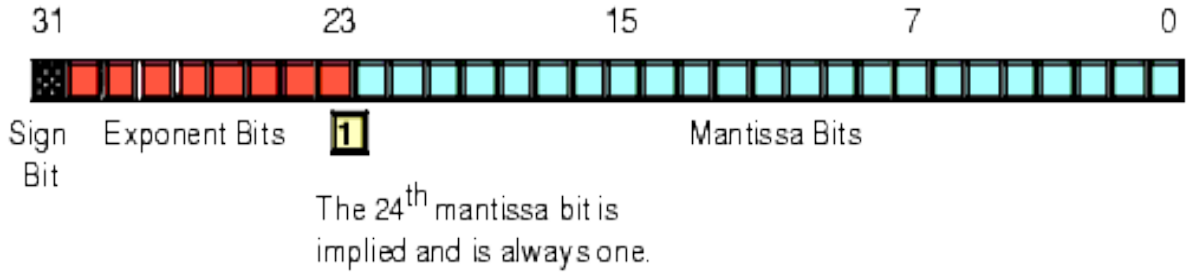
Part I: Creating a personality that adds two 32-bit single precision floating point numbers

This tutorial will introduce Floating Point numbers and addition and walk you through the steps to create a Convey personality, generate a floating point adder using Xilinx IP Core generator and use it in the Convey personality you created to add two 32-bit single precision floating point numbers.

Introduction to Floating point numbers and addition:

The following block represents 32-bit precision floating point numbers



Variables in floating point numbers

- Sign: 0 = positive, 1 = negative
- Exponent: 8 bits
- Fraction: 23 bits
- Mantissa = 1 + F
- Bias = 127 for single precision
- Formula: $(-1)^S * (1 + F) * 2^E$

Converting a decimal number to floating point number steps

1. Convert a Decimal number to Binary number
   975.75_{10} >> 1111001111.11_2

2. Normalize the number
   $1.11100111111_2 * 2^9$

3. From this normalized number we can fill all 32-bits of floating point number

   Sign bit = 0 (number is positive)
   Exponent = Bias + 9 = 127 + 9 = 136_{10} = $1000\ 1000_2$
   Fraction part will contain all the bits after decimal point.

Hence, Floating Point representation of 975.75_{10} is

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	Exponent								Fraction																						
0	1	0	0	0	1	0	0	0	1	1	1	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0

Summation of 2 floating point numbers:

- A. 0 0101 0110 111 0011 1000 0000 1100 0011 (number A)
- B. 0 0101 0011 100 1110 0101 1111 0001 1111 (number B)
- E_A 0101 0110 = 86 (exponent part of number A)
- E_B 0101 0011 = 83 (exponent part of number B)
- Normalizing the smaller number (B), shifting to the right by 86-83 = 3 such that exponents of both the numbers match.
- B. 1.100 1110 0101 1111 0001 1111 * $\mathbf{2^{83}}$

  0. 001 1001 1100 1011 1110 0011 * $\mathbf{2^{86}}$
- A. 1. 111 0011 1000 0000 1100 0011 * $\mathbf{2^{86}}$

  10. 000 1101 0100 1100 1010 0110 * $\mathbf{2^{86}}$

  1. 000 0110 1010 0110 0101 0011 * $\mathbf{2^{87}}$

  A+B (32-bit)

  0 0101 0111 000 0110 1010 0110 0101 0011

Steps:

Run the "newCnyProject" script. You can find it on the class wiki page (http://wikis.ece.iastate.edu/cpre584/index.php/Convey_PDK_Tutorial). Name your project. This project will use "Fpadd" as its name. Pick a number for your project. This number can be between 65000 and 65535. This tutorial will use 65005. Don't worry about this number conflicting with other personalities. The script creates a new personality directory for your personality.

```
cd
cd $CNYSCRIPTS/newCnyProject
```

Try making and running your application.

```
cd caeFpadd/appFpadd
make
./run
```

Now, we want our personality to add two 32-bit single precision floating point numbers. This tutorial will create a personality that takes in two 32-bit floating point numbers and returns their sum.

First let's start at the program that will be running on the host processor. We input two floating point numbers, add them and return the sum. Edit appFpadd.cpp with your favorite text editor. Then replace:

```
#TODO: replace with own cp call
cout << "@user:calling coprocessor" << endl;
copcall_fmt(sig, cpTalk, "");
cout << "@user:calling coprocessor" << endl;
```

with:

```
float input1 = 0.75;
float input2 = 0.50;
float output;
cout << "Input1: " << input1 << endl;
cout << "Input2: " << input2 << endl;
//TO DO: call coprocessor
cout << "Calling Assembly unit";
//print output sum value
cout << "output sum received back = " << output << endl;
```

Now we want an assembly function to run on the coprocessor that the host can call. Edit cpFpadd.s. Go to the cpTalk function and replace its name with 'summation'. Arguments are usually passed using the registers and the mov instruction. The same method is used for the return values. Add the following to cpFpadd.s:

```
    .globl  summation
    .type   summation. @function
    .signature pdk=65005
summation:
```

```
     mov.ae0 %s1, $0, %aeg
     mov.ae0 %s2, $1, %aeg
     caep00.ae0 $0
     mov.ae0 %aeg, $0, %s1
rtn
```

This will run but since we have not written anything for caep00 the instruction does nothing (nop).

[Note: The assembly code calls a coprocessor instruction caep00 and passes the input float values to the application engine general registers (aeg). Note that floating point values are passed in S registers, starting with S1. If you want to pass integer values or addresses, use A registers starting with A8. For more details, refer chapter 11 of the Convey Programmers Guide.]

Now, go back to FpaddApp.cpp
Add the function reference:

```
extern "C" float summation();
```

At the TODO statement add the following:

```
output = f_copcall_fmt(sig, summation,"ss",input1,input2);
```

[Note: In this step we referenced the 'summation' function and also included the coprocessor interface routine f_copcall_fmt. Since our return value is a float, we use f_copcall_fmt. You might want to use some other routine for a different type of return value. For more details on copcall routines, refer Appendix G of the Convey Programmer's Guide.]

