sc2 Reference Manual

Maya Gokhale
Los Alamos National Laboratory
MSD440
Los Alamos, NM 87545
maya@lanl.gov

Abstract

sc2 is a new implementation of the Streams-C[3] language and compiler. This manual describes the Streams-C language and sc2 compiler structure. The Streams-C programming model is that of communicating processes. A system consists of a collection of processes that communicate using streams and signals. Processes can run either in software on conventional processors (SP) or in hardware on FPGA processors (HP). Processes (and their associated stream and signals) can be created dynamically by other software processes.

The sc2 compiler, which consists of several passes using the Stanford University Intermediate Format (SUIF) infrastructure, is used to compile FPGA processes. The compiler translates a subset of C into Register-Transfer-Level (RTL) VHDL that is synthesizable on FPGAs.

The sc2 compiler passes are freely available for noncommercial use in source form from Los Alamos National Laboratory. Please contact the authors for more information.

1. Introduction

This manual describes the Streams-C language and compiler. sc2 is a new implementation of Streams-C [3], a parallel language following the communicating processes model. The language consists of a small collection of additions to C in the form of new data types and intrinsic calls. The sc2 compiler translates the C program into Register-Transfer-Level (RTL) VHDL that can be synthesized to FPGAs.

In the manual we describe the language extensions, define the subset of C that we can translate to VHDL, and sketch the compiler implementation. sc2 defines *processes* that communicate data over *streams* and events via *signals*. A process may be a software process (SP) on a conventional processor or a hardware process (HP) on an FPGA. The sc2 compiler is used to translate HP processes to VHDL. SP processes, which use a software library included with sc2, are compiled by the native C compiler of the system.

The original Streams-C compiler targeted the Annapolis MicroSystems Wildforce FPGA board. The sc2 compiler release provides hardware libraries for the AMS Firebird, which contains a single Xilinx Virtex E on a 64-bit PCI bus. The sc2 software library is based on Linux.

1.1. Scope of this Reference Manual

The purpose of this reference manual is to present the unique features of Streams-C, along with examples of its use. This reference guide is *not* intended to be a reference on the ANSI C programming language. Also, it is assumed that the reader is familiar with parallel programming and the communicating process model.

1.2. Conventions

All C syntax conventions are followed for Streams-C. The code appearing in the body of a Streams-C process does not introduce any non-C syntax. New constructs appear in the form of predefined types and intrinsic functions and in directives (in comment blocks) to the sc2 compiler.

Whenever a Streams-C construct – predefined type, intrinsic function, etc. is referenced – it will be shown in **bold face**, with all C and user-supplied information interspersed in the construct in regular type. All sc2 predefined types and intrinsic functions have the preface **sc_** as part of their name.

An explanation of language features that are restricted in the current implementation of the compiler are identified by '**(I)**' preceding or following the manual text, as appropriate. Examples of *illegal code* are preceded by an *italic* warning stating that the example is illegal.

A loose BNF notation is used to describe syntax. When a category in angle brackets contains the string "name," it must be a C identifier, eg. <stream_name> denotes the name of a stream, and the name must conform to C identifier format.

2. Programming Model

The concept of using Field Programmable Gate Arrays (FPGAs) as customizable compute engines began in the late 1980's, and since then, many realizations of that concept have been developed that have delivered the promised performance acceleration. However, that 10x-100x of performance that can be obtained for suitable applications on Reconfigurable Computers (RCCs) has been at the cost of 10x-100x increase in difficulty of application development. FPGA-based systems offer the programmability of software, allowing a vast number of applications and application variants to be mapped onto them. Despite many promising research efforts, the mainstream of application development must use Computer-Aided Design (CAD) tools that are oriented towards hardware rather than software development, characterized by high cost of the tool set, long compile times if a reasonable level of abstraction and portability are desired, and most important, the necessity of developing and completely understanding the cycle-by-cycle behavior of millions of gates spanning multiple FPGA chips and fixed function units.

In recent years, a concerted effort has been launched to remedy the design tool problem. Many of these tools are embedded in high level design simulation environments (Ptolemy, Khoros, MatLab, Handel C)([7], [8])or target "dusty deck" sequential procedural code ([4]), while others target low level, technology-specific, optimized designs ([1], [6]).

The sc2 language represents an intermediate approach between those very high level and very low level design tools. Our target machine is an attached parallel processor such as the Annapolis Microsystems Firebird. These PCI or VME accelerators sit on the I/O bus of a conventional workstation or PC. They include multiple FPGAs interconnected by both fixed and programmable resources. The FPGAs have access to local or shared SRAM chips, and have some relatively slow method of communicating with the workstation.

With current compiler technology, parallelization of the application and mapping to the FPGA board architecture are best performed by the application developer. This is in keeping with methods of programming conventional parallel machines, in which the application developer usually manually parallelizes the program and inserts message-passing and synchronization logic. However, it is our thesis that the application developer should not have to be a hardware designer in order to develop reasonably efficient programs, that clock-cycle-level of specification should not be required. With this middle approach, software engineers knowledgeable in parallel programming can create applications on FPGA-based processors.

The sc2 model embodies the above design goals. Our programming model is targeted at stream-oriented FPGA applications. Characteristics of stream-oriented computing include high-data-rate flow of one or more data sources; fixed size, small stream payload (one byte to one word); compute-intensive operations, usually low precision fixed point, on the data stream; access to small local memories holding coefficients and other constants; and occasional synchronization between computational phases.

