Implementing Simulink Designs on SRC-6 System

David Meixner, Volodymyr Kindratenko[1], David Pointer

Innovative Systems Laboratory
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign

October 2006

1. Introduction

In this report we describe the process of programming the SRC-6 reconfigurable computer [1] using MathWorks Simulink® [2] with Xilinx System Generator™ for DSP [3] and Xilinx Blockset [4]. By design, the SRC-6 is programmed in the SRC MAP C programming language [5]. Code development for the SRC-6 platform closely resembles code development for a conventional microprocessor-based system, except for explicit code to support data transfer between the system memory and FPGA-controlled memory. The SRC Carte™ development environment [5] allows the developer to bring in third-party subroutines, called macros, that can be used to extend the functionality of the original language. These macros are typically implemented in Verilog Hardware Description Language (HDL) and are brought into the MAP C program via configuration files that define the interface between the macros and MAP C language. We describe a method of using the VHDL and Verilog sources for SRC macros generated from the MathWork's Simulink-based designs.

The ability to introduce Simulink-based designs into the Carte framework opens up new possibilities in programming the SRC-6 system. The main advantage of using Simulink-based designs is the ability to use the fixed-point numeric type, which is not directly available in MAP C. This leads to reduced FPGA resource utilization as one can avoid the need to use larger numerical types for problems that require a reduced numerical range. Other benefits include the ability to use low-level FPGA resources (e.g., BRAM) directly and access to Xilinx IP cores, such as FFT and CORDIC algorithms

2. Simulink Design

The model one wishes to implement should be created using the Xilinx Blockset in Simulink. The input and output ports are 'gateway in' and 'gateway out' blocks, respectively. These should be labeled (in lowercase letters) the same way as the variables to be used in the MAP C code. Note that Xilinx System Generator will convert all variables to lowercase. Also, the "IOB Timing Constraint" for these blocks should be set to "None". The 'gateway in' block also allows the designer to specify the bit width and binary point of the inputs; more on this follows. As a final note, the "Use Placement Information for Core" option should be deselected for all blocks that have this option. Errors might arise during the compilation if this option is selected.

As an example, Figure 1 shows a simple Simulink design which takes three inputs: a, b, and c, and outputs q = (a+b)*c. For this example, all signals here have a bit width of 40 and a binary point of 30. We also note that the overall latency of this design is 5.

The next step is to set up the Xilinx System Generator parameters as shown in Figure 2. Particularly, the

[1] Corresponding author: kindr@ncsa.uiuc.edu

following items are important:

- *Compilation Type: HDL Netlist*
- *Part: The FPGA to be used*
- *Target Directory: The directory where the files will be output*
- *Synthesis Tool: Synplify Pro*
- *Hardware Description Language: VHDL**
- *FPGA Clock period (ns): 10*

*) Verilog can also be used as the hardware description language instead of VHDL. However, some System Generator blocks can only be translated into VHDL. There is a way to get around this and translate these blocks into Verilog, and this method is described in the Xilinx System Generator user guide. Nevertheless, when the option is available, it simplifies the process to choose VHDL as the hardware description language.

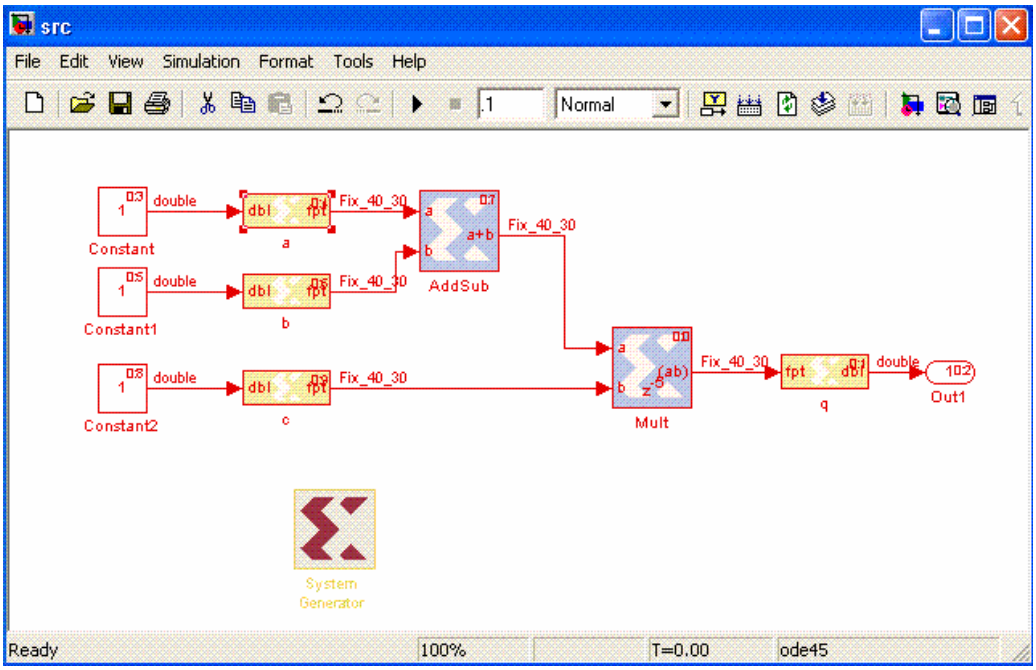

Figure 1. Simple Simulink example.

Once the model generation is complete, several files will be created (here <*design*> is the name of the model):

- <*design*>*_files.vhd* - this file contains most of the HDL for the design.
- <*design*>*_clk_wrapper.vhd* - this file is an HDL wrapper that drives clocks and clock enables.
- <*design*>*_clk_wrapper.ncf* - this file contains timing constraints for the design
- *conv_pkg.vhd* - this file contains constants and functions used by <*design*>*_fils.v*.
- *.edn* files - these are files, in addition to the HDL, that implement parts of the design.
- *synplify_*<*design*>*.prj* – this is a project file used by Synplify Pro to compile the design.

In the example shown in Figure 1, called src.mdl, System Generator produces the following files:

- src_files.vhd, src_clk_wrapper.vhd, conv_pkg.vhd
- adder_subtracter_virtex2_7_0_84f1dba84ee809b9.edn
- multiplier_virtex2_7_0_b018b3a1b259a550.edn
- src_clk_wrapper.ncf, synplify_src.prj

These files need to be copied into the macro directory of the MAP C code.

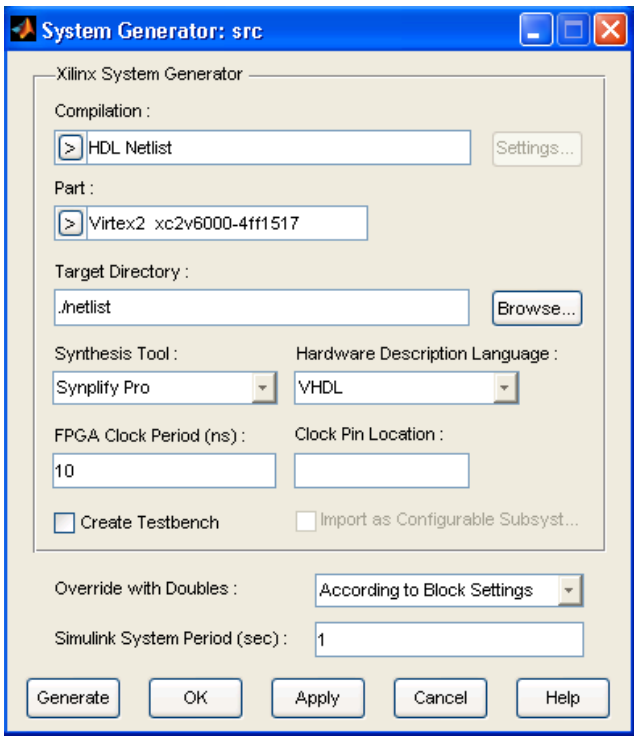

Figure 2. Xilinx System Generator configuration.

If Verilog HDL was selected rather than VHDL, the same files will be outputted, with the *.vhd files being replaced by *.v files.

3. MAP C design

First, if the code was generated using VHDL, the *synplify_<design>.prj* file needs to be modified. The **disable_io_insertion** option by default is set to **false**: "set_option –disable_io_insertion false". It needs to be changed to **true**. For our example, the updated file appears as follows:

```
add_file -vhdl -lib work "conv_pkg.vhd"
add_file -vhdl -lib work "src_files.vhd"
add_file -vhdl -lib work "src_clk_wrapper.vhd"
```

```
project -result_file "src_clk_wrapper.edf"

set_option -top_module fmdemod_clk_wrapper
set_option -technology virtex2p
set_option -part xc2v6000
set_option -package ff1517
set_option -speed_grade -4
set_option -default_enum_encoding sequential
set_option -symbolic_fsm_compiler false
set_option -resource_sharing true
set_option -frequency 100
set_option -fanout_limit 1000
set_option -maxfan_hard false
set_option -disable_io_insertion true   #this is the modified line#
set_option -write_verilog false
set_option -write_vhdl false
set_option -write_apr_constraint false
```

Also, the file name should be changed from *synplify_<design>.prj* to *<deisgn>_clk_wrapper.prj*.

