In Proc. 9th *Military and Aerospace Programmable Logic Devices (MAPLD) International Conference*
September, 2006, Washington, DC.

On Using Simulink to Program SRC-6 Reconfigurable Computer

David Meixner, Volodymyr Kindratenko[§], David Pointer
National Center for Supercomputing Applications (NCSA), University of Illinois at Urbana-Champaign (UIUC)
E-mail addresses: dmeixner@uiuc.edu, kindr@ncsa.uiuc.edu, pointer@ncsa.uiuc.edu

Abstract

By design, the SRC-6 reconfigurable computer is programmed in the MAP C programming language within the framework provided by the SRC Carte™ development environment. The functionality of the original language can be extended via third party subroutines, called macros. These macros, typically implemented in Verilog Hardware Description Language, are brought into the MAP C program via configuration files that define the interface between the macros and the MAP C language. In this paper, we describe a process of using the Verilog source for SRC macros generated from the MathWorks Simulink® designs built using Xilinx System Generator™ for DSP and Xilinx Blockset. We also describe an example application that takes advantage of this programming model.

Introduction

In this paper we describe a 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 Verilog source for SRC macros generated from the MathWork's Simulink-based designs.

The rest of the paper is organized as follows. First, we provide a brief overview of the SRC-6 reconfigurable computer and its programming model. Next, we present steps necessary to integrate a Simulink-based design with MAP C source code (part of this work also appears in [6]). We briefly discuss issues related to numerical accuracy of using fixed-precision arithmetic instead of floating-point arithmetic. We then present an application example which shows the benefits of using Simulink instead of the native MAP C implementation.

SRC-6 reconfigurable computer

The SRC-6 MAPstation used in this work consists of a dual-CPU Xeon board, one MAP Series C and one MAP Series E processor, and an 8 GB common memory module, all interconnected with a 1.4 GB/s low latency Hi-Bar™ switch (Figure 1). The SNAP™ Series B interface board is used to connect the CPU board to the Hi-Bar switch.

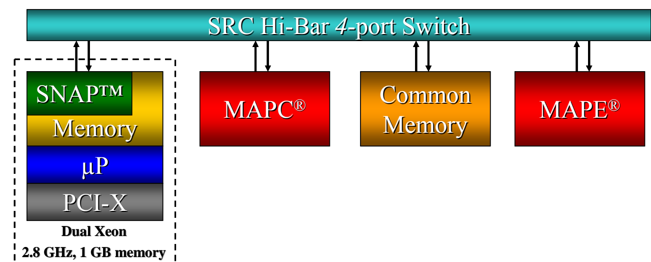

Figure 1: SRC-6 MAPstation used in the course of this study.

The MAP Series C processor module consists of two user FPGAs, one control FPGA, and memory (Figure 2). There are six banks (A-F) of on-board memory (OBM); each bank is 64 bits wide and 4 MB deep for a total of 24 MB. There is an additional 4 MB of dual-ported memory dedicated to data transfer between the two FPGAs. The two user FPGAs in the MAP Series C are Xilinx Virtex-II XC2V6000 FPGAs. The FPGA clock rate of 100 MHz is set from within the SRC programming environment. The MAP Series E processor module is identical to the Series C module with the exception of the user FPGAs: The two user FPGAs in the MAP Series E are Xilinx Virtex-II Pro XC2VP100 chips.

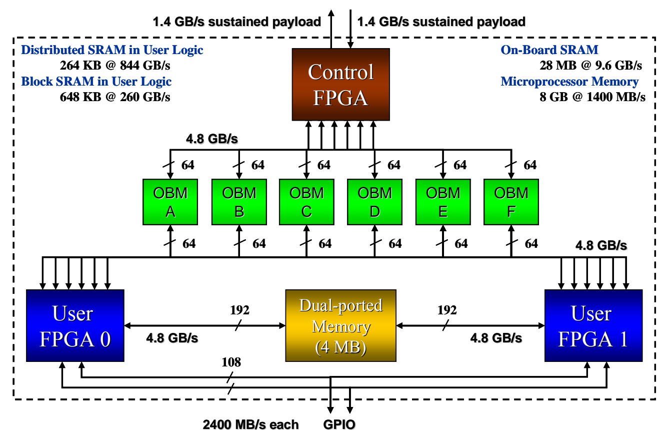

Figure 2: MAP Series C processor module.

[§] Corresponding author

Software for the SRC MAPstation is developed in the MAP C programming language using the Carte™ version 2.1 programming environment. The Intel C (icc) version 8.1 compiler is used to generate the CPU side of the combined CPU/MAP executable. The SRC MAP C compiler produces the hardware description of the FPGA design for our final, combined CPU/MAP target executable. This intermediate hardware description of the FPGA design is passed to Xilinx ISE place and route tools to produce the FPGA bit file. Finally, the linker is invoked to combine the CPU code and the FPGA hardware bit file into a unified executable. Figure 3 provides an overview of the code development process.

