LAB A

Due 3.12.2023

Application Acceleration with High-Level Synthesis

11120EE521800

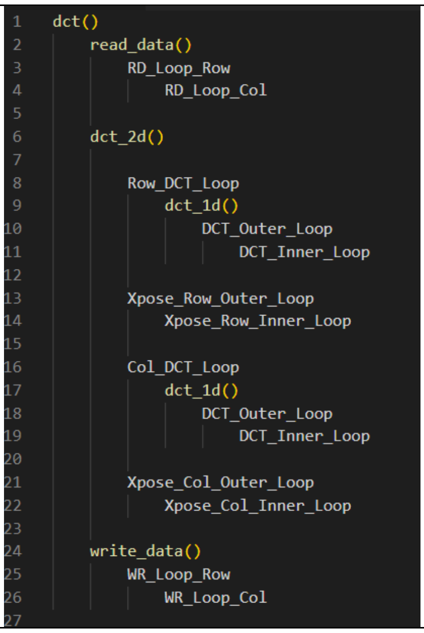
Department of Electrical Engineering

National Tsing Hua University, Taiwan

Name: 吳永玹

ID: 111022533

- Introduce the DCT code structure, hierarchy, loops. And code snippet

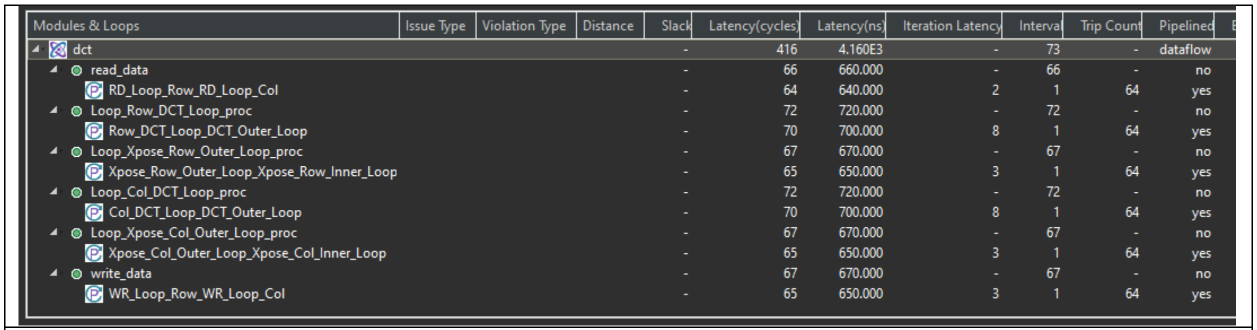


```
dct()
    read_data()
        RD_Loop_Row
            RD_Loop_Col

    dct_2d()

        Row_DCT_Loop
            dct_1d()
                DCT_Outer_Loop
                    DCT_Inner_Loop

        Xpose_Row_Outer_Loop
            Xpose_Row_Inner_Loop

        Col_DCT_Loop
            dct_1d()
                DCT_Outer_Loop
                    DCT_Inner_Loop

        Xpose_Col_Outer_Loop
            Xpose_Col_Inner_Loop

    write_data()
        WR_Loop_Row
            WR_Loop_Col
```

Fig.1

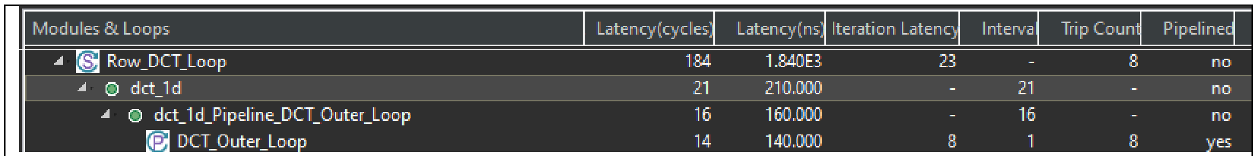
The hierarchy of DCT code is as shown in Fig.1. The top function is `dct()`. `dct()` first reads data into fpga BRAM by calling `read_data()`, then calls `dct_2d()` to process the data, finally writes the processed data out using `write_data()`. Both `read_data()` and `write_data()` use two level nested loop to read/write data, since the input/output data is in the form of 2d array. `dct_2d()` first perform 1d dct transform on the rows (`Row_DCT_Loop`) of input data, then perform same transform on its columns (`Row_DCT_Loop`). In order to reuse the same function `dct_1d()`, the code transpose (`Xpose_Row_Outer/Inner_Loop`) the output of `dct_1d()` after perform 1d dct on rows, then pass this to `dct_1d()` again, then transpose (`Xpose_Col_Outer/Inner_Loop`) the output back. `dct_1d()` internally uses two-level nested loops to perform product of input row vector of length 8 with a constant 8x8 coefficient matrix.

- Explain the calculation of Latency of loop, sub loop, and function from Latency/Loop table



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined
dct				-	416	4.160E3	-	73	-	dataflow
read_data				-	66	660.000	-	66	-	no
RD_Loop_Row_RD_Loop_Col				-	64	640.000	2	1	64	yes
Loop_Row_DCT_Loop_proc				-	72	720.000	-	72	-	no
Row_DCT_Loop_DCT_Outer_Loop				-	70	700.000	8	1	64	yes
Loop_Xpose_Row_Outer_Loop_proc				-	67	670.000	-	67	-	no
Xpose_Row_Outer_Loop_Xpose_Row_Inner_Loop				-	65	650.000	3	1	64	yes
Loop_Col_DCT_Loop_proc				-	72	720.000	-	72	-	no
Col_DCT_Loop_DCT_Outer_Loop				-	70	700.000	8	1	64	yes
Loop_Xpose_Col_Outer_Loop_proc				-	67	670.000	-	67	-	no
Xpose_Col_Outer_Loop_Xpose_Col_Inner_Loop				-	65	650.000	3	1	64	yes
write_data				-	67	670.000	-	67	-	no
WR_Loop_Row_WR_Loop_Col				-	65	650.000	3	1	64	yes