Your FpaddApp.cpp should look like below:

```
     float input1 = 0.75;
     float input2 = 0.50;
     float output;

     cout << "Input1: " << input1 << endl;
     cout << "Input2: " << input2 << endl;

     cout << "Calling Assembly unit";
     output = f_copcall_fmt(sig, summation,"ss",input1,input2);

     cout << "output sum received = " << output << endl;
```

Make and see if it compiles.

Verilog part:
Some of the Verilog has been written to help you create your personality. This includes some instruction decoding and aeg register logic. We need to add logic to use aeg0. Also, we need to add logic to execute caep00. Let's start with the aeg register logic. Go the following section of code and add the code in blue to the existing code:

```
//*************************************************************
***********
// PERSONALITY SPECIFIC LOGIC
//*************************************************************
***********
   reg[31:0] sumout;
   reg[31:0] input1;
   reg[31:0] input2;
   wire[31:0] c_sumout;
   wire c_return_sum;
   reg return_sum;
   reg oper_nd;
   wire c_oper_rfd;
   reg r_enable;
   wire f_idle;

//
   // AEG[0..NA-1] Registers
   //
    localparam NA = 51;
    localparam NB = 6;       // Number of bits to represent NAEG

    assign cae_aeg_cnt = NA;

    //output of aeg registers
    wire [63:0]  w_aeg[NA-1:0];

    genvar g;
    generate for (g=0; g<NA; g=g+1) begin : g0
      reg [63:0] c_aeg, r_aeg;

      always @* begin
        case (g)
```

```
//TODO: add cases for registers to be written to
         0:begin
          if(return_sum) begin
             c_aeg[31:0] = sumout;
          end
          else
             c_aeg = r_aeg;
          end
            default: c_aeg = r_aeg;
        endcase
      end

      wire c_aeg_we = inst_aeg_wr && inst_aeg_idx[NB-1:0] == g;

      always @(posedge clk) begin
        if (c_aeg_we) begin
            r_aeg <= cae_data;
            $display("writing: %x", cae_data);
        end
        else
            r_aeg <= c_aeg;
        end
      assign w_aeg[g] = r_aeg;
    end endgenerate
```

Now add the logic to correctly stall the processor.

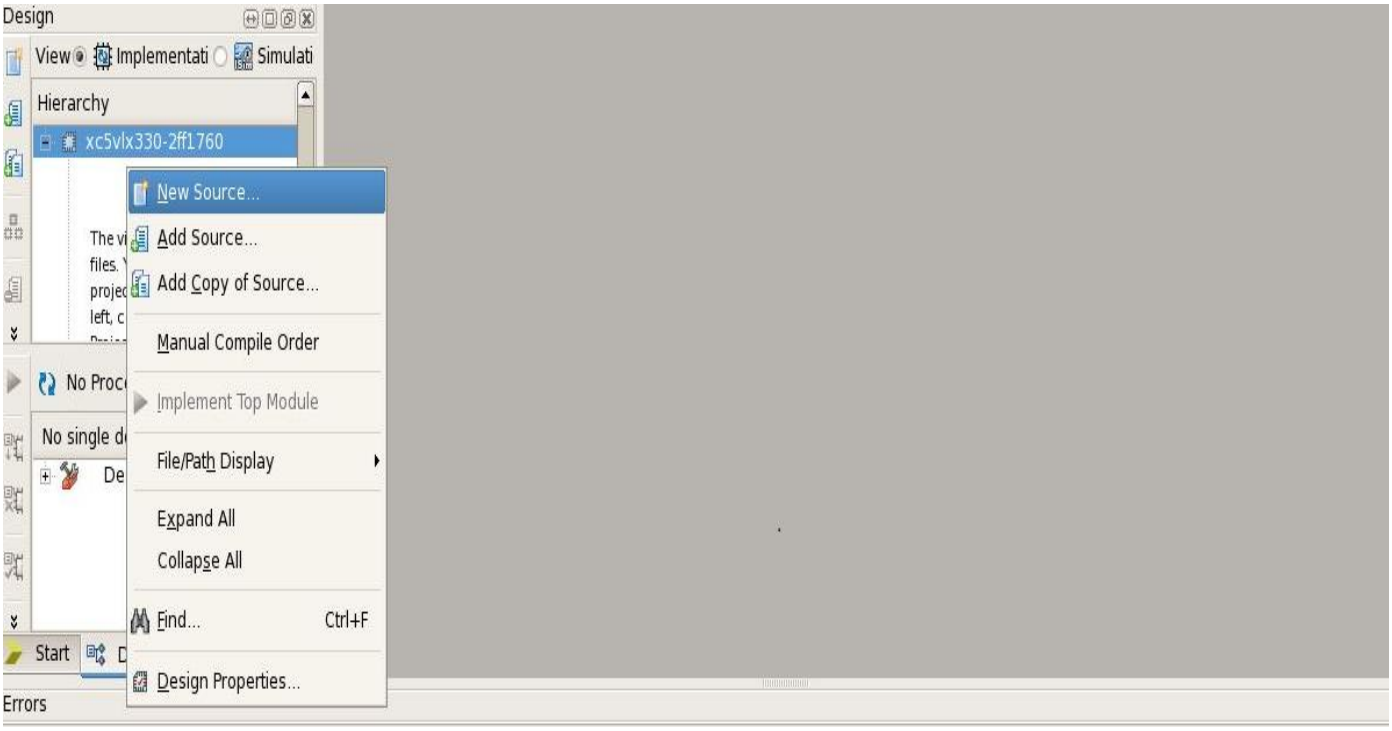
```
//logic for using cae IMPORTANT. cae_idle should be 0 when
executing a custom instruction and 1 otherwise.
//cae_stall should be 1 when when exectuting a custom instruction
and 0 otherwise.
   wire c_caep00;
   reg r_caep00;
   assign c_caep00 = inst_caep == 5'd0 && inst_val;
   always @(posedge clk) begin
     r_caep00 <= c_caep00;
   end
   assign cae_idle = !r_caep00 && f_idle;
   assign cae_stall = c_caep00 || r_caep00 || !f_idle;
```

