20th April 2020

IMAGE PROCESSING ON FPGA

Implementing Sobel Edge Detection Filter on Xillinx ZedBoard

Aditya Saripalli
INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

1.Image Processing

1.1 Introduction

Image processing is a method to perform some operations on an image, in order to get an enhanced image or to extract some useful information from it. It is a type of signal processing in which input is an image and output may be image or characteristics/features associated with that image. Nowadays, image processing is among rapidly growing technologies. It forms core research area within engineering and computer science disciplines - Artificial Intelligence and Machine Learning.

Image processing basically includes the following 3 steps:

- Importing the image via image acquisition tools.
- Analyzing and manipulating the image.
- Output in which result can be altered image or report that is based on image analysis.

The purpose of image processing can be divided into 5 groups:

- *Visualization* : Observe the objects that are not visible.
- *Image sharpening and Restoration* : To create a better image.
- *Image Retrieval* : Seek for the image of interest.
- *Measurement of pattern* : Measures various objects in an image.
- *Image Recognition* : Distinguish the objects in an image.

There are 2 types of methods used for image processing namely, Analog and Digital image processing.

- ***Analog*** image processing can be used for the hard copies like printouts and photographs. Image analysts use various fundamentals of interpretation while using these visual techniques.
- ***Digital*** image processing techniques help in manipulation of the digital images by using computers. The three general phases that all types of data have to undergo while using digital technique are pre-processing, enhancement, and display, information extraction.

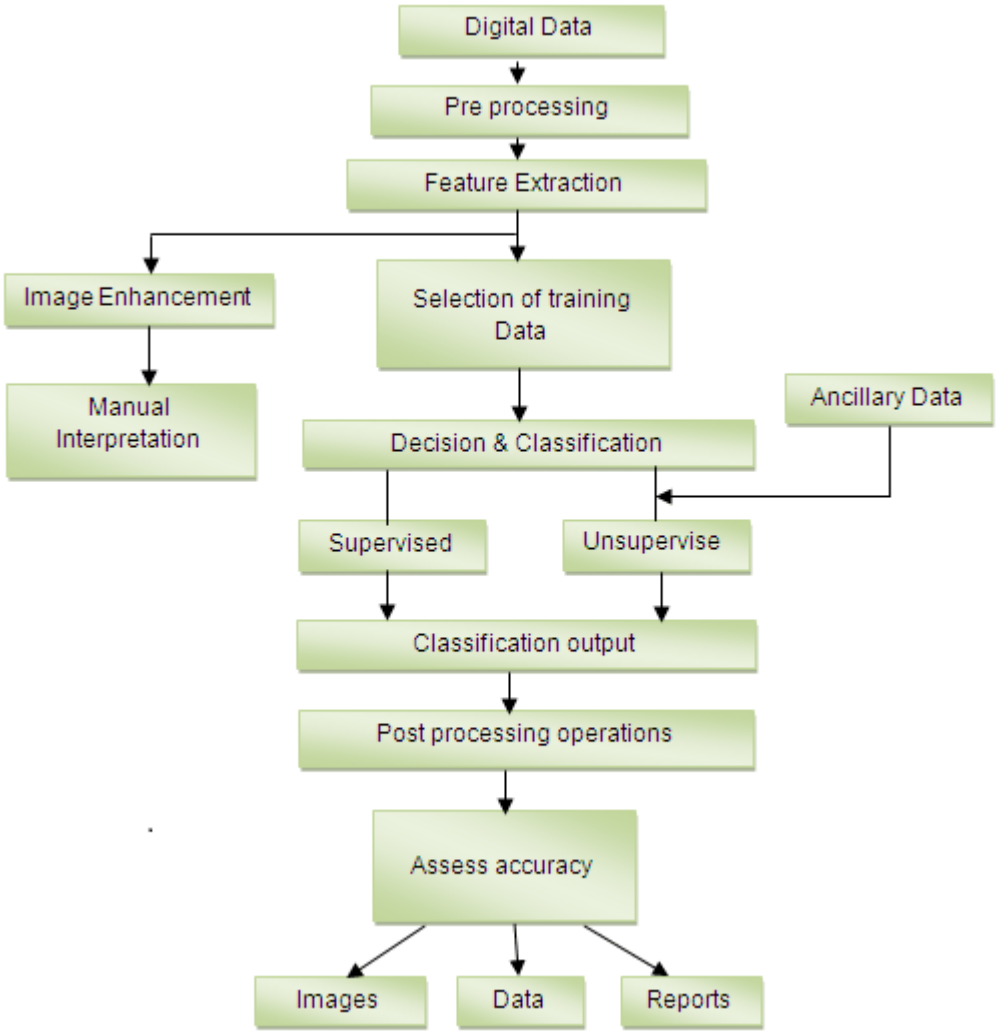
Digital Processing techniques help in manipulation of the digital images by using computers. Raw data from imaging sensors contains deficiencies so to get over such flaws and to extract the originality of information, it must undergo various phases of processing. The three general phases that all types of data must undergo while using digital technique are pre-processing, Enhancement and Display, Information Extraction.

Digital Image Processing has wide range of applications in various domains like - Intelligent Transportation Systems, Moving Object Tracking, Defense surveillance, Biomedical Imaging Techniques, Automatic Visual Inspection Systems, and many more.

1.2 Image Transformations

There are several techniques used in image processing that are performed on the digital images to extract some useful information. Wide range of algorithms can be applied to input data which can avoid problems such as noise and signal distortion during processing.

The different phases involved in the digital image processing are shown below in the flow chart.



There are several image transformation and enhancement techniques like Smoothening, Image sharpening, Convolution, Image Filtering, Masking, Image blending, Stitching (panoramic processing), Edge Detection, Gray level processing, and many more.

Out of all the above techniques, for the current discussion, I would like to focus on one image filtering technique – *"Edge Detection"* of digital images.

1.3 Edge Detection

In image processing, edge detection is a very important task. Edge detection is the main tool in pattern recognition, image segmentation and scene analysis. The concept of edge detection is used to detect the location and presence of edges by making changes in the intensity of an image. Different operations are used in image processing to detect edges. It can detect the variation of grey levels, but it quickly gives response when a noise is detected. It is a type of filter which is applied to extract the edge points in an image. Sudden changes in an image occurs when the edge of an image contour across the brightness of the image.

In image processing, edges are interpreted as a single class of singularity. In a function, the singularity is characterized as discontinuities in which the gradient approaches are infinity. As we know that the image data is in the discrete form, so edges of the image are defined as the local maxima of the gradient.

Mostly edges exist between objects and objects, primitives and primitives, objects and background. The objects which are reflected back are in discontinuous form. Methods of edge detection study to change a single pixel of an image in gray area.

Edge detection is mostly used for the measurement, detection and location changes in an image gray. Edges are the basic feature of an image. In an object, the clearest part is the edges and lines. With the help of edges and lines, an object structure is known. That is why extracting the edges is a very important technique in graphics processing and feature extraction.

The basic idea behind edge detection is as follows:

- To highlight local edge operator, use edge enhancement operator.
- Define the edge strength and set the edge points.

Among all the famous edge detection operators like – Robert's Cross Operator, Laplacian of Gaussian, Prewitt Operator, the most common and widely used operator is "Sobel Edge Detection Operator" – which is the focus of this discussion too.

1.4 Sobel Edge Detection

The Sobel edge detection operator extracts all the edges of an image, without worrying about the directions. The main advantage of the Sobel operator is that it provides *differencing* and *smoothing* effect. The operator is implemented as the sum of two directional edges and the resulting image is a unidirectional outline in the original image.

Sobel Edge detection operator consists of 3x3 convolution kernels, as shown below. Gx is a simple kernel and Gy is rotated by 90°.

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

These Kernels are applied separately to input image because separate measurements can be produced in each orientation – Gx and Gy.

The gradient magnitude is computed as follows:

$$|G| = \sqrt{Gx^2 + Gy^2}$$

For faster computations, an approximate magnitude is computed as below:

$$|G| = |Gx| + |Gy|$$

In the next few sections, I will try to explain how to implement a Sobel Edge Detection operator in VDHL/Verilog (by using Vivado High Level Synthesis toolset) and compare the performance of the application on FPGA (Xillinx ZedBoard) with the standalone implementation of the Sobel Edge Detection on intel x86 architecture using OpenCV libraries.

2.Introduction to FPGA

2.1 Overview

Software is the basis of all applications. Many of the products people use today began as a software model or prototype. Based on the performance and programmability constraints of the system, the software engineer is tasked with determining the best implementation platform to get a project to market. To accomplish this task, the software engineer is aided by both programming techniques and a variety of hardware processing platforms.

On the programming side, previous decades yielded advances in object-oriented programming for code reuse and parallel computing paradigms for boosting algorithm performance. The advancements in programming languages, frameworks, and tools allowed the software engineer to quickly prototype and test different approaches to solve a problem. This need to quickly prototype a solution leads to two interesting questions.

- How to analyze and quantify one algorithm against another ?
- Where to execute the algorithm ?

The first question is not the focus of current discussion, however regarding where to run an algorithm, there is an increasing focus on parallelization and concurrency. Although the interest in the parallel and concurrent execution of software programs is not new, the renewed and increased interest is aided by certain trends in processor and Application Specific Integrated Circuit (ASIC) design.

There are two ways of getting more performance out of a software algorithm - a custom-integrated circuit or an FPGA.

The first and most expensive option is to turn the algorithm over to a hardware engineer for a custom circuit implementation. Despite advancements in fabrication process node technology that have yielded significant improvements in power consumption, computational throughput, and logic density, the cost to fabricate a custom-integrated circuit or ASIC for an application is still high. At each processing node, the cost of fabrication continues to increase to the point where this approach is only economically viable for applications that ship in the range of millions of units.

