Hardware, Software, and Tools for an AsAP2 Many-Core System

By

NIMA MOSTAFAVI
B.S. (University of California Berkeley) July, 2011

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Bevan M. Baas, Chair

S. J. Ben Yoo

Venkatesh Akella

Committee in charge
2014

Abstract

This thesis describes the design and implementation of hardware and software which enable the integration of the 167-processor AsAP2 processor array chip into a system containing DRAM memory, mass storage, and high-speed interconnect. A new daughtercard utilizes high-speed LVDS interconnect for data and control interfaces between the array processor chip and a commercial FPGA board which serves as a hub for the entire system. Verilog code on the FPGA board automates programming of the AsAP2 chip using PCIe or JTAG connections and also enables the AsAP2 input and output data interfaces to communicate through a PCIe connection to the host computer. A scheduler program written in Perl reduces instruction counts and increases performance of AsAP2 assembly code by line re-arranging, register forwarding, and register renaming. Many Perl and C programs on the host computer simplify the conversion of the assembly programs and the input and output data in human readable format on the host computer to the machine code formats transferable to the FPGA board and vice versa. This interface design enables the design and implementation of many features such as SSD hard drives, DDR3 memories, and fiber optic networks that lead to the use of the low power AsAP2 chip in a large enterprise system.

Acknowledgments

First, I would like to sincerely thank my advisor Professor Bevan Baas for his support and guidance throughout my years at UC Davis. I was extremely lucky to have the opportunity of working in VLSI Computation Lab under his supervision, and it was thoroughly an enriching and delightful experience. I would also like to thank Professor S. J. Ben Yoo for his constant support and valuable advice through my graduate study. I would like to thank Professor Venkatesh Akella for his time and consideration in reviewing my thesis.

There were many people without their help this work could not be accomplished. I would like to express my appreciation to Aaron Stillmaker, Jon Pimentel, and Jeremy Webb for helping me with learning the tools, and their endless support and valuable advice through the projects. I would also like to thank all the people at VCL lab Aaron Stillmaker, Jon Pimentel, Jonathan Earl, Timothy Andreas, Michael Braly, Bin Liu, Emmanuel Adeagbo, Brent Bohnenstiehl, and Dean Truong for providing me with a friendly and exciting environment, inspiring me to keep following my research aspirations while helping me with my work.

I would also like to thank Roberto Proietti and Zheng Cao for helping me with many of my decisions in this project.

I would like to recognize Patty Hunter for her support in designing the daughter card, Ted Park from Green Circuits for fabricating and loading the boards, Eli Billauer from Xillybus for providing me with the PCIe interface.

I am grateful for the support from our sponsors ST Microelectronics, UC Micro, NSF Grants, SRC GRC Grants, C2S2 Grants, Intel Corporation, Intellasys Corporation, UC Davis Faculty Research Grant, and SEM. I am also thankful for the support of Mentor Graphics, Xilinx, and Cadence for the tools provided.

Finally, my special thanks goes to my beloved family for all their patience throughout these years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There is a high demand for systems with low power consumption and huge storage that can perform desired operations of an application on a massive input data in high speed and efficiency [1], [2]. For this purpose, the first generation of many core processors such as Array of Simple Asynchronous Processor (AsAP) with 36 tiled-based cores and low area for digital signal processing [5], [6] was designed and evaluated [7], [8]. Later the second generation of AsAP called AsAP2 with 167 heterogeneous [9] cores was introduced with a low-area [10], [11] circuit switch inerprocessors networking [12] and shared input queues . In this architecture the concept of Globally Asynchronous Locally Synchronous (GALS) processor was introduced to have different cores being able to run at different clock frequencies using independent oscillators [14], [15] that helps reduce the power consumption of low activity cores by reducing their clock frequencies [16]. Due to scalability of this design [17], there has also been many efficient scalable sorting algorithm running on large data sets using many core processors [18] as well as many encryption algorithms such as AES [19] to be used on enterprise systems or data centers.

As a very good example of high performance low power cores, AsAP2 is selected to be used with two targeted goals in this document. First, is the interface of AsAP2 with different devices such as memories, hard drives, displays, etc for Digital Signal Processing (DSP) and embedded system [20] applications. Second interconnect of large enterprise system with many blades. Since the second targeted goal can include the first goal, enterprise system is only discussed briefly.

1.1 AsAP2 Interface Considerations

The AsAP2 interface is a simple parallel input and output data interface with a serial programming interface. By enabling the connection between this simple interface and other devices many applications and programs such as AES encryption [19], H.264 video decoding [21] for 720p HDTV [22], ultrasound signal processing [23], real-time 802.11a baseband receiver [24], and fast fourier transform [25] can be tested on the many core platform with very low power consumption using dynamic voltage and frequency scaling [26] implemented in AsAP2 chip. In addition, The parallelism of many designs such as H.264 encoder [27] and AES [28] can be exploit. However, in order for a AsAP2 package to communicate with other devices it must be connected to an arbiter to convert this simple system to the correct standard used in the peripheral device. In order to achieve this goal a daughter card connecting AsAP2 to an arbiter device such as FPGA board must be considered. In this design due to the difference in clock frequency domains between these different devices and AsAP2 output or input processors, dual-clock FIFOs [29] are used. In addition to These connections many more software tools must be written to provide the required interface to program the AsAP2 processors to run any application efficiently.

1.2 Design of the Large Enterprise System

The desired enterprise system is a combination of AsAP2 and optical interconnect network with DDR Memories and SSD Hard drives. The desired system with large optical network and compute units can be broken down into many sub blocks in the network level with the highest level being called a *Macroblade*, second level of the design is called *Miniblade*, the next level in hierarchy below the Miniblade is called a *Microblade*, and finally the lowest level is called a *Picoblade* [30]. Figure 1.1 displays the Macroblade view of this design while Figure 1.2 displays a detailed view of a Picoblade.

This document is only dedicated to the design and implementation of some parts of the Picoblade. In the next subsection, the logical view of a Picoblade is described then the physical view of the Picoblade is explained.

1.2.1 Logical Design of a Picoblade

In order to meet the requirements of this design, the design should have storage units, network connectors, processing units, processor programming connectors, input data units, output

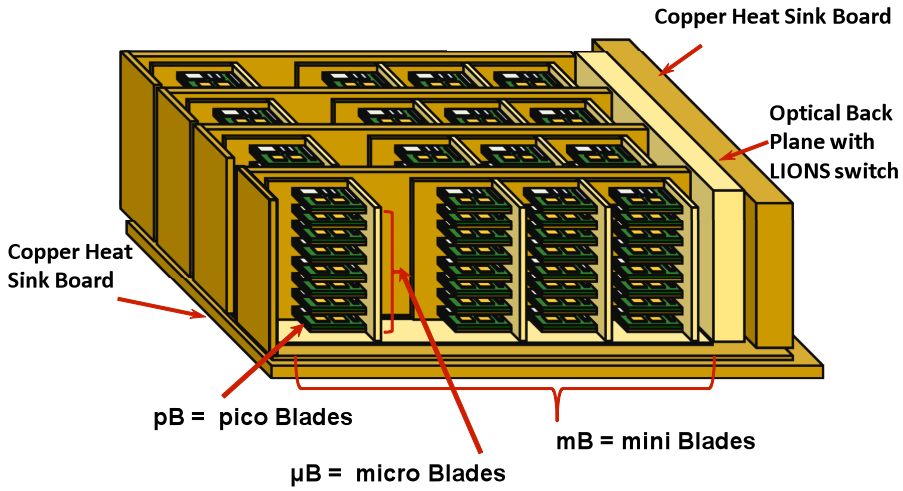


Figure 1.1: The schematic displaying the Macroblade level of the enterprise system (above Figure reproduced with permission from Prof. S. J. B. Yoo)

data units, and control units. Based on this the tasks has been divided into two separate types of Picoblade units as follows:

1. Computational Picoblade
2. System interface Picoblade

Figure 1.2 displays a diagram of a simple Miniblade displaying the two types of Picoblades. The red rectangle on this picture represents a host computer, and each green rectangle represents a Microblade. There are two Microblades in this picture.

1. Computational Picoblades: The computational Picoblades in this design only have storage units, network connectors, processing units, processor programming connectors, and a control unit. These Picoblades don't require an input/output data unit in addition to the internal network connection since all the information is received/sent from/to the outside of this system through only one Picoblade called a system interface Picoblade. The system interface Picoblade is designed to transfer data between other Picoblades and a server (host) computer, so they can be displayed or used later. This Picoblade has all the components that a computational unit has with the addition of the input and output data unit.

   In order to achieve these goals in a computational Picoblades, A Xilinx FPGA (Field - Programmable Gate Array) board (Xilinx Virtex 7 VC 709 [31]) is used to act as the control unit and connects all the parts together. AsAP2 processor arrays are used in this system as the computational unit. AsAP2 is connected to the FPGA board using a custom designed

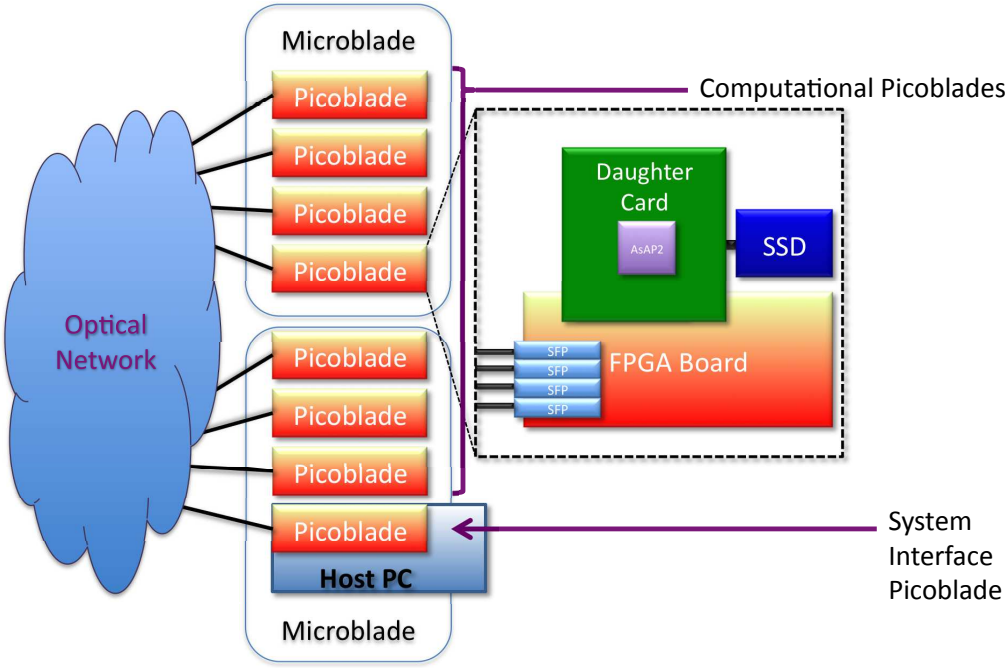


Figure 1.2: The schematic displaying both types of Picoblades in the Miniblade level

daughter card. Each one of these daughter cards has two SATA (Serial Advanced Technology Attachment) connectors, an AsAP2 chip, and one FMC (FPGA Mezzanine Card) connector. The AsAP2 chip and the two SATA SSDs (Solid State Drives) can be connected to each computational Picoblade (FPGA board) using the FMC connector. Even though SSDs connect to AsAP2 daughter board, but they won't be communicating to AsAP2 chip directly. These connections are all made through the control unit (FPGA board) to make them more flexible. Each computational FPGA (Picoblade) is connected to the network via four SFP (Small form-factor pluggable) high-speed optical connectors.

A control unit (FPGA Board) in a computational unit is programmed through a JTAG (Joint Test Action Group) interface before the FPGA verilog is fully tested. Afterward, it can be reprogrammed through the same JTAG interface or just by using the BPI (Byte Peripheral Interface) Parallel NOR flash memory on the Virtex 7 FPGA board. The AsAP2 Chip can be programmed using either the BRAM (Block Random Access Memory) on the FPGA board or the USB (Universal Serial Bus) to UART (Universal Asynchronous Receiver-Transmitter) Bridge. Computational Picoblades distribute the input data coming in from the system interface Picoblade between all the SSD Drives before the main program starts. Then, they start their computational intensive computation and compute the results. Finally,

these results get transferred back to the system interface Picoblade to be sent out to the user interface or the host computer.

2. System Interface Picoblade: This unit acts as a data collector or a data distributor in the network system as well as doing the computations like other computational Picoblades. It uses a PCIe (Peripheral Component Interconnect Express) connection to burst in the input information from a host computer or a server into this optical network system. Also, after the computation is done, all internal computational Picoblades send their final results back into this Picoblade using the optical network to be transferred back to the server. System interface Picoblade has all the components of a computational Picoblade, but it also has an additional capability of communicating with outside world. This node can be used as a firewall blocking all the outside accesses to the internal information stored in the SSD drives in case of an intrusion. Also these units instead of using the USB to UART Bridge connection for reprogramming use PCIe to reprogram AsAP2 instructions into BRAM in addition to programming the FPGA board using the JTAG connection. The AsAP2 chip can be rapidly reprogrammed using the BRAM.

1.3 Previous Work

The AsAP2 chip has been designed, fabricated, and tested using a daughter card designed by previous members of the VCL group by connecting to an Avnet Virtex-5 LX evaluation kit via an Expansion (EXP) connector. This board uses a single ended connection to transfer data between the AsAP2 chip and the FPGA board. The design interface has been implemented in a way that a new AsAP2 program gets loaded to AsAP2 chip using a UART connection to the FPGA board from the host every time. The start command is sent from the host computer every time to start the AsAP2 program, so a constant host connection to the board is required and the board can't be programmed and removed from the host for remote functionality. The verilog code on FPGA board converts the 8 bit UART input data to the correct format before sending it to the AsAP2 chip to get it programmed. The input to AsAP2 chip has not been implemented on the old FPGA board. The output from AsAP2 chip can only be streamed out to a set of pins to be displayed on a digital oscilloscope or a similar tool.

The written assembly code had to be manually optimized and written with all the hardware detailed instructions such as no operation instructions (NOPs). There had been a scheduler program

written with limited functionality that wasn't included in the system interface to be used to optimize or add extraneous instructions such as NOPs.

1.4 Project Contributions

1. A new AsAP2 daughter card was designed from scratch. This redesign provides higher communication speed and compatibility with the new FPGA board. This complete redesign was made since a new FPGA has been required to enable the capability of optical modules in the network. This new FPGA board requires a different connection that doesn't connect to the previously designed daughter card.

2. The verilog code for the FPGA board has been completely rewritten to provide an easier interface for programming, input, and output. The programs can be loaded to the FPGA board in two different ways while it doesn't require a host instantiating the start signal. The program can be loaded and start signal can be initiated directly from the FPGA board to the AsAP2 chip. This interface also gives the possibility to read multiple lines from BRAM and make modification to the program stored on the FPGA board BRAM. Xillybus provided a PCIe interface to transfer input and output data between the host and the FPGA board in addition to the reprogramming interface previously mentioned. The Xillybus template programs were modified, and additional programs were added to convert a simple ASCII input data and programs to a binary format to be sent to the FPGA board. Also, new programs were added to take the output data or instruction from AsAP2 to ASCII format. These programs act as an interface between the PCIe logic and human ASCII visible input and output.

3. A scheduler and a converter were completed to convert a simple assembly or simulator pseudo C program to optimized code by inserting all required extra instructions to be either transferred back to the simulator for further simulation or to the assembler to be programmed into the AsAP2 chip. This peripheral software reports the memory usage and instruction count of the assembly program.

4. Different host software has been written in Perl and C to provide the required machine format from a higher-level human readable format for the inputs and output data to the AsAP2 as well as the programming interface on the host computer.