Fig.2



Modules & Loops	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined
Row_DCT_Loop	184	1.840E3	23	-	8	no
dct_1d	21	210.000	-	21	-	no
dct_1d_Pipeline_DCT_Outer_Loop	16	160.000	-	16	-	no
DCT_Outer_Loop	14	140.000	8	1	8	yes

Fig.2-1

Calculation of `latency` for function

Take `read_data` function in Fig.2 above for example. Since the `read_data` function in Fig.2 is not pipelined, its `latency` will be equal to its `interval`. From Fig.2 we see that its interval is 66, which is calculated using the `Interval`, `trip count`, and `Iteration latency` of its underling loop as follow:

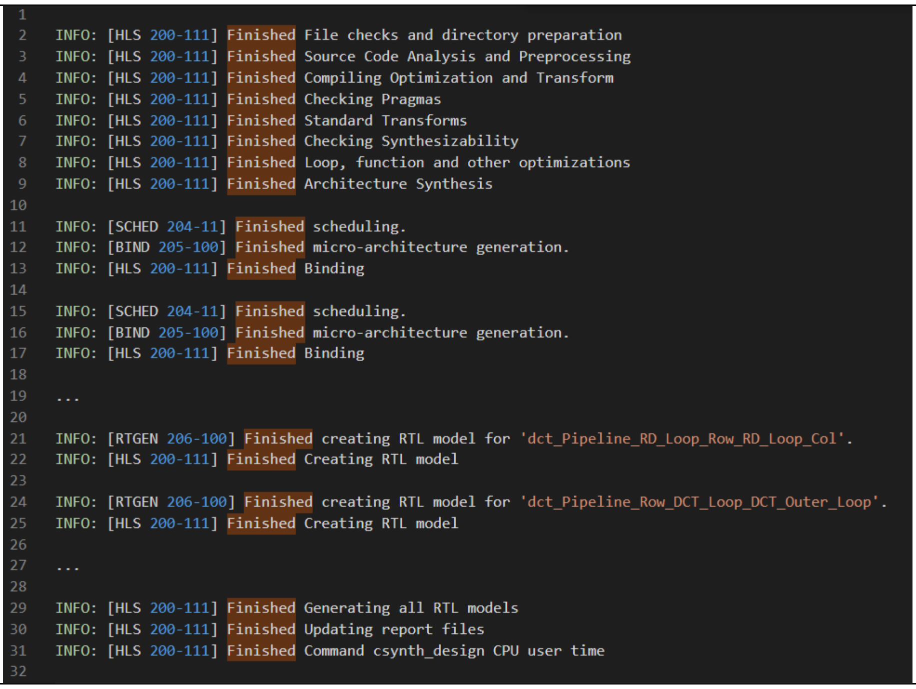
(Interval) * (trip count) + (Iteration latency) = 1 * 64 + 2 = 66

Calculation of `latency` for loop

The latency for the loop `Row_DCT_Loop` is calculated from its own `Iteration latency` and `trip count`:

(Iteration latency) * (trip count) = 23 * 8 = 184

- Examine the synthesis log to list what steps the tool takes during synthesis.

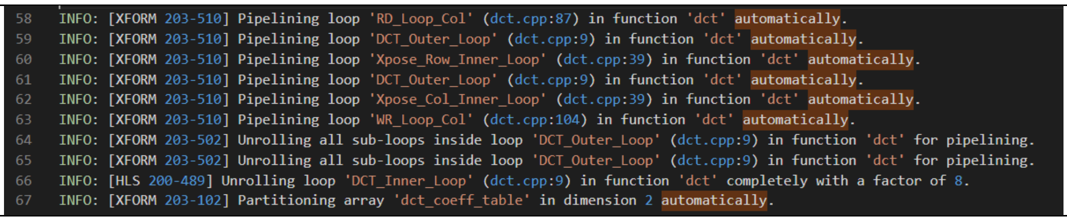


```

INFO: [HLS 200-111] Finished File checks and directory preparation
INFO: [HLS 200-111] Finished Source Code Analysis and Preprocessing
INFO: [HLS 200-111] Finished Compiling Optimization and Transform
INFO: [HLS 200-111] Finished Checking Pragmas
INFO: [HLS 200-111] Finished Standard Transforms
INFO: [HLS 200-111] Finished Checking Synthesizability
INFO: [HLS 200-111] Finished Loop, function and other optimizations
INFO: [HLS 200-111] Finished Architecture Synthesis

INFO: [SCHED 204-11] Finished scheduling.
INFO: [BIND 205-100] Finished micro-architecture generation.
INFO: [HLS 200-111] Finished Binding

INFO: [SCHED 204-11] Finished scheduling.
INFO: [BIND 205-100] Finished micro-architecture generation.
INFO: [HLS 200-111] Finished Binding

...

INFO: [RTGEN 206-100] Finished creating RTL model for 'dct_Pipeline_RD_Loop_Row_RD_Loop_Col'.
INFO: [HLS 200-111] Finished Creating RTL model

INFO: [RTGEN 206-100] Finished creating RTL model for 'dct_Pipeline_Row_DCT_Loop_DCT_Outer_Loop'.
INFO: [HLS 200-111] Finished Creating RTL model

...

INFO: [HLS 200-111] Finished Generating all RTL models
INFO: [HLS 200-111] Finished Updating report files
INFO: [HLS 200-111] Finished Command csynth_design CPU user time

```

Fig.3

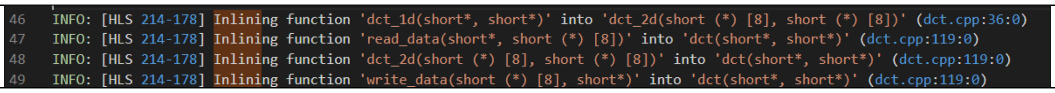
Fig.3 lists all steps the tool takes during C-synthesis. This is extracted from the synthesis log of my solution 1.

