SIRC: An Extensible Reconfigurable Computing Communication API

Ken Eguro
Embedded and Reconfigurable Computing Group
Microsoft Research
Redmond, WA USA
e-mail: eguro@microsoft.com

***Abstract*—Reconfigurable computing applications often need to divide computation between software running on a conventional desktop processor and hardware mapped to an FPGA. However, the reconfigurable computing development platforms available today either do not provide a sufficient mechanism for the communication and synchronization that is needed or else employ a complex & proprietary API specific to a given toolflow or device, limiting code portability. The Simple Interface for Reconfigurable Computing (SIRC) project provides a straightforward, portable and extensible open-source communication and synchronization API. It consists of both a software-side interface and a hardware-side interface that allows C++ code running on a host PC to communicate and synchronize with a Verilog-based circuit mapped to a FPGA. One key feature of this API is that both the hardware and software user interfaces can remain consistent across all platforms and future releases. This allows applications built for existing systems to migrate to different platforms without significant modification to user code.**

Keywords-FPGA, communication API, open-source

I. INTRODUCTION

Even among the applications that are best-suited for execution on an FPGA, there is often the need to perform some part of the computation in software. This division of labor makes a mechanism for communicating between the software and hardware portions of an application necessary.

A popular way of building a reconfigurable computing platform is to attach a commercially available FPGA development board to a conventional desktop computer via a standard interface (e.g. Xilinx ML-series board communicating with a PC via Ethernet or PCI-E). Although inexpensive, these systems generally lack the high-level hardware and software support that users need to easily communicate and synchronize at the application level.

This lack of support for higher-level functionality means that significant engineering is required before development can begin on the actual target application. Developers must build interface logic and low-level software to use the raw functionality provided by vendors of hardware IP and software drivers. Developers may be very hesitant to invest the necessary time and effort to make these essential pieces. Time-to-market plays a role, as does the fact that any engineering effort would likely be very specific to a given platform. Changing the communication interface, operating system, FPGA board, or lower-level IP that is used could require repeating the entire process in the future.

One potential solution to this problem is to use a sophisticated commercial development toolflow that offers a higher-level synchronization and communication protocol (e.g. Impulse C). Unfortunately, these development tools can be expensive, making them inaccessible to small research and development groups. Furthermore, the user APIs that these development tools present are often complex and proprietary, preventing the developer from porting their application to other toolflows or to unsupported platforms.

There has been work in the research community looking into the HW/SW interface for reconfigurable computing systems. However, the focus of these projects has largely been on creating programming abstractions for developers to easily model the parallelism or communication relationships between their hardware and software modules (e.g. multi-threading [1] or message passing [2]). While appropriate for large-scale computations in which the HW/SW co-design problem is complex, these programming models may be overly complicated and introduce unnecessary overhead in smaller-scale systems in which the computation is not split over multiple hardware modules and the division between what should be done in hardware versus software is clear.

For reconfigurable computing and FPGAs in general to gain traction outside of the most skilled or well-funded application developers, the community needs a simple, open-source communication API. Towards this end, the Simple Interface for Reconfigurable Computing (SIRC) project [3] was designed with a particular emphasis on ease-of-use, future extensibility, and backwards compatibility.

II. USAGE MODEL & DESIGN OVERVIEW

SIRC defines both a high-level software-side API and a high-level hardware-side API to allow C++ code running on a host processor to communicate and synchronize with a Verilog-based circuit mapped to a FPGA. To provide the most approachable system possible, SIRC operates in a master/slave style arrangement in which commands issued by the user's software directs the execution of the user's hardware. The expectation is that the user's software will follow this basic operational model:

1) send one or more pieces of data from the host PC to an input buffer connected to their circuit on the FPGA

2) signal the FPGA to start execution on that data
3) wait until the circuit indicates that computation is complete
4) retrieve the results from an output buffer, also connected to the FPGA-based logic

In turn, the hardware side of the user's application will:

1) wait until signaled by the software API to begin execution
2) fetch input data from an input buffer, compute on that data and place the results into an output buffer
3) signal the software API that execution has completed and return to an idle state

For simplicity sake, this batched execution model operates in a single-threaded manner for a given partnered pair of one instance of the software API and one instance of the hardware API (i.e. only one operation is performed at a time and overlapped reads and writes are not allowed). As will be discussed in more detail later, multiple sets of API interfaces can be used together to support streaming-style execution with overlapped I/O.

As seen in Fig. 1, the software and hardware APIs focus on four fundamental components: two data buffers, a parameter register file, and an execution control module. The sizes of the bulk-transfer input and output buffers are compile-time constants and can be customized by the user. The parameter register file is primarily intended to be used for runtime data that seldom changes (e.g. a session hash key). The execution control module is not only used to signal when the user logic can begin execution and when it is complete, it also enforces read and write synchronization between the user's software and hardware. The user's software begins with exclusive read/write access to the register file and I/O buffers. When the user's hardware is given permission to execute, exclusive read/write access is transferred to the user's circuit until it indicates that execution is complete or until execution is aborted by the software API.

