = i3 - i4;

  % even stage 2 - 4A
  b0 = a0 + a3;
  b1 = a1 + a2;
  b2 = a0 - a3;
  b3 = a1 - a2;

  % even stage 3 - 6A 4S
  c0 = b0 + b1;
  c1 = b0 - b1;
  c2 = b2 + b2/4 + b3/2;
  c3 = b2/2 - b3 - b3/4;

  % odd stage 2 - 12A 8S
  % NB a4/4 and a7/4 are each used twice, so this really is 8 shifts, not
  ↪  10.
  b4 = a7/4 + a4 + a4/4 - a4/16;
  b7 = a4/4 - a7 - a7/4 + a7/16;
  b5 = a5 + a6 - a6/4 - a6/16;
  b6 = a6 - a5 + a5/4 + a5/16;

  % odd stage 3 - 4A
  c4 = b4 + b5;
  c5 = b4 - b5;
  c6 = b6 + b7;
  c7 = b6 - b7;

  % odd stage 4 - 2A
  d4 = c4;
  d5 = c5 + c7;
  d6 = c5 - c7;
  d7 = c6;

  % permute/output
  out = [c0; d4; c2; d6; c1; d5; c3; d7];

  % total: 36A 12S
end
```

Modify your `dct.c` and verify the correctness of your results. Generate the output data file `output.dat`. Create a new Vitis HLS solution with butterfly DCT and compare it with the previous DCT by Vitis HLS Report Comparison.

5 Introduction to PYNQ Overlays

5.1 Load a PYNQ overlay

Overlays, or hardware libraries, are programmable/configurable FPGA designs that extend the user application from the Processing System of the Zynq into the Programmable Logic. Overlays can accelerate a software application or customize a hardware platform for a particular application.

For example, image processing is a typical application where the FPGAs can provide acceleration. A software programmer can use an overlay in a similar way to a software library to run some of the image processing functions (e.g. edge detection, thresholding, etc.) on the FPGA fabric. Overlays can be loaded to the FPGA dynamically, as required, just like a software library. In this example, separate image processing functions could be implemented in different overlays and loaded from Python on demand.

PYNQ provides a Python interface to allow overlays in the PL to be controlled from Python running in the PS. FPGA design is a specialized task that requires hardware engineering knowledge and expertise. PYNQ overlays are created by hardware designers and wrapped with this PYNQ Python API. Software developers can then use the Python interface to program and control specialized hardware overlays without needing to design an overlay themselves. This is analogous to software libraries created by expert developers, which are then used by many other software developers working at the application level.

By default, an overlay (bitstream) called base is downloaded into the PL at boot time. The base overlay can be considered a reference design for a board. New overlays can be installed or copied to the board and loaded into the PL as the system runs.

An overlay usually includes:

- A bitstream to configure the FPGA fabric
- A Vivado design HWH file to determine the available IP
- Python API that exposes the IPs as attributes

The PYNQ `Overlay` class can be used to load an overlay. An overlay is instantiated by specifying the name of the bitstream file. Instantiating the Overlay also downloads the bitstream by default and parses the HWH file.

For the base overlay, we can use the existing `BaseOverlay` class; this class exposes the IPs available on the bitstream as attributes of this class.

Open PYNQ. If the browser shows `DNS_PROBE_FINISHED_NXDOMAIN`, run

```
ipconfig /flushdns
```

Note that the Jupyter notebook home directory is `root@pynq:/home/xilinx/jupyter_notebooks`. You can check by `ls` command. You can also access the file explorer of Windows by `\\pynq`. However, operating files are directories in Terminal is much faster than file explorer.

Create a new Python 3 notebook.

```
from pynq.overlays.base import BaseOverlay
base_overlay = BaseOverlay("base.bit")
```

Once an overlay has been instantiated, the `help()` method can be used to discover what is in an overlay. The help information can be used to interact with the overlay. Note that if you try the following code on your own board, you may see different results depending on the version of PYNQ you are using, and which board you have.

```
help(base_overlay)
```

This will give a list of the IPs and methods available as part of the overlay.

From the `help()` printout above, it can be seen that in this case, the overlay includes a `gpio_dict` instance:

```
"""
leds : AxiGPIO

     4-bit output GPIO for interacting with the green LEDs LD0-3
"""
```

Running `help()` on the `gpio_dict` object will provide more information about the object, including details of its API.

```
help(base_overlay.leds)
```

The API can be used to control the object. For example, the following cell will turn on LD0 on the board.

```
base_overlay.leds[0].toggle()
```

Information about other IPs can be found from the overlay instance in a similar way, as shown below.

