Politecnico di Torino

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA ELETTRONICA

INTEGRATED SYSTEMS ARCHITECTURE

Laboratory 2

Bernunzo Angela 198721
Busignani Fabio 197883
Gianoglio Emanuele 200090

February 2, 2014

Contents

Introduction

The aim of this laboratory is to design an optimized ASIP for the Discrete Cosine Transform starting from the given C code. In order to do that, we used **TCE**, which is a *TTA based Co-design Enviroment*.
TTA (Transport Trigger Architecture) is a kind of architecture that works only with move operations. The computation made by the TTA functional unit starts only when the input set as a trigger is filled through a move operation.
Just before starting with the DCT problem, we followed the available tutorial based on the CRC algorithm to understand the whole functionality of TCE.
The starting point is the basic algorithm without any optimization (Sec.1).
Exploiting the TCE tools we will evaluate the number of cycles required to complete the computation of the algorithm using the minimal architecture available, which describes a minimalistic architecture containing just enough resources that the TCE compiler can still compile program for it.
After that, we will optimize the algorithm and the architecture in order to reduce the number of cycles required to complete the computation. The last step is to generate the VHDL code for the implemented architecture, simulate and synthesize it for a 100 MHz target clock frequency.

1 Starting Point Architecture

Starting with prode, the graphic tool used to design the architecture for the ASIP, we can explore the minimal starting point architecture (Fig.1).

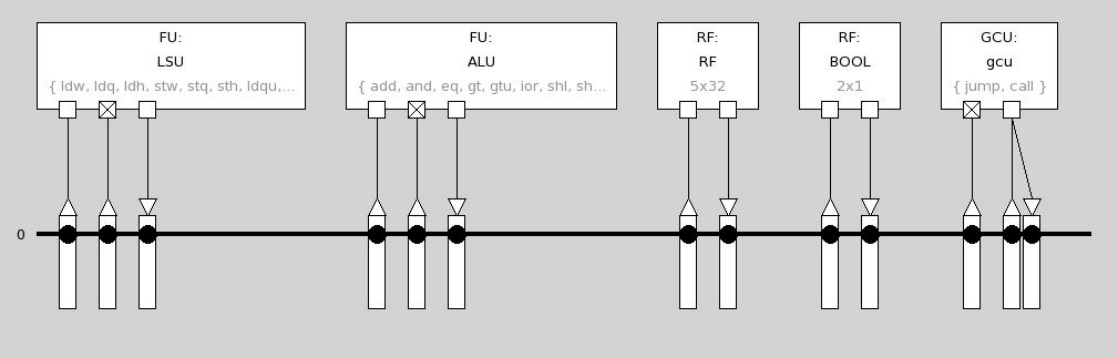

Figure 1: Starting point architecture

It consists in an architecture with one ALU, one RF, one boolean RF, one LSU and one GCU. Each of them is connected to the other by one single bus. Through the TCE compiler it is created the program (.tpef file) that can be executed on the available architecture, in this particular case the program consists in the DCT algorithm written in the available C files.
Now it is possible to simulate the execution of the generated program on the architecture, and to discover how many cycles are needed to complete it. This is done on the command line shell and the result is that the program takes 21299 cycles to be executed (Fig.2).

```
[isa4@isa DCT1]$ tcecc -O3 -a start.adf -o dct.tpef -k y dct.c main.c
[isa4@isa DCT1]$ ttasim -a start.adf -p dct.tpef

(ttasim) info proc cycles
21299
```

Figure 2: Starting number of cycles

2 Accelerating the Algorithm

In order to achieve an improvement of the speed, and thus a reduction of the required number of cycles,we add a custom operation in the DCT algorithm. Analyzing the algorithm, we chose to customize the lifting operations.
First of all we created the new operation module named lifting in the Operation Set Editor. It contains all the operations that we want to customize: LIFTPI81, LIFTPI82, LIFTPI161, LIFTPI162, LIFT3PI161 and LIFT3PI162.

