INAUGURAL - DISSERTATION
zur
Erlangung der Doktorwürde
der
Naturwissenschaftlich-Mathematischen Gesamtfakultät
der
Rupert - Karls - Universität
Heidelberg

vorgelegt von
M.Sc. Yuning Yang
aus Henan, VR China

Tag der mündlichen Prüfung:

Design and Implementation of a Scalable Hardware Platform for High Speed Optical Tracking

Gutachter: Prof. Dr. Reinhard Männer

Zweiter Gutachter:

Abstract

Optical tracking has been an important subject of research since several decades. The utilization of optical tracking systems can be found in a wide range of areas, including military, medicine, industry, entertainment, etc.

In this thesis a complete hardware platform that targets high-speed optical tracking applications is presented. The implemented hardware system contains three main components: a high-speed camera which is equipped with a 1.3M pixel image sensor capable of operating at 500 frames per second, a CameraLink grabber which is able to interface three cameras, and an FPGA+Dual-DSP based image processing platform. The hardware system is designed using a modular approach. The flexible architecture enables to construct a scalable optical tracking system, which allows a large number of cameras to be used in the tracking environment.

One of the greatest challenges in a multi-camera based optical tracking system is the huge amounts of image data that must be processed in real-time. In this thesis, the study on FPGA based high-speed image processing is performed. The FPGA implementation for a number of image processing operators is described. How to exploit different levels of parallelisms in the algorithm to achieve high processing throughput is explained in detail. This thesis also presents a new single-pass blob analysis algorithm. With an optimized FPGA implementation, the geometrical features of a large number of blobs can be calculated in real-time.

At the end of this thesis, a prototype design which integrates all the implemented hardware and software modules is demonstrated to prove the usability of the proposed optical tracking system.

Zusammenfassung

Optisches Tracking ist seit vielen Jahren ein wichtiger Forschungsgegenstand. Anwendungen optischer Trackingsysteme können in vielen Gebieten gefunden werden, unter anderem in militärischen, medizinischen und industriellen Systemen sowie der Unterhaltungsindustrie.

In dieser Arbeit wird eine komplette Hardware-Plattform vorgestellt, die optisches Tracking bei hohen Frameraten ermöglicht. Das entwickelte Hardwaresystem besteht aus 3 wichtigen Teilen: Eine Hochgeschwindigkeitskamera, die mit einem 1,3-Megapixel-Bildsensor ausgestattet ist und bis zu 500 Bilder pro Sekunde liefern kann; ein Camera-Link Grabber, an den drei Kameras angeschlossen werden können sowie ein FPGA+Dual-DSP-Board zur Bildverarbeitung. Das Hardwaresystem ist modular aufgebaut. Die flexible Architektur ermöglicht es, ein skalierbares optisches Trackingsystem mit einer großen Anzahl an Kameras zu erstellen.

Eine große Herausforderung in Multi-Kamera-Systemen stellt die Datenmenge dar, die in Echtzeit verarbeitet werden muss. In dieser Arbeit werden FPGA-basierte Systeme zur Hochgeschwindigkeitsbildverarbeitung untersucht. Etliche FPGA-Implementierungen für Bildverarbeitungsoperatoren werden beschrieben. Es wird detailliert darauf eingegangen, wie verschiedene Ebenen der Parallelität in den Algorithmen ausgenutzt werden können, um eine hohe Datendurchsatzrate zu erreichen. Des weiteren wird in der Arbeit ein neuer Single-Pass-Algorithmus zur Blob-Analyse präsentiert. Mit Hilfe einer optimierten FPGA-Implementierung können geometrische Eigenschaften einer großen Anzahl an Blobs in Echtzeit berechnet werden.

Zum Abschluß der Arbeit wird ein Prototyp vorgestellt, der alle entwickelten Hard- und Softwaremodule vereint und so die Nützlichkeit des vorgestellten Trackingsystems zeigt.

Acknowledgements

A lot of people have contributed either directly or indirectly to the work of this thesis. It is an honor for me to acknowledge their efforts.

First and foremost, I want to express my sincerest gratitude to my advisor, Prof. Dr. Reinhard Männer, who provided me the financial support and guidance throughout my research.

My thanks go to all my colleagues at Heidelberg University, especially Andreas Wurz and Wenxue Gao, from whom I always got valuable answers to my technical questions.

I also want to sincerely thank Christiane Glasbrenner and Andrea Seeger for their everyday support.

I own many thanks to my friends, Paul Jarmola and Dr. Markus Adameck, for proof-reading my thesis.

Lastly, no words can express my deep gratitude to my family - my parents, my wife and my daughter, for their love and encouragement throughout my life.

Contents

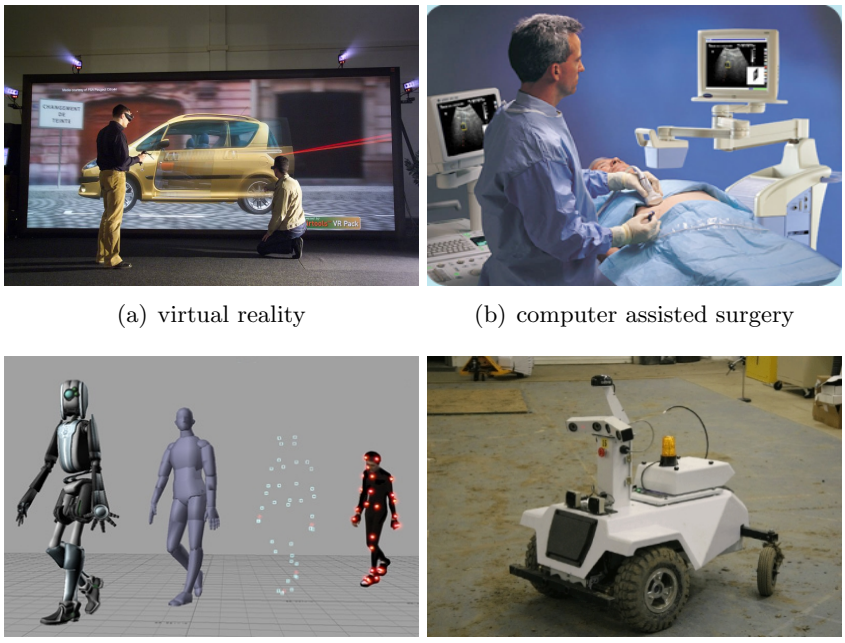
List of Figures

List of Tables

1. Introduction

1.1. Tracking Technologies

Tracking or motion tracking is the process of determining the position and orientation of moving objects in three-dimensional space. It has been an active, interesting and important subject of research since several decades. The utilization of tracking systems is nowadays necessary in a wide range of areas, including military, medicine, industry and entertainment. For example, in the well-known Virtual Reality (VR) and Augmented Reality (AR) applications, the position and orientation of user's head must be obtained to calculate the correct perspective of the world from the user's point of view. Additionally, one or both of the user's hands are tracked to provide the capability of 3D interaction [MG96]. In medical applications, such as Computer Assisted Surgery (CAS), the location and the angle of surgery instruments are determined by a tracking device, sometimes also the motion of the patient needs to be tracked to provide the surgeon with real-time guidance during the medical intervention. In Figure 1.1 some other applications of tracking systems are illustrated.



(a) virtual reality (b) computer assisted surgery

(c) human motion analysis (d) robot navigation

Figure 1.1.: Example applications of tracking systems

The specification of a point in 3D space requires three coordinates (x, y, z). However, rather than using points, many applications also need to have the orientation of the target specified by three angles known as *pitch, roll* and *yaw*, see Figure 1.2. Thus a tracking task often requires six degrees of freedom (6 DoF) [Han93].

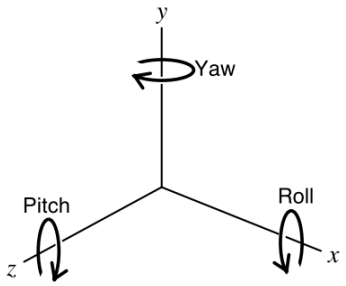


Figure 1.2.: 6 DoF in position and orientation

To obtain the 6 DoF information of one or multiple objects, there are mainly five tracking technologies in use today: Mechanical, Electromagnetic, Acoustic, Inertial and Optical [MAB92].

Mechanical - A mechanical tracker makes physical connections between the tracked object and the system. Typically potentiometers or optical encoders are used to measure the rotation of a joint between rigid linkages. Given the angle of each joint and length of the rods or wires, together with the known position of the fixed hardware, it is possible to calculate the position of the tracked object [BS05].

Electromagnetic - An electromagnetic tracker comprises a transmitter and a receiver. An oscillating magnetic field generated in the three orthogonal coils of the transmitter is sensed by the receiver in three corresponding coils. The position and orientation of the object being tracked can be calculated by measuring the intensities of the received magnetic field [MAB92].

Acoustic - In acoustic tracking, one or more emitters are mounted on a target. They send out ultrasonic pulses, which are received by several sensors (microphones). The time taken for the sound pulses to reach the sensors is measured and the distance is calculated based on the speed of sound in air [MAB92]. The location and the orientation of the target can then be triangulated from these measurements.

Inertial - Inertial trackers make use of accelerometers and gyroscopes to compute the relative change in position and orientation from the appearing acceleration and angular velocity in the moving target with respect to an inertial reference coordinate system. With a known absolute start position and start orientation the actual position and orientation of the target can be determined [LKPB02].

Optical - Optical tracking utilizes optical sensors, typically video cameras, to detect visual features associated with the tracked objects and so to determine their positions in 3D space. Usually markers, such as light emitters (active markers) or light reflectors

(passive markers), are attached to the targets to make them more distinguishable. Using image processing technologies, the 2D location of markers in each image captured by the cameras can be obtained. The position and orientation of the targets are then calculated based on the geometric relationship between the 2D marker location and the 3D camera position known a priori. Figure 1.3 illustrates a typical setup of an optical tracking system.

Figure 1.3.: Typical setup of an optical tracking system

Compared to other tracking technologies, optical tracking is more widely used in many areas, especially VR/AR, surgery guidance and motion analysis. The main advantages of optical tracking systems lie in the fact that they are

- inherently more accurate and reliable as optical sensors are not subject to distortions due to ferromagnetic metals, like electromagnetic techniques, or from drift problems, like inertial sensors.

- much less intrusive since the markers attached to the objects are normally very small and lightweight.

- capable of allowing larger tracking volume and higher update rate, as well as tracking many objects simultaneously.

Optical tracking systems can be divided into two categories: inside-out and outside-in [Meh06]. The schematic diagram is shown in Figure 1.4.

- Inside-out - In inside-out tracking the cameras are placed on the object being tracked and observe features in the surrounding environment. Markers are placed

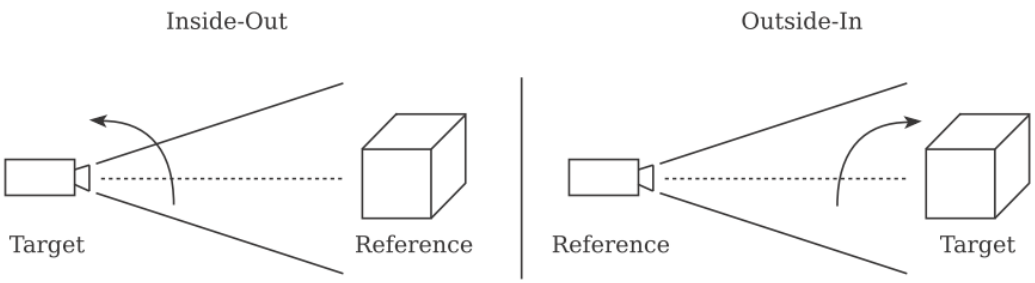


Figure 1.4.: Inside-out and outside-in optical tracking systems

at statical positions that can be observed by the cameras. For VR/AR applications inside-out tracking is frequently used.

- Outside-in - In outside-in tracking the cameras are mounted at fixed positions and are oriented towards the objects to be tracked. The objects move freely in the tracking volume, which is determined by the visible ranges of the cameras. Outside-in tracking is widely utilized in human motion analysis and surgery guidance applications where objects to be tracked must be allowed to move with minimum limitation.

1.2. Objective

Optical tracking also suffers from some disadvantages. One of the major problems faced by most optical tracking systems is occlusion. This problem happens when one or multiple markers are partially or totally invisible from cameras, resulting in insufficient information for calculating the position of the objects to be tracked.

An effective solution to the occlusion problem is to introduce more cameras into the tracking environment, or using a *many-camera* tracking system. When occlusion occurs from one view, robust tracking can still be achieved using other views' information if the occluded markers are visible to other cameras. Meanwhile, some other advantages can be gained, e.g. larger tracking range. As a camera has a fixed number of pixels, one can either achieve larger visible range with lower resolution using wide angle lens, or smaller visible range with higher resolution using narrow angle lens, but not both. Therefore, larger tracking range for a given resolution can only be obtained by combining the visible ranges of increased number of cameras placed at different locations [Che02]. Furthermore, the overall system robustness can be improved due to the redundancy provided by a many-camera system. For example, when one or multiple cameras stop functioning due to some special reasons (such as power failure), tracking can still proceed as long as a sufficient number of cameras are working in the tracking volume. In order to implement the above described concept, the system must have a

high degree of flexibility. For instance, adding or removing cameras to/from the tracking environment should not involve a great deal of effort. This requires the tracking system to be scalable.

The main purpose of this thesis is to develop an optical tracking system with a scalable architecture, which allows a large number of cameras to be easily integrated into the tracking environment. In addition, this system should fulfill the following requirements:

- **High Accuracy** - Accuracy is measured by the maximum error between the reported position and the real position. In the case of VR and AR, normally the positioning error should not exceed 0.5mm and the orientation error should be below 0.1 degree [WF02]. For medical applications, such as CAS, more decent tracking accuracy is required.

- **High Update Rate** - Update rate is defined as the amount of measurements that can be performed per second (measured in Hz). The maximum update rate of applications with normal real-time requirement, like VR/AR, is mostly limited within 60Hz. Nowadays the demand for high-speed tracking ($\geq$ 200Hz) is substantially growing because tracking at high update rate enables the possibility to capture details of fast movements.

- **Low Latency** - Latency is the time lag between the moment a target's movement occurs and the moment it is reported. It determines the responsiveness of the tracking system. The need for tracking with low latency ($<$ 5ms) becomes more and more critical, because the effectiveness of many interactive applications (such as surgery navigation) depends highly on how quickly the system can respond to the movement of the target being tracked.

To achieve high accuracy, it is often necessary to use high resolution ($\geq 1M$ pixel) cameras, since poor resolution leads to larger error in the measurement of 2D target position on the image plane of cameras, which can significantly degrade the accuracy of 3D target location. The update rate of an optical tracking system is mainly limited by the camera frame rate and the system processing capability. With the dramatic improvement of image sensor technologies, more and more high-speed cameras are available today for use in applications requiring high update rate. Latency is often related to update rate. However, high update rate does not necessarily mean low latency. In order to achieve low tracking latency, it is also required to buffer as few as possible image data during the processing.

One of the greatest challenges for developing such a system is how to handle the large amount of image data produced by multiple high resolution cameras running at high frame rate. Suppose that there are three cameras working in a typical tracking scenario, each of which is running at 200Hz with a resolution of one mega pixel (1024$\times$1024).

These cameras yield a total of 600 Mega pixels per second. With the increment of the number of cameras, the total input data rate can quickly reach multi-gigabyte-per-second. Considering that most computer vision algorithms are computationally expensive, even the up-to-date high-performance PC can not handle such a data rate in real time. Finding a solution to process the huge amount of image data at high speed is a major objective of this thesis.

1.3. Existing Optical Tracking Systems

Optical tracking has been a topic both in industry and in academic research since many years. In this section some representative optical tracking systems are presented. Since the work of this thesis focuses on outside-in tracking, only this type of systems are covered. For more comprehensive overview of currently available optical tracking systems, refer to [Rib01] and [Bua05].

1.3.1. Commercial Systems

Vicon Tracker - Vicon Motion Systems Ltd provides high-end solutions for optical tracking. Vicon Tracker is a passive infrared marker tracking system that consists of multiple cameras equipped with IR LEDs and IR optical filters, and a set of retro-reflective markers positioned on the objects to be tracked. Retro-reflective markers are passive markers, which do not generate light themselves. They are normally coated with retro-reflective material that reflects the IR radiation from the IR LEDs into the direction of the incoming radiation. The IR-pass optical filter mounted on the camera lenses filters out other spectrum and only keeps the one reflected by the markers. As a result, markers will appear as bright spots in the image of the cameras which can easily be detected.

Figure 1.5.: Vicon MX camera

The system calculates the center of each marker and reconstructs its 3D position in the tracking volume. At least 4 markers and 3 cameras are required to provide 6 DoF

information. The Vicon cameras shown in Figure 1.5 are designed, developed and built specifically for motion tracking applications. The Vicon MX13 camera, for example, is equipped with a CMOS image sensor which works in infrared (875nm wavelength) region. The frame rate with full resolution (1280 × 1024) is 482fps. The on-board processing capability allows complex marker detection algorithms to be performed in real-time.

The independently measured positional accuracy (average absolute error through a large 3D space) reported by the vendor is 0.1mm and the angular accuracy is 0.15 degree. The overall tracking latency is limited within 10ms in general. The update rate ranges from 200Hz to 1000Hz depending on the resolution of the camera.

ARTTrack System - Similar to Vicon Tracker, the ARTTrack System developed by Advanced Realtime Tracking GmbH is an infrared-based marker tracking system. The system utilizes two or more cameras to obtain the tracking information.

All ART tracking cameras are equipped with a low-noise CCD sensor and embedded processors to accelerate the analysis of the marker data. The cameras also have built-in infrared flashes (880nm wavelength) to illuminate the tracked objects. Flashes are synchronized by an external sync signal, which is provided to each camera. One ARTTrack system can contain maximumly 16 cameras. Spherical retro-reflective markers are used in the system. To get the 6 DoF information, at least 4 markers must be attached to the object.

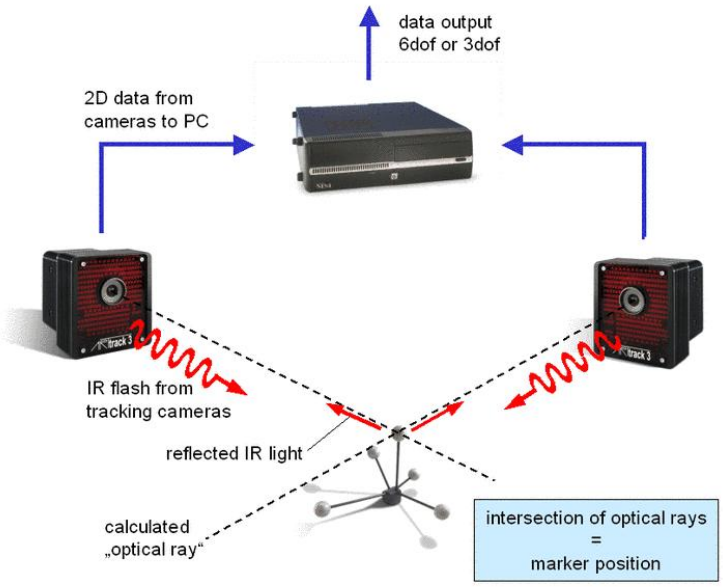


Figure 1.6.: ARTTrack System

The maximum update rate of the ARTTrack System is 60Hz. According to the product information from the vendor, the accuracy results are 0.4mm in position estimation and 0.12 degree in orientation estimation.

Raptor-4 Digital RealTime System - Like Vicon Tracker and ARTTrack, the Raptor-4 Digital RealTime System from Motion Analysis Corporation is a passive optical tracking system based on retro-reflective markers.

The system consists of the Raptor-4 digital cameras and the Cortex software, which capture complex motion at high accuracy. The Raptor-4 Digital Camera has a CMOS image sensor with a large pixel array (2352×1728). Each camera is equipped with 323 LEDs around the lens. The powerful on-board processing capability allows the system to operate at 166Hz with full resolution and up to 10,000Hz with partial resolution.

The accuracy data is not directly provided by the vendor. However, according to the technical specification, Raptor-4 is the only optical tracking system that satisfies the accuracy requirement of the broadcast tracking applications - 1/100th of a degree.

Figure 1.7.: Raptor-4 Digital Camera

Optotrak Certus Motion Capture System - The Optotrak certus motion capture system is a 6 DOF motion measurement system made by Northern Digital Inc. Unlike Vicon, ARTTrack and Raptor-4 trackers Optotrak makes use of active marker technology. Active markers are infrared light emitting devices, mostly using LEDs. The disadvantage of active markers is that each marker requires wires and electronic circuits, making them less compact than passive markers. However, active markers usually appear as even brighter spots in the captured images than passive markers and are thus more easily detectable.

In the Optotrak system, synchronized markers are placed on the moving objects which are tracked by three cameras mounted on a rigid base. Each camera is equipped with an infrared optical filter. The active markers (LEDs) are connected to a central control unit, which turns on and off the LEDs in sequence. In this way, the LED that has been activated can be identified by the system at any point in time.

To get the 6 DOF information, at least one LED must be visible for the position estimation and at least three must be visible for the orientation estimation. Maximum number of markers supported by the system is 512. The maximum marker scanning frequency reaches 4600Hz, which corresponds to about 1500Hz of overall system frame rate. According to the technical specifications provided by the vendor, the 3D accuracy in position is 0.1mm. There was no information available regarding the accuracy in

orientation.

Figure 1.8.: Optotrak Certus Motion Capture System

MicronTracker - MiconTracker is a real-time sub-millimeter 6 DOF optical tracking system designed and manufactured by Claron Technology Inc. Unlike all the above introduced optical tracking systems, which use infrared passive or active markers, MicronTracker are fully passive, using visible light to detect and track objects.

The key technology of MicronTracker is to mark objects by small checkered target regions called Xpoints. Each Xpoint contains an intersection of 4 high-contrast black and white regions as shown in Figure 1.9. Advanced computer vision algorithms are used to detect the Xpoint and calculate the position of the intersection point. Since each of the four boundary lines of the Xpoint independently serves to pinpoint the location of the target, portions of the Xpoint region hidden by smudges does not strongly affect the accuracy. As a result, the MicronTracker provides more robustness than traditional infrared tracking systems. Additionally Xpoints contain both location and orientation information which greatly reduces mismatches between targets seen by different cameras. This is also a clear advantage against infrared markers, which do not contain any geometrical information other than the locations of their center.

The tracking accuracy of the MicronTracker is guaranteed by a very accurate detection algorithm and high precision custom calibration of the camera. According to the manufacture specifications, the static jitter (measured by single target at a distance of 75cm) reaches 0.007mm RMS (Root Mean Square). However, MicronTracker does not outperform infrared tracking systems in terms of update rate and latency, since all complicated detection algorithms are done by software. For instance, the maximum measurement rate supported by MicronTracker(Sx60 Model) is 48Hz. The overall system latency reported by the manufacture is around 30ms.

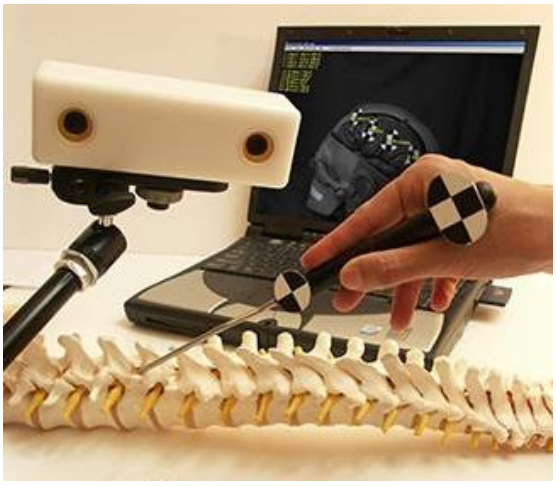


Figure 1.9.: MicronTracker

1.3.2. Systems Developed in Academic Research

POSTTRACK - POSTTRACK is an optical motion tracking system presented by Chung *et al.* [CKKP01]. This system utilizes four gray-scale cameras, each of which is equipped with IR illuminating LEDs and an IR pass optical filter. The user is required to wear one or more retro-reflective markers. A marker needs to be seen by at least two of the four cameras to make the tracking possible. The Open source computer vision libraries (OpenCV) is used for camera calibration, which includes the calculation for camera intrinsic and extrinsic parameters by having the four cameras reference on known visual features. After calibrating the cameras, the 2D locations of the center of gravity of the markers are calculated. This step is followed by matching the markers between the four captured images. Afterwards the 3D position of each marker are computed.

POSTRACK makes use of standard PC for all the calculations required by the tracking algorithms, which greatly limits the system performance in terms of tracking update rate. For example, the achievable update rate can only reach 15Hz.

Figure 1.10.: Camera and markers used in POSTRACK

Personal Space Station - Jurriaan D. Mulder *et al.* [MJvR03] presented a low

cost optical head tracking system for desktop VR/AR applications called Personal Space Station. Two FireWire cameras with VGA resolution (640×480) image sensor are utilized, which are available at a low price of less than 100Euro. Personal Space Station does not make use of IR illuminations and retro-reflective markers. Instead, a marker pattern consisting of 3 black circular dots on a white background arranged in a triangular form is used in the system. The camera calibration and 3D reconstruction algorithms are similar to those utilized by the POSTTRACK. The system provides an update rate of 30Hz and a delay of 66 ms.

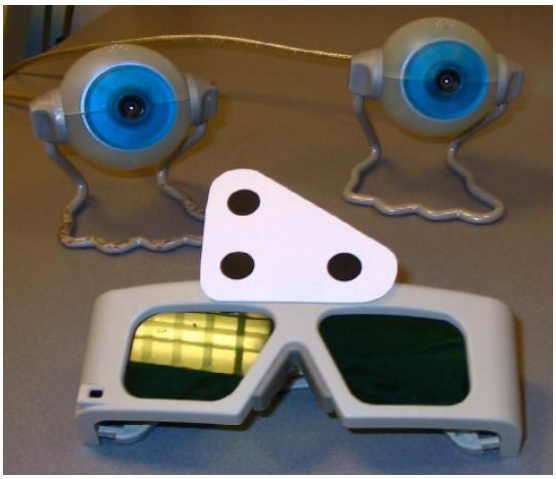


Figure 1.11.: Personal Space Station hardware: cameras and the dot pattern marker

Cyclope Tracker - The innovation introduced by the Cyclope tracker [Mat05] is that it is a 6 DOF optical tracking system based on a single camera. The camera is equipped with infrared LEDs mounted on the lens. Retro-reflective markers that have a pre-defined pattern are used. To build a 3D target, 4 markers must be fixed on a rigid structure. Given the 3D geometric configuration of 4 markers and their 2D positions in the image, Cyclope tracker calculates the position and the orientation of the reference frame attached to the markers with respect to the camera reference frame.

Figure 1.12.: Cyclope camera

Up to 4 targets can be simultaneously tracked by Cyclope Tracker. The maximum system update rate is 60Hz. The latency is limited within 40ms. The achievable accuracy is 1.1mm in translation and 0.3 degree in rotation at the depth of 1.5 meter.

None of the optical tracking systems introduced above demonstrates a highly scalable architecture while simultaneously provides the features of high accuracy, high speed as well as low tracking latency, which is the purpose of the development of this thesis.

1.4. Contribution

The main contribution of this thesis includes the development of a hardware platform that satisfies the high demands of a scalable optical tracking system and the study on performing high-speed image processing tasks.

The hardware system consists of multiple high-speed cameras with mega pixel resolution and an image processing platform. The camera features a modular architecture, which allows a wide variety of image sensors to be equipped, adding more flexibility to the system. The image processing platform employs FPGA (Field Programmable Gate Array) and DSPs (Digital Signal Processors) to provide the capability of processing large amount of image data in real time.

Parallel implementation for a number of image processing algorithms on the FPGA has also been explored, including color conversion, noise reduction, edge detection, morphological filter and blob analysis. All the implemented algorithms are capable of processing incoming image data on-the-fly, which not only fulfills the high data rate requirement but also guarantees low system latency.

At the end of this thesis, a prototype optical tracking system is demonstrated to prove the usability of the proposed concept.

1.5. Thesis Outline

The remainder of this thesis is organized as follows: Chapter 2 briefly introduces the mathematical fundamentals required for optical tracking. Chapter 3 describes the hardware design of the proposed optical tracking system in detail. Chapter 4 deals with the FPGA implementation of 2D image processing algorithms that are frequently used in optical tracking. In Chapter 5 a prototype design that integrates all implemented hardware and software modules is presented to demonstrate the feasibility and performance of the proposed tracking system. Chapter 6 summarizes the work done within this thesis and discusses some directions for future developments.

2. Fundamentals

Optical tracking is a complicated problem covering a wide range of aspects of computer vision. This chapter attempts to provide a conceptual overview of the entire tracking process. The mathematical background necessary to understand the problems presented in different tracking steps is briefly introduced. Throughout this chapter, where not explicitly mentioned, the book *Multiple View Geometry in Computer Vision* [HZ04] is used as a reference.

The overall process of optical tracking contains four main stages:

1) *Camera Calibration* - Establishing an accurate model of the cameras used in the tracking environment.

2) *2D Feature Extraction* - Identifying and locating features of objects to be tracked.

3) *Correspondence Matching* - Matching features associated with the same object among images captured by each camera.

4) *3D Reconstruction* - Calculating 3D coordinates of the tracked objects.

2.1. Camera Calibration

Camera calibration is an essential part of optical tracking where the relationship between the 3D world defined by the physical tracking area and the 2D image plane defined by the image captured by each camera is determined.

2.1.1. Pinhole Camera Model

The pinhole camera model is the simplest and an ideal camera model that is suitable for many computer vision and computer graphics applications. It defines a geometric mapping between the 3D world and a 2D image.

As illustrated in Figure 2.1, a pinhole camera is modeled by its optical center C and the image plane R. The line through C and orthogonal to R is called the optical axis. The point at which the optical axis intersects R is referred to as the principal point p_0. f represents the focal length which is determined by the distance between C and R.

A point in 3D space M is mapped to an image point m where the line through C and M intersects with R. Using similar triangles shown in Figure 2.2, we can derive

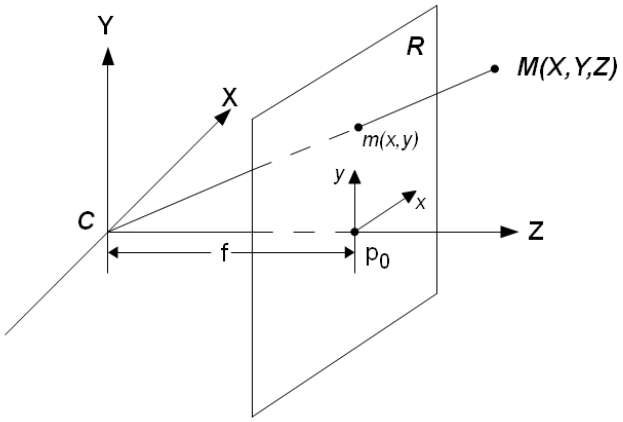


Figure 2.1.: Pinhole camera model

that the point $M(X, Y, Z)^T$ is mapped to the point $m(fX/Z, fY/Z, f)^T$ on the image plane.

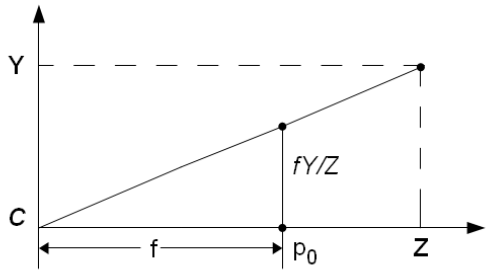


Figure 2.2.: Similar triangles of a pinhole camera model

Ignoring the final image coordinate, we obtain the following mapping relationship:

$$(X, Y, Z)^T \mapsto (fX/Z, fY/Z)^T \tag{2.1}$$

If we use homogeneous vectors to represent the world and image points, then Equation 2.1 can be written in terms of matrix multiplication as

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \mapsto \begin{pmatrix} fX \\ fY \\ Z \end{pmatrix} = \begin{bmatrix} f & & & 0 \\ & f & & 0 \\ & & 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{2.2}$$

Let M represent the homogeneous vector of a world point in 3D-space $(X, Y, Z, 1)^T$ and m represent the homogeneous vector of an image point in 2D-space, we can write Equation 2.2 compactly as

$$m = PM \tag{2.3}$$

where P is a 3×4 matrix called camera projection matrix. It can be expressed as

$$P = \begin{bmatrix} f & & \\ & f & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & 0 \\ & 1 & & 0 \\ & & 1 & 0 \end{bmatrix} = K[I_3|0] \tag{2.4}$$

The 3×3 matrix K in Equation 2.4 is called the camera calibration matrix. So far, it solely depends on the focal length f.

In Equation 2.1 we assumed that the origin in the image space is at the principal point. In practice, the origin of an image usually lies in the top-left corner of the image. Thus, we can write Equation 2.1 more generally as

$$(X, Y, Z)^T \mapsto (fX/Z + u_0, fY/Z + v_0)^T \tag{2.5}$$

where $(u_0, v_0)^T$ are the coordinates of the principal point. Equation 2.5 can be expressed in homogeneous coordinates as

$$M = \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \mapsto m = \begin{pmatrix} fX + Zu_0 \\ fY + Zv_0 \\ Z \end{pmatrix} = \begin{bmatrix} f & & u_0 & 0 \\ & f & v_0 & 0 \\ & & 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{2.6}$$

Now we can refine the camera calibration matrix K to

$$K = \begin{bmatrix} f & & u_0 \\ & f & v_0 \\ & & 1 \end{bmatrix} \tag{2.7}$$

Then Equation 2.6 can be rewritten as

$$m = K[I|0]M \tag{2.8}$$

In general, points in space will be expressed in terms of a world coordinate system. The camera coordinate system and the world coordinate system are related via a rotation and a translation, see Figure 2.3. Now let M be a 3-vector representing the coordinates of a point in the world coordinate frame and M' be a 3-vector representing the same point in the camera coordinate frame. We can express M' by

$$M' = R(M - C) \tag{2.9}$$

where R is a 3×3 rotation matrix that represents the orientation of the camera

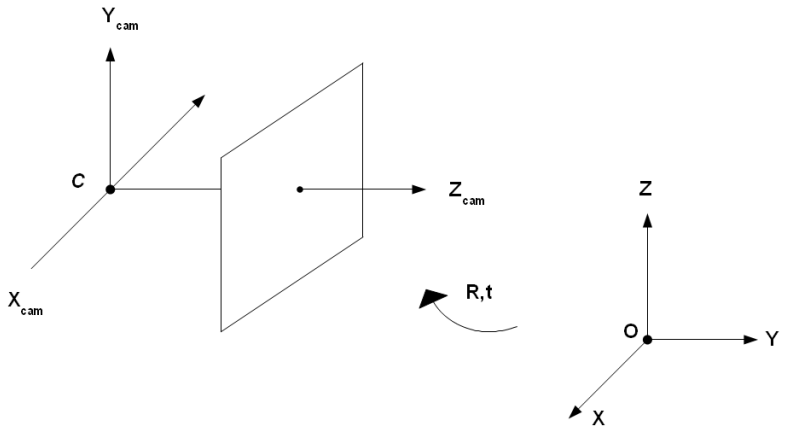


Figure 2.3.: Transformation between the camera and world coordinate frames

coordinate frame and C represents the coordinates of the camera center in the world coordinate frame. Using homogeneous coordinates, Equation 2.9 can be written as

$$M' = \begin{bmatrix} R & -RC \\ 0 & 1 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \begin{bmatrix} R & -RC \\ 0 & 1 \end{bmatrix} M \tag{2.10}$$

Combining this together with Equation 2.8, we get

$$m = KR[I|-C]M \tag{2.11}$$

where M is in a world coordinate system.

Until now it is assumed that the image coordinates are scaled by the same factor in both horizontal (x) and vertical (y) directions. However we must consider the fact that pixels of a real CCD or CMOS camera are not squared, which means that we have unequal scale factors between x and y directions of the image when measuring point coordinates in pixels. Let m_x and m_y be the number of pixels per unit distance in the x and y direction in image coordinates, then we can write the camera calibration matrix in the general form as

$$K = \begin{bmatrix} \alpha_x & & u_0 \\ & \alpha_y & v_0 \\ & & 1 \end{bmatrix} \tag{2.12}$$

where $\alpha_x = fm_x$ and $\alpha_y = fm_y$ represent the focal length in the x and y direction in terms of pixel dimensions.

There is one more parameter we need to consider, which is referred to as the *skew* parameter s. For most normal cameras, s will be zero. However, it can take non-

zero values under unusual conditions, e.g., the image plane is not perpendicular to the optical axis. By adding the skew parameter s, the camera calibration matrix can be extended as

$$K = \begin{bmatrix} \alpha_x & s & u_0 \\ & \alpha_y & v_0 \\ & & 1 \end{bmatrix} \tag{2.13}$$

Based on Equation 2.11, we can generalize the projection matrix P to the form

$$P = KR[I|-C] \tag{2.14}$$

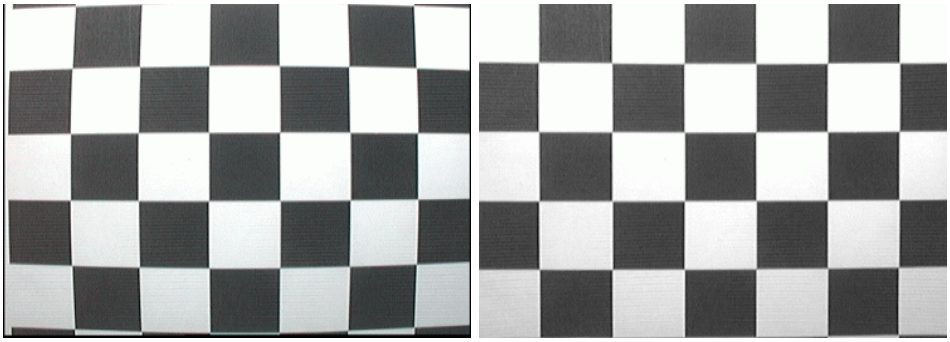
or

$$P = K[R|t] \tag{2.15}$$

where $t = -RC$. This is the general mapping given by a pinhole camera. In Equation 2.15, the parameters contained in K are called the *internal* camera parameters, and the parameters determined by R and C are called *external* camera parameters.

2.1.2. Lens Distortion

Lens distortion is an unavoidable artifact that can be found in any camera image. Even cameras equipped with high quality lens are subject to some level of optical distortion. In Figure 2.4 a raw image with distortion and the image after rectification are shown.



(a) distorted image (b) undistorted image

Figure 2.4.: Image distortion

Lens distortion can be modeled by a combination of the radial and the tangential distortion [HS97b]. The radial distortion is often approximated using the following expression

$$\begin{pmatrix} \hat{u} \\ \hat{v} \end{pmatrix} = \begin{pmatrix} u(k_1 r^2 + k_2 r^4 + \cdots) \\ v(k_1 r^2 + k_2 r^4 + \cdots) \end{pmatrix} \tag{2.16}$$

where u and v are image coordinates, $\hat{u}$ and $\hat{v}$ are distorted image coordinates, k_i are

radial distortion coefficients and $r = \sqrt{u^2 + v^2}$.

The tangential distortion vector can be expressed by

$$\begin{pmatrix} \hat{u} \\ \hat{v} \end{pmatrix} = \begin{pmatrix} 2p_1uv + p_2(r^2 + 2u^2) \\ p_1(r^2 + 2v^2) + 2p_2uv \end{pmatrix} \tag{2.17}$$

where p_1 and p_2 are tangential distortion coefficients.

In an optical tracking system, where cameras are used for measurement purpose, a compensation for lens distortion must be performed to ensure reasonable tracking results.

2.1.3. Calibration

The purpose of camera calibration is to determine the internal and external parameters used in the camera model discussed above. Since the distortion parameters do not change when moving a camera, they are usually regarded as internal parameters.

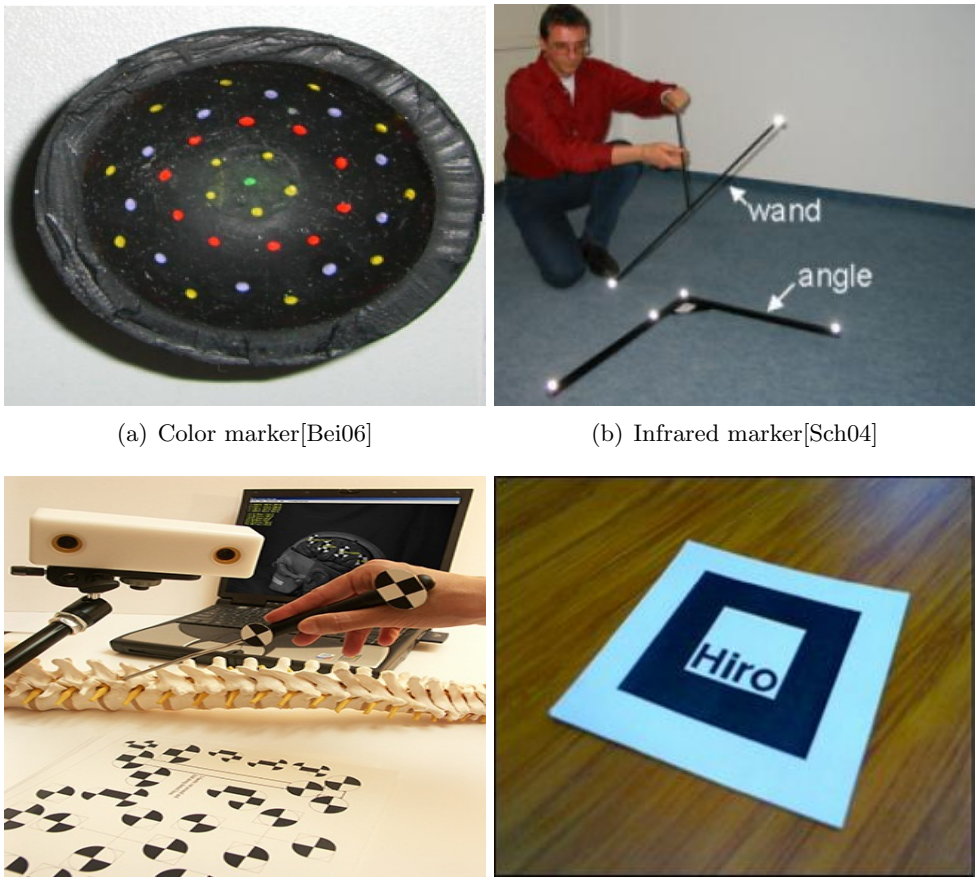
A wide range of different camera calibration methods have been reported in the literature. One widely used technique was developed by Zhang [Zha00]. Zhang's approach requires the camera to observe a planar pattern (normally chessboard pattern) shown at a few (at least two) different orientations. At first, corners of the checkerboard pattern are extracted and located with sub-pixel precision. Then the algorithm computes the projective transformation between the corner points of n different images. Afterwards, the camera internal and external parameters are recovered using a closed-form solution, while the third- and fifth-order radial distortion terms are recovered within a linear least-squares solution. A final nonlinear minimization of the re-projection error refines all the recovered parameters.

Since camera calibration is not the focus of this thesis, a more detailed introduction is not given here. For deeper discussions on this subject, refer to [Hem03].

2.2. 2D Feature Extraction

2D Feature extraction is a crucial step in optical tracking. Before the 3D coordinates of an object can be reconstructed, the 2D location of the object in images captured by multiple cameras must be known. The final accuracy of the tracking system is largely dependent on how precisely the object can be located in each image. Determining where in the image the object is requires an analysis of the image, which typically involves looking for the feature points.

Features are distinctive image points corresponding to objective 3D scene elements that are in most instances accurately locatable and recur in successive images, which make them explicitly trackable over time [TS04]. A wide variety of feature point or interest point detectors have been reported in the literature. A comprehensive overview



(a) Color marker[Bei06] (b) Infrared marker[Sch04]

(c) Xpiont marker[MT009] (d) 2D-barcode marker[ABD04]

Figure 2.5.: Example of markers used in optical tracking

of the current methods for feature extraction is provided in [SMB00]. To avoid complex and computationally expensive image processing, optical tracking systems often use specific or pre-defined markers that can easily be detected [LM03]. Figure 2.5 illustrates some examples.

Extracting feature points from an image is something human beings are naturally talented at, however for a computer it is a complex and computationally intensive process since it often involves analyzing a large amount of image data. When designing a solution to the problem of feature extraction, several aspects need to be considered, including accuracy, reliability and complexity, etc. There is no general solution for selecting the best features and determining the most appropriate feature extraction algorithm, since this problem is always application dependent.

2.3. Correspondence Matching

When more than one feature points are found in images, ambiguities may arise. This problem occurs when trying to establish the correspondence between feature points

in one image and another [Cou03]. Finding Correspondence is an important step in optical tracking. Prior to any 3D reconstruction operations, it is necessary to identify feature points in two or more views that represents the same point in the 3D space.

To solve the correspondence problem, one straightforward approach is to select a feature pixel in an image and then search through a 2D region around that pixel in the other image to find the corresponding point. The algorithm is based on the calculation of Sum-of-Absolute-Differences(SAD) or Sum-of-Squared-Differences (SSD), through which corresponding points between images can be obtained by finding the minimum SAD or SSD in an area-based block matching process [SSZ01]. This approach has the main drawback of high computational cost and low accuracy. As a consequence, it is rarely used in optical tracking systems.

Epiploar Geometry

The epipolar geometry [PS07] provides an alternative for finding the relationship between feature points in images captured by different cameras.

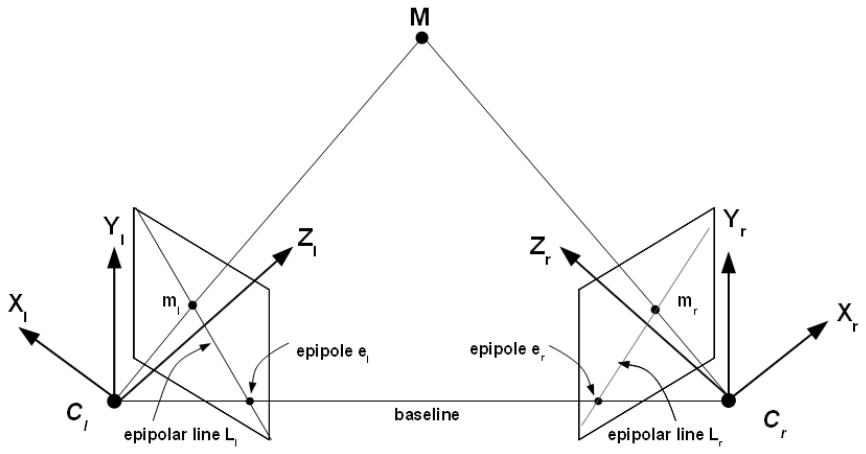


Figure 2.6.: Epipolar geometry

Consider a stereo setup composed by two pinhole cameras whose principal axes are non-colinear, as illustrated in Figure 2.6. Let C_l and C_r represent the optical centers of the left camera and the right camera respectively. A 3D point M is projected onto both image planes, resulting in the 2D point pair m_l and m_r. The epipolar plane is determined by the point M and the two camera optical centers C_l and C_r. Given m_l, its corresponding point in the right image is constrained to lie on a line called the epipolar line L_r, which is the intersection of the epipolar plane with the image plane of the right camera. And the same is valid for m_r, whose corresponding point must lie in the epipolar line L_l in the image plane of the left camera.

Based on this observation, we can make use of the *epipolar constraints* to find the correspondence between two feature points. The *epipolar constraints* is that, for a given

point p_1 in image 1, its possible matches in image 2 must lie on the epipolar line of p_1. As a result, the search space for a correspondence finding is reduced to one dimension. More detailed discussion on epiploar geometry can be found in [FP02].

2.4. 3D Reconstruction

Once the coordinates of the corresponding feature points from deferent views and the camera parameters are known, we are ready to calculate the position of the point in 3D space. This process is referred to as 3D reconstruction, or sometimes triangulation.

In Figure 2.6 we see that for two image points m_l and m_r in two different views that represent the same point in 3D space M, there exist two back projected rays that pass through the corresponding image center and image point intersecting at M. Or in other words, there will be a 3D point M that gives $m_l = P_l M$ and $m_r = P_r M$, where P_l and P_r represent the projection matrix of the left and the right camera respectively.

This however, is the ideal case. In practice, due to inaccuracies in the camera calibration and inherent localization errors of the feature points, the back projected rays will not exactly intersect at the point M, as shown in Figure 2.7.

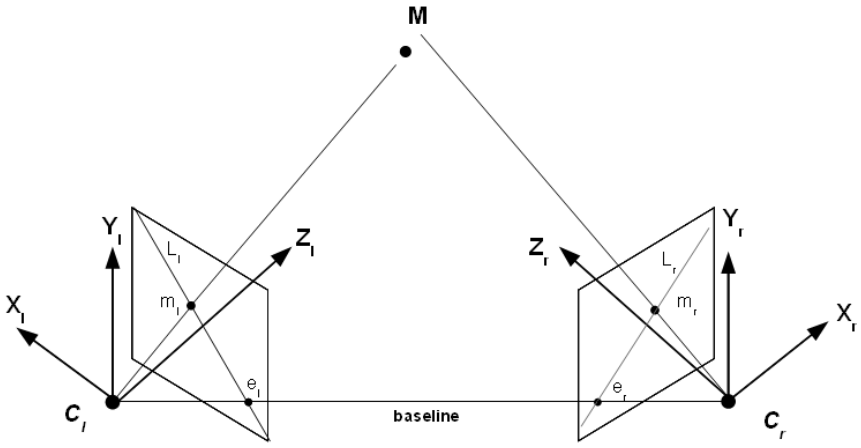


Figure 2.7.: Epipolar geometry with measurement errors

The idea of 3D triangulation is to estimate a point $\hat{M}$, that satisfies

$$\hat{m}_l = P_l \hat{M} \text{ and } \hat{m}_r = P_r \hat{M} \tag{2.18}$$

And $\hat{M}$ is estimated so that it minimizes the reprojection error, which can be calculated by the summed squared distances between the projection points $\hat{m}_l$ and $\hat{m}_r$ of $\hat{M}$ and the measured image points m_l and m_r.

There exist a large number of reconstruction approaches. A comprehensive survey is given in [HS97a].

2.5. Summary

This chapter briefly introduces the mathematical fundamentals that are necessary to understand the entire process of optical tracking, which mainly include camera calibration, 2D feature extraction, correspondence matching and 3D reconstruction. As can be concluded, optical tracking is a very comprehensive subject. This thesis does not intend to address all problems presented in every aspect of optical tracking. The research of this thesis focuses on the hardware system design and high-speed image processing for the 2D feature extraction.

3. Hardware Design

This chapter introduces the hardware architecture and the design methodology of the implemented optical tracking system. The hardware design in this thesis includes the development of a high-speed camera, a CameraLink simulator, two different types of CameraLink grabber and a multi-purpose image processing platform called *PowerEye*. The following sections first explain the considerations behind the design decisions and then describe each hardware component in detail.

3.1. Design Considerations

3.1.1. Choice of Processors

As mentioned in previous chapters, optical tracking is a very computationally expensive process. Over the past decades, optical tracking systems have taken advantage of high-end host PC to handle image processing tasks. Today's high speed optical tracking applications, however, are making host PC performance reach its limits due to the huge amounts of image data to be processed. Special hardware resources are strongly needed to overcome the limitations of host PC.

At present, there are a number of processors available for high speed image processing. The most commonly used ones are: Application Specific Integrated Circuits (ASICs), Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs) and Digital Signal Processors (DSPs).

- **ASICs** - ASICs are usually dedicated for high volume production. These devices are customized for some particular use. Thus, they contain components that are very specific and necessary for performing only one given task [Art08]. Once manufactured, the functions of an ASIC can not be changed. Due to the true customized hardware architecture, ASICs enable highly optimized implementation of image processing algorithms at high clock rates. In most cases, ASICs provide the best performance in the matter of computational capability and power consumption.

- **GPUs** - GPUs were originally developed for video games, where massive floating-point operations need to be executed for real-time 3D graphics rendering. Nowadays using GPUs to accelerate 2D/3D image processing is becoming more and more popular. GPU computing is enabled by its massively parallel architecture

which consists of hundreds of processor cores operating together to crunch through the data set in the application. Figure 3.1 illustrates a typical schematic layout.

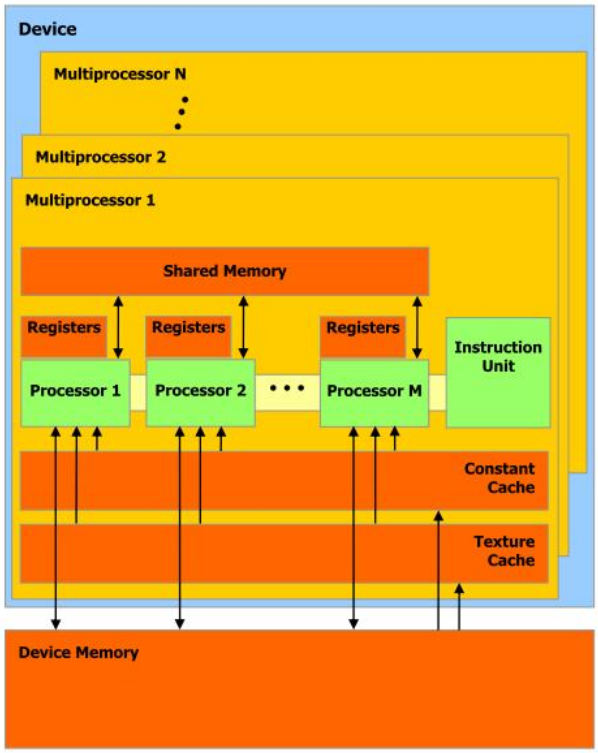


Figure 3.1.: GPU architecture [CU008]

It can be seen that there are many processing cores inside the GPU, each grouped into multiprocessors. Operations are performed by threads that are grouped into blocks, which are in turn arranged on a grid. Each block is executed by a single processor. If there are enough resources available, several blocks can be active at the same time on a processor. The processor will time-slice the blocks to improve performance, one block performing calculations while another is waiting for a memory read [AR08]. Clearly this high parallel processing architecture leads to a great enhancement in terms of computing performance. For instance, the state-of-the-art GPU Nvidia GTX295 features 240 processor cores with each operating at 1.2GHz. The peak performance has reached 2TFLOPS (Tera FLoating point Operations Per Second) [Lie09].

- **FPGAs** - FPGAs are devices that contain a huge number of logic elements, configurable interconnects (routing) and I/O blocks [Hed08, Mac05]. The name *field programmable* indicates that this kind of devices can be reconfigured by the designer after manufacturing, which makes them very useful in a wide range of applications. A typical structure of an FPGA is illustrated in Figure 3.2 .

  Each logic block can be programmed to perform both arithmetic and logical oper-

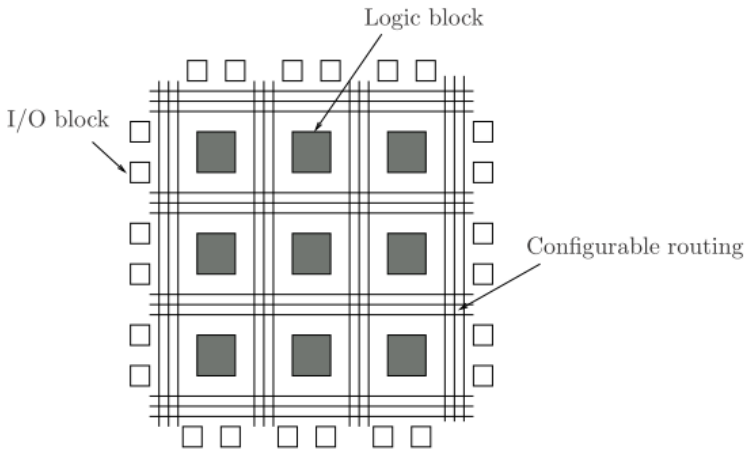


Figure 3.2.: Conceptual architecture of an FPGA

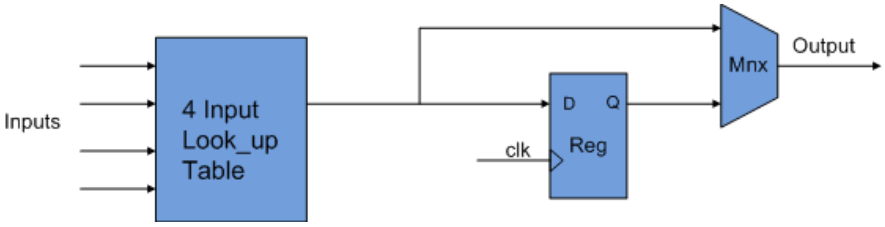


Figure 3.3.: Simplified logic block structure of a typical FPGA

ations with limited complexity. A classic FPGA logic block consists of a 4-input lookup table (LUT), a flip-flop register, and a multiplexer as shown in Figure 3.3. LUT is a constant delay device that stores combinational logic results. Significant processing gains are achievable by using LUTs since the device does not have to perform expensive computations to determine the result of a combinational function [Far09]. For a long period, 4-input LUTs were the industry standard. In recent years, manufacturers have started moving to 6-input LUTs based logic blocks to implement more complex functions with reduced logic level [Xil09]. A simplified logic block structure of the up-to-date Virex-5 FPGA from Xilinx Inc. is shown in Figure 3.4.

With the similar architecture to ASICs, FPGAs also provide the possibility to exploit parallelism when executing image processing operations. In comparison with general purpose processors (GPPs), which require several cycles to accomplish one operation, multiple computations can be performed by an FPGA in a single clock cycle using the available massive parallel hardware resources. Because the required clock cycles are significantly reduced, FPGAs can operate with much slower clocks and still provide a performance boost. The lower clock speeds result in lower power consumption, making FPGAs power-efficient. In addition,

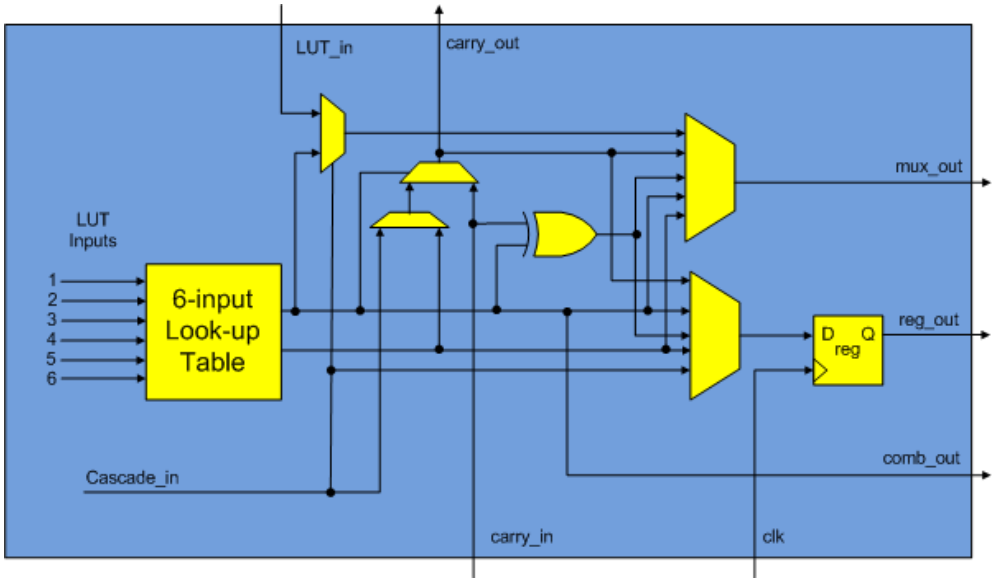


Figure 3.4.: Structural diagram of Virtex-5 logic block

modern FPGAs are equipped with dedicated on-chip hardware blocks, e.g. multipliers, RAMs, and in some cases even embedded CPUs to offer more flexibility and higher computational capability. Today, FPGAs are being used in a wide range of applications to accelerate image processing tasks.

- **DSPs** - DSPs are microprocessors with specialized architecture designed to efficiently implement computational algorithms. They were originally developed for optimizing one-dimensional signal processing in telecommunication areas, nowadays DSPs are more and more emerging into the image processing domain. One advantage of DSPs over GPPs is the switch from the Von Neumann to the Harvard architecture, in which independent program bus and data bus are available. The processor can simultaneously access two or more separate memory banks through separate communication buses, thereby loading data operands and fetching instructions concurrently [Cop08, Far09]. To further improve the computing efficiency, modern high performance DSPs feature the *Very Long Instruction Word* (VLIW) architecture, which enables multiple instructions to be fetched and executed at the same time. As an example, the *Texas Instruments* TMS320C64x DSP contains eight independent functional units, with six ALUs supporting single 32-bit, dual 16-bit, or quad 8-bit arithmetic and two multipliers supporting four 16×16-bit multiplies or eight 8×8-bit multiplies per clock cycle. When operating at 1GHz, 4000 million MACs per second (MMACS) with 16-bit multiply operations or 8000 MMACS with 8-bit multiply operations can be achieved. Figure 3.5 illustrates the structural diagram of the TMS320C64x DSP core.

The processors described above have their own field of applications. Each of them

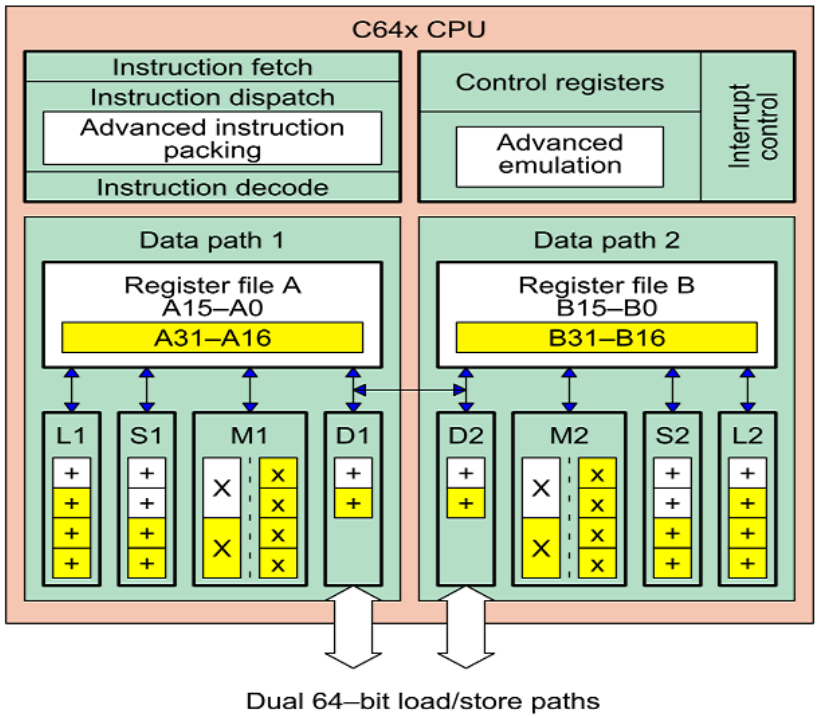


Figure 3.5.: Block diagram of TMS320C64x DSP core [Tex01]

owns its benefits in some aspects and disadvantages in others. Trade-offs often have to be made to provide a balanced implementation. There are several factors that affect the final decision of which technology to choose, such as performance, energy efficiency, cost, and ease of development.

ASICs offer the best performance in terms of computational power, however, the biggest problem with ASICs is the long design period due to their non-programmability, and the extremely high non-recurring engineering costs, which can easily exceed 1 million dollars. GPUs are much cheaper and provide excellent programming flexibility, but the high power consumption (normally >180W) makes them only suited to a PC-based system. Moreover, GPUs are inferior to FPGAs when the algorithm is data-intensive and involves a large number of memory access operations [Cop08, CCLW05], which is typically the case of low-level image processing (e.g. 2D convolution) required in most optical tracking systems.