The sc2 language is actually a small set of annotations and library functions callable from a conventional C program. The annotations are used to declare a **process**, **stream**, or **signal**, and to assign resources on the FPGA board to those objects. The library functions are used to communicate stream data between the processes.

sc2 follows the Communicating Sequential Processes (CSP) [5] parallel programming model. The implementation is a combination of annotations and library functions callable from C. This is for pragmatic reasons as our compiler is built within the framework of the SUIF compiler infrastructure, which best supports C and Fortran. In our model, there are three distinguished objects: processes, streams, and signals. A process is an independently executing object with a process body (the "run function") that is given by a C subroutine. A process can run on a conventional processor or on an FPGA chip.

```
/// PROCESS_FUN <function_name>
/// IN_STREAM <stream_element_data_type_name> <stream_name>
... other input streams ...
/// OUT_STREAM <stream_element_data_type_name> <stream_name>
... other output streams ...
/// IN_SIGNAL <signal_element_data_type_name> <signal_name>
... other input signals ...
/// OUT_SIGNAL <signal_element_data_type_name> <signal_name>
... other output signals ...
/// PARAM <parameter_type_name> <parameter_name>
... one parameter declaration per process ...
/// PROCESS_FUN_BODY
...  C code ...
/// PROCESS_FUN_END
```

Figure 1. Format of a Streams-C Run Function

An FPGA process must be written in a subset of C (defined below in Section 6). In addition, intrinsic functions to perform stream or signal operations may be referenced. Processes may be initiated dynamically during execution. A process runs until it exits with a return statement, control reaches the end of the process body, or it is terminated by its initiating process.

The sc2 compiler synthesizes hardware circuits for one or more FPGAs as well as a set of communicating processes on conventional processors. The compiler includes previously reported features ([2]) extended to pipelined stream computation, so that the generated hardware/software is capable of pipelining a computation across multiple FPGAs and the conventional processor. Our system includes a functional simulation environment, allowing the programmer to simulate the collection of parallel processes and their communication at the functional level.

3. Declaring Processes, Streams, and Signals

3.1. Graphical Description

3.2. Textual Description

This section describes the format for describing the specific processes, streams, and signals that a Streams-C program uses. These directives are embedded in specially formatted blocks. Each directive must be on one line. Each directive is prepended by "///" starting with the first character of the line. Next, a keyword identifying the directive must appear, followed by parameter(s) to the directive.

The first set of directives describe the "run function" of a process. This is the body of code that gets executed when the associated process is initiated. The PROCESS_FUN directive gives a name to the run function, input and output streams and signal parameters, followed by an optinal parameter to be passed to the process when it is initiated. After the parameter, the body of the function appears as normal C code, usually containing variable declarations, stream and/or signal communication, and computation. A keyword directive is used to mark the end of the run function.

The function name is a C identifier. The stream and signal names are also identifiers and can be used within stream operations within the body of the C code. The data type of stream or signal elements precedes the name of the stream or signal as in normal C syntax. A single parameter to the process is also permitted. The format of the PROCESS_FUN directive is shown in Figure 1.

To describe a process to Streams-C, the PROCESS directive is used. A process has an associated run function and is of type "SP" (software process) or "HP" (hardware process). The PROCESS directive optionally may be used to declare a 1-dimensional array of processes.

The use of <array_spec> means that the system contains <integer> number of processes. Arrays of processes are often useful to describe systolic computation. The type of each process must be given as either SP for software process or HP for hardware process. If omitted, SP is assumed. The optional ON clause maps the process onto a specific resource of the system

```
/// PROCESS  <process_name> [<array_spec>] PROCESS_FUN <process_fun_name>
       [TYPE [SP | HP]] <on_spec>
<array_spec> ::= '[' <integer> ']'
<on_spec>    ::= [ ON <resource_name> ]
```

Figure 2. Format of a Streams-C Process Directive

```
/// CONNECT <process_name> [<process_ref1>.<port>] [ <fifo_spec> ] \
            <process_name> [<process_ref2>.<port>] [ <fifo_spec> ]

<process_ref1> ::=  '[' <integer> ']' |
                  '[' <integer> .. <integer> ']'

<process_ref2> ::=  '[' <integer> ']' |
                  '[' '!' [+|-] <integer> ']'

<port>        ::= stream or signal name from a PROCESS_FUN directive

<fifo_spec>   ::= FIFO_SIZE <uint>
```

Figure 3. Format of a Streams-C CONNECT Directive

such as a specific FPGA chip or processor. The default for a software process is sc_host; default for hardware processes is FPGA. Figure 2 shows the format of the PROCESS directive.

The last directive CONNECT is used to connect processes via streams and signals. To connect two processes, the name of one process's stream or signal is associated with the name of another process's stream or signal. In Figure 3, the stream or signal formal parameter defined in the PROCESS_FUN directive is generically referred to as a port.

(I) The CONNECT directive must be specified from "source" to "destination."
The <process_ref> is the name of a process that has been declared in a PROCESS directive. If the name denotes an array of processes, a subscript may be used to select a single process instance, a range of process instances, or a process instance relative to other instances.

The relative notation "[! + integer]" or "[! - integer]" is used in conjunction with a range of process instances. The "!" is used in the second process reference and takes on each value in the range specified by the first process instance. This notation is useful for connecting processes in a systolic array. For an example of the use of this directive to declare and connect an array of processes, see Figure 4.