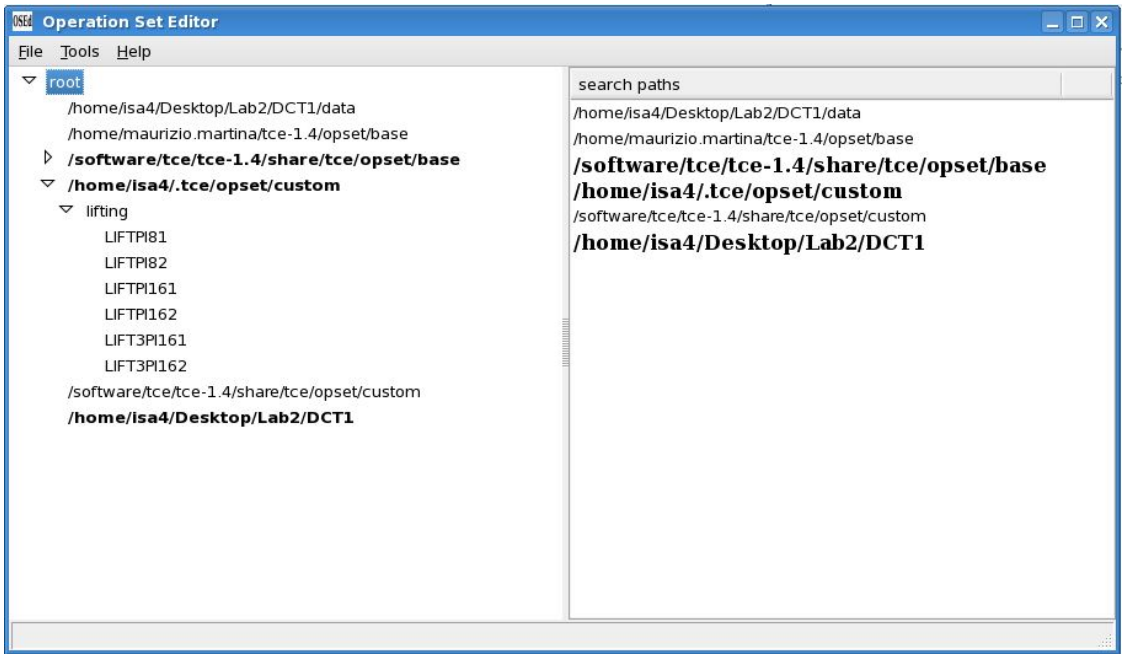

Figure 3: Operation Set Editor

In order to permit to simulate a program that use these custom operations with the TCE processor simulator we added, for each operation, the simulation behavior model as follow:

```
/**
 * OSAL behavior definition file.
 */

#include "OSAL.hh"
#include "dct.h"
```

```
7
  OPERATION(LIFTPI81)
9 TRIGGER

11 sample_t x1= INT(1);
   sample_t x2= INT(2);
13 sample_t res=0;
   sample_t temp=0;
15
   temp= (x2*L1_8) >> S1_8;
17 res= x1+temp;

19 IO(3)=static_cast<signed>(res);

21 return true;
   END_TRIGGER;
23 END_OPERATION(LIFTPI81);


25
   OPERATION(LIFTPI82)
27 TRIGGER

29 sample_t x1= INT(1);
   sample_t x2= INT(2);
31 sample_t res=0;
   sample_t temp=0;
33
   temp= (x1*L2_8) >> S2_8;
35 res= x2-temp;

37 IO(3)=static_cast<signed>(res);

39 return true;
   END_TRIGGER;
41 END_OPERATION(LIFTPI82);


43
   OPERATION(LIFTPI161)
45 TRIGGER

47 sample_t x1= INT(1);
   sample_t x2= INT(2);
49 sample_t res=0;
   sample_t temp=0;
51
   temp= (x2*L1_16) >> S1_16;
53 res= x1+temp;

55 IO(3)=static_cast<signed>(res);
```

```
57 return true;
   END_TRIGGER;
59 END_OPERATION(LIFTPI161);


61
   OPERATION(LIFTPI162)
63 TRIGGER

65 sample_t x1= INT(1);
   sample_t x2= INT(2);
67 sample_t res=0;
   sample_t temp=0;
69
   temp= (x1*L2_16) >> S2_16;
71 res= x2-temp;

73 IO(3)=static_cast<signed>(res);

75 return true;
   END_TRIGGER;
77 END_OPERATION(LIFTPI162);


79
   OPERATION(LIFT3PI161)
81 TRIGGER

83 sample_t x1= INT(1);
   sample_t x2= INT(2);
85 sample_t res=0;
   sample_t temp=0;
87
   temp= (x2*L1_316) >> S1_316;
89 res= x1+temp;

91 IO(3)=static_cast<signed>(res);

93 return true;
   END_TRIGGER;
95 END_OPERATION(LIFT3PI161);


97
   OPERATION(LIFT3PI162)
99 TRIGGER

101 sample_t x1= INT(1);
    sample_t x2= INT(2);
103 sample_t res=0;
    sample_t temp=0;
105
    temp= (x1*L2_316) >> S2_316;
```

```
107 res= x2−temp;

109 IO(3)=static_cast<signed>(res);

111 return true;
    END_TRIGGER;
113 END_OPERATION(LIFT3PI162);
```

This code describes the behavior of each operation present inside the module Lifting. The code of the behavior is taken from the C source file of the DCT algorithm. Now we have the complete set of custom operation in the operation set database.
The further step is to create a new functional unit in the processor architecture. The starting point is the previous mentioned minimal architecture, in which we add one functional unit. It will be able to execute our custom operations. The new functional unit must have two inputs and one output accordingly to the definition of the lifting operations. This functional unit is still a TTA, so that we set one input as trigger.
We add the LIFTING functional unit to the architecture as shown in the figure, and we fully connect the architecture in order to connect the added functional unit to the rest of the processor.