The implemented hardware system is based on an FPGA+DSP architecture, because for our application these two devices provide the best compromise of performance, cost, energy efficiency, flexibility and development period. FPGA is an ideal choice for low-level image processing that typically involves applying the same repetitive function to each pixel in the image. Such algorithms are usually data-intensive and have a high degree of parallelism. They can be easily and effectively mapped onto an FPGA. Clearly

an optical tracking system can benefit from the parallel processing capabilities offered by FPGAs, however, it is also often necessary to implement functions requiring large control loops and complex branches rather than pixel-by-pixel iterations. Implementing such functions in an FPGA can quickly eat up the available logic resources and reduce the overall system performance [Wil04]. DSPs provide an ideal complement to FPGAs, since they enable the implementation of complicated functions in software (using C/C++). Therefore, the decision was made to make a combination of FPGAs and DSPs to reap the benefits of both. Data-intensive, repetitive processing tasks (e.g., 2D image filters) can be performed in the beginning of the processing pipeline within the FPGA, while the DSP remains free for less computationally expensive but control/math-intensive processing functions (e.g., correspondence matching and 3D reconstruction).

3.1.2. Modular Design

As mentioned in Section 1.2, scalability is an important consideration for the hardware system design. Highly scalable architecture would allow new functions to be implemented without complete redesign of the hardware platform. To achieve this objective, the hardware system was developed using a modular design approach.

A modular system consists of a number of self contained modules, which can be easily removed and replaced without significantly changing the architecture of the remaining parts of the system. The replacing module may have a different function or performance, but it should be able to interface with the existing components. If the overall hardware system can be partitioned in a modular way, new hardware modules can be added incrementally to the system, thereby achieving improved scalability [Par06].

From the hardware perspective, the optical tracking system implemented in this thesis can be divided into three separate subsystems, namely the camera system, the CameraLink grabber and the PowerEye image processing system. Additionally a CameraLink simulator was developed for simplifying the system verification. The following sections describe each subsystem in detail.

3.2. High Speed Camera

Cameras play an essential role in optical tracking. They function like human eyes, capturing information from the real world. Human eyes have excellent information acquisition capability (high image resolution) but also have spatial and temporal limitations [MDP07]. For example, human vision temporal resolution is close to 100 milliseconds [TFM96], which will lead to the failure of detecting fast movements. With the dramatic improvement of image sensor technologies, more and more high-speed image sensors are available. This provides the possibility to build high-speed cameras that are capable of capturing fast moving objects. In this thesis, a modular high-speed camera was

developed. Figure 3.6 depicts the high-level block diagram of the camera system.

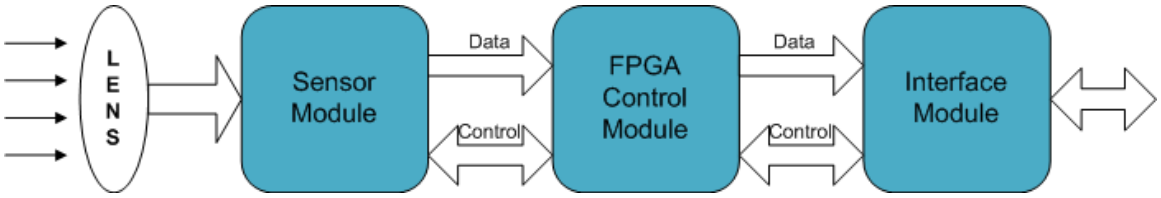


Figure 3.6.: High-level block diagram of the camera system

As can be seen, the realized high speed camera consists of three hardware modules. The first module - the sensor module [1] - carries the photo-electrical signal conversion sensor. The main component of the second module is an FPGA device that bridges the sensor module and the interface module. The interface module is responsible for transmitting the pixel data to and communicating with the outside world. Various circuit boards inside the camera are stacked together using high speed connectors. A photograph of the overall camera system is illustrated in Figure 3.7.

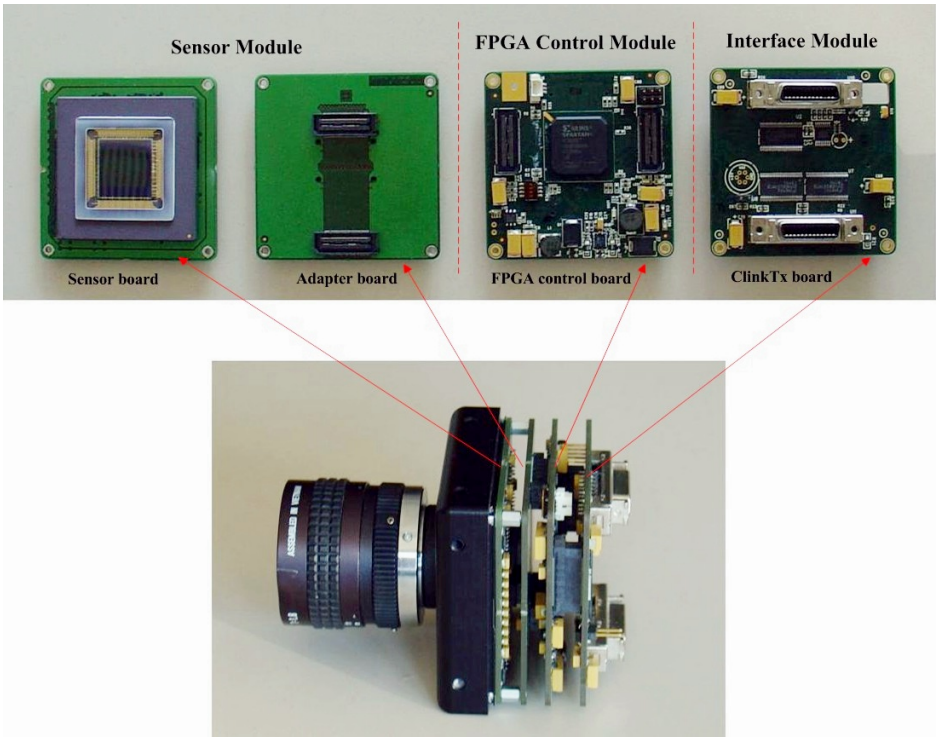


Figure 3.7.: Photograph of the camera

[1] The sensor module was designed and produced by VRMagic GmbH (www.vrmagic.com). It was adopted for the development of the camera system in this thesis.

3.2.1. Sensor Module

An image sensor is a device that converts the light intensities to electronic signals. There exist two different types of image sensors that are widely used in digital cameras: Charge Coupled Device (CCD) sensors and Complementary Metal Oxide Semiconductor (CMOS) sensors. The relative advantages of CCD and CMOS sensors have been discussed frequently in the literature. Today, in many aspects they still remain complementary [Zur01]. However, in recent years CMOS sensors have presented more and more advantages in comparison with CCD sensors, especially for high-speed applications, since they

i) are capable of achieving faster frame rate;

ii) support random pixel access;

iii) have less smear and blooming effects;

iv) require less complicated circuits for operation and consume much less power.

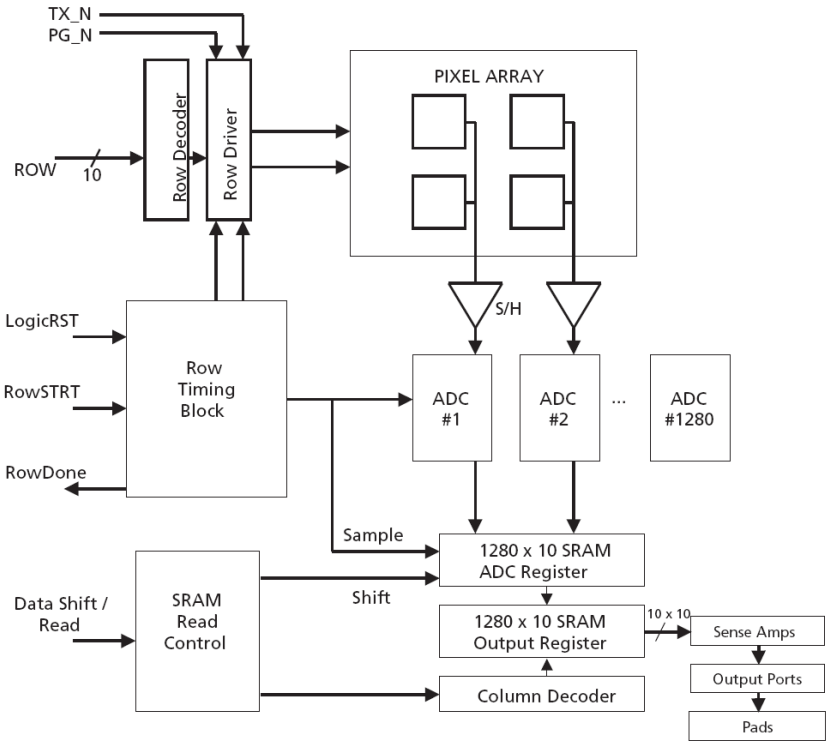


Figure 3.8.: MT9M413 sensor functional block diagram

Taking these advantages in consideration, the MT9M413 sensor from Micron Technology Inc. was chosen. The MT9M413 is a 1280H×1024V (1.3 Mega pixel) CMOS digital image sensor capable of 500 frames-per-second (fps) operation when running

at full speed (66MHz). With a partial resolution the maximum frame rate can reach 10,000 fps. Other features of this sensor include the TrueSNAP electronic shutter, which allows simultaneous exposure of the entire pixel array, the 10-bit on-chip analog-to-digital converters (ADCs), which are self-calibrating, and a fully digital interface [Mic06]. The MT9M413 sensor is available in both monochrome and color mode. Figure 3.8 illustrates the sensor functional block diagram.

As can be seen from Figure 3.7, the sensor module is made up by two circuit boards: the sensor board and the adapter board. The structural diagram is given in Figure 3.9.

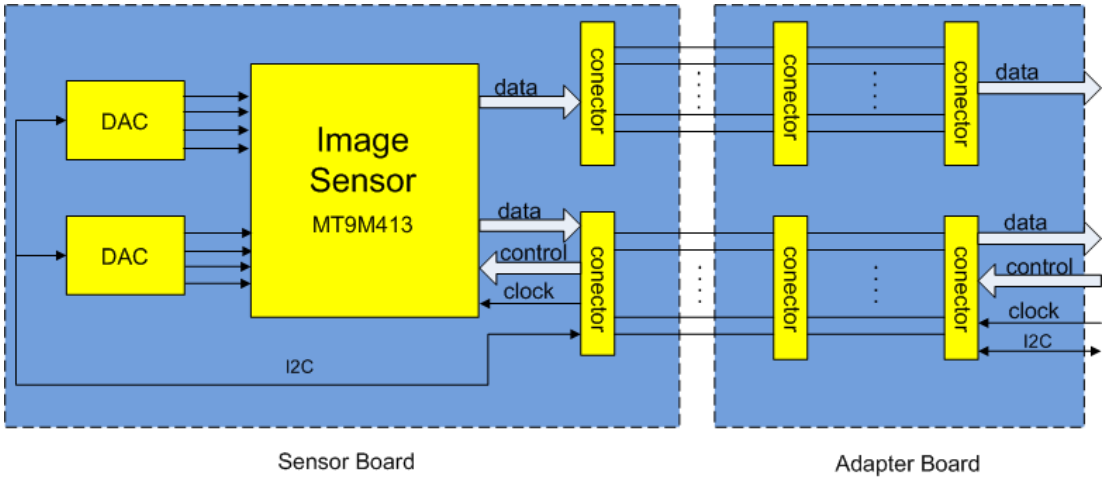


Figure 3.9.: Block diagram of the sensor module

The sensor board contains the MT9M413 image sensor mounted on the top of the PCB and all required external circuitry, including digital-to-analog converters (DACs), decoupling capacitors and signal connectors on the bottom. The MT9M413 sensor has ten 10-bit-wide digital output ports. Thus, it outputs a total of 100 bits pixel data per clock cycle. The on-chip analog-to-digital converters (ADCs) require various reference voltages for the bias setting operation and the fixed pattern noise (FPN) calibration. For this reason, the sensor board is equipped with two programmable DACs (DAC6573), each of which is capable of generating four configurable voltages. Adjusting the voltage level can be realized by programming the internal registers of the DACs via the I2C bus.

The adapter board was designed to make the connection between the sensor board and the FPGA control board. It introduces extra routing length of the high speed signals transmitted from sensor to FPGA. To minimize signal reflections along long transmission lines, which can cause serious signal integrity problems, series termination resistors are placed close to every output pin of the image sensor.

3.2.2. FPGA Control Module

The MT9M413 sensor needs a controller to guide it through the full sequence of its operation [Mic06]. An FPGA control board shown in Figure 3.7 was developed to manage the whole camera system. This module has two main tasks. First, it generates all control signals required for operating the sensor, and synchronizes the output data stream from the sensor with the interface module. Second, it reads and executes control commands from the host, such as the on-line configuration of exposure time, gain, frame rate, region of interest (ROI), etc.

In the initial phase of the hardware development, the first thought was to integrate a high-end FPGA (e.g. Virtex-4 or Virtex-5) into the camera system to perform both sensor control and high data rate image processing tasks. But this solution introduces a new problem: since the architecture is not optimized for power-efficient applications, high-end FPGAs normally consume a lot of power and result in significant amounts of heat during the operation. Considering that the FPGA will be working in a closed camera housing, the large heat dissipation could greatly increase the temperature-sensitive dark current noise of the image sensor, which in turn limits the system performance. Furthermore, equipping every camera with a high-end FPGA will potentially raise the overall system costs. Therefore, it was the decision to utilize an energy-efficient, low-cost FPGA in the camera system to perform sensor control and data communication operations. The module containing high performance FPGA and DSPs (the PowerEye system introduced in Section 3.5) that aims to process multiple pixel streams simultaneously was isolated from the camera.

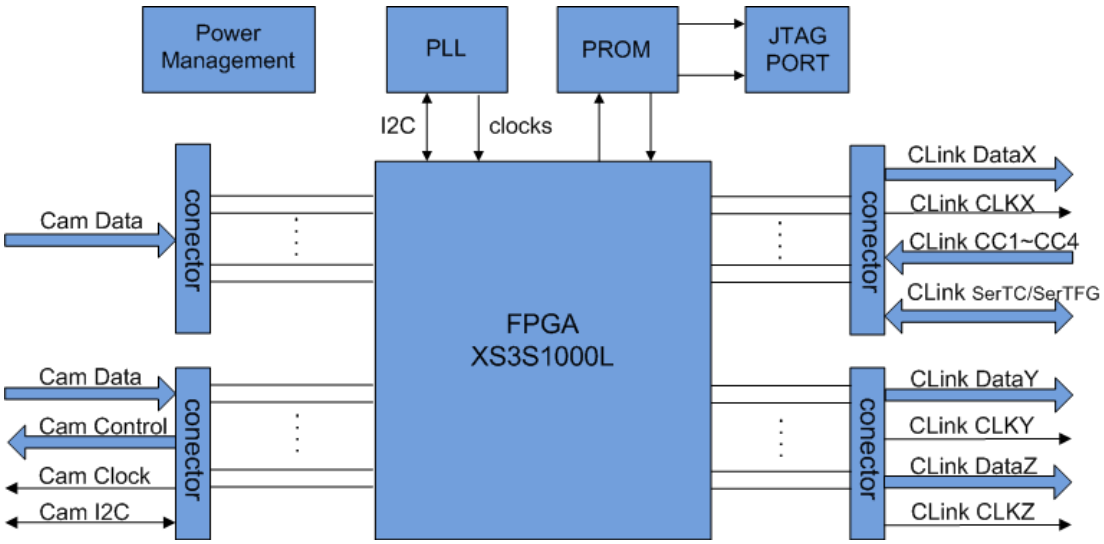


Figure 3.10.: Block diagram of the FPGA control board

Figure 3.10 shows the block diagram of the FPGA control board used in the camera system. The key component is a low power, mid-size Xilinx Spartan3 FPGA

(XC3S1000L). This device features 1 Mega system gates, 24 dedicated 18x18 multipliers as well as rich on-chip memory resources, which make it suitable for camera control and low-complexity image pre-processing tasks, such as color transformations. The on-board power management circuitry provides all necessary voltages to power the whole camera system, including the 1.2V FPGA core, the 2.5V FPGA Vccaux, and the 3.3V power supply required by FPGA I/O banks and the image sensor. A programmable PLL which is capable of generating up to four clock sources was employed to drive the FPGA internal logic, the image sensor, as well as the interface module. In addition, two pairs of signal connectors are available on the board: one is dedicated for interfacing with the sensor module; the other is used to connect the interface module. The FPGA generates all necessary signals for controlling the image sensor. The received raw image data are organized into pixel streams, which are then forwarded to the interface module.

3.2.3. Interface Module

High speed image sensor requires a high data transfer rate. For instance, if the MT9M413 image sensor is running at full frame rate (500fps) and maximum resolution (1280×1024×10bit), a communication link with sustained transfer rate of 782 MBytes (or 6250Mb/s) per second will be needed. Currently, there are four commonly used high bandwidth camera interface standards [EN08]: FireWire 800 or IEEE-1394b, Gigabit Ethernet or GigE, USB 3.0 and CameraLink. Table 3.1 depicts the general specifications of these interfaces.

Category	**IEEE-1394b**	**Gigabit-Ethernet**	**USB 3.0**	**Camera-Link**
Topology	Peer-to-Peer	Networked	Master-Slave	Master-Slave
Maximum bit rate	800 Mb/s	1000 Mb/s	5000 Mb/s	6800 Mb/s
Maximum sustained bit rate	640 Mb/s (80%)	900 Mb/s (90%)	4000 Mb/s (80%)	6800 Mb/s (100%)
Maximum cable distance	4.5 m	100 m	3 m	10 m

Table 3.1.: Specifications of digital camera interface standards

CameraLink was chosen as the communication interface for the camera system, because it provides the highest data transfer bandwidth, and is in fact the only candidate that satisfies the transmission bandwidth requirement of our high-speed camera. Moreover, CameraLink supports a long cable connection (up to 10 meter) between cameras and frame grabbers, making it convenient for applications where cameras must be located at long distances from the host. The CameraLink specification [Aut01] defines:

- a standard connector that is used on both camera and frame grabber;

- a standard cable to connect camera and frame grabber;
- image formats for transmitting data from camera to frame grabber;
- standard camera control inputs, i.e. CC1, CC2, CC3, and CC4;
- a standard method for transmitting serial communication data (SerTC/SerTFG) between camera and frame grabber;

The key technology of CameraLink is the Channel Link [Nat06], a data transmission method suggested by National Semiconductor Inc. Channel Link consists of a transmitter and receiver pair. The chipset operates with 3.3V at a maximum clock speed of 85MHz. The transmitter accepts parallelly 28 bits LVTTL data signals and a single-ended clock. The data is then serialized 7:1, and the resulting four data streams as well as the clock signal are transmitted over five LVDS pairs. When operating at full speed, 28 bits LVTTL data can be transmitted at 595 Mbps per LVDS channel. The receiver accepts the LVDS pairs, then de-serializes and converts them back into 28 bits LVTTL data signals plus one single-ended clock.

Since a single Channel Link chip is limited to 28 bits, some cameras may require several chips to achieve sufficient data transfer bandwidth. The CameraLink interface has therefore three configurations with the following naming convention:

(1) CameraLink Base - with three data ports, single Channel Link chip and single cable connector, supporting up to 255 MByte/s transfer rate.

(2) CameraLink Medium - with six data ports, two Channel Link chips and two cable connectors, supporting up to 510 MByte/s transfer rate.

(3) CameraLink Full - with ten data ports, three Channel Link chips and two cable connectors, supporting up to 850 MByte/s transfer rate.

The CLinkTx (CameraLink Transmitter) board shown in Figure 3.7 is responsible for handling the high data rate image transmission. It incorporates the connector, signals, pinout, and chipset in compliance with the CameraLink specification. Three National Semiconductor DS90CR287 chips serve as the high-speed image data transmitter. A DS90LV048A differential line receiver is utilized to receive the camera control signals (CC1∼CC4). The serial data communication (SerTC/SerTFG) defined in the CameraLink specification is implemented by a DS90LV019 differential line transmitter/receiver. CLinkTx was designed to support all three CameraLink configurations. The structural block diagram is shown in Figure 3.11.

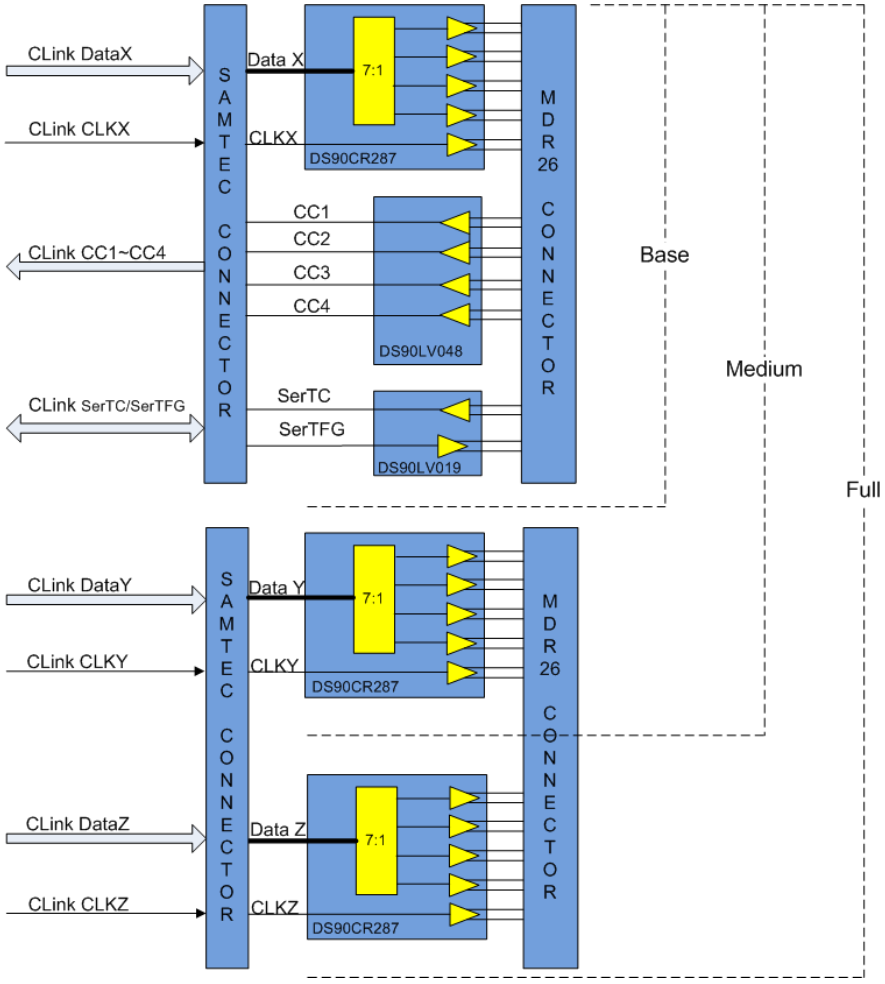


Figure 3.11.: Block diagram of the CLinkTx interface module

3.3. CameraLink Grabber

The CameraLink grabber captures digital image data from one or multiple CameraLink cameras and converts the received LVDS signals back to single-ended signals. Two different types of CameraLink grabber board were developed, which are named CLinkRx-TripleBase and CLinkRx-FULL respectively.

3.3.1. CLinkRx-TripleBase

The CLinkRx-TripleBase board shown in Figure 3.12 was designed to simultaneously capture image streams from three separate CameraLink cameras that are configured in Base mode. Three independent Channel Link chip sets are used on the board, each of which consists of one DS90CR288A as image data receiver, one DS90LV047A as camera control signal transmitter and one DS90LV019 for the serial communication. The incoming LVDS signals carrying the image data are de-serialized and converted to single-ended 3.3V LVTTL signals, which are later routed to a 180-pin high speed con-

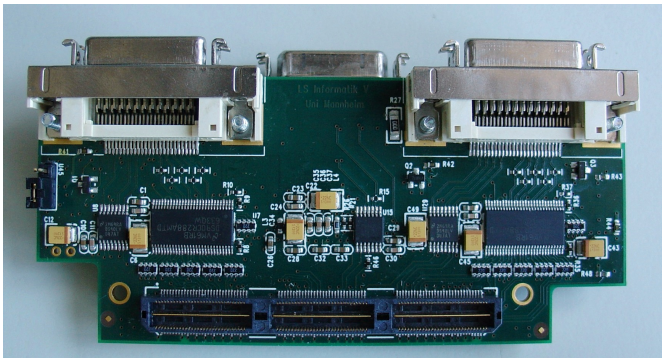


Figure 3.12.: Photograph of the CLinkRx-TripleBase board

nector (Samtec QTH-090-01-L-D-A). Termination for both differential and single-ended signals is provided on board. The schematic block diagram of CLinkRx-TripleBase is shown in Figure 3.13.

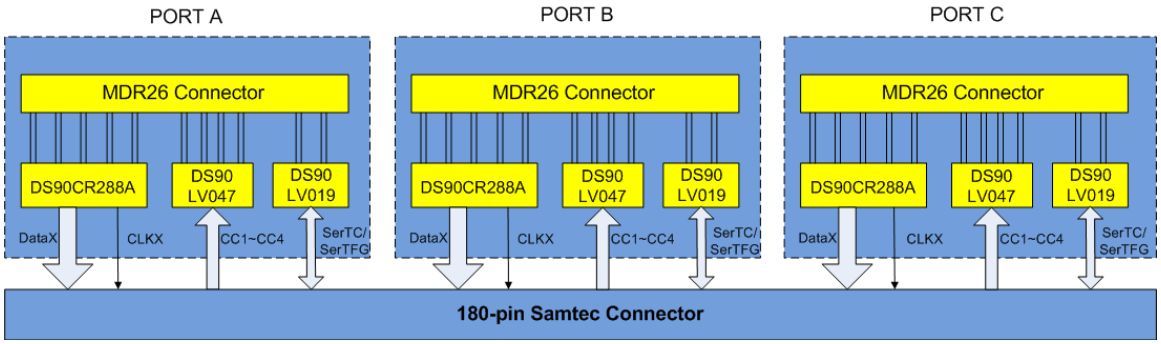


Figure 3.13.: Block diagram of the CLinkRx-TripleBase board

3.3.2. CLinkRx-FULL

When the MT9M413 sensor is running at full speed (1280×1024×10b@500fps), a CameraLink Full interface will be needed since the image data rate (782 MByte/s) exceeds the maximum bandwidth provided by CameraLink Base and Medium. The CLinkRx-FULL board is able to connect a single camera and supports all three levels of CameraLink interface - Base, Medium and Full. As shown in Figure 3.14, Port A operates in the same way as the Base mode interface, transferring the image data via channel X. Port B consists of two Channel Link receiver (DS90CR288A) chips that transmit data through channel Y and channel Z simultaneously. The three on-board Channel Link receivers are capable of transporting a total of 80 bits image data per clock cycle - 24 bits for channel X, 28 bits for Channel Y and 28 bits for channel Z. Thus, when clocked at 85MHz, a substantial transmission rate of 850 MByte/s can be achieved.

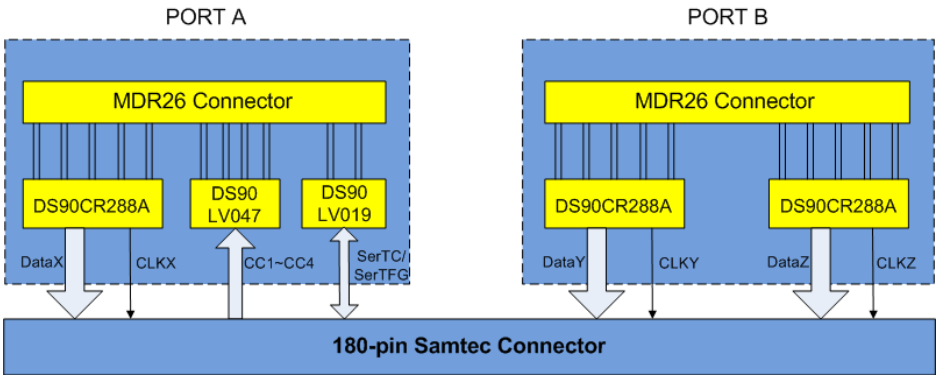


Figure 3.14.: Block diagram of the CLinkRx-Full board

3.4. CameraLink Simulator

As described previously, our hardware system is divided into multiple independent modules in order to achieve high modularity. However, this concept also increases the complexity of the hardware debugging process, especially when the camera, the CameraLink grabber and the image processing system (PowerEye) are incorporated together.

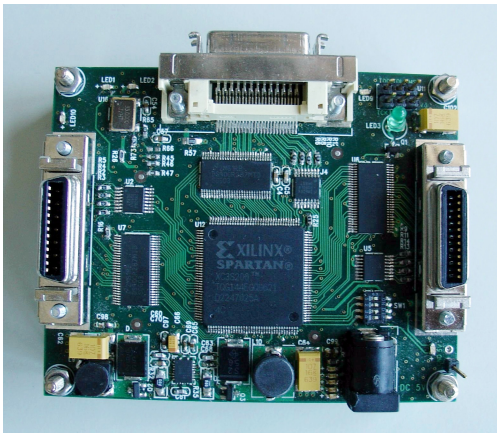


Figure 3.15.: Photograph of the CLinkSim board

A CameraLink simulator - the CLinkSim board shown in Figure 3.15 was developed for the purpose of simplifying the system verification. The idea is to replace the complete camera system, which contains multiple circuit boards, by a single-board simulator, so that the CameraLink grabber and the image processing system can be tested without connecting a real camera. CLinkSim is basically a video pattern generator based on a Spartan-3 FPGA (XC3S200) device. The FPGA can be programmed to simulate the timing characteristics of one or multiple real cameras and to output video streams fully in compliance with the CameraLink standard. CLinkSim supports both CameraLink Base and Medium configurations. It can also be used to simulate the behavior of up to three cameras configured in the Base mode simultaneously. The

block diagram of CLinkSim is shown in Figure 3.16.

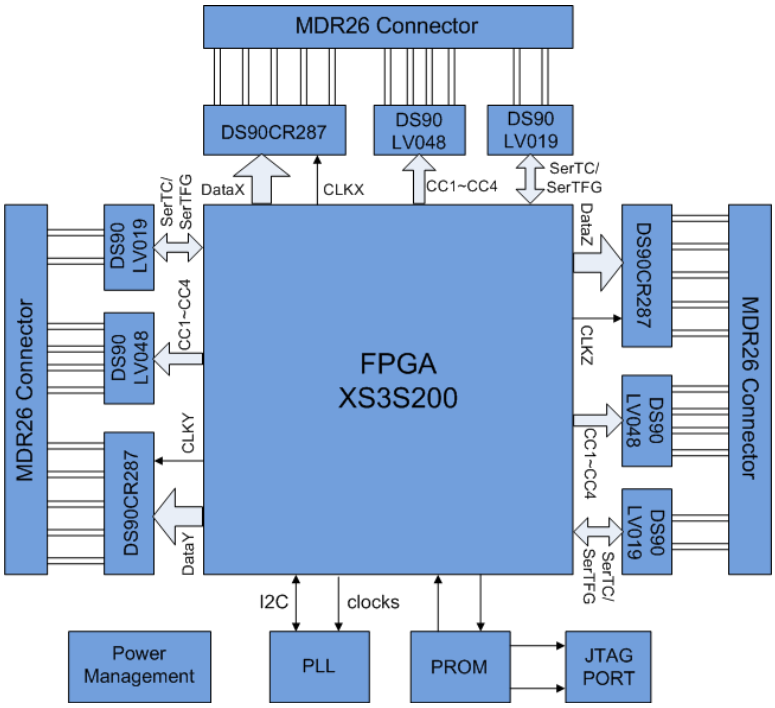


Figure 3.16.: Block diagram of the CLinkSim board

3.5. PowerEye

PowerEye was designed as a general-purpose hardware platform that can be used in a wide range of image processing applications. It can be plugged into a PC, serving as an accelerator for the PC CPU, or can operate in a stand-alone manner, functioning as an embedded system. There are some basic elements which should be considered when designing a generic image processing system, such as processors, frame buffer memory and host interfaces.

Image processors are responsible for performing all necessary computations for implementing the required 2D/3D image processing algorithms. The processors on PowerEye consist of a high-density FPGA and two high-performance DSPs. The advantages of such a processing architecture have already been discussed in Section 3.1.1.

Frame buffer memory is an important issue for an image processing system. Many image processing algorithms need to access the image data in some order other than the one it is presented by the video source. Thus, it is necessary to collect the image data for each frame in a frame buffer memory that is randomly accessible by the processor[And96]. On PowerEye, various memory resources can be utilized as frame buffers, which include 9 Mega bytes ZBTSRAM directly connected to the FPGA, and 128 Mega bytes SDRAM mapped to the DSP memory space.

An image processing system should provide one or multiple interfaces to connect with a host (e.g., a PC) for the purpose of system setup, control and data visualization [And96]. PowerEye has access to three industrial standard interfaces, i.e. PCI-Express, Gigabit Ethernet and USB2.0. In addition, a 110-bit general purpose expansion connector and a 12-pair high-speed LVDS link connector are provided on-board for use of interfacing different types of peripherals.

Figure 3.17 and Figure 3.18 illustrate the photograph and the block diagram of PowerEye respectively. The details of each functional block are introduced in the following subsections.

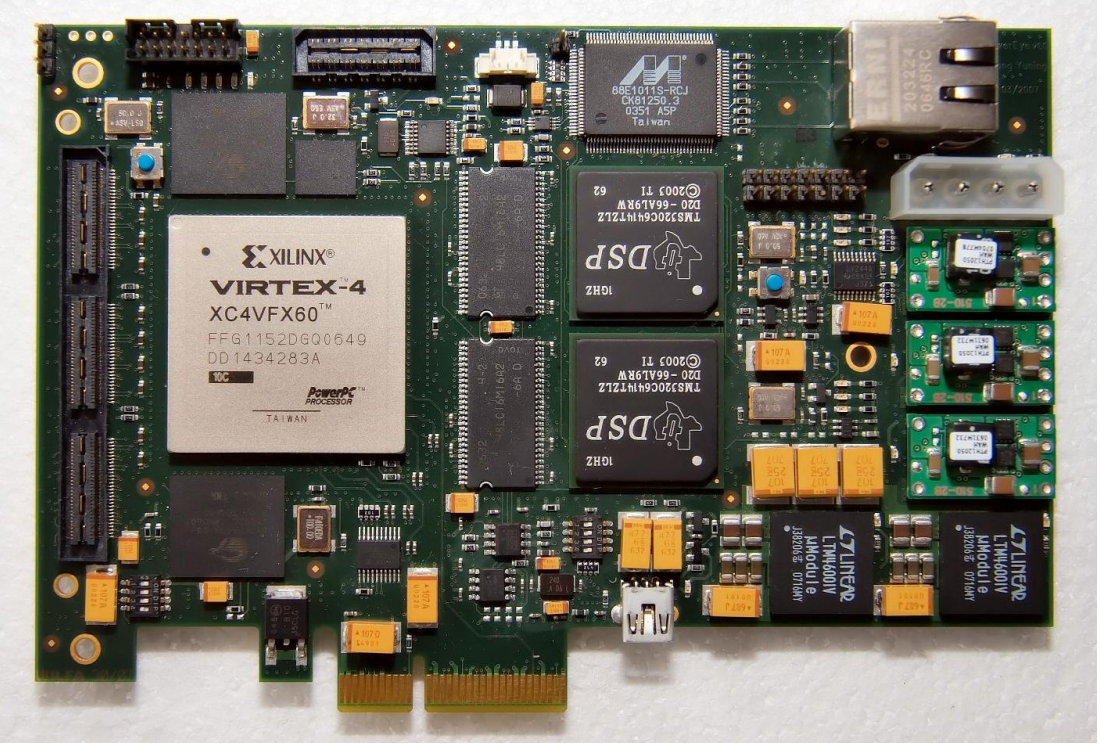


Figure 3.17.: Photograph of the PowerEye board

3.5.1. FPGA

The FPGA selected for PowerEye is one of the Virtex-4 devices [Xil07] from Xilinx Inc., which are fabricated with 90nm copper CMOS process. Besides a significant increase in density of logic gates compared with former devices, the Virtex-4 FPGAs offer several architectural advantages, e.g., the Digital Clock Managers (DCMs) which provide various clock management features, including clock deskew, frequency synthesis, and phase shifting, the I/O blocks that support a wide range of voltage standards from 1.5V to 3.3V, the on-chip true dual-port synchronous block RAMs which are capable of running at 500MHz, and the rich hard-IP core blocks including PowerPC processors, tri-mode Ethernet MACs, 6.5 Gb/s serial transceivers and dedicated XtremeDSP slices.

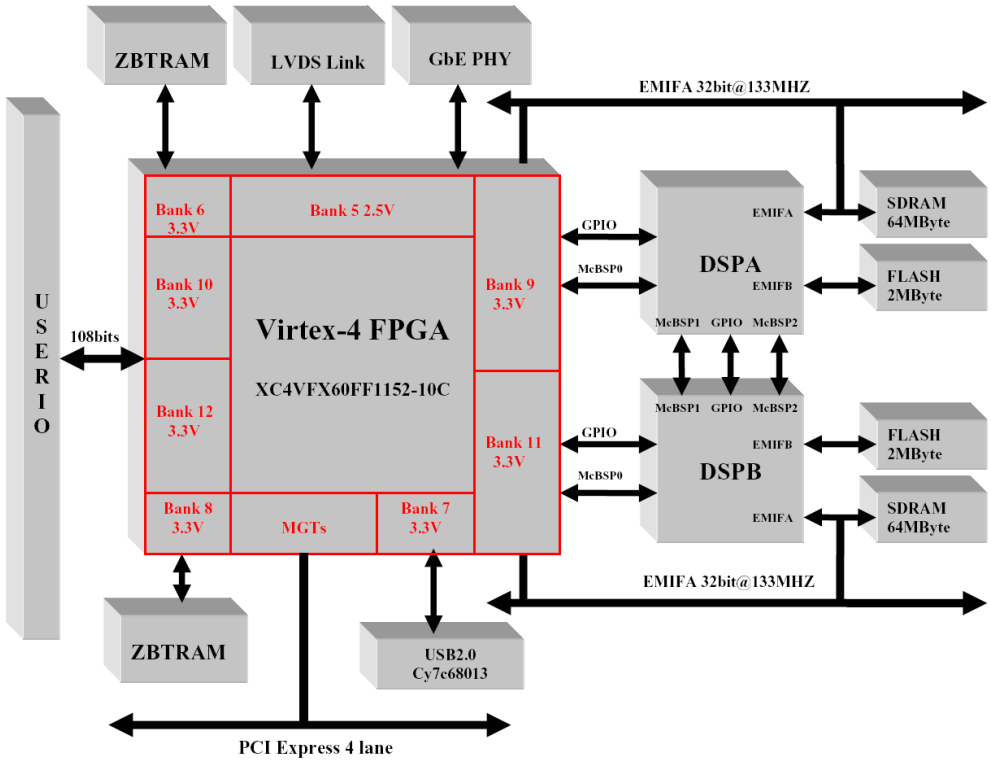


Figure 3.18.: PowerEye block diagram

Considering that the FPGA must be able to interface a number of peripherals, the XC4VFX60-FF1152 with a large amount of I/O pins (576 programmable I/O pins in total) was chosen. This device has a footprint that is fully compatible with the XC4VFX100 FPGA. If required, the current FPGA on PowerEye can be replaced by a larger device containing more logic and memory resources without changing anything regarding the hardware design.

3.5.2. ZBTSRAM

Although the XC4VFX60 FPGA contains a total of 522k bytes on-chip block RAM, an entire image produced by the MT9M413 sensor (1280×1024) consumes 1.3M bytes. Thus an external frame buffer is required to store the image data.

The choice was made between Synchronous Dynamic RAM (SDRAM) and Static RAM (SRAM). SDRAMs, such as the DDR2- or DDR3-SDRAMs, have the advantages of high storage capacity and low cost. Nevertheless, they suffer from a significant amount of access latency. If the image data must be randomly read from or written to a frame buffer, the achievable throughput with SDRAMs can be very low. SRAMs, on the other side, support very low access latency (typically ≤ 2 clock cycles) and provide medium-size storage space. They are capable of accessing random data at non-sequential addresses with constant delay [Alt08]. This characteristic makes SRAMs

more effective for use not only as image frame buffers, but also as large Look-Up-Tables (LUTs) holding the data that is too large to fit in the FPGA on-chip memory.

On PowerEye, the Virtex-4 FPGA has direct access to a total of 9M bytes fast ZBTSRAM (CY7C1460-250BZC). ZBT stands for "Zero Bus Turnaround". As the name implies, this kind of SRAM devices are designed to sustain 100% bus bandwidth by eliminating turnaround cycles when there is a transmission from Read to Write, or vice versa. The ZBTSRAMs on PowerEye are organized in two independent 1M×36bit banks, as the control path, the address and data buses as well as the clock inputs are unique to each bank with no sharing of signals. This architecture allows to construct a ping-pong frame buffer, which enables simultaneous image capture and processing to improve the system performance. Since both ZBTSRAM banks can be clocked at 200MHz, a maximum data transfer rate of 3.6GByte/s is achievable.

3.5.3. DSP

In addition to the Virtex-4 FPGA, PowerEye contains two TMS320C6414T DSPs clocked at 1 GHZ, providing a peak performance of 16,000 MIPS (Million Instruction per Second).

The TMS320C6414T is one of the high-performance fixed-point DSPs from Texas Instruments. It has a two-level cache architecture. The first level (L1 cache) is a set of 128 kbit of program cache and 128 kbit of data cache. The second level (L2 cache) consists of an 8 Mbit memory space that is shared between program and data space.

The TMS320C6414T DSP is equipped with a powerful and diverse set of peripherals, in particular three multi-channel full duplex buffered serial ports (McBSPs), a user-configurable 16bit or 32bit host port interface (HPI), 16 bits general purpose input/output (GPIO), and an external memory interface (EMIF) supporting a glueless interface to a variety of external memory devices, including SDRAM, SRAM, and first-in first-out (FIFO) buffers. The TMS320C6414T DSP provides two EMIF options: EMIF-A (configurable to 32bit or 64bit data width) and EMIF-B (configurable to 8bit or 16bit data width).

Another powerful feature of the TMS320C6414T DSP is its Enhanced Direct Memory Access (EDMA) engine, which aims at efficiently transmitting large amounts of data between the core CPU, device peripherals, and the L2 cache without any CPU intervention. The EDMA engine has 64 channels for independent transfers, most of which are specifically triggered by a distinct peripheral event. All of the channels, however, can be manually triggered by the CPU as long as the particular synchronization event for the respective channel is not activated. More detailed information about the TMS320C6414T DSP can be found in [Tex09].

3.5.4. SDRAM

A total of 128M bytes of SDRAM memory is available on PowerEye, since each DSP is equipped with 64M bytes SDRAMs mapped to the CE0 memory space. The SDRAMs are clocked at 133MHz and connected directly to the DSP EMIF-A port. They can be used as a complement to the ZBTSRAMs for applications requiring a large amount of data to be stored during the runtime.

3.5.5. FLASH

Flash belongs to the non-volatile memory used frequently in embedded systems. On PowerEye, a 2M-Byte FLASH is connected to each DSP in the EMIF-B CE1 memory space. It holds the boot code for the DSP and optional user-defined parameters. The FLASH is a 16-bit wide device which can also be configured under 8-bit mode. The boot code must be stored in 8-bit format because this is the mode the DSP requires. Since the TMS320C6414T DSP only provides 20 address lines on its EMIF-B bus, one GPIO (GPIO9) pin is connected to the highest address bit of the FLASH for page mode access to the full 2M-Byte memory space. As a result, the FLASH memory is seen from each DSP as two separated 1M-Byte pages.

3.5.6. Inter-processor Communication

In all multi-processor systems, one of the most crucial aspects of determining the overall system performance is the inter-processor communication efficiency in terms of both latency and data transfer bandwidth. Great attention was paid to this aspect in the early phase of the hardware architecture design. Figure 3.19 shows the multi-path inter-processor communication architecture that was implemented on PowerEye.

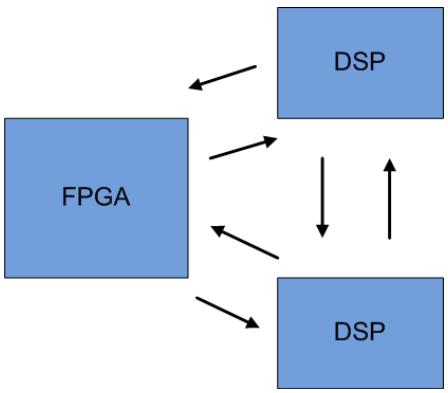


Figure 3.19.: Multi-path inter-processor communication

FPGA to DSP Communication

The interface selection between FPGA and DSP is driven by application characteristics as well as the available interfaces of the processor. On PowerEye, two different DSP communication interfaces are utilized to connect the FPGA: the 32-bit EMIF-A and the McBSP.

The 32-bit EMIF-A was selected as a high-speed communication channel between DSP and FPGA due to its high data transfer rate and the possibility of using the DSP EDMA engine. The block diagram of this communication method is given in Figure 3.20.

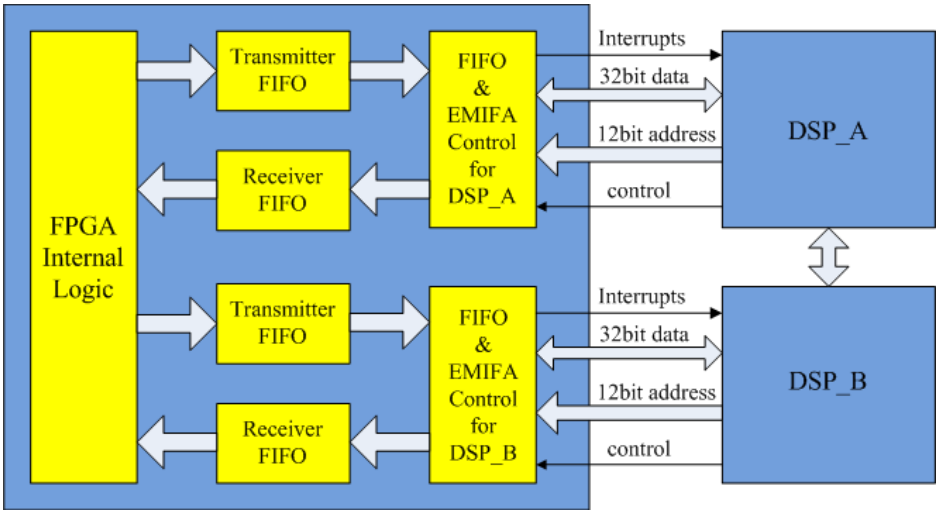


Figure 3.20.: FPGA to DSP communication via EMIF-A

The EMIF&FIFO Control block in the FPGA decodes the EMIF control signals from the DSP and forwards the data to the receiver FIFO. When the amount of data in the receiver FIFO reaches a predefined threshold, the FPGA starts to read the data and process them. The FPGA sends the results to the transmitter FIFO if it has a certain amount of free space. When the amount of data in the transmitter FIFO reaches a threshold, the FPGA issues an interrupt and sends a DMA transfer request to the DSP. The DSP EDMA engine will then move the data from the transmitter FIFO to its local memory. Since EMIF-A is a 32bit bus that is capable of operating at 133 MHz, a peak data transfer rate of 532 MByte/s can be achieved.

The second option for the FPGA to DSP communication is enabled by McBSP, which is a full duplex serial interface capable of running at up to 125 MHz (two 125Mbps streams). As shown in Figure 3.18, the FPGA has access to the McBSP0 port of both DSPs. Compared with the EMIF interface, the achievable data transfer rate via McBSP is significantly lower. However, this communication method costs much less FPGA logic resources due to the simplicity of the transmission protocol. Figure 3.21 illustrates the block diagram of this communication scheme.

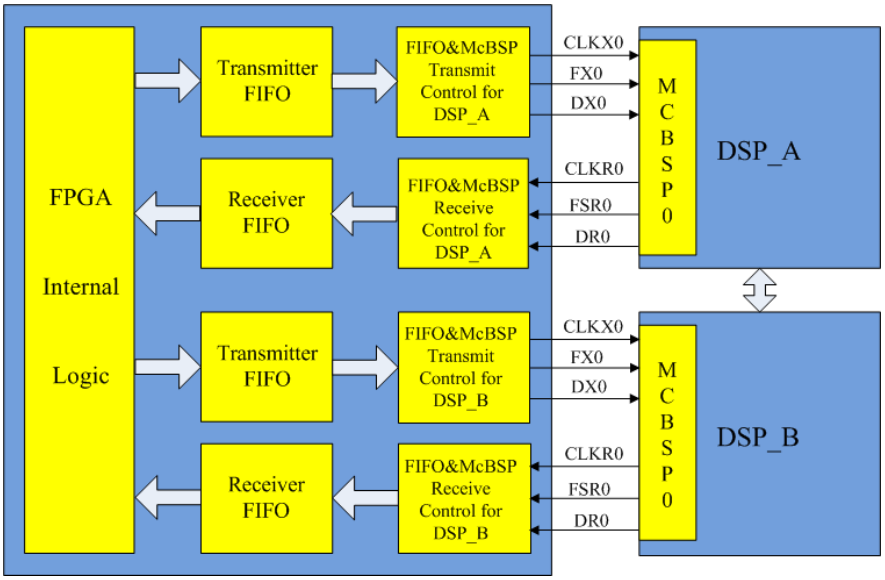


Figure 3.21.: FPGA to DSP communication via McBSP

DSP to DSP Communication

The DSPs on PowerEye were also designed to be capable of exchanging data with each other. As illustrated in Figure 3.18, the two DSPs are directly linked by their McBSP1 and McBSP2 ports. During the communication, the McBSP transmitter behaves as a master that generates both clock and frame synchronization signals for data transfer. The other McBSP portion then acts as a slave awaiting these signals from the master. For instance, the McBSP1 transmit portion of DSPA is the master of both clock and frame to the McBSP1 receiver of DSPB, and the McBSP1 transmitter of DSPB is configured to be the clock and frame master to the McBSP1 receiver of DSPA. The same arrangement applies for the McBSP2 port.

The main drawback of the communication via McBSP is its low data transfer bandwidth due to the sequential nature of McBSP. An alternative is to use dual port RAM for more efficient inter-DSP communication. A dual port RAM is a static memory with dual access ports. Each port has separate address, data and control signals, which are ideal for communication between two asynchronous devices [IDT04]. On PowerEye, this communication scheme can be easily implemented by using the FPGA internal RAM resources. The XC4VFX60 FPGA has a total of 232 pieces of block memory, each of which is organized as a 512x36-bit dual port RAM. They can be seamlessly connected to the 32-bit EMIF-A port of the DSP. During the communication the dual port RAM can be considered as a shared buffer that is accessible to both DSPs. The sender DSP sends data to the shared buffer and notifies the receiver. The receiver reads the data out and messages the sender that the buffer is free. It is important to maintain coherency when multiple devices access data from the shared memory. For applications that require high speed data exchange between DSPs, the EDMA-driven

large block transfer can be used to maximize the communication bandwidth. Tests in lab show that a substantial inter-DSP transfer rate of 400MByte/s can be achieved at the frame size of 4k bytes per DMA.

3.5.7. Interface

PowerEye provides PCI-Express, Gigabit Ethernet and USB2.0 as standard interfaces for communication with external devices, like a PC. In addition, two expansion connectors are mounted on board that can be used to interface a variety of peripherals.

PCI-Express

PCI-Express is an enhancement to the well-known PCI architecture where the parallel bus has been replaced with a scalable, fully serial interface. The differences in the electrical interface are transparent to the software, so existing PCI software implementations are compatible.

The XC4VFX60 FPGA enables the possibility to implement a PCI-Express interface by use of the embedded Multi-Gigabit Transceiver (MGT) blocks. MGT is a full duplex serial transceiver for point-to-point transmission applications and can operate at any serial bit rate in the range of 622 Mb/s to 6.5 Gb/s per channel. XC4VFX60 provides a total of 16 MGTs. Four of them are utilized on PowerEye to build a 4-lane PCI-Express interface. Each lane has a unidirectional transmit and receive differential pair supporting a transfer data rate of 2.5 Gb/s.

Using PowerEye in a PCI-Express application requires the logic-level implementation of the PCI-Express protocol inside the FPGA. Xilinx provides a PCI-Express Endpoint IP core with a physical interface to the MGT ports. However, the implementation of PCI-Express is beyond the scope of this thesis due to limited time of development.

Gigabit Ethernet

Ethernet is the most widely used technology for local area networks (LAN). It is very flexible, easy to implement and highly scalable. Since more than 25 years Ethernet has been covering 97% of all installed network connections.

An Ethernet device contains mainly three basic components: a physical layer transceiver (PHY), a media access controller (MAC) and a protocol stack defined in the OSI Reference Model.

The XC4VFX60 FPGA integrates four hard Tri-Mode Ethernet MAC cores that support 10/100/1000 Mb/s data rates and are designed to comply with the IEEE Std 802.3-2002 specifications. The Ethernet MAC features the IEEE standard GMII-/RGMII interface for accessing the PHY. An Ethernet PHY (88E1011S from Marvell Inc.) connected to a RJ-45 jack is equipped on-board to implement the physical layer transmission. The XC4VFX60 FPGA provides the possibility to build a protocol stack

both in software (using the embedded PowerPC processors) and in hardware (using logic resources). In this thesis, a hardware based UDP stack was implemented to achieve maximum transfer bandwidth and lowest latency.

USB 2.0

An on-board micro-controller (Cy7c68013 from Cypress Inc.) enables data communication between PowerEye and PC via a standard USB 2.0 interface. The micro-controller is a single-chip device that integrats a 8051 core, a Serial Interface Engine (SIE) and a USB 2.0 transceiver supporting both full-speed (12Mbps) and high-speed (480 Mbps) operations. A dedicated 64Kb serial EEPROM is provided to maintain the firmware. The EEPROM can also be accessible by the FPGA via the I2C bus.

I/O Expansions

Besides the standard interfaces, there are two expansion connectors available on PowerEye, namely the 110-bit general-purpose user I/O connector and the 12-pair LVDS Link connector, which were designed for easy interfacing with additional off-board components.

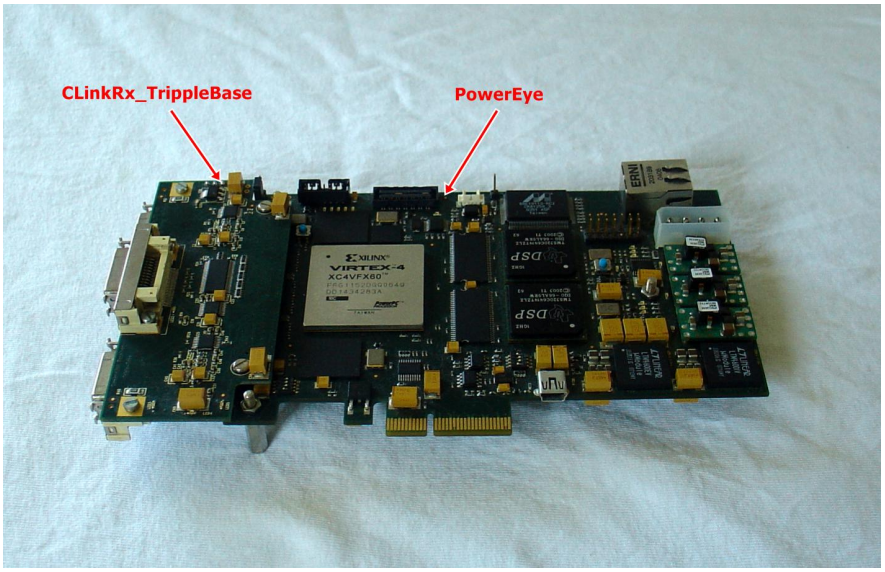


Figure 3.22.: Connection between PowerEye and CLink-TripleBase

The general-purpose user I/O is made up by 110 single-ended signals routed to a 180-pin high density connector (Samtec QSH-090-01-L-D-A) near the left edge of the board. These signals are directly connected to the FPGA programmable I/O pins that are distributed in two different banks. The remaining pins of the connector are either connected to ground, for ensuring good signal return path, or connected to the 12V/5V/3.3V power planes of PowerEye, serving as power supplies for an off-board component. The aim of this general-purpose interface is to allow different hardware

modules to be interconnected together without them actually being designed to specifically fit each other, thereby adding more reusability to the whole system. In our optical tracking system, the general-purpose user I/O is used to interface with the CameraLink grabber board, as shown in Figure 3.22.

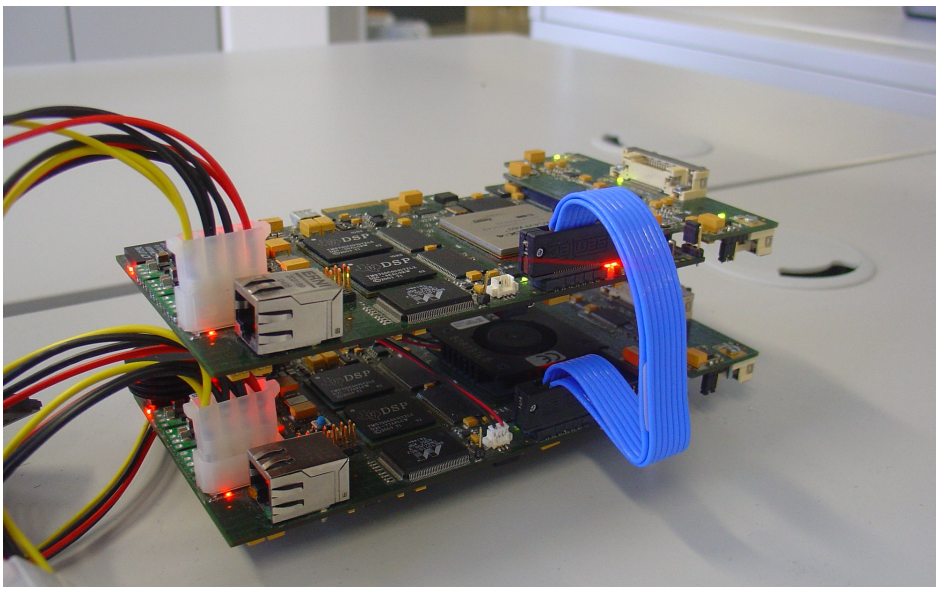


Figure 3.23.: Connection between two PowerEye boards via the LVDS Link

The 12-pair LVDS Link was designed for high speed board-to-board communication as illustrated in Figure 3.23. All 12-pair LVDS signals are connected to the same FPGA bank powered with 2.5V. These differential signals are distributed across a QSE-014-DP connector on the board edge, and are routed with 100ohm differential trace impedance and matched length. Since the maximum speed of the Virtex-4 LVDS I/O reaches 1Gbps, the LVDS Link can serve as a high-speed data communication channel, through which the image data can be transmitted in real-time between two PowerEye boards.

3.5.8. Clock Distribution

Figure 3.24 depicts the block diagram for the clock distribution scheme on PowerEye.

- **sys_clk** - A 50 MHz and a 100 MHz oscillator provide the system clock inputs to the FPGA. They are typically used to generate clocks with various frequencies and phases within the FPGA fabric.

- **user I/O clocks** - Four user I/O pins are connected directly to the FPGA global clock inputs. This allows an off-board component to supply reference clocks for PowerEye.

- **programmable clocks** - To adjust the clock frequencies for one or multiple on-board components in runtime, a phase-locked loop (PLL) capable of operating

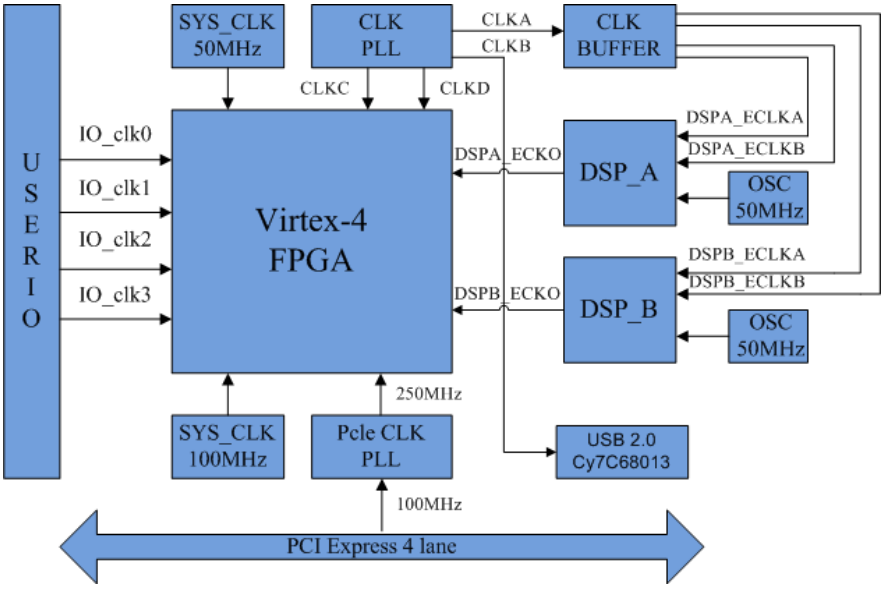


Figure 3.24.: Clock distribution on PowerEye

over a wide range of frequency is equipped on PowerEye. When combined with a reference oscillator, the PLL is able to output up to four independent clocks (CLKA, CLKB, CLKC and CLKD) with various frequencies. On PowerEye, CLKA is used to drive the DSP EMIF ports. CLKB sources the USB 2.0 microcontroller. CLKC and CLKD are connected to the FPGA global clock input pins.

- **DSP_clk** - Each DSP is connected to a clock oscillator with fixed frequency (50 MHz). The PLL inside the DSP multiplies the source clock frequency with a configurable value to generate the internal operating clock.

- **EMIF_clk** - The EMIF signals of the TMS320C6414T DSP can be synchronized either by an internal clock or externally by a reference clock. For inter-processor communication, the second option is preferred, since the internal clocks of the on-board DSPs are not synchronized to each other. On PowerEye, a Zero-Delay clock buffer that accepts one reference clock (CLKA output from the programmable PLL) and generates four zero-delayed low-jitter clocks is used to source the EMIF ports of both DSPs. Since all these clocks are phase locked with the reference clock, a strict synchronization of the EMIF signals belonging to different DSPs is achievable. This greatly decreases the design complexity of the EMIF-based inter-processor communication.

- **PCIe_clk** - In order to function as a PCI-Express device, PowerEye must use the 100 MHz reference clock provided over the PCI-Express card edge to be frequency-locked with the host system. The Virtex-4 FPGA requires a 250 MHz

clock for the MGT clock source to meet the requirements specified in the PCI-Express standard. A PCI-Express jitter attenuator is utilized to convert the 100 MHz PCI Express clock to the required 250 MHz while still meeting the jitter tolerance of the Virtex-4 MGTs.

3.5.9. Power Management

Power management is one of the key challenges that have to be faced in complex circuit design. PowerEye can be powered by 12V DC either from the external power supply connector or from the PCI Express slot. The on-board power management circuitry generates various voltages required by different hardware components. Figure 3.25 gives the high level block diagram of the power management circuitry.

