Development of a pattern generator and logic analyzer based on FPGA

Juan Compadre Ochando

Tutor: Rafael Gadea Gironés

Cotutor: David Martínez Lerín, José María Monzó Ferrer

Trabajo Fin de Máster presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Máster en Ingeniería Telecomunicación

Curso 2018-19

Valencia, 4 de septiembre de 2018

Abstract

In the evaluation of different chips it is necessary to generate patterns in their input signals and check that the output signals are correct. These patterns must cover all the possibilities of the chip to ensure its correct performance in any situation and working mode. The objective of this TFM is to implement a pattern generator using an FPGA that is also capable of capturing the outputs of the chip. The use of the FPGA allows total control over which signals are sent and received through their pins, where the edges are located and, in addition, allows to work at a higher frequency than several current solutions.

In this project, the patterns will be generated at bit level using Python scripts on a PC and will be uploaded to the FPGA through a USB port. The task of the FPGA will be to send these patterns and receive signals from the chip to be evaluated.

INDEX

1. Introduction and motivation

The objective of this master's thesis is the development of a platform for the evaluation of the digital part of chips. In the evaluation of any chip, it is necessary to use an interface. We do that by providing different inputs and then capturing and analysing the outputs. The variety of inputs to be tested depend on the characteristics of the chip and its working mode(s). Therefore, defining the correct patterns is the best starting point for a good IC evaluation.

This project, named Falcon, is created due to the need of a flexible and configurable tool that provides the capability of defining a digital pattern at bit level, selecting exactly where the edges will be located in a simple manner. It is also necessary to generate and capture these patterns at a higher frequency than current solutions.

Falcon is a digital pattern generator and logic analyzer that provides a very flexible and easy solution for IC evaluation. It is based on a FPGA platform and its objective is to provide a highly configurable and customizable solution, suitable for a high diversity of applications (ADC testing, scan patterns debug ...) and capable of working at high frequencies.

Its high level PC driver is designed to be simple and intuitive, it is based on Python but it can be called from any programming language.

2. Falcon project

2.1. Development platform

Falcon is implemented in a FPGA evaluation kit board, due to the high flexibility for design that these devices offer. The design of Falcon needs to be highly configurable and offer a very low latency. Furthermore, it is necessary to control all the signals at bit level and control all the delays between any input and any output.

FPGAs offer this low latency and this detailed control over the hardware, which makes possible the accurate control over signals, delays, clock frequencies, etc. The designer uses hardware description languages to program the hardware of the FPGA. In addition, using the FPGA pins, it is possible to develop any interface, from configurable GPIO ports (which is the case of Falcon), to more complex interfaces using protocols like SPI, I2C or RS-232.

But the biggest benefit of using an FPGA is the possibility of reprogramming the implemented design. During the development phase, it is very simple to try different implementations of different designs and analyse the results with laboratory equipment.

FPGAs usually include more resources than programmable logic. Traditionally, they include DSPs, RAM memory, DMAs, USB controller, etc. Some FPGAs also include an embedded ARM microprocessor, which facilitates the use of some resources, like the USB connection or the DMAs. Including this processor means that the designer will need to develop some firmware using a high level programming language, like C.

The FPGA used for this project is a Zynq 7000, from Xilinx, which includes an ARM Cortex-A9 Based Application Processor Unit [1]. This FPGA is included in the evaluation kit MicroZed, which provides connection to the I/O pins of the FPGA, a USB port and 1 GB of DDR3 SDRAM memory, 128 Mb of QSPI Flash and a Micro SD card interface.

2.2. General architecture and functionality

Following the objectives mentioned in the previous section, Falcon needs to be very flexible and capable of working at high frequency. It should be possible to control, at a single bit level, the pattern sent to the DUT. Furthermore, the frequency for sending and capturing signals should be configurable.

For these two reasons, the functionality of the FPGA part is to store and send/receive the data for the DUT evaluation. The FPGA is completely dedicated to control the data transfers at the required conditions. The data is not created nor processed inside the FPGA, this is carried out by several Python drivers and scripts.

In a host PC, a Python environment runs different scripts that generate the exact data, at a bit level, that the user desires to send to the DUT. The result of these scripts is a vector of digital values (with a maximum of 32 bits), each of them representing the state of every GPIO pin in a specific clock cycle. The entire vector represents the data that will be sent to the DUT during as many clock cycles as components of the vector.

The Python environment also executes the driver to communicate with Falcon. This communication consists in an USB connection which delivers commands from the PC to Falcon and receives the different responses. In addition, the USB interface is used to send the generated tx data to Falcon, in order to be stored; and to transmit the received data in Falcon, from the DUT, to the PC.

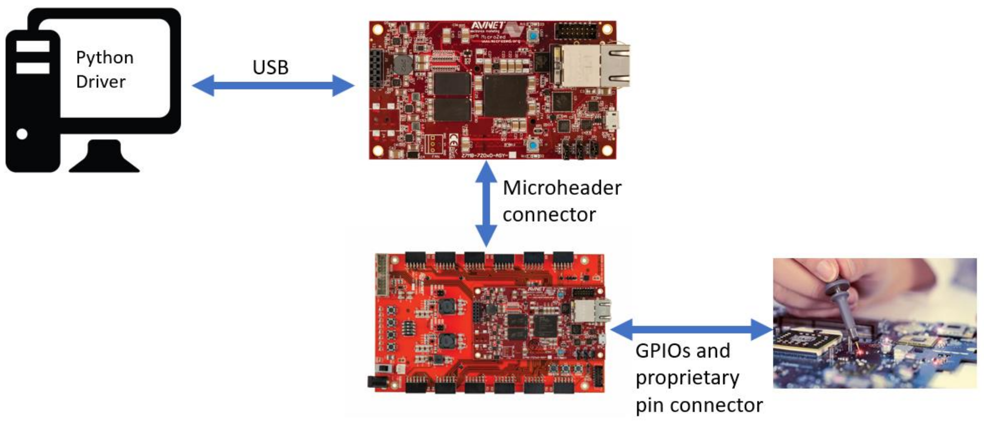


Figure 1 Falcon connection and interfaces

The SOM (System on Module, the MicroZed evaluation kit) selected for the project contains DDR memory (among many other resources), which is the memory used for storing the generated data. Once stored, when the Python driver has configured the pins, the working frequency, and all the custom IPs, if the appropriate command is sent, the tx data is sent through the pins. Some additional functionalities are: waiting for a trigger to start the operation, repeat the data sent as a pattern and rearm the trigger automatically. At the same time, if commanded by the user, it is possible to capture data using the GPIOs configured as input. This data will be stored in the DDR memory, in a region separated from the tx data.

The FPGA Zynq 7010 contains a resource named DMA [1], which is in charge of all data transfers in the system. The DMA allows the processor to continue executing its operations while the data transfers are being performed. The processor only needs to configure the DMA and it will rise an interruption when the transfer is complete.

In the next figure we can see a simplified diagram of the architecture of Falcon.

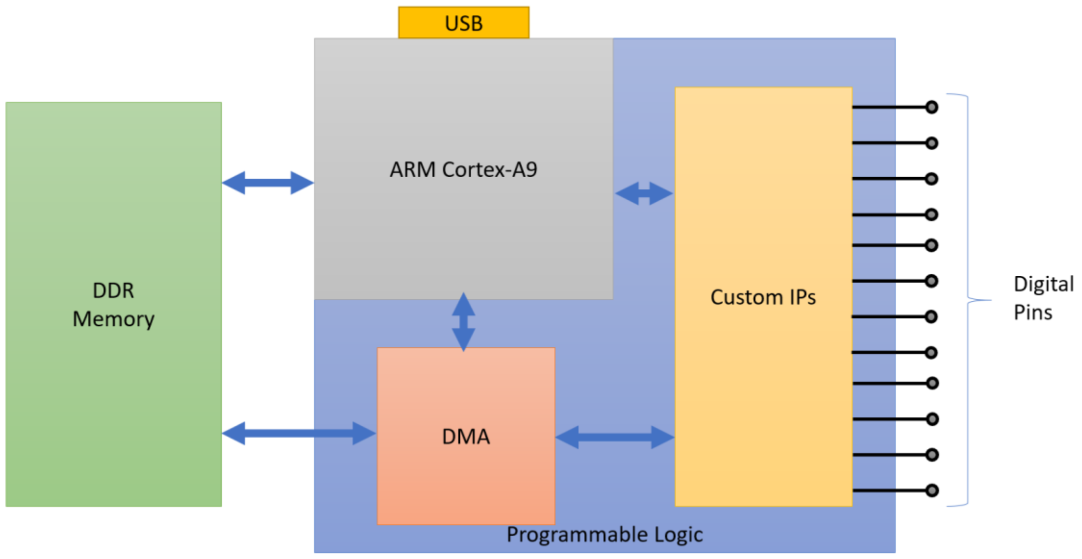


Figure 2 Simplified Falcon Architecture

The processor is the centre of the system and controls the USB, DDR, DMA and other Custom IPs. The data received through USB is stored by the processor in the DDR memory. Then it configures the custom IPs and the DMA in order to perform the operation that the user command indicates. The DMA takes the data from the DDR and send it to the Custom IPs, which code/decode it, guarantee a constant flow and send through the correct pins.

For the input data, the operation is the other way around. The DMA takes the data captured by the Custom IPs and stores it in the DDR memory. An interruption is raised and the processor can send the received data through the USB connection to the Python driver in the PC.

The digital pins of the FPGA are connected to an interposer board. As it will be explained in following sections, this board is equipped with level shifters, which allows to select the digital levels that the DUT requires. It also provides two type of connectors for the GPIO pins and SMA connectors for clocks.

Working modes

Falcon uses different working modes to transmit and capture data:

- Send and/or receive a certain number of bits in a unique operation. This is the simplest mode. Falcon will start sending data when the trigger conditions are met, until the data finishes. At the same time, it can capture a certain number of bits, starting when the trigger is activated and finishing when all the data has been captured.
- Repeatedly send a pattern. In this mode, Falcon will store a pattern in DDR memory and it will be sent through the pins every time the trigger conditions are met, with a maximum of repetitions set by the user. Similarly, a fixed amount of data can be captured with every trigger until the total amount of data has been captured .
- Send divided data. In this mode, Falcon can divide a unique set of data in chunks and send them one by one when the trigger signal arrives. Simultaneously, a chunk of data of the selected length can be captured with every trigger until all the data has been captured, as in the previous mode.

Clock frequency

The frequency of the GPIO pins can be set in two ways. The FPGA Zynq 7000 has an internal PLL which provides a configurable frequency clock. This PLL can be configured by writing some registers, so it is possible to change its frequency at any time.

On the other hand, the FPGA has several input pins for clocks, which can feed the internal clock tree. These pins allow to use an external clock for driving the GPIOs

3. Hardware

In this section of the project, we will look into the hardware design and analyse how Falcon is designed using the resources available in our FPGA (Zynq 7010) and SOM evaluation kit (MicroZed).

As seen previously, the FPGA Zynq 7010 is an FPGA which also includes an embedded microprocessor, apart from other resources that will also be used. Using a processor along with the FPGA facilitates the development of the project, as it is possible to perform tasks writing firmware, that would require very complex IPs designed using Verilog or VHDL.

Most of the Xilinx IPs available in the design suite Vivado are designed to be configured from the embedded ARM microprocessor. Even for some of them it is compulsory to be connected to certain signals of the uP. In our case all the IPs are configured or controlled by the uP, both the custom IPs (developed from scratch for this project) and the Xilinx proprietary IPs.

Regarding the clock domains, Falcon design will use two clock domains. One domain will be set by a clock whose frequency is fixed and cannot change. This domain is called "internal fixed clock domain" and its purpose is to feed all the IPs that are not related with the output and capture of data through the FPGA pins. It is also used to write all the registers in the IPs, both custom IPs and Xilinx proprietary.

The other clock domain is called "Pins clock domain" and it includes all the IPs that are directly related with the capture or output of data through the pins. These IPs are only the custom IPs. Unlike the previous clock domain, in this case the frequency can be set by the user. This configurability allows the user to select at which frequency the data will be sent or captured through the FPGA pins.

The reason to separate the design in two clock domains is to simplify the design, only changing the working frequency of the IPs controlling the pins, leaving the other IPs working always at the same frequency. In the next figure we can see a diagram showing the Falcon design with all the used IPs.

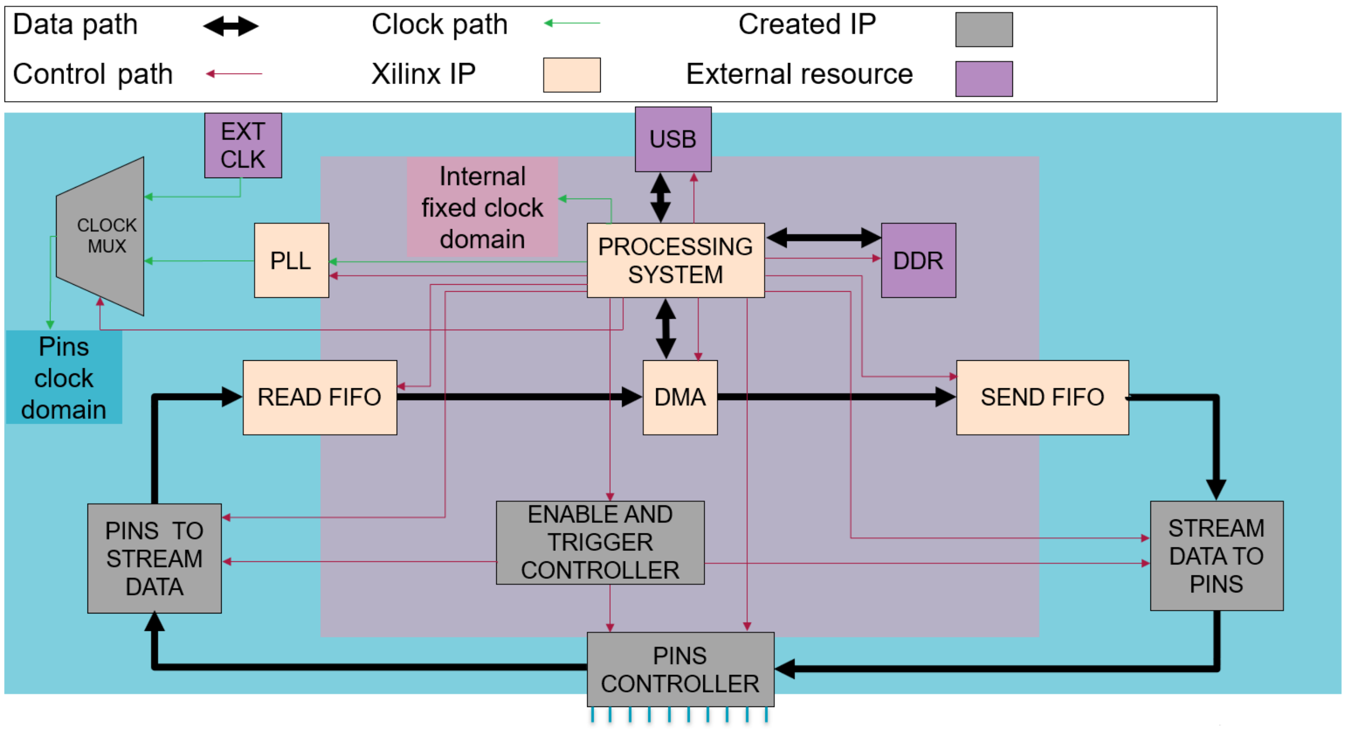


Figure 3 Falcon design. IP level

First of all, the blue colour and the red colour represent the pins clock domain and the internal fixed clock domains respectively. As we can see, each domain includes different IPs, although some of them (FIFOs and PLL) are the connection between both domains.

Custom IPs and Xilinx IPs can be differentiated, being the custom IPs more focused on the pins clock domain and the Xilinx IPs in the fixed clock domain. The design of the custom IPs, including the RTL code, will be explained in following sections.

We can also see the use of external resources, some of them located in the SOM evaluation kit, like the USB connector or the DDR memory. The input port for the external clock can be selected from some of the pins of the FPGA. This will be developed in following sections.

Finally, it is possible to distinguish between the data path, the control path and the clock path. As it will be discussed, control path and data path use different protocols to transport the information.

The processing system IP is the IP representing the microprocessor. It is the centre of the design and provides different control and data signals that feed all the other IPs. Although the microprocessor itself works at a frequency of 600 MHz, this clock is not used in the rest of the logic, so it has not been considered. This IP provides a connection with the DDR memory and the USB connector. Furthermore, it provides an internal clock, which will be used to set the frequency of the internal fixed clock domain and as an input for the PLL.

The PLL IP is a Xilinx IP which accepts one input clock and outputs a clock of a different frequency. The output frequency is set writing some registers and can be modified "on the fly", which gives more flexibility to the design. The output clock of the PLL is one of the two clocks that can set the frequency of the pins clock domain.

The other clock that can set the frequency of this domain is the external clock, which is a clock been feed to one of the pins of the FPGA which can accept an input clock. For selecting one clock or another, a custom IP has been designed. This IP acts as a multiplexor, a special one that can handle clock signals. Its output can be selected writing a value in a register and can also be changed "on the fly" but taking care of the possible glitches.

The DMA is one of the most important IPs. As it has been explained before, it moves data from one point to another, realising the processor from this task. In our design, the DMA works moving data from the DDR to the IP that will send this data through the pins. In the same way, the DMA will move the captured data from the IP to the DDR memory.

As the data needs to cross from one clock domain to another in both send and receive operations, two FIFOs have been implemented. A FIFO is one of the best solutions for data to cross between different clock domains, as it will be stored using the origin clock and read using the destination clock. In this case, the selected FIFOs are a Xilinx IP, as it has a good performance and a lot of options (like full and empty flags), which will be very useful in the rest of the design.

DMA and FIFOs use a protocol named AXI Stream Protocol (AXIS), which permits to stream data using some control signals. When sending data through pins, the DMA sends the data following this protocol, so this data stream flow needs to be correctly received and converted in raw data, discarding the control signals, in order to be sent through the pins. The same situation occurs when capturing data. The data captured is raw data corresponding to the state of the pins during different clock cycles. This data needs to be encapsulated in the AXIS protocol in order to be sent to the DMA.

The custom IPs "Stream Data to Pins" and "Pins to Stream Data" implement this functionality. As these IPs send and receive the data to and from the FIFO, this data is already in the pins clock domain, so these IPs will work in this same domain. But apart from converting the AXIS protocol to raw data, these IPs also control when and how much data is sent or captured. They are in charge of receiving the trigger and enable signals and act in consequence, dividing the data in patterns as configured.

In relation to the trigger and enable signals, the custom IP "Enable and Trigger Controller" is in charge of providing these two types of signals. This IP consists on several registers containing information about the selected options for the enable and trigger of each individual pin. This information is used to drive some combinational logic which will provide the correct signal to the IPs "Stream Data to Pins" and "Pins to Stream Data".

3.1. Processing system

The Processing System IP is the interface around the Zynq-7000 platform processing system, acting as a logic connection between the PS and the PL. This IP allows to connect the resources of the PS with those from the PL [2]. In the following figure we can see a diagram of the resources available in the PS and how they can be connected to the PL:

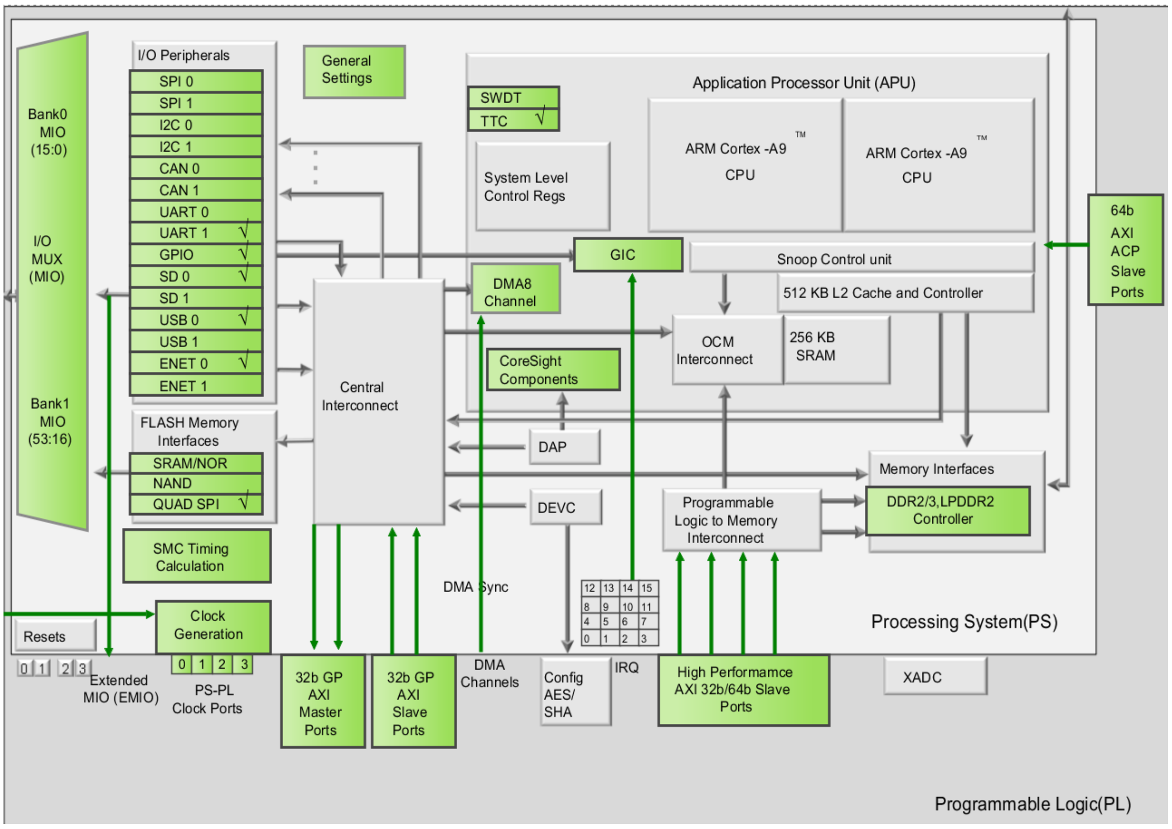


Figure 4 Processing System Diagram

The resources from the PS that will be connected to the PL in this project are the AXI interfaces, the PL clocks and the interrupts.