```
help(base_overlay.video)
```

Save your Jupyter notebook as `lab3_1.ipynb` with outputs.

5.2 Partial configuration

From image v2.4, PYNQ supports partial bitstream reconfiguration. The partial bitstreams are managed by the overlay class. It is always recommended to use the .hwh file along with the .bit for the overlay class.

5.2.1 Preparing the files

There are many ways to prepare bitstreams. Users can choose to follow the project flow or the software flow to implement a partial reconfiguration Vivado project. For more information, please refer to the documentation page on partial reconfiguration.

After each reconfiguration, the PL status will update to reflect the changes on the bitstream, so that new drivers can be assigned to the new blocks available in the bitstream. To achieve this, users have to provide the metadata file (.hwh file) along with each full/-partial bitstream. The .hwh file is typically located at: <project_name>/<design_name>.gen/sources_1/bd/<design_name>/hw_handoff/.

Keep in mind that each partial bitstream needs a .hwh file.

In this lab, we will use boards/Pynq-Z1/gpio_pr/notebooks/partial_reconfig from `https://github.com/yunqu/partial_reconfig_example`.

5.2.2 Loading full bitstream

It is straightforward to download a full bitstream. By default, the bitstream will be automatically downloaded onto the PL when users instantiate an overlay object.

```
from pynq import Overlay
overlay = Overlay('gpio_pr.bit')
```

To download the full bitstream again:

```
overlay.download()
```

Note that no argument is provided if a full bitstream is to be downloaded.

Another thing to note, is that if the Vivado project is configured as a partial reconfiguration project, the .hwh file for the full bitstream will not contain any information inside a partial region, even if the full bitstream always has a default Reconfiguration Module (RM) implemented. Instead, the .hwh file only provides information on the interfaces connecting to the partial region. So for the full bitstream, do not be surprised if you see an empty partial region in the .hwh file. The complete information on the partial regions are revealed by the .hwh files of the partial bitstreams, where each .hwh file reveals one possible internal organization of the partial region.

Run

```
from pprint import pprint
pprint(overlay.interrupt_pins)
pprint(overlay.pr_dict)
```

5.2.3 Loading partial bitstream

Typically, the partial regions are hierarchies in the block design of the bitstream. In an overlay object, the hierarchical blocks are exposed as attributes of the object. In the following example, let us assume there is a hierarchical block called gpio_0 in the design. There are two ways to download a partial bitstream.

The first way, using the `download()` method of the `DefaultHierarchy` class, please see `pynq.overlay.DefaultHierarchy.download()`.

The second way, using `pr_download()` method of the `Overlay` class. For this, users have to specify the partial region as well as the partial bitstream, see `pynq.overlay.Overlay.pr_` download().

```
overlay.pr_download('gpio_0', 'led_5.bit')
```

Observe the LEDs change on your board. Take a photo of your board. To load a different RM:

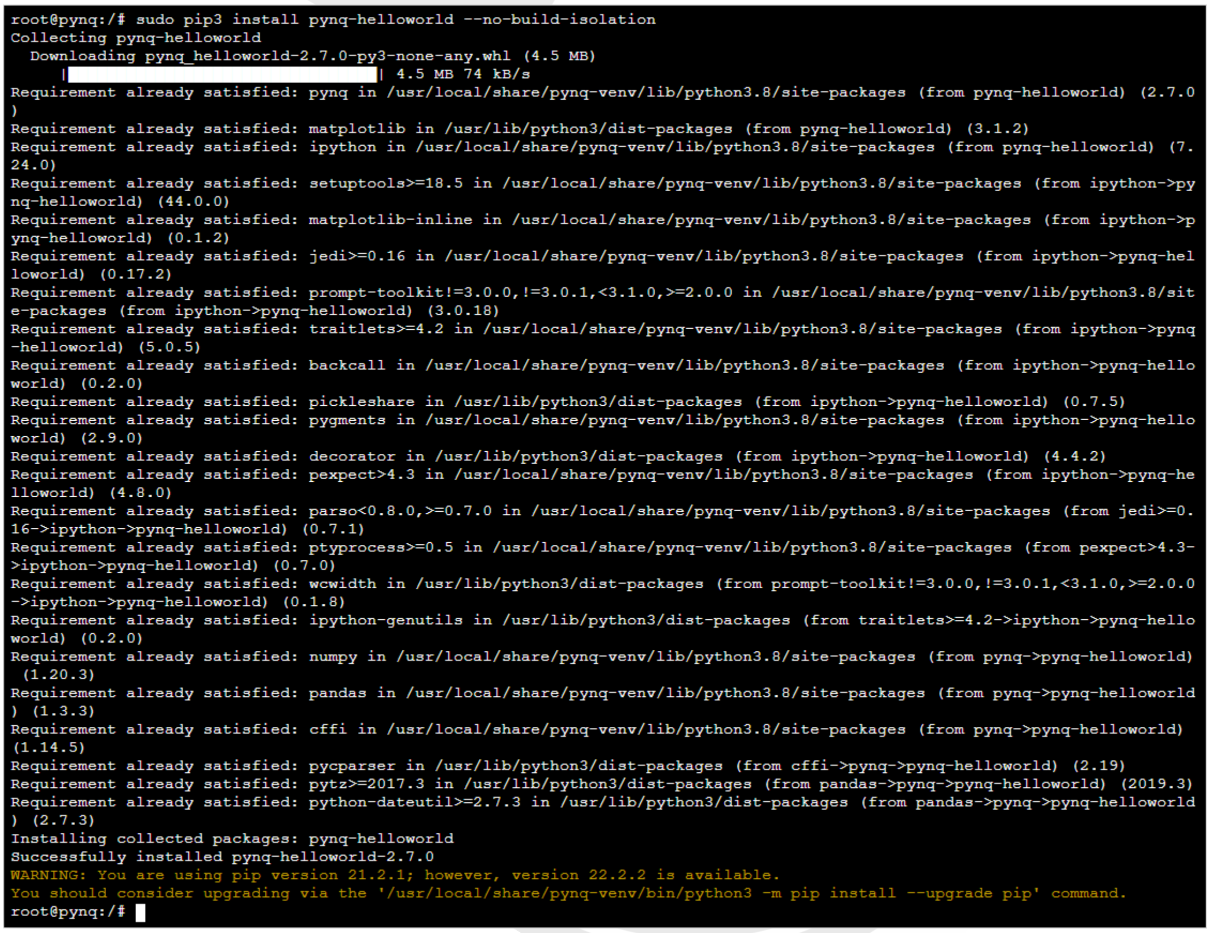
```
overlay.pr_download('gpio_0', 'led_a.bit')
```

Take another photo of your board. Save your Jupyter notebook as `lab3_2.ipynb` with outputs.

5.3 PYNQ HelloWorld

Open a terminal on your PYNQ board and run:

```
sudo pip3 install pynq-helloworld --no-build-isolation
```



```
root@pynq:/# sudo pip3 install pynq-helloworld --no-build-isolation
Collecting pynq-helloworld
  Downloading pynq_helloworld-2.7.0-py3-none-any.whl (4.5 MB)
     |████████████████████████████████| 4.5 MB 74 kB/s
Requirement already satisfied: pynq in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from pynq-helloworld) (2.7.0
)
Requirement already satisfied: matplotlib in /usr/lib/python3/dist-packages (from pynq-helloworld) (3.1.2)
Requirement already satisfied: ipython in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from pynq-helloworld) (7.
24.0)
Requirement already satisfied: setuptools>=18.5 in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from ipython->py
nq-helloworld) (44.0.0)
Requirement already satisfied: matplotlib-inline in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from ipython->p
ynq-helloworld) (0.1.2)
Requirement already satisfied: jedi>=0.16 in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from ipython->pynq-hel
loworld) (0.17.2)
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0 in /usr/local/share/pynq-venv/lib/python3.8/sit
e-packages (from ipython->pynq-helloworld) (3.0.18)
Requirement already satisfied: traitlets>=4.2 in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from ipython->pynq
-helloworld) (5.0.5)
Requirement already satisfied: backcall in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from ipython->pynq-hello
world) (0.2.0)
Requirement already satisfied: pickleshare in /usr/lib/python3/dist-packages (from ipython->pynq-helloworld) (0.7.5)
Requirement already satisfied: pygments in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from ipython->pynq-hello
world) (2.9.0)
Requirement already satisfied: decorator in /usr/lib/python3/dist-packages (from ipython->pynq-helloworld) (4.4.2)
Requirement already satisfied: pexpect>4.3 in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from ipython->pynq-he
lloworld) (4.8.0)
Requirement already satisfied: parso<0.8.0,>=0.7.0 in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from jedi>=0.
16->ipython->pynq-helloworld) (0.7.1)
Requirement already satisfied: ptyprocess>=0.5 in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from pexpect>4.3-
>ipython->pynq-helloworld) (0.7.0)
Requirement already satisfied: wcwidth in /usr/lib/python3/dist-packages (from prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0
->ipython->pynq-helloworld) (0.1.8)
Requirement already satisfied: ipython-genutils in /usr/lib/python3/dist-packages (from traitlets>=4.2->ipython->pynq-hello
world) (0.2.0)
Requirement already satisfied: numpy in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from pynq->pynq-helloworld)
 (1.20.3)
Requirement already satisfied: pandas in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from pynq->pynq-helloworld
) (1.3.3)
Requirement already satisfied: cffi in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from pynq->pynq-helloworld)
(1.14.5)
Requirement already satisfied: pycparser in /usr/lib/python3/dist-packages (from cffi->pynq->pynq-helloworld) (2.19)
Requirement already satisfied: pytz>=2017.3 in /usr/lib/python3/dist-packages (from pandas->pynq->pynq-helloworld) (2019.3)
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/lib/python3/dist-packages (from pandas->pynq->pynq-helloworld
) (2.7.3)
Installing collected packages: pynq-helloworld
Successfully installed pynq-helloworld-2.7.0
WARNING: You are using pip version 21.2.1; however, version 22.2.2 is available.
You should consider upgrading via the '/usr/local/share/pynq-venv/bin/python3 -m pip install --upgrade pip' command.
root@pynq:/# █
```

Currently, this repository is compatible with pynq package v2.7.0.

Go to your Jupyter home folder (on our boards, this is /home/xilinx/jupyter_notebooks), and run the following to deliver the notebooks:

```
pynq get-notebooks pynq-helloworld -p .
```

Run all cells of pynq-helloworld/resizer_pl.ipynb and pynq-helloworld/resizer_ps.ipynb. Take screenshots of your outputs. Explain your results.

5.4 Creating overlays

Go through the "Developing a Single IP" section and the "Creating a Driver" section in Overlay Tutorial. The sample Jupyter notebook is [14] [13]:

