Icestick IO

Benjamin Braun & Nate Chizgi

Spring 2017

CS 448H: Special Topics in Computer Graphics – Agile Hardware Design

Table of Contents

[Table of Contents 2](#_Toc477983624)

[Abstract 3](#_Toc477983625)

[1.0 Introduction 3](#_Toc477983626)

[2.0 Design 4](#_Toc477983627)

[2.1 Application Interface 4](#_Toc477983628)

[2.2 Communication Protocol Abstraction 4](#_Toc477983629)

[2.3 Compilation 5](#_Toc477983630)

[3.0 Implementation 5](#_Toc477983631)

[3.1 Making UART Fast and Reliable 5](#_Toc477983632)

[3.2 Printf Circuit 6](#_Toc477983633)

[3.3 Compilation and CPU side handler 7](#_Toc477983634)

[4.0 Evaluation 8](#_Toc477983635)

[5.0 Lessons Learned & Limitations 10](#_Toc477983636)

[5.1 CPU integration 10](#_Toc477983637)

[5.2 Reliable communication 10](#_Toc477983638)

[5.3 Printf statement collision, timestamping, scalability 10](#_Toc477983639)

[5.4 Multiple concurrent display types 11](#_Toc477983640)

[6.0 Related work 11](#_Toc477983641)

[7.0 Future work 11](#_Toc477983642)

[8.0 Conclusions 12](#_Toc477983643)

# Abstract

For a paltry $30, anyone can gain access to an icestick FPGA, a beginner device with a fully open source tool chain and dynamic user base. With a small amount of Verilog (or Magma) knowledge, it is straightforward to develop one’s own custom HW solution, download it to the icestick, and maybe see a few LEDs blink. This is often the point where beginners get left behind, or where more experienced developers extend a great deal of time and energy. Debugging more complicated programs becomes a daunting task requiring investment in external equipment (e.g. logic analyzer), and/or writing a customized data streaming interface to a CPU to gain visibility into their program. Many useful applications will then require dynamic interaction to and from a CPU to form the complete solution – e.g. data processing, image or video rendering, or just simple functional unit testing.

This paper presents a library which gives off the shelf input/output communication capabilities between a CPU and an icestick FPGA. The library balances simplicity with flexibility by providing interfaces for different forms of communication, all while abstracting away, and remaining generic to, the underlying communication protocol used (i.e. UART, SPI, PCIe, etc.). With this library, adding print statements to a complex icestick design takes fewer than 10 lines of magma code. Applications using the library can achieve high throughputs, even with the basic UART connection, of ~0.7MBps, and 350,000 print statements per second.

# 1.0 Introduction

Providing a streamlined off-the-shelf input/output capability between a CPU and an FPGA is an important and often overlooked part of developing meaningful user applications on the FPGA. We have decided to tackle this problem with the hopes of giving increased accessibility and usability to the devices. Our focus is on the icestick FPGA since it is an inexpensive commercially available device with a fully open-source toolchain.

We are targeting common communication patterns such as:

* **Visual Data**: graphics rendered on FPGA displayed on CPU screen in real-time
* **Debugging / Log Data**: printf() “statements” on FPGA which show in CPU terminal
* **Stream Data**: file on CPU fed into FPGA program, and output written back to a file on CPU

The icestorm pipeline comes equipped to flash a static program to the device, but it does not offer much for high speed input/output streaming between the device and the CPU. A specific goal of ours is to implement a solution capable of the high throughput requirements of our target communication patterns. An unmodified Icestick has only a built-in UART connection available which we are interfacing with, however our library abstracts away this input/output connection type.

Additionally, a complete solution requires the CPU side program to be provided which has specialized communication type handling, data streaming support, and other features to guarantee a robust data stream with efficient I/O. This can be challenging to support for multiple OSes and we have decided to provide a solution for linux which can be easily ported to IOS/Windows.

This paper first describes the design of the library programming interface, how the library abstracts away the underlying connection primitive, and how the library fits into the compilation flow of an Icestick application. We then describe the current implementation of the library and evaluate the library using a few simple applications. Finally, we list limitations of the library, discuss prior work, and outline how the library can grow in the future.