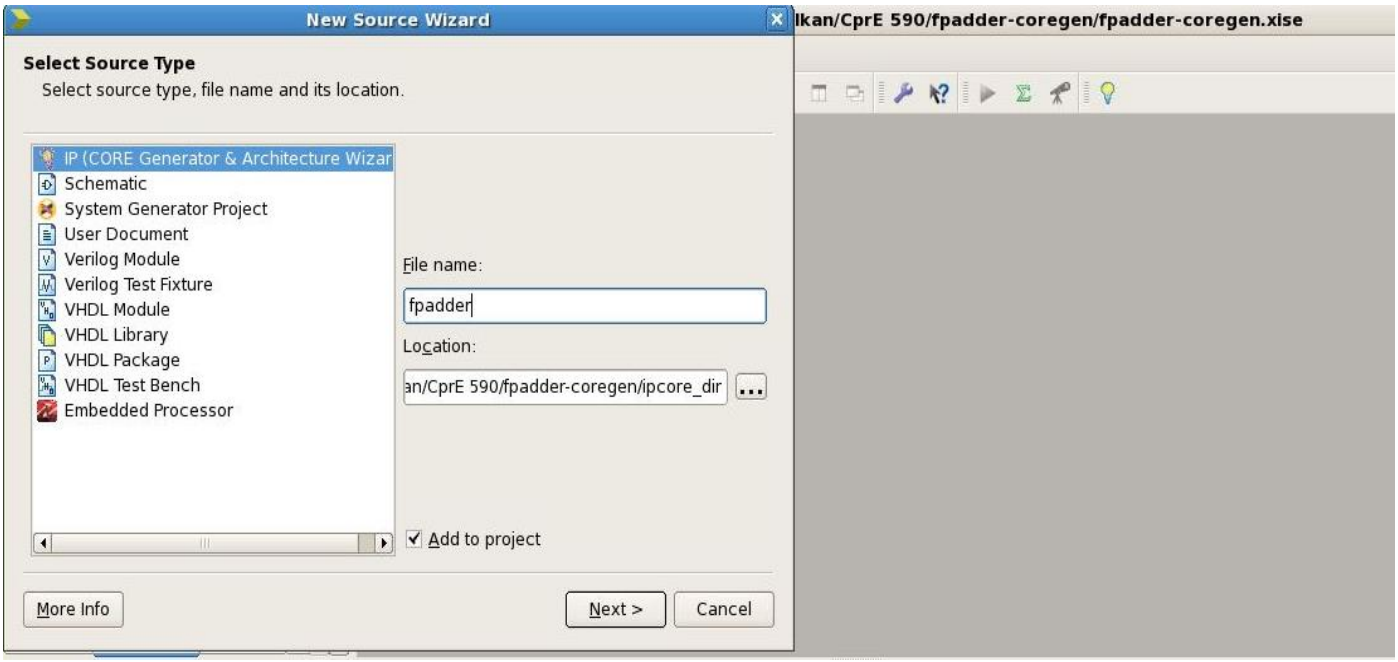
Now add the logic for addition of numbers. Floating point numbers cannot be added using the conventional + sign. They need to be added as mentioned in the introduction above. Hence we instantiate another module called fpadder_top which handles the floating point summation. Note that we are using VHDL to write the fpadder_top module. To read more about how to instantiate a VHDL module in a Verilog code, refer the wiki page.

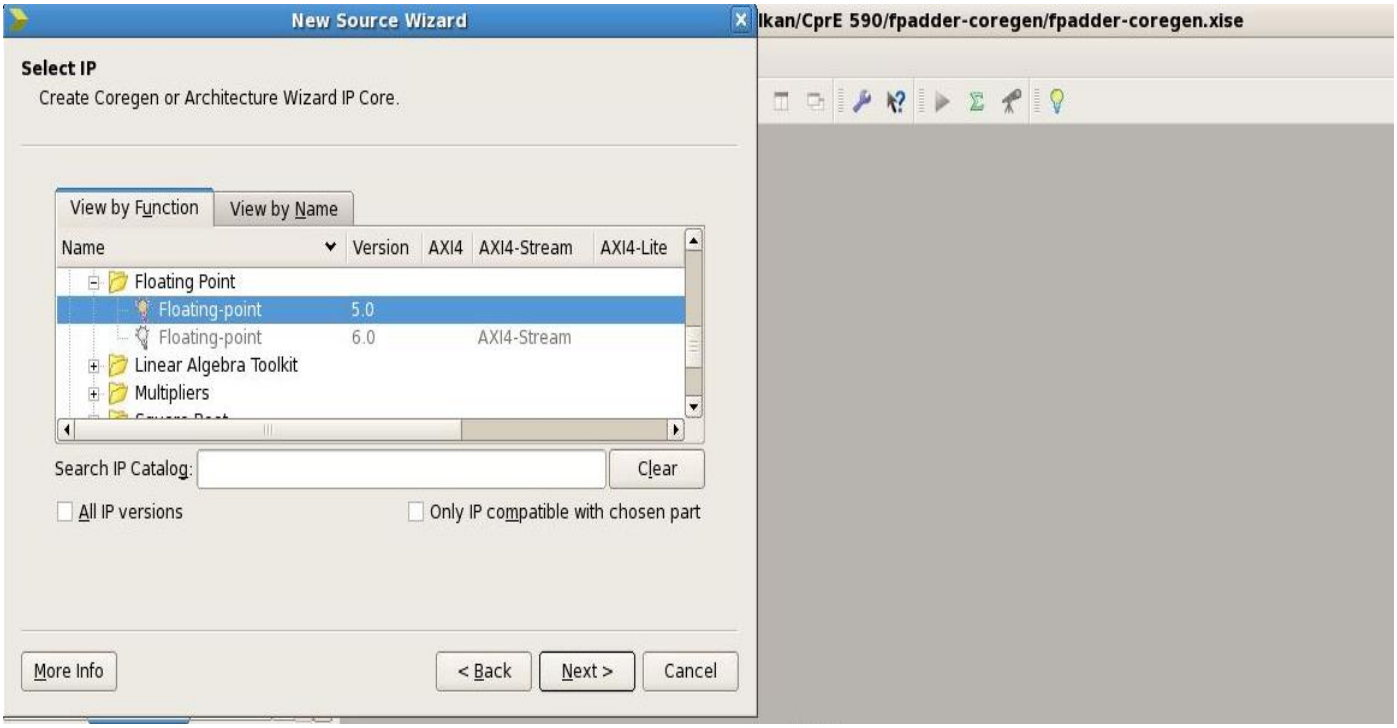
```
    always @(posedge clk_per) begin
        if(inst_caep == 5'd0 && inst_val) begin
          r_enable <= 1'b1;
          input1 <= w_aeg[0][31:0];
          input2 <= w_aeg[1][31:0];
        end
     else begin
        r_enable <= 1'b0;
        return_sum <= 1'b0;
        if(c_return_sum) begin
          sumout <= c_sumout;
          return_sum <= 1'b1;
        end
     end
   end


fpadder_top test_fpadder_top(
.clk(clk_per),
.enable(r_enable),
.reset(reset_per),
.a_in(input1),
.b_in(input2),
.result_out(c_sumout),
.result_rdy(c_return_sum),
.idle(f_idle)
);
```