```
from pynq import Overlay

overlay = Overlay('../pynq_adder.bit')
overlay?
add_ip = overlay.scalar_add
add_ip?
add_ip.register_map
add_ip.register_map.a = 3
add_ip.register_map.b = 4
add_ip.register_map.c
add_ip.write(0x10, 4)
add_ip.write(0x18, 5)
add_ip.read(0x20)
from pynq import DefaultIP

class AddDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:hls:hls_adder:1.0']

    def add(self, a, b):
        self.write(0x10, a)
        self.write(0x18, b)
        return self.read(0x20)
overlay = Overlay('../pynq_adder.bit')
overlay?
overlay.scalar_add.add(15,20)
```

Save your Jupyter notebook as `lab3_3.ipynb` with outputs.

5.5 Questions

1. Please list out any advantages of using overlay.
2. In general, when you choose the software approach vs. hardware acceleration approach, what are the considerations? Any tradeoffs?

6 (Optional) FIR Filter on PYNQ

6.1 Implementation and optimization of an FIR filter by HLS

According to Lab 3 slides, implement a functionally correct, but not optimized, 128-tap FIR filter. This is your baseline implementation. Use the provided script.tcl to create your project. Try the following optimization:

- Variable Bitwidths: It is possible to specify a very precise data type for each variable in your design. The number of different data types is extensive: floating point, integer, and fixed point, all with varying bit widths and options. The data type provides a tradeoff between accuracy, resource usage, and performance.
- Pipelining: Pipelining increases the throughput typically at the cost of additional resources. The initiation interval (II) is an important design parameter that changes performance and resource usage.
- Removing Conditional Statements: If/else statements and other conditionals can limit the possible parallelism and often require additional resources. If the code can be rewritten to remove them, it can make the resulting design more efficient. This is known as code hoisting.
- Loop Partitioning: Dividing the loop into two or more separate loops may allow for each of those loops to be executed in parallel (via unrolling), enable loop level pipelining, or provide other benefits. This may increase performance and resource usage.
- Memory Partitioning: The storage of the arrays in memory plays an important role in area and performance. On the one hand, you could put an array entirely in one memory (e.g., BRAM). But this limits the number of read accesses and write accesses per cycle. Or you can divide the array into two or more memories to increase the number of ports. Or you could instantiate each of the variables as its own register, which allows simultaneous access to all of the variables at every clock cycle.

Combine any number of optimizations to get your best architecture. A design with high throughput will likely take a lot of resources. A design that has small resource usage likely will have lower performance, but that could still be the best depending on the application goals. It is possible that some optimizations may not have a big (or any effect). Some optimizations may only work when you use them in combination with others. This is what makes the design space exploration process difficult.

6.2 From HLS to PYNQ

1. HLS to RTL: Run C Synthesis, Implementation, and export RTL in Vitis HLS. *Sometimes you may need to set port types before synthesis: Make sure that fir_test.cpp is open. Open Directive and right click on the options, and select Modify Directive. In the drop-down, select INTERFACE and set all the ports to s_axilite by right-clicking on available options. Repeat the same for all the options.*

2. RTL to Bitstream: Add FIR Filter IP to your block design in your Vivado project with ZYNQ7 Processing System. Generate bitstream. Before closing Vivado, we need to note our IP and its port addresses. Under Sources, open fir_test_fir_io_s_axi.v, scroll down and note addresses for in and out ports. We need these addresses for our host program. For example, the addresses to pay attention to are 0x00 (control bus ap_ctrl), 0x10 (output), and 0x18 (input). These are the addresses you will need to use to write data to the fabric from the ARM core, start the fabric to run your design and generate your outputs, and then read your outputs from the fabric into the ARM core on the Pynq board.

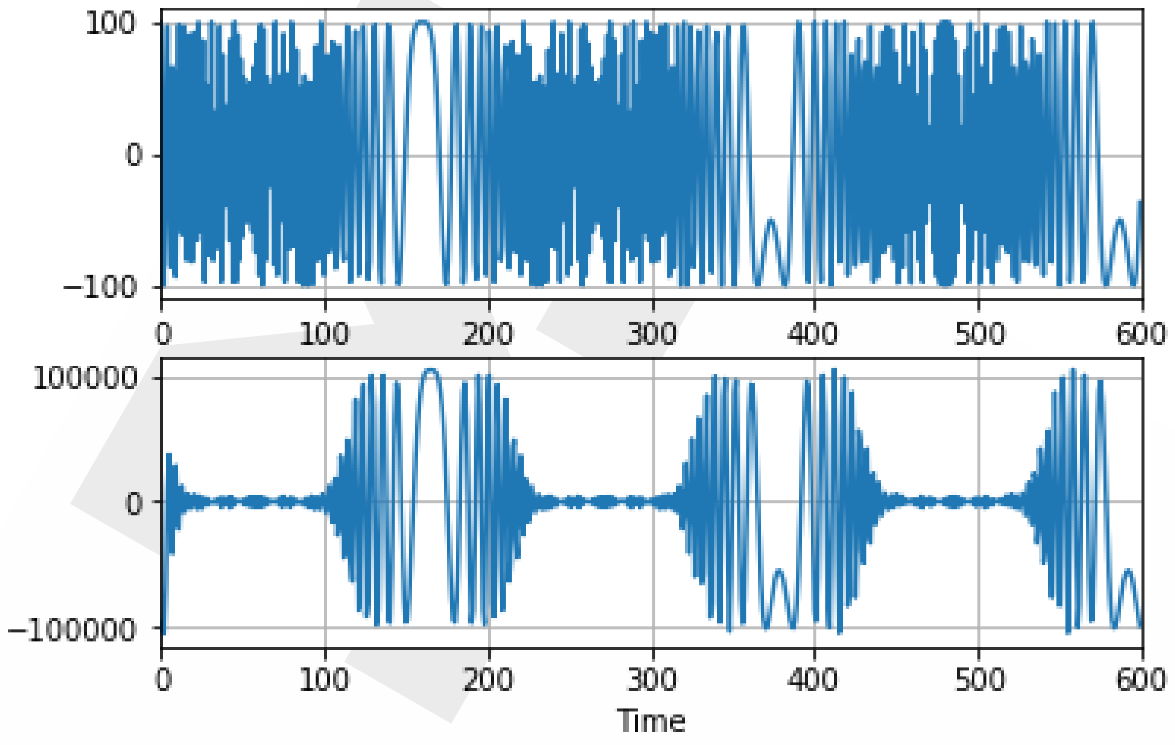
3. Bitstream on PYNQ: Make sure to name the .bit file and the .hwh file with the same name. For example, we name them "design_1_wrapper.bit" and "design_1_wrapper.hwh".

6.3 Verification on PYNQ

Write a host program host_fir.ipynb. The example usage of MMIO is