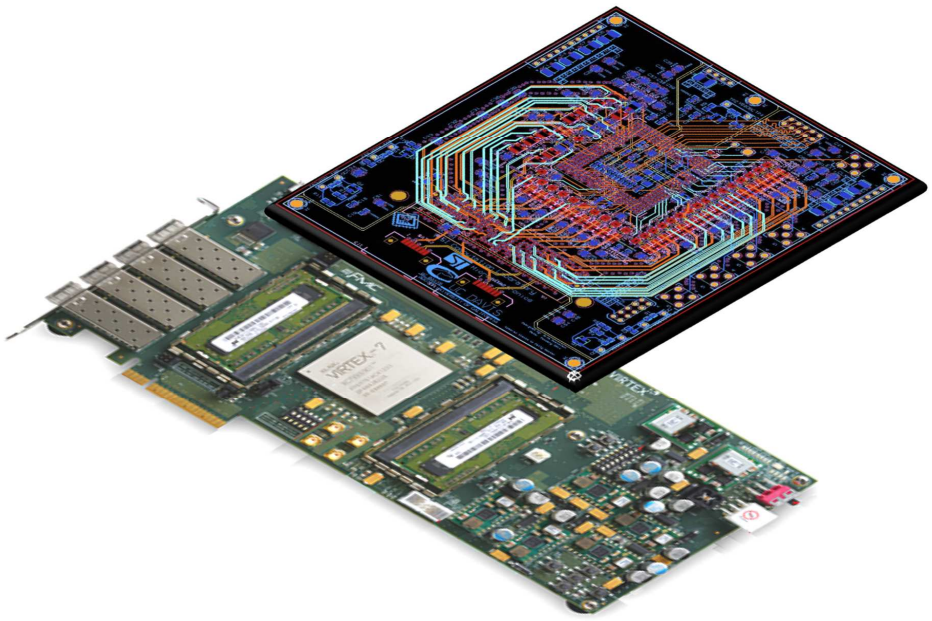


Figure 1.3: A physical view of the designed Picoblade with a layout view of the daughter card

1.5 Organization

Chapter 2 discusses the AsAP2 chip and its functionalities. This chapter introduces the AsAP2 chip built by students in VLSI Computational Lab (VCL) [32]. The focus of this chapter is mostly on the power, input, and output data interfaces of this chip in addition to the programming of this chip since there are being considered many times in the future chapters.

Chapter 3 discusses the process and consideration on building the AsAP2 daughter card used to connect the AsAP2 chip to the FPGA board in addition to possible future hard drive connections. This board has been fabricated with all the components loaded on the board.

Chapter 4 discusses the design and logic used in the verilog code used to design the internal communication logics in the FPGA board to program and communicate between AsAP2 and the FPGA board. Figure 1.3 displays a simple picture of how the daughter card (PCB board) connects to the FPGA board in a Picoblade.

Chapter 5 discusses host programs and modifications done in order to make the previously implemented host codes to run with the new system. This section mostly discusses about how the interface has been modified for the new interface to program a FPGA board for a system interface Picoblade using a JTAG connection.

Chapter 6 discusses the PCIe interface bring up, and the host interface for reprogramming

and data input and output verilog logic.

Chapter 7 briefly discusses about the small battery powered system that was build to power the previously build daughter card. This section describes the decisions and the full schematic design of different components in that demo.

Finally Chapter 8 concludes the discussion about this project and gives some future opportunities to improve and complete this project.

Chapter 2

AsAP2 Chip Interfaces

The Asynchronous Array of Simple Processors 2 [10] (AsAP2) consists of 164 fine-grained, homogeneous programmable processors in addition to three special purpose processors for motion estimation, veterbi decoding, and fast fourier transform (FFT). Each of these 164 programmable processors can be programmed to run a different set of instructions running at their independent clock frequencies below 1.3GHz.

In order to program AsAP2, programmer requires writing their program in AsAP2 assembly language and for each of the 164 processors independently. In order to do this each program has been labeled with their location as is shown in Figure 2.1.

2.1 Input and Output Data Interface

The input and output data interfaces of AsAP2 chip run at 2.5 V with single ended connections. This 19-bit interface consists of 16 data signals, one clock signal, one request signal, and one valid signal.

2.1.1 Input Data Interface

Inputs to the AsAP2 chip can go to any of the 12 processors on the first column (proc 0,0 to proc 0,11 on Figure 2.1) of the AsAP2 array. Each processor can be statically assigned the input data stream coming in from pins marked as *in_data*[0:15] (DI#) in Figure 2.2.

Three signals *in_valid*(IV), *in_clk*(IC), and *in_request*(IR) are additional signals coming to or leaving the input processor. The *in_clk* is the input clock to the input buffer of the input processor. The *in_request* and the *in_valid* are the handshaking signals between the transmitter and

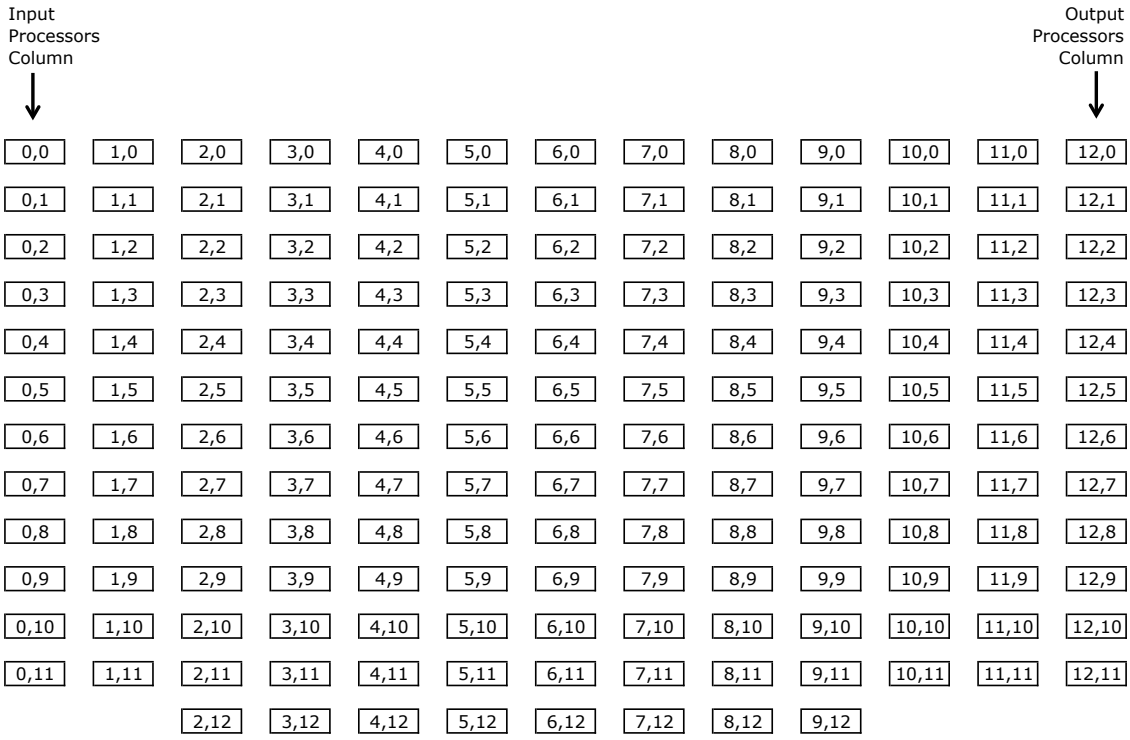


Figure 2.1: The schematic view of AsAP2 processors with their corresponding labels in the AsAP2 chip array

the receiving processor in AsAP2 chip. The AsAP2 processor sets the *in_request* signal high (logical one) to show it is ready to receive data, and the input source, sending the data to AsAP2 chip, sets the *in_valid* signal high (logical one) when the data coming to AsAP2 chip is valid and low (logical zero) when there is no data available to be sent to the AsAP2 input processor.

2.1.2 Output Data Interface

The outputs from AsAP2 chip can go to any of the 12 processors on the last column (proc 12,0 to proc 12,11 on Figure 2.1) of the AsAP2 array. Similar to the input data interface, each processor can statically be assigned to be the output source of the AsAP2 chip. These output pins are labeled *out_data*[0:15](DO#) in Figure 2.2.

Similar to the input data interface, the output data interface also has three additional signals *out_clk* (OC), *out_valid* (OV), and *out_request* (OR). The *out_clk* signal outputs the clock signals that the output processor runs to process and produce the output data. The *out_valid* signal displays whether the data on *out_data* lines are valid or not while the *out_request* signals the output processor when the receiver is ready to receive the output data from the AsAP2 output processor.

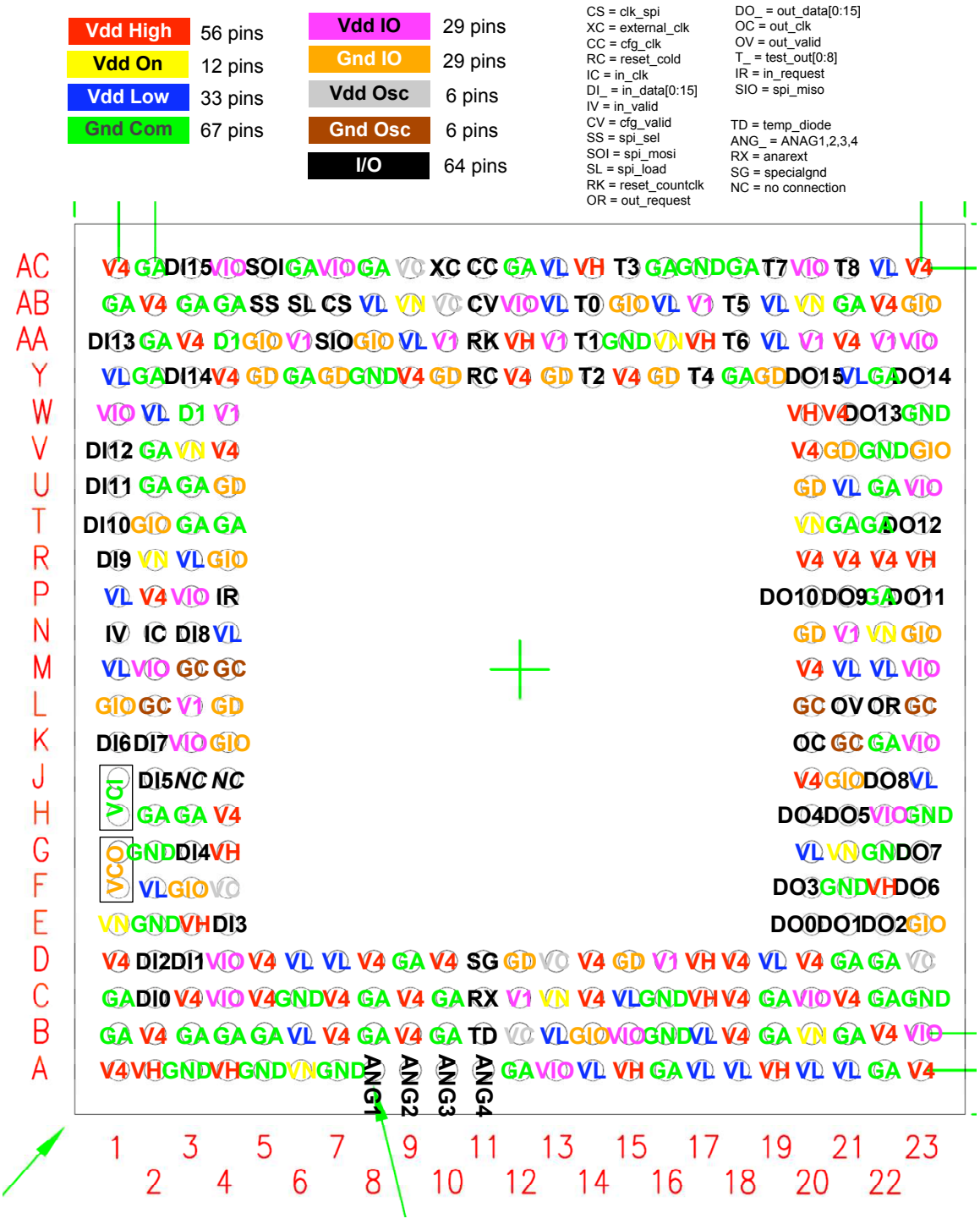


Figure 2.2: The AsAP2 pinout diagram. This is an actual pin readout from the package when facing the pins side

2.2 Programming Interface

The AsAP programming interface is designed for simplicity. It is a serial interface that loads the instruction and configuration information into the AsAP2 chip. Each processor's instruction memory (IMEM) can store up to 128 instructions. In addition, each processor's clock frequency can be separately configured in addition to may other settings that can be set on each processor. In general the programming interface can be divided into two different sections of physical and functional (Instruction Format).

2.2.1 Physical Interface

Physical programming interface is divided into two different sections. The Serial Programming Interface (*SPI*) and the Configuration Interface (*CFG*).

2.2.1.1 Serial Programming Interface

The *SPI* consists of five signals *clk_spi* (CS, also called *spi_clk*), *spi_sel* (SS), *spi_mosi* (SOI), *spi_load* (SL), and *spi_miso* (SIO) as shown in Figure 2.2. The *clk_spi*, labeled as SCK in *spi_slave.v* file of the AsAP2 verilog source code, is the clock signal that the input serial bits of data come into AsAP2 programming unit in spi_slave.v file.

In the AsAP2 programming design, the master is the source that sends program data to AsAP2 chip and the slave is the AsAP2 programming unit. Considering the master and slave rule, *spi_mosi* (Serial Programming Interface — Master Out Slave In) sends the programming data bits to AsAP2 from the source of programming while *spi_miso* (Serial Programming Interface — Master In Slave Out) returns the input data bits in a loop back format, back to the master for error checking and signal integrity.

The *spi_sel*, labeled as *CSn* in spi_slave.v file of AsAP2 verilog source code, is an active low input signal to AsAP2 chip to signals AsAP2 programming unit that the input MOSI signal is valid. This signal stays low only when the actual programming data is being sent to be loaded in AsAP2 chip. While the two MSBs (Most Significant Bits) of the programming packets (Explained in Instruction Format in Section 2.2.2 on page 13) are being loaded, this signal stays high.

Finally, *spi_load* signal, labeled as LOAD_EN in spi_slave.v file, is an active high input signal to AsAP2 programming unit that only goes high when one packet has completely arrived to the AsAP2 chip to be stored as a valid packet.

2.2.1.2 Configuration Interface

The configuration interface consists of two signals, *cfg_valid* (CV) and *cfg_clk* (CC) as shown on Figure 2.2. The *cfg_clk* signal is the clock frequency at which the input programming packets (explained in instruction format in Section 2.2.2) enter the configuration unpack unit (in *cfg_unpack.v* file of AsAP2 verilog source code). This signal runs 40× slower than *spi_clk* signal since for every 20 *spi_clk* clocks *cfg_clk* make one transition from high to low or low to high to latch. Each 20-bit packet gets sent only when the *cfg_clk* is low, but when the *cfg_clk* goes high the received 20 bits gets latched in for another 20 cycles of *spi_clk*.

The *cfg_valid* signal is an active high input signal that signals the configuration unpack unit (in cfg_unpack.v file of AsAP2 verilog source code) that the current stored packet by AsAP2 programming unit in spi_slave.v is a valid packet and must be unpacked based on the two MSBs of the programming packets described in the next section.

2.2.2 Instruction Format

Input program packets could be :

1. Instructions that go to Instruction Memory (IMEM)
2. Configuration data that go to Configuration Memory or CMEM (not to be confused with Dynamic Configuration Memory (DCMEM) that can be modified during the program execution)
3. Other configuration and signals such as Configuration (CFG) pre-instruction runs to fill Data Memories (DMEM) or set other DCMEM settings before processors run the main program.

Each instruction or CFG pre-instruction sent to AsAP2 to program AsAP2 processors consists of a maximum total of 56 bits. There are 21 bits for address and 35 bits for the instruction itself. Each configuration settings sent to CMEM of AsAP2 to configure AsAP2 processors also consists of a total of 37 bits. There are 21 bits for address and 16 bits for data. This information is based on *parse.c* from *aprog* module as a part of the host computer programming chain described more in Chapter 6.

The serial packets coming to the AsAP2 programming unit are divided into four different kinds of packets with all 20 bits long labeled as Address Upper, Address Lower, Data Upper, and Data Lower. The two most significant bits (MSBs — bits 19 and 18) specify the type of packet. The least significant bits (LSBs — bits 0 – 17) are either address or data (both data and address were referred to data in previous sections) bits. Bit 18 describes whether the information in the packet is

a Data packet (bit 18 set to 1 — high) or Address packet (bit 18 set to 0 — low). Bit 19 describes whether the data or address information in the packet is for the upper half or the lower half of the data or the address bits. When this bit is set to 0 (low) upper half is selected, and it is the lower half of the address or data bits when it is set to 1 (high).

2.3 Test Signals Interface

Testing Interface of AsAP2 consists of 9 test signals *test_out*[0:8] (T#) as shown on Figure 2.2. These test signals are used to output different values based on the configuration settings. These signals can be used to output the clock of a processor or other information about the FIFOs, stall, and program counter in a processor.

2.4 Power Delivery

2.4.1 Input Source Voltages

AsAP2 has a total of 136 pins on its package for input voltages as shown on Figure 2.2. These pins feed five separate power rails. The first power rail is *VddHigh* taking 56 pins. This power rail is used for high clock frequency load when the high performance is desired. This voltage can go up to 1.3 V. The second power rail is *VddLow* taking only 33 pins. This power rail is normally set below *VddHigh* for low power operations. This voltage rail is normally used with lower frequencies to minimize the power dissipation of the chip. The third power rail is *VddOn* taking 12 pins on the package. These pins power the always-on logic such as the DVFS circuitry on AsAP2 chip. This voltage can set to maximum voltage of 1.3 V. The fourth power rail is *VddOsc* and uses 6 pins. This power rail is used for the oscillator logic block on each processor. This pin can be set to a maximum voltage of 1.3 V. The fifth power rail is *VddIO* taking 29 pins. This power rail is used to power input/output receivers and drivers of the AsAP2 chip. These receivers and drivers run at 2.5 V.

