Politecnico di Torino
III Facoltà di Ingegneria

Design of an ASIP with TCE Integrated Systems Architectures

Master degree in Electronic Engineering

Group: 02

Alessi Valeria 198141
Renzi Alessandro 197783
Tiralongo Antonio 200021

February 5, 2014

Contents

CHAPTER 1

Introduction

A transport triggered architecture (TTA) is a kind of CPU design in which programs directly control the internal transport buses of a processor. Computation happens as a side effect of data transports: writing data into a triggering port of a functional unit triggers the functional unit to start a computation. Due to its modular structure, TTA is an ideal processor template for Application Specific Instruction-set Processors (ASIP) with customized datapath, but without the inflexibility and design cost of fixed function hardware accelerators.

Typically a transport triggered processor has multiple transport buses and multiple functional units connected to the buses, which provides opportunities for instruction level parallelism. This is why the TTA architecture resembles the Very Long Instruction Word (VLIW) architecture. A TTA instruction word is composed of multiple slots, one slot per bus, and each slot determines the data transport that takes place on the corresponding bus.

The goal of this lab is to design with TCE an Application Specific Instruction-set Processor (ASIP) to compute the Discrete Cosine Transform (DCT).

CHAPTER 2

Starting Point Architecture

The initial architecture is composed of:

- One BUS
- One ALU
- One LSU
- One RF
- One BOOL RF

The initial benchmark of the DCT algorithm execution on this architecture (performed using *Proxim*) reported the following data:

```
cycles: 21299

buses:

B1              79.9052% (17019 writes)

sockets:

lsu_i1          14.4796% (3084 writes)
lsu_o1          9.43706% (2010 writes)
lsu_i2          5.04249% (1074 writes)
RF_i1           10.5404% (2245 writes)
RF_o1           23.48% (5001 writes)
bool_i1         1.49303% (318 writes)
gcu_i1          1.64796% (351 writes)
gcu_i2          0.0751209% (16 writes)
gcu_o1          0.154937% (33 writes)
```

```
ALU_i1          23.311% (4965 writes)
ALU_i2          23.3156% (4966 writes)
ALU_o1          24.8134% (5285 writes)

operations executed in function units:

LSU:
LDW             65.1751% of FU total (2010 executions)
STW             34.8249% of FU total (1074 executions)
TOTAL           14.4796% (3084 triggers)

ALU:
ADD             55.5086% of FU total (2756 executions)
AND             6.10272% of FU total (303 executions)
EQ              6.40483% of FU total (318 executions)
SHL             12.3666% of FU total (614 executions)
SHR             6.40483% of FU total (318 executions)
SHRU            6.10272% of FU total (303 executions)
SUB             1.00705% of FU total (50 executions)
XOR             6.10272% of FU total (303 executions)
TOTAL           23.311% (4965 triggers)

gcu:
JUMP            95.1567% of FU total (334 executions)
CALL            4.8433% of FU total (17 executions)
TOTAL           1.64796% (351 triggers)


operations:

ADD             12.9396% (2756 executions)
AND             1.4226% (303 executions)
CALL            0.079816% (17 executions)
EQ              1.49303% (318 executions)
JUMP            1.56815% (334 executions)
LDW             9.43706% (2010 executions)
SHL             2.88276% (614 executions)
SHR             1.49303% (318 executions)
SHRU            1.4226% (303 executions)
STW             5.04249% (1074 executions)
SUB             0.234753% (50 executions)
XOR             1.4226% (303 executions)
```

```
FU port guard accesses:

register accesses:

RF:
0               3104 reads,    0 guard reads,      66 writes
1               60 reads,      0 guard reads,      46 writes
2               403 reads,     0 guard reads,      706 writes
3               1007 reads,    0 guard reads,      1000 writes
4               427 reads,     0 guard reads,      427 writes
TOTAL 5 registers used

BOOL:
0               0 reads,       1152 guard reads,   318 writes
TOTAL 1 registers used
```

As we can see from the profiling information, there are a lot of transactions on the system bus, the majority of which are between the ALU and the register file. So a first attempt to improve the performances was to add **a system bus** and **three more registers** in the register file.

CHAPTER 3

Algorithm Acceleration

The proposed modifications lead to the following performances:

```
cycles: 5303

buses:

B1              61.7198% (3273 writes)
B2              96.1908% (5101 writes)

sockets:

lsu_i1          4.69546% (249 writes)
lsu_o1          2.82859% (150 writes)
lsu_i2          1.86687% (99 writes)
RF_i1           35.2065% (1867 writes)
RF_o1           56.3832% (2990 writes)
bool_i1         5.99661% (318 writes)
gcu_i1          6.61889% (351 writes)
gcu_i2          0.301716% (16 writes)
gcu_o1          0.622289% (33 writes)
ALU_i1          51.6123% (2737 writes)
ALU_i2          51.6123% (2737 writes)
ALU_o1          51.6689% (2740 writes)

operations executed in function units:

LSU:
LDW             60.241% of FU total (150 executions)
STW             39.759% of FU total (99 executions)
TOTAL           4.69546% (249 triggers)
```

```
ALU:
ADD             19.8027% of FU total (542 executions)
AND             11.0705% of FU total (303 executions)
EQ              11.6186% of FU total (318 executions)
SHL             22.4333% of FU total (614 executions)
SHR             11.6186% of FU total (318 executions)
SHRU            11.0705% of FU total (303 executions)
SUB             1.31531% of FU total (36 executions)
XOR             11.0705% of FU total (303 executions)
TOTAL           51.6123% (2737 triggers)

gcu:
JUMP            95.1567% of FU total (334 executions)
CALL            4.8433% of FU total (17 executions)
TOTAL           6.61889% (351 triggers)


operations:

ADD             10.2206% (542 executions)
AND             5.71375% (303 executions)
CALL            0.320573% (17 executions)
EQ              5.99661% (318 executions)
JUMP            6.29832% (334 executions)
LDW             2.82859% (150 executions)
SHL             11.5784% (614 executions)
SHR             5.99661% (318 executions)
SHRU            5.71375% (303 executions)
STW             1.86687% (99 executions)
SUB             0.678861% (36 executions)
XOR             5.71375% (303 executions)

FU port guard accesses:

register accesses:

RF:
0               251 reads,      0 guard reads,      35 writes
1               45 reads,       0 guard reads,      46 writes
2               67 reads,       0 guard reads,      67 writes
3               963 reads,      0 guard reads,      347 writes
4               640 reads,      0 guard reads,      352 writes
5               361 reads,      0 guard reads,      359 writes
```

```
6               632 reads,      0 guard reads,      631 writes
7               31 reads,       0 guard reads,      30 writes
TOTAL 8 registers used

BOOL:
0               0 reads,        288 guard reads,    318 writes
TOTAL 1 registers used
```