The second option is to use an FPGA, which addresses the cost issues inherent in ASIC fabrication. FPGAs allow the designer to create a custom circuit implementation of an algorithm using an off-the-shelf component composed of basic programmable logic elements. This platform offers the power consumption savings and performance benefits of smaller fabrication nodes without incurring the cost and complexity of an ASIC development effort. Like an ASIC, an algorithm implemented in an FPGA benefits from the inherent parallel nature of a custom circuit.

2.2 What is an FPGA ?

A programmable hardware whose sub-system configuration can be modified even after fabrication, falls under the category of Reconfigurable System. And the most predominant integrated circuit that supports reconfigurable computing is FPGA, an acronym for **F**ield **P**rogrammable **G**ate **A**rray. An FPGA is a type of integrated circuit (IC) that can be programmed for different algorithms after fabrication. FPGA enables you to program product features, adapt to new standards, and reconfigure hardware for specific applications even after the product has been installed in the field — hence the term *field-programmable*. Whereas *gate arrays* refer to two-dimensional array of logic gates present in its architecture.

Modern FPGA devices consist of up to two million logic cells that can be configured to implement a variety of software algorithms. Although the traditional FPGA design flow is more like a regular IC than a processor, an FPGA provides significant cost advantages in comparison to an IC development effort and offers the same level of performance in most cases. Another advantage of the FPGA when compared to the IC is its ability to be dynamically reconfigured. This process, which is the same as loading a program in a processor, can affect part or all the resources available in the FPGA fabric.

All modern personal computers including desktops, notebooks, smartphones, and tablets, are examples of general-purpose computers. General-purpose computing incorporates 'Von Neumann' approach, which states that an instruction fetch, and a data operation cannot occur simultaneously. Therefore, being sequential machines, their performance is also limited.

On the other hand, we have the Application Specific Integrated Circuits (ASICs) which are customized for a task like a digital voice recorder or a high-efficiency Bitcoin miner. An ASIC uses a spatial approach to implement only one application and provides maximum performance. However, it can't be used for the tasks other than those for which it has been originally designed.

So, how about trading off the performance of ASICs for the flexibility of general-purpose processors?

FPGAs act as a middle ground between these two architectural paradigms. Having said that, FPGAs are less energy efficient when compared to ASICs and also not suitable for large volume productions. However, they are reprogrammable and have low NRE costs when compared to an ASIC.

ASICs and FPGAs have different value propositions. Most device manufacturers typically prefer FPGAs for prototyping and ASICs for very large production volumes. FPGAs used to be chosen for lower speed and complex designs in the past, but nowadays FPGAs can easily surpass the 500 MHz performance benchmark.

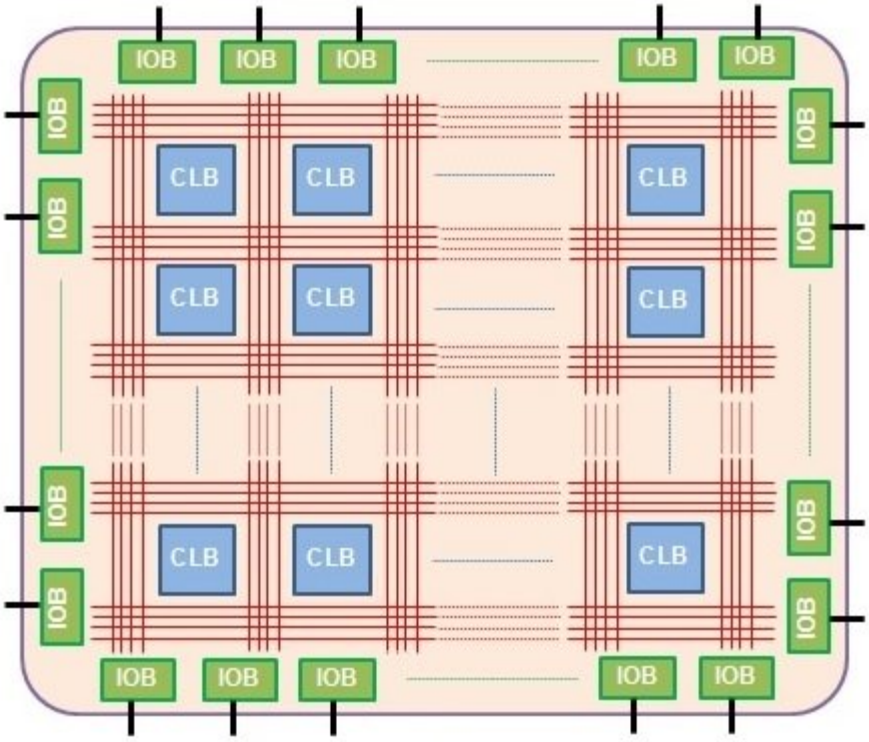
3. FPGA Architecture Details

3.1 Overview

The basic structure of an FPGA is composed of the following elements:

- **Look-up table (LUT):** This element performs logic operations.
- **Flip-Flop (FF):** This register element stores the result of the LUT.
- **Wires:** These elements connect elements to one another.
- **Input/Output (I/O) pads:** These physically available ports get data in and out of the FPGA.

The combination of these elements results in the basic FPGA architecture shown in the figure below.



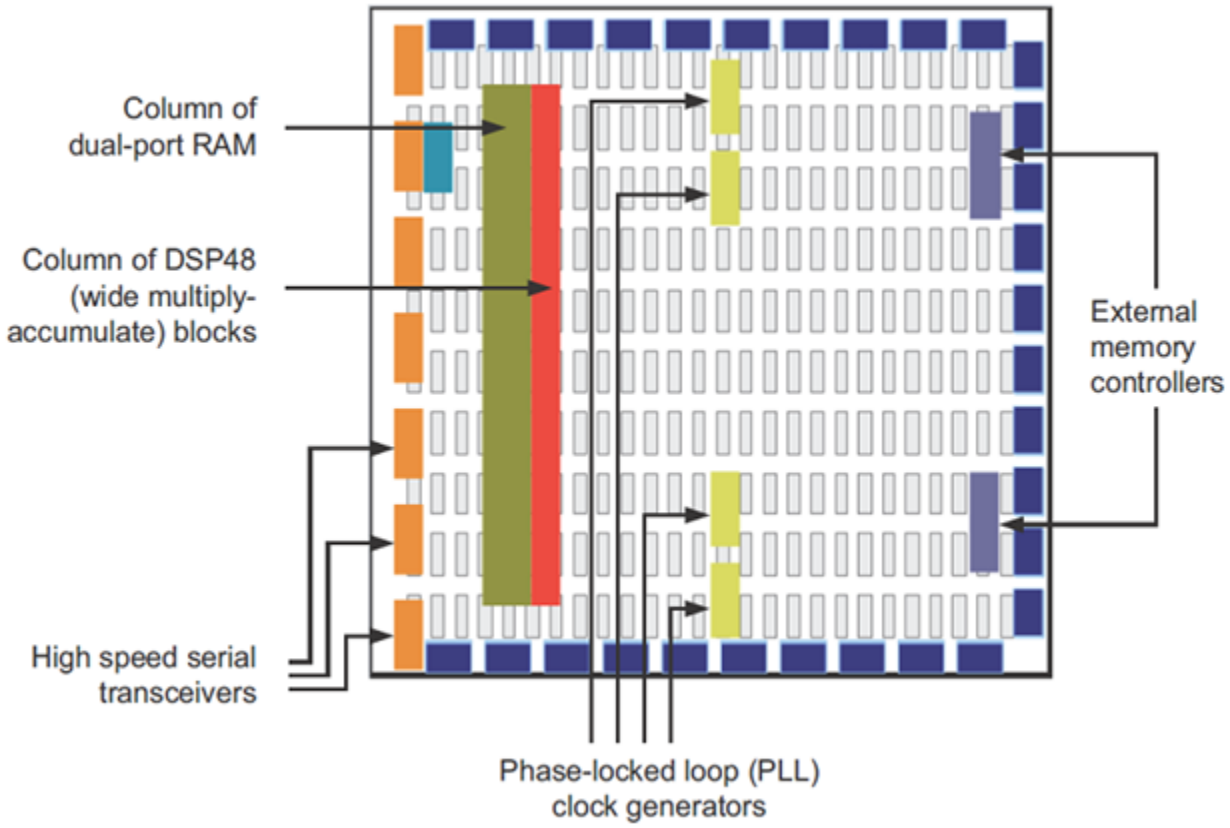
Although this structure is sufficient for the implementation of any algorithm, the efficiency of the resulting implementation is limited in terms of computational throughput, required resources, and achievable clock frequency.

Contemporary FPGA architectures incorporate the basic elements along with additional computational and data storage blocks that increase the computational density and efficiency of the device. These additional elements, which are discussed in the following sections, are:

- Embedded memories for distributed data storage
- Phase-locked loops (PLLs) for driving the FPGA fabric at different clock rates

- High-speed serial transceivers
- Off-chip memory controllers
- Multiply-accumulate blocks

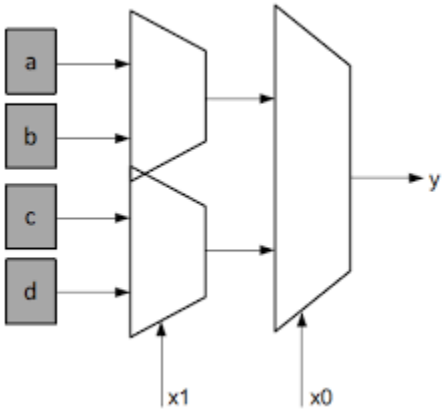
The combination of these elements provides the FPGA with the flexibility to implement any software algorithm running on a processor and results in the contemporary FPGA architecture as shown below:



3.2 Look-up Table (LUT)

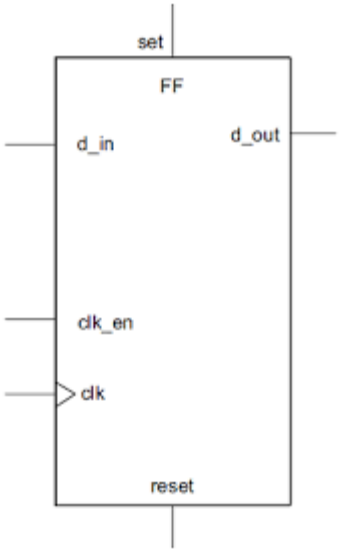
The LUT is the basic building block of an FPGA and is capable of implementing any logic function of N Boolean variables. Essentially, this element is a truth table in which different combinations of the inputs implement different functions to yield output values. The limit on the size of the truth table is N, where N represents the number of inputs to the LUT. For the general N-input LUT, the number of memory locations accessed by the table is – $\mathbf{2^N}$**.** Which allows the table to implement number of functions to be – $\mathbf{2^{N^N}}$. A typical value for N in FPGA devices is 6.