```python
from pynq import Overlay
from pynq import MMIO
ol =
↪ Overlay("/home/xilinx/jupyter_notebooks/fir_test/design_1_wrapper.bit")
↪ # designate a bitstream to be flashed to the FPGA
ol.download() # flash the FPGA
fir_ip = MMIO(0x43C00000, 0x10000) # (IP_BASE_ADDRESS, ADDRESS_RANGE),
↪ told to us in Vivado
inp = 5
fir_ip.write(0x18, inp) # write input value to input address in fabric
print("input:", fir_ip.read(0x18)) # confirm that our value was written
↪ correctly to the fabric
fir_ip.write(0x00, 1) # set ap_start to 1 which initiates the process we
↪ wrote to the fabric
print("output:", fir_ip.read(0x10)) # read corresponding output value
↪ from the output address of the fabric
```

Use `matplotlib` to plot the input data vs. time and the output data vs. time. The sample image by the 11-tap FIR filter is as shown below:



You are expected to show a similar image using the output of your 128-tap FIR filter.

7 Deliverables

1. Section 2:
   (a) Section 2.4: answer the questions
   (b) Section 2.5: the screenshot of the Performance Estimates and Utilization Estimates of the Vitis HLS Report Comparison.
   (c) Section 2.6: the screenshot of the Performance Estimates and Utilization Estimates of the Synthesis Details Report after `DATAFLOW` directive is applied
   (d) Section 2.6: the screenshot of the Performance Estimates and Utilization Estimates of the Synthesis Details Report after configuring Dataflow commands
   (e) Section 2.7 (Optional): the screenshot of your implementation report and the exported RTL zip file
2. Section 3:
   (a) Section 3.4: answer the questions in Step 4
   (b) Section 3.5: the screenshot of the Utilization Estimates detail of memory in Step 5
   (c) Section 3.7: the screenshot of the Performance Estimates and the Utilization Estimates of the Vitis HLS Report Comparison between all five solutions
   (d) Section 3.9: the modified `dct.c` code and the generated output data file `output.dat`
   (e) Section 3.9 (Optional): the recovered image
3. Section 4 (Optional): the modified `dct.c` code, `output.dat`, and the report comparison (if you finish this section, no need to finish Section 6)
4. Section 5:
   (a) Section 5.1: the lab3_1.ipynb code with outputs
   (b) Section 5.2: the two photos of the board and the lab3_2.ipynb code with outputs
   (c) Section 5.3: the screenshots of the outputs of resizer_pl.ipynb and the outputs of resizer_ps.ipynb
   (d) Section 5.4: the lab3_3.ipynb code with outputs
   (e) Section 5.5: answer to the questions
5. Section 6 (Optional): the `fir.cpp` code and the `host_fir.ipynb` including the plots (if you finish this section, no need to finish Section 4)

8 Grading policy

Factors	Percentage
Section 2	35% + 2% bonus
Section 3	35% + 3% bonus
Section 4	20% bonus (not take effect concurrently with Section 6)
Section 5	30%
Section 6	20% bonus (not take effect concurrently with Section 4)

A Peer Evaluation Form

Part	Your work	Your partner's work	Your score	Your partner's score
Section 2				
Section 3				
Section 4				
Section 5				
Section 6				

B Troubleshooting

B.1 Vitis HLS Command Prompt Make

If you failed to run `make` in Vitis, read [15]. This is an old bug since at least Vitis HLS 2021.1. It was reported by Prof. Guo and me last year, but it is still not fixed, as Xilinx always does not care about thousands of user-reported bugs on its support website, especially for those on the Windows platform. Generally, your system PATH does not contain `gcc`, so just change Line 3 of `vitis_hls_cmd.bat` to