The speedup from the starting architecture is **400**%. The second bus implies a more efficient transfer between the ALU and the register file, while the added register contributes to lower the transfers from and to the memory. Due to the different architecture, the usage percentage of the different functional units is now biased toward the ALU, in fact at this time it manages half of the operations executed on the processor. In particular there are a lot of additions and shifts, that are the core computation of the DCT algorithm. So it seems to be straightforward to add a **second ALU** which implements these two operations in parallel to the first.
The benchmark result is the following:

```
cycles: 4970

buses:

B1              72.7968% (3618 writes)
B2              95.5533% (4749 writes)

sockets:

lsu_i1          5.01006% (249 writes)
lsu_o1          3.01811% (150 writes)
lsu_i2          1.99195% (99 writes)
RF_i1           37.4245% (1860 writes)
RF_o1           60.0201% (2983 writes)
bool_i1         6.39839% (318 writes)
gcu_i1          7.06237% (351 writes)
gcu_i2          0.321932% (16 writes)
gcu_o1          0.663984% (33 writes)
ALU_i1          32.2535% (1603 writes)
ALU_i2          32.2535% (1603 writes)
ALU_o1          32.2938% (1605 writes)
ALU2_i1         22.8169% (1134 writes)
ALU2_i2         22.8169% (1134 writes)
ALU2_o          22.837% (1135 writes)
```

```
operations executed in function units:

LSU:
LDW             60.241% of FU total (150 executions)
STW             39.759% of FU total (99 executions)
TOTAL           5.01006% (249 triggers)

ALU:
ADD             1.24766% of FU total (20 executions)
AND             18.9021% of FU total (303 executions)
EQ              19.8378% of FU total (318 executions)
SHL             0.124766% of FU total (2 executions)
SHR             19.8378% of FU total (318 executions)
SHRU            18.9021% of FU total (303 executions)
SUB             2.24579% of FU total (36 executions)
XOR             18.9021% of FU total (303 executions)
TOTAL           32.2535% (1603 triggers)

ALU2:
ADD             46.0317% of FU total (522 executions)
SHL             53.9683% of FU total (612 executions)
TOTAL           22.8169% (1134 triggers)

gcu:
JUMP            95.1567% of FU total (334 executions)
CALL            4.8433% of FU total (17 executions)
TOTAL           7.06237% (351 triggers)


operations:

ADD             10.9054% (542 executions)
AND             6.09658% (303 executions)
CALL            0.342052% (17 executions)
EQ              6.39839% (318 executions)
JUMP            6.72032% (334 executions)
LDW             3.01811% (150 executions)
SHL             12.3541% (614 executions)
SHR             6.39839% (318 executions)
SHRU            6.09658% (303 executions)
STW             1.99195% (99 executions)
SUB             0.724346% (36 executions)
XOR             6.09658% (303 executions)
```

```
FU port guard accesses:

register accesses:

RF:
0               251 reads,     0 guard reads,      35 writes
1               45 reads,      0 guard reads,      46 writes
2               69 reads,      0 guard reads,      69 writes
3               964 reads,     0 guard reads,      348 writes
4               637 reads,     0 guard reads,      349 writes
5               356 reads,     0 guard reads,      353 writes
6               631 reads,     0 guard reads,      631 writes
7               30 reads,      0 guard reads,      29 writes
TOTAL 8 registers used

BOOL:
0               0 reads,       1440 guard reads,   318 writes
TOTAL 1 registers used
```

This time the speedup with respect to the previous version is only **1%**. The explanation has to be searched into the DCT algorithm. If we analyze the code, we find that in the sequence of calls to the lift operations there are a lot of data dependencies:

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

  ytmp3 = x[7];
```

```
ytmp4 = x[1];
ytmp3 = lift_pi8_1(ytmp3, ytmp4);
ytmp4 = lift_pi8_2(ytmp3, ytmp4);
ytmp3 = lift_pi8_1(ytmp3, ytmp4);

ytmp1 = lift_pi16_1(ytmp1, ytmp4);
ytmp4 = lift_pi16_2(ytmp1, ytmp4);
ytmp1 = lift_pi16_1(ytmp1, ytmp4);

ytmp2 = lift_3pi16_1(ytmp2, ytmp3);
ytmp3 = lift_3pi16_2(ytmp2, ytmp3);
ytmp2 = lift_3pi16_1(ytmp2, ytmp3);
```

This data dependency between two calls prevents to exploit the improved parallelism of the architecture, therefore it is useless to add more functional units, since the code would be in any case unable to exploit them. So the best option to further improve the performances is to implement in hardware the lifting operations in order to reduce the time needed for their execution. The question now is: is it possible or convenient?

CHAPTER 4

Creating the Custom Operations

Looking at the **lifting operations** implementation we can see that they are very similar to each other. There is always a multiplication by a constant, a shift by eight, and an addition or a subtraction. It is very likely that the large number of additions found in the benchmarks are caused by these multiplications, so it should be useful to add an hardware multiplier in order to execute the multiplications in a few clock cycles, instead of an expensive software loop. Moreover, since a constant hardware shift comes barely for free it seems to be another good way to speedup the functions.
Now we have to check if these hypothesis are true; to do so we must simulate again the execution, but this time adding the behavior of the new module, in charge of implementing the lifting operations. This is done introducing a new module in the architecture with the tool *OSEd* (Operation Set Editor), then we add to this module all the operations we want to simulate. These operations will become part of the instruction set of the ASIP.
Once the new instructions have been added, we have to define their simulation behavior. The simulation behavior is specified in C language, supported by a specific library (OSAL.hh). The behavior for the instructions is the following:

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

#include "OSAL.hh"

OPERATION(LIFT_PI8_1)
        TRIGGER
        int x1 = INT(1);
        int x2 = INT(2);
        int result = 0;
```

```
        result = x1 + ((x2*51) >> 8);

        IO(3) = result;

        END_TRIGGER
END_OPERATION(LIFT_PI8_1)


OPERATION(LIFT_PI8_2)
        TRIGGER
        int x1 = INT(1);
        int x2 = INT(2);
        int result = 0;

        result = x2 - ((x1*98) >> 8);

        IO(3) = result;

        END_TRIGGER
END_OPERATION(LIFT_PI8_2)


OPERATION(LIFT_PI16_1)
        TRIGGER
        int x1 = INT(1);
        int x2 = INT(2);
        int result = 0;

        result = x1 + ((x2*25) >> 8);

        IO(3) = result;

        END_TRIGGER
END_OPERATION(LIFT_PI16_1)


OPERATION(LIFT_PI16_2)
        TRIGGER
        int x1 = INT(1);
        int x2 = INT(2);
        int result = 0;

        result = x2 - ((x1*50) >> 8);
```

```
        IO(3) = result;

        END_TRIGGER
END_OPERATION(LIFT_PI16_2)


OPERATION(LIFT_3PI16_1)
        TRIGGER
        int x1 = INT(1);
        int x2 = INT(2);
        int result = 0;

        result = x1 + ((x2*78) >> 8);

        IO(3) = result;

        END_TRIGGER
END_OPERATION(LIFT_3PI16_1)


OPERATION(LIFT_3PI16_2)
        TRIGGER
        int x1 = INT(1);
        int x2 = INT(2);
        int result = 0;

        result = x2 - ((x1*142) >> 8);

        IO(3) = result;

        END_TRIGGER
END_OPERATION(LIFT_3PI16_2)
```