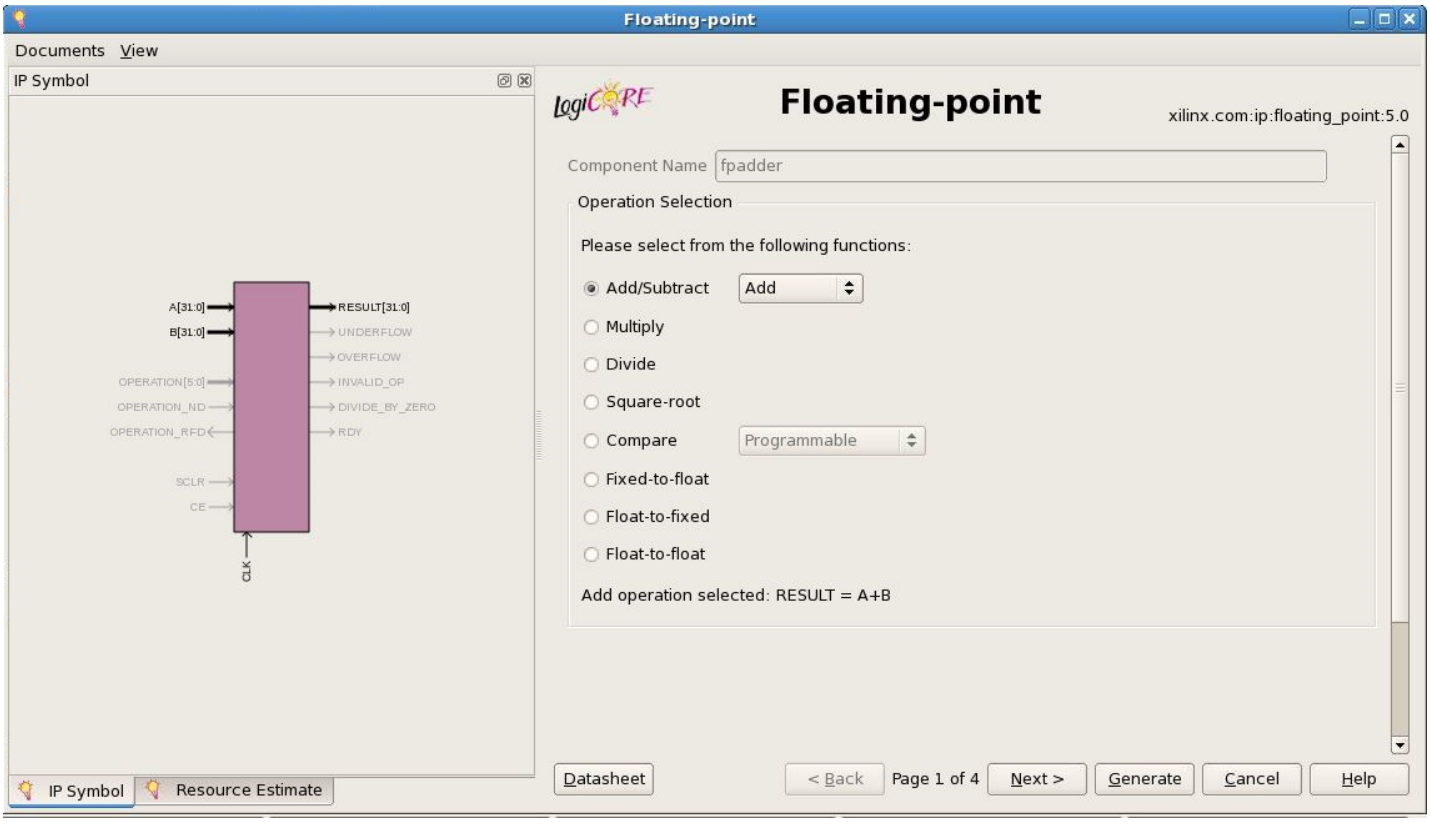
Now, let us write the VHDL file for the floating point addition. There are two parts of the VHDL file: (a) the top level module- fpadder_top and (b) the floating point adder core generated by Xilinx IP Coregen.

