International Journal of Soft Computing 1(4): 261-270, 2006
© Medwell Online, 2006

Adaptive Streams-C Design Methodology for SoRC Based on Modern FPGA

[1,2]Mtibaa, A., [2]A. Sboui and [3]E. Bourennane
[1]Ecole Nationale d'Ingénieurs de Monastir, Avenue Ibn ElJazzar 5019 Monastir Tunisie
[2]Laboratoire EmE de la FS Monastir, Avenue de l'environnement- 5019 Monastir Tunisie
[3]Laboratoire LE2i, CNRS UMR 5158, BP 47870 Université de Bourgogne, Dijon France

Abstract: In recent years, the use of FPGA "Field Programmed Gate Arrays" as the main suitable circuit for SoC "System on Chip" prototyping has launched an ever-increasing FPGA capacity mass production. FPGA based systems offer the programmability of software, allowing various applications to be mapped to them. Those circuits, specially the modern FPGAs, witch actually containing one or more embedded processor cores (both hard and soft cores) are quickly becoming a mainstream architecture for high performance application. Hoverer the lack of co-design tools witch fully support those new architectures has introduced the so-called Design Productivity Crisis. To remedy this gap, a great effort has launched in researching of an CAD "Computer Aid design" tool that is oriented towards hardware rather than software development with a reasonable abstraction level, witch established standards industry-wide and/or the efficient interface synthesis tools making the integration of IPs "Intellectual Property" from different provider and/or technologies in the same SoC will be more easy. In this study we aim to extend the Streams-C design and synthesis environment in order to support those new challenges.

Key words: FPGA, SoC, SoRC, IP, streams-C, powerPC, coreconnect

INTRODUCTION

Great influence has been added to area and speed SoC designs by On Chip Communication Architectures OCCA. This influence is expected to be even more pronounced on System on Reconfigurable Chip SoRC. To date, little researches has been analysed the OCCA suitability of SoC for SoRC[1]. We describe in this document our contribution to implement a flexible communication module extracted from Streams-C component interface witch based on FIFO blocs and evaluate the interconnection structures performances in comparison to conventional buses.

We describe in the first part, the Streams-C programming model, the second part interest in an adaptive Streams-C design methodology in order to support the latest challenges witch occurred in FPGA's embedded resources and the programmable logic evolution[2,3], the third part give a solution to adapt Streams-C hardware component witch targeted to Virtex and Virtex E to Virtex II Pro FPGA.

In this study, we aim to extend the Streams-C tool methodology in order to support the the latest FPGA SoRC architectures.

STREAMS-C PROGRAMMING MODEL AND RELATIONSHIP TO OTHER ON-CHIP INTERFACE

The Streams-C langage have an intermediate level (between the high level and the low level design tools). This language consists in a set of library functions callable from a C language program[2].

The Streams-C compiler synthesizes the hardware part of the application architecture for multiples Virtex-E FPGA based board and the software part like a multithreaded software program for host computer control processor. Our platform architecture is based on "Annapolis Microsystem FireBird ™ PCI board" which has the following block diagram (Fig. 1). The specification of the behavior architecture is done with high level constructions or components which permit to model the application like a set of competitor processes. Those communicate between them using tubes of data called Stream. The Stream components are based on producer-consumer non blocking protocol. The processing model used in Streams-C is inspired from CSP (Communicating Sequential Processes) formalism like a parallel programming model[2,4].

Its implementation is a combination of annotations and library functions callable from C. This is for pragmatic

Corresponding Author: Mtibaa, A., Ecole Nationale d'Ingénieurs de Monastir, Avenue Ibn ElJazzar 5019 Monastir Tunisie

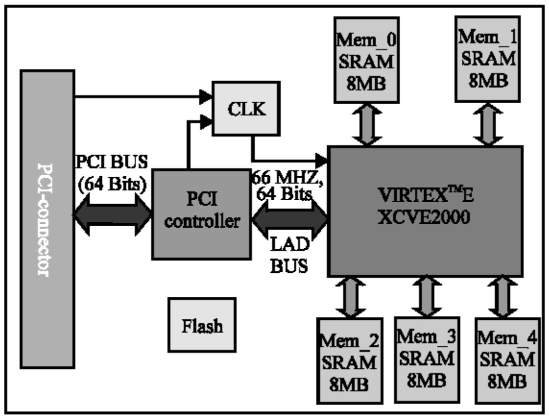

Fig. 1: Firebird PCI board bock diagram

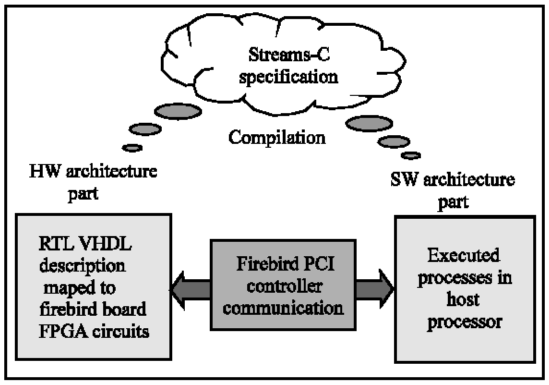
Streams-C specification
Compilation
HW architecture part
SW architecture part
RTL VHDL description maped to firebird board FPGA circuits
Firebird PCI controller communication
Executed processes in host processor

Fig. 2: Streams-C compilation phase

reasons as streams-C builds on several years of compiler development within the SUIF C-processing framework[5].