Within this step we also have to specify a latency for the new instructions, but since we have not yet synthesized the processor we don't know it at the moment, so we can only make a raw estimation of this latency. For now we can take three clock cycles as a reasonable latency.
Then we compile the behavior into a plugin module that the simulator can call whenever one of these instructions has to be executed. Once the behavior has been compiled, the module has to be added to the architecture as a functional unit and connected to the buses. This is done using *ProDe* (Processor Design) tool.

CHAPTER 5

Custom operations in C code

In order to force the code to use the custom instruction, instead of the traditional assembly code, based on the basic instruction set, we have to re-implement the lifting functions using a specific syntax, defined into a library provided with TCE. Basically, we threat the new instructions as functions to be called with some arguments, which are the data to be provided to the input ports of the functional unit. The following code contains the modified lifting functions:

```
/// Compute the first lifting step of the pi/8 rotation
///\param x1 first value
///\param x2 second value
static sample_t lift_pi8_1(sample_t x1, sample_t x2)
{
  sample_t output;
  _TCE_LIFT_PI8_1(x1, x2, output);

  return output;
}

/// 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 output;
  _TCE_LIFT_PI8_2(x1, x2, output);

  return output;
}

/// 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 output;
  _TCE_LIFT_PI16_1(x1, x2, output);

  return output;
}

/// 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 output;
  _TCE_LIFT_PI16_2(x1, x2, output);

  return output;
}

/// 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 output;
  _TCE_LIFT_3PI16_1(x1, x2, output);

  return output;
}

/// 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 output;
  _TCE_LIFT_3PI16_2(x1, x2, output);

  return output;
}
```

At this point it is possible to perform the simulation to verify the performances of the new architecture. The simulation report is the following:

```
cycles: 400

buses:

B1              92.5% (370 writes)
B2              56.5% (226 writes)

sockets:

lsu_i1          26.5% (106 writes)
lsu_o1          15.25% (61 writes)
lsu_i2          11.25% (45 writes)
RF_i1           34.25% (137 writes)
RF_o1           62.25% (249 writes)
gcu_i1          0.75% (3 writes)
gcu_i2          0.25% (1 writes)
gcu_o1          0.75% (3 writes)
ALU_i1          34.25% (137 writes)
ALU_i2          34.25% (137 writes)
ALU_o1          35% (140 writes)
LIFTER_i1       3.75% (15 writes)
LIFTER_i2       3.75% (15 writes)
LIFTER_o1       4.5% (18 writes)

operations executed in function units:

LSU:
LDW             57.5472% of FU total (61 executions)
STW             42.4528% of FU total (45 executions)
TOTAL           26.5% (106 triggers)

ALU:
ADD             82.4818% of FU total (113 executions)
SHL             5.83942% of FU total (8 executions)
SUB             11.6788% of FU total (16 executions)
TOTAL           34.25% (137 triggers)

LIFTER:
LIFT_PI8_1      40% of FU total (6 executions)
LIFT_PI8_2      20% of FU total (3 executions)
LIFT_PI16_1     13.3333% of FU total (2 executions)
```

```
LIFT_PI16_2     6.66667% of FU total (1 executions)
LIFT_3PI16_1    13.3333% of FU total (2 executions)
LIFT_3PI16_2    6.66667% of FU total (1 executions)
TOTAL           3.75% (15 triggers)

gcu:
JUMP            33.3333% of FU total (1 executions)
CALL            66.6667% of FU total (2 executions)
TOTAL           0.75% (3 triggers)


operations:

ADD             28.25% (113 executions)
CALL            0.5% (2 executions)
JUMP            0.25% (1 executions)
LDW             15.25% (61 executions)
LIFT_3PI16_1    0.5% (2 executions)
LIFT_3PI16_2    0.25% (1 executions)
LIFT_PI16_1     0.5% (2 executions)
LIFT_PI16_2     0.25% (1 executions)
LIFT_PI8_1      1.5% (6 executions)
LIFT_PI8_2      0.75% (3 executions)
SHL             2% (8 executions)
STW             11.25% (45 executions)
SUB             4% (16 executions)

FU port guard accesses:

register accesses:

RF:
0               91 reads,      0 guard reads,      5 writes
1               0 reads,       0 guard reads,      1 writes
2               27 reads,      0 guard reads,      27 writes
3               35 reads,      0 guard reads,      27 writes
4               32 reads,      0 guard reads,      28 writes
5               31 reads,      0 guard reads,      25 writes
6               15 reads,      0 guard reads,      8 writes
7               18 reads,      0 guard reads,      16 writes
TOTAL 8 registers used

BOOL:
```

```
TOTAL 0 registers used
```

As we can see the total number of operations has dramatically dropped thanks to the new instructions. Now the total execution time is only **400 clock cycles**, which means 14 times faster than the previous architecture and 84 times faster than the initial one.