If the code was generated using Verilog, the *<design>_clk_wrapper.v* file needs to be modified. Near the bottom, just before the line ***module <design>_clk_wrapper***, there is a line of code that reads ***`include "conv_pkg.v"***. After this, another *include* statement should be added: ***`include "<deisgn>_files.v"***. For our example, the updated text appears as follows:

```
`include "conv_pkg.v"
`include "src_files.v"         /* this is the line that was added */

module srcmap_clk_wrapper (a, b, c, ce, ce_clr, clk, q);

--SystemGenerator code here--

endmodule
```

Next, a black box definition for the design needs to be created. This includes the inputs and outputs defined in the Simulink model, as well as signals *ce, ce_clr,* and *clk,* created by the System Generator. If the design does not have any delays, the signals *ce, ce_clr, and clk* will not be generated, and should therefore not be included. For this example, the multiplier has a delay of 5, so the clock signals are generated. The black box appears as follows (remember that we are using 40-bit fixed-point numbers):

```
module src_clk_wrapper (a, b, c, ce, ce_clr, clk, q);
    input [39:0] a;
    input [39:0] b;
    input [39:0] c;
    input ce;
    input ce_clr;
    input clk;
    output [39:0] q;
endmodule
```

The next step is to create the info file. This file maps the operators and calls from the source program to

macros and signal names in the HDL code. Note that the inputs and outputs are 64 bits in the source code, but we only use the bottom 40 bits in our HDL code. The *clk* signal should always be mapped to *CLOCK*. Also, LATENCY value should be set accordingly. Lastly, the debug function performs the same operation as our Simulink design, except the result is shifted 20 bits to the right as is required for proper fixed point multiplication. The resultant info file is as follows:

```
BEGIN_DEF "src"
    MACRO = "src_clk_wrapper";
    STATEFUL = NO;
    EXTERNAL = NO;
    PIPELINED = YES;
    LATENCY = 5;

    INPUTS = 3:
      I0 = INT 64 BITS (a[39:0])  // explicit input
      I1 = INT 64 BITS (b[39:0])  // explicit input
      I2 = INT 64 BITS (c[39:0])  // explicit input
      ;
    OUTPUTS = 1:
      O0 = INT 64 BITS (q[39:0])  // explicit output
      ;

    IN_SIGNAL : 1 BITS "ce"     = "1'b1";
    IN_SIGNAL : 1 BITS "ce_clr" = "1'b0";
    IN_SIGNAL : 1 BITS "clk"    = "CLOCK";


    DEBUG_HEADER = #
    void src__dbg (long long v0, long long v1, long long v2, long long *res);
    #;

    DEBUG_FUNC = #
   void src__dbg (long long v0, long long v1, long long v2, long long *res) {
          *res = (v0+v1)*v2 >> 20;
          }
    #;
END_DEF
```

Next, the path for <*design*>*_clk_wrapper.vhd* or <*design*>*_clk_wrapper.v* to the MACROS line of the Makefile should be added. For this example, the line would read:

```
MACROS = macros/src_clk_wrapper.vhd
```

The last step is to provide a MAP C function prototype:

```
void addmult(int64_t a, int64_t b, int64_t c, int64_t *q);
```

A complete Makefile for this example can be found in Appendix A. At this point, the macro is ready to be called from the main MAP C program.

4. Floating Point and Fixed Point Conversions

Because Xilinx blockset for Simulink contains blocks for fixed point operations only, we need a way to convert floating point numbers to fixed point numbers that can be used with the HDL code. The two functions, *float2fix and double2fix*, convert single precision and double precision values, respectively, to fixed point values. Similarly, the functions, *fixed2float* and *fix2double*, convert fixed point values to single precision and double precision floating point values, respectively. These functions can be found in the appendix, written both in C and MAP C.

For the floating point and fixed point conversions to work, bit shift operations must be performed. This means that floating point values need to be held as integers since bit shift operations cannot be performed on floating point numbers. This type conversion can be taken care of in the C functions via the use of *union* data type. However, when using the MAP C functions, this conversion is not done automatically, so all floating point values should be passed in as integer types and floating point values returned are returned as integer types.

Because we are now able to use variables of arbitrary widths, it is important to know how to take advantage of this feature to save space. For a fixed-point number with I integer bits, and F fractional bits, the range is

$$[-2^{(I-1)}, 2^{(I-1)}-(½)^F] \approx \pm 2^{(I-1)}$$

and the precision is

$$(½)^F.$$

When working with floating point numbers, the precision of the number is limited by the size of the mantissa. The mantissa is composed of a leading bit (1) and the fraction bits (23 or 52 bits for single or double precision numbers, respectively). There is also a bit for the sign, so there are a total of 25 or 54 precision bits for single- or double-precision numbers, respectively. This means that the fixed-point representations need no more than 25 or 54 bits, not including leading or trailing zeros. For example,

0111.1111 1111 1111 1111 1011 0101 1101 // 32-bit fixed-point

= +1.11 1111 1111 1111 1111 1011 x 2^2 // single-precision float

= 0111.1111 1111 1111 1111 1011 0000 0000 // corresponding value

Notice that the final corresponding value of the floating point value contains only 24 bits, not counting the 8 trailing zeros. Tables 1 and 2 show the range and precision that can be obtained by placing the decimal point at various locations. For example, if working with 32-bit values, placing the decimal point after the 5^{th} bit, and therefore leaving 25-5=20 bits left for the integer portion, allows for a precision of

$$(½)^5 = 3.125e\text{-}2$$

and an approximate range of

$$\pm 2^{(20-1)} = \pm 524288$$

Of course, the fixed point numbers can be smaller than 25 or 54 bits if such an accurate precision is not needed. For instance, in the example used throughout this paper, the values were 40 bits with the decimal point placed after the 30^{th} bit. This allows for a precision of $(½)^{30}$ = 9.313225746154785156 25e-10 and a range of $\pm 2^{(10-1)} = \pm 512$.

32-BIT RANGE

DECIMAL POINT	RANGE (+/-)	PRECISION
(0)	16777216	1.00000000000000000000000e+00
(1)	8388608	5.00000000000000000000000e-01
(2)	4194304	2.50000000000000000000000e-01
(3)	2097152	1.25000000000000000000000e-01
(4)	1048576	6.25000000000000000000000e-02
(5)	524288	3.12500000000000000000000e-02
(6)	262144	1.56250000000000000000000e-02
(7)	131072	7.81250000000000000000000e-03
(8)	65536	3.90625000000000000000000e-03
(9)	32768	1.95312500000000000000000e-03
(10)	16384	9.76562500000000000000000e-04
(11)	8192	4.88281250000000000000000e-04
(12)	4096	2.44140625000000000000000e-04
(13)	2048	1.22070312500000000000000e-04
(14)	1024	6.10351562500000000000000e-05
(15)	512	3.05175781250000000000000e-05
(16)	256	1.52587890625000000000000e-05
(17)	128	7.62939453125000000000000e-06
(18)	64	3.81469726562500000000000e-06
(19)	32	1.90734863281250000000000e-06
(20)	16	9.53674316406250000000000e-07
(21)	8	4.76837158203125000000000e-07
(22)	4	2.38418579101562500000000e-07
(23)	2	1.19209289550781250000000e-07
(24)	1.00000000	5.96046447753906250000000e-08
(25)	0.50000000	2.98023223876953125000000e-08
(26)	0.25000000	1.49011611938476562500000e-08
(27)	0.12500000	7.45058059692382812500000e-09
(28)	0.06250000	3.72529029846191406250000e-09
(29)	0.03125000	1.86264514923095703125000e-09
(30)	0.01562500	9.31322574615478515625000e-10
(31)	0.00781250	4.65661287307739257812500e-10
(32)*	0.00390625	2.32830643653869628906250e-10

*unsigned values only

Table 1. 32-bit range and precision

64-BIT RANGE

DECIMAL POINT	RANGE (+/-)	PRECISION
(0)	9007199254740992	1.0000000000000000000000000000000000000e+00
(1)	4503599627370496	5.0000000000000000000000000000000000000e-01
(2)	2251799813685248	2.5000000000000000000000000000000000000e-01
(3)	1125899906842624	1.2500000000000000000000000000000000000e-01
(4)	562949953421312	6.2500000000000000000000000000000000000e-02
(5)	281474976710656	3.1250000000000000000000000000000000000e-02
(6)	140737488355328	1.5625000000000000000000000000000000000e-02
(7)	70368744177664	7.8125000000000000000000000000000000000e-03
(8)	35184372088832	3.9062500000000000000000000000000000000e-03
(9)	17592186044416	1.9531250000000000000000000000000000000e-03
(10)	8796093022208	9.7656250000000000000000000000000000000e-04
(11)	4398046511104	4.8828125000000000000000000000000000000e-04
(12)	2199023255552	2.4414062500000000000000000000000000000e-04
(13)	1099511627776	1.2207031250000000000000000000000000000e-04
(14)	549755813888	6.1035156250000000000000000000000000000e-05
(15)	274877906944	3.0517578125000000000000000000000000000e-05
(16)	137438953472	1.5258789062500000000000000000000000000e-05
(17)	68719476736	7.6293945312500000000000000000000000000e-06
(18)	34359738368	3.8146972656250000000000000000000000000e-06
(19)	17179869184	1.9073486328125000000000000000000000000e-06
(20)	8589934592	9.5367431640625000000000000000000000000e-07
(21)	4294967296	4.7683715820312500000000000000000000000e-07
(22)	2147483648	2.3841857910156250000000000000000000000e-07
(23)	1073741824	1.1920928955078125000000000000000000000e-07
(24)	536870912	5.9604644775390625000000000000000000000e-08
(25)	268435456	2.9802322387695312500000000000000000000e-08
(26)	134217728	1.4901161193847656250000000000000000000e-08
(27)	67108864	7.4505805969238281250000000000000000000e-09
(28)	33554432	3.7252902984619140625000000000000000000e-09
(29)	16777216	1.8626451492309570312500000000000000000e-09
(30)	8388608	9.3132257461547851562500000000000000000e-10
(31)	4194304	4.6566128730773925781250000000000000000e-10
(32)	2097152	2.3283064365386962890625000000000000000e-10
(33)	1048576	1.1641532182693481445312500000000000000e-10
(34)	524288	5.8207660913467407226562500000000000000e-11
(35)	262144	2.9103830456733703613281250000000000000e-11
(36)	131072	1.4551915228366851806640625000000000000e-11
(37)	65536	7.2759576141834259033203125000000000000e-12
(38)	32768	3.6379788070917129516601562500000000000e-12
(39)	16384	1.8189894035458564758300781250000000000e-12
(40)	8192	9.0949470177292823791503906250000000000e-13
(41)	4096	4.5474735088646411895751953125000000000e-13
(42)	2048	2.2737367544323205947875976562500000000e-13
(43)	1024	1.1368683772161602973937988281250000000e-13
(44)	512	5.6843418860808014869689941406250000000e-14
(45)	256	2.8421709430404007434844970703125000000e-14
(46)	128	1.4210854715202003717422485351562500000e-14
(47)	64	7.1054273576010018587112426757812500000e-15
(48)	32	3.5527136788005009293556213378906250000e-15
(49)	16	1.7763568394002504646778106689453125000e-15
(50)	8	8.8817841970012523233890533447265625000e-16
(51)	4	4.4408920985006261616945266723632812500e-16
(52)	2	2.2204460492503130808472633361816406250e-16
(53)	1.00000000000	1.1102230246251565404236316680908203125e-16
(54)	0.50000000000	5.5511151231257827021181583404541015625e-17
(55)	0.25000000000	2.7755575615628913510590797170227050781e-17
(56)	0.12500000000	1.3877787807814456755295395851135253906e-17
(57)	0.06250000000	6.9388939039072283776476979255676269531e-18
(58)	0.03125000000	3.4694469519536141888238489627838134766e-18
(59)	0.01562500000	1.7347234759768070944119244813919067383e-18
(60)	0.00781250000	8.6736173798840354720596224069595336914e-19
(61)	0.00390625000	4.3368086899420177360298112034797668457e-19
(62)	0.00195312500	2.1684043449710088680149056017398834229e-19
(63)	0.00097656250	1.0842021724855044340074528008699417114e-19
(64)*	0.00048828125	5.4210108624275221700372650053972e-20