This model is mainly based on three distinguished objects, processes, streams and signals. A process is an independently executing object with a process body that is given by C subroutine. A process can run on the host processor or on an FPGA chip. For an FPGA process, the process body accesses only local data and is written in a subset of C supported by the Streams-C compiler. In Addition, intrinsic functions to perform stream or signal operations may be referenced. All declared processes are initiated when the program begins and run until their subroutine bodies complete.

This study is well adapted to the parallelization of image processing and signal algorithms[2,6].

Those can be expressed like a succession of independent processing modules. Each module use like input data the results produced by one or several previous modules. This communication model allows that every of processing module can be described like a task which communicate in one side with process and the one with the following stage. In Streams-C tool, the parallelism (in task level) is explicitly done by the designer and it appears in the competition between the processes forming the program. During the compilation phase of C language based specification, every task is transformed to an autonomous hardware entity synthesizable architecture (associated to hardware processes) in RTL VHDL level, or being compiled to be executed like a simple set of processes on the host processor which communicate with our reconfigurable resource FireBird board (Fig. 2).

The management of communication between the different tasks forming the application is taked place by means of flux on which the elementary input/output operations are possible. This flux can be seen like a tampon file or buffer with sequential access, in which the producer process writes its results and the consumer reads its data. Streams-C materializes this flux by a FIFO type structure.

The streams-C hardware libraries: It was written in synthesizable RTL-level VHDL with heavy use of generics. Depending on the context of use, the compiler instantiates specific modules from these libraries. Its stream library consists of approximately 30 modules. Each stream module uses a standard interface to communicate with the process on one side and the I/O port on the other side. The modules make use of the various "Firebird" channel mapping such as nears test neighbor, Crossbar, FIFOs, or intra-FPGA communication. Two protocols, Valid Tag and buffered, have been implemented. The valid Tag Protocol delivers a data item accompanied by a one-bit tag every clock cycle. The tag indicates whether the data item is valid. The Buffered protocol fills a buffer on stream write and then empties the buffer on stream read. If the buffer nears full, the stream read module sends a signal to the stream write module to stall, which the stream write module propagates to the pipeline controller and instruction sequencer of the producing process.

The streams-C software libraries: Which use the POSIX threads package Pthreads to simulate (at functional level) software processes and streams communications in software application architecture part.

PROPOSED BASED INTERFACE METHODOLOGY

In the early stages of SoRC design, cores were designed with many different interfaces and communication protocols[7,8].

Integrating such cores in a SoRC often required suboptimal glue logic to be inserted. In order to avoid this problem, standards for on-chip bus structures were developed. Currently there are a few publicly available bus architectures from leading manufactures, such as the

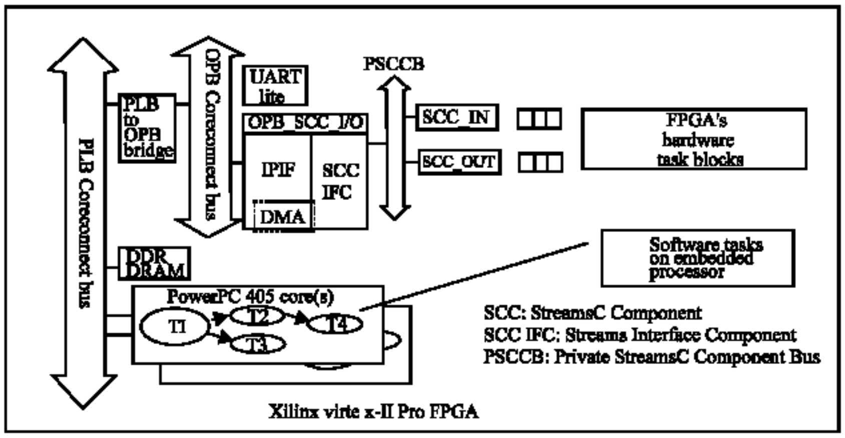

Fig. 3: SoC based on Hw/Sw streams-c component

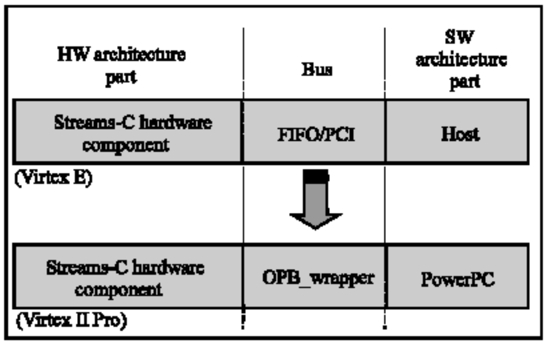

Fig. 4: Streams C adaptive architecture

The cores are predesigned and preverified to work with the CoreConnect bus architecture and protocols, thus allowing for reuse from chip to chip.