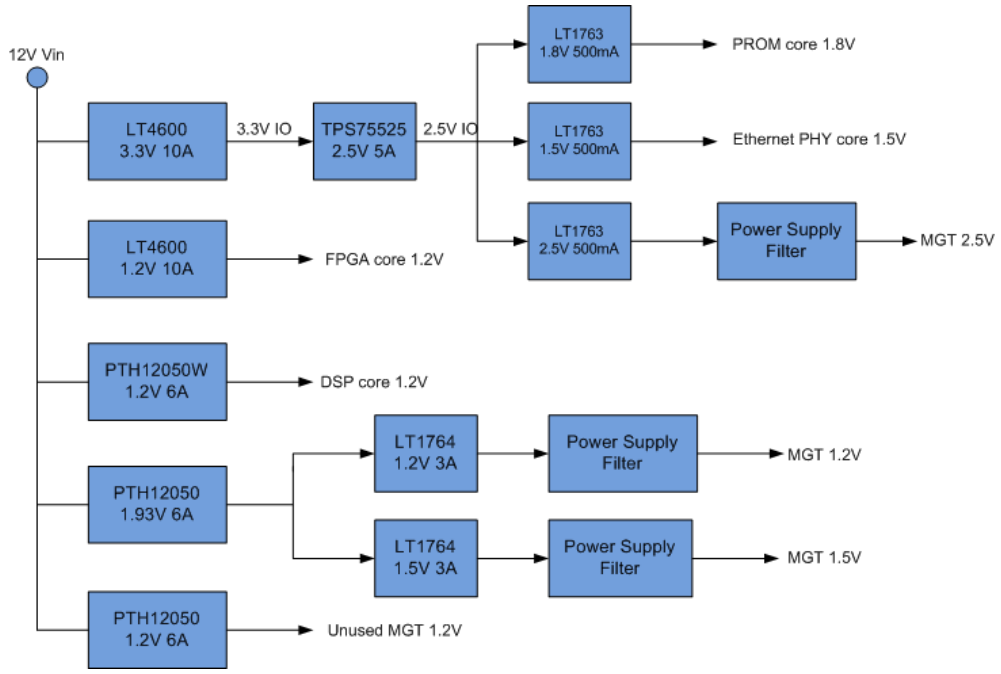


Figure 3.25.: Block diagram of power management

The +1.2V FPGA core and +3.3V power rails are realized by two LT4600 power modules independently. LT4600 is a switching DC/DC regulator capable of furnishing up to 10A current. The DSP core power supply is regulated by the PHT12050 module, which sources up to 6A current. A 5A linear regulator is used to convert +3.3V to +2.5V for the Ethernet PHY I/O power supply and the 2.5V FPGA VCCO, as well as the FPGA Vccaux. The +1.5V and +1.8V voltages for the Ethernet PHY and FPGA PROM are derived from the on-board 3.3V power plane using two linear regulators.

The MGTs of the Virtex-4 FPGA are powered by linear voltage regulators, since the MGTs are very sensitive to noise of the power. Moreover, dedicated passive high-frequency filtering circuitry is used to ensure clean power supply for the MGT blocks.

3.5.10. Printed Circuit Board (PCB) Design

PowerEye operates at high frequency with a complicated circuit structure. High speed circuit design rules were strictly followed during the PCB development.

The major concern for the PCB design of PowerEye was signal integrity. Several important issues in this regard have been taken into account, which mainly include crosstalk reduction, impedance control, signal delay matching, and supply voltage by-passing.

The root of the crosstalk problem is the large mounts of signal routes on the board. High routing density can cause serious cross talk problems that drastically degrades the overall performance. Thus, the primary method for improving signal integrity is to reduce the routing density[CKRB03]. This can be achieved by increasing the number of PCB layers when the the dimension of the PCB has already been defined. PowerEye was fabricated with a 15cm×10cm PCB consisting of 14 layers, which include eight signal layers, three power planes and three ground planes. The PCB stack-up architecture is shown in Figure 3.26.

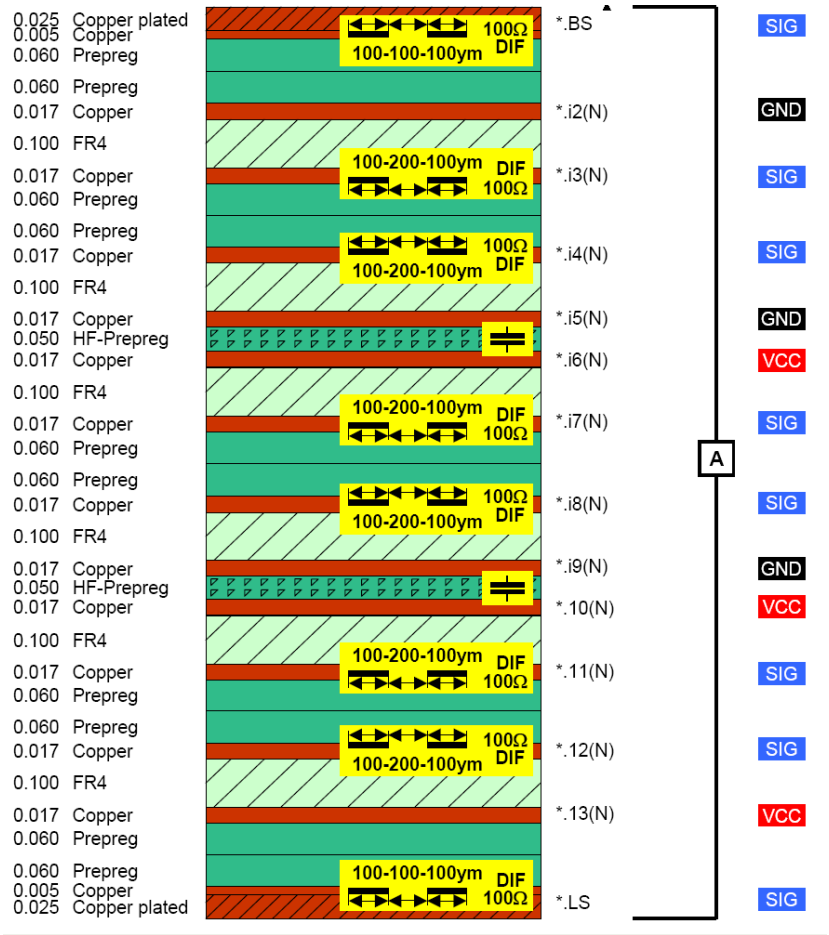


Figure 3.26.: PowerEye PCB stack-up

As can be seen, every signal plane in the PCB is adjacent to a reference plane, which

ensures that the return currents always travel as near as possible to their corresponding trace. Signals in adjacent layers were routed perpendicularly, so that the horizontal and vertical layers alternate. This significantly limited the crosstalk between signal traces of adjacent layers.

Impedance mismatch of transmission line on PCB can cause signal reflection problems, especially in high frequency environments. The impedance of 50Ω for single-ended signals and the impedance of 100Ω for differential signals were chosen for approximate matching with signal drivers and signals on off-board components. The trace widths of the single-ended signals were adjusted for each layer in order to maintain constant transmission line impedance across different layers. Series resistor terminations have been used on long traces to attenuate the signal reflections. All differential signal trace pairs were routed together with fixed spacing. PCB layer thicknesses and trace widths were chosen to ensure uniform impedance value throughout the board as shown in Figure 3.26.

High speed signals, such as the data and address buses of the ZBTSRAMs, DSP EMIF signals and the clocks were routed with matched lengths to ensure consistent signal delay, which is important for all high speed synchronous designs.

Ground bounce is another problem due to high switching activities on the high pin-count FPGA and DSPs. In extreme cases, the voltage drop can be significant enough to make the on-board ICs momentarily malfunction and produce errors [CKRB03]. Therefore, as many as possible bypassing capacitors should be provided to ensure proper system operation. There are over 500 bypassing capacitors placed on PowerEye with capacitance values ranging from $22pF$ to $680\mu F$.

3.6. System Setup

As described in the previous sections, the implemented hardware system features a very flexible architecture that can be easily scaled up. Depending on the requirements, three different setups can be utilized for various optical tracking applications.

3.6.1. 3-Camera System

The basic setup is a 3-Camera system shown in Figure 3.27, where three cameras configured in the CameraLink Base mode are connected to PowerEye via the CLinkRx-TripleBase grabber board.

On PowerEye, three image data streams are fed simultaneously to the FPGA, where the feature point extraction takes place. The 2D feature point coordinates are calculated by the FPGA for each camera in parallel. The results are then sent to one or both DSPs, which are responsible for correspondence matching and 3D reconstruction that require low computational cost but highly complex math and control operations. A host, such as a laptop, can access PowerEye via the standard Gigabit Ethernet or USB

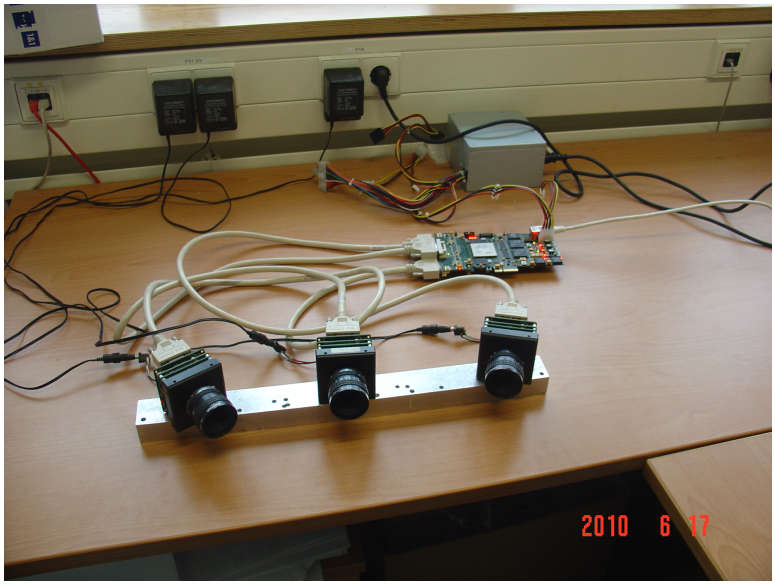


Figure 3.27.: 3-camera system

2.0 interface to initialize the cameras, visualize the 3D tracking data and perform the overall system control at runtime.

3.6.2. 6-Camera System

It is easy to extend the 3-camera system to a 6-camera system by cascading two PowerEye boards via the high-speed LVDS link, as illustrated in Figure 3.28.

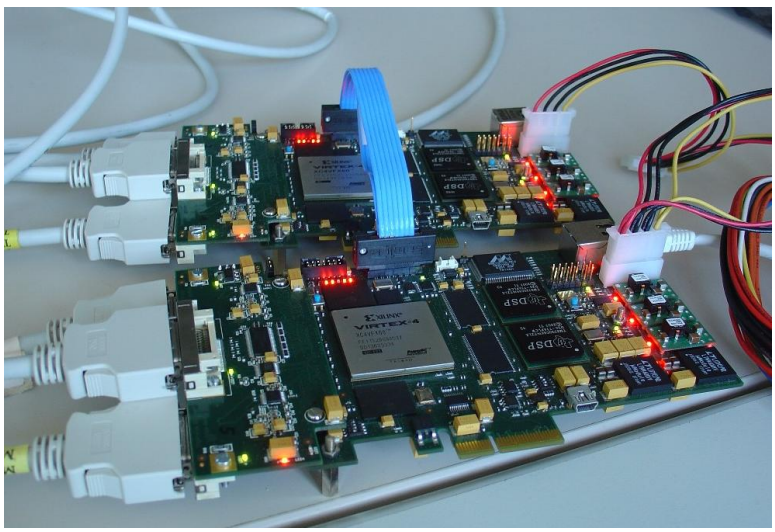


Figure 3.28.: 6-camera system

In this system, the two PowerEye boards can work in a Master-Slave fashion. The FPGAs of both master and slave PowerEye calculate the 2D feature point positions for each camera independently as in the 3-camera system. The slave FPGA (FPGA on

the slave PowerEye) sends its local results to the master FPGA, where the 2D feature point information from all the 6 cameras are collected. One of the master DSPs is responsible for establishing the feature point correspondence across the cameras, while the second master DSP remains free for the final 3D reconstruction.

It is obvious that the inter-board communication is an important issue that can influence the overall system performance. As discussed in Section 3.5.7, the LVDS link provides enough data transfer bandwidth for exchanging 2D feature point information between master and slave PowerEye. It even allows real-time raw image data to be transmitted from one PowerEye to another.

3.6.3. Many-Camera System

For applications that require even more cameras, it is possible to build a networked many-camera system using the Gigabit Ethernet interface available on PowerEye. Figure 3.29 illustrates an architectural diagram.

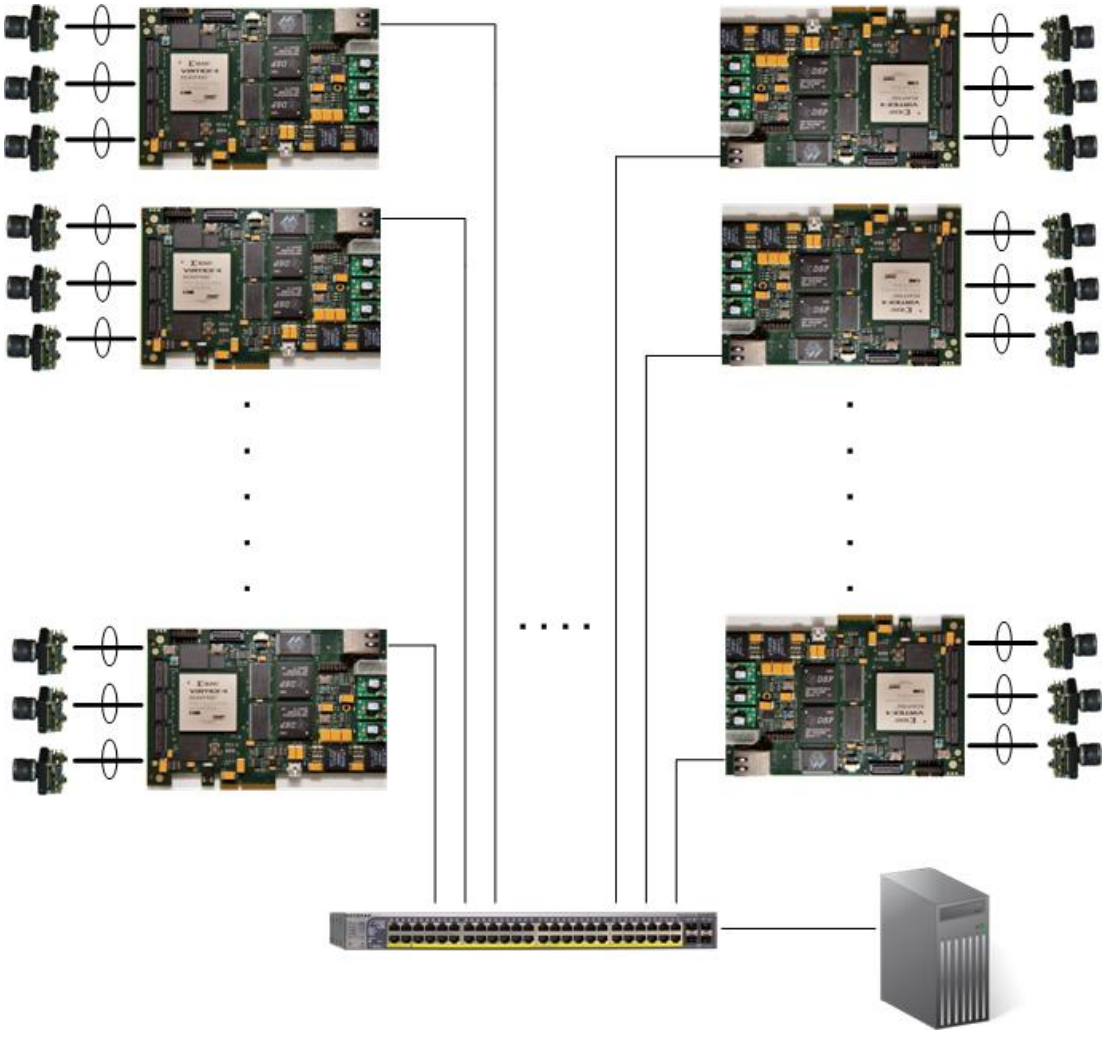


Figure 3.29.: Many-camera system

In such a system, every three cameras are grouped together and connected to a PowerEye board, forming a processing node. The FPGA and DSPs on each PowerEye board perform the 2D image analysis and calculate the feature point positions in images captured by the local cameras. The results are then sent to a central processor or a server over Ethernet. The server can be a standard PC, which integrates the 2D information from various processing nodes and estimates the 3D position and orientation of the targets to be tracked. Interconnecting processing nodes with the server can be easily achieved using standard Ethernet switches. Since the 2D object information consumes very little bandwidth of Gigabit Ethernet, we can scale to large numbers of cameras to cover a large tracking volume.

One of the most challenging problems in an optical tracking system with a large number of cameras is camera synchronization. When tracing moving objects, we must ensure that the images are captured at exactly the same moment in time. If the images from various cameras are captured at different moments, corresponding feature points from different views may not represent the same point in space. This will in turn lead to significant amount of error in the final 3D reconstruction step. Thus, it is a reasonable requirement to have strictly time-synchronized cameras to guarantee correct tracking results. The problem of camera synchronization becomes non-trivial as the number of cameras increases. In this thesis, an efficient solution to this problem is presented. More details are explained in Section 5.3.4.

3.7. Summary

In this chapter, the hardware design for the proposed optical tracking system is described. The complete hardware system is divided into three sub-systems : the high-speed camera, the CameraLink grabber and the PowerEye image processing system. The camera features a modular architecture, allowing to easily adapt with various image sensors and interfaces. The currently used MT9M413 CMOS sensor is able to output 500 images per second at the resolution of 1280×1024. The camera integrates a low-cost FPGA for sensor control and communication with the outside world. CameraLink was chosen as the camera interface to transmit large amounts of pixel data in real-time. Two different CameraLink grabbers have been developed, which can be used to interface three CameraLink BASE cameras and one CameraLink Full camera respectively. The PowerEye image processing system takes advantages of both FPGA and DSP to perform complex image processing algorithms at high frame rate. A CameraLink Simulator capable of simulating the behavior of three cameras simultaneously was implemented for simplifying the system verification. The flexible hardware architecture allows to construct a highly scalable optical tracking system.

4. FPGA accelerated 2D Image Processing

Image processing is mostly very demanding of computing power due to the huge volume of data to be processed. Even some of the simplest algorithms are computationally intensive, often requiring multiple additions and multiplications for each pixel [And96]. In an optical tracking system, the highest computational load comes from the 2D image processing tasks related to calculating the location of the feature points in each image captured by different cameras. This chapter focuses on high speed 2D image processing with FPGA based hardware acceleration. The implemented algorithms are frequently used in optical tracking applications, including color segmentation, noise reduction, edge detection, morphological filtering and blob analysis.

4.1. Color Segmentation

Segmentation is a commonly used technique in optical tracking, where target objects are isolated from the rest of the scene in the image, and then tracked by analyzing the change of positions in successive frames [CTJG05]. One simple approach is the color segmentation, in which objects are segmented by analyzing the color information associated with each pixel. Since the calculation only involves simple repetitive operations, color segmentation can be easily mapped to an FPGA. The segmentation process normally consists of two steps:

i) color space conversion

ii) color thresholding

4.1.1. Color Space Conversion

A color space is the format in which pixels are represented when captured, stored or transmitted in an image processing system. There are three color spaces frequently used for image segmentation - RGB, YUV and HSI. A summary of the characteristics of these color spaces is presented in [col09].

Bayer to RGB

The bayer pattern color filter array (CFA) is widely used in single-sensor color cameras to add color information to the raw pixels [Bra94]. As shown in Figure 4.1, the odd

rows of the filter array are comprised of alternating green and blue pixels, and even rows are comprised of alternating red and green pixels. As a result, half of the total number of pixels are green (G), while a quarter of total number are assigned to both red (R) and blue (B). This color filtering scheme corresponds to the fact that human eyes are most sensitive to the color green.

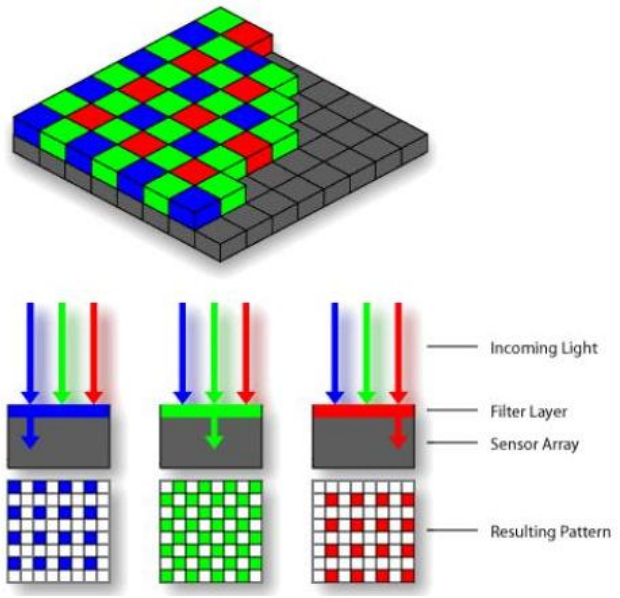


Figure 4.1.: Bayer filter [col09]

The MT9M413 color sensor is equipped with a bayer CFA. To apply color segmentation to the captured images, it is first necessary to convert the bayer color to the RGB color space. This process requires interpolating the two missing colors for each pixel. The most commonly used approach is the neighbor interpolation, which utilizes the color information of pixels in the 3×3 neighborhood to compute the missing colors of each pixel. The algorithm can be described by Equation 4.1, where four possible cases depicted in Figure 4.2 must be considered.

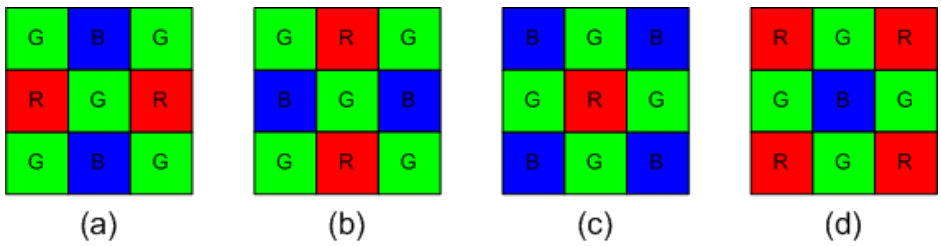


Figure 4.2.: Four possible cases of the bayer color interpolation

$$case(a): \begin{cases} R(x,y) = \frac{p(x-1,y)+p(x+1,y)}{2} \\ G(x,y) = \frac{p(x,y)+p(x-1,y-1)+p(x+1,y-1)+p(x-1,y+1)+p(x+1,y+1)}{5} \\ B(x,y) = \frac{p(x,y-1)+p(x,y+1)}{2} \end{cases} \tag{4.1a}$$

$$case(b): \begin{cases} R(x,y) = \frac{p(x,y-1)+p(x,y+1)}{2} \\ G(x,y) = \frac{p(x,y)+p(x-1,y-1)+p(x+1,y-1)+p(x-1,y+1)+p(x+1,y+1)}{5} \\ B(x,y) = \frac{p(x-1,y)+p(x+1,y)}{2} \end{cases} \tag{4.1b}$$

$$case(c): \begin{cases} R(x,y) = p(x,y) \\ G(x,y) = \frac{p(x-1,y)+p(x+1,y)+p(x,y-1)+p(x,y+1)}{4} \\ B(x,y) = \frac{p(x-1,y-1)+p(x+1,y-1)+p(x+1,y-1)+p(x+1,y+1)}{4} \end{cases} \tag{4.1c}$$

$$case(d): \begin{cases} R(x,y) = \frac{p(x-1,y-1)+p(x+1,y-1)+p(x+1,y-1)+p(x+1,y+1)}{4} \\ G(x,y) = \frac{p(x-1,y)+p(x+1,y)+p(x,y-1)+p(x,y+1)}{4} \\ B(x,y) = p(x,y) \end{cases} \tag{4.1d}$$

In Equation 4.1, $p(x,y)$ represents the intensity of the pixel located at (x,y) in the raw image, while $R(x,y)$, $G(x,y)$ and $B(x,y)$ are the interpolated RGB colors. The FPGA implementation for the above listed equations is quite straightforward. Figure 4.3 illustrates a block diagram for Equation 4.1a. Equation 4.1b, 4.1c and 4.1d can be implemented in a similar manner.

As shown in Figure 4.3, the incoming pixels are shifted into a delay logic consisting of two line buffers and six registers. The delay logic provides simultaneous access to nine adjacent pixels that form a 3×3 pixel window. The pixels are summed up by adders, marked with ADD, and then divided by dividers, marked with DIV. The output is a pixel stream, in which each pixel is represented in the RGB format.

Color segmentation can be performed directly in the RGB color space. However, this method is frequently unreliable, because segmentation in the RGB space is very sensitive to luminance changes. The reason lies in the high correlation among the R, G and B components [Tom86, Cel90]. For instance, if the luminance changes, all the three components will change accordingly. To achieve better segmentation results, the RGB color often needs to be converted to other color spaces, e.g. YUV or HSI, in which the luminance and the chrominance components are separated.

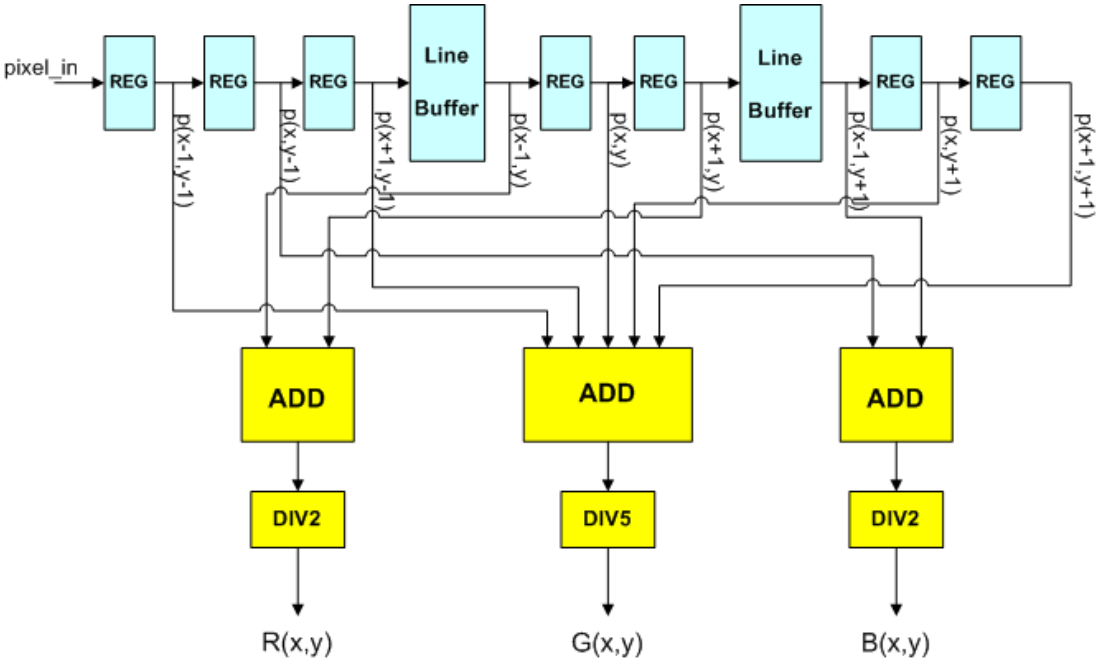


Figure 4.3.: FPGA implementation for Equation 4.1a

RGB to YUV

The YUV color space is widely used in video and broadcasting [Rus02]. It is also one of the successful color models for accurate color segmentation, since all information about the luminance is given by the Y component, while the U and V components representing the chrominance are independent from the luminance. A standard RGB to YUV transformation is given by the following formula:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & 0.081 \end{bmatrix} \times \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{4.2}$$

Performing floating point calculations on an FPGA will cause a large area cost. As can be observed in Equation 4.2, the R, G and B components of each pixel only need to be multiplied by a constant value. A simple scaling operation illustrated by Equation 4.3 enables the calculation to be converted from floating-point to fixed-point.

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 306 & 601 & 117 \\ -173 & -339 & 512 \\ 512 & -429 & 83 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \times \frac{1}{2^{10}} \tag{4.3}$$

Figure 4.4 depicts the hardware implementation of the fixed-point RGB to YUV color conversion. As can be seen, the circuitry only requires seven fixed-point multipliers and three adders to perform the desired computation.

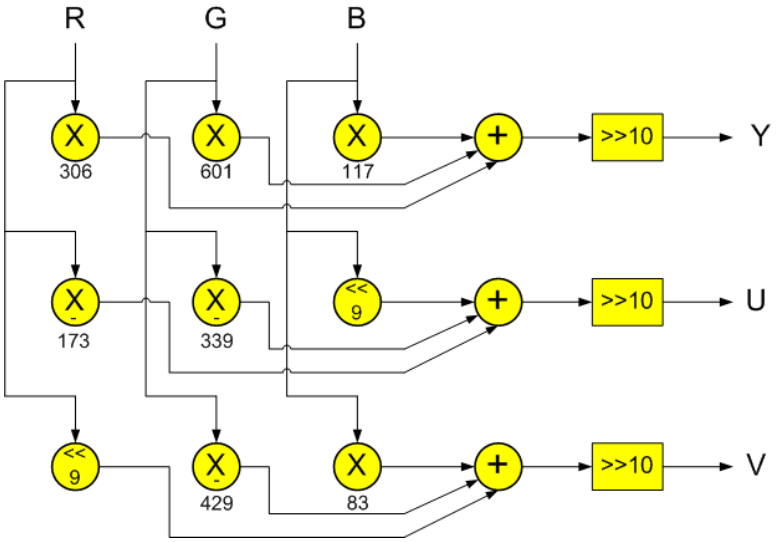


Figure 4.4.: RGB to YUV color conversion

RGB to HSI

The HSI color space is an important color model for image segmentation applications, where color information is represented by hue (H) and saturation (S) values, while the intensity (I), which describes the brightness of an image, is determined by the amount of the light [JKS95].

The hue component describes the color in the form of an angle between 0° and 360°, where 0° means red, 120° means green, and 240° means blue. The saturation component indicates how much the color is polluted with white color and ranges from 0 to 1. The range of intensity is also [0, 1], where 0 represents black, and 1 represents white. The HSI model decouples the intensity component from the color-carrying information (hue and saturation), which makes it very valuable for color segmentation.

The conversion from RGB to HSI is defined as follows:

$$I = \frac{R+G+B}{3} \tag{4.4a}$$

$$S = 1 - \frac{3}{R+G+B} \times MIN(R,G,B) \tag{4.4b}$$

$$H = cos^{-1}\left[\frac{1/2[(R-G)+(R-B)]}{\sqrt{(R-G)^2+(R-B)(G-B)}}\right] \tag{4.4c}$$

Equation 4.4 has a significantly higher computational load than Equation 4.3, as it contains complicated trigonometric and square root functions. Although a logic-level implementation using the CORDIC [And98] method is achievable, large amounts of hardware resources will be required. In practice, such functions are often performed by means of Look-Up-Tables (LUTs).

In its simplest form the LUT approach involves pre-calculating the results of an expression or function for each possible input [JGB04]. The results are loaded either into an FPGA local RAM (e.g. BlockRAM) or in an off-chip memory, depending on the size of the input vectors. During the execution, the results that correspond to the input (memory address) are readout from the LUT with constant delay, which eliminates the need for performing complicated computations. The ZBTSRAMs on PowerEye provide sufficient access bandwidth and storage capacitance, and thus present an ideal choice for the implementation of a LUT based RGB to HSI color conversion.

4.1.2. Color Thresholding

Thresholding is a point operation that performs a comparison on each pixel to make a distinction between the interesting objects and the surrounding background. Let us take the color segmentation in the HSI space as an example. Typically a segmentation criterion is defined, which consists of three pairs of threshold values applied to each color component. This can be described by Equation 4.5.

$$I_b(x,y) = \begin{cases} 1 & \text{if } (H_{min} < H(x,y) < H_{max}) \text{ AND} \\ & \quad (S_{min} < S(x,y) < S_{max}) \text{ AND} \\ & \quad (I_{min} < I(x,y) < I_{max}) \\ 0 & \text{else} \end{cases} \tag{4.5}$$

In Equation 4.5, the threshold is composed of three pairs of upper limit value and lower limit value. If all the H, S and I values of a pixel fall within the desired range, the output is 1, otherwise the output is 0. The result is a binary image ($I_b(x,y)$), where pixels with value 1 represent the *foreground* pixels, while pixels with value 0 are referred to as the *background* pixels. Color thresholding in the YUV space can be performed in a similar way.

Figure 4.5 depicts a simplified block diagram of a color segmentation system implemented on FPGA. The raw 8-bit pixels output by the MT9M413 sensor with Bayer CFA are captured by the Video Grabber module. The Bayer-to-RGB Conversion module interpolates the missing two colors for each pixel and yields an output image in the RGB888 format, where each color component is sampled with 8-bit. Since the on-board ZBTSRAM only provides a 20-bit address space, the lowest two bits of the R and B color components are dropped. This is done by the RGB Subsampling module. The resulting 20 bits RGB686 color values are then used to address the LUT stored in the ZBTSRAM, where the RGB to HSI color conversion takes place. The LUT is pre-calculated and can be loaded into the ZBTSRAM dynamically. Once the color conversion is performed, the image can be thresholded by choosing the preferred upper and lower limits for H, S and I, depending on the color characteristics of the objects

and the lighting conditions of the environment. The result is a binary image, where pixels are valued by either 1 (foreground) or 0 (background).

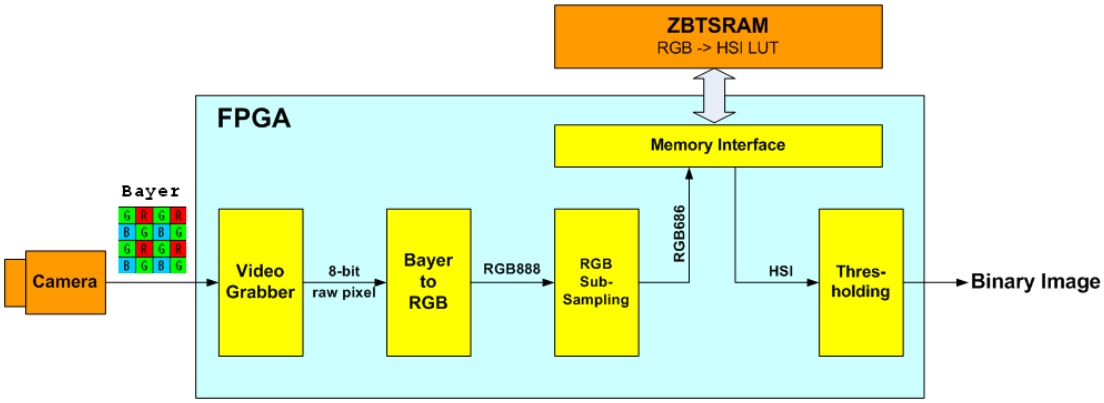
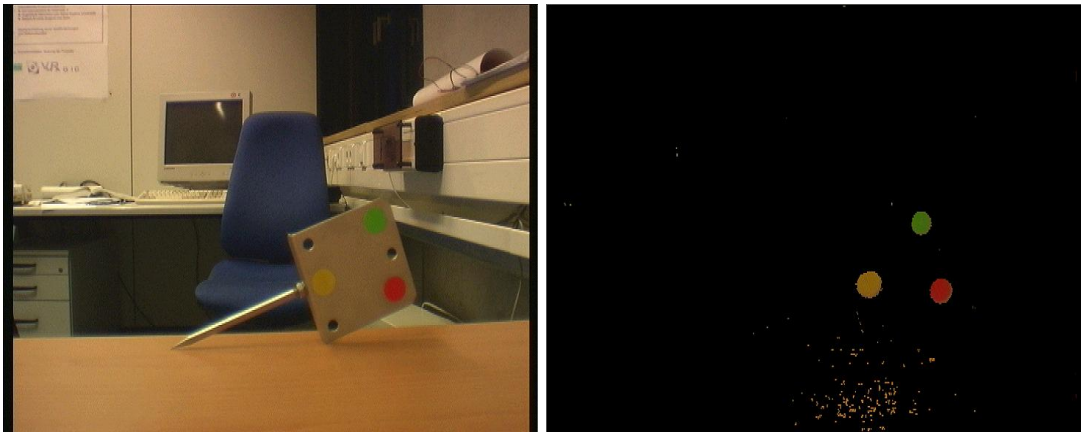


Figure 4.5.: Color segmentation on FPGA

Figure 4.6 illustrates an example of color segmentation. As can be seen, three round markers in different colors (red, green and yellow) are isolated from the background.



(a) Original image (b) Image after color segmentation

Figure 4.6.: An example of color segmentation

4.2. Noise Reduction

Images captured by a camera are often corrupted by random variations in intensity values, called noise [JKS95]. Noise reduction is a necessary step in most optical tracking systems, since the final tracking accuracy can be strongly affected by noise. In the frequency domain, noise is typically dominant for the high frequencies, whereas an image contains mostly low frequency information. Therefore, image noise can be efficiently reduced using low-pass filters.

In this section, the 2D image convolution is firstly introduced, which constitutes

the basis of the implementation for numerous image processing filters. Then three frequently used low-pass filters for image noise reduction, i.e., the mean filter, the Gaussian smoothing filter and the median filter, are described.

4.2.1. 2D Image convolution

A large number of image processing algorithms are based on filters in the frequency domain. For instance, image smoothing can be represented by low-pass filters, and image sharpening can be implemented by high-pass filters. One straightforward method is to convert the original image into its frequency domain using the Fourier Transform, and apply the desired filters afterwards [Chi06]. For instance, applying a low-pass filter means zeroing all frequency components above a cutoff frequency, applying a high-pass filter requires removing all the frequencies below some threshold frequency. Such filtering operations in the frequency domain can be performed using simple multiplication operations. The result after the frequency filtering is then converted back to the spatial domain by an Inverse Fourier Transform.

The main disadvantage of this method is the high computational complexity due to the Fourier Transform which could result in large FPGA logic resource consumption. An alternative approach is to perform a convolution in the spatial domain of an image. The convolution theorem states that multiplication in the frequency domain is equivalent to convolution in the time domain [ISUB05]. For an FPGA, the spatial domain convolution is often faster and easier than the Fourier Transform.

In two dimensional continuous space, the convolution of two functions $f(x, y)$ and $g(x, y)$ produces a resulting function $h(x, y)$. This can be formally defined as:

$$h(x,y) = f(x,y) * g(x,y) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(q,r)g(x-q, y-r)dqdr \tag{4.6}$$

In two dimensional discrete space, which is the case of the 2D image processing, the convolution is defined as follows:

$$I'[x,y] = I[x,y] * w[x,y] = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} I[m,n]w[x-m, y-n] \tag{4.7}$$

where I and I' represent the original image and the processed image respectively, and w is referred to as the convolution kernel or mask with the size given by $M \times N$. From an algorithmic perspective, 2D image convolution is a local process in which a convolution mask is slided over the input image to calculate the output pixel values. For each pixel located at (x, y) in I, a $M \times N$ window centered at this pixel is extracted, then all pixels in this window are multiplied by the corresponding weight defined by w, afterwards the products are summed up to produce the output pixel value. Figure 4.7 depicts a conceptual view of the 2D image convolution using a 3×3 mask.

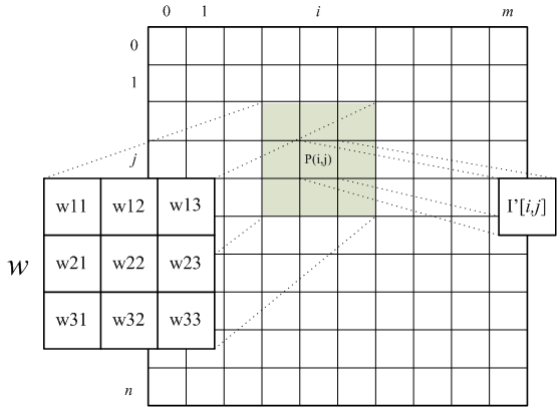


Figure 4.7.: 2D image convolution using a 3×3 mask

Despite its simple representation, the 2D image convolution task is both computationally expensive and memory-access-intensive. With a $M \times N$ convolution mask, the calculation requires $M \times N$ multiplications and $M \times N - 1$ additions, as well as $M \times N$ accesses to the incoming pixel data to get the result of a single output pixel [ZXH07]. Using an FPGA, one can exploit the inherent parallelism of the computation, and thus achieve high processing throughput. Figure 4.8 shows the structure of an FPGA-based implementation for a generic 2D image convolution.

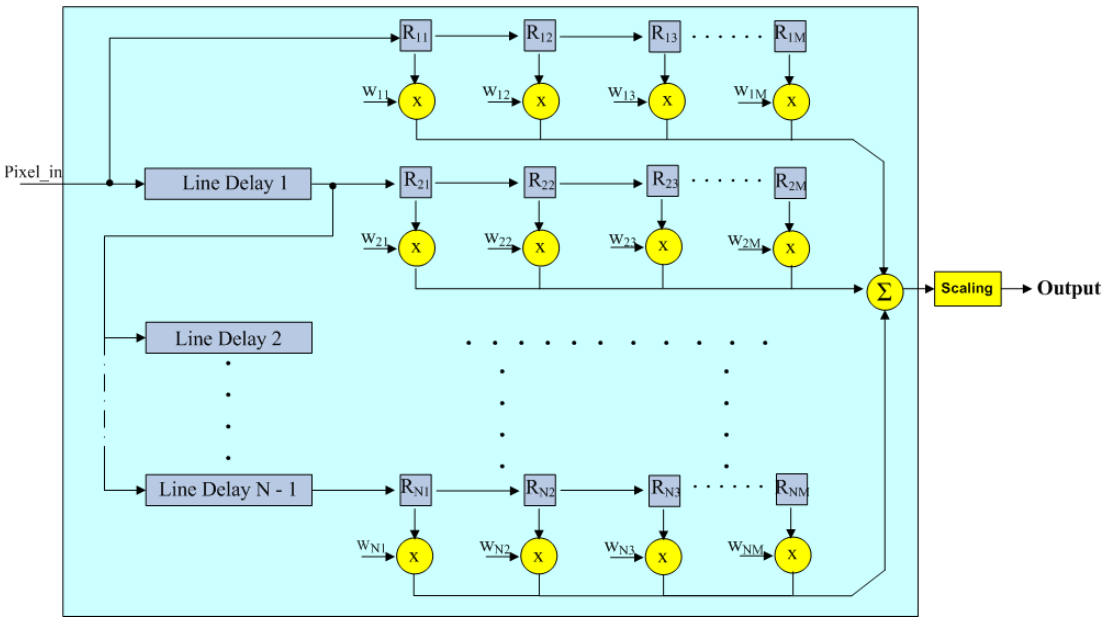


Figure 4.8.: FPGA based generic 2D image convolution

To speed up the 2D convolution, simultaneous access to $M \times N$ pixels belonging to the convolution window must be provided. This allows the calculation for all pixels in the window to be performed in parallel. As shown in Figure 4.8, $N - 1$ line buffers and N sets of register arrays, each consisting of M shift registers, are utilized for this

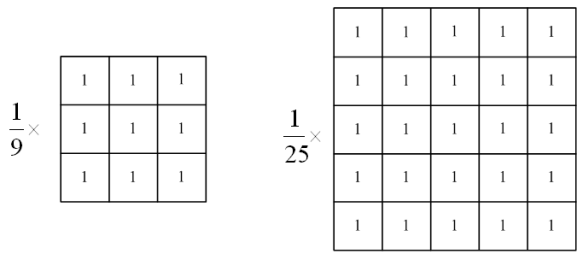
purpose. A line buffer with the depth equal to the number of pixels in each line of the image is capable of delaying a pixel for a whole image line, and a shift register delays a pixel by one time step. The incoming pixels are shifted into the FPGA in a raster scan order. As soon as $N-1$ raster lines and M pixels in the current line are loaded, all pixels belonging to the first $M \times N$ convolution window are available. At each node of the shift register array, the pixels are multiplied with the appropriate filter coefficients determined by the convolution mask, and then all the multiplier products are added together by a adder tree to produce the result. Typically, scaling is applied at the final output. When a new pixel is shifted in, the convolution window will move to the next position. The same process will be repeated until all pixels in the image are scanned.

4.2.2. Mean Filter

Mean filter is based on a local averaging operation where the value of each pixel is replaced with the average value of all the pixels in the local neighborhood:

$$I'[x,y] = \frac{1}{M} \sum_{(m,n)\in w} I[m,n] \tag{4.8}$$

In Equation 4.8, M is the total number of pixels defined by w. Mean filter can be calculated by convolving the input image with a mask, in which all coefficients have the value 1, as shown in Figure 4.9. Pixel values are summed with equal weight, then the sum is divided by a scaling factor, which is determined by the size of the convolution mask.



(a) 3x3 mean filter (b) 5x5 mean filter

Figure 4.9.: Convolution mask for mean filter

4.2.3. Gaussian Smoothing Filter

Gaussian filters are a class of linear smoothing filters with the weights chosen according to the shape of a Gaussian function given by Equation 4.9, where σ determines the sharpness of the Gaussian function.

$$G(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{x^2}{2\sigma^2}} \tag{4.9}$$

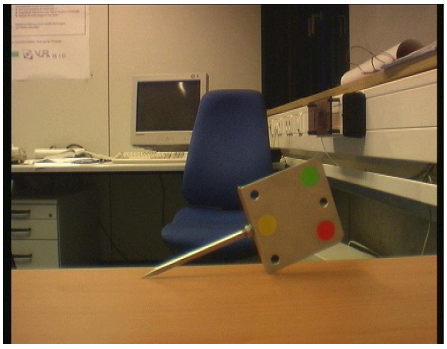
In the case of image processing, a two-dimensional zero-mean discrete Gaussian function is often used:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}} \tag{4.10}$$

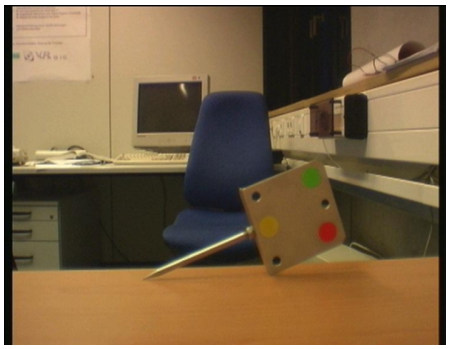
For the practical calculation, the Gaussian function is normally quantized into discrete values in order to develop a convolution kernel of a specific size. Figure 4.10 illustrates a 5×5 integer-valued convolution kernel that approximates a Gaussian function with a σ of 1.0.

$$\frac{1}{256} \times \begin{array}{|c|c|c|c|c|} \hline 1 & 4 & 6 & 4 & 1 \\ \hline 4 & 16 & 24 & 16 & 4 \\ \hline 6 & 24 & 36 & 24 & 6 \\ \hline 4 & 16 & 24 & 16 & 4 \\ \hline 1 & 4 & 6 & 4 & 1 \\ \hline \end{array}$$

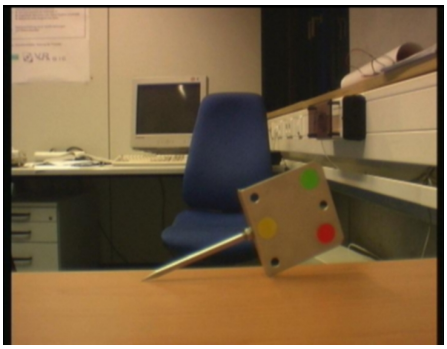
Figure 4.10.: 5×5 Gaussian convolution kernel with $\sigma = 1.0$



(a) Original image



(b) 3×3 Mean filter



(c) 5×5 Gaussian smoothing filter

Figure 4.11.: Effect of mean and Gaussian smoothing filter

4.2.4. Median Filter

Both the mean filter and the Gaussian smoothing filter are linear low pass filters, which have the disadvantage of blurring sharp discontinuities in intensity values while trying to suppress the noise in an image. This effect is shown in Figure 4.11.

The intensity discontinuities usually contain useful information for image analysis, e.g. edges and corners. Consequently they should be preserved as well as possible while performing the noise reduction. An alternative approach is the non-linear median filter, which is very effective to remove the impulsive noise from an image, while preserve the sharp edges at the same time [HS93]. The idea is to replace each pixel value with the median value in the local neighborhood. This can be done by first sorting all the pixel values from the surrounding neighborhood into numerical order and then picking the middle pixel value as the output [Bax94]. Figure 4.12 illustrates an example using a 3×3 neighborhood.

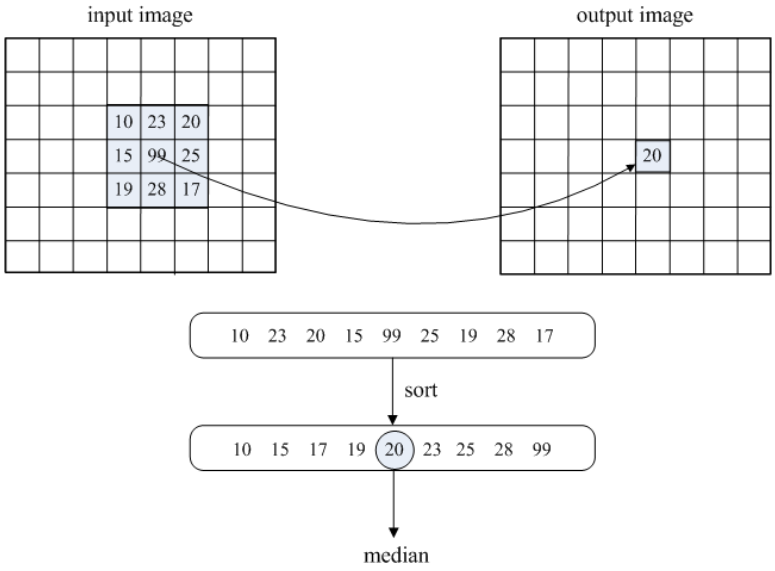


Figure 4.12.: Median filter using a 3×3 neighborhood, from [RPBM06]

Median filter has a high computational cost, because for sorting N pixels the temporal complexity is $O(N \times logN)$, even with the most efficient sorting algorithms. In Figure 4.13 a block diagram of an FPGA based 3×3 median filter is illustrated. Unlike the architecture of the 2D convolution shown in Figure 4.8, the median filter is composed of a set of comparators rather than adders and multipliers.

The FPGA circuitry contains a 3×3 pixel moving window that allows the current pixel and all the 8 neighborhood pixels to be accessed simultaneously. A sorting network consisting of 19 Processing Nodes (PN) performs pixel sorting in multiple stages to produce the result. As detailed in Figure 4.14, each PN is formed by a comparator together with two 2 : 1 multiplexers for sorting two input pixels. The lower input exits the node on the top (port L), while the higher leaves on the bottom (port H). The

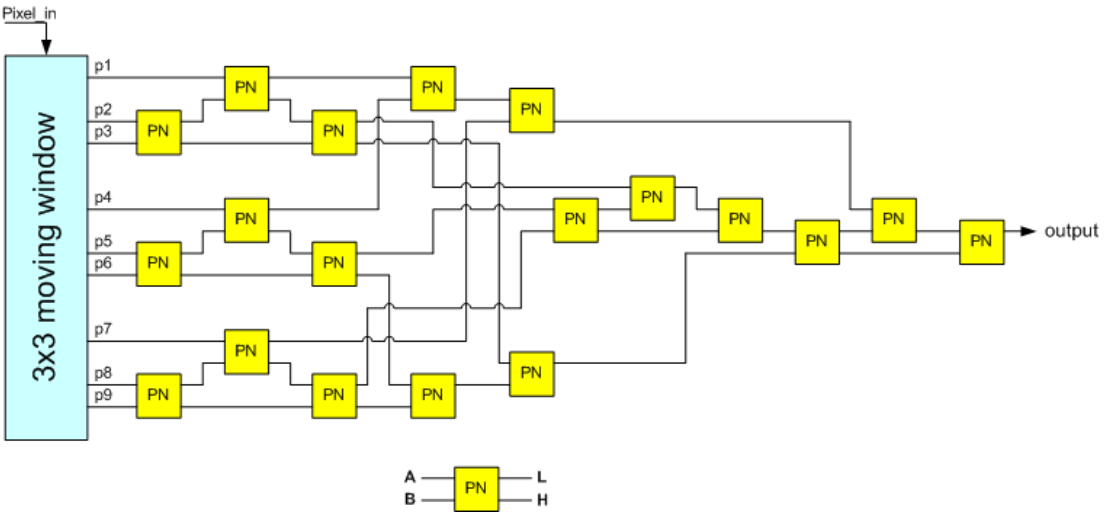


Figure 4.13.: FPGA block diagram for the 3×3 median filter

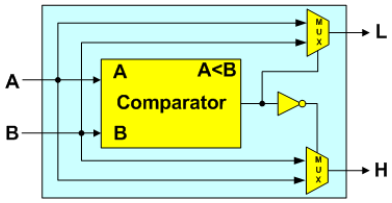


Figure 4.14.: Sturcture of the Processing Node in a media filter

intermediate comparison results are fed into PNs in the subsequent stage. The final result representing the median value of the pixels in a 3×3 window comes from the L port of the last PN.

Figure 4.15 illustrates the performance of the median filter. As can be seen, the orignal image is strongly corrupted by impluse noise. After applying a 3×3 median filter the noise is effectively removed while most image details, such as sharp edges, are retained.

(a) Original image with impulse noise (b) 3 × 3 median filter

Figure 4.15.: Effect of 3×3 median filter

4.3. Edge Detection

Edges are considered to be one of the most important features that provide valuable information for image analysis. In an image, edges typically occur on the boundary between two different regions, resulting in significant local changes in pixel intensity [Lin96]. Edge detection is a fundamental tool used in many image processing applications as a precursor step to feature extraction and object segmentation [FS09].

Most edge detectors make use of a gradient operator that calculates the level of variance between different pixels. In the case of image processing, the gradient is defined as a vector which represents the two-dimensional equivalent of the first derivative [GW06]:

$$G[f(x,y)] = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \partial f/\partial x \\ \partial f/\partial y \end{bmatrix} \tag{4.11}$$

The magnitude and the direction of the gradient are given by Equation 4.12 and Equation 4.13 respectively.

$$G(x,y) = \sqrt{G_x^2 + G_y^2} \tag{4.12}$$

$$\alpha(x,y) = arctan(\frac{G_y}{G_x}) \tag{4.13}$$

In Equation 4.13, the angle α is measured with respect to the x axis.

Sobel Edge Detector

The Sobel edge detector is one of the most frequently used edge detectors in image analysis. It utilizes two 3×3 kernels that are convolved with the original image to calculate the approximations of the derivatives in two perpendicular directions - one for horizontal, and the other for vertical. Let I be the input source image, and G_x and G_y be the resultant images containing the horizontal and vertical derivative information respectively, then the computation can be described as follows:

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * I \qquad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * I \tag{4.14}$$

where $*$ represents the 2D convolution. In hardware implementation, the calculation of the gradient vector magnitude is often simplified using Equation 4.15:

$$G(x,y) = |G_x| + |G_y| \tag{4.15}$$

A pixel located at (x,y) in the resulting edge image is considered an edge pixel $E(x,y)$, if its magnitude $G(x,y)$ exceeds some predefined threshold T.

$$E(x,y) = \begin{cases} 1 & \text{if } G(x,y) > T \\ 0 & \text{else} \end{cases} \tag{4.16}$$

The threshold T in Equation 4.16 is typically chosen using the cumulative histogram of $G(x,y)$ such that 5 to 10 percent of pixels with largest gradients are declared as edges [Jai88].

Figure 4.16 depicts the FPGA implementation for the above described Sobel edge detector. Since the convolution masks used for calculating G_x and G_y only contain values of 0, ± 1 and ± 2, the computation can be performed by a combination of simple logic elements such as adders, subtractors and shift registers.

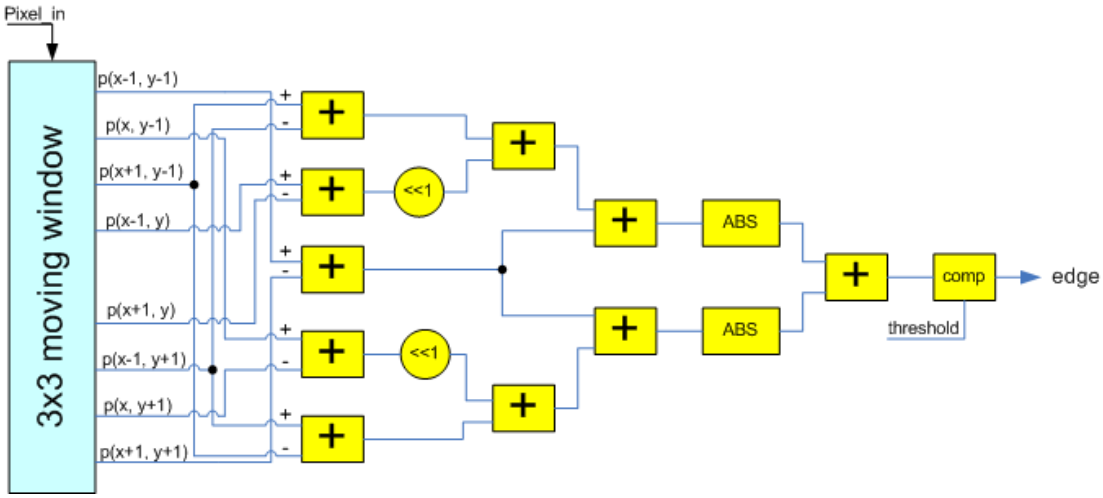
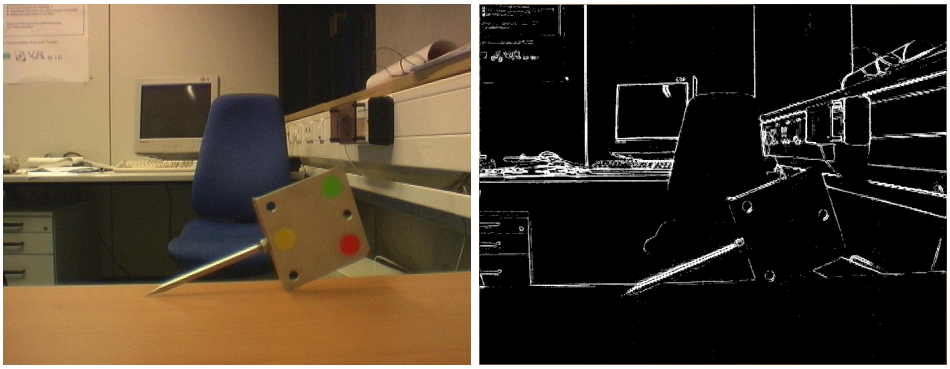


Figure 4.16.: 3×3 Sobel edge detector

Figure 4.17 shows the result of the Sobel edge detector described above.



(a) Original image (b) Sobel edge image

Figure 4.17.: Result of Sobel edge detection

4.4. Morphological Filter

The morphological filters are a set of digital image processing filters based on the concept of mathematical morphology. They are widely used for image pre-processing, such as thinning, thickening, skeletonization and pruning [Abd07].

Dilation and Erosion are two basic morphological operations, since most existing morphological filters can be realized by a combination of these two operations [Soi99]. Dilation and Erosion are normally applied to binary images, although there are gray level versions. In this section, we only deal with binary images, where pixels are grouped into foreground (with value 1) and background (with value 0).

4.4.1. Dilation

Let A be the input binary image and B represent the structuring element (SE) used to process A. Dilation is defined by the following equation:

$$A \oplus B = \left\{ z | [(\hat{B}) \cap A] \subseteq A \right\} \tag{4.17}$$

The implementation of Dilation is performed by moving the SE over the image and setting the center pixel to 1 if any pixel in the neighborhood has the value 1; otherwise setting the center pixel to 0. Applying dilation to an image can increase the sizes of objects, fill holes and connect areas that are separated by spaces smaller than the size of the SE.

4.4.2. Erosion

Erosion on the other hand can be considered a narrowing of features on an image. Again define A as the input binary image and B as the structuring element, the process of Erosion can be described by:

$$A \ominus B = \{ Z | (B)_z \subseteq A \} \tag{4.18}$$

Erosion is performed by moving the SE over the image and setting the center pixel to 1 if all of the pixels in the neighborhood have the value 1; otherwise setting the center pixel to 0. This operation decreases the sizes of objects and removes small anomalies by subtracting objects with a radius smaller than the structuring element.

4.4.3. Opening and Closing

Dilation and Erosion can be combined into complex sequences, such as Opening and Closing. As mentioned before, Erosion can be used to eliminate small clumps of undesirable foreground pixels. However, it will affect all regions of foreground pixels. Opening overcomes this problem by performing Erosion followed by Dilation on an image:

$$A \circ B = (A \ominus B) \oplus B \tag{4.19}$$

Dilation is able to fill small background holes in images. Meanwhile, it also distorts all regions of pixels indiscriminately. One can reduce some of this effect by performing an erosion on the image after an dilation, i.e., a closing which is defined as:

$$A \bullet B = (A \oplus B) \ominus B \tag{4.20}$$

Designing a morphological filter needs to consider the connectivity path among pixels in an image. Two fundamental pixel connectivity schemes shown in Figure 4.18 are frequently used in morphological image analysis.

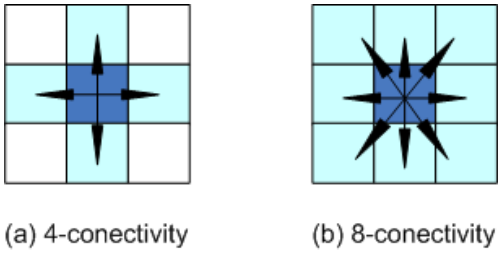


Figure 4.18.: 4- and 8-connectivity

For a pixel p with the coordinate (x, y) the set of pixels in the neighborhood is given by:

$$N_4(p) = \{(x+1, y), (x-1, y), (x, y+1), (x, y-1)\} \tag{4.21}$$

$$N_8(p) = N_4(p) \cup \{(x+1, y+1), (x-1, y+1), (x-1, y-1), (x+1, y-1)\} \tag{4.22}$$

According to Equation 4.21 and Equation 4.22, two pixels p and q are *4-connected* if q is from the set $N_4(p)$ and *8-connected* if q is from $N_8(p)$.

In Figure 4.19 the hardware diagram for a dilation filter supporting both the 4- and 8-connectivity is depicted. The circuitry only contains a set of OR gates since the process of dilation is to output the value 1 if any of the pixels in the neighborhood is 1, otherwise 0.

The architecture of the erosion filter is the same as that of the dilation filter, apart from using AND gates instead of OR gates. Opening and closing functions can be easily implemented by cascading dilation and erosion filters in the desired order.

