DMA over PCIe with FPGA.

Sven Stubbe, Jan Marjanovič (both DESY)
2019-12-03

MTCA Pre-Workshop 2019

MicroTCA TECHNOLOGY LAB
HELMHOLTZ RESEARCH FOR GRAND CHALLENGES
DESY.

Abstract

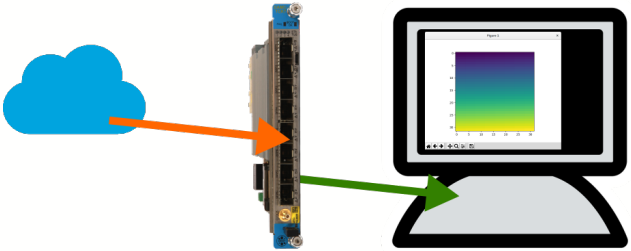
The objective of this tutorial session is to build a simple data acquisition system, in which the data is moved from an external device through a FPGA over PCIe into a CPU memory. There the data will be made available to user applications. We will use the DAMC-TCK7, its Board Support Package and a MicroTCA crate as a starting point. Then we will add a couple of Xilinx IP blocks and use Xilinx Linux driver to interface with FPGA modules. Our final goal is to have the data available for getting operated using a small Python application.

Speaker: Sven Stubbe, MicroTCA Technology Lab
Date: Tuesday, 3 December 2019 13:30-15:00
Venue: CFEL, SR I-III

Introduction

Motivation for this talk:

Student with a task to transfer data from a detector to his software/Matlab/...



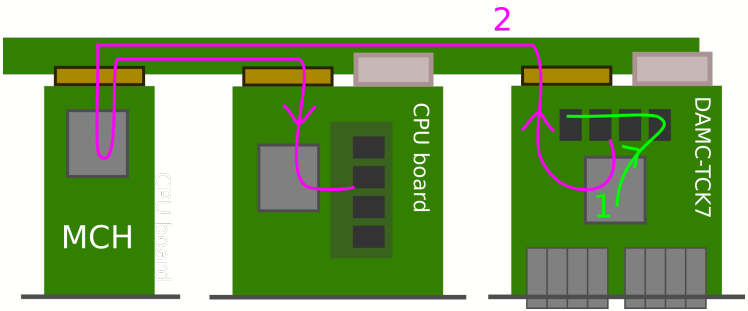
What is Direct Memory Access (DMA)?

From Wikipedia:

Direct memory access (DMA) is a feature of computer systems that allows certain hardware subsystems [this is us] *to access main system memory (random-access memory), independent of the central processing unit (CPU).*

What is Direct Memory Access (DMA)?

Real example: move the data from a PCIe board into CPU memory



Direct memory access (DMA) is a feature of computer systems that allows certain hardware subsystems to access main system memory (random-access memory), independent of the central processing unit (CPU).

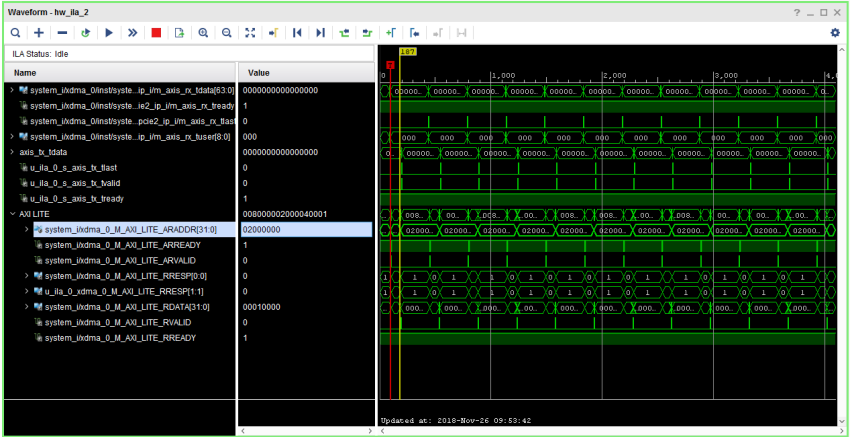
When is DMA needed?

When we need to transfer large amounts of data quickly.

Alternative to DMA transfers is some kind of memcpy():
processor reads the data into its registers, and writes the data out of registers. The CPU mangages the data transfers;
several CPU cycles for each data word are needed

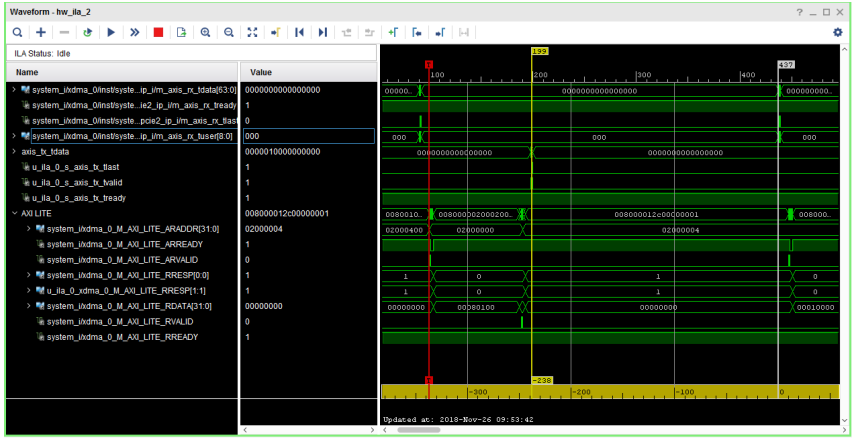
When is DMA needed?

memcpy() over the PCIe in Linux kernel space results in $\sim$ 2 MB/s transfer rate



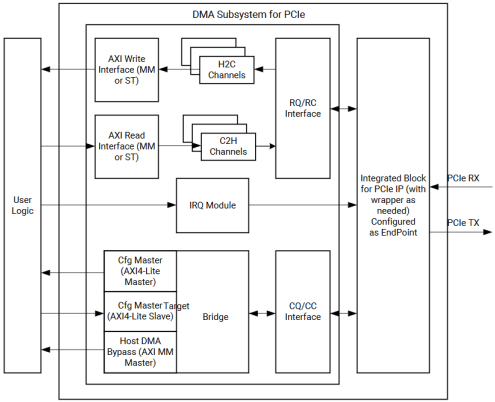
When is DMA needed?

A large delay between new MRd request (@437) and completion (@199) can be noted; this is not under our control



How does it look like?