Our methodology[12] consist in the substitution of the software part of our application which is a set of processes executed in the host computer to a set of hardware processes executed in PowerPC processor core into the FPGA circuit for a final issue to create a full Hw/Sw embedded system as a SoRC prototype (Fig. 3).

The hardware component of Streams-C compiler is targeted for Virtex E FPGA, so we must take the technologic and architectural constraints before move our design to Virtex II Pro FPGA family (Fig. 4).

Our work is targeted to XUP Virtex II Pro Development system board which contains one Virtex II Pro FPGA with two embedded PowerPC processor core[13] (Fig. 5).

We propose in this adaptive methodology to implement a wrapper component called OPB_Wrap as a CoreConnect OPB side embedded architecture bus. This OPB_Wrap is generated like a Streams-C hardware component and based on an existent Xilinx soft IP called

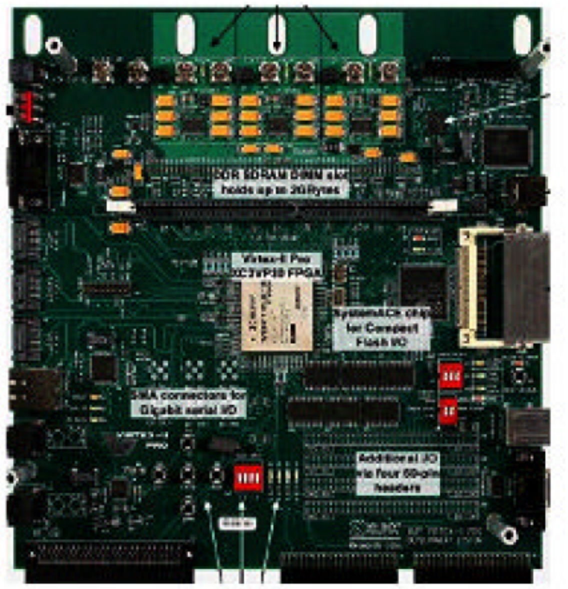

Fig. 5: XUP Virtex II Pro development system board[3]

We propose in this adaptive methodology to implement a wrapper component called OPB_Wrap as a CoreConnect OPB side embedded architecture bus. This OPB_Wrap is generated like a Streams-C hardware component and based on an existent Xilinx soft IP called OPB_IPIF[13] with its pseudo full configuration (Fig. 6). This soft IP is supported by both the Virtex-E and the Virtex-II Pro FPGA circuit families.

OPB WRAPPER COMPONENTS SPECIFICATIONS

The block diagram of Streams-C OPB Wrapper Sc_OPB_Wrap" is showed in the Fig. 7.
The principal features of the "OPB_wrap" are resumed as follow:

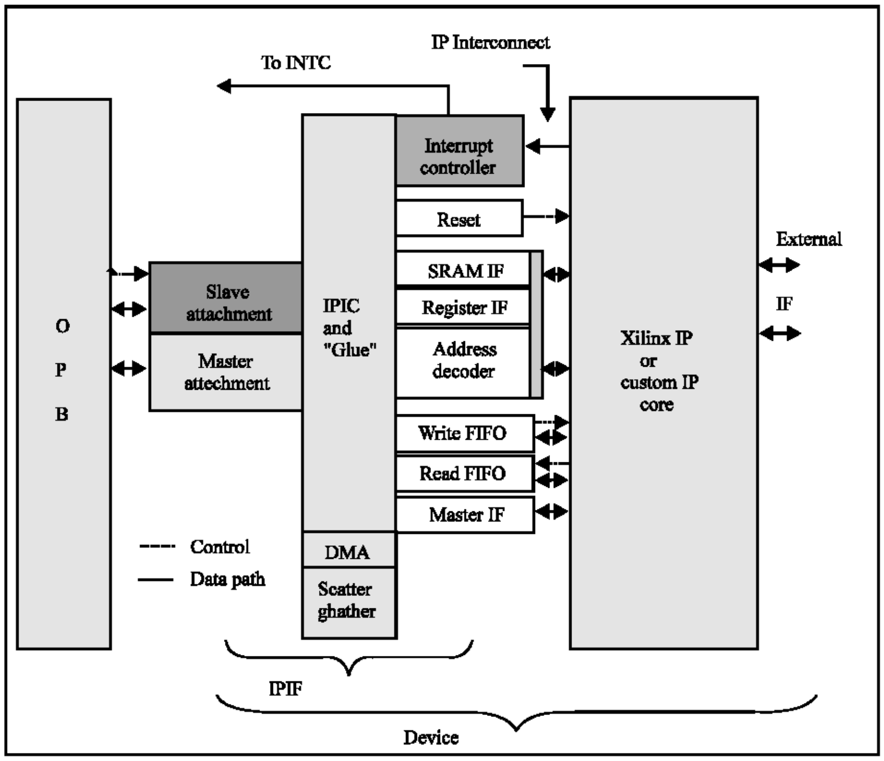

Fig. 6: OPB IPIF block diagram

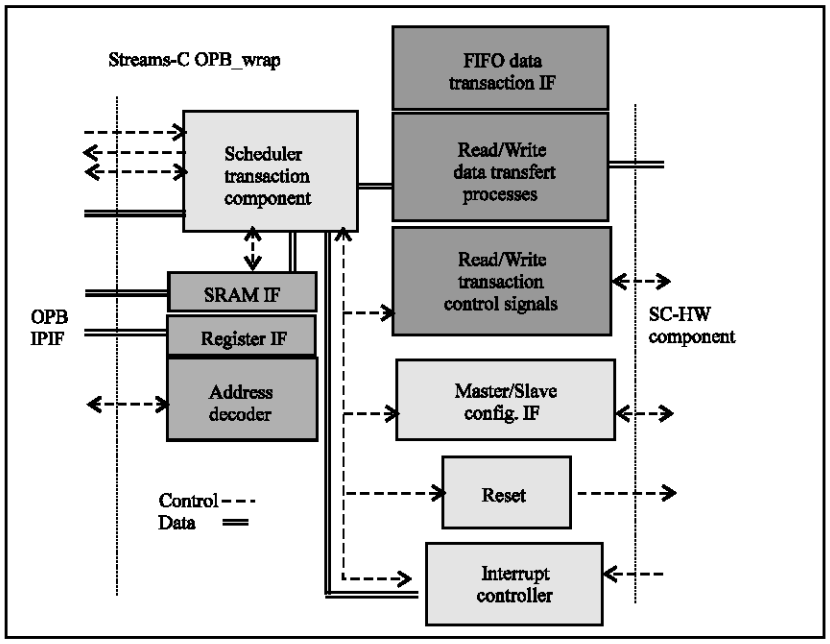

Fig. 7: OPB _Wrap Component block diagram

- The FIFO data transaction is a module that contains Read/ Write hardware processes specified in Streams-C with its controls signals interfaces between the Streams-C hardware component (Sc_Hw_Comp) and the Xilinx Scheduler Transaction Component (Sch_Tr_Comp);
- The interrupt controller is a module which receives interruptions from Sc_Hw_Comp and send interrupt packet to Sch_Tr_Comp, this packet contain the class Scheduler_c : public sc_hw_process {
  public:
  (const char* process_name,

```
unsigned int new_id,
sc_system* sys,
processor* pe)
: sc_hw_process(process_name, new_id, sys, pe)
{
}
void* run(){};
};
```

Fig. 8: Streams-C void process class

```
/// PROCESS_FUN ReadWrite_FIFO_run
/// IN

                                        1: Process declaration

/// PROCESS_FUN Read_Sc_Hw_Comp_FIFO

                                        2: Streams (s) declaration

/// IN_STREAM sc_uint8 [256]  IP_To_Sch_FifoInStream
/// IN_STREAM sc_uint8[256] OPBIPIF_To_Sch_FifoInStream

/// OUT_STREAM sc_uint8[256] Sch_To_IP_FifoInStream
/// OUT_STREAM sc_uint8[256] Sch_To_OPBIPIF_FifoInStream

                                        3: Signal(s) declaration

/// IN_SIGNAL  sc_uint32 OPBIPIF_To_Sch_Enable_Sig
/// OUTvSIGNAL   sc_uint32 Sch_To_IP_Enable_Sig

/// OUT_SIGNAL   sc_uint32 Sch_To_OPBIPIF_Ready_Sig
/// OUT_SIGNAL   sc_uint32 Sch_To_IP_Ready_Sig

/// OUT_SIGNAL   sc_uint32 Sch_To_OPBIPIF_Full_Sig
/// OUT_SIGNAL   sc_uint32 Sch_To_OPIPIF_Empty_Sig

/// OUTvSIGNAL   sc_uint32 Sch_To_OPBIPIF_WriteEnable_Sig
/// OUT_SIGNAL   sc_uint32 Sch_To_OPIPIF_ReadEnable_Sig


/// PROCESS_FUN_BODY

             C subroutine code

/// PROCESS_FUN_END
```

Fig. 9: Streams-C Hw process declaration

source interrupt, the current operation defined in Sc_Hw_Comp status registers and the destination of the interruption. This interrupt packet (Int_Pack) is processed by Sch_Tr_Comp which according to the nature of Sc_Hw_Comp current operation and its interruption priority defined by the user in Sch_Tr_Comp specification, this last send interrupt events to the OPB IPIF[13].