*unsigned values only

Table 2. 64-bit range and precision

5. Image Processing Example

In this example, we use the image processing model provided with the Xilinx Blockset, called sysgenconv5x5.mdl (Figure 3). This model filters an image using n-tap MAC FIR filters. One of nine

different filters can be chosen by changing the mask value on the 5x5 Filter block.

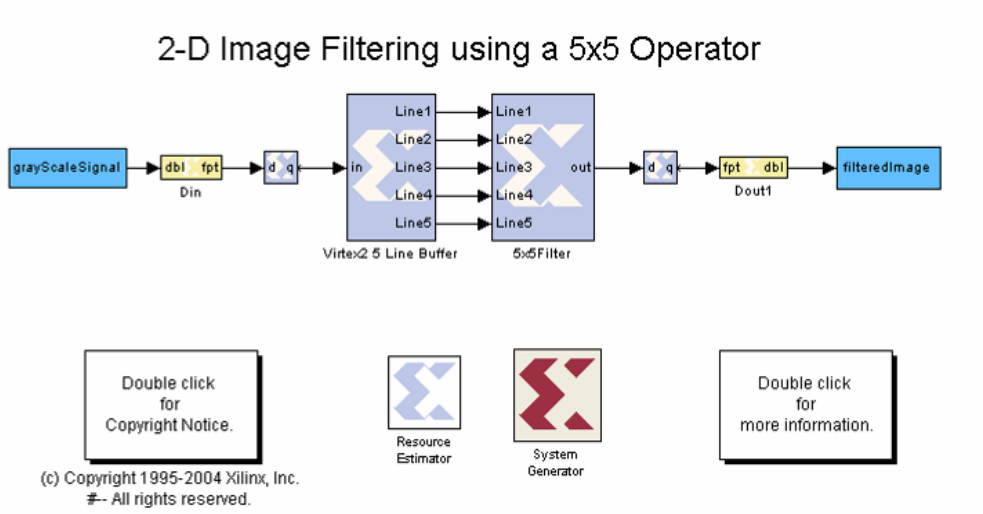

Figure 3. 2D image filter example provided with Xilinx Blockset for Simulink.

The design needs to be slightly modified to be ported to SRC-6. Since the 5-tap MAC FIR filters are clocked at a rate 5 times faster than the input, we need to ensure that the MAP only sends one value to the macro for every five clock cycles. This is done by defining this macro as a periodic macro, with period 5. The MAP compiler requires that periodic macros have a "clr" and "start" input bit, and a "done" output bit. Even though these are not necessary for this design to run properly, they must be added as shown in Figure 4.

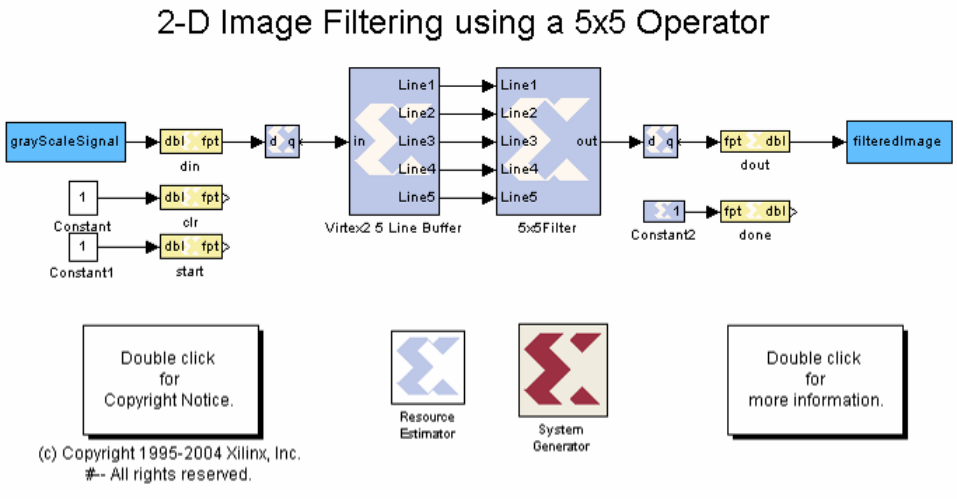

Figure 4. Modified design suitable for using on SRC-6.

Another modification that must be made is to change the "IOB Timing Constraint" property on the gateway in and gateway out blocks from "Data Rate; Set 'FAST' Attribute" to "None". If this is not done, the compiler will display errors.

Once the design is set up properly, the Xilinx System Generator can be run, followed by the development path outlined in Section 3. The main.c file should simply read values from the unfiltered image, pass them to the MAP C subroutine, and then get the output values from the MAP and write them to another file or display the filtered image. The MAP C code simply passes the image pixel values to the macro and retrieves the output values. The main difference between this example and the previous one is the info file. This is where we define that this is a periodic macro, with period 5. This is done by declaring the period to be 5, and including the "clr" and "start" input bits and the "done" output bit.

```
BEGIN_DEF "conv"
    MACRO = "sysgenconv5x5_clk_wrapper";
    LATENCY = 2650;
    STATEFUL = NO;
    EXTERNAL = NO;
    PERIOD = 5;

    INPUTS = 3:
      I0 = INT 64 BITS (din[7:0])         // explicit input

      I1 = INT 1 BITS (clr)               // implicit input
      I2 = INT 1 BITS (start)             // implicit input
      ;
    OUTPUTS = 2:
      O0 = INT 64 BITS (dout[7:0])        // explicit output

      O1 = INT 1 BITS (done)              // implicit output
      ;

    IN_SIGNAL : 1 BITS "ce"     = "1'b1";
    IN_SIGNAL : 1 BITS "ce_clr" = "1'b0";
    IN_SIGNAL : 1 BITS "clk"    = "CLOCK";


    DEBUG_HEADER = #
        void conv__dbg(long long x, long long *y);
        #;

    DEBUG_FUNC = #
        void conv__dbg(long long x, long long *y)
        {
            *y = 255-x;;
        }
        #;
END_DEF
```

6. FM Radio Example

This next example demonstrates how to design an FM radio demodulator in Simulink that can be used on the SRC Portable system. The hardware setup for this example consists of an SRC Portable MAP [6]

with one Gsample/sec analog-to-digital converter board connected to the GPIO port of the primary FPGA and an RF front-end board (Figures 5 and 6). The wide-band RF receiver front-end used in this study to acquire RF samples is essentially the same design as provided by the GNU Software Radio project [7].