The CPU let the endpoint manage the the data transfers on its own:

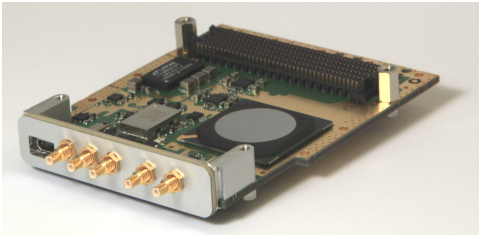


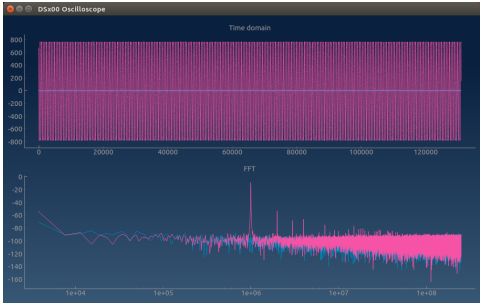
DMA/Bridge Subsystem for PCI Express® Overview (from pg195)

Examples

Example 1: DAQ for DFMC-DS800

- DFMC-DSx00 is an FMC mezzanine card with two-channel, 800 MSPS, 12-bit ADC
- Data rate is very high: 2400 MB/s
- DAQ application on Xilinx KCU105 eval kit





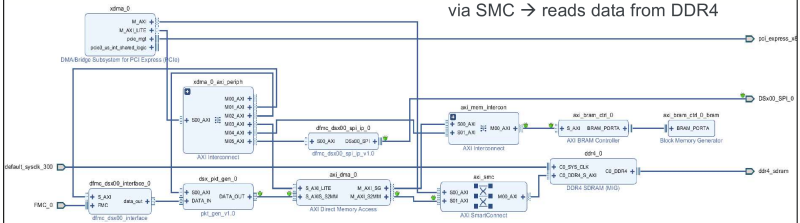
Example 1: DAQ for DFMC-DS800

DAQ-FPGA

- DFMC-DSX00-INTERFACE connected to LVDS data lanes → delay calibration
- DSX_PKT_GEN generates AXI stream pakets readable by AXI DMA core

(see also THOA01, IBIC 2018, J.Marjanovic for data post-processing)

- AXI_DMA dumps data into DDR4 via AXI SmartConnect
- XDMA PCIe core controls cores via AXI Lite
- XDMA has full AXI connection to DDR4 via SMC → reads data from DDR4



7th IBIC, Shanghai 2018

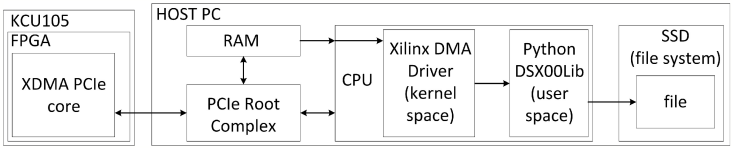
MICROTCA TECHNOLOGY LAB HELMHOLTZ RESEARCH FOR GRAND CHALLENGES

from http://ibic2018.vrws.de/talks/thoa02_talk.pdf

Example 1: DAQ for DFMC-DS800

DAQ-Software

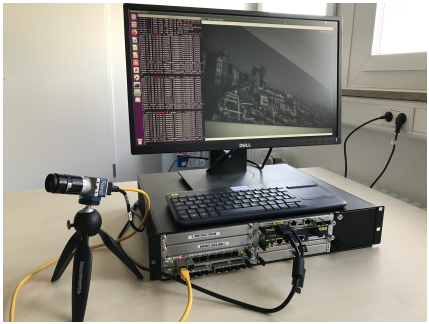
- straight forward solution for DAQ
- XDMA transfers sample data from DDR4 (KCU105) into host PC's RAM via PCIe root complex (DMA controller)
- Xilinx DMA driver (kernel space) provides pointer for DMA transfers into RAM
- Python library DSX00Lib can access transfered data in kernel space
- data can be dumped into file on SSD/HDD
- over 1 Mio. samples per channel can be stored on KCU105 (2GB DDR4)
- 8k samples can be stored on DAMC-FMC25 (256 MB DDR2)



from `http://ibic2018.vrws.de/talks/thoa02_talk.pdf`

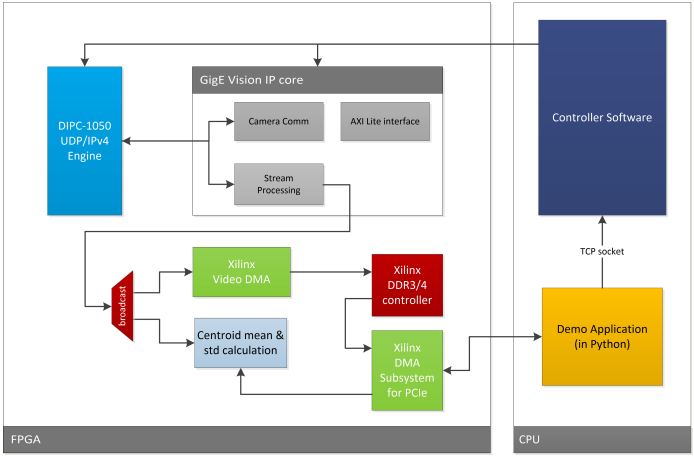
Example 2: GigE Vision camera interfacing

- DIPC-7050 is an IP stack for interfacing GigE Vision cameras with a FPGA
- Devices are running with up to (10)Gb Ethernet



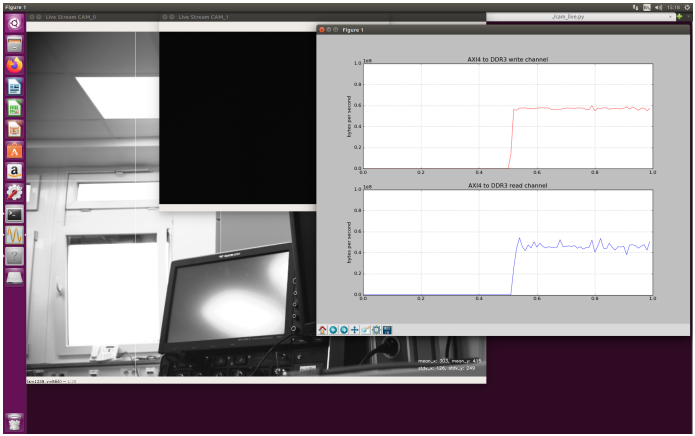
Talk on Thursday at 11:45:
”Certification and improvements of MicroTCA Technology Lab’s GigE Vision Stack”