- The SRAM interface (respectively Register IF) is targeted for IP which support this type of transfer, in our case this interface consist in FPGA local RAM bocks which is shared between the OPB IPIF and the Sch_Tr_Comp. The register interface consist in few register (point address) decodes, read/write request and acknowledge signals. The SRAM interface consist in a address range with its address block decode, address bits and transaction control signal.
- The scheduler transaction component is the main module in Sc_OPB_Wrap, this block schedule the different data operations (FIFO communication between OPB IPIF and the Sc_Hw_Comp, SRAM communication between Sc_OPB_Wrap and OPB IPIF), configuration signals (via the Master/Slave Config IF), interrupt events and reset signal. The scheduler module is parameterized by the user like the different operation priorities, the master slave Sc_Hw_Comp configuration, The size or /and depth of different FIFO, SRAM and Registers.

Sc_OPB_Wrap specification: As we mentioned previously that an process may be either a host process or an FPGA process. Host process can do file I/O and use the full C language. FPGA processes must adhere to the supported C language subset[14,15].

The compiler can synthesize logic for operations using unsigned integers of arbitrary bit length and arrays of integers. The programs must use structured programming constructs. In lining of function calls is currently being implemented.

Preprocessing for hardware side of synthesis: On the hardware side, the app.sc file is processed and translated into one file called app.cf before being processed by the synthesis compiler. It includes:

- The architecture definition of the hardware (in pragmas);
- Include files, such as macros for the hardware compiler and other things;
- Extern declarations of functions that are running in software and their pragmas;
- Definitions of functions running in hardware and their pragmas;
- Pragmas for each process declared in ///PROCESS;
- Pragmas for each connection declared in ///CONNECT.

Function headers have void return values and take processes and parameters as arguments.