# 2.0 Design

## 2.1 Application Interface

We wanted our library to provide a generic communication layer between the Icestick hardware and the CPU. This is a challenge because different kinds of data may have different implied semantics, for example video data is usually streamed continuously while debugging or log data may come in short bursts and from multiple locations in a design. Furthermore, we wanted our library to be usable by beginning developers as well as usable in larger hardware design projects. Hence, we focused on a simple application interface that was targeted to the kind of data being communicated (i.e. visual, print statements, etc.):

* **ifio\_connection**(...)
  + Define connection properties. Pass result as argument to below:
* **ifio\_simpledisplay**(width, height, framerate, details)
  + make a width x height display @ framerate, other args in “details”
* **ifio\_display**()
  + “Advanced” version of simpledisplay (details omitted)
* **ifio\_printf**(bool valid, format string, arg\_description)
  + Prints data to CPU stdout on any clock cycle where “valid” line transition -> 1
* **ifio\_stream\_process**(“file\_in.txt”,”file\_out.txt”)
  + Exposes data from “file\_in.txt” to FPGA whenever FPGA sets “ready”
  + Exposes FPGA output to “file\_out.txt” whenever FPGA sets “valid”

## 2.2 Communication Protocol Abstraction

The library treats the underlying communication protocol in a modular way, to allow us to run over either UART (provided by the Icestick) or other protocols such as PCIe or SPI (which require hardware modifications to achieve on the Icestick.) Note that the library hides the details of the underlying communication protocol entirely from the client design, see the discussion of the library interface above.

Some of the challenge in supporting multiple communication protocols is that the protocols may run at a different clock domain than the Icestick design, and the protocols may require data to be sent in large blocks or batches, possibly with some support for retransmitting dropped blocks. We believe these different protocols can be packaged into a module with the following client-facing ports for sending data:

**- Ready wire**

* + The communication module sets this high whenever it can accept another data word.

**- Valid wire**

* + Should be pulsed for a single cycle when the client has data to send and the Ready wire is high.

**- Data**

* + We feed a fixed number of bits of data to the communication module at a time by latching the bits in a single cycle whenever Ready and Valid are both high. Unless the module can immediately accept more data the very next cycle, ready will go low until it can.
  + Choosing the bit-width of data requires some care. A bit-width of 8 is convenient for communicating text data (assuming 8-bit characters) while bit-width of 16, 32, or 64 might be useful for sending pixel or audio frame data. Ideally, the communication module can be parameterized for different bit-widths and our library chooses the correct one for the communication mode.

A similar abstraction works for receiving data as well, simply reverse the meaning of the Ready and Valid bits. The clock described above should match the client clock, internally the module should handle any discrepancy between that clock and the clock of the actual communication primitive.

Furthermore, the module should guarantee that a data word, sent on a cycle when Ready and Valid are both high, is reliably communicated to the destination. Failures to communicate should cause the communication module to keep Ready low until some explicit Reset signal is received.

This abstraction might need to be revisited for some more exotic communication protocol, i.e. one that requires data retransmission. Regardless, UART can be easily packaged into such a black box, see the Implementation section for details.

## 2.3 Compilation

Use of our library adds one step to compiling an Icestick design, which yields a binary that runs on the CPU to communicate with the icestick. The individual calls to the application interface add hardware to implement the communication and also add some metadata to a new “iospec” file. This iospec file is turned into C code for the communication binary for the CPU with an “iocc” script we provide.

# 3.0 Implementation

The current implementation provides only the ifio\_connection and ifio\_printf application calls, and only supports the UART connection type. The UART connection type accepts 8 bits of data in a cycle. We support a variable number of printf statements, each with a variable number of variable bit-width arguments.

## 3.1 Making UART Fast and Reliable

We found that the FTDI chip on the Icestick will support up to 12,000,000 baud which results in a reasonable 0.715MBps streaming FPGA -> CPU. This is compared to the UART demoed in class which achieved a mere 7 kBps. In order to use this higher speed, we set the baudrate of the FTDI chip using a function provided by the libftdi library on the CPU side. Then, the UART circuit we generate ticks according to a baud wire that pulses at that same rate. Running the FTDI at a lower baudrate (necessary if the Icestick is clocked to less than 12MHz) or adjusting the baud wire in the generated UART circuit (necessary if the Icestick is clocked above 12MHz) would ideally be done automatically by our library, however this is not currently implemented. This could be done by accepting a “baudrate” parameter to ifio\_connection when using the UART mode.

As mentioned in the Design section, we require the communication module to be reliable. By design, UART can be a lossy communication protocol if the CPU is running slowly and fails to read from the communication buffer before it overflows. From the reverse perspective, the FPGA may start feeding outputs to the CPU faster than the CPU is ready to accept them, leading to lost data.