CHAPTER 6

Functional Unit implementation

In order to be able to generate the VHDL code of the processor, we must provide an implementation for our custom functional unit.
This implementation is composed of a package, which contains the constants representing the opcodes for each instruction that the functional unit is able to perform, and an entity in which there is the actual implementation.

```
package lifter_opcodes is

   constant LIFT_3PI16_1  : std_logic_vector(2 downto 0) := "000";
   constant LIFT_3PI16_2  : std_logic_vector(2 downto 0) := "001";
   constant LIFT_PI16_1   : std_logic_vector(2 downto 0) := "010";
   constant LIFT_PI16_2   : std_logic_vector(2 downto 0) := "011";
   constant LIFT_PI8_1    : std_logic_vector(2 downto 0) := "100";
   constant LIFT_PI8_2    : std_logic_vector(2 downto 0) := "101";

end lifter_opcodes;
```

The entity must be compliant with a given interface stated by the TTA architecture. There must be a generic parameter *dataw*, which will contain the bus width. For each data input port there must be a load signal which will trigger the input flip-flop. One of the input ports must be a trigger port, with a further input, the opcode.

```
entity lifter is
   generic (dataw: integer := 32);

   port (
   t1data  : in  std_logic_vector(dataw-1 downto 0);
   t1load  : in  std_logic;
   t1opcode : in std_logic_vector(2 downto 0);
```

```
   o1data  : in std_logic_vector(dataw-1 downto 0);
   o1load  : in std_logic;

   r1data  : out std_logic_vector(dataw-1 downto 0);

   clk     : in std_logic;
   rstx    : in std_logic;
   glock   : in std_logic
   );
end lifter;
```

The functional unit architecture is the following: there is a flip-flop on each input port, triggered by the corresponding external signal, then, depending on the opcode, one of the two input ports is selected as one operand for the multiplier. The other operand *mul_B* is selected between several constants, one for each opcode. Part of the process in charge of selecting the second operand of the multiplier is shown below.

```
case t1opcode is
   when LIFT_PI8_1 =>
      mul_B <= conv_std_logic_vector(51, dataw);
      add_sub <= '0';

   when LIFT_PI8_2 =>
      mul_B <= conv_std_logic_vector(98, dataw);
      add_sub <= '1';

   when LIFT_PI16_1 =>
      mul_B <= conv_std_logic_vector(25, dataw);
      add_sub <= '0';

   when LIFT_PI16_2 =>
      mul_B <= conv_std_logic_vector(50, dataw);
      add_sub <= '1';

   when LIFT_3PI16_1 =>
      mul_B <= conv_std_logic_vector(78, dataw);
      add_sub <= '0';

   when LIFT_3PI16_2 =>
      mul_B <= conv_std_logic_vector(142, dataw);
      add_sub <= '1';
```

```
   when others =>
      null;
end case;
```

The output of this multiplier *mul_OUT* is shifted by eight and sent as second operand to an adder (add/sub). The shifting operation is performed in the pipeline register, that is placed between the multiplier and the adder, to cut the critical path. The eight least significant bits are discarded and the final result is stored in *mul_reg*.

```
pipe_mul_reg: process (clk, rstx)
   begin -- process regs
      if rstx = '0' then                -- asynchronous reset (active low)
         mul_reg <= (others => '0');
      elsif clk'event and clk = '1' then -- rising clock edge
         if glock = '0' then
            mul_reg <= mul_OUT(dataw-1+8 downto 8);
         end if;
      end if;
   end process pipe_mul_reg;

add_B <= mul_reg;
```

The first operand of the adder *add_A* is the input port which was not selected as an operand for the multiplier. The second operand *add_B*, coming from the multiplier, will be added or subtracted to the first one depending on the opcode. In fact, in the process used to select the right constant for the multiplier, we also set a control signal *add_sub*, that will decide which kind of operation has to be executed.

```
proc_add: process(add_sub, add_A, add_B)
   begin
      if add_sub = '0' then
         add_OUT <= add_A + add_B;
      else
         add_OUT <= add_A - add_B;
      end if;
   end process proc_add;
```

Then, the result will be sent to the output buffer *r1reg*, that is directly connected to the output port *r1data*.
Before generating the processor, the functional unit implementation has to be added to the Hardware Database, using the tool *HDBEditor*. Here we have to specify the name of the module, the opcodes, the ports and the VHDL file. At this point the module can be used in the processor generation.

CHAPTER 7

Generating the processor

Once the functional unit has been added to the Hardware Database, it is possible to generate the VHDL implementation of the whole processor, ready to be synthesized. This can be done using the tool *ProGe* (Processor Generator). It will take as input the processor architecture in the *adf* format and a file in the *idf* format, generated by *ProDe*, in which we have to select the implementation for each functional unit that is present in the architecture.

CHAPTER 8

Logic Synthesis

Logic synthesis has been carried out using Design Compiler, the RTL Synthesis tool by Synopsys.
As a preliminary step, it was necessary to copy in the working directory the synopsys_dc.setup file, that is used for initializing design parameters and variables, declare design libraries, and so on. The commands in this file are executed when Design Compiler is invoked.
The first command in the synthesis flow is analyze that reads the HDL source files and checks for syntactical errors. For example:

```
analyze -f vhdl -lib WORK ../src/toplevel.vhdl
```

Then, the elaborate command is invoked and it is in charge to:

- Translate the design into a technology-independent design (GTECH), starting from the intermediate files generated during analysis.
- Allow changing of parameter values (generics) defined in the source code.
- Replace the VHDL arithmetic operators in the code with DesignWare components.