- Check the tool automatic perform pipeline, inline?



```
INFO: [XFORM 203-510] Pipelining loop 'RD_Loop_Col' (dct.cpp:87) in function 'dct' automatically.
INFO: [XFORM 203-510] Pipelining loop 'DCT_Outer_Loop' (dct.cpp:9) in function 'dct' automatically.
INFO: [XFORM 203-510] Pipelining loop 'Xpose_Row_Inner_Loop' (dct.cpp:39) in function 'dct' automatically.
INFO: [XFORM 203-510] Pipelining loop 'DCT_Outer_Loop' (dct.cpp:9) in function 'dct' automatically.
INFO: [XFORM 203-510] Pipelining loop 'Xpose_Col_Inner_Loop' (dct.cpp:39) in function 'dct' automatically.
INFO: [XFORM 203-510] Pipelining loop 'WR_Loop_Col' (dct.cpp:104) in function 'dct' automatically.
INFO: [XFORM 203-502] Unrolling all sub-loops inside loop 'DCT_Outer_Loop' (dct.cpp:9) in function 'dct' for pipelining.
INFO: [XFORM 203-502] Unrolling all sub-loops inside loop 'DCT_Outer_Loop' (dct.cpp:9) in function 'dct' for pipelining.
INFO: [HLS 200-489] Unrolling loop 'DCT_Inner_Loop' (dct.cpp:9) in function 'dct' completely with a factor of 8.
INFO: [XFORM 203-102] Partitioning array 'dct_coeff_table' in dimension 2 automatically.
```

Fig.4



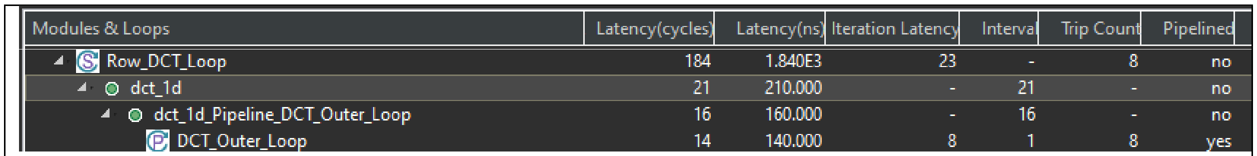
```
INFO: [HLS 214-178] Inlining function 'dct_1d(short*, short*)' into 'dct_2d(short (*) [8], short (*) [8])' (dct.cpp:36:0)
INFO: [HLS 214-178] Inlining function 'read_data(short*, short (*) [8])' into 'dct(short*, short*)' (dct.cpp:119:0)
INFO: [HLS 214-178] Inlining function 'dct_2d(short (*) [8], short (*) [8])' into 'dct(short*, short*)' (dct.cpp:119:0)
INFO: [HLS 214-178] Inlining function 'write_data(short (*) [8], short*)' into 'dct(short*, short*)' (dct.cpp:119:0)
```

Fig.5

From Fig.4 and Fig.5 we see that indeed that tool automatically perform several pipelining and inlinings when I do not specify any directive.

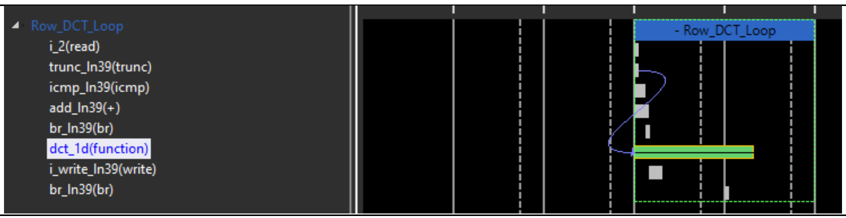
- Explain Schedule Viewer in greater detail, try to understand the operation/variable dependency and its latency.

Following I'll explain the schedule viewer by examples.



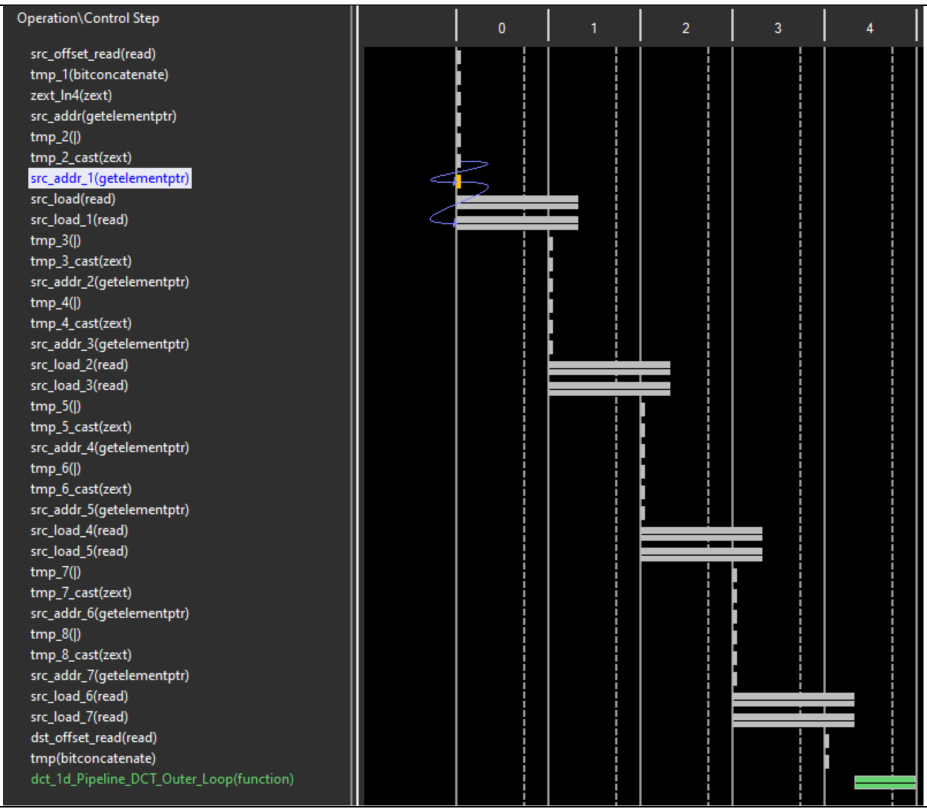
Modules & Loops	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined
Row_DCT_Loop	184	1.840E3	23	-	8	no
dct_1d	21	210.000	-	21	-	no
dct_1d_Pipeline_DCT_Outer_Loop	16	160.000	-	16	-	no
DCT_Outer_Loop	14	140.000	8	1	8	yes