2.4.2 Ground

There are a total of 102 pins associated to ground as shown on Figure 2.2. These pins break down into three categories. The first category is *GndCom* taking 67 pins. *GndCom* is the ground corresponding to *VddHigh*, *VddLow*, and *VddOn* source voltages ground connection. The second category is *GndOsc* taking 6 pins. This ground corresponds to *VddOsc*. Finally, the third

category is *GndIO* taking 29 pins corresponding to *VddIO* pins.

2.5 Other Signals

Four signals are output analog pins, and they are connected to one particular processor in the AsAP2 array to measure some Analog values. These four signals measure *VddHigh*, *VddLow*, *VddOn*, and *Clk* values. The last signal is an input clock signal and mainly used for testing purposes. These signals are analog high precision signals that require special attention for correct display values.

Chapter 3

Daughter Card Design

The Printed Circuit Board (PCB) design of a daughter card was one of the key parts of the project, and required very careful design and consideration due to high cost of both Fabrication and AsAP2 chip used on this board. Many changes were made due to many limitations between the two interfaces after in-depth consideration.

3.1 Design Goals

The main goals of this design are summarized as follows:

1. Correct Functionality: One of the main goals of this design is correct functionality of the devices. The PCB connects to a Field Programmable Gate Array (FPGA) board built by Xilinx called "Xilinx Virtex-7 FPGA VC709 Connectivity Kit" using a FPGA Mezzanine Card (FMC) connector, so AsAP2 chip on the PCB can connect and communicate with the programmed FPGA board logic. In addition, PCB requires implementing a connection to Solid State Drives (SSDs) using a Serial Advanced Technology Attachment (SATA) connection for mass storage capability through the FPGA board.

2. High Reliability: One of the main focuses of this design is the high reliability of the design. This term is used more in the context of this chapter mainly in Section 3.3.5. Daughter cards that produce different results due to emission, Electromagnetic Interference (EMI), or other environmental factors are not desired in the design of a printed circuit board, and they require special attention.

3. High Speed: This design goal requires a large amount of consideration due to crucial limiting

factors, such as maximum clock frequency, imposed on the input and the output data interfaces of the AsAP2 chip using single ended signals in the pervious designs of AsAP2 daughter card. Some of these considerations try to improve the design of the PCB to provide a faster interface than the previous design. This faster speed results in more results in less time

4. High Yield: High yield is desired in any mass produced design [33], and this design is no exception. High yield design reduces the extra overhead cost by decreasing damaged and unwanted boards. Following the design rules given by the fabrication and assembly company increases the design yield greatly.

5. Low Total Cost: One of the most important factors in the design of any device or system is the cost. The total cost equation considered for the PCB design is as follow.

$$TotalCost = NRECost + PCBFabricationCost + BoardComponentsCost + LoadingBoardCost[+ConsultantCost] \quad (3.1)$$

ConsultantCost is an optional part of this analysis that can be removed from the equation especially when a professional designer designs the PCB.

To further explore this option below shows an example list of cost for each item for the AsAP2 designed board. This quote is given based on fabrication and assembly of 32 PCBs by Green Circuits [34].

$$NonRecurringExpense(NRE) = \$400.00 \quad (3.2)$$

$$PCBFabricationCost = \$74.00\ cost/board * 32 = \$2,368.00 \quad (3.3)$$

$$BoardComponentsCost = \$212.10\ cost/board * 32 = \$6,787.20 \quad (3.4)$$

$$LoadingBoardCost = \$75.00\ cost/board * 32 = \$2,400.00 \quad (3.5)$$

$$ConsultantCost = NotEnclosed \quad (3.6)$$

The total cost for 32 boards is $\$11,955.20$.

3.2 Daughter Card Design at a Glance

The new designed board for AsAP2, called *testboard2*, has 12 layers and made of Flame Retardant 4 (FR4) material with dielectric constant (Er) of 4.2 according to Green Circuits. The

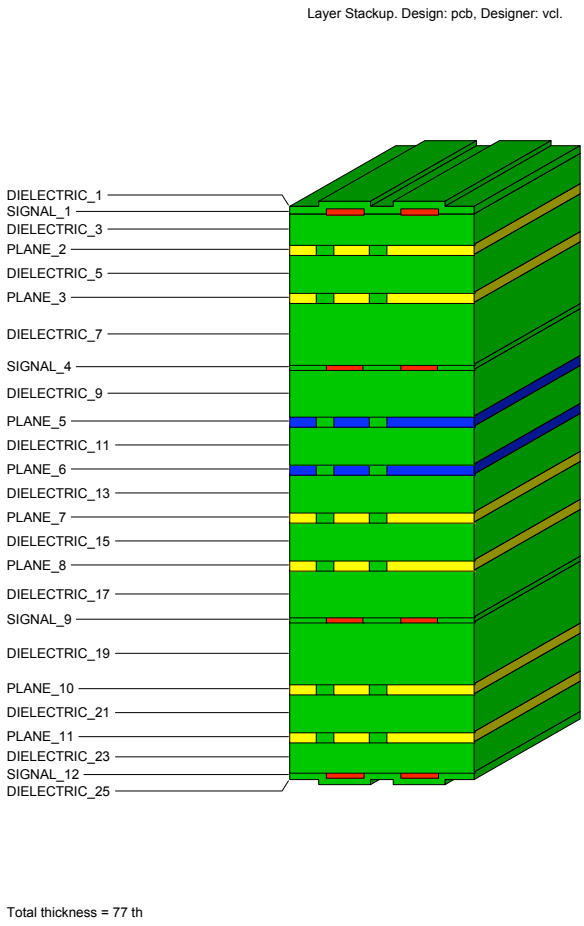


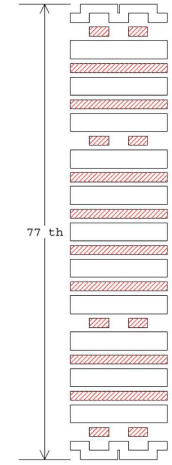
Figure 3.1: Layer stack up of the testboard2 scaled thickness drawing

board thickness is 77 mils. Figure 3.1 is a scale drawing of the PCB layers.

This board consists of 4 signal layers, 3 power layers, 4 ground layers, and 1 layer with both ground and power planes. The ground layers have been divided into three different ground Regions of *GNDG* (labeled as GG on the PCB), *GNDSATA* (labeled as GS on the PCB), and *GNDOSC* (labeled as GO on the PCB). The power layers have been divided into 5 different power regions of *VDDON* (labeled as VN on the PCB), *VDDOSC* (labeled as VO on the PCB), *VDDH* (labeled as VH on the PCB), *VDDL* (labeled as VL on the PCB), and *VDDIO* (labeled as VIO on the PCB).

The width of the signals in the signal layers was sized to match 50 ohms while the width of power and ground signals in the signal layers were selected to minimize resistance. Figure 3.2 shows the width and the thickness of the different signal layers of this design in addition to the impedance of these signals.

The PCB has two SATA connectors to connect SSD drives, one FMC connector to connect



Layer Stackup. Design: pcb, Designer: vcl.

Number of layers: 25
Total thickness = 77 th

NN	Layer Name th	Type th	Usage th	Thickness th	Technology th	Er th	Metal th	Bulk R ohm-m	T coef 1/°C	Loss Tangent 1/°C	Test Width th	Z0 ohm
1	DIELECTRIC_1	Dielectric	Solder Mask	1		1				0.1		
2	SIGNAL_1	Metal	Signal	0.7		<Auto>	Copper	1.724e-008	0.00393	0.1	7	51.5
3	DIELECTRIC_3	Dielectric	Substrate	4	Prepreg	4.2				0.02		
4	PLANE_2	Metal	Split/Mixed	1.4		<Auto>	Copper	1.724e-008	0.00393	0.02	12	34.8
5	DIELECTRIC_5	Dielectric	Substrate	5	Core	4.2				0.02		
6	PLANE_3	Metal	Split/Mixed	1.4		<Auto>	Copper	1.724e-008	0.00393	0.02	12	31.4
7	DIELECTRIC_7	Dielectric	Substrate	8	Prepreg	4.2				0.02		
8	SIGNAL_4	Metal	Signal	0.7		<Auto>	Copper	1.724e-008	0.00393	0.02	5	52.4
9	DIELECTRIC_9	Dielectric	Substrate	6	Core	4.2				0.02		
10	PLANE_5	Metal	Split/Mixed	1.4		<Auto>	Copper	1.724e-008	0.00393	0.02	12	31.4
11	DIELECTRIC_11	Dielectric	Substrate	5	Prepreg	4.2				0.02		
12	PLANE_6	Metal	Split/Mixed	1.4		<Auto>	Copper	1.724e-008	0.00393	0.02	12	25.1
13	DIELECTRIC_13	Dielectric	Substrate	5	Core	4.2				0.02		
14	PLANE_7	Metal	Split/Mixed	1.4		<Auto>	Copper	1.724e-008	0.00393	0.02	12	25.1
15	DIELECTRIC_15	Dielectric	Substrate	5	Prepreg	4.2				0.02		
16	PLANE_8	Metal	Split/Mixed	1.4		<Auto>	Copper	1.724e-008	0.00393	0.02	12	31.4
17	DIELECTRIC_17	Dielectric	Substrate	6	Core	4.2				0.02		
18	SIGNAL_9	Metal	Signal	0.7		<Auto>	Copper	1.724e-008	0.00393	0.02	5	52.4
19	DIELECTRIC_19	Dielectric	Substrate	8	Prepreg	4.2				0.02		
20	PLANE_10	Metal	Split/Mixed	1.4		<Auto>	Copper	1.724e-008	0.00393	0.02	12	31.4
21	DIELECTRIC_21	Dielectric	Substrate	5	Core	4.2				0.02		
22	PLANE_11	Metal	Split/Mixed	1.4		<Auto>	Copper	1.724e-008	0.00393	0.02	12	34.8
23	DIELECTRIC_23	Dielectric	Substrate	4	Prepreg	4.2				0.02		
24	SIGNAL_12	Metal	Signal	0.7		<Auto>	Copper	1.724e-008	0.00393	0.1	7	51.5
25	DIELECTRIC_25	Dielectric	Solder Mask	1		1				0.1		

Figure 3.2: The stack up view of the PCB with trace width impedance values

to the FPGA board, and one AsAP2 chip to do the processing in addition to other parts and connectors for the total of 302 components.

The board dimensions are 4850 mil (width) x 4250 mil (height). These dimensions were chosen so the PCB can be attached to the FPGA board via FMC connector while it is installed into the special computer purchased for this project since the FPGA board is too big to fit in most desktop computers.

3.3 Daughter Card Evolution and Redesign Factors

The design of the daughter card can be broken into four different evolution sections. The first evolution describes briefly about the previous fabricated daughter card, called *testboard1*, designed for a different FPGA board. In the second section, the first version of the new PCB, called *testboard2-v1*, is described. Next the second version of the PCB, called *testboard2-v2*, is described in details. Finally, the final version of the Test board, called *testboard2-v3*, is explained. This final version is the version that has been fabricated, so it can be attached to the VC709 FPGA board.

3.3.1 Testboard1

The original AsAP2 daughter card was designed, so it can be installed on a FPGA board using an Expansion (EXP) connector. Figure 3.3 shows the layout view of the original daughter

Table 3.1: testboard1 trace lengths

	Max Length (mil)	Min Length (mil)	Difference	Time* (ps)
Input	2341.37	875.79	1465.58	249.15
Output	2272.75	1066.9	1205.85	204.99

*The Time column values are based on 170 $\frac{psec}{in}$ traces.

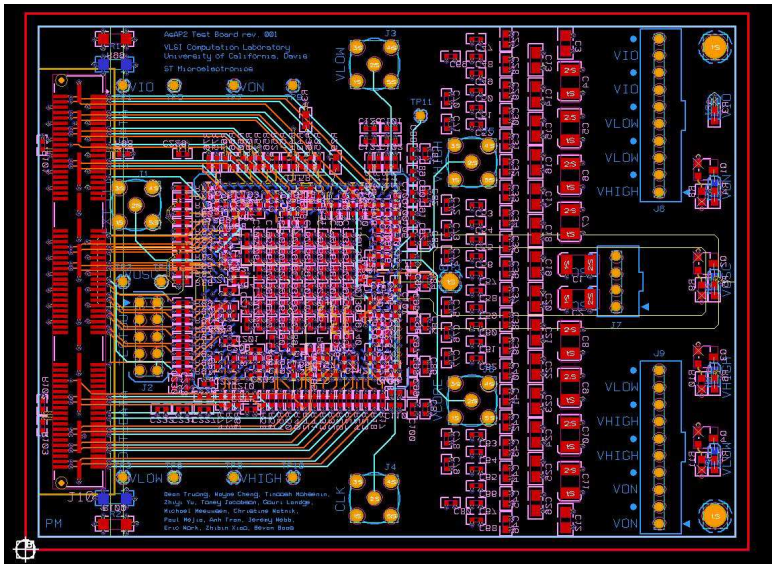


Figure 3.3: The layout view of the original daughter card — testboard1

card with only the main signals displayed.

The maximum and minimum length of the signals in this design is as follow where 170ps/inch is calculated using Equation 3.7:

$$PCBSignalSpeed^{-1} = \frac{Speedoflight}{\sqrt{Er}} = \frac{3 \times 10^8}{\sqrt{4.2}} \frac{\text{m}}{\text{sec}} \times 4 \times 10^{-}11 \frac{\text{sec} \times \text{in}}{\text{psec} \times \text{m}} = 0.0058554 \frac{\text{in}}{\text{psec}}$$

$$PCBSignalSpeed = 170.7 \frac{\text{psec}}{\text{in}} \tag{3.7}$$

Table 3.1 displays the regional trace length for the maximum and the minimum traces as well as the difference in length between the maximum and minimum traces and the corresponding delay difference between the two signals in testboard1 design. This table displays two regions of input and output where the input region is considered as all the in coming, outgoing, and clock

signals that are required to input a value to the AsAP2 chip, and the output region is defined similarly to input region except it is for all the signals used to send out a value from AsAP2 chip. The trace difference delay timing is achieved based on 170 psec/In PCB signal speed calculated in Equation 3.7.

3.3.2 Testboard2-v1

The first attempt to make testboard1 design compatible with the requirement of the project, was made to modify the implementation of the PCB, so it can be connected to VC709 FPGA board using the FMC connector as well as having the capability of stacking up two boards on top of each other. With this requirement, two AsAP2 chips could connect to the same FPGA board.

In this version of the designed board, all the connections to AsAP2 are single ended, similar to the previous board design, and the number of decoupling capacitors used are kept the same as well. Also, the dimensions of the board are kept the same.

The main modification from the old design is the replacement of the EXP connector with two FMC connectors (one male and one female connector). The male FMC connector is used to connect to the FPGA board while the female FMC connector is used to connect the second daughter card on top of the first daughter card. All the power, ground, and test connectors are moved to the sides of the board, and all of them are replaced by right angled parts instead of straight parts to provide enough gap, so the two boards can stack on top of each other.

Also, two SATA connectors are added to the PCB. These connectors are connected to the FPGA board through the FMC connector using a Low Voltage Differential Signaling (LVDS) connection. This design includes an additional power region for *VDDSATA* with decoupling capacitors, matching network resistors, power source connectors, and ground connectors for the SATA LVDS signals.