Example 2: GigE Vision camera interfacing



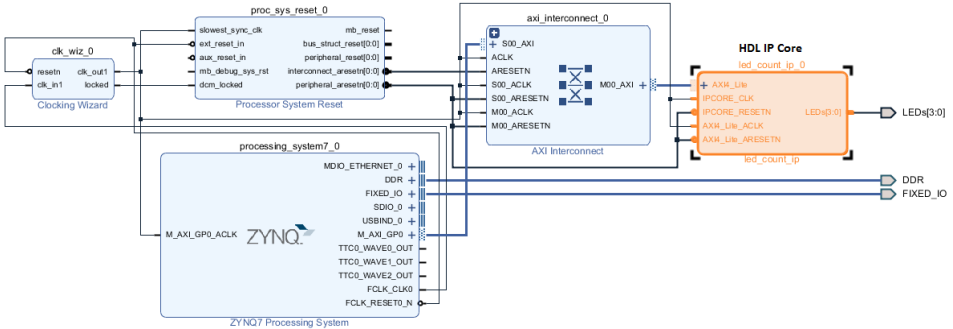
DIPC-7050 GigE Vision Stack base configuration

Example 2: GigE Vision camera interfacing



Running DIPC-7050 demo application with AXI performance monitor

Basics



from `https://www.mathworks.com/help/hdlcoder/examples/define-and-register-custom-board-and-reference-design-for-zynq-workflow.html`

Protocols: AXI4

- (Almost) all Xilinx IPs use AMBA AXI protocols for interfacing between each other
- Provide good interface for testing

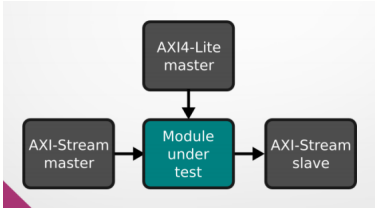


image from:

https://indico.desy.de/indico/event/15974/session/6/contribution/15/material/slides/0.pdf

another example (with Vivado):

https://github.com/j-marjanovic/chisel-stuff/tree/master/example-3-vivado-ip/vivado_tb_project

- Save documentation and communication efforts

Protocols: AXI4-Stream

Very common in Xilinx IPs for data streams
E.g.: FFT, 1/10/40/100 Gigabit Ethernet, high-perf PCIe

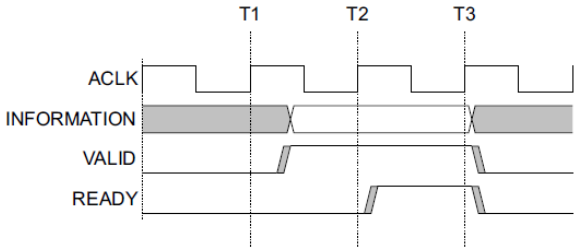


Figure A3-2 VALID before READY handshake

from AMBA® AXI™ and ACE™ Protocol Specification, available at

`https://www.arm.com/products/silicon-ip-system/embedded-system-design/amba-specifications`

Protocols: AXI4-Lite

Very common in Xilinx IPs for access to control/status registers

Composed of 5 independent channels:

- read address (M $\rightarrow$ S)
- read data (S $\rightarrow$ M)
- write address (M $\rightarrow$ S)
- write data (M $\rightarrow$ S)
- write response (S $\rightarrow$ M)

Three ways to implement AXI4-Lite interface:

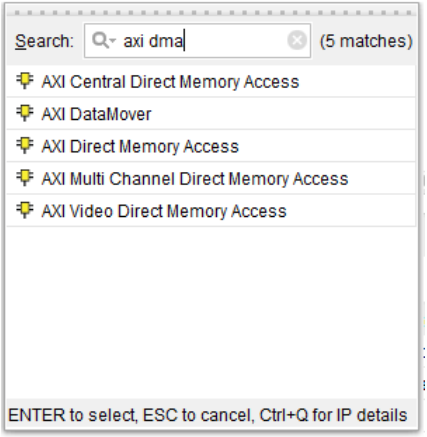
- Vivado Wizard (Tools/Create and Package New IP)
- write the interface yourself (2 state machines)
- Vivado HLS (in C and C++, bundle all inputs/outputs into one interface)

Protocols: AXI4

Similar to AXI4-Lite, but support burst transfers (typicaly 16, 32 data beats per burst).
Very common in Xilinx IP for high-throughput interfaces (DDR3/DDR4, PCIe DMA, DMA)

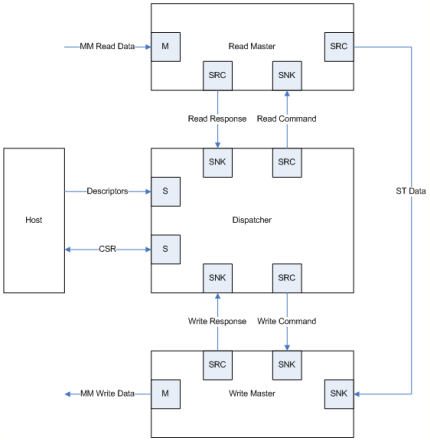
AXI DMA

Vivado provides a large variety of AXI DMA:



Other DMAs

- Altera mSGDMA
  `https://fpgawiki.intel.com/wiki/Modular_SGDMA`
  - IMHO the nicest SW interface
  - uses FIFOs for commands and responses
- home-made DMA
  - necessary in Virtex-5 times
  - weeks to implements
  - weeks to test all edge cases
- with Vivado HLS



Not really needed for this talk, but nice to understand what is going on behind the scenes.

Xilinx XDMA takes AXI transactions and creates PCIe TLP packets; these are passed to the Xilinx PCIe IP.

Figure 2-1: Non-Posted Read Transaction Protocol

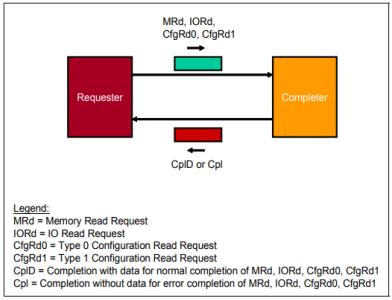
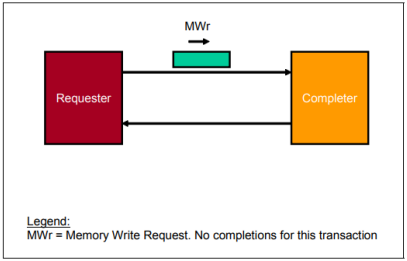


Figure 2-4: Posted Memory Write Transaction Protocol



from `https://www.mindshare.com/files/ebooks/PCI%20Express%20System%20Architecture.pdf`

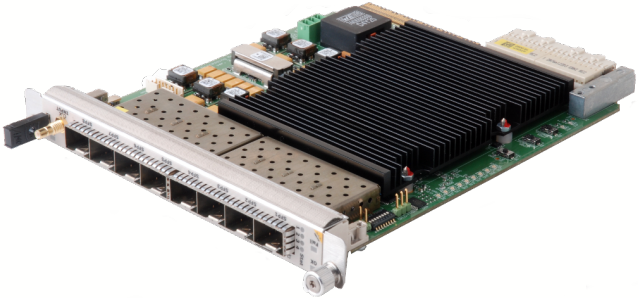
Demo

Task list

- ☐ Decide for the hardware and physically setup the crate
- ☐ Compile the firmware image and program the FPGA
- ☐ Check PCIe link state and load linux driver
- ☐ Configure the registers in FPGA firmware via PCIe access
- ☐ Run the user application and check the results