The hardware implementation of a LUT can be thought of as a collection of memory cells connected to a set of multiplexers. The inputs to the LUT act as selector bits on the multiplexer to select the result at a given point in time. It is important to keep this representation in mind, because a LUT can be used as both a function compute engine and a data storage element. The figure shows this functional representation of the LUT.



3.3 Flip-Flop

The flip-flop is the basic storage unit within the FPGA fabric. This element is always paired with a LUT to assist in logic pipelining and data storage. The basic structure of a flip-flop includes a data input, clock input, clock enable, reset, and data output. During normal operation, any value at the data input port is latched and passed to the output on every pulse of the clock. The purpose of the clock enable pin is to allow the flip-flop to hold a specific value for more than one clock pulse. New data inputs are only latched and passed to the data output port when both clock and clock enable are equal to one. The figure below shows the structure of a flip-flop.



3.4 Storage Elements

The FPGA device includes embedded memory elements that can be used as random-access memory (RAM), read-only memory (ROM), or shift registers. These elements are block RAMs (BRAMs), UltraRAM blocks (URAMS), LUTs, and shift registers (SRLs).

The BRAM is a dual-port RAM module instantiated into the FPGA fabric to provide on-chip storage for a relatively large set of data. The two types of BRAM memories available in a device

can hold either 18 k or 36 k bits. The number of these memories available is device specific. The dual-port nature of these memories allows for parallel, same-clock-cycle access to different locations.

In terms of how arrays are represented in C/C++ code, BRAMs can implement either a RAM or a ROM. The only difference is when the data is written to the storage element. In a RAM configuration, the data can be read and written at any time during the runtime of the circuit. In contrast, in a ROM configuration, data can only be read during the runtime of the circuit. The data of the ROM is written as part of the FPGA configuration and cannot be modified in any way.

As previously discussed, the LUT is a small memory in which the contents of a truth table are written during device configuration. Due to the flexibility of the LUT structure in Xilinx FPGAs, these blocks can be used as 64-bit memories and are commonly referred to as distributed memories. This is the fastest kind of memory available on the FPGA device, because it can be instantiated in any part of the fabric that improves the performance of the implemented circuit.

The shift register is a chain of registers connected to each other. The purpose of this structure is to provide data reuse along a computational path, such as with a filter. For example, a basic filter is composed of a chain of multipliers that multiply a data sample against a set of coefficients. By using a shift register to store the input data, a built-in data transport structure moves the data sample to the next multiplier in the chain on every clock cycle.

3.5 FPGA Parallelism vs Processor Architecture

When compared with processor architectures, the structures that comprise the FPGA fabric enable a high degree of parallelism in application execution. To examine the benefits of the FPGA execution paradigm, let us have a brief review of processor program execution.

3.5.1 Program execution on a processor

A processor, regardless of its type, executes a program as a sequence of instructions that translate into useful computations for the software application. This sequence of instructions is generated by processor compiler tools, such as the GNU Compiler Collection (GCC), which transform an algorithm expressed in C/C++ into assembly language constructs that are native to the processor. The job of a processor compiler is to take a C function of the form:

```
z = a + b;
```

and transform into assembly code as:

```
ADD  $R1, $R2, $R3
```

The assembly code above defines the addition operation to compute the value of z in terms of the internal registers of a processor. The code states that the input values for the computation are stored in registers R1 and R2, and the result of the computation is stored in register R3. This

code is simple, and it does not express all the instructions needed to compute the value of z. This code only handles the computation after the data has arrived at the processor. Therefore, the compiler must create additional assembly language instructions to load the registers of the processor with data from a central memory and to write back the result to memory. The complete assembly program to compute the value of z is as follows:

```
LD      a, $R1
LD      b, $R2
ADD     $R1, $R2, $R3
ST      $R3, c
```

The code above shows that even a simple operation, such as the addition of two values, results in multiple assembly instructions. The computational latency of each instruction is not equal across instruction types. Depending on the location of a and b, the LD operations take a different number of clock cycles to complete.

If the values are in the processor cache, these load operations complete within a few tens of clock cycles. If the values are in the main, double data rate (DDR) memory, the operations take between hundreds and thousands of clock cycles to complete. If the values are in a hard drive, the load operations take even longer to complete. This is why software engineers with cache hit traces spend so much time restructuring their algorithms to increase the spatial locality of data in memory to increase the cache hit rate and decrease the processor time spent per instruction.

3.5.2 Program execution on a FPGA

The FPGA is an inherently parallel processing fabric capable of implementing any logical and arithmetic function that can run on a processor. It is not hindered by the restrictions of a cache and a unified memory space. The computation of z is compiled into several LUTs required to achieve the size of the output operand. For example, assume that in the original software program the variable a, b, and z are defined with the short data type. This type, which defines a 16-bit data container, gets implemented as 16 LUTs (1 LUT is equivalent to 1 bit of computation). The LUTs used for the computation of z are exclusive to this operation only. Unlike a processor, where all computations share the same ALU, an FPGA implementation instantiates independent sets of LUTs for each computation in the software algorithm.

In addition to assigning unique LUT resources per computation, the FPGA differs from a processor in both memory architecture and the cost of memory accesses. In an FPGA implementation, the compiler can arrange memories into multiple storage banks as close as possible to the point of use in the operation. This results in an instantaneous memory bandwidth, which far exceeds the capabilities of a processor.

With regard to computational throughput and memory bandwidth, the compiler can exercise the capabilities of the FPGA fabric through the processes of *scheduling*, *pipelining*, and *dataflow*.

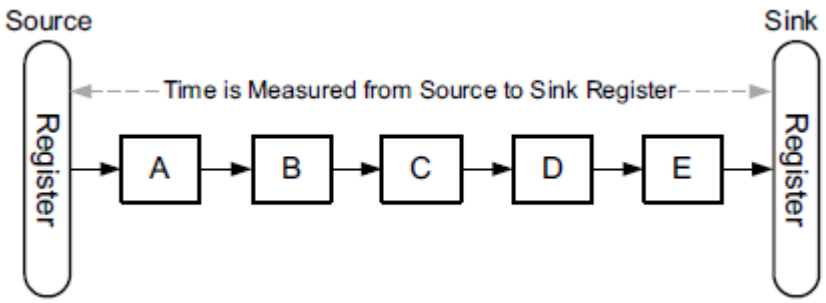
Although transparent to the user, these processes are integral stages of the software compilation process that extract the best possible circuit-level implementation of the software application.

Scheduling

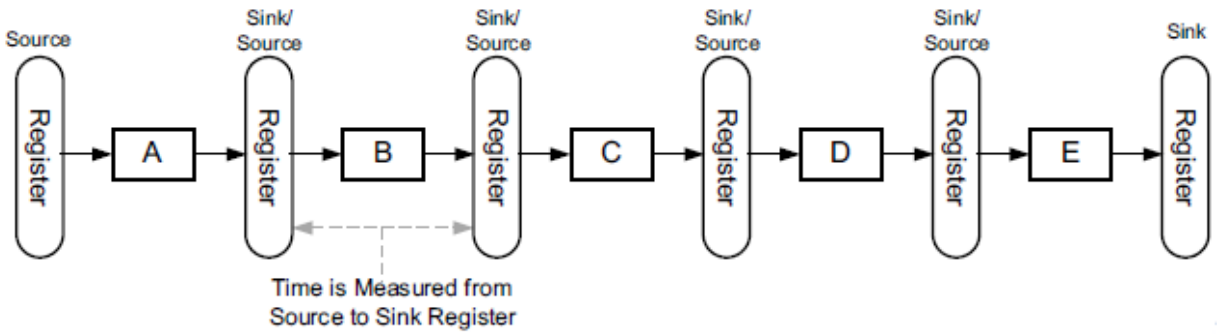
Scheduling is the process of identifying the data and control dependencies between different operations to determine when each will execute. In traditional FPGA design, this is a manual process also referred to as parallelizing the software algorithm for a hardware implementation. Vivado HLS analyzes dependencies between adjacent operations as well as across time. This allows the compiler to group operations to execute in the same clock cycle and to set up the hardware to allow the overlap of function calls. The overlap of function call executions removes the processor restriction that requires the current function call to fully complete before the next function call to the same set of operations can begin. This process is called pipelining.

Pipelining

Pipelining is a digital design technique that allows the designer to avoid data dependencies and increase the level of parallelism in an algorithm hardware implementation. The data dependence in the original software implementation is preserved for functional equivalence, but the required circuit is divided into a chain of independent stages. All stages in the chain run in parallel on the same clock cycle. The only difference is the source of data for each stage. Each stage in the computation receives its data values from the result computed by the preceding stage during the previous clock cycle.



FPGA implementation without pipelining



FPGA implementation with pipelining

4. Vivado High Level Synthesis (HLS)

4.1 Overview

The Xilinx Vivado High-Level Synthesis (HLS) compiler provides a programming environment similar to those available for application development on both standard and specialized processors. Vivado HLS shares key technology with processor compilers for the interpretation, analysis, and optimization of C/C++ programs. The main difference is in the execution target of the application.