Fig.5-1



```
Row_DCT_Loop
    i_2(read)
    trunc_ln39(trunc)
    icmp_ln39(icmp)
    add_ln39(+)
    br_ln39(br)
    dct_1d(function)
    i_write_ln39(write)
    br_ln39(br)
```

Fig.5-2



```
Operation\Control Step
    src_offset_read(read)
    tmp_1(bitconcatenate)
    zext_ln4(zext)
    src_addr(getelementptr)
    tmp_2(|)
    tmp_2_cast(zext)
    src_addr_1(getelementptr)
    src_load(read)
    src_load_1(read)
    tmp_3(|)
    tmp_3_cast(zext)
    src_addr_2(getelementptr)
    tmp_4(|)
    tmp_4_cast(zext)
    src_addr_3(getelementptr)
    src_load_2(read)
    src_load_3(read)
    tmp_5(|)
    tmp_5_cast(zext)
    src_addr_4(getelementptr)
    tmp_6(|)
    tmp_6_cast(zext)
    src_addr_5(getelementptr)
    src_load_4(read)
    src_load_5(read)
    tmp_7(|)
    tmp_7_cast(zext)
    src_addr_6(getelementptr)
    tmp_8(|)
    tmp_8_cast(zext)
    src_addr_7(getelementptr)
    src_load_6(read)
    src_load_7(read)
    dst_offset_read(read)
    tmp(bitconcatenate)
    dct_1d_Pipeline_DCT_Outer_Loop(function)
```

Fig.5-3: `dct_1d``'s schedule.

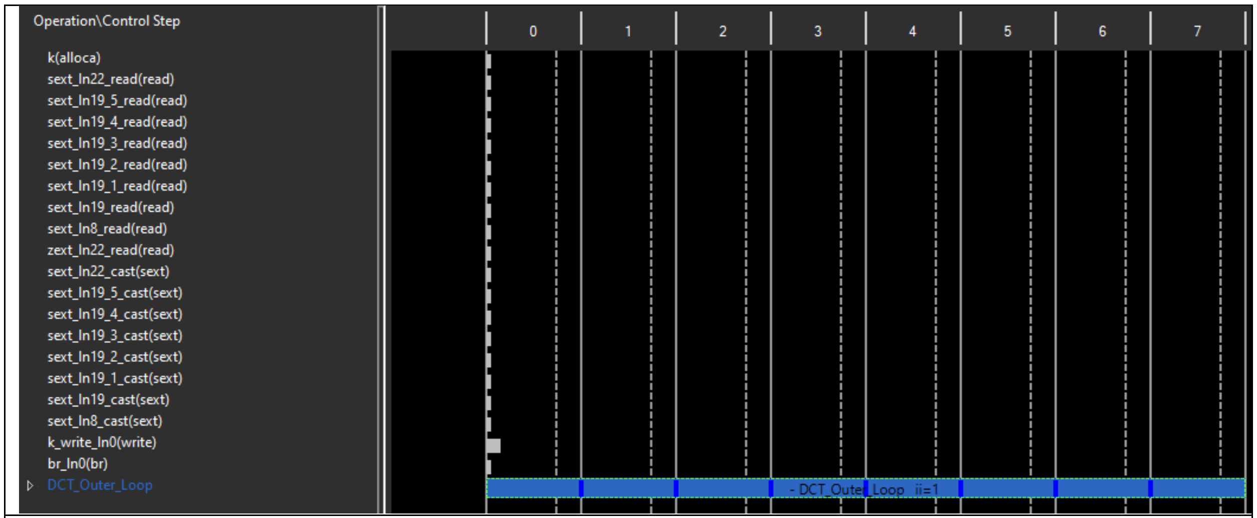


Fig.5-4

Fig.5-1 through Fig.5-4 are taken from solution3.