Hardware Setup

DAMC-TCK7 is commercially available at N.A.T.:
`https://www.nateurope.com/products/NAT-AMC-TCK7.html`



Hardware Setup

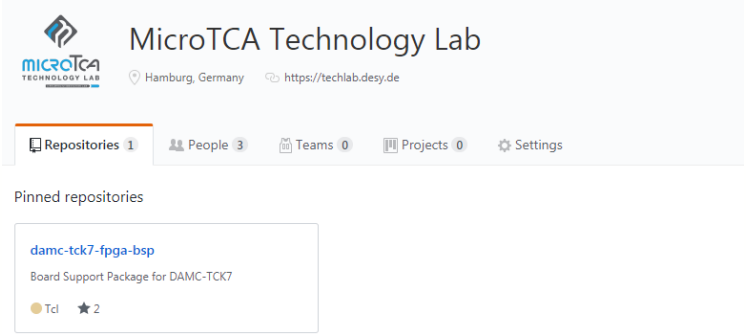
DAMC-TCK7 is set up together with a Concurrent AM G64 CPU here in the front:

Task list

- ☑ Decide for the hardware and physically setup the crate
- ☐ Compile the firmware image and program the FPGA
- ☐ Check PCIe link state and load linux driver
- ☐ Configure the registers in FPGA firmware via PCIe access
- ☐ Run the user application and check the results

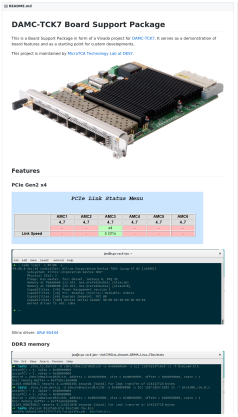
Start from the Board Support Package

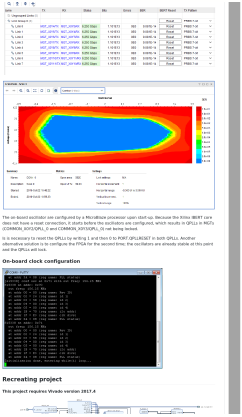
Available at `https://github.com/MicroTCA-Tech-Lab/damc-tck7-fpga-bsp`

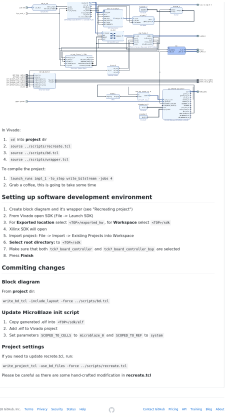


Start from the Board Support Package

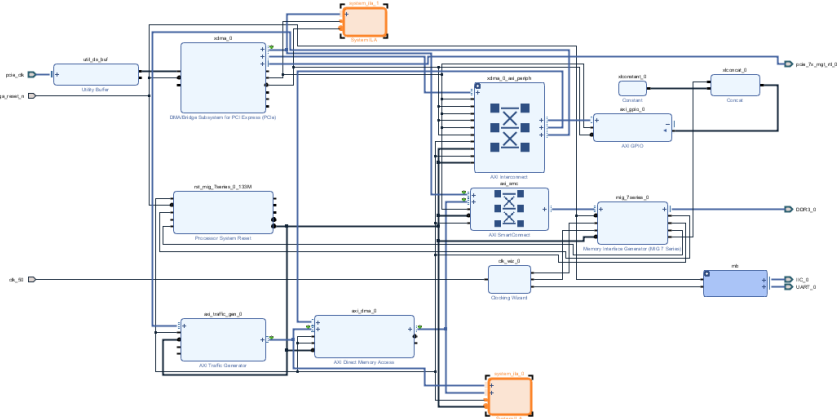
User can transfer the data from the CPU board to the DDR3 on DAMC-TCK7 board







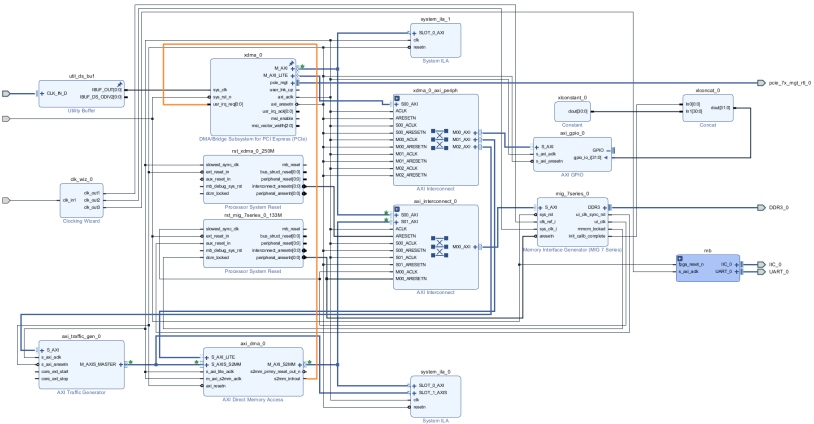
Vivado IP Integrator overview



Two Integrated Logic Analyzers (ILAs) on important AXI connections are highlighted

Vivado IP Integrator overview

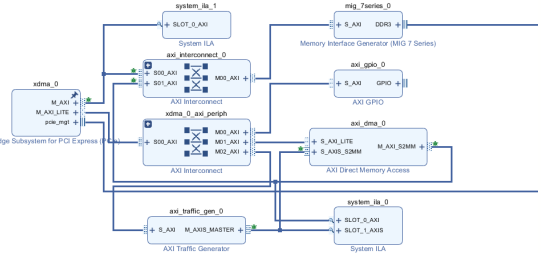
Adding interrupt requests that will be monitored by user application