By targeting an FPGA as the execution fabric, Vivado HLS enables a software engineer to optimize code for throughout, power, and latency without the need to address the performance bottleneck of a single memory space and limited computational resources. This allows the implementation of computationally intensive software algorithms into actual products, not just functionality demonstrators.

Vivado HLS analyzes all programs in terms of:

- Operations
- Conditional statements
- Loops
- Functions

Operations

Operations refer to both the arithmetic and logical components of an application that are involved in computing a result value. When working with operations, the main difference between Vivado HLS and other compilers is in the restrictions placed on the designer. With a processor compiler, the fixed processing architecture means that the user can only affect performance by limiting operation dependency and manipulating memory layout to maximize cache performance. In contrast, Vivado HLS is not constrained by a fixed processing platform and builds an algorithm-specific platform based on user input. This allows an HLS designer to affect application performance in terms of throughput, latency, and power.

Conditional statements

Conditional statements are program control flow statements that are typically implemented as if, if-else, or case statements. These coding structures are an integral part of most algorithms and are fully supported by all compilers, including HLS. The only difference between compilers is how these types of statements are implemented.

With a processor compiler, conditional statements are translated into branch operations that might or might not result in a context switch. The introduction of branches disrupts the maximum instruction execution packing by introducing a dependence that affects which instruction is fetched next from memory. This uncertainty results in bubbles in the processor execution pipeline and directly affects program performance. In an FPGA, a conditional

statement does not have the same potential impact on performance as in a processor. Vivado HLS creates all the circuits described by each branch of the conditional statement. Therefore, the runtime execution of a conditional software statement involves the selection between two possible results rather than a context switch.

Loops

Loops are a common programming construct for expressing iterative computation. HLS fully supports loops and can even do transformations that are beyond the capabilities of a standard processor compiler. For example, assume that a loop takes 4 clock cycles per iteration regardless of the implementation platform. On a processor, the compiler is forced to schedule loop iterations sequentially for a total run time of 40 cycles.

HLS does not have this limitation. Because HLS creates the hardware for the algorithm, it can alter the execution profile of a loop by pipelining iterations. Loop iteration pipelining extends the concept of operation parallelization from within loop iterations to across iterations. To reduce iteration latency, the first automatic optimization applied by Vivado HLS is operator parallelization to the loop iteration body. The second optimization is loop iteration pipelining. This optimization requires user input, because it affects the resource consumption and input data rates of the FPGA implementation.

HLS can parallelize or pipeline the iterations of a loop to reduce computation latency and increase the input data rate. The user controls the level of iteration pipelining by setting the loop initialization interval (II). The II of a loop specifies the number of clock cycles between the start times of consecutive loop iterations.

Functions

Functions are a programming hierarchy that can contain operators, loops, and other functions. The treatment of functions in both HLS and processor compilers is similar to that of loops.

HLS can parallelize the execution of both loops and functions. With loops, this transformation is typically referred to as pipelining, because there is a clear hierarchy difference between operators and loop iterations. With functions, operations outside of a loop body and within loops are in the same hierarchical context, which might lead to confusion if the term pipelining is used. To avoid potential confusion when working with HLS, the parallelization of function call execution is referred to as dataflow optimization.

The dataflow optimization instructs HLS to create independent hardware modules for all functions at a given level of program hierarchy. These independent hardware modules are capable of concurrent execution and self-synchronize during data transfer.

4.2 HLS Optimization Pragmas

In both accelerator and system-on-chip development environments, the hardware kernel must be synthesized from the OpenCL, C, or C++ language into the register transfer level (RTL) that can be implemented into the programmable logic of a FPGA. The Vivado High Level Synthesis (HLS) tool synthesizes RTL from the OpenCL, C, and C++ language descriptions.

The HLS tool is intended to work with the development environment project without interaction. However, the HLS tool also provides pragmas that can be used to optimize the design - reduce latency, improve throughput performance, and reduce area and device resource usage of the resulting RTL code. These pragmas can be added directly to the source code for the kernel.

The optimization types supported by HLS pragmas are listed in the table below.

Type	**Attributes**
Kernel Optimization	• pragma HLS allocation • pragma HLS expression_balance • pragma HLS latency • pragma HLS reset • pragma HLS resource • pragma HLS stable
Function Inlining	• pragma HLS inline • pragma HLS function_instantiate
Interface Synthesis	• pragma HLS interface
Task-level Pipeline	• pragma HLS dataflow • pragma HLS stream
Pipeline	• pragma HLS pipeline • pragma HLS occurrence
Loop Unrolling	• pragma HLS unroll • pragma HLS dependence
Loop Optimization	• pragma HLS loop_flatten • pragma HLS loop_merge • pragma HLS loop_tripcount
Array Optimization	• pragma HLS array_map • pragma HLS array_partition • pragma HLS array_reshape
Structure Packing	• pragma HLS data_pack

5. Xillinx ZedBoard

5.1 ZYNQ7000 SoC

The Zynq-7000 family is based on the Xilinx SoC architecture. These products integrate a feature-rich dual or single-core ARM Cortex-A9 MPCore based processing system (PS) and Xilinx programmable logic (PL) in a single device, built on a state-of-the-art, high-performance, low-power (HPL), 28 nm, and high-k metal gate (HKMG) process technology. The ARM Cortex-A9 MPCore CPUs are the heart of the PS which also includes on-chip memory, external memory interfaces, and a rich set of I/O peripherals.

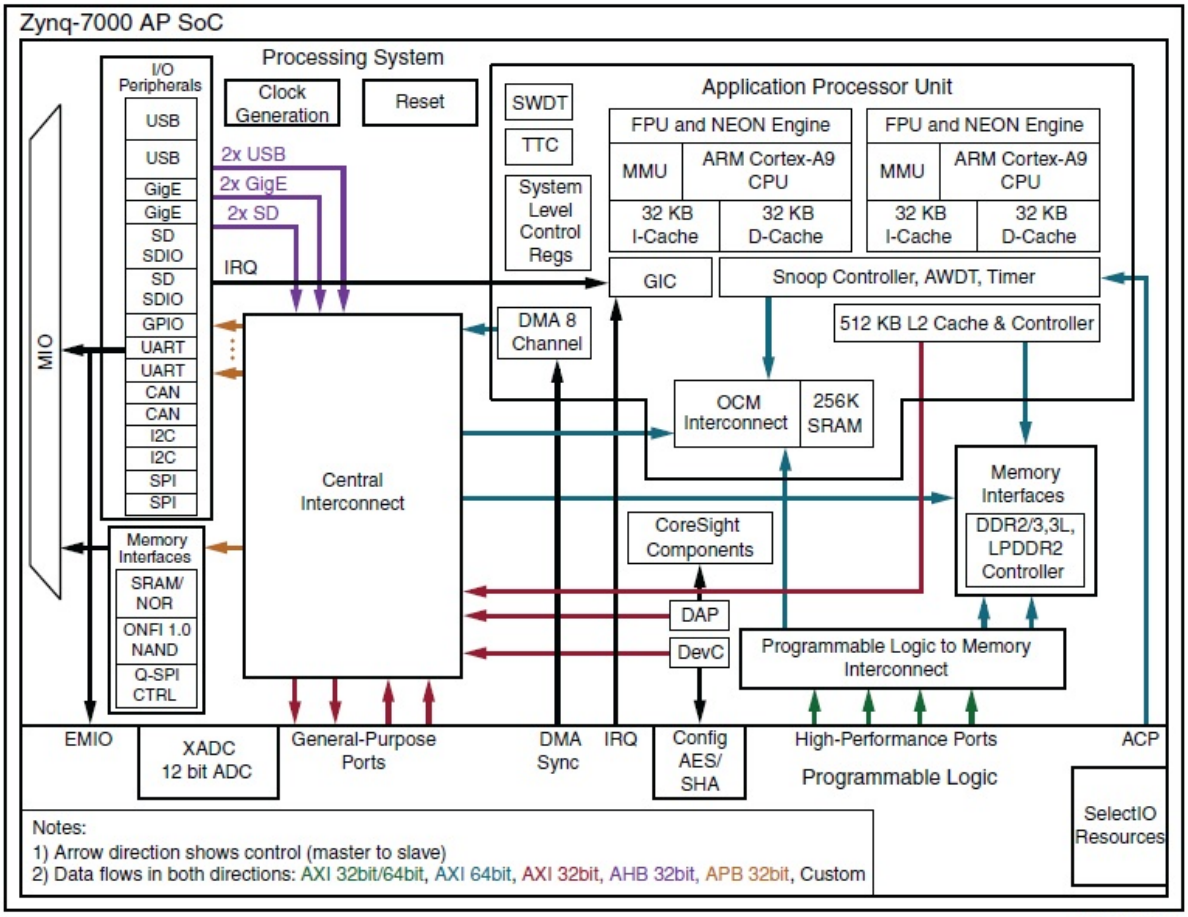
The Zynq-7000 family offers the flexibility and scalability of an FPGA, while providing performance, power, and ease of use typically associated with ASIC and ASSPs. The range of devices in the Zynq-7000 SoC family enables designers to target cost-sensitive as well as high-performance applications from a single platform using industry-standard tools. While each device in the Zynq-7000 family contains the same PS, the PL and I/O resources vary between the devices. As a result, the Zynq-7000 SoC devices are able to serve a wide range of applications.

The Zynq-7000 architecture conveniently maps the custom logic and software in the PL and PS respectively. It enables the realization of unique and differentiated system functions. The integration of the PS with the PL provides levels of performance that two-chip solutions cannot match due to their limited I/O bandwidth, loose-coupling and power budgets.

The processor(s) in the PS always boot first, allowing a software centric approach for PL system boot and PL configuration. The PL can be configured as part of the boot process or configured at some point in the future. Additionally, the PL can be completely reconfigured or used with partial, dynamic reconfiguration (PR). PR allows configuration of a portion of the PL. This enables optional design changes such as updating coefficients or time-multiplexing of the PL resources by swapping in new algorithms as needed. This latter capability is analogous to the dynamic loading and unloading of software modules. The PL configuration data is referred to as a bitstream.