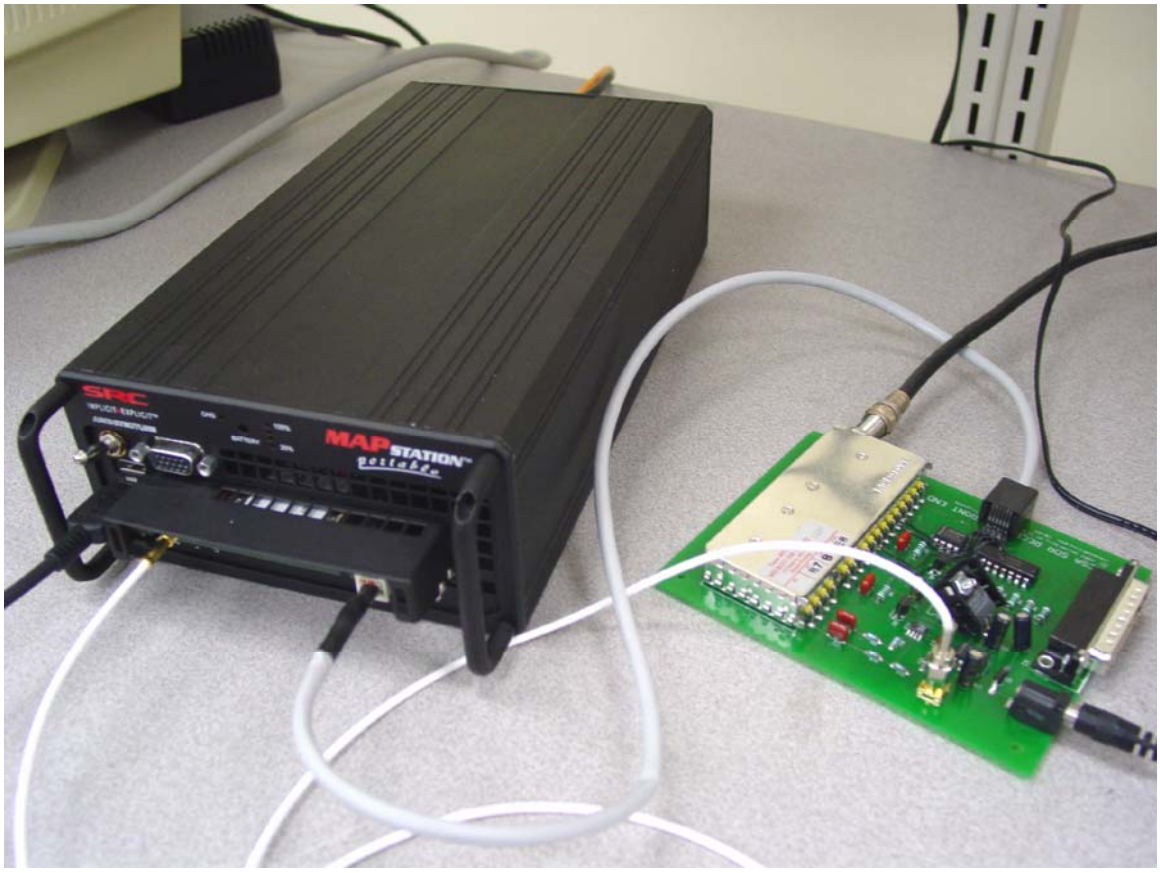

Figure 5. SRC Portable MAP with wide-band RF front-end board.

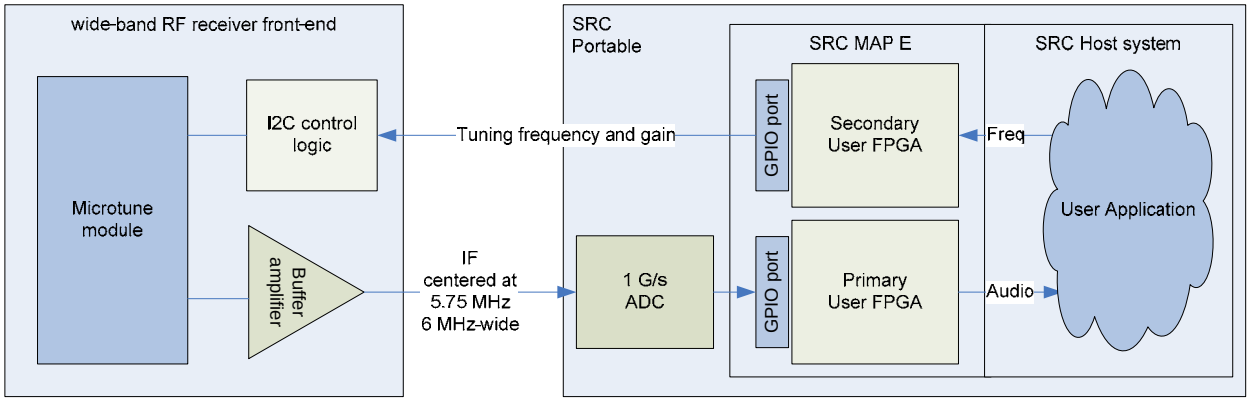

Figure 6. Block diagram of the Software-Defined Radio setup.

The core of the receiver is a commercially available 4937 DI5 Microtune 3x7702 cable TV tuner module manufactured by Microtune, Inc. The tuner uses a double-conversion approach with a 43.75 MHz IF frequency and a second conversion to 5.75 MHz IF frequency. It downconverts a 6 MHz-wide channel located anywhere within the 50-860 MHz range to IF centered at 5.75 MHz. The output signal level is adjustable via two gain-control inputs. Band selection and tuning is done via I2C bus connected to the GPIO port of the secondary FPGA on the SRC Portable MAP. Figure 6 provides a functional diagram of the overall system.

The Simulink software radio design is shown in Figure 7. The input from the ADC is clocked at 100 MHz and is 11 bits wide with the decimal point at 10. The two delays help to buffer the input (Figure 8). The signal is then filtered through a bandpass FIR filter (Figure 9), with cutoff frequencies at 5.75 MHz and 5.76 MHz. (This filter was designed using the FDATool.) The filter is also set to decimate at a 5:1 ratio, so the output sampling frequency is 20 MHz. After the bandpass filter, the signal is downconverted to the baseband by multiplying the input signal by a cosine and sine wave, each with a frequency of 5.75 MHz, the carrier frequency (Figure 10). This simply places the I and Q channels of the FM signal on the baseband.

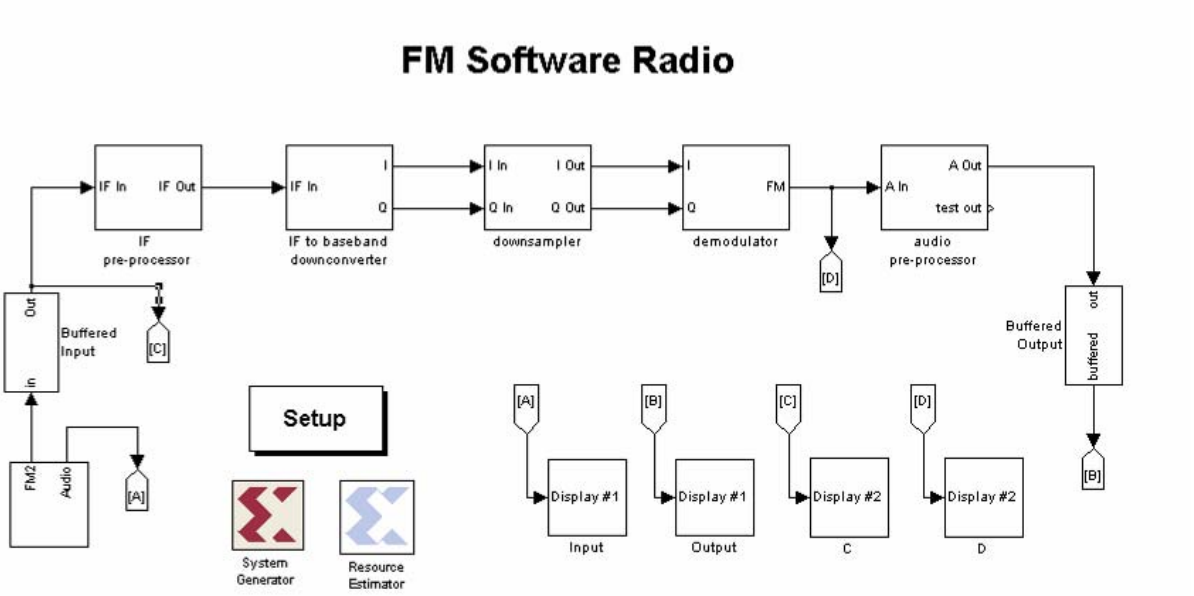

Figure 7. Simulink FM radio receiver design.

At the downsampling stage (Figure 11), the input signal is downsampled by 2 so that the input sampling rate of the lowpass FIR is 10 MHz. This is done because this particular filter is clocked at twice the sampling rate (20 MHz), and the FPGA clock (100 MHz) needs to be an integer multiple of the sampling rates in the design. If this was not done then the filter would be clocked at 40 MHz, which would cause timing problems since 100 MHz is not an integer multiple of 40 MHz. The filter is simply a 20-tap lowpass FIR filter, with the coefficients designed using the MATLAB's *fir1* subroutine that creates a 20 coefficient lowpass FIR filter with normalized cutoff frequency at 0.01. After the filter, the signal is downsampled by 20 which leads to an output sampling rate of 500 kHz.

The FM demodulation is performed by measuring the change in angle between the I and Q channels, which is implemented by taking the arctangent of the two signals and subtracting it from the arctangent of

the previous sample (Figure 12). To get the final audio signal, a gain of 20000 is applied, followed by a 32-tap MAC FIR filter with a normalized cutoff frequency at 0.15 (Figure 13). Finally, the output is converted to a 16 bit value with the decimal point at 15. Like the input, the output is also buffered using two delays (Figure 14).

Figure 8. Buffered input.

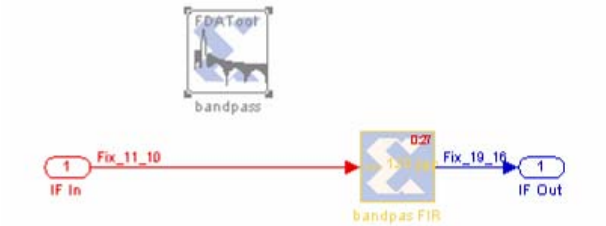

Figure 9. Bandpass filter.

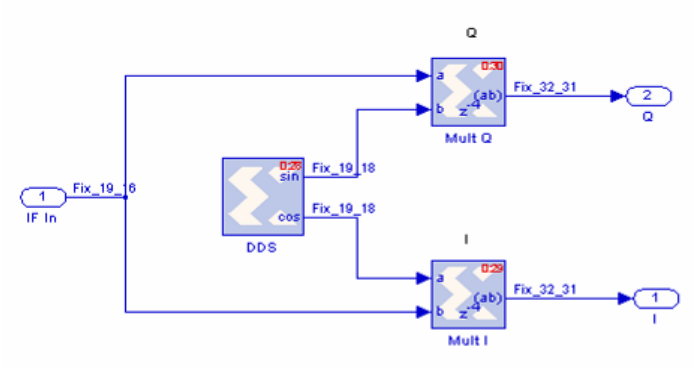

Figure 10. IF to Baseband Converter.