At this point, if the elaboration completed successfully, the design is represented in GTECH format, which is an internal, equation-based, technology-independent design format.
Before synthesizing the design with compile_ultra command the clock signal with a period of 10 ns was generated with the following command:

```
create_clock -name CK -period 10.0 CK
```

Moreover, other parameters like clock uncertainty, input and output delay, output load and the wire model were specified. After synthesis we have observed the following results:

- Timing
- Area
- Power

8.1 Timing

```
TIMING ANALYSIS
---------------------------------------------------------------
  Startpoint: fu_LIFTER/add_sub_reg
              (rising edge-triggered flip-flop clocked by CLOCK)
  Endpoint: fu_LIFTER/mul_reg_reg[31]
            (rising edge-triggered flip-flop clocked by CLOCK)
  Path Group: CLOCK
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  toplevel           tsmc090_wl40          fast

  Point                                                    Incr       Path
  ---------------------------------------------------------------------------
  clock CLOCK (rise edge)                                  0.00       0.00
  clock network delay (ideal)                              0.00       0.00
  fu_LIFTER/add_sub_reg/CK (EDFFX2)                        0.00       0.00 r
  fu_LIFTER/add_sub_reg/Q (EDFFX2)                         0.15       0.15 f
  fu_LIFTER/U202/Y (CLKINVX32)                             0.23       0.38 r
  fu_LIFTER/U15/Y (INVX18)                                 0.16       0.54 f
  fu_LIFTER/U140/Y (AOI22X2)                               0.19       0.73 r
  fu_LIFTER/U197/Y (BUFX12)                                0.23       0.96 r
  .
  .
  .
  fu_LIFTER/U203/Y (XOR2X1)                                0.20       9.61 f
  fu_LIFTER/U669/Y (MX2X1)                                 0.18       9.80 f
  fu_LIFTER/mul_reg_reg[31]/D (DFFRQX2)                    0.00       9.80 f
  data arrival time                                                   9.80

  clock CLOCK (rise edge)                                 10.00      10.00
  clock network delay (ideal)                              0.00      10.00
  clock uncertainty                                       -0.07       9.93
  fu_LIFTER/mul_reg_reg[31]/CK (DFFRQX2)                   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                                                    -9.80
--------------------------------------------------------------------------
slack (MET)                                                           0.10
```

The delay report shows delay calculation in two sections: the first section for data arrival time and the second for data required time.
The data arrival time is the time required for a signal to travel from a starting point to an end point of a path.
The data required time is the maximum time a signal has for travelling that path.
The time difference between data required time and data arrival time is called slack or timing margin of the path. If the slack is negative, there is a timing violation on that path.
In our case the slack is equal to 0, which means that the maximum clock frequency is 100 Mhz. However, it is important to highlight that during this phase we are not considering parasitics.
The critical path is on the multiplier path, and goes from the output of the register that selects the multiplier operand (add_sub_reg) to the input of the pipeline register, that divides the adder and the multiplier blocks (mul_reg_reg). Only a portion of the critical path is shown in figure 8.1.

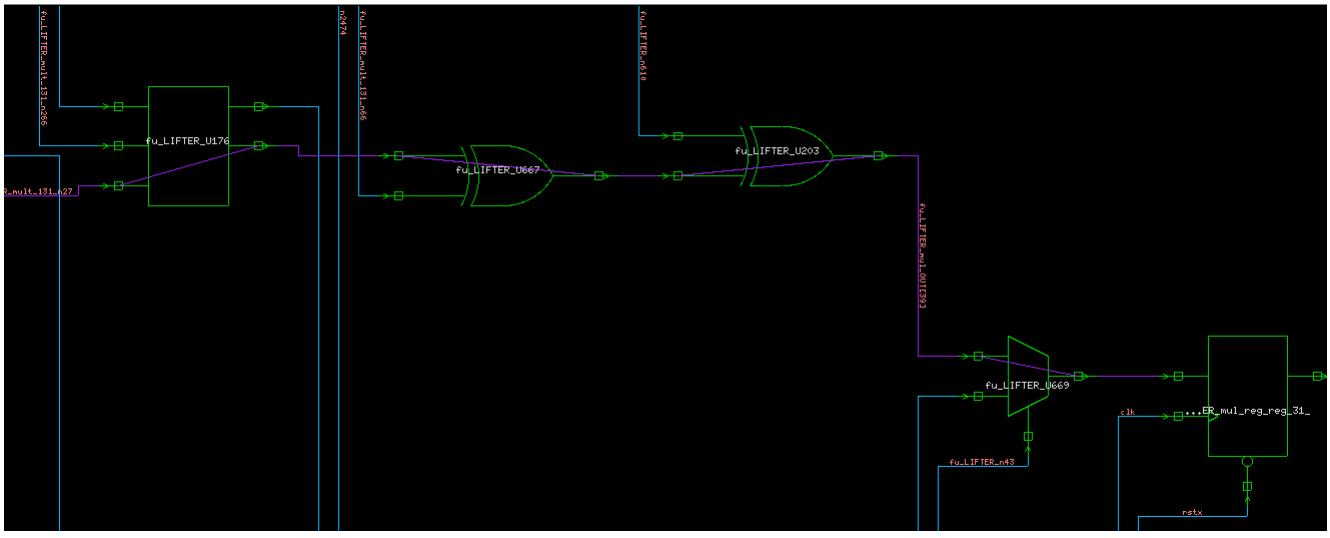

Figure 8.1: Critical path

8.2 Area

```
AREA ANALYSIS
```

```
--------------------------------------------------------------
Number of ports:              242
Number of nets:              3598
Number of cells:             3345
Number of references:         145

Combinational area:       28476.604943
Noncombinational area:    15377.140808
Net Interconnect area:    3112682.750000

Total cell area:          43853.746094
Total area:               3156536.496094

--------------------------------------------------------------
```

The above report simply shows the total area of the design. It is the sum of three components: combinational, noncombinational, and net interconnect area.
The area due to logic cells in the design is made by the combinational (basic logic gates like ANDs, ORs, etc) and the noncombinational (registers) factors. The third factor affecting the area (net interconnect area) is due to the wires connecting these cells.
It is the dominant value, since heavy routing is required to connect all functional units of the processor. By simply looking at the Design Schematic, created by Design Vision, it is practically impossible to distinguish the RTL blocks, since wires are the dominant part of the view.

8.3 Power

```
POWER ANALYSIS
--------------------------------------------------------------
Cell Internal Power  =   1.6800 mW   (61%)
Net Switching Power  =   1.0537 mW   (39%)
                         ---------
Total Dynamic Power  =   2.7337 mW  (100%)

Cell Leakage Power   = 123.4962 uW

--------------------------------------------------------------
```

The internal power is caused by the charging of internal loads as well as by the short-circuit current between N and P transistors of a gate, when both are on. As we can see from the report, it is the dominant component of the dynamic power, dissipated any time the capacitive load of a net charges or discharges.

Dynamic power has been computed assuming a switching activity equal to 0.5, which is a non realistic value, since some nodes could switch more frequently than others.