We found that buffer overruns can be prevented by adding a CPU -> FPGA backpressure mechanism which effectively requires the CPU to signal ACK of a received buffer before the FPGA can reuse that buffer. We send data in blocks of 4080 bytes from the FPGA, and use ACKS to ensure there are never more than 2 such blocks “in-flight” at a time.

The UART connection has unused control pins available which have been repurposed for this backpressure mechanism:

* DTR : ‘ACK’ pin – provides a CPU -> FPGA buffer free signal
* RTS : ‘CPU READY’ pin – provides a CPU -> FPGA signal indicating CPU is ready to accept data

## 3.2 Printf Circuit

The printf circuit needed to have sufficient flexibility to be meaningful for a user application. Thus, we have designed it to have:

* A variable number of print statements (support currently between 1 and 4 with easy extension beyond this)
* A variable number of arguments to each statement, each argument with a variable bit-width
* Asynchronous and disjoint operation between each print statement
* Either level set or rising/falling edge triggered valid signals
* Latched arguments at valid signal in a single cycle, allowing the user circuit to continue operation

The overall circuit is broken into several sub-circuits

* **Input argument latch**: latches data arguments when printf is busy
* **Winner selection**: if any valid line goes high, pick one of the messages to transmit
* **Busy indicator**: Keep circuit busy until message has completed
* **Reset logic**: printf reset mechanism based on global and local conditions
* **Message counter**: Cycle through length of data in message
* **Backpressure mechanism**: Pause data input to connection when CPU is not ready to accept data
* **Connection block**: UART connection

The overall circuit for this is shown in the following figure:

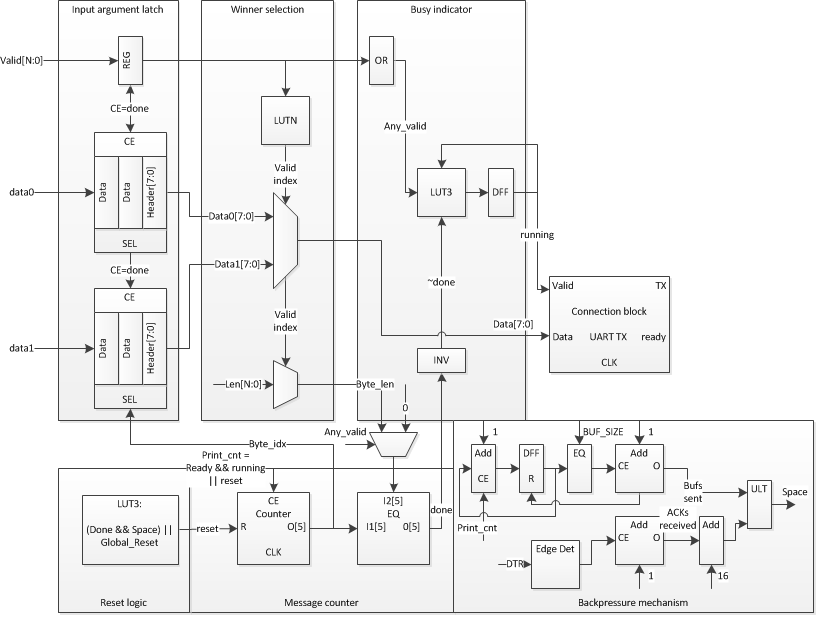


Figure 1: Printf circuit diagram

## 3.3 Compilation and CPU side handler

The iospec file generated is a raw JSON file describing each printf and a connection object. This JSON file is converted to a C file in the iocc script by using Python’s built-in string template module. The resulting C program relies on the libftdi and libusb Linux libraries to communicate with the Icestick connected over USB.

An example magma flow is shown in the following figure:

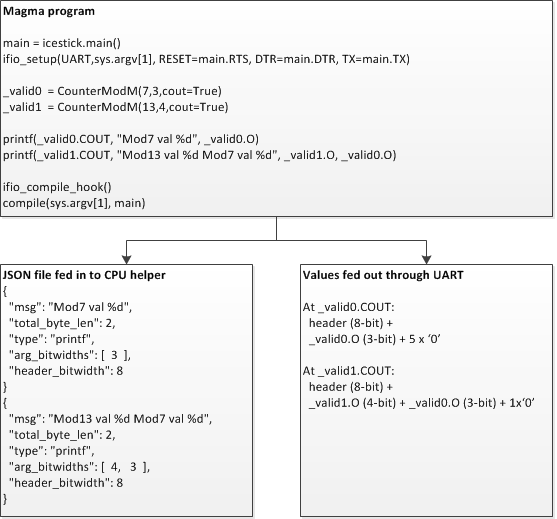


Figure 2: Printf magma program flow

# 4.0 Evaluation

We have implemented both printf and display routines and applied them to a series of real world applications.