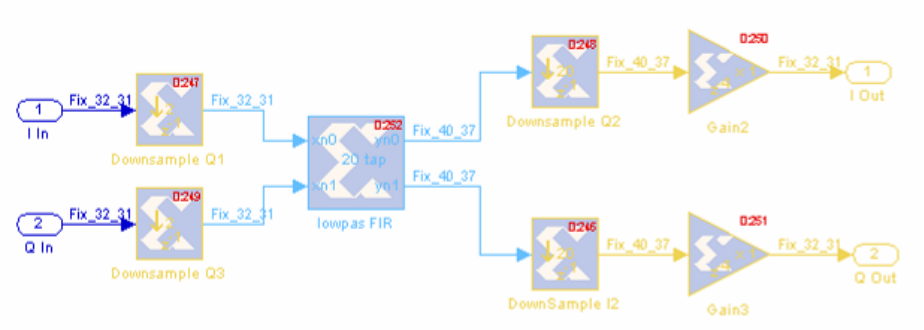

Figure 11. Downsampler.

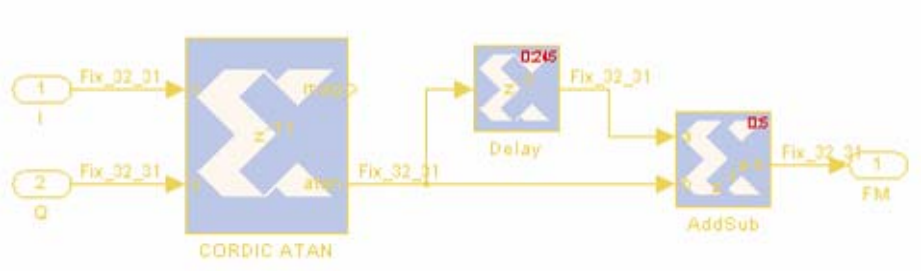

Figure 12. Demodulator.

Figure 13. Audio Pre-Processor.

Figure 14. Buffered Output.

The design is simulated and verified in Simulink. Figure 15 shows a chirp signal that serves as an input into an FM modulator (probe A in Figure 7). The modulated signal is then passed to the receiver design, and the recovered signal (probe B in Figure 7) is shown in Figure 15.

The ADC board returns 16 samples (1 frame) on each call. The values are unsigned int16_t, ranging from 0 to 1023, sampled at 1 Gigasample/second. Since the FPGA runs at 100 MHz, it is not in sync with the ADC, and a valid frame is not returned on every clock cycle. However, on average, the ADC returns 5 valid frames (80 samples) for every 8 clock cycles. Therefore, the data can be downsampled to 100 MHz by taking every 10th valid value over every 8 clock cycles. In the code below, this is implemented as follows. There is one counter, *fifocount*, which counts from 0 to 4, and increments each time there is a new valid frame from the ADC. For each valid frame, when *fifocount* is 0, store values *d0* and *d10* (samples 0 and 10). On the next count, store *d4* and *d14* (samples 20 and 30). Then store *d8* (sample 40) on the next count, followed by *d2* and *d12* (samples 50 and 60) on the following count, and finally store *d6* (sample 70) on the fifth count. There is also another counter, *count*, which counts from 0 to 7 and increments every clock cycle. When *count* is 0, pass *e0* as input to the macro, when *count* is 1, pass *e1*, and so on. This method ensures that data is only read from the ADC when there is a new valid frame, yet valid data is sent as input to the macro on every clock cycle. Finally, the audio is stored by writing every

2268th sample to memory. This provides an output audio rate of 100 MHz / 2268 ≈ 44.1 kHz.

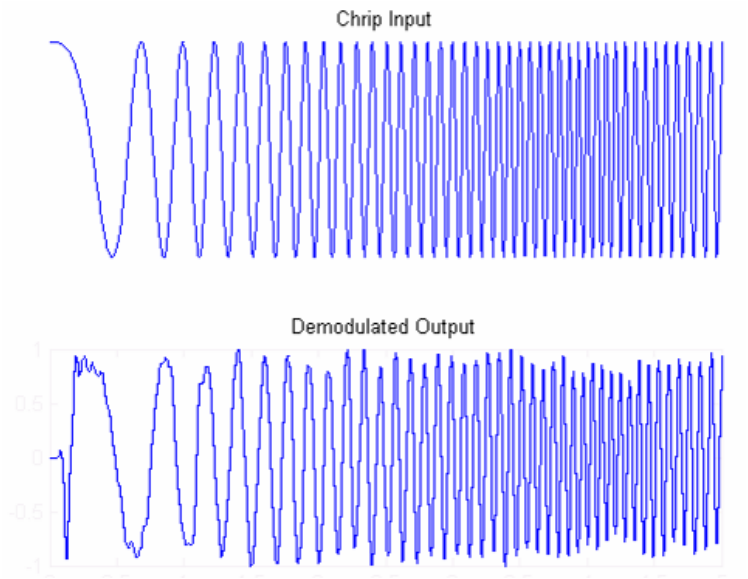

Figure 15. Simulink design simulation.

```
#include <libmap.h>

void tuner_subr();  // secondary chip subroutine prototype
void fmdemod(int16_t d, int16_t *x);  // Simulink-based subroutine prototype

void adc_subr(int n, int64_t res[], int64_t *tune, int mapno)
{
    int16_t d0, d1, d2, d3, d4, d5, d6, d7;
    int16_t d8, d9, d10, d11, d12, d13, d14, d15;
    int16_t sample0, sample1, sample2, sample3,
    int16_t sample4, sample5, sample6, sample7;
    int16_t x0, x1, x2, x3;
    int16_t fm_in, audio_out;
    int16_t clk_enable, fifo_valid;
    int64_t tuner_in, tuner_out;
    int nbytes;
    int iteration, index, iterreset, indexreset;
    int count, countreset, fifocount, fiforeset;

    OBM_BANK_A (al, long long, MAX_OBM_SIZE)

    // tuner code
    tuner_in = *tune;
    send_to_bridge_a(tuner_in);
    recv_from_bridge_b(&tuner_out);
    *tune = tuner_out;
```

```
   // adc code
   clk_enable = 1;
   iterreset  = 1;
   indexreset = 1;
   fiforeset  = 1;
   countreset = 1;

 do
 {
   adc_raw_data (clk_enable, &d0, &d1, &d2, &d3, &d4, &d5, &d6, &d7,
                 &d8, &d9, &d10, &d11, &d12, &d13, &d14, &d15, &fifo_valid);

   // increment every time there is a valid frame
   cg_count_ceil_32 (fifo_valid, 0, fiforeset, 4, &fifocount);
   fiforeset = 0;

   // counter that goes from 0-7
   cg_count_ceil_32 (1, 0, countreset, 7, &count);
   countreset = 0;

   // increment the index every 2268*4 cycles (4 output samples)
   cg_accum_add_32_np (1, iteration == 2268*4, -1, indexreset, &index);
   indexreset = 0;

   // count up to 2268*4 (1 output sample ever 2268 cycles)
   cg_count_ceil_32_np (1, 0, iterreset, 2268*4, &iteration);
   iterreset = 0;

   // collect every 10th data sample
   if (fifo_valid)
   {
       if (fifocount == 0)
       {
           sample0 = d0;  //data value 0
           sample1 = d10; //data value 10
       }
       if (fifocount == 1)
       {
           sample2 = d4;  //data value 20
           sample3 = d14; //data value 30
       }
       if (fifocount == 2) sample4 = d8;  //data value 40
       if (fifocount == 3)
       {
           sample5 = d2;  //data value 50
           sample6 = d12; //data value 60
       }
       if (fifocount == 4) sample7 = d6;  //data value 70
   }

   if (count == 0) fm_in = sample0;
   if (count == 1) fm_in = sample1;
   if (count == 2) fm_in = sample2;
   if (count == 3) fm_in = sample3;
   if (count == 4) fm_in = sample4;
   if (count == 5) fm_in = sample5;
   if (count == 6) fm_in = sample6;
```

```
    if (count == 7) fm_in = sample7;

    // Convert fm_in to a number between -1024 and 1023.
    // When read as an 11-bit fixed point number, with the
    // decimal point at 10, this is a number between -1 and 1.
    fmdemod(fm_in*2-1024, &audio_out);

    // Sample output at 100 MHz / 2268 ~= 44.1 kHz
    if (iteration == 2268*1) x0 = audio_out;
    if (iteration == 2268*2) x1 = audio_out;
    if (iteration == 2268*3) x2 = audio_out;
    if (iteration == 2268*4)
    {
        x3 = audio_out;
        comb_16to64(x3, x2, x1, x0, &al[index]);
    }
  } while (index < n);

    // dma bank a
    nbytes = n * sizeof(int16_t);
    DMA_CPU (OBM2CM, al, MAP_OBM_stripe (1,"A"), res, 1, nbytes, 0);
    wait_DMA (0);
}
```

7. Computational Kernel Example

The ability to use fixed-point arithmetic with a user-defined bit width allows one to save the FPGA resources when compared to the use of 'native' floating-point data types. This, in turn, allows one to implement additional compute logic on the chip, thus shortening the overall calculation time. We demonstrate this in the following application.

In astronomy, the two-point angular correlation function (TPACF), $\omega(\theta)$, encodes the frequency distribution of angular separations, θ between objects on the celestial sphere as compared to randomly distributed objects across the same space [8]. Qualitatively, a positive value of $\omega(\theta)$, indicates that objects are found more frequently at angular separations of θ than expected for randomly distributed objects, and $\omega(\theta)$=0 indicates a random distribution of objects. Precise computation of the TPACF can involve calculation of autocorrelation and cross-correlation components for different angular separations θ for a large number of celestial objects. As an example, the problem of computing the autocorrelation function for this particular application can be expressed as follows:

- Input: Set of points $x_1,\ldots, x_n$ with Cartesian coordinates distributed on the surface of the 3-sphere and a small number b of bins: $[\theta_0, \theta_1), [\theta_1, \theta_2),\ldots, [\theta_{b-1}, \theta_b]$.
- Output: For each bin, the number of unique pairs of points (x_i, xj) for which the dot product is in the respective bin: $B_k = |\{ij: \theta_{k-1} <= x_i \cdot x_j < \theta_k\}|$.

