DISTRIBUTED COMPUTING PROCESSOR FOR SIGNAL PROCESSING APPLICATIONS

Krystal Peterson, Samuel Richter, and Adam Schafer
with Steve Grant and Kurt Kosbar
Telemetry Learning Center
Department of Electrical and Computer Engineering
Missouri University of Science and Technology
Rolla, MO, USA

ABSTRACT

Many signal processing, data analysis and graphical user interface algorithms are computationally intensive. This paper investigates a method of off-loading some of the calculations to remotely located processors. Inexpensive, commercial off the shelf processors are used to perform operations such as fast Fourier transforms and other numerically intensive algorithms. The data is passed to the processors, and results collected, using conventional network interfaces such as TCP/IP. This allows the processors to be located at any location, and also allows potentially large caches of processors to be shared between multiple applications.

INTRODUCTION

Advances in digital technology means that computing power is sometimes not an issue for signal processing, data analysis and graphical user interface operations. This is particularly true for desktop and server applications, or where size and power constraints are not severe. Desktop and server technology provide orders of magnitude more processing power than what was available even a few years ago.

However, there are applications where processing power is still an issue. For example, some complex signal processing algorithms that were not attempted in the past, are now being requested. As handheld devices become more prevalent, there is a desire to use them as the user interface. Even when these devices have the ability to perform advanced calculations, the limitation on battery life, and the need to quickly update displays, may cause them to be unable to process the data in a timely manner. Yet another application is with big data, where a tremendous amount of data must be sorted, searched, or otherwise manipulated to extract the information the user requires.

The data processing for these numerically intensive applications can be off-loaded to some sort of server, or cluster of computers. There are ample software and hardware tools available to accommodate this. But in some applications, one may need just a little more processing power, and a conventional server, or laptop system, may not be necessary. There are also situations where the processing power needs to be scalable with the algorithm complexity or data set size. It would be helpful if one could conveniently double or triple the computing power without having to reconfigure the system. It would also be convenient if the communication between the user interface and computing engines were over a commercial off the shelf network, such as TCP/IP over wired or wireless ethernet.

This paper describes a project that is intended to address many of these issues. We investigated methods of executing a common numerically intensive algorithm, the fast Fourier transform (FFT), on a remote processor. While the approach can be used for many different types of processors, we focused on a Zynq-7000 series programmable system-on-chip based device. This type of device can be run independently of a host computer. Data can be transferred in, and results returned, over a conventional ethernet bus, making the solution scalable.

The device runs a Linux operating system in an ARM processor to provide control and other interface functions, while providing a programmable logic array to speed numerically intensive calculations. In our application the processor will read data from the ethernet port, and store it in local memory, as illustrated in figure 1. The programmable logic will read the data from memory, perform the FFT operation, and store the results back to memory. When the processor becomes aware the FFT has completed, it will transmit it back to the user interface via ethernet. We chose to implement the processor code in embedded C [1], while the FFT system was implemented in VHDL [2, 3].

SYSTEM ON CHIP

Each node of the distributed signal processor will be based on a system on chip (SoC) device. There are many such devices which are commercially available. The high level block diagram of the SoC we require for this project is shown in figure 2.

The minimum input/output function would be to communicate with a gigabit ethernet connection. However, for programming and debugging purposes, it would be helpful to have at least one universal serial bus (USB) and a flash memory card interface such as a SD card interface. Many SoC devices have a wide range of input/output ports, so we did not anticipate this modest requirement would be a significant design constraint.

We found SoC that support additional I/O such as video interfaces, user interfaces, and a variety of bus structures. It was tempting to select one of these, as programming and debugging could be easier if one was able to have a conventional personal-computer style interface with the SoC. Unfortunately, the cost of these SoC were substantially higher than ones that had more limited interfaces. Since we would like the system to be easily scalable, it was decided to limit cost by using a SoC with more modest interface options.