(I) Do not use external memory in an array of hardware processes. The Streams-C model does not arbitrate external memory access between multiple hw processes. Choose local memory such a block ram or CLB ram. Local memory is not implemented for arrays of processes yet.

The <fifo_spec> is used to set the size of the FIFOs at the sender and receiver respectively. If omitted, 16-element FIFOs are used. Allowable fifo depths are 16, 32 and 64.

(I) the connections between processes must be one-to-one. Broadcast patterns are not supported. Many to one connections are not supported.

In this example there are two software processes called setup and finish, and 10 instances of a hardware process p. The first instance of p receives stream data from setup. Instances 1 through 9 receive data from the previous instance. The ninth instance outputs data to the finish process.

4. Predeclared Integer Data Types

Streams-C provides predefined unsigned and signed integer data types for selected bit lengths ranging from 1 to 128, as shown in Figure 5. The bit lengths we support are 1, 2, 4, 6, 8, 12, 16, 18, 20, 24, 32, 40, 48, 64, 128. A simple convention is used to name these predefined types. Signed types have the name **sc_int**<bit_length>. Unsigned types have the name

```
/// PROCESS_FUN setup_run
/// OUT_STREAM sc_uint4 data
/// PROCESS_FUN_BODY
...  beginning of C code ...
/// PROCESS_FUN_END

/// PROCESS_FUN finish_run
/// IN_STREAM sc_uint4 processed_data
/// PROCESS_FUN_BODY
...  beginning of C code ...
/// PROCESS_FUN_END

/// PROCESS_FUN p_run
/// IN_STREAM sc_uint4 str1
/// OUT_STREAM sc_uint8 str2
/// PROCESS_FUN_BODY
...  beginning of C code ...
/// PROCESS_FUN_END

/// PROCESS setup PROCESS_FUN setup_run

/// PROCESS p [10] PROCESS_FUN p_run TYPE HP

/// PROCESS finish PROCESS_FUN finish_run

/// CONNECT p[0].str1 setup.data
/// CONNECT p[1 .. 9].str1  p[!-1].str2
/// CONNECT p[9].str2    finish.processed_data
```

Figure 4. CONNECT Directives Example

Signed	Unsigned
	sc_uint1
sc_int2	sc_uint2
sc_int4	sc_uint4
sc_int6	sc_uint6
sc_int8	sc_uint8
sc_int12	sc_uint12
sc_int16	sc_uint16
sc_int18	sc_uint18
sc_int20	sc_uint20
sc_int24	sc_uint24
sc_int32	sc_uint32
sc_int40	sc_uint40
sc_int48	sc_uint48
sc_int64	sc_uint64
sc_int128	sc_uint128

Figure 5. Streams-C Data Types

sc_uint<bit_length>. Variables of these types may be used in a Streams-C program. A stream may have one of these Streams-C integer types as its data element type (see Section 5.2).

Note: Currently we do not implement all the sc_int types, since the ART library cannot be used reliably until it is made threadsafe. We currently declare all the types except for 40, 48 and 128 bits. The 1, 2, 4, 6, 12, 18, 20 and 24 bit types are not implemented correctly for software processes and for hardware processes in simulation. Streams and signals are currently limited to a maximum of 32 bits.

(I) For hardware synthesis of streams and signals, data types upto 32 bits are supported for hardware-to-software connections. All the Streams-C data types are supported in the hardware library and can be used for hardware-to-hardware streams and signal connections.

(I) Due to the limited number of different types of integers in C, the Streams-C software simulation only supports bit sizes of 8, 16, and 32 accurately. Other sizes are treated as the next largest supported size. Any size over 32 is not supported in software simulation.

Example:

sc_int128 data; not supported
sc_int8 data_o; accurately supported
sc_int12 data_i; treated as a 16 bit variable
sc_int4 data_s; treated as a 8 bit bariable

(I) For hardware synthesis explicit (and implicit) casting for both signed and unsigned itegers is supported. In multiplication, to prevent an exponential growth in the size of variables, the result will be cast to the size of the largest of the operands. In order to get the full result of a multiply, the operands need to be cast before the multiply is carried out. When casting down, the least significant bits are passed to the result and the most significant bits are lost. When casting up, the variable is extended (signed or unsigned) depending on the nature of the cast.

Example:

sc_uint8 foo;
sc_uint4 bar;
sc_uint8 result8;
sc_uint12 result12;
bar = foo; lowest 4 bits of foo passed to bar
result8 = foo * bar; 8 bit value stored in result
result12 = (sc_uint12)(foo * bar); *illegal: 8 bit result from multiply is cast to 12 bits*
result12 = (sc_uint12)foo * bar; correct: 12 bit result from multiply stored in result12
result12 = (sc_uint12)foo*(sc_uint12)bar; correct: 12 bit result from multiply stored in result12