In this design the orientation of the AsAP2 chip was kept the same as before so all the single ended signals for input and output regions (i.e. input region is based on all the in coming and outgoing signals and clock signals related to inputting data to AsAP2 chip) were routed from the sides of the chip (Figure 3.4 Displays the layout view of this design) with minimal length matching considerations. The auto route feature built in to the software couldn't completely route the design due to complexity of the design, so every signals was had routed. The maximum and minimum trace length values are shown in Table 3.2 for different regions of AsAP2 signals where 170 ps/inch used in this table is calculated based on Equation 3.7 for the maximum and minimum trace length

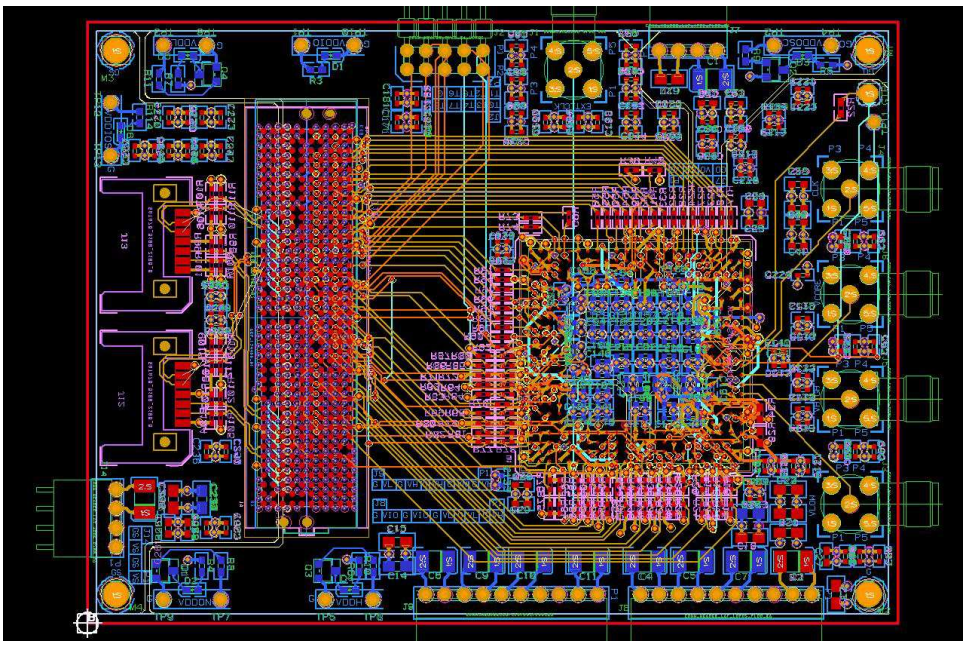


Figure 3.4: The layout view of the first version of the new daughter card — testboard2-v1

Table 3.2: Testboard2-v1 trace lengths

	Max Length (mil)	Min Length (mil)	Difference	Time* (ps)
Input	3014.12	1350.03	1664.09	282.90
Output	2611.35	1293.61	1317.74	224.01

*The Time column values are based on 170 $\frac{psec}{in}$ traces.

difference delay timing. As it is visible the difference between the maximum and minimum signals has increase, but they are still in the same orders of magnitude.

3.3.3 Testboard2-v2

This version of the board is designed with many modifications to the first version of the testboard2 design while still keeping the dimensions of the board the same. There are still two FMC connectors, and all the connectors are kept in the right-angled format and on the sides of the board.

The main change is done on the SATA connector side of the board. The extra matching resistors and decoupling capacitors are removed from the board because the resistive matching is only required at the source or the destination of a signal and the PCB is not located at either end of the SATA connections. The *VDDSATA* power region is completely removed because it isn't used at all with the connectors when the resistor matching is removed from the board, and it is replaced with another *GNDSATA* region. This design reduces the distance of the return loop current from

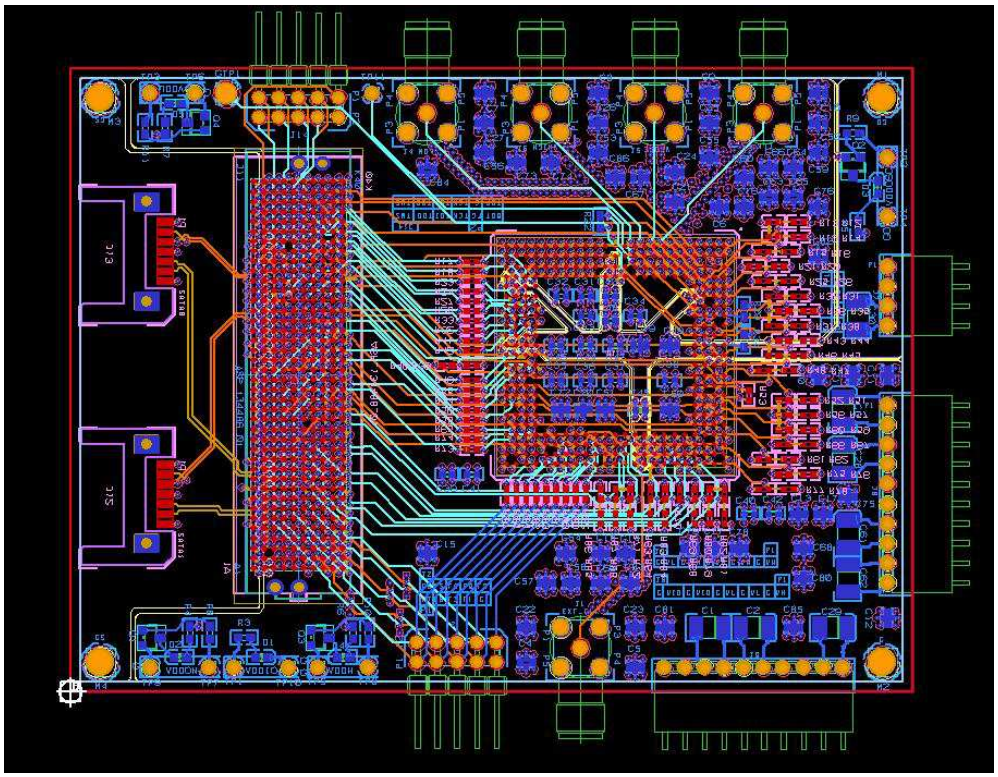


Figure 3.5: The layout view of the second version of the new daughter card — testboard2-v2

the LVDS SATA signals on the 9th layers of the board since instead of a *VDDSATA* region on the 10th layer a *GNDSATA* region is used next to the SATA signals on the 9th layer. This ground region is mainly added in addition to the *GNDSATA* on the 11th layer that was mainly used to reduce the distance of the return loop current from the SATA signals on the 12th layer of the board. Figure 3.5 displays this modification.

In addition to the SATA connector area modification, most of capacitors are modified to X2Y capacitors for higher decoupling of the input powers from ground. This modification reduces the number of capacitors required in the design to almost half. Some of the capacitors are still kept the same as before just because of the low quantity of these capacitors required for some power and ground regions of the design such as *VDDOSC* to *GNDOSC* decoupling capacitors.

The final modification is done on the orientation of the AsAP2 chip to reduce the difference between the length of each signal region on the AsAP2 input and output signals. This reduction is summarized in Table 3.3 where 170ps/inch is calculated using Equation 3.7. This modification has reduced the different between the signal lengths in the same region by about 1000 mil. After this modification, the traces were hand routed again and many parts were moved around.

Table 3.3: Testboard2-v2 trace lengths

	Max Length (mil)	Min Length (mil)	Difference	Time* (ps)
Input	2872.79	2568.19	304.6	51.78
Output	1422.57	1082.6	339.97	57.79

*The Time column values are based on 170 $\frac{psec}{in}$ traces.

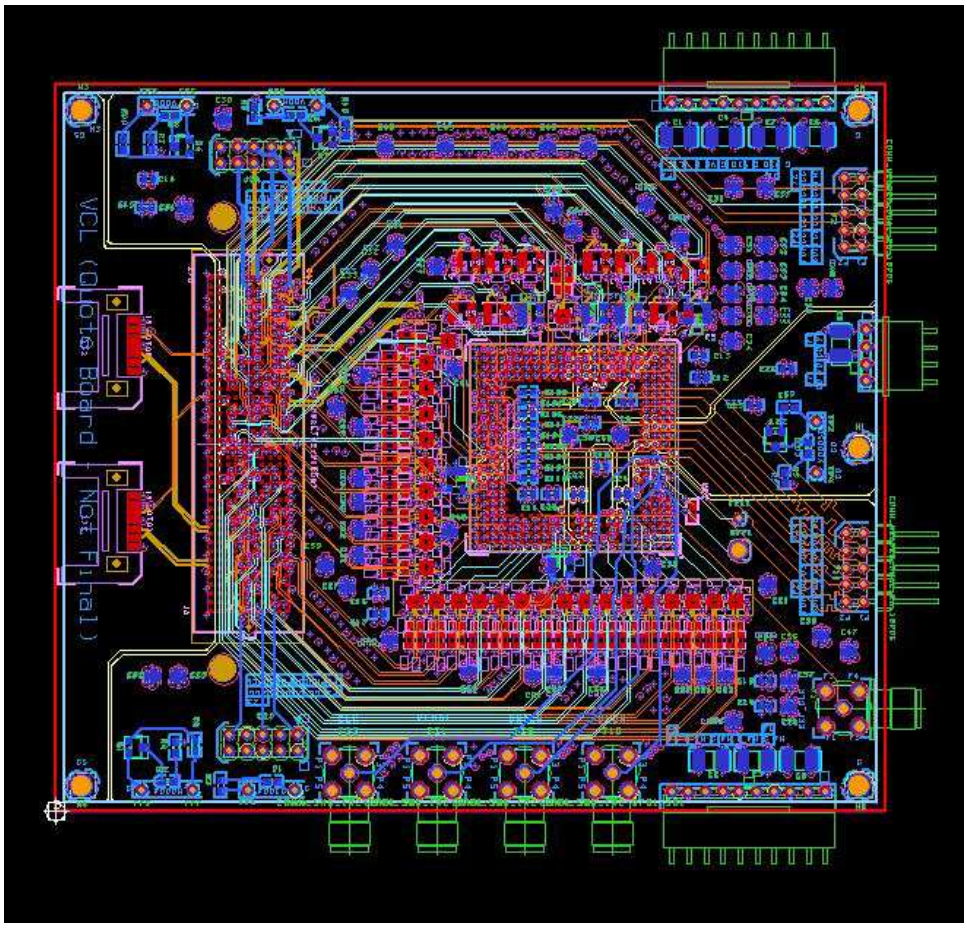


Figure 3.6: The layout view of the third version of the new daughter card — testboard2-v3 — phase one (Quoted Version)

3.3.4 Testboard2-v3

The final version of the PCB is done in two phases. The first phase is considered pre consultants changes (Quoted Board) — Figure 3.6, and the second phase is considered as post consultants changes (Fabricated Board) — Figure 3.7 on page 29.

3.3.4.1 Phase One

One of the big issues found in the previous designs of the PCB was the voltage difference between the FPGA board's High Precision (HP) single ended drivers/receivers operating at maximum of 1.8 V and the AsAP2 single ended drivers/receivers operating at 2.5 V. This different in voltage values, results in some major changes in the design. There are two possible solutions to this issue. The first possible solution is the use of voltage level shifters to change the voltage levels to the desired voltage values. The second possible solution is to use LVDS drivers and receivers that generate signals according to the LVDS standard and are available on both sides.

After some search on different web sites based on bit rate of the voltage level shifters and differential drivers/receivers the following result was gathered.

From the Table 3.4, the maximum bit rate for an industrial voltage level shifter between 1.8 V and 2.5 V is about 320 Mbps (fifth Item on the table), and since the input and outputs of AsAP2 require to transfer the clock as well as data and control signals, only maximum operating clock frequency of 160 MHz can be achieved using voltage level shifters.

Based on Table 3.5 the maximum bit rate achievable by a LVDS transmitter at 2.5 V is 630 Mbps. Also the maximum bit rate achievable by a LVDS receiver is 500 Mbps. Considering the clock signal in both cases, a clock frequency of 315 MHz for the LVDS transmitter and 250 MHz for LVDS receiver is achievable. Also since LVDS signals swing at low differential voltage of 250 mV with the common voltage of 1.2 V, voltage level change from 2.5 V to 1.8 V is not required with a LVDS signal pair. However, two signal traces instead one is required to transmit each signal value. These two signals are complementary of each other and are labeled as _p and _n for positive and negative respectively.

Based on these results the LVDS drivers and receivers were used in implementation of the final version of the PCB. This decision also enforced the reduction in the number of possible PCBs connecting to the FPGA board because the increase in the number of trace signals used on the FMC connector has doubled. This increase disables the capability of connecting two daughter cards stacking up on top of each other that resulted in removal of the female FMC connector from the daughter card. In addition to the previous solution, there is also the possibility of converting each signal to LVDS at AsAP2 side and then back to single ended right before entering the FMC connector and vise versa to keep the possibility of stacking up two daughter cards on top of each other, but due to increase in the number of components this option was not implemented.

In the new design *test_out* pins also require special considerations. Previously these pins

Table 3.4: Level shifters

#	MFG	Model	Bit Rate (Mbps)			Bits	Dimension (mmxmm)	Distr	Pin Count	Link
			to 1.8 V	to 2.5 V	Max					
1	MAXIM	MAX13055E MAX13058E	100	100	100	8		Digi-key	24–28	Manual
2	ST	ST4G3234	210	260	380	4	2.04×1.41	Digi-key	11	Manual
3	ST	ST1G3234	210	260	380	1	1.36×1.02	Digi-key	5	Manual
4	ON Semi	NLSV4T244				4	1.70×2.00	Digi-key	12	Manual
5	TI	SN74AVCH2T45	320	320	500	2	1.918×.918 BG	Digi-key	8	Manual
6	ON Semi	NLSV8T244				8	4.00×2.00	Digi-key	20	Manual
7	ON Semi	NLSV4T3234				4	1.41×2.04	Digi-key	11	Manual
8	TI	TXB0104	60	60	100	4	1.89×1.39	Digi-key	12	Manual
9	TI	TXB0106-Q1	60	60	100	6	5.10×6.60	Digi-key	16	Manual
10	TI	SN74AVC2T45	320	320	500	2	4.25×3.15	Digi-key	8	Manual
11	Maxim	MAX3002	30	30	35	8	6.55×6.60	Digi-key	20	Manual
12	ON Semi	NLSX3014	100	100	100	4	1.70×2.00	Digi-key	12	Manual
13	TI	SN74LVC1T45	75	140	420	1	1.418×.918 BG	Digi-key	6	Manual
14	ST	ST2378E			13	8	2.46×1.98	Digi-key	20	Manual
15	TI	TXS0102			24	2	1.918×.918	Digi-key	8	Manual
16	TI	SN74AVC4T245	200	200	380	4	2.65×1.85	Digi-key	16	Manual
17	ON Semi	NLSX3012	140	140	140	2	1.80×1.20	Digi-key	8	Manual
18	TI	SN74AVCH1T45	200	200	380	1	1.418×.918	Digi-key	6	Manual
19	TI	SN74AVC1T45	320	320	500	1	1.418×.918	Digi-key	6	Manual
20	ON Semi	NLSX5011	100	140	100	1	1.2×1.0	Digi-key	6	Manual
21	TI	TXS0108E	60	60	60	8	3.10×2.60 BG	Digi-key	20	Manual
22	TI	SN74AVC2T45-Q1	320	320	500	2	2.10×3.20	Digi-key	8	Manual
23	TI	SN74LVC1T45	75	140	420	1	1.418×.918	Digi-key	6	Manual
24	TI	SN74AVC2T245	320	320	500	2	1.85×1.45	Digi-key	10	Manual
25	ON Semi	NLSX3013	100	100	100	8	2.03×2.54	Digi-key	20	Manual
26	TI	TXS0104E	24	24	24	4	1.89×1.39	Digi-key	12	Manual
27	TI	TXS0101	21	21	24	1	1.418×.918	Digi-key	6	Manual
28	ON Semi	NLSX3018	100	100	100	8	4.00×2.00	Digi-key	20	Manual

Table 3.5: LVDS drivers/receivers

#	MFG	Model	Bit Rate (Mbps)			Bits	Dimension (mmxmm)	Distr	Pin Count	Link
			to 1.8 V	to 2.5 V	Max					
1	TI	SN65LVDS1	630	630	630	1	3.05×3.00	Digi-key	5	Manual
2	TI	SN65LVDS4	500	500	500	1	2.35×1.85	Mouser	10	Manual

were routed to a right angle connector as well as FMC connector. This could increase the capacitance and inductance of these traces by generating stubs in the path of these signals. In addition, due to voltage difference between the AsAP2 chip and the FPGA board, a direct connection to the FMC connector is not feasible anymore. The follow is the list of the possible solutions.

1. Routing the *test_out* pins only differentially to the FMC connector
2. Routing the *test_out* pins using only single ended wires to a right angled connector (selected option)
3. Routing the *test_out* pins using single ended wires to a connector and then differentially to the FMC connector
4. Routing the *test_out* pins using single ended wires to a right-angled connector and then to the FMC connector after being voltage level shifted

The first option wasn't implemented because the *test_out* pins are normally used for testing purposes. These signals get connected to the oscilloscope in most cases and sending that signal only to the FPGA board isn't a good choice for this design.

The second option was implemented due to ease of implementation and ease of use for testing. Only a series termination to 50 Ω resistance is required for this option, and *test_out* signals can be probed with oscilloscope or any other device.

The third option wasn't selected because of the introduction of an additional capacitance and inductance added to these signals. The increase in capacitance and inductance is due to the long stubs created by the right-angled connector on the single ended signals that cause reduction in reliability of these signals.

The fourth option wasn't also selected since it requires an additional 1.8 V power plane with all the required connectors and decoupling capacitors to be added to the existing design. This implementation would increase the number of required parts in the design and also the cost of the design.

The next modification is the orientation of AsAP2 chip. The orientation is changed to the old orientation used in the first version of the testboard2 because the positive and the negative LVDS signals are required to be coupled together closely and any via or separation between the two signals degrade he signal integrity and reliability of the signal by introducing differential noise.