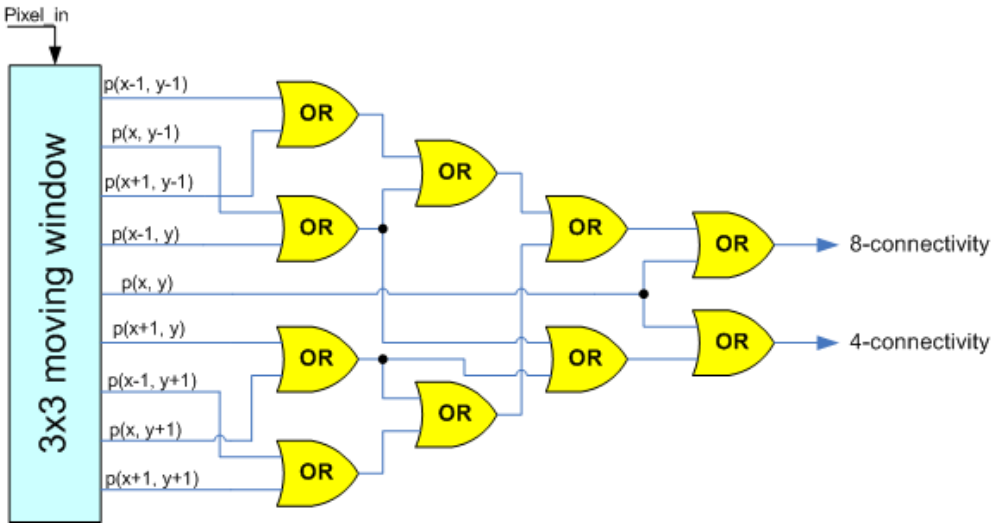


Figure 4.19.: Implementation of a Dilation filter

4.5. Parallel Image Processing on FPGA

The 2D image processing algorithms introduced so far are either pixel operators (e.g. color conversion) or neighborhood operators (e.g. noise reduction, edge detection and morphological filtering), both of which are dependent only on a limited portion of the input image data and require no recursive operations. They are all suited for implementation utilizing a certain type of parallelism because of the lack of data dependencies [AS89].

Using an FPGA, one can exploit the inherent parallelism in the algorithm to achieve significant speed upgrade compared to the sequential software implementation. For most image processing tasks, there are three levels of parallelism to be exploited: instruction-level, data-level and task-level.

4.5.1. Instruction-level Parallelism

The most direct approach for applying instruction-level parallelism is *Pipelining*, which is a common technique widely used in modern general-purpose processors. The operation of such processors is based on the well-known *fetch-decode-execute-store* process. First, an instruction is fetched from memory, then it is decoded to address the relevant functional unit. After the execution takes place in the functional unit, the results are written back into memory. Pipelining allows the processor to overlap multiple instructions to reduce the execution time [dR06]. For instance, at a certain moment instruction A can be decoded while simultaneously instruction B is being fetched from memory. In this way, the processor can process two or more instructions at the same time (in parallel).

In FPGA computing, it is possible to construct a much deeper pipeline allowing a

large number of instructions to be overlapped and executed concurrently. Let us take the 5×5 Gaussian convolution introduced in Section 4.2.3 as an example.

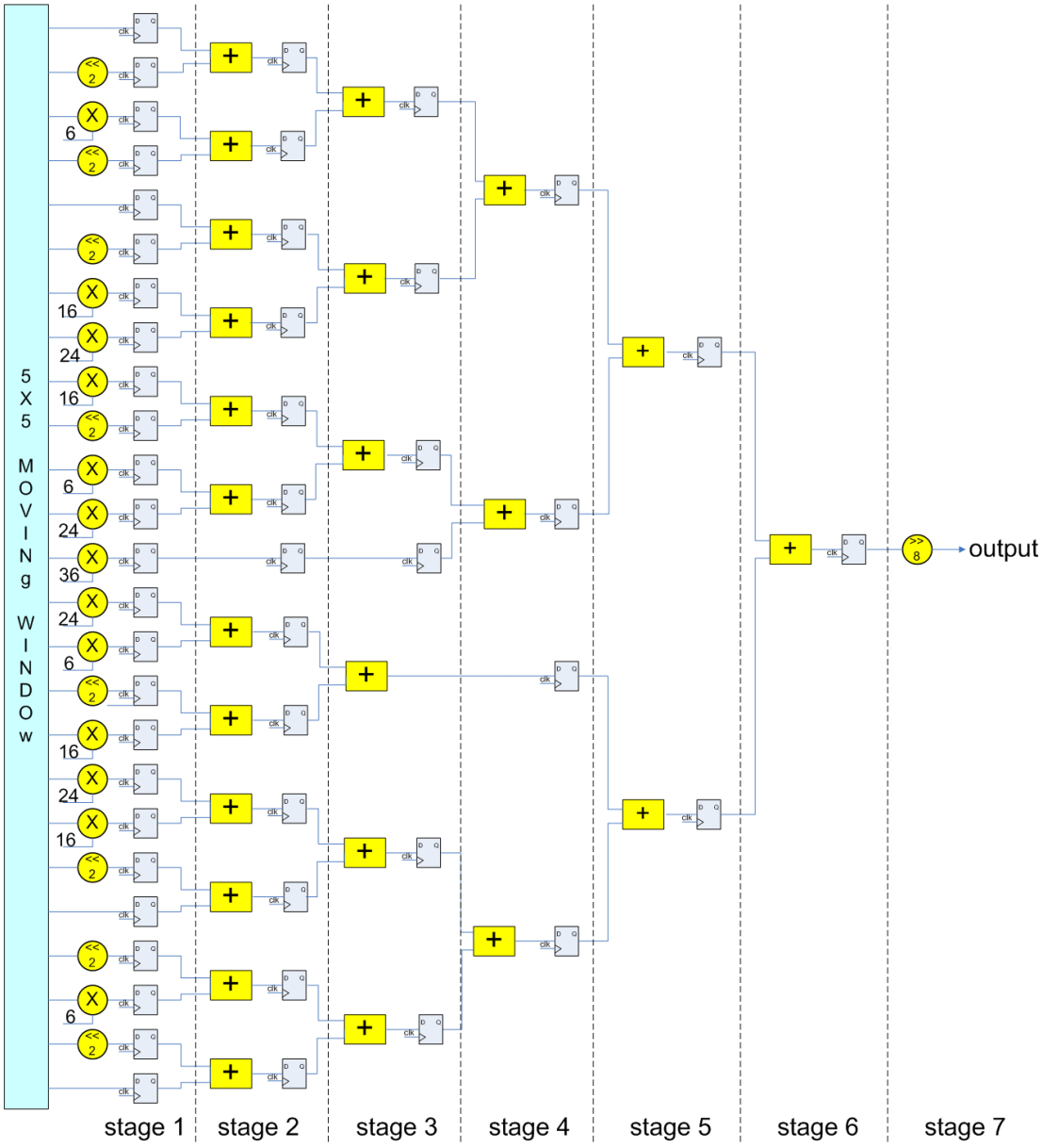


Figure 4.20.: Pipelined 5×5 Gaussian filter

Figure 4.20 depicts a pipelined 5×5 convolution architecture. The calculation is divided into 7 pipeline stages. In the first stage, 25 pixels in the 5×5 window are multiplied with the weights given by the convolution mask. Since the FPGA adder primitive only supports two input operands, 24 adders distributed in 5 pipeline stages (stage 2 to stage 6) are employed to sum up the products of the multipliers. The final result is generated after the scaling operation performed by the last stage.

In Figure 4.20, all pipeline stages are connected in series, so that the output of one stage is the input of the next. In order to hold the intermediate results, registers

are placed between adjacent stages. Information that flows throughout the pipeline is controlled by a common clock applied to all the registers simultaneously. With this architecture, successive pixel windows can be streamed into the pipeline and get processed in an overlapped fashion. For instance, when the first pixel window is received, pipeline stage 1 may multiply the pixels with the weights. After stage 1 has finished the multiplications, stage 2 may begin adding the products while stage 1 receives and begins processing the second pixel window. Similarly, the 7th pixel window can be fed into stage 1 while the 1st pixel window is being processed by stage 7.

It is easy to understand that at the start of the processing 7 clock cycles are required to fill the pipeline before the first output pixel is available. However once the pipeline is filled, a new output pixel is generated every clock cycle. Therefore, processing throughput of the the pipeline architecture illustrated in Figure 4.20 is 1 pixel per clock cycle and the latency is n clock cycles, where n equals the number of the pipeline stages.

The pipelining strategy can be applied not only to a specified operator but also to a complete image processing system which requires a sequence of different operations to be performed on an image. For example, an object tracking system may comprise preprocessing, segmentation, feature extraction and target recognition. The intermediate result of one task is just the input of the next task. Therefore, different processing modules can be cascaded in a pipelined manner to allow a large number of operations to be executed in parallel [KDF01].

An important consideration for designing an image processing pipeline is how to maximize the operating frequency, as it is a clear way to improve the processing throughput. In an FPGA, all logical elements have an inherent propagation delay. That is to say, it takes a finite amount of time for any digital signal passing from one point to another. The maximum operating frequency of a pipeline is determined by the largest critical path delay between successive registers. A straightforward way to increase the clock speed is to break down the critical path by inserting additional pipeline stages at the expense of increased latency. However, this solution does not allow an infinitely high clock frequency due to the physical speed limitation of the FPGA (normally below 250MHz). For a real-time system, the image data to be processed must flow through the pipeline at least as fast as it is coming in. This requires that the processing pipeline must be clocked at a higher frequency than the frequency of the incoming pixel clock. Therefore, using a single path pipeline is not always adequate to high-speed optical tracking applications, where the image data rate can easily exceed 250MPixel/s.

4.5.2. Data-level Parallelism

It is possible to further parallelize the processing by distributing the image data across multiple processing elements (PEs) that perform a set of operations on a large number of data sets at the same time, as shown in Figure 4.21. This kind of parallelism is often referred to as data-level parallelism.

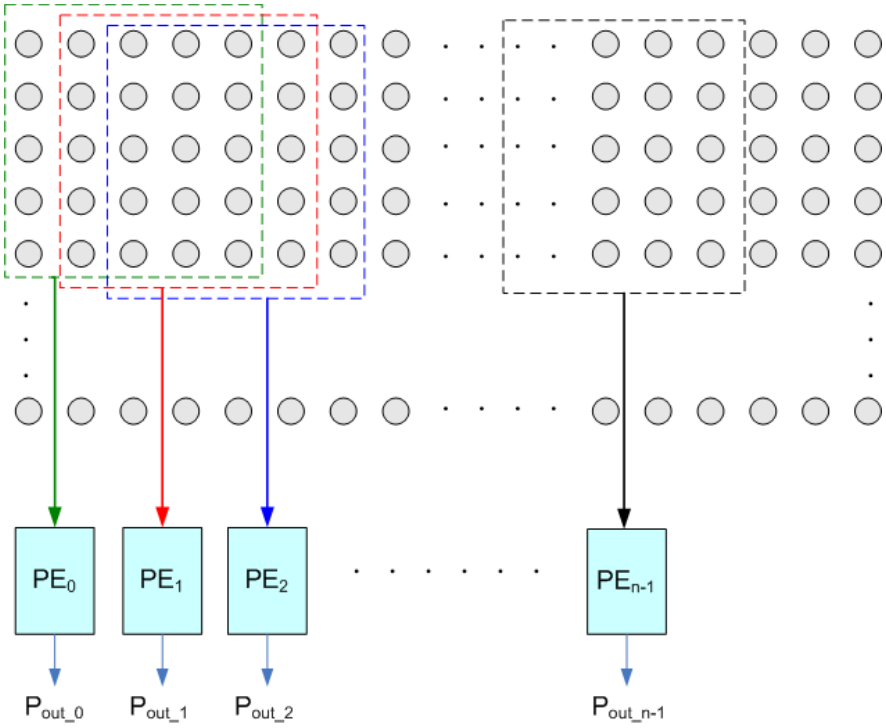


Figure 4.21.: Data-level parallelism using a PE array

In Figure 4.21, a PE array consisting of a number of PEs enables the possibility to process multiple adjacent pixel windows simultaneously. Each PE contains the identical pipeline, e.g. the 5×5 Gaussian smoothing filter illustrated in Figure 4.20, which produces one output pixel every clock cycle. As a result, N output pixels can be achieved per clock cycle, where N represents the number of PEs. Ideally the processing for a complete image could be accomplished within one clock cycle, if N equals the number of pixels in the image. In practice, however, it is only possible to fit a limited amount of PEs into the FPGA due to the hardware resource limitation. Thus, it is important to find an area-efficient implementation in order to maximize the number of PEs that are feasible to the target FPGA.

In image processing, a large number of 2D filters are based on symmetric and separable convolution kernels. A typical example is the 5×5 Gaussian smoothing filter introduced in Section 4.2.3. Figure 4.22 shows that such a 2D convolution can be divided into two 1D convolutions.

Equation 4.23 gives the expression for a generic separable 2D convolution.

$$\begin{aligned} I'[x,y] &= I[x,y] * w[x,y] \\ &= I[x,y] * f[x,y] * g[x,y] \\ &= \sum_{m=0}^{M-1} (\sum_{n=0}^{N-1} I[x-m, y-n] \cdot f[n]) \cdot g[n] \end{aligned} \tag{4.23}$$

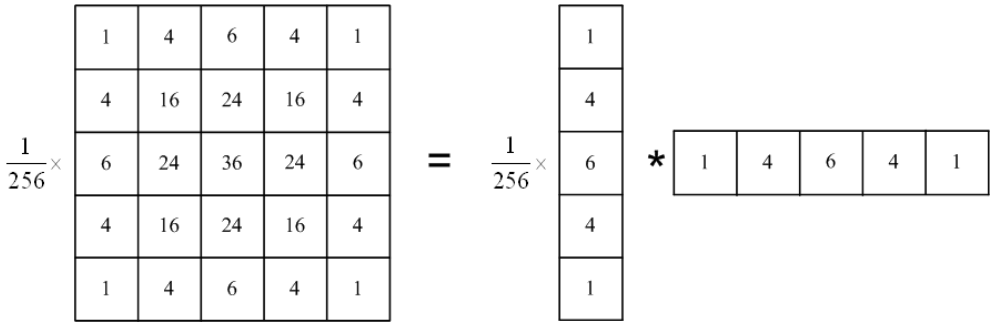


Figure 4.22.: Separable Gaussian convolution kernel

The hardware implementation is illustrated in Figure 4.23. In comparison with the architecture shown in Figure 4.8, a great number of LUTs, flip-flops and multipliers are saved.

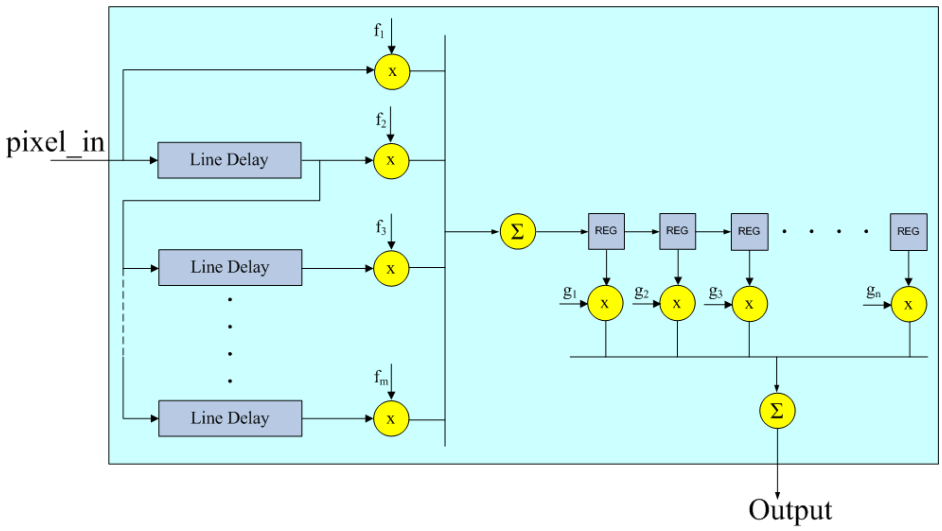


Figure 4.23.: FPGA based separable 2D image convolution

For non-separable convolution filters, area reduction can be achieved when taking the data reuse potential into account. As can be observed from Figure 4.21, a significant portion of pixels are overlapped among adjacent windows. This implies that these pixels can be reused for the computation of neighboring outputs. Therefore, it is not necessary to duplicate the processing element N times to build an N-PE array. The logic block used for processing the overlapped pixels can be shared by adjacent PEs. As a result, significant FPGA logic resources in terms of LUTs, flip-flops and multipliers can be saved.

Using the area reduction techniques described above, it is possible to implement a parallel image processing system in a single chip FPGA with good performance/area trade-off. As an example, we have successfully integrated 24 parallel PEs into the FPGA on PowerEye, each of which performs the 5×5 Gaussian convolution filter.

Since the FPGA is able to operate at 200MHz, a substantial processing throughput of $4800 Mpixels/s$ can be achieved. This translates into processing 4800 images per second at the resolution of 1024×1024. According to the performance evaluation presented in [CCLW05], the FPGA demonstrates a 60× acceleration over a GeForce 6800GT GPU, and more than 500× speed-up over a 3GHz Pentium 4 processor.

4.5.3. Task-level Parallelism

Task parallelism is the third level of parallelism that the FPGA can readily exploit. By examining the operation to be performed, multiple tasks may be identified which can operate independently without introducing any data hazards (data dependency conflicts) [dR06]. This kind of parallelism can be demonstrated by a typical scenario in an optical tracking system, where the FPGA handles multiple pixel streams from different cameras simultaneously. The processing tasks for each camera may be identical or different, depending on the functions to be implemented.

Although many image processing operators exhibit a high degree of parallelism due to their local nature and lack of data dependencies, there exist some complex cases where the operations are global and data dependency among pixels can not be avoided. In such cases, it is difficult to parallelize the algorithm using the parallelization techniques described above. A good example for this is the blob analysis discussed in the following section.

4.6. Blob Analysis

Blob analysis is one of the most fundamental image analysis tools, which is frequently used in infrared marker tracking applications. Since infrared markers usually have identical color, they are difficult to be tracked using the color segmentation approach discussed in Section 4.1. Instead of analyzing the color information associated with each pixel, blob analysis utilizes a region based segmentation method to identify the objects and determine their features.

Blob is defined as a maximally connected region of foreground pixels in a binary image. The task of blob analysis is to isolate blobs from the background and calculate their geometric features, which involve typically three processing stages. In the first stage the input gray-level or color image is converted to a binary image consisting of a number of regions against the background. Next, connected components labelling is performed to assign each region or blob a unique label, enabling distinct blobs to be distinguished. In the last stage, each blob is analyzed on the basis of the label information, and a number of geometric features, e.g., area, center of gravity, bounding box, are extracted. Figure 4.24 illustrates a simple example.

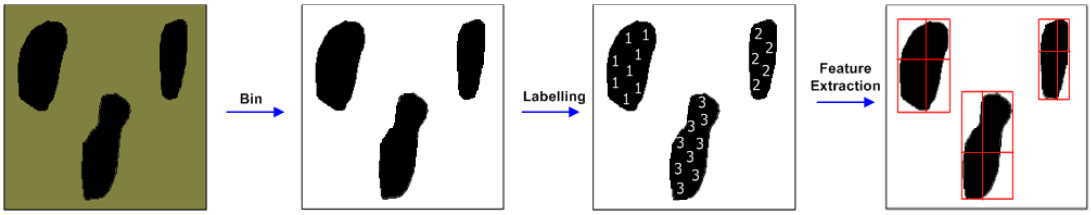


Figure 4.24.: Example of blob analysis

4.6.1. Classical Algorithm

In classical algorithms, Connected Components Labelling (CCL) is an essential step in blob analysis that focuses on separating blobs from the background. The input or source is a binary image in which all interesting objects are formed by foreground pixels. CCL translates the source image into a label image which has the property that all adjacent foreground pixels are assigned the same label as they belong to the same component, and each component owns a unique label to distinguish from each other. When analyzing pixel connectivities, either the 4- or the 8-connectivity scheme shown in Figure 4.18 can be used depending upon the application. In this thesis, we only consider the case of 8-connectivity.

For image processed in the raster-scan order, the algorithms for CCL require the information of labels in row N and row $N-1$ while labelling row N [RAA95]. Figure 4.25 shows the four neighbor pixels (assuming 8-connectivity is used) that need to be analyzed for labelling the current pixel p. The neighbor pixels are named $p1$, $p2$, $p3$, $p4$ to represent the left, the upper-left, the upper-center, and the upper-right pixel with respect to p.

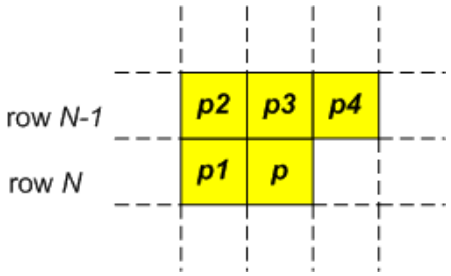


Figure 4.25.: 8-connectivity used for CCL

Since the shape of an object can be arbitrary, CCL involves significant data computation and communication and is therefore computationally intensive [WlCCcL03]. The most classical labelling algorithms [RP66][RK82] rely on two subsequent raster-scans through the image. In the first scan a temporary label is assigned to each foreground pixel by examining the labels of the neighbors that have already been visited. Specifically, if the neighborhood pixels ($p1, p2, p3, p4$ in Figure 4.25) are all background pixels, a new label is generated and assigned to the current pixel p. If the neighborhood con-

tains a single foreground pixel which was labelled by L, then L will be copied to p. If the neighborhood contains more than one foreground pixels, there are two possibilities to be considered. If $p1, p2, p3, p4$ were assigned the same label then p is given that label also. The second case is that the neighbors carry different labels, which occurs typically when a "U" shaped object is encountered. The two branches of "U" were assumed to be two individual regions, and therefore have been labelled by M and N respectively, where $M \neq N$. When they join at the pixel p, label M and label N must be merged. More precisely, one of the two labels is assigned to p and an equivalence pair is generated indicating that label M and label N belong to the same object. During the first scan, different labels may be associated with the same component [dSB99]. Thus, after completion of the first scan the equivalence pairs need to be resolved. This involves grouping the equivalence pairs into equivalence classes and assigning a unique class identifier to each equivalence class. Either the minimum or the maximum label in each equivalence class can be adopted as the class identifier. Then, a second pass rescans the image so as to replace each temporary label by the identifier of its equivalence class. As a result, each pixel belonging to the same object is assigned the same unique label. A simple example shown in Figure 4.26 illustrates how the classical two-pass CCL works.

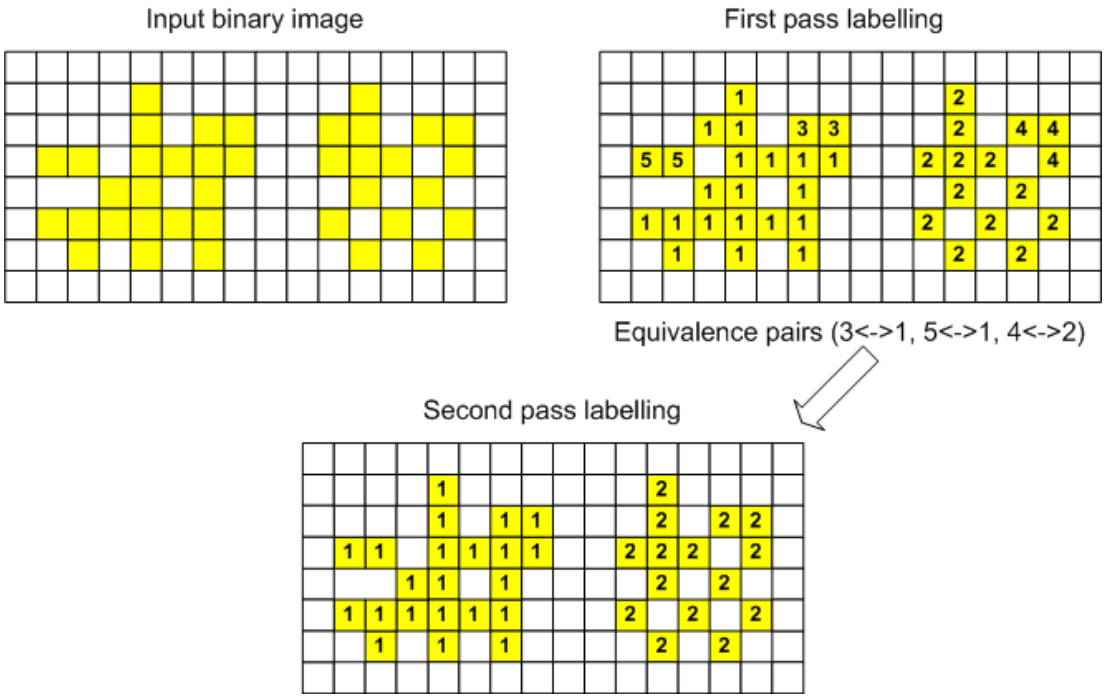


Figure 4.26.: Two-pass connected components labelling

Once the labels are determined, all objects in image can be easily distinguished. Afterwards, typically in a third pass, the features of each object are calculated according to the label information.

It is obvious that the classical algorithm is not suitable for hardware implementation.

First, the algorithm is difficult to be parallized due to its sequential nature. In the step of CCL, prior to assigning the current pixel p a label, the labels of the adjacent pixels must be provided, which implies a high degree of data dependency. As a consequence, pixels have to be processed one at a time (in serial). Since labelling one pixel may take several (say N, where $N > 1$) clock cycles, the FPGA needs to be clocked at $N\times$ the frequency of the pixel clock to process the incoming image data in real time. This, however, is unrealistic when the image data rate reaches several hundred-mega-pixel per second. Second, the algorithm requires multiple raster-scan passes over the image. Hence, a complete image frame needs to be buffered before the next pass starts. This not only demands high memory access bandwidth but also results in significant processing delay (the duration of one frame), which is unacceptable to applications where the system latency is critical. In many optical tracking systems, it is required that pixels must be processed "on-the-fly" to maintain minimum latency.

In this thesis, a high performance single-pass blob analysis is developed. The implemented algorithm is very area-effective, utilizing less than 7% logic resources of the target FPGA on PowerEye. Moreover, the algorithm requires only one raster-scan pass to calculate desired blob features. The following sections describe the proposed algorithm as well as the FPGA implementation in detail.

4.6.2. Proposed Algorithm

The developed blob analysis is a single pass algorithm which allows the progressively scanned streaming image data to be processed directly from the camera without any off-chip buffering. The algorithm consists of two main processing blocks: labelling and merging.

The labelling process in our algorithm follows the rules of the classical CCL:

- Background pixels are labeled by 0.
- If all neighboring pixels are background, then a new label is assigned to the current pixel.
- If only a single non-zero label is found among the neighbors, that label is copied to the current pixel.
- If two different non-zero labels are found among the neighbors, a merging condition occurs. The smaller label is assigned to the current pixel, and an equivalence pair is registered to indicate that these two labels belong to the same blob.

Labels assigned to each pixel are fed into the feature merging block, in which the following three geometrical features are calculated:

- Area - the total number of pixels that belong to a blob.

$$A = \sum_{(x,y)\in B} 1 \tag{4.24}$$

- Center of gravity - the equilibrium point at which the entire mass of an object is concentrated [Kol07]. The center of gravity of a blob $CoG = [x, y]^T$ can be calculated by

$$CoG = \left[\frac{\sum_{(x,y)\in B} xI(x,y)}{\sum_{(x,y)\in B} I(x,y)}, \frac{\sum_{(x,y)\in B} yI(x,y)}{\sum_{(x,y)\in B} I(x,y)} \right]^T \tag{4.25}$$

  where $I(x, y)$ represents the intensity value of pixel p located at (x, y).

- Bounding-Box - the minimum rectangle that completely surrounds a blob. Bounding-Box is commonly represented by the coordinates of two corner points: $[left, top]$ and $[right, bottom]$.

The basic idea of the proposed single pass blob analysis is to calculate the blob features while performing the labelling. This removes the need for producing a label image and thus saves the second re-labelling pass which is necessary in classical algorithms. It is important to mention that the aim of blob analysis is not assigning pixels in each blob the same unique label as the classical CCL does. In fact, the outputs of blob analysis should be properties or features of blobs rather than labels. Therefore, in our algorithm labelling is only an intermediate step in which pixels belonging to the same blob are allowed to have different labels. Using a dedicated merging scheme, blob features can still be correctly calculated.

Since pixels of the image sensor are scanned one by one from left to right and top to bottom, a blob might be considered as multiple individual or temporary blobs before it can be completely seen by the camera. Thus, the features of a blob can not be determined until all pixels belonging to the blob are available in the data stream. In order to accomplish the feature extraction in one scan pass, a feature RAM which is addressed by labels associated with each blob, is used to store the temporary blob features. Whenever a new label is generated, a new entry is created in the feature RAM. If two different labels are found to be equivalent, the features of the corresponding blobs are merged immediately, which can be achieved by accumulating the features of the involved blobs.

Let us take the calculation for the sum of the x coordinates (denoted as sum_x) as an example. Suppose two blobs that are labeled by L_i and L_j (with $L_j > L_i$) respectively join at the current pixel p. We assign p the smaller label (L_i). To get the merged sum_x, the sum_x value of both blob$[L_i]$ and blob$[L_j]$ stored in the feature RAM are read out and added together, then the result is accumulated with the x coordinate of

p. The new sum_x is written into the feature RAM at the address pointing to L_i, while the entry at the address L_j is discarded.

Selecting the smaller label as the representative works for simple blobs. When analyzing blobs with complex shapes, the above described procedure may return wrong results. An example shown in Figure 4.27 demonstrates such a scenario.

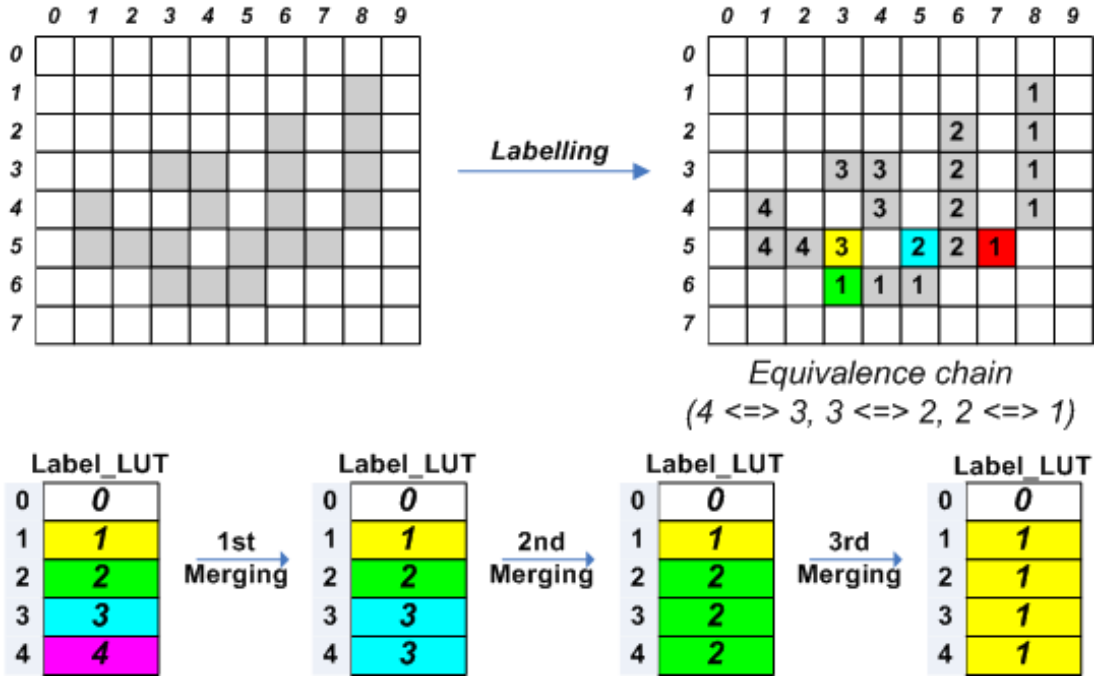


Figure 4.27.: Blob merging

The first merging condition in Figure 4.27 occurs in row 5 where the pixel is marked by yellow. To perform the feature merging, we first assign label 3 to the yellow pixel and register an equivalence pair (4 <=> 3). Then the features of blob[4] and blob[3] stored in the Feature RAM(denoted as FeatureRAM[4] and FeatureRAM[3] respectively) as well as the features of the current pixel are accumulated. Afterwards the new features are written into FeatureRAM[3]. FeatureRAM[4] is out-of-date and thus can be deleted. The same operation is performed when the blue and the red pixel are available in the pixel stream. After the completion of row 5, the features at address 1 in the feature RAM (FeatureRAM[1]) represent the current status of the blob. In addition, we get three equivalence pairs, namely (4 <=> 3), (3 <=> 2), (2 <=> 1), forming an equivalence chain. So far the algorithm works, however, this procedure can lead to an incorrect merging operation when processing the green pixel in row 6. As can be seen, this pixel has two neighbors which have been labelled previously by 4 and 3 respectively. It is important to point out that the features of the green pixel can not be merged with FeatureRAM[3], since FeatureRAM[3] has already been declared out-of-date after being merged with FeatureRAM[2]. The correct operation is to merge its features with FeatureRAM[1], in which the updated blob features are maintained.

The relationship between label 3 and label 1 can be established when we resolve the equivalence chain. It is not difficult to find out that label 1 is the root of the chain, to which all labels point. Because only the root label is associated with the updated blob, we should use the root rather than the temporary label to perform the feature merging operation. Therefore, the key point of our algorithm is to resolve the equivalence chain during the labelling process, so that the feature extraction block always receives the root labels associated with each blob.

Resolving the equivalence chain is non-trivial work. Searching for the root label can be very time consuming, especially when the chain contains significant amount of equivalence pairs. In the classical CCL, the equivalence pairs are represented as rooted trees and the root can be determined using Union-Find algorithms [DST92]. Unfortunately, such algorithms are not suited for the FPGA implementation because they all suffer from high computational complexity and require a second scan pass throughout the image.

In our implementation, this problem is addressed by means of a label look-up table (Label_LUT). The Label_LUT is a simple one-dimensional array with the size equal to the maximum label value. It is addressed by the temporary labels assigned to each blob and stores for each temporary label its corresponding root. For example, Label_LUT[i] is the root label to which label i points. At the beginning, each entry of the Label_LUT is initialized to contain a value equal to its index, i.e., Label_LUT[i] = i, where i = 0, 1, ..., max_label. This means that, initially, each possible label represents a distinct blob. When two labels, L_i and L_j, with $L_j > L_i$, are found to be equivalent, a Parallel Search and Multiple Update (PSMU) operation will be performed. More precisely, all entries of Label_LUT containing the value of Label_LUT[L_j] are updated by Label_LUT[L_i]. The C code for this process is given by:

```
for(address = 0; address < max_label; address++){
    if (Label_LUT[address] == Label_LUT[Lj])
        Label_LUT[address] = Label_LUT[Li];
}
```

PSMU is the key operation to keep the Label_LUT always in its correct, updated state. Suppose the current pixel p has two neighbors which are labelled by L_i and L_j respectively. To perform the correct merging operation, we first compare $Label\_LUT[L_i]$ with $Label\_LUT[L_j]$ to check whether the roots of L_i and L_j (denoted as R_{L_i} and R_{L_i} respectively) are different or not. For the former case, if $R_{L_i}<R_{L_j}$, we assign R_{L_i} to p and accumulate FeatureRAM[R_{L_i}] and FeatureRAM[R_{L_j}] together with the features of p. The new features are then written into FeatureRAM[R_{L_i}] and the Label_LUT is updated in the way of PSMU. For the later case, we assign R_{L_i} to p and only have to accumulate FeatureRAM[R_{L_i}] with the features of p. Since L_i and L_j point to the same root, there is no need to update the Label_LUT.

In the example shown in Figure 4.27, initially, Label_LUT is set to [0 1 2 3 4]; after finding the equivalence pair (4 <=> 3), Label_LUT is updated to [0 1 2 3 3]; then, after finding (3 <=> 2) and (2 <=> 1) Label_LUT becomes [0 1 1 1 1]. When we get to the the green pixel in row 6, we can easily find out that both temporary labels in the neighborhood (label 4 and label 3) point to the same root (label 1). Thus, by checking the Label_LUT it is now clear to assign label 1 to the green pixel and merge its features with FeatureRAM[1].

So far our algorithm can be summarized as follows:

```
Initialize_Label_LUT();
Initialize_FeatureRAM();

for x=0 to image_width, y=0 to image_height loop
  if p is a foreground pixel then
    if neighbors are all background pixels then
      Assign_Label(p, new_label);
      MergeFeature(FeatureRAM[new_label], p);
    else if neighbors contain only one foreground pixel labelled by L then
      R_L = Label_LUT[L];
      Assign_Label(p, R_L);
      MergeFeature(FeatureRAM[R_L], p);
    else if neighbors contain more than one foreground pixels then
      /* L1,L2,L3,L4 are temporary labels assigned to p1,p2,p3,p4 respectively */
      R_L1 = Label_LUT[L1];
      R_L2 = Label_LUT[L2];
      R_L3 = Label_LUT[L3];
      R_L4 = Label_LUT[L4];
      if R_L1 = R_L2 = R_L3 = R_L4 then
        Assign_Label(p, R_L1);
        MergeFeature(FeatureRAM[R_L1], p);
      else
        Label_min = min(R_L1, R_L2, R_L3, R_L4);
        Label_max = max(R_L1, R_L2, R_L3, R_L4);
        Assign_Label(p, Label_min);
        MergeFeature(FeatureRAM[Label_min], p);
        MergeFeature(FeatureRAM[Label_min], FeatureRAM[Label_max]);
        Update_Label_LUT(Label_min, Label_max);
      end if;
    end if;
  end if;
end loop;
```

4.6.3. Speed Optimization

As mentioned earlier, the algorithm of blob analysis is sequential in nature. Processing one pixel may cost multiple clock cycles. In order to handle a large amount of pixels in real time, which is required by high speed optical tracking, two strategies are used to speed up our algorithm.

Strategy 1

The first strategy makes reuse of labels during the scan, which not only saves large amounts of logic resources but also increases the processing speed. Both Label_LUT and feature RAM require one entry for each label. The worst case for the possible number of blobs in an $N \times N$ image is $N/2 \times N/2$ as illustrated in Figure 4.28.

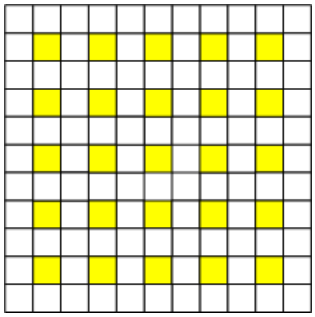


Figure 4.28.: Worst case number of blobs

Although in practice the actual number of blobs is normally less than that of the worst case, the Label_LUT and the feature RAM can still be large enough to become unfeasible for an FPGA due to the fact that a single blob may consume multiple labels. Moreover, the larger the Label_LUT, the more cycles are required for finding the root and performing the update operation. Therefore, if labels are able to be reused, significant gain in terms of both area and speed can be obtained.

The reuse of labels is not a new concept. A general algorithm for determining connected components in digital images by reusing temporary labels is presented in [DST92]. The minimum number of labels is $2N/3$ for any $N \times N$ image. In [KGH02] a more effective method which aggressively reuses the labels is described. Only $N/2 + 1$ labels are needed for an $N \times N$ image. The minimum amount of labels required by the proposed blob analysis algorithm is also $N/2 + 1$. In addition, labels are reused in a simpler way, making the algorithm more suitable for the FPGA implementation.

We borrow the term *Dead* label from [KGH02] to represent the label which can be possibly reused during the scan. There are two situations that lead to labels being declared as *Dead*:

1. A label may die an *Equivalence Death*, if it is found equivalent to some label and is not the root. In Figure 4.29, after finding the equivalence pair (3 <=> 2), we may consider label 3 as a *Dead* label because it is equivalent to label 2 and is not the root. Similarly, after finding (4 <=> 1), label 4 can be declared as a *Dead* label.

2. A label may die a *Blob Death*, if all pixels belonging to the blob have been scanned. In such a case, the root label associated with the blob is declared as a *Dead* label.

In Figure 4.29, label 1 and label 2 become *Dead* after the corresponding blobs are completely scanned.

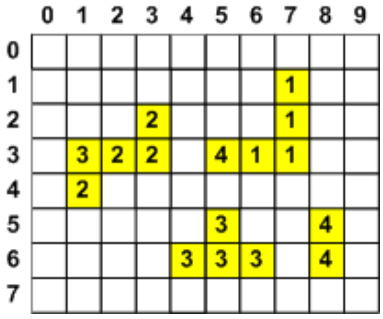


Figure 4.29.: Reuse of labels

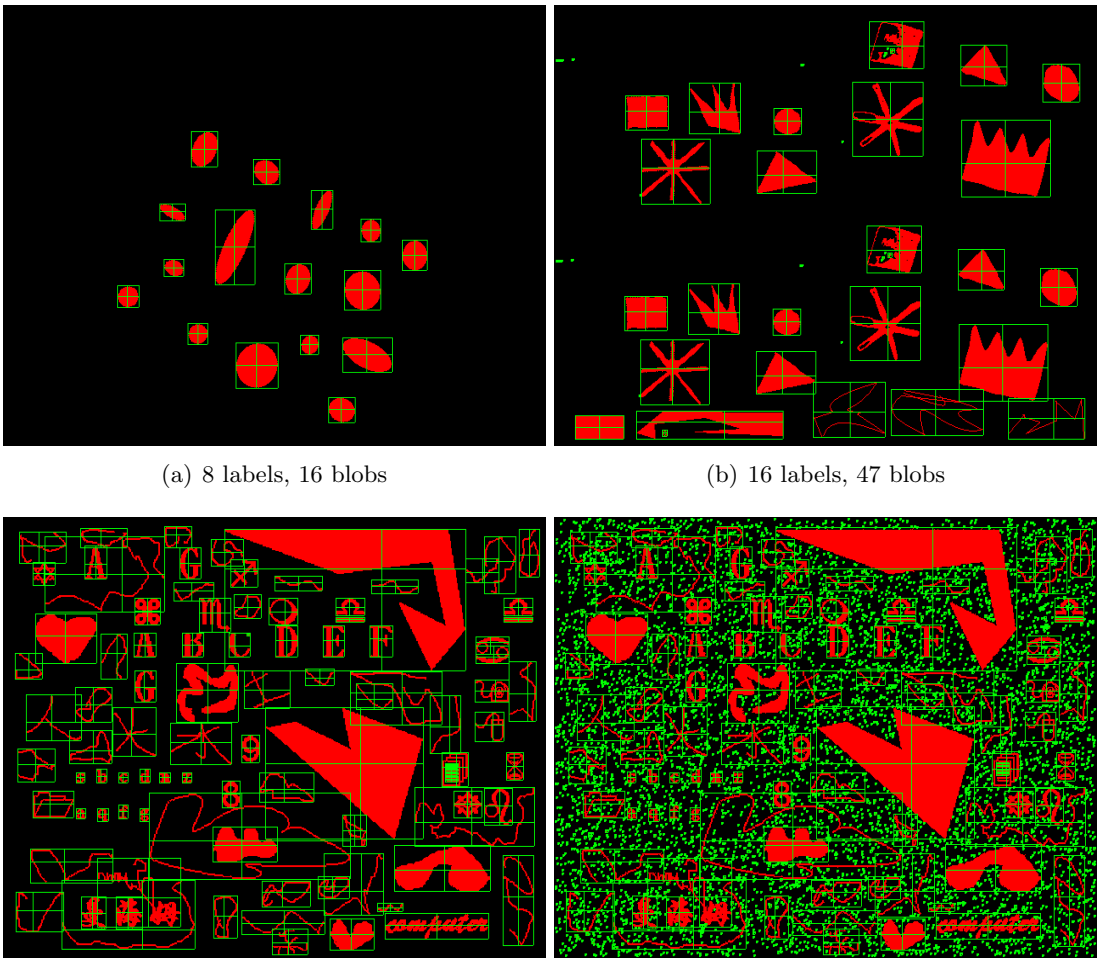
Although *Dead* labels can be reused, some constraints must be applied to avoid incorrect labelling and merging operations. In Figure 4.29, at the beginning Label_LUT is initialized as [0 1 2 3 4]. After finding the equivalence pair (3 <=> 2), we update Label_LUT to [0 1 2 2 4] and declare label 3 as a *Dead* label. If we reuse label 3 immediately in the current row, i.e., instead of assigning label 4 to the pixel at $(5, 3)$ we label it with 3, then an equivalence pair (3 <=> 1) will be generated after scanning the pixel at (6,3). As a result, Label_LUT will become [0 1 2 1 4]. Thus, when we get to the pixel at $(1, 4)$, we will have to label it 1 after checking the Label_LUT. However, label 2 is the correct label that should be assigned to this pixel.

As demonstrated by the example described above, immediate reuse of labels may cause unexpected update of Label_LUT, which can result in incorrect labelling operation for the subsequent pixels. To solve this problem, it is suggested in [KGH02] that all previous instances of a *Dead* label in the current row must be relabelled by its root prior to reusing it. However, relabelling is the operation we are trying to avoid, since it costs significant number of clock cycles. In our algorithm this problem is addressed in a different way. Suppose label L is assigned to pixel p and declared as *Dead* in row N. If we reuse L neither in row N nor in row $N + 1$, the Label_LUT[L] can be kept unchanged in row N and row $N + 1$. As a result, every pixel adjacent to p in row $N + 1$ will be labelled by its correct root label. After row $N + 1$, there will be no pixels connected to p. Thus L can be safely reused in row $N + 2$ and subsequent rows.

For the second case, since the root label is the only representative label of the blob, it will not be associated with any pixels belonging to the blob after the blob is completely scanned. Therefore, it is safe to reuse a label immediately after it dies a *Blob Death*. In the proposed algorithm, we use the x coordinate of the last pixel of a blob to determine the blob completion. To describe this process more precisely, we need to define two special pixel masks, namely *leave_blob* and *finish_blob*. *leave_blob* is a pixel mask in which the current pixel p is a background pixel and the neighbor pixel $p1$ is a foreground pixel. *Finish_blob* defines a pixel mask with the following criteria: the current pixel p

and its neighbors $p1$ and $p3$ are all background pixels, meanwhile $p2$ is a foreground pixel. If a *leave_blob* mask is encountered, the x coordinate of p is written into an array named *last_x* at the address pointing to the root label of $p1$, or $last\_x[R_{p1}]$. Whenever a pixel mask that satisfies the criteria of *finish_blob* is found, we first read the root label of $p2$ from the Label_LUT and then compare the x coordinate of the current pixel p with $last\_x[R_{p2}]$. If $x = last\_x[R_{p2}]$, we claim that the blob is completed. R_{p2} can then be reused immediately. And the features stored in FeatureRAM$[R_{p2}]$ can be sent to the host, e.g., a PC, for further analysis.

Clearly the data structure used to store the labels is a stack. During the scan, if a label is found to be *Dead*, it is pushed into the stack for the future reuse. Whenever a new label is required, we pop one label from the stack and check its validity before using it. Reusing labels enables detecting a large number of blobs with small amount of labels. As can be seen in Figure 4.30, it is possible to calculate features of more than 5,000 blobs in an image with only 64 labels.



(a) 8 labels, 16 blobs

(b) 16 labels, 47 blobs

(c) 32 labels, 96 blobs

(d) 64 labels, 5759 blobs

Figure 4.30.: Performance of reusing labels

Strategy 2

The second strategy comes out of the realization that the algorithm of blob analysis processes mostly foreground pixels. For most applications, background pixels contribute the dominate portion of the image. Thus, if background pixels that require no processing are filtered out, significant reduction of the overall execution time can be achieved.

Furthermore, it is possible to optimize the processing of foreground pixels by exploiting the fact that the neighbors in a pixel mask are not independent [WOS09]. For instance, $p3$ is a neighbor of p, $p1$, $p2$ and $p4$, which implies that all other foreground pixels in the mask must share the same root label with $p3$. Therefore, if $p3$ is a foreground pixel, it is the only pixel that needs to be checked to process the the current pixel p. As a result, the number of neighbors to be examined is reduced from four to one.

The 8-connectivity pixel mask shown in Figure 4.25 consists of 5 binary pixels, which gives a total of 32 cases. In order to dig out the optimization potential of every pixel mask, we enumerate all the 32 cases, and group the required operations into 10 categories, as illustrated in Figure 4.31.

Pixel Mask	Category	Pixel Mask	Category	Pixel Mask	Category	Pixel Mask	Category
	new_blob		copy_p1		NOP		leave_blob
	copy_p4		comp_p1p4		NOP		leave_blob
	copy_p3		copy_p3		NOP		leave_blob
	copy_p3		copy_p1		NOP		leave_blob
	copy_p2		copy_p1		finish_blob		leave_blob
	comp_p2p4		copy_p1		finish_blob		leave_blob
	copy_p2		comp_p1p4		NOP		leave_blob
	copy_p2		copy_p1		NOP		leave_blob

Figure 4.31.: Pixel Mask Categorization

The operations needed by each category are summarized as follows:

- **new_blob** - pop a new label from the label stack, assign it to p, create a new entry in blob feature RAM.

- **copy_p1** - get the temporary label of $p1$ (L_{p1})[1], assign it to p, merge the features of p with $FeatureRAM[L_{p1}]$.
- **copy_p2** - get the root label of $p2$ (R_{p2}) by checking Label_LUT, assign it to p, merge the features of p with $FeatureRAM[R_{p2}]$.
- **copy_p3** - get the root label of $p3$ (R_{p3}) by checking Label_LUT, assign it to p, merge the features of p with $FeatureRAM[R_{p3}]$.
- **copy_p4** - get the root label of $p4$ (R_{p4}) by checking Label_LUT, assign it to p, merge the features of p with $FeatureRAM[R_{p4}]$.
- **comp_p1p4** - get the temporary label of $p1$ (L_{p1}) and the root label of $p4$ (R_{p4}), compare L_{p1} with R_{p4}. If $L_{p1} = R_{p4}$, assign L_{p1} to p, merge the features of p with $FeatureRAM[L_{p1}]$; if $L_{p1} < R_{p4}$, assign L_{p1} to p, merge the features of p with $FeatureRAM[L_{p1}]$ and $FeatureRAM[R_{p4}]$, push R_{p4} into the label stack, update Label_LUT; if $L_{p1} > R_{p1}$, assign R_{p4} to p, merge the features of p with $FeatureRAM[L_{p1}]$ and $FeatureRAM[R_{p4}]$, push L_{p1} into the label stack, update Label_LUT.
- **comp_p2p4** - get the root label of $p2$ (R_{p2}) and the root label of $p4$ (R_{p4}), compare R_{p2} with R_{p4}. If $R_{p2} = R_{p4}$, assign R_{p2} to p, merge the features of p with $FeatureRAM[R_{p2}]$; if $R_{p2} < R_{p4}$, assign R_{p2} to p, merge the features of p with $FeatureRAM[R_{p2}]$ and $FeatureRAM[R_{p4}]$, push R_{p4} into the label stack, update Label_LUT; if $R_{p2} > R_{p4}$, assign R_{p4} to p, merge the features of p with $FeatureRAM[R_{p2}]$ and $FeatureRAM[R_{p4}]$, push R_{p2} into the label stack, update Label_LUT.
- **leave_blob** - get the temporary label of $p1$ (L_{p1}), write the x coordinate of p into the $last\_x[L_{p1}]$.
- **finish_blob** - get the root label of $p2$ (R_{p2}), if $last\_x[R_{p2}] = x$, declare the completion of $FeatureRAM[R_{p2}]$, push R_{p2} into the label stack.
- **NOP** - No operation.

As can be seen, the maximum number of pixels to be examined is 2, which is required by pixel masks that fall into the **comp_p1p4** and **comp_p2p4** categories. In all other cases, it is only necessary to check one pixel in the neighborhood.

Combing the above described two optimization strategies together, a significant improvement in terms of processing speed can be achieved. Performance analysis shows

[1]Since $p1$ is the last pixel that has been labelled, the label assigned to $p1$ must be the root label. Therefore, there is no need to check the Label_LUT to get R_{p1}.

that based on an FPGA implementation, the proposed algorithm is able to satisfy the throughput requirements of our high speed optical tracking system.

4.6.4. Hardware Implementation

Figure 4.32 illustrates the high level block diagram of the FPGA implementation for the above described algorithm.

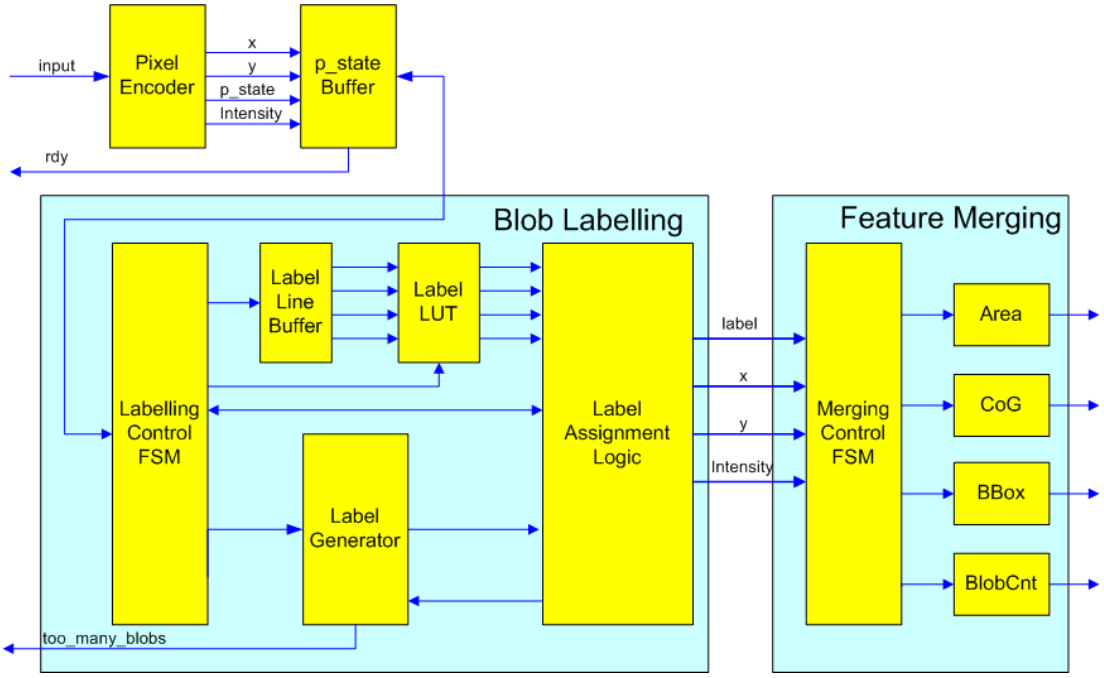


Figure 4.32.: Block Diagram of FPGA based blob analysis

Pixel Encoder

The pixel encoder module performs the pixel mask categorization. Each category is given a unique 4-bit code named *p_state* as shown in Figure 4.33.

Category	p_state	Category	p_state	Category	p_state	Category	p_state	Category	p_state
new_blob	1000	copy_p1	1001	copy_p2	1010	copy_p3	1011	copy_p4	1100
Category	p_state	Category	p_state	Category	p_state	Category	p_state	Category	p_state
comp_p1p4	1101	comp_p2p4	1110	finish_blob	0001	leave_blob	0010	NOP	0000

Figure 4.33.: Pixel state encoding

It is obvious that each bit of *p_state* is simply a function of input pixels. Using a Karnaugh-Map, the calculation of *p_state* can be expressed as:

$$p\_state(0) = \bar{p_1} \bullet p_2 + p \bullet \bar{p_2} \bullet p_3 + \bar{p} \bullet \bar{p_1} \bullet p_2 \bullet \bar{p_3} \tag{4.26a}$$

$$p\_state(1) = p \bullet \bar{p_1} \bullet p_2 + p_3 + \bar{p} \bullet p1 \tag{4.26b}$$

$$p\_state(2) = p \bullet \bar{p_3} \bullet p4 \tag{4.26c}$$

$$p\_state(3) = p \tag{4.26d}$$

As illustrated in Figure 4.33, if all bits of *p_state* are zero, the corresponding pixel mask must belong to the NOP category. In this case, the *p_state* will not be sent to the subsequent processing modules. As a result, a large amount of input pixels are filtered out. The pixel encoder also synchronizes the x and y coordinates as well as the pixel intensity with each valid *p_state*, since they are necessary for calculating the blob features, such as center of gravity and bounding-box.

p_state Buffer

As mentioned previously, there is a high degree of data dependency between pixels in blob analysis. Thus, pixels can not be streamed smoothly into the processing pipeline at every clock cycle. The current pixel must wait until the processing of the previous pixel has been completed. The *p_state* buffer is used to provide a flow control mechanism. It is composed of an asynchronous FIFO with the depth of 4096. The results from the pixel encoder are first written into the buffer before getting processed. Whenever the processing pipeline is free, a valid *p_state* together with the x and y coordinates are read out and then fed into the processing pipeline. If the number of data stored in the buffer reaches a predefined threshold, the *rdy* signal is deasserted to inform the previous module that no input pixel can be accepted at the moment.

Blob Labelling

The Blob Labelling block performs the labelling process in blob analysis. It consists of five main sub-modules, namely the control state machine (Control_FSM), the Label Line Buffer, the Label_LUT, the Label Generator and the Label Assignment Logic as shown in Figure 4.32.

Control_FSM The Control_FSM is responsible for control of the whole labelling process. It starts with decoding the *p_state* and decides which operation to be performed. When the current pixel is completely processed, it issues a read command and fetches the next *p_state* from the buffer.

Label Line Buffer The Label Line Buffer functions as a delay element that synchronizes the labels assigned to $p2$, $p3$, $p4$ in the previous line with the label assigned to $p1$ in the current line. It ensures that the labels of all neighborhood pixels are ready

for use when the current pixel is being processed. The label line buffer is realized by a RAM with the depth equal to the number of pixels in an image line.

Label_LUT The Label_LUT memorizes the root for each temporary label and performs the PSMU operation introduced in 4.6.2. In this thesis, two different implementations of Label_LUT are evaluated.

The first implementation is based on a RAM architecture which executes the PSMU in serial. The primary advantage is that the RAM based Label_LUT is very area-effective. However, three cycles are necessary to accomplish the read, compare and write operations. As a result, it takes $3N$ clock cycles for each PSMU, where N represents the maximum number of labels used in the design. If there is a large amount of PSMU to be executed during the scan, this implementation can become very inefficient.

The second implementation is intended to perform the PSMU operation in a single clock cycle. It features a parallel architecture as shown in Figure 4.34.

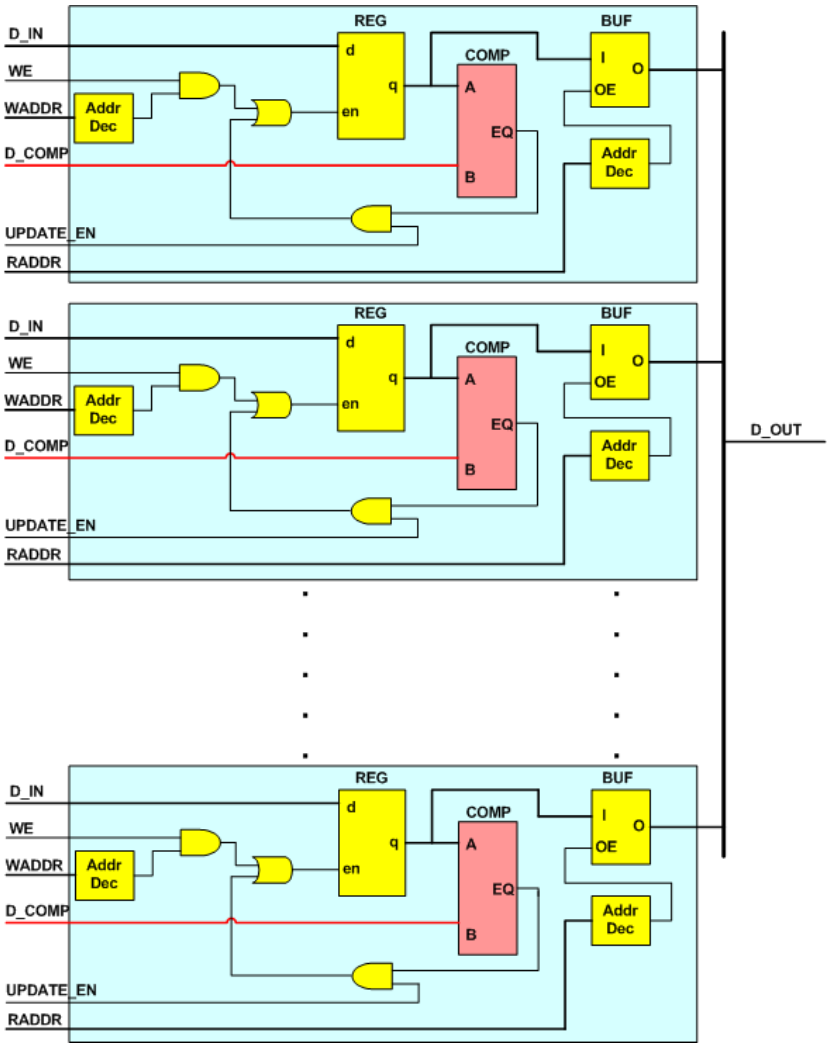


Figure 4.34.: parallel implementation of Label_LUT

The parallel Label_LUT is made up by a number of identical memory cells, each of

which contains a register, two address decoders and a output buffer. In comparison with conventional RAMs, there is additionally a comparator in each memory cell that is responsible for comparing the data stored in the register and the data driven by the D_COMP bus. If a match is found, registers containing the value of D_COMP will be updated by the data presented on the the D_IN bus. In this way, the PSMU operation can be executed in one clock cycle.

The main drawback of the parallel Label_LUT is its large area cost. As illustrated in Figure 4.35, the amount of required FPGA LUTs and flip-flops increases dramatically as the number of labels increases.

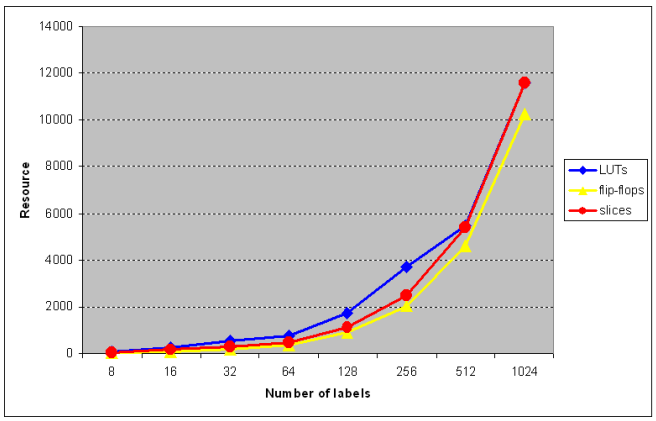


Figure 4.35.: Parallel Label_LUT resource utilization

The concept of reusing labels enables the usage of a parallel Label_LUT with a reasonable resource cost and satisfactory performance. In our FPGA design, the number of labels is set to 64, which allows to build a parallel Label_LUT with only 767 FPGA LUTs and 390 flip-flops. As demonstrated in Figure 4.30, 64 labels should be enough for most optical tracking applications.

Label Assignment The Label Assignment module has two main tasks:

1. select the correct label among the neighbors, assign it to the current pixel p,
2. determine the value of D_COMP and D_IN to update the Label_LUT.

As summarized in Figure 4.31, the possible labels that can be assigned to the current pixel are:

- the temporary label of $p1$ (L_{p1}),
- the root label of $p2$ (R_{p2}),
- the root label of $p3$ (R_{p3}),
- the root label of $p4$ (R_{p4}),
- the smaller label between L_{p1} and R_{p4},

- the smaller label between R_{p2} and R_{p4},
- the new label popped from the label generator.

Similarly, labels to be updated in the Label_LUT can be selected from one of the following two options:

- the larger label between L_{p1} and R_{p4},
- the larger label between R_{p2} and R_{p4}.