III. SOFTWARE API

The software API consists of a SIRC C++ class and its 12 member functions. Table I shows the SIRC class included in our initial release of the project. Although this particular class implements communication over Gigabit Ethernet, the class name and the type of argument passed to constructor are the only items that will differ between SIRC software classes, regardless of platform or communication mechanism. This allows any SIRC class to be substituted for any other with minimal changes to user C++ code.

The argument passed to the constructor is a unique identifier for the intended hardware-side SIRC counterpart of the newly created class object. In the case of the Ethernet communication class shown in Table I and Fig. 2, the constructor is called with the MAC address of a specific instance of the SIRC hardware API that will be programmed onto the FPGA at execution time. This is the one and only instance of the hardware API with which the software object will communicate.

One problem with the constructor of a C++ class is that it simply succeeds or fails – it either returns a valid or null pointer. While this is sufficient for the purposes of memory allocation, there are any number of reasons for which the constructor (or any other of the member functions) of a SIRC class object may fail. Some of the errors may be fatal and persistent, and some may be recoverable and transient. Thus, it is important to provide the user with the reason for a given

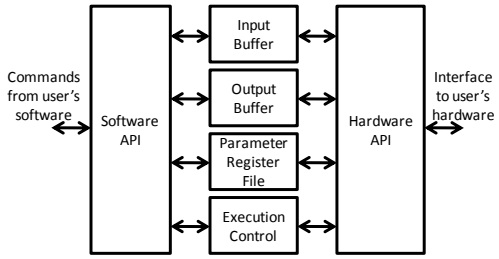

Figure 1. High-level relationship between the software & hardware APIs

TABLE I. EXAMPLE OF SIRC SOFTWARE API CLASS AND MEMBER FUNCTIONS: CONNECTION VIA GIGABIT ETHERNET

ETH_SIRC (uint8_t *FPGA_ID)	Class constructor. Initializes software side of API and verifies that the matching hardware-side interface can be contacted.
~ ETH_SIRC()	Class destructor.
int8_t getLastError()	Return error code returned by last execution of a class function
BOOL sendWrite(uint32_t startAddress, uint32_t length, uint8_t *buffer)	Send block of data from software buffer to FPGA API input buffer.
BOOL sendRead(uint32_t startAddress, uint32_t length, uint8_t *buffer)	Read block of data from FPGA API output buffer back to software buffer.
BOOL sendParamRegisterWrite(uint8_t regNumber, uint32_t value)	Write value to FPGA API parameter register.
BOOL sendParamRegisterRead(uint8_t regNumber, uint32_t *value)	Read value from FPGA API parameter register.
BOOL sendRun()	Signal FPGA API that user circuit can begin execution. Transfer buffer/register file R/W access to user logic.
BOOL waitDone(uint8_t maxWaitTime)	Wait up to *N* seconds for the user circuit to signal execution is complete and buffer/register file R/W control has been relenquished
BOOL sendReset()	Abort execution of user circuit and return buffer/register file R/W control to software.
BOOL sendWriteAndRun(uint32_t startAddress, uint32_t inLength, uint8_t *inData, uint8_t maxWaitTime, uint8_t *outData, uint32_t maxOutLength, uint32_t *outputLength)	Combination of sendWrite, sendRun, waitDone and sendRead functions. For performance reasons, use this function rather than separate explicit function calls.
BOOL sendConfiguration(char *path)	Reconfigure FPGA using provided bitstream file

```
1  int main(){
2    uint8_t MAC[6]  = {0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA};
3    uint8_t input[4] = {1, 2, 3, 4}; uint8_t output[4]; uint32_t outLength;
4    *ETH _SIRC fpga = new ETH_SIRC(&MAC);
5    if(!fpga || fpga->getLastError() != 0) exit(-1);
6    if(!fpga->configure("pathtobitstream"))          //Configure FPGA
7      cerr << "error " << fpga->getLastError(); exit(-1);
8    if(!fpga->sendWrite(0, 4, &input))               //Send input data
9      cerr << "error " << fpga->getLastError(); exit(-1);
10   if(!fpga->sendRun())                             //Begin execution
11     cerr << "error " << fpga->getLastError(); exit(-1);
12   if(!fpga->waitDone(2))                           //Wait up to 2 seconds
13     cerr << "error " << fpga->getLastError(); exit(-1);
14   if(!fpga->sendRead(0, 4, &output))               //Retrieve results
15     cerr << "error " << fpga->getLastError(); exit(-1);
     //Execute again, but use the faster, more concise batch command
16   if(!fpga->sendWriteAndRun(0, 4,& input, 1, &output, 4,
                               &outLength))
17     cerr << "error " << fpga->getLastError(); exit(-1);
18   return 0;
19 }
```

Figure 2. Using the software API to process 4 bytes of data on FPGA

failure. This is where the *getLastError()* function comes into play. As shown in Fig. 2, *getLastError()* should be called after the constructor returns a valid pointer, or after any of the other functions return false. *getLastError()* will return a constant error code that is defined in the SIRC class definition. While these error codes may be specific to a given SIRC class, by convention, fatal errors (e.g. memory allocation failure) are defined as negative constants and errors that may be resolved by retrying a command (e.g. a read command is issued while the hardware logic is still executing) are defined as positive constants.

That said, relatively few of the potential errors returned by a SIRC class should truly be runtime-recoverable by the user. This is because the functions provided by a SIRC class are intended to be robust session-level commands. For example, Ethernet is a fundamentally unreliable communication medium. Thus, our implementation of the *ETH_SIRC* class includes automatic packetization and a TCP-like acknowledgement/retry mechanism. In the course of normal operation, the user does not need to be notified of every dropped packet. However, in the event that multiple consecutive attempts are not acknowledged, the system will retry until a pre-defined threshold has been exceeded. At that point, the API will return an error. This error should be considered fatal, because it is likely that it is caused by some physical problem that will not be resolved by simply calling the function again.

Although most of the functions shown in Table I are relatively straightforward, *sendWriteAndRun()* requires further explanation. As shown in Fig. 2, this function is almost equivalent to explicitly calling *sendWrite()*, *sendRun()*, *waitDone()* and *sendRead()* in series. One difference is purely conceptual – combining these operations into a single function call strongly re-enforces the batch processing nature of the API. However, there is also an implicit functional difference. Calling *sendRead()* separately requires the user to know the size of the results that are expected back from the hardware. This may be the case if the size of the output is purely a function of the size of the input provided or because the user's logic relays the size of the output through the parameter registers. *sendWriteAndRun()*, however, does not require that the user know the size of the output response. Rather, it can receive an arbitrary amount of output data and it returns this information using the *outputLength* variable. For both conceptual reasons and for performance reasons, *sendWriteAndRun()* is the preferred method of interfacing with the user's hardware logic. The other functions are included for the sake of simplicity, completeness, and debugging.

IV. HARDWARE API

A straightforward and standard interface for the hardware portion of a reconfigurable computing application is particularly important – arguably, even more so than the software interface. This is because the design and debugging effort required for the hardware will likely dwarf that required to build the accompanying software, regardless of how simple the hardware interface is or how skilled the developer. Thus, anything that can shorten the process and make a hardware implementation more portable is valuable.

The abstractions presented by the SIRC software API are intended to closely mirror the physical features of the SIRC hardware API. For example, as seen in Fig.3 and Fig. 4, when the user calls *sendRun()* from software, the execution control module asserts the *userRunValue* signal. This notifies the user's circuit that it can begin execution. At the same time, the handshaking protocols on the I/O buffers and the parameter register file become active. The user's logic can then submit read requests to the input buffer via *inputMemoryReadReq* and *inputMemoryReadAdd*. The user's circuit should hold the values for the request steady until *inputMemoryReadAck* is asserted. This signals that the read request was accepted by the API logic. The user's circuit can obtain the input data back from the buffer on the *inputMemoryReadData* bus when *inputMemoryRead-DataValid* is asserted. The circuit can then compute on this

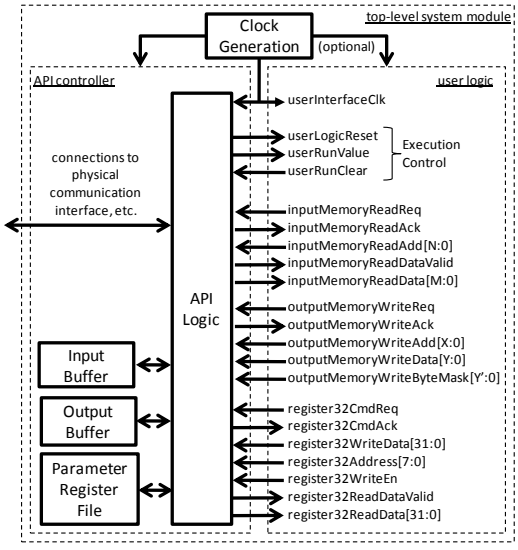

Figure 3. Illustration of Hardware API

```
1 module simpleExample(input userInterfaceClk, input userRunValue…)
2   initial begin ... end
3   always @(posedge userInterfaceClk) begin
4     case(currState)
5       IDLE: begin
6         if(userRunValue)begin
7           currState <=RUN;  inputMemoryReadAdd <=0;
8           outputMemoryWriteAdd <=0;  inputMemoryReadReq <=1;
9       end end
10      RUN: begin
          //Should we request a different input value?
11        if(inputMemoryReadAck && outputMemoryWriteAck)
12          inputMemoryReadAdd <= inputMemoryReadAdd + 1;
          //Are we computing on a new input?
13        if(inputMemoryReadDataValid) outputMemoryWriteReq <=1;
14        else outputMemoryWriteReq <=0;
          //Did we output a new result?
15        if(outputMemoryWriteReq && outputMemoryWriteAck) begin
16          if(outputMemoryWriteAdd < 3) //Did we output the last result?
17            outputMemoryWriteAdd <=outputMemoryWriteAdd+1;
18          else begin
19            currState <= FINISH; userRunClear <= 1;
20            outputMemoryWriteReq <= 0;
21      end end end
22      FINISH: begin
23        currState <= IDLE; userRunClear <= 0;
24        inputMemoryReadReq <=0;
25      end
26    endcase
27    outputMemoryWriteData <=inputMemoryReadData + 1;
28  end
29 endmodule
```

Figure 4. Using the hardware API to increment 4 bytes of received data

data and submit results to the output buffer over a similar handshaking interface.

As mentioned earlier, multiple sets of API interfaces can be used simultaneously to allow overlapped I/O and streaming execution. For example, if the user creates two instances of the software-side API (in two different threads), each one will communicate independently with one of two hardware-side APIs. Each of these hardware APIs will have their own set of I/O buffers, parameter registers, and associated handshaking/execution control signals. The user logic can then alternate between these two independent hardware interfaces in a double-buffering manner.

V. Details of the Ethernet SIRC Implementation

The current implementation of the SIRC software API requires the Virtual Network Services driver freely available from Microsoft. The documentation included with our source code provides detailed instruction regarding how to download, install, and set up this driver. It has been tested using both Visual Studio 2005 and Visual Studio 2008.

The current hardware implementation of the Ethernet SIRC API targets Xilinx Virtex 5 "transceiver" devices – i.e. the LXT, SXT, TXT, FXT chips, but not the baseline LX series. The LX series is not supported because our implementation uses the on-board Ethernet MAC only found on the "T" devices. Our implementation also utilizes two pieces of Xilinx Core Generator IP, the EMAC wrapper core and the block memory generator. These cores are available free of charge from Xilinx, but must be generated by the end user. Lastly, the default pin layout of the project targets the Xilinx ML505, ML507 and Digilent XUP-V5 development boards. With the proper modifications to the pin layout, any board with a compatible FPGA and an external GMII Ethernet PHY can be used. The documentation included with our source code provides detailed instructions regarding how to generate, customize and compile the SIRC hardware API.

In our testing of the full SIRC system, *sendWrite()* and *sendRead()* were able to achieve effective transfer rates of ~950Mbps (98% theoretical max) for transfers greater than 512KB and consistently over 450Mbps for smaller transfers. Furthermore, all of the transmission and execution functions incurred less than 65µs of roundtrip latency beyond the minimum theoretical transfer or FPGA-based logic execution time.

Alongside any community-based support, we intend to consistently maintain the SIRC API. For example, new versions of the system supporting PCI Express, DDR2 memory and multi-user/multi-device platforms are currently under development. These will be released later this year.

VI. Conclusions

In this paper, we have presented SIRC, a software-hardware communication and synchronization API specifically designed for reconfigurable computing platforms. The high-level, platform-agnostic design of the API allows us to completely separate the implementation details a given communication mechanism from all user-generated application code. Combined with its open-source nature, the FPGA community can add support for new communication protocols and devices with full compatibility with all applications past and future. Developers can freely migrate their applications to different supported platforms with virtually no source code modifications. Furthermore, the API uses easy-to-understand session-level operations. This makes it possible for developers to build high-performance computing applications with nothing but a basic knowledge of C++ and Verilog. No knowledge of drivers, operating systems or communication protocols is necessary.

These characteristics dramatically lower the cost, effort and risk generally associated with developing applications on reconfigurable hardware. We believe that this will attract a new breed of FPGA users/developers and encourage reconfigurable devices in a more widespread range of applications.

References

[1] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J Stevens, F. Baijot, and E. Komp, "Achieving Programming Model Abstractions for Reconfigurable Computing," IEEE Transactions on VLSI, vol. 16, No. 1, Jan. 2008, pp. 34-44.

[2] M. Saldana, A. Patel, C. Madill, D. Nunes, D. Wang, H. Styles, A. Putnam, R. Wittig, and P. Chow "MPI as an Abstraction for Software-Hardware Interaction for HPRCs," presented at HPRCTA 2008.

[3] Download from http://research.microsoft.com/en-us/people/eguro/