Return Type	Name	Arg 1	Arg 2	Arg 3	Arg 4
void	**sc_initiate**	(qualified) name	parameters		
void	**sc_terminate**	(qualified) name			
void	**sc_stream_open**	name			
void	**sc_stream_close**	name			
sc_error_type	**sc_error**	name			
Boolean	**sc_stream_eos**	name			
Int type	**sc_stream_read**	name			
void	**sc_stream_write**	name	value		
<**sc_int_type**>	**sc_wait**	name	name ...		
void	**sc_post**	name	value		
<**sc_int_type**>	**sc_bit_extract**	value	start bit	number of bits	
void	**sc_bit_insert**	destination	dest start bit	number of bits	source
int	**sc_my_id**				
void	**sc_load_mem_from_array**	memory name	array	offset in memory	number of qwords
void	**sc_unload_mem_to_array**	memory name	array	offset in memory	number of qwords
void	**sc_load_bram_from_array**	memory name	array	offset in memory	number of qwords
void	**sc_unload_bram_to_array**	memory name	array	offset in memory	number of qwords
void	**set_memories**				

Figure 6. Streams-C Intrinsic Functions

(I) Arrays must have base type that matches the memory to which the array is allocated. For example, if an array is allocated to an 8-bit on-chip RAM, then the base type of the (possibly multidimensional) array must have a size of 8 bits.

5. Intrinsic Functions

Streams-C includes several predefined intrinsic functions. There are intrinsic functions related to process, stream and signal operations and those related to bit manipulation. The process functions allow you to initiate and destroy processes. The stream functions allow you to open, close, read, and write streams and check for the end of a stream as well as the error indicator. The signal functions allow you to post an event and wait for an event. The bit manipulation functions allow you to insert and extract bits. The predefined intrinsic functions are tabulated in Figure 6 and described below.

5.1 Functions to Manage Processes

There are three functions provided to manage processes. Processes are defined using the directives outlined in Section 3.2. A process object begins execution after the **sc_initiate** function is called. The process name is the first parameter to the function. Optionally, a specific instance of an array process may be specified with an array reference. If the process name is that of an array of processes, and the array reference is omitted, the entire array is initiated. The remaining parameter is the argument to the process(es).
The intrinsic function **sc_terminate** closes down a process, process instance, or range of processes.
The sc_my_id function returns the process's id number. Each process has a unique id number starting at 1.

(I) Id numbers are assigned in the order that processes are declared in the .sc file.

sc_initiate(<process_name> ['[' <integer>']'] [, <process_parameters>])
sc_terminate(<process_name>, <integer>)
sc_my_id ()

The initiate and terminate intrinsic functions may only be called from the C "main" or from other software processes. They may not be called from hardware processes.

5.2 Stream Processing Functions

The first operation to be performed on a stream is the stream open. Since a stream formal parameter to a process must be either input or output, it is not necessary in a stream open function call to specify a direction (unlike file I/O):
sc_stream_open(<stream_name>)
The stream open resets the stream internal state. There are no error conditions associated with a stream open.
When the stream is no longer needed, it may be closed:
sc_stream_close(<stream_name>)
The stream close writes an "end-of-stream" token to the output stream. There are no error conditions associated with a stream close.
Some stream operations might result in an error. To check for an error on a stream, the **sc_error** function may be called:
sc_error_type sc_error(<stream_name>)

(I) Currently only one error is defined: overflow, which is 1. No error is a zero.

On an input stream, two additional operations may be performed: end-of-stream test and stream read. The end-of-stream test checks to see whether a "close" operation was performed on the stream by the stream writer. It does this by checking the current element at the head of the stream. If this elemen is the distinguished "end-of-stream" token, a "true" value is returned; otherwise a "false" value is returned.

The stream read tries to read the next stream element, and blocks if the stream is empty. A read operation on a closed stream returns zero and sets the end-of-stream flag.
Boolean **sc_stream_eos**(<stream_name>)
This operation is only allowed on input streams.
<sc_[u]int_type> **sc_stream_read**(<stream_name>)

This function returns a stream element of the Streams-C integer data type associated with the stream (see Section 4 for a list of Streams-C integer data types). If the stream is closed, zero is returned, and a subsequent call to **sc_stream_eos** returns True. If an error was encountered, a subsequent call to **sc_error**(<stream_name>) returns the error and clears the error indicator. The stream read function is only allowed on input streams.

Output streams may be written:
sc_stream_write(<stream_name>, <value>)
The stream must be a writable stream, and the value must be coercible to the stream data type.

5.3 Signal Functions

Signals are used for directed occasional communication between processes, typically for synchronization. Two operations are permitted on signals: post and wait. A parameter may be passed with the signal.
<**sc_int_type**> **sc_wait**(<signal_name_list>)
The process receives a signal posted by the signal writer. If a signal has not yet been posted, the process waits. If the wait statement specifies more than one signal, the statement returns whenever one of the signals in the list has been posted.
sc_post(<signal_name>, <value>)
The signal is posted along with the parameter, over-writing any previously posted signals. The process continues immediately. If acknowledgement is desired, the receiving process should post a different signal back to the sender, and the sender should wait for the acknowledging signal.

(I) Multiple signal posts can result in over-writen data, reference Streams-C examples apps/sig2 showing recommended use for signal posts.

(I) If a numeric value is posted with a **sc_post** call, the value must be cast to the same type as the output signal.

5.4 Bit Manipulation Functions

There are two functions that are useful for bit manipulation: extracting bits and inserting bits. For both these functions, the start bit is the low order bit and the number of bits counts from the low order bit.
<**sc_int_type**> **sc_bit_extract**(<value>, <start_bit>, <number_of_bits>)
This function returns a contiguous range of bits extracted from <value>, starting at the specified start bit for the specified number of bits.
sc_bit_insert(<destination>,<d_start_bit>, <number_of_bits>, <source>)

The source is inserted into the destination starting at bit <d_start_bit> for <number_of_bits>. Truncation or sign extension are performed depending on the underlying data type.