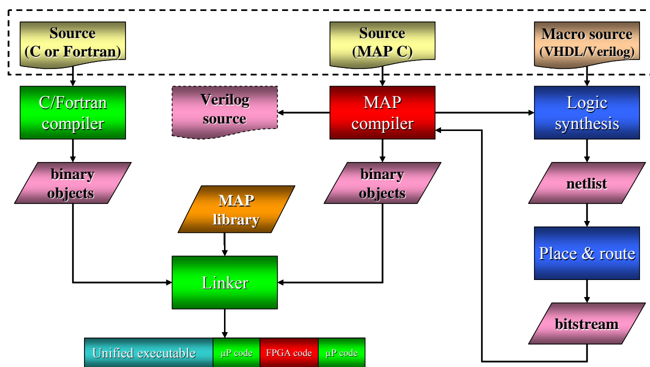

Figure 3: SRC-6 code development process.

Functionality of the MAP C language can be extended via third-party user macros written in an HDL, such as Verilog, and integrated with the Carte programming environment via a **black box** file that defines the interface information for all of the user macros and an **info** file that establishes the mapping between operators and calls in the source program and the macros and signal names in the Verilog code generated by the MAP compiler. We use this feature of the Carte development environment to bring in Verilog source generated from Simulink-based designs.

MathWorks Simulink model design

The Simulink model one wishes to incorporate into SRC-6's Carte framework should be created using the Xilinx Blockset for Simulink. Figure 4 shows a simple Simulink example which takes three inputs: *a*, *b*, and *c*, and outputs *q=(a+b)*c*. For this example, all signals have a bit width of 40 and a binary point of 30. The input and output ports are '*gateway in*' and '*gateway out*' blocks, respectively. They should be labeled in lowercase letters using the same names as the variables to be used in the resulting macro. The '*gateway in*' blocks also allow one to specify the bit width and binary point of the inputs; however, they do not perform the actual data type conversion. The programmer is responsible for performing the required data type conversion; more on this follows

Once the Simulink model is created, the next step is to set up and run the System Generator. Its parameters should be set up as follows:

- Compilation Type: HDL Netlist
- Part: the FPGA to be used, e.g., Virtex2 xc2v6000-4ff1517 for MAP Series C processor
- Target Directory: the directory where the Verilog files will be outputted
- Hardware Description Language: Verilog
- FPGA Clock period (ns): 10

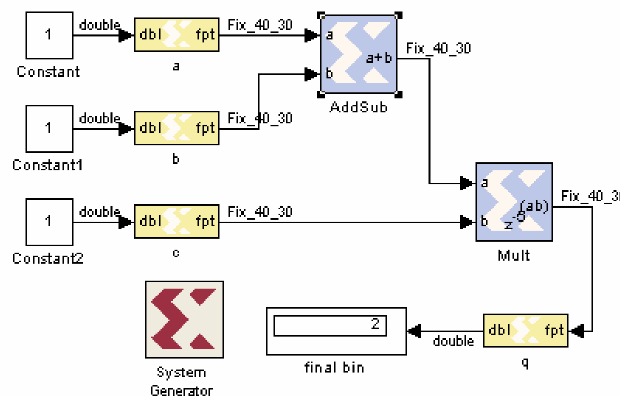

Figure 4: Simple Simulink Example, addmult.mdl.

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

- <*design*>_files.v – contains most of the HDL for the design.
- <*design*>_clk_wrapper.v – an HDL wrapper that drives clocks and clock enables.
- conv_pkg.v – contains constants and functions used by <*design*>_fils.v.
- .edn files – implementation of the parts of the design.

In the above example, the following files are produced: addmult_files.v, addmult_clk_wrapper.v, conv_pkg.v, adder_subtracter_virtex2_7_0_84f1dba84ee809b9.edn, and multiplier_virtex2_7_0_b018b3a1b259a550.edn. These files need to be copied into the macro directory of the target MAP C implementation.

Integration with Carte framework

First, <*design*>_clk_wrapper.v file needs to be modified to include <*deisgn*>_files.v in addition to conv_pkg.v:

```
--System Generator code here--

`include "conv_pkg.v"
`include "addmult_files.v" /* line added manually */

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

--System Generator code here--
```

Note that some Xilinx blocks (i.e. FIR filters) cannot be generated using Verilog, so VHDL has to be used instead. When using VHDL source instead of Verilog, one needs to include the <design>_clk_wrapper.prj file in the macros directory instead of adding the "`include <design>_files.v" line, and modify the following line:

```
"set_option -disable_io_insertion false"
```

so that it is set to "true".