```
3  set PATH=%~dp0;%PATH%;%~dp0..\tps\win64\msys64\usr\bin;%~dp0..\tps\win64
   ↪  \msys64\mingw64\bin
```

After doing that, you may also change Line 6 to:

```
6  set VITIS_HLS_HOME=%~dp0..
```

B.2 Vitis HLS FIFO Configuration

This is an old bug since at least Vitis HLS 2021.1. It was reported by Prof. Guo and me last year, but it is still not fixed. In viis_hls.log, you can see:

```
INFO: [RTMG 210-285] Implementing FIFO
↪  'Y_scale_c_U(yuv_filter_fifo_w8_d3_S)' using Shift Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'U_scale_c_U(yuv_filter_fifo_w8_d3_S)' using Shift Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'V_scale_c_U(yuv_filter_fifo_w8_d3_S)' using Shift Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_yuv_channels_ch1_U(yuv_filter_fifo_w8_d2457600_A)' using Vivado
↪  Default RAMs.
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_yuv_channels_ch2_U(yuv_filter_fifo_w8_d2457600_A)' using Vivado
↪  Default RAMs.
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_yuv_channels_ch3_U(yuv_filter_fifo_w8_d2457600_A)' using Vivado
↪  Default RAMs.
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_yuv_width_U(yuv_filter_fifo_w16_d2_S)' using Shift Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_yuv_height_U(yuv_filter_fifo_w16_d2_S)' using Shift Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_scale_channels_ch1_U(yuv_filter_fifo_w8_d2457600_A)' using Vivado
↪  Default RAMs.
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_scale_channels_ch2_U(yuv_filter_fifo_w8_d2457600_A)' using Vivado
↪  Default RAMs.
```

```
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_scale_channels_ch3_U(yuv_filter_fifo_w8_d2457600_A)' using Vivado
↪  Default RAMs.
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_scale_width_U(yuv_filter_fifo_w16_d2_S)' using Shift Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_scale_height_U(yuv_filter_fifo_w16_d2_S)' using Shift Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'start_for_yuv_scale_U0_U(yuv_filter_start_for_yuv_scale_U0)' using
↪  Shift Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'start_for_yuv2rgb_1_U0_U(yuv_filter_start_for_yuv2rgb_1_U0)' using
↪  Shift Registers.
```

The channels are still using Vivado Default RAMs instead of Shift Registers.

Solution 1. Modify Line 10 of your yuv_filter.prj/solution3/solution3.aps from

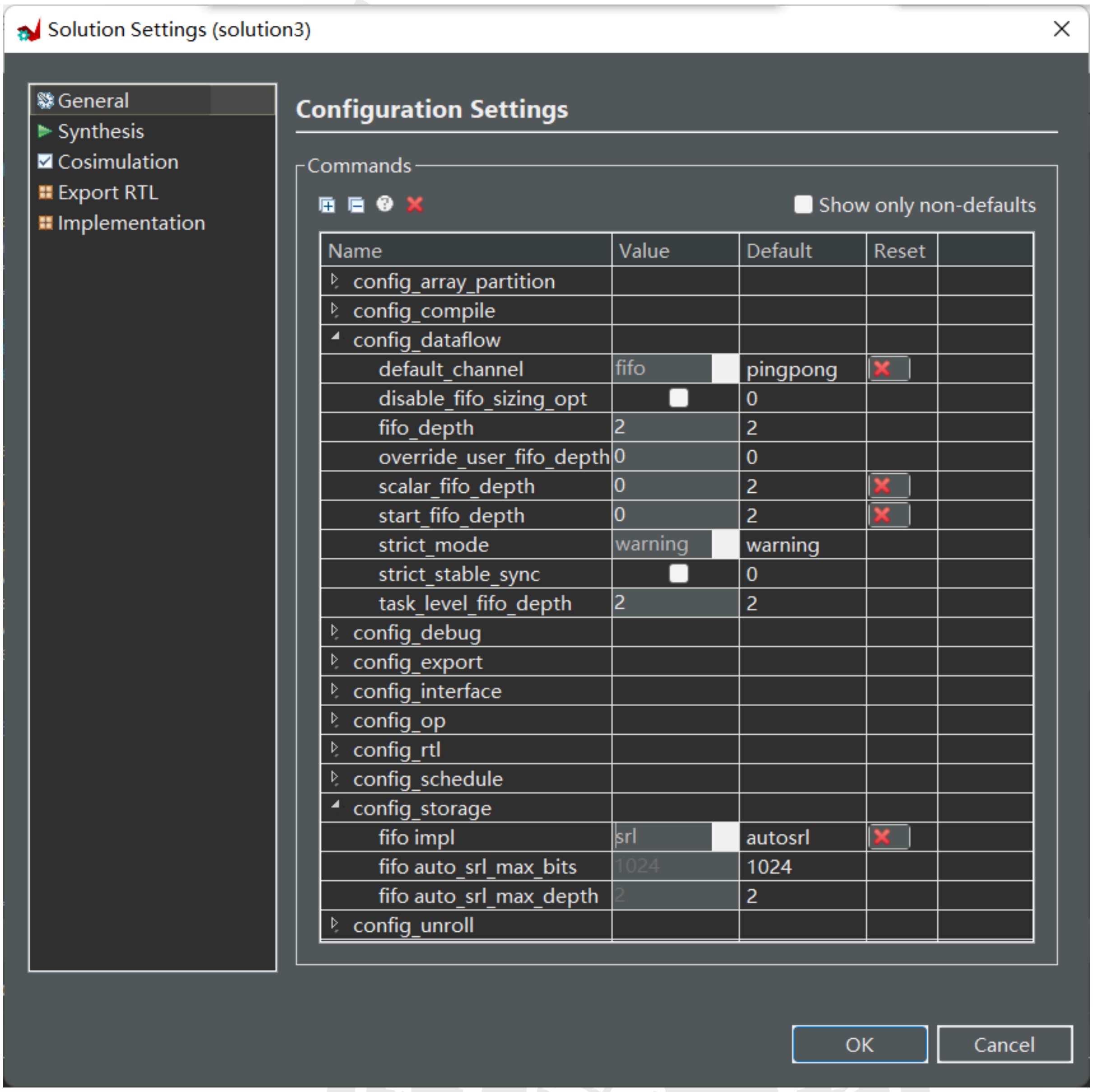
```
> <config_dataflow default_channel="fifo" scalar_fifo_depth="0"
↪  start_fifo_depth="0"/>
```

to