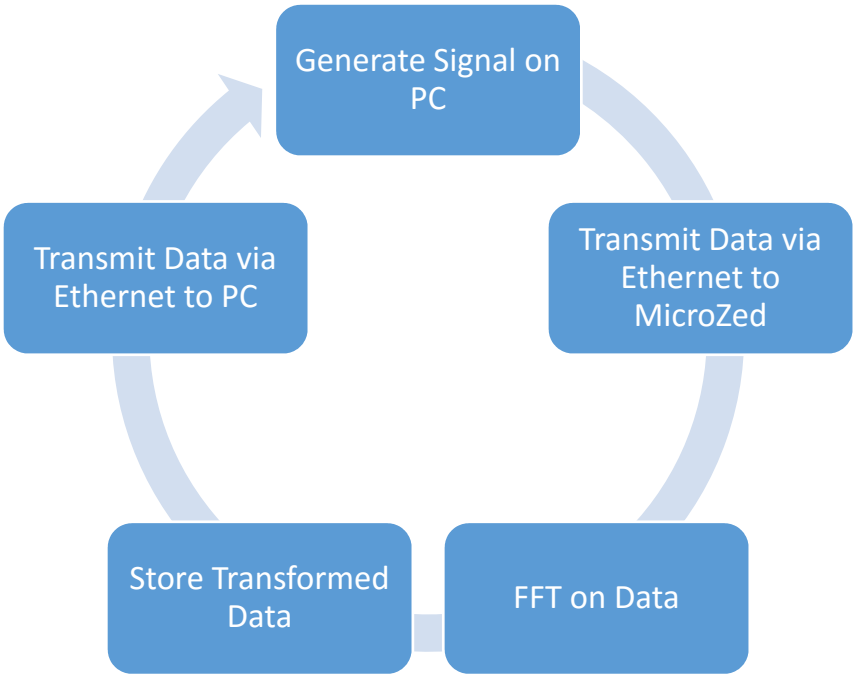

Figure 1. High Level Structure

Data will be moved onto, and off of, the SoC using transmission control protocol/internet protocol, TCP/IP over a 100 Mbyte or 1 GByte eithernet connection. These are best effort delivery protocols, with no specification on latency. This brings up the problem of buffering both input and output data. To use the SoC efficiently, when the PC is able to establish a connection to the SoC, it may be necessary to transfer a substantial amount of data, to provide the SoC with enough of a computing challenge that it has something to work on when the TCP/IP connection is handling traffic from other users. When the SoC is done with its processing, it may be necessary to buffer the outputs in local memory, while waiting for bandwidth on the TCP/IP link. Because of these requirements, it was decided that external memory may be necessary for the SoC, in addition to memory required for booting the SoC system. As we did not want to limit ourselves to one specific memory format, we looked for SoC that allowed multiple formats.

The core of the SoC consists of a processor, and programmable logic array. The processor block may consist of either a single processor, or perhaps two or more cores [4]. We anticipate the processing load will be reasonably light, as it involves primarily moving data to/from memory and

to/from the PC, along with some status and handshaking data which will be transferred between the processor and PC. Since these processing tasks are rather modest, a single core processor is anticipated to be sufficient.