Shown below is the ZYNQ7000 SoC Block diagram which illustrates the functional blocks of the Zynq-7000 SoC. The PS and the PL are on separate power domains, enabling the user of these devices to power down the PL for power management if required. The Zynq-7000 SoC is composed of the following major functional blocks:

- Processing System (PS)
  - Application processor unit (APU)
  - Memory interfaces
  - I/O peripherals (IOP)
  - Interconnect
- Programmable Logic (PL)



5.2 ZedBoard Overview

ZedBoard is a low-cost development board for the Xilinx Zynq-7000 SoC. This board contains everything necessary to create a Linux, Android, Windows or other OS/RTOS-based design. Additionally, several expansion connectors expose the processing system and programmable logic I/Os for easy user access. Take advantage of the Zynq-7000 SoC's tightly coupled ARM processing system and 7 series programmable logic to create unique and powerful designs with the ZedBoard.

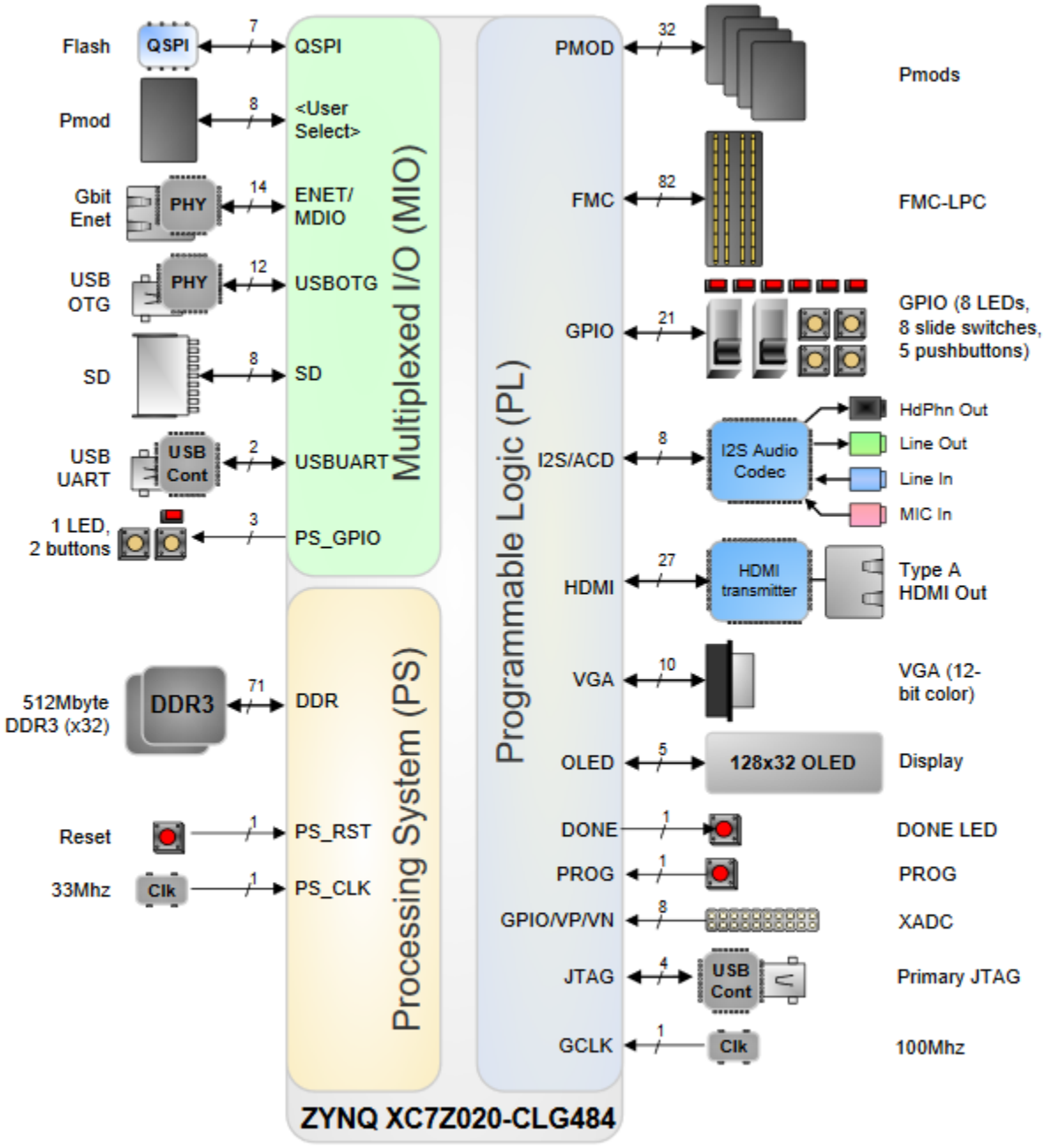
5.3 Key Features and Interfaces on ZedBoard

The following are the key features and interfaces provided by the Xillinx ZedBoard.

- Zynq-7000 SoC XC7Z020-CLG484-1
- 512 MB DDR3
- 256 Mb Quad-SPI Flash
- 4 GB SD card
- Onboard USB-JTAG Programming
- 10/100/1000 Ethernet
- USB OTG 2.0 and USB-UART
- PS & PL I/O expansion (FMC, Pmod, XADC)

- Multiple displays (1080p HDMI, 8-bit VGA, 128 x 32 OLED)
- I2S Audio CODEC

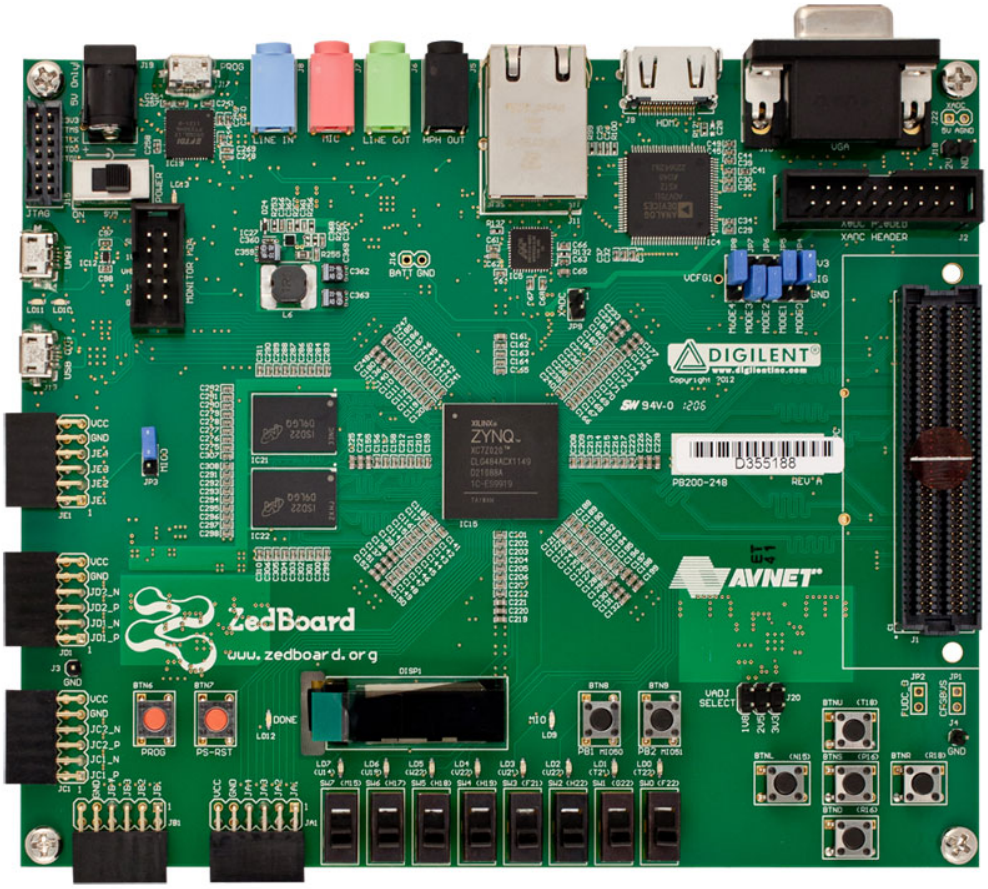
Shown below is the block diagram of a typical ZedBoard.



ZedBoard Block Diagram

Combining a dual Corex-A9 Processing System (PS) with 85,000 Series-7 Programmable Logic (PL) cells, the Zynq-7000 AP SoC can be targeted for broad use in many applications. The ZedBoard's

robust mix of on-board peripherals and expansion capabilities make it an ideal platform for both novice and experienced designers.



ZedBoard with ZYNQ7000 SoC

6. AXI Streaming Interface

6.1 What is AXI

The Advanced eXtensible Interface (AXI), part of the ARM Advanced Microcontroller Bus Architecture 3 (AXI3) and 4 (AXI4) specifications, is a parallel high-performance, synchronous, high-frequency, multi-master, multi-slave communication interface mainly designed for on-chip communication.

AXI has been introduced in 2003 with the AMBA3 specification. In 2010, a new revision of AMBA (AMBA4) - defined the AXI4, AXI4-Lite and AXI4-Stream protocol. AXI is royalty-free and its specification is freely available from ARM.

AXI offers a wide spectrum of features, including:

- Separate address/control and data phases
- Support for unaligned data accesses
- Burst-based transfers, with a single transmission of the starting address
- Separate and independent read and write channels
- Support for outstanding transactions
- Support for out-of-order transaction completion
- Support for atomic operations

AXI specifies many optional signals, which can be optionally included depending on the specific requirements of the design, making AXI a versatile bus for numerous applications. While the communication over an AXI bus is between a single master and a single slave, the specification includes detailed description and signals to include N:M interconnects, able to extend the bus to topologies with more masters and slaves.