5.5 Miscellaneous Constructs

5.6. Memory Functions

Two functions are provided to write to and read from memories on the FPGA. The memories are named mem_0, mem_1, mem_2, mem_3 and mem_4 for the Firebird board.
sc_load_mem_from_array(<memory_name>,<array>, <memory_offset>, <number_qwords>)
sc_unload_mem_to_array(<memory_name>,<array>, <memory_offset>, <number_qwords>)
Two functions are provided to write to and read from block rams on the FPGA. The memories are named DP_1_<number>, DP_2_<number> for the Firebird board.
sc_load_bram_from_array(<memory_name>,<array>, <memory_offset>, <number_qwords>)
sc_unload_bram_to_array(<memory_name>,<array>, <memory_offset>, <number_qwords>)
(I) The current release of the compiler supports only on type named bram_0, reference Streams-C example apps/bram1.
A function to initialize external memories for synthesis
set_memories()

5.7. Macros

The IF_SIM macro is provided for convenience. The body of the macro is executed in simulation mode and omitted in synthesis mode. A corresponding IN_NSIM macro is provided to include code for synthesis but not simulation. Similarly, #ifdef SC_SIMULATION can be used for code intended for simulation mode (and #ifndef SC_SIMULATION for synthesis mode).

Other convenient directive include: /// HARDWARE_INCLUDE /// HARDWARE_INCLUDE_END These directives are used before and after include files and macros that must be included in the generated .cf file for synthesis. Most include files such as stdio.h or math.h are not included with the input to the synthesis compiler. Putting include files into the hardware include block ensures that those files will get included in the synthesis compile.

Similarly, for code run on the host, the directives: /// SOFTWARE_INCLUDE /// SOFTWARE_INCLUDE_END are provided. These directives are used before and after include files and macros that you want included in the generated _sim.cpp and _syn.cpp files.

6. Hardware-Supported Subset of C

- Dynamic memory allocation is not supported
- Pointers are not permitted. Indirect reference must be accomplished through array reference.
- C variable must be declared at the beginning of the process. i.e. cannot declare i in a for loop, for(int i=0; i<n; i++)
- Arrays of processes are implemented, but local block ram memory for each process in the array is not.

7. Style Issues

7.1. Optimization Hints

- Use functions sc_bit_insert() and sc_bit_extract() instead of the shift left or shift right (”<<” or ”>>”) operators.

7.2. Simulation vs. Synthesis

Occasionally it is useful to write the code in one way for simulation and slightly different for synthesis. The IF_SIM macro is provided for this purpose.

We have found that C compiler bugs sometimes cause large locally allocated arrays to get corrupted. Thus to circumvent the bug in simulation, the array is globally allocated during simulation and locally allocated for synthesis.

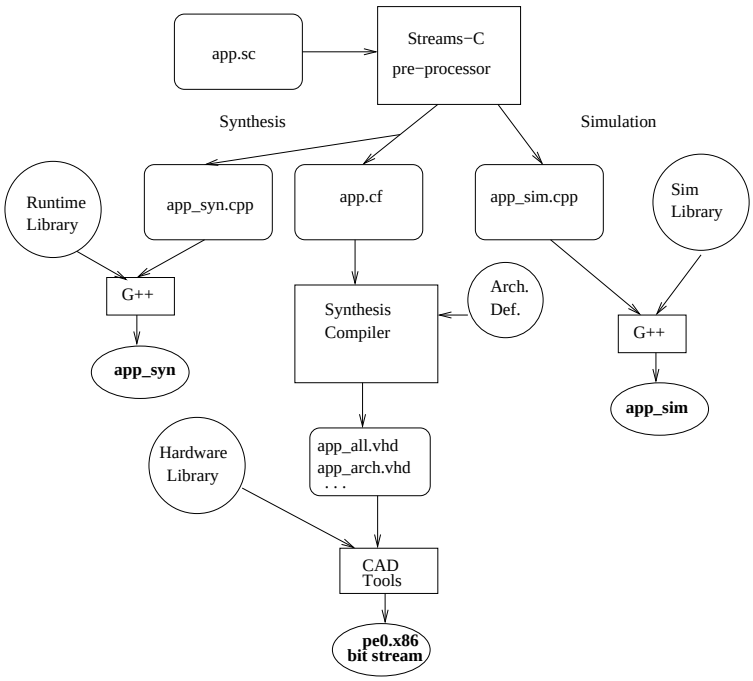

Figure 7. Organization of the Compiler

8. Using sc2

8.1. Compiler Organization

A Streams-C program may be simulated at the functional level (see figure 7). Our functional simulator uses the Linux pthreads package to support concurrent processes and stream communication. At this level, the programmer can use conventional software debuggers and "print" statements to understand the parallel program's concurrency behavior. The programmer can detect many potential deadlock and livelock conditions, and get a good approximation for buffer sizes required for correct program execution.

Our simulation tools use the `///` annotations to generate a C++ program that links the process function body with the simulation library. The generated C++ source program is then linked with our "ptstreams" library to produce a Linux executable that can run on the Linux workstation. The process of compiling for simulation is described in greater detail in Section 8.2.