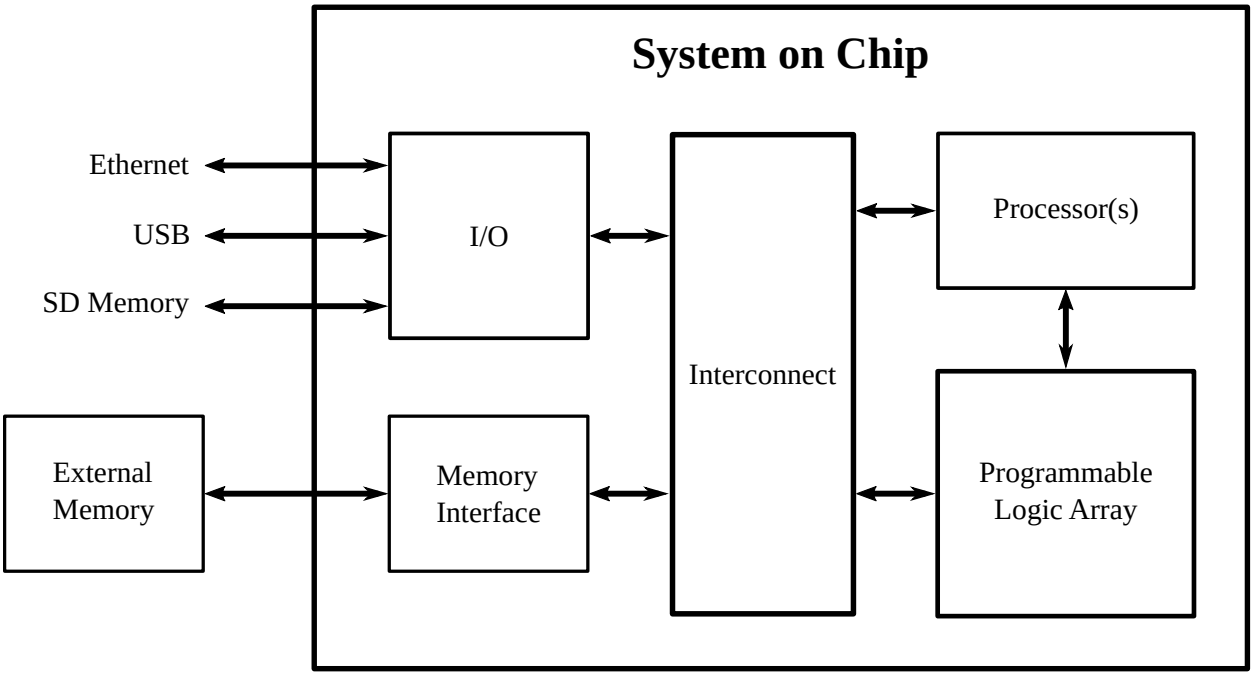

Figure 2. Minimum SoC Architecture

The programmable logic array (PLA) will be responsible for performing the discrete Fourier transform (DFT) / fast Fourier transform (FFT) operations [3], and little else. Programmable arrays that are easily configured for FFT operations were sought. The size of the PLA was not a large concern during the conceptual design stage. A larger PLA may be necessary when larger FFT operations need to be performed. The FFT algorithm is very scalable, and the code used to configure the PLA can be easily modified to accommodate larger FFT operations. For that reason, during the conceptual design stage we focused on SoC with modest sized PLA.

There are numerous SoC devices available that satisfy the basic constrains outlined above. It is not the intention of this work to document all the options currently available. This would be a rather futile task anyways, as the field is rapidly evolving. The particular solution we choose was a Zynq-7000 series SoC from Xilinx. An approximate block diagram for this series SoC is shown in figure 3. The number of interfaces is well beyond what this application requires, as is the dual core ARM processor. The cost was in the range we were looking for, and the support software available for developing the system was attractive.

SYSTEM ON MODULE

Our initial intention was to design a printed circuit board around the SoC selected above. While this still might be an option for large scale systems, for development purposes we opted for a system on module (SoM) solution. The module, or development board, contained some supporting interfaces and hardware that the SoC required. A block diagram of the SoM used for this project is shown in figure 4.

The USB interface was helpful for programming and debugging the SoM, along with other interface busses available on the board. The RJ45 / 10-100-1G Ethernet port will be the way data is transferred to and from the device during operation. The Micro SD card socket was not absolutely necessary, but does provide another programming and debugging option which we found helpful. A photograph of the board is in figure 5, where the RJ45 jack can be used to get a sense of scale.