Vivado IP Integrator overview

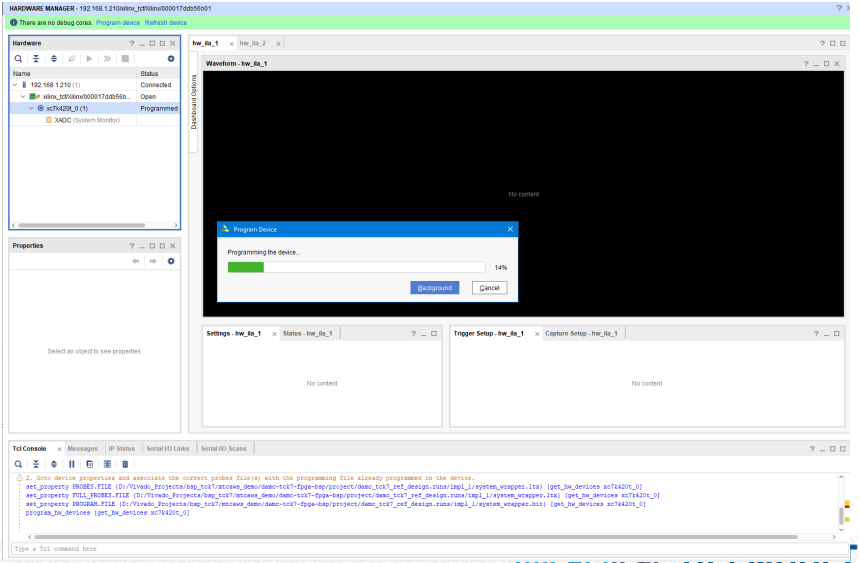
At this point our design includes:

- DMA/Bridge Subsystem for PCI Express (from BSP)
- Memory Interface Generator (MIG) (from BSP)
- AXI DMA
- AXI Traffic Generator



Block design reduced to AXI connections

Program FPGA

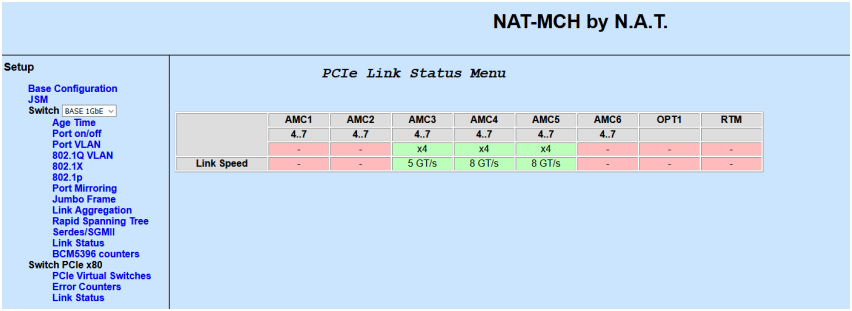


Task list

- ☑ Decide for the hardware and physically setup the crate
- ☑ Compile the firmware image and program the FPGA
- ☐ Check PCIe link state and load linux driver
- ☐ Configure the registers in FPGA firmware via PCIe access
- ☐ Run the user application and check the results

Check link state

PCIe link states can be observed by accessing the MCH (telnet or webinterface):



NAT-MCH by N.A.T.

Setup

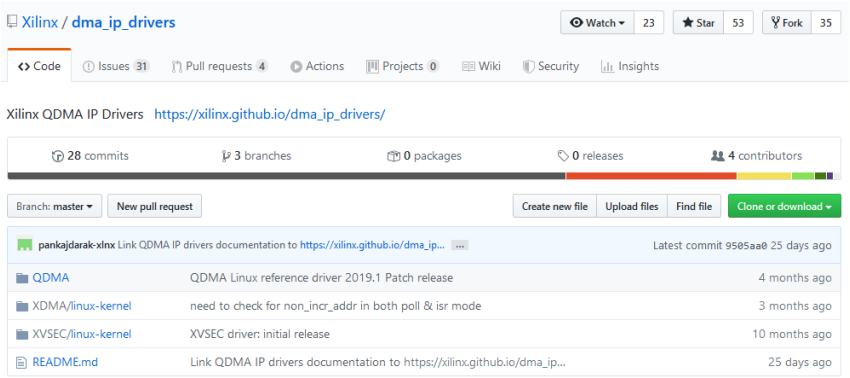
- Base Configuration
- JSM
- Switch BASE 1GbE
  - Age Time
  - Port on/off
  - Port VLAN
  - 802.1Q VLAN
  - 802.1X
  - 802.1p
  - Port Mirroring
  - Jumbo Frame
  - Link Aggregation
  - Rapid Spanning Tree
  - Serdes/SGMII
  - Link Status
  - BCM5396 counters
- Switch PCIe x80
  - PCIe Virtual Switches
  - Error Counters
  - Link Status

PCIe Link Status Menu

	AMC1	AMC2	AMC3	AMC4	AMC5	AMC6	OPT1	RTM
	4..7	4..7	4..7	4..7	4..7	4..7		
	-	-	x4	x4	x4	-	-	-
Link Speed	-	-	5 GT/s	8 GT/s	8 GT/s	-	-	-

Linux driver

Linux driver provided by Xilinx (AR# 65444) needs to be downloaded and compiled.
Sources are vailable at `https://github.com/Xilinx/dma_ip_drivers`.



Linux driver

Get PCIe bus and device informations via `lspci`:

```
> sudo lspci -s 06:00 -v
06:00.0 Serial controller: Xilinx Corporation Device 7024 (prog-if 01 [16450])
        Subsystem: Xilinx Corporation Device 0007
        Physical Slot: 3
        Flags: bus master, fast devsel, latency 0, IRQ 212
        Memory at 9ac00000 (32-bit, non-prefetchable) [size=1M]
        Memory at 9ad00000 (32-bit, non-prefetchable) [size=64K]
        Capabilities: [40] Power Management version 3
        Capabilities: [48] MSI: Enable+ Count=1/1 Maskable- 64bit+
        Capabilities: [60] Express Endpoint, MSI 00
        Capabilities: [100] Device Serial Number 00-00-00-00-00-00-00-00
        Kernel driver in use: xdma
```

Interface to user space

On the CPU side , Xilinx DMA driver provides several char devices:

```
$ ll /dev | grep xdma
drwxr-xr-x  3 root root          60 Jun 20 17:19 xdma
crw-rw-rw-  1 root root     238,  36 Jun 20 17:19 xdma0_c2h_0
crw-rw-rw-  1 root root     238,   1 Jun 20 17:19 xdma0_control
crw-rw-rw-  1 root root     238,  10 Jun 20 17:19 xdma0_events_0
crw-rw-rw-  1 root root     238,  11 Jun 20 17:19 xdma0_events_1
[...]
crw-rw-rw-  1 root root     238,  24 Jun 20 17:19 xdma0_events_14
crw-rw-rw-  1 root root     238,  25 Jun 20 17:19 xdma0_events_15
crw-rw-rw-  1 root root     238,  32 Jun 20 17:19 xdma0_h2c_0
crw-rw-rw-  1 root root     238,   0 Jun 20 17:19 xdma0_user
crw-rw-rw-  1 root root     238,   2 Jun 20 17:19 xdma0_xvc
```