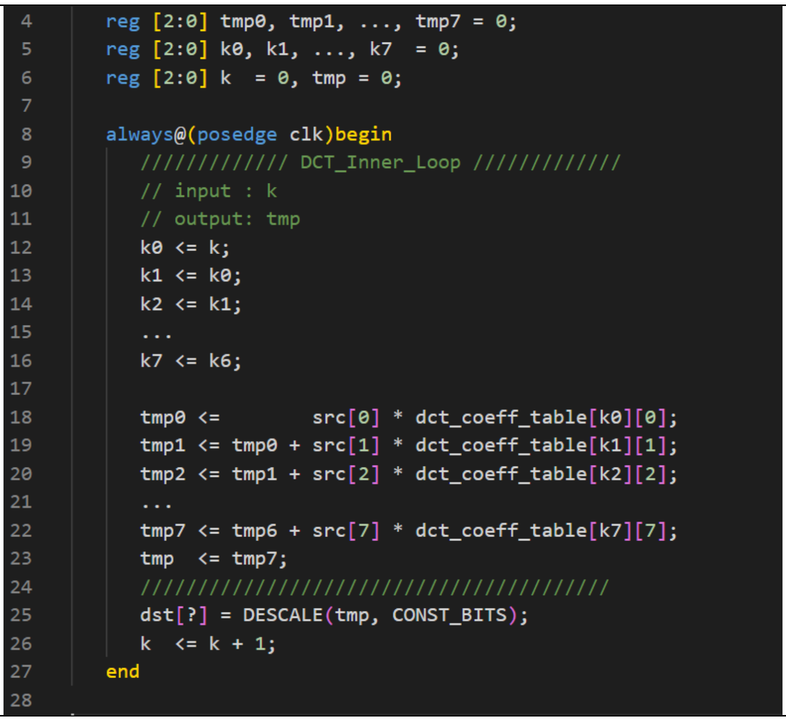
- From Fig.5-1 we see that `DCT_Outer_Loop` has `iteration latency` of 8, and we can get this info from schedule view: from Fig.5-4 we see the blue bar spans over 8 clock cycles from 0 to 7.
- From Fig.5-1 we see that the II of `DCT_Outer_Loop` is 1, and we can get this info from schedule view: from Fig.5-4 we see that on the blue bar it displays the text “ii=1”.
- From Fig.5-1 we see that `dct_1d` has `interval` of 21, and we can get this info from schedule view: from Fig.5-3 we see that `dct_1d` function’s schedule spans 5 clock cycles, and at the last clock cycles, it passes data into a multi-cycle function `dct_1d_Pipeline_DCT_Outer_Loop`, whose `interval` is 16. Therefore, the `interval` of `dct_1d` is 5 + 16 = 21.
- From Fig.5-1 we see that `Row_DCT_Loop` has `iteration latency` of 23, and we can get this info from schedule view: from Fig.5-2 we see that the loop `Row_DCT_Loop` spans over 2 clock cycles, and in the middle it calls the multi-cycle function `dct_1d`, which has `interval` of 21. Therefore, the minimum `interval` (== `iteration latency` for sequential loop) of `Row_DCT_Loop` is 2 + 21 = 23.

- For each optimization, show the observation of deficiency first.

TABLE-2

	Changes made	**Reason**
sol1 -> sol2	Pipeline inner loops for all nested loops	Before one data finished processing, the next data cannot begin -> Most resources are idle.
Sol2 -> sol3	Change `dct_1d` to completely unroll inner loop, and then pipeline outer loops.	Some operations in the outer loop level prevent loop flattening due to data dependency, which makes it no benefit in pipelining only the inner loop.
Sol3 -> sol4	Parallelize memory read by partitioning `buf_2d_in` and `col_inbuf`.	Reading from single 2-port BRAM found to be the bottle neck.
Sol4 -> sol5	Enable inter-functions/loops parallelization by applying dataflow optimization to functions/loops of `dct`.	Same reason as the one in `sol1 -> sol2`, but at functions/loops level.
Sol5 -> sol6	Inline `dct_2d` function into its upper level function `dct`.	Many functions/loops are hidden under `dct_2d` and therefore are not them subject to dataflow optimization applied in previous step. Inlining `dct_2d` makes functions/loops of `dct_2d` to become functions/loops of `dct`, which makes them also subject to dataflow optimization applied in previous step.

- Show why Row_DCT_Loop/COL_DCT_loop cannot do the flatten?

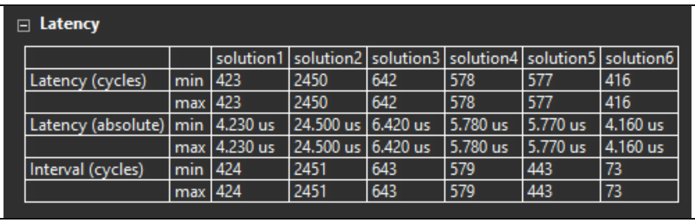


```
4     reg [2:0] tmp0, tmp1, ..., tmp7 = 0;
5     reg [2:0] k0, k1, ..., k7  = 0;
6     reg [2:0] k  = 0, tmp = 0;
7
8     always@(posedge clk)begin
9        //////////// DCT_Inner_Loop ////////////
10       // input : k
11       // output: tmp
12       k0 <= k;
13       k1 <= k0;
14       k2 <= k1;
15       ...
16       k7 <= k6;
17
18       tmp0 <=        src[0] * dct_coeff_table[k0][0];
19       tmp1 <= tmp0 + src[1] * dct_coeff_table[k1][1];
20       tmp2 <= tmp1 + src[2] * dct_coeff_table[k2][2];
21       ...
22       tmp7 <= tmp6 + src[7] * dct_coeff_table[k7][7];
23       tmp  <= tmp7;
24       ////////////////////////////////////////
25       dst[?] = DESCALE(tmp, CONST_BITS);
26       k  <= k + 1;
27    end
28
```

Fig.8

When we only pipeline the inner loop `DCT_Inner_Loop`, the resulting Verilog code would look like the one in Fig.8, where the pipelined `DCT_Inner_Loop` (the region between the '/'-barriers) can now take one new `k` value from outer loop in every clock cycle. Everything outside of `DCT_Inner_Loop` is not pipelined. We now see a problem: line25 will use the output `tmp` from `DCT_Inner_Loop`, but what the `?` in `dst[?]` should be? One correct way is make it `k7 + 3'd1`, but since `k7` in internal to `DCT_Inner_Loop` and it created as a result of performing pipelining on `DCT_Inner_Loop`, so it is not accessible to everything outside of `DCT_Inner_Loop`. To make it possible for line25 to use `k7`, we need to perform pipeline on `DCT_Outer_Loop` instead.

- For each optimization steps, show latency and delay timing.



