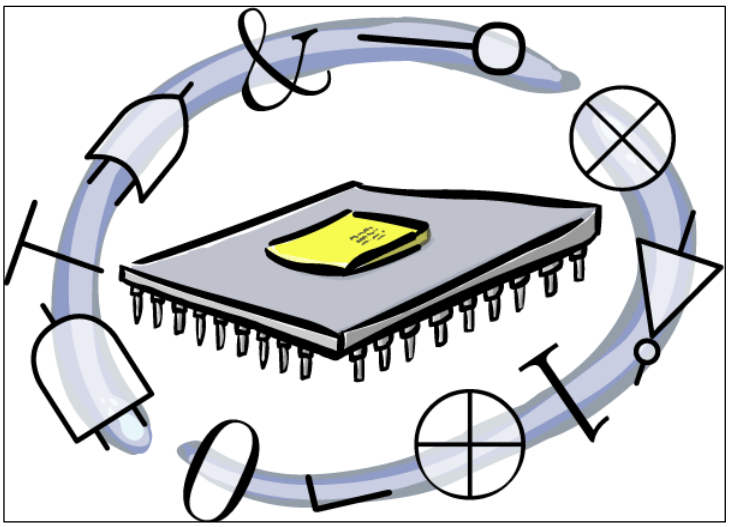
FPGA Analysis and Design of Embedded Processor Cores

A) Introduction

This report serves as the first report in compliance with the undergraduate independent study course titled “FPGA Analysis and Design of Embedded Processor Cores” during the 2005 fall semester at Arizona State University. The objective is to study the benefits of FPGA’s with embedded processors. A comparative study will be made with a few of the more popular options available today. This would include both “hard” cores (e.g. PowerPC in the Vertex-II Pro FPGA) and “soft” cores (e.g. PicoBlaze 8-bit microcontroller core offered by Xilinx).

The Field Programmable Gate Array (FPGA) is a relatively new technology that has made a huge impact on embedded computing. The FPGA is a programmable logic device that gives its designer, not its manufacture; the flexibility to implement virtually any customized digital logic circuit. This has given digital logic designers a revolutionary new level of flexibility at an increasingly smaller cost and development time. The FPGA is based on CMOS technology and does not have any on-chip non-volatile memory. Therefore it has to be programmed or configured at each power up using an off-chip non-volatile memory source such as flash memory. This initializing boot up time is very small and will meet most application specifications. The programming languages used to program FPGAs are called Hardware Descriptive Languages (HDLs) of which there are two commonly used; VHDL and Verilog.

So why would a designer use an FPGA? One very big reason is integration. The ability to decrease the number of components on your application board saves cost and power consumption. In the early days of FPGAs, they would be used to replace 7400 style discreet logic chips. But as CMOS technology has improved, FPGAs have been getting cheaper and bigger, not only making them more cost effective, but also increasing the complexity of logic implemented inside the FPGA providing for more and more integration.



Not only have the size of the FPGA increased, but a number of key resources such as on-chip memory, hardware multipliers, and Digital Clock Managers (DCMs) have been built into an FPGA making it possible to implement an entire System-On-Chip (SOC) application. This could include one or more “soft-core” processors described entirely with an HDL language and programmed into the FPGA’s reconfigurable logic. This is now referred to as a system-on-a-programmable-chip (SOPC) application. Another feature of some of the latest FPGAs include an actual hardware processor core manufactured into the FPGA and interfaced with the FPGA’s reconfigurable logic. They are called “hard-core” processors.

B) Hardware or software? Reasons to use an FPGA embedded core.

When it comes to considering the use of an embedded processor in your FPGA, there are two different perspectives. One perspective is from a designer already accustomed to developing applications with a stand alone microprocessor. They may or may not have already used an FPGA for other logic functions and are considering embedding the processor within the FPGA. The question for this person might be; why integrate the processor into the FPGA? The other perspective is from a designer who may never have used a microprocessor and is only concerned with best utilizing the FPGA for the application at hand. For which case, the question then becomes; why use a processor at all? I intend on answering both these questions below.

First, there is the inherent flexibility of an FPGA and then there is the freedom to split your application's tasks between hardware and software. First the flexibility.

Never before has an embedded application designer had such flexibility as that available in today's FPGA solutions. In a traditional embedded microprocessor application, a designer is restricted to the number of I/O pins available, the peripheral functions available, as well as which functions can be implement on which pins. This can make a Printed Circuit Board (PCB) design very complicated and costly. An FPGA is free to rout any logic function or peripheral function to any I/O pin on the chip. An example of a peripheral function includes PWM, USART, I2C, SPI, and Timers. A processor implemented on an FPGA is free to customize just the right number of needed peripherals need making for a more efficient application. Similar functions can also share resources providing for even more efficiency.

Also, once the PCB and hardware is built, the number of available changes or upgrades becomes very limited. With an FPGA solution, simple modifications to the HDL code can fix bugs or implement significant system level upgrades providing for increased life span and support for minimal cost. A FPGAs design and testing can continue even after the application has been manufactured by re-programming the FPGA in-field. Then there's the issue of portability. An FPGA's HDL Intellectual Property (IP) is much easier to port to newer technologies, packages, or even different platforms entirely when compared to the traditional stand alone microprocessor which has to be partially or fully re-designed each time it changes platforms. A processor's migration into an FPGA provides gained flexibility, performance, decreased costs, and sooner time to market. No more obsolescence here.

Also, a single processor has a limit to the number of tasks in can implement at once. As the number tasks increases, so does the complexity and effort required to test and debug. Using multiple processors solves this problem by simplifying the software development. Doing this is much easier on an FPGA. The only limitations are the size of the logic within the FPGA as well as the number pins available on the package. Tools such as the Chipscope from Xilinx make monitoring and debugging signals within the FPGA very easy.

Now the question, should a task be done in hardware or software?

Some background; A traditional standalone processor executes software or code usually written in C which is made up of a listing of sequential instructions. Indeed, almost anything can be done in code by sequentially manipulating data stored in memory or registers and then manipulating the voltage presented at I/O pins by again, writing to the correct register. This can get very complex, slow, and very tedious. So a traditional processor will also come with a set of hardware modules called peripherals that will implement certain functions in hardware. A processor will manipulate these peripherals through registers. An example would be a PWM module. Instead of writing code that will time out and manually toggle the I/O pin at the correct duty cycle and period, we can simply write a duty cycle and period value to the correct registers and the hardware module will automatically run the I/O pin at the correct duty cycle and period. This leaves the processor free to do other stuff. A processor, like a Digital Signal Processor (DSP) for example, can also assign a hardware module to a dedicated instruction code like multiplication. It is common for a DSP to have a Multiply and Accumulate (MAC) unit that can complete a multiplication and addition operation in one instruction. This dedicated hardware greatly improves performance and speed. The processor manufacture dictates both the instruction set and the peripheral set. Wouldn't it be neat to customize

the entire processor to have exactly the hardware and functionality you need and nothing you don't need? With an FPGA you can.

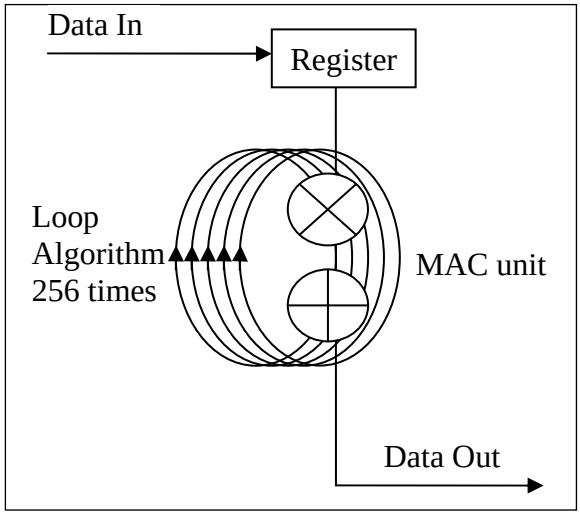


Figure 1 A 256 tap FIR filter on a DSP

And this is where the original question comes in. What should be done in dedicated hardware and what should be done in software run by the embedded processor? Do we need a processor at all? A processor, by its very nature, executes its tasks sequentially or serially. It does do a very good job of time sharing its resources between tasks. But there is a limit to the number of things it can do at one time. The FPGA has an added benefit of being naturally conducive to parallelism. Custom logic can be configured to perform many things at once. The figures one an two demonstrate the difference when implementing an FIR filter. A DSP processor will execute a MAC operation 256 times sequentially in a loop for each output sample. The FPGA can compute the whole sample at once using dedicated registers and multipliers for each tap of the filter.

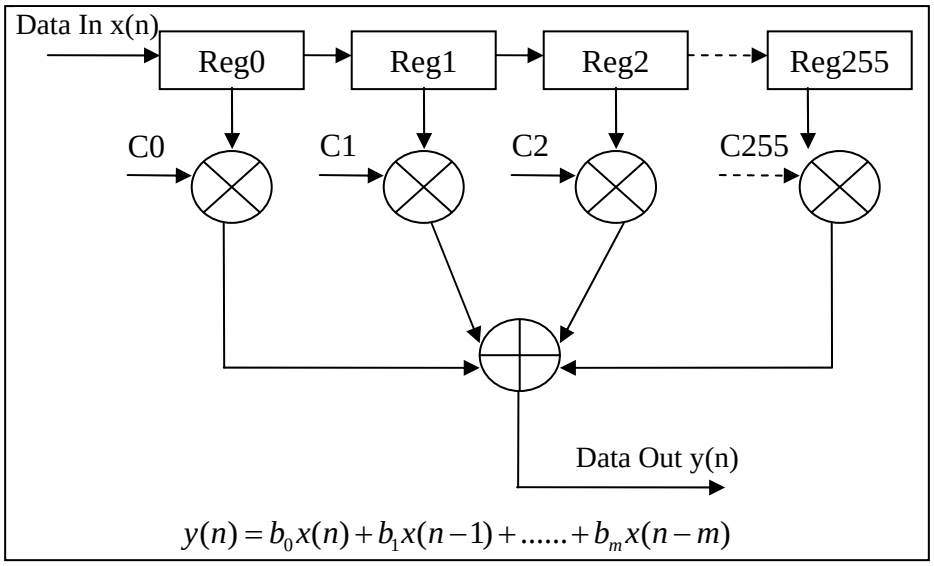


Figure 2 A 356 tap FIR filter on an FPGA

So why not do everything in parallel? The FPGA has its limits. Some of the lower end FPGA's have very few hardware multipliers. The excellent memory management and time sharing properties of a processor can make use of the hardware multipliers in an efficient manner. More efficient than simply adding more lower speed custom logic multipliers. The embedded processors are also great for implementing complex but non-time critical state machines.

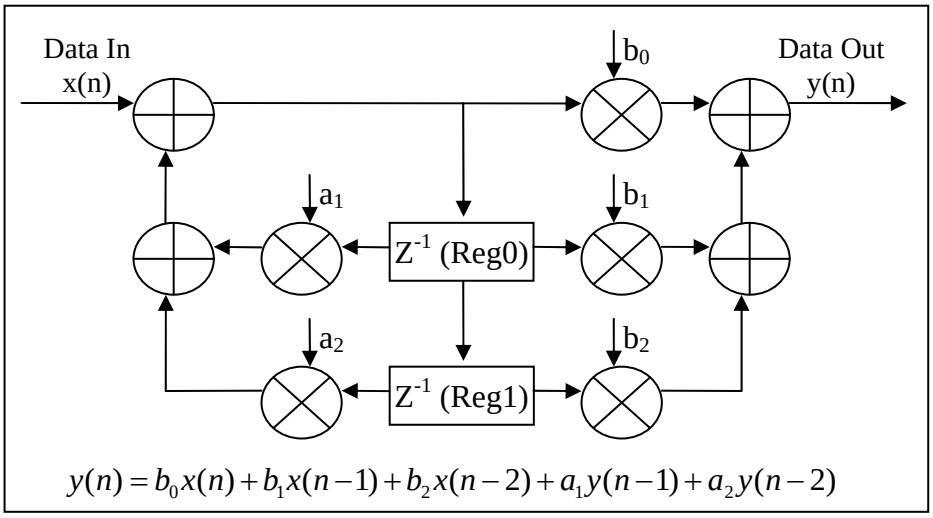


Figure 3 An IIR filter on an FPGA

C) Soft cores

Some of the more popular soft core processors include the ones offered by the FPGA manufactures. For example, the Nios from Altera and the MicroBlaze from Xilinx provide a wide range of customizable options. Table 1 shows some of there respective features.

TABLE I
FEATURES OF COMMERCIAL SOFT PROCESSOR CORES

Feature	Nios 3.1	MicroBlaze 3.2
Datapath	16 or 32 bits	32 bits
Pipeline Stages	5	3
Frequency	up to 150 MHz[1]	up to 150 MHz[1]
Gate Count	26,000–40,000	30,000–40,000
Register File	up to 512	32 general purpose
	(window size: 32)	and 32 special purpose
Instruction Word	16 bits	32 bits
Instruction Cache	Optional	Optional
Hardware Multiplier	Optional	Optional

Most of these kinds of processors will be programmable in C. C is the de-facto standard in the embedded systems market and will provide for easily portable IP.

These soft processor cores will also come with the tools needed to develop with them. Xilinx for example sells it's Embedded Development Kit (EDK) which includes all the needed IP, including the actual MicroBlaze soft core, The GNU C code development tools, as well as the Processor Core Configuration Tool which automates most of the configuration of the processor. Some of the configuration options include register file size, hardware multiply and divide, interrupts, and I/O hardware. Core Connect is another peace of IP that provides the best means foe connecting the processor to various peripherals.

Another very common soft core is the PicoBlaze also offered by Xilinx. PicoBlaze is a very small 8-bit microcontroller which can fit in only 96 CLBs of the FPGA. Originally called (K)constant Coded Programmable State Machine (KCPSM), it is a core used in less time critical applications and excels in implementing complex state machines that may be far more difficult to implement in custom logic. Other good uses include applications where many things are being integrated into the FPGA. Many of the devices that you would need to then interface are expecting to communicate with a processor. A Hatachi based LCD for example expects an initialization process heavily reliant on the data being in a specific order with particular timing. The PicoBlaze makes for a great co-processor. When a MicroBlaze application for example is getting complicated, sometimes the easiest thing to do is to offload some of the non-critical tasks to the PicoBlaze.

In an effort to decrease complexity and increase ease of use, Xilinx has recently produced a powerful new tool called System Generator that allows one to design, develop, and simulate entire applications using Simulink from Mathworks. System Generator is a Simulink add on that provides its own custom processing blocks that can be synthesized into VHDL and programmed into an FPGA. This gives the user a powerful new method for implementing MATLAB/Simulink style algorithms and control systems targeted directly for an FPGA.

One such block implements the PicoBlaze soft core. Figure 4 shows an example implementation used to interface to hardware buttons, switches, and LEDs. The PicoBlaze is well suited to this kind of task as user interfaces are usually not very time critical. This PicoBlaze can then be used by any other part of the application to interface to the user.

D) FPGA re-configuration

With so many different things being integrated onto a single FPGA, re-use of IP becomes more and more important. Dynamic re-configuration of an FPGA is a new area of study that involves dynamically changing a FPGAs configuration during operation. This has the possibility of improving area, timing and power characteristics. The ability to change, update, or even learn new functionality during the lifetime operation of an application provides for an awesome range of possibilities.

Currently available FPGAs do not have the technology to re-configure any one portion of its circuitry without interrupting the whole thing. Even the smallest change in the logic configuration requires a re-programming of the entire chip. Some applications can work this way. For example, an intelligent device can hold more than one FPGA configuration in non-volatile memory. During production, it can easily be set to configure the FPGA with a soft core charged with testing it's functionality and then once finished, it can re-configure the FPGA for final operation.

However, researchers are finding applications for Dynamic re-configuration of any one portion of the FPGA would be very useful. For example, A paper titled "Investigating Dynamic Reconfiguration of FPGA based IP Cores" has found a class of circuits that can take advantage of such a technology.

References:

1. TechXclusives articles, http://www.xilinx.com/xlnx/xweb/xil_tx_home.jsp "Performance + Time = Memory (Cost savig with 3-D design)" by Ken Chapman
2. Karen Parnell, Roger Bryner, "Comparing and Contrasting FPGA and Microprocessor System Design and Development" Xilinx WP213 (v1.1) July 21, 2004
3. Tyson S. Gall, "System-on-a-Programmable-Chip Development Platforms in the Classroom" IEEE transactions on Education, Vol. 47, No 4, Nov 2004
4. Ken Chapman, "PicoBlaze KCPSM3 8-bit Micro Controller for Spartan-3, Virtex-II and Virtex-IIPRO" Xilinx KCPSM3_Manual.pdf Rev. 7 October 2003 z
5. TechXclusives articles, http://www.xilinx.com/xlnx/xweb/xil_tx_home.jsp "Creating Embedded Microcontrollers (Programmable State Machines)" by Ken Chapman
6. "PowerPC Processor Reference Guide" www.xilinx.com EDK 6.1 September 2, 2003
7. Roman Lysecky, "A Study of the Speedups and Cometitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning"
8. Jeremy Kowalczyk, "Multiprocessor Systems" Xilinx WP162 (v1.1) April 10, 2003
9. Patrick Lysaght, John MacBeth, "Investigatin Dynamic Reconfiguration of FPGA Based IP Cores"
10. Tiecai Li,"ECOMIPS: An Economic MIPS CPU Design on FPGA" Proceedings of the 4th IEEE International Workshop on System-on-Chip for Real-Time Applications (IWSOC'04)
11. Jan Gray, "Building a RISC CPU and System-on-a-Chip in an FPG" Copyright © 1998-2000, Gray Research LLC.
12. Kevin Morris, "Prime-time Processing, Are Embedded Systems on FPGA Ready?" FPGA and Programmable Logic Journal February 8th, 2005
13. TechXclusives articles, http://www.xilinx.com/xlnx/xweb/xil_tx_home.jsp "The Root of All Evil" by Richard Griffin

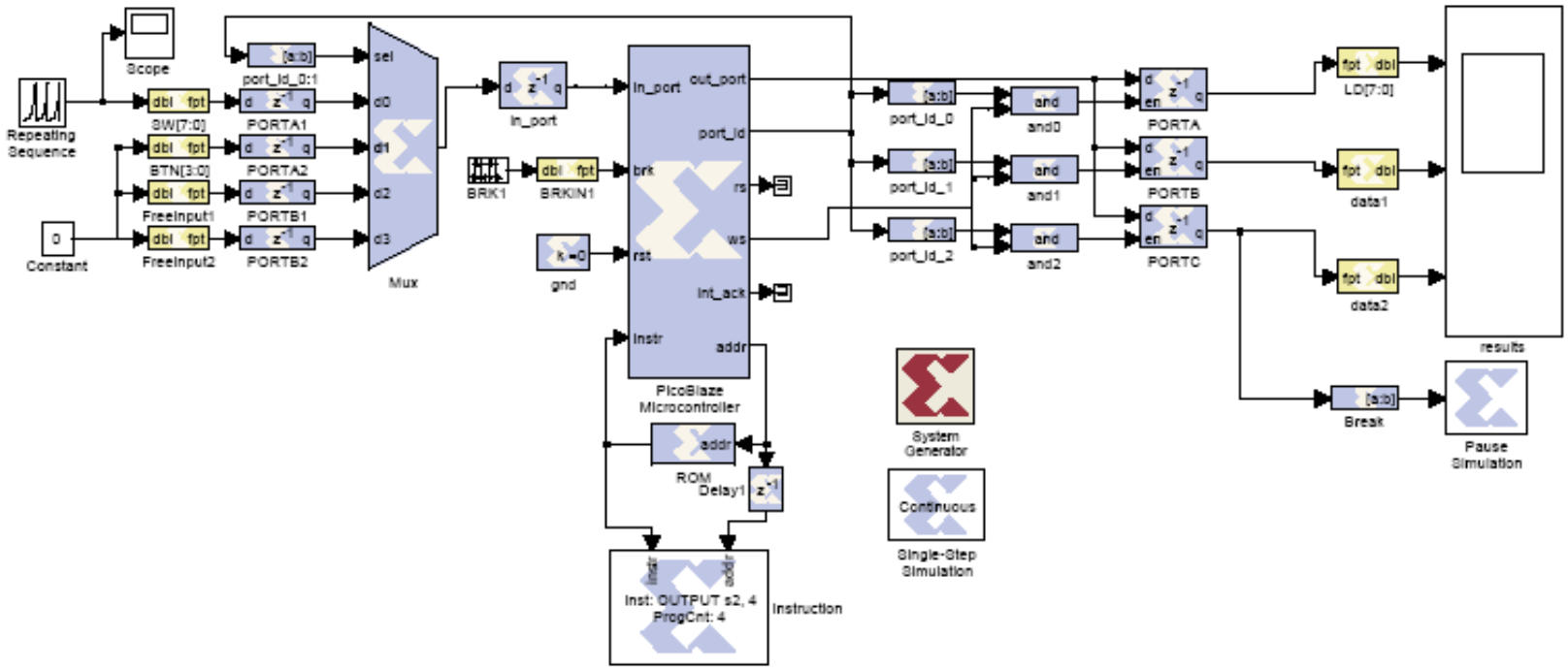


Figure 3 PicoBlaze application