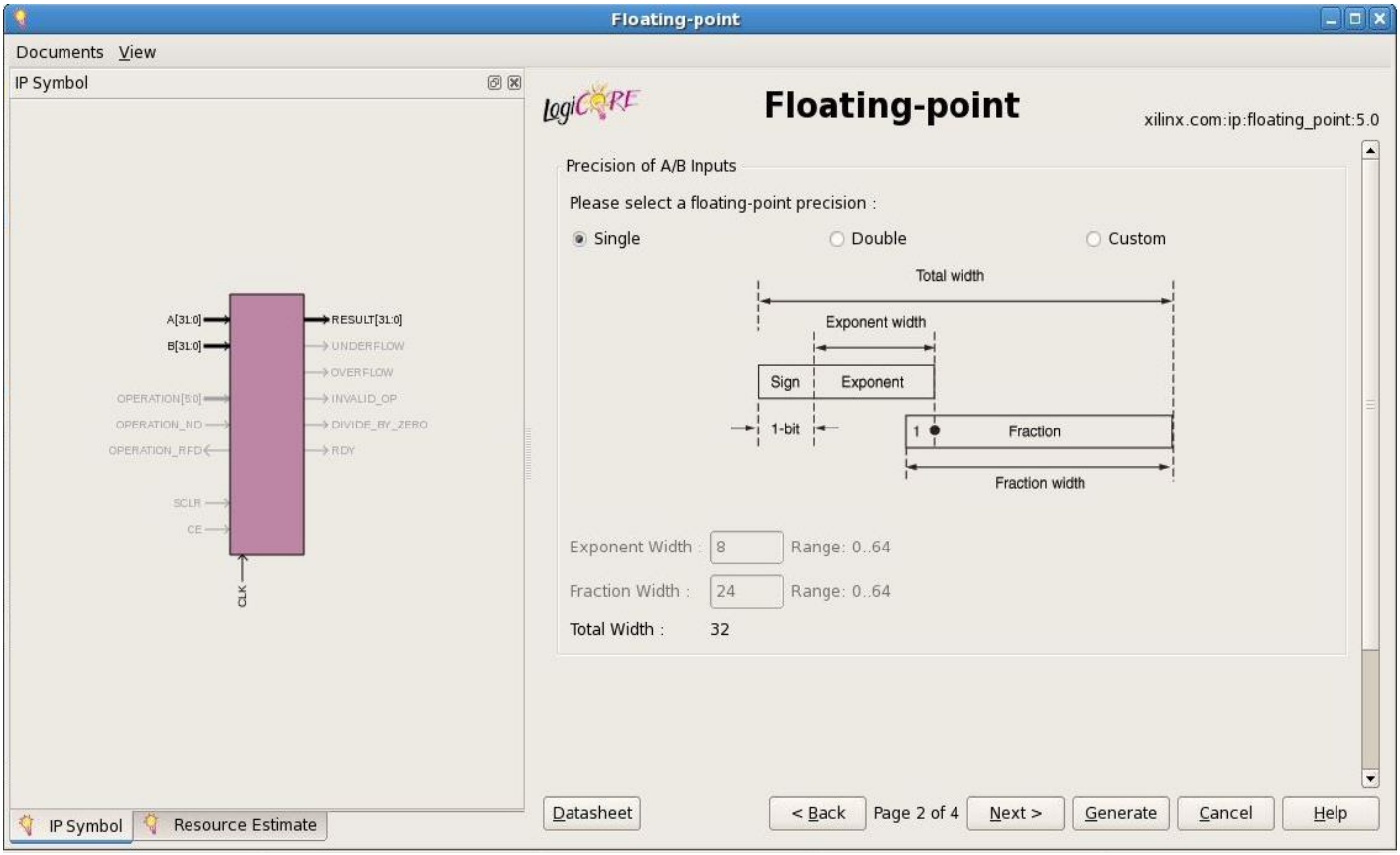
Creating Coregen:
Write ‘ISE &’ in terminal
Create a new project in ISE and follow the following steps