CHAPTER 9

Place and route

To perform the Place and Route we used Cadence SoC Encounter, the steps needed are:

- Importing the design
- Floorplanning
- Power planning and routing
- Cell placing
- Clock tree synthesis
- Filler Placement
- Signal routing
- Timing analysis
- Design analysis and verification

Importing the design First we need to import the synthesized netlist generated by Design Compiler; to do so we customized the configuration file specifying all the needed files like the libraries and the same netlist.

Floorplanning Within this step the software set the area to be assigned to the design, in our case it is set in the center of the chip with a margin with respect to the die boundary.

Power planning and routing The power planning aim is to place the necessary metal stripes to provide power to the entire chip. First two rings are added around the chip boundary for VCC and GND, then some vertical stripes are placed in order to reduce the path length of the current, and so the voltage drop along the power lines. At this point the horizontal wires are placed to power the standard cells and complete the power planning.

Cell placing Now the library cells will be placed according to the synthesized netlist. Encounter will try to place these cells in the best possible way to limit the routing congestion.
A screenshot of the cell placement is shown in figure 9.1.

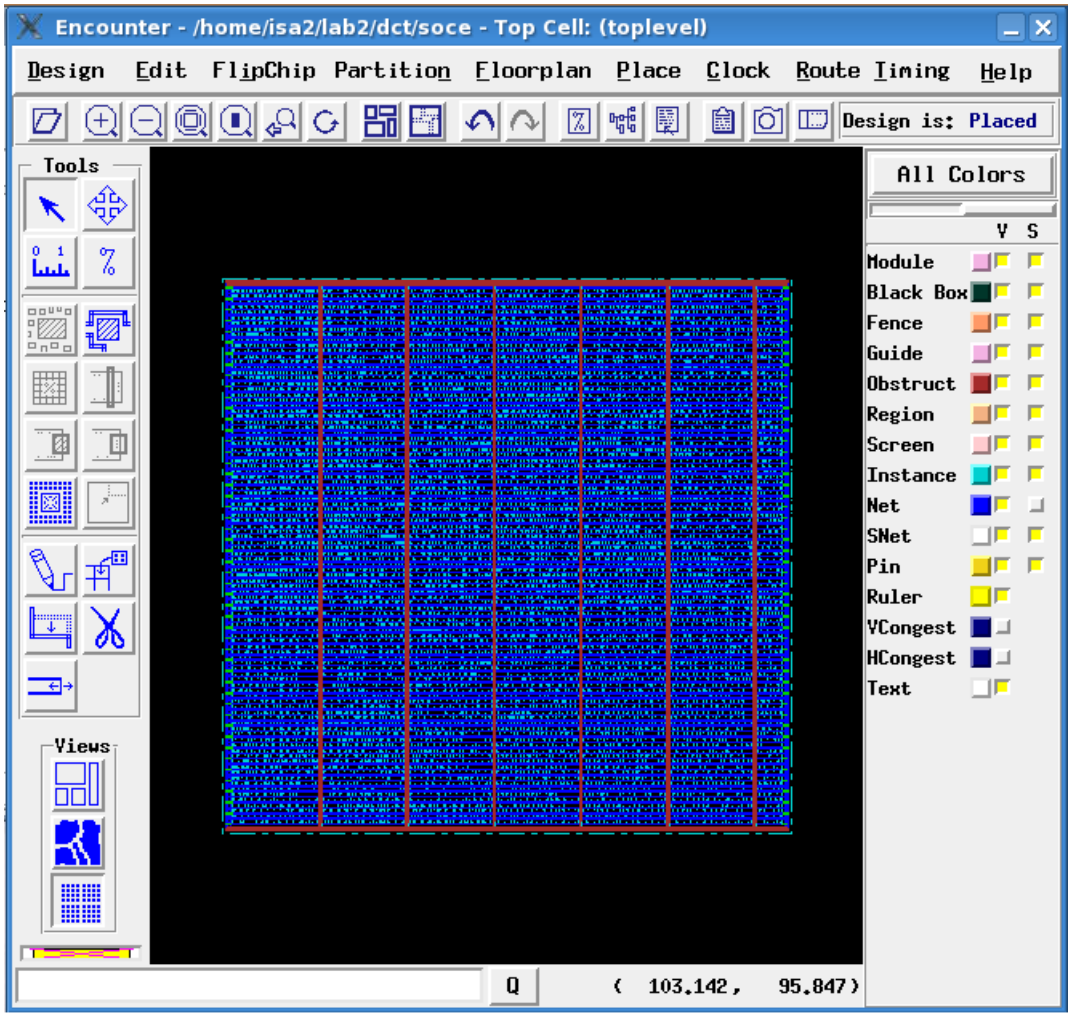

Figure 9.1: Cell placement

Clock tree synthesis Clock tree synthesis (CTS) is the process of insertion of buffers or inverters along the clock paths of the design, in order to achieve minimum skew or

balanced skew. The goal of CTS is to minimize the load and the delay of the clock tree.
In the *.ctstch* file we specify a tree of three levels, where the first level uses CLKBUFX1 buffers (where X1 indicates that they are able to drive only one gate), the second level uses CLKBUFX4 buffers, while the third one uses CLKBUFX8 buffer. As we go up in the tree the load capacitance seen by buffers increases, so we need to increase also their driving capability.
After clock synthesis all results are stored in the report file and a verilog file, with all buffers that have been added to the design, is generated.

```
------------------------------Clock Tree Report--------------------------------
Nr. of Subtrees                 : 0
Nr. of Sinks                    : 946
Nr. of Buffer                   : 25
Nr. of Level (including gates) : 3
Max trig. edge delay at sink(R): rf_RF_reg_reg_6__6_/CK 330.3(ps)
Min trig. edge delay at sink(R): fu_LIFTER_mul_reg_reg_23_/CK 301.5(ps)

                                  (Actual)              (Required)
Rise Phase Delay                : 301.5~330.3(ps)       0~10000(ps)
Fall Phase Delay                : 341.5~371.1(ps)       0~10000(ps)
Trig. Edge Skew                 : 28.8(ps)              10000(ps)
Rise Skew                       : 28.8(ps)
Fall Skew                       : 29.6(ps)
Max. Rise Buffer Tran           : 194.6(ps)             10000(ps)
Max. Fall Buffer Tran           : 187.4(ps)             10000(ps)
Max. Rise Sink Tran             : 134.5(ps)             10000(ps)
Max. Fall Sink Tran             : 121.4(ps)             10000(ps)



***** NO Transition Time Violation *****

***** NO Capacitance Violation *****
-------------------------------------------------------------------------------
```

25 buffers have been inserted with a rise skew of 28.8 ps and a fall skew of 29.6 ps.