AMBA AXI4, AXI4-Lite and AXI4-Stream have been adopted by Xilinx and many of its partners as main communication buses in their products. There are 3 types of AXI4 interfaces:

- **AXI4** For high-performance memory-mapped requirements
- **AXI4-Lite** For simple, low-throughput memory-mapped communication (for example, to and from control and status registers)
- **AXI4-Stream** For high-speed streaming data

6.2 AXI Benefits

AXI4 is widely adopted in Xilinx product offerings, providing benefits to *Productivity*, *Flexibility*, and *Availability*.

Productivity: by standardizing on the AXI interface, developers need to learn only a single protocol for IP.

Flexibility: Providing the right protocol for the application:

- AXI4 is for memory-mapped interfaces and allows high throughput bursts of up to 256 data transfer cycles with just a single address phase.
- AXI4-Lite is a light weight, single transaction memory-mapped interface. It has a small logic footprint and is a simple interface to work with both in design and usage.
- AXI4-Stream removes the requirement for an address phase altogether and allows unlimited data burst size. AXI4-Stream interfaces and transfers do not have address phases and are therefore not considered to be memory-mapped.

Availability: By moving to an industry-standard, you have access not only to the Vivado IP Catalog, but also to a worldwide community of ARM partners.

- Many IP providers support the AXI protocol.
- A robust collection of third-party AXI tool vendors is available that provide many verification, system development, and performance characterization tools. As you begin developing higher performance AXI-based systems, the availability of these tools is essential.

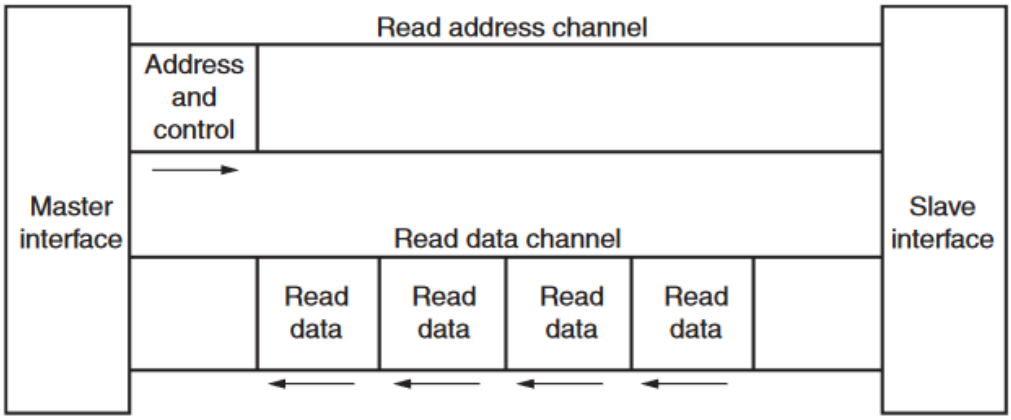
6.3 How AXI works

The AXI specifications describe an interface between a single AXI master and AXI slave, representing IP cores that exchange information with each other. Multiple memory-mapped AXI masters and slaves can be connected together using AXI infrastructure IP blocks. The Xilinx AXI Interconnect IP and the newer AXI SmartConnect IP contain 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.

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

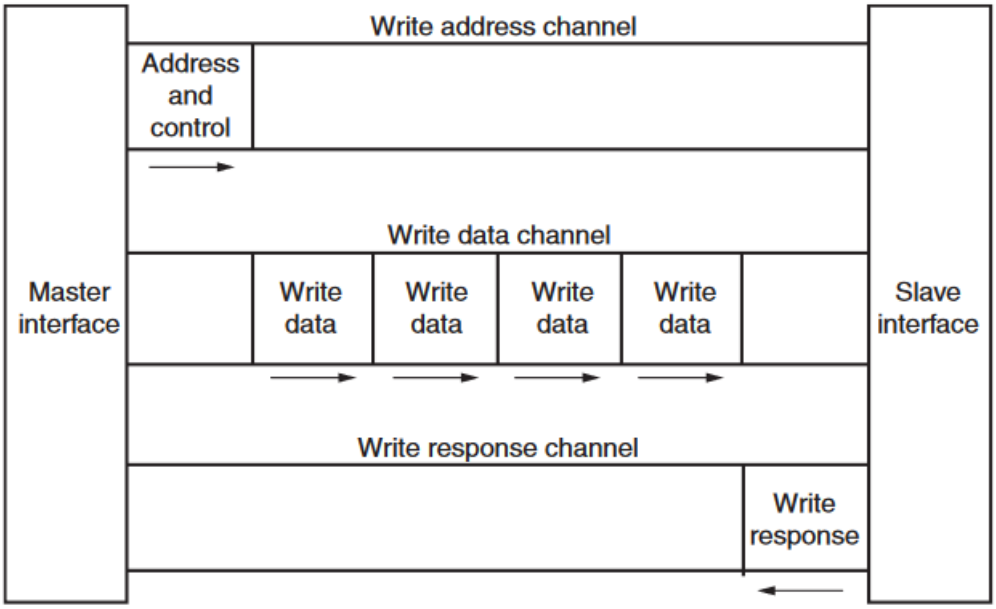
- Read Address Channel
- Write Address Channel
- Read Data Channel
- Write Data Channel
- Write Response Channel

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 following figure shows how an AXI4 read transaction uses the read address and read data channels.



Channel Architecture of Reads

The following figure shows how a write transaction uses the write address, write data, and write response channels.



Channel Architecture of Writes

As shown in the preceding figures, AXI4:

- Provides separate data and address connections for reads and writes, which allows simultaneous, bidirectional data transfer.
- Requires a single address and then bursts up to 256 words of data.

The *AXI4 protocol* describes options that allow AXI4-compliant systems to achieve very high data throughput. Some of these features, in addition to bursting, are – data upsizing and downsizing, multiple outstanding addresses, and out-of-order transaction processing.

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.

AXI4-Lite protocol is similar to AXI4 with some exceptions: The most notable exception is that bursting is not supported.

The *AXI4-Stream protocol* defines a single channel for transmission of streaming data. The AXI4-Stream channel models the write data channel of AXI4. Unlike AXI4, AXI4-Stream interfaces can burst an unlimited amount of data.

6.4 AXI Interconnect IP

AXI Interconnect IP connect one or more AXI memory-mapped master devices to one or more memory-mapped slave devices. he AXI Interconnect can be used in all memory-mapped designs.
The following subsections describe the possible use cases:

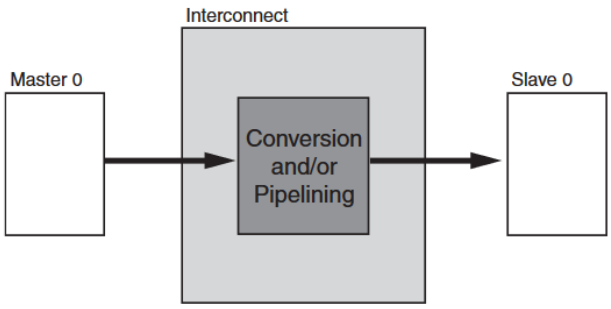
- Conversion Only
- N-to-1 Interconnect
- 1-to-N Interconnect
- N-to-M Interconnect

Conversion Only

The AXI Interconnect core can perform various conversion and pipelining functions when connecting one master device to one slave device. These are:

- Data width conversion
- Clock rate conversion
- AXI4-Lite slave adaptation
- AXI-3 slave adaptation
- Pipelining, such as a register slice or data channel FIFO

In these cases, the AXI Interconnect core contains no arbitration, decoding, or routing logic. There could be incurred latency, depending on the conversion being performed. The following figure shows the one-to-one or conversion use case.

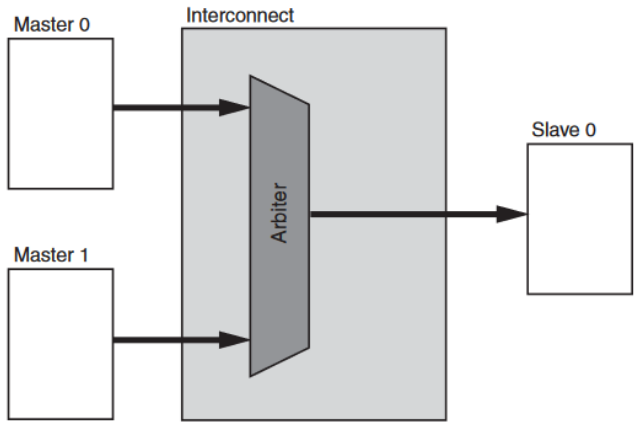


1-to-1 Conversion AXI Interconnect Use Case

N-to-1 AXI Interconnect

A common degenerate configuration of AXI Interconnect core is when multiple master devices arbitrate for access to a single slave device, typically a memory controller. In these cases, address decoding logic might be unnecessary and omitted from the AXI Interconnect core (unless address range validation is needed). Conversion functions, such as data width and clock rate conversion, can also be performed in this configuration.

The following figure shows the N-to-1 AXI interconnection use case.

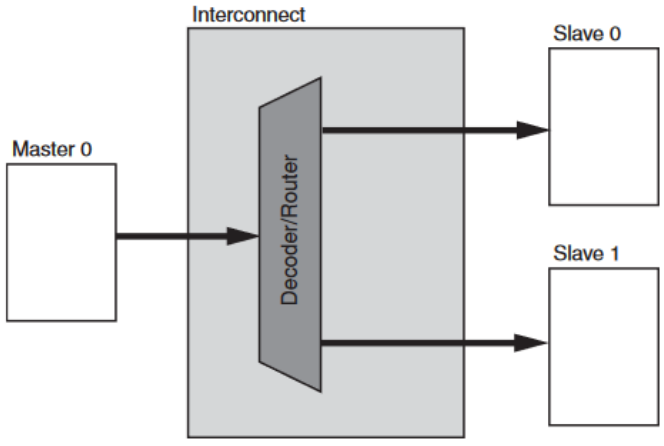


N-to-1 AXI interconnect

1-to-N AXI interconnect

Another degenerative configuration of the AXI Interconnect core is when a single master device, typically a processor, accesses multiple memory-mapped slave peripherals. In these cases, arbitration (in the address and write data paths) is not performed.