```
> <config_dataflow default_channel="fifo" fifo_depth="2"
↪  scalar_fifo_depth="0" start_fifo_depth="0"/>
```

and redo C Synthesis.

Solution 2. Solution Settings (solution3)->General->Configuration Settings->Commands->config_storage->fifo impl, change value from default ‘autosrl‘ to ‘srl‘ and redo C Synthesis.


Solution Settings (solution3)

Configuration Settings

Commands

Name	Value	Default	Reset
config_array_partition			
config_compile			
config_dataflow			
default_channel	fifo	pingpong	✕
disable_fifo_sizing_opt	☐	0	
fifo_depth	2	2	
override_user_fifo_depth	0	0	
scalar_fifo_depth	0	2	✕
start_fifo_depth	0	2	✕
strict_mode	warning	warning	
strict_stable_sync	☐	0	
task_level_fifo_depth	2	2	
config_debug			
config_export			
config_interface			
config_op			
config_rtl			
config_schedule			
config_storage			
fifo impl	srl	autosrl	✕
fifo auto_srl_max_bits	1024	1024	
fifo auto_srl_max_depth	2	2	
config_unroll			

After that, you should see the following information in vitis_hls.log:

```
INFO: [RTMG 210-285] Implementing FIFO
↪  'Y_scale_c_U(yuv_filter_fifo_w8_d3_S)' using Shift Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'U_scale_c_U(yuv_filter_fifo_w8_d3_S)' using Shift Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'V_scale_c_U(yuv_filter_fifo_w8_d3_S)' using Shift Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_yuv_channels_ch1_U(yuv_filter_fifo_w8_d2457600_S)' using Shift
↪  Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_yuv_channels_ch2_U(yuv_filter_fifo_w8_d2457600_S)' using Shift
↪  Registers.
```