Two straight connectors and one right angle connector are added to the design at this stage. The right angled connector is used to bring out the Test Data In (*TDI*), Test Data Out (*TDO*),

Test Mode Select (TMS), and Test ClocK (TCK) out of FPGA board and accessible for testing. For this project the *TDI* signal should be shorted to the *TDO* signal for connectivity of the Joint Test Action Group (JTAG) signal.

Pins PG_M2C, PRSNT_M2C_L, and PG_C2M from the FMC connector have been brought to one of the straight connectors next to ground to be shorted using a jumper in order to select different selections explained shortly. These signals can be connected to GNDG plane using a jumper while they are internally connected to a pull up resistor connecting to 3.3 V (at 3.3 V with out a jumper to GNDG). The shorted PG_M2C shows that the board is defective while shorted PRSNT_M2C_L switches off the JTAG by pass between *TDI* and *TDO*, so the *TDI*, *TDO*, TMS, and TCK can be used for JTAG Chaining. The PG_C2M is an output pin, but it has brought to a pin in case it is required in the future with the FPGA board while it can be left alone for this project. These are based on page 23 of Xilinx VC709 Evaluation Board for the Virtex-7 FPGA User Guide [35] and VC709 board schematics [36].

The other straight connector is used for power connection. VIO_B_M2C connections are input power connections that should be connected to 1.8 V to provide power to the HB bank (bank 35) of the Virtex 7 chip. These connections are directed to the connector pins to be connected using a jumper to VADJ (1.8 V) that are being directed to a same connector from the FMC connector.

3.3.4.2 Phase Two

After a design review and the consultant recommendations, the final modifications were done on the design. Figure 3.7 displays the final layout view of this design. The final signal layer gerber view of this design is available at Appendix A.

In this design almost all of the Silk Screen drawings and references were modified to have a clearer representation of each signal. The LVDS signals were distanced such a way that they have minimal cross talk between each pair. The *test_out* traces were length matches, so they can be easily connected to oscilloscope for measurement.

The straight connector connecting the FPGA power signal VIO_B_M2C to 1.8 V was replaced with Zero ohm resistors. This was done because the jumper implementation using the straight connector would introduce undesired inductance, and it would reduce the reliability and functionality of the FPGA board bank.

The right-angled connector was removed and the *TDI* signal of the FPGA JTAG signal was shorted to the *TDO* signal of the FPGA using a zero ohm resistor instead of a jumper to increase

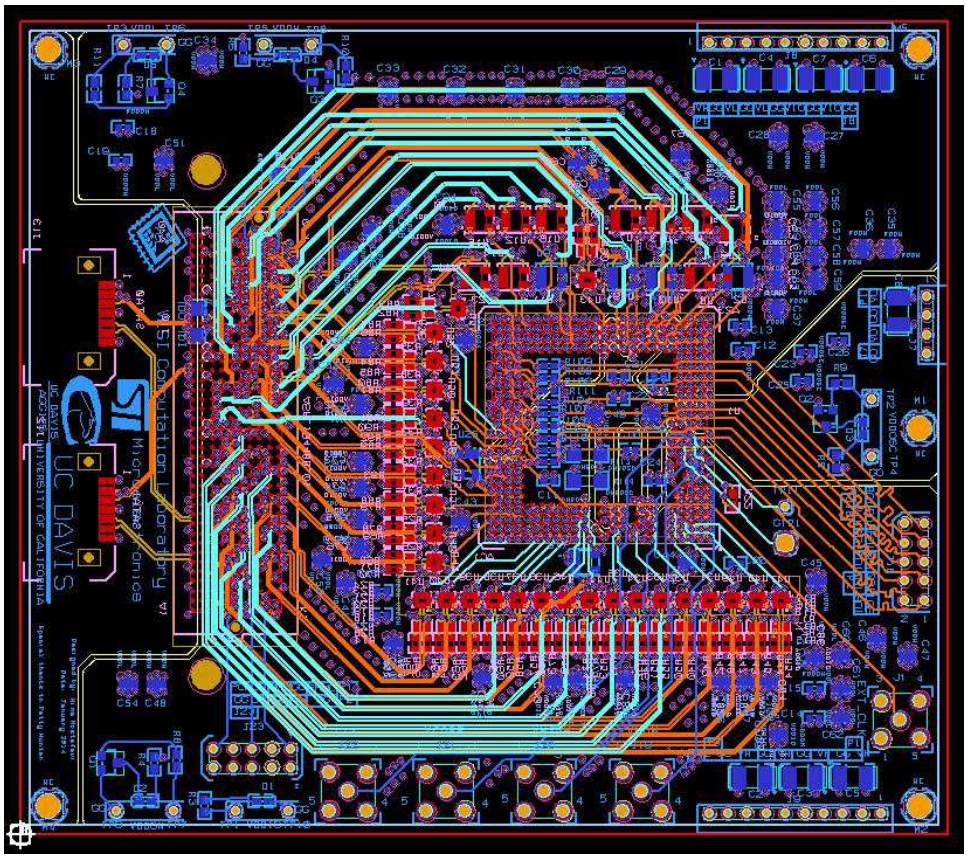


Figure 3.7: The layout view of the third version of the new daughter card — testboard2-v3 — phase two (Fabricated Version)

the reliability of this signal.

3.3.5 Other Design Considerations and Factors

As previously mentioned FPGA supply runs at 1.8 V while AsAP2 IO runs at 2.5 V in order to compensate for this difference LVDS drivers and receivers have been used. These LVDS Integrated Circuit (IC) chips run at 2.5 V using *VDDIO* voltage, and because of increase in the number of components on *VDDIO* the number of X2Y bypass Capacitors for VDDIO has been doubled to compensate for the additional load due to LVDS ICs connected to *VDDIO* in the final design. The failsafe resistors connecting the LVDS signals to *VDDIO* and *GNDG* before the LVDS receivers are picked based on page 13 of TI SN65LVDS4 data sheet [37] where *VCC* is equal to 2.5 V.

The 100 Ω differential and the 50 Ω single ended daughter card trace resistance matching are picked based on the 50 Ω impedance matching on the FPGA board. All the inputs and outputs

connection on the FMC connector of VC709 are HP type, but they don't have the capability of using Digitally Controlled Impedance (DCI) drivers and receivers. They are only matched to 50 Ω.

The *test_out*, 5.9 Ω, serial terminations resistors are picked assuming the *test_out* drivers are 44 Ω. These resistors can be replaced by shorting wires in case these values come out to be in accurate while a 50 Ω matching active probe can be used on the scope to compensate for the 50 Ω mismatching issues. The 6 Ω resistance is calculated based of given assumption in the AsAP2 drivers as follow:

$$LoadCapacitance = C\_L = 25\ \text{pF}(drivers) + 15\ \text{pF}(trace) + 13\ \text{pF}(scope) + 10\ \text{pF}(probe) + 17\ \text{pF}(forcompensationofprobeandviascapacitance) = 80\ \text{pF} \quad (3.8)$$

$$LowtoHighPropagationDelay = Tp\_LH = .69 \times Reqp \times C\_L \quad (3.9)$$

$$HightoLowPropagationDelay = Tp\_HL = .69 \times reqn \times C\_L \quad (3.10)$$

Equations 3.9 and 3.10 are based on Digital Integrated Circuits — A Design Perspective by Jan M. Rabaey and etl. [38].

$$AverageTp\_LH = Tp\_LH\_avg = (2.1\ \text{ns} + 2.8\ \text{ns})/2 = 2.45\ \text{ns} \quad (3.11)$$

$$AverageTp\_HL = Tp\_HL\_avg = (1.9\ \text{ns} + 2.8\ \text{ns})/2 = 2.35\ \text{ns} \quad (3.12)$$

The values used in Equations 3.11 and 3.12 are based on the classified document accessible by request from VLSI Computational Lab (VCL).

$$AveragePropagationDelay(bothHLandLH) = Tp\_avg = (2.45\ \text{ns} + 2.35\ \text{ns})/2 = 2.4\ \text{ns} \quad (3.13)$$

$$DriverResistance = R\_Tp\_avg = tp\_avg/(.69{*}C\_L) = 2.4\ \text{ns}/(.69{*}80\ \text{pF}) = 43.478 \approx 44\ \Omega \quad (3.14)$$

Based on the final result of 44 Ω for AsAP2 driver output resistance, in order to match to 50 Ω transmission line resistance, a 6 Ω resistor in series with the transmission line is required.

Table 3.6: Length matching values for testboard2-v3

Signal Name	Single Ended Length (mil)	Single Ended Delay (ns)	Diff_P Length (mil)	Diff_N Length (mil)	Delay Difference(ns)	Total Length (mil)	Total Delay (ns)	Steps to One Decimal	Steps (Decimal)	Step Skew (Decimal)	Steps Skew (Binary)
CONFIG SIGNAL LENGTHS											
RESET_COLD	617.55	0.1	1194.58	1194.85	0.2	1812.4	0.3	3.6	4		
RESET_ COUNTCLK	664.32	0.11	1205.95	1205.75	0.2	1870.27	0.31	1.9	2		
SPI_CLK	491.59	0.08	1345.97	1346.62	0.23	1838.21	0.31	1.9	2	15	01111
SPI_LOAD	400.04	0.07	1368.94	1368.16	0.23	1768.98	0.3	2.1	2	15	01111
SPI_MISO	275.45	0.04	1437	1437.36	0.25	1712.81	0.29	2.3	2	15	01111
SPI_MOSI	411.36	0.07	1377.26	1377.47	0.23	1788.83	0.3	2.1	2	15	01111
SPI_SEL	384.08	0.06	1335.56	1335.37	0.23	1719.64	0.29	2.3	2	15	01111
CFG_CLK	746.51	0.13	1660.49	1660.03	0.28	2407	0.41	0	0	13	01101
CFG_VALID	693.99	0.12	1228.38	1228.61	0.21	1922.6	0.33	1.5	2	15	01111
TEST OUT SIGNAL LENGTHS											
TEST_OUT0	2715.28	0.41	0	0	0	2715.28	0.41	0.6	1		
TEST_OUT1	2717.47	0.42	0	0	0	2717.47	0.42	0.4	0		
TEST_OUT2	2700.53	0.44	0	0	0	2700.53	0.44	0	0		
TEST_OUT3	2805.96	0.4	0	0	0	2805.96	0.4	0.8	1		
TEST_OUT4	2717.12	0.44	0	0	0	2717.12	0.44	0	0		
TEST_OUT5	2732	0.41	0	0	0	2732	0.41	0.6	1		
TEST_OUT6	2704.99	0.43	0	0	0	2704.99	0.43	0.2	0		
TEST_OUT7	2730.09	0.42	0	0	0	2730.09	0.42	0.4	0		
TEST_OUT8	2766.14	0.42	0	0	0	2766.14	0.42	0.4	0		
IN DATA SIGNAL LENGTHS											
IN_CLK	425.11	0.07	2846.15	2844.19	0.49	3271.26	0.56	8.7	9	15	01111
IN_DATA0	704.45	0.12	5188.16	5187.85	0.89	5892.61	1.01	0	0	6	00110
IN_DATA1	671.24	0.11	4782.63	4783.29	0.82	5454.53	0.93	1.5	2	8	01000
IN_DATA2	522.6	0.09	4422.98	4422.46	0.76	4945.58	0.85	3.1	3	9	01001
IN_DATA3	577.93	0.1	3999.23	4000.63	0.69	4578.56	0.79	4.2	4	10	01010
IN_DATA4	515.8	0.09	3558.86	3558.5	0.61	4074.66	0.7	6	6	12	01100
IN_DATA5	454.72	0.08	4627.23	4626.69	0.8	5081.95	0.88	2.5	3	9	01001
IN_DATA6	377.19	0.06	4216.66	4217.8	0.73	4594.99	0.79	4.2	4	10	01010
IN_DATA7	405.31	0.07	3857.61	3856.89	0.66	4262.92	0.73	5.4	5	11	01011
IN_DATA8	546.66	0.09	3136.21	3135.49	0.54	3682.87	0.63	7.3	7	13	01101
IN_DATA9	435.78	0.07	2734.46	2734.74	0.47	3170.52	0.54	9	9	15	01111
IN_DATA10	504.19	0.08	2514.1	2514.77	0.43	3018.96	0.51	9.6	10	16	10000
IN_DATA11	575.53	0.1	1988.79	1988.91	0.34	2564.44	0.44	11	11	17	10001
IN_DATA12	649.8	0.11	1757.67	1757.13	0.3	2407.47	0.41	11.5	12	18	10010
IN_DATA13	626.99	0.11	1659.08	1657.95	0.28	2286.07	0.39	11.9	12	18	10010
IN_DATA14	527.3	0.09	1078.65	1078.66	0.18	1605.96	0.27	14.2	14	20	10100
IN_DATA15	445.48	0.07	1081.1	1081.63	0.18	1527.11	0.25	14.6	15	21	10101
IN_REQUEST	295.88	0.05	3064.23	3064.44	0.53	3360.32	0.58	8.3	8	14	01110
IN_VALID	335.33	0.06	3328.55	3328.76	0.57	3664.09	0.63	7.3	7	13	01101

Table 3.7: Length matching values for testboard2-v3 (continues)

Signal Name	Single Ended Length (mil)	Single Ended Delay (ns)	Diff_P Length (mil)	Diff_N Length (mil)	Delay Difference(ns)	Total Length (mil)	Total Delay (ns)	Steps to One Decimal	Steps (Decimal)	Step Skew (Decimal)	Steps Skew (Binary)
OUT DATA SIGNAL LENGTHS											
OUT_CLK	382.65	0.06	2536.72	2536.05	0.44	2919.37	0.5	7.7	8	15	01111
OUT_DATA0	611.1	0.1	4636.39	4636.31	0.8	5247.49	0.9	0	0	7	00111
OUT_DATA1	478.5	0.08	4045.68	4046.25	0.7	4524.75	0.78	2.3	2	9	01001
OUT_DATA2	708.05	0.12	4282.32	4281.78	0.74	4990.37	0.86	0.8	1	8	01000
OUT_DATA3	475.01	0.08	3656.3	3656.31	0.63	4131.32	0.71	3.7	4	11	01011
OUT_DATA4	471.45	0.08	3222	3222.57	0.56	3694.02	0.64	5	5	12	01100
OUT_DATA5	494.52	0.08	3662.33	3663.74	0.63	4158.26	0.71	3.7	4	11	01011
OUT_DATA6	613.25	0.1	3303.45	3303.09	0.57	3916.7	0.67	4.4	4	11	01011
OUT_DATA7	554.4	0.09	2889.05	2888.74	0.5	3443.45	0.59	6	6	13	01101
OUT_DATA8	573.33	0.1	3034.5	3034.39	0.52	3607.83	0.62	5.4	5	12	01100
OUT_DATA9	415.03	0.07	1740.43	1739.21	0.28	2155.46	0.35	10.6	11	18	10010
OUT_DATA10	397.53	0.07	2497.76	2498.63	0.43	2896.16	0.5	7.7	8	15	01111
OUT_DATA11	539.9	0.09	1988.89	1989.44	0.34	2529.34	0.43	9	9	16	10000
OUT_DATA12	587.53	0.1	1887.58	1885.98	0.32	2475.11	0.42	9.2	9	16	10000
OUT_DATA13	274.24	0.05	1633.57	1633.76	0.28	1908	0.33	11	11	18	10010
OUT_DATA14	571.7	0.1	1708.81	1709.66	0.29	2281.36	0.39	9.8	10	17	10001
OUT_DATA15	435.75	0.07	1473.59	1473.71	0.23	1909.46	0.3	11.5	12	19	10011
OUT_REQUEST	303.52	0.05	2663	2663.71	0.45	2967.23	0.5	7.7	8	15	01111
OUT_VALID	429.32	0.07	2946.82	2945.49	0.51	3376.14	0.58	6.2	6	13	01101
SATA SIGNAL LENGTHS											
SATA0_C2M	0	0	536.66	536.53	0.09	536.66	0.09				
SATA0_M2C	0	0	1292.8	1292.45	0.19	1292.8	0.19				
SATA1_C2M	0	0	972.5	972.2	0.17	972.5	0.17				
SATA1_M2C	0	0	537.06	537.74	0.08	537.74	0.08				
FPGA SIGNAL LENGTHS											
TDI	165.28	0.84	0	0	0	165.28	0.84				
TDO	179.98	0.86	0	0	0	179.98	0.86				
TEST SIGNALS LENGTHS											
VCORE	2746.5	0.41	0	0	0	2746.5	0.41				
VHIGH	2189.64	0.32	0	0	0	2189.64	0.32				
VLOW	1965.39	0.29	0	0	0	1965.39	0.29				
ASAP_CLK	3223.75	0.48	0	0	0	3223.75	0.48				
EXTERNAL_CLK	2850.35	0.49	0	0	0	2850.35	0.49				

Table 3.8: Testboard2-v3 trace lengths

	Max Length (mil)	Min Length (mil)	Difference	Time* (ps)
Config	2407	1713	694.19	0.12
Test out	2805.96	2701	105.43	0.04
Input	5892.61	1527	4365.5	0.76
Output	5247.49	1908	3339.5	0.6
SATA	1292.8	536.7	756.14	0.11
FPGA	179.98	165.3	14.7	0.02
Test	3223.75	1965	1258.4	0.2

*The Time column values are based on 170 psec/in traces.