Figure 4.36 shows the simplified circuit architecture of the Label Assignment module. A 7-way multiplexer decides which label should be output as the label of p. Two comparators together with two multiplexers select the labels that should be passed to the D_COMP bus and the D_IN bus of the Label_LUT respectively. Moreover, an $UPDATE\_EN$ signal determines whether it is necessary to update the Label_LUT.

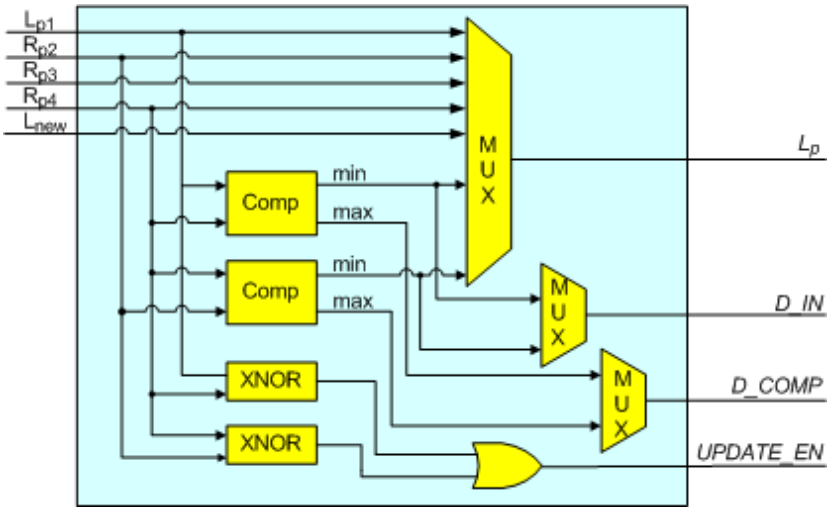


Figure 4.36.: Structure of the Label Assignment module

Label Generator the Label Generator is formed by a synchronous FIFO with the size of 64. If a new_blob operation is decoded, the next available label is popped from the FIFO. Whenever a label is found to be *Dead*, this label is pushed into the FIFO for future reuse. Since labels can not be reused arbitrarily, it is necessary to check the validity of a label before using it. As explained in Section 4.6.3, if a label dies an *Equivalence Death* in row N, it can only be reused in row M, where $M > N + 1$. In our FPGA design, when a label is declared *Dead*, the current row number is recorded into a RAM at the address pointing to this label. When a label is popped from the Label Generator, we compare the pre-recorded row number attached to this label with the current scan row number to check if it can be legally reused. For the case where a label dies a *Blob Death*, we write the maximum line number of the image into the RAM, since this kind of label can be reused immediately after being declared *Dead*.

It is obvious that the proposed algorithm can not support the detection of an infinite number of blobs, even labels can be reused. A warning must be given in case the image contains too many blobs. In our FPGA design, this functionality is implemented by asserting the *too_many_blobs* signal if one of the following two conditions is satisfied:

1. there is no label available in the Label Generator, when a new_blob operation is decoded,
2. a *Dead* label is illegally reused.

Feature Merging

The Feature Merging module takes the label information, the x and y coordinates as well as the pixel intensity from the labelling block as input to calculate the blob features. As each pixel is assigned a label, the feature RAM is updated to reflect the current state of each blob. When two blobs are merged, the update of the feature RAM requires a read of the existing features of both blobs followed by a write of the update features. This enables a Dual-Port RAM to be used for the feature data storage. Figure 4.37 shows the circuit structure for calculation of the blob center of gravity. Other features, such as area and bounding-box can be implemented in a similar manner.

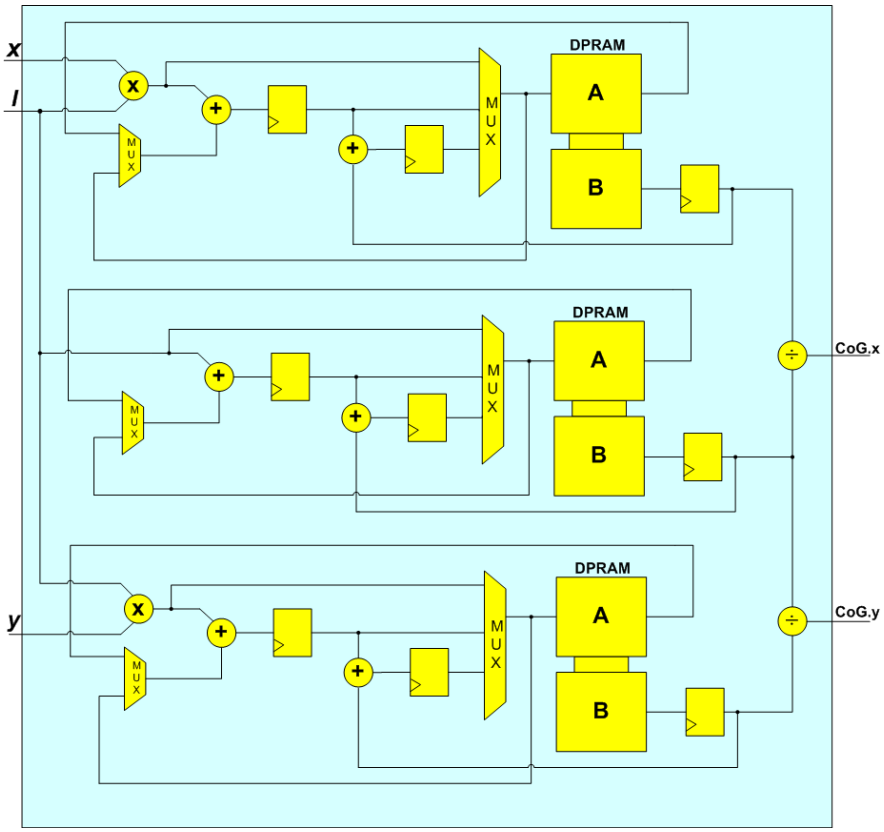


Figure 4.37.: Circuit structure for the blob CoG calculation

FPGA Resource Utilization

The logic resource utilization for the above described implementation is shown in Table 4.1. The target FPGA is the XC4VFX60 device used on PowerEye. As can be seen, the implemented design consumes a very small portion of the available FPGA hardware resources.

	Used	**Available**	**Utilization**
Number of Slices	1653	25280	6.6%
Number of Block RAMs	12	232	5.2%
Number of DSP Blocks	2	128	1.6%

Table 4.1.: Blob analysis FPGA resource utilization

4.6.5. Performance Estimation

As discussed in Section 4.6.3, the proposed algorithm groups pixels to be processed into categories that require different processing time. For instance, labelling the pixel that falls into the **comp_p1p4** category contains the following operations: fetch the temporary label of $p4$, check Label_LUT to get the root label of $p4$, compare it with the label of $p1$, push the *Dead* label into the label stack and update Label_LUT. Since the last two operations can be executed simultaneously, a total of 4 clock cycles are needed. In comparison, labelling a pixel belonging to the **copy_p1** category requires only 2 clock cycles. As a result, time needed to process an image depends highly on the image content.

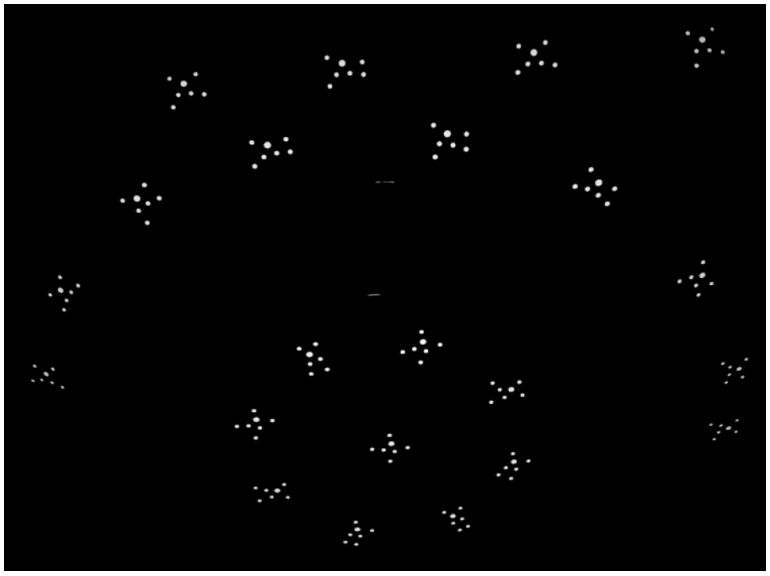
The processing performance in terms of speed can be evaluated by the number of frames that can be processed per second. To estimate the frame rate, we need to count the number of required clock cycles for every image, which is given by:

$$N = \sum_i N_{p_i} C_i \tag{4.27}$$

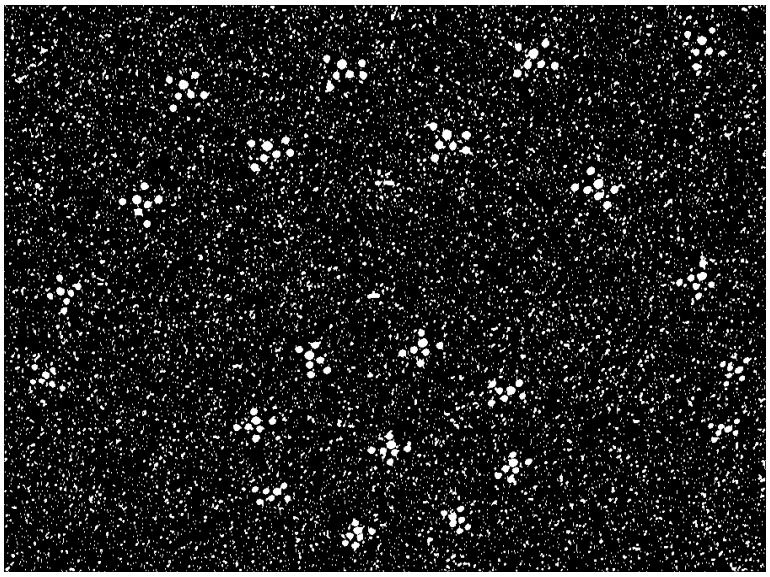
where N_p is the number of pixels that belong to the same category and C represents the clock cycles needed for processing such a pixel. Once N is obtained, we can easily calculate the processing frame rate by dividing the number of FPGA operating frequency by N. Let us take the two images in Figure 4.38 as an example.

The 1.3 Mega pixel (1280×1024) sized image shown in Figure 4.38(a) was captured by the *WheelWatch* photogrammetry system that measures the wheel position of a driving car in real time [WBM04]. A total of 154 infrared markers are attached to the vehicle wheel which are seen as white blobs. We consider this image as the typical case scenario in optical tracking.

The algorithm calculates the area, the centre of gravity and the bounding box for each blob. According to the pixel statistics listed in Table 4.2, only a very small portion



(a) typical case



(b) worst case

Figure 4.38.: Typical and worst case scenario of blob analysis

of pixels need to be processed. And most of them belong to the simplest category (**copy_p1**). As a result, 15,496 clock cycles are required for processing the entire image. Since implemented FPGA design has a maximum operating clock frequency of 150MHz, a processing frame rate of 9,679fps can be obtained.

In Figure 4.38(b), the same image used for the typical case scenario is strongly corrupted by salt noise. After the image binarization, there will be a large number of small blobs cased by the noise pixels. We use this image to simulate the worst case scenario. The proposed algorithm detects 16,786 blobs at the cost of 143,308 clock cycles per image. When being clocked at 150MHz, the FPGA is able to process 1,046

	typical case		extreme case	
category	Number of pixels	Number of required cycles	Number of pixels	Number of required cycles
new_blob	194	2	16921	2
copy_p1	4706	2	5242	2
copy_p2	197	3	742	3
copy_p3	423	3	950	3
copy_p4	190	3	704	3
comp_p1p4	37	4	89	4
comp_p2p4	0	4	20	4
leave_blob	1004	2	19309	2
finish_blob	370	3	17580	3
total	7121	15496	61557	143308

Table 4.2.: Pixel statistic for the typical and worst case scenario of blob analysis

such images per second. It is clear that the throughput requirement of our high-speed camera can still be satisfied.

4.7. Summary

High-speed optical tracking systems require excessive processing power due to huge amount of pixel data yielded by multiple high-speed cameras. This chapter describes the FPGA implementation for a number of image processing algorithms which are frequently used in optical tracking applications. One can easily exploit the parallelism for some image processing operators, such as color segmentation, noise reduction, edge detection and morphological filtering. Using an FPGA, a processing throughput of several thousand Mega pixels per second can be obtained. A more complicated case studied in this chapter is the blob analysis, in which a high degree of data dependency between pixels can not be avoided. A new high performance, one pass blob analysis algorithm is presented. With a highly optimized FPGA implementation, it is possible to process several thousand images per second depending upon the application.

5. System Integration and Evaluation

This chapter presents a prototype design that proves the usability of the proposed optical tracking system. First, the hardware setup is briefly introduced. Then two FPGA designs that focus on operating the camera system and performing the calculation of 2D object positions at high frame rate are described in detail. Finally, the system performance in terms of accuracy, speed and latency is evaluated.

5.1. Hardware Setup

Figure 5.1 illustrates the hardware setup of the prototype system.

Figure 5.1.: Hardware setup of the prototype system

Due to the lack of image sensors, the prototype system contains two cameras that are connected to a PowerEye board via the CLinkRx-TripleBase adapter. Each camera is equipped with a monochrome MT9M413 sensor and an infrared illuminator. Configured in the CameraLink Base mode, the camera outputs 205 images per second at the resolution of 1.1 Mega pixel (1080×1024). The tracking targets are spherical, retro-reflective markers that reflect infrared light emitted by the illuminators. The goal is to calculate the 2D position of each marker in real time. A laptop which accesses PowerEye via Ethernet is used for the purpose of system control and data visualization. The data flow throughout the entire system can be briefly described as follows:

i) Each image sensor captures images under the control of the camera FPGA.

ii) The camera FPGA receives pixel data from the sensor, packs the data into CameraLink streams and sends them to the CLinkTx board.

iii) The CLinkTx board converts the single-ended LVCMOS signals from the FPGA to LVDS signals that are transmitted over a CameraLink cable.

iv) The CLinkRx-TripleBase board receives two LVDS data streams from the cameras simultaneously, converts them back into single-ended LVCMOS signals that are fed to the FPGA on PowerEye.

v) The PowerEye FPGA performs all necessary operations for calculating the 2D marker positions and sends the results to PC via Gigabit Ethernet.

vi) A software program running on the PC visualizes the image data as well as the tracking data to verify the correctness of the results delivered by the hardware.

The camera FPGA and the PowerEye FPGA play an essential role in the prototype system. The following sections detail the designs for the two FPGAs respectively.

5.2. Camera FPGA Design

The camera FPGA is responsible for the control of the whole camera system, which mainly includes guiding the image sensor through the full sequence of its operation and packing the high-speed data from the sensor into pixel streams that comply with the CameraLink standard. It also manages low-speed communication between sensor and PowerEye. Figure 5.2 depicts the high level block diagram of the camera FPGA design.

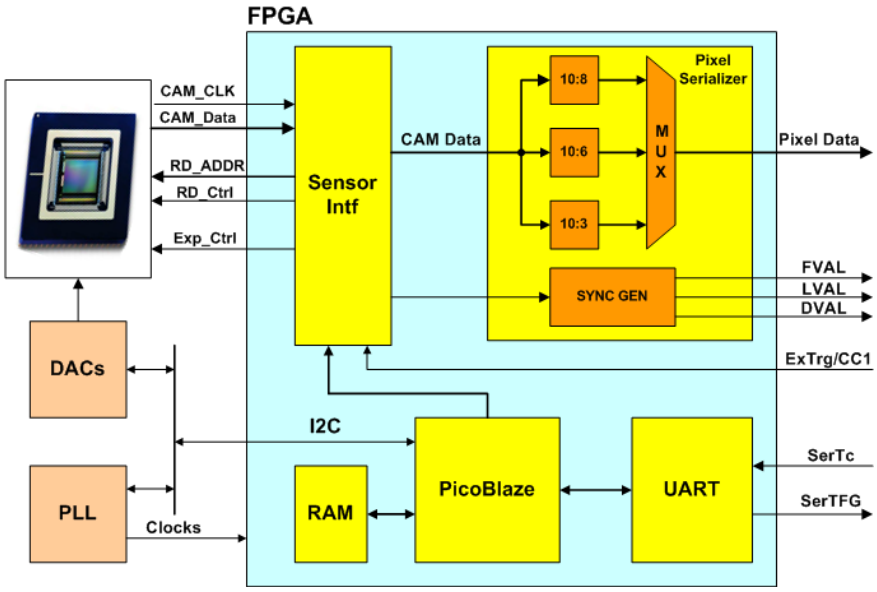


Figure 5.2.: Block diagram of the camera fpga design

5.2.1. Sensor Interface

The Sensor Interface module illustrated in Figure 5.2 generates all required signals to operate the MT9M413 image sensor. Two main control procedures, the sensor exposure and the pixel read-out, are implemented in this module.

Exposure Control

The MT9M413 sensor integrates the *TrueSNAP* electronic shutter which allows simultaneous exposure of the entire pixel array. The exposure procedure is controlled by two signals: PG_N and TX_N, as shown in Figure 5.3. Clearing pixels and starting new photo-signal integration are enabled by making PG_N low. Forcing TX_N low allows to transfer the data into pixel memory. The exposure time is determined by the interval between the falling edges of PG_N and TX_N [Mic06].

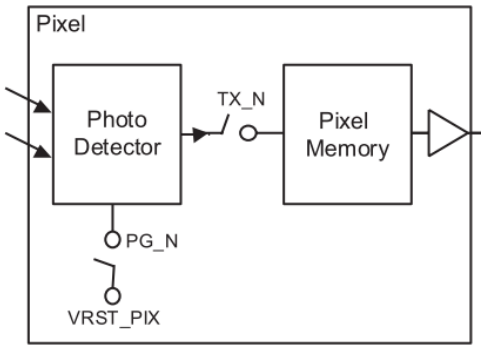


Figure 5.3.: MT9M413 exposure control [Mic06]

The camera FPGA supports the following two exposure modes:

- Free-run mode - The FPGA generates an internal signal to continuously trigger the exposure for each frame at a programmed frame rate.
- External trigger mode - The sensor exposure is controlled by an externally generated trigger (ExTrg) signal. This mode is frequently used in multi-camera systems for the purpose of camera synchronization. In the prototype system, the PowerEye FPGA broadcasts an ExTrg signal to all cameras over CameraLink. Each camera starts the exposure at the positive edge of ExTrg and ends when the falling edge of ExTrg arrives. As shown in Figure 5.2, the camera control pin CC1 is selected to transfer the ExTrg, since the CC1 signals for all cameras were routed with matched length throughout the entire system.

Pixel read-out Control

Pixel read-out involves the process of selecting the pixel rows to be read, storing the digitized data in the ADC register and transferring the data to the output register.

The MT9M413 sensor can be configured in two different read-out mode: sequential and simultaneous. In the sequential mode, a frame is read out after the exposure. In the simultaneous mode, the current frame can be exposed while the previous frame is being read out. The camera FPGA was designed to support both sequential and simultaneous read-out.

The read-out control also implements the ROI (region of interest) function for applications where the full image resolution (1280×1024) is not needed. A ROI refers to a rectangular window that is specified by its position and size within the full frame. In the FPGA design, generating a ROI is realized by two counters - one for horizontal and the other for vertical. These counters are controlled by four registers which set the start and stop positions separately in both directions. This allows the definition of a ROI at any location with the size ranging from one pixel to the entire sensor.

5.2.2. Pixel Serializer

As mentioned in Section 3.2.1, the MT9M413 sensor integrates 10-bit ADCs to perform the analog-to-digital conversion and clocks out 10 pixels in parallel. Inside the FPGA, the least two significant bits of each pixel are dropped, which results in a total of 80 bits pixel data per clock cycle. These pixels need to be serialized to comply with the transmission protocol defined by the CameraLink standard. As illustrated in Figure 5.2, three different serializers together with a 3:1 multiplexer are integrated in the Pixel Serializer module so that all the three possible configurations of CameraLink can be supported. For example, when the camera is configured in CameraLink BASE mode, the 10:3 serializer is activated, as CameraLink BASE defines three pixel data ports. Similarly, the 10:6 and the 10:8 serializer can be selected for the Medium mode and Full mode respectively. In the prototype system, cameras are configured in the BASE mode, since they must be connected to the CLinkRx_TripleBase board which only supports the CameraLink BASE configuration.

The SYNC_GEN block in the Pixel Serializer generates the FVAL (frame valid), LVAL (line valid) and DVAL (data valid) signals to mask valid image data, as specified in the CameraLink standard. Figure 5.4 illustrates the pixel data timing diagram under the simultaneous read-out mode.

5.2.3. UART Interface

In addition to the signals for pixel data and camera control, the CameraLink standard also defines a RS232 compatible asynchronous serial interface for changing camera operating mode and parameters. It can also be used to query the camera about its current status. This interface is enabled by the UART module shown in Figure 5.2. The implemented UART interface supports half-duplex communication with a standard 11-bit frame format - one start bit, eight data bits, one parity bit and one stop bit.

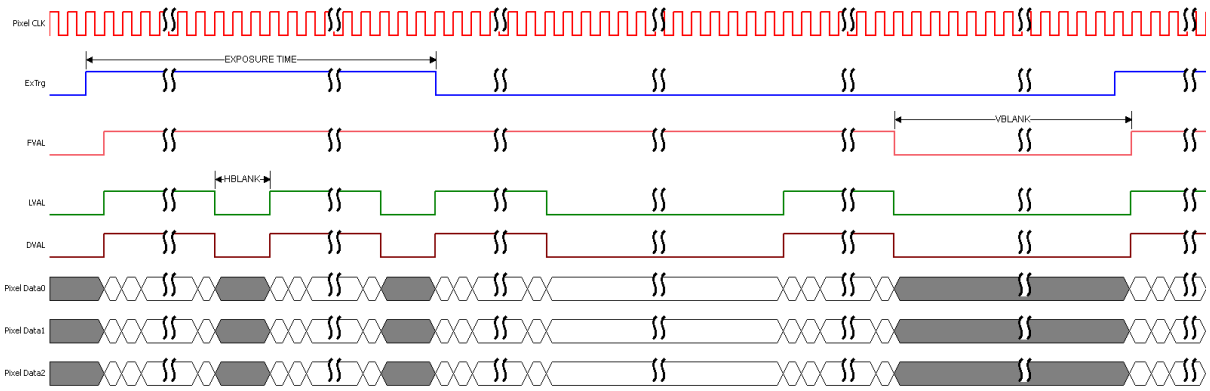


Figure 5.4.: Pixel data timing diagram simultaneous read-out mode

5.2.4. PicoBlaze Processor

The camera FPGA integrates an 8-bit PicoBlaze micro processor, which handles all low-speed operations. The PicoBlaze processor consists of a Havard CPU core, a 1K×16-bit instruction memory and a 1K×8-bit data memory. One of its main tasks is to implement an I2C master interface to initialize the on-board PLL and DACs.

The PLL can be programmed to provide clock sources with desired frequency that are used for driving the image sensor, FPGA internal logic and the ChannelLink transceivers on the CLinkTx board. Figure 5.5 illustrates the clock distribution structure of the camera system.

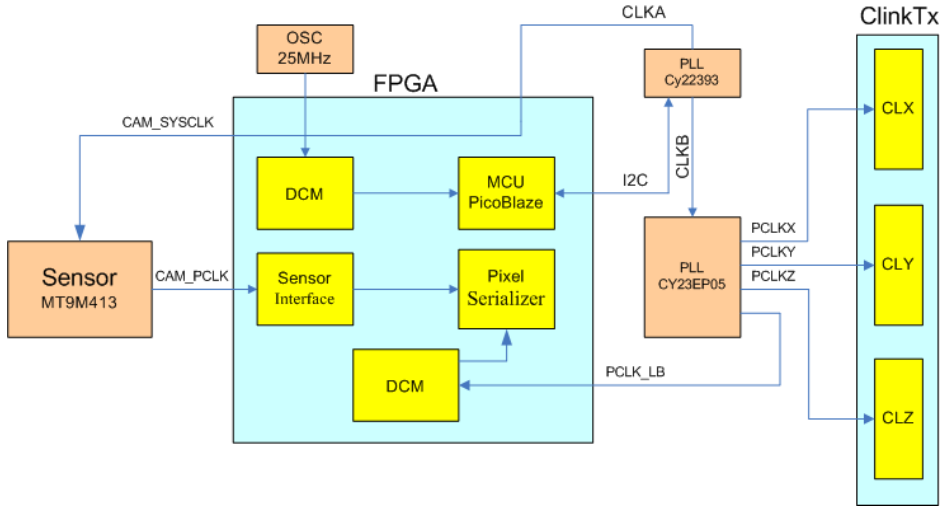


Figure 5.5.: Clock distribution of the camera system

The DACs must be configured via the I2C bus to generate proper reference voltages for the sensor gain and offset adjustment as well as the fixed pattern noise correction.

During the run-time, all communication data traveling to and from the UART interface, e.g., ROI and exposure time configuration, inquiry request of camera status,

etc., are collected and analyzed by the PicoBlaze processor and forwarded to the expected destinations afterwards. The advantage of using a PicoBlaze processor is that it enables implementing complex control operations in software and thus increases the system flexibility.

5.3. PowerEye FPGA Design

The main task of the PowerEye FPGA is to analyze pixel data from the cameras and perform all the necessary operations to calculate the 2D maker positions. Moreover, it also handles off-chip memory management, data communication with PC over Ethernet and camera synchronization. The high-level FPGA design block diagram is illustrated in Figure 5.6. Despite the fact that only two cameras are used in the prototype system, three independent pixel processing pipelines are integrated in the FPGA to provide the capability of processing pixel streams from three cameras simultaneously.

5.3.1. Video Input Controller

The Video Input Controller is where the entire image acquisition process takes place. The inputs to this module are three CameraLink video streams, each of which contains three parallel 8-bit pixel data ports as shown in Figure 5.6. In order to improve the processing parallelism and the memory bandwidth utilization, a 3:4 pixel deserializer is applied to each camera which packs every four pixels into a 32-bit word. Asynchronous FIFOs are utilized in the pixel deserializer to synchronize signals between camera clock domain and FPGA internal clock domain.

5.3.2. Pixel Processing Pipeline

The Pixel Processing Pipeline performs all necessary operations to calculate the 2D marker positions in each captured image. To achieve high processing throughput, all window-based modules are designed to process parallelly 4 pixels every clock cycle, using the technique described in Section 4.5.1. The entire pipeline consists of six processing stages: Gaussian Smoothing, Sobel Edge Detection, Region Filling, Morphological Filtering, Blob Analysis and Blob Classification.

Gaussian Smoothing

The incoming image from the Video Input Controller, denoted as I, is first smoothed by a Gaussian filter for the purpose of noise suppression. This process is implemented by convolving I with a 5×5 Gaussian smoothing kernel shown in Figure 4.10.

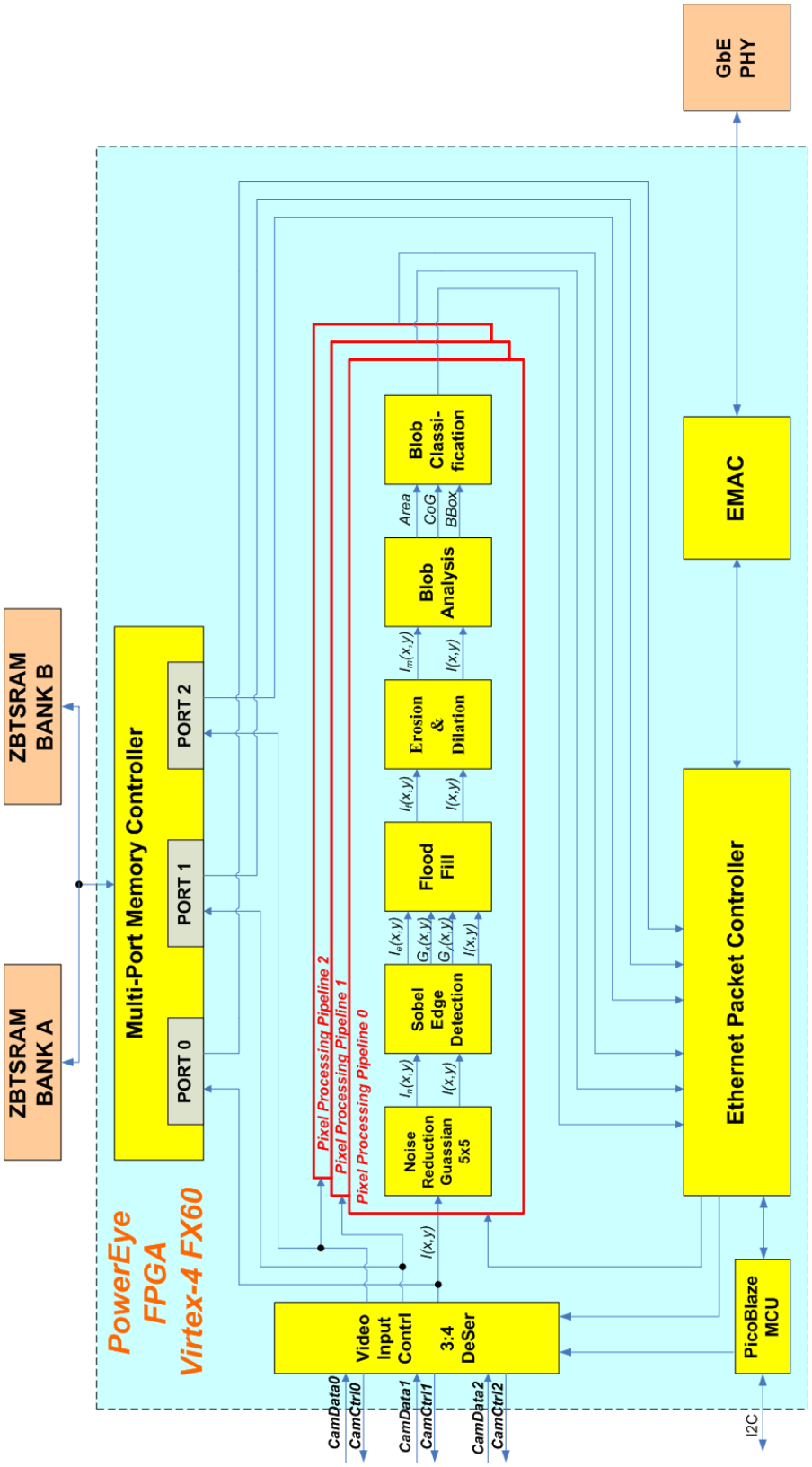


Figure 5.6.: Block diagram of the PowerEye FPGA design

Sobel Edge Detection

The second stage of the Pixel Processing Pipeline is an edge detection process. The goal is to extract the contour of each marker. The Sobel edge detector that not only determines the edge pixels but also calculates the edge direction is chosen. As described in Section 4.3, the Sobel edge detector utilizes two 3×3 kernels which are convolved with the original image to calculate approximation of the derivatives - one for horizontal (G_x) and the other for vertical (G_y). In the implemented design, two optimized Sobel convolution kernels given by Equation 5.1 are used since they produce much less error of local direction than the kernels defined in Equation 4.14 [JGH99].

$$G_x = \begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix} * I \qquad G_y = \begin{bmatrix} 3 & 10 & 3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix} * I \tag{5.1}$$

The edge strength at each point can be determined by the sum of the absolute values of the derivatives in both directions, as presented in Equation 4.15.

The output of this module is an edge image (I_e), where pixels close to the marker border have a high intensity value. As a result, marker contours can be easily extracted by binarizing I_e with a given threshold. The threshold must be carefully chosen so that the contour extracted for each marker defines a closed region. Moreover, the sign of G_x provides the information to distinguish the left and right side of the contour. For a pixel on the contour of a marker, its G_x is negative on the left side and positive on the right side.

Hole Filling

Once the edge detection is complete, we have a binary image where each pixel is marked as either an edge pixel or a non-edge pixel. Markers in the binary image are seen as hollow circles, which are filled by the Region Filling module. The region filling approach presented in [WBM04] is adopted in the implemented design since the algorithm can be easily parallized. The criterion that determines whether a pixel should be filled or not is described as follows:

Suppose pixel p located at (x, y) is the current pixel under scan. $p1$ and $p2$ represent pixels located at $(x-1, y)$ and $(x, y-1)$ respectively. Then p is filled if it is an edge pixel, or both the $p1$ and $p2$ are filled and the G_x of $p1$ is not a positive number.

Morphological Filtering

The next step of the pixel processing pipeline is a morphological filter. The objective is to remove small foreground regions which can be caused by noise. An erosion filter and a dilation filter are integrated in the module. The order for which filter is applied first is configurable. By changing the order of erosion/dilation, both opening and closing

operations are achievable. In order to provide more processing flexibility, both erosion and dilation filters are designed to support bypassing the input pixels directly to the output without them being processed.

Blob Analysis

The Blob Analysis module labels all foreground pixel regions and calculates their 2D geometrical features, including center of gravity, area and bounding-box. The algorithm and the FPGA implementation have been detailed in Section 4.6.

Blob Classification

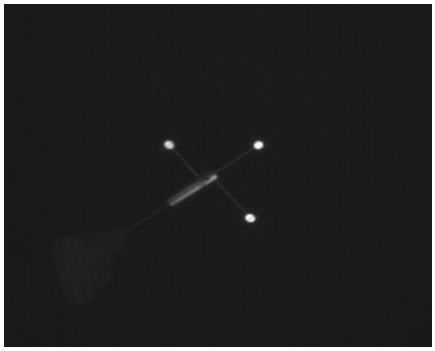
The Blob Classification module refines the tracking result according to the information of blob area and bounding-box. By checking the area, blobs that are too small or too large are filtered out. Moreover, based on the assumption that all markers are circular in shape a simple roundness check is performed. If the width of the bounding box is close to its height, we probably have a circle rather than long rectangular blob. Additionally, the blob extent is calculated as a double-check. The extent of a blob is defined as the area of this blob divided by the area of its bounding-box. Given the circular form, the extent value of a marker should be between 0.8 and 0.9. Thus, any blob with an extent out of this range is eliminated.

Figure 5.7 demonstrates an example, which shows the resulting images produced by the Pixel Processing Pipeline.

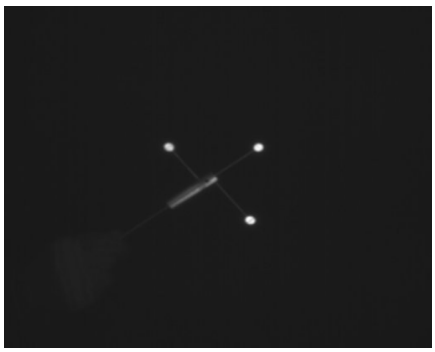
5.3.3. Multi-Port Memory Controller

Memory management is an important aspect in the PowerEye FPGA design. In order to verify the tracking results during the run-time, the FPGA must provide the possibility to transmit the original or intermediate result image data from one or multiple high-speed cameras to PC. In the prototype system, Gigabit Ethernet is the only available communication path between PowerEye and PC, which has a physical bandwidth limitation of 1Gbps. Since the pixel data rate is much higher than 1Gbps, a complete image frame needs to be buffered. Storing image data produced by multiple high-speed cameras in real time requires not only large storage capacitance but also high memory bandwidth. This makes the off-chip ZBTSRAMs on PowerEye an ideal choice for use as a frame buffer.

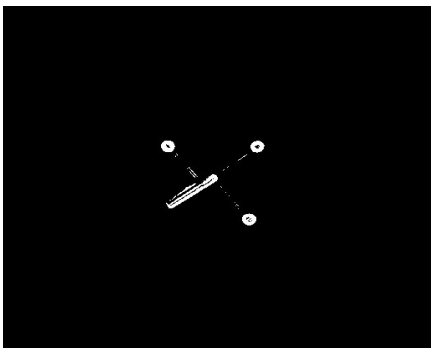
As mentioned in Section 3.5.2, PowerEye features two independent ZBTSRAM banks, allowing to construct a ”ping-pong” architecture. More precisely, the memory bank that is being read from and written to the ZBTSRAM can alternate every frame. This avoids concurrent reads and writes on the same chip and thus improves the image data transmission efficiency.



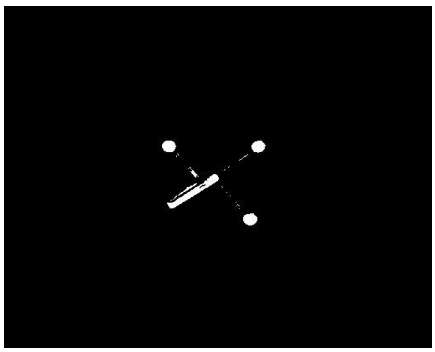
(a) Original image



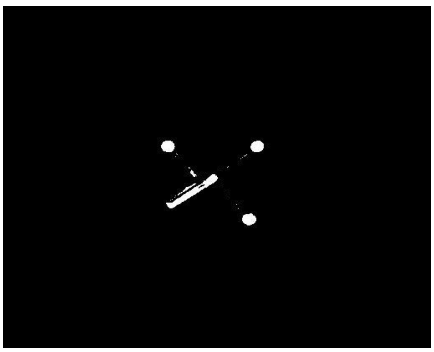
(b) Gaussian Smoothing



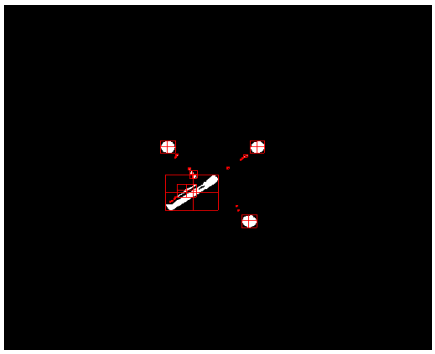
(c) Sobel Edge Detection



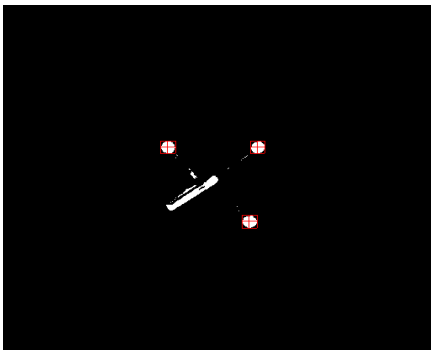
(d) Hole Filling



(e) Morphological Filtering



(f) Blob Analysis



(g) Blob Classification

Figure 5.7.: Intermediate results produced by the Pixel Processing Pipeline

One great challenge for managing the image data storage in PowerEye is how to handle concurrent pixel streams from multiple cameras. As mentioned in Section 5.3.1, the Video Input Controller packs every 4 pixels into a 32-bit word for each camera, yielding a total of 96 bits data per clock cycle. However, one ZBTSRAM bank has only a 36-bit wide data bus. In the PowerEye FPGA design, this problem is addressed by a Multi-Port Memory Controller (MPMC).

The implemented MPMC is a triple-port memory controller which can arbitrate up to three concurrent pixel streams. Each memory port has a 36-bit wide data interface. Either a read or a write access can be served at a time, since the address control is shared between read and write. A read buffer and a write buffer with the depth of 32 are integrated in each memory port so that the read and write requests can be buffered and get processed only when the memory bus is available. The arbiter assigns time slices to each port for accessing the ZBTSRAM and arbitrates all three ports in a round-robin fashion. Figure 5.8 illustrates the block diagram of MPMC.

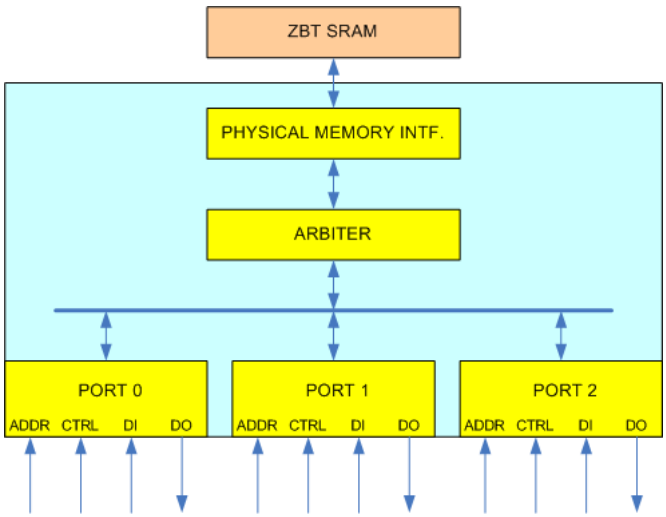


Figure 5.8.: Block diagram of the MPMC

In the prototype system, all CameraLink pixel streams have an operating clock frequency of 80MHz. After the 3:4 pixel deserialization performed by the Video Input Controller, the pixel clock is degraded to 60MHz. In order to write three pixel streams concurrently into the memory without any data lost, the ZBTSRAM must be clocked at least at $3 \times 60MHz = 180MHz$. An important consideration for designing a high speed memory interface is minimizing the clock skew, which is caused by delay in the clock path. Clock skew potentially reduces the overall design performance by increasing setup times and lengthening clock-to-output delays, both of which increase the clock cycle period [Xil06]. The solution presented in [Bap00] is adopted in the PowerEye FPGA design. The key point is to route the clock signals driving the both ZBTSRAM banks back to the FPGA and use them as feedback inputs to the Digital Clock Managers (DCMs). Inside the FPGA, three DCMs are employed to minimize the clock

skew as shown in Figure 5.9: one to de-skew and generate a 2× controller clock and two to de-skew and generate a board-level 2× clock. As a result, both the controller and the ZBTSRAMs are driven by de-skewed clocks. Based on this architecture, the physical interface of the MPMC can operate at 186MHz, which satisfies the clock rate requirement.

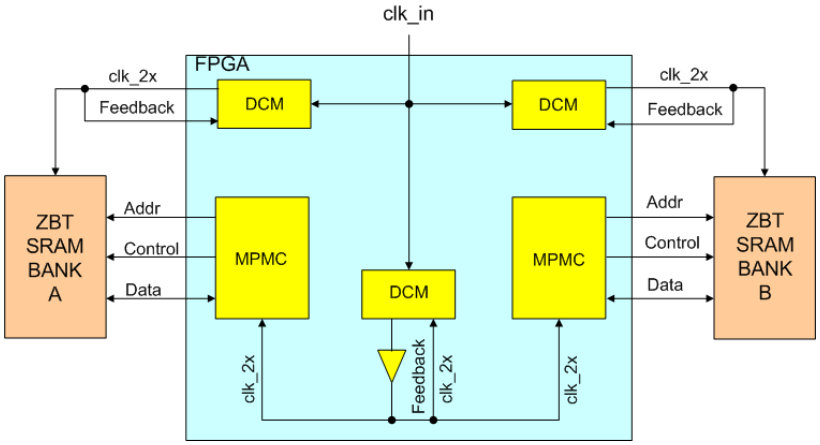


Figure 5.9.: Clock de-skew scheme of MPMC

5.3.4. Ethernet Packet Controller

As mentioned in Section 3.5.7, the PowerEye FPGA contains embedded Gigabit EMAC blocks. Together with the on-board PHY, a Gigabit Ethernet connection to an external device, e.g., a network, a PC, or another PowerEye board, can be established. Although the EMAC greatly simplifies the design for data communication over Ethernet, extra logic is still required for creating packets that comply with the network transmission protocol. This functionality is implemented by the Ethernet Packet Controller module. In addition, this module also handles camera synchronization for applications where a large number of cameras and PowerEye boards are used in the tracking environment. The high level block diagram is shown in Figure 5.10.

As can be seen, there are three main types of packets traveling over the Ethernet Packet Controller:

1. video packet, which carries the raw image data reading from the MPMC and the tracking results produced by the Pixel Processing Pipelines.

2. configuration packet, which contains the configuration information for the complete system.

3. PTP (Precise Timing Protocol) packet, which is dedicated for time synchronization between different network nodes.

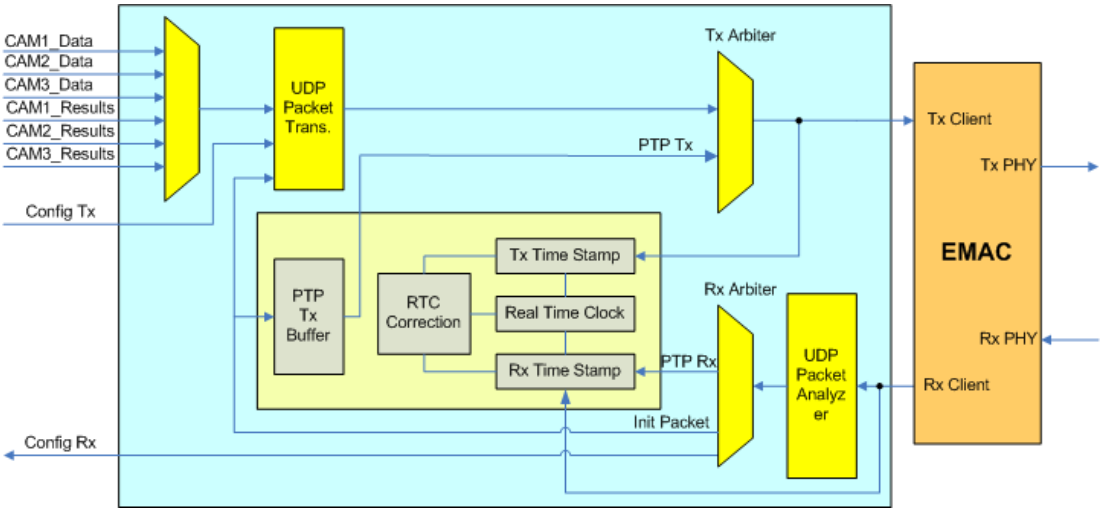


Figure 5.10.: Block diagram of the Ethernet Packet Controller

UDP Packet Transmitter

The Packet Transmitter is responsible for organizing the video data and the configuration data in standard Ethernet packets, so that they can be accepted by another Ethernet device.

Considering that high network bandwidth utilization and low transmission latency are critical for the overall system performance, the User Datagram Protocol (UDP) [Pos80] was chosen as the network communication protocol. UDP is a transport layer protocol defined in the well-known Open Systems Interconnection (OSI) model [Zim80].

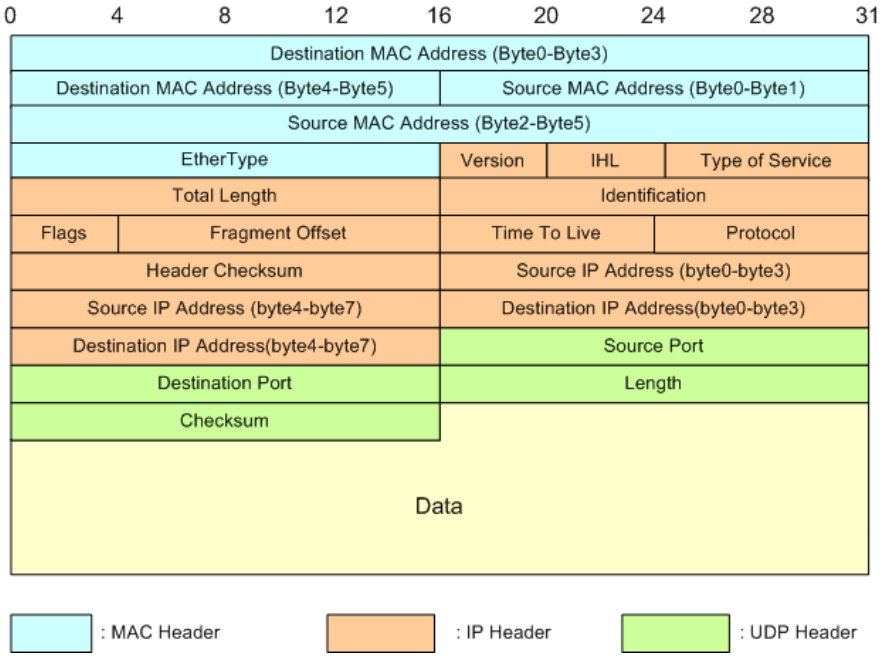


Figure 5.11.: UDP packet format

As shown in Figure 5.11 , a UDP packet consists of four data segments: the 14-byte MAC header, the 20-byte IP header, the 8-byte UDP header and the payload data to be transmitted. All data fields of the headers can be considered as static, except for the following parts:

- *Total Length* and *Header Checksum* belonging to the IP header,
- *Length* and *Checksum* belonging to the UDP header.

The calculation for UDP checksum is optional, which means it can be set to zero without the packet being rejected by the receiver [ABS10] [Pos80]. The computation of IP header checksum is defined as the 16-bit one's complement of the one's complement sum of all 16-bit words in the header [Int81]. If we exclude the *Total Length*, all fields in the IP header section are constant. Therefore, the IP header checksum can be seen as a function of *Total Length*. To simplify the FPGA design, the length of the payload data for both video packet and configuration packet is set to be equal to the width of the image. Then the *Total Length* of the IP header and the *Length* of the UDP header can simply be calculated as :

$$Total\_Length = 20(IP\_header) + 8(UDP\_header) + Image\_Width \tag{5.2a}$$

$$Length = 8(UDP\_header) + Image\_Width \tag{5.2b}$$

Since the resolution of the camera will rarely be changed after initialization, we can pre-calculate the IP header checksum by software. As a result, the entire header required for generating the video and configuration packets can be stored in a header RAM. After power-on, we send a special *init* packet from PC to PowerEye. The *init* packet contains basic information used for initializing the head of UDP packets carrying video and configuration data. This includes pre-calculated IP header length, IP head checksum, UDP length and all other static data fields such as MAC addresses, IP addresses, UDP port numbers, etc.

When the payload data is ready to be sent, the constant header fields stored in the header RAM are read out and transmitted first. Since the transmission of the header takes only 44 clock cycles, a small amount of payload data need to be buffered. The transmission of the payload data can start immediately after all header bytes are sent. This removes the need of buffering the entire payload data for calculating the length and checksum, and thus significantly reduces the transmission latency.

In order to measure the transmission throughput when sending data from FPGA to PC, the following experiment was performed. The FPGA was programmed to generate a large number of UDP packets which are continuously transfered to PC. The packets contain fixed header stored in the header RAM. A serial number is attached to the

data field of each packet so that packet loss can be detected. On the PC side, a C++ application based on the open source *WinPcap* library [WP11] was developed to receive the packets. Packet loss and corruption are monitored by checking the serial number and the IP header checksum. According to the measurement results shown in Figure 5.12, when sending packets with the size greater than 1024 bytes, a substantial data transfer (without any packet loss or corruption) rate of 887Mbps can be achieved. This allows to transmit approximately 110 images per second from PowerEye to PC at the resolution of 1024×1024.

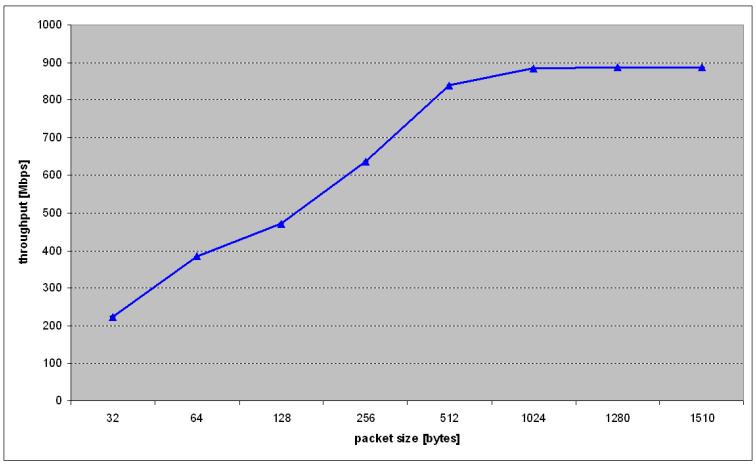


Figure 5.12.: Results of the Gigabit Ethernet transmission throughput measurement

The transmission latency over Gigabit Ethernet was measured by performing a round-trip test. The FPGA sends a UDP packet to PC. Meanwhile, a timer inside the FPGA starts. The PC acknowledges back as soon as the packet from the FPGA is received. The timer stops at the moment when the acknowledge arrives in the FPGA and the round-trip time (RTT) is recorded. The one-way latency can be estimated by dividing the RTT in half. Figure 5.13 illustrates the results of the latency measurement.

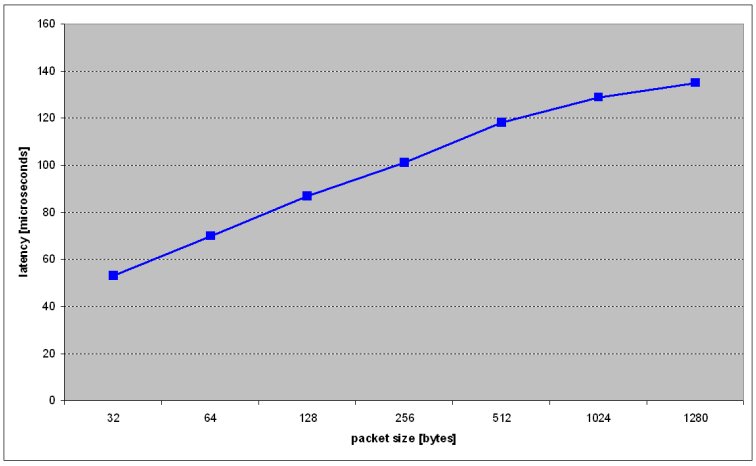


Figure 5.13.: Results of the Gigabit Ethernet transmission latency measurement

Precise Time Synchronization

- **The problem of camera synchronization**

As already mentioned in Section 3.6.3, camera synchronization is an important issue in a multi-camera based optical tracking system. Images delivered by various cameras must be captured at the same moment to ensure correct tracking results.

In the prototype system, synchronization of two cameras connected to one PowerEye board is easy to achieve. Dedicated hardware architecture enables that the PowerEye FPGA can send a common external trigger signal (ExTrg) to the cameras, which starts the exposure and pixel readout processes for both image sensors simultaneously.

In some more complicated cases, where a large number of cameras and PowerEye boards are used, the problem now becomes how to synchronize the ExTrg signals from different PowerEye boards with each other. ExTrg is activated by the PowerEye FPGA immediately after power-on. There are two main reasons that cause the ExTrg signals generated by various PowerEye boards to be asynchronous. First, the PowerEye boards can not be powered up at the same time. All PowerEye boards in the tracking environment are powered independently. In practice, it is difficult to activate the power for all PowerEye boards at exactly the same moment. The second reason for the synchronization problem is that each PowerEye FPGA has a unique oscillator with an inherent error in frequency, which means that each oscillator has its own output frequency near the nominal value but not exactly equals the nominal value. The tolerance is often measured in PPM (parts per million). Suppose we have two clock oscillators running at 100MHz with a frequency tolerance of $\pm$100PPM (the typical value for commercial crystal oscillators), in 10 seconds one clock can run 2ms faster or slower than the other for the worst case scenario. Moreover, the output frequency of a crystal oscillator can drift due to change of environmental conditions, such as temperature. Therefore, even if we can power up all PowerEye boards simultaneously, the ExTrg signals will not keep synchronized since the temporal drifting caused by clock oscillators can be accumulated over time.

- **Precision Time Protocol**

Considering that Ethernet is the only way to interconnect a large number of PowerEye boards (see Figure 3.29), the IEEE1588 standardized Precision Clock Synchronization Protocol (also known as Precision Time Protocol or PTP) [LE02] is adopted to solve the problem of camera synchronization. PTP provides a mechanism to allow network devices exchanging their time information. By precisely time-stamping special packets as they leave and arrive at the network nodes, PTP can measure and compensate offset, delay and clock drifting among different devices to provide a common time base at sub-microsecond precision to all nodes on the network. For our application, if all PowerEye boards in the tracking environment are running on the basis of the same time reference,

it is easy to create synchronized ExTrg signals to achieve strict camera synchronization. The following describes how the precise time synchronization works in PTP.

In a PTP network, one device is selected to provide the grand-master clock. The master device usually has the "best" or most accurate clock, to which the local clock of other (slave) devices must be synchronized. The master broadcasts its time to all slave nodes at a predefined interval (also known as Sync Interval). The slave nodes collect the time information from the master and adjust their local clock accordingly. It is important to mention that the Sync Interval must be chosen to avoid significant overheads on the bus traffic. Commonly used values for the Sync Interval are 1, 2, 8, 16 and 64 seconds.

In PTP, four messages are defined to manage the time information traveling between master and slave: *Sync*, *Followup*, *DelayReq*, and *DelayResp*. Figure 5.14 and Figure 5.15 illustrate how the offset and delay between master and slave are calculated and compensated.

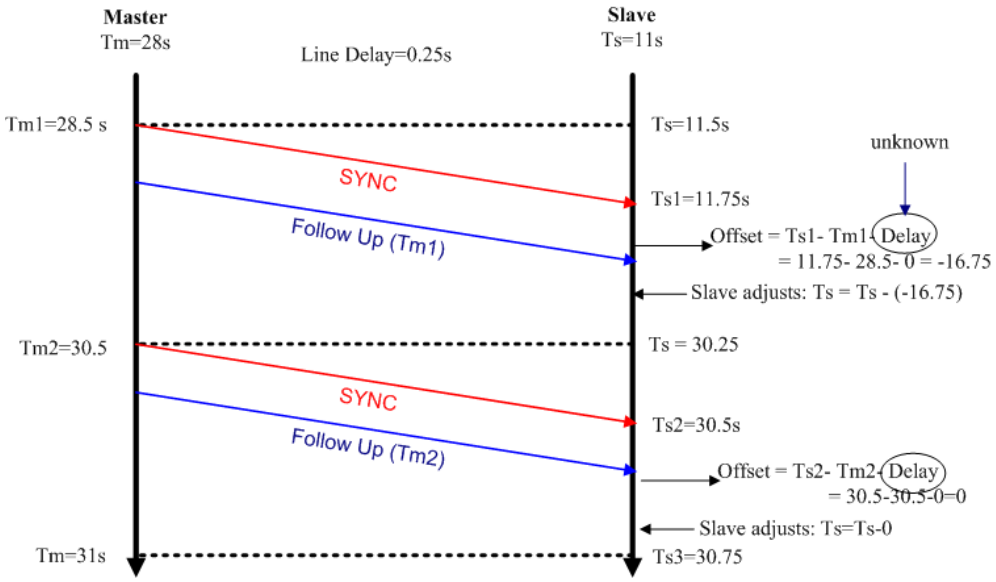


Figure 5.14.: PTP offset measurement

In Figure 5.14, the master inserts its local system time into a *Sync* message, then sends the message out. A *Followup* message containing the time when the *Sync* message actually leaves the master (T_{m1}) is sent later to the slave. On the slave side, the time when the Sync message arrives (T_{s1}) is recorded. After receiving the *Followup* message, the slave may calculate the offset with respect to the master under the assumption that there is no delay on the communication path. Since then, the slave can synchronize itself to the master with a constant error due to the unknown transmission delay.

The correction for the transmission delay works as follows. The slave sends a *DelayReq* message to the master and time-stamps the moment when the message actually leaves (T_{s3}). The master records the time when the *DelayReq* message arrives (T_{m3}),

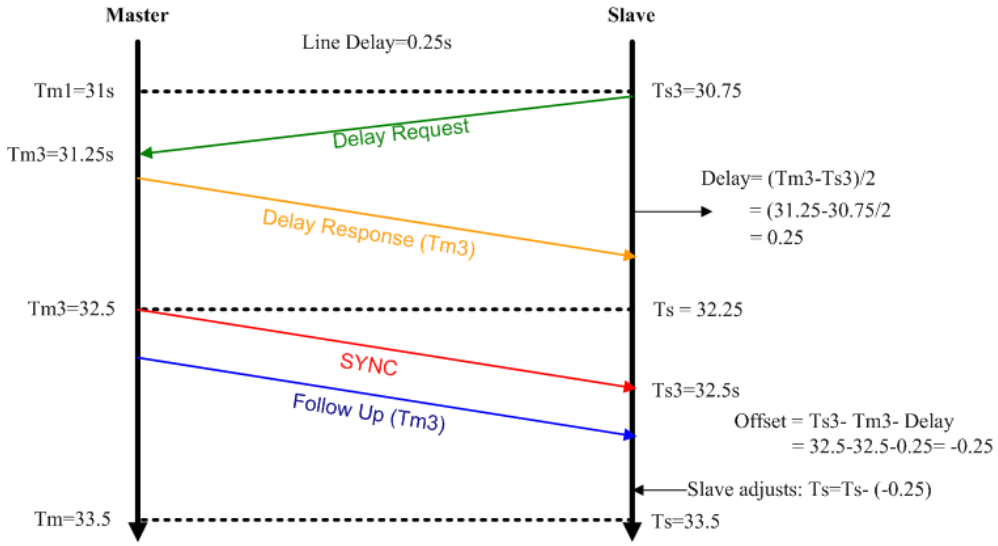


Figure 5.15.: PTP delay measurement

puts it into the *DelayResp* message and sends it back to the slave. Based on the assumption that the communication path is symmetric, the transmission delay can be calculated as shown in Figure 5.15.

As mentioned previously, in order to avoid occupying significant Ethernet bandwidth, the corrections for offset and delay are performed at a relatively long interval, for instance, once a second. Therefore, to achieve sub-microsecond accuracy and maintain linear time, it is also necessary to estimate and compensate the clock drifting between slave and master so that the slave clock runs at the same speed of the master clock.

- **PTP in FPGA**

The key components used to implement the PTP based time synchronization are shown in Figure 5.10, which mainly include the Real Time Clock (RTC) logic, the Tx and Rx Time Stamping logic, the RTC Correction module and the PTP Tx Buffer.

The RTC is formed by a 48-bit seconds field counter and a 32-bit nanoseconds field counter. When the nanoseconds field counter reaches 1×10^9 (1 second), it resets back to zero and the seconds field counter increments by 1. In the implemented FPGA design, both the master RTC and the slave RTC are clocked at 125MHz, since this is a readily available clock source used to drive the transmission logic of the Gigabit Ethernet MAC. Thus, by default the nanoseconds counter increments by 8 on every rising edge of the 125 MHz clock. During the operation, the slave clock frequency must be fine tuned to match the frequency of the master clock. For this reason, the RTC nanoseconds counter is extended by a 16-bit sub-nanoseconds counter, so that the nominal 8 ns increment step can be adjusted at a high degree of accuracy ($1/2^{16}$ ns). The fine-tuned RTC increment step is a 20-bit integer value calculated by the RTC correction module. The upper 4 bits overlaps with the lower 4 bits of the RTC nanoseconds counter and the

lower 16 bits align with the 16-bit sub-nanoseconds counter, as shown in Figure 5.16.