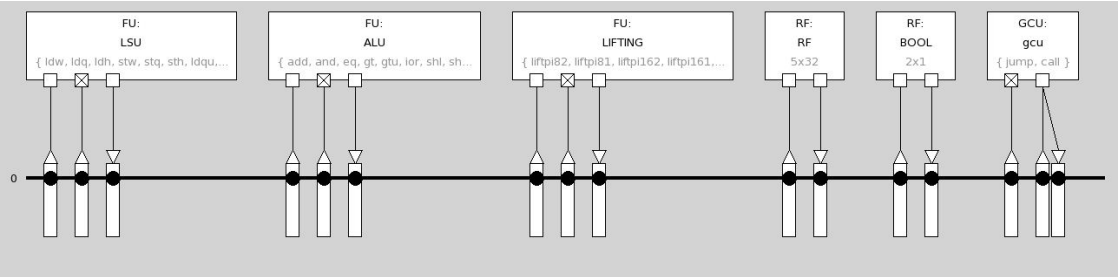

Figure 4: Accelerating architecture with functional unit **Lifting**

In order to get some benefit from this customized architecture, we have to modify the C code to exploit the added custom operations during the execution of the program. Following the example of the given tutorial we modify the code using the macros of the custom operations. The result of this modification can be seen in the *dct_custom.c* file.
Now we compile the code and we simulate it on the new customized architecture. To do that we follow the same procedure used with the starting point architecture and algorithm. Through the TCE compiler we generate the program which will be executed on the custom architecture and we simulate it.
The result is an heavly improvement in terms of cycles: 990 cycles instead of 21299 without custom operations.

During this step we also enabled the bus trace, it means that the simulator writes on a text file the bus values of the processor from every cycle of execution. This data will be used in order to verify the correctness of the RTL implementation of the processor, which is the next step.

```
[isa4@isa DCT1]$ tcecc -O3 -a custom.adf -o dct_custom.tpef -k y dct_custom.c main.c
[isa4@isa DCT1]$ ttasim -a custom.adf -p dct_custom.tpef

(ttasim) info proc cycles
990
```

Figure 5: Number of cycles after accelerating

3 VHDL Implementation

Before generating the VHDL of the whole processor, we have to add the hardware description of the custom functional unit in the hardware database (HDB) of TCE. In order to do that we use the HDBEditor and we add our functional unit, named lifting, in the database. We associate its VHDL description and we assign the opcodes for each operation involved in the algorithm as shown in the table.

OPERATION	OPCODE
LIFT3PI161	0
LIFT3PI162	1
LIFTPI161	2
LIFTPI162	3
LIFTPI81	4
LIFTPI82	5

The VHDL code of the lifting functional unit is available at the end of the report.
Now it is possible to generate the VHDL code for the entire architecture. We do that through prode tool, that is able to generate the HDL implementation of the graphical drawn processor. In particular it create three directories: the first contains the VHDL of the functional units and register files, the second contains the interconnection network and the third contains the testbench files.
The last step before simulating the processor, is to create instruction memory and data memory starting from the program generated by the compiler when we used the TCE simulator (.tpef file). This is done by command line and the result is the file .img that contains the instruction memory image of the processor.

4 Simulation and Verification

Now everything is done and the last step of the implementation of the processor can be performed. We have the VHDL of the processor and the testbench, thus we can simulate it. In order to do that we use the GHDL simulator and, after having modified the given file .sh with the correct number of test cycles, we compile and simulate the architecture. The bus trace is still enabled and the simulation writes a new file in which there is the conent of the bus in every clock cycle. Comparing this file with the previous one generated during the architecture simulation, we obtain that the trace are equal, this means that the RTL generation is completed succesfully and everything is ready for the synthesis.

5 Increasing Performance

A further step can be done in order to improve the performance of the custom architecture. The processor that we are using is minimalistic, thus, by adding resources, we can improve its performance and we are able to speed up the execution of our algorithm.
We can achieve the wanted results for example by adding transport buses.
In our case we try to add 3 buses and see what happen. After having added the

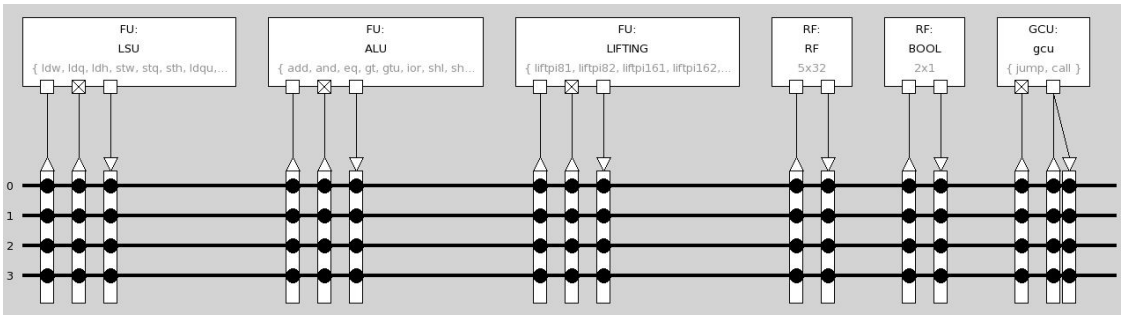

Figure 6: Architecture with four buses

additional buses we simulated the architecture and the results is a reduction in terms of process cycles: 428 instead of 990.

```
[isa4@isa DCT1]$ ttasim -a modified_4_bus.adf -p dct_custom.tpef

(ttasim) info proc cycles
428
```