There are no matching resistors between the LVDS ICs and AsAP2 pins since these traces are not considered transmission lines. The validity of this statement is based on Equation 3.15 which is a simple rule of thumb to show whether a trace is a transmission line with the requirement of resistance matching or not.

$$Length\ Of\ Trace \times \frac{Clock\ Frequency}{Trace\ Velocity} \geq 0.02 \tag{3.15}$$

With velocity of 146,385,010.9 m/s based on Equation 3.16 and maximum trace length of 750 mil or 0.01905 m running at 315 MHz, the left side of equation equal to 0.02 which is equal to the right side of the equation which is 0.02. Since all traces are kept shorter than 750 mils, these traces won't act as transmission line and won't require resistance matching. Tables 3.6 and 3.7 display the delay and trace length for all the main signals on the PCB while Table 3.8 displays the maximum and minimum trace length and their difference in length and delay for each signal group.

$$TraceVelocity = \frac{Speedoflight}{\sqrt{Er}} = \frac{3 \times 10^8}{\sqrt{4.2}}\ \frac{\text{m}}{\text{sec}} = 146385010.9\ \frac{\text{m}}{\text{sec}} \tag{3.16}$$

All clock signals on AsAP2 input, output, and *config* Signals have been connected to CC (Clock Capable) pins. The *out_clk*, *in_clk*, and *spi_clk* have been connected to MRCC (Multi Region Clock Capable) while the *cfg_clk* is connected to SRCC (Single Region Clock Capable) pin (this was due to unavailability of MRCC pins on that bank). The CC pins are used for clock signals based on page 29 of Xilinx, 7 Series FPGAs Clocking Resources [39], User Guide, where it is explained why all input clock signals are required to be connected to CC pins.

All the *out_data* signals and the out clock signal are connected to HB bank (bank 36). All the *in_data* signals and the in clock signal are connected to HA bank (bank 35). Finally, all the AsAP2 programming and configuration signals and their clocks have been connected to bank LA17–

Table 3.9: The FMC connector pin locations connecting to the AsAP2 signals

	MALE FMC CONNECTOR										
	K	J	H	G	F	E	D	C	B	A	
1	NC	GNDG	NC	GNDG	PG_M2C	GNDG	PG_M2C	GNDG	NC	GND_SATA	1
2	GNDG	NC	PRSNT_M2C_L	NC	GNDG	NC	GNDG	NC	NC	NC	2
3	GNDG	NC	GNDG	NC	GNDG	NC	GNDG	NC	NC	NC	3
4	NC	GNDG	NC	GNDG	IN_CLK_P	GNDG	NC	GNDG	NC	GND_SATA	4
5	NC	GNDG	NC	GNDG	IN_CLK_N	GNDG	NC	GNDG	NC	GND_SATA	5
6	GNDG	IN_DATA11_p	GNDG	NC	GNDG	IN_DATA4_P	GNDG	NC	NC	S1_M2C_P	6
7	IN_DATA12_P	IN_DATA11_N	NC	NC	IN_VALID_P	IN_DATA4_N	GNDG	NC	NC	S1_M2C_N	7
8	IN_DATA12_N	GNDG	NC	GNDG	IN_VALID_N	GNDG	NC	GNDG	NC	GND_SATA	8
9	GNDG	IN_DATA10_P	GNDG	NC	GNDG	IN_DATA3_P	NC	GNDG	NC	GND_SATA	9
10	IN_DATA13_P	IN_DATA10_N	NC	NC	IN_REQUEST_P	IN_DATA3_N	GNDG	NC	NC	S0_M2C_P	10
11	IN_DATA13_N	GNDG	NC	GNDG	IN_REQUEST_N	GNDG	NC	NC	NC	S0_M2C_N	11
12	GNDG	IN_DATA9_P	GNDG	NC	GNDG	IN_DATA2_P	NC	GNDG	NC	GND_SATA	12
13	IN_DATA15_P	IN_DATA9_N	NC	NC	IN_DATA7_P	IN_DATA2_N	GNDG	GNDG	NC	GND_SATA	13
14	IN_DATA15_N	GNDG	NC	GNDG	IN_DATA7_N	GNDG	NC	NC	NC	NC	14
15	GNDG	IN_DATA8_P	GNDG	NC	GNDG	IN_DATA1_P	NC	NC	NC	NC	15
16	IN_DATA14_P	IN_DATA8_N	NC	NC	IN_DATA6_P	IN_DATA1_N	GNDG	GNDG	NC	GND_SATA	16
17	IN_DATA14_N	GNDG	NC	GNDG	IN_DATA6_N	GNDG	SPI_CLK_P	GNDG	NC	GND_SATA	17
18	GNDG	NC	GNDG	NC	GNDG	IN_DATA0_P	SPI_CLK_N	NC	NC	NC	18
19	NC	NC	NC	NC	IN_DATA5_P	IN_DATA0_N	GNDG	NC	NC	NC	19
20	NC	GNDG	NC	GNDG	IN_DATA5_N	GNDG	NC	GNDG	NC	GND_SATA	20
21	GNDG	NC	GNDG	SPI_SEL_P	GNDG	NC	NC	GNDG	NC	GND_SATA	21
22	NC	NC	NC	SPI_SEL_N	NC	NC	GNDG	CFG_CLK_P	NC	NC	22
23	NC	GNDG	NC	GNDG	NC	GNDG	GNDG	CFG_CLK_N	NC	NC	23
24	GNDG	OUT_DATA14_P	GNDG	SPI_LOAD_P	GNDG	OUT_DATA0_P	NC	GNDG	NC	GND_SATA	24
25	OUT_DATA15_P	OUT_DATA14_N	NC	SPI_LOAD_N	NC	OUT_DATA0_N	GNDG	GNDG	NC	GND_SATA	25
26	OUT_DATA15_N	GNDG	NC	GNDG	NC	GNDG	NC	NC	NC	S1_C2M_P	26
27	GNDG	OUT_DATA12_P	GNDG	SPI_MOSI_P	GNDG	OUT_DATA2_P	NC	NC	NC	S1_C2M_N	27
28	OUT_DATA13_P	OUT_DATA12_N	SPI_MISO_P	SPI_MOSI_N	OUT_DATA8_P	OUT_DATA2_N	GNDG	GNDG	NC	GND_SATA	28
29	OUT_DATA13_N	GNDG	SPI_MISO_N	GNDG	OUT_DATA8_N	GNDG	TCK	GNDG	NC	GND_SATA	29
30	GNDG	OUT_DATA11_P	GNDG	NC	GNDG	OUT_DATA1_P	TDI	NC	NC	S0_C2M_P	30
31	OUT_DATA9_P	OUT_DATA11_N	RESET_COLD_P	NC	OUT_DATA5_P	OUT_DATA1_N	TDO	NC	NC	S0_C2M_N	31
32	OUT_DATA9_N	GNDG	RESET_COLD_N	GNDG	OUT_DATA5_N	GNDG	NC	NC	NC	GND_SATA	32
33	GNDG	OUT_DATA10_P	GNDG	NC	GNDG	OUT_DATA3_P	TMS	NC	NC	GND_SATA	33
34	OUT_REQUEST_P	OUT_DATA10_N	RESET_COUNTCLK_P	NC	OUT_DATA7_P	OUT_DATA3_N	NC	GNDG	NC	NC	34
35	OUT_REQUEST_N	GNDG	RESET_COUNTCLK_N	GNDG	OUT_DATA7_N	GNDG	GNDG	NC	NC	NC	35
36	GNDG	OUT_VALID_P	GNDG	NC	GNDG	OUT_DATA6_P	NC	GNDG	NC	GND_SATA	36
37	OUT_CLK_P	OUT_VALID_N	CFG_VALID_P	NC	OUT_DATA4_P	OUT_DATA6_N	GNDG	NC	NC	GND_SATA	37
38	OUT_CLK_N	GNDG	CFG_VALID_N	GNDG	OUT_DATA4_N	GNDG	NC	GNDG	NC	NC	38
39	GNDG	VIO_HB_M2C_J39	GNDG	VADJ_1P8V_G39	GNDG	VADJ_1P8V_E39	GNDG	NC	NC	NC	39
40	VIO_HB_M2C_K40	GNDG	VADJ_1P8_H40	GNDG	VADJ_1P8V_F40	GNDG	NC	GNDG	NC	GND_SATA	40
	K	J	H	G	F	E	D	C	B	A	

33 (bank 34) of the Virtex 7 board in order to be able to use local clock signal for these signals to meet the timings for all of these signals (based on Xilinx 7 Series FPGAs Clocking Resources [39]). Table 3.9 displays the FMC pin connection to AsAP2 chip signals after becoming differential. In this figure blue represent the input signals, and purple represent the output signals. All the programming signals are colored yellow, while green is used to represent the SATA connections.

The reason only one right-angled component comes out of the top side of the PCB on Figure 3.7 is that this side of the board is inaccessible when the board is installed inside the computer via the PCIe connection in a system interface Picoblade. The Top right, right-angled connector starts right where the PCB comes out of the designated desktop computer.

Two zero ohm resistors are used to connect the VADJ_1P8Vs voltages to the power input of VIO_HB_M2Cs with very wide traces to reduce the voltage drop in these connections.

As shown on Figure 3.8, the *GndCom*, *GndIO*, and *GndOsc* are separate from each other while they are only connected through a low resistance of substrate in the AsAP2 chip. All the

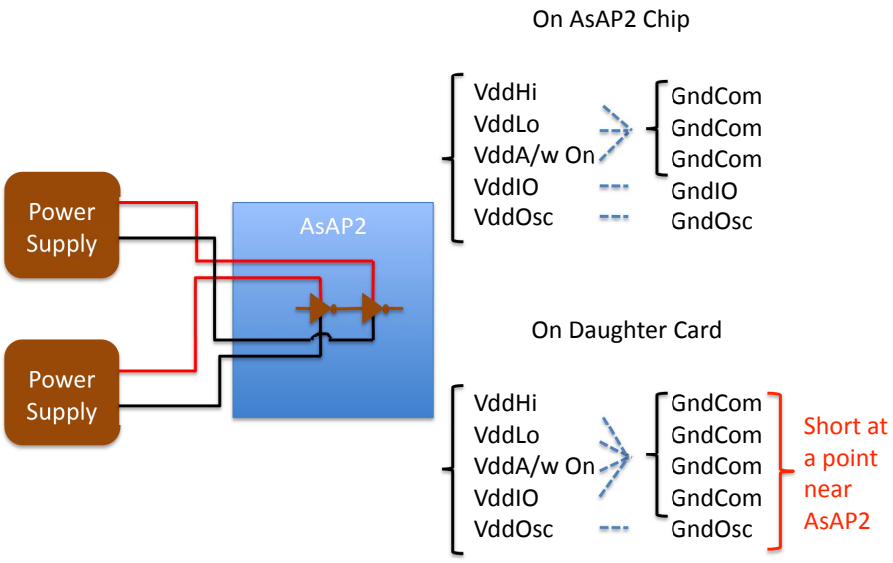


Figure 3.8: The schematic shows the ground connections on the AsAP2 and the fabricated daughter card

connections coming from *GndIO* and *GndCom* go to the same power plane while the *GndOsc* is connected to a separate plane on the PCB. There are two zero ohm resistors on the PCB connect the *GndOsc* to the *GndCom*. This has been implemented in this form to reduce the noise coupling of the oscillator circuit to the rest of the circuit by connecting the two grounds only at one low resistance point close to the AsAP2 chip. This reduces the current flow through the substrate while keeping the two grounds connected.

One of the potential big issues in the designed PCB is the issue of the bottled neck ground return current. Figure 3.9 displays this issue with a rectangle as the current coming from the right connector pins (displayed with arrows) can't be easily distributed through the bottle neck on the left side as is shown on this figure. An improvement to this potential problem has been shown on Figure 3.8. In this picture, the bottle necking via has been moved and returns current path has been opened (as shown on figure with a rectangle) the ground plane for the return current (opposite direction of the arrows).

With this potential issue, the fabricated board has been turned on using the power supplies. Using a bench voltmeter the voltage difference between the two grounds with out the FPGA connection has been tested, and it almost equal to zero. The exact voltage difference is 0.004 mV. Similar test is performed on a regular ground pins to a different ground pin on different locations of the board and voltage difference of 0.006 mV is observed, so the resistance between the two grounds

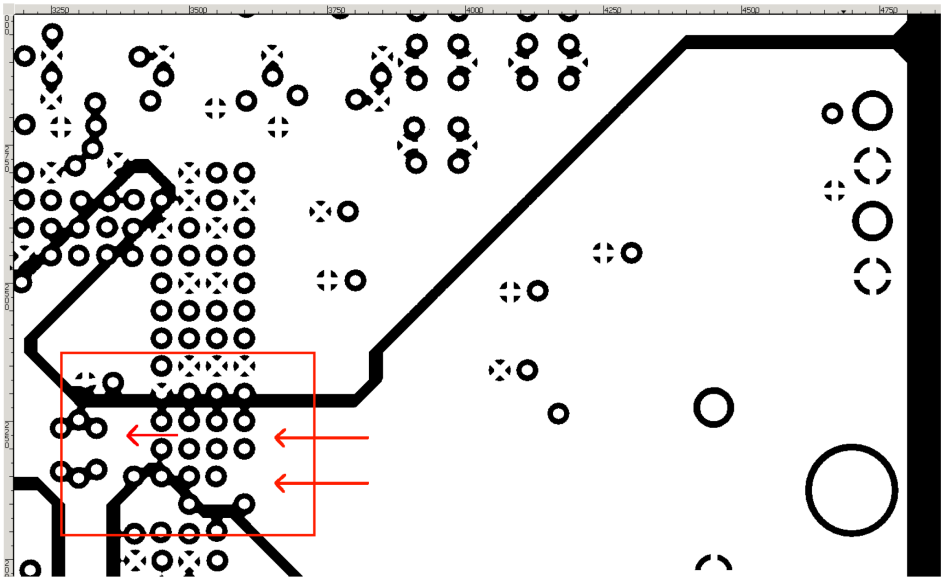


Figure 3.9: The fabricated board with potential issue

is minimal. Considering these supplies are all shorted on the supply side to common ground of the power strip and the Zero ohm resistor connecting the two different *GndCom* and *GndOsc*.

Inductive filters at the power source input to the PCB have been avoided due to the unique behavior of the AsAP2 chip. Globally Asynchronous Locally Synchronous (GALS) design of AsAP2 chip requires each processor to turn on and off with any desired frequency which result in burst of current surge from the power supplies in any time and at any order. If these current surges are not responded due to use of inductors in the inductive filters such as pi filters, current burst can get slowed which can result in drop in voltage on the processor requesting the current. For this reason in the design of the PCB inductive filters have been avoided and only decoupling capacitors have been used.

Lastly, a special consideration is requires in connecting the X2Y capacitor on the PCB. The via connection of these capacitors on the PCB is that each pad connects to the PCB using two via connections instead of just one which improves the performance of these capacitors by reducing the loop inductance of these capacitors. More information about X2Y capacitor connections can be found in "Get the Most from X2Y Capacitors with Proper Attachment Techniques" [40]. All the other capacitors have been connected to the PCB with one via close to the pad based on Figure 2–1 on page 22 of "7 Series FPGAs PCB Design and Pin Planning Guide" [41].

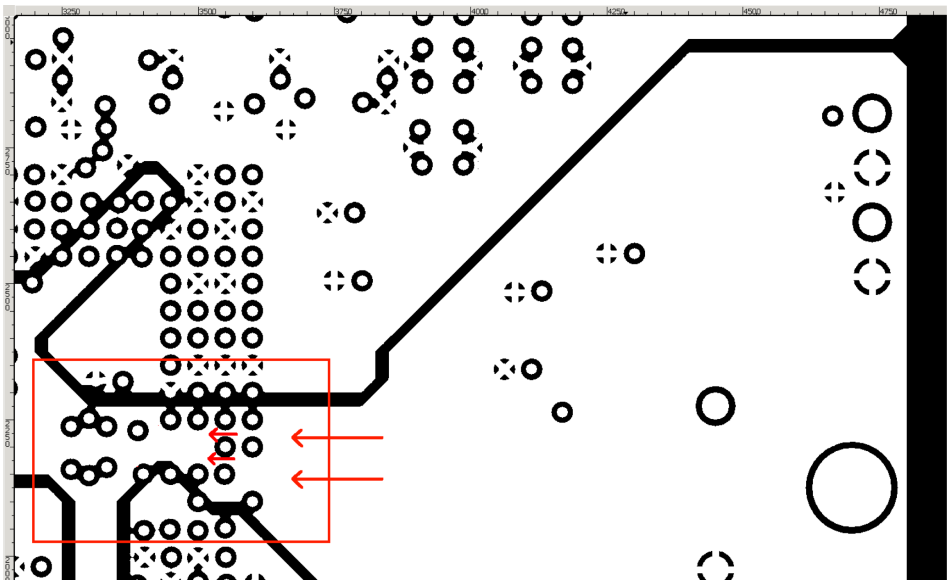


Figure 3.10: The design without a bottleneck

3.3.5.1 Design Consultant

The following are the recommendation and changes that the consultant has made to the board.