When the program is compiled for synthesis, there are both software and hardware "object" files generated. The software executable contains all software processes as well as the runtime system to communicate with hardware processes. The hardware bit stream(s) contain all the hardware processes as well as the hardware libraries for stream communication and sequence control. Compiling for synthesis is described in greater detail in Section 8.2.

Note: A pre-processor converts the .sc file into C++ or C for the simulation or synthesis process respectively. The pre-processor parses only the "///" directives, and does not detect syntax errors in the user's code. Thus you should expect to get error messages from the C++ or sc2 synthesis compiler relative to the generated code.

8.2. How to compile for VHDL simulation and synthesis

To compile a <program_name>ṡc file for synthesis, use the standard Streams-C makefile with a target of <program_name>_all.vhd. The hw_lib manual describes VHDL simulation, synthesis and place and route procedures to create a bit stream for the target hardware.

There are numerous phases and intermediate files created on the path from .sc to the .vhd output. Any of the intermediate targets may be specified as a target, if desired.

```
#pragma SC PROCESS_FUN <function_name> [<stream__signal_list>] [<parameter>]

<stream_signal_list> ::= <stream_signal> [ <stream_signal> ... ]
<stream_signal>      ::= <instream> | <outstream> | <insignal> | <outsignal>

<instream>      ::= IN_STREAM <type_name> <stream_name>
<outstream>     ::= OUT_STREAM <type_name> <stream_name>
<insignal>      ::= IN_SIGNAL <type_name> <signal_name>
<outsignal>     ::= OUT_SIGNAL <type_name> <signal_name>

<parameter>      ::= PARAM <type_name> <parameter_name>
```

Figure 8. PROCESS_FUN Pragma

```
#pragma SC PROCESS <process_name> <array_spec> <process_fun_spec> \
<typespec>  <on_spec>

<array_spec>    ::= ARRAY <int>
<process_fun_spec> ::= PROCESS_FUN <name>
<type_spec>     ::= TYPE HP|SP
<on_spec>       ::= ON <resource_name>
```

Figure 9. PROCESS Pragma

1. .sc → .cf converts the "///" annotations into C pragmas for the synthesis compiler
2. .cf → .snt applies the SUIF snoot C parser (as modified for Streams-C)
3. .snt → .spf applies numerous porky passes to eventually generate an input file into the sc2 compiler passes.
4. .spf → .spg performs the first sc2 pass to *normalize* the syntax tree and enforce the sc2 C subset.
5. .spg → .zsd is the major sc2 pass that *schedule*s each hardware process. A porky pass is then applied to dismantle control structure.
6. .zsd → .vhd runs the *vhdl generator* over the scheduled SUIF representation.

Streams-C Pragma Format

When compiling for synthesis, a script converts each "///" block in the .sc file into a C pragma. The normal SUIF processing incorporates the pragmas into the syntax tree. The normalize, schedule and vhdl generator use the pragma information during translation. Each pragma is tagged with the keyword "SC" followed by the name of the option (PROCESS_FUN, PROCESS, or CONNECT) and the additional qualifiers applicable to each option. The pragma must be on a single line. The format of the PROCESS_FUN pragma is shown in Figure 8.

The PROCESS pragma is shown in Figure 9. Each clause is required. If there is a single process instance, the array-spec is written "ARRAY 1." If no on_spec was given in the .sc, a default name FPGA is used. The default for software processes is sc_host.

The CONNECT pragma is shown in Figure 10. If the process name specifies an array of processes, a specific instance or a range may be chosen. The PORT refers to a stream or signal previously defined in a PROCESS_FUN pragma. The to_range uses the "!" symbol to denote all instances in the <from_range>, and the second to_range parameter may be a signed integer denoting an offset from the range.

The memory pragma shown in Figure 11 selects the type of memory used by a given array in a hardware process. See the *sc2 Hardware Library Reference Manual* for more information on the types, width and size of external memories and block RAMs available. The memories are defined in the .def file (in apps/arch) which is specific to each target and these files are provided for the Firebird.

```
#pragma SC CONNECT <process_name> <from_spec> <port_spec> <fifo_spec> [<register_base>]\
  TO <process_name> <to_spec> <port_spec> <fifo_spec> [<register_base>]

<from_spec>     ::= INSTANCE <int> | <from_range>
<from_range>    ::= RANGE <int> <int>

<port_spec>     ::= PORT <name>

<to_spec>       ::= INSTANCE <int> | <to_range>
<to_range>      ::= RANGE ! <int>

<fifo_spec>     ::= FIFO_SIZE <uint>

<register_base> ::= REGISTER_BASE <hex>
```

Figure 10. CONNECT Pragma

```
#pragma SC memory <memory_type> <array_name>
```

Figure 11. Memory Pragma

The pipeline pragma is shown in Figure 12. The pipeline pragma should be inserted after the loop control statement. The current version of the compiler automatically pipelines **for** loops and **while** loops.

(I) The inner-most loops of nested loops can be pipelined. See example /streamsc/apps/fastfold/totalfold.sc

The unroll pragma is shown in Figure 13. It should be inserted after the loop control statement and takes one parameter, which is the unroll factor. For example, ”unroll 4” means that four copies of the loop body should be instantiated. It is useful to unroll small, fixed iteration loops so that all the iterations can occur in parallel.

(I) When using the unroll pragma, large unroll factors combined with large loop body may cause the scheduling phase of the compiler to run out of memory.