This problem can be solved in $\log(b)(n-1)n/2$ steps by sequentially looping through all unique pairs of points in the data set, computing their dot product, and applying a binary search algorithm to identify the bin the dot product belongs to. The following is the core of the algorithm written in C:

```
for (int i = 0; i < n-1; i++) {
    for (int j = i+1; j < n; j++)  {
```

```
        double dot = x[i] * x[j] + y[i] * y[j] + z[i] * z[j];
        int k, min = 0, max = nbins;
        while (max > min+1) {
            k = (min + max) / 2;
            if (dot >= binb[k]) max = k;
            else min = k;
        };
        if (dot >= binb[min]) bin[min] += 1;
        else if (dot < binb[max]) bin[max+1] += 1;
        else bin[max] += 1;
    }
}
```

The core can be implemented in MAP C as is, however, it will not constitute an efficient FPGA implementation because only the most inner loop used for the binary search algorithm will be pipelined by the MAP C compiler. This inefficiency can be avoided if the number of bins b is known ahead of time and thus the binary search loop can be manually unrolled. As a result, the next most inner loop will be fully pipelined, and thus the entire problem can be solved in just (n-1)n/2 steps. For example, assuming that b<32, the following is an efficient MAP C implementation of the autocorrelation core:

```
for (i = 0; i < n-1; i++) {
    pi_x = x[i]; pi_y = y[i]; pi_z = z[i];
    #pragma loop noloop_dep
    for (j = i+1; j < n; j++) {
        cg_count_ceil_32 (1, 0, j == (i+1), 3, &bank);
        dot = pi_x * x[j] + pi_y * y[j] + pi_z * z[j];
        select_pri_8_32val( (dot < bv31), 31, (dot < bv30), 30,
             (dot < bv29), 29, (dot < bv28), 28, (dot < bv27), 27,
             (dot < bv26), 26, (dot < bv25), 25, (dot < bv24), 24,
             (dot < bv23), 23, (dot < bv22), 22, (dot < bv21), 21,
             (dot < bv20), 20, (dot < bv19), 19, (dot < bv18), 18,
             (dot < bv17), 17, (dot < bv16), 16, (dot < bv15), 15,
             (dot < bv14), 14, (dot < bv13), 13, (dot < bv12), 12,
             (dot < bv11), 11, (dot < bv10), 10, (dot < bv09), 9,
             (dot < bv08), 8,  (dot < bv07), 7,  (dot < bv06), 6,
             (dot < bv05), 5,(dot < bv04), 4,  (dot < bv03), 3,
             (dot < bv02), 2,  (dot < bv01), 1,  0, &indx);
        if (bank == 0) bin1a[indx] += 1;
        else if (bank == 1) bin2a[indx] += 1;
        else if (bank == 2) bin3a[indx] += 1;
        else bin4a[indx] += 1;
   }
}
```

In this implementation, double-precision floating-point arithmetic is used in the calculation of the dot product, point coordinates are stored in the OBM banks, care is taken to avoid read-after-write data dependency, and select_pri_8_32val macro is used to implement a sequence of if/if else statements.

The dataset and random samples used to calculate TPACF in this study are the sample of photometrically classified quasars and the random catalogs first analyzed in this context by [8]. The dataset and each of the random realizations contains 97178 points (n=97178). We use five bins per decade of scale with θ_{min}=0.01 arcminutes and θ_{max}=10000 arcminutes. Thus, angular separations are spread across 6 decades of scale and require 30 bins (b=30). Covering this range of scales requires the use of double-precision floating-point arithmetic as single-precision floating-point numbers are not sufficient to accurately compute θ values smaller than 1 arcminute. However, a closer look at the numerical range of the bin boundaries shows that just 41 bits of the mantissa are sufficient to cover the required range of scales.

Thus, instead of using double-precision floating-point arithmetic, we can use fixed-point arithmetic via macros implemented in Simulink. These macros will replace the dot product calculation and select_pri_8_32val macro as follows:

```
dot_product(pi_x, pi_y, pi_z, x[j], y[j], z[j], &dot);
bin_mapper(dot, bv01, bv02, bv03, bv04, bv05, bv06,
                bv07, bv08, bv09, bv10, bv11, bv12, bv13,
                bv14, bv15, bv16, bv17, bv18, bv19, bv20,
                bv21, bv22, bv23, bv24, bv25, bv26, bv27,
                bv28, bv29, bv30, bv31, bv32, &indx);
```

In this implementation, point coordinates are stored in the OBM banks as before. However, instead of the double-precision floating-point representation we use 48 bits fixed-point representation with 42 bits allocated for the fractional part and 6 bits allocated for the integer part and the sign. The few extra bits in excess of the 41 bits previously mentioned as sufficient to represent the bin boundaries are used to eliminate effects associated with rounding and overflow. The conversion between the point coordinates stored in the double-precision floating-point format and the fixed-point format takes place on the CPU before the data is translated to the OBM banks. The overhead associated with the data type conversion is negligible compared to the overall computational time.

The Simulink-based dot product macro implementation is shown in Figure 16 and the bin mapping macro design is shown in Figure 17 and Figure 18. Table 3 shows reports for loop pipelining, FPGA resource utilization, and place and route time for the complete design. Thus, SLICE utilization is reduced by 22% as compared to the original design. This was expected because 31 double-precision floating-point comparison operators are now replaced with much smaller 48-bits-wide fixed-point comparison operators and a cascade of 1-bit adders.

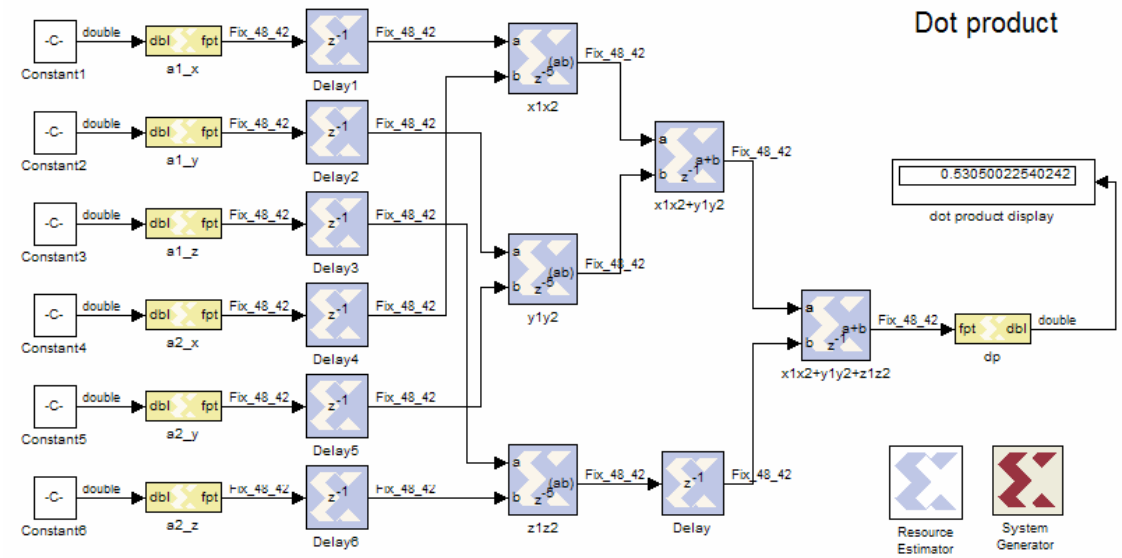

Figure 16. Dot product macro implemented in Simulink.

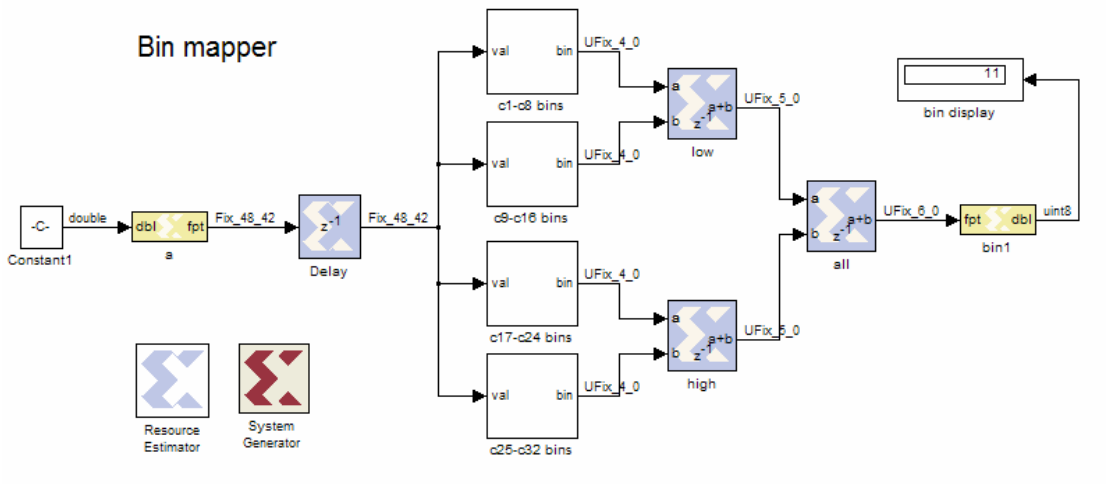

Figure 17. Bin mapping macro implemented in Simulink.