Getting placement guidelines from the fabrication company is one of the crucial points in designing a PCB. Another important point is that to SubMiniature Version B (SMB) routing should be more isolated from other routes and even shielded. This can be accomplished with a top backfill of the ground with stitching to give the trace routes a proper return path.

Spacing traces or at least trace bundles apart from each other by a 3–5 wide rule is crucial in signal integrity of the signals. Accordion loops are done to length match, a minimum rule of thumb is 5-wide rule in each loop and loop-to-loop is recommended. Typically the accordion patterns are grouped in tighter bundles. In order to length match on the 45-degree angles a trombone serpentine is preferred by adding a few loops longer along the angle.

Balance/Center placement of capacitors between SMBs to allow for finger or wrench tightening of at least 100 mils Is required for easy part access. All other placement (possibility of a package-to-package rules set up in the CES clearance) package rules might be desired. Tighter spacing is only possible for a “next gen” assembly house. That is usually quite a bit more expensive. If at all possible, getting the assembly house spacing standards and setting those up and seeing that the economical standard spacing is met is very important.

The components at the edge of a Ball Grid Array (BGA) chip would usually be placed in

200 mils distance, or they are placed in 100 mils in more relaxed versions. If at all possible sro-to-sro should be closer to 75 mils, leaving pad to pad at 50mils. Anything less is again considered "next gen" and commands more expensive processes.

Making sure the first pin for the components are specified with a dot or a small number on the silk screen next to the pad for small components is really important for future probing the pins after assembly. Also, labeling all the decoupling capacitors with the kind of *Vdd* they are connected to help for debugging the possible issues in the future. In general, all the components should be clearly labeled on silkscreen. Especially resistors should be labeled with their REFerence DEScription (REFDES) for debugging purposes.

3.4 Design Tools and Limitations

This board was design using the Mentor Graphics [42] tools. The "DxDesigner 2005" was the tool used to design the schematic view of the design. Appendix B on page 111 displays this schematic view. Bill of Materials (BOM) can be generated in this tool as well.

In order to insert new parts "Library Manager" is used. This tool gives different tool accesses to generate the required part from the schematic view to layout view of a part. "Symbol Editor" is used to create a symbolic view used in schematic view. The "Padstack Editor" can produce platted or non-platted holes and pads with different sizes and shape. These generated pads and holes get used in "Cell Editor" tool in the Library manager to generate the layout view of the part including the silkscreen, placement, and the assembly of the part. Finally "Part Editor" connects the cell (layout) view of the design to the symbolic (schematic) view of the part.

Finally, "Expedition PCB 2005" from Mentor Graphics tools was used to generate the layout view of the whole design. This tool can connect to DxDesigner to bring the connections and the layout view of the designed parts. In this tool "Constraint Editor" (also called CES) can be invoked to set the design rules such as minimum trace distances (clearance), differential pair assignments, trace grouping, and regional rules. The desired trace width size ranges can be set in this tool for each layer of the layout design. From the Constraint Editor the "Stackup" tool can be called to set the layer thickness and Dielectric constant (Er) of the layers as well as calculating the trace width required to achieve desired impedance. In Expedition PCB, the plane boundaries can be set as well as the part locations. Any additional silkscreen labeling and trace routing is done in Expedition PCB. Expedition PCB provides an automated routing option, but this normally is not used since it isn't as clean as hand routing the traces.

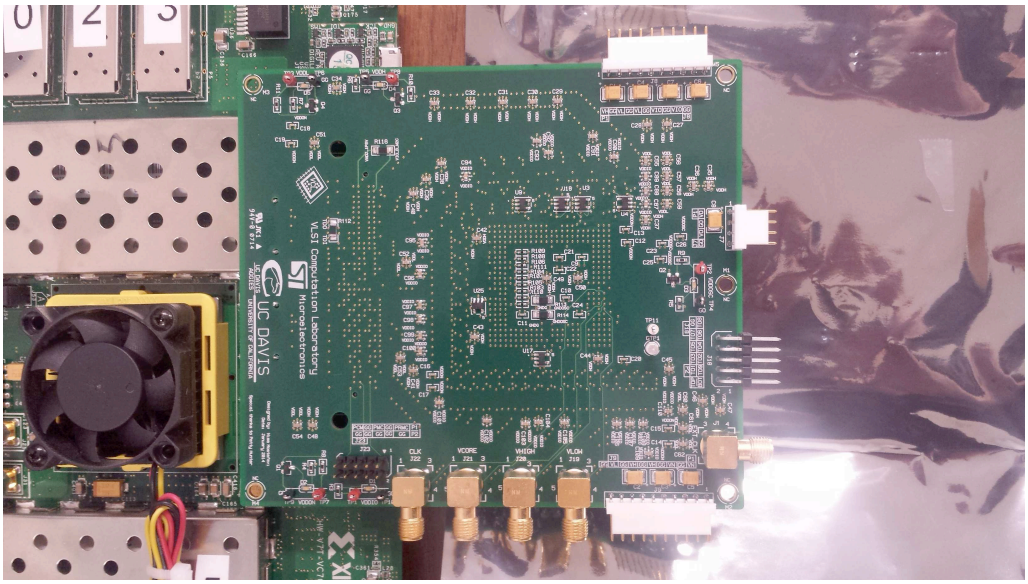


Figure 3.11: The top view of the fabricated board (testboard2-v3 — Phase 2) connected to the FPGA board

The "Forward and Backward Annotation" has special importance in the use of Mentor Graphics tools. The forward annotation forwards the changes done in DxDesigner to the Expedition PCB while the backward annotation forwards the changes done in the Expedition PCB back to the DxDesigner. In case this link gets broken sometimes text modifications in the schematic documentation or roll back in the design might be required.

Expedition PCB also can run DRC rule check on the design when the DRC rules are normally manually set using the fabrication company's "Capability Specs". In addition to DRC checker Expedition PCB can generate both "ODB++" and "Gerber" outputs depending on the fabrication company's Request. Finally, if the parts are being loaded by the fabrication company and they require the "Pick and Place for UBX" file to specify the location and the orientation of the part. The Expedition PCB can export this file in the "Generic AIS" format.

3.5 Final Board and Fabrication

Designed daughter card has been Fabricated and loaded by Green Circuits [34]. The top view of the design is shown on Figure 3.11. From the top view of the board all the connectors are visible. Figure 3.12 displays the bottom view of the fabricated board. In this view most of the crucial components of the board are visible such as the AsAP2 chip in the middle, the FMC connector next to the AsAP2 chip, and the two SATA connectors on the other side.

Figure 3.12: The bottom view of the fabricated board (testboard2-v3 — Phase 2)

All the surface mounted components on the board are loaded using reflow process. The Gerber files and the BOM file are sent to the Green circuits for the fabrication of the board with the following notes in a text file:

Board Outline.gdo	Board Outline for reference
EtchLayer1Top.gdo	Top Signal Layer
EtchLayer2.gdo	Signal 2
EtchLayer3.gdo	Signal 3
EtchLayer4.gdo	Signal 4
EtchLayer5.gdo	Signal 5
EtchLayer6.gdo	Signal 6
EtchLayer7.gdo	Signal 7
EtchLayer8.gdo	Signal 8
EtchLayer9.gdo	Signal 9
EtchLayer10.gdo	Signal 10
EtchLayer11.gdo	Signal 11
EtchLayer12Bottom.gdo	Bottom Signal Layer
GeneratedSilkscreenTop.gdo	Top Silkscreen
GeneratedSilkscreenBottom.gdo	Bottom Silkscreen

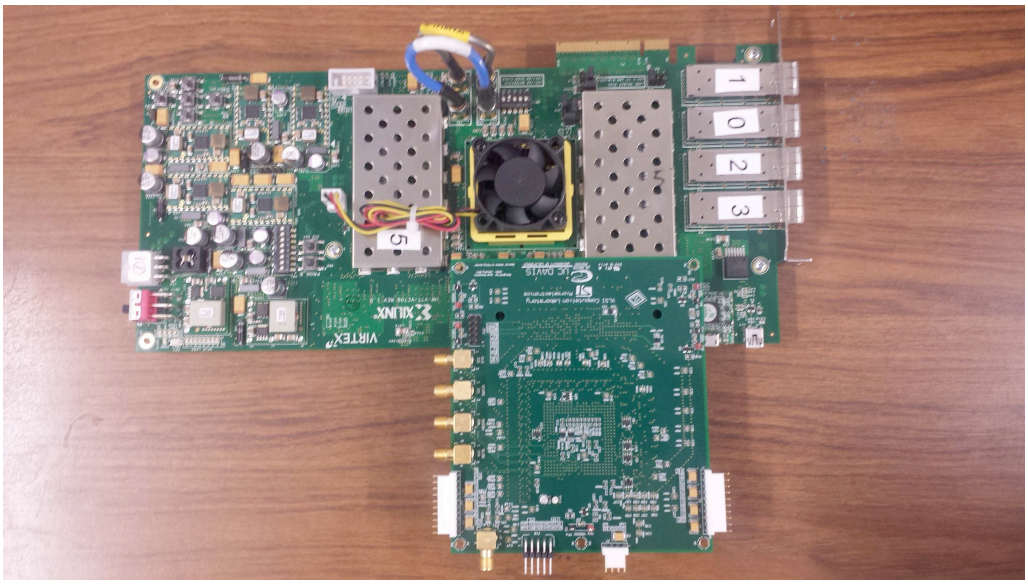


Figure 3.13: The picture shows the FPGA board and the connected daughter cards together

SoldermaskTop.gdo	Top solder mask
SoldermaskBottom.gdo	Bottom solder mask
SolderPasteTop.gdo	Top paste mask
SolderPasteBottom.gdo	Bottom paste mask

Drill Files

ThruHoleNonPlated	Non plated drills
ThruHolePlated	Plated drills

Contact information:
Name, Phone number, and email address

The first column of these notes points the name of the file while the second column described the file.

Green Circuits also requested for a pick and place file for loading the components in either text or excel format. The text format was generated and sent to the Green Circuits.

Figure 3.13 displays the final fabricated board loaded on VC 709 FPGA board.

Chapter 4

FPGA Verilog Code Design and Logic

The Xilinx Virtex-7 FPGA VC709 Connectivity Kit [31] featuring the "XC7VX690T - 2FFG1761C" FPGA is used for the AsAP2 setup with optical interconnect. Figure 4.1 displays a view of this board.

The VC709 FPGA board uses a USB-JTAG interface to load the bitfile of the desired verilog program. It can also use a Byte Peripheral Interface (BPI), a Parallel NOR Flash memory, to store the program, so the program can be reloaded to the FPGA board. The BPI can be used when the USB-JTAG interface of the board is not connected to a computer via a Universal Serial Bus (USB). This FPGA board contains two 4 GB Double Data Rate 3 (DDR3) memories running at 1866 Mbps. These memories can be accessed using internally programmed logic in the FPGA. This board has four Small Form-Factor Pluggable (SFP)/Enhanced Small Form-Factor Pluggable (SFP+) cages for optical transceivers which is one of the main reasons for selecting this FPGA board. A Universal Asynchronous Receiver-Transmitter (UART) to USB Bridge is also available on the VC709 FPGA board for other signaling purposes such sending and receiving data at slow speed.

Another important feature of this FPGA board is the "Peripheral Component Interconnect - Express" (PCI-E or PCIe) connection on this board that can be used to connect the system interface Picoblade to the host computer. Chapter 5 on page 56 describes this feature in more detail. The Xilinx FPGA board has a FMC connector to connect the AsAP2 daughter card to this board.

Originally, Vivado was used to write, synthesis, place and route (PAR), generate bitfile, and load the FPGA program into the FPGA, but most of the programs have been transferred to

Figure 4.1: The Xilinx Virtex-7 FPGA VC709 Connectivity Kit

run on Xilinx ISE software in the final phases of this projects due to the requirement of PCIe code to run on ISE for high reliability.

One design decision worth mentioning is the Jumper settings on the daughter card that allows the FPGA board *TDI* to be connected to the *TDO*. Based on Figure 4.2 from the "VC709 Evaluation Board for the Virtex-7 FPGA User Guide" [35] when *FMC1_HPC_PRSNT_M2C_B* (labeled as *PRMC* on the daughter card) is shorted to ground, switch U27 turns off. Since the *TDI* to TDO is already shorted on the PCB board, this connection still stays continuous.

4.1 Architecture and Testing Setup

The desired project has many parts that should be implemented on the FPGA board to get all the interfaces communicating with each other. This document only discusses the parts that have already been implemented or are in the testing phase. Figure 4.3 displays all the different parts for this project.

This chapter focuses on the AsAP2 interface side of the FPGA board while Chapter 5 briefly discusses how the PCIe interface has been brought up and used as a part of this project.

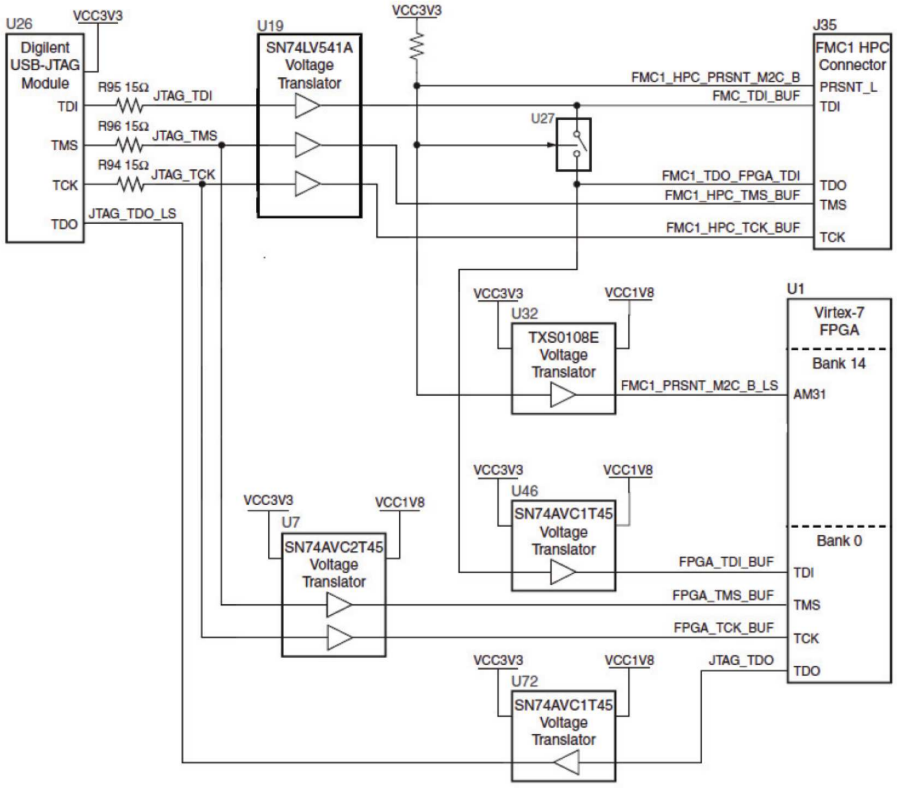


Figure 4.2: This schematic displays the *TDI*/TDO signal connections on the VC 709 FPGA board.