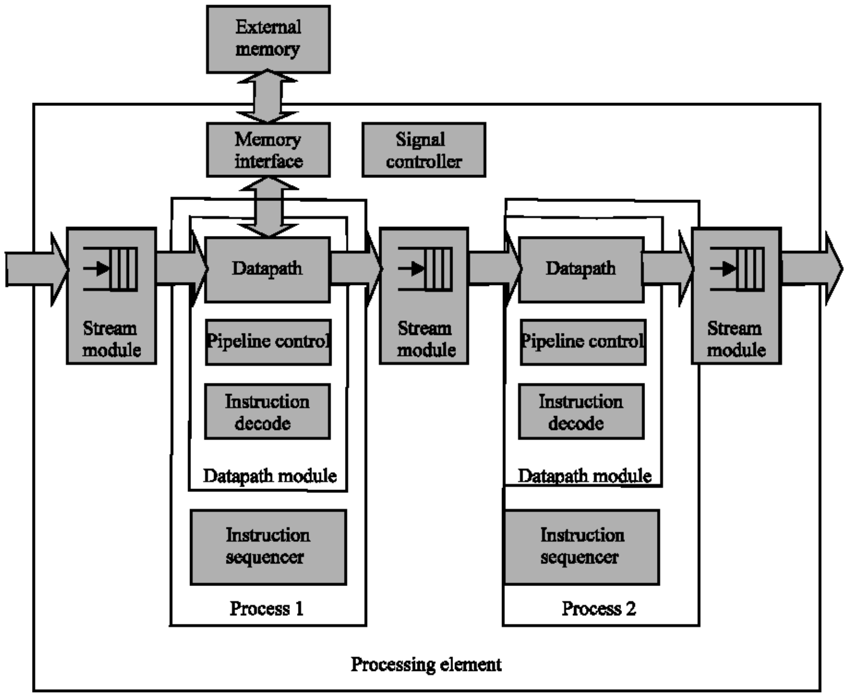

Fig. 10: Processing element structure

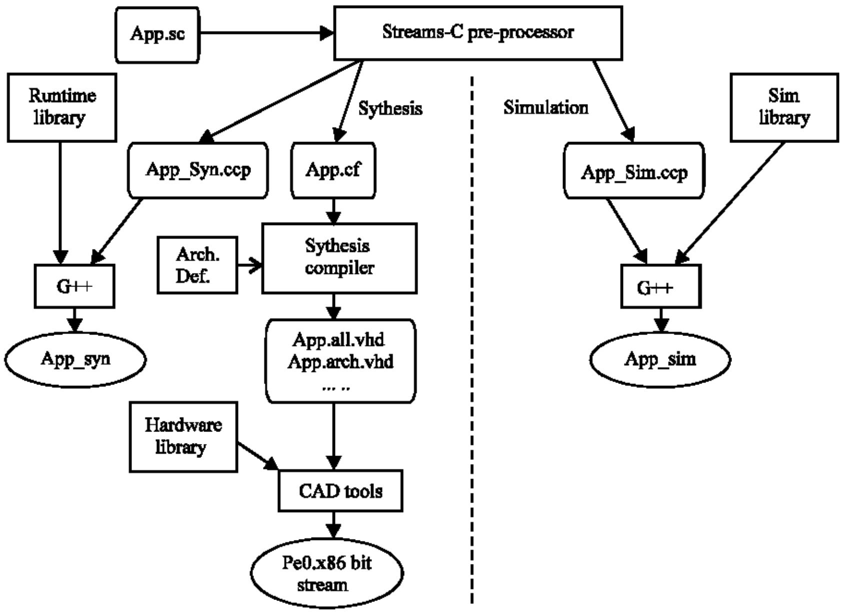

Fig. 11: Streams-C compiler organization

Preprocessing for simulation: The app.sc file is processed and translated into app sim.cpp. This is done in a similar manner to preprocessing for the software side of synthesis; however every process is a derived class of sc_sw_process. All connections are represented and all run functions are included. There are, however, no register assignments or startup/shutdown of processors. In our example the Sch_Tr_Comp is modelled by a set of void class declaration (Fig. 8 and Fig. 9) containing hardware processes. This component contains mainly three processes; one for FIFO data transaction; one for SRAM interfacing and the last for memories

```
FSM: process (State, RdEn, WrEn,
              Mem64_Akk,
              Mem64_High_Data_Valid,
              Mem64_Low_Data_Valid)
Begin   -- process FSM
   Mem64_Req <= '0';
  Mem64_Write <= '0';
  Mem64_Low_Enable <= '1';
  Mem64_High_Enable <= '1';
  Stall <= "0";
  case State is
    when ST0 =>
        NextState <= ST0;
        if (RdEn = "1")
          then
            Mem64_Req <= '1'
            NextState <= ST1;
        end if;
    when ST1 =>
       if((Mem64_Low_Data_Valid = '1')
 and (Mem64_High_Data_Valid = '1'))
        then
        Stall <= "1";
           NextState <= ST0;
        else
          Stall <= "1";
          NextState <= ST1;
       end if;
  end case;
END process FSM;
```

Fig. 12: Generated FSM for control flow

initializations (Register interface can be modelled as a FIFO transaction).

Every hardware process is an independent executing entity (PE) (Fig. 10) which contain Stream(s) or/and signal(s) declarations and C subroutine body. Among these main processes for "Sch_Tr_Comp" in Streams-C specification is shown in Fig. 9. In this declaration (FIFO Hw process in "read" mode), this one of these three processes which have been declared with their own stream and signals names.

The process body which is a sort of C subroutine and in this declaration it uses the streams and signals of the two other processes and in the same time wait an Int_Pack data from the interrupt controller; in our testbensch example, the IP Sc_Hw_Comp is configured as slave, with 32x256 FIFO buffer, with high priority FIFO transaction mode.

The Sc_Hw_Comp send three sort of interruption (read, write and initialization); In any case of those interruptions, the Sch_Tr_Comp extract the correspondent operation from Int_Pack and route it for OPBIPIF to the destination component in CoreConnect bus.

SCHEDULER TRANSACTION COMPONENT SYNTHESIS

The Streams-C compiler (Fig. 11) builds on the infrastructure build for the NAPA C compiler[17] which is based on SUIF (Standford University Intermediate Format)[17] compiler infrastructure and the Malleable Architecture Generator "MARGE" data path generator. The Streams-c compiler includes extraction and scheduling of data path blocks from the Abstract Syntax Tree AST[13], pipeling of For and while loops and generation of a control program to sequence the generated hardware blocks.

The Streams-C compiler coverts the process body (C code) into three main parts (Fig. 10); a basic block, a data path and a control flow part. The basic block is a straight line section of code without branches. The pipeline block appears like an inner loop body of a loop with independent iterations (do-all loop) than can be pipelined by the compiler. The control flow part is targeted for loops that cannot be pipelined by the compiler. From this part, Streams-C generates an FSM to sequence the basic and the pipeline blocks of the data path.

The compiler generates a complete VHDL architecture for each chip. The structure of the generated entities is illustrated in Fig. 12. Each chip contains zero or more processes and zero or more stream. A process contains the instruction sequencer and the data path module. Within a data path module is an instruction decoder, zero or more pipeline controllers and the data path circuits. The stream module is specific to the size of the stream payload, the type of resource used by the stream (FIFO, Crossbar, Inter-PE bus) and the synchronisation protocol (valid tag, buffered). The compiler also generates a multi-threaded control program. Before initialling host processes, the control program loads the FPGA configuration bit files and initializes local memories. Then processes also communicate via streams. When a stream connects "Host processes", the same library is used as for simulation.

When a stream connects an FPGA process with any other process, driver calls are issued to read or write the Firebird hardware FIFOs.