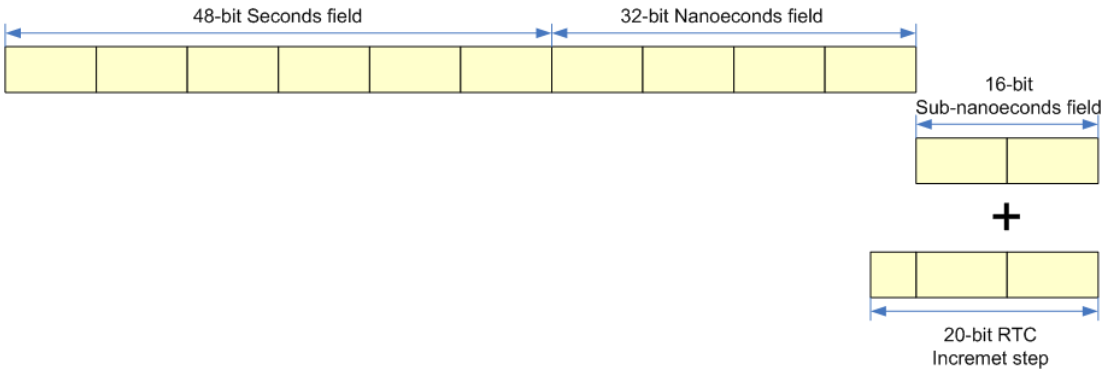


Figure 5.16.: Real Time Clock (RTC)

Time stamping plays an essential role in PTP. Whenever a PTP packet is transmitted or received, the current value of the RTC is sampled. The final time synchronization accuracy directly depends on the time stamp accuracy. Clearly, the most accurate method is to time stamp all PTP messages at the physical layer. This is however not feasible for the FPGA, since the FPGA has no direct access to the physical layer signals. As illustrated in Figure 5.10, in the PowerEye FPGA design, PTP time stamps are taken on the client interface of the Embedded Ethernet MAC. Since the Ethernet MAC has a known fixed latency, high accuracy can still be achieved.

The RTC Correction module functions only when the PowerEye board is identified as a slave device in the network. It is responsible for calculating the offset and delay values to update the slave RTC using the time stamps extracted from the PTP packets. Furthermore, this module also determines the value of the RTC increment step to compensate the clock drifting with respect to the master. The adjustment of the increment step takes place at the frequency of the *Sync* message. Suppose at time T_1 the master sends a *Sync* message to the slave. A counter on the slave is cleared when the *Sync* message arrives and starts counting on every rising edge of the local clock immediately afterwards. At time T_2 the master sends another *Sync* message. The slave counter stops counting and latches the current count value (clk_cnt) when the second *Sync* message is received. If both T_1 and T_2 are normalized in nanoseconds, the increment step of the slave RTC can be estimated by $(T_2 - T_1)/clk\_cnt$. The result is represented by a 20-bit value to maintain high accuracy.

According to the IEEE1588 standard, PTP is an application layer protocol built over UDP. The PTP Tx buffer stores the constant data fields of all necessary PTP massages, including MAC addresses, IP addresses, UDP ports, ect. Whenever a PTP message needs to be sent, it is only necessary to update a few bytes of the complete frame, such as message type, message length, time stamp. The UDP Packet Analyzer shown in Figure 5.10 monitors all received packets from the Ethernet MAC. If the packet is identified to be a PTP message, the integrated time stamp is extracted for further

analysis.

Figure 5.17 shows a setup used to evaluate the performance of the implemented PTP. Two PowerEye boards are interconnected via a Gigabit Ethernet switch. The PowerEye board on the top serves as a PTP master, while the other functions as a slave. In order to test the difference between the slave RTC and the master RTC as application realistically as possible, the FPGAs on both PowerEye boards output a Pulse Per Second (PPS) signal which is connected to an oscilloscope. Measurements show that the time deviation between master and slave was limited in ±300 ns (max. jitter), which is good enough for the purpose of camera synchronization.

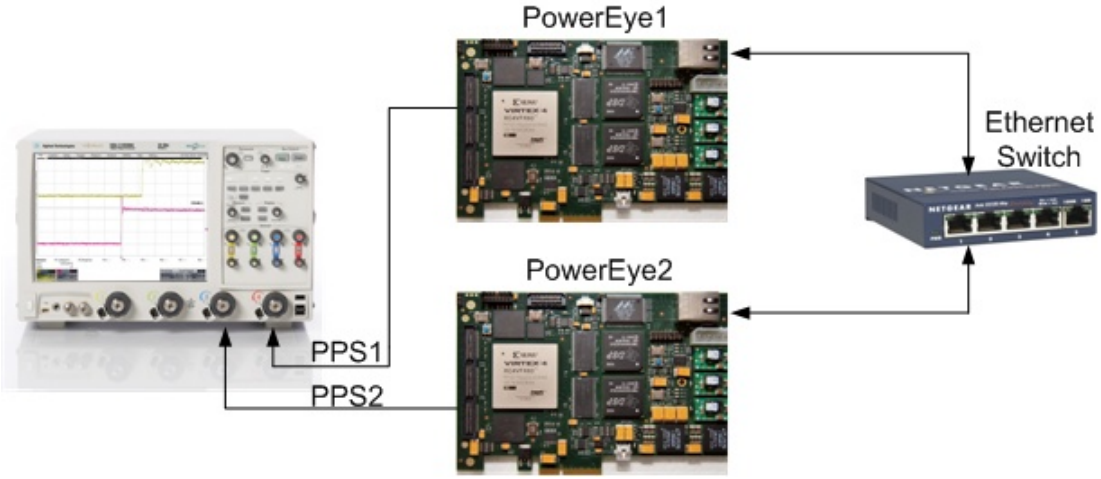


Figure 5.17.: Time deviation measurement between two PowerEye boards

5.4. Performance Evaluation

This section presents the performance evaluation of the above described prototype system in terms of processing throughput, latency and accuracy.

5.4.1. Processing Throughput

The processing throughput is determined by the maximum number of pixels or the maximum number of images at a known resolution that can be processed per second.

In the implemented PowerEye FPGA design, all three Pixel Processing Pipelines are able to be clocked at 150MHz. Since all window-based processing modules, including Gaussian Smoothing, Sobel Edge Detection, Hole Filling and Morphological Filtering, are designed to process 4 pixels every clock cycle, up to $600 Mpixels/s$ can be processed in real-time. Although the blob analysis module processes pixels in serial rather than in parallel, a throughput of $600 Mpixels/s$ can still be achieved according to the performance evaluation presented in Section 4.6.5. Therefore, we may conclude that the

processing power provided by PowerEye is $600Mpixels/s$ or 600 frames per second at the resolution of 1M pixel for each camera.

However, such a processing throughput is not achievable due to the bandwidth limitation of CameraLink. In the prototype system, the cameras are connected to a PowerEye board via the CLinkRx_TripleBase board, which supports only the CameraLink Base interface. As a result, both cameras are configured in the CameraLink Base mode with the operating clock frequency of 80MHz. Since CameraLink Base has three parallel pixel ports, the theoretical pixel transfer rate can reach $240Mpixels/s$. In practice, the achievable pixel transfer rate will be degraded due to the horizontal blanking time and the vertical blanking time (*hblank* and *vblank* shown in Figure 5.4). Given a resolution, the maximum frame rate of the camera can be calculated by the following equation.

$$frame\_rate(fps) = \frac{CameraLink\_clock\_rate(Hz)}{(Image\_Width/3 + hblank) \times Image\_Height + vblank} \quad (5.3)$$

In order to obtain a frame rate of more than 200 fps, the resolution of the MT9M413 sensor is reduced from 1280 × 1024 to 1080 × 1024 in the prototype system. The *hblank* and *vblank* values are set to 16 clock cycles and 5200 clock cycles respectively. According to Equation 5.3, the camera can output 205 frames per second.

As a result, the real processing throughput of the prototype system is $225Mpixels/s$ or 205 frames per second at the resolution of 1080 × 1024 for each camera.

5.4.2. Latency

Latency can be defined as the time that elapses between a movement and the report of that movement. For optical tracking applications, latency is usually expressed in milliseconds. In the presented prototype system, there are mainly three sources that cause latency:

1. After the sensor has been exposed to light, the camera FPGA needs time to read pixels out.
2. The PowerEye FPGA needs to buffer a certain amount of pixels for 2D image processing.
3. It takes time to transmit the tracking results to PC via Ethernet.

Figure 5.18 illustrates a timing chart for better understanding the latency introduced by the hardware system.

The latency estimation is done based on the following two assumptions:

1. Latency is the interval between the time when the image sensor finishes exposure and the time when the host PC gets the 2D marker positions from PowerEye.

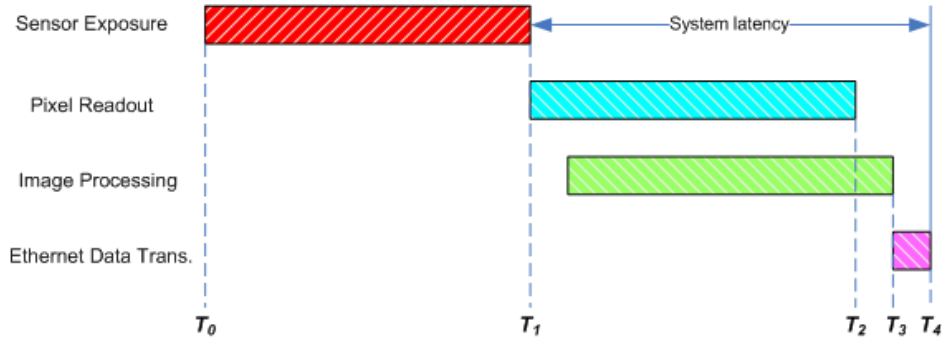


Figure 5.18.: System latency timing

2. The cameras run at 205 fps with the resolution of 1080 × 1024.

In Figure 5.18, time required to readout a complete frame (from T_1 to T_2) can be calculated as follows.

$$T_2 - T_1 = (Image\_Width/3 + hblank) \times Image\_Height \times T_{clk} \tag{5.4}$$

where T_{clk} represents the clock period of the CameraLink interface. Given that the CameraLink interface operates at 80MHz and $hblank = 16$ clock cycles, we can get a latency of 4.813 ms which is caused by the process of pixel readout.

Latency introduced by the PowerEye FPGA is determined by the number of pixels that have to be buffered. Since all modules in the Pixel Processing Pipeline are designed to process pixels "on-the-fly", only a small amount of pixels are buffered. More precisely, 4 image lines for Gaussian Smoothing, 2 image lines for Sobel Edge Detection, 1 image line for Hole Filling, 4 image lines for Morphological Filtering and 1 image line for Blob Analysis. Therefore, we can get that $(T_3 - T_2)$ equals the duration of 12 image lines or 0.057 ms.

The transmission delay between PowerEye and PC Ethernet via Ethernet depends on the size of the packet. As mentioned in Section 5.3.4, the UDP Packet Transmitter organizes pixel data and tracking results into packets with constant size. When the camera is running at the resolution of 1080 × 1024, the UDP packet size is fixed to 1122. According to Figure 5.13, the transmission latency over Ethernet $(T_4 - T_3)$ is around 0.11 ms.

By summing up the numbers listed above, we can get the result that the overall system latency $(T_4 - T_1)$ of the prototype system is 4.98 ms.

5.4.3. Accuracy

The tracking accuracy is evaluated by analyzing the static jitter of the extracted 2D marker positions. In the accuracy measurement setup both cameras and infrared markers were placed at stationary positions. The distance between cameras and markers was

about 1.5 meter. The tracking results for 20,000 frames were recorded. It was measured that the 2D positions of the marker center of gravity jitter on the x axis by averagely 0.029 pixels and on the y axis by averagely 0.033 pixels, and with a standard deviation of 0.014 and 0.019 pixels respectively.

5.5. Summary

This chapter presents a prototype system that incorporates all implemented hardware and software modules in this thesis. Two FPGA designs are described in detail. The Camera FPGA is responsible for sensor control and image data transmission via the CameraLink interface. The PowerEye FPGA integrates three Pixel Processing Pipelines to process pixel streams from three high speed cameras simultaneously. Moreover, a PTP based solution is presented to solve the camera synchronization problem for applications where a large number of cameras and PowerEye boards are used in the tracking environment. At last, the performance evaluation of the prototype system is performed.

6. Summary and Future Work

This chapter summarizes the work presented in this thesis and discusses some options for the future development.

6.1. Summary

In this thesis, a complete hardware platform was designed, implemented and evaluated, for the purpose of high-speed optical tracking.

The developed hardware system consists of three main components: the high-speed camera, the CameraLink grabber and the PowerEye image processing platform. The high-speed camera is equipped with a 1.3M pixel (1280×1024) CMOS sensor, which is capable of operating at 500 frames per second. A low-cost, low-power FPGA is used for operating the image sensor as well as handling the data communication with the host. In order to transfer large amounts of image data produced by the sensor in real-time, CameraLink was chosen as the camera interface. Two different CameraLink grabbers, namely the CLinkRx_TripleBase and the CLinkRx_FULL were developed for connecting cameras with PowerEye. CLinkRx_TripleBase was designed to interface three cameras that are all configured in the CameraLink BASE mode. Using CLinkRx_FULL, it is possible to interface one camera which is configured in the CameraLink FULL mode. PowerEye was designed as a high performance image processing platform, which employs both FPGA and DSPs to perform complex image processing algorithms at high frame rate. In addition to the core processors, high-speed memories, such as the ZBTSRAMs, and rich interface resources, including USB2.0, Gigabit Ethernet, PCI-Express and the 110-bit general-purpose user I/O as well as the 12-pair LVDS Link, are equipped on-board to boost the system performance. The overall hardware system was designed in a modular manner to achieve high scalability. One can scale up from a 3-camera system to a many-camera system which allows a large number of cameras to be introduced in the tracking environment.

One of the most challenging problem presented in a multi-camera based optical tracking system is how to handle the large amounts of image data in real-time. Within the scope of this thesis, the study on FPGA accelerated 2D image processing was performed. The FPGA implementation for a number of image processing algorithms, which are frequently used for optical tracking applications, has been described. For some of the algorithms, such as color segmentation, noise reduction, edge detection

and morphological filtering, one can easily exploit different levels of parallelism. Taking advantage of the parallel processing architecture of the FPGA, a high processing throughput of several thousand Mega pixels per second can be achieved. A new blob analysis algorithm was presented in this thesis. The proposed algorithm supports the detection of tens of thousand blobs and requires only one raster-scan pass throughout the image. Using a highly optimized FPGA implementation, both high processing throughput and low latency can be obtained.

In order to prove the usability of the proposed hardware concept, a prototype system was developed. Individual hardware and software modules were integrated to demonstrate a functioning tracking system. The obtained tracking update rate (205 fps at the resolution of 1080×1024) and latency ($< 5ms$) fulfill the design objective of this work.

Another important issue in a multi-camera based optical tracking system is camera synchronization. A time synchronization solution was implemented using the PTP protocol. By exchanging time information between different network nodes, it is possible to synchronize a large number of cameras at sub-microsecond precision.

6.2. Future Work

As can be observed from previous discussions, the performance of the proposed optical tracking system is not limited by the computing power of PowerEye but by the transmission bandwidth of CameraLink. To achieve higher tracking update rate and lower latency, it is necessary to utilize more advanced transmission technologies. Recently, the Automated Imaging Association (AIA) has released the draft of the CameraLink HS standard - the next generation of CameraLink. CameraLink HS features high bandwidth (up to 33.6 Gbit/s), data reliability, low jitter and built-in fault tolerance with CRC (cyclical redundancy check). The standard is expected to be finalized and released in 2012. Integrating the CameraLink HS interface inside the FPGA can be considered to be one of the future development areas.

The work in this thesis focuses on the hardware system design and the FPGA based high-speed 2D image processing. However, a complete optical tracking system should also include camera calibration, feature point matching and 3D reconstruction. At the moment, the two DSPs on PowerEye remain free for accomplishing these jobs. It is an important aspect of the future research to develop suitable algorithms for the missing parts in optical tracking and map them into DSPs.

Bibliography

[ABD04] Daniel F. Abawi, Joachim Bienwald, and Ralf Dorner. Accuracy in optical tracking with fiducial markers: An accuracy function for artoolkit. In *Proceedings of the 3rd IEEE/ACM International Symposium on Mixed and Augmented Reality*, ISMAR '04, pages 260–261, Washington, DC, USA, 2004. IEEE Computer Society.

[Abd07] Abdallah S. Abdallah. Investigation of new techniques for face detection. Master's thesis, Virginia Polytechnic Institute and State University, 2007.

[ABS10] Nikolaos Alachiotis, Simon A. Berger, and Alexandros Stamatakis. Efficient pc-fpga communication over gigabit ethernet. In *CIT*, pages 1727–1734. IEEE Computer Society, 2010.

[Alt08] Altera Inc. *Memory System Design*, 2008.

[And96] Ray Andraka. A dynamic hardware video processing platform. Technical report, Andraka Consulting Group, 1996.

[And98] Ray Andraka. A survey of cordic algorithms for fpga based computers. In *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 191–200, New York, NY, USA, 1998. ACM.

[AR08] A. Gray A. Richardson. Utilisation of the gpu architecture for hpc. Technical report, University of Edinburgh, 2008.

[Art08] Clemens Arth. *Visual Surveillance on DSP-Based Embedded Platforms*. PhD thesis, Graz University of Technology, 2008.

[AS89] M. R. AZIMI-SADJADI. Parallel and pipeline architectures for 2-d block processing. *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS,*, 36, NO.3:443–448, 1989.

[Aut01] Automated Imaging Association. *Camera Link Specifications of the Camera Link Interface Standard for Digital Cameras and Frame Grabbers*, 2001. Ver 1.1.

[Bap00] Shekhar Bapat. Synthesizable 200 mhz zbt sram interface. *Xilinx Application Note, XAPP136*, 2000.

[Bax94] G.A. Baxes. *Digital Image Processing. Principles & Applications.* Wiley & Sons, 1994.

[Bei06] Florian Beier. Optisches tracking von deformierbaren objekten. *Diploma thesis, Dept. of Mathematics and Computer Science, University of Mannheim*, 2006.

[Bra94] D. H. Brainard. Bayesian method for reconstructing color images from trichromatic samples. In *Proceedings of the IS&T 47th Annual Meeting*, pages 375–380, Rochester, NY, USA, May 1994.

[BS05] A.G. Buaes and P. Santos. A survey on the available optical tracking systems for ar/vr indoor applications. 2005.

[Bua05] A.G. Buaes. A survey on the available optical tracking systems for ar/vr indoor applications. Technical report, PPGEE, UFRGS; Porto Alegre, 2005.

[CCLW05] Ben Cope, Peter Y. K. Cheung, Wayne Luk, and Sarah Witt. Have gpus made fpgas redundant in the field of video processing? In *FPT*, pages 111–118, 2005.

[Cel90] Mehmet Celenk. A color clustering technique for image segmentation. *Comput. Vision Graph. Image Process.*, 52(2):145–170, 1990.

[Che02] Xing Chen. *Design of many-camera tracking systems for scalability and efficient resource allocation.* PhD thesis, Stanford, CA, USA, 2002. Adviser-Hanrahan, Patrick M.

[Chi06] Lixin Chin. Fpga based embedded vision systems. Master's thesis, The University of Western Australia, 2006.

[CKKP01] Jaeyong Chung, Namgyu Kim, Gerard Jounghyun Kim, and Chan-Mo Park. Postrack: A low cost real-time motion tracking system for vr application. In *In International conference on Virtual Systems and MultilMedia*, pages 383–392. IEEE, 2001.

[CKRB03] Chen Chang, Kimmo Kuusilinna, Brian Richards, and Robert W. Brodersen. Implementation of bee: a real-time large-scale hardware emulation engine. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, FPGA '03, pages 91–99, New York, NY, USA, 2003. ACM.

[col09] *Color Space*, 2009. `http://en.wikipedia.org/wiki/Color_space`.

[Cop08] Benjamin Thomas Cope. *Video Processing Acceleration using Reconfigurable Logic and Graphics Processors*. PhD thesis, Department of Electrical and Electronic Engineering, Imperial College London, 2008.

[Cou03] Simon Coulson. Real time positioning and motion tracking for simulated clay pigeon shooting environments. Master's thesis, Imperial College London, 2003.

[CTJG05] D. G. Bailey C. T. Johnston and K. T. Gribbon. Optimisation of a colour segmentation and tracking algorithm for real-time fpga implementation. In *Proceedings of Image and Vision Computing Conference New Zealand (IVCNZ '05)*, pages 422–427, Dunedin, New Zealand, 2005.

[CU008] *NVIDIA CUDA Compute Unified Device Architecture Programming Guide v2.0*, 2008.

[dR06] Bart de Ruijsscher. Fpga based accelerator for real-time skin segmentation. Master's thesis, Delft University of Technology, 2006.

[dSB99] Luigi di Stefano and Andrea Bulgarelli. A simple and efficient connected components labeling algorithm. In *ICIAP '99: Proceedings of the 10th International Conference on Image Analysis and Processing*, page 322, Washington, DC, USA, 1999. IEEE Computer Society.

[DST92] Michael B. Dillencourt, Hannan Samet, and Markku Tamminen. A general approach to connected-component labeling for arbitrary image representations. *J. ACM*, 39(2):253–280, 1992.

[EN08] A. Postula B.C. Lovell E. Norouznezhad, A. Bigdeli. A high resolution smart camera with gige vision extension for surveillance applications. *2nd ACM/IEEE International Conference on Distributed Smart Cameras (ICDSC'08)*, 2008.

[Far09] John Patrick Farrell. Digital hardware design decisions and trade-offs for software radio systems. Master's thesis, Virginia Polytechnic Institute and State University, 2009.

[FP02] David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Prentice Hall Professional Technical Reference, 2002.

[FS09] Akira Asano Febriliyan Samopa. Hybrid image thresholding method using edge detection. *International Journal of Computer Science and Network Security*, VOL.9 No.4, 2009.

[GW06] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[Han93] Chris Hand. A survey of 3-d input devices. *Technical Report CS TR94/2; Department of Computer Science, De Montfort University;*, 1993.

[Hed08] Hugo Hedberg. *Image Processing Architectures for Binary Morphology and Labeling*. PhD thesis, Department of Electrical and Information Technology, Lund University, 2008.

[Hem03] E. E. Hemayed. A survey of camera self-calibration. In S. Kawada, editor, *IEEE Conference on Advanced Video and Signal Based Surveillance, Proceedings*. IEEE Comp Soc, TC Pattern Anal & Machine Intelligence; Int Soc Informat Fus, IEEE Computer Soc, 2003.

[HS93] R.M. Haralick and L.G. Shapiro. *Computer and Robot Vision*. Addison-Wesley Publishing, first edition, 1993.

[HS97a] Richard I. Hartley and Peter F. Sturm. Triangulation. *Computer Vision and Image Understanding*, 68(2):146–157, 1997.

[HS97b] Janne Heikkila and Olli Silven. A four-step camera calibration procedure with implicit image correction. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, 0:1106, 1997.

[HZ04] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[IDT04] Dual port memory simplifies wireless base station design. Technical report, Integrated Device technology, Inc., 2004.

[Int81] Internet Engineering Task Force. *RFC 791 Internet Protocol - DARPA Inernet Programm, Protocol Specification*, September 1981.

[ISUB05] A. Amira I. S. Uzun and A. Bouridane. Fpga implementations of fast fourier transforms for real-time signal and image processing. *Vision, Image and Signal Processing, IEE Proceedings*, 152:283–296, 2005.

[Jai88] Anil K. Jain. *Fundamentals of Digital Image Processing*. Prentice Hall, September 1988.

[JGB04] C. T. Johnston, K. T. Gribbon, and D. G. Bailey. Implementing image processing algorithms on fpgas. In *Proc. Eleventh Electronics New Zealand Conference, Palmerston North, New Zealand*, pages 118–123, 2004.

[JGH99] Bernd Jähne, Peter Geissler, and Horst Haussecker, editors. *Handbook of Computer Vision and Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.

[JKS95] Ramesh Jain, Rangachar Kasturi, and Brian G. Schunck. *Machine vision*. McGraw-Hill, Inc., New York, NY, USA, 1995.

[KDF01] Z. Ji K. Dong, M.Hu and B. Fang. Research on architectures for high performance image processing. In *Proceedings of the Fourth International Workshop on Anvanced Parallel Processing Technologies*, 2001.

[KGH02] V. Khanna, P. Gupta, and C.J. Hwang. Finding connected components in digital images by aggressive reuse of labels. 20(8):557–568, June 2002.

[Kol07] Norbert Koller. *Fully Automated Repair of Surface Flaws using an Artificial Vision Guided Robotic Grinder*. PhD thesis, Institute for Automation University of Leoben, 2007.

[LE02] Kang Lee and John Eidson. Ieee1588 standard for a precision clock synchronization protocol for networked measurement and control systems. In *In 34 th Annual Precise Time and Time Interval (PTTI) Meeting*, pages 98–105, 2002.

[Lie09] Manfred Liebmann. High performance computing with graphics processing units. Technical report, Department of Mathematics, University of Wyoming, 2009.

[Lin96] Tony Lindeberg. Edge detection and ridge detection with automatic scale selection. In *CVPR '96: Proceedings of the 1996 Conference on Computer Vision and Pattern Recognition (CVPR '96)*, page 465, Washington, DC, USA, 1996. IEEE Computer Society.

[LKPB02] P. Lang, A. Kusej, A. Pinz, and G. Brasseur. Inertial tracking for mobile augmented reality. In *Proc. of the IMTC 2002*, volume 2, pages 1583–1587, Anchorage, USA, 2002.

[LM03] Robert Van Liere and Jurriaan D. Mulder. Optical tracking using projective invariant marker pattern properties. In *In Proceedings of the IEEE Virtual Reality Conference 2003 (2003*, pages 191–198. IEEE Press, 2003.

[MAB92] Kenneth Meyer, Hugh L. Applewhite, and Frank A. Biocca. A survey of position trackers. *Presence: Teleoper. Virtual Environ.*, 1(2):173–200, 1992.

[Mac05] W. James MacLean. An evaluation of the suitability of fpgas for embedded vision systems. In *CVPR '05: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Workshops*, page 131, Washington, DC, USA, 2005. IEEE Computer Society.

[Mat05] Hervé Mathieu. The Cyclope : A 6 DOF optical tracker based on a single camera. In *2nd INTUITION International Workshop "VR/VE & Industry: Challenges and opportunities", November, 2005*, Senlis, France, 2005.

[MDP07] R. Mosqueron, J. Dubois, and M. Paindavoine. High-speed smart camera with high resolution. *EURASIP J. Embedded Syst.*, 2007(1):23–23, 2007.

[Meh06] Michael Mehling. Implementation of a low cost marker based infrared light optical tracking system. Master's thesis, Vienna University of Technology, 2006.

[MG96] Tomasz Mazuryk and Michael Gervautz. Virtual reality history, applications, technology and future. Technical report, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 1996.

[Mic06] Micron Technology, Inc. *MT9M413 Cmos Image Sensor Data Sheet*, 2006.

[MJvR03] Jurriaan D. Mulder, Jack Jansen, and Arjen van Rhijn. An affordable optical head tracking system for desktop vr/ar systems, 2003.

[MT009] *MicronTracker*, 2009. `http://www.clarontech.com/`.

[Nat06] National Semiconductor. *Channel Link Design Guide*, 2006.

[Par06] Nikhil Paruchuri. *The Kuhabs, Kubesat & KuteSAT-1 Technical Report, Design of a Modular Platform for Picosatellites.* PhD thesis, University of Kansas, 2006.

[Pos80] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[PS07] Ying Piao and Jun Sato. Computing epipolar geometry from unsynchronized cameras. In *ICIAP*, pages 475–480, 2007.

[RAA95] Ramana V. Rachakonda, Peter M. Athanas, and A. Lynn Abbott. High-speed region detection and labeling using an fpga based custom computing platform. In *FPL '95: Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications*, pages 86–93, London, UK, 1995. Springer-Verlag.

[Rib01] Miguel Ribo. State of the Art Report on Optical Tracking, 2001.

[RK82] A. Rosenfeld and A. C. Kak. *Digital Picture Processing*, volume 2, chapter 10.5 Iterative Segmentation: "Relaxation", pages 152–184. Academic Press, 1982.

[RP66] Azriel Rosenfeld and John L. Pfaltz. Sequential operations in digital picture processing. *J. ACM*, 13(4):471–494, 1966.

[RPBM06] Daggu Venkateshwar Rao, Shruti Patil, Naveen Anne Babu, and V Muthukumar. Implementation and evaluation of image processing algorithms on reconfigurable architecture using c-based hardware descriptive languages, 2006.

[Rus02] John C. Russ. *Image Processing Handbook, Fourth Edition.* CRC Press, Inc., Boca Raton, FL, USA, 2002.

[Sch04] Marc Schneberger. Infrared optical tracking systems mathematical and operation principles. 2004. `http://www.ar-tracking.de`.

[SMB00] Cordelia Schmid, Roger Mohr, and Christian Bauckhage. Evaluation of interest point detectors. *Int. J. Comput. Vision*, 37(2):151–172, 2000.

[Soi99] P. Soille. *Morphological Image Analysis: Principles and Applications.* Springer-Verlag, 1999.

[SSZ01] D. Scharstein, R. Szeliski, and R. Zabih. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. In *SMBV '01: Proceedings of the IEEE Workshop on Stereo and Multi-Baseline Vision (SMBV'01)*, page 131, Washington, DC, USA, 2001. IEEE Computer Society.

[Tex01] Texas Instruments Inc. *TMS320C64x Technical Overview (Rev. B)*, 2001.

[Tex09] Texas Instruments Inc. *Datasheet of TMS320C6414T, TMS320C6415T, TMS320C6416T Fixed-point Digital Signal Processors (Rev. M)*, 2009.

[TFM96] S. Thorpe, D. Fize, and C. Marlot. Speed of processing in the human visual system. *Nature*, 381(6582):520–522, June 1996.

[Tom86] S. Tominaga. Color image segmentation using three perceptual attributes. In *Proceedings of the Conference Vision and Pattern Recognition*, pages 628–630, Los Alamitos, CA, 1986. IEEE Computer Society.

[TS04] Prithiraj Tissainayagam and David Suter. Assessing the performance of corner detectors for point feature tracking applications. *Image Vision Comput.*, 22(8):663–679, 2004.

[WBM04] Georg Wiora, Pavel Babrou, and Reinhard Männer. Real time high speed measurement of photogrammetric targets. In *DAGM-Symposium*, volume 3175 of *Lecture Notes in Computer Science*, pages 562–569. Springer, 2004.

[WF02] Greg Welch and Eric Foxlin. Motion tracking: No silver bullet, but a respectable arsenal. *IEEE Comput. Graph. Appl.*, 22(6):24–38, 2002.

[Wil04] Richard Williams. Increase image processing system performance with fpgas. Technical report, Hunt Engineering (UK) Ltd., 2004.

[WlCCcL03] Kuang-Bor Wang, Tsorng lin Chia, Zen Chen, and Der chyuan Lou. Parallel execution of a connected component labeling operation on a linear array architecture. *Journal of Information Science and Engineering*, 19:353–370, 2003.

[WOS09] Kesheng Wu, Ekow J. Otoo, and Kenji Suzuki. Optimizing two-pass connected-component labeling algorithms. *Pattern Anal. Appl.*, 12(2):117–135, 2009.

[WP11] *WinPcap*, 2011. `http://www.winpcap.org/`.

[Xil06] Xilinx Inc. *Using Digital Clock Managers (DCMs) in Spartan-3 FPGAs*, 2006.

[Xil07] Xilinx Inc. *Virtex-4 Family Overview*, 2007.

[Xil09] Xilinx Inc., http://www.xilinx.com/support/documentation/virtex-6.htm. *Virtex-6 FPGA Configurable Logic Block User Guide*, 2009.

[Zha00] Zhengyou Zhang. A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(11):1330–1334, 2000.

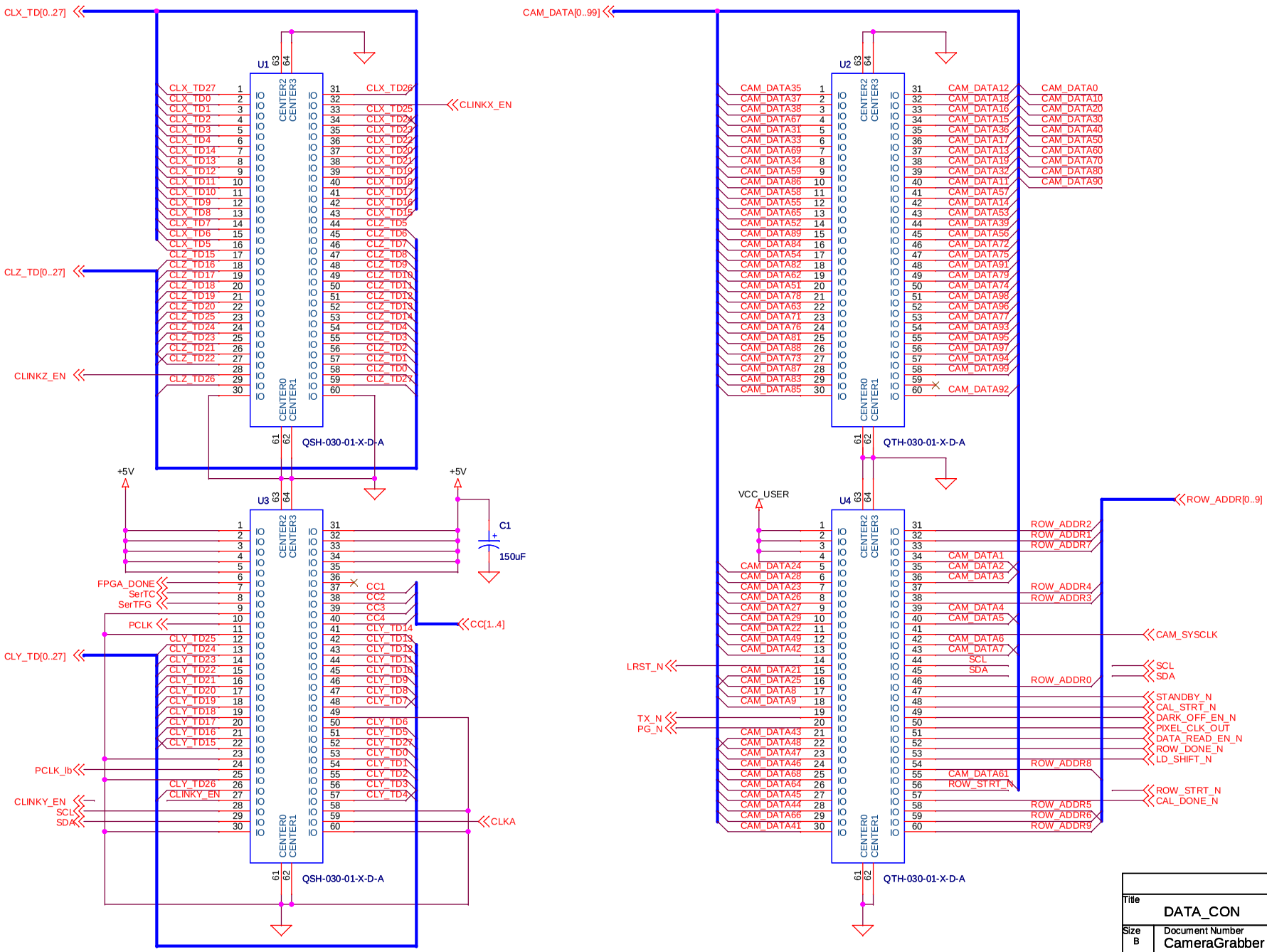
[Zim80] H. Zimmermann. OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.

[Zur01] Nicolas Blanc Zurich. Ccd versus cmos – has ccd imaging come to an end? *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 154–158, 2001.

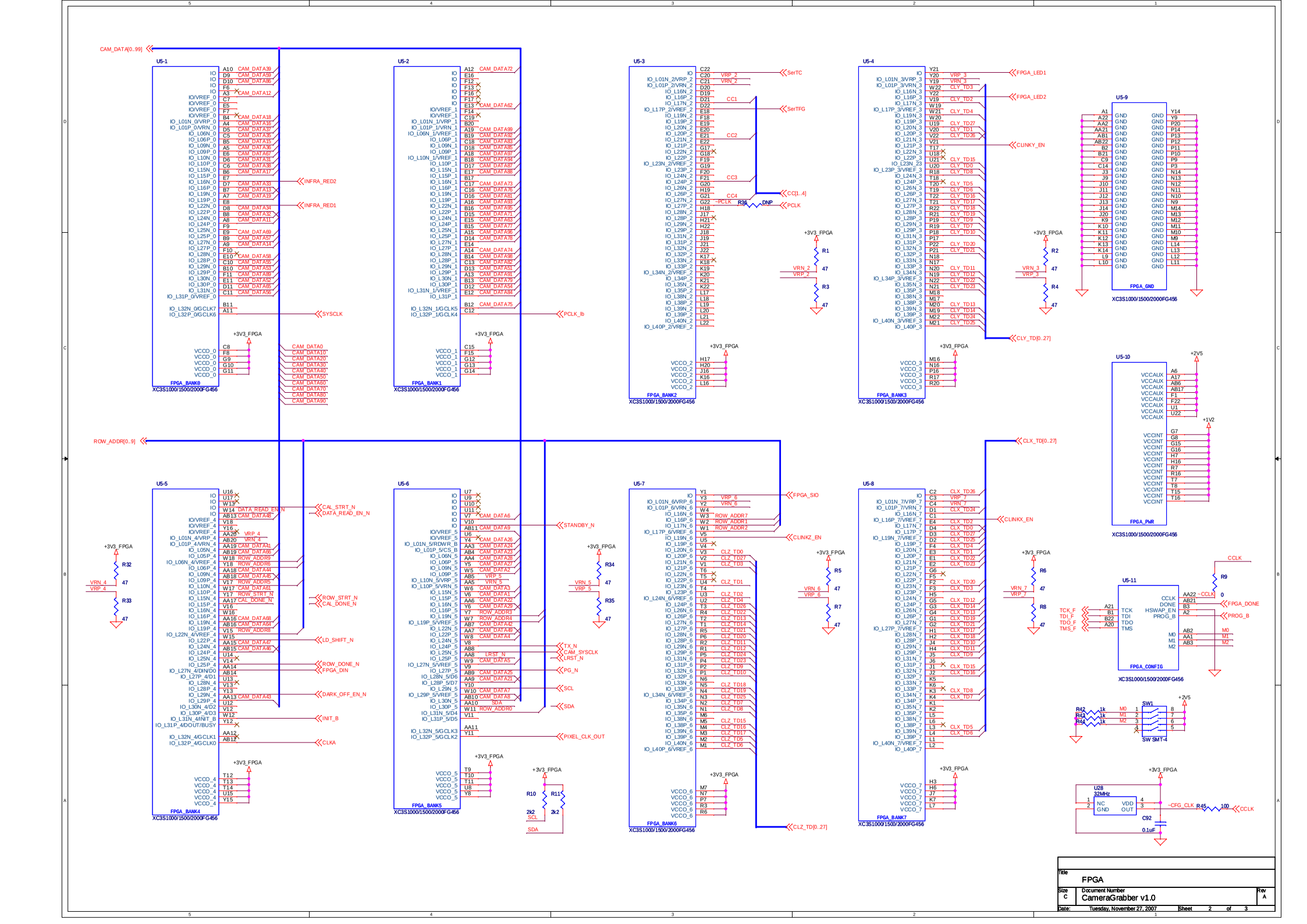
[ZXH07] Hui Zhang, Mingxin Xia, and Guangshu Hu. A multiwindow partial buffering scheme for fpga-based 2-d convolvers. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 54(2):200–204, 2007.

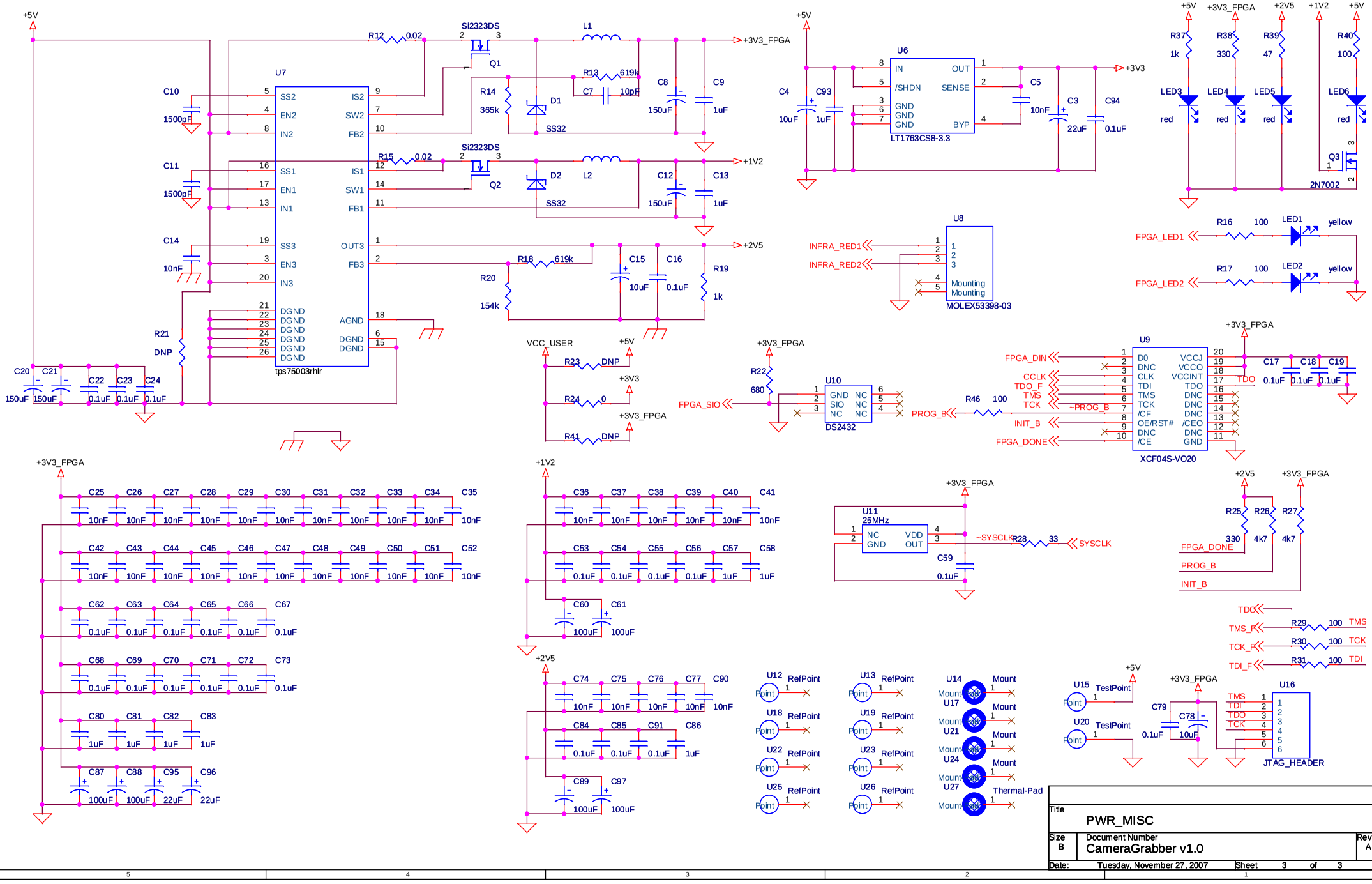
A. Schematics

A.1. Camera FPGA Board



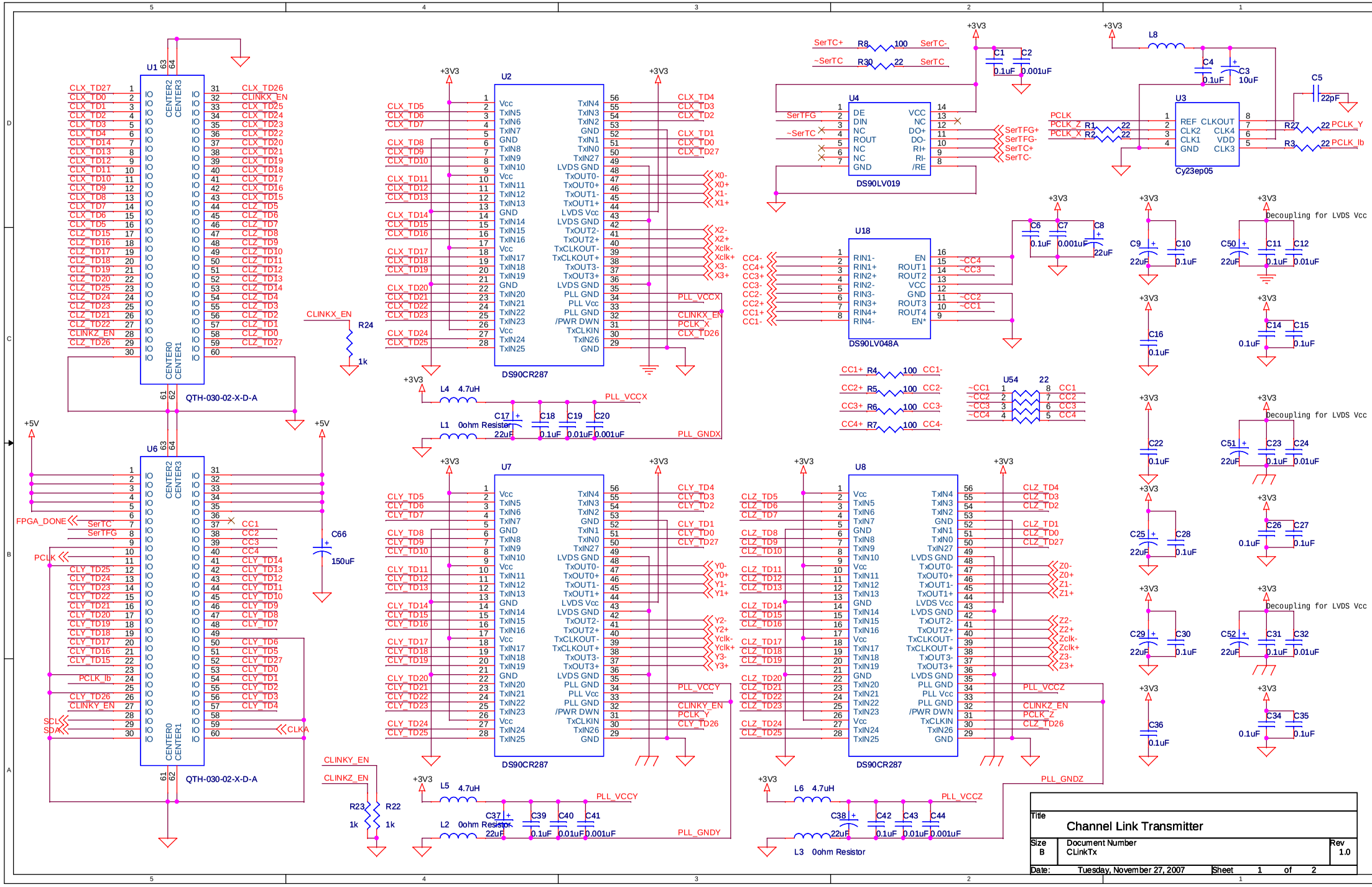
Title	DATA_CON	
Size B	Document Number CameraGrabber v1.0	Rev A
Date: Tuesday, November 27, 2007	Sheet 1 of 3	

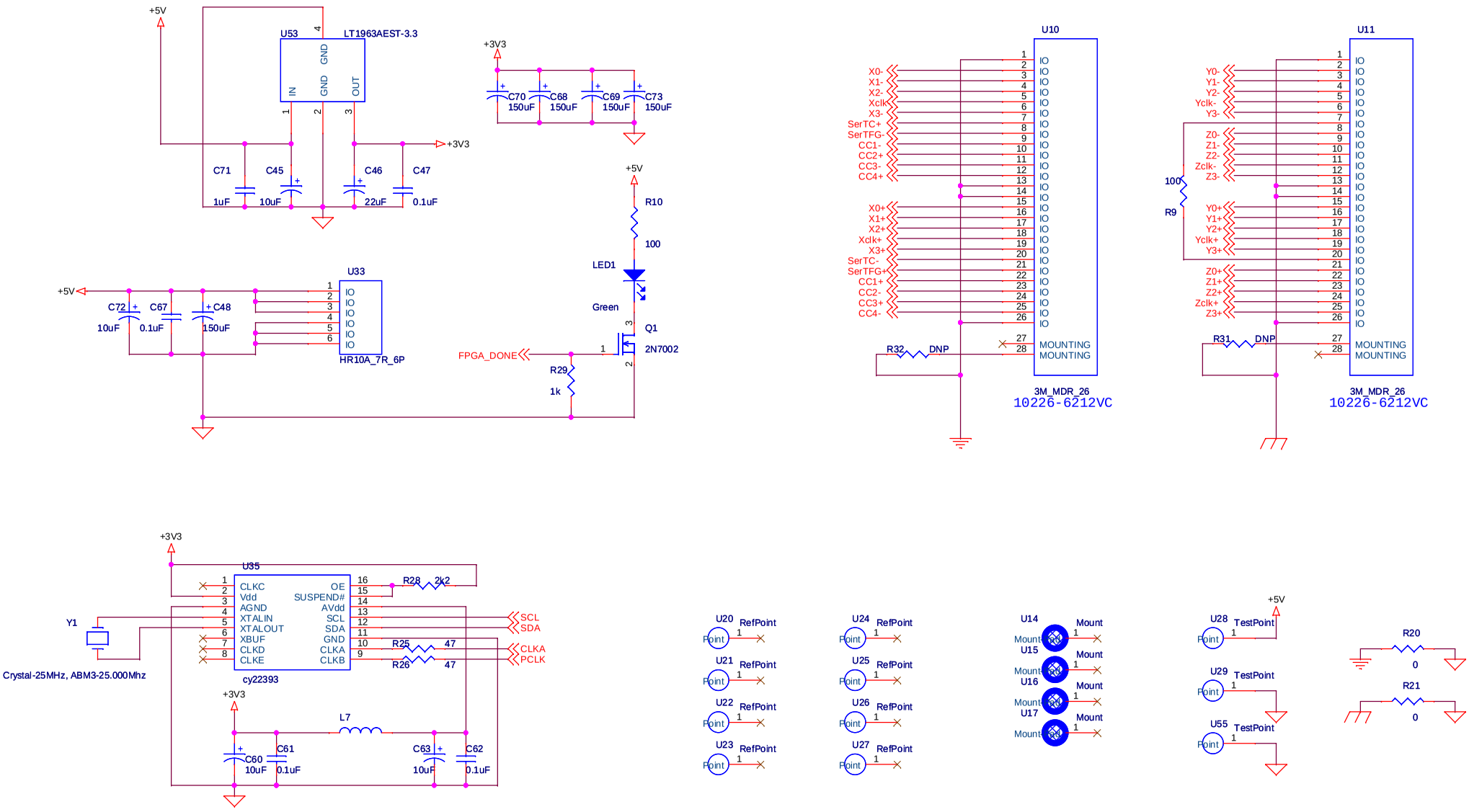

CAM_DATA[0..99]
U5-1
FPGA_BANK0
XC3S1000/1500/2000FG456
INFRA_RED2
INFRA_RED1
SYSCLK
U5-2
FPGA_BANK1
XC3S1000/1500/2000FG456
PCLK_lb
U5-3
FPGA_BANK2
XC3S1000/1500/2000FG456
SerTC
SerTFG
CC[1..4]
PCLK
R36
DNP
U5-4
FPGA_BANK3
XC3S1000/1500/2000FG456
FPGA_LED1
FPGA_LED2
CLINKY_EN
CLY_TD[0..27]
U5-9
FPGA_GND
XC3S1000/1500/2000FG456
U5-10
FPGA_PWR
XC3S1000/1500/2000FG456
ROW_ADDR[0..9]
U5-5
FPGA_BANK4
XC3S1000/1500/2000FG456
CAL_STRT_N
DATA_READ_EN_N
ROW_STRT_N
CAL_DONE_N
LD_SHIFT_N
ROW_DONE_N
FPGA_DIN
DARK_OFF_EN_N
INIT_B
CLKA
U5-6
FPGA_BANK5
XC3S1000/1500/2000FG456
STANDBY_N
TX_N
CAM_SYSCLK
LRST_N
PG_N
SCL
SDA
PIXEL_CLK_OUT
U5-7
FPGA_BANK6
XC3S1000/1500/2000FG456
FPGA_SIO
CLINKZ_EN
CLZ_TD[0..27]
U5-8
FPGA_BANK7
XC3S1000/1500/2000FG456
CLX_TD[0..27]
CLINKX_EN
U5-11
FPGA_CONFIG
XC3S1000/1500/2000FG456
FPGA_DONE
PROG_B
SW1
SW SMT-4
U28
32MHz
CCLK
~CFG_CLK
C92
0.1uF
+3V3_FPGA
+2V5
+1V2
Title
FPGA
Size
C
Document Number
CameraGrabber v1.0
Rev
A
Date:
Tuesday, November 27, 2007
Sheet
2
of
3



Title	PWR_MISC
Size B	Document Number CameraGrabber v1.0 Rev A
Date: Tuesday, November 27, 2007	Sheet 3 of 3

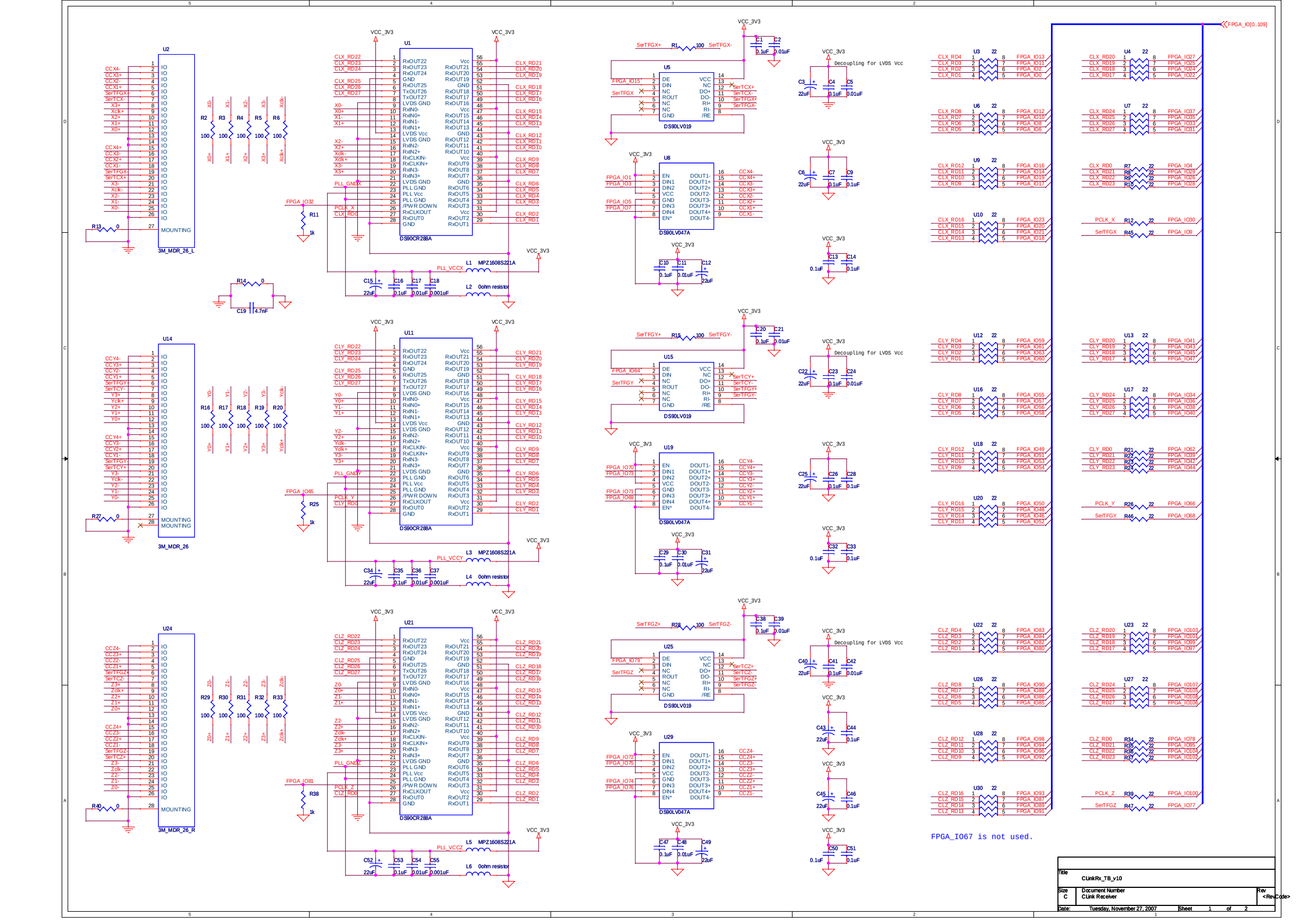
A.2. CLinkTx Board

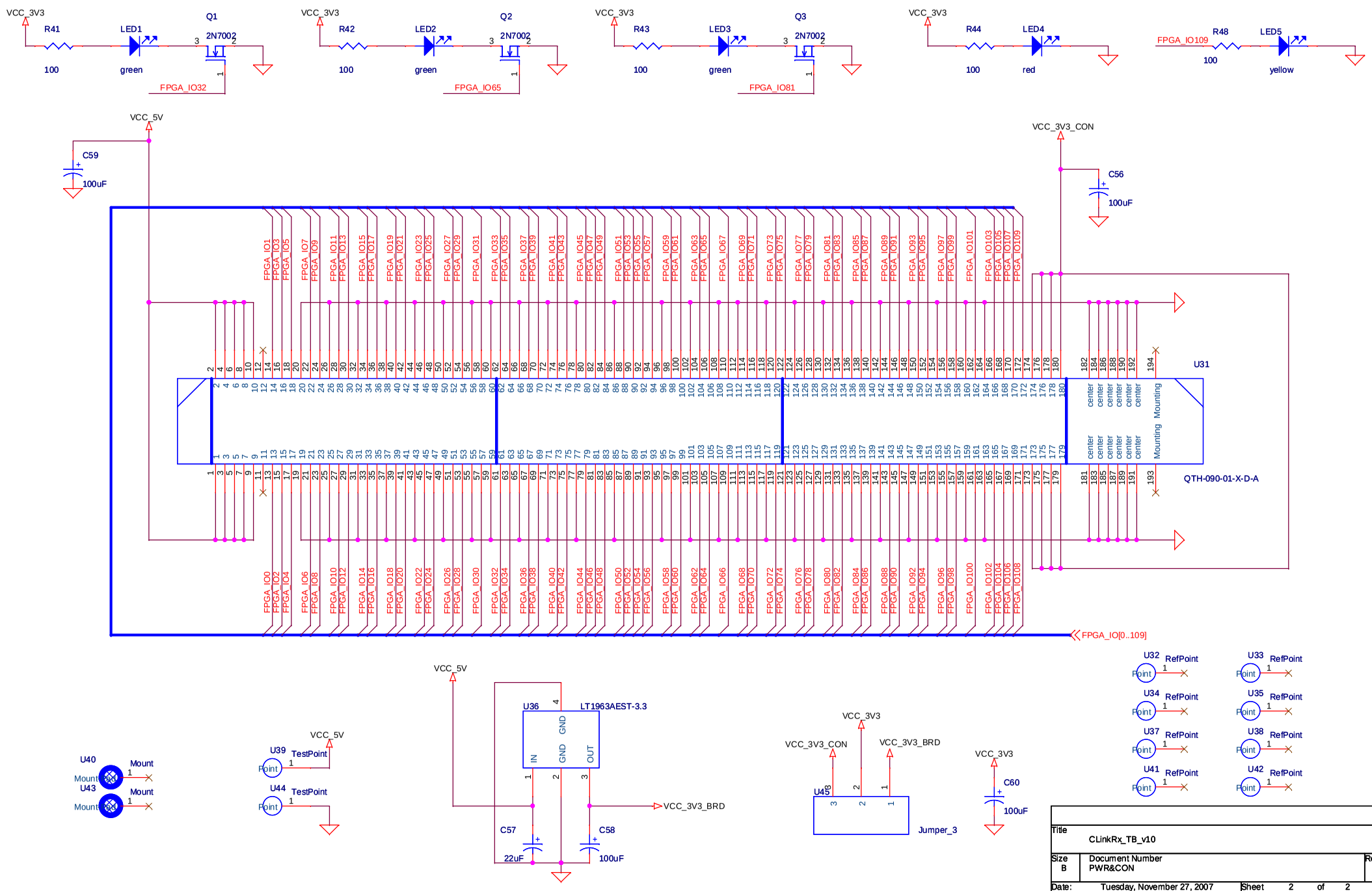

U1
QTH-030-02-X-D-A
U2
DS90CR287
U3
Cy23ep05
U4
DS90LV019
U6
QTH-030-02-X-D-A
U7
DS90CR287
U8
DS90CR287
U18
DS90LV048A
U54
+3V3
+5V
Decoupling for LVDS Vcc
Title
Channel Link Transmitter
Size B
Document Number CLinkTx
Rev 1.0
Date: Tuesday, November 27, 2007
Sheet 1 of 2



Title	PWR_MDR26_CON	
Size B	Document Number CLinkTx	Rev 1.0
Date: Tuesday, November 27, 2007	Sheet 2 of 2	