```
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_yuv_channels_ch3_U(yuv_filter_fifo_w8_d2457600_S)' using Shift
↪  Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_yuv_width_U(yuv_filter_fifo_w16_d2_S)' using Shift Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_yuv_height_U(yuv_filter_fifo_w16_d2_S)' using Shift Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_scale_channels_ch1_U(yuv_filter_fifo_w8_d2457600_S)' using Shift
↪  Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_scale_channels_ch2_U(yuv_filter_fifo_w8_d2457600_S)' using Shift
↪  Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_scale_channels_ch3_U(yuv_filter_fifo_w8_d2457600_S)' using Shift
↪  Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_scale_width_U(yuv_filter_fifo_w16_d2_S)' using Shift Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'p_scale_height_U(yuv_filter_fifo_w16_d2_S)' using Shift Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'start_for_yuv_scale_U0_U(yuv_filter_start_for_yuv_scale_U0)' using
↪  Shift Registers.
INFO: [RTMG 210-285] Implementing FIFO
↪  'start_for_yuv2rgb_1_U0_U(yuv_filter_start_for_yuv2rgb_1_U0)' using
↪  Shift Registers.
```

B.3 Vitis HLS Export RTL ERROR IMPL

Check yuv_filter.prj/solution3/impl/ip/hs_err_pid38996.log, reboot your computer, and re-run implementation. You can specify IP Location, and it will automatically export RTL there.

C Change Log

Fall 2022: Yihua Liu

- merged Lab 3 and Lab 4 from Word to LaTeX
- added an introduction to YUV model and updated the starter code
- updated Section 5
- added Section 3, 4, 6, and the troubleshooting section

Fall 2021: Prof. Xinfei Guo

- created this lab of optimizing performance through pipelining as Lab 4

- created this lab of PYNQ overlays as Lab 3

References

[1] amritkumar2306 et al. *synthesis is taking forever*. Mar. 18, 2021. URL: https://support.xilinx.com/s/question/0D52E00006hpQZqSAM/synthesis-is-taking-forever?language=en_US.

[2] Houjie Bi and Jian Wang. *New generation video compression coding standard - H.264/AVC*. 2nd ed. Beijing: Posts & Telecom Press, Nov. 2009. ISBN: 978-7-115-21436-2.

[3] Siheng Chen. *ECE4880J: Computer Vision Homework 3: Image Processing*. University of Michigan - Shanghai Jiao Tong University Joint Institute, June 1, 2022.

[4] Microsoft Corporation. *Converting Between YUV and RGB*. Apr. 8, 2004. URL: https://learn.microsoft.com/en-us/previous-versions/windows/embedded/ms893078(v=msdn.10).

[5] Antonio D'Angelo. *Lecture 11: Discrete Cosine Transform*. Department of Mathematics and Computer Science, University of Udine, 2013. URL: http://users.dimi.uniud.it/~antonio.dangelo/MMS/2013/lessons/L11lecture.pdf.

[6] Tri Dao et al. *Butterflies Are All You Need: A Universal Building Block for Structured Linear Maps*. June 13, 2019. URL: https://dawn.cs.stanford.edu/2019/06/13/butterfly/.

[7] James Harold Davenport. *CM30142 Chapter 4. Video and Audio Compression*. Department of Computer Science, University of Bath, 2011. URL: http://www.cs.bath.ac.uk/jpff/CM30142/compression.pdf.

[8] Lena Forsén. *The Lenna Story*. 1972. URL: http://www.lenna.org/lena_std.tif.

[9] Fabian "ryg" Giesen. *Bink 2.2 integer DCT design, part 1*. Kirkland, King County, Washington, USA, Nov. 4, 2013. URL: https://fgiesen.wordpress.com/2013/11/04/bink-2-2-integer-dct-design-part-1/.

[10] Keith Jack. *Video Demystified*. HighText Publications, 2013. ISBN: 1-878707-09-4.

[11] Steven G. Johnson. *Butterfly-FFT*. Sept. 19, 2005. URL: https://upload.wikimedia.org/wikipedia/commons/9/98/Butterfly-FFT.png.

[12] A. Kassem, M. Hamad, and E. Haidamous. "Image compression on FPGA using DCT". In: *2009 International Conference on Advances in Computational Tools for Engineering Applications*. 2009, pp. 320–323. DOI: 10.1109/ACTEA.2009.5227881.

[13] Yihua Liu. *Creating Pynq Overlay*. Oct. 25, 2021. URL: https://blog.csdn.net/yihuajack/article/details/120931539.

[14] Yihua Liu. *PYNQ-Adder*. Oct. 25, 2021. URL: https://github.com/yihuajack/PYNQ-Adder.

[15] Yihua Liu. *Vitis HLS 2021.1 Command Prompt make not an internal or external command*. Oct. 31, 2021. URL: https://blog.csdn.net/yihuajack/article/details/121047540.

[16] MathWorks. *Discrete Cosine Transform*. Version R2022b. 2022. URL: https://www.mathworks.com/help/images/discrete-cosine-transform.html.

[17] Tonyle. *Example of U-V color plane, Y' value = 0.5, represented within RGB color gamut.* June 8, 2009. URL: https://upload.wikimedia.org/wikipedia/commons/f/f9/YUV_UV_plane.svg.

[18] Wikibooks. *JPEG - Idea and Practice/The compression and encoding — Wikibooks, The Free Textbook Project.* [Online; accessed 8-October-2022]. 2018. URL: %5Curl%7Bhttps://en.wikibooks.org/w/index.php?title=JPEG_-_Idea_and_Practice/The_compression_and_encoding&oldid=3424423%7D.

[19] Wikipedia contributors. *Butterfly diagram — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Butterfly_diagram&oldid=1111230153. [Online; accessed 8-October-2022]. 2022.

[20] Wikipedia contributors. *Discrete cosine transform — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Discrete_cosine_transform&oldid=1113261571. [Online; accessed 4-October-2022]. 2022.

[21] Wikipedia contributors. *JPEG — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=JPEG&oldid=1113230705. [Online; accessed 8-October-2022]. 2022.

[22] Wikipedia contributors. *YUV — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=YUV&oldid=1114229318. [Online; accessed 6-October-2022]. 2022.

[23] Wolfram. *FourierDCTMatrix.* Version 13.1. 2022. URL: https://reference.wolfram.com/language/ref/FourierDCTMatrix.html.en.

[24] Xilinx. *Vitis High-Level Synthesis User Guide (UG1399).* Version v2022.1. June 7, 2022.