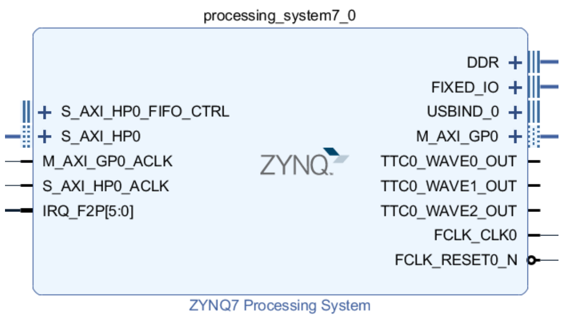


Figure 5 Processing System IP

3.1.1. AXI interfaces

The AXI I/O interface group contains AXI interfaces between the processing system and the programmable logic. The AXI interfaces include two general purpose master ports, two general purpose slave ports along with four high performance ports and an accelerator coherency port (ACP).

The AXI interfaces will be used to connect with all the other IPs of the PL, including the DMA. For those interfaces using an AXI MM Slave interface, it will be necessary to be connected to a General Purpose AXI Master port. However, there is only two ports of this type available. To solve this problem, a new IP called "AXI Interconnect" will be used.

This IP contains a configurable number of AXI-compliant master and slave interfaces and can be used to route transactions between one or more AXI masters and slaves [3]. This way, it is possible to connect multiple AXI interfaces coming from different IPs to a single AXI port in the processing system. It can be done with both master interfaces and slave interfaces. In Falcon design, there are two instances of this IP, which allows to connect all the IPs using the AXI interfaces to the ports of the processing system.

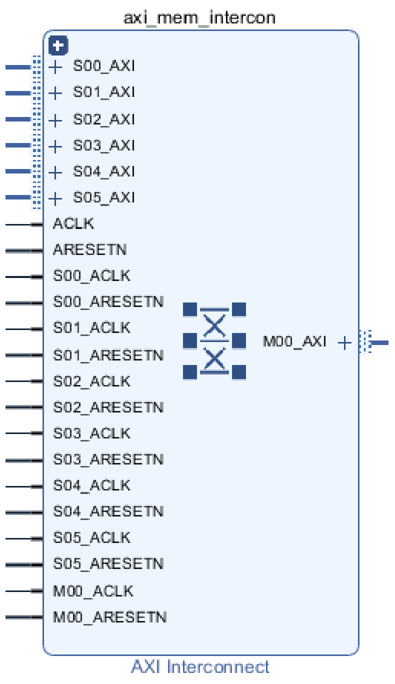


Figure 6 AXI Interconnect IP

It is also necessary to provide the correct reset signal for all this IPs using an AXI interface. For this task, there is another IP named “Processor System Reset” which is in charge of adapting the reset provided by the processing system [4]. This IP offers a wide range of possibilities for configuring the reset. Some of them are:

- Supports asynchronous external reset input and is synchronizes it with the clock
- Both the external and auxiliary reset inputs are selectable as active-High or active-Low
- Selectable minimum pulse width for reset inputs to be recognized
- Power On Reset generation

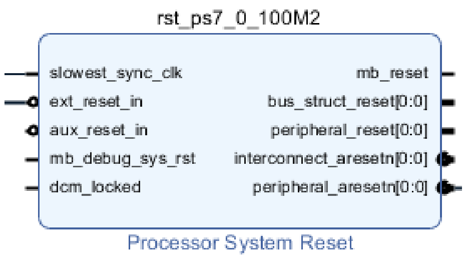
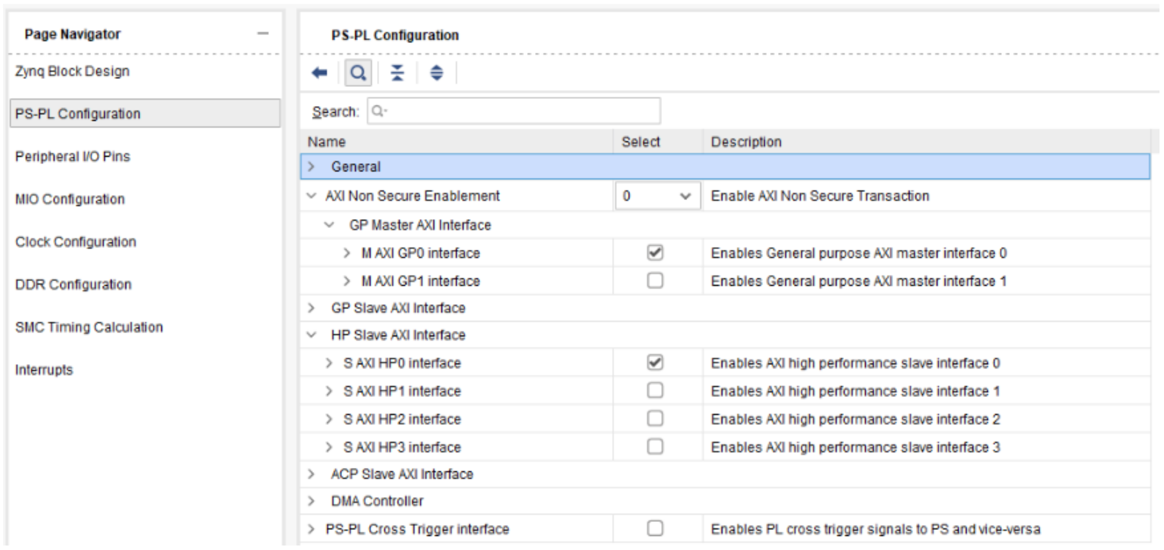


Figure 7 Processor System Reset IP

For those IPs with a Master AXI Interface, which in our design refers to the DMA only, there is the possibility of using a high-performance AXI port, which only have slave interfaces in the processing system [2]. It allows to store data in either on-chip memory or the DDR SDRAM through the SDRAM controller. If multiple connections to this port are required, the IP AXI interconnect should be used.

Typically, high-performance cores instantiated within the PL will use an AXI streaming interface. AXI streaming has no address channel and therefore needs to be converted to a memory-mapped AXI channel before it can be used to store data in a memory-mapped location, like the DDR memory. The normal method for accomplishing this is to use an AXI DMA block in the PL, as will be explained in following sections.


Page Navigator

- Zynq Block Design
- PS-PL Configuration
- Peripheral I/O Pins
- MIO Configuration
- Clock Configuration
- DDR Configuration
- SMC Timing Calculation
- Interrupts

PS-PL Configuration

Search:

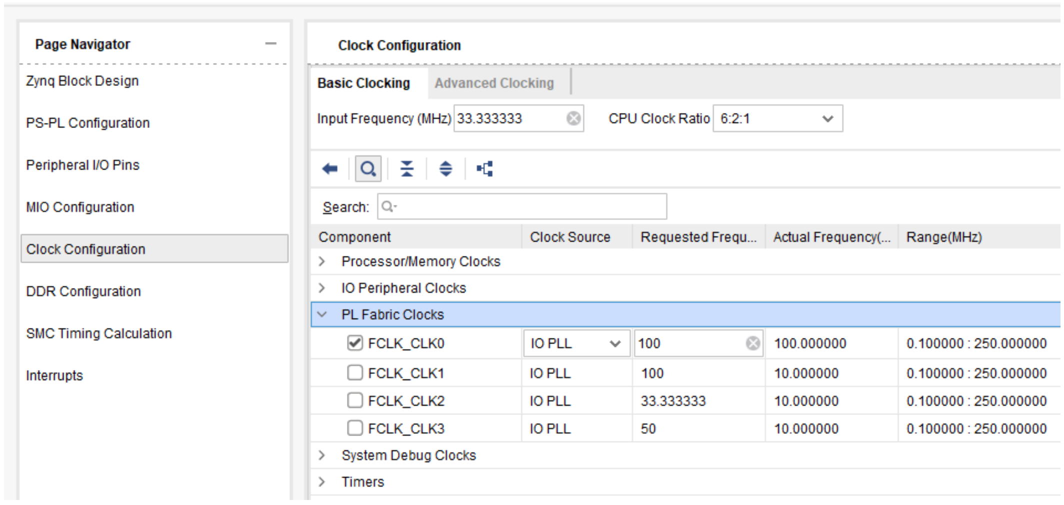
Name	Select	Description
General		
AXI Non Secure Enablement	0	Enable AXI Non Secure Transaction
GP Master AXI Interface		
M AXI GP0 interface	✓	Enables General purpose AXI master interface 0
M AXI GP1 interface	☐	Enables General purpose AXI master interface 1
GP Slave AXI Interface		
HP Slave AXI Interface		
S AXI HP0 interface	✓	Enables AXI high performance slave interface 0
S AXI HP1 interface	☐	Enables AXI high performance slave interface 1
S AXI HP2 interface	☐	Enables AXI high performance slave interface 2
S AXI HP3 interface	☐	Enables AXI high performance slave interface 3
ACP Slave AXI Interface		
DMA Controller		
PS-PL Cross Trigger interface	☐	Enables PL cross trigger signals to PS and vice-versa

Figure 8 AXI Interfaces in Processing System

3.1.2. PL Clocks

The PL, apart from having its own clock management generation and distribution, receives four clock signals from the clock generator in the PS. These clocks are completely asynchronous to each other. They produce clock waveforms suitable for PL use.

In Falcon design, one of these clocks is used to drive the "internal fixed clock domain". It is used in all IPs inside this domain and also in those in the pins clock domain which have an AXI MM interface. Apart from that, this clock is feed into the PLL in order to generate a clock with a different frequency that can be used in the pins clock domain. The selected frequency is 100 MHz.


Page Navigator

- Zynq Block Design
- PS-PL Configuration
- Peripheral I/O Pins
- MIO Configuration
- Clock Configuration
- DDR Configuration
- SMC Timing Calculation
- Interrupts

Clock Configuration

Basic Clocking | Advanced Clocking

Input Frequency (MHz) 33.333333 CPU Clock Ratio 6:2:1

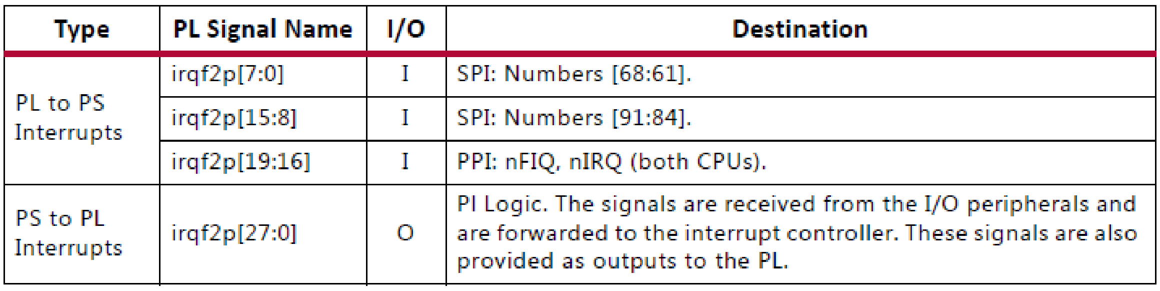
Search:

Component	Clock Source	Requested Frequ...	Actual Frequency(...	Range(MHz)
Processor/Memory Clocks				
IO Peripheral Clocks				
PL Fabric Clocks				
✓ FCLK_CLK0	IO PLL	100	100.000000	0.100000 : 250.000000
☐ FCLK_CLK1	IO PLL	100	10.000000	0.100000 : 250.000000
☐ FCLK_CLK2	IO PLL	33.333333	10.000000	0.100000 : 250.000000
☐ FCLK_CLK3	IO PLL	50	10.000000	0.100000 : 250.000000
System Debug Clocks				
Timers				

Figure 9 PL Clocks configuration

3.1.3. Interrupts

The processing system allows a certain number of interrupts from the PS to the PL, and a number of output interrupts from the PL to PS. In the next table, the available ports for both type of interrupts are listed:



Type	PL Signal Name	I/O	Destination
PL to PS Interrupts	irqf2p[7:0]	I	SPI: Numbers [68:61].
	irqf2p[15:8]	I	SPI: Numbers [91:84].
	irqf2p[19:16]	I	PPI: nFIQ, nIRQ (both CPUs).
PS to PL Interrupts	irqf2p[27:0]	O	Pl Logic. The signals are received from the I/O peripherals and are forwarded to the interrupt controller. These signals are also provided as outputs to the PL.

Figure 10 Interrupts between PL and PS

In our design, we are interested in the interrupts from the PL to the PS, as the DMA IP will assert some interrupts that need to be handled. The port used is the irqf2p, whose number of interrupts can vary from 1 to 16 depending on the designer selection. The number of interrupts connected to irqf2p is calculated and the logic ensures the correct order of an interrupt assignment.

Additionally, to connect different interrupts to this port, it is necessary to vectorize them, so that the processing system IP determines how many interrupts are connected. This can be easily done with a Xilinx IP called “Concat”. This IP simply has inputs which can be single bit signals (or multiple bit signals) and outputs the concatenated vector of all of them. All the interrupts from our system going to the PS, will be concatenated before being connected to the port irqf2p.

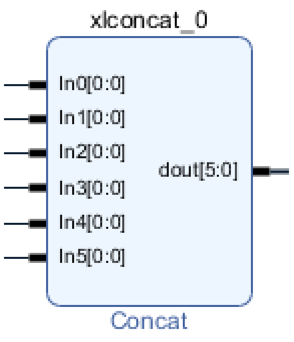


Figure 11 Concat IP

3.2. AXI

In this design, it is necessary to use buses to communicate different IPs and the PS and PL, sending and receiving control signals and data. Xilinx provide a family of buses, the AXI buses. These buses can be used to solve the communication necessities in our design.

AXI is part of ARM AMBA, a family of micro controller buses first introduced in 1996. Xilinx worked closely with ARM to define the AXI4 specification for high-performance FPGA-based systems and designs [5].

There are three types of AXI4 interfaces:

- **AXI4:** For high-performance memory-mapped data transfers.
- **AXI4-Lite:** For simple, low-throughput memory-mapped communication (for example to and from control and status registers). Is similar to AXI4 with some exceptions, the most notable is that bursting is not supported.
- **AXI4-Stream:** For high-speed streaming data. AXI4-Stream interfaces and transfers do not have address phases and are therefore not considered to be memory-mapped.

Some of the benefits of using AXI in our design are:

- By standardizing on the AXI interface, only a single protocol will be used and developers need to learn only this protocol.
- The different types of AXI protocol provide an accurate solution for different necessities of the design.
- By moving to an industry-standard, developers have access not only to the Vivado IP Catalog, but also to a worldwide community of ARM partners.

The AXI specifications describe an interface between a single AXI master and AXI slave, representing IP cores that exchange information with each other. At this point we can distinguish between the AXI Memory Map protocols and the AXI Stream protocol.

3.2.1. Memory-Mapped Protocol

In memory-mapped protocols (AXI4, and AXI4-Lite), all transactions involve the concept of transferring a target address within a system memory space and data. Memory-mapped systems often provide a more homogeneous way to view the system, because the IP operates around a defined memory map.

Both AXI4 and AXI4-Lite interfaces consist of five different channels:

- Read Address Channel
- Write Address Channel
- Read Data Channel
- Write Data Channel
- Write Response Channel

Providing separate data and address connections for reads and writes allows simultaneous, bidirectional data transfer. Data can move in both directions between the master and slave simultaneously, and data transfer sizes can vary. The limit in AXI4 is a burst transaction of up to 256 data transfers. AXI4-Lite allows only one data transfer per transaction.

The AXI4 protocol describes options that allow AXI4-compliant systems to achieve very high data throughput. At a hardware level, AXI4 allows systems to be built with a different clock for each AXI master-slave pair. In addition, the AXI4 protocol allows the insertion of register slices (often called pipeline stages) to aid in timing closure.

The following figures show an AXI4 read and write transaction using the address and data channels.

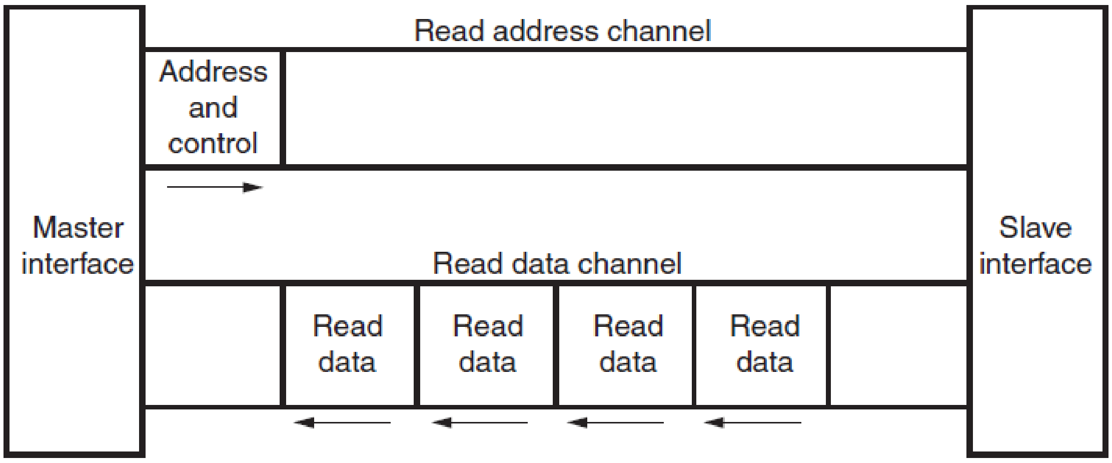


Figure 12 AXI Memory Map Read operation

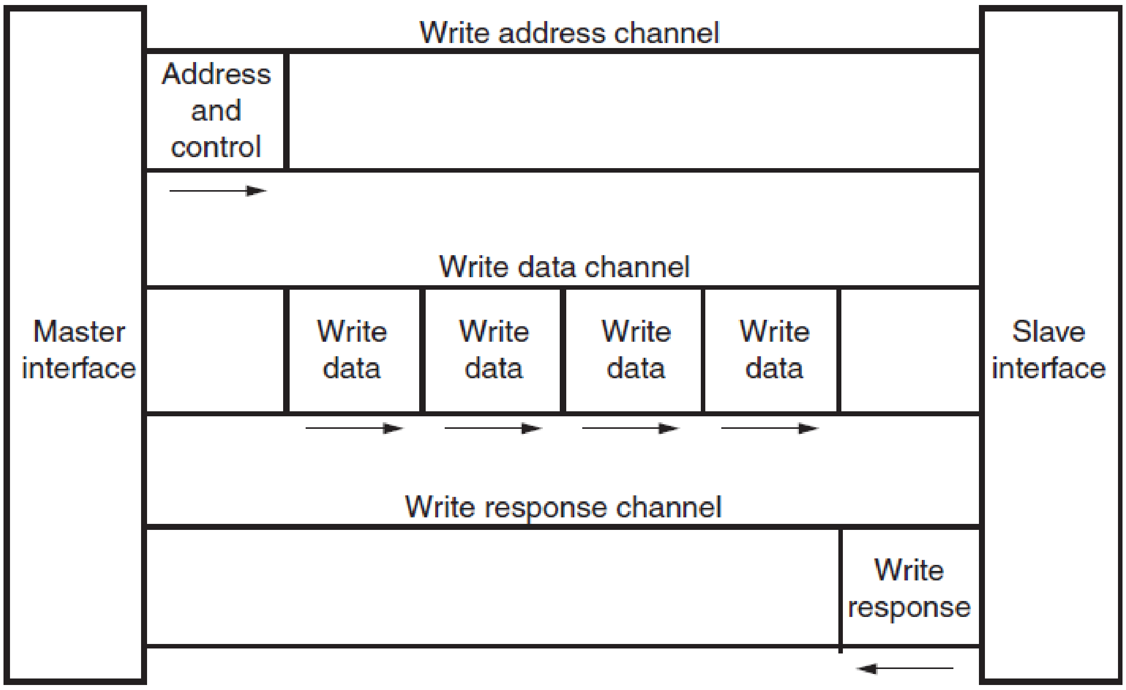


Figure 13 AXI Memory Map Write operation

3.2.2. AXI4-Stream Protocol

The AXI4-Stream protocol is used for applications that typically focus on a data-centric and data-flow paradigm where the concept of an address is not present or not required. Each AXI4-Stream act as a single unidirectional channel with a handshaking data flow.

At this lower level of operation (compared to the memory-mapped protocol types), the mechanism to move data between IPs is defined and efficient, but there is no unifying address context between IP. The AXI4-Stream IP can be better optimized for performance in data flow applications, but also tends to be more specialized around a given application space.

As defined by the AMBA standard, AXI4-Stream protocol can use up to 11 signals, some of them being optional. For this project, only 6 signals, one of them optional will be used.

- ACLK: The global clock signal. All signals are sampled on the rising edge of ACLK.
- ARESETn: The global reset signal. ARESETn is active-LOW.
- TVALID: sourced from the master side. Indicates that the master is driving a valid transfer.
- TREADY: sourced from the slave. Indicates that the slave can accept a transfer in the current cycle. A transfer only takes place when both TVALID and TREADY are asserted.
- TDATA sourced from the master. It is the primary payload that is used to provide the data that is passing across the interface. The width of the data payload is an integer number of bytes.
- TLAST sourced from the master. It is use for dividing the transferred data into packers. When asserted indicates the end of a packet, being the next payload of data the start of the next packet.

Handshake process

The TVALID and TREADY handshake determines when information is passed across the interface. A two-way flow control mechanism enables both the master and slave to control the rate at which the data and control information is transmitted across the interface. For a transfer to occur both the TVALID and TREADY signals must be asserted. Either TVALID or TREADY can be asserted first or both can be asserted in the same ACLK cycle.

A master is not permitted to wait until TREADY is asserted before asserting TVALID. Once TVALID is asserted it must remain asserted until the handshake occurs. A slave is permitted to wait for TVALID to be asserted before asserting the corresponding TREADY. If a slave asserts TREADY, it is permitted to deassert TREADY before TVALID is asserted. Some situations are showed down below:

- TVALID before TREADY handshake: In Figure 14 the master asserts the TVALID signal. Once the master asserted, the data or control information from the master must remain unchanged until the slave drives the TREADY signal HIGH, indicating that it can accept the data and control information. In this case, transfer takes place once the slave asserts TREADY HIGH.