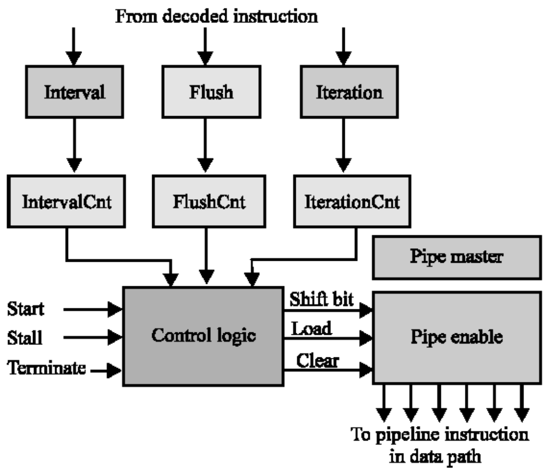

Fig. 13: Pipeline controller structure

The Streams-c compiler builds on the infrastructure built for NAPA C compiler"[16]. The processor module converts the annotations and the SC_macro calls into pragmas that are parsed by Streams-C compiler. This architecture, based on the SUIF compiler infrastructure and the "MARGE" data path generator includes extraction and scheduling of data path blocks from the Abstract Syntax Tree AST, pipelining of For loops and generation of a control program to sequence the generated hardware blocks.

The new version of Streams-C compiler include semantic validation of process and Streams pragmas, pipelining and control of while loops, state machine generation for on board sequencing and an efficient stream communication hardware library, components of which are instantiated by the compiler to effect the desired stream communications.

Pipeline control: A pipeline controller is instantiated whenever there is a pipeline instruction in the data path. There are two types of pipeline controller: definite iteration, where the number of iterations is fixed at loop entry and indefinite iteration, where it is not.

There is an enable register with one bit for each level of the pipeline. The pipeline control module sets the appropriate bits in this shift register to control data path operations at each level. A pipeline controller must first be initialized with: the number of pipeline stages, the pipeline initialization interval; the number of flush cycles and the number of iterations (for definite iteration controller).
Then the pipeline instruction is issued by the sequencing state machine. The initiation interval determines the number of pipeline stages that are active at the same time. The pipeline enable register is a shift register, with a '1' being shifted in every "initiation interval" clock cycles. Since some of the operations occurring withing a pipeline level might stall, the pipeline controller is also stallable. On a stall signal, the enable register is set to '0's. When the stall ends, the original contents of the enable register are restored. Fig. 13 shows the pipeline control structure. The iteration Cnt register is used for definite iterations and the Terminate signal is used for indefinite iterations. The PipeMaster is used to restore the PipeEnable register after a stall.

Pragma processing: The purpose of pragma processing is first to ensure that the process and stream declarations are consistent and the to use the information to map each process onto the correct chip and select the appropriate instance of a stream communication module for each stream used in the program. The compiler has no build-in information about the target architecture. This information is kept externally in header and architecture definition files, allowing the compiler to easily be re-targeted to different board. The header files establish names for each FPGA chip and for each path between chips. The names are the link into an architecture definition file that describes stream library modules. For example, for the Wildforce architecture (our first architecture), if a process is on P1 and writes a stream to a process on the host, the stream uses the hardware FIFO1out connection on the Firebird board and a stream module is selected for instantiation on P1 that observes the FIFO protocol and connects to those pins on P1. Similarly, in the host process, the stream read calls the Firebird driver to read FIFO1.

Process and signal attributes: Those attributes of the processes and streams must be defined in annotations. There are seven stream declarations in our scheduler transaction module. Every declaration gives the stream name and its own type.

For synthesis, a physical path is specified. There follow the bit width of the stream element, the number of buffers, the number of readers and the first bit used. The example also shows process specifications for read_fifo, scheduler and purge; these give the physical input and output streams corresponding to the logical streams used by the process. In addition, for synthesis, the process is mapped to a physical chip and the name of the process's subroutine body is given.

Functional simulation: A Streams-C program may be simulated at the functional level (Fig. 11). Its functional simulator uses the Linux Pthreads package to support concurrent processes and the stream communication. At

Table 1: Synthesis results for OPB_Wrap

	Sch_Tr comp	FIFO IF	SRAM/ IF	Add_ dec	Int_ cnt
Slices	432	385	341	170	150
Luts	612	517	480	229	168
Block RAM4	2	2	1	2	

this level, the programmer can use conventional software debuggers and print statement to understand the parallel program's concurrency behaviour. 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 /// annotation and the SC_macro calls to generate a C++ program that links the process function body with the simulation library. The generated C++ source program is then linked with the Pstreams library to produce a Linux executable that can run on the Linux workstation.

An architecture definition file also describes characteristics of memories accessible to the FPGAs. Read and write latencies for each memory are used by the pipeline scheduler, so that loops that access memory (as well as read/write streams) are pipelined correctly.

Instruction sequencing: Withing a process body, the compiler analyzes the AST and partitions the tree into data path, encompassing basic blocks and pipeline blocks and control flow. A basic block is simply a straight line section of code without branches. A pipeline block is an inner loop body of a loop with independent iterations (do-all loop) that can be pipelined by the Streams-C pipelining algorithms. Control flow is used for loops that cannot be pipelined. We note that "if-statements do" not usually result in control flow, as the compiler converts "If-statements into guards controlling execution of the if-body.