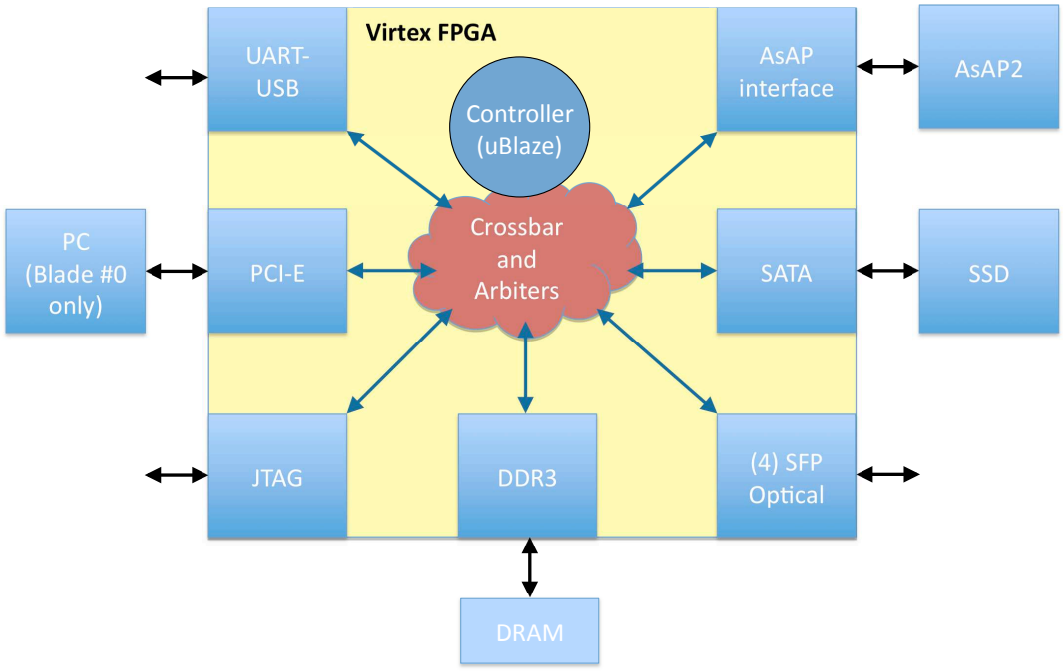


Figure 4.3: Schematic view of the internal architecture of the FPGA board.

4.2 Programming and Logic of AsAP2 Interface

In this design the FPGA board communicates to the AsAP2 daughter card through a FMC connector. The list of all the AsAP2 connections into FMC pins on the FPGA board and into the corresponding Virtex 7 chip pins for the physical constraint file is shown in Table 4.1.

In order to interface with the AsAP2 chip, two items require close attention. The first item is the programming interface to program AsAP2 and the second item is the input and output data interface.

4.2.1 AsAP2 Programmer

In order to program AsAP2, a part of the Block Random Access Memory (BRAM) on the FPGA has been assigned to hold the AsAP2 program and configuration file. Another part of BRAM is assigned to hold the run commands sent to the AsAP2 chip. Currently the verilog code to instantiate both blocks of memory has been written and simulated. The programming memory block has been written with data width of 64 bits and up to 30K address locations (15 bits for addressing) of program memory (*prog_mem* instance). The run or compute memory has been instantiated (prog_mem instance) with data width of 64 bits and up to 256 Memory locations (8 bits for addressing). There are three possible ways to modify the information on BRAM for this project, but only 2 have been implemented so far. The implemented methods are through JTAG (.coe input file mainly used in testing phases) and the PCIe interface mainly used after testing phase is done. The PCIe interface is only available to the system interface Picoblade in our system design. The third possible method that hasn't been implemented is the use of UART to modify the AsAP2 program and run commands on the BRAM.

For testing purposes, the functionality of the AsAP2 interface has been broken into the following modes and these are set using the switches on the FPGA board. Changing each switch position to a state can change this functionality to a state machine. The state transitions can be done in the order and one function at a time.

```
Switch 0: Calibration == 0, Run Mode == 1
Switch 1: Run Mode: Programming == 0, Compute == 1
          Calibration: Static Calibration == 0, Dynamic Calibration == 1
Switch 2: Calibration:
          Dynamic Calibration: Output Calibration == 0,
                               Input Calibration == 1
```

Table 4.1: Connections between the FPGA board pins and the AsAP2 pins.

FPGA Connections To the Bottom Daugther Card							
Signal	Symbol	Pin	TYPE	FMC Pin		U1* FPGA Pin	Direction from FPGA view
IN_DATA	DI	0	P	E18	HA20	B34	output
			N	E19		A34	
		1	P	E15	HA16	B39	
			N	E16		A39	
		2	P	E12	HA13	B36	
			N	E13		A37	
		3	P	E9	HA09	E32	
			N	E10		D32	
		4	P	E6	HA05	G32	
			N	E7		F32	
		5	P	F19	HA19	B32	
			N	F20		B33	
		6	P	F16	HA15	C33	
			N	F17		C34	
		7	P	F13	HA12	B37	
			N	F14		B38	
		8	P	J15	HA14	E37	
			N	J16		E38	
		9	P	J12	HA11	J37**	
			N	J13		J38**	
		10	P	J9	HA07	C38	
			N	J10		C39	
		11	P	J6	HA03	H33	
			N	J7		G33	
		12	P	K7	HA02	E33	
			N	K8		D33	
		13	P	K10	HA06	G36	
			N	K11		G37	
		14	P	K16	HA17_CC	C35	
			N	K17		C36	
		15	P	K13	HA10	H38	
			N	K14		G38	
IN_VALID	IV	--	P	F7	HA04	F34	output
			N	F8		F35	
IN_CLK	IC	--	P	F4	HA00_CC	E34	output
			N	F5		E35	
IN_REQUEST	IR	--	P	F10	HA08	J36	input
			N	F11		H36	
TEST_OUT	TO	0	--	--		--	--
		1	--	--		--	--
		2	--	--		--	--
		3	--	--		--	--
		4	--	--		--	--
		5	--	--		--	--
		6	--	--		--	--
		7	--	--		--	--
		8	--	--		--	--
RESET_COUNTCLK	RK	--	P	H34	LA30	V30	output
			N	H35		V31	
RESET_COLD	RC	--	P	H31	LA28	L29	output
			N	H32		L30	
TEST_CLK	TCK	--	--	D29		U19.14	output
TEST_DATA_IN	TDI	--	--	D30		U19.18	--
TEST_DATA_OUT	TDO	--	--	D31		T10	--
TEST_MODE_SELECT	TMS	--	--	D33		U19.17	output
VADJ_1P8V_E39	VJ	--	--	E39		--	output
VADJ_1P8V_F40	VJ	--	--	F40		--	output
FMC1_VIO_B_M2C_J39	VHB	--	--	J39		--	input
VDDH	VH	--	--	--		--	--
VDDIO	VIO	--	--	--		--	--
VDDOSC	VO	--	--	--		--	--
GNDG	GG/G	--	--	--		--	--

Signal	Symbol	Pin	TYPE	FMC Pin		U1 FPGA Pin	Direction from FPGA view
OUT_DATA	DO	0	P	E24	HB05	K27	input
			N	E25		J27	
		1	P	E30	HB13	P25	
			N	E31		P26	
		2	P	E27	HB09	H23	
			N	E28		G23	
		3	P	E33	HB19	L25	
			N	E34		L26	
		4	P	F37	HB20	P21	
			N	F38		N21	
		5	P	F31	HB12	K24	
			N	F32		K25	
		6	P	E36	HB21	P22	
			N	E37		P23	
		7	P	F34	HB16	N25	
			N	F35		N26	
		8	P	F28	HB08	H25	
			N	F29		H26	
		9	P	K31	HB10	M22	
			N	K32		L22	
		10	P	J33	HB15	M21	
			N	J34		L21	
		11	P	J30	HB11	K22	
			N	J31		J22	
		12	P	J27	HB07	G26	
			N	J28		G27	
		13	P	K28	HB06	K23	
			N	K29		J23	
		14	P	J24	HB01	H28	
			N	J25		H29	
		15	P	K25	HB00_CC	J25	
			N	K26		J26	
OUT_VALID	OV	--	P	J36	HB18	G21	input
			N	J37		G22	
OUT_CLK	OC	--	P	K37	HB17_CC	M24	input
			N	K38		L24	
OUT_REQUEST	OR	--	P	K34	HB14	J21	output
			N	K35		H21	
SPI_MOSI	SOI	--	P	G27	LA25	K29	output
			N	G28		K30	
SPI_MISO	SIO	--	P	H28	LA24	R30	input
			N	H29		P31	
SPI_SEL	SS	--	P	G21	LA20	Y29	output
			N	G22		Y30	
SPI_CLK	CS	--	P	D20	LA17_CC	L31	output
			N	D21		K32	
SPI_LOAD	SL	--	P	G24	LA22	R28	output
			N	G25		P28	
CFG_CLK	CC	--	P	C22	LA18_CC	M32	output
			N	C23		L32	
CFG_VALID	CV	--	P	H37	LA32	V29	output
			N	H38		U29	
PG_M2C	PMC	--	--	D1		AL32	input
PG_C2M	PCM	--	--	F1		AN34	output
PRSNT_M2C_L	PRMC	--	--	H2		AM31	input
VADJ_1P8V_G39	VJ	--	--	G39		--	output
VADJ_1P8V_H40	VJ	--	--	H40		--	output
FMC1_VIO_B_M2C_K40	VHB	--	--	K40		--	input
VDDL	VL	--	--	--		--	--
VDDON	VN	--	--	--		--	--
GNDOSC	GO	--	--	--		--	--
GNDSATA	GS	--	--	--		--	--

FPGA Connections To the Bottom Daughter Card						
Signal	Symbol	Pin	TYPE	FMC Pin	U1* FPGA Pin	Direction from FPGA view
SATA1_RX (J12)	S1_M2C_P	6	P	A6	B8	input
	S1_M2C_N	5	N	A7	B7	
SATA1_TX (J12)	S1_C2M_P	2	P	A26	C2	output
	S1_C2M_N	3	N	A27	C1	
SATA0_RX (J13)	S0_M2C_P	6	P	A10	A6	input
	S0_M2C_N	5	N	A11	A5	
SATA0_TX (J13)	S0_C2M_P	2	P	A30	B4	output
	S0_C2M_N	3	N	A31	B3	

* U1 connections are displayed for Xilinx Virtex 7 FPGA VC 709 board according to ug887 documentation.
** Changed Based on VC709 Schematics

Pushing the center bottom on the FPGA board sends different commands to the FPGA based on the way these switches are set. These modes are defined in the *top.v* where the main connections are connected and the main modules are instantiated. A constant delay has been set for the *SPI* and *config* lines based on Table 3.6 on page 31 in *spi_delay_tap_const* and *spi_miso_delay_tap_const* defined as follows:

```
spi_delay_tap_const = {spi_mosi(15 taps), spi_sel(15 taps),
                spi_load(15 taps), cfg_clk(13 taps), cfg_valid(15 taps)}
spi_miso_delay_tap_const = {spi_miso(15 taps)}
```

4.2.1.1 Programmer Logic and Programming

AsAP2 programming logic breaks each instruction or configuration line into 4 chunks of 20 bits to program each AsAP2 processor. As described earlier these chunks are defined as upper address, lower address, upper data, and lower data. First, the 64-bit line is read from BRAM programming memory, and then it gets broken into the four different sections described before.

The *AsAp_programming.v* file (*prog_arb* instance) contains the programming logic for the FPGA board side of the AsAP2 programming and is in charge of sending programming and configuration lines to AsAP2 chip. This code sends lines from both run memory and programming memory to the AsAP2 chip.

The state machine implemented for this logic contains five states: IDLE, MEMREAD, MEMDEC, SENDDATA, and WAIT.

The IDLE state is the initial state and no action is done in this state until the start signal get set high (switch 0 in the FPGA is set to Run Mode, and the center push button is pushed). Then the state changes to MEMREAD. In the MEMREAD state depending on the position of the switch 1 either lines from programming memory (switch 1 in programming) are read or lines from the run memory (switch 1 in compute) are read. Figure 4.4 displays this state machine.

After reading a line from either BRAM (program or run memory), the state changes to MEMDEC to decode the line. As mentioned earlier each instruction or *config* gets broken into four chunks. Four different sections are broken according to the following line where *dout* is the 64 bits output line from either BRAM and *addr_data_buf* holds the four chunk of the line in one 80-bit location.

```
addr_data_buf = {UPPERADDRESS_BITS, 15'h0 ,dout[60:58],
        LOWERADDRESS_BITS, dout[57:40], UPPERDATA_BITS, RESET_BIT,
```

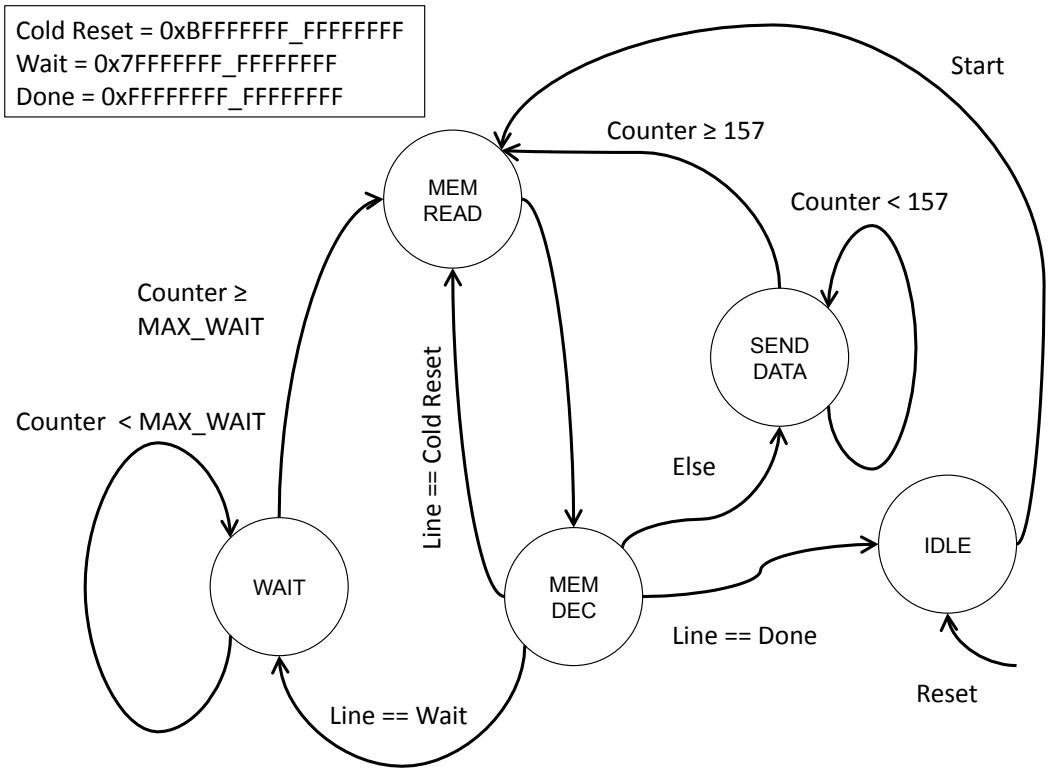


Figure 4.4: State machine view of the programmers logic .

```
dout[34:18], LOWERDATA_BITS, dout[17:0]};
```

and the following are constant values:

```
UPPERADDRESS_BITS = 2'b00
LOWERADDRESS_BITS = 2'b10
UPPERDATA_BITS = 2'b01
LOWERDATA_BITS = 2'b11
RESET_BIT = 1'b0
```

The next state after decoding is either IDLE where 64 ones has been read, MEMREAD where 64'hBFFFFFFF _ FFFFFFFF has been read to display a cold reset, WAIT where 64'h7FFFFFFF _ FFFFFFFF has been read after a config program has been loaded and should finish running, or SENDDATA where the 4 chunks of data get sent to the AsAP2 chip.

In the WAIT state, a counter with minimum width of 8 bits counts up until it reaches a maximum value set by *MAX_WAIT* (default set to 250) running at the *SPI* clock frequency. This wait is only due to some config programs run on core before running the main program, so they can set some memory values before the main program starts. The default value for this wait is set to

250 since the same 8-bit counter used in SENDDATA state is used, and the total of 256 values has been rounded to 250 for simplicity. This state gets invoked after a delay command is read from the BRAM.

In the SENDDATA state, a counter counts up to 197 while the 4 chunks of data get sent. The following describes the state of each signal related to the counter value.

The *shift_en* signal gets set when the counter is between the intervals of [2, 22), [42, 62), [82, 102), [122, 142), [162, 182) where '[' or ']' represent inclusion, but ')' and '(' represent exclusion. Bits start getting shifted out to *spi_mosi* after two *spi_clk* clock cycles after the *shift_en* gets set to one.

The *cfg_clk* signal is low when the counter is between the intervals of [3, 23) , [43, 63), [83, 103), [123, 143), [163, 183) while the rest of time this value is set to high.

The *spi_load* signal gets set high only when the counter is at 24, 64, 104, 144, and 184.

The *cfg_valid* signal gets set high when the counter is at 22, 23, 62, 63, 102, 103, 142, 143, 182, and 183.

Finally, *spi_sel* signal gets reset to low when counter is in the decimal sets of [2, 24), [42, 64), [82, 104), [122, 144), and [162,184) and set back high for any other counter value.

The maximum value that the counter counts up to is 197. Each chunk that gets transmitted to the AsAP2 chip is 20 bits and requires 40 cycles to get transmitted. During the first 20 cycles the *cfg_clk* signal is low while the data get shifted in. then *cfg_clk* goes back high to latch the data for the next 20 cycles (The second 20 cycles can be reduced after complete test of correct functionality of the connection). To send 4 chunks of data it requires 160 cycles, but there is an additional transmitted chunk that gets discarded just to make the program line get stored