Next, a black box definition for the design is created. It includes the inputs and outputs defined in the Simulink model, as well as the 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 definition appears as follows (remember that we are using 40-bit fixed-point numbers):

```
module addmult_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 links 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. Lastly, the debug function can be included as well. The resultant info file is as follows:

```
BEGIN_DEF "addmult"
   MACRO = "addmult_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";

END_DEF
```

Next, we add the path for <*design*>_clk_wrapper.v to the MACROS line of the Makefile. For this example, the line would read:

```
MACROS = macros/addmult_clk_wrapper.v
```

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);
```

Now the macro can be called from MAP C code just as any other macro:

```
addmult(a, b, c, &q);
```

Numerical conversion

Since Simulink performs all calculations with fixed-point arithmetic, floating-point numbers must be converted to fixed-point representation in order to take advantage of variable resolution. This conversion can occur either on the microprocessor side before the data is transferred from the main memory to the MAP or on the MAP before the data is used by the Simulink-based design. We have implemented subroutines both in C for the execution on the microprocessor and in MAP C for the inclusion in the MAP's algorithm implementation. Appendix A includes C subroutines that convert between the floating-point and fixed-point representations.

When converting between floating-point and fixed-point numerical representations, there can be a loss of numerical resolution. The size (number of bits) and the binary point of the fixed-point numbers determine its *range* and *precision*, so a larger range or higher precision requires more bits to be used. This is a tradeoff that must be taken into consideration by the programmer. For instance, in the above example the variables are 40 bits wide with the decimal point placed after the 30th bit. This allows for an accuracy of 9.31322574615478515625e-10 and a maximum range of just under $\pm 2^{10}$, since there are 10 bits for the integer and 30 bits for the decimal.

Example application

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 [7]. 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, x_j) 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 [7]. 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 5 and the bin mapping macro design is shown in Figure 6 and Figure 7. Table 1 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.

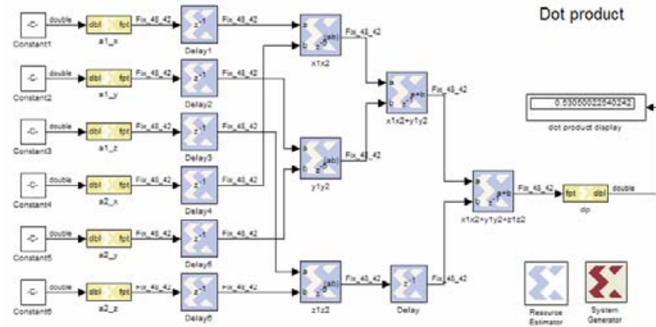

Figure 5: Dot product macro implemented in Simulink.

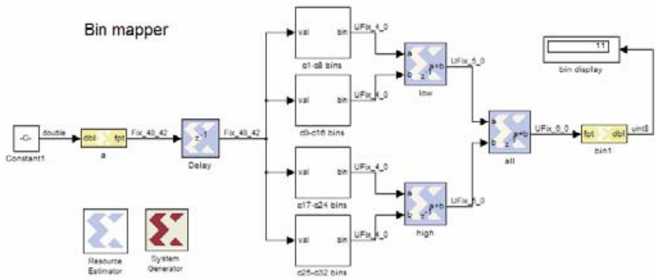

Figure 6: Bin mapping macro implemented in Simulink.

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

Table 1. Comparison of two implementations.

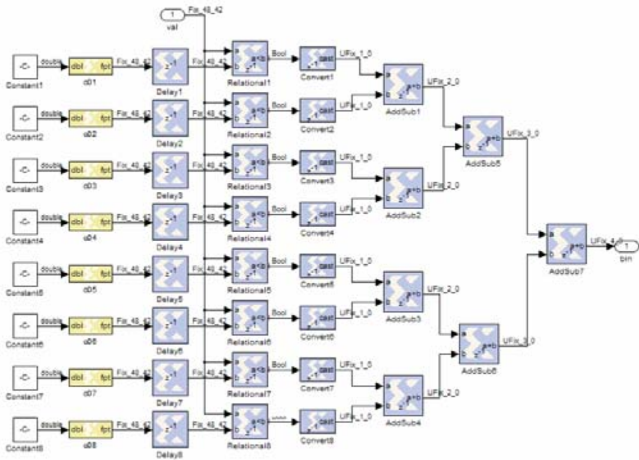

Figure 7: One of four sub-components from the bin mapping macro.

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? It turns out that 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.

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, although not explored in this paper, 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 publication.

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 2.1 Guide, 2005.

[6] D. Meixner, V. Kindratenko, D. Pointer, Running Simulink-based Designs on SRC-6, The High Performance Embedded Computing (HPEC'06).

[7] 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

```
/* double2fix
    input: flp - a 64-bit floating-point number
           binpt - the placement of the decimal point
    output: a 64-bit integer holding the fixed-point representation
*/

long long double2fix (double flp, int binpt) {

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

    union {double d; long long l;} temp;
    long long sign, exp, man, fixed;

    temp.d = flp;
    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;

}



/* fix2double
    input: fixed - a 64-bit integer holding a fixed-point number
           binpt - the placement of the decimal point
           width - the bit width of the fixed-point number
    output: a double holding the floating-point representation
*/

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;

}
```