- Rise skew is calculated based on rise edge at the clock root.
- Rise skew is calculated based on rise edge at the clock root.
- Fall skew is calculated based on fall edge at the clock root.

- Triggering edge Skew is calculated based on arrival times of active signals on clock pins.
- Transition time is the time taken by the clock signal to make a transition from 20% to 80% of the maximum value.

The clock skew is not equal to 0 because paths are not perfectly balanced. However, it is a small value with respect to the clock period.
Main causes of clock skew could be:

- unequal wire length;
- unequal buffer delay;
- unequal load;
- IR-drop.

Filler placement This step is required for technological reasons to guarantee continuity in N and P wells in each row. It consists of filling the holes on the die with filler cells.

Signal routing This step is divided into two phases: the first phase is a sort of raw routing, a planning of the wire position. The second phase is fine routing of the wires which connect all the cells; here Encounter will try to find the best solution for the wires positioning. A screenshot of the signal routing is shown in Fig.9.2.

Timing analysis To do the timing analysis we first have to specify the operating conditions like temperature, power supply voltage and the process variations, then we must extract the parasitics (resistance and capacitance) of each wire in our design. The **ExtractRC** command returns two files *.setload* and *.setres* that are used to set capacitance and resistance for each net respectively. They have to be included in the *.sdc* file, that is used to fix timing constraints.

```
#/************************************************
# * Timing constraint file in SDC format
# ************************************************/
set_wire_load_model -name tsmc090_wl40 -library fast
create_clock [get_ports clk]  -name CLOCK  -period 10  -waveform {0 5}
source mydct.setload
source mydct.setres
```

Now we are ready for the timing analysis, we load the timing constraints (used also in the synthesis) and launch the analysis.

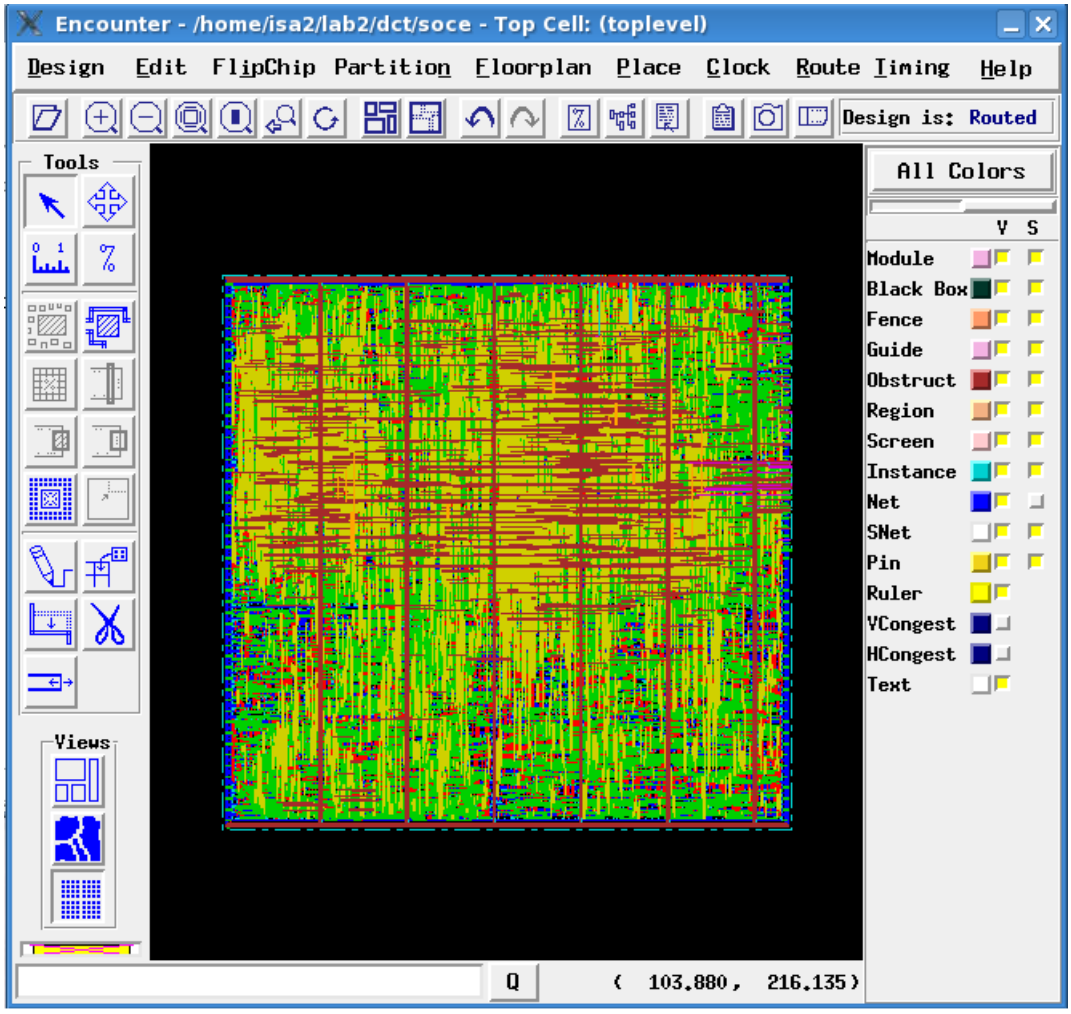

Figure 9.2: Signal routing

As we anticipated in the Logic Synthesis chapter, the maximum operating frequency, found in that phase, is not a real value, since we were not considering resistance and capacitance parasitic values for metal wires. Starting from the minimum declared period (10 ns) we performed Timing Analysis and we found no violating paths.

```
------------------------------------------------------------
*info: Report constrained paths
*      Path type: max (data)
*      Format: long
*** Found 0  violating paths ***
------------------------------------------------------------
```

It means that the DCT can really be clocked at 100 Mhz.