Figure 7: Number of cycles after added three buses

Moreover we may be interested in the processor utilization statistics for what concern the operations, so that we will be able to improve this statistics by adding resources like RFs or ALUs.

```
operations:

ADD             42.757% (183 executions)
CALL            0.46729% (2 executions)
JUMP            0.233645% (1 executions)
LDW             24.7664% (106 executions)
LIFT3PI161      0.46729% (2 executions)
LIFT3PI162      0.233645% (1 executions)
LIFTPI161       0.46729% (2 executions)
LIFTPI162       0.233645% (1 executions)
LIFTPI81        1.40187% (6 executions)
LIFTPI82        0.700935% (3 executions)
SHL             1.86916% (8 executions)
STW             16.3551% (70 executions)
SUB             3.73832% (16 executions)
```

Figure 8: Statistical of operations

Let's see in the case of the 4 bus optimization that there are a lot of load and store operations. This can be optimized by adding one additional RF in the architecture.

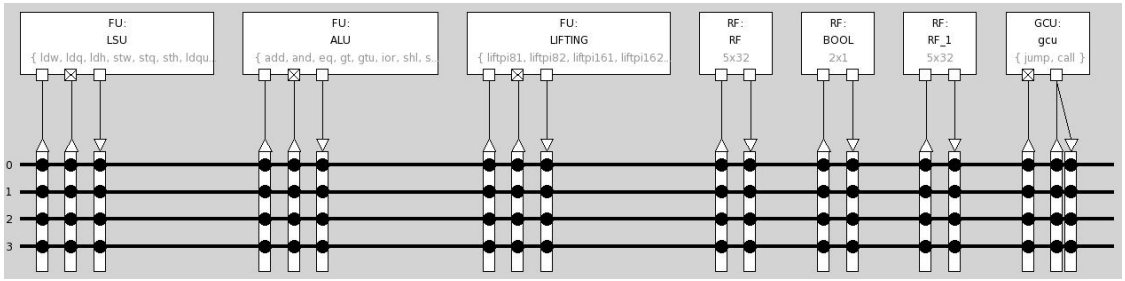

Figure 9: Architecture with an additional register file

We have done this optimization and we performed another time the simulation in order to check if there is an improvement both in the cycles and in the statistics. The result is good, in fact we obtained 163 cycles instead of 428, and the number of load and store operations is strongly reduced.

```
[isa4@isa DCT1]$ ttasim -a modified_RF.adf -p dct_custom.tpef

(ttasim) info proc cycles
163
```

Figure 10: Number of cycles with an additional register file

```
operations:

ADD             44.1718% (72 executions)
CALL            1.22699% (2 executions)
JUMP            0.613497% (1 executions)
LDW             23.3129% (38 executions)
LIFT3PI161      1.22699% (2 executions)
LIFT3PI162      0.613497% (1 executions)
LIFTPI161       1.22699% (2 executions)
LIFTPI162       0.613497% (1 executions)
LIFTPI81        3.68098% (6 executions)
LIFTPI82        1.84049% (3 executions)
SHL             4.90798% (8 executions)
STW             16.5644% (27 executions)
SUB             9.81595% (16 executions)
```

Figure 11: Statistical of operations with an additional register file

Load operations are reduced from 106 to 38, while store operations are reduced from 70 to 27. Another possible optimization is to add an ALU that performs addition operation. In fact, if we see at the statistics in the case of the architecture with one single ALU, we notice that the addition operation takes a large amount of percentage on the total usage (75%).

```
operations executed in function units:

LSU:
LDW             58.4615% of FU total (38 executions)
STW             41.5385% of FU total (27 executions)
TOTAL           39.8773% (65 triggers)

ALU:
ADD             75% of FU total (72 executions)
SHL             8.33333% of FU total (8 executions)
SUB             16.6667% of FU total (16 executions)
TOTAL           58.8957% (96 triggers)

LIFTING:
LIFTPI81        40% of FU total (6 executions)
LIFTPI82        20% of FU total (3 executions)
LIFTPI161       13.3333% of FU total (2 executions)
LIFTPI162       6.66667% of FU total (1 executions)
LIFT3PI161      13.3333% of FU total (2 executions)
LIFT3PI162      6.66667% of FU total (1 executions)
TOTAL           9.20245% (15 triggers)

gcu:
JUMP            33.3333% of FU total (1 executions)
CALL            66.6667% of FU total (2 executions)
TOTAL           1.84049% (3 triggers)
```

Figure 12: Statistical of units

Now let's see what happen with the dedicated ALU.