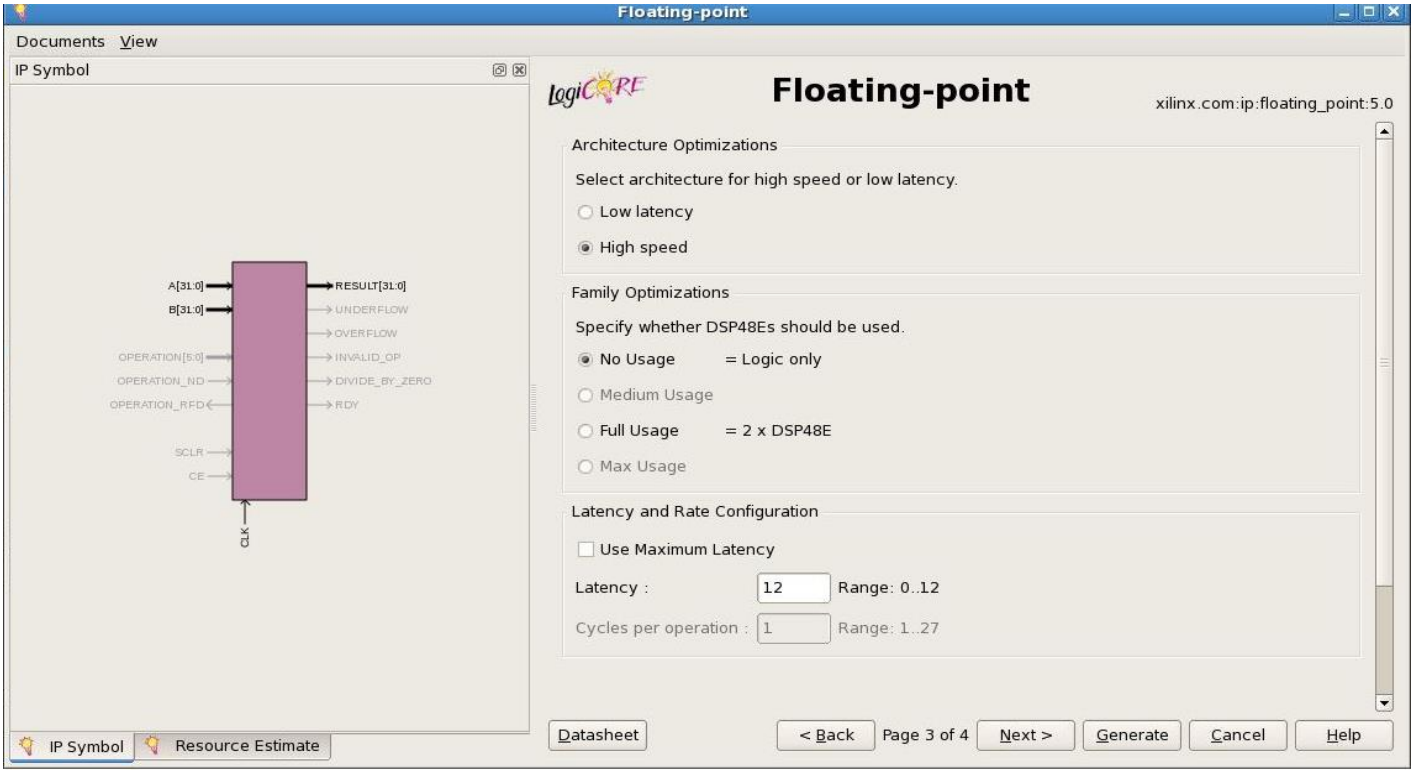

Design
View
Implementati
Simulati
Hierarchy
xc5vlx330-2ff1760
New Source...
Add Source...
Add Copy of Source...
Manual Compile Order
Implement Top Module
File/Path Display
Expand All
Collapse All
Find...
Ctrl+F
Design Properties...
Start
Errors

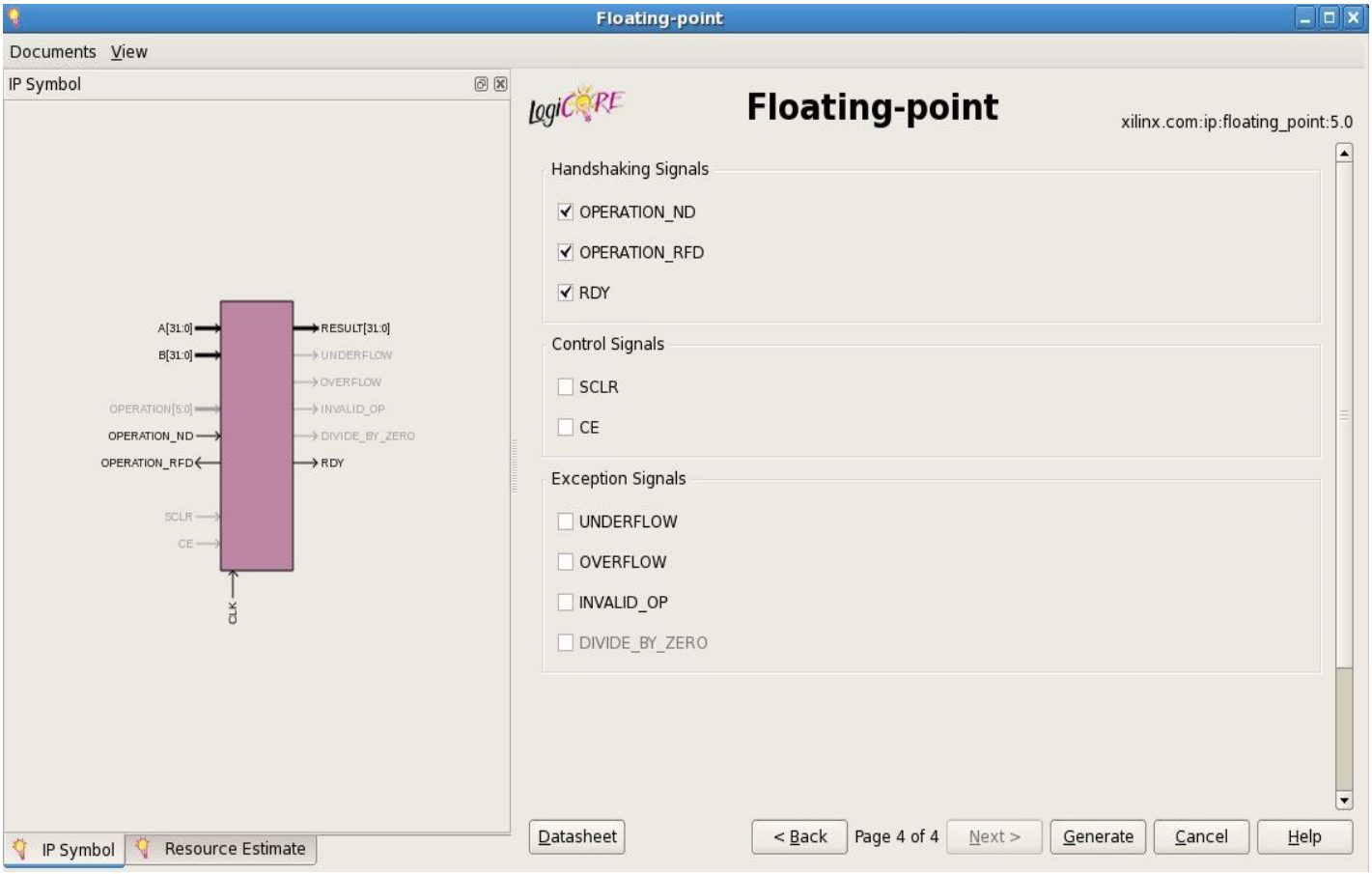

New Source Wizard
Select Source Type
Select source type, file name and its location.
IP (CORE Generator & Architecture Wizar
Schematic
System Generator Project
User Document
Verilog Module
Verilog Test Fixture
VHDL Module
VHDL Library
VHDL Package
VHDL Test Bench
Embedded Processor
File name:
fpadder
Location:
an/CprE 590/fpadder-coregen/ipcore_dir
Add to project
More Info
Next >
Cancel
lkan/CprE 590/fpadder-coregen/fpadder-coregen.xise