Design analysis and verification Since there are no timing violations we can proceed with the design verification. It checks if there are floating wires and if there are geometric issues related to the design rules, imposed by the technology.

```
-----------------------------------------------------------
Begin Summary ...
  Cells       : 0
  SameNet     : 0
  Wiring      : 0
  Antenna     : 0
  Short       : 0
  Overlap     : 0
End Summary

No DRC violations were found
-----------------------------------------------------------
```

Finally, the total number of gates is reported below.

```
-----------------------------------------------------------
Gate area 2.1168 um^2
Level 0 Module toplevel Gates = 20784
Cells = 5632
Area = 43997.0 um^2
-----------------------------------------------------------
```

APPENDIX A

lift.vhd

```
library ieee;
use ieee.std_logic_1164.all;

package lifter_opcodes is

   constant LIFT_3PI16_1  : std_logic_vector(2 downto 0) := "000";
   constant LIFT_3PI16_2  : std_logic_vector(2 downto 0) := "001";
   constant LIFT_PI16_1   : std_logic_vector(2 downto 0) := "010";
   constant LIFT_PI16_2   : std_logic_vector(2 downto 0) := "011";
   constant LIFT_PI8_1    : std_logic_vector(2 downto 0) := "100";
   constant LIFT_PI8_2    : std_logic_vector(2 downto 0) := "101";

end lifter_opcodes;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.lifter_opcodes.all;

entity lifter is

   generic (dataw: integer := 32);

   port (
   t1data   : in  std_logic_vector(dataw-1 downto 0);
   t1load   : in  std_logic;
   t1opcode : in std_logic_vector(2 downto 0);
```

```
   o1data  : in std_logic_vector(dataw-1 downto 0);
   o1load  : in std_logic;

   r1data  : out std_logic_vector(dataw-1 downto 0);

   clk     : in std_logic;
   rstx    : in std_logic;
   glock   : in std_logic
   );
end lifter;


architecture rtl of lifter is

   signal t1reg : std_logic_vector(dataw-1 downto 0);
   signal o1reg : std_logic_vector(dataw-1 downto 0);
   signal r1reg : std_logic_vector(dataw-1 downto 0);

   signal mul_A: std_logic_vector(dataw-1 downto 0);
   signal mul_B: std_logic_vector(dataw-1 downto 0);
   signal mul_OUT: std_logic_vector(2*dataw-1 downto 0);

   signal mul_reg: std_logic_vector(dataw-1 downto 0);

   signal add_A: std_logic_vector(dataw-1 downto 0);
   signal add_B: std_logic_vector(dataw-1 downto 0);
   signal add_OUT: std_logic_vector(dataw-1 downto 0);
   signal add_sub: std_logic;

begin

   regs_op1 : process (clk, rstx)
   begin -- process regs
      if rstx = '0' then               -- asynchronous reset (active low)
         t1reg <= (others => '0');
      elsif clk'event and clk = '1' then -- rising clock edge
         if glock = '0' then
            if t1load = '1' then
               t1reg <= t1data;

               case t1opcode is
                  when LIFT_PI8_1 =>
```

```
                mul_B <= conv_std_logic_vector(51, dataw);
                add_sub <= '0';

              when LIFT_PI8_2 =>
                mul_B <= conv_std_logic_vector(98, dataw);
                add_sub <= '1';

              when LIFT_PI16_1 =>
                mul_B <= conv_std_logic_vector(25, dataw);
                add_sub <= '0';

              when LIFT_PI16_2 =>
                mul_B <= conv_std_logic_vector(50, dataw);
                add_sub <= '1';

              when LIFT_3PI16_1 =>
                mul_B <= conv_std_logic_vector(78, dataw);
                add_sub <= '0';

              when LIFT_3PI16_2 =>
                mul_B <= conv_std_logic_vector(142, dataw);
                add_sub <= '1';

              when others =>
                null;
            end case;

        end if;
      end if;
   end if;
end process regs_op1;


regs_op2: process (clk, rstx)
begin -- process regs
   if rstx = '0' then               -- asynchronous reset (active low)
      o1reg <= (others => '0');
   elsif clk'event and clk = '1' then -- rising clock edge
      if glock = '0' then
         if o1load = '1' then
            o1reg <= o1data;
         end if;
      end if;
```

```
   end if;
end process regs_op2;


proc_mux: process(add_sub, t1reg, o1reg)
begin
   if add_sub = '0' then
      mul_A <= o1reg;
      add_A <= t1reg;
   else
      mul_A <= t1reg;
      add_A <= o1reg;
   end if;
end process proc_mux;


mul_OUT <= mul_A * mul_B;


pipe_mul_reg: process (clk, rstx)
begin -- process regs
   if rstx = '0' then               -- asynchronous reset (active low)
      mul_reg <= (others => '0');
   elsif clk'event and clk = '1' then -- rising clock edge
      if glock = '0' then
         mul_reg <= mul_OUT(dataw-1+8 downto 8);
      end if;
   end if;
end process pipe_mul_reg;


add_B <= mul_reg;


proc_add: process(add_sub, add_A, add_B)
begin
   if add_sub = '0' then
      add_OUT <= add_A + add_B;
   else
      add_OUT <= add_A - add_B;
   end if;
end process proc_add;
```

```
   output_reg: process (clk, rstx)
   begin -- process regs
      if rstx = '0' then               -- asynchronous reset (active low)
         r1reg <= (others => '0');
      elsif clk'event and clk = '1' then -- rising clock edge
         if glock = '0' then
            r1reg <= add_OUT;
         end if;
      end if;
   end process output_reg;


   r1data <= r1reg;
```

lift.cc

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

#include "OSAL.hh"

OPERATION(LIFT_PI8_1)
   TRIGGER
   int x1 = INT(1);
   int x2 = INT(2);
   int result = 0;

   result = x1 + ((x2*51) >> 8);

   IO(3) = result;

   END_TRIGGER
END_OPERATION(LIFT_PI8_1)


OPERATION(LIFT_PI8_2)
   TRIGGER
   int x1 = INT(1);
   int x2 = INT(2);
   int result = 0;
```

```
   result = x2 - ((x1*98) >> 8);

   IO(3) = result;

   END_TRIGGER
END_OPERATION(LIFT_PI8_2)


OPERATION(LIFT_PI16_1)
   TRIGGER
   int x1 = INT(1);
   int x2 = INT(2);
   int result = 0;

   result = x1 + ((x2*25) >> 8);

   IO(3) = result;

   END_TRIGGER
END_OPERATION(LIFT_PI16_1)


OPERATION(LIFT_PI16_2)
   TRIGGER
   int x1 = INT(1);
   int x2 = INT(2);
   int result = 0;

   result = x2 - ((x1*50) >> 8);

   IO(3) = result;
```