Linux on FPGA Platforms - Control Software to Connect Custom Peripherals

Moritz Kretz
Institute of Computer Engineering of the University of Heidelberg
B6, 26
68131 Mannheim, Germany
kretz@stud.uni-heidelberg.de

Andreas Kugel
Institute of Computer Engineering of the University of Heidelberg
B6, 26
68131 Mannheim, Germany
andreas.kugel@ziti.uni-heidelberg.de

ABSTRACT

Accessing custom hardware peripherals from a soft-CPU realized on FPGA fabric is a common task. We use a Virtex-5 FPGA with a MicroBlaze soft-CPU running a standard Linux kernel as the core of our embedded system. In order to enable processes on the Linux system to communicate with custom peripherals on the FPGA a device driver is implemented to take advantage of the fast simplex link (FSL) bus and the resulting performance regarding throughput and latency is measured.

1. INTRODUCTION

In current large scale physics experiments like ATLAS [2] FPGAs are commonly used in the read-out chain of the detector, for example for buffering or data formatting ([6] [1, p. 24ff]). In the case of the ATLAS pixel detector, the FPGAs on the read-out driver cards (ROD, [3]) are themselves embedded in a landscape of other components like DSPs. A single board computer (SBC) running Linux controls the ROD cards that are placed in a VME crate.
Further up the data acquisition chain commodity computers are used for buffering and event building/filtering, also running a common Linux distribution[1].

It is desirable to have a unified software environment for most of the numerous hardware devices being used across the experiment, as this would simplify software development, allow for flexible deployment of functionality and reduce debugging efforts. As Linux is the de facto standard for scientific computing we investigate the use of a standard kernel on an FPGA to integrate custom hardware peripherals into a common software environment.
One possible way to communicate with hardware from the Linux environment (for example for setting or getting register values) is by mapping physical memory from /dev/mem to userspace addresses. Another possibility is specific to the MicroBlaze soft-CPU introduced in the next section: a bus interface can be used to transfer data between CPU and peripherals. To access this bus from userspace a device driver needs to be implemented.

[1]http://www.scientificlinux.org

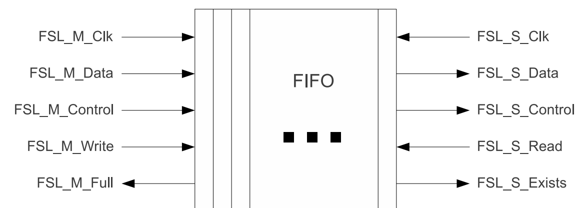


Figure 1: Fast Simplex Link peripheral [9]

2. THE MICROBLAZE SOFT-CPU

The MicroBlaze is a 32 bit RISC-architecture soft-CPU developed by Xilinx for use on their FPGA devices. It can be heavily customized to the needs of the target application by configuring its properties such as instruction and data cache sizes, use of a memory management unit, use of a floating point unit etc. To complete the usefulness of the final embedded system, IP cores like an interrupt controller, memory controller, ethernet core or custom designed peripherals can be added. These peripherals could include co-processor-like hardware, that accelerates specific time-consuming operations, or interfaces to other hardware components connected to the FPGA. They are then connected to the MicroBlaze with either the Processor Local Bus (PLB), more recently the Advanced eXtensible Interface (AXI), or - as in this work - the FSL. As of 2009 the Linux vanilla kernel supports the MicroBlaze architecture out of the box.

2.1 FSL Interface

Compared to other buses such as the On Chip Peripheral Bus (OPB) the fast simplex link (FSL) is a simple method to connect the MicroBlaze to hardware peripherals (or even to other MicroBlaze instances). It provides a uni-directional point-to-point link between the master (transmitter) and the slave (receiver). Therefore one needs two FSL instances for bidirectional communication - for example to send back the result of a calculation that was performed in hardware[2]. The

[2]*Streamlink* as it is being used in the MicroBlaze configuration wizard refers to two FSL instances, with the MicroBlaze being the master on one instance and the slave on the other. Later on we might refer to a stream link simply as an FSL.

FSL is implemented as a FIFO buffer that either works in asynchronous or synchronous mode, depending on its configuration. There are more configuration options for the IP core such as

- FIFO depth (ranging from 1 to 8192 words — 16 in this implementation)
- bus width (1 bit to 32 bits - if connected to a MicroBlaze, the bus width is always 32 bits as the MicroBlaze registers are 32 bits wide
- location of the FIFO implementation (LUT-RAM or BRAM, LUT-RAM in this implementation)

Figure 1 shows the block diagram of the FSL. The signals FSL_M_Write and FSL_M_Read are used to control writing to and reading from the FIFO, while FSL_M_Full and FSL_S_Exists are asserted to indicate when the FIFO is full or has valid data to be read by the slave. These signals can later be used to generate interrupts needed by the Linux device driver.

The relevant assembler instructions for reading from and writing to the FSL are `nput` and `nget` [8, p. 169f, p. 142f]. The blocking variants `put` and `get` will stall the CPU until data can be written to (i.e. the FIFO is not full) or read from (i.e. data is available) the corresponding FSL interface, while the non-blocking versions will set the carry bit of the status register to 0, if the read or write operation was successful. The *dynamic* instructions `nputd` and `ngetd` allow to determine the FSL interface at runtime by looking it up from a register. All instructions accessing an FSL interface need to be executed in privileged mode (kernel mode).

2.2 Embedded System Setup

We use the the Xilinx ML506 Virtex-5 development platform[3] for our test setup. A vanilla Linux Kernel (2.6.37) is employed. The embedded system consists of one MicroBlaze instance in the following configuration: 125 MHz clock, barrel shifter, 32 bit multiplier, 256 byte instruction and data cache with 8 word cache lines, virtual memory management and 2 memory protection zones, debug intereface, FSL stream link interfaces.

We use several peripherals, the noteworthy ones being

- interrupt controller for handling interrupts
- timer
- UART for serial communication
- DDR2-RAM controller to access 256 MB of memory
- ethernet MAC for networking support

Figure 2 gives an overview of the major peripherals and the busses used in the embedded system.

[3]http://www.xilinx.com/products/devkits/HW-V5-ML506-UNI-G.htm

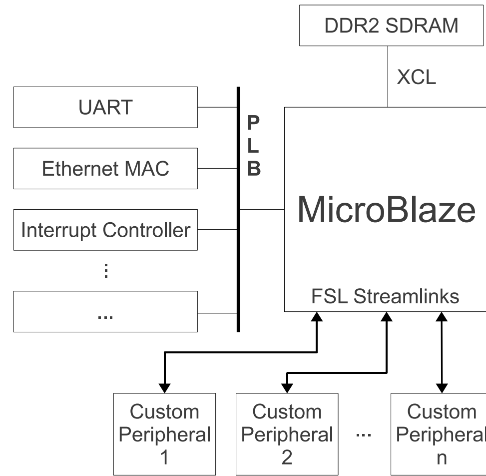


Figure 2: The embedded system with the peripherals centered around the MicroBlaze CPU.

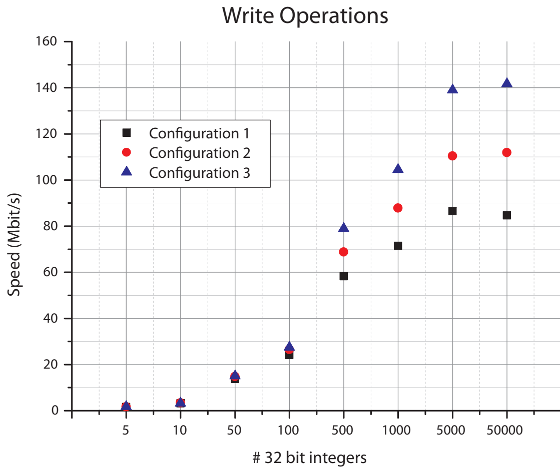
3. FSL DEVICE DRIVER

In order to communicate with the FSL from userspace, we have to supply a device driver that takes care of data transfers and provides a convenient interface for the userspace processes accessing the FSL.

For userspace processes the FSL device looks like a file (flagged by the filesystem as a device node), supporting open, close, read and write system calls, a subset of the file operations specified in the POSIX standard [5]. The file operations are implemented in the driver module, reading data from userspace and passing it to the FSL, while also reading from the FSL when there is incoming data and then writing it to userspace. As the kernel module cannot directly access userspace memory, the functions `copy_from_user` and `copy_to_user` are being used for that purpose (`get_user` and `put_user` being simpler, but faster alternatives).

The non-blocking variants immediately return to the userspace program once the FSL FIFO is full (write operation) or no more data can be read from the FSL - i.e. the FIFO is empty (read operation).
For the blocking read and write functionality we do not return to the calling application if an FSL operation fails, but wait until it can be performed. There are different concepts for realizing that kind of behavior, mainly polling (busy-waiting) or an interrupt based approach. While polling is a simple and effective method for achieving the desired behavior, it is also not very efficient compared to a solution with interrupts, due to massive CPU overhead and we therefore decided to implement the blocking variant with interrupts.

It is important to note that the non-blocking instructions `nputd` and `ngetd` are used throughout the whole implementation - even for the blocking read and write variants. The blocking instructions would stall the MicroBlaze completely until the FSL read/write can be performed and not give the Linux scheduler the chance to put the calling process to sleep and give CPU-time to another process.



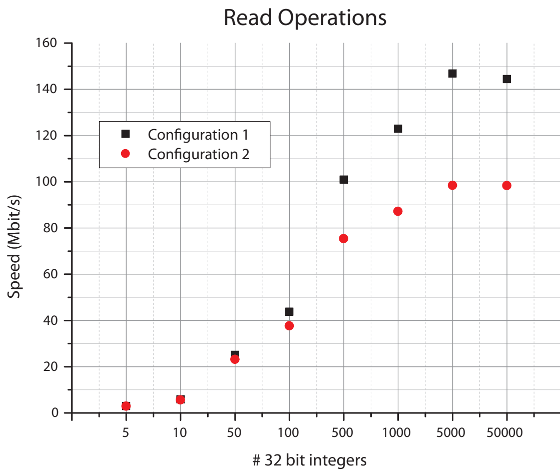


Figure 3: Throughput of FSL read and write operations for different implementations of the driver

3.1 Driver Performance

As one is interested to deliver a good performance for FSL access, we have to understand the different effects that have an impact on throughput and latency. The MicroBlaze runs at 125 MHz and is therefore theoretically able to perform 125 MIPS which is equal to a maximum throughput of 4 Gbit/s (125 MHz x 32 bit, assuming one FSL read/write operation each clock cycle). Nonetheless this is only a theoretical value to get an idea of the upper bound on throughput. It further is important to note that the CPU is part of the data-path and therefore presumed to be the limiting performance factor.

Throughput

A test program that generates different write and read patterns for FSL access is used to assess the throughput performance of the system. It takes two arguments S and N: first the amount of data S to be written or read (in terms of 32 bit integers) and second the number of iterations performed over the whole array N). The program opens the device file, reads (or writes) S data words N times, and finally closes the file again.

Figure 4: Write performance for two cache configurations of the MicroBlaze

Figure 3 shows the throughput for different values of S for several implementations of the non-blocking write/read functions (the values of N were chosen in the range of 1000 to 250000 resulting in a total runtime between 10 and 35 seconds per data point).

Write performance was assessed for three different implementations. Configuration 1 uses an internal kernel module buffer and the `copy_from_user` function. In configuration 2 the `get_user` function is used as opposed to the `copy_from_user` function in configuration 1. It only copies single words from userspace memory, but this is unproblematic as this value is directly written to the link. Configuration 3 employs the faster `__get_user` function, but we manually need to make sure the pointer to the memory location supplied by the write system call is valid by calling the kernel function `access_ok`[4] before accessing the memory.

We notice that for small amounts of data being written there is almost no performance difference between the approaches, suggesting that CPU time is mostly spent in the application. As the data size increases, configuration 3 cleary outperforms configuration 2 (which itself is faster than configuration 1). This is not unexpected for our write scenario: the internal driver module buffer used in configuration 1 adds additional instructions and memory accesses and does not help for our implementation.

There seem to exist upper bounds on throughput for larger data sizes, which are mainly caused by two limiting factors: CPU speed and memory access latency. CPU time at the upper end of the data size axis will mostly be spend in the write loop of the driver module. The write loop consists of 12 instructions yielding a maximum theoretical throughput of 333 Mbit/s and leaves little room for optimization.
Memory access on the other hand can easily be sped up by increasing the cache sizes of the MicroBlaze (resulting in more used resources on the FPGA) and thus reducing the number of cache misses.

[4] see [4, p. 142f] for an API reference

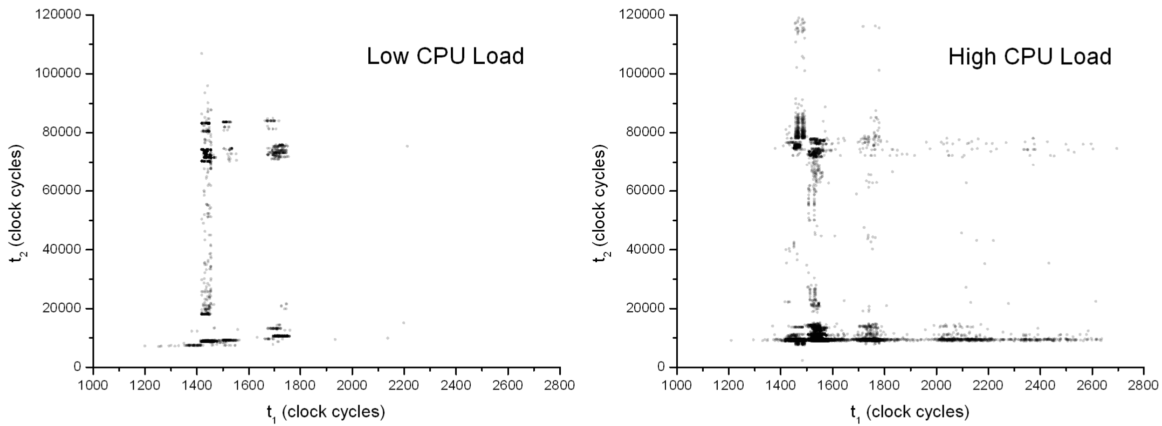


Figure 5: Distribution of interrupt latencies for idle system and system under load

System Load		Mean	σ	Min.	Median	Max.	99.9%
t1	Unloaded	1473	137	1200	1440	9415	1752
t1	Loaded	1605	612	1217	1558	20722	11754
t2	Unloaded	16425	20760	6951	8753	192346	84006
t2	Loaded	14658	26684	7159	9077	1627081	139813

Table 1: Results of the second series of latency measurements. The last column shows the latency thresholds, for which 99.9% of the measured latencies are shorter. Units in clock cycles.

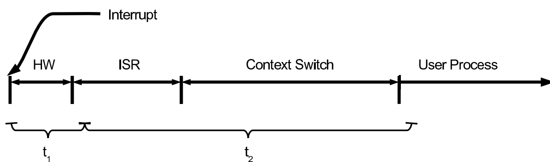


Figure 6: Latencies for a blocking FSL operation, according to [7]

For the read operations configuration 1 uses `put_user` and configuration 2 `__put_user` together with the `access_ok` function.

Interestingly enough, configuration 1 with `put_user` is faster than configuration 2, opposed to the earlier result for configuration 2 of the write performance measurement. For this we do not have a plausible explanation.
The other features of the figure are analogical to the ones of the write measurement discussed earlier.

In Figure 4 write performance for two different cache sizes is compared: the cache 1 configuration features 256 Byte instruction and data caches (equivalent to configuration 3 in Figure 3), while the cache 2 configuration has two 32 kByte caches.

First we notice the performance improvement of the larger cache configuration not only for small values of S, but also at the upper end of the scale. The performance improvement for smaller values of S is most likely due to the increased instruction cache size, while at larger data sizes the increased data cache size pays off. We can clearly see that for S=1000 and S=5000 the throughput is plateauing at around 200 Mbit/s: cache misses are practically non-existent and the penalty imposed by context level switches does not play a major role anymore.

We note that for the last data point S=50000 performance decreases again for the second configuration. The data does not completely fit in the cache any longer and therefore we have an increased number of cache misses. The speed difference compared to the smaller cache configuration is solely explainable by the different instruction cache sizes.

Latency

Figure 6 illustrates the chain of events for a blocking FSL operation. After the interrupt is generated by the FSL there is a delay caused by hardware and the initial interrupt handling of the kernel until the ISR of the driver module is finally called. We call the time interval t_1 from generation of the interrupt until execution of the custom ISR *interrupt latency*. Next, the ISR is executed and the process is woken up again (duration t_2) and the FSL operation executed - we call the overall latency $t_1 + t_2$ *preemption latency*.

The two time spans t_1 and t_2 were measured with the help of a custom VHDL module. In the ISR of the driver module an FSL operation on an unused streamlink was added as the first instruction in order to detect the call of the ISR. The VHDL module triggers a counter on interrupt generation, write operation to the unused FSL, and read operation on the FSL and then writes the measured latencies to BRAMs on the FPGA (256 kBit in total, corresponding to 16384 tuples of t_1 and t_2). On the Linux system one process writes data to the FSL and another one reads from the FSL (blocking). Our first test series was carried out on an idle system, while for the second data series load and interrupts were generated by copying multiple files via NFS/Ethernet resulting

in a load average of >1.0 of the system.

Table 1 presents the results of the measurement of t_1 and t_2. According to our sample the interrupt latency t_1 lies within a relatively small margin around the mean values of the latency distributions. There are few outliers on the upper end having almost no effect on the mean values for both measurements, though. Nonetheless, the interrupt latency for a system under load is noticeably higher than for an idle system.
The distributions for t_2 on the other hand are quite asymmetric and scattered. The median is a good figure to assess the expected range of values for t_2 (that make up the most part of the preemption latency). There are more outliers in the data sets (with values up to 1.6 million clock cycles, that means over 100 times slower than the median latency) and therefore system behavior becomes more and more unpredictable under load.

Figure 5 shows the two datasets for an idle system and for a system under load. While most data points are tightly arranged for the unloaded system, they are more scattered under load. Still one can make up several cluster regions even for the loaded system that depict recurring execution and memory access patterns.

4. CONCLUSIONS

We implemented a Linux device driver to access custom hardware peripherals from a standard Linux kernel running on a MicroBlaze soft-CPU. Measuring the performance of our implementation we noticed a strong dependency of the throughput on the cache sizes of the MicroBlaze. With a large enough cache data rates of up to 200 Mbit/s could be achieved.
Further the driver latencies of loaded and unloaded systems were investigated - a minimum preemption latency of 65 μs was observed for the unloaded system, while 99.9% of the measured times were faster than 686 μs (unloaded) and 1210 μs (loaded) respectively. The latency distribution was widely spread for both cases.

5. REFERENCES

[1] G. Aad et al. ATLAS pixel detector electronics and sensors. *J. Instrum.*, 3:P07007, 2008.

[2] G. Aad et al. The ATLAS Experiment at the CERN Large Hadron Collider. *J. Instrum.*, 3:S08003, 2008.

[3] G. Balbi et al. A PowerPC-based control system for the read-out-driver module of the ATLAS IBL. *J. Instrum.*, 7(02):C02016, 2012.

[4] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers*. O'Reilly, 2005.

[5] IEEE. *1996 (ISO/IEC) [IEEE/ANSI Std 1003.1, 1996 Edition] Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application: Program Interface (API) [C Language]*. IEEE, 1996.

[6] A. Kugel. *The ATLAS ROBIN - A High Performance Data-Acquisition Module*. PhD thesis, Universität Mannheim, 2009.

[7] P. Laurich. A comparison of hard real-time Linux alternatives, 2004.

[8] Xilinx. *MicroBlaze Processor Reference Guide*, 2008.

[9] Xilinx. *LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11c)*, April 2010.