Example

For the example program shown in Section 8.3, the following .cf directives are generated:

```
#pragma SC PROCESS_FUN host1_run OUT_STREAM sc_int32 "output_stream"  PARAM int "iterations"

#pragma SC PROCESS_FUN host2_run IN_STREAM sc_int32 "input_stream"

#pragma SC PROCESS_FUN controller_run IN_STREAM sc_int32 "input_stream" OUT_STREAM
sc_int32 "output_stream"

#pragma SC PROCESS_FUN pe0_proc_run IN_STREAM sc_int32 "input_stream" OUT_STREAM
sc_int32 "output_stream"

#pragma SC PROCESS controller ARRAY 1 PROCESS_FUN controller_run TYPE HP ON PE0

#pragma SC PROCESS pe0_proc ARRAY 1 PROCESS_FUN pe0_proc_run TYPE HP ON PE0


#pragma SC pipeline
```

Figure 12. Pipeline Pragma

```
#pragma SC unroll <number of iterations of the loop>
```

Figure 13. Unroll Pragma

```
#pragma SC PROCESS host1 ARRAY 1 PROCESS_FUN host1_run TYPE SP ON sc_host


#pragma SC PROCESS host2 ARRAY 1 PROCESS_FUN host2_run TYPE SP ON sc_host

#pragma SC CONNECT host1 INSTANCE 0 PORT "output_stream" FIFO_SIZE 16 TO controller
INSTANCE 0 PORT "input_stream" FIFO_SIZE 16 REGISTER_BASE 0x0

#pragma SC CONNECT controller INSTANCE 0 PORT "output_stream" FIFO_SIZE 16 TO pe0_proc
INSTANCE 0 PORT "input_stream" FIFO_SIZE 16

#pragma SC CONNECT pe0_proc INSTANCE 0 PORT "output_stream" FIFO_SIZE 16  REGISTER_BASE
0x3 TO host2 INSTANCE 0 PORT "input_stream" FIFO_SIZE 16
```

8.3. Examples

This section contains several Streams-C examples. The first example is a simple pass-through pipeline and illustrates how to define processes and streams, and how to use the stream intrinsics.

8.3.1 Example 1: Passing Data in a Pipeline

This program strm2.sc has two software processes and two hardware processes, as shown in Figure 14. The first software process host1, with run function host1_run opens an output stream and writes a sequence of integers to the stream. The bound on the loop ("iterations") is set by the input argument to the program invocation (eg. the invocation "strm 400" causes a sequence of integers from 0 to 399 to be written to the output stream).

The stream sent by host1 goes to a hardware process controller with run function controller_run. This process simply forwards the stream to the next hardware process, pe0_proc_run.

pe0_proc_run has two phases. First it copies its input stream to memory. When the whole stream has been read into memory, it reads back the data in reverse order and writes to its output stream.

The final process, host2, using run function host2_run, reads the stream from pe0_proc_run and prints out the data received from the stream.

The main program parses the input parameter and then initiates all the processes. A terminate is not necessary, as the processes terminate upon exit from the run function.

The IF_SIM macro is provided for convenience. The body of the macro is executed in simulation mode and omitted in synthesis mode.