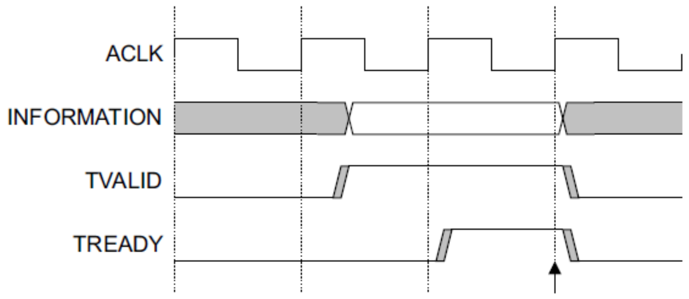


Figure 14 TVALID before TREADY handshake

- TREADY before TVALID handshake: In Figure 15 the slave asserts TREADY before the data and control information from the master is valid. This indicates that the destination can accept the data and control information. In this case, transfer takes place once the master asserts TVALID.

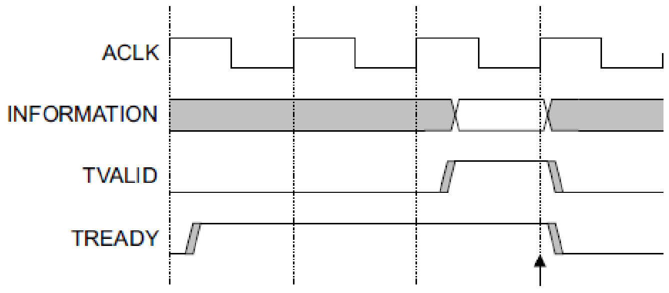


Figure 15 TREADY before TVALID handshake

Packets and TLAST

A packet is a grouping of bytes that are transmitted together across the interface. Infrastructure components can typically be made more efficient by dealing with transfers that are grouped together in packets. An AXI4-Stream packet is similar to an AXI4 burst.

When asserted during a data transmission, TLAST indicates the end of a packet boundary. When deasserted, it indicates that another transfer can follow.

For data streams that have no concept of packets or frames the default value of TLAST is indeterminate and it can be set to low, high or being periodically generated.

Any component that has the concept of packet boundaries must include a TLAST signal. When included on a component interface, the TLAST signal must be preserved through the interconnect. This is the case of Falcon, where the DMA IP works with packets and therefore it is necessary to provide the TLAST signal in all the IPs working with the AXI4-Stream Protocol.

The following figure represents the assertion and deassertion of TLAST, indicating the start and the end of two different stream packets.

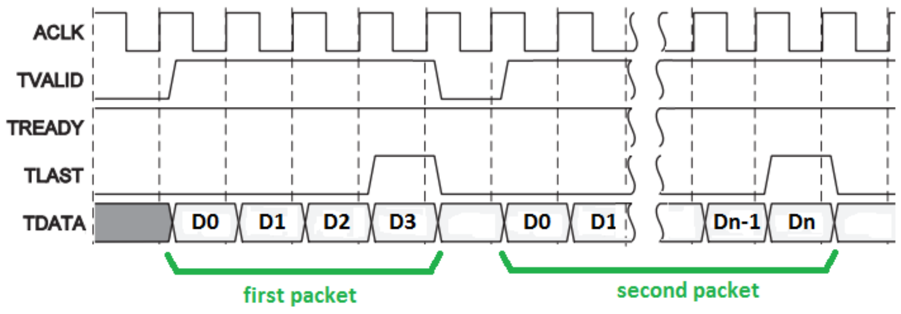


Figure 16 TLAST signal

3.3. DMA

DMA (Direct Memory Access) is a technique that allows a device to send or receive data to and from the main memory, bypassing the CPU. This technique allows a system to liberate the CPU from tasks of copying memory. It allows the CPU to perform other operations while the transfer is in progress. When the transfer is complete, the DMA will generate an interrupt. DMA not only offloads a system's processing elements but can transfer data at much higher rates than processor reads and writes.

DMA channels are used to communicate data between different addresses of system memory, different peripherals or from a peripheral to system memory [6]. In Falcon, this last option is the one used. DMA transfers data from the DDR memory to the IP that drives the pins, and from the IP for capturing data to the DDR memory.

In figure 17 we can see a diagram of the Xilinx IP "AXI DMA".

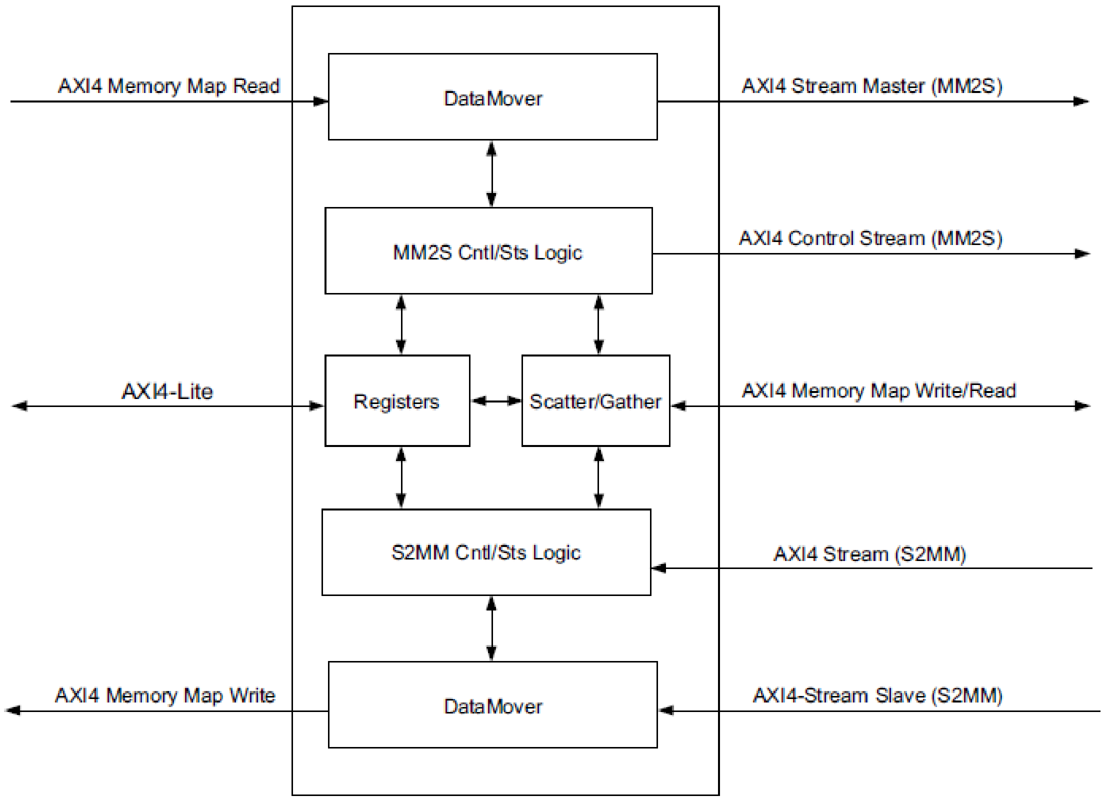


Figure 17 AXI DMA IP diagram

The AXI DMA Xilinx IP core provides high-bandwidth direct memory access between AXI4 memory mapped and AXI4-Stream IP interfaces [6]. This perfectly fits our design, as the objective is to send data in mapped memory (DDR) to IPs that needs a stream of data and vice versa. Initialization, status, and management registers are accessed through an AXI4-Lite slave interface, which has we have seen, it is available from the processing system.

The main high-speed DMA data movement between system memory and stream IP is through the AXI4 Read Master to AXI4 memory-mapped to stream (MM2S) Master, and AXI stream to memory-mapped (S2MM) Slave to AXI4 Write Master. The MM2S channel and S2MM channel operate independently and they can be transferring data at the same time. Furthermore, the AXI DMA provides byte-level data realignment allowing memory reads and writes starting at any byte offset location.

The data port AXI4 Stream Master (port M_AXIS_MM2S) and AXI4 Stream Slave (port S_AXIS_S2MM) are connected using an AXI4 Stream Interface to both FIFOs, as this are the ports that carry the stream data.

An AXI Control stream is also available in the MM2S (port M_AXI_MM2S) channel for sending user application data to the target IP. In the S2MM (port M_AXI_S2MM) channel, an AXI Status stream is used for receiving user application data from the target IP. These master control channels and an additional one for the SG mode (also master control channel) which will be explained in the following lines, are connected with an AXI MM interface to the processing system (through the AXI Interconnect IP).

Additionally, the interrupts generated in the DMA (ports mm2s_introut and s2mm_introut) are connected to the processing system through the “concat”, along with the other interrupts generated in the PL.

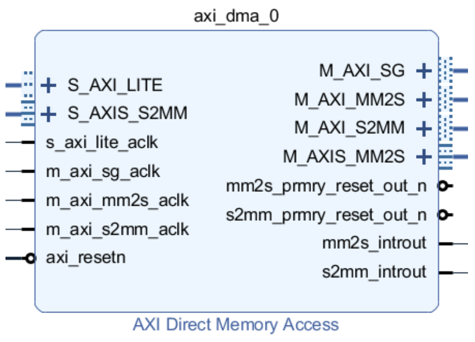


Figure 18 AXI DMA IP

As it will be seen in the firmware section, DMA can work in different modes. In the simple mode, every time the CPU wants to make a transmission, it configures the DMA and continues doing other tasks until the interrupt arrives. If then the CPU needs to perform another data transfer, it needs to program again the DMA.

The Scatter-gather mode solves this problem [7]. When the required series of transfers are known and we like AXI DMA to perform these transfers automatically, the scatter gather mode becomes useful. It allows to perform all the transfers without the need of the CPU to program the DMA for each of them. It creates structures called Buffer Descriptors, which store information about the different transfers that are going to be performed.

Another advantage of this mode is that it allows to provide data transfers from one non-contiguous block of memory to another by dividing the transfer in a series of smaller contiguous-block transfers, which correspond with the Buffer Descriptors.

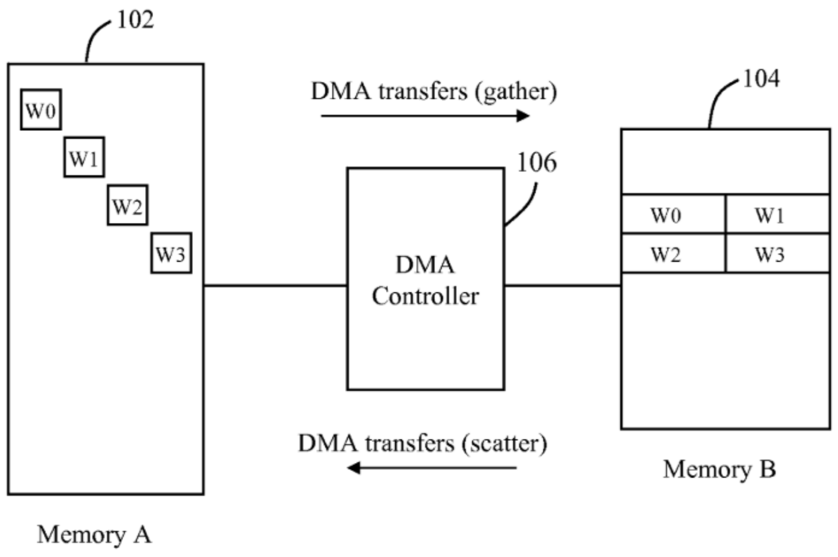


Figure 19 Non-contiguous DMA transfers

However, there is a limitation in this mode, the transfers have a maximum length. The Buffer Descriptors (BD) use DDR memory to store their information and the region they are allowed to use is limited [8]. For this reason, it is not possible to create an infinite number of Buffer Descriptors.

When the transfer configured by one BD has been completed, the DMA engine automatically starts the transmission defined by the next BD, without any gap between both transmissions. When the transmission defined by a BD has been used, it can be reconfigured with a new transfer of data.

But the BD buffers are configured by the firmware running on the ARM processor. Depending on the frequency at which the DMA is transferring the data from or to the DDR, it is possible that, eventually, all the transfers have been completed and there is no BD buffer configured to continue with the transfer of data. This happens when the DMA is working at a frequency higher than the one to which the BD are configured. When this situation occurs, the DMA is only capable of provide transfers in bursts, instead of an indefinite continuous transfer.

For Falcon we need a continuous flow of data, which can be of a length much longer than the limit of the DMA IP. In addition, the AXI DMA IP works at a fixed frequency, as it is inside the internal fixed clock domain. In order to obtain a continuous flow of data in the pins clock domain (whose frequency can be selected at any moment), it is necessary to insert some buffering, which is implemented with FIFOs. This is developed in the next section.

With this configuration, the DMA will stream data to the Write FIFO and the Read FIFO will stream data to the DMA. The DMA allows to move the data in groups of 4 bytes every cycle, which can be seen as words of 32 bits. This size is the reason why there are 32 GPIOs connected to the FPGA pins and why Falcon can send and capture words of 32 bit maximum.

3.4. FIFOs

As seen in the previous section, the DMA cannot provide a constant flow of data. It depends on the total amount of data, the size of packets and how these packets divide the data. Furthermore, the DMA works at a fixed frequency, set directly by the PL clock. It means that the data will be feed and sink at this exact frequency, without any possibility of change. To solve these two problems, FIFOs are used, one for transmission and another for reception.

A FIFO (First In First Out) is a memory for temporarily storing data while it is being moved from one place to another. Its structure is design so that the first element stored will be the first element to leave the FIFO, acting like a queue. A FIFO can be implemented using a shift register or using other memory structures, like a circular buffer [9].

A common implementation uses dual port RAM, creating a circular buffer with two pointers. It allows to use a pointer for writing and a different pointer for reading. The write pointer is incremented when a new element is written in memory. The read pointer is incremented when an element has been read from memory. When the pointers reach the end of memory, the next position will be the beginning of the memory (circular buffer). This way, the First In First Out performance is achieved.

The control logic (or software) manages these pointers and takes care of the limit situations. When the write pointer is just before the read pointer, the FIFO is full and cannot accept more data, the write pointer cannot move. When the read pointer is at the same position as the write pointer, it means that the FIFO is empty, and the next written element will be the next read element.

It is possible to distinguish between synchronous and asynchronous FIFOs. While the first type uses the same clock for both write and read operations, the second type uses a clock for write data into the FIFO and another for reading from it. This feature allows a data flow to cross between two clock domains.

In the FPGA Zynq 7000, the FIFOs are implemented with Block RAM, which is a resource available in almost all FPGAs [10]. Xilinx provides a FIFO IP which uses the AXI Stream protocol, has a configurable size and some configurable flags. It is called AXIS4 Stream Data FIFO.

This IP can accept data directly from the DMA, as the AXIS protocol is common in both resources. The output of the FIFO also works with the AXIS protocol, so the data will have to be decoded with some custom IP after this step.

AXIS4 Stream Data FIFO can be configured as asynchronous so that it needs two clocks, for reading and for writing. In the case of the TX FIFO (see figure 20), the write clock is the internal fixed clock, because the data coming from the DMA is in this clock domain. As the objective is to cross to the configurable clock domain, the read port will use the configurable clock, so that the data is taken from the FIFO at this frequency. At this point, the clock domain has been successfully changed. For the RX FIFO, the change in the clock domain is done in the opposite way. The write clock is the configurable clock and the read clock is the fixed clock.

As seen in the previous section, the DMA feeds data to the FIFO in bursts. However, for our application, a constant flow of data is required. The TX FIFO can solve this problem. When starting a transmission, the TX FIFO is desired to be as full as possible. The reason is to increase the time that the FIFO can stream data, without becoming empty, when no data is being written to it. This time will depend on the initial data in the FIFO, the size of the gaps between burst of the DMA and the difference between clocks.

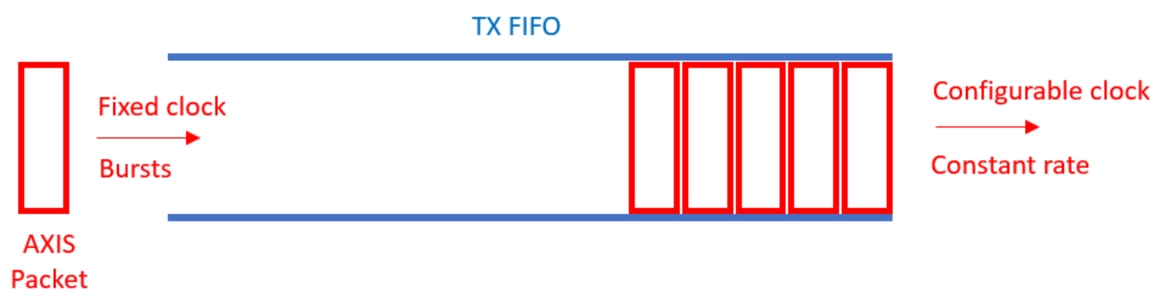


Figure 20 TX FIFO

The difference between the frequency of the two clock domains is critical for this margin time. If the two frequencies are very similar, or even the configurable frequency is higher than the fixed one, the FIFO will not be able to maintain a constant flow of data at its output for a long time. On the other hand, if the configurable frequency is low enough with respect to the fixed frequency, the gaps in the DMA transmission will be compensated and the constant flow of data can be maintained for an indefinite time. This is the normal behaviour of Falcon, although for some special cases, the first one can be used.

In the FIFO IP used, there is a flag called "almost full" which indicates when only one more packet can be stored in the FIFO. This flag is particularly useful for indicating when the tx transmission can be started.

In the RX side, a similar problem can be observed. As the DMA takes the data from the RX FIFO in bursts, it is necessary to avoid the situation in which the RX FIFO is full. This would cause a problem because the new data would be lost for not having space in the FIFO. Again, the difference in frequencies of both clock domains and the size of the bursts are very significant to define the time that the data can be read without any loss. A configurable frequency higher or very similar to the fixed internal frequency can be tolerated for some time, but eventually some data will be lost. As before in the tx side, this working mode can be used for some situations under control, but in the normal working mode the fixed frequency will be high enough with respect to the configurable frequency so that a constant reception of data can be maintained for an indefinite time.

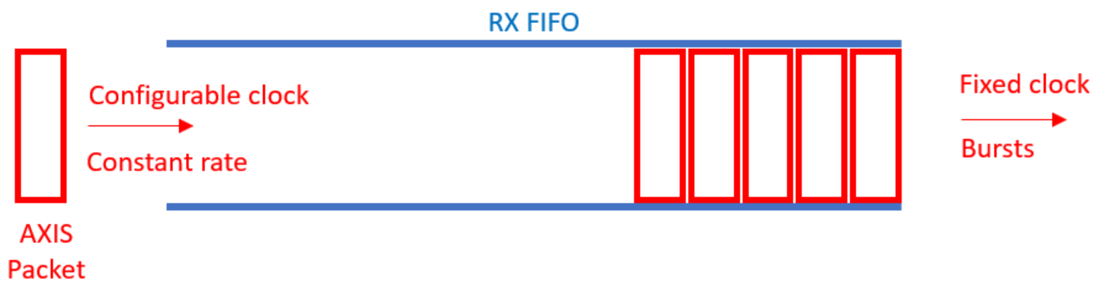


Figure 21 RX FIFO

3.5. Stream Data to Pins IP

Together with the IP “Pins to Stream Data”, the IP “Stream Data to Pins” (also called “write pins”) is in charge of convert between the AXIS stream data and raw data to be send or captured. They also control when the send or capture operations start and end, depending on the configured mode and the trigger.

This IP takes the continuous stream of data provided by the Write FIFO and decodes the AXIS protocol in order to obtain the raw data that will be sent through the pins. For decoding the protocol, it first needs to implement an AXIS slave interface in order to receive the stream correctly. On the other hand, it uses a 32-bit wide output to provide the raw data to the IP “Pins Controller”.

Apart from the input and output interfaces, this IP uses some registers in order to store information like the total number of data that will be sent in one operation, the length of the patterns, the state of the transmission, etc.

For implementing these registers, Xilinx provide a template which implements the registers and the necessary logic to write and read them, using the AXI4 Memory Mapped protocol. This template consists in a module which is instantiated in the top module of the custom IP and provides the necessary signals to create an AXI4 MM interface. All the custom IPs that need registers use this template with little modifications.

This IP is composed of three modules. The first one is the mentioned module with the registers, using the AXI4 MM protocol, as it needs to read and write information from a mapped memory. The second module is where the main logic is placed and uses the AXI4 Stream protocol, as it has to interact with the FIFO. Finally, the third module is the top module, where the other two are instantiated.

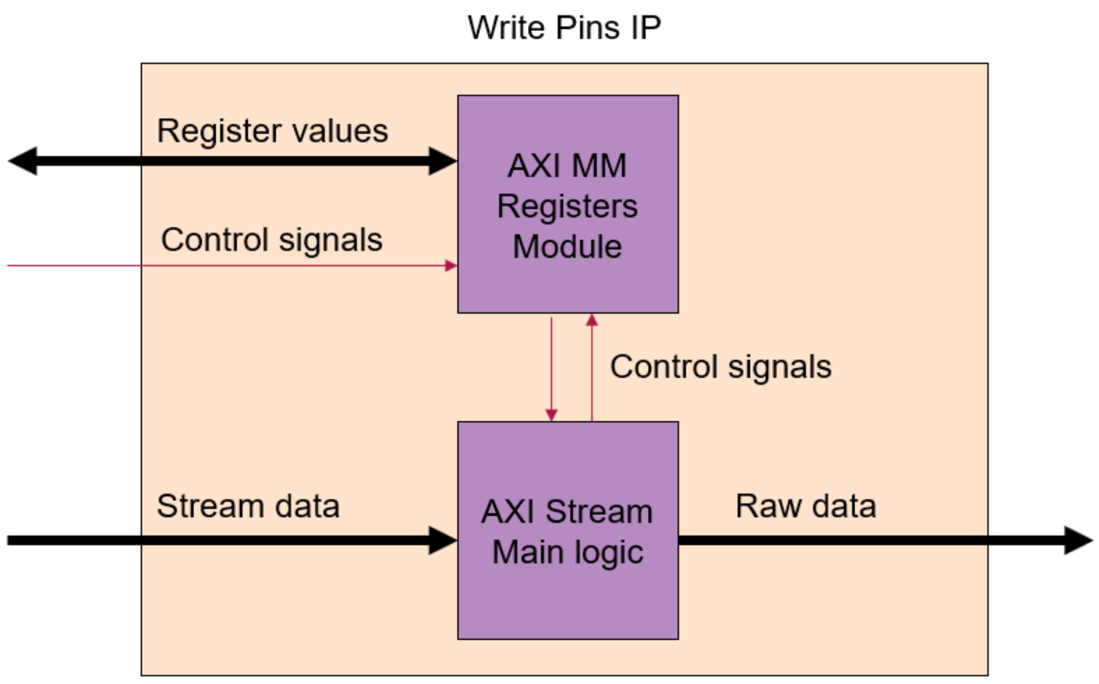


Figure 22 Stream Data to Pins IP

Top Module (myip_write_pins_v1)

This module only works as a wrapper to join the other two modules. It is composed of the instantiations of these two modules and provides the necessary connections between the modules and to the exterior. There is no additional logic.