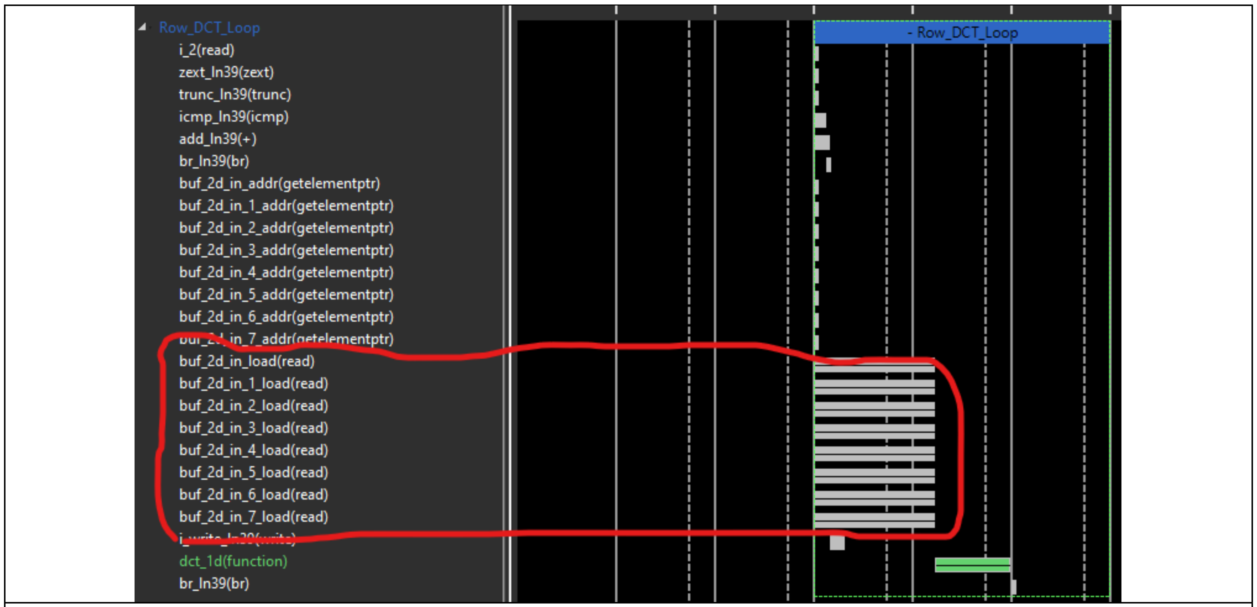
Latency

		solution1	solution2	solution3	solution4	solution5	solution6
Latency (cycles)	min	423	2450	642	578	577	416
	max	423	2450	642	578	577	416
Latency (absolute)	min	4.230 us	24.500 us	6.420 us	5.780 us	5.770 us	4.160 us
	max	4.230 us	24.500 us	6.420 us	5.780 us	5.770 us	4.160 us
Interval (cycles)	min	424	2451	643	579	443	73
	max	424	2451	643	579	443	73

Fig.9

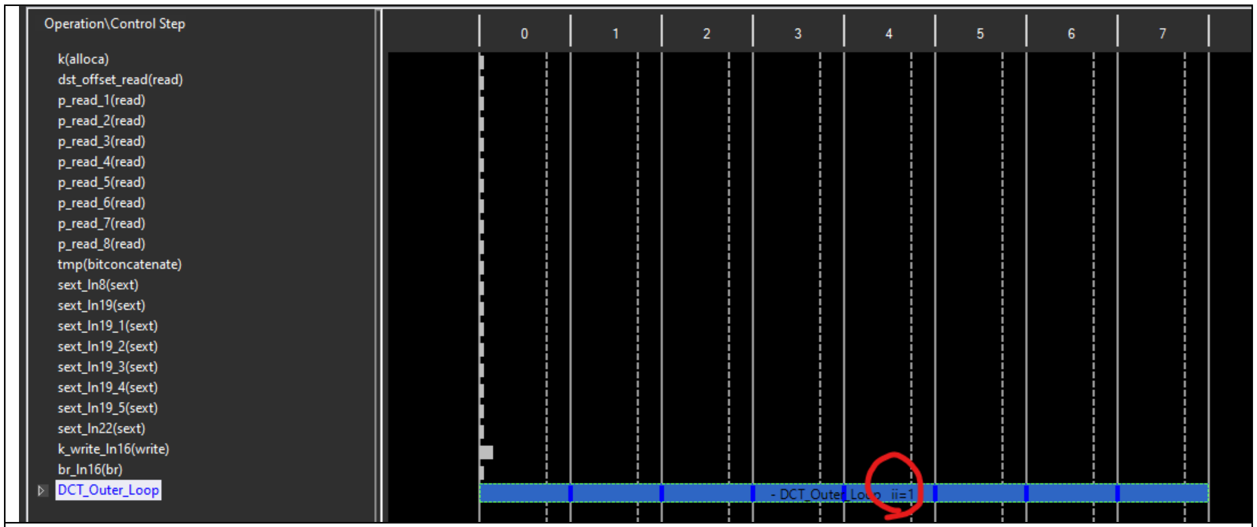
From Fig.9 we see that all solutions have shorter `interval` and `latency` compare with their previous ones except for solution2. The reason why solution1 is so good is that: the tools automatically perform several inlinings and loop flattenings for solution1, who have no any directives applied to it by me yet, but the tool performs not so much automatic inlinings and loop flattenings for other solutions to which I've already applied some directives.

- Show memory block and how array partition helps, show Resource profile, scheduler waveform, how the memory is implemented?