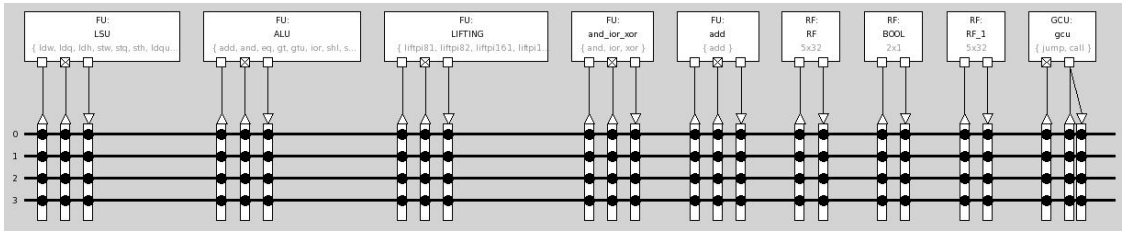

Figure 13: Architecture with an additional register file

```
operations executed in function units:

LSU:
LDW             58.4615% of FU total (38 executions)
STW             41.5385% of FU total (27 executions)
TOTAL           44.5205% (65 triggers)

ALU:
ADD             38.4615% of FU total (15 executions)
SHL             20.5128% of FU total (8 executions)
SUB             41.0256% of FU total (16 executions)
TOTAL           26.7123% (39 triggers)

LIFTING:
LIFTPI81        40% of FU total (6 executions)
LIFTPI82        20% of FU total (3 executions)
LIFTPI161       13.3333% of FU total (2 executions)
LIFTPI162       6.66667% of FU total (1 executions)
LIFT3PI161      13.3333% of FU total (2 executions)
LIFT3PI162      6.66667% of FU total (1 executions)
TOTAL           10.274% (15 triggers)

add:
ADD             100% of FU total (57 executions)
TOTAL           39.0411% (57 triggers)

gcu:
JUMP            33.3333% of FU total (1 executions)
CALL            66.6667% of FU total (2 executions)
TOTAL           2.05479% (3 triggers)
```

Figure 14: Statistical of units with an additional register file

We notice that the percentage of the addition in the ALU is strongly reduced (from 75% to 38%), and thus we can imagine that this optimization further speeds up our processor. Looking at the cycle count we see as expected that the number of cycles is reduced: from 163 to 146.

```
[isa4@isa DCT1]$ ttasim -a modified_ALU.adf -p dct_custom.tpef

(ttasim) info proc cycles
146
```

Figure 15: Number of cycles with an additional register file

With these optimizations we understand that by adding resources at the processor we are able to speed up the execution of the algorithm. Of course there is an overhead in terms of the area of the whole circuit.

6 Synthesis

Before starting with the synthesis, we regenerated the VHDL code exploiting prode tool, but this time without bustrace generation. The generated VHDL refers to the architecture with only the custom functional unit and not with the all added resources when we tried to speed up the execution.
Now all the files of the architecture we need for the synthesis are available and we can start working in the Synopsys environment.
First we analyze and elaborate the .vhdl file and then we are able to apply the required constraints: the working frequency must be 100 MHz. In order to do that we create the clock signal with period of 10 ns and we bind it to the clock pin in our architecture.
We launch the synthesis and we save the results in two files. The first is the timing report, in which we can check if our architecture is feasible with the clock signal imposed, and the second is the area report, to have an idea of the occupied area of our network.
From the timing report we see that there is a positive slack, it means that the 100 MHz clock signal works correctly and all data arrive correctly to the the destinations in the 10 ns clock period. We see that the slack is very little, only

```
clock MYCLK (rise edge)                                 10.00      10.00
clock network delay (ideal)                              0.00      10.00
clock uncertainty                                       -0.07       9.93
fu_LIFTING/reg_out_reg[31]/CK (DFFSX2)                   0.00       9.93 r
library setup time                                      -0.03       9.90
data required time                                                  9.90
--------------------------------------------------------------------------
data required time                                                  9.90
data arrival time                                                  -9.89
--------------------------------------------------------------------------
slack (MET)                                                         0.01
```

Figure 16: Timing report

0.01 ns, but it is sufficient to accept the imposed clock frequency, moreover we can speed up the system until the slack is equal to 0, then, the maximum achievable clock frequency is:

$$\frac{10^9}{10.00 - 0.01} = 100.1\ MHz.$$

s not much more than the set frequency but that is. For what concern the area we see from the report that the total area occupied by our architecture is about 2340884 μm^2.

```
Library(s) Used:

    fast (File: /home/maurizio.martina/libtsmc/synopsys/fast.db)

Number of ports:                187
Number of nets:                3382
Number of cells:               3113
Number of references:           162

Combinational area:       24113.174382
Noncombinational area:    13105.814281
Net Interconnect area:    2303665.250000

Total cell area:          37218.988281
Total area:               2340884.238281
```

Figure 17: Area Report

7 Place and Route

The last step of this laboratory is to perform the place and route operations with Encounter. We start importing the design as described in the available tutorial, and following the steps, we first set the area needed for our architecture by structuring the floorplan.
The further step is to create the power supply stripes and to distribute it around the whole chip.
At this point we can place the cells of our architecture through the command Place, so that each cell has its own position inside the chip.
Before completing the design with the routing step, we set the clock tree following the instructions.
Now we are ready to complete the design. We create the connections among the cells through the commands TrialRoute and NanoRoute and then we extract the timing and geometry informations.