From the AST representation of the control flow, the compiler generates a state machine to sequence the basic and pipeline blocks of the data path. Each block is called an "instruction" in the data path. Conditional expressions are evaluated in the data path and cause state changes in the sequencer.

Synthesis results: Written in Streams-C, the OPB_Wrap synthesis is resumed in the Table 1. Theses results are provided by Synplify synthesis tool and for Virtex II Pro targeted architecture. The Xilinx OPBIPIF (with its full configuration), this last has used like resources the following values: Slices: 567, Luts: 817, Block RAMSs: 4.

DISCUSSION

Current methods of interfacing IP cores to system buses have their own problems. The Xilinx IPIF module only supports Xilinx platforms and OPB/PLB bus standards. An interface methodology has been proposed aiming to ease the interface process and resolve some of these problems. The interface methodology attempts to separate the interface and the function of the IP, in order to eliminate the complex interfacing process for custom and third-party IP cores onto SoC integration platforms. The methodology particularly addresses reconfigurable System-on-Chip. The interface adaptor logic provides an automated interconnection between the IP core and different communication architectures. In the future, once this methodology has been investigated and proven, it can be incorporated into a more automated CAD tool. This CAD tool will allow designers to mix-and-match different IP core functions with the most appropriate system interconnection structure.

There are still significant issues to be resolved concerning the methodology. An Interface Adaptor Logic and a generic IP core will take longer to develop then an ordinary IP core with interface incorporated. If a communication protocol has been updated, the adaptor module will need to be modified.

CONCLUSION

This study has proposed a new methodology that will improve the reusability of IP cores and promote more complete explorations of the most appropriate system interconnection architectures for individual chips. There are several SoRC integration platforms that are available on the market today, each with their unique architecture and each with their own system-bus protocol. Interfacing IP cores to a particular bus protocol has often been a difficult process, requiring significant time to understand the complex bus protocol. Platform vendors recognise this problem and supply system tools to ease the interfacing process. These tools however only support their own platform system bus protocol.

This new study, of a more general set of wrappers has been proposed to address this situation.

A preliminary Wrapper specification has been proposed in this study. The next major task is to implement our first version of the Wrapper generator software. We plan to update this specification and have a small set of those generators and generic IP cores available in the near future.

REFERENCES

1. Reinaldo, A. Bergamaschil and R. Lee. William Designing Systems-on-Chip Using Cores. IBM T. J. Watson Research Center, Yorktown Heights, NY, IBM Microelectronics, Raleigh, NC.

2. Maya, G. Streams-C compilator. Los Alamos National Laboratory.
3. Xilinx, 2005. XUP Virtex II Pro Development system board. Hardware Reference Manual.
4. Maya Gokhale and Jan stone. Stream-Oriented FPGA Computing in the Streams-C High level Language. Los Alamos lab Publications.
5. Hoare, C.A.R., 1978. Communication Sequential processes. Communication of the ACM, pp: 666-677.
6. Ben, A. Abdelali and A. Mtibaa, 2005. Toward hardware implementation of the Compact Color Descriptor for real time video indexing. Advances in Engineering Software J., 36: 475-486, Elsevier Publishers.
7. Tien-Lung N. Lee,and W. Bergmann, 2003. An Interface Methodology for Retargettable FPGA Peripherals. School of ITEE, The University of Queensland, Brisbane Australia.
8. Andy S. Lee and Neil W. Bergmann. On Chip communication architecture for reconfigurable system on chip. School of ITEE, The University of Queensland, Brisbane Australia.
9. IBM, 1999. The CoreConnect TM Bus Architecture.http://www.chips.ibm.com/product/coreconnect/docs/crcon_wp.pdf.
10. ARM. AMBA Specification Overview. http://www.arm.com/Pro+Peripherals/AMBA.
11. IBM. Blue Logic Technology. http://www.chips.ibm.com/bluelogic.
12. Sboui, A., 2006. Study of the communication between embedded processors in the Virtex-II Pro FPGA and the synthesized components by the StreamsC tool. NTSID: Master degree of ENIS High School, Electrical, sfax-Tunisia.
13. Xilinx, 2003. OPB IPIF Architecture. http://www.xilinx.com/ipcenter/catalog/logicore/docs/opb_ipif.pdf.
14. Arnout, G., 2000. SystemC Standard. Proceedings of the ASPDAC.
15. Flake, P. and S. Davidmann, 2000. Superlog, a Unified Design Language for System-on-Chip. Proceedings of the ASP-DAC.
16. Maya Gokhale and Jan stone, 1988. Napa C: Compiling for hybrid Risc/fpga architecture. Proceeding of IEEE symposium on FPGA as computing machines.
17. Standford University, 1988. Standford University Intermediate Format: The National Compiler Infrastructure Project, http://suif.Stanford /edu/suif/.