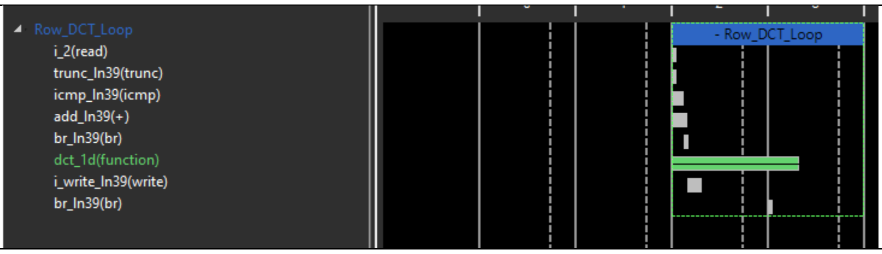
```
Row_DCT_Loop
    i_2(read)
    zext_ln39(zext)
    trunc_ln39(trunc)
    icmp_ln39(icmp)
    add_ln39(+)
    br_ln39(br)
    buf_2d_in_addr(getelementptr)
    buf_2d_in_1_addr(getelementptr)
    buf_2d_in_2_addr(getelementptr)
    buf_2d_in_3_addr(getelementptr)
    buf_2d_in_4_addr(getelementptr)
    buf_2d_in_5_addr(getelementptr)
    buf_2d_in_6_addr(getelementptr)
    buf_2d_in_7_addr(getelementptr)
    buf_2d_in_load(read)
    buf_2d_in_1_load(read)
    buf_2d_in_2_load(read)
    buf_2d_in_3_load(read)
    buf_2d_in_4_load(read)
    buf_2d_in_5_load(read)
    buf_2d_in_6_load(read)
    buf_2d_in_7_load(read)
    i_write_ln39(write)
    dct_1d(function)
    br_ln39(br)
```

Fig.10(a): solution4, with array partition.



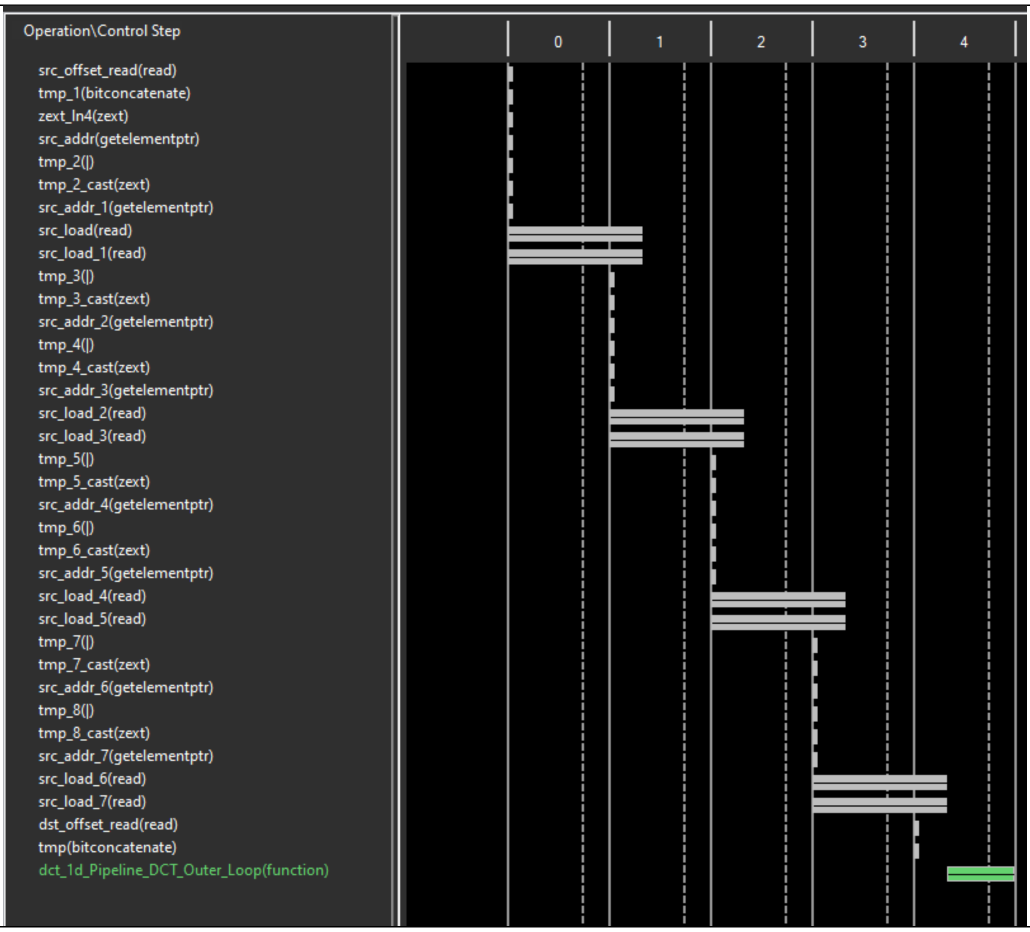
```
Operation\Control Step
    k(alloca)
    dst_offset_read(read)
    p_read_1(read)
    p_read_2(read)
    p_read_3(read)
    p_read_4(read)
    p_read_5(read)
    p_read_6(read)
    p_read_7(read)
    p_read_8(read)
    tmp(bitconcatenate)
    sext_ln8(sext)
    sext_ln19(sext)
    sext_ln19_1(sext)
    sext_ln19_2(sext)
    sext_ln19_3(sext)
    sext_ln19_4(sext)
    sext_ln19_5(sext)
    sext_ln22(sext)
    k_write_ln16(write)
    br_ln16(br)
  DCT_Outer_Loop
```

Fig.10(b): solution4, with array partition.



- Row_DCT_Loop
  - i_2(read)
  - trunc_ln39(trunc)
  - icmp_ln39(icmp)
  - add_ln39(+)
  - br_ln39(br)
  - dct_1d(function)
  - i_write_ln39(write)
  - br_ln39(br)

Fig.11(a): solution3, without array partition.