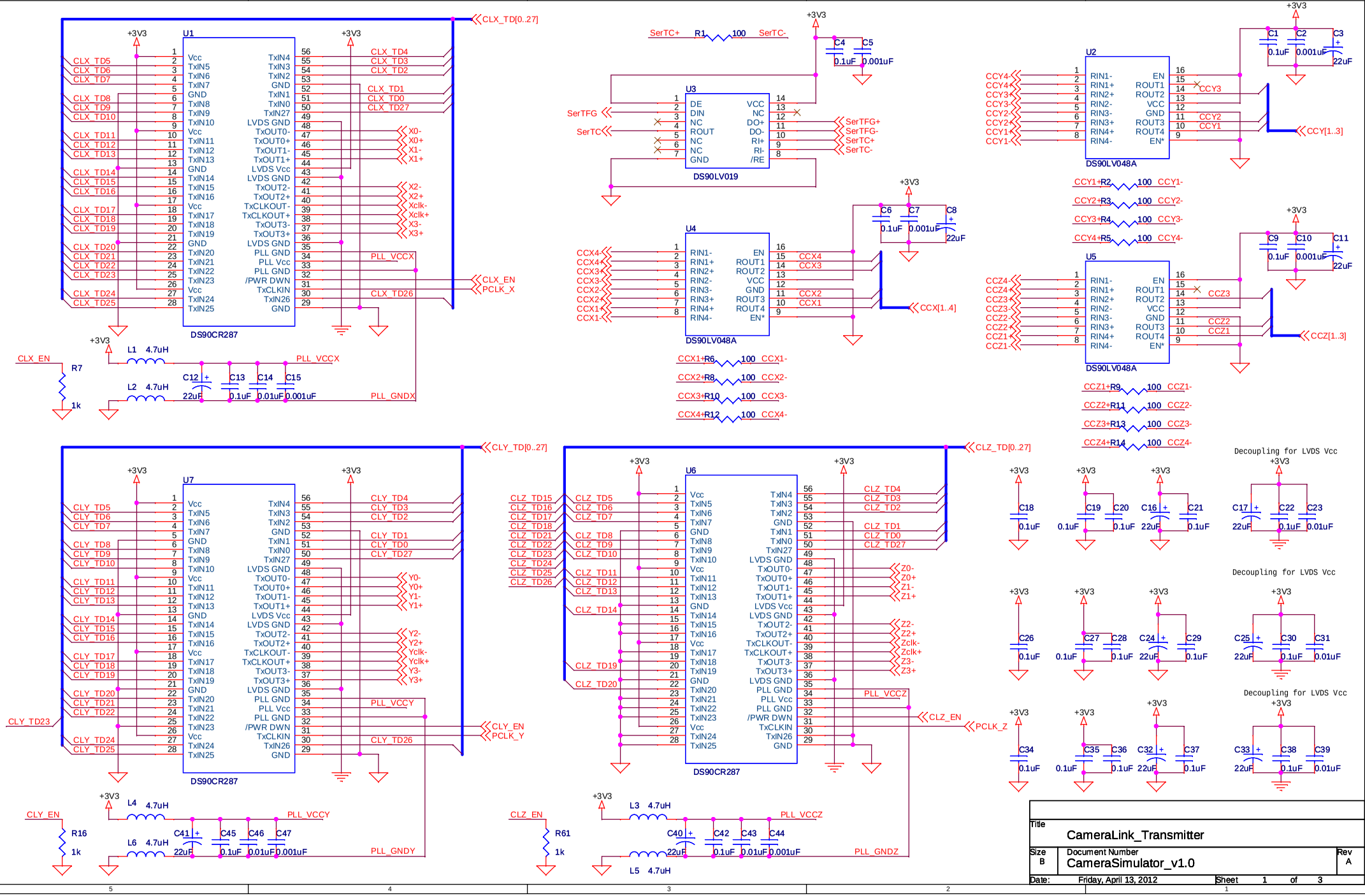
A.3. CLinkRx_TripleBase Board

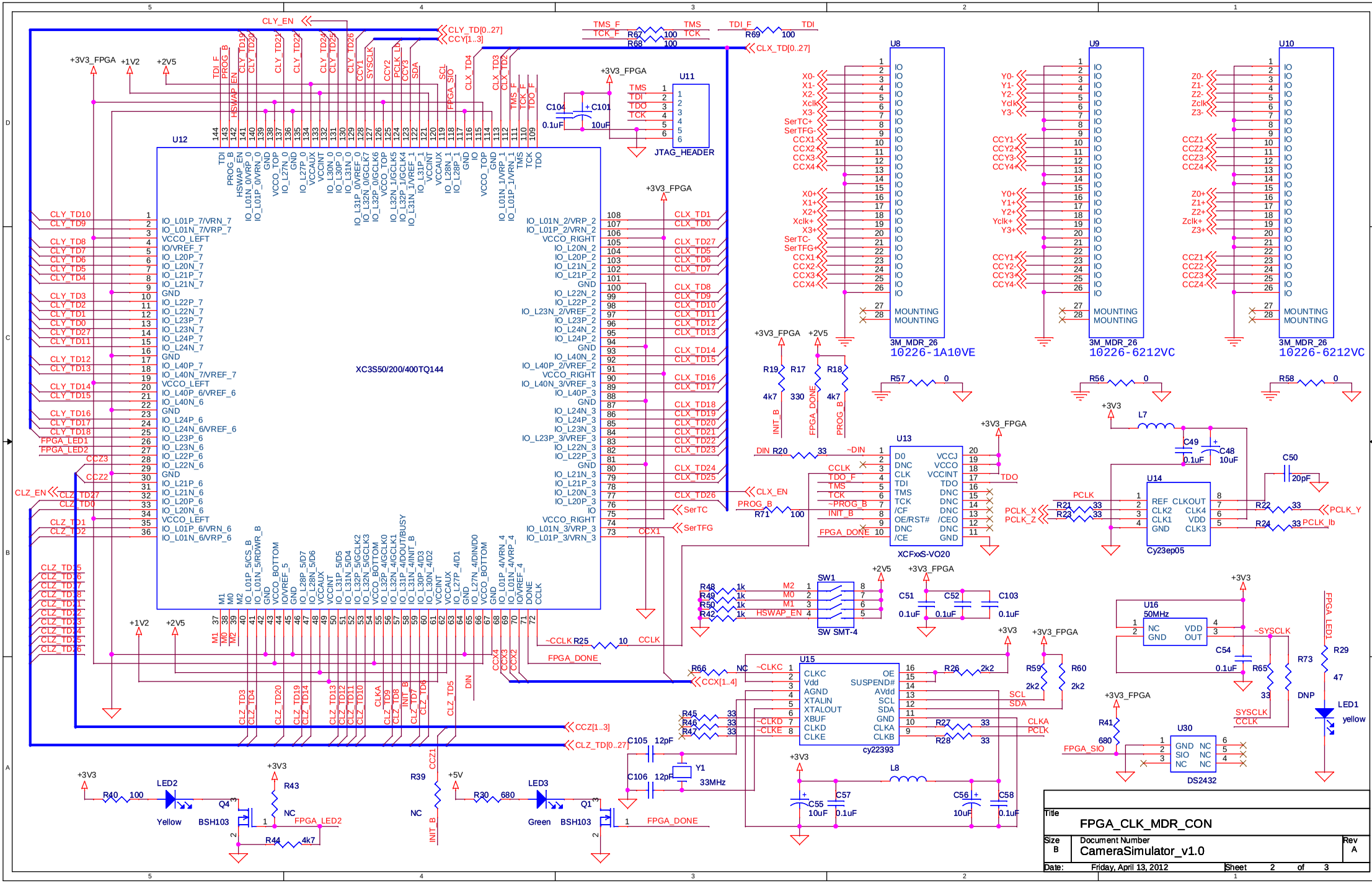

U2
3M_MDR_26_L
U1
DS90CR288A
U5
DS90LV019
U8
DS90LV047A
U14
3M_MDR_26
U11
U15
U19
U24
3M_MDR_26_R
U21
U25
U29
U3
U4
U6
U7
U9
U10
U12
U13
U16
U17
U18
U20
U22
U23
U26
U27
U28
U30
VCC_3V3
Decoupling for LVDS Vcc
FPGA_IO[0..109]
FPGA_IO67 is not used.
Title
CLinkRx_TB_v10
Size C
Document Number
CLink Receiver
Rev
<RevCode>
Date: Tuesday, November 27, 2007
Sheet 1 of 2



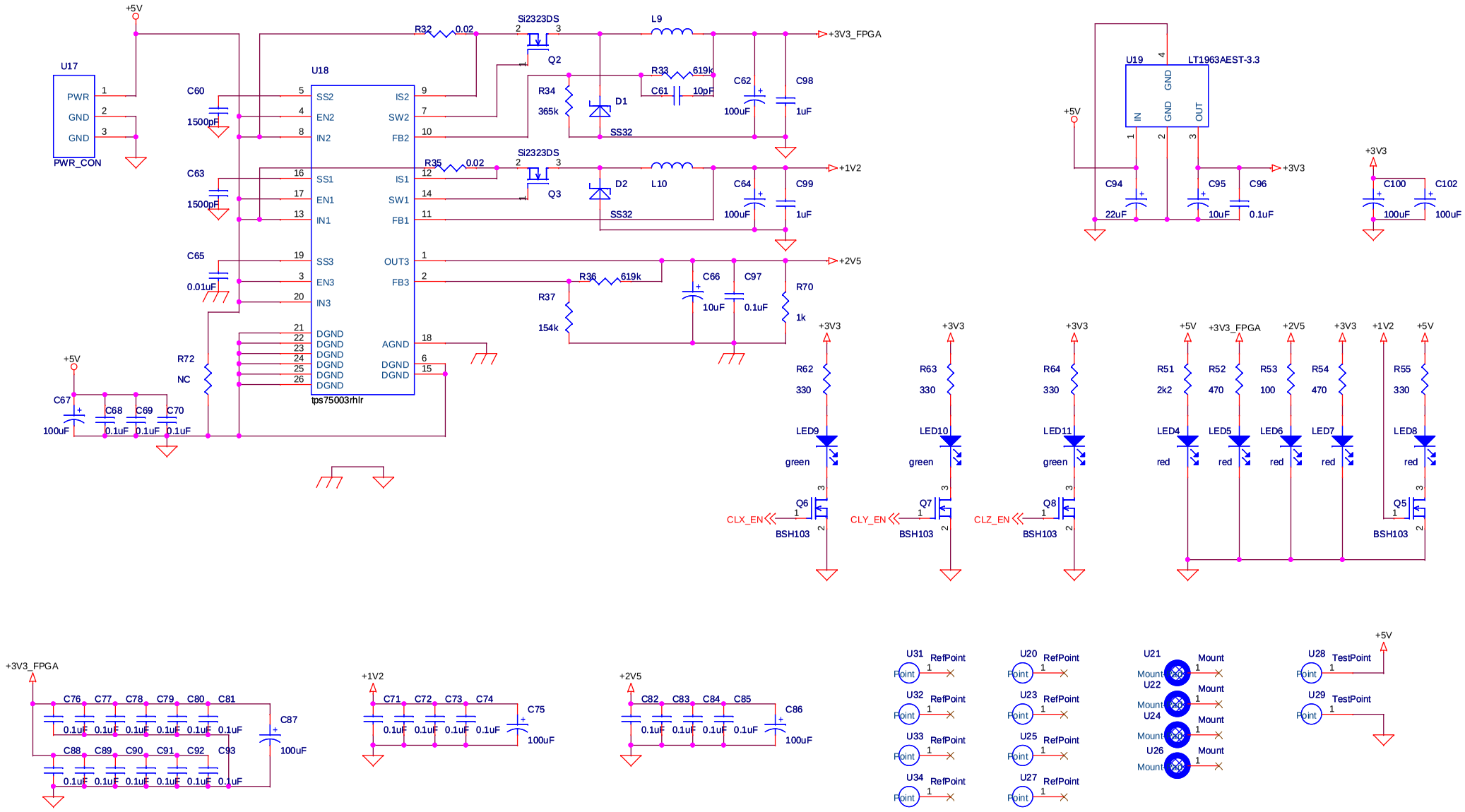
Title	CLinkRx_TB_v10	
Size B	Document Number PWR&CON	Rev <Rev
Date: Tuesday, November 27, 2007	Sheet 2 of 2	

A.4. CameraLink Simulator Board


U1
DS90CR287
CLX_TD[0..27]
CLX_EN
PCLK_X
PLL_VCCX
PLL_GNDX
U3
DS90LV019
SerTC+
SerTC-
SerTFG
SerTFG+
SerTFG-
SerTC+
SerTC-
U2
DS90LV048A
CCY[1..3]
U4
DS90LV048A
CCX[1..4]
U5
DS90LV048A
CCZ[1..3]
U7
DS90CR287
CLY_TD[0..27]
CLY_EN
PCLK_Y
PLL_VCCY
PLL_GNDY
U6
DS90CR287
CLZ_TD[0..27]
CLZ_EN
PCLK_Z
PLL_VCCZ
PLL_GNDZ
Decoupling for LVDS Vcc
Title
CameraLink_Transmitter
Size B
Document Number
CameraSimulator_v1.0
Rev A
Date: Friday, April 13, 2012
Sheet 1 of 3



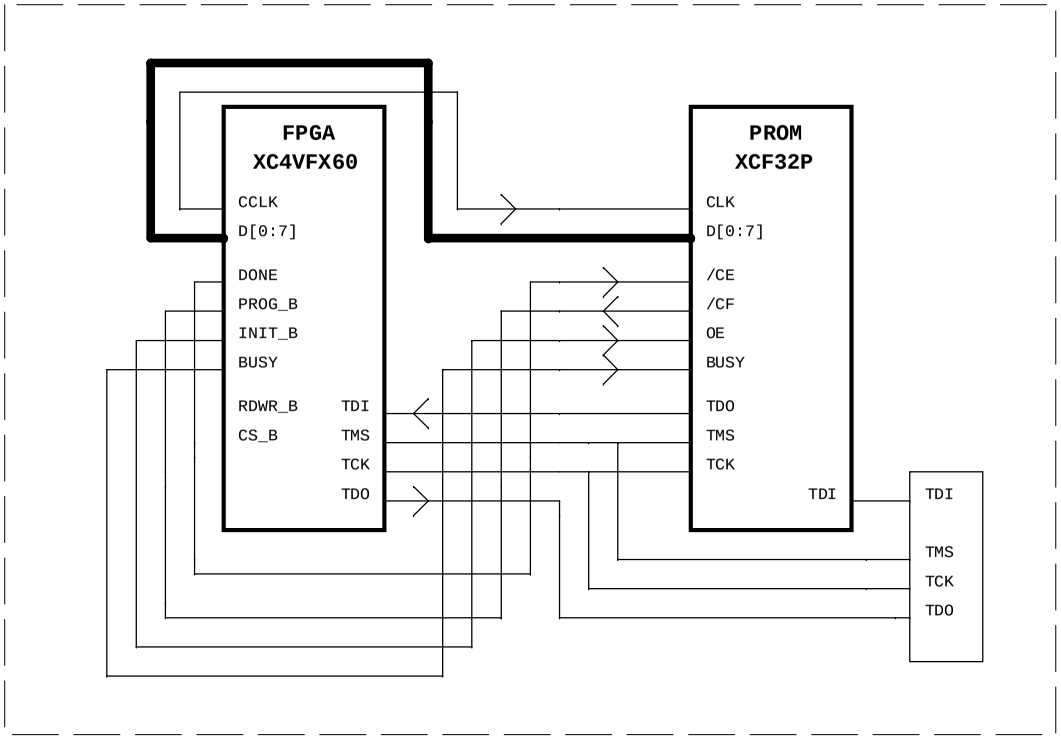
Title	FPGA_CLK_MDR_CON	
Size B	Document Number CameraSimulator_v1.0	Rev A
Date: Friday, April 13, 2012	Sheet 2 of 3	



Title	POWER_Decoupling	
Size B	Document Number CameraSimulator_v1.0	Rev A
Date: Friday, April 13, 2012	Sheet 3 of 3	

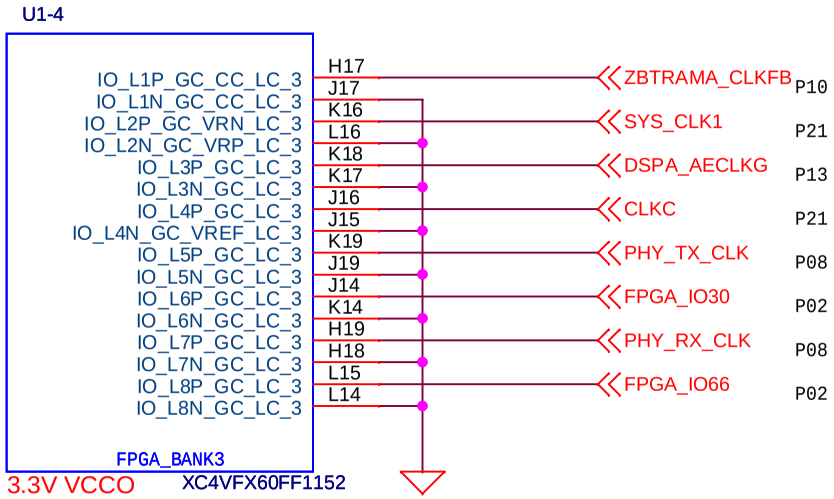
A.5. PowerEye Board

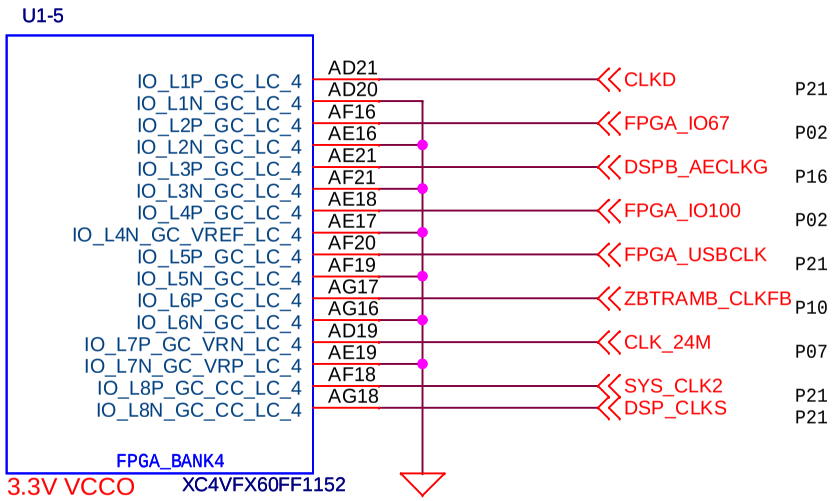
Schematic Contents



FPGA CONFIGURATION AND JTAG CHAIN

Title: <Title>		
Size: B	Document Number: <Doc>	Rev: 1.0
Date: Wednesday, April 11, 2012	Sheet 0 of 0	





FPGA GLOBAL CLOCK DISTRIBUTION

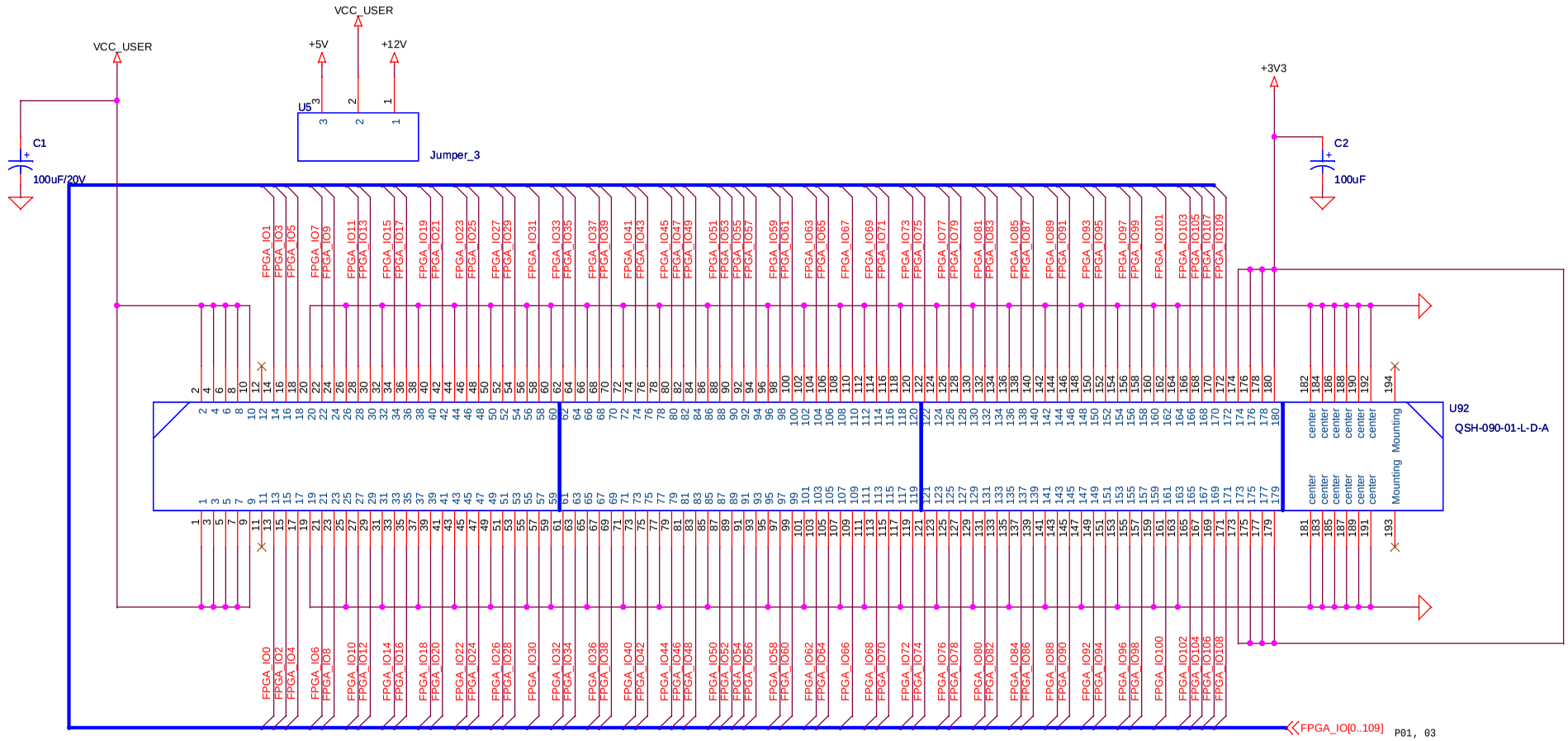
TOP:

1. ZBTRAMA_CLKFB(FEEDBACK FROM ZBTRAMA)
2. FPGA_IO30(FROM USERIO CONNECTOR)
3. FPGA_IO66(FROM USERIO CONNECTOR)
4. DSPA_AECLKG(FROM DSPA ECLKOUT2)
5. PHY_RX_CLK(FROM GbE PHY)
6. PHY_TX_CLK(FROM GbE PHY)
7. CLKC(FROM PROGRAMMABLE CLOCK PLL)
8. SYS_CLK1(FROM EXTERNAL OSCILLATOR)

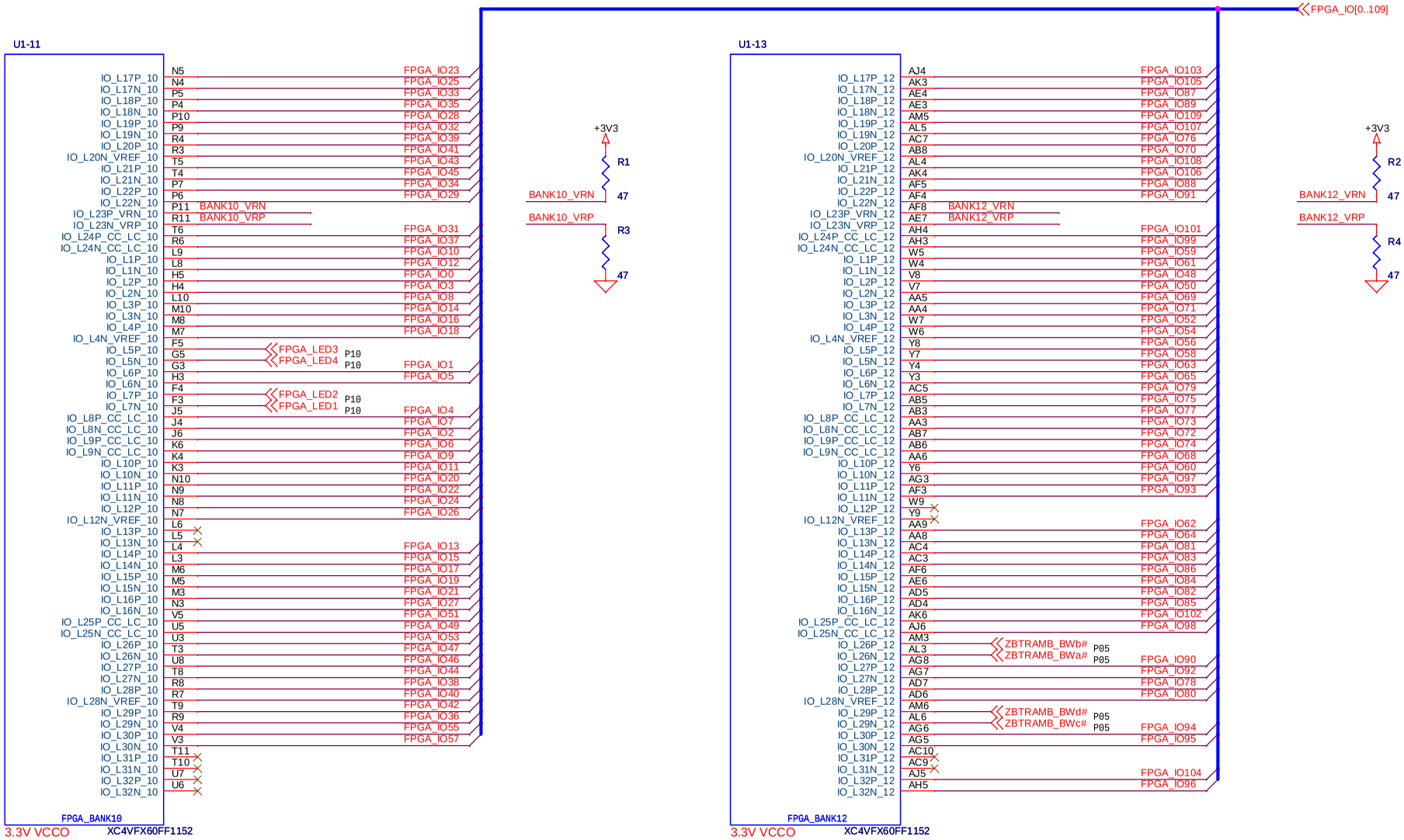
BOTTOM:

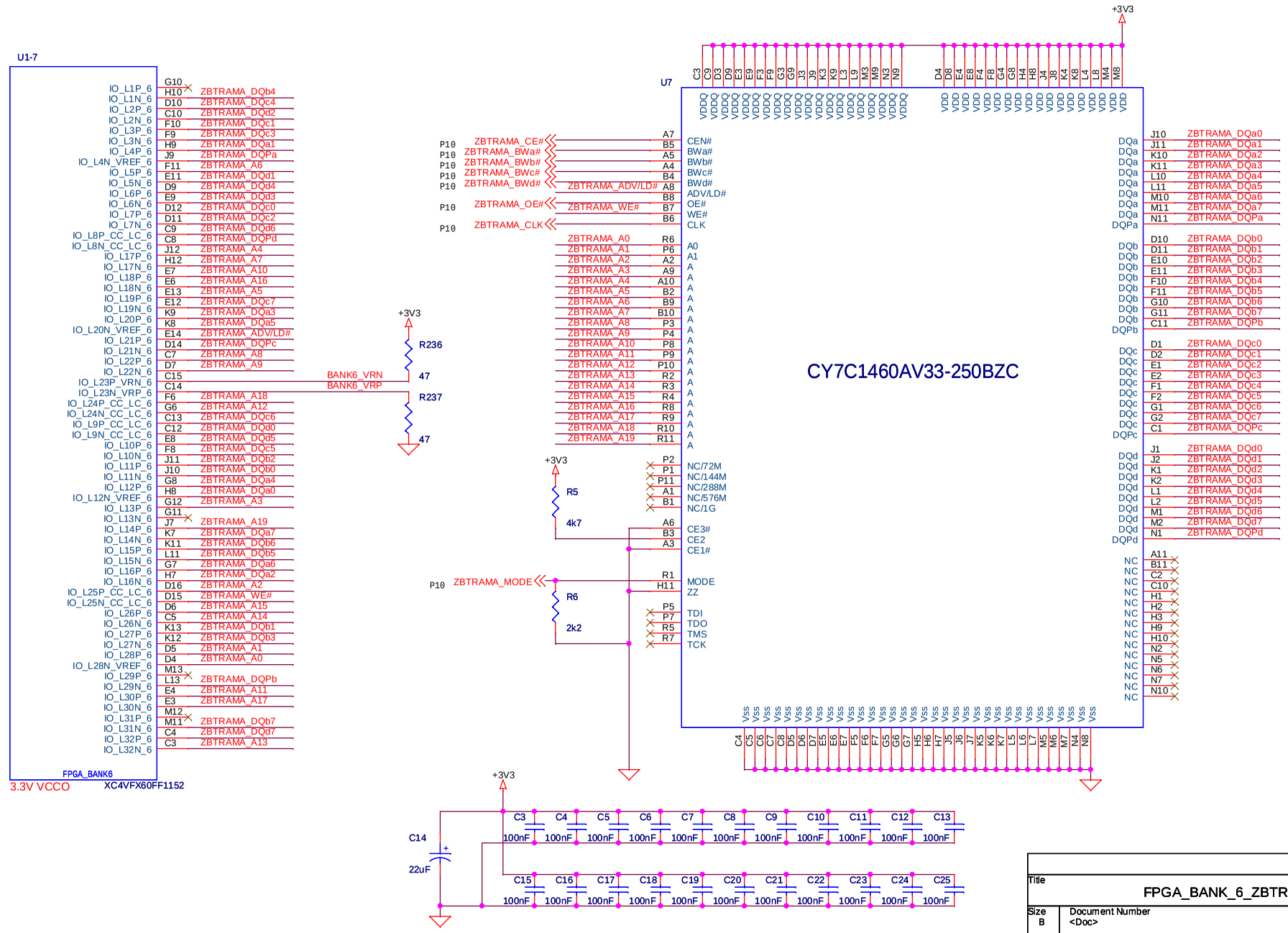
1. ZBTRAMB_CLK(FEEDBACK FROM ZBTRAMB)
2. FPGA_IO67(FROM USERIO CONNECTOR)
3. FPGA_IO100(FROM USERIO CONNECTOR)
4. DSPB_AECLKG(FROM DSPB ECLKOUT2)
5. FPGA_USBCLK(FROM PROGRAMMABLE CLOCK PLL)
6. SYS_CLK2(FROM EXTERNAL OSCILLATOR)
7. CLKD(FROM PROGRAMMABLE CLOCK PLL)
8. CLK_24M(FROM FX2 CLKOUT)

Title: FPGA_BANK_3_4_GCLK		
Size: A	Document Number: <Doc>	Rev: 1.0
Date: Wednesday, April 11, 2012	Sheet 1 of 22	

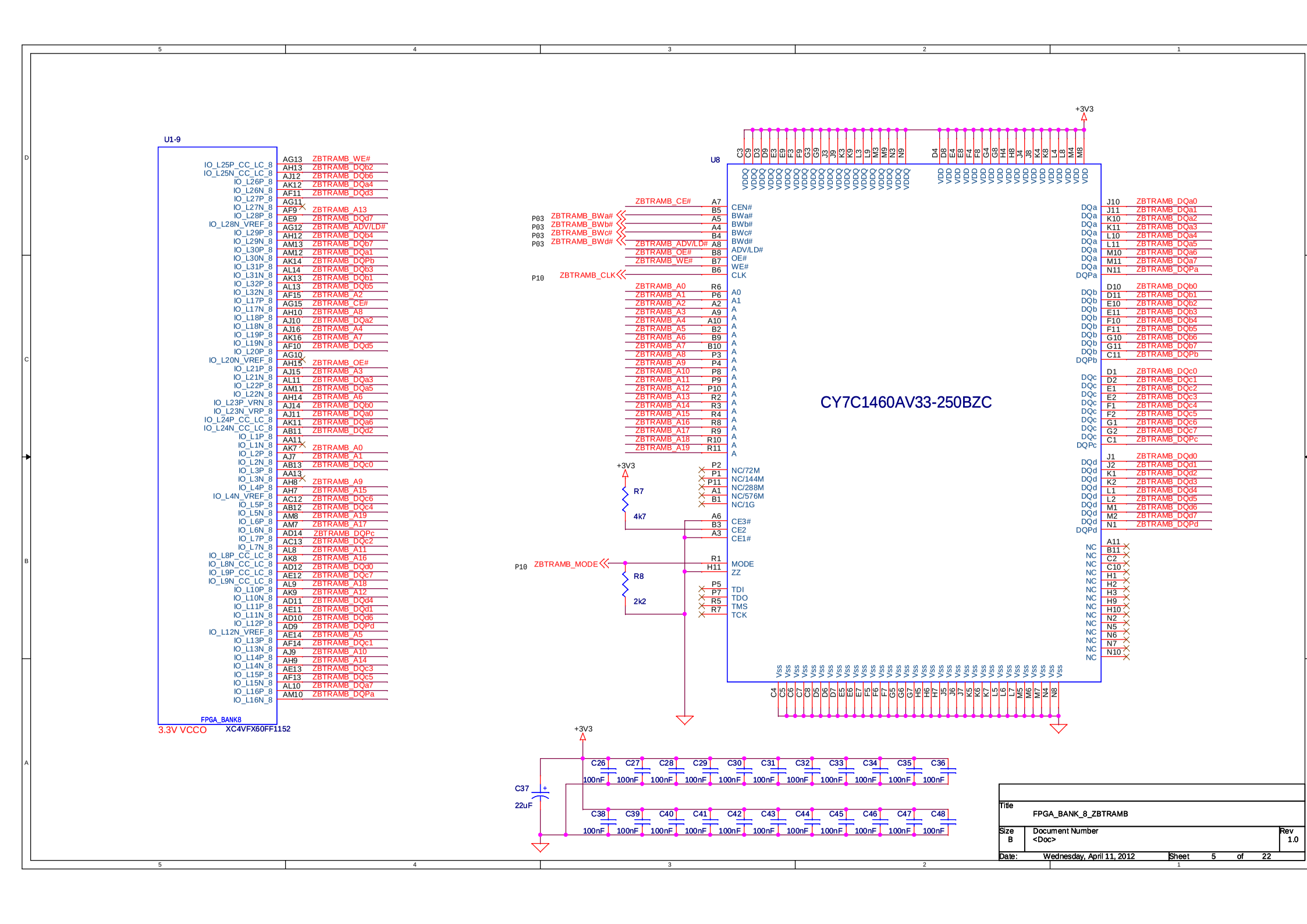


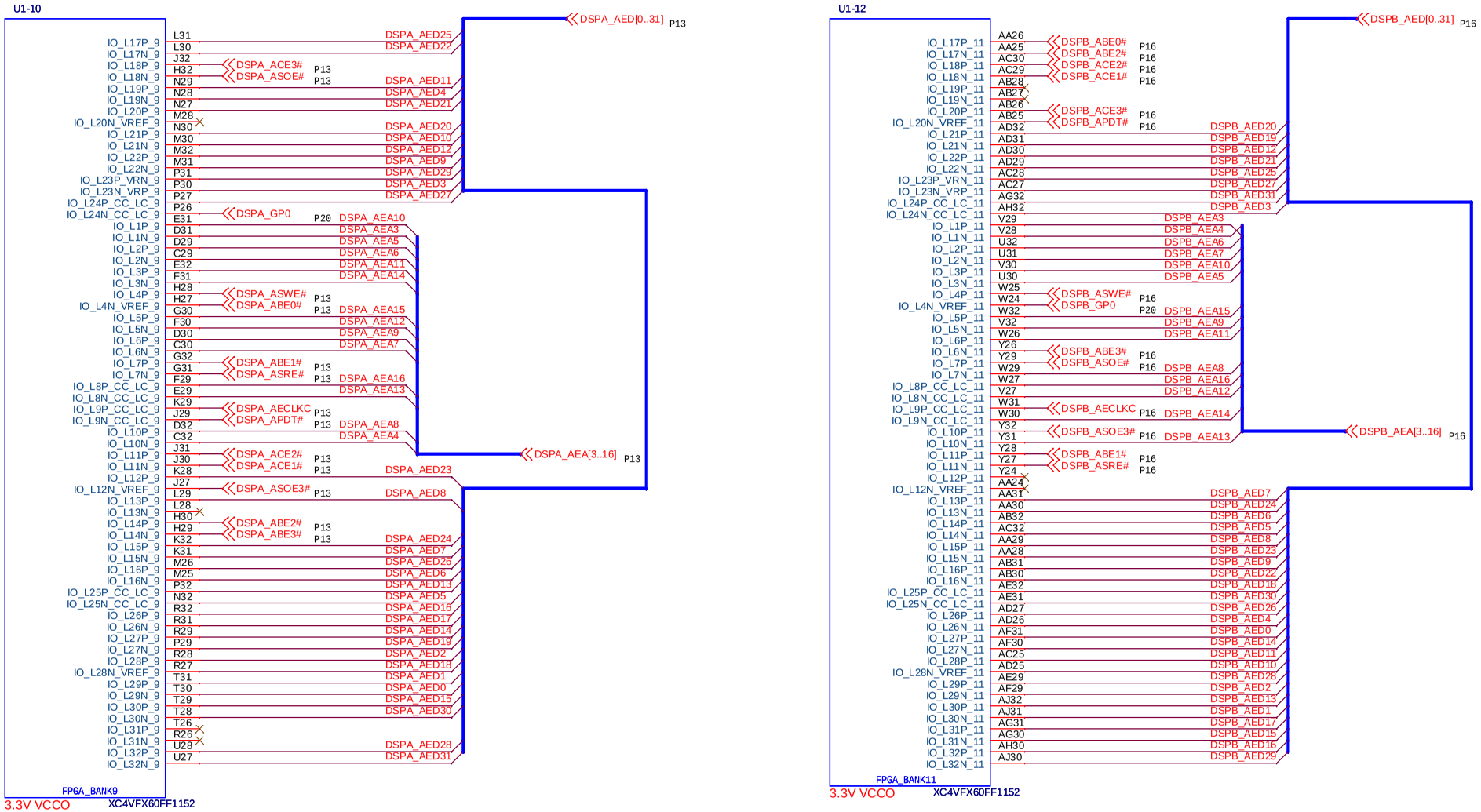
Title	FPGA_USERIO_CONNECTOR	
Size B	Document Number <Doc>	Rev 1.0
Date: Wednesday, April 11, 2012	Sheet 2 of 22	


U1-11
IO_L17P_10
IO_L17N_10
IO_L18P_10
IO_L18N_10
IO_L19P_10
IO_L19N_10
IO_L20P_10
IO_L20N_VREF_10
IO_L21P_10
IO_L21N_10
IO_L22P_10
IO_L22N_10
IO_L23P_VRN_10
IO_L23N_VRP_10
IO_L24P_CC_LC_10
IO_L24N_CC_LC_10
IO_L1P_10
IO_L1N_10
IO_L2P_10
IO_L2N_10
IO_L3P_10
IO_L3N_10
IO_L4P_10
IO_L4N_VREF_10
IO_L5P_10
IO_L5N_10
IO_L6P_10
IO_L6N_10
IO_L7P_10
IO_L7N_10
IO_L8P_CC_LC_10
IO_L8N_CC_LC_10
IO_L9P_CC_LC_10
IO_L9N_CC_LC_10
IO_L10P_10
IO_L10N_10
IO_L11P_10
IO_L11N_10
IO_L12P_10
IO_L12N_VREF_10
IO_L13P_10
IO_L13N_10
IO_L14P_10
IO_L14N_10
IO_L15P_10
IO_L15N_10
IO_L16P_10
IO_L16N_10
IO_L25P_CC_LC_10
IO_L25N_CC_LC_10
IO_L26P_10
IO_L26N_10
IO_L27P_10
IO_L27N_10
IO_L28P_10
IO_L28N_VREF_10
IO_L29P_10
IO_L29N_10
IO_L30P_10
IO_L30N_10
IO_L31P_10
IO_L31N_10
IO_L32P_10
IO_L32N_10
N5 FPGA_IO23
N4 FPGA_IO25
P5 FPGA_IO33
P4 FPGA_IO35
P10 FPGA_IO28
P9 FPGA_IO32
R4 FPGA_IO39
R3 FPGA_IO41
T5 FPGA_IO43
T4 FPGA_IO45
P7 FPGA_IO34
P6 FPGA_IO29
P11 BANK10_VRN
R11 BANK10_VRP
T6 FPGA_IO31
R6 FPGA_IO37
L9 FPGA_IO10
L8 FPGA_IO12
H5 FPGA_IO0
H4 FPGA_IO3
L10 FPGA_IO8
M10 FPGA_IO14
M8 FPGA_IO16
M7 FPGA_IO18
F5
G5 FPGA_LED3 P10
G3 FPGA_LED4 P10 FPGA_IO1
H3 FPGA_IO5
F4
F3 FPGA_LED2 P10
J5 FPGA_LED1 P10 FPGA_IO4
J4 FPGA_IO7
J6 FPGA_IO2
K6 FPGA_IO6
K4 FPGA_IO9
K3 FPGA_IO11
N10 FPGA_IO20
N9 FPGA_IO22
N8 FPGA_IO24
N7 FPGA_IO26
L6
L5
L4 FPGA_IO13
L3 FPGA_IO15
M6 FPGA_IO17
M5 FPGA_IO19
M3 FPGA_IO21
N3 FPGA_IO27
V5 FPGA_IO51
U5 FPGA_IO49
U3 FPGA_IO53
T3 FPGA_IO47
U8 FPGA_IO46
T8 FPGA_IO44
R8 FPGA_IO38
R7 FPGA_IO40
T9 FPGA_IO42
R9 FPGA_IO36
V4 FPGA_IO55
V3 FPGA_IO57
T11
T10
U7
U6
FPGA_BANK10
3.3V VCCO
XC4VFX60FF1152
+3V3
R1
BANK10_VRN
47
BANK10_VRP
R3
47
FPGA_IO[0..109]
U1-13
IO_L17P_12
IO_L17N_12
IO_L18P_12
IO_L18N_12
IO_L19P_12
IO_L19N_12
IO_L20P_12
IO_L20N_VREF_12
IO_L21P_12
IO_L21N_12
IO_L22P_12
IO_L22N_12
IO_L23P_VRN_12
IO_L23N_VRP_12
IO_L24P_CC_LC_12
IO_L24N_CC_LC_12
IO_L1P_12
IO_L1N_12
IO_L2P_12
IO_L2N_12
IO_L3P_12
IO_L3N_12
IO_L4P_12
IO_L4N_VREF_12
IO_L5P_12
IO_L5N_12
IO_L6P_12
IO_L6N_12
IO_L7P_12
IO_L7N_12
IO_L8P_CC_LC_12
IO_L8N_CC_LC_12
IO_L9P_CC_LC_12
IO_L9N_CC_LC_12
IO_L10P_12
IO_L10N_12
IO_L11P_12
IO_L11N_12
IO_L12P_12
IO_L12N_VREF_12
IO_L13P_12
IO_L13N_12
IO_L14P_12
IO_L14N_12
IO_L15P_12
IO_L15N_12
IO_L16P_12
IO_L16N_12
IO_L25P_CC_LC_12
IO_L25N_CC_LC_12
IO_L26P_12
IO_L26N_12
IO_L27P_12
IO_L27N_12
IO_L28P_12
IO_L28N_VREF_12
IO_L29P_12
IO_L29N_12
IO_L30P_12
IO_L30N_12
IO_L31P_12
IO_L31N_12
IO_L32P_12
IO_L32N_12
AJ4 FPGA_IO103
AK3 FPGA_IO105
AE4 FPGA_IO87
AE3 FPGA_IO89
AM5 FPGA_IO109
AL5 FPGA_IO107
AC7 FPGA_IO76
AB8 FPGA_IO70
AL4 FPGA_IO108
AK4 FPGA_IO106
AF5 FPGA_IO88
AF4 FPGA_IO91
AF8 BANK12_VRN
AE7 BANK12_VRP
AH4 FPGA_IO101
AH3 FPGA_IO99
W5 FPGA_IO59
W4 FPGA_IO61
V8 FPGA_IO48
V7 FPGA_IO50
AA5 FPGA_IO69
AA4 FPGA_IO71
W7 FPGA_IO52
W6 FPGA_IO54
Y8 FPGA_IO56
Y7 FPGA_IO58
Y4 FPGA_IO63
Y3 FPGA_IO65
AC5 FPGA_IO79
AB5 FPGA_IO75
AB3 FPGA_IO77
AA3 FPGA_IO73
AB7 FPGA_IO72
AB6 FPGA_IO74
AA6 FPGA_IO68
Y6 FPGA_IO60
AG3 FPGA_IO97
AF3 FPGA_IO93
W9
Y9
AA9 FPGA_IO62
AA8 FPGA_IO64
AC4 FPGA_IO81
AC3 FPGA_IO83
AF6 FPGA_IO86
AE6 FPGA_IO84
AD5 FPGA_IO82
AD4 FPGA_IO85
AK6 FPGA_IO102
AJ6 FPGA_IO98
AM3
AL3 ZBTRAMB_BWb# P05
AG8 ZBTRAMB_BWa# P05 FPGA_IO90
AG7 FPGA_IO92
AD7 FPGA_IO78
AD6 FPGA_IO80
AM6
AL6 ZBTRAMB_BWd# P05
AG6 ZBTRAMB_BWc# P05 FPGA_IO94
AG5 FPGA_IO95
AC10
AC9
AJ5 FPGA_IO104
AH5 FPGA_IO96
FPGA_BANK12
3.3V VCCO
XC4VFX60FF1152
+3V3
R2
BANK12_VRN
47
BANK12_VRP
R4
47
Title
FPGA_BANK_10_12_USERIO
Size B
Document Number <Doc>
Rev 1.0
Date: Wednesday, April 11, 2012
Sheet 3 of 22

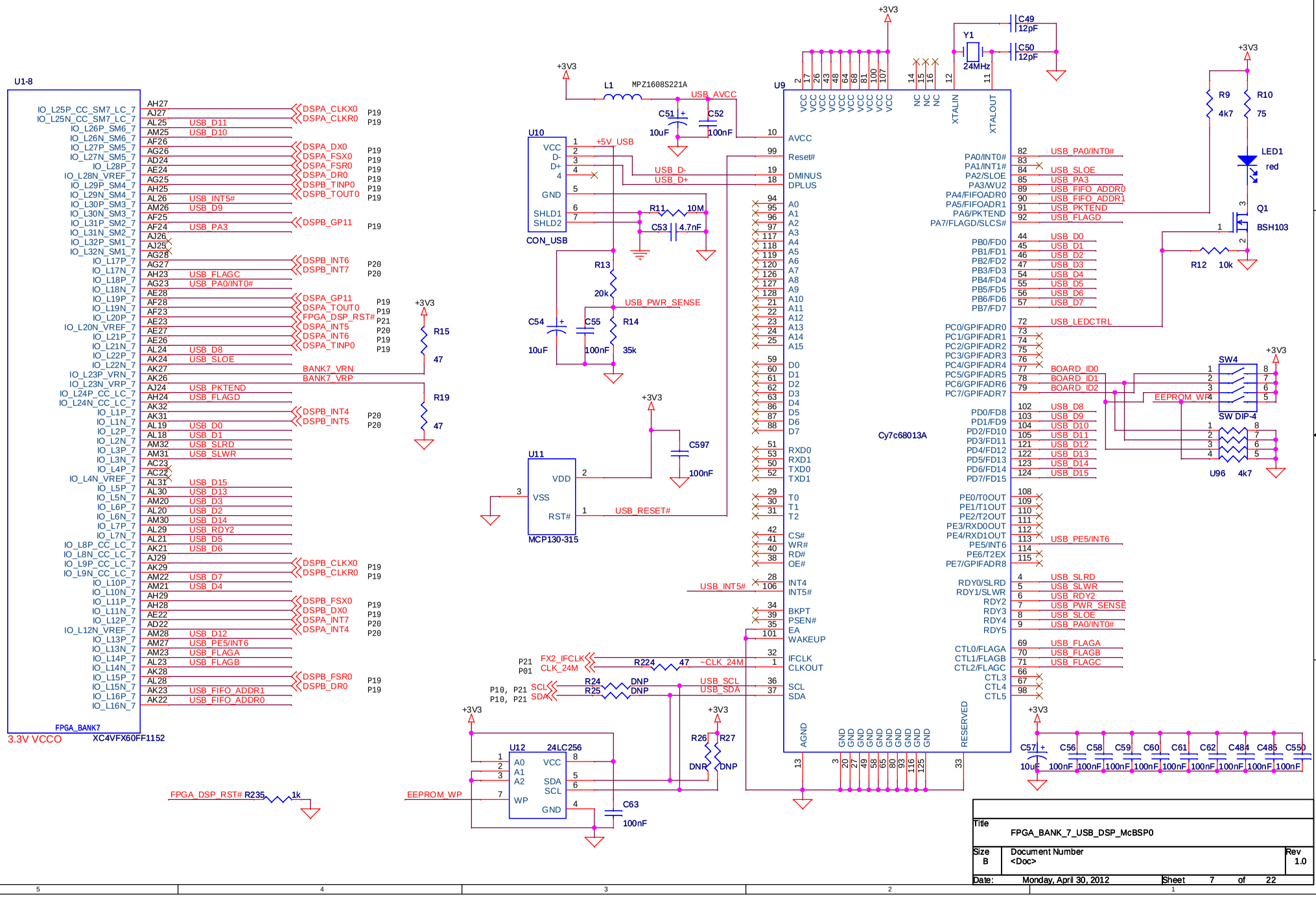


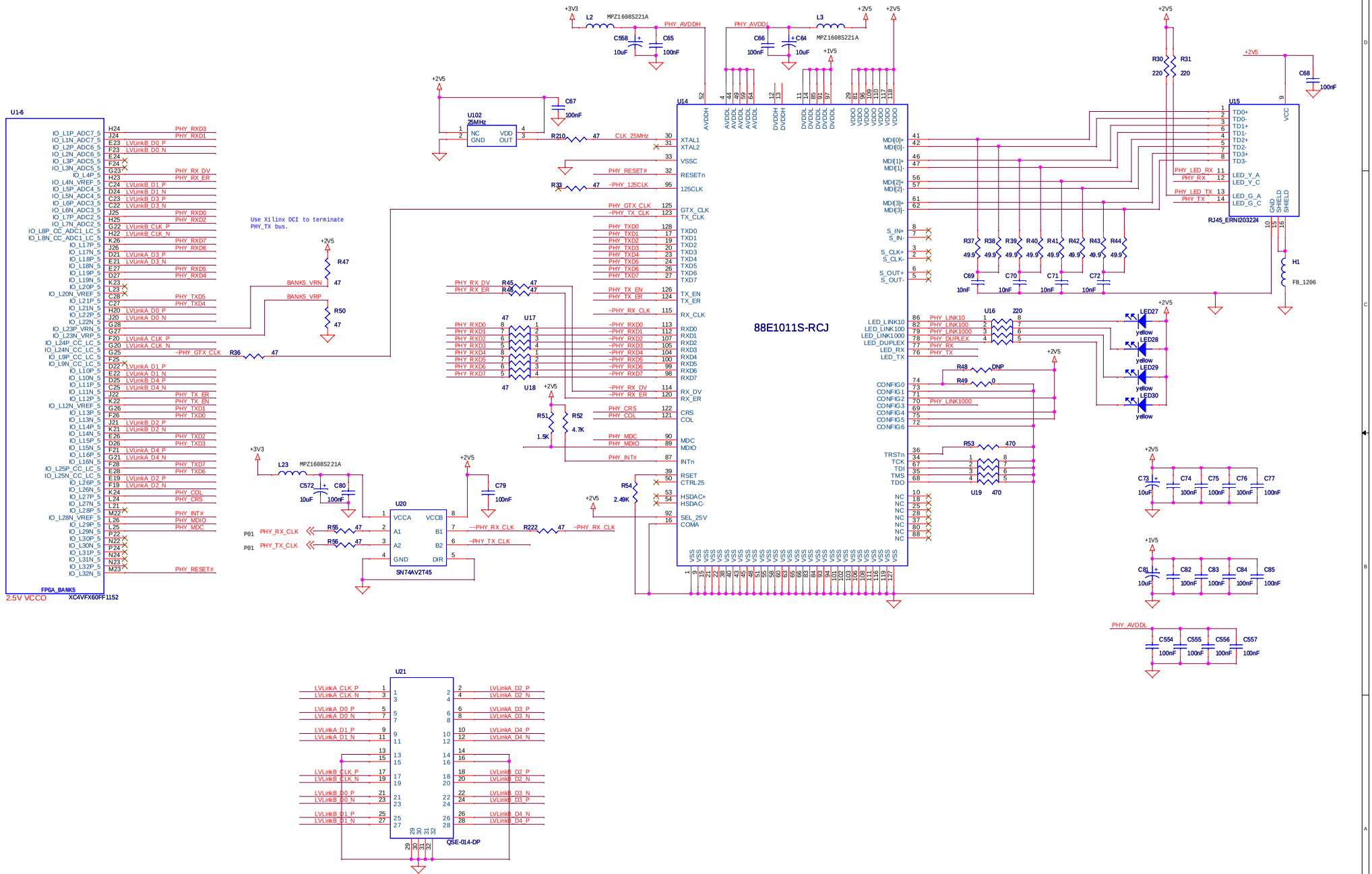
Title	FPGA_BANK_6_ZBTRAMA	
Size B	Document Number <Doc>	Rev 1.0
Date: Wednesday, April 11, 2012	Sheet 4 of 22	


U1-9
IO_L25P_CC_LC_8 AG13 ZBTRAMB_WE#
IO_L25N_CC_LC_8 AH13 ZBTRAMB_DQb2
IO_L26P_8 AJ12 ZBTRAMB_DQb6
IO_L26N_8 AK12 ZBTRAMB_DQa4
IO_L27P_8 AF11 ZBTRAMB_DQd3
IO_L27N_8 AG11
IO_L28P_8 AF9 ZBTRAMB_A13
IO_L28N_VREF_8 AE9 ZBTRAMB_DQd7
IO_L29P_8 AG12 ZBTRAMB_ADV/LD#
IO_L29N_8 AH12 ZBTRAMB_DQb4
IO_L30P_8 AM13 ZBTRAMB_DQb7
IO_L30N_8 AM12 ZBTRAMB_DQa1
IO_L31P_8 AK14 ZBTRAMB_DQPb
IO_L31N_8 AL14 ZBTRAMB_DQb3
IO_L32P_8 AK13 ZBTRAMB_DQb1
IO_L32N_8 AL13 ZBTRAMB_DQb5
IO_L17P_8 AF15 ZBTRAMB_A2
IO_L17N_8 AG15 ZBTRAMB_CE#
IO_L18P_8 AH10 ZBTRAMB_A8
IO_L18N_8 AJ10 ZBTRAMB_DQa2
IO_L19P_8 AJ16 ZBTRAMB_A4
IO_L19N_8 AK16 ZBTRAMB_A7
IO_L20P_8 AF10 ZBTRAMB_DQd5
IO_L20N_VREF_8 AG10
IO_L21P_8 AH15 ZBTRAMB_OE#
IO_L21N_8 AJ15 ZBTRAMB_A3
IO_L22P_8 AL11 ZBTRAMB_DQa3
IO_L22N_8 AM11 ZBTRAMB_DQa5
IO_L23P_VRN_8 AH14 ZBTRAMB_A6
IO_L23N_VRP_8 AJ14 ZBTRAMB_DQb0
IO_L24P_CC_LC_8 AJ11 ZBTRAMB_DQa0
IO_L24N_CC_LC_8 AK11 ZBTRAMB_DQa6
IO_L1P_8 AB11 ZBTRAMB_DQd2
IO_L1N_8 AA11
IO_L2P_8 AK7 ZBTRAMB_A0
IO_L2N_8 AJ7 ZBTRAMB_A1
IO_L3P_8 AB13 ZBTRAMB_DQc0
IO_L3N_8 AA13
IO_L4P_8 AH8 ZBTRAMB_A9
IO_L4N_VREF_8 AH7 ZBTRAMB_A15
IO_L5P_8 AC12 ZBTRAMB_DQc6
IO_L5N_8 AB12 ZBTRAMB_DQc4
IO_L6P_8 AM8 ZBTRAMB_A19
IO_L6N_8 AM7 ZBTRAMB_A17
IO_L7P_8 AD14 ZBTRAMB_DQPc
IO_L7N_8 AC13 ZBTRAMB_DQc2
IO_L8P_CC_LC_8 AL8 ZBTRAMB_A11
IO_L8N_CC_LC_8 AK8 ZBTRAMB_A16
IO_L9P_CC_LC_8 AD12 ZBTRAMB_DQd0
IO_L9N_CC_LC_8 AE12 ZBTRAMB_DQc7
IO_L10P_8 AL9 ZBTRAMB_A18
IO_L10N_8 AK9 ZBTRAMB_A12
IO_L11P_8 AD11 ZBTRAMB_DQd4
IO_L11N_8 AE11 ZBTRAMB_DQd1
IO_L12P_8 AD10 ZBTRAMB_DQd6
IO_L12N_VREF_8 AD9 ZBTRAMB_DQPd
IO_L13P_8 AE14 ZBTRAMB_A5
IO_L13N_8 AF14 ZBTRAMB_DQc1
IO_L14P_8 AJ9 ZBTRAMB_A10
IO_L14N_8 AH9 ZBTRAMB_A14
IO_L15P_8 AE13 ZBTRAMB_DQc3
IO_L15N_8 AF13 ZBTRAMB_DQc5
IO_L16P_8 AL10 ZBTRAMB_DQa7
IO_L16N_8 AM10 ZBTRAMB_DQPa
FPGA_BANK8
3.3V VCCO
XC4VFX60FF1152
U8
+3V3
CY7C1460AV33-250BZC
ZBTRAMB_CE# A7 CEN#
P03 ZBTRAMB_BWa# B5 BWa#
P03 ZBTRAMB_BWb# A5 BWb#
P03 ZBTRAMB_BWc# A4 BWc#
P03 ZBTRAMB_BWd# B4 BWd#
ZBTRAMB_ADV/LD# A8 ADV/LD#
ZBTRAMB_OE# B8 OE#
ZBTRAMB_WE# B7 WE#
P10 ZBTRAMB_CLK B6 CLK
ZBTRAMB_A0 R6 A0
ZBTRAMB_A1 P6 A1
ZBTRAMB_A2 A2 A
ZBTRAMB_A3 A9 A
ZBTRAMB_A4 A10 A
ZBTRAMB_A5 B2 A
ZBTRAMB_A6 B9 A
ZBTRAMB_A7 B10 A
ZBTRAMB_A8 P3 A
ZBTRAMB_A9 P4 A
ZBTRAMB_A10 P8 A
ZBTRAMB_A11 P9 A
ZBTRAMB_A12 P10 A
ZBTRAMB_A13 R2 A
ZBTRAMB_A14 R3 A
ZBTRAMB_A15 R4 A
ZBTRAMB_A16 R8 A
ZBTRAMB_A17 R9 A
ZBTRAMB_A18 R10 A
ZBTRAMB_A19 R11 A
P2 NC/72M
P1 NC/144M
P11 NC/288M
A1 NC/576M
B1 NC/1G
+3V3
R7
4k7
A6 CE3#
B3 CE2
A3 CE1#
P10 ZBTRAMB_MODE
R8
2k2
R1 MODE
H11 ZZ
P5 TDI
P7 TDO
R5 TMS
R7 TCK
J10 ZBTRAMB_DQa0
J11 ZBTRAMB_DQa1
K10 ZBTRAMB_DQa2
K11 ZBTRAMB_DQa3
L10 ZBTRAMB_DQa4
L11 ZBTRAMB_DQa5
M10 ZBTRAMB_DQa6
M11 ZBTRAMB_DQa7
N11 ZBTRAMB_DQPa
D10 ZBTRAMB_DQb0
D11 ZBTRAMB_DQb1
E10 ZBTRAMB_DQb2
E11 ZBTRAMB_DQb3
F10 ZBTRAMB_DQb4
F11 ZBTRAMB_DQb5
G10 ZBTRAMB_DQb6
G11 ZBTRAMB_DQb7
C11 ZBTRAMB_DQPb
D1 ZBTRAMB_DQc0
D2 ZBTRAMB_DQc1
E1 ZBTRAMB_DQc2
E2 ZBTRAMB_DQc3
F1 ZBTRAMB_DQc4
F2 ZBTRAMB_DQc5
G1 ZBTRAMB_DQc6
G2 ZBTRAMB_DQc7
C1 ZBTRAMB_DQPc
J1 ZBTRAMB_DQd0
J2 ZBTRAMB_DQd1
K1 ZBTRAMB_DQd2
K2 ZBTRAMB_DQd3
L1 ZBTRAMB_DQd4
L2 ZBTRAMB_DQd5
M1 ZBTRAMB_DQd6
M2 ZBTRAMB_DQd7
N1 ZBTRAMB_DQPd
A11 B11 C2 C10 H1 H2 H3 H9 H10 N2 N5 N6 N7 N10
+3V3
C37 22uF
C26 C27 C28 C29 C30 C31 C32 C33 C34 C35 C36 100nF
C38 C39 C40 C41 C42 C43 C44 C45 C46 C47 C48 100nF
Title FPGA_BANK_8_ZBTRAMB
Size B
Document Number <Doc>
Rev 1.0
Date: Wednesday, April 11, 2012
Sheet 5 of 22

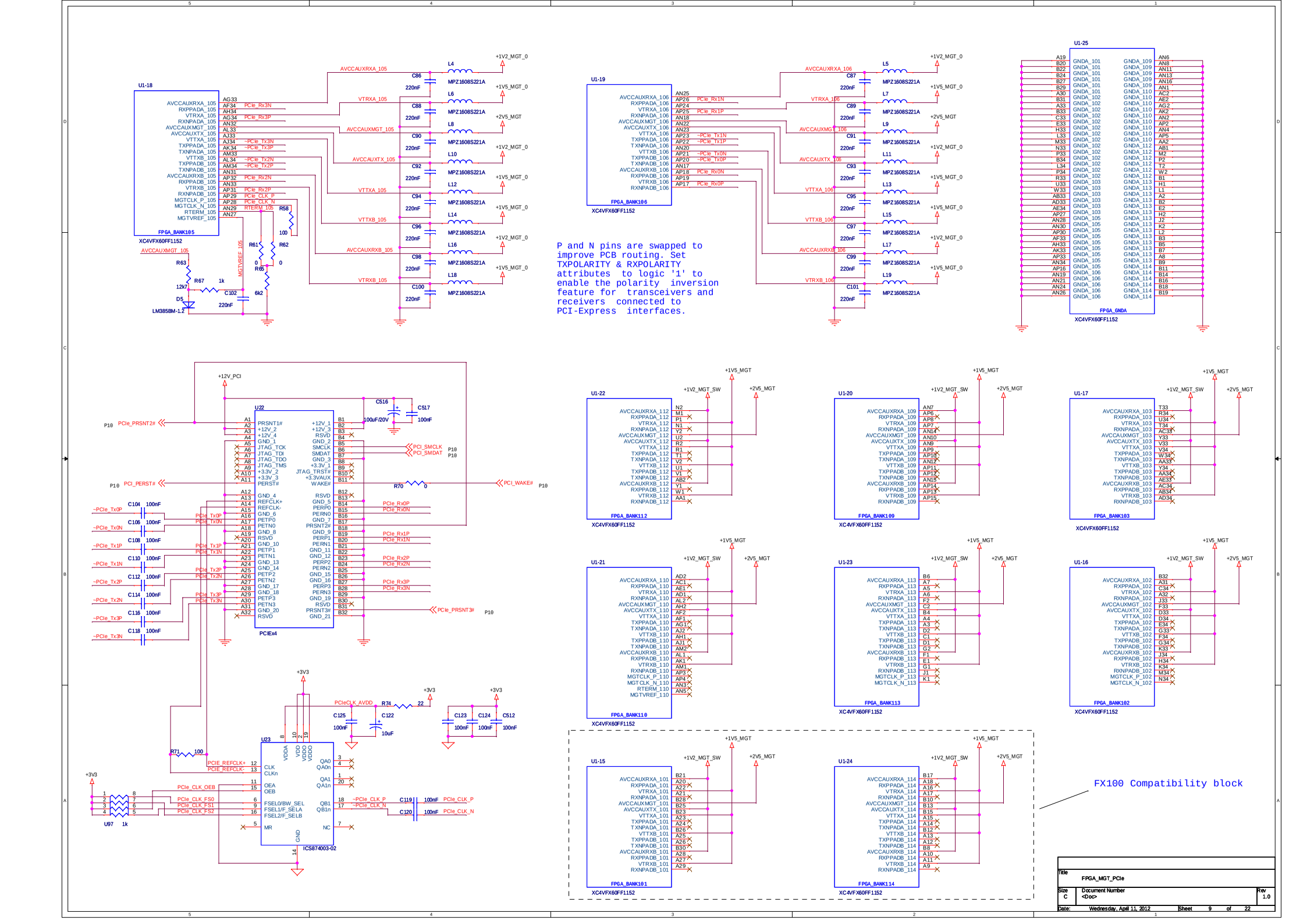


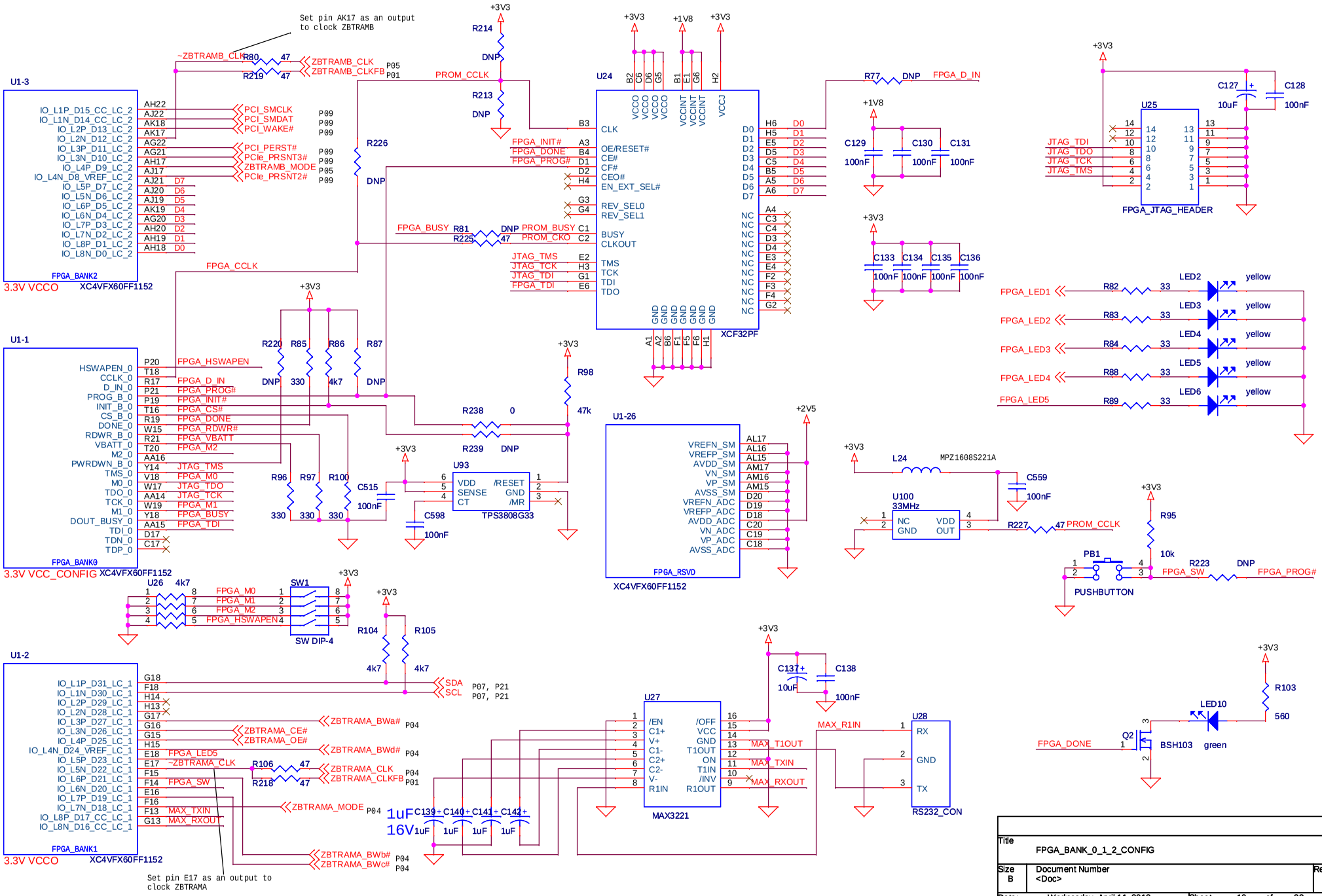
Title	FPGA_BANK_9_11_DSP_EMIFA	
Size B	Document Number <Doc>	Rev 1.0
Date: Wednesday, April 11, 2012	Sheet 6 of 22	