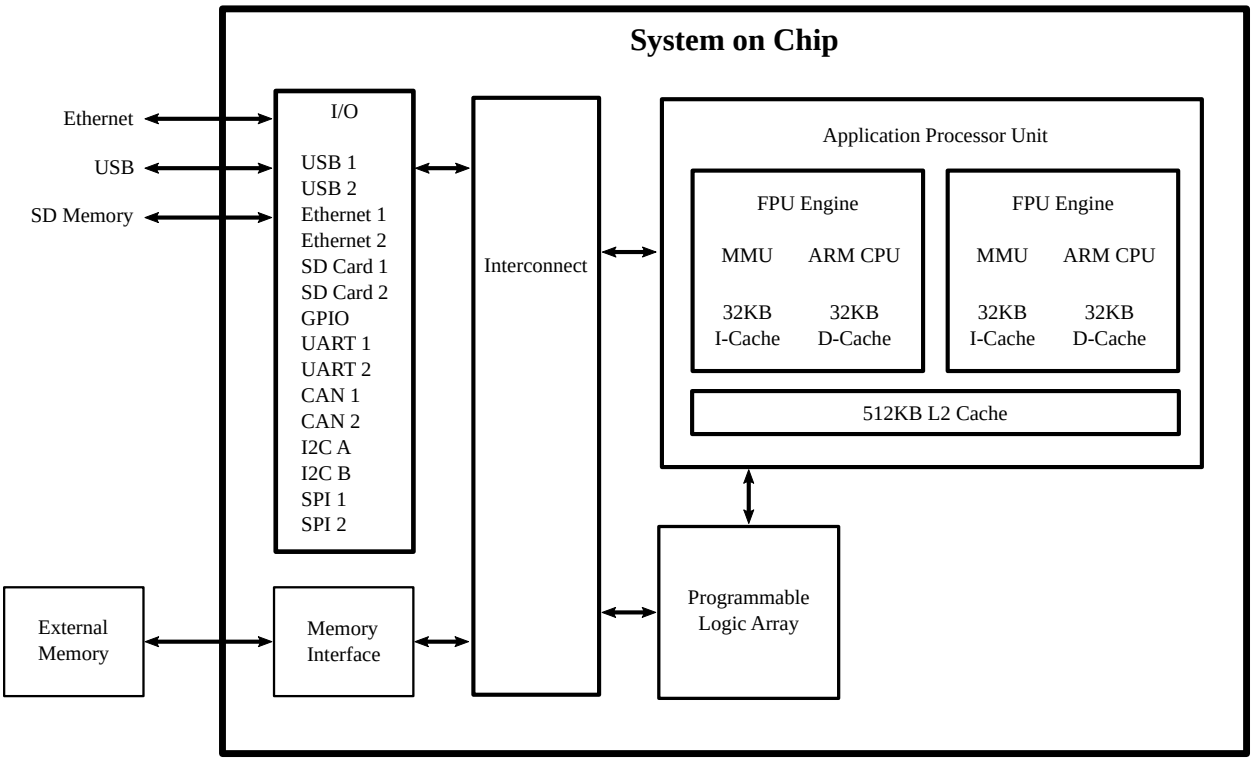

Figure 3. SoC Architecture

OPERATING SYSTEMS

Prior to the start of coding, we needed to select an operating system for the application processing unit for the SoC [5, 6]. There are a number of choices for this.

NO OPERATING SYSTEM

This is sometimes referred to as a bare metal system [7]. The processor code is all written by the end user, and does not use any libraries, scheduling, or other software typically available on machines that are running an operating system. This is an attractive approach in so much as it can make most efficient use of the hardware. Since we did not anticipate implementing advanced or complex programs on the board, we initially decided to go with a bare metal system.

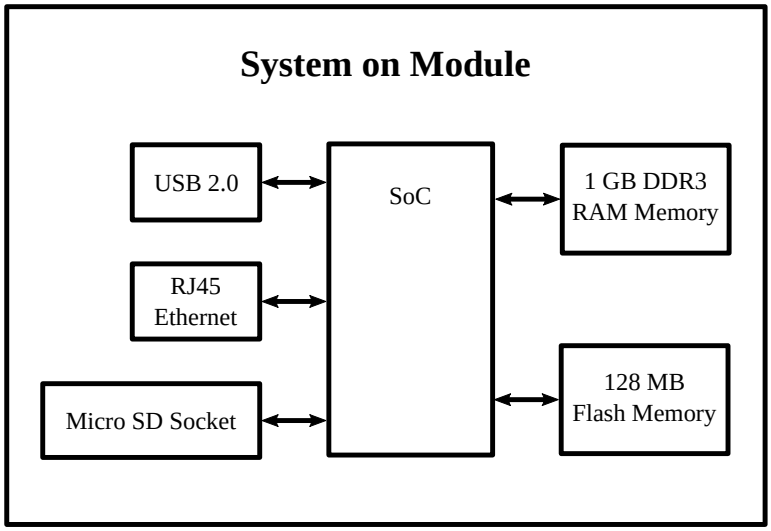

Figure 4. SoM Architecture

The bare metal approach was useful for much of our code, but we ran into a significant issue when it came to the Ethernet TCP/IP connections. For ease of use, we wanted to be able to communicate with the device from any other TCP/IP equipped machine, such as a typical personal computer. This required that we implement at least part, if not all, of the TCP/IP stack protocol. This protocol is well documented, but involved many more details that we had time to implement. Based on this constraint, we elected to forgo the no operating system approach.

UNIX / LINUX OPERATING SYSTEM

There are a variety of Unix / Linux operating systems that will run on the processors on the SoC [5]. Since we were sure all of these would implement TCP/IP protocols, it solved the problem of having to code the communications protocol stack from scratch. However, like all operating systems, there was a substantial amount of code we had to load onto the device. In addition, these

operating systems tend to slow the execution of the signal processing operations. The more advanced scheduling and file structure systems were not needed for our design, and appeared to significantly impact the speed of operation.