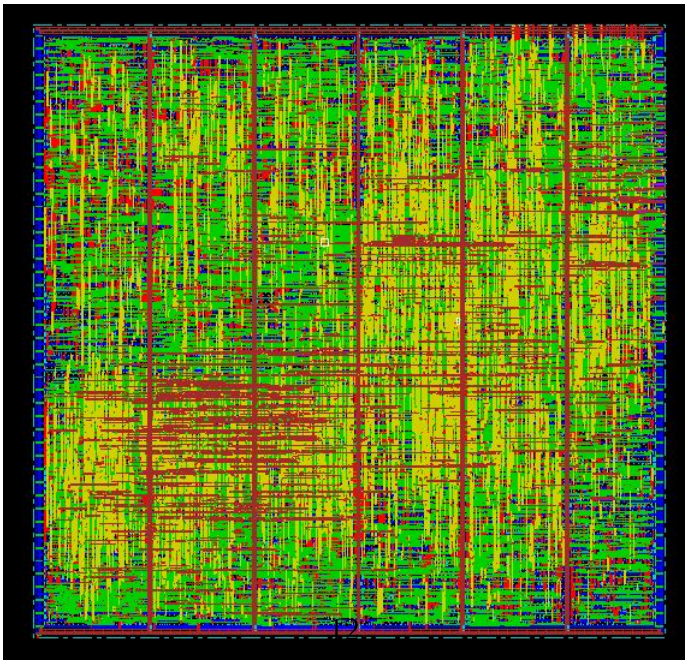

Figure 18: Place and route

```
Summary report for Module: Top Cell
============================================

============== Design Statistics============
         Number of Pins:      16856
      Number of IO Pins:        187
         Number of Nets:       4707
Average Pins Per Net (Signal): 3.5810e+00

============== Chip Utilization============
              Core Size: 5.3166e+04 um^2
              Chip Size: 5.7016e+04 um^2
  Effective Utilization: 1.0874e+00
    Number of Cell Rows:         90

================ Module Information===========
           No. of Cells: 10915
             No. of IOs: 187
             Total Area: 5.701570e+04 um^2
Total Clock Wire Length: NA

================= Wire Info ==================
                           Internal   External
            No. of nets:       4490        186
     No. of connections:      11389       1163
   Total net length (X): 7.2751e+04 um 3.1983e+03 um
   Total net length (Y): 7.0014e+04 um 1.3201e+03 um
       Total net length: 1.4276e+05 um 4.5185e+03 um
```

Figure 19: Area Report

From the results, we notice that there are no violation both in timing and in geometry. The results in terms of area are shown in the following figure.