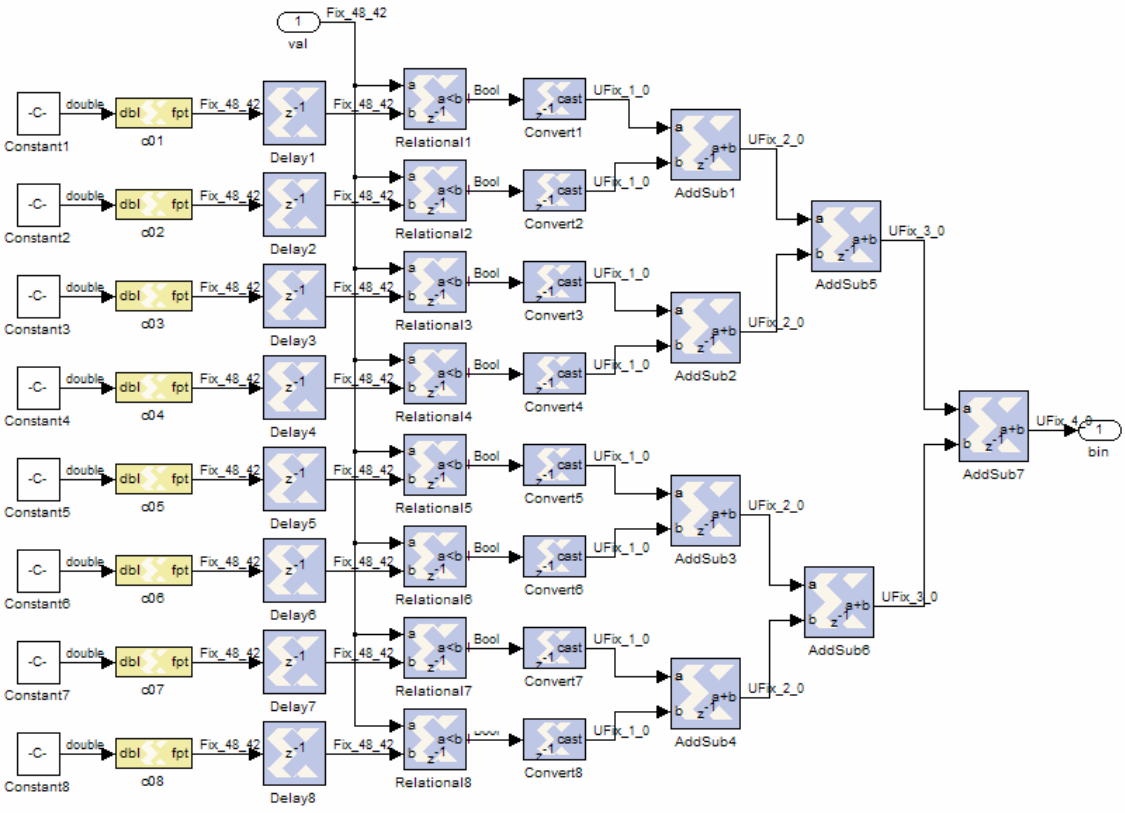

Figure 18. One of four sub-components from the bin mapping macro.

	Original	*Modified*
pipeline depth (clocks)	49	29
MULT18X18s	12%	18%
RAMB16s	5%	5%
SLICEs	63%	41%
PAR time (minutes)	76	25

Table 3. Comparison of two implementations.

Note that MULT18X18s utilization has increased from 12% to 18%. This was not expected; it can be explained, however, by a less efficient implementation of the multipliers in Xilinx Blockset. In addition, the time needed to place and route the entire design on the chip is reduced by a factor of 3. This is mainly due to the smaller overall size of the design.

So, what did we gain by implementing parts of this application in Simulink rather than in native MAP C? When using the native MAP C implementation, we are able to place only two such compute kernels on the FPGA before we run out of available SLICEs. However, when using the Simulink-based kernel implementation, because of the reduced usage of the SLICEs, we are able to place 4 compute kernels per chip, thus achieving twice the performance of the native MAP C implementation.

8. Conclusions

We have demonstrated how a Simulink-based design created with Xilinx System Generator and Xilinx Blockset can be integrated with the native SRC MAP C code. The ability to introduce Simulink-based designs into the Carte framework opens up new possibilities in programming the SRC-6 system. The main advantage of using Simulink-based designs is the ability to use the fixed-point numeric type, which is not directly available in MAP C. This leads to reduced FPGA resource utilization as one can avoid the need to use larger numerical types for problems that require a reduced numerical range. Other benefits include the ability to use low-level FPGA resources (e.g., BRAM) directly and access to Xilinx IP cores, such as FFT and CORDIC algorithms, etc.

We should point out that instead of using Simulink, one can implement the same functionality using an HDL directly. However, using HDL is a more involved process as it requires a set of skills typically possessed by those involved with hardware design, whereas the Simulink environment provides a higher level of abstraction and is more familiar to software developers.

Acknowledgements

This work was funded by the National Science Foundation grant SCI 05-25308. We would like to thank Jon Huppenthal, David Caliga, Dan Poznanovic, and Dr. Jeff Hammes, all from SRC Computers Inc., for their help and support with the SRC-6 system. The TPACF work was performed in collaboration with Dr. Adam Myers and Dr. Robert Brunner from the Department of Astronomy at the University of Illinois at Urbana-Champaign and funded by NASA grants NAG5-12578, NAG5-12580, and NNG06GH15G. Special thanks to Trish Barker from NCSA's Office of Public Affairs for help in preparing this report.

References

[1] SRC Computers Inc., Colorado Springs, CO, SRC Systems and Servers Datasheet, 2005.

[2] http://www.mathworks.com/products/simulink/

[3] http://www.xilinx.com/ise/optional_prod/system_generator.htm

[4] http://www.xilinx.com/products/software/sysgen/blockset.htm

[5] SRC Computers Inc., Colorado Springs, CO, SRC C Programming Environment v 2.2 Guide, 2006.

[6] SRC Computers Inc., Colorado Springs, CO, SRC Systems and Servers Datasheet, 2006.

[7] J. Bernhard, A. Betts, M. Hall, V. Kindratenko, M. Pant, D. Pointer, T. Rimovsky, V. Welch, P. Zawada, and Z. Zhang, Software Defined Radio Project Technical Report, NCASSR technical report, June 1, 2004.

[8] A. Myers, R. Brunner, G. Richards, R. Nichol, D. Schneider, D. Vanden Berk, R. Scranton, A. Gray, and J. Brinkmann, First Measurement of the Clustering Evolution of Photometrically Classified Quasars, The Astrophysical Journal, 2006, 638, 622.

Appendix A: Code Example from Section 3

```
// main.c for the example presented in Section 3

#include <libmap.h>
#include <stdlib.h>

#define SIZE 32
#define BINPT 30
#define WIDTH 40

void my_operator (double A[], double B[], double C[], double Q[], int binpt,
int width, int m, int mapnum);

int main ()
{
    int i;
    double D[SIZE];

    double *A = (double*)Cache_Aligned_Allocate(SIZE * sizeof(double));
    double *B = (double*)Cache_Aligned_Allocate(SIZE * sizeof(double));
    double *C = (double*)Cache_Aligned_Allocate(SIZE * sizeof(double));
    double *Q = (double*)Cache_Aligned_Allocate(SIZE * sizeof(double));

    for (i=0; i<SIZE; i++)
    {
        A[i] = (double)random()/RAND_MAX;
        B[i] = (double)random()/RAND_MAX;
        C[i] = (double)random()/RAND_MAX;
        D[i] = A[i] + B[i];
        D[i] = D[i] * C[i];
    }

    map_allocate(1);

    my_operator(A, B, C, Q, BINPT, WIDTH, SIZE, 0);
```

```
    for (i=0; i<SIZE; i++)
    {
        printf ("(%lf + %lf) * %lf = %lf(map) vs. %lf(cpu) ",
                 A[i], B[i], C[i], Q[i], D[i]);
        if (D[i] != Q[i])  printf ("-error %.30f", D[i]-Q[i]);
        printf ("\n");
    }

    map_free(1);
}
```

```
// src.mc for the example presented in Section 3

#include <libmap.h>

void src (long long, long long, long long, long long *);
long long double2fix (long long, int);
long long fix2double (long long, int);

void my_operator (double A[], double B[], double C[], double Q[], int binpt,
int width, int m, int mapnum)
{
    OBM_BANK_A (AL, long long, MAX_OBM_SIZE)
    OBM_BANK_B (BL, long long, MAX_OBM_SIZE)
    OBM_BANK_C (CL, long long, MAX_OBM_SIZE)
    OBM_BANK_D (DL, long long, MAX_OBM_SIZE)
    int i;

    long long a, b, c, q;

    DMA_CPU (CM2OBM, AL, MAP_OBM_stripe (1, "A"), A, 1, m*8, 0);
    wait_DMA (0);

    DMA_CPU (CM2OBM, BL, MAP_OBM_stripe (1, "B"), B, 1, m*8, 0);
    wait_DMA (0);

    DMA_CPU (CM2OBM, CL, MAP_OBM_stripe (1, "C"), C, 1, m*8, 0);
    wait_DMA (0);

    for (i=0; i<m; i++)
    {
      a = double2fix (AL[i], binpt);
      b = double2fix (BL[i], binpt);
      c = double2fix (CL[i], binpt);

      src (a, b, c, &q);

      DL[i] = fix2double (q, binpt, width);
    }

    DMA_CPU (OBM2CM, DL, MAP_OBM_stripe (1, "D"), Q, 1, m*8, 0);
    wait_DMA (0);
}

#include <double2fix.mc>
#include <fix2double.mc>
```

```
# Makefile for the example presented in Section 3

FILES       = main.c
MAPFILES    = src.mc
BIN         = src

MACROS      = macros/src_clk_wrapper.vhd
MY_BLKBOX   = macros/blk.v
MY_NGO_DIR  = macros
MY_INFO     = macros/info

MY_MCCFLAGS       = -inline=name:double2fix,fix2double:1
MY_MFTNFLAGS      =

CC          = icc
FC          = ifort
LD          = icc

MY_CFLAGS   =
MY_FFLAGS   = -w95

MAKIN   ?= $(MC_ROOT)/opt/srcci/comp/lib/AppRules.make
include $(MAKIN)
```