Figure 5. SoM Board

We also had some concerns about the efficiency of interfacing Linux to the programmable logic portion of the board. To make most efficient use of the programmable logic, we need some predictable latency, and timing, in our code. Operating systems such as Linux appear to use best effort scheduling, so it is at best difficult to determine how long a series of commands will take to execute, or exactly which order commands will execute.

Linux has built in support for multiple processors, which would allow us to use both of the processing units available on the SoC, without having to put much effort into thinking about how to do this. It is unclear if this is a big advantage for our application, as we expect the programmable logic array to do the majority of the numerically intensive calculations.

REAL TIME OPERATING SYSTEM

The next solution we investigated was some of the real time operating systems (RTOS) available [5]. These appeared better suited to interface to the programmable logic array, necessary to implement the FFT operations. Some also allowed TCP/IP interfaces, although we struggled to get these to execute in as seamless a manner as the Linux based solutions. The RTOS also made it more difficult to make efficient use of the multiple processors available in the application processing unit.

HYBRID APPROACH

We ultimately decided to use a hybrid approach, using both a Linux based system and a RTOS. The dual processor architecture is set up so it is possible to use different operating systems with the two different processors. We use one of the processors to implement a Linux kernel, primarily to handle communication with TCP/IP devices for control, to collect data, and to deliver the final results.

The Linux processor stores the data we need to process in RAM, and reads out the results from the same memory. The RTOS then tracks the operations of the RAM and programmable logic array. The RTOS programs senses when the Linux system has provided new data to be transformed in RAM and schedules a time to conduct the operation in the programmable logic. The RTOS provides that programmable logic with the data on where to locate the information in RAM, and when to start the necessary transforms. When the programmable logic has completed the transforms, it notifies the RTOS, which tells the Linux processor it can transfer the data to the outside world.

We had some concerns, and ran into some problems, with getting the RTOS and Linux processors to properly share memory resources. Linux is built to recognize and use memory management units (MMU), to help keep the various programs from interfering with each other. The RTOS that we tried did not work as well with the MMU, causing collisions and other problems with memory access, which we were still working to resolve.

CONCLUSION

It appears to be feasible to implement numerically intensive computing tasks, such as FFT operations, on the programmable logic portion of a SoC device. To allow for easy transfer of data from other devices over a commercial protocol such as TCP/IP over ethernet, it is helpful to use one of the SoC processors to run a version of an open source operating system such as Linux. As the SoC frequently come with multiple processor cores, the second core can be used either as a bare-metal device with no operating system, or one can use a real time operating system to optimize the transfer of data from the local memory to and from the programmable logic array.

A full custom printed circuit board could be the most economical way to implement these processors when a large number of them are required, or if there are special design constraints that justify the non-recurring costs associated with such a project. But for limited runs and prototype systems, commercially available Systems-on-Module solutions are current available that appear to provide sufficient memory and interface resources to allow one to quickly implement these algorithms.

REFERENCES

[1] Pont, M. J., Embedded C, Addison-Wesley, Indianapolis, IN, USA, 2002

[2] U. Meyer-Baese, Digital Signal Processing with Field Programmable Gate Arrays, 4th Ed., Springer, New York, 2014

[3] Z. Derafshi, J. Frounchi and H. Taghipour, "A High Speed FPGA Implementation of a 1024-Point Complex FFT Processor", IEE International Conference on Computer and Network Technology, April 2010, Bankok Thailand

[4] Z Ou et al., "Energy- and Cost-Efficiency Analysis of ARM-Based Clusters", 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, May 2012, Ottawa, ON

[5] H. Posadas et. al., "Early Modeling of a Linux-Based RTOS Platforms in a SystemC Time-Approximate Co-simulation Environment", 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, May 2010, Carmona, Sevile

[6] S. Mohanty and V. Prasanna, "Rapid System-Level Performance Evaluation and Optimization for Application Mapping onto SoC Architectures", 15th Annual IEEE International ASIC/SOC Confderence, Sept. 2002

[7] D. Hardin, "Real-Time Objects on the Bare Metal: An Efficient Hardware Realization of the Java Virtual Machine", Proceedings of the Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2001, Magdeburg