AXI MM Registers Module (myip_write_pins_v1_0_S00_AXI)

The objective of this module is to provide the necessary configuration data for the IP. AS mentioned before, it uses a template provided by Xilinx. In this template, some signals with the value of the registers can be added so they are directly used by the main logic of the custom IP. For this custom IP, the signals of the AXI MM module are:

- **AXI MM interface signals**: s00_axi_aclk, s00_axi_aresetn, s00_axi_awaddr, s00_axi_awprot, s00_axi_awvalid, s00_axi_awready, s00_axi_wdata, s00_axi_wstrb, s00_axi_wvalid, s00_axi_wready, s00_axi_bresp, s00_axi_bvalid, s00_axi_bready, s00_axi_araddr, s00_axi_arprot, s00_axi_arvalid, s00_axi_arready, s00_axi_rdata, s00_axi_rresp, s00_axi_rvalid, s00_axi_rready,

- **end_of_transfer**: it is a single bit input signal coming from the main logic. It is used as an interrupt to the PS to indicate that the configured transfer has been completed.

- **fifo_almost_full**: it is a single bit input signal coming from the main logic. It asserts a bit of a register to indicate that the write FIFO has only one position left. This bit will be read by the firmware to start the transmission at a point when the data has been buffered as much as possible.

- **total_words**: this 32 bit wide signal copies the value of a register which is written to indicate the size of the transmission. This size is indicated in words of 32 bits.

- **reset_words_counter**: this single bit signal copies the value of a bit in a register. When asserted, a counter in the main logic for the total number of words sent will be reset.

- **repeated_pattern_mode**: it is a single bit signal. When it is 0, it means the pattern mode is deactivated and a unique set of data will be sent. When it is 1, the data will be sent in patterns.

- **pattern_len**: this 32 bit signal sets the size of the patterns when working in this mode.

A total of 6 registers are used to store all the needed information to configure this IP. In the template, the logic of the register fifo_almost_full has been modified so that these registers are read-only from the part of the firmware.

AXI Stream Main Logic (myip_write_pins_v1_0_S00_AXIS)

This module contains the main logic of the IP and is in charge of receiving the AXIS protocol, extract the data and send it to the next IP following the configuration set in the registers.

The I/O signals of this module are:

- **S_AXIS_ACLK:** The clock which drives the AXIS protocol. It is the clock of the pins clock domain.

- **S_AXIS_ARESETN:** Reset signal of the AXIS protocol. Active at low level.

- **S_AXIS_TREADY:** TREADY signal of the AXIS protocol. It is an output signal going to the AXIS master interface. It indicates that the Write Pins IP is ready to receive data.

- **S_AXIS_TVALID:** TVALID signal of the AXIS protocol. It is an input signal coming from the AXIS master interface. It indicates that the Write FIFO is ready to send data.

- **S_AXIS_TLAST:** TLAST signal of the AXIS protocol. It is an input signal and indicates the end of a DMA packet.

- **S_AXIS_TDATA:** TDATA signal of the AXIS protocol. It is a 32 bit input signal coming from the AXIS master interface and transports the payload with the transferred data.

- **enable:** It is an activation signal coming from the enable and trigger controller IP. When asserted, the transmission can start if the rest of the conditions are met.

- **trigger:** This signal starts the transmission one cycle after it is asserted, if the rest of the conditions are met. It is an input signal coming from the enable and trigger controller IP

- **total_words:** It is a 32 bit input signal coming from the registers module and contains the information of the length of the transmission, in words of 32 bits.

- **reset_words_counter:** this input signal also comes from the register's module. When it is asserted, the words counter is set to 0.

- **end_of_transfer:** An output signal going to the register's module. It is asserted when the transmission has finished. It will write a 1 in a register so that firmware is able to read it and determine if the transmission is still running.

- **repeated_pattern_mode:** This input signal configures the logic to work in pattern mode when its value is 1.

- **pattern_len:** This input 32 bit signal contains the information about the length of the patterns. It comes from the register´s module

- **pins_out:** It is a 32 bit output signal. It contains the raw data that will be sent to the IP Pins Controller.

This module is based in an FSM of two states and two counters which control the change of state in the FSM. Additionally, there is some combinational logic to provide other outputs.

- **Words counter**: It is incremented when a new 32-bit word has been received and sent through the port "pins_out" to the next custom IP. The counter is set to 0 when the AXIS reset is activated or when the reset_words_counter signal is asserted. When it arrives to the maximum value it stays in this value to indicate that the transfer has finished.

- **Pattern words counter**: This counter controls the size of the pattern when the transfer works in pattern mode (selected by the signal repeated pattern mode). It is incremented when a new 32-bit word has been received and sent through the port "pins_out" to the next custom IP. It is automatically set to zero when the count reaches the value set by "pattern_len"

Regarding the FSM machine, its two states are:

- **IDLE**: In this state the IP is not transmitting or receiving any data. The AXIS signal TREADY is de-asserted so the write FIFO does not send data (as seen in previous sections, it is necessary that TREADY and TVALID signals are asserted to transfer data).
  This is the initial state. The FSM will remain in this state until the words counter becomes lower than the maximum count (typically 0 when the reset occurs) and the trigger and enable signal are asserted.
  When these three conditions are met, the state goes to the next state "SEND".

- **SEND**: In this state, the transfer is performed. The TREADY signal is asserted so, if Write FIFO asserts TVALID, the transmission happens. If these two signals are asserted, the payload coming from TDATA is sent through the port "pins_out" as 32 bit raw data. This data will be output through the FPGA pins.
  Also, when TREADY and TVALID are asserted the words counter is incremented and the pattern counter is incremented if the "repeated pattern mode" signal is asserted.

From this state, the FSM will go to “IDLE” if the enable signal is de-asserted, if the words counter reaches the end of the count or if, being in pattern mode, the pattern counter finishes and no trigger arrives in that same clock cycle to continue with the transmission.

Finally, there is some logic which generates two signals. First, the “pins_out” signal outputs the payload in the AXIS signal TDATA when the transmission is being performed. But when the transmission stops, it is desired to keep the last value at the output. For this purpose, the output value at every cycle of the transmission is stored in a register so that it can be used when the transmission stops. The “pins_out” signal is the output of a multiplexor whose inputs are the TDATA signal and the register with the last value. Its selection signal is the and combination of TREADY and TLAST, which indicates that the transmission is being performed.

The other generated signal is “end_of_transfer”, which is more simple. It is asserted when the words counter reaches the end of the count. As explained before, this information is stored in a register so that the firmware is able to determine is the DMA operation has finished.

3.1. Pins to Stream Data IP

This IP is the complementary to the “Pins to Stream Data IP”, as it performs the same operation but in the other way around. It takes the captured 32-bit words of raw data from the IP “Pins controller” and encapsulates it in the AXI Stream protocol in order to be sent to the Read FIFO. Therefore, this IP needs to implement an AXIS Master interface, to provide the data stream.

Like the other IP, some registers are needed in order to configure how the transmission is going to perform. It is necessary to define the number of words to transfer, the length of the patterns, the transmission mode (unique stream or pattern), etc.

Again, we use the Xilinx template to implement these registers, as it already provides the necessary logic to write and read them from the firmware, using the AXI MM protocol. This module allows the IP to have an AXI MM interface through which it can communicate with the processing system.

This IP is composed of three modules too. The first one is the register's module with the AXI4 MM interface, as it needs to read and write information from a mapped memory. The second module is where the main logic for the transmission is placed and, as communicates directly with the AXIS FIFO, it has an AXI4 Stream interface. Finally, the third module is the top module, where the other two are instantiated.

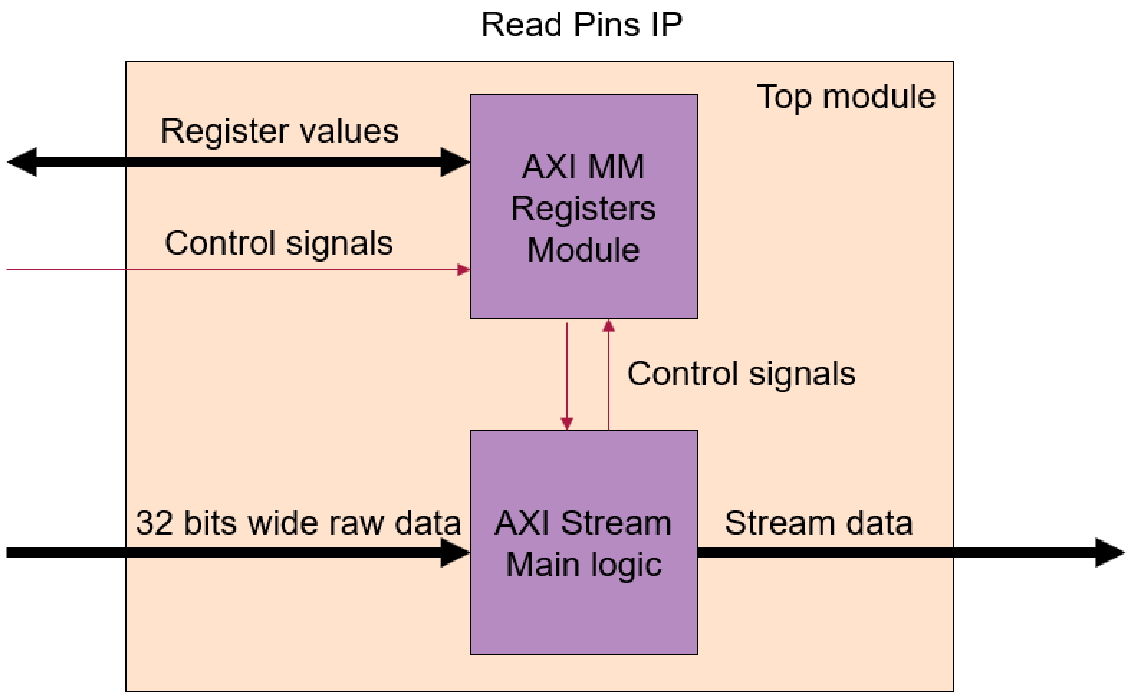


Figure 23 Pins to Stream Data IP

AXI MM Registers Module (myip_read_pins_v1_0_S00_AXI)

The objective of this module is to store the necessary configuration values for the rest of the logic of the IP, using the template provided by Xilinx. To complete this template, some signals copying the value of the selected registers have been added, so that they provide this information to the other module. The I/O signals of the AXI MM module are:

- **AXI MM interface signals**: s00_axi_aclk, s00_axi_aresetn, s00_axi_awaddr, s00_axi_awprot, s00_axi_awvalid, s00_axi_awready, s00_axi_wdata, s00_axi_wstrb, s00_axi_wvalid, s00_axi_wready, s00_axi_bresp, s00_axi_bvalid, s00_axi_bready, s00_axi_araddr, s00_axi_arprot, s00_axi_arvalid, s00_axi_arready, s00_axi_rdata, s00_axi_rresp, s00_axi_rvalid, s00_axi_rready,

- **words_per_packet**: This signal is new in this IP and does not appear in the Write Pins IP. It is a 32 bit wide signal which indicates by how many words a DMA packet is composed. As the AXIS TLAST signal is used to mark the end of a packet, the value of this signal determines the periodicity at which TLAST will be activated.

- **number_of_output_words:** a 32 bit signal that indicates the size of the transmission. It copies the value of a register which is written by the firmware to set this configuration. This size is indicated in words of 32 bits

- **reset_words_counter**: this single bit signal copies the value of a bit in a register. When asserted, a counter in the main logic for the total number of words sent will be reset.

- **repeated_pattern_mode**: a single bit signal. When it is 0, it means the pattern mode is deactivated and a unique set of data will be sent. When it is 1, the data will be sent in patterns.

- **pattern_len**: this 32 bit signal sets the size of the patterns when working in this mode.

A total of 6 registers are used to store all the needed information to configure this IP. In this case there are no read-only registers, as the information in the registers is going from the firmware to the PL logic.

AXI Stream Main Logic (myip_read_pins_v1_0_M00_AXIS)

This module contains the main logic of the IP and its functionality consist in receiving the raw data captured and encapsulate it using the AXI Stream protocol, in order to be sent to the FIFO. For this operation it needs the configuration options provided by the previous module. In addition, it needs to implement an AXIS interface.

The I/O signals of this module are:

- **M_AXIS_ACLK:** The clock which drives the AXIS protocol. It is the clock of the pins clock domain.

- **M_AXIS_ARESETN:** Reset signal of the AXIS protocol. Active at low level.

- **M_AXIS_TREADY:** TREADY signal of the AXIS protocol. It is an input signal coming from the Read FIFO. It indicates that it is ready to receive data new data.

- **M_AXIS_TVALID:** TVALID signal of the AXIS protocol. It is an output going to the Read FIFO. It indicates that this IP is ready to send data and that the data at its output is already valid.

- **M_AXIS_TLAST:** TLAST signal of the AXIS protocol. It is an output signal and indicates the end of a DMA packet when asserted

- **M_AXIS_TDATA:** TDATA signal of the AXIS protocol. It is the 32-bit output signal going from the AXIS master interface to the Read FIFO. It transports the payload with the captured data.

The rest of the signals are only input signals

- **input_pins**: 32bit wide signal coming from the IP “Pins Controller”, it contains the captured data that is going to be encapsulated.

- **enable:** It is an activation signal coming from the enable and trigger controller IP. When asserted, the transmission can start if the rest of the conditions are met.

- **trigger:** This signal starts the transmission one cycle after it is asserted, if the rest of the conditions are met. It is an input signal coming from the enable and trigger controller IP

- **words_per_packet**: coming from the register´s module, indicates how many words fits in a DMA packet and therefore, when the TLAST signal is going to assert.

- **number_of_output_words:** coming from the register´s module, contains the information of the length of the transmission, in words of 32 bits.

- **reset_words_counter:** coming from the register´s module. When it is asserted, the words counter is set to 0.

- **repeated_pattern_mode:** coming from the register´s module, configures the logic to work in pattern mode when its value is 1.

- **pattern_len**: coming from the register´s module, this 32-bit signal contains the information about the length of the patterns.

Similar to the previous IP, this one also uses an FSM with two states and some counters to control it, apart from signals generated with combinational logic.

- **Output words counter**: It is incremented when a new 32-bit word of raw data has been received through the port “input_pins” and has been encapsulated and send to the Read FIFO. The counter is set to 0 when the AXIS reset is activated or when the reset_words_counter signal is asserted. When it arrives to the maximum value it stays in this value to indicate that the transfer has finished and to stop the IP.

- **Pattern words counter**: Again, this counter controls the size of the pattern when the transfer works in pattern mode (selected by the signal repeated pattern mode). Its count is incremented when a new 32-bit word has been processed, which implies that the signals TREADY and TVALID are asserted. The count is automatically set to zero when the count reaches the value set by “pattern_len”.

- **Packet counter:** This new counter is used to determine when the DMA packet finishes and therefore the TLAST signal must be asserted. The maximum count is determined by the signal “words_per_counter”. It will be incremented exactly at the same time than the other two counters, that is when the signals TREADY and TVALID are asserted and so a new word has been transferred. The count will be set to 0 when a reset is applied, when the maximum value is reached or when the output words counter reaches its maximum value.

With regards to the FSM machine, the two states are the same than in the other IP:

- **IDLE**: In this state the IP is not transmitting or receiving any data. The AXIS signal TVALID is de-asserted so the read FIFO does not receive any data. This is the initial state.
  The FSM will remain in this state until the words counter becomes lower than the maximum count (typically 0 when the reset occurs) and the trigger and enable signal are asserted.
  When these three conditions are met, the state goes to the next state “SEND”.

- **SEND**: In this second state, the transfers are performed. The TVALID signal is asserted so, if the TREADY signal from the Read FIFO is also asserted, the transmission of data will occur. For encapsulating the captured raw data in the AXIS Protocol, it is necessary to send this raw data through the TDATA port only when the TVALID and TREADY signals are asserted.

  From this state, the FSM will go to “IDLE” if the enable signal is de-asserted, if the words counter reaches the end of the count or if, being in pattern mode, the pattern counter finishes and no trigger arrives in that same clock cycle to continue with the transmission.

And finally, there are two signals that are generated using combinational logic. The first one is the TDATA signal, which comes out of a multiplexor. Similarly to the previous IP, if the TREADY and TVALID signals are asserted, the data is taken directly from the “input_pins” signal, as both signals have the same width, 32 bits. However, if the transfer is not active, we need the TDATA signal to remain in its last valid value. For this purpose, the output value is registered every clock cycle while the transfer is active. When one of the two signals is not active, the output of the multiplexor will be the last stored value.

And the last generated signal is the AXIS TLAST signal. As it sets the end of the DMA packets, it is based on the words per packet counter. When this counter reaches its maximum value, if the transfer is active, the TLAST signal is asserted. But there is the possibility that the size of the transmission is not multiple of the size of the DMA packet. In this case, the last packet will be shorter. For this reason, TLAST is also asserted when the output words counter reaches its maximum count.

3.2. Pins Controller IP

At this point, we have analysed of the data is taken from the memory, how it is sent using the AXI Stream protocol by the DMA and how our IPs convert this stream in raw data to be sent through the pins. The opposite process occurs when the operation is to capture data. The last step in this process is to control the pins, capturing its value or sending data through them. This is the functionality of the "PINS CONTROLLER IP".

In the FPGA the pins are input/output, but for our application, it is necessary to be able to select either input or output for every pin. Selecting this option for the pin is important because we need the electronic to either provide a voltage to drive the pin or capture the voltage in the pin. These different behaviours require changes in the electronic, which make necessary to implement an electronic structure which allows to change the behaviour of every pin. This way, it is possible to configure some pins to send data and some other pins to capture data.

In order to implement this structure, we will use primitives [11]. Primitives are the basic embedded components that compound a FPGA. These components can be PLLs, BUFFERS, RAM, etc. When a primitive is instantiated in the design, it will be present in the final result after the synthesis. The opposite term of primitive are macros, which are tools to instantiate a group of primitives which is complex to instantiate one by one. After the synthesis, the macros expand to their underlying primitives.

One of the primitives that Xilinx provide is a double buffer to use with bidirectional signals. It is called "IOBUF"

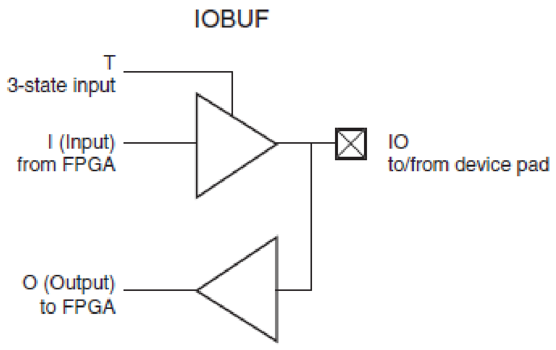


Figure 24 IO Buffer

It uses an input and an output buffer. This output buffer is a 3-state buffer, which can be controlled with an active high pin (T). It is necessary one of these primitives for every pin, so has we use 32 GPIO pins, it is necessary to instantiate this primitive 32 times (this can be easily done with the generate function in Verilog).

If a pin is configured as output, the T pin will be de-asserted. This way, when an output signal comes from the FPGA and arrives to this structure, the output buffer will drive the pin. The buffer also isolates the internal circuit of the FPGA from the circuit connected to the pin, so that the input impedance of what is connected does not interfere with the internal electronic of the FPGA . It also provides the necessary current to drive the circuit connected to the pin, until a certain limit.

On the contrary, when a pin is configured as input, the T pin is asserted and the output buffer is in 3-state, as if this connection did not exist. This time, it is the input buffer the one that isolates the internal circuit of the FPGA from the one that is driving the pin, so that the output impedance of the driver does not create internal problems in the FPGA.

This IP is compounded of two modules: the top module containing the instantiation of the primitives and a module with a configuration register.

The top module instantiates an IOBUF primitive for every pin and connects them to the buses going to the IPs “Pins to Stream Data” and “Data to Stream Pins”. The signals in this module are:

- **out_pins:** Input signal of 32 bits. This is the bus containing the raw data that is going to be sent through the pins. It comes from the IP “Stream data to pins”. Although its size is 32 bits, only the bits corresponding with a pin configured as output will carry a valid value.

- **in_pins:** Output signal of 32 bits. This is the bus containing the captured raw data that is going to be sent to the IP “Pins to Stream Data”. Although its size is 32 bits, only the bits corresponding with a pin configured as input will carry a captured value and the others will be discarded by the firmware.

- **inout_pins:** This is a 32 bit input/output signal. It directly connects the output of each IOBUF primitive to its assigned pin.

- **AXI MM interface signals**: s00_axi_aclk, s00_axi_aresetn, s00_axi_awaddr, s00_axi_awprot, s00_axi_awvalid, s00_axi_awready, s00_axi_wdata, s00_axi_wstrb, s00_axi_wvalid, s00_axi_wready, s00_axi_bresp, s00_axi_bvalid, s00_axi_bready, s00_axi_araddr, s00_axi_arprot, s00_axi_arvalid, s00_axi_arready, s00_axi_rdata, s00_axi_rresp, s00_axi_rvalid, s00_axi_rready.