A VHDL

A.1 Lifting

```
library IEEE;
use IEEE.std_logic_1164.all;

package lift_opcodes is

  constant OPC_3PI161 : std_logic_vector(2 downto 0) := "000";
  constant OPC_3PI162 : std_logic_vector(2 downto 0) := "001";
  constant OPC_PI161: std_logic_vector(2 downto 0) := "010";
  constant OPC_PI162: std_logic_vector(2 downto 0) := "011";
  constant OPC_PI81 : std_logic_vector(2 downto 0) := "100";
  constant OPC_PI82 : std_logic_vector(2 downto 0) := "101";



end lift_opcodes;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
--use IEEE.numeric_std.all; --per la funzione shift_right
use work.lift_opcodes.all;

entity lifting is
  generic (
    busw : integer := 32);
  port(
    in1  : in std_logic_vector (busw-1 downto 0);
    in2  : in std_logic_vector (busw-1 downto 0);
    t1opcode  : in std_logic_vector(2 downto 0);
    in1_load : in std_logic;
    in2_load : in std_logic;
    rst : in std_logic;
    clk  : in std_logic;
    glock : in std_logic;
    result : out std_logic_vector (busw-1 downto 0)
  );
end lifting;


architecture behavior of lifting is

constant L1_8  : integer := 51;
constant S1_8  : integer := 8;
```

```
44 constant L2_8  : integer := 98;
   constant S2_8  : integer := 8;
46 constant L1_16  : integer := 25;
   constant S1_16  : integer := 8;
48 constant L2_16  : integer := 50;
   constant S2_16 : integer := 8;
50 constant L1_316 : integer := 78;
   constant S1_316 : integer := 8;
52 constant L2_316 : integer := 142;
   constant S2_316 : integer := 8;
54
   signal reg_out : std_logic_vector(busw-1 downto 0); -- register out
56
   begin -- rtl
58 regs: process(clk,rst)
     variable mult : signed(2*busw-1 downto 0);
60   variable reg_x1,reg_x2,tmp,reg_out_tmp : signed(busw-1 downto 0);
     begin
62     if rst = '0' then
         reg_out <= (others => '0');
64     elsif clk'event and clk = '1' then
         if glock = '0' then
66         if in2_load = '1' then
             reg_x2 := signed(in2);
68         end if;
           if in1_load = '1' then
70           reg_x1 := signed(in1);
           end if;
72
           case t1opcode is
74           when OPC_PI81 =>
               mult := reg_x2 * conv_signed(L1_8,busw);
76             tmp := signed(sxt(std_logic_vector(mult(busw-1 downto 8)),busw));
           reg_out_tmp := reg_x1 + tmp;
78         when OPC_PI82 =>
               mult := reg_x1 * conv_signed(L2_8,busw);
80             tmp := signed(sxt(std_logic_vector(mult(busw-1 downto 8)),busw));

82         reg_out_tmp := reg_x2 - tmp;
           when OPC_PI161 =>
84           mult := reg_x2 * conv_signed(L1_16,busw);
             tmp := signed(sxt(std_logic_vector(mult(busw-1 downto 8)),busw));
86           reg_out_tmp := reg_x1 + tmp;
           when OPC_PI162 =>
88           mult := reg_x1 * conv_signed(L2_16,busw);
             tmp := signed(sxt(std_logic_vector(mult(busw-1 downto 8)),busw));
90           reg_out_tmp := reg_x2 - tmp;
           when OPC_3PI161 =>
92           mult := reg_x2 * conv_signed(L1_316,busw);
             tmp := signed(sxt(std_logic_vector(mult(busw-1 downto 8)),busw));
```

```
94              reg_out_tmp := reg_x1 + tmp;
              when OPC_3PI162 =>
96              mult := reg_x1 * conv_signed(L2_316,busw);
                tmp := signed(sxt(std_logic_vector(mult(busw-1 downto 8)),busw));
98              reg_out_tmp := reg_x2 - tmp;
              when others => null;
100         end case;
            reg_out <= conv_std_logic_vector(reg_out_tmp,busw);
102       end if;
        end if;
104   end process regs;

106   result <= reg_out;

108 end behavior;
```