Interface to user space

On the CPU side , Xilinx DMA driver provides several char devices:

```
$ ll /dev | grep xdma
drwxr-xr-x  3 root root           60 Jun 20 17:19 xdma
crw-rw-rw-  1 root root     238,  36 Jun 20 17:19 xdma0_c2h_0
crw-rw-rw-  1 root root     238,   1 Jun 20 17:19 xdma0_control
crw-rw-rw-  1 root root     238,  10 Jun 20 17:19 xdma0_events_0
crw-rw-rw-  1 root root     238,  11 Jun 20 17:19 xdma0_events_1
[...]
crw-rw-rw-  1 root root     238,  24 Jun 20 17:19 xdma0_events_14
crw-rw-rw-  1 root root     238,  25 Jun 20 17:19 xdma0_events_15
crw-rw-rw-  1 root root     238,  32 Jun 20 17:19 xdma0_h2c_0
crw-rw-rw-  1 root root     238,   0 Jun 20 17:19 xdma0_user
crw-rw-rw-  1 root root     238,   2 Jun 20 17:19 xdma0_xvc
```

c2h
AXI4 port (DMA from device)

h2c
AXI4 port (DMA to device)

user
AXI4-Lite port

events_N
interrupts

Task list

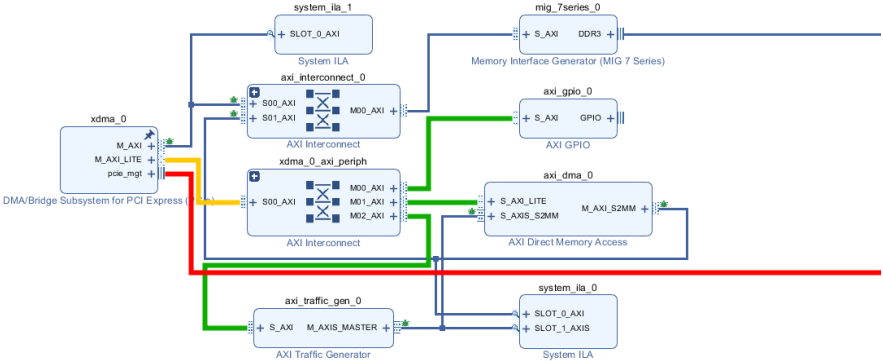
- ☑ Decide for the hardware and physically setup the crate
- ☑ Compile the firmware image and program the FPGA
- ☑ Check PCIe link state and load linux driver
- ☐ Configure the registers in FPGA firmware via PCIe access
- ☐ Run the user application and check the results

Interface to user application

To access the status and control registers (on AXI4-Lite interface), Python variant

```python
class HwAccessPcie(object):

  def __init__(self):
    user_filename = "/dev/xdma/card0/user"
    self.fd_user = os.open(user_filename, os.O_RDWR)
    self.mem = mmap.mmap(self.fd_user, 4 * 1024 * 1024)

  def __close__(self):
    self.mem.close()
    os.close(self.fd_user)

  def _rd32(self, addr):
    bs = self.mem[addr:addr + 4]
    return struct.unpack("I", bs)[0]

    def _wr32(self, addr, data):
    bs = struct.pack("I", data)
    self.mem[addr:addr + 4] = bs
```

Configurable IP blocks



Block design reduced to AXI connections

Register map

- BAR0
  - 0x0000 AXI GPIO
    - 0x0000 inputs
      - bit[0] = DDR3 ready
  - 0x1000 AXI DMA
    - 0x1030 S2MM_DMACR
      - bit[0] = Run/Stop
      - bit[1] = RSVD (const 1)
    - 0x1034 S2MM_DMASR
      - bit[0] = Halted
      - bit[1] = Idle
      - bit[11:4] = various errors
    - 0x1048 S2MM_DA
    - 0x104C S2MM_DA_MSB
    - 0x1058 S2MM_LENGTH
  - 0x10000 AXI TPM

Manual control

Use tool provided by Xilinx (`reg_rw`) to ”peek” and ”poke” registers.

1. Check DDR3 status

   ```
   ./reg_rw /dev/xdma/card0/user 0x0
   ```

   Expected value: 0x1
2. Check if DMA is in idle

   ```
   ./reg_rw /dev/xdma/card0/user 0x1034
   ```

   Expected value: 0x1

Manual control

Get the data from AXI Traffic Generator into on-board DDR3 memory

1. Enable DMA

```
./reg_rw /dev/xdma/card0/user 0x1030 w 0x1
```

2. Check if DMA is not idle anymore

```
./reg_rw /dev/xdma/card0/user 0x1034
```

3. Set DMA destination address

```
./reg_rw /dev/xdma/card0/user 0x1048 w 0x80000000
```

4. Start transfer

```
./reg_rw /dev/xdma/card0/user 0x1058 w 0x2000
```

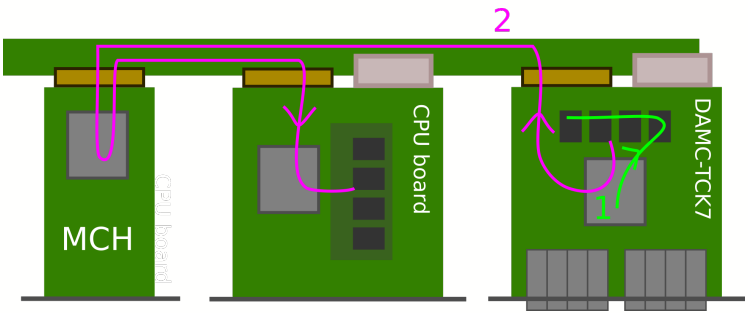
Task list

- ☑ Decide for the hardware and physically setup the crate
- ☑ Compile the firmware image and program the FPGA
- ☑ Check PCIe link state and load linux driver
- ☑ Configure the registers in FPGA firmware via PCIe access
- ☐ Run the user application and check the results

Application

1. AXI DMA will always be ready to accept data.
2. When an entire packet is received, AXI DMA raises interrupt request.
3. The user-space application monitors the interrupt request. When the AXI DMA has finished storing data into on-board DDR3, the user-space application is alerted.
4. The user-space application then copies the data from on-board DDR3 to its internal buffer.
5. The content of the buffer is compared to the SW implementation of LFSR.
6. A software script is used to re-trigger AXI Traffic Generator.