Operation\Control Step

- src_offset_read(read)
- tmp_1(bitconcatenate)
- zext_ln4(zext)
- src_addr(getelementptr)
- tmp_2(|)
- tmp_2_cast(zext)
- src_addr_1(getelementptr)
- src_load(read)
- src_load_1(read)
- tmp_3(|)
- tmp_3_cast(zext)
- src_addr_2(getelementptr)
- tmp_4(|)
- tmp_4_cast(zext)
- src_addr_3(getelementptr)
- src_load_2(read)
- src_load_3(read)
- tmp_5(|)
- tmp_5_cast(zext)
- src_addr_4(getelementptr)
- tmp_6(|)
- tmp_6_cast(zext)
- src_addr_5(getelementptr)
- src_load_4(read)
- src_load_5(read)
- tmp_7(|)
- tmp_7_cast(zext)
- src_addr_6(getelementptr)
- tmp_8(|)
- tmp_8_cast(zext)
- src_addr_7(getelementptr)
- src_load_6(read)
- src_load_7(read)
- dst_offset_read(read)
- tmp(bitconcatenate)
- dct_1d_Pipeline_DCT_Outer_Loop(function)

Fig.11(b) : solution3, without array partition.

TABLE-1: Compare latencies and interval of `dct_1d` functions of sol3 and sol4.

	Latency	Iteration latency	Interval
Solution3	21		21
Solution4	16		16

Solution4 is derived by applying array partition on soluton3. From Fig.9 we see that both `latency` and `interval` of solution4 are shorter than that of solution3.

From Fig.11(b) we see that there at most can only be two reads (`src_load` operations) in one clock cycle from the `src` array in the function `dct_1d`. To read all 8 data, it takes 4 + 1 clock cycles. The extra 1 cycle comes from the fact that it takes 1 cycle to compute address and 1 cycle to get data from BRAM.

For the case with array partition, from Fig.10(a) we see that the 8 memory reads take place all in 2 clock cycles.

TABLE-1 above shows comparison between the latencies and interval of the `dct_1d` functions of sol3 and sol4. We see that in applying array partition, both latency and interval reduced by 5 cycles.

- In the dataflow optimization, is the three blocks dct_2d, read_data , write_data running in parallel? How do you tell lit?

Yes, since the interval of `dct` is less than the sum of individual latencies for `read_data`, `dct_2d`, `write_data`.

- Run C/RTL co simulation. Introduce Dataflow Graph, and dataflow viewer.

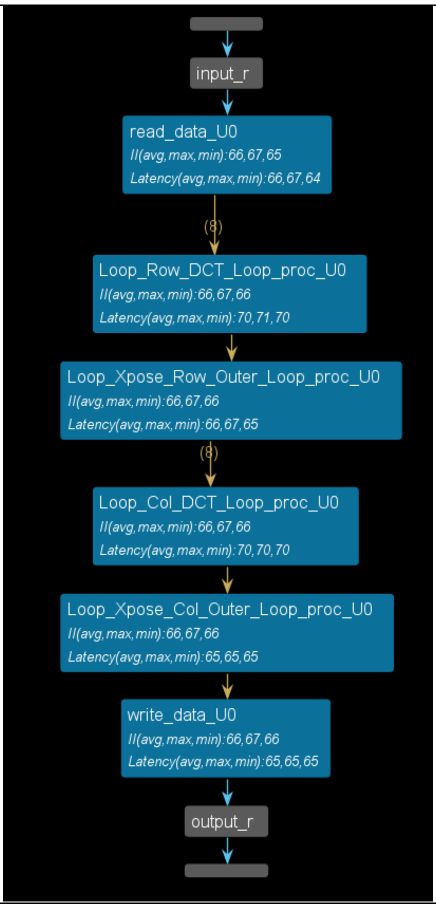
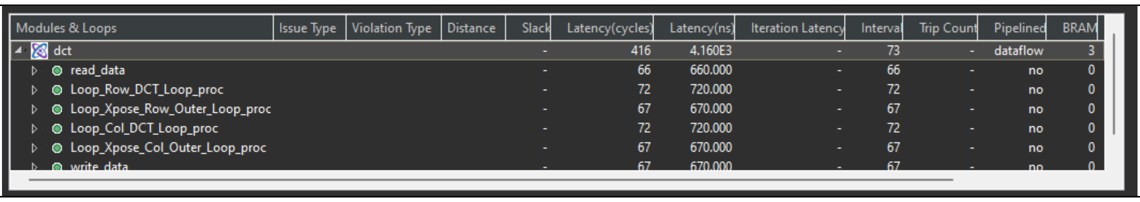

input_r
read_data_U0
II(avg,max,min):66,67,65
Latency(avg,max,min):66,67,64
(8)
Loop_Row_DCT_Loop_proc_U0
II(avg,max,min):66,67,66
Latency(avg,max,min):70,71,70
Loop_Xpose_Row_Outer_Loop_proc_U0
II(avg,max,min):66,67,66
Latency(avg,max,min):66,67,65
(8)
Loop_Col_DCT_Loop_proc_U0
II(avg,max,min):66,67,66
Latency(avg,max,min):70,70,70
Loop_Xpose_Col_Outer_Loop_proc_U0
II(avg,max,min):66,67,66
Latency(avg,max,min):65,65,65
write_data_U0
II(avg,max,min):66,67,66
Latency(avg,max,min):65,65,65
output_r

Fig.12: Data flow graph for solution6.

- Use Dataflow viewer perform throughput analysis, FIFO sizing, channel status.

The throughput determined by `II`, and overall `II` of a system of functions is the maximum of the `II`'s of the al functions on the longest path. From Fig.12 we see that the largest `II` is 67.

- Analyze the final interval achieved? Can it be further optimized?



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM
dct				-	416	4.160E3	-	73	-	dataflow	3
read_data				-	66	660.000	-	66	-	no	0
Loop_Row_DCT_Loop_proc				-	72	720.000	-	72	-	no	0
Loop_Xpose_Row_Outer_Loop_proc				-	67	670.000	-	67	-	no	0
Loop_Col_DCT_Loop_proc				-	72	720.000	-	72	-	no	0
Loop_Xpose_Col_Outer_Loop_proc				-	67	670.000	-	67	-	no	0
write_data				-	67	670.000	-	67	-	no	0

Fig.13

The final interval achieve is 73, as shown in Fig.13.

- Github Link

https://github.com/Ri-chard-Wu/AAHLS