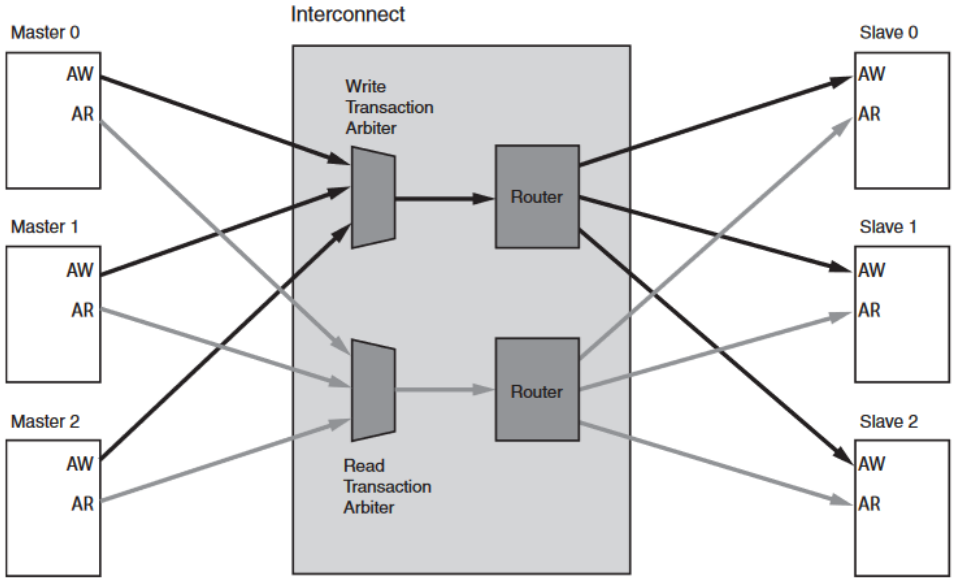
The following figure shows the 1 to N Interconnect use case.



1-to-N AXI interconnect

N-to-M AXI Interconnect

The N-to-M use case of the AXI Interconnect features a Shared-Address Multiple-Data (SAMD) topology, consisting of sparse data crossbar connectivity, with single-threaded write and read address arbitration, as shown in figure below.



Shared Write and Read Address Arbitrations

Parallel write and read data pathways connect each SI slot (attached to AXI masters on the left) to all the MI slots (attached to AXI slaves on the right) that it can access, according to the configured sparse connectivity map.

When more than one source has data to send to different destinations, data transfers can occur independently and concurrently, provided AXI ordering rules are met.

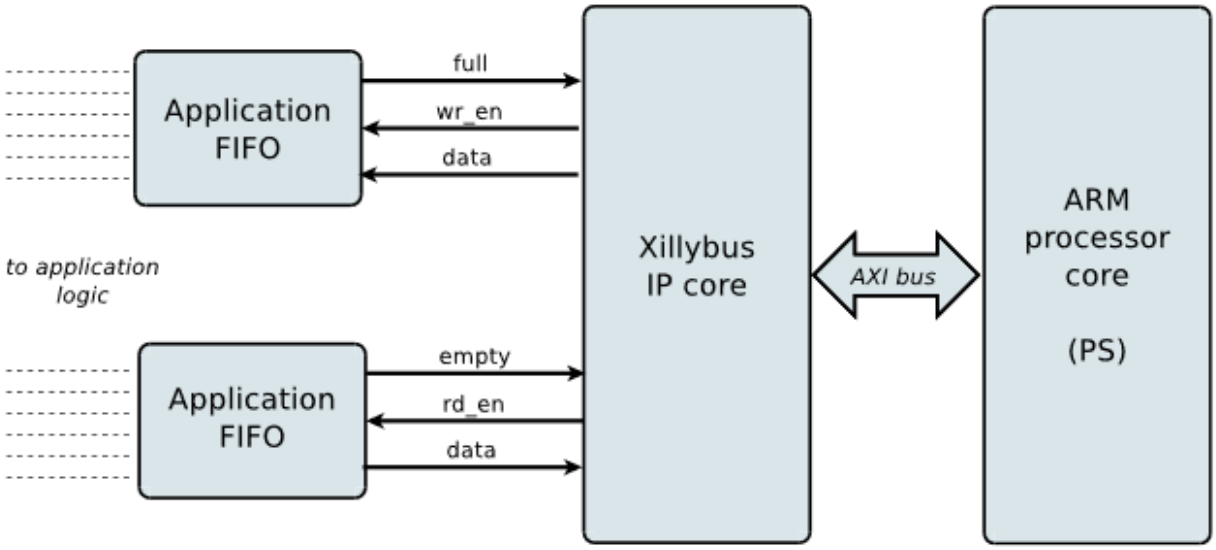
The write address channels among all SI slots (if > 1) feed into a central address arbiter, which grants access to one SI slot at a time, as is also the case for the read address channels. The winner of each arbitration cycle transfers its address information to the targeted MI slot and pushes an entry into the appropriate command queue(s) that enable various data pathways to route data to the proper destination while enforcing AXI ordering rules.

You can connect the slave interface of one AXI Interconnect core module to the master interface of another AXI Interconnect core with no intervening logic. Cascading multiple AXI Interconnects allow systems to be partitioned and potentially better optimized.

7. Xillybus and Xillinux

7.1 Xillybus IP Core

Xillybus is a straightforward, portable, intuitive, efficient DMA-based end-to-end turnkey solution for data transport between an FPGA and a host running Linux or Microsoft Windows. It's available for personal computers and embedded systems using the PCIe Express bus as the underlying transport, as well as ARM-based processors, interfacing with the AMBA bus (AXI3/AXI4).



As shown above, the application logic on the FPGA only needs to interact with standard FIFOs.

For example, writing data to the lower FIFO in the diagram makes the Xillybus IP core sense that data is available for transmission in the FIFO's other end. Soon, the Xillybus reads the data from the FIFO and sends it to the host, making it readable by the user space software. The data transport mechanism is transparent to the application logic in the FPGA, which merely interacts with the FIFO.

On its other side, the Xillybus IP core implements the data flow utilizing the AXI bus, generating DMA requests on the processor core's bus. The application on the computer interacts with device files that behave like named pipes. The Xillybus IP core and driver stream data efficiently and intuitively between the FIFOs in the FPGAs and their respective device files on the host.

The number of streams, their direction and other attributes are defined by customer to achieve an optimal balance between bandwidth performance, synchronization, and design simplicity.

7.2 How to use Xillybus

Xillybus was designed to present the Linux host with a simple and well-known inter-face, having a natural and expected behavior. The host driver generates device files that behave like named pipes. They are opened, read from and written to just like any file, but behave much like pipes between processes or TCP/IP streams. To the program running on the host, the difference is that the other side of the stream is not another process (over the network or on the same computer), but a FIFO in the FPGA. just like a TCP/IP stream, the Xillybus stream is designed to work well with high-rate data transfers as well single bytes arriving or sent occasionally.

Since the interface with Xillybus is all through device files that are accessed like just any file, any practical programming language can be used, with no need for a special module, extension or any other adaption. If a file can be opened with the chosen language, it can be used to access the FPGA through Xillybus.

One driver binary supports any Xillybus IP core configuration: The streams and their attributes are auto detected by the driver as it's loaded into the host's operating system, and device files are created accordingly. These device files are accessed as/dev/xillybus_something.

Also, at driver load, DMA buffers are allocated in the host's memory space, and the FPGA is informed about their addresses. The number of DMA buffers and their size are separate parameters for each stream. These parameters are hardcoded in the FPGA IP core for a given configuration and are retrieved by the host during the discovery process.

A handshake protocol between the FPGA and host makes an illusion of a continuous data stream. Behind the scenes, DMA buffers are filled, handed over and acknowledged. Techniques similar to those used for TCP/IP streaming are used to ensure an efficient utilization of the DMA buffers, while maintaining responsiveness for small pieces of data.

Since Xillybus I/O is carried out just like any device file I/O in Linux, there is apparently no need for a programming guide, as common programming practices can be employed.

Please refer ref#N for complete details on how to access the device streams using UNIX I/O programming.

7.3 Xillinux Distribution

Xillinux is a complete, graphical, Lubuntu 16.04-based Linux distribution for the Zynq-7000 device, intended as a platform for rapid development of mixed software / logic projects. The currently supported boards are Z-Turn Lite, Zedboard, MicroZed and Zybo.

Like any Linux distribution, Xillinux is a collection of software which supports roughly the same capabilities as a personal desktop computer running Linux. Unlike common Linux distributions, Xillinux also includes some of the hardware logic, in particular the VGA adapter.

With the Zedboard the distribution is organized for a classic keyboard, mouse and monitor setting. It also allows command-line control from the USB UART port.

Xillinux is also a kickstart development platform for integration between the device's FPGA logic fabric and plain user space applications running on the ARM processors. With its included Xillybus IP core and driver, no more than basic programming skills and logic design capabilities are needed to complete the design of an application where FPGA logic and Linux based software work together.

The bundled Xillybus IP cores eliminate the need to deal with the low-level internals of kernel programming and interface with the processor, by presenting a simple and yet efficient working environment to the application designers.

7.4 Current use case

Hosting Xillinux on ZedBoard gives user the flexibility of making the device (ZedBoard) a flexible environment for FPGA development. Using Xillinux as host environment, we can easily communicate with the Xillybus device streams by treating them as normal UNIX I/O devices.

Using Xillybus IP core as the base framework, one can easily integrate the user defined IP core created using Vivado HLS. In our case, I will create a Sobel Filter IP Core using Vivado HLS and integrate it with customized Xillybus IP Core. The customized Xillybus IP core will have the following additional data read/write streams added:

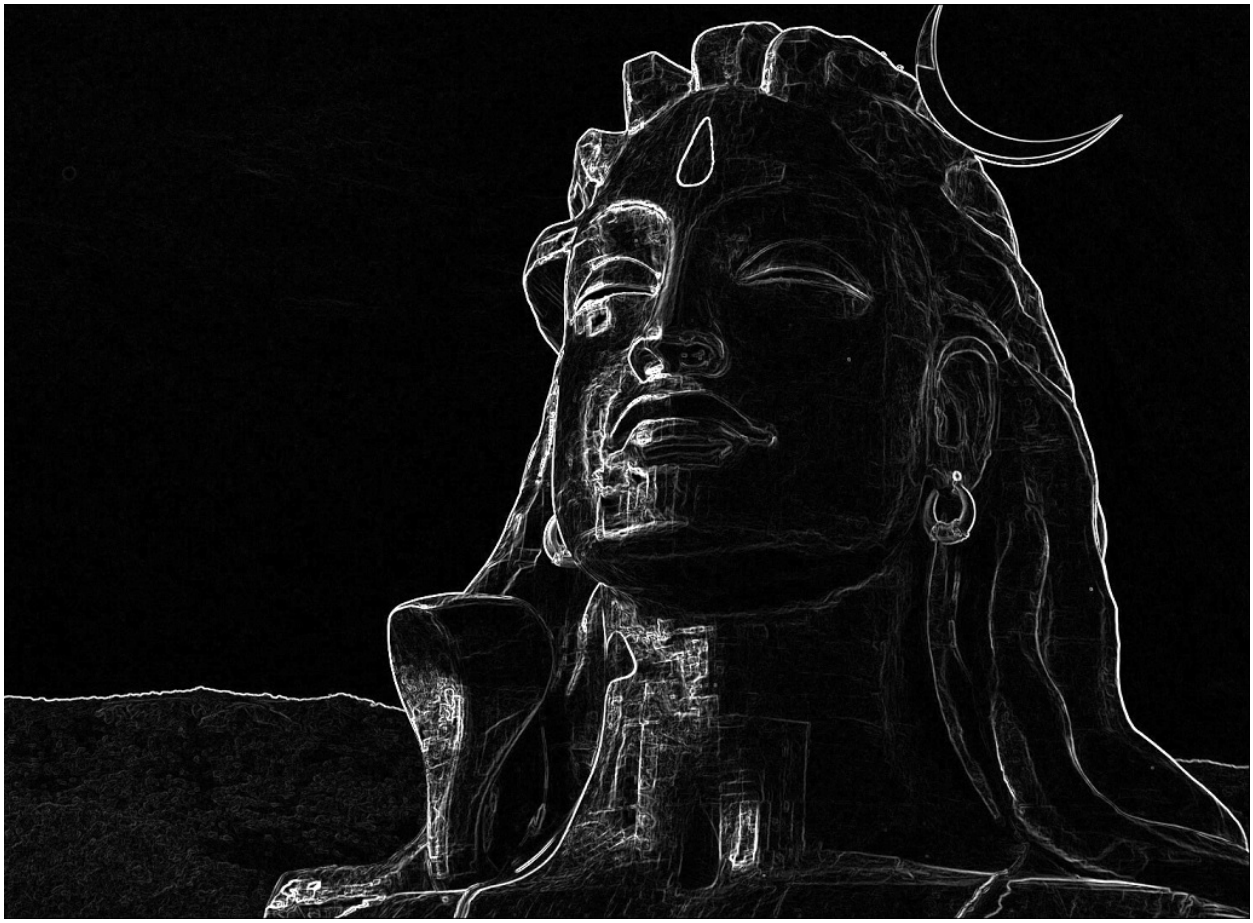
- xillybus_read_16_1 (Upstream – FPGA to host – 16 bits)
- xillybus_read_16_2 (Upstream – FPGA to host – 16 bits)
- xillybus_write_16_1 (Downstream – host to FPGA – 16 bits)
- xillybus_write_16_2 (Downstream – host to FPGA – 16 bits)

The reason behind adding these streams is, Sobel Filter IP core source internally uses 16-bit unsigned integer data for storing the black and white image streams. Adding these additional streams will give user more flexibility and ease of operating with the 16-bit integer data.

8. Sobel Edge Detection on x86 vs FPGA

8.1 Sobel Edge Detection (Original Image)

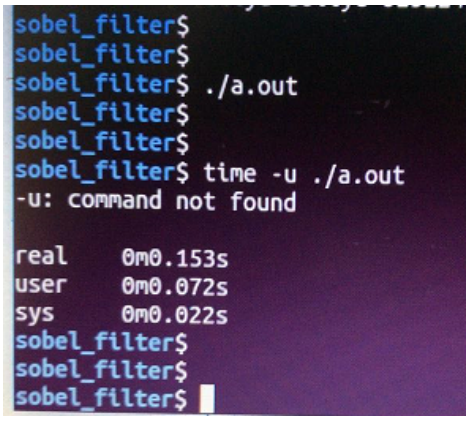
8.2 Sobel Edge Detection output on x86 system



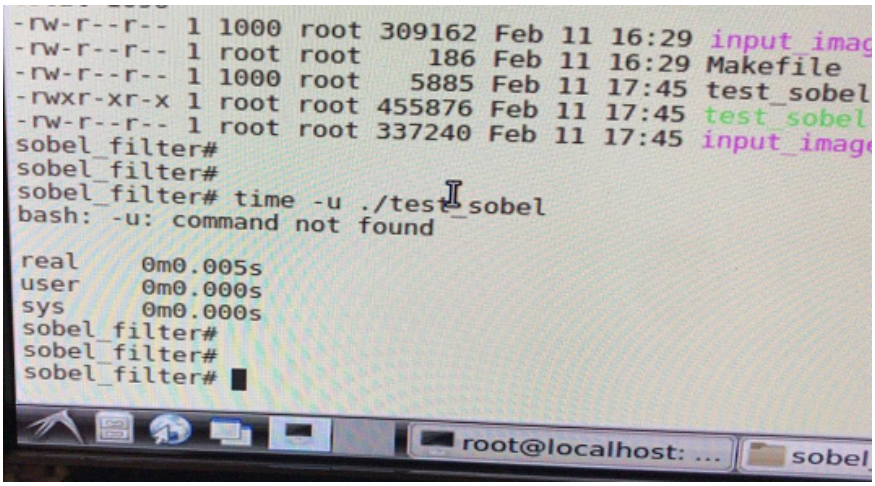
8.3 Sobel Edge Detection output on FPGA (ZedBoard)

8.4 Timing in x86 vs FPGA

After running the Sobel Filter algorithm on both Intel x86 system (hosting Ubuntu 18.03 LTS) and ZedBoard FPGA hosting Xillinux the timing statistics are as shown below:



Sobel Edge Detection on Intel x86 i5 CPU



Sobel Edge Detection on ZedBoard

Architecture	**Time taken to run Sobel Edge Detection Algorithm**
Intel x86 Core i5 CPU with 8 GB RAM	0.153 milli seconds
ZedBoard FPGA with ZYNQ7000 SoC with 512 MB BRAM (Block RAM)	0.005 milli seconds

Which means we did achieve an acceleration of 30x with ZedBoard. ZedBoard FPGA executed the Sobel Edge Detection algorithm **"30 times faster"** than intel x86 i5 CPU.

9.Source Code (GitHub)

9.1 GitHub

The complete source code for implementing the Sobel Edge Detection algorithm on a ZedBoard can be found at:
https://github.com/adisarip/sobel_edge_detection_zedboard

The source tree has the following structure:

```
sobel_edge_detection_zedboard
  ├── README.txt
  ├── report
  |   └── Sobel_Edge_Detection_on_Xillinx_ZedBoard_FPGA.pdf
  ├── src
  |   ├── opencv_standalone
  |   |   └── opencv_sed_test.cpp
  |   ├── vivado_hls_zedboard
  |   |   ├── zboard_sed_test.cpp
  |   |   ├── zboard_sed_top.cpp
  |   |   └── zboard_sed_top.h
  |   └── xillinux_test_sed
  |       ├── Makefile
  |       └── test_sobel.cpp
  ├── test_images
  |   ├── input_adiyogi.jpg
  |   ├── output_adiyogi_sed_x86.jpg
  |   └── output_adiyogi_sed_zedboard.jpg
  ├── vivado_block_design
  |   ├── sed_ip_core_block_design.pdf
  |   └── zynq_full_system_block_design.pdf
  └── xillybus_custom_core
          └── xillybus_ip_core_for_sed.pdf
```

10. References

10.1 Textbooks and Research Papers

- *Donald G. Bailey: Design for Embedded Image Processing on FPGAs*
- *Donald G. Bailey*. https://ieeexplore.ieee.org/book/6016259

10.2 AXI Interface:

- *AMBA4 AXI4-Stream Protocol Specification* : https://developer.arm.com/docs/ihi0051/a
- *Vivado AXI Reference Guide* : https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

10.3 Vivado HLS:

- *Introduction to FPGA Design Using High-Level Synthesis* : https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf
- *Vivado Design Suite User Guide: High-Level Synthesis* : https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug902-vivado-high-level-synthesis.pdf#nameddest=xApplyingOptimizationDirectives
- *Vivado HLS Optimization Pragmas* : https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/hls-pragmas-okr1504034364623.html

10.4 Xillybus & Xillinux:

- *Getting started with Xillinux for Zynq-7000* : http://xillybus.com/downloads/doc/xillybus_getting_started_zynq.pdf
- *Getting started with Xillybus on a Linux host* : http://xillybus.com/downloads/doc/xillybus_getting_started_linux.pdf
- *Xillybus Block Design flow* : http://xillybus.com/downloads/doc/xillybus_block_design_flow.pdf
- *Creating a custom Xillybus IP core* : http://xillybus.com/downloads/doc/xillybus_custom_ip.pdf
- *Xillybus host application programming guide for Linux* : http://xillybus.com/downloads/doc/xillybus_host_programming_guide_linux.pdf