B C

B.1 Custom DCT

```
#include "dct.h"
2 #include "tceops.h"
/// Compute a WHT step
4 ///\param x1 first value
///\param x2 second value
6 ///\param f choose add/subtract
///\return WHT step result
8 static sample_t wht_step(sample_t x1, sample_t x2, int f)
{
10   if (f == 0)
       return x1+x2;
12   else
       return x1-x2;
14 }

16 /// Compute the first lifting step of the pi/8 rotation
///\param x1 first value
18 ///\param x2 second value
static sample_t lift_pi8_1(sample_t x1, sample_t x2)
20 {
   sample_t tmp;
22
   tmp = (x2*L1_8) >> S1_8;
24
   return x1+tmp;
26 }

28 /// Compute the second lifting step of the pi/8 rotation
```

```
///\param x1 first value
///\param x2 second value
static sample_t lift_pi8_2(sample_t x1, sample_t x2)
{
  sample_t tmp;

  tmp = (x1*L2_8) >> S2_8;

  return x2-tmp;
}

/// Compute the first lifting step of the pi/16 rotation
///\param x1 first value
///\param x2 second value
static sample_t lift_pi16_1(sample_t x1, sample_t x2)
{
  sample_t tmp;

  tmp = (x2*L1_16) >> S1_16;

  return x1+tmp;
}

/// Compute the second lifting step of the pi/16 rotation
///\param x1 first value
///\param x2 second value
static sample_t lift_pi16_2(sample_t x1, sample_t x2)
{
  sample_t tmp;

  tmp = (x1*L2_16) >> S2_16;

  return x2-tmp;
}

/// Compute the first lifting step of the 3pi/16 rotation
///\param x1 first value
///\param x2 second value
static sample_t lift_3pi16_1(sample_t x1, sample_t x2)
{
  sample_t tmp;

  tmp = (x2*L1_316) >> S1_316;

  return x1+tmp;
}

/// Compute the second lifting step of the 3pi/16 rotation
///\param x1 first value
///\param x2 second value
```

```
static sample_t lift_3pi16_2(sample_t x1, sample_t x2)
{
  sample_t tmp;

  tmp = (x1*L2_316) >> S2_316;

  return x2-tmp;
}

/// Compute the WHT
///\param *x input buffer pointer
///\param *y output buffer pointer
void wht(volatile sample_t *x, sample_t *y)
{
  /// temporary buffer
  sample_t ytmp[2][8];
  /// old data pointer
  sample_t *yold;
  /// new data pointer
  sample_t *ynew;
  /// temporary buffer index
  int yidx;

  /// index
  int i;

  /// level in the WHT butterfly structure
  int level;
  /// current block in level
  int block;
  /// number of blocks per current level
  int block_number;
  /// block offset
  int offset;
  /// displacement of the second input/output respect to the first one
  int m;

  /// values initialization
  m = N/2;
  yidx=0;

  /// buffer initialization
  yold = ytmp[yidx];
  ynew = ytmp[yidx^1];

  /// copy input data in the local buffer
  for (i=0; i<N; i++)
    yold[i] = x[i] << FBITS;

  /// compute the WHT in a butterfly fashion
```

```
  for (level=0; level<LOG2N; level++)
  {
    /// numer of blocks in current level is 2^level
    block_number = 1 << level;
    offset = 0;
    for (block=0; block<block_number; block++)
    {
      for (i=0; i<m; i++)
      {
  /// compute one butterfly
  ynew[i+offset] = wht_step(yold[i+offset], yold[i+offset+m], 0);
  ynew[i+offset+m] = wht_step(yold[i+offset], yold[i+offset+m], 1);
      }
      /// update the block offset
      offset += (m << 1);
    }
    /// exchange buffers
    yidx ^=1;
    yold = ytmp[yidx];
    ynew = ytmp[yidx^1];
    /// update the displacement
    m >>= 1;
  }

  /// copy results on output buffer
  for (i=0; i<N; i++)
    y[i] = yold[i];

#ifdef DEBUG
  printf("WHT results\n");
  for (i=0; i<N; i++)
    printf("%d\n", yold[i]);
#endif

}

/// Compute the Lifting Steps
///\param *x input buffer pointer
///\param *y output buffer pointer
void ls(sample_t *x, volatile sample_t *y)
{
  /// temporary results
  sample_t ytmp1;
  sample_t ytmp2;
  sample_t ytmp3;
  sample_t ytmp4;

#ifdef DEBUG
  /// index
  int i;
```

```
#endif

  /// Walsh and B order
  /// 0, 3, 6, 5, 4, 7, 2, 1

  /// not altered
  y[0] = x[0];
  y[4] = x[3];

  /// \pi/8 lifting steps
  ytmp1 = x[6];
  ytmp2 = x[5];
_TCE_LIFTPI81(ytmp1,ytmp2,ytmp1);
  //ytmp1 = lift_pi8_1(ytmp1, ytmp2);
_TCE_LIFTPI82(ytmp1,ytmp2,ytmp2);
  //ytmp2 = lift_pi8_2(ytmp1, ytmp2);
_TCE_LIFTPI81(ytmp1,ytmp2,ytmp1);
  //ytmp1 = lift_pi8_1(ytmp1, ytmp2);
  y[2] = ytmp1;
  y[6] = ytmp2;

  /// \pi/8 lifting steps
  /// \pi/16 lifting steps
  /// 3\pi/16 lifting steps
  ytmp1 = x[4];
  ytmp2 = x[2];
_TCE_LIFTPI81(ytmp1,ytmp2,ytmp1);
  //ytmp1 = lift_pi8_1(ytmp1, ytmp2);
_TCE_LIFTPI82(ytmp1,ytmp2,ytmp2);
  //ytmp2 = lift_pi8_2(ytmp1, ytmp2);
_TCE_LIFTPI81(ytmp1,ytmp2,ytmp1);
  //ytmp1 = lift_pi8_1(ytmp1, ytmp2);

  ytmp3 = x[7];
  ytmp4 = x[1];
_TCE_LIFTPI81(ytmp3,ytmp4,ytmp3);
  //ytmp3 = lift_pi8_1(ytmp3, ytmp4);
_TCE_LIFTPI82(ytmp3,ytmp4,ytmp4);
  //ytmp4 = lift_pi8_2(ytmp3, ytmp4);
_TCE_LIFTPI81(ytmp3,ytmp4,ytmp3);
  //ytmp3 = lift_pi8_1(ytmp3, ytmp4);

_TCE_LIFTPI161(ytmp1,ytmp4,ytmp1);
  //ytmp1 = lift_pi16_1(ytmp1, ytmp4);
_TCE_LIFTPI162(ytmp1,ytmp4,ytmp4);
  //ytmp4 = lift_pi16_2(ytmp1, ytmp4);
_TCE_LIFTPI161(ytmp1,ytmp4,ytmp1);
  //ytmp1 = lift_pi16_1(ytmp1, ytmp4);

_TCE_LIFT3PI161(ytmp2,ytmp3,ytmp2);
```

```
  // ytmp2 = lift_3pi16_1(ytmp2, ytmp3);
_TCE_LIFT3PI162(ytmp2,ytmp3,ytmp3);
  // ytmp3 = lift_3pi16_2(ytmp2, ytmp3);
_TCE_LIFT3PI161(ytmp2,ytmp3,ytmp2);
  // ytmp2 = lift_3pi16_1(ytmp2, ytmp3);

  y[1] = ytmp1;
  y[3] = ytmp2;
  y[5] = ytmp3;
  y[7] = ytmp4;

#ifdef DEBUG
  printf("DCT_results\n");
  for (i=0; i<N; i++)
    printf("%d\n", y[i]);
#endif

}
```