Icestick FastIO

Benjamin Braun & Nate Chizgi

Spring 2017

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

Table of Contents

[Abstract 3](#_Toc477864260)

[1.0 Overview 3](#_Toc477864261)

[2.0 Implementation 4](#_Toc477864262)

[2.1 Connection Type 4](#_Toc477864263)

[2.2 Printf Circuit 4](#_Toc477864264)

[2.3 CPU side handler 5](#_Toc477864265)

[3.0 Evaluation 6](#_Toc477864266)

[4.0 Lessons Learned & Limitations 7](#_Toc477864267)

[4.1 CPU integration 7](#_Toc477864268)

[4.2 Reliable communication 8](#_Toc477864269)

[4.3 Printf statement collision and timestamping 8](#_Toc477864270)

[4.4 Multiple concurrent display types 8](#_Toc477864271)

[5.0 Related work 8](#_Toc477864272)

[6.0 Conclusions 8](#_Toc477864273)

# 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 formulation the complete solution – e.g. data processing, image or video rendering, or just simple functional unit testing.

This project provides a library which gives off the shelf input/output communication capabilities between a CPU and an icestick FPGA.

# 1.0 Overview

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.

It is important to have a simple, user friendly interface but still provide sufficient flexibility required of developers. By targeting only a subset of the more common patterns, we are able to keep an intuitive easy to use API:

* **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”

# 2.0 Implementation

We have implemented printf capability for demo purposes. The circuit implemented supports:

* Connection Type
  + Fast Reliable UART capable of 0.715 MBps
* Printf Circuit
  + Variable number of print statements with dynamic # of variable bit-width arguments
* CPU side handler
  + Supports CPU ready & ack/nack feedback for reliable connection
  + Modifications for faster connection types
  + Connection pattern type JSON file input autogenerated from circuit

## 2.1 Connection Type

We found that the FTDI chip provided will support up to 12,000,000 bad 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. The overclocked UART requires modifications to the FTDI driver provided as part of our CPU side handler.

In addition to speed, we require a reliable connection with the CPU. By design, UART can be a lossy communication protocol if the CPU is running slowly and failing to read the buffer before it is overrun. Additionally, FPGA may start feeding UART outputs to CPU before the CPU is ready to accept them. Thus, we have added in a CPU -> FPGA backpressure mechanism which effectively requires the CPU to signal ACK of a received buffer before the FPGA can reuse that buffer. The internal FTDI buffer has been broken up into 16 buffers each of length 510\*4 bytes.

The UART connection has unused control pins available which have been repurposed for above:

* 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

## 2.2 Printf Circuit

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

* Variable number of print statements (support currently between 1 and 4 with easy extension beyond this)
* 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 to allow user circuit to continue operation

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



Figure 1: Printf circuit diagram

## 2.3 CPU side handler

On the CPU side, there is an FTDI driver provided which supports higher data-rate UART, as well as ACK/READY feedback mechanism and printf driver. Magma compilation flow provides a JSON file which is input to the CPU side file which provides message information. An example magma flow is shown in the following figure:



Figure 2: Printf magma program flow

# 3.0 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 a simple application based on homework # 2:

* Pattern display – display LED lightshow pattern 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:

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

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



Figure 3: simpledisplay() magma interface

# 4.0 Lessons Learned & Limitations

We experienced several interesting challenges when working on this project.

## 4.1 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.

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.

## 4.2 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.

## 4.3 Printf statement collision and timestamping

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 fastio\_setup() routine may be a good optional parameter which optionally attaches an N-bit counter timestamp to each message header.

## 4.4 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.

# 5.0 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 openFramworks (openframeworks.cc) have related goals. The processing project has been successful in attracting artists and creators with little programming experience to programming.

# 6.0 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 display patterns (printf and simpledisplay). 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. One can also envision adding support for CPU side interface to pause and resume all circuits on the FPGA and add in an interactive debugging platform between the CPU and the FPGA which allows user to read or write arbitrarily at any register within the circuit when paused. This can allow for a dynamic and interactive GDB option with the icestick FPGA through the comfort of a CPU interface.