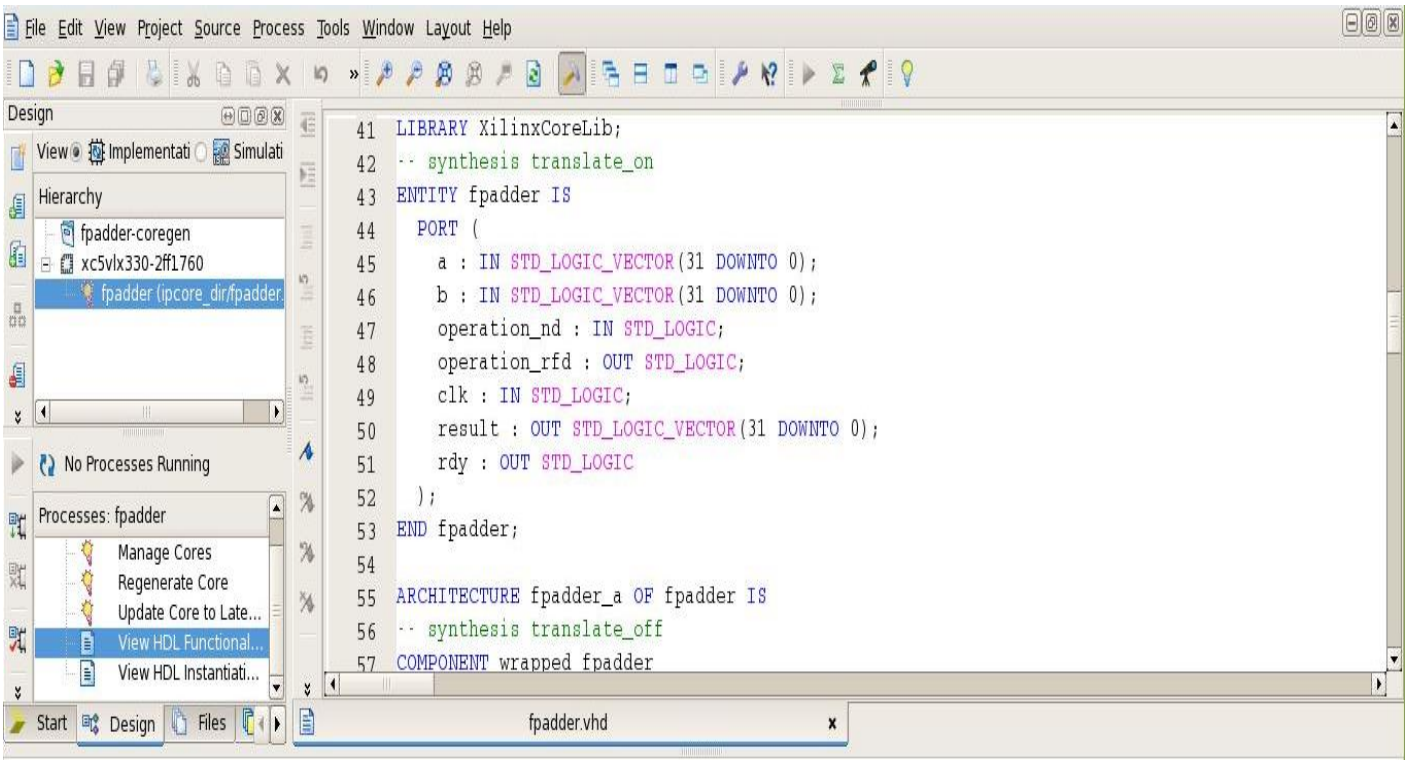

New Source Wizard
Select IP
Create Coregen or Architecture Wizard IP Core.
View by Function
View by Name
Name
Version
AXI4
AXI4-Stream
AXI4-Lite
Floating Point
Floating-point 5.0
Floating-point 6.0 AXI4-Stream
Linear Algebra Toolkit
Multipliers
Search IP Catalog:
Clear
All IP versions
Only IP compatible with chosen part
More Info
< Back
Next >
Cancel
lkan/CprE 590/fpadder-coregen/fpadder-coregen.xise


Floating-point
Documents View
IP Symbol
A[31:0]
B[31:0]
OPERATION[5:0]
OPERATION_ND
OPERATION_RFD
SCLR
CE
CLK
RESULT[31:0]
UNDERFLOW
OVERFLOW
INVALID_OP
DIVIDE_BY_ZERO
RDY
IP Symbol
Resource Estimate
LogiCORE
Floating-point
xilinx.com:ip:floating_point:5.0
Component Name
fpadder
Operation Selection
Please select from the following functions:
Add/Subtract
Add
Multiply
Divide
Square-root
Compare
Programmable
Fixed-to-float
Float-to-fixed
Float-to-float
Add operation selected: RESULT = A+B
Datasheet
< Back
Page 1 of 4
Next >
Generate
Cancel
Help


Floating-point
Documents View
IP Symbol
A[31:0]
B[31:0]
OPERATION[5:0]
OPERATION_ND
OPERATION_RFD
SCLR
CE
CLK
RESULT[31:0]
UNDERFLOW
OVERFLOW
INVALID_OP
DIVIDE_BY_ZERO
RDY
IP Symbol
Resource Estimate
Floating-point
xilinx.com:ip:floating_point:5.0
Precision of A/B Inputs
Please select a floating-point precision :
Single
Double
Custom
Total width
Exponent width
Sign
Exponent
1-bit
1
Fraction
Fraction width
Exponent Width : 8 Range: 0..64
Fraction Width : 24 Range: 0..64
Total Width : 32
Datasheet
< Back
Page 2 of 4
Next >
Generate
Cancel
Help