Appendix B: Floating point and fixed point conversion in C

```
/* float2fix.c
   input: flp - a 32-bit floating point number
          binpt - the placement of the decimal point for the fixed point number
   output : a signed 32-bit integer holding the fixed point representation of the
            floating point  number
*/
int float2fix (float flp, int binpt)
{
    if (binpt == 31 && flp == -1) return (1 << 31);

    union {float f; int i;} temp;
    temp.f = flp;

    int sign, exp, man, fixed;

    if (temp.f == 0) return 0;

    sign = temp.i >> 31;
    exp = ((temp.i >> 23) & 0xFF) - 127;
    man = temp.i & 0x7FFFFF;

    if (binpt-(23-exp) > 0)
        fixed = (man | 0x800000) << (binpt-(23-exp)) & 0x7FFFFFFF;
    else
        fixed = (man | 0x800000) >> ((23-exp)-binpt);

    if (sign < 0) fixed = ~fixed+1;

    return fixed;
}
```

```
/* double2fix.c
   input: flp - a 64-bit floating point number
            binpt - the placement of the decimal point for the fixed point number
   output : a signed 64-bit integer holding the fixed point representation of the
            floating point number
*/
long long double2fix (double flp, int binpt)
{
    if (binpt == 63 && flp == -1) return ((long long)1 << 63);

    union {double d; long long l;} temp;
    temp.d = flp;

    long long sign, exp, man, fixed;

    if (temp.d == 0) return 0;

    sign = temp.l >> 63;
    exp = ((temp.l >> 52) & 0x7FF) - 1023;
    man = temp.l & 0xFFFFFFFFFFFFF;

    if (binpt-(52-exp) > 0)
        fixed = (man | 0x10000000000000) << (binpt-(52-exp)) & 0x7FFFFFFFFFFFFFFF;
    else
        fixed = (man | 0x10000000000000) >> ((52-exp)-binpt);

    if (sign < 0)
        fixed = ~fixed+1;
    return fixed;
}
```

```
/* fix2float.c
   input: fixed - a signed 32-bit integer holding a fixed point number
          binpt - the placement of the decimal point for the fixed point number
          width - the bit width of the fixed point number
   output: a float holding the floating point representation of the fixed point number
*/
float fix2float (int fixed, int binpt, int width)
{
    union {float f; int i;} flp;
    int sign = 0;
    int exp = 127;
    int man;

    if ((binpt == 31) && (fixed == 1<<31)) return -1;
    else if ((fixed >> (width-1)) & 1)
    {
        if (width != 32) fixed = fixed | (-1 << width);
        fixed = ~fixed+1;
        sign = 1;
    }
    man = fixed;

    if (binpt == 31)
    {
        binpt--;
        exp--;
    }

    if (fixed == 0) exp = 0;
```

```
    else if ((fixed >> binpt) > 1)
    {
        while ((fixed >> binpt) > 1)
        {
            fixed = fixed >> 1;
            exp++;
        }
    }
    else
    {
        while ((fixed >> binpt) < 1)
        {
            fixed = fixed << 1;
            exp--;
        }
    }

    if ((man >> 23) < 1 && man != 0)
    {
        while (man >> 23 < 1)
            man = man << 1;
    }
    else
    {
        while ((man >> 23) > 1)
            man = man >> 1;
    }

    man = man & 0x7FFFFF;
    flp.i = (sign << 31) | (exp << 23) | man;
    return flp.f;
}
```

```
/* fix2double.c
   input: fixed – a signed 64-bit integer holding a fixed point number
          binpt - the placement of the decimal point for the fixed point number
          width - the bit width of the fixed point number
   output: a double holding the floating point representation of
           the fixed point number
*/
double fix2double (long long fixed, int binpt, int width)
{
    union {double d; long long l;} flp;
    long long sign = 0;
    long long exp = 1023;
    long long man;

    if ((binpt == 63) && (fixed == (long long)1<<63)) return -1;
    else if ((fixed >> (width-1)) & 1)
    {
        if (width != 64)
            fixed = fixed | ((long long)-1 << width);
        fixed = ~fixed+1;
        sign = 1;
    }
    man = fixed;

    if (binpt == 63)
    {
        binpt--;
```

```
        exp--;
    }

    if (fixed == 0) exp = 0;
    else if ((fixed >> binpt) > 1)
    {
        while ((fixed >> binpt) > 1)
        {
            fixed = fixed >> 1;
            exp++;
        }
    }
    else
    {
        while ((fixed >> binpt) < 1)
        {
            fixed = fixed << 1;
            exp--;
        }
    }

    if ((man >> 52) < 1  && man != 0)
    {
        while (man >> 52 < 1)
            man = man << 1;
    }
    else
    {
        while ((man >> 52) > 1)
            man = man >> 1;
    }

    man = man & 0xFFFFFFFFFFFFF;
    flp.l = (sign << 63) | (exp << 52) | man;
    return flp.d;
}
```

Appendix C: floating point and fixed point conversion in MAP C

```
/* float2fix.mc
   input: flp - a 32-bit floating point number
          binpt - the placement of the decimal point for the fixed point number
   output : a signed 32-bit int holding the fixed point representation of
            the floating point number
*/
int float2fix(int flp, int binpt)
{
    int  sign, exp, man, fixed;

    if (binpt == 31 && flp == 0xBF800000) return (1 << 31);
    if (flp == 0) return 0;

    sign = flp >> 31;
    exp = ((flp >> 23) & 0xFF) - 127;
    man = flp & 0x7FFFFF;

    if (binpt-(23-exp) > 0) fixed = (man | 0x800000) << (binpt-(23-exp)) & 0x7FFFFFFF;
    else fixed = (man | 0x800000) >> ((23-exp)-binpt);
```

```
    if (sign < 0) fixed = ~fixed+1;
    return fixed;
}
```

```
/* double2fix.mc
   input: flp - a 64-bit floating point number
          binpt - the placement of the decimal point for the fixed point number
   output : a 64-bit integer holding the fixed point representation of
            the floating point number
*/
long long double2fix(long long flp, int binpt)
{
    long long sign, exp, man, fixed;

    if (binpt == 63 && flp == 0xBFF0000000000000) return ((long long)1 << 63);
    if (flp == 0) return 0;

    sign = flp >> 63;
    exp = ((flp >> 52) & 0x7FF) - 1023;
    man = flp & 0xFFFFFFFFFFFFF;

    if (binpt-(52-exp) > 0)
        fixed = (man | 0x10000000000000) << (binpt-(52-exp)) & 0x7FFFFFFFFFFFFFFF;
    else
        fixed = (man | 0x10000000000000) >> ((52-exp)-binpt);

    if (sign < 0) fixed = ~fixed+1;
    return fixed;
}
```

```
/* fix2float.mc
   input: fixed - a 32-bit value holding a fixed point number
          binpt - the placementof the decimal point for the fixed point number
          width - the bit width of the fixed point number
   output: a 32-bit integer holding the floating point representation of
           the fixed point number
*/

int fix2float(int fixed, int binpt, int width)
{
    int exp = 127;
    int man = fixed;
    int sign = 0;

    if ((binpt == 31) && (fixed == 1 << 31)) return 0xBF800000;
    else if ((fixed >> (width-1)) & 1)
    {
        if (width != 32) fixed = fixed | (-1 << width);
        fixed = ~fixed+1;
        sign = 1;
    }

    man = fixed;

    if (binpt == 31)
    {
```

```
        binpt--;
        exp--;
    }

    if (fixed == 0) exp = 0;
    else if ((fixed >> binpt) > 1)
    {
        while ((fixed >> binpt) > 1)
        {
            fixed = fixed >> 1;
            exp++;
        }
    }
    else if ((fixed >> binpt) < 1)
    {
        while ((fixed >> binpt) < 1)
        {
            fixed = fixed << 1;
            exp--;
        }
    }

    if ((man >> 23) < 1 && man != 0)
    {
        while (man >> 23 < 1)
            man = man << 1;
    }
    else
    {
        while ((man >> 23) > 1)
            man = man >> 1;
    }

    man = man & 0x7FFFFF;
    return (sign << 31) | (exp << 23) | man;
}
```

```
/* fix2double.mc
   input: fixed - a signed 64-bit value holding a fixed point number
          binpt - the placement of the decimal point for the fixed point number
          width - the bit width of the fixed point number
   output: a 64-bit integer holding the floating point representation of
           the fixed point number
*/

long long fix2double(long long fixed, int binpt, int width)
{
    long long exp = 1023;
    long long man = fixed;
    long long sign = 0;

    if ((binpt == 63) && (fixed == (long long)1<<63)) return 0xBFF0000000000000;
    else if ((fixed >> (width-1)) & 1)
    {
        if (width != 64) fixed = fixed | ((long long)-1 << width);
        fixed = ~fixed+1;
        sign = 1;
    }

    if (binpt == 63)
```

```
    {
        binpt--;
        exp--;
    }

    if (fixed == 0) exp = 0;
    else if ((fixed >> binpt) > 1)
    {
        while ((fixed >> binpt) > 1)
        {
            fixed = fixed >> 1;
            exp++;
        }
    }
    else
    {
        while ((fixed >> binpt) < 1)
        {
            fixed = fixed << 1;
            exp--;
        }
    }

    if ((man >> 52) < 1 && man != 0)
    {
        while (man >> 52 < 1)
            man = man << 1;
    }
    else
    {
        while ((man >> 52) > 1)
            man = man >> 1;
    }

    man = man & 0xFFFFFFFFFFFFF;
    return (sign << 63) | (exp << 52) | man;
}
```