Quick reminder



Content of `App.cpp`

```
/// implements LFSR as described in "AXI Traffic Generator v3.0"
class AxiTrafficGenLfsr;

/// interface to Xilinx DMA driver
class XdmaIsr;

/// general interface for register access (using wr32 and rd32 methods)
class XdmaRegAccess;

/// AXI DMA controller
class HwAxiDmaControl;

/// data consumer
class DataConsumer;

/// application capturing data and checking data integrity
int main();
```

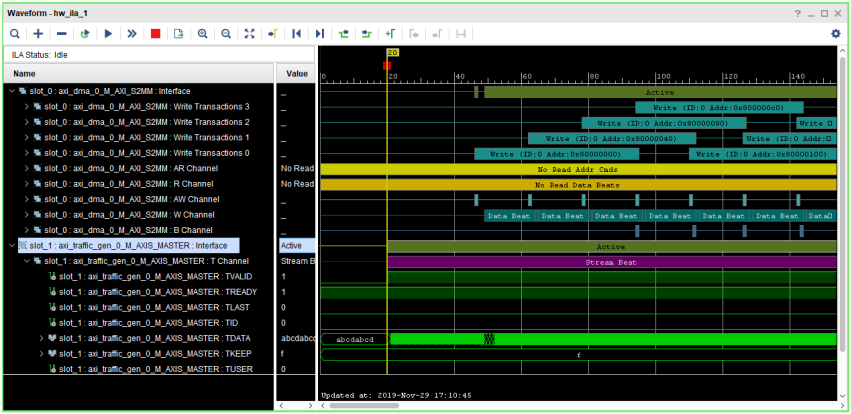
AXI Traffic Generator

We use the following script to trigger the AXI Traffic Generator.

```
#! /bin/bash
# script to configure and trigger AXI Traffic generator

AXI_TG_BASE=0x10000

AXI_TG_ADDR_ST_CTRL=0x30
AXI_TG_ADDR_ST_CONF=0x34
AXI_TG_ADDR_TR_LEN=0x38


# set delay between packets
%reg_rw /dev/xdma/card0/user $(($AXI_TG_BASE + $AXI_TG_ADDR_ST_CONF)) w 0x10000

# set packet len, number of packets
reg_rw /dev/xdma/card0/user $(($AXI_TG_BASE + $AXI_TG_ADDR_TR_LEN)) w 0x101ff

# trigger the traffic generation
reg_rw /dev/xdma/card0/user $(($AXI_TG_BASE + $AXI_TG_ADDR_ST_CTRL)) w 1
```

DMA transfer

If we trigger ILA on AXI4-Stream VALID and READY being both ’1’ we are able to capture the DMA transfer into on-board DDR3 memory.

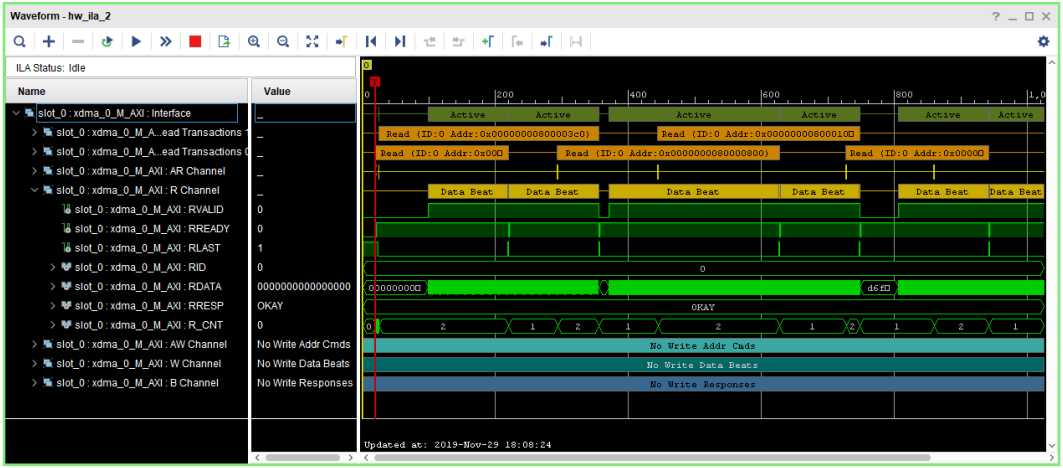


DMA from device

1. Data is now in on-board (DAMC-TCK7 board) DDR3 memory.
2. We need to use the second DMA to bring it into CPU.

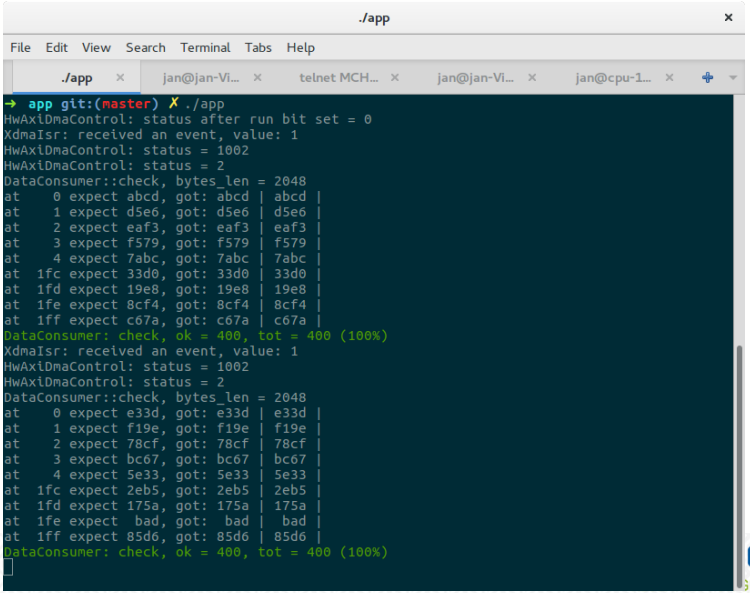
DMA from device

Trigger on ARREADY and ARVALID to xdma/M_AXI:



(Note the incrementing address)

Output from application



Conclusion

Conclusion

- Xilinx provides a lot of infrastructure to help start the development
- MTCA Tech Lab is commited to provide open-source Board Support Packages for new boards (DAMC-TCK7, Zynq MPSoC FMC+ carrier, ...)
- Linux drivers are also provided by Xilinx
  (`https://www.xilinx.com/support/answers/65444.html`)
  Servers are a huge market and require better support for drivers/interfaces. Xilinx DMA provides everything out of the box.

TRANSFER MTCA TO RESEARCH AND INDUSTRY

- Custom developments
- High-end test & measurement services
- System configuration & integration
- LLRF design

MicroTCA training coursers:

Next dates (see `https://techlab.desy.de/services/training/` for more):

- Basic: January 15-16, April 7-8, ...
- Advanced: February 26-27, June 3-4, ...