Floating-point
Documents View
IP Symbol
A[31:0]
B[31:0]
OPERATION[5:0]
OPERATION_ND
OPERATION_RFD
SCLR
CE
CLK
RESULT[31:0]
UNDERFLOW
OVERFLOW
INVALID_OP
DIVIDE_BY_ZERO
RDY
IP Symbol
Resource Estimate
Floating-point
xilinx.com:ip:floating_point:5.0
Architecture Optimizations
Select architecture for high speed or low latency.
Low latency
High speed
Family Optimizations
Specify whether DSP48Es should be used.
No Usage = Logic only
Medium Usage
Full Usage = 2 x DSP48E
Max Usage
Latency and Rate Configuration
Use Maximum Latency
Latency : 12 Range: 0..12
Cycles per operation : 1 Range: 1..27
Datasheet
< Back
Page 3 of 4
Next >
Generate
Cancel
Help


Floating-point
Documents View
IP Symbol
A[31:0]
B[31:0]
OPERATION[5:0]
OPERATION_ND
OPERATION_RFD
SCLR
CE
CLK
RESULT[31:0]
UNDERFLOW
OVERFLOW
INVALID_OP
DIVIDE_BY_ZERO
RDY
IP Symbol
Resource Estimate
Floating-point
xilinx.com:ip:floating_point:5.0
Handshaking Signals
OPERATION_ND
OPERATION_RFD
RDY
Control Signals
SCLR
CE
Exception Signals
UNDERFLOW
OVERFLOW
INVALID_OP
DIVIDE_BY_ZERO
Datasheet
< Back
Page 4 of 4
Next >
Generate
Cancel
Help


File Edit View Project Source Process Tools Window Layout Help
Design
Hierarchy
fpadder-coregen
xc5vlx330-2ff1760
fpadder (ipcore_dir/fpadder
No Processes Running
Processes: fpadder
Manage Cores
Regenerate Core
Update Core to Late...
View HDL Functional...
View HDL Instantiati...
Start
Design
Files
fpadder.vhd
41 LIBRARY XilinxCoreLib;
42 -- synthesis translate_on
43 ENTITY fpadder IS
44 PORT (
45 a : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
46 b : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
47 operation_nd : IN STD_LOGIC;
48 operation_rfd : OUT STD_LOGIC;
49 clk : IN STD_LOGIC;
50 result : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
51 rdy : OUT STD_LOGIC
52);
53 END fpadder;
54
55 ARCHITECTURE fpadder_a OF fpadder IS
56 -- synthesis translate_off
57 COMPONENT wrapped_fpadder

Copy fpadder.vhd, fpadder.ngc and fpadder.xco files to coregen folder in your project folder.

fpadder_top.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity fpadder_top is
    Port ( clk : in  STD_LOGIC;
           enable : in  STD_LOGIC;
        reset : in STD_LOGIC;
           a_in : in  STD_LOGIC_VECTOR (31 downto 0);
           b_in : in  STD_LOGIC_VECTOR (31 downto 0);
           result_out : out  STD_LOGIC_VECTOR (31 downto 0);
           result_rdy : out  STD_LOGIC;
           idle : out  STD_LOGIC);
end fpadder_top;

architecture Behavioral of fpadder_top is

COMPONENT fpadder
    PORT(
         a : IN  std_logic_vector(31 downto 0);
         b : IN  std_logic_vector(31 downto 0);
         operation_nd : IN  std_logic;
         operation_rfd : OUT  std_logic;
         clk : IN  std_logic;
         result : OUT  std_logic_vector(31 downto 0);
         rdy : OUT  std_logic
        );
    END COMPONENT;

     type mc_state is (IDLE_ST, ADD_ST, OUT_ST);
     signal state : mc_state := IDLE_ST;
     signal a : std_logic_vector(31 downto 0);
     signal b : std_logic_vector(31 downto 0);
     signal operation_nd : std_logic;
     --signal clk : std_logic;
     signal operation_rfd : std_logic;
   signal result : std_logic_vector(31 downto 0);
```

```
   signal rdy : std_logic;

begin

     fpadder1: fpadder PORT MAP (
          a => a,
          b => b,
          operation_nd => operation_nd,
          operation_rfd => operation_rfd,
          clk => clk,
          result => result,
          rdy => rdy
        );

     process(clk)
   begin
          if rising_edge(clk) then
               if reset = '1' then
                    idle <= '1';
                    result_out <=
"00000000000000000000000000000000";
                    result_rdy <= '0';
               else
                    case state is
                         when IDLE_ST =>
                              idle <= '1';
                              result_out <=
"00000000000000000000000000000000";
                              result_rdy <= '0';
                              if enable = '1' then
                                   operation_nd<= '1';
                                   a <= a_in;
                                   b <= b_in;
                                   idle <= '0';
                                   state <= ADD_ST;
                              else
                                   state <= IDLE_ST;
                              end if;

                              when ADD_ST =>
                                   if rdy = '1' then
```

```
                                        result_out <= result;
                                        result_rdy <= '1';
                                        state <= OUT_ST;
                                    else
                                        state <= ADD_ST;
                                    end if;

                                when OUT_ST =>

                                    state <= IDLE_ST;
                            end case;

                        end if;

                    end if;

        end process;

end Behavioral;
```

Steps for generating bit file.

Now you can create a bit file to be loaded on the coprocessor.

```
cd ../phys
make
```

This should take 1 to 2 hours, so use NX client and suspend the session and come back later to finish.

```
make release
```

#takes about 5 minutes

```
cd ..
cp ../caefpadd.release/13_04_27_05/cae_fpga.tgz
personalities/65005.1.1.1.0/ae_fpga.tgz
```

Copy your project folder to convey machine using scp command.

To run your application on actual hardware use following commands

```
cd appfpadd
./loadcp
./runcp
```

Conclusion

Using this tutorial, you will be able to create your own personality which uses coregent to add two floating point numbers.

Extension to this can be creating a personality which reads number from memory controller, adds the number and stores the sum back in memory controller.