As mentioned before, this IP requires one register to configure the pins as input or output. As it has been done in previous IPs, this IP uses the Xilinx template that contains registers and the logic to create an AXI Memory Mapped interface to write and read them.

The signals of the module myip_select_io_dir_v1_0_S00_AXI are:

- **AXI MM interface signals**: s00_axi_aclk, s00_axi_aresetn, s00_axi_awaddr, s00_axi_awprot, s00_axi_awvalid, s00_axi_awready, s00_axi_wdata, s00_axi_wstrb, s00_axi_wvalid, s00_axi_wready, s00_axi_bresp, s00_axi_bvalid, s00_axi_bready, s00_axi_araddr, s00_axi_arprot, s00_axi_arvalid, s00_axi_arready, s00_axi_rdata, s00_axi_rresp, s00_axi_rvalid, s00_axi_rready.

- **in_out_select:** it’s a 32-bit output signal, containing the value of an entire register. Each of its bits represent if the corresponding GPIO is configured as input (1) or output (0).

3.3. PLL IP

As described in the diagram of figure 3, there are two options for setting the frequency of the pins clock domain. One of them is the output of a configurable PLL, which allows to set the desired frequency by writing some registers.

A PLL is an electronic circuit which uses a voltage-driven oscillator that automatically adjusts to match the frequency of an input signal. PLLs are used to generate, stabilize, modulate, demodulate, filter or recover a signal from a "noisy" communications channel. It can be used to generate a clock at a desired frequency from a clock working at a different frequency. This last application is used in out design, to obtain a desired frequency given an input fixed clock, coming from the processing system (PL clock at 100 MHz).

Xilinx provides an IP with this functionality. It is called “Clocking Wizard”. This IP simplifies the creation of HDL source code wrappers for clock circuits, as it integrates all the necessary logic and primitives [12]. It automatically calculates the voltage-controlled oscillator (VCO) frequency for the primitives and provides multiply and divide values based on input and output frequency requirements. It also has some advantages like the Safe Clock Startup feature, that enables the output when the clock is valid and stable.

But the big advantage of this IP is that provides an AXI4-Lite interface for dynamically reconfiguring the clocking primitives. It allows this IP to automatically change the output frequency when a new configuration is applied.

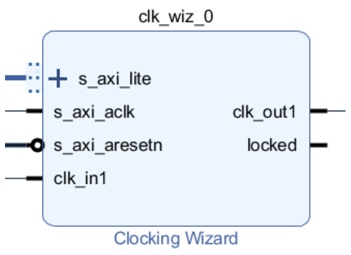


Figure 25 Clocking Wizard IP

This IP is connected to the AXI Interconnect block. Its input clock comes directly from the PL clock provided in the processing system IP and its output is feed to the “Clock Mux IP”.

3.4. Clock Mux IP

This IP acts as a clock multiplexor, allowing the firmware to select which clock will set the frequency of the pins clock domain. This mux selects between the internal generated clock in the PLL and an external clock coming to the FPGA through a specific pin.

When working with clock signals, it is necessary to put extra attention in the resources used, as these signals present more restrictions than the control or data signals. First, they present a higher fanout, as they feed many different elements. Second, they are the signals with the highest switching speed. Third, they need to be as clean as possible, as they provide the temporal reference for the logic. And finally, they phase delay that they present between different points should be acceptable.

For satisfy these restrictions, the clock signals need to use the specific clock resources available in the FPGA and be routed through the clock tree, which is a clock distribution network helps ensure that critical timing requirements are satisfied.

Again, it will be necessary to use a primitive that implements a clock mux using the available clock resources. The primitive “BUFGCTRL” can be configured in multiple ways in order to achieve different types of clock multiplexor [13].

Usually, it is recommended to use synchronous mux for clocks, which means that the output clock will change its frequency in the precise instant that will not produce a glitch. However, this requires both input clocks to be running during the change and more important, during initialization. This is not the case of Falcon, where the external clock will not always be available. If the external clock is not available during initialization, the pins clock domain will not have any driving clock. For this reason, it is not possible to implement a synchronous multiplexor.

The selected configuration is an asynchronous multiplexor, which does not prevent from glitches but allows to change the source clock at any time, being the other active or not. To avoid the danger of glitches, the firmware will take care of when it is secure to change the select signal of the multiplexor. The connection of the primitive to implement the desired clock multiplexor can be seen in Figure 26.

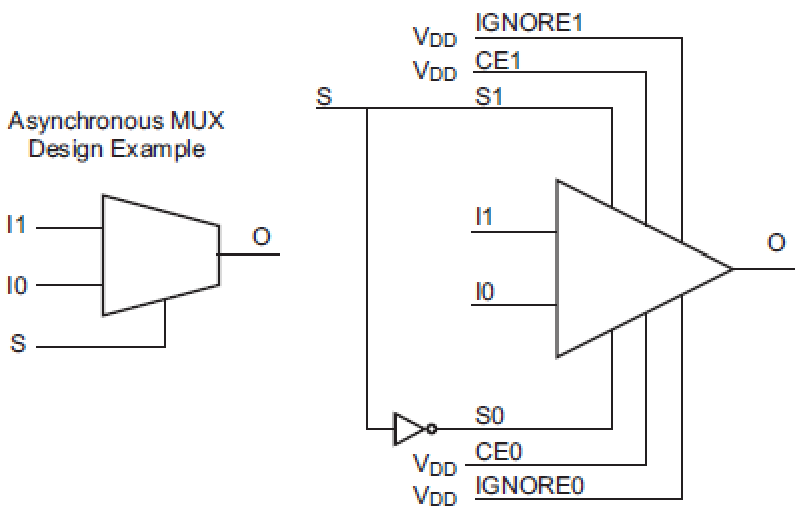


Figure 26 BUFGCTRL primitive

Regarding the modules composing the IP, there is a top module containing the instantiation of the primitive and a module containing a register for setting the select signal of the mux, again using the template provided by Xilinx.

The I/O signals of the top module are:

- **clk_in_1:** input clock signal. It is the first source clock.
- **clk_in_2**: input clock signal. It is the second source clock.
- **clk_out:** output signal. It is the output of the multiplexor.

The I/O signals of the top module are:

- **AXI MM interface signals**: s00_axi_aclk, s00_axi_aresetn, s00_axi_awaddr, s00_axi_awprot, s00_axi_awvalid, s00_axi_awready, s00_axi_wdata, s00_axi_wstrb, s00_axi_wvalid, s00_axi_wready, s00_axi_bresp, s00_axi_bvalid, s00_axi_bready, s00_axi_araddr, s00_axi_arprot, s00_axi_arvalid, s00_axi_arready, s00_axi_rdata, s00_axi_rresp, s00_axi_rvalid, s00_axi_rready.

- **select_clock:** single bit signal to select which clock will be used for the pins clock domain. It is one bit written in a register.

3.5. Enable and trigger controller

At this point, it is necessary an IP controlling the exact clock cycle when the transmissions start. This IP will provide both the enable and trigger signals for the IPs “Stream data to Pins” and “Pins to Stream Data.

The enable signal is a one-bit signal that will be generated and allow the transmission to start, but by itself it does not mean that the transmission starts. For that it is necessary the trigger conditions to occur and therefore the trigger signal to be asserted. The normal procedure is to assert the enable signal and then wait for the trigger signal to be asserted too. But it can also happen the case in which the trigger signal is already asserted and enabling the enable signal starts the transmission. In any case, it is important to understand that the combination of these two signals start the transmission (either a unique transmission or a pattern).

Both for the trigger and for the enable, it is necessary to generate two signals, one for the IP “Stream Data to Pins” and other for “Pins to Stream Data”, as these two IPs can operate independently for send and receive operations. For generating this signals it is necessary to use some configuration registers. Again, in this IP, there will be a module which provides registers and the logic to read and write them from the firmware using the AXI Memory Mapped interface.

The enable signals will be set depending on the value of a unique register that the firmware can write. There are 4 possible states:

- **Both enable signals deactivated:** this is the default state and will be applied if the value of the enable register is 0.
- **Read enable signal asserted and write enable signal deactivated:** for only read operations. This state is achieved when the register value is 1.
- **Write enable signal asserted and read enable signal deactivated**: for only write operations. This state is achieved when the register value is 2.
- **Both enable and write signals asserted:** used for simultaneous read and write operations. In this case the value of the register will be 4.

Regarding the trigger signals, their generation will be more complex. It will need the captured value of the input pins, apart from some registers for configuration. In addition, the trigger signal does not need to be active during all the transmission, as it only sets the start of it. Being active for one cycle is enough for the IPs "Stream data to pins" and "Pins to stream data" to start their transmissions.

There are different types of trigger. The trigger can be an edge, in which we can differentiate between positive and negative edge; or it can be a level, which can be high or low level. Different pins can be configured to produce the trigger in different conditions, and they can be configured to not take part into the trigger condition.

In order to take into account all these possibilities, it is necessary to use several registers, containing all the configuration options for the 32 pins. These registers are:

- **read_trigger_select_pattern_edge**: for the read trigger signal, selects which pins will produce a trigger with a level (the 32 bits will create a "pattern") and which will produce the trigger with an edge. The 32 bits of the register are directly related with the 32 GPIOs. Writing a 0 means the trigger is level active, while writing a 1 means edge active.
- **write_trigger_select_pattern_edge**: the same information than the previous register but for the write trigger signal.
- **read_trigger_enable**: for the read trigger signal, selects which pins are enabled to produce a trigger. Again, the 32 bits of the register are directly related with the 32 GPIOs. Writing a 0 means the pin is not enabled to produce a trigger, while writing a 1 means the contrary.
- **write_trigger_enable**: the same information than the previous register but for the write trigger signal.
- **read_trigger_pattern**: for the pins that have been configured to produce the read trigger with a level, configures this level. It can be seen as a pattern created by the 32 bits of the register.

- **write_trigger_pattern**: the same information than the previous register but for the write trigger signal.
- **read_trigger_edge_type**: for the pins that have been configured to produce the read trigger with an edge, configures if the edge is positive (1) or negative (0).
- **write_trigger_edge_type**: the same information than the previous register but for the write trigger signal.

The logic producing the trigger signal will be all combinatorial, to allow our system to start any operation only one cycle after the trigger conditions are met. However, to detect the edges it will be necessary to register the input pins, so this will be the only sequential logic.

Each of the input signals coming from the ip “Pins Controller” will be registered to store its previous state at any moment. This delayed signal can be compared with the current signal to see if an edge has been produced.

The following circuit acts as an edge detector, providing an output which is asserted when a positive edge is produced, and another output asserted when the edge is negative. The trigger circuit for a single bit is represented.

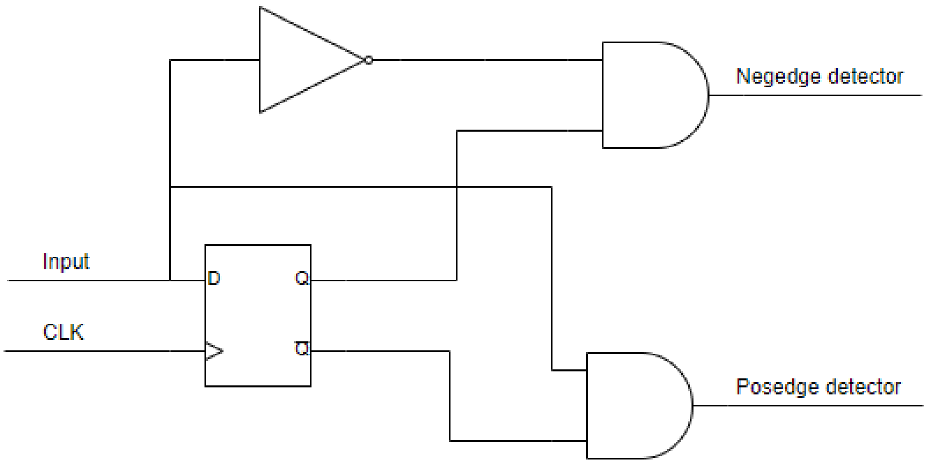


Figure 27 Edges detection

Once the information about edges is obtained, the rest of the logic is pure combinational. The edge information is passed through two “and” gates, one for the positive edge and another for the negative. The or combination of these two outputs will go to another and gate which selects between edge or pattern.

On the other hand, for the trigger level, the input goes to a xnor gate, whose output will be asserted if the two inputs are equal, meaning the pin and the selected trigger level for that pin are the same. As before, this output will pass through a and gate for selecting trigger level or edge.

Finally, the level output and the pattern output pass through an or gate and here we obtain the trigger signal for a single bit.

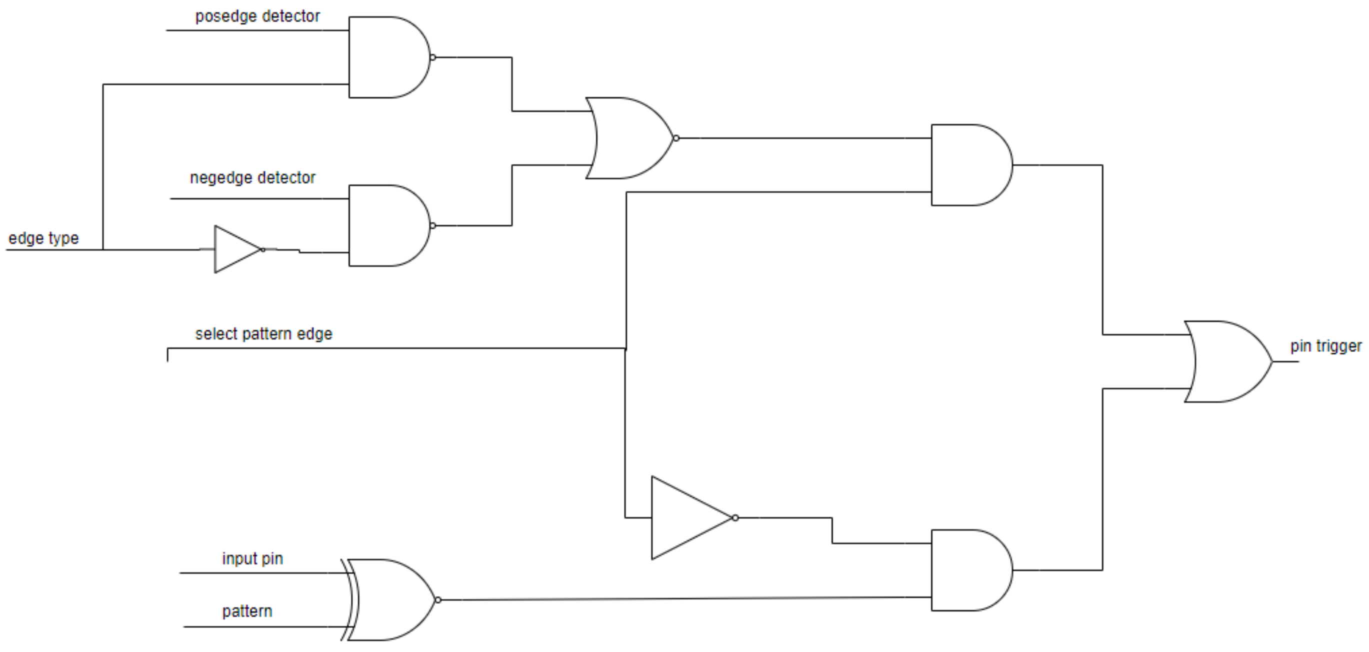


Figure 28 Trigger combinational logic

All the trigger signal for a single bit will go to an “or” gate, in order to consider only those triggers pins that have been enabled. Inverting the enable signal going to the or gate, the output will be a 0 only when the enable signal is enabled and the trigger is not asserted. All the other cases will produce a 1 at the output.

After this step, a “and” operation is performed with all the trigger signals, so if any of them is 0, the output will be 0. This is the reason for obtaining always 1 at the output of the or gates, except when the trigger should happen for a specific pin and it has not happened.

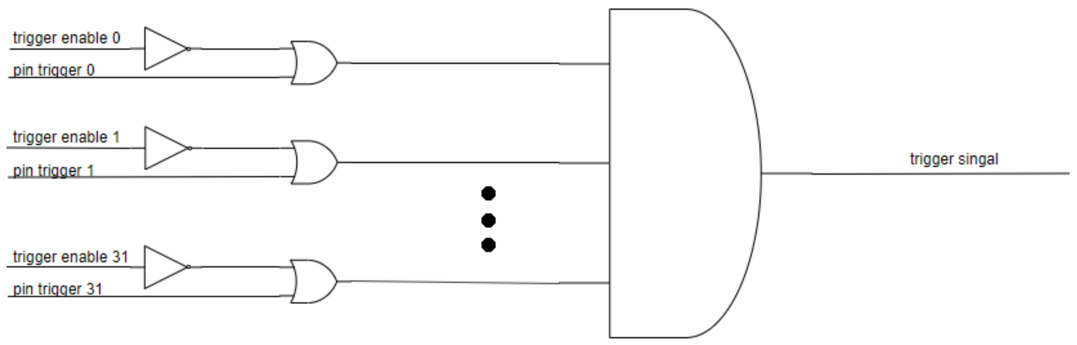


Figure 29 Join all triggers

Producing the trigger this way, the disabled pins for trigger will not stop the final trigger signal to be asserted. And if all the pins are disabled for trigger, it will be always active, so that our system can continue working. In that case, the enable signal will start the transmission operations automatically.

The signals of the two modules of this IP are:

myip_enable_trigger_controller_v1_0_S00_AXI

- **AXI MM interface signals**: s00_axi_aclk, s00_axi_aresetn, s00_axi_awaddr, s00_axi_awprot, s00_axi_awvalid, s00_axi_awready, s00_axi_wdata, s00_axi_wstrb, s00_axi_wvalid, s00_axi_wready, s00_axi_bresp, s00_axi_bvalid, s00_axi_bready, s00_axi_araddr, s00_axi_arprot, s00_axi_arvalid, s00_axi_arready, s00_axi_rdata, s00_axi_rresp, s00_axi_rvalid, s00_axi_rready.

- **Enable register**: contains the value for the desired enable option, explained at the beginning of this section.

- **Trigger registers** (explained previously): read_trigger_select_pattern_edge, write_trigger_select_pattern_edge, read_trigger_enable, write_trigger_enable, read_trigger_pattern, write_trigger_pattern, read_trigger_edge_type, write_trigger_edge_type.

myip_trigger_controller_v1_0 (top module)

- **AXI MM interface signals**
- **input_pins:** captured value of pins, at a frequency set by pins_clk.
- **pins_clk**: clock signal of the pins clock domain
- **pins_rst**: reset signal of the pins clock domain
- **read_trigger**: output signal. Trigger for read operations.
- **write_trigger**: output signal. Trigger for write operations.
- **read_enable:** : output signal. Enable for read operations.
- **write_enable**: output signal. Enable for write operations.

4. Firmware

This section deals about the design of the firmware part of Falcon. Recapitulating up to this point, we have analysed the design of the hardware part of Falcon. It is composed of a DMA, some IPs to send and capture the data to/from the pins and other IPs which control how these transactions are done. As shown in each of these IPs, almost all of them need some configuration which is provided through registers that can be written by the user.

The platform in which the Falcon project is developed, contains both an FPGA with programmable logic and an embedded ARM microprocessor. This microprocessor allows to design certain functionalities in a very simple way. Using the C language, it is possible to write scripts for controlling all the Xilinx IPs and peripherals using the provided APIs, and it is possible to write the registers of the custom IPs in a very simple and transparent way.

In this project the firmware has different functions. Some of the most important are:

- Controls the USB connection, creating the endpoints and providing the required configuration and internal USB functionality, everything through the API provided by Xilinx. This is needed for the communication with the PC.
- Controls and map the DDR memory which is present in the evaluation kit. The PS allows all the resources to access to this memory.
- Configures and controls the DMA IP. This IP needs to be configured in order to provide information about which data is going to send. It also produces interrupts that need to be handled
- Writes the registers of the custom IPs, in order to provide all the necessary configuration. This configuration can be the length of a pattern, the clock used for the pins, the type of trigger…
- In general, controls the flow of the different tasks commanded through the USB from the Python driver.

4.1. Firmware functionality and flow

In this section, the different functionalities of the firmware will be explained. In addition, the flow between them will be analysed.

The functionalities that the firmware implement include the orders that can be commanded from the Python driver and the initialization of the platform itself.

4.1.1. USB initialization

This functionality configures the USB interface in order to communicate with the PC. The configuration includes, among other tasks, the creation of endpoints, the device descriptor and configuring the interruptions and writing the handler functions.

A USB system is a unidirectional communications hierarchy consisting in a single USB host and a number of devices from 1 to 127. The host controls all communications within the system, it sends communication requests and Devices respond to these requests. USB devices do not initiate data transfer events [14].

Transferring data involves the host reading and writing to a set of memory locations located on each Device. These memory locations are called Endpoints, which are essentially in and out baskets. Device endpoints are found in numbered pairs, as each endpoint number has an IN and an OUT endpoint. Out endpoints carry data coming “out” of the host, while “in” endpoints contain data being sent to the host.

Every USB device reserves its endpoint 0 as a unique endpoint called the “Control Endpoint”. It contains a description of the USB Device which is read by the host during enumeration. This endpoint allows the host to send configuration commands to the device. The rest of the endpoints can be configured for carry the data transfers.

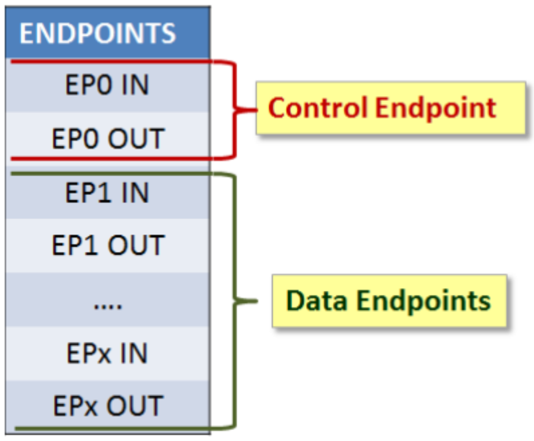


Figure 30 USB Endpoints

In Falcon, two endpoints will be used. The Endpoint 0 for control, as this cannot be chosen, and the endpoint 1 for data transfers. Using the Xilinx USB API it is possible to configure the number of endpoints and its size, for both input and output.

When an endpoint in the device (which will be the evaluation kit) receives a transfer, it rises and interrupt. This interrupt should be attended, and a handle function executed. So, in addition to the endpoints, the USB interrupt handle functions also have to be configured during USB initialization.

There will be two handle functions, one for each endpoint. In the case of the control endpoint, the handle function will be executed when the Falcon is connected to the PC. Therefore, its function is to send the device information in order to establish the connection properly. On the other hand, the function of the endpoint 1 handler function is to obtain the received data and perform the correct operation with it.

The obtained data can be either a command, which will execute a certain task of the firmware, or it can be data that has to be stored in DDR for being send when the corresponding command is received. In any case, the handler function will send an ACK message to the host if the command is going to be executed or the data has been correctly stored. If the system is performing any operation that is not compatible with executing the new command, a BUSY signal will be sent so that the host can try again after some time.

Finally, this function needs to configure the device descriptor and the configuration descriptor, so that the device can be identified and configured in the host PC. The device descriptor is read by the host during enumeration. The purpose of the device descriptor is to inform to the host what specification of USB the device uses to and how many configurations are available on the device. On the other hand, the configuration descriptor informs the host about how many interfaces are in the device and its characteristics. In addition, it informs about how much power the device will consume for every configuration, in order to choose the most appropriated one.

4.1.2. Interrupt System Configuration

As it has been mentioned before, the USB, DMA and the “Stream to Pins” custom IP use interrupts for executing certain functions when an event occurs. For these interrupts to be handled it is necessary to configure the General Interrupt Controller (GIC). The GIC (figure 31) is a centralized resource for managing interrupts sent to the CPUs from the PS and PL. It enables, disables, masks, and prioritizes the interrupt sources and sends them to the CPU in a programmed manner as the CPU interface accepts the next interrupt [15].

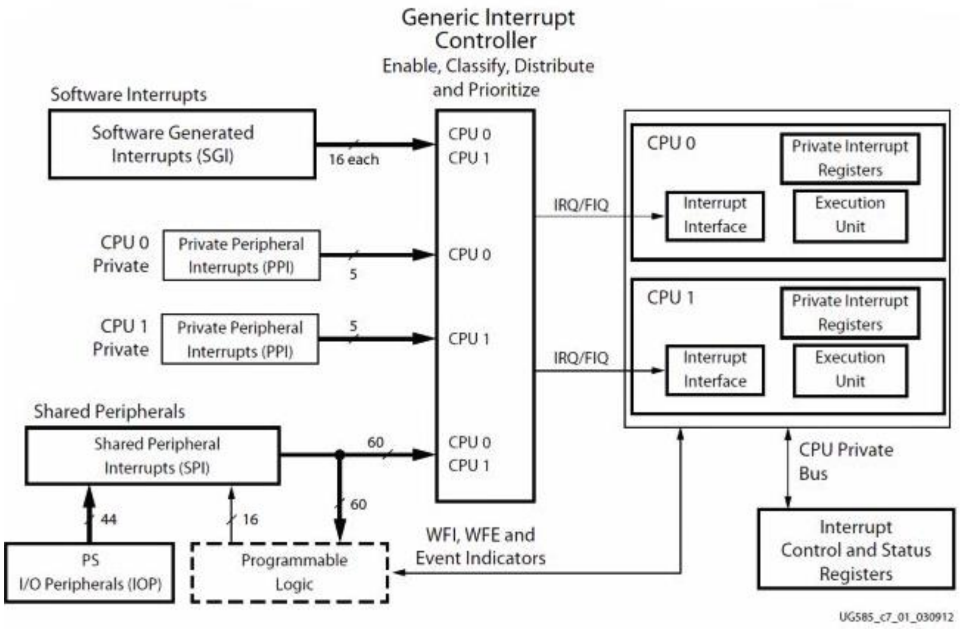

Generic Interrupt Controller
Enable, Classify, Distribute and Prioritize
Software Interrupts
Software Generated Interrupts (SGI)
16 each
CPU 0
CPU 1
CPU 0 Private
Private Peripheral Interrupts (PPI)
5
CPU 1 Private
Private Peripheral Interrupts (PPI)
5
Shared Peripherals
Shared Peripheral Interrupts (SPI)
60
60
44
16
PS I/O Peripherals (IOP)
Programmable Logic
WFI, WFE and Event Indicators
IRQ/FIQ
IRQ/FIQ
CPU 0
Private Interrupt Registers
Interrupt Interface
Execution Unit
CPU 1
Private Interrupt Registers
Interrupt Interface
Execution Unit
CPU Private Bus
Interrupt Control and Status Registers
UG585_c7_01_030912

Figure 31 GIC Diagram

Every interrupt should have an associate handler function, which will be executed when the interrupt is attended. When an interrupt is raised, the processor stops executing its current instructions, saves the state of the execution at that time and executes the handler function associated to the interrupt that it is attending. After completing this function, it restores the state of the execution when it was interrupted and continues with the rest of the instructions. One important detail is that the processor should clear the interrupt in the handler function so that it is not attended again and can be raised when the event happens again.

The interrupts have different priorities, so in case that two are raised at the same time, the processor can choose which one to execute first. Furthermore, in some processors a higher priority interrupt can interrupt the execution of the handler of a lower priority interrupt. This is not the case of out processor, in which the interrupts cannot interrupt each other. It is possible to change this behaviour but for our purpose it is better to leave that way.

In our firmware, the function to configure the GIC performs 4 steps for each interrupt:

1. Sets the trigger type for the interrupt events, which is rising edge sensitive in all of them
2. Configures the priority for each interrupt, which is the same for the initialization process
3. Connects the interrupt with its associated handler so that the processor knows which function to execute when it is produced
4. Enables the interrupt, so that it can start to be attended. This sometimes is not desired because the source of the interrupt is not yet configured and could create an unexpected behaviour (as we will see, this happens with the "end of transfer" interrupt.

The number of interrupts configured for this project are 4: two for indicating then end of the DMA operations (send data from DDR and to DDR), one which is raised when the tx operations has ended, and the last one is the USB interrupt which indicates that the USB has received some data (this interrupt includes the two interrupts for both endpoints).

4.1.3. Configuring memory regions and pointers

Falcon works reading data from the DDR memory and sending it through the GPIO pins and receiving data from the GPIO pins and storing it in DDR memory. For developing these two general operations it needs to maintain a control of which part of the memory is used to store captured data and which part stores the data that is going to be send.

To avoid overlapping problems, the DDR memory has been divided in three regions of data. One is used to store the data that will be sent through the pins when the correct command is executed. This first region will be written by a function that takes the data received from the USB and writes it to this region of DDR memory. This region will be read by the DMA when a tx operation is commanded.

The second region of memory is used to store the captured data. It is written by the DMA when a rx operation is commanded and it is read by a function that sends the data through the USB connection to the PC. Finally, the third region is used to store the buffers of the DMA, which need to be in DDR memory. This region is very little compared to the other two.

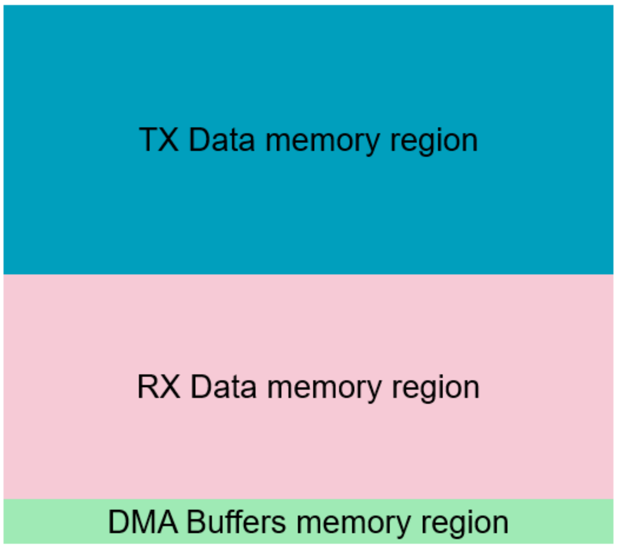


Figure 32 DDR Memory division

By default, and the initial instructions will configure it this way, the tx and rx region of memory are almost equal in size, being a slightly smaller than half of the memory because we need to allocate the DMA buffers region too. This division can be changed by the user after initialization so that the distribution can be best suited for different sets of operations.

It is necessary to highlight that when performing an operation, it is the python driver which will tell the firmware in which memory address is located the data that is going to be sent or in which address the captured data is going to be stored. The firmware cannot distinguish between different sets of data. It can only process an initial address and a pattern length. When using different sets of data (sent or captured), the Python driver will provide a valid memory address.

While executing any tx or rx operation, the firmware needs to use a pointer to indicate the DMA and the functions accessing the DDR memory, from where to read or where to store the data. Falcon use two global pointers to memory. The first one is used to store and send data to/from DDR through the USB connection. The second pointer is used to indicate the DMA where to store the captured data.

These pointers can be configured at any time by the Python driver but they are given an initial value. For the first pointer, which will be called from now on “main memory pointer” this initial value is the first address of the tx region of memory. For the second pointer, which will be called from now on “rx memory pointer”, the initial value is the first address of the rx memory region.

4.1.4. PLL configuration

The PLL provides one of the two option for the frequency of the pins clock domain. It needs to be initialized and configure so that it starts to output a proper clock. The frequency of its output clock can be changed at any time, between a minimum and a maximum value, which in our project is from 1MHz to 100MHz.

To configure the PLL (whose IP is called “clock wizard”), it is necessary to write some registers containing the values of the multiply and divide factors. When changing the output frequency it is necessary to wait until the clock is stable to perform other operations. If the clock does not achieve the desired frequency after some time, an error is produced and it needs to be notified to the user.

The provided API includes functions that can perform these tasks. It is only necessary to include a call to these functions in our code, providing the necessary parameters.

When initializing Falcon, the function to initialize the clock wizard IP is called. And when the user sends a command to change the clock frequency, a function which automatically calculates the value of the multiply and divide factors and configures the PLL is called. It is only necessary to indicate which is the desired frequency. This frequency should be between the limits, otherwise the PLL will not achieve it and an error will be generated.

4.1.5. Pins configuration

The custom IP “Pins Controller” allows to select which GPIO pins act as input pins and which act as output pins. This configuration is set by writing a register. As this IP is a custom IP, the API generated by the tools only provides a function to write registers in the AXI Memory Mapped module.

By default, the value of this register is 0, which means that all the pins act as output pins. During initialization, this configuration is changed to an initial value which configures the GPIOs 0 to 15 as output and the GPIOs 16 to 31 as input. The pins can be configured at any moment by writing this register.

4.1.6. Trigger configuration

The trigger of Falcon has been implemented by writing a custom IP (see previous section). This IP can manage different types of trigger (level, edge) with different polarities for each of the GPIO pins. The configuration for these triggers is provided by writing four 32 bit registers providing individual configuration for each of the 32 GPIO pins:

- Trigger enable: selects which pins will contribute to the trigger (asserted with 1)
- Select pattern edge: when the value is 0 indicates that the trigger will be produced with a level (or pattern). When the value is 1 the trigger will be raised with an edge.
- Pattern: indicates which level will produce the trigger if the level option is selected.
- Edge type: indicates if the edge will be positive or negative

Similarly to the pins configuration, as this IP is a custom IP with an AXI interface, the generated API provides a function to write the registers. In the default configuration, all triggers are disabled, so the enable signal will activate the transfers through the pins.

4.1.7. Manage the different commands

When the user sends a command from the PC using the Python driver, the firmware receives a string which identifies this command. Apart from this string, different commands require different parameters, which will be sent just after the command string. For example, when the configuration of the pins is going to be changed, apart from the specific command, it is necessary to send 32 bits with the new configuration. Different commands use different number of parameters, with different lengths.

One of the most important functionalities of the firmware is to manage the reception and execution of commands. When receiving a new packet, the USB interruption handle function takes the first 32 bits which contains the string identifying the new command. This string is stored in a global variable, so that the current command is accessible to any function.

Once the command has been identified, a function is executed and it will perform all the necessary actions to execute this command. First, depending on which command is, different parameters will be read from the data received in the USB packet. With these parameters, a different list of functions will be executed. When all of them have been completed, the system will be ready to execute a new command.

A slightly different behaviour occurs when the command consist in storing new data in DDR. As the amount of data can be much longer than a USB packet, it will be split in different packets. The first packet will only carry the command string and the amount of data that will be stored. When this command is received, a flag is activated indicating that the next USB packets will carry data that is going to be stored in DDR. When the commanded amount of data has been stored, the command has finished and the flag is deactivated. This technique avoids sending the store command in all USB packets, which can be a lot. It also improves the performance of the transfer.

4.1.8. Configuration and control of DMA

The DMA IP is in charge of all transfers from DDR memory to the GPIO pins and from GPIO pins to DDR memory. Using this resource allows the processor to execute other tasks while the transfer is being performed. As it has been analysed in previous sections, the DMA works using the "scatter gather mode", so that it can take different sets of data from non-contiguous memory regions. To manage the different pieces of data, it creates structures called Buffer Descriptors (BD), which store information about the different transfers that are going to be performed. These BD need a memory region in DDR, and the maximum number of BD will depend on the size of this region (this has been discussed in previous sections).

The BD are grouped in structures called “BD Rings”. One ring is created for tx transactions and another for rx transactions. They manage the individual BD and divides them in four groups:

- Free: These BDs can be allocated a new transaction, but they still need to be configured providing a pointer to memory, a transaction size and some control bits, everything through the API.
- Pre-process: The BDs that have been configured and are still under the API control. The application modifies these BDs through driver API to prepare them for DMA transactions.
- Hardware: The BDs that have been enqueued to hardware. These BDs are under hardware control and may be in a state of awaiting hardware processing, in process, or processed by hardware. The application should not change BDs while they are in this group.
- Post-process: The BDs that have been processed by hardware and have been extracted from the Hardware group. These BDs are under application control and the transfer status can be checked. They can be returned to the Free group

The functions used to send or receive data through pins control the status and performance of the BDs. Depending on the type of operation (only send, only receive) and the size of the data, it may be possible that the BD have to be reused (as the number of BD is limited by the memory region defined for them).

Once the BDs have been sent to hardware, the DMA takes care of its execution and the processor can proceed with other tasks. In the cases when the tx transaction needs more BDs than available so that they have to be reconfigured several times, the processor can reconfigure the finished BD while others are being executed. Correctly configured and using the capacity of the FIFO, this technique allows to provide a continuous data flow for an amount of data that does not fit in the available BD buffers.

In this kind of operations before starting the transmission, all the available BD buffers are configured. Then, when a group of buffers of a selected size have completed its transaction, an interrupt is raised. The processor executes the handler function which checks how many BD can be set free and reconfigures them if more data needs to be transferred.

The same situation occurs for the rx operation, but in this case, there is no need for achieving a continuous data flow, as the received data is going to be stored in DDR and read after the whole operation has concluded. In addition, the data is read from the RX FIFO at a higher rate than it is stored. The only requirement here is that the RX FIFO should never be full, as it would stop new data from being received.

4.2. Available commands

Having seen all the functionality that the firmware implements, we can see how they can be combined to perform different operations that can be commanded from the user side. These operations are:

- **Move main pointer:** This command moves the main memory pointer to a memory address indicated by a parameter. It is important to remember that is the Python driver the one in control of the memory pointers to different sets of data. This a basic operation which needs to be performed before any other command that uses the DDR memory.
- **Move rx pointer:** similar to the previous command, in this operation, the rx memory pointer is moved to another address, given by a parameter sent from the Python driver.
- **Move rx base address:** this command allows to move the separation between memory regions. The tx memory region will always start in the lowest available position, and the rx memory region is located just after. Moving the initial position of the rx region will resize both regions.

- **Store data:** Consists on receiving data through the USB connection and storing it in the DDR memory. The amount of data is indicated in a parameter. It requires the main memory pointer to be moved to a specific memory address, which is given and controlled by the Python driver. As the data that can be received may be longer than a single USB packet, this function asserts a flag that indicates that all the following USB packets only contain data to stored and it will be done automatically. When all the data has been transferred, this flag is automatically deasserted.
- **Send and receive:** This is the basic operation for sending and capturing data. Before executing this command, the main memory pointer need to be moved to the initial position where the data to send is stored and the rx memory pointer to the first position where the captured data will be stored. When executed, this command will configure the DMA to send and capture the amount of data indicated by parameters, and then enables the IPs "Stream data to pins" and "Pins to Stream Data". At this point the read trigger and write trigger signals will start both send and receive operations.
- **Send:** This operation allows to only send data without performing any capture. It configures the DMA for transferring the selected data and enables the IP "Stream data to pins". When the write trigger signal is produced, the transmission starts.
- **Receive:** Complementary to the previous command, it performs a capture operation, given an amount of data. In this case, it only enables the IP "Pins to Stream Data" and waits for the read trigger.
- **Divide send and receive:** This operation works similar to the "send and receive", but it divides the tx data in pieces of a selected size. Each of these pieces will be sent when a write trigger arrives. The same behaviour applies to the capture operation, with every trigger a piece of data of the selected size will be captured.
- **Repeated pattern send and receive:** This command sends the same set of data with every write trigger, for a selected number of repetitions. At the same time, with every read trigger, it captures a selected amount of data, as in the previous command.

- **Read:** When a command using the capture option has been executed, the obtained data has been stored in DDR memory. This command takes this data from memory and sends it through the USB to the PC, for being processed.
- **Pins**: This command is used to configure which pins act as input and which act as output. This information has been received in a 32-bit parameter, indicating the direction of each bit.
- **Select clock:** Selects which clock will be used for the pins clock domain. It uses a parameter which will be used as the select signal in the IP “Clock Mux”.
- **Frequency:** Configures the output frequency of the PLL. This value can be selected from 5MHz to 100 MHz
- **Tx trigger:** Receives four parameters in order to configure the four registers needed for the trigger of the tx operations.
- **Rx trigger:** Receives four parameters in order to configure the four registers needed for the trigger of the rx operations.

4.3. Firmware operation flow

Although there is almost no restriction for using any command at any time, the firmware has been designed following a basic operation flow, which should be followed for obtaining valid results. Due to the importance of this, the Python drivers are design to execute the necessary commands for any operation following the expected order. This operation flow has been represented in the diagram of Figure

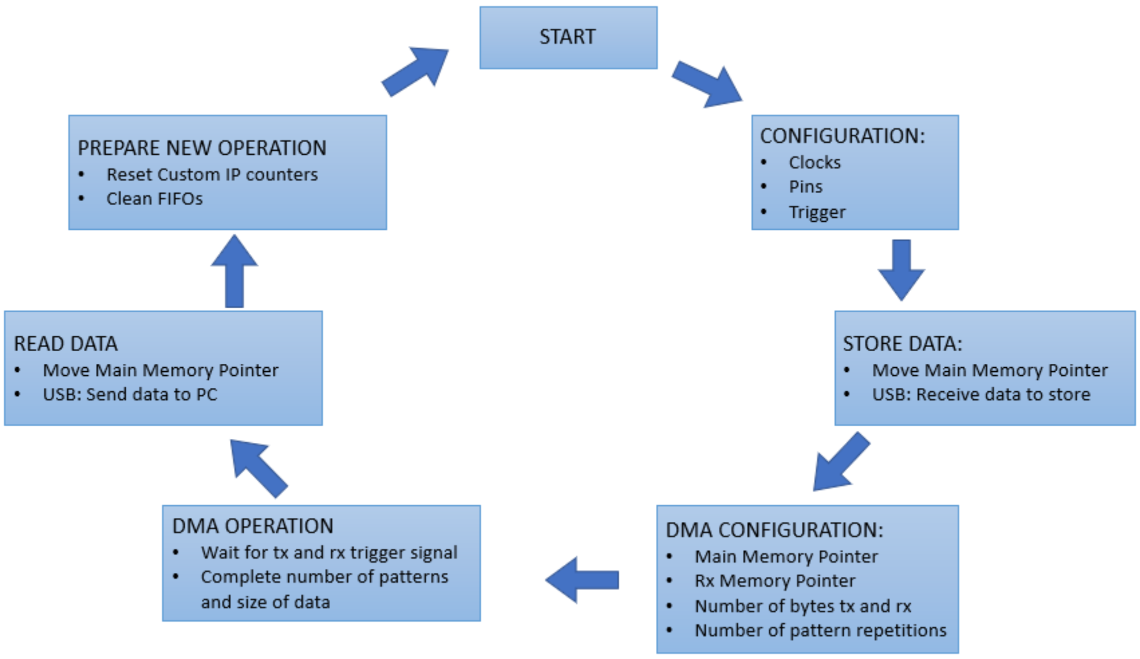

START
CONFIGURATION:
• Clocks
• Pins
• Trigger
STORE DATA:
• Move Main Memory Pointer
• USB: Receive data to store
DMA CONFIGURATION:
• Main Memory Pointer
• Rx Memory Pointer
• Number of bytes tx and rx
• Number of pattern repetitions
DMA OPERATION
• Wait for tx and rx trigger signal
• Complete number of patterns and size of data
READ DATA
• Move Main Memory Pointer
• USB: Send data to PC
PREPARE NEW OPERATION
• Reset Custom IP counters
• Clean FIFOs

Figure 33 Operation flow diagram

The firsts commands that need to be executed are those which configure the clocks, the direction of the GPIO pins and the type of trigger. This is so important that there is a default configuration in case the user do not start with this procedure.

After that, the firmware expects the user to store some data in DDR, although if the operation that is going to be performed is only a receive operation, this step is not necessary. For storing data in memory, the main memory pointer needs to be moves first to indicate the first memory address. Then, the data received through the USB will be stored in DDR.

At this point, the DMA can be configured, providing a pointer to the tx data and a pointer to the memory where the received data will be stored. Apart from the pointers, the characteristics of the operation should be configured: send and/or receive, number of patterns, length of pattern, etc. Once this is done, the DMA operation is executed and the system will wait for the trigger(s).

When all the triggers have been produced and all the patterns have been sent and captured, the DMA operation has finished and the captured data can be read. The main memory pointer is moved to the first address of the captured data and the USB will send this data to the PC.

Finally, some resets in the IPs of the PL are performed in order to restore all the configuration to the default values and leave the system prepared for a new transfer.

5. Python Driver

In this section, the Python part which is running in the PC will be explained. This part consists in a driver and some additional scripts which make possible to control Falcon and send and receive data. The communication between the PC and Falcon is made through USB connection.

The Python scripts use the inputs of the user to generate the list of bytes that will be sent to Falcon using USB. This list of bytes represents the command that is going to be executed, the parameters (if needed) and the necessary data (if any).

The objective of this driver is that the user only uses Python to work with Falcon, being the Verilog and the C part completely transparent.

5.1. USB connection

To use the USB interface, the Python driver relies in the module PyUSB, which provides easy access to the host machine's USB system. It works on any platform running a Python version higher than 2.4. It can be installed using pip [16].

USB devices must be configured through some standard requests after the connection has been established. PyUSB contains methods that can search for a specific USB device, obtain its available configurations, search for its endpoints, set interfaces, etc.

Apart from PyUSB, our PC needs to recognize correctly the USB device we are connecting. As the device descriptor, configuration descriptor and string descriptor have been modified in order to personalize our USB device, the PC will not be able to find a compatible driver from the OS point of view.

The solution is to install libusbK. This is a complete driver/library solution for vendor class USB device interfaces. LibusbK is not an end-user component, it is a solution for situations in which windows can't recognize the USB device, but it needs an additional driver for the device to work.

5.2. Structure

Python is an object-oriented programming language and much of its benefits come from structuring our code creating classes. In the Falcon project, there will be 3 different types of classes: a general Falcon class, a “write data” class and a “read data” class, each of them containing different methods for different functionalities.

5.2.1. Falcon class

This class is used to interface directly with Falcon. It stores different information about Falcon configuration, such as input and output pins, memory region separation, trigger configuration, etc… Its methods create the list of bytes to send to Falcon, using the string associated to every command and the parameters or additional data. They also create the correct configuration strings from the configuration instructions in Python.

The methods of this class are:

- **config_usb**: uses the pyusb module to establish the USB connection
- **get_endpoint**: obtains the endpoint object using the pyusb module. With this object it is possible to send and receive data through USB.
- **receive_usb**: reads the USB buffer in order to receive data. A timeout can be assigned so that an error will be raised if no data has arrived in the selected time
- **store_pattern**: returns an object of the class “WriteDataContainer” (it will be seen afterwards), which automatically stores data in the DDR of Falcon.
- **capture_pattern**: returns an object of the class “ReadDataContainer” (it will be seen afterwards), which automatically sends a command to Falcon to capture a pattern.
- **read_pattern**: returns an object of the class “ReadDataContainer”, but using a flag to avoid capturing the pattern automatically.

- **store_data**: takes a list of bytes that is going to be stored in DDR in Falcon an divides it to be efficiently sent through USB. It takes care of sending first the store command and the parameter with the length of the data.
- **send_data**: Sends command to Falcon to send data through pins. It also sends the parameter with the length.
- **receive_data:** Sends command to Falcon to receive data through pins. It also sends the parameter with the length.
- **read_data**: Sends command to Falcon to read data from the DDR memory. It receives packets of 512 bytes and joins them into a unique list.
- **send_and_receive_data:** Sends command to Falcon to send and receive data through pins. Both operations are executed at the same time, the triggers can be different
- **repeated_pattern_rearm_send_and_receive_data:** Sends command to Falcon to send and receive data through pins. In this case, the send data will be repeated with every trigger, which is automatically rearmed.
- **rearm_send_and_receive_data:** Sends command to Falcon to send and receive data through pins. In this case, the send data will be divided in chunks of a selected size, which will be sent with every trigger activation.
- **move_main_pointer**: Sends command to Falcon to move the main memory pointer to the desired position.
- **move_rx_pointer:** Sends command to Falcon to move the rx memory pointer to the desired position.
- **move_rx_base:** Sends command to Falcon to move the rx first address to the desired position.
- **set_internal_freq:** Sends command to Falcon to set the PLL clock frequency for the pins clock domain.
- **select_clock:** Sends command to Falcon to select between internal and external clock for the pins clock domain.
- **configure_pins**: Sends command to Falcon to configure pins. It receives a list that indicates which pins are output, by default pins are input. With the list, it creates a 32 bit number which indicates the direction of every GPIO pin.

- **get_tx_start_pointer**: Returns a pointer to free memory to store tx data in DDR. This is used when new data has to be sent to Falcon.
- **get_rx_start_pointer**: Returns a pointer to free memory to store captured data in DDR. This is used when new data must be sent to Falcon.
- **set_tx_trigger:** Sends command to Falcon to configure the tx trigger. It uses 4 lists containing information about the pins which produce the trigger at high level, at low level, with a posedge and with a negedge.
- **set_rx_trigger:** Sends command to Falcon to configure the rx trigger. It uses 4 lists containing information about the pins which produce the trigger at high level, at low level, with a posedge and with a negedge.
- **check_trigger_pin:** auxiliar method to check if more than one trigger option has been assigned to any GPIO pin.
- **send_usb:** takes a byte list and sends it using the configured endpoint.

5.2.2. WriteDataContainer class

This class manages all the operations related with the process of storing data in DDR and sending it through the GPIOs. For every set of data that has to be stored and sent, one object of this class will be used. It will store information about the configuration of the pins and the memory pointers.

This class is composed of the following methods:

- **store:** Stores the data in a given list to DDR, after obtaining a pointer to free memory. It codes the data to match the exact distribution of the output pins (the word of data represents a number which has to fit in the output pins distribution, where the most significative and the least has been selected)
- **send_and_receive:** Starts a send and receive transaction. Needs information about the rx pointer and data length. First, it moves the pointers. If the return_data flag is activated, it returns the captured data. If not, it returns an

object of the class “ReadDataContainer”, which can read this data at any moment using the “read” method.

- **repeated_pattern_rearm_send_and_receive:** Calculates the total send and receive length for a repeated pattern with trigger rearm operation and executes this transaction. Like the previous method, first it moves the pointers. If the return_data flag is activated, it returns the captured data.
- **rearm_send_and_receive:** executes a rearm send and receive transaction. It also calculates the tx and rx divisions, moves the pointers and returns the captured data or the “ReadDataContainer” object.
- **send**: This method only moves the main memory pointer and starts a simple send transaction.

5.2.3. ReadDataContainer

This class manages all the operations related with the process of capturing data using the GPIO pins and reading it from DDR memory. For every set of data that has to be captured and read, one object of this class will be used. It will store information about the configuration of the pins.

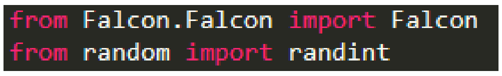
The methods which are available in this class are:

- **receive:** starts a simple receive transaction given an amount of data. It first moves the rx memory pointer.
- **read:** reads from DDR the data associated with the object. Once the data has been received, it decodes it taking into account the configuration of the output pins, to obtain the 32 bit word.

5.3. Example of user usage:

In this last section, an example of usage of Falcon is going to be analysed. This is a typical script that the users of Falcon have to write in order to use it. The example shows the configuration of Falcon for executing a repeated rearm send and receive operation.

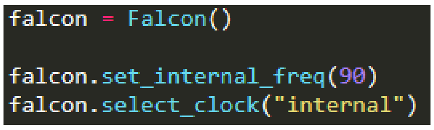
First, the Falcon class is imported, which inside has imported all the necessary modules. The random class will also be necessary for this example.



```
from Falcon.Falcon import Falcon
from random import randint
```

Figure 34 Imports

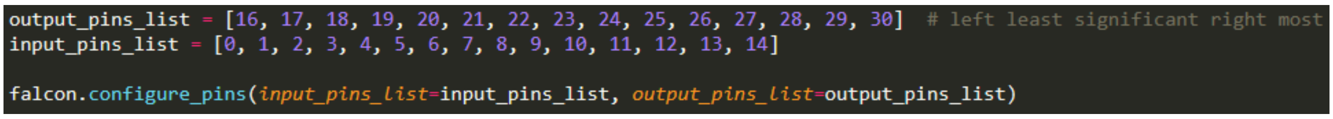
The Falcon object is created and the first action is to set the internal frequency of the PLL. In this case, this is the clock that will be used for the pins clock domain, so it is selected with the "select_clock" method and the parameter "internal", which refers to the PLL (not to the internal fixed frequency clock, at which we have no access).



```
falcon = Falcon()

falcon.set_internal_freq(90)
falcon.select_clock("internal")
```

Figure 35 Clock configuration

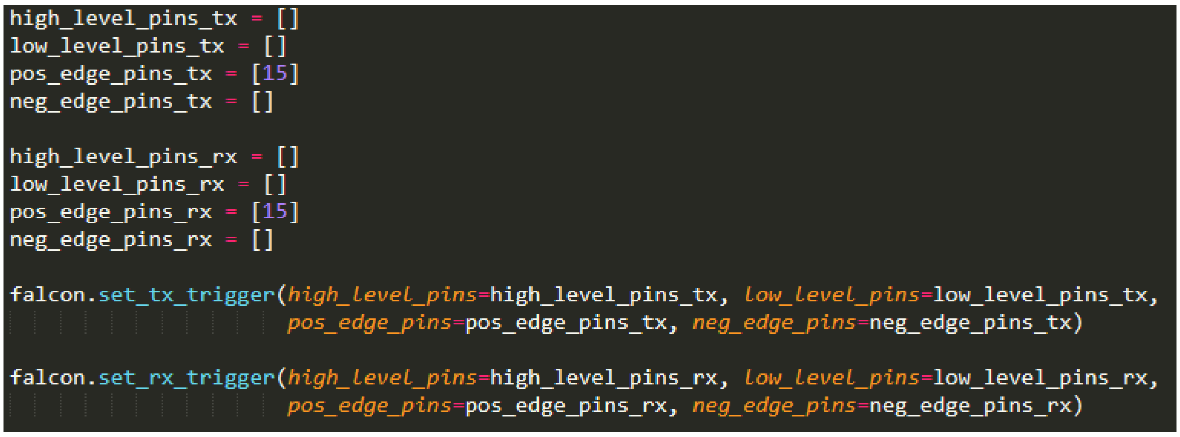
Then, the pins are configured. Two lists containing the GPIO numbers which will be input and output are necessary to use the "configure_pins" method.



```
output_pins_list = [16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]  # left least significant right most
input_pins_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

falcon.configure_pins(input_pins_list=input_pins_list, output_pins_list=output_pins_list)
```

Figure 36 Pins configuration

And finally, the trigger is configured for both tx and rx. In this case, the trigger is the same for both operations. The trigger will, be active when a positive edge appears on GPIO 15.



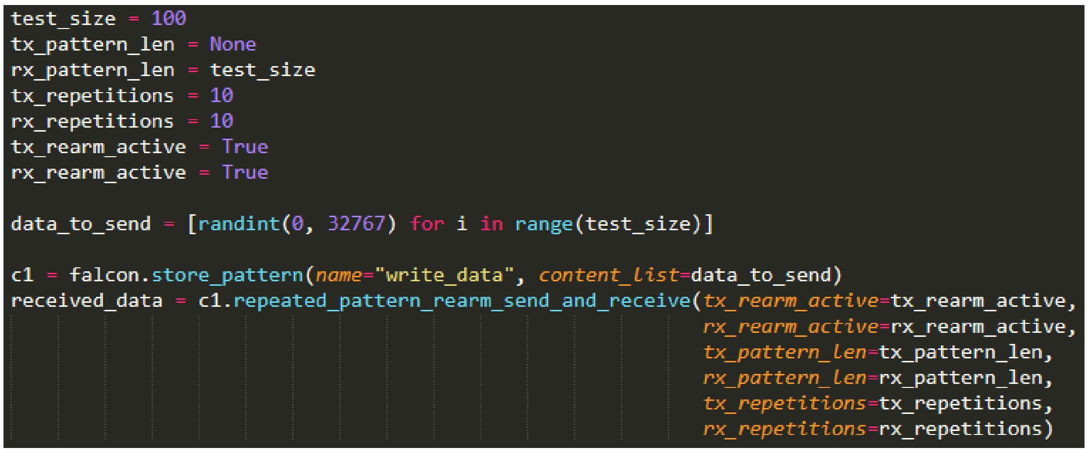
```
high_level_pins_tx = []
low_level_pins_tx = []
pos_edge_pins_tx = [15]
neg_edge_pins_tx = []

high_level_pins_rx = []
low_level_pins_rx = []
pos_edge_pins_rx = [15]
neg_edge_pins_rx = []

falcon.set_tx_trigger(high_level_pins=high_level_pins_tx, low_level_pins=low_level_pins_tx,
                      pos_edge_pins=pos_edge_pins_tx, neg_edge_pins=neg_edge_pins_tx)

falcon.set_rx_trigger(high_level_pins=high_level_pins_rx, low_level_pins=low_level_pins_rx,
                      pos_edge_pins=pos_edge_pins_rx, neg_edge_pins=neg_edge_pins_rx)
```

Figure 37 Trigger configuration

At this point, Falcon has been correctly configured and we are ready to perform any operation sending or capturing data. In this case the operation will send a pattern composed of 100 random words of 15 bits (as we have configured 15 output GPIOs) for 10 times. At the same time and using the same trigger, patterns of the same size will be captured, another 10 times.

First, the data of the tx pattern has to be stored using the method "store_pattern", from the flacon class. This method returns a object of the class "WriteDataContainer", which is related with this pattern stored in DDR of Falcon.

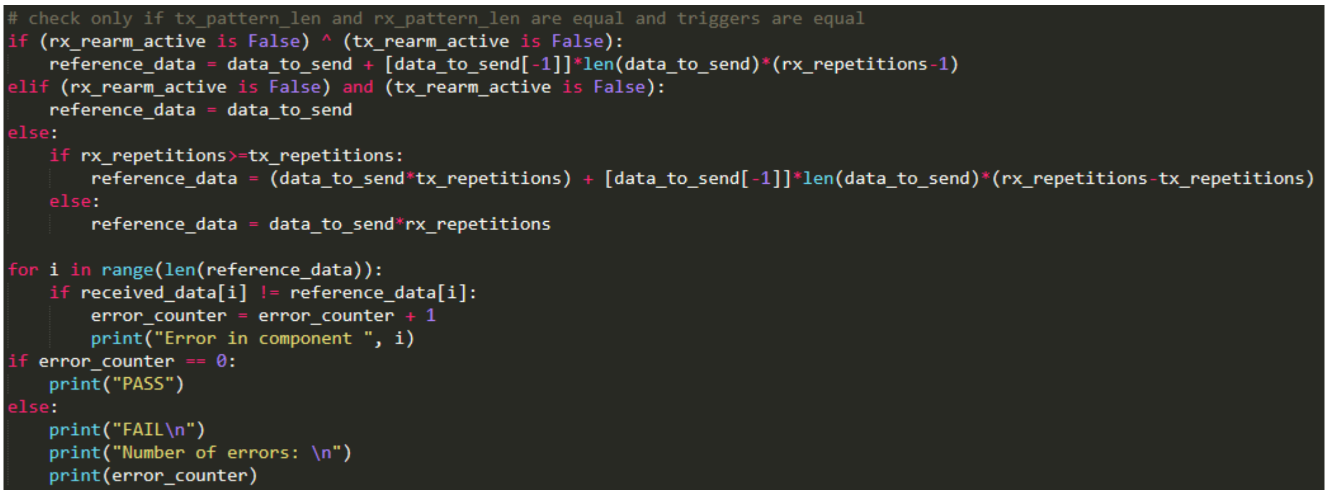
Finally, the "repeated_pattern_rearm_send_receive" method executes the desired operation and returns the list of read words. Deactivating the return_data flag it can also be configured to return an object of the class "ReadDataContainer", which can read this data from DDR at any moment using the "read" method.



```
test_size = 100
tx_pattern_len = None
rx_pattern_len = test_size
tx_repetitions = 10
rx_repetitions = 10
tx_rearm_active = True
rx_rearm_active = True

data_to_send = [randint(0, 32767) for i in range(test_size)]

c1 = falcon.store_pattern(name="write_data", content_list=data_to_send)
received_data = c1.repeated_pattern_rearm_send_and_receive(tx_rearm_active=tx_rearm_active,
                                                           rx_rearm_active=rx_rearm_active,
                                                           tx_pattern_len=tx_pattern_len,
                                                           rx_pattern_len=rx_pattern_len,
                                                           tx_repetitions=tx_repetitions,
                                                           rx_repetitions=rx_repetitions)
```

Figure 38 Send and receive operation

In this case, the input pins are directly connected with the output pins, so the received data should be the exact same data that has been sent. Additionally, at the end of this test, a checker has been created to verify that the send data and the captured data are the same.



```
# check only if tx_pattern_len and rx_pattern_len are equal and triggers are equal
if (rx_rearm_active is False) ^ (tx_rearm_active is False):
    reference_data = data_to_send + [data_to_send[-1]]*len(data_to_send)*(rx_repetitions-1)
elif (rx_rearm_active is False) and (tx_rearm_active is False):
    reference_data = data_to_send
else:
    if rx_repetitions>=tx_repetitions:
        reference_data = (data_to_send*tx_repetitions) + [data_to_send[-1]]*len(data_to_send)*(rx_repetitions-tx_repetitions)
    else:
        reference_data = data_to_send*rx_repetitions

for i in range(len(reference_data)):
    if received_data[i] != reference_data[i]:
        error_counter = error_counter + 1
        print("Error in component ", i)
if error_counter == 0:
    print("PASS")
else:
    print("FAIL\n")
    print("Number of errors: \n")
    print(error_counter)
```

Figure 39 Checker

6. Results and ampliations

This project provides a new flexible evaluation tool which covers situations where accurate control over the signals is required. It can provide very complex patterns and trigger configurations for IC evaluation, being especially useful for ADCs characterization. Falcon makes possible a complete evaluation of the newest ICs reducing the required time for the evaluation process.

The next figures are the captures of the screen of an oscilloscope, showing a simple example of transmission executed in Falcon. In figure 40, signal 3 is the trigger signal, going into Falcon through a GPIO pin. Signal 2 is the clock, which is being feed externally to Falcon. And signal 1 and 0 are the outputs of GPIO pins.

In this case, the trigger of Falcon has been configured to be activated with a posedge of the GPIO pin in signal 3. When the trigger arrives and the edge has been captured by the clock (in the positive edge), at the next clock cycle Falcon starts to send data through the configured GPIOs. In the example, 4 bits are sent through every GPIO pin. As they alternate between 1 and 0 and the value of both signals are always different, this output can be used as a differential clock at half the frequency than the clock going to Falcon.

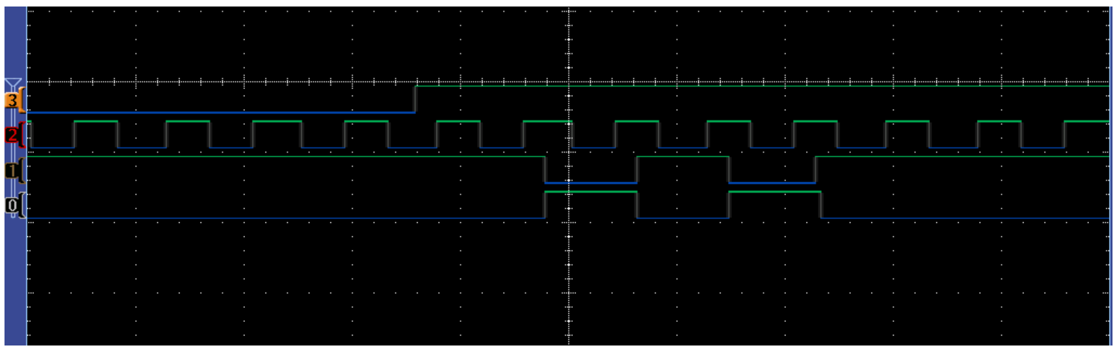


Figure 40 Differential clock triggered with posedge

The transmitted data can be any pattern that the user generates in the PC. At the same time, Falcon can read the values of the input GPIOs.

Another example can be seen in figure 41, where the transmitted pattern is the same that was send before but this time it is repeated 4 times and using a trigger active with the negedge of the configured GPIO pin.

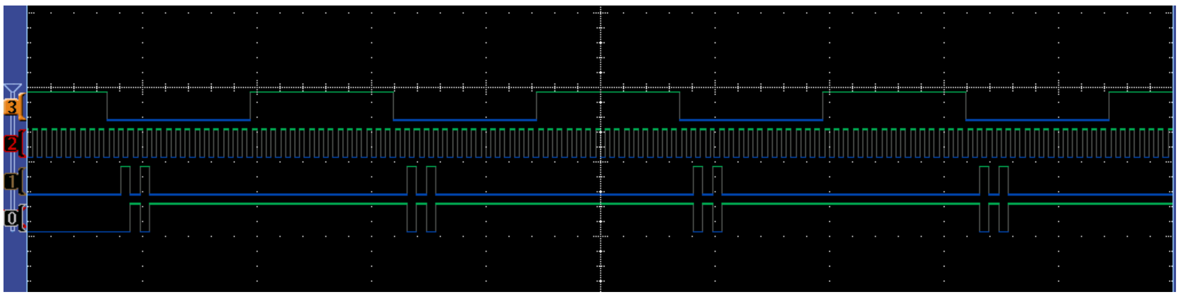


Figure 41 Patterns triggered with negedge

With the current implementation, Falcon can work transmitting data continuously at a rate of 95 MHz. If the data is not continuous and the packets are short (tens of bits) the rate can be much higher, around 150 MHz, where some integrity signal problems start to appear. As mentioned in previous sections, the limit is the size of the tx FIFO.

The first idea to continue developing this project is to address the timing problems that appear when the frequency of the internal fixed clock is incremented further than 100 MHz. Using some directives for the implementation process has helped solving some of the paths that did not achieve the timing objectives, but still some research and changes in the rtl code are needed to overcome all the problems.

Incrementing this frequency will allow to increment the maximum frequency of the pins clock domain for continuous data transmission. As the tx FIFO would be filled faster, the data going to the pins from the tx FIFO can do it at a higher rate without the FIFO becoming empty.

Another possible and very interesting ampliation consist in duplicate the tx and rx circuit which includes DMA, FIFOs, Stream to Data and Data to Stream IPs and Enable and trigger controller IPs. This duplication would allow to have two different channels that can work individually, with different triggers, different enables, etc. In this case, the pins would need to be shared and an IP for controlling this access would be needed.

This ampliation can be very useful in cases when two different patterns need to be sent at the same time or with two different trigger signals. Using the current design with only one channel, only one pattern can be configured with one trigger, but using two channels, is it possible to send a pattern when a posedge arrives, and a totally different pattern when a negedge appears in the trigger signal.

7. Bibliography

[1] Xilinx. Zynq-7000 SoC Data Sheet.
www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[2] Xilinx. Processing System 7 v5.5
www.xilinx.com/support/documentation/ip_documentation/processing_system7/v5_5/pg082-processing-system7.pdf

[3] Xilinx. AXI Interconnect.
www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf

[4] Xilinx. Processor System Reset Module.
www.xilinx.com/support/documentation/ip_documentation/proc_sys_reset/v5_0/pg164-proc-sys-reset.pdf

[5] ARM. AMBA® 4 AXI4-Stream Protocol.
static.docs.arm.com/ihi0051/a/IHI0051A_amba4_axi4_stream_v1_0_protocol_spec.pdf

[6] Xilinx. AXI DMA v7.1.
www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf

[7] EEjournal. How Does Scatter/Gather Work?. www.eejournal.com/article/20170209-scatter-gather/

[8] Xilinx. Header file of driver API for the AXI DMA engine
github.com/Xilinx/embeddedsw/blob/master/XilinxProcessorIPLib/drivers/axidma/src/xaxidma.h

[9] Texas Instruments. FIFO Architecture, Functions, and Applications.
www.ti.com/lit/an/scaa042a/scaa042a.pdf

[10] Xilinx. AXI4-Stream FIFO.
www.xilinx.com/support/documentation/ip_documentation/axi_fifo_mm_s/v4_1/pg080-axi-fifo-mm-s.pdf

[11] Xilinx. 7 Series FPGAs SelectIO Resources.
www.xilinx.com/support/documentation/user_guides/ug471_7Series_SelectIO.pdf

[12] Xilinx. Clocking Wizard v6.0.
https://www.xilinx.com/support/documentation/ip_documentation/clk_wiz/v6_0/pg065-clk-wiz.pdf

[13] Xilinx. 7 Series FPGAs Clocking Resources.
https://www.xilinx.com/support/documentation/user_guides/ug472_7Series_Clocking.pdf

[14] Microchip. Universal Serial Bus. https://microchipdeveloper.com/usb

[15] Xilinx. GIC. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842191/GIC

[16] PyUSB. PyUSB 1.0. https://github.com/pyusb/pyusb

8. Appendix 1. Interposer board

The evaluation kit used in the design of Falcon can be connected to some boards which provide connections for the FPGA pins and provide power. In the project of Falcon, there are some special needs, which require the design of a custom interposer board.

These requirements are:

- Port to connect the MicroZed evaluation kit.
- Connection of GPIO pins through a simple and accessible connector of 32 pins.
- Connection of GPIO pins using a proprietary connector, which is used in other proprietary boards
- Use of two SMA connectors to feed clocks to the FPGA.
- Use of 1 SMA connector to output the clock of the pins clock domain.
- The possibility to select the high level voltage of the digital signals, using level shifters.
- I/O buffers for future implementations of the I2C protocol.
- A voltage reference circuit of 3.3V.
- A power circuit which provides enough current for the FPGA

The 3.3 V reference circuit uses a ADP1706 CMOS linear regulator to achieve this output. It has a test point so that the user can check its output at any moment

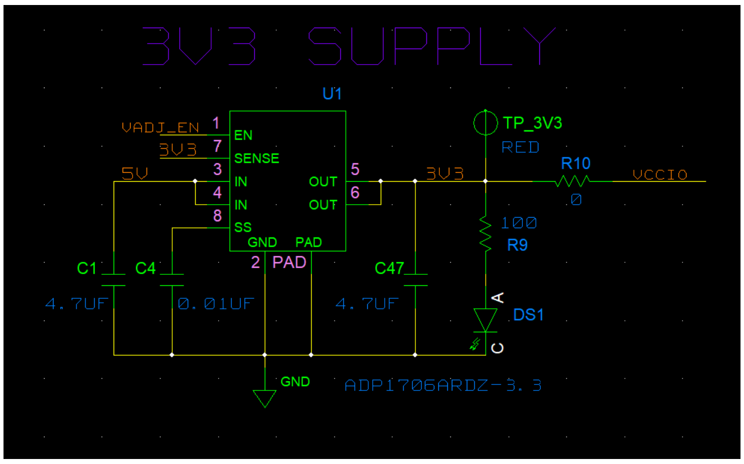


Figure 42 Voltage reference circuit

A set of jumpers allow to select the voltage level for the GPIOs 0 to 15, 16 to 31 and the experimental I2C and SPI interfaces.

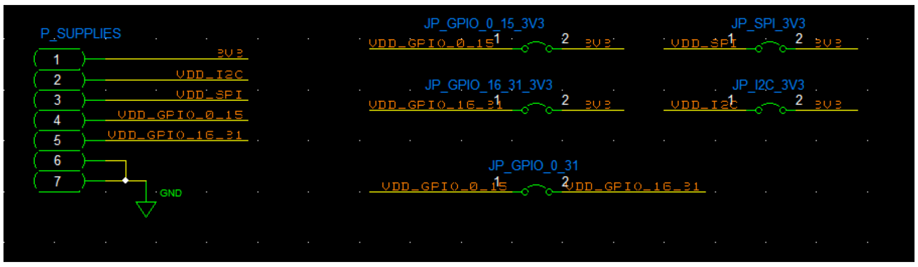
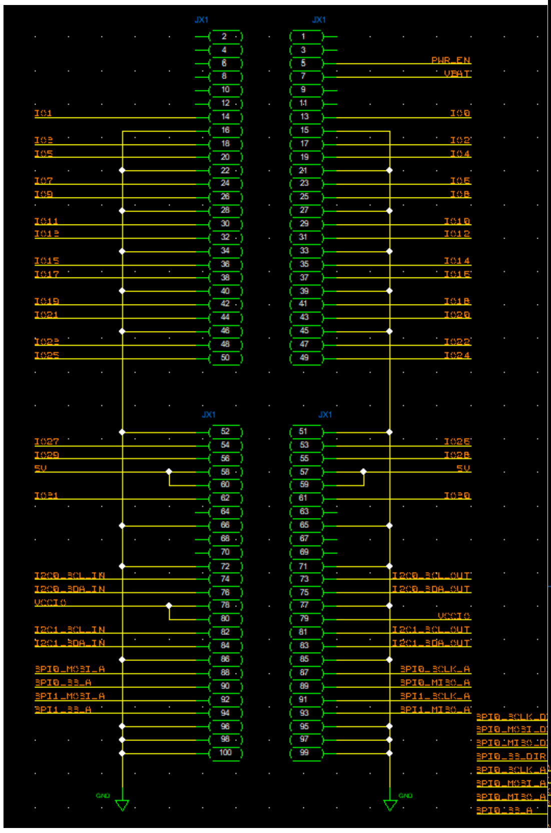


Figure 43 Voltage selectors

The connection to the MicroZed evaluation kit is done through two ports of 100 pins. These ports connect the pins of the FPGA with the interposer board and include the power supply and enable signals from the interposer board to the evaluation kit. The main part of the pins of the second connector are used to configure the direction of the level shifters. As it can be seen in figure, some of these ports are connected to SMA connectors, which are in charge of transporting clock signals. The selected pins are those who are connected to clock resources in the FPGA and therefore are capable of carrying this kind of signals.



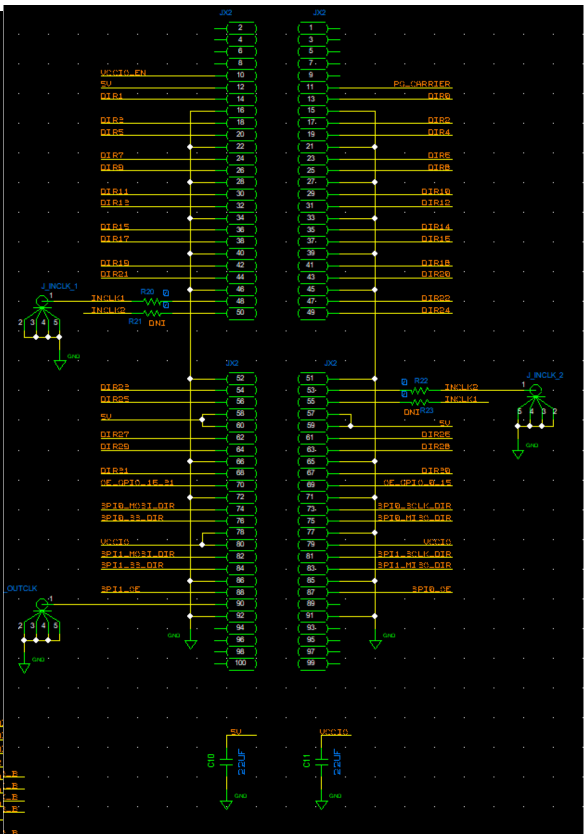


Figure 44 Microzed connection port 1

Figure 45 Microzed connection port 2

Regarding the level shifters, the SN74AVC4T774 level shifter, from TI is used. It needs one signal to determine the direction of the voltage level conversion, which comes from one pin of the FPGA, as seen before. Each of this level shifter can handle up to 4 signals, so 8 will be necessary in our design.

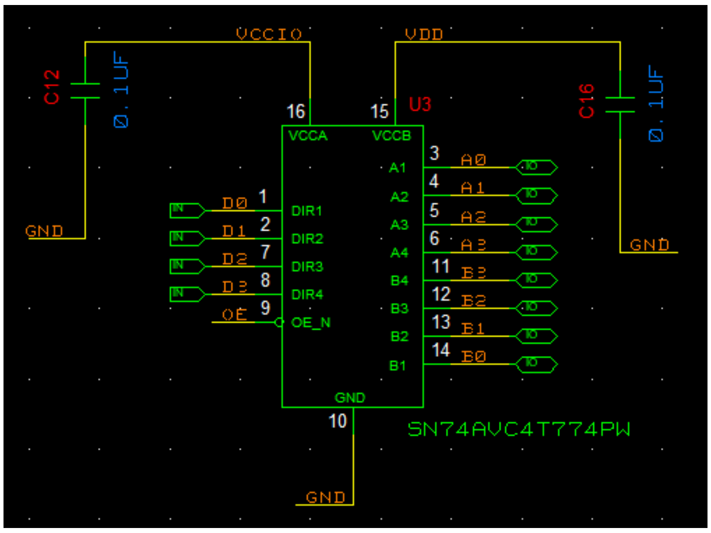


Figure 46 Level shifters

It has been mentioned that there are some pins dedicated to I2C connections. As the SCL and SDA components of this protocol are bidirectional, it is necessary to implement I/O buffers before the level shifter. The buffers used are the SN74LVC2G125DCU, from TI. The schematic, shown in figure 47, also includes pull-up resistors.

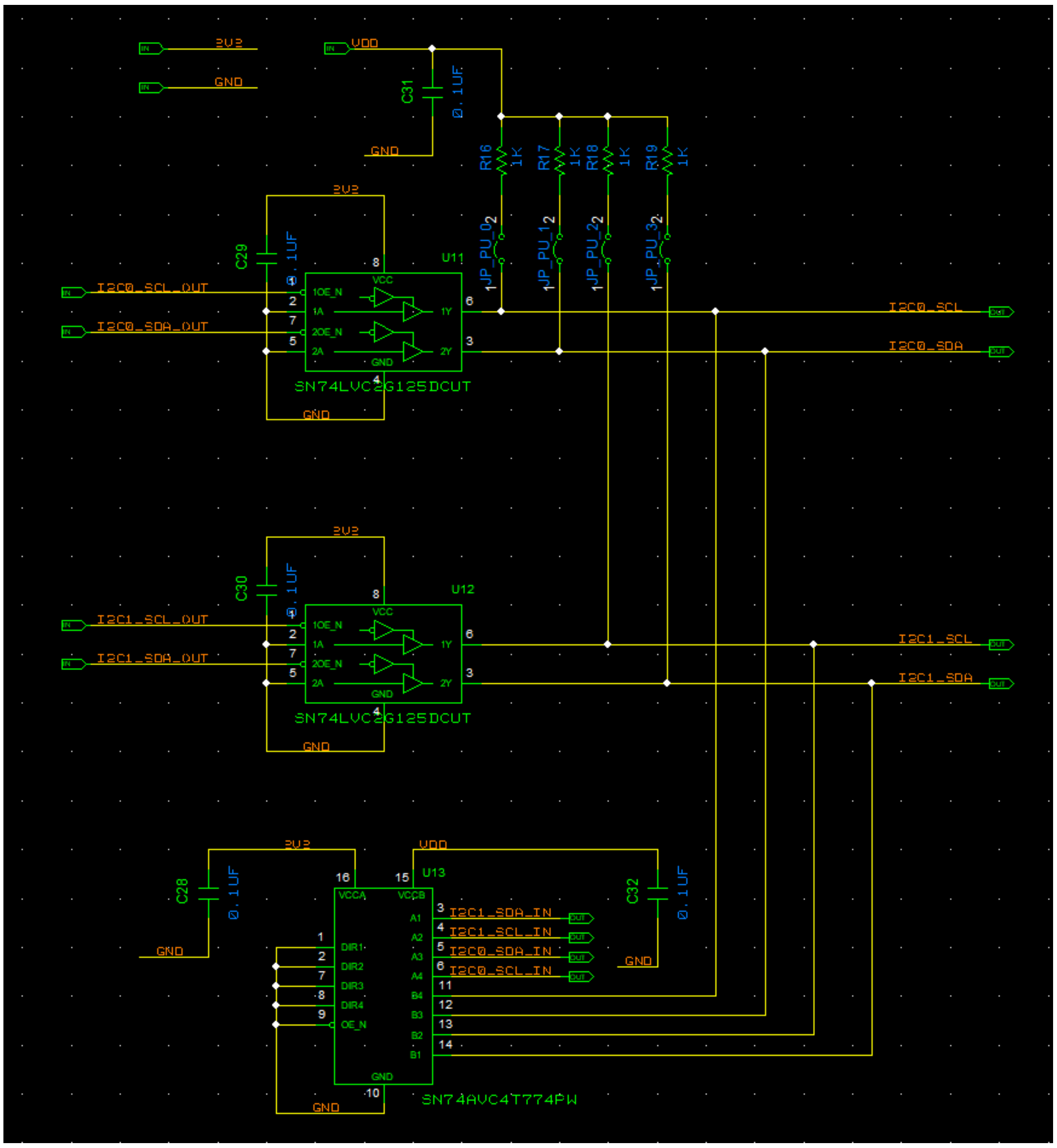


Figure 47 I2C connections

All the GPIO signals, after passing through the level shifters, go to two connectors, the simple GPIO connector and the proprietary connector. The experimental I2C and SPI signals only go to the proprietary connector. The schematic is shown in figures 48 and 49

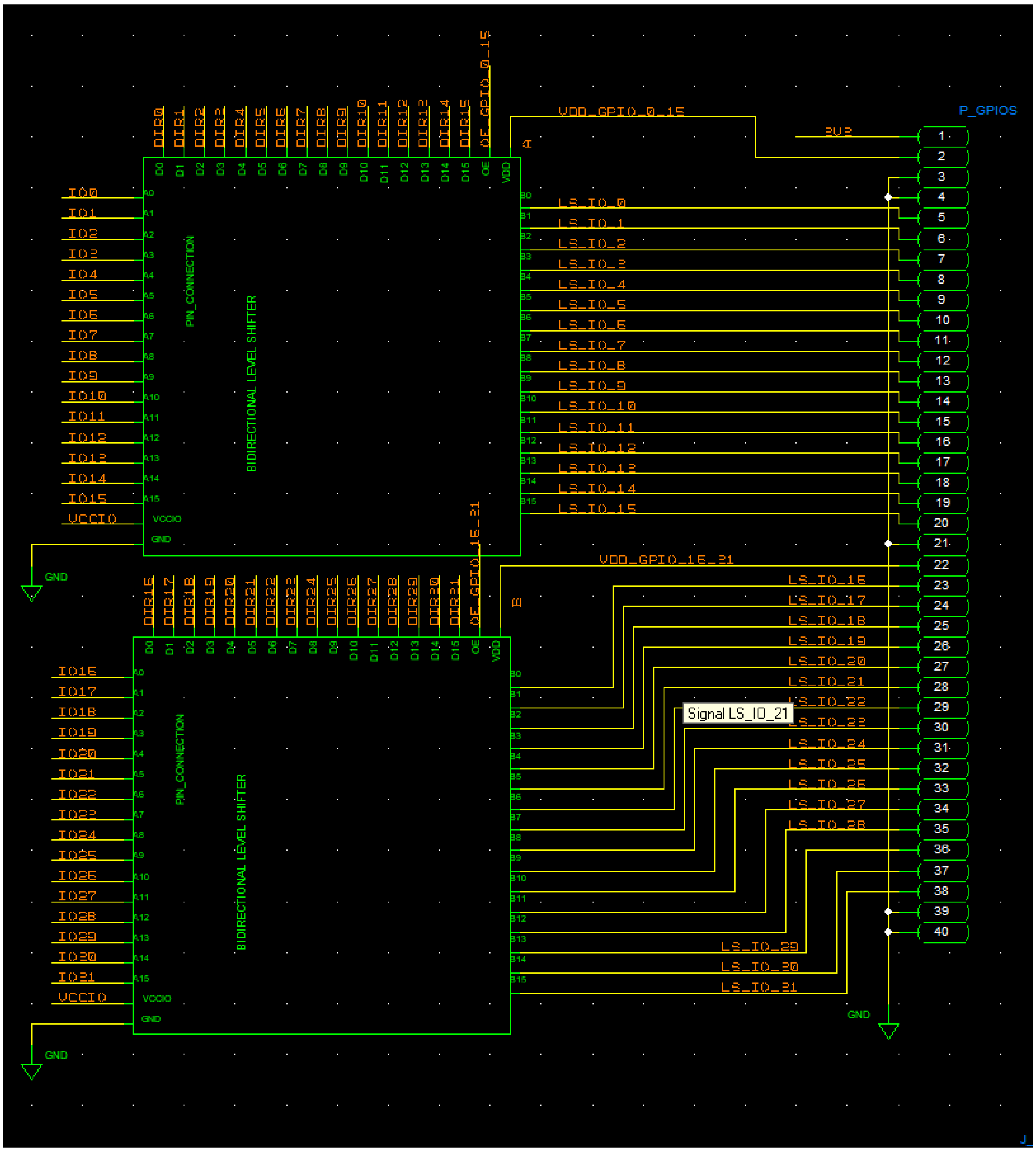


Figure 48 GPIO connector

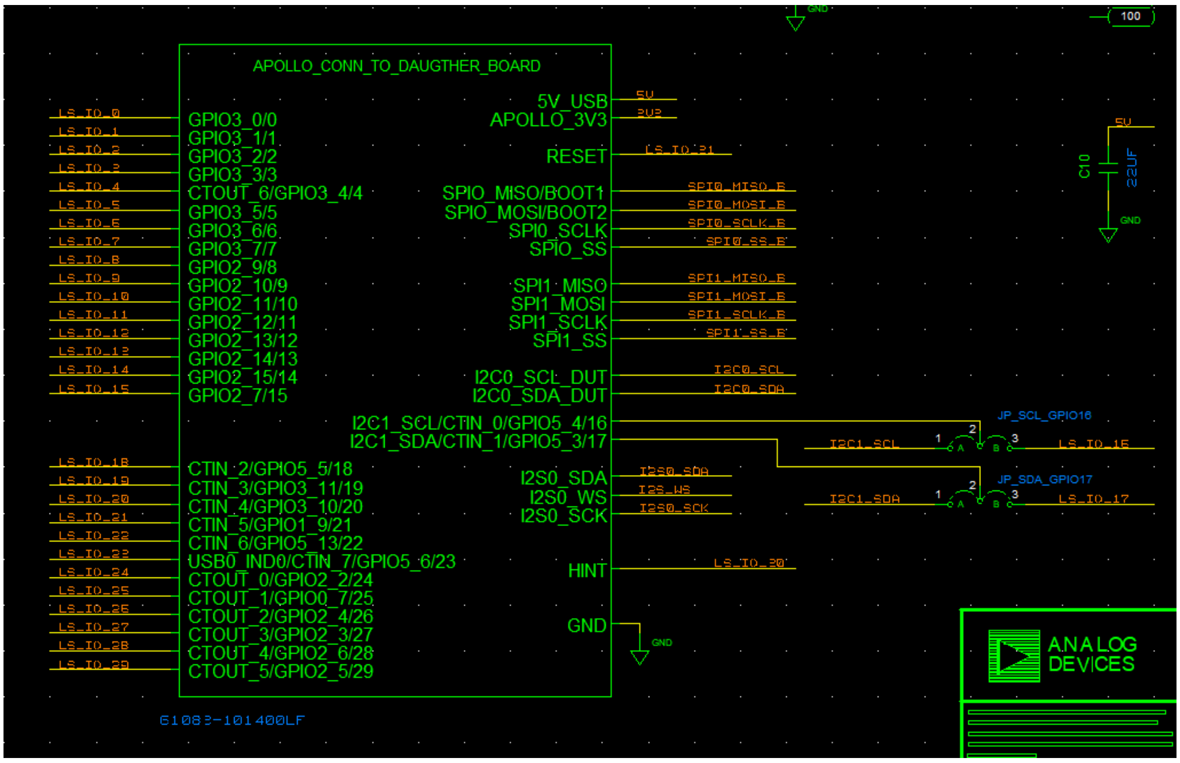


Figure 49 Proprietary connector