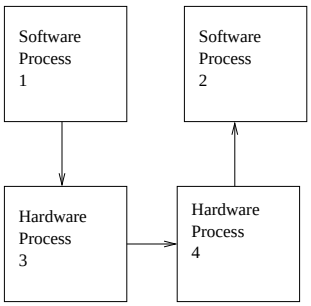

Figure 14. Process interconnection for Example 1

```
/* Example streams-c test program
 * Algorithm:
 *   host1 writes integer values 0 to "iterations" out to the
 *   controller process via a stream.  The controller writes
 *   the values to the pe0_proc process via a stream.  The
 *   pe0_proc stores the integers in an array and then
 *   writes them to the host2 process in reverse order and the
 *   host2 prints them out.
 *
 *   tests stream modules, pipeline modules, memory interface
 */

#include <stdio.h>

#define MAX 5000

/// SOFTWARE_INCLUDE
// After place and route, uncomment these and set to values
// then recompile to create new _syn.cpp.
//#define SC_UCLK <some value>
//#define SC_MCLK <some value>
/// SOFTWARE_INCLUDE_END

void usage(char* ProgramName)
{
  printf("USAGE:  %s <# iterations> (%d is default, max is %d)\n",
 ProgramName, MAX, MAX);
}

int parse_input_pars(int argc, char **argv)
{
  char* ProgramName;

   ProgramName = argv [0];

  printf("StreamC Memory test\n");

  int iterations = MAX;
```

```
  if(argc == 2) {
    int i;

    sscanf(argv[1], "%i", &i);
    if (( i>=0) && (i<=MAX))
      iterations = i;
    else usage(ProgramName);
  } else if (argc != 1)
    usage(ProgramName);
  return iterations;
  }

/// PROCESS_FUN host1_run
/// OUT_STREAM sc_int32 output_stream
/// PARAM int iterations
/// PROCESS_FUN_BODY
  sc_int32 i;

  printf("Process host1 entered\n");

  printf("iterations = %d\n", iterations);

  sc_stream_open(output_stream);

  printf("Process host1 opened stream: output_stream\n");

  for(i=0; i<iterations; i++) {
    printf("Process host1 writing stream: output_stream with: %d\n", (int)i);
    sc_stream_write(output_stream, i);
  }

  printf("Prcess host1 Closing stream output_stream\n");

   sc_stream_close(output_stream);

  printf("Process host1 exiting\n");

/// PROCESS_FUN_END


/// PROCESS_FUN host2_run
/// IN_STREAM sc_int32 input_stream
/// PROCESS_FUN_BODY

  sc_int32 j;

  printf("Process host2 entered\n");

  sc_stream_open(input_stream);

  printf("Process host2 opened stream: input_stream\n");
```

```
  printf("Process host2 reading stream: input_stream\n");
  j = sc_stream_read(input_stream);

  while(!sc_stream_eos(input_stream)) {
    printf("Process host2 read %d from stream: input_stream\n", (int)j);
    j = sc_stream_read(input_stream);
  }

  printf("Process host2 Closing stream input_stream\n");
   sc_stream_close(input_stream);

  printf("Process host2 exiting\n");


/// PROCESS_FUN_END


/// PROCESS_FUN controller_run
/// IN_STREAM sc_int32 input_stream
/// OUT_STREAM  sc_int32 output_stream
/// PROCESS_FUN_BODY

  sc_int32 i;

  IF_SIM(printf("Process controller entered\n"));

  sc_stream_open(input_stream);
  IF_SIM(printf("Process controller opened stream: input_stream\n"));

  sc_stream_open(output_stream);
  IF_SIM(printf("Process controller opened stream: output_stream\n"));

  while(!sc_stream_eos(input_stream)) {
#pragma SC pipeline
    i = sc_stream_read(input_stream);
    IF_SIM(printf("Process controller read %d from stream: input_stream\n", (int)i));
    sc_stream_write(output_stream, i);
  }

  IF_SIM(printf("Process controller Closing stream input_stream\n"));
   sc_stream_close(input_stream);

  IF_SIM(printf("Process controller Closing stream output_stream\n"));
   sc_stream_close(output_stream);

  IF_SIM(printf("Process controller exiting\n"));

/// PROCESS_FUN_END


/// PROCESS_FUN pe0_proc_run
/// IN_STREAM  sc_int32 input_stream
/// OUT_STREAM  sc_int32 output_stream
```

```
/// PROCESS_FUN_BODY

  sc_int32 i, i1;
  sc_int32 A[5000];

  IF_SIM(printf("Process pe0_proc entered\n"));

  sc_stream_open(input_stream);
  IF_SIM(printf("Process pe0_proc opened stream: input_stream\n"));

  sc_stream_open(output_stream);
  IF_SIM(printf("Process pe0_proc opened stream: output_stream\n"));

  i = 0;
  while(! sc_stream_eos(input_stream)) {
#pragma SC pipeline
    i1 = sc_stream_read(input_stream);
    IF_SIM(printf("Process pe0_proc read %d from stream: input_stream\n", (int)i1));
    A[i] = i1;
    i++;  /* max of 5000 is enforced at host */
  }

  for (i=i-1; i>=0; i--) {
#pragma SC pipeline
    sc_stream_write(output_stream, A[i]);
    IF_SIM(printf("Process pe0_proc wrote %d from stream: input_stream\n", (int)A[i]));
  }

  IF_SIM(printf("Process pe0_proc Closing stream input_stream\n"));
   sc_stream_close(input_stream);

  IF_SIM(printf("Process pe0_proc Closing stream output_stream\n"));
   sc_stream_close(output_stream);

  IF_SIM(printf("Process pe0_proc exiting\n"));

/// PROCESS_FUN_END


//
// process definitions
//

/// PROCESS controller PROCESS_FUN controller_run TYPE HP ON PE0

/// PROCESS pe0_proc PROCESS_FUN pe0_proc_run TYPE HP ON PE0

/// PROCESS host1 PROCESS_FUN host1_run

/// PROCESS host2 PROCESS_FUN host2_run

//
// connections
```

```
//

/// CONNECT host1.output_stream controller.input_stream
/// CONNECT controller.output_stream pe0_proc.input_stream
/// CONNECT pe0_proc.output_stream host2.input_stream

void main(int argc, char *argv[]) {
  int iterations = parse_input_pars(argc, argv);
  sc_initiate(host2);
  sc_initiate(controller);
  sc_initiate(pe0_proc);
  sc_initiate(host1, iterations);
}
```

9. Compiler Implementation Notes

References

[1] Xilinx corporation http://www.xilinx.com/xilinxonline/jbits.htm. 1999.

[2] M. Gokhale and J. Stone. Napa c: Compiling for a hybrid risc/fpga architecture. *Proceedings of the IEEE Symposium on FPGAs as Custom Computing Machines*, 1998.

[3] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *IEEE International Symposium on FPGAs for Custom Computing Machines*, 2000.

[4] M. Hall et al. Defacto: A design environment for adaptive computing technology. *Proceedings of the 6th Reconfigurable Architectures Workshop (RAW'99)*, 1999.

[5] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.

[6] B. Hutchins et al. Jhdl-an hdl for reconfigurable systems. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 1998.

[7] E. Pauer, P. Fiore, J. Smith, and C. Myers. Algorithm analysis and mapping environment for adaptive computing systems. *FPGA2000*, 2000.

[8] S. Periyayacheri et al. Library functions in reconfigurable hardware for matrix and signal processing operations in matlab. *Proc. 11th IASTED Parallel and Distributed Computing and Systems Conference (PDCS'99)*, November 1999.