U1-8
FPGA_BANK7
3.3V VCCO
XC4VFX60FF1152
U9
Cy7c68013A
U10
CON_USB
U11
MCP130-315
U12
24LC256
L1
MPZ1608S221A
Y1
24MHz
SW4
SW DIP-4
LED1
red
Q1
BSH103
+3V3
USB_AVCC
USB_PWR_SENSE
USB_RESET#
EEPROM_WP
FPGA_DSP_RST#
Title
FPGA_BANK_7_USB_DSP_McBSP0
Size
B
Document Number
<Doc>
Rev
1.0
Date:
Monday, April 30, 2012
Sheet
7
of
22

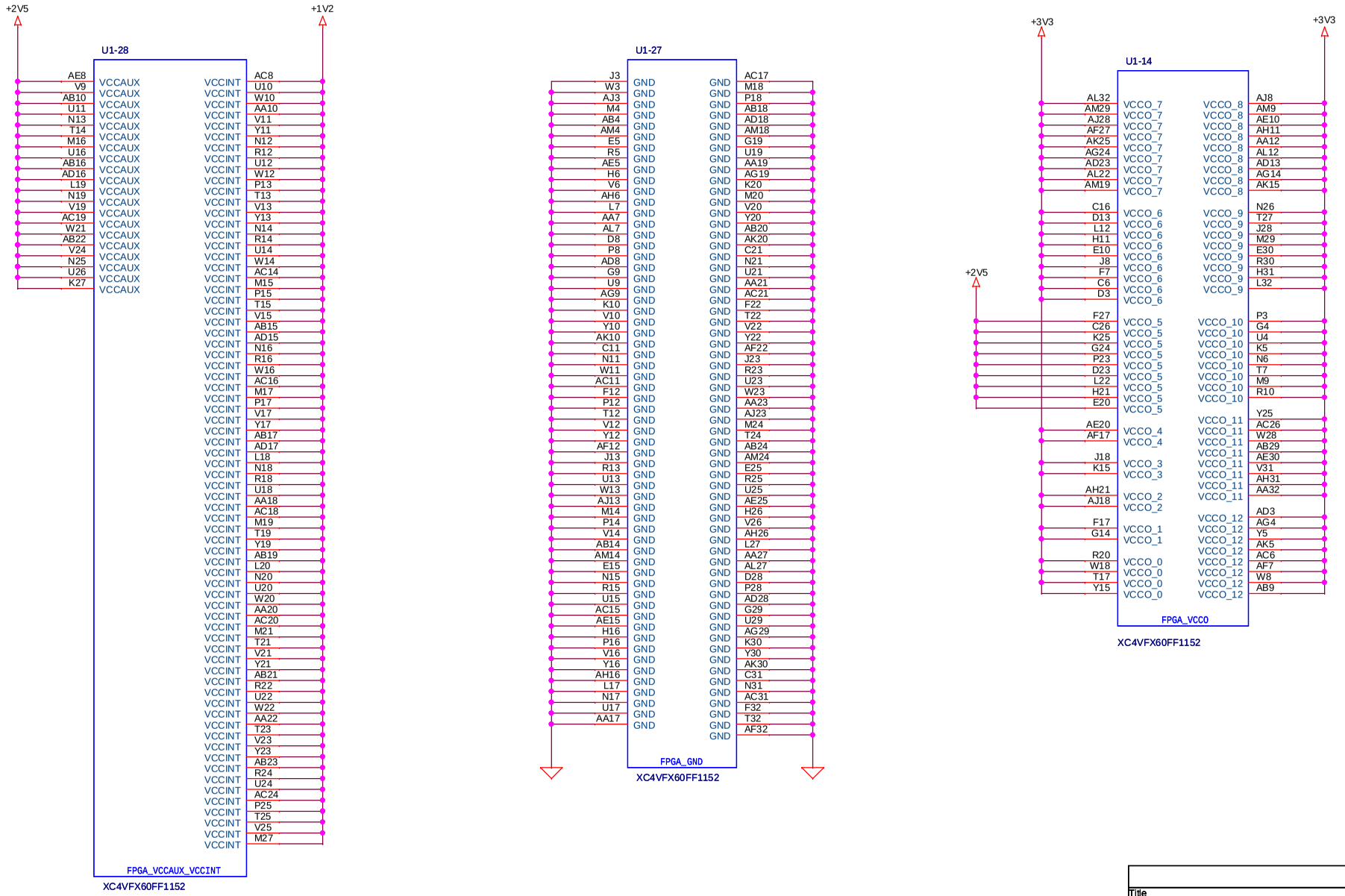


Title	FPGA_BANK_5_GbE_LVDSLink		
Size C	Document Number <Doc>		Rev 1.0
Date: Monday, April 30, 2012	Sheet 8 of 22		

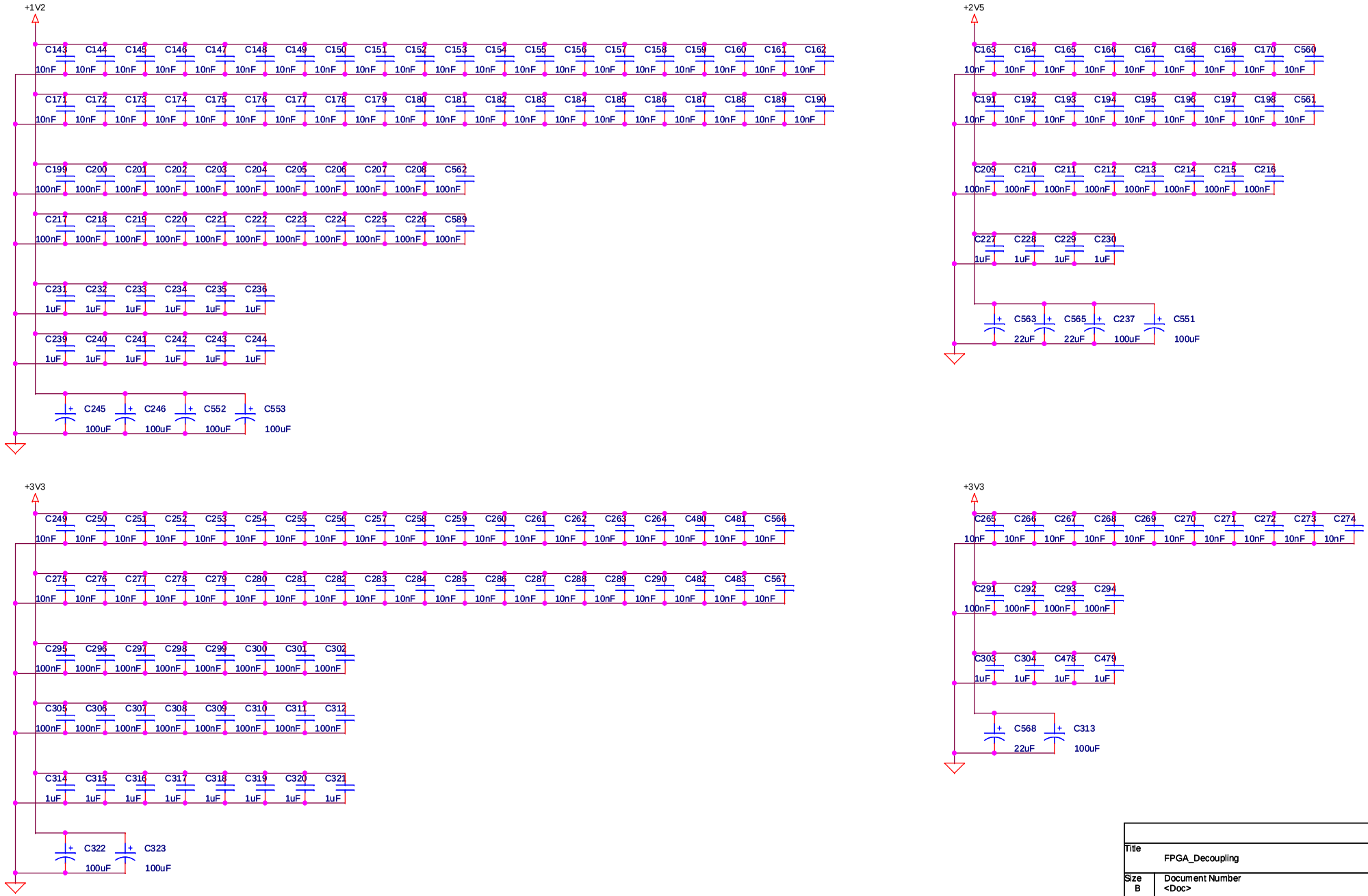

U1-18
FPGA_BANK105
XC4VFX60FF1152
U1-19
FPGA_BANK106
XC4VFX60FF1152
U1-25
FPGA_GNDA
XC4VFX60FF1152
P and N pins are swapped to improve PCB routing. Set TXPOLARITY & RXPOLARITY attributes to logic '1' to enable the polarity inversion feature for transceivers and receivers connected to PCI-Express interfaces.
U22
PCIEx4
U23
ICS874003-02
U97
U1-22
FPGA_BANK112
U1-20
FPGA_BANK109
U1-17
FPGA_BANK103
U1-21
FPGA_BANK110
U1-23
FPGA_BANK113
U1-16
FPGA_BANK102
U1-15
FPGA_BANK101
U1-24
FPGA_BANK114
FX100 Compatibility block
Title
FPGA_MGT_PCIe
Size C
Document Number <Doc>
Rev 1.0
Date: Wednesday, April 11, 2012
Sheet 9 of 22

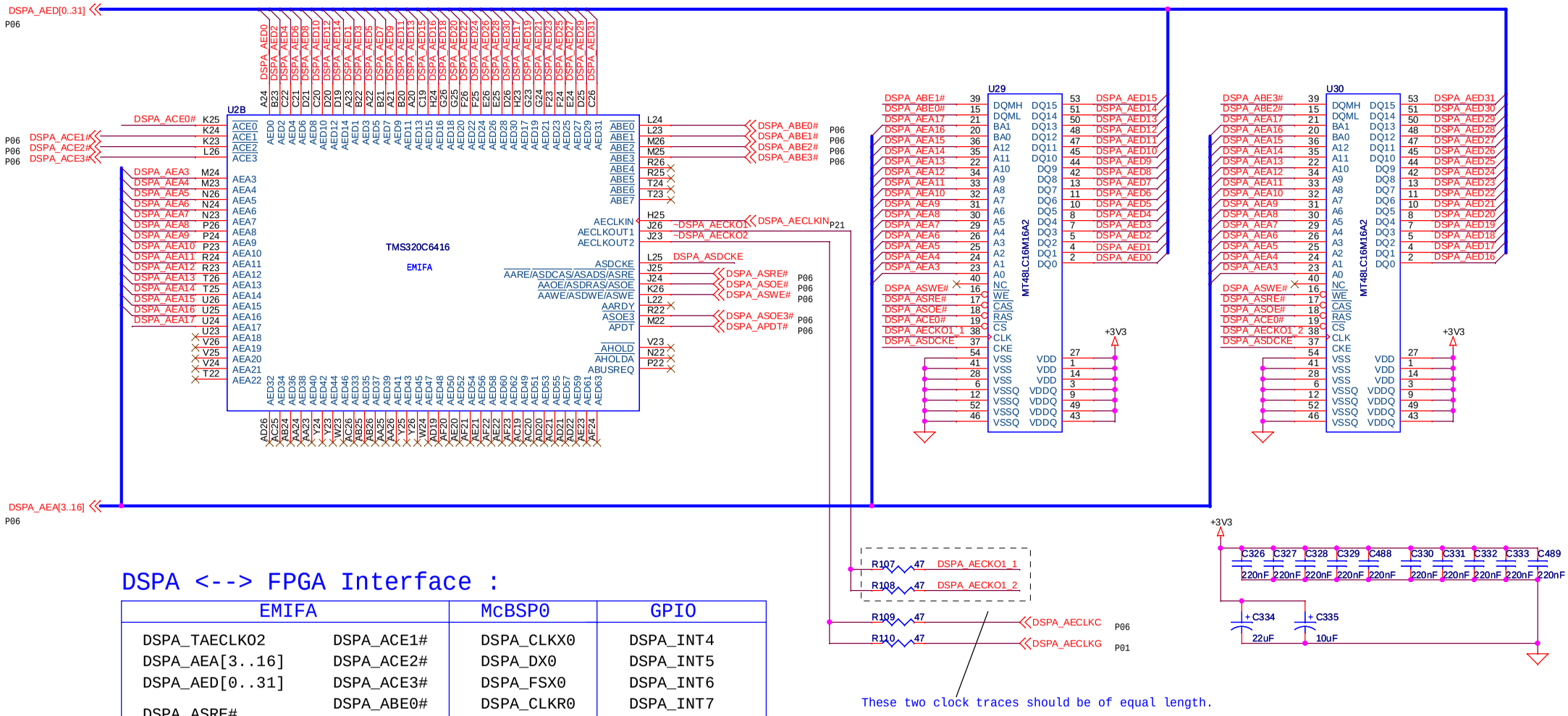


Title	FPGA_BANK_0_1_2_CONFIG	
Size B	Document Number <Doc>	Rev 1.0
Date: Wednesday, April 11, 2012	Sheet 10 of 22	



Title	FPGA_PWR_GND	
Size B	Document Number <Doc>	Rev 1.0
Date: Wednesday, April 11, 2012	Sheet 11 of 22	

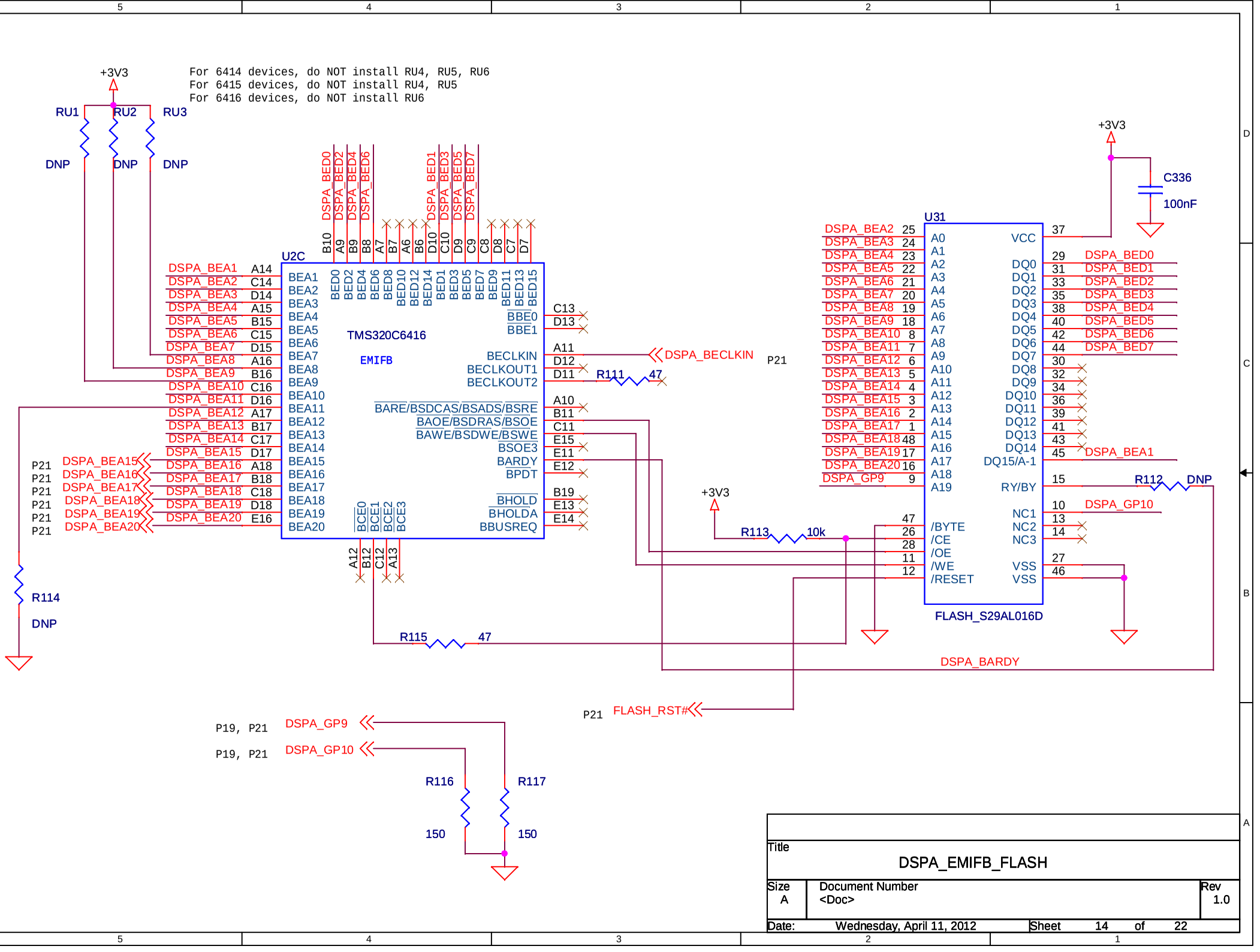

+1V2
C143 C144 C145 C146 C147 C148 C149 C150 C151 C152 C153 C154 C155 C156 C157 C158 C159 C160 C161 C162
10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF
C171 C172 C173 C174 C175 C176 C177 C178 C179 C180 C181 C182 C183 C184 C185 C186 C187 C188 C189 C190
10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF
C199 C200 C201 C202 C203 C204 C205 C206 C207 C208 C562
100nF 100nF 100nF 100nF 100nF 100nF 100nF 100nF 100nF 100nF 100nF
C217 C218 C219 C220 C221 C222 C223 C224 C225 C226 C589
100nF 100nF 100nF 100nF 100nF 100nF 100nF 100nF 100nF 100nF 100nF
C231 C232 C233 C234 C235 C236
1uF 1uF 1uF 1uF 1uF 1uF
C239 C240 C241 C242 C243 C244
1uF 1uF 1uF 1uF 1uF 1uF
C245 C246 C552 C553
100uF 100uF 100uF 100uF
+2V5
C163 C164 C165 C166 C167 C168 C169 C170 C560
10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF
C191 C192 C193 C194 C195 C196 C197 C198 C561
10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF
C209 C210 C211 C212 C213 C214 C215 C216
100nF 100nF 100nF 100nF 100nF 100nF 100nF 100nF
C227 C228 C229 C230
1uF 1uF 1uF 1uF
C563 C565 C237 C551
22uF 22uF 100uF 100uF
+3V3
C249 C250 C251 C252 C253 C254 C255 C256 C257 C258 C259 C260 C261 C262 C263 C264 C480 C481 C566
10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF
C275 C276 C277 C278 C279 C280 C281 C282 C283 C284 C285 C286 C287 C288 C289 C290 C482 C483 C567
10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF
C295 C296 C297 C298 C299 C300 C301 C302
100nF 100nF 100nF 100nF 100nF 100nF 100nF 100nF
C305 C306 C307 C308 C309 C310 C311 C312
100nF 100nF 100nF 100nF 100nF 100nF 100nF 100nF
C314 C315 C316 C317 C318 C319 C320 C321
1uF 1uF 1uF 1uF 1uF 1uF 1uF 1uF
C322 C323
100uF 100uF
+3V3
C265 C266 C267 C268 C269 C270 C271 C272 C273 C274
10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF 10nF
C291 C292 C293 C294
100nF 100nF 100nF 100nF
C303 C304 C478 C479
1uF 1uF 1uF 1uF
C568 C313
22uF 100uF
Title
FPGA_Decoupling
Size B
Document Number <Doc>
Rev 1.0
Date: Wednesday, April 11, 2012
Sheet 12 of 22

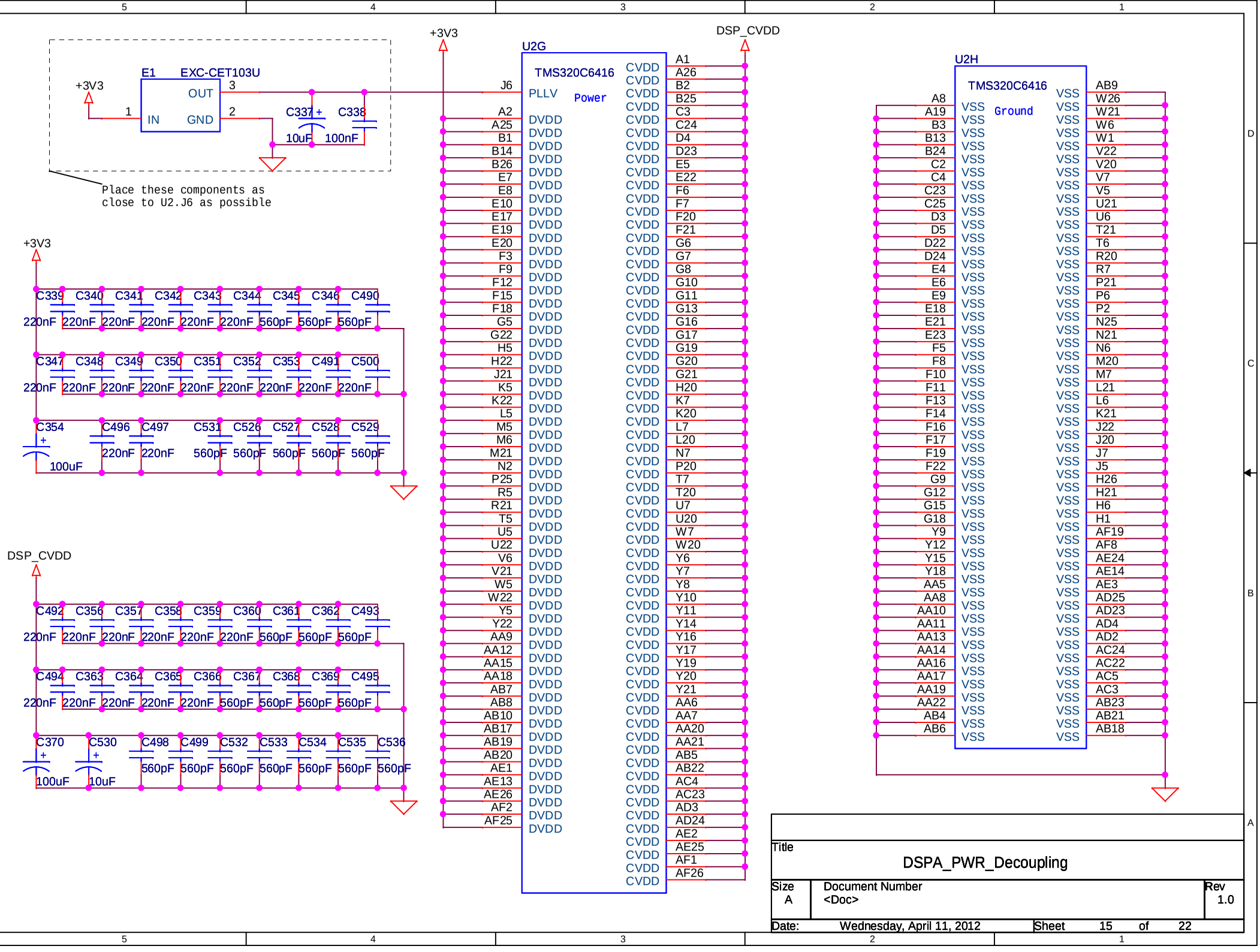


DSPA <--> FPGA Interface :

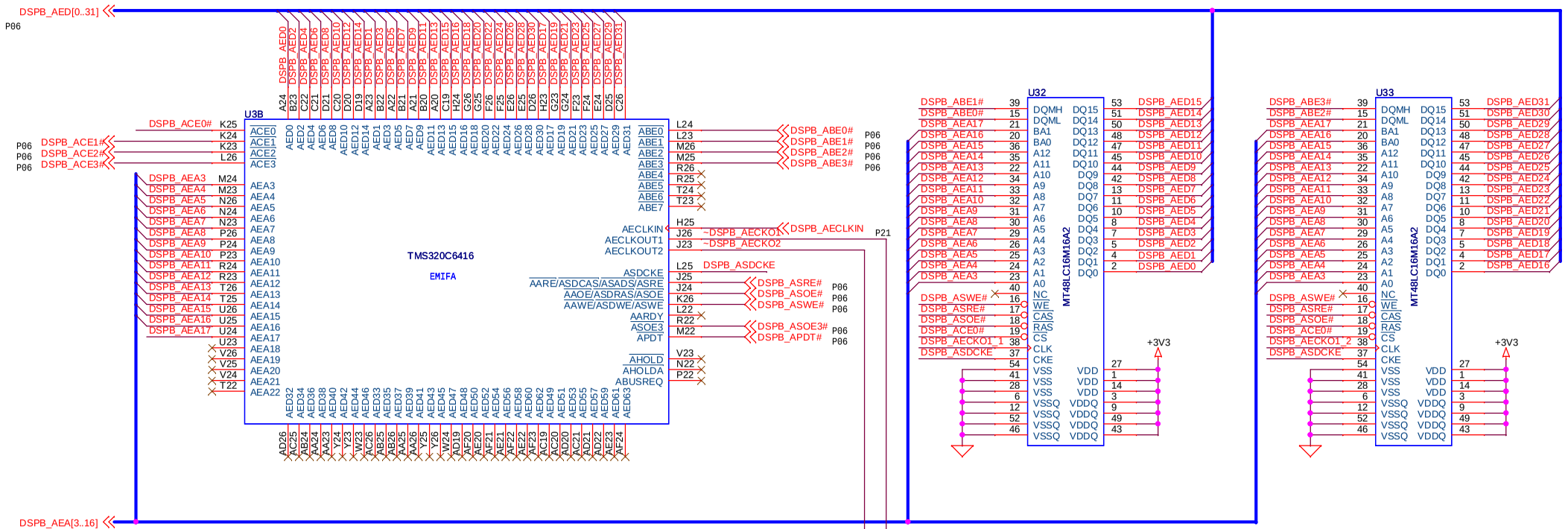
EMIFA		McBSP0	GPIO
DSPA_TAECLKO2	DSPA_ACE1#	DSPA_CLKX0	DSPA_INT4
DSPA_AEA[3..16]	DSPA_ACE2#	DSPA_DX0	DSPA_INT5
DSPA_AED[0..31]	DSPA_ACE3#	DSPA_FSX0	DSPA_INT6
DSPA_ASRE#	DSPA_ABE0#	DSPA_CLKR0	DSPA_INT7
DSPA_ASOE#	DSPA_ABE1#	DSPA_DR0	DSPA_GP0
DSPA_ASWE#	DSPA_ABE2#	DSPA_FSR0	DSPA_GP11
DSPA_ASOE3#	DSPA_ABE3#		TIMER
DSPA_APDT#			DSPA_TINP0
			DSPA_TOUT0

Title	DSPA_EMIFA_SDRAM		
Size B	Document Number <Doc>		Rev 1.0
Date: Wednesday, April 11, 2012		Sheet 13 of 22	


For 6414 devices, do NOT install RU4, RU5, RU6
For 6415 devices, do NOT install RU4, RU5
For 6416 devices, do NOT install RU6
+3V3
RU1
RU2
RU3
DNP
DNP
DNP
U2C
TMS320C6416
EMIFB
DSPA_BEA1
DSPA_BEA2
DSPA_BEA3
DSPA_BEA4
DSPA_BEA5
DSPA_BEA6
DSPA_BEA7
DSPA_BEA8
DSPA_BEA9
DSPA_BEA10
DSPA_BEA11
DSPA_BEA12
DSPA_BEA13
DSPA_BEA14
DSPA_BEA15
DSPA_BEA16
DSPA_BEA17
DSPA_BEA18
DSPA_BEA19
DSPA_BEA20
DSPA_BED0
DSPA_BED1
DSPA_BED2
DSPA_BED3
DSPA_BED4
DSPA_BED5
DSPA_BED6
DSPA_BED7
DSPA_BECLKIN
P21
R111
47
R114
DNP
R115
47
R116
150
R117
150
P19, P21
DSPA_GP9
P19, P21
DSPA_GP10
P21
FLASH_RST#
DSPA_BARDY
+3V3
R113
10k
U31
FLASH_S29AL016D
C336
100nF
R112
DNP
DSPA_GP10
DSPA_GP9
DSPA_BEA1
Title
DSPA_EMIFB_FLASH
Size
A
Document Number
<Doc>
Rev
1.0
Date:
Wednesday, April 11, 2012
Sheet
14
of
22



Title	DSPA_PWR_Decoupling	
Size A	Document Number <Doc>	Rev 1.0
Date: Wednesday, April 11, 2012	Sheet 15 of 22	

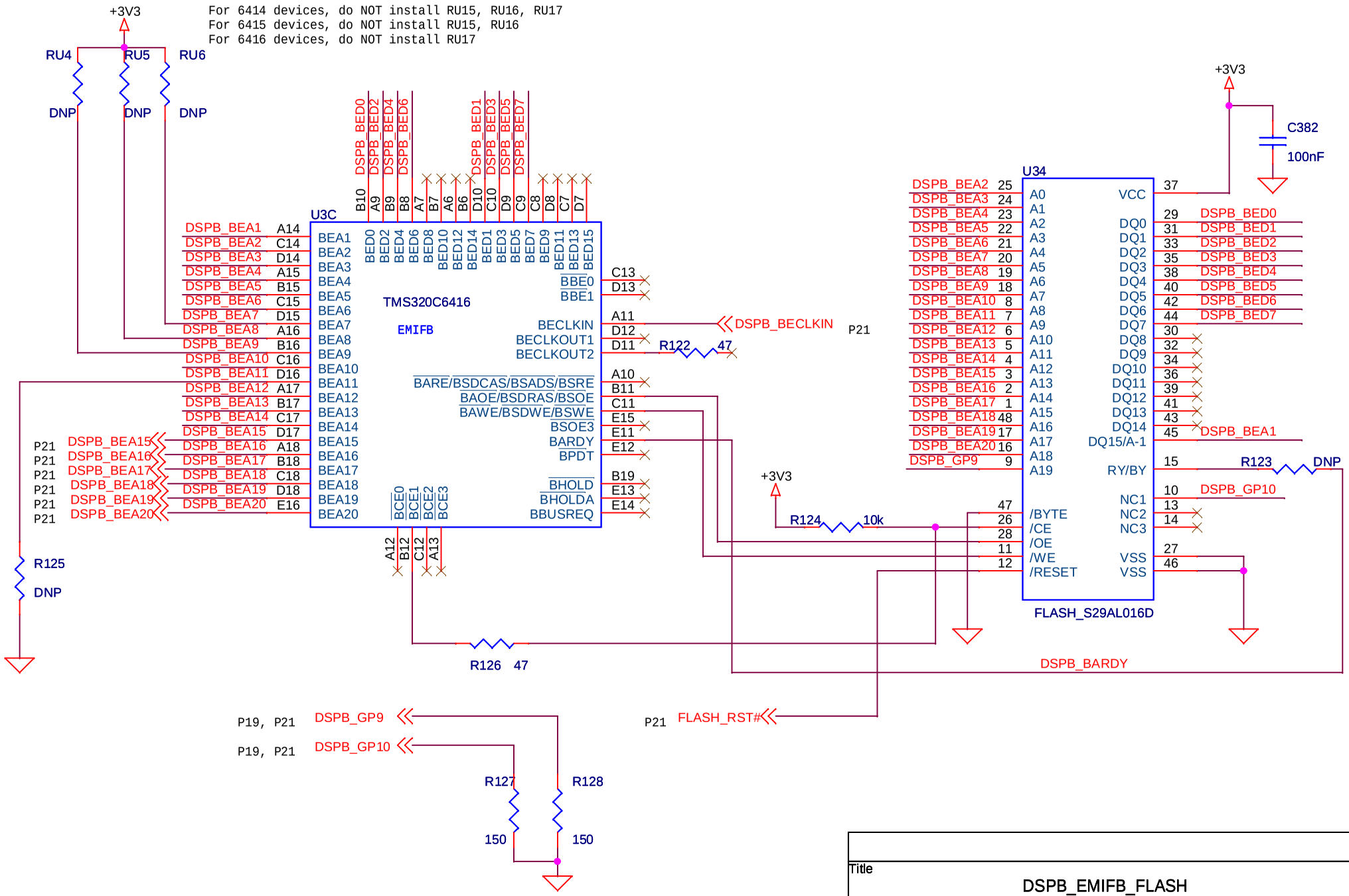


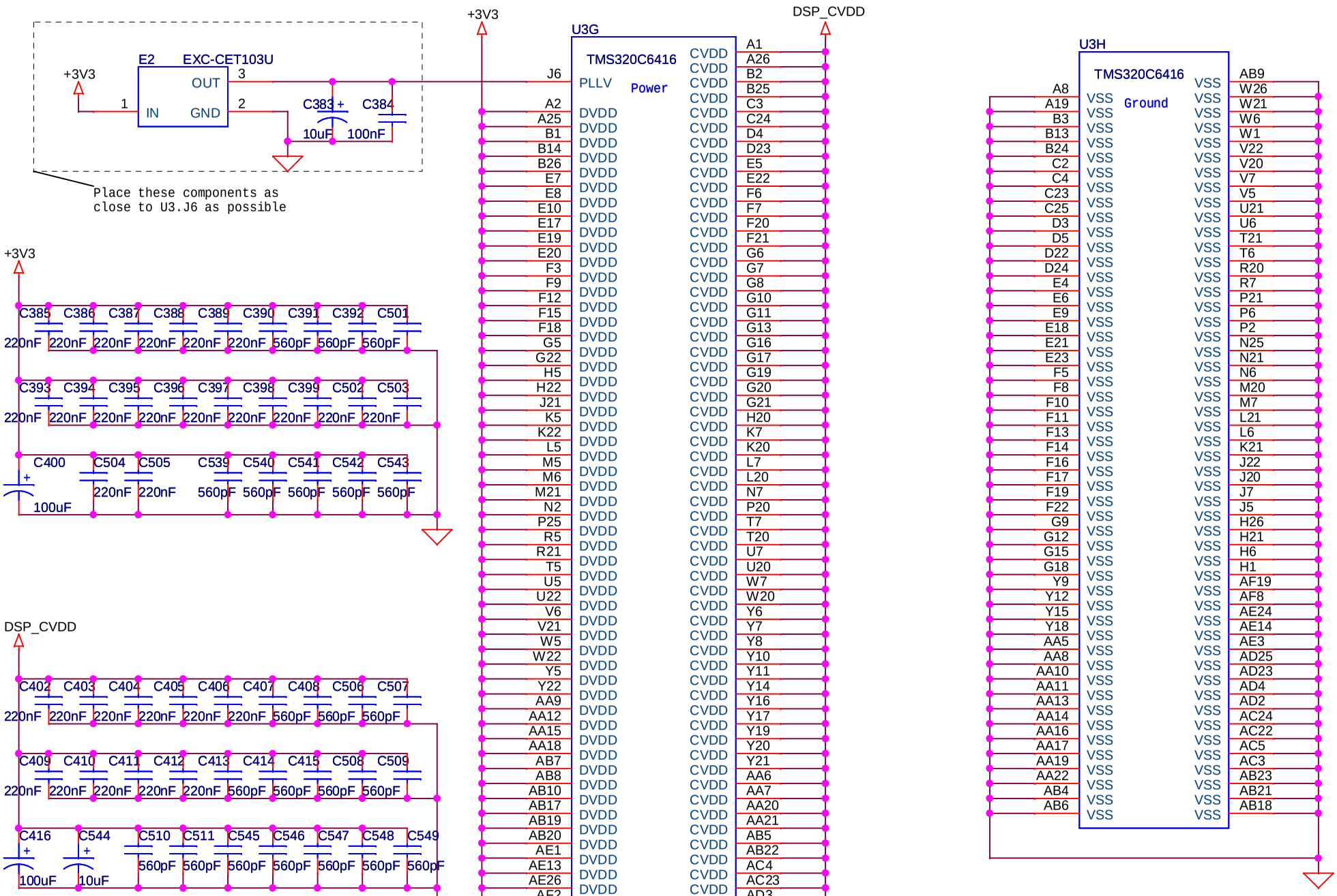
DSPB <--> FPGA Interface :

EMIFA		McBSP0	GPIO
DSPB_TAECLKO2	DSPB_ACE1#	DSPB_CLKX0	DSPB_INT4
DSPB_AEA[3..16]	DSPB_ACE2#	DSPB_DX0	DSPB_INT5
DSPB_AED[0..31]	DSPB_ACE3#	DSPB_FSX0	DSPB_INT6
DSPB_ASRE#	DSPB_ABE0#	DSPB_CLKR0	DSPB_INT7
DSPB_ASOE#	DSPB_ABE1#	DSPB_DR0	DSPB_GP0
DSPB_ASWE#	DSPB_ABE2#	DSPB_FSR0	DSPB_GP11
DSPB_ASOE3#	DSPB_ABE3#		
DSPB_APDT#			

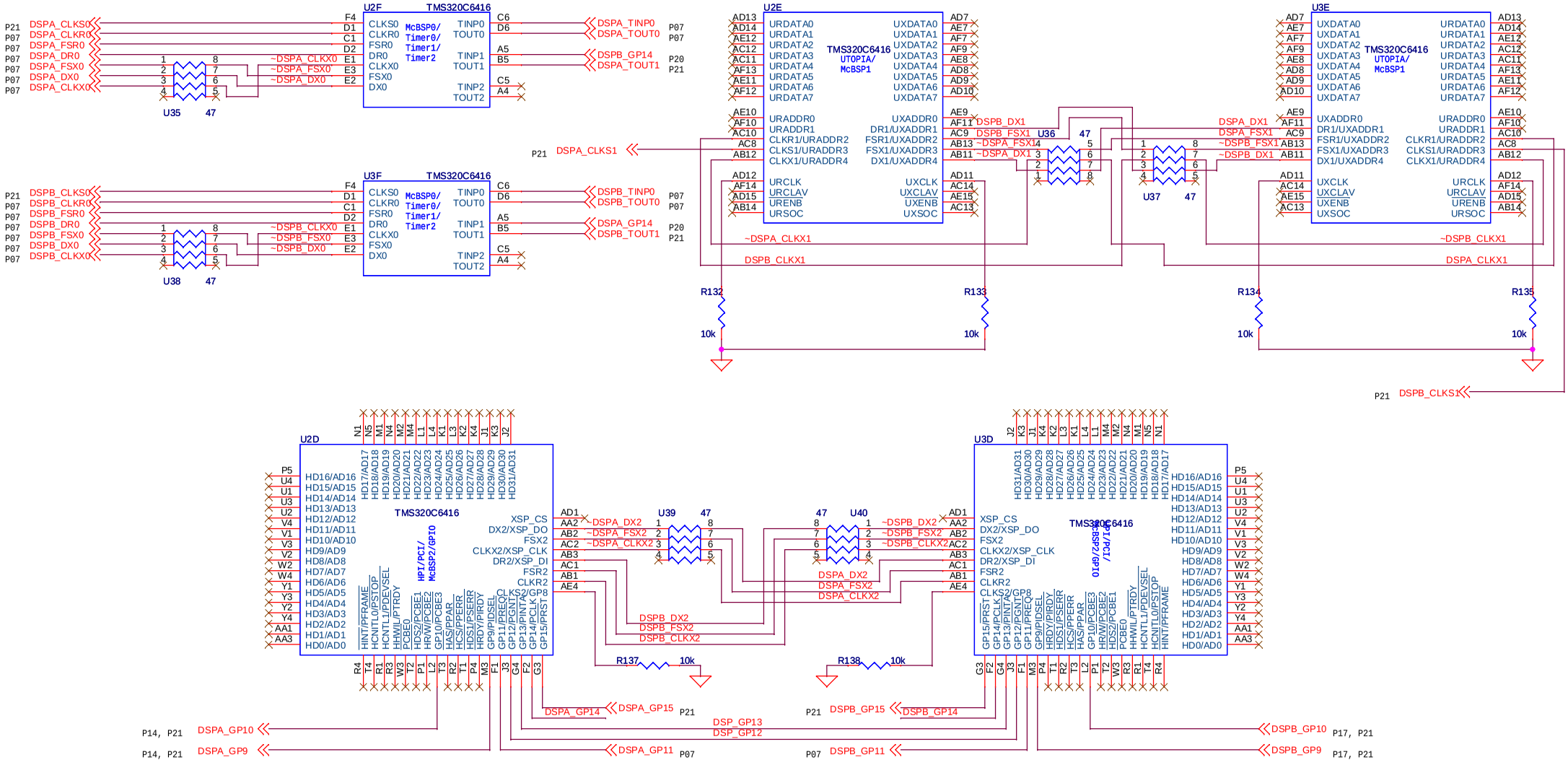
TIMER
DSPB_TINP0
DSPB_TOUT0

Title	DSPB_EMIFA_SDRAM		
Size B	Document Number <Doc>		Rev 1.0
Date: Wednesday, April 11, 2012	Sheet 16 of 22		

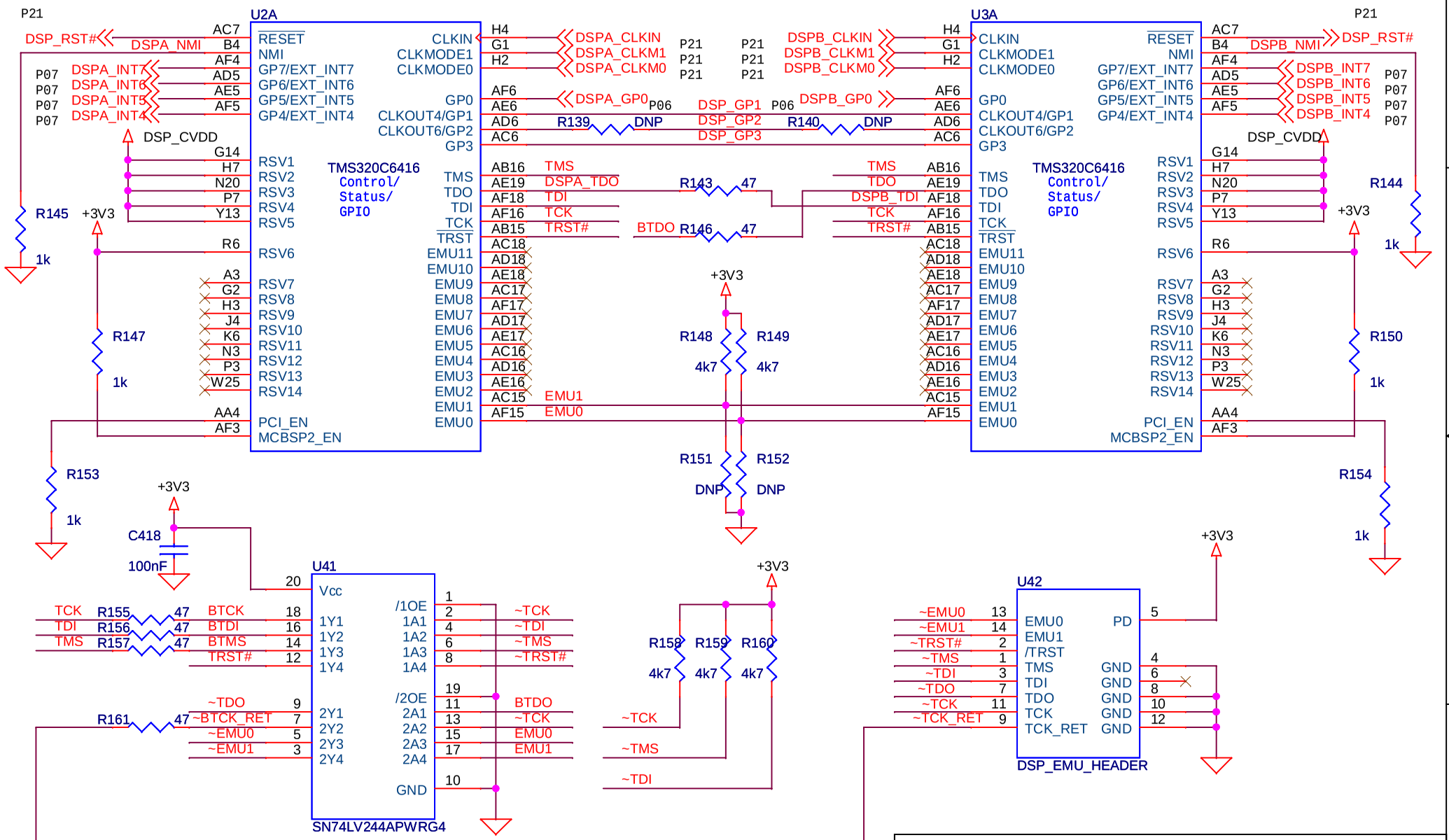

For 6414 devices, do NOT install RU15, RU16, RU17
For 6415 devices, do NOT install RU15, RU16
For 6416 devices, do NOT install RU17
+3V3
RU4
RU5
RU6
DNP
DNP
DNP
U3C
TMS320C6416
EMIFB
DSPB_BEA1
DSPB_BEA2
DSPB_BEA3
DSPB_BEA4
DSPB_BEA5
DSPB_BEA6
DSPB_BEA7
DSPB_BEA8
DSPB_BEA9
DSPB_BEA10
DSPB_BEA11
DSPB_BEA12
DSPB_BEA13
DSPB_BEA14
DSPB_BEA15
DSPB_BEA16
DSPB_BEA17
DSPB_BEA18
DSPB_BEA19
DSPB_BEA20
DSPB_BED0
DSPB_BED1
DSPB_BED2
DSPB_BED3
DSPB_BED4
DSPB_BED5
DSPB_BED6
DSPB_BED7
DSPB_BECLKIN
P21
R122
47
R123
DNP
R124
10k
R125
DNP
R126
47
R127
150
R128
150
P19, P21
DSPB_GP9
P19, P21
DSPB_GP10
P21
FLASH_RST#
DSPB_BARDY
U34
FLASH_S29AL016D
+3V3
C382
100nF
Title
DSPB_EMIFB_FLASH
Size
A
Document Number
<Doc>
Rev
1.0
Date:
Wednesday, April 11, 2012
Sheet
17
of
22



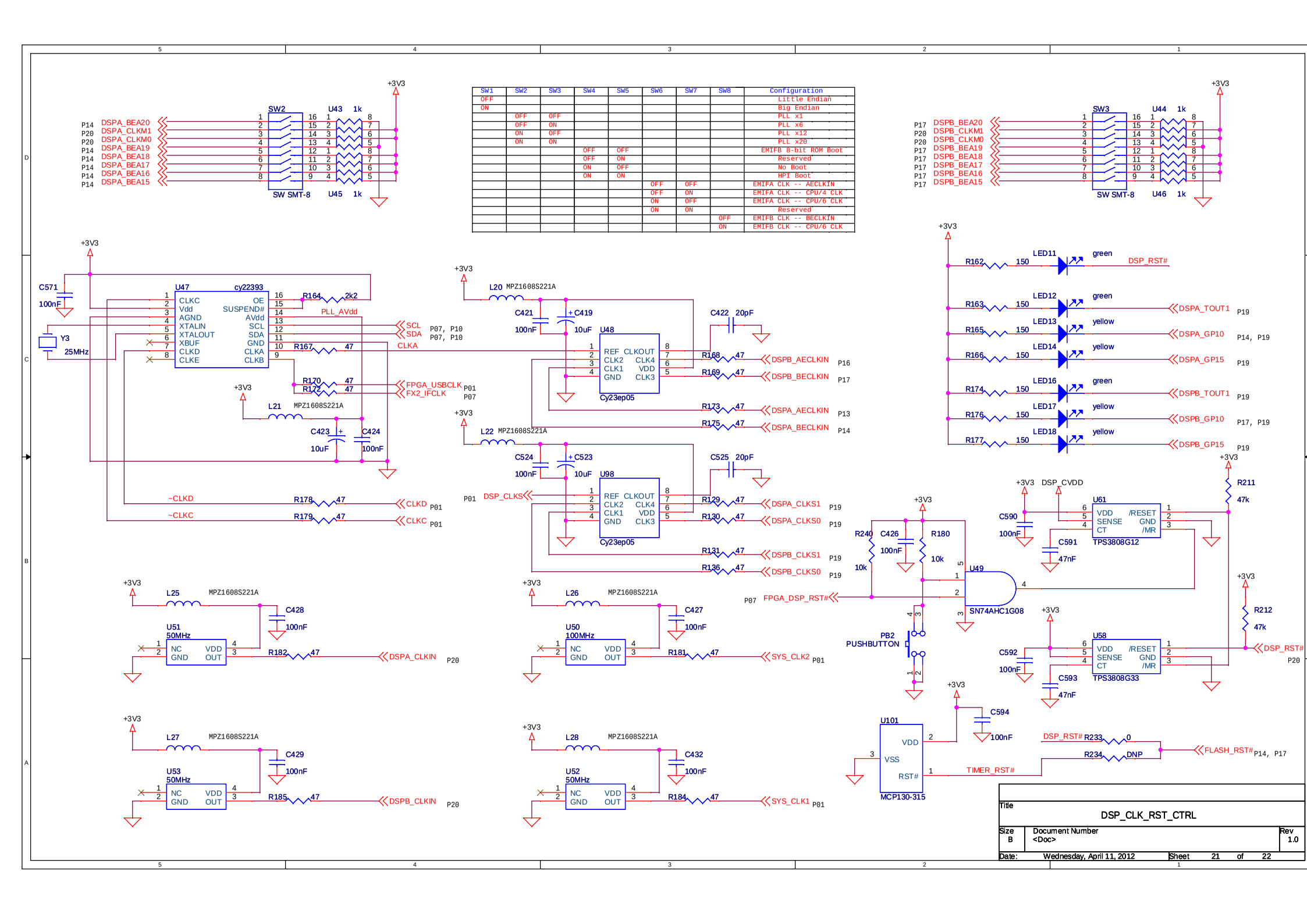
Title	DSPB_PWR_Decoupling	
Size: A	Document Number: <Doc>	Rev: 1.0
Date: Wednesday, April 11, 2012	Sheet 18 of 22	

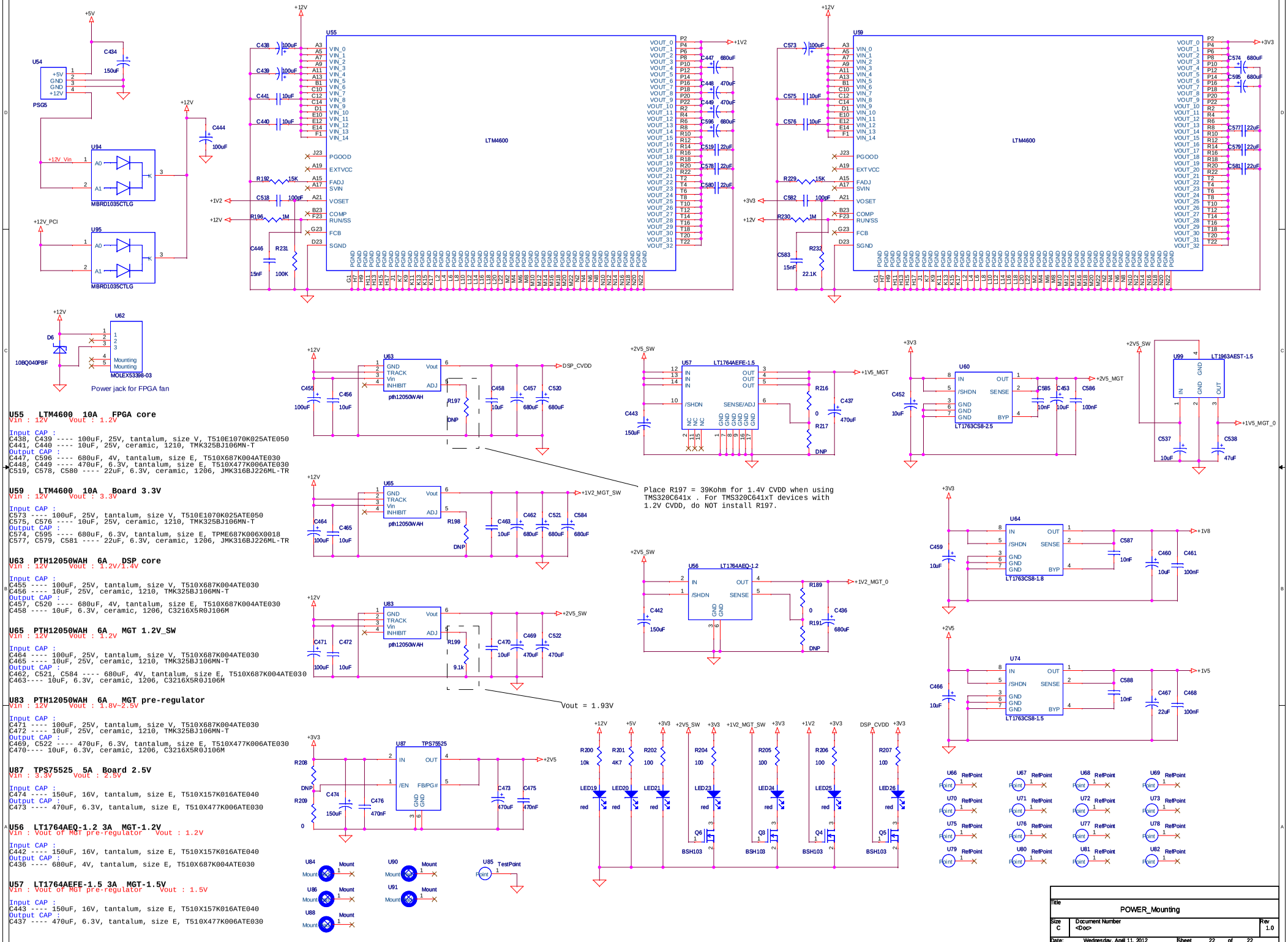


Title	DSP_McBSP	
Size B	Document Number <Doc>	Rev 1.0
Date: Wednesday, April 11, 2012	Sheet 19 of 22	



Title	DSP_JTAG	
Size A	Document Number <Doc>	Rev 1.0
Date: Wednesday, April 11, 2012	Sheet 20 of 22	


SW1 SW2 SW3 SW4 SW5 SW6 SW7 SW8 Configuration
OFF Little Endian
ON Big Endian
OFF OFF PLL x1
OFF ON PLL x6
ON OFF PLL x12
ON ON PLL x20
OFF OFF EMIFB 8-bit ROM Boot
OFF ON Reserved
ON OFF No Boot
ON ON HPI Boot
OFF OFF EMIFA CLK -- AECLKIN
OFF ON EMIFA CLK -- CPU/4 CLK
ON OFF EMIFA CLK -- CPU/6 CLK
ON ON Reserved
OFF EMIFB CLK -- BECLKIN
ON EMIFB CLK -- CPU/6 CLK
SW2 U43 1k SW SMT-8 U45 1k
SW3 U44 1k SW SMT-8 U46 1k
U47 cy22393 Y3 25MHz C571 100nF R164 2k2 PLL_AVdd R167 47 R170 47 R171 47
SCL SDA CLKA FPGA_USBCLK FX2_IFCLK ~CLKD ~CLKC R178 47 R179 47 CLKD CLKC
L20 MPZ1608S221A L21 MPZ1608S221A L22 MPZ1608S221A
C421 100nF C419 10uF C422 20pF C423 10uF C424 100nF C524 100nF C523 10uF C525 20pF
U48 U98 Cy23ep05 REF CLKOUT CLK2 CLK4 CLK1 VDD GND CLK3
R168 47 R169 47 R173 47 R175 47 R129 47 R130 47 R131 47 R136 47
DSPB_AECLKIN DSPB_BECLKIN DSPA_AECLKIN DSPA_BECLKIN DSP_CLKS DSPA_CLKS1 DSPA_CLKS0 DSPB_CLKS1 DSPB_CLKS0
L25 L26 L27 L28 MPZ1608S221A
C428 C427 C429 C432 100nF
U51 U50 U53 U52 50MHz NC GND VDD OUT
R182 47 R181 47 R185 47 R184 47
DSPA_CLKIN SYS_CLK2 DSPB_CLKIN SYS_CLK1
LED11 LED12 LED13 LED14 LED16 LED17 LED18 green yellow
R162 R163 R165 R166 R174 R176 R177 150
DSP_RST# DSPA_TOUT1 DSPA_GP10 DSPA_GP15 DSPB_TOUT1 DSPB_GP10 DSPB_GP15
R240 10k C426 100nF R180 10k U49 SN74AHC1G08 PB2 PUSHBUTTON FPGA_DSP_RST#
DSP_CVDD U61 U58 TPS3808G12 TPS3808G33 VDD /RESET SENSE GND CT /MR
C590 C591 C592 C593 100nF 47nF
R211 47k R212 47k
U101 MCP130-315 VDD VSS RST# C594 100nF R233 0 R234 DNP TIMER_RST# FLASH_RST#
Title DSP_CLK_RST_CTRL
Size B Document Number <Doc> Rev 1.0
Date: Wednesday, April 11, 2012 Sheet 21 of 22



U55 LTM4600 10A FPGA core
Vin : 12V Vout : 1.2V

Input CAP :
C438, C439 ---- 100uF, 25V, tantalum, size V, T510E1070K025ATE050
C441, C440 ---- 10uF, 25V, ceramic, 1210, TMK325BJ106MN-T
Output CAP :
C447, C596 ---- 680uF, 4V, tantalum, size E, T510X687K004ATE030
C448, C449 ---- 470uF, 6.3V, tantalum, size E, T510X477K006ATE030
C519, C578, C580 ---- 22uF, 6.3V, ceramic, 1206, JMK316BJ226ML-TR

U59 LTM4600 10A Board 3.3V
Vin : 12V Vout : 3.3V

Input CAP :
C573 ---- 100uF, 25V, tantalum, size V, T510E1070K025ATE050
C575, C576 ---- 10uF, 25V, ceramic, 1210, TMK325BJ106MN-T
Output CAP :
C574, C595 ---- 680uF, 6.3V, tantalum, size E, TPME687K006X0018
C577, C579, C581 ---- 22uF, 6.3V, ceramic, 1206, JMK316BJ226ML-TR

U63 PTH12050WAH 6A DSP core
Vin : 12V Vout : 1.2V/1.4V

Input CAP :
C455 ---- 100uF, 25V, tantalum, size V, T510X687K004ATE030
C456 ---- 10uF, 25V, ceramic, 1210, TMK325BJ106MN-T
Output CAP :
C457, C520 ---- 680uF, 4V, tantalum, size E, T510X687K004ATE030
C458 ---- 10uF, 6.3V, ceramic, 1206, C3216X5R0J106M

U65 PTH12050WAH 6A MGT 1.2V_SW
Vin : 12V Vout : 1.2V

Input CAP :
C464 ---- 100uF, 25V, tantalum, size V, T510X687K004ATE030
C465 ---- 10uF, 25V, ceramic, 1210, TMK325BJ106MN-T
Output CAP :
C462, C521, C584 ---- 680uF, 4V, tantalum, size E, T510X687K004ATE030
C463---- 10uF, 6.3V, ceramic, 1206, C3216X5R0J106M

U83 PTH12050WAH 6A MGT pre-regulator
Vin : 12V Vout : 1.8V~2.5V

Input CAP :
C471 ---- 100uF, 25V, tantalum, size V, T510X687K004ATE030
C472 ---- 10uF, 25V, ceramic, 1210, TMK325BJ106MN-T
Output CAP :
C469, C522 ---- 470uF, 6.3V, tantalum, size E, T510X477K006ATE030
C470---- 10uF, 6.3V, ceramic, 1206, C3216X5R0J106M

U87 TPS75525 5A Board 2.5V
Vin : 3.3V Vout : 2.5V

Input CAP :
C474 ---- 150uF, 16V, tantalum, size E, T510X157K016ATE040
Output CAP :
C473 ---- 470uF, 6.3V, tantalum, size E, T510X477K006ATE030

U56 LT1764AEQ-1.2 3A MGT-1.2V
Vin : Vout of MGT pre-regulator Vout : 1.2V

Input CAP :
C442 ---- 150uF, 16V, tantalum, size E, T510X157K016ATE040
Output CAP :
C436 ---- 680uF, 4V, tantalum, size E, T510X687K004ATE030

U57 LT1764AEFE-1.5 3A MGT-1.5V
Vin : Vout of MGT pre-regulator Vout : 1.5V

Input CAP :
C443 ---- 150uF, 16V, tantalum, size E, T510X157K016ATE040
Output CAP :
C437 ---- 470uF, 6.3V, tantalum, size E, T510X477K006ATE030

Place R197 = 39Kohm for 1.4V CVDD when using TMS320C641x . For TMS320C641xT devices with 1.2V CVDD, do NOT install R197.

Vout = 1.93V

Title	POWER_Mounting
Size C	Document Number <Doc>
Rev	1.0
Date:	Wednesday, April 11, 2012
Sheet	22 of 22