Printf was evaluated with a few simple applications – e.g.

* Basic evaluation
  + printf(counter.COUT, “Counter time tick”);
* Circuit unit test framework
  + printf(edge(test.done), “In (%d, %d, %d) out %d”, test.in0, test.in1, test.in2, test.out);
* Pattern switch messaging (from homework assignment 2)
  + printf(pattern.switch, “Change to next pattern %d”, pattern.index);

Display was evaluated with the following application:

* Gradient Display: Render time-varying gradient on FPGA and display on CPU

For both examples, a very lightweight userspace API is exposed and intuitive to use. [Figure 2](#_2.3_CPU_side) above shows the extent of code required to instantiate printf in the use space – it is limited to (1) setup routine which passes the top level information into the printf circuitry as well as the connection type, (2) the actual print statements, and (3) the compilation hook:

* ifio\_setup(UART,sys.argv[1], RESET=main.RTS, DTR=main.DTR, TX=main.TX)
* printf(\_valid0.COUT, "Mod7 val %d", \_valid0.O)
* ifio\_compile\_hook()

Similarly for display(), we have shown the magma code required to interact with the display connection type for our gradient display application:

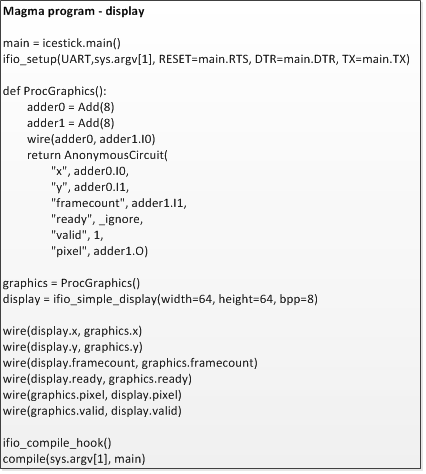


Figure 3: display() magma interface

For the basic counter print example, using a CounterModM that resets every 7 cycles at the default 12MHz Icestick clock will saturate the throughput of the UART, yielding 375,000 print statements per second on the CPU. This matches the theoretical throughput of the UART, as 12MHz baud with 16 cycles to emit an 8-bit data word leads to 750,000 bytes/second throughput (0.715MBps) and, for this example, each print statement transmits 2 bytes (1 byte header + 1 byte data.)

# 5.0 Lessons Learned & Limitations

We experienced several interesting challenges when working on this project.

## 5.1 CPU integration

We have provided a complete solution with both FPGA side circuitry as well as CPU side driver modifications and support programs. This has meant that the FTDI drivers and CPU hooks output from the program are dependent on the OS, the FTDI drivers being used, and other CPU specific information. Thus, having a generalized out-of-the box solution will not necessarily work across a wide range of CPUs. Currently we have limited the functionality to arch-linux OS and provide sufficient information to port the solution to Mac-OS or Windows.

Additionally, we have clocked the FPGA FTDI chip at its maximum rate of 12,000,000 baud, and modified the linux FTDI drivers to support this data-rate. We have heard that Mac-OS can only support up to 3,000,000 baud, which is another OS-specific limitation which would prevent our current solution from being picked up without modification to support dynamic baud rate in the connection configuration.

Generally speaking – one of the main things which we are trying to provide off-the-shelf support for is to have seamless CPU side integration and communication with the icestick – however there are still a number of hurtles which the programmer may need to go through in order to get our solution to work on the CPU side, thus potentially hurting part of our purpose.

## 5.2 Reliable communication

Providing a reliable communication interface from the FPGA to the CPU through UART required a substantial amount of circuitry. Just the backpressure mechanism to prevent dropped packets on the CPU ended up taking up more space than the rest of the printf and UART circuitry combined! Generally speaking, our solution should take up as little of the FPGA as possible, to allow for maximum user space circuitry and to keep the compilation time reasonable.

Future revisions should focus on circuit optimization perhaps at the expense of robustness of the connection by supporting dropped packet handling on the CPU side. This can perhaps be achieved through modifications to the data stream and header formats. Generally speaking, this requires additional research.