Thank you

https://techlab.desy.de

Deutsches Elektronen-Synchrotron DESY
A Research Centre of the Helmholtz Association
Notkestr. 85, 22607 Hamburg, Germany

Backup slides

System check

Before we start with example, let's check if everything is OK with the system:

- ☑ FPGA programmed
- ☑ PCIe link is up (**show_link_state**, **show_ekey**, front-panel LED)
- ☑ PCIe device enumerated (**lspci**)
- ☑ Driver loaded (**lspci -v**)

memcpy meas: signals to watch

```
system_i/xdma_0/inst/system_xdma_0_0_pcie2_to_pcie3_wrapper_i/
  pcie2_ip_i/s_axis_tx_tdata[*]
  pcie2_ip_i/s_axis_tx_tuser[*]
  pcie2_ip_i/s_axis_tx_tkeep[*]
  pcie2_ip_i/s_axis_tx_tvalid
  pcie2_ip_i/s_axis_tx_tready
  pcie2_ip_i/s_axis_tx_tlast
  pcie2_ip_i/m_axis_rx_tdata[*]
  pcie2_ip_i/m_axis_rx_tuser[*]
  pcie2_ip_i/m_axis_rx_tlast
  pcie2_ip_i/m_axis_rx_tready
  pcie2_ip_i/m_axis_rx_tvalid
```

memcpy meas: memcpy implementation

Inspired by `drivers/net/wireless/broadcom/brcm80211/brcmfmac/pcie.c#L486`

```
static void xdma_memcpy_from_dev (
    struct xdma_dev *lro,
    u32 offs,
    void *dstaddr,
    u32 len
) {
    void __iomem *address = lro->bar[lro->user_bar_idx] + offs;
    u32* dst32;

    len = len / 4;
    dst32 = (u32*)dstaddr;

    while (len) {
        *dst32 = ioread32(address);
        address += 4;
        dst32++;
        len--;
    }
}
```

memcpy meas: time measurement

see also https://www.kernel.org/doc/html/v4.15/driver-api/device-io.html

```
long char_ctrl_ioctl(struct file *filp, unsigned int cmd, unsigned long arg) {
    struct xdma_dev *lro;
    struct xdma_char *lro_char = (struct xdma_char *)filp->private_data;
    u64 t0_ns, t1_ns, duration_ns;

    BUG_ON(!lro_char);
    BUG_ON(lro_char->magic != MAGIC_CHAR);
    lro = lro_char->lro;
    BUG_ON(!lro);
    BUG_ON(lro->magic != MAGIC_DEVICE);

    printk(KERN_DEBUG "JAN: memcpy started, size: %lu\n", sizeof(tmp));
    t0_ns = ktime_get_ns();
    xdma_memcpy_from_dev(lro, USR_DDR3_OFFS, (void*)tmp, sizeof(tmp));
    t1_ns = ktime_get_ns();
    printk(KERN_DEBUG "JAN: memcpy done\n");
    duration_ns = t1_ns - t0_ns;
    printk(KERN_DEBUG "JAN: duration %llu ns (t1 = %llu, t0 = %llu)\n",
                      duration_ns, t1_ns, t0_ns);
    return 0;
}
```

memcpy meas: CPU 1

On Concurrent Tech AM900x
kernel 4.4.0-139-generic
Intel(R) Core(TM) i7-3612QE CPU @ 2.10GHz

```
[55775.132977] JAN: memcpy started, size: 1048576
[55775.498411] JAN: memcpy done
[55775.498414] JAN: duration 560049231 ns (t1 = 111312945419962, t0 = 111312385370731)

[55793.478833] JAN: memcpy started, size: 1048576
[55793.844333] JAN: memcpy done
[55793.844335] JAN: duration 468041144 ns (t1 = 111352256875650, t0 = 111351788834506)

[55819.083870] JAN: memcpy started, size: 1048576
[55819.449562] JAN: memcpy done
[55819.449564] JAN: duration 520045715 ns (t1 = 111401605213633, t0 = 111401085167918)
```

memcpy meas: CPU 2

On ADLINK AMC-1000
kernel 4.15.0-39-generic
Intel(R) Core(TM)2 Duo CPU L7400 @ 1.50GHz

```
[  357.333248] JAN: memcpy started, size: 1048576
[  357.792152] JAN: memcpy done
[  357.792157] JAN: duration 458898891 ns (t1 = 357792150659, t0 = 357333251768)

[  360.061151] JAN: memcpy started, size: 1048576
[  360.520068] JAN: memcpy done
[  360.520073] JAN: duration 458910762 ns (t1 = 360520066218, t0 = 360061155456)

[  362.647225] JAN: memcpy started, size: 1048576
[  363.106262] JAN: memcpy done
[  363.106268] JAN: duration 459030051 ns (t1 = 363106259232, t0 = 362647229181)
```

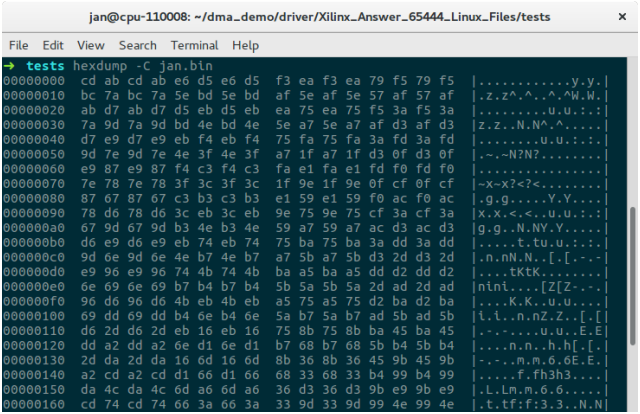
memcpy meas: CPU 3

On external CPU
kernel 4.15.0-39-generic
Intel(R) Core(TM) i5-2500 CPU @ 3.30GHz

```
[  225.678066] JAN: memcpy started, size: 1048576
[  226.128774] JAN: memcpy done
[  226.128775] JAN: duration 450702965 ns (t1 = 225831915216, t0 = 225381212251)

[  229.493646] JAN: memcpy started, size: 1048576
[  229.944742] JAN: memcpy done
[  229.944743] JAN: duration 451090885 ns (t1 = 229647854461, t0 = 229196763576)

[  232.285649] JAN: memcpy started, size: 1048576
[  232.736626] JAN: memcpy done
[  232.736627] JAN: duration 450971647 ns (t1 = 232439716846, t0 = 231988745199)
```

Hexdump

We inspect the data transfered with `dma_from_device` and note the pattern which was generated by AXI Traffic Generator (2x16-bit values).



LFSR Implementation used for Random Generation

The below linear feed-back shift register (LFSR) is used for random address or data generation:

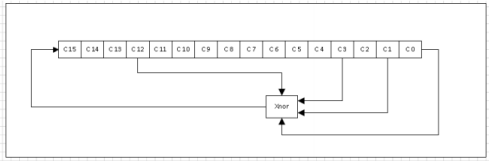


Figure 2-5: **LFSR**

Initially, the C0 to C15 flops are loaded with the input seed value. Later, it behaves as a shift register as per the architecture.

from Xilinx AXI Traffic Generator v3.0

`https://www.xilinx.com/support/documentation/ip_documentation/axi_traffic_gen/v3_0/pg125-axi-traffic-gen.pdf`