## 5.3 Printf statement collision, timestamping, scalability

Currently when two print statements collide at a particular clock, we drop one of them based on a fixed-order arbitration scheme. Thus, we see missing print statements in more advanced examples we have put together. This can be improved or enhanced in several ways – most trivially, instead of dropping a message, we can buffer a colliding message to be sent serially after the current message completes. This would require a relatively small amount of change to our existing circuitry. We can also indicate a ‘drop’ bit in the next message header if we end up dropping a message due to FPGA feeding messages too quickly to the I/O connection.

We have also implemented a timestamp for each print statement, however we are currently taking the timestamp from the CPU side. Thus, the value logged is relative to when the message gets processed on the CPU, and not necessarily when the FPGA sent the message. This gives good approximate timing information at no cost of I/O bandwidth, but is not acceptable for more precise requirements such as waveform viewing. Adding something along the lines of “fpga\_timestamp=True” to the ifio\_setup() routine may be a good parameter which optionally attaches an N-bit counter timestamp to each message header.

Adding a printf adds connections for the argument wires into latches, which are then muxed down to select a specific printf to print at a time. What this means is that adding many printfs to a design adds hardware overhead proportional to the number of printfs, and furthermore creates a large number of long-distance routing connections that may increase compile times. Hence, for the purpose of debugging, recommended practice is to enable or disable sets of printfs based on some debugging flags at compile-time (i.e. enable all printfs regarding a particular component that is being debugged.)

## 5.4 Multiple concurrent display types

Supporting multiple display types requires asynchronous interleaving of different types onto the same connection type. For example, if we want print statements to run while also streaming a display to the CPU, we will need to make the mechanism robust to different data type collisions and perhaps limit the maximum length of a message to prevent long blocking of one type vs another. This would require a fancier header and additional information fed to the CPU to stitch/interleave data types. Perhaps embedding the sub-message length with the header or simply embed a header vs data bit with each UART packet can be done to support dynamic interleaving.

# 6.0 Related work

Within our class, there are several examples of related work. Pat has given a UART which streams “Hello world” several times a second from the FPGA, Ross has also demonstrated a real time edge detection on an FPGA in class. In Pat’s example, the usefulness is limited due to the low data rate of the UART provided as well as the full message passing from the FPGA and lack of multiple asynchronous message and argument support. Ross’s example was inspiring, however a lot of customized CPU and FPGA I/O effort was put into this demonstration which is not applicable to the Icestick, and it also required custom bootstrapping between the FPGA and the CPU and a custom CPU-side driver.

Much of our goal is to give FPGA accessibility to more novice programmers and to this extent the processing project ([www.processing.org](http://www.processing.org/)) and openFrameworks (openframeworks.cc) have related goals. The processing project has been successful in attracting artists and creators with little programming experience to programming.

# 7.0 Future work

We have only implemented a single connection type (UART), and only the TX side of UART. This should be extended to a variety of connection types which may be supported on Icestick with HW modifications, or come with built-in support on other FPGAs. We shall also extend the library to cover other more interesting bi-direction applications which would require implementation of a functional UART RX (or other connection type RX) on the FPGA.

We have only implemented a couple of commonly used communication patterns (printf and display). This provides a good initial coverage for debug and graphics rendering on the FPGA, but limits the usability for many user applications. Adding bi-directional streaming is the key addition which would best expand the application coverage. Additionally, adding timestamp and waveform viewing option would be nice to have a primitive limited scope analyzer capability.

An interesting application of our library is as a component of an interactive debugger. Here, the debugger should have some feature to pause and resume all circuits on the FPGA. The debugger could then communicate to the FPGA via our library to print out some values from the FPGA as the debugging progresses (the debugger would need to be aware of which part of the circuit belongs to our library and allow those circuits to resume while pausing all other parts.) Hence, the designer of the debugger need not know the details of communication primitives and how to achieve reliable, fast transmission with them. A note is that we probably want to generate a static library that the debugger CPU-side binary links against rather than a whole CPU-side binary for this application. Here, the ifio\_printf primitive is actually very close to what the debugger needs, however it may need to be a modified form that just outputs the raw byte arguments without a format statement. And, we should probably emit a CPU-side static library that links against the debugger binary rather than a whole CPU-side binary for such an application. But, these are straightforward extensions to the library.

# 8.0 Conclusions

We presented a library for communication between the Icestick and the CPU that is fast, easy-to-use, flexible, and designed with extensibility in mind. We are eager to collaborate in furthering the library and look forward to pull requests.

We thank Dr. Pat Hanrahan for running such an interesting course and developing Magma, and Ross Daly for providing interesting feedback as we developed this library.

The code can be obtained at the following url: https://github.com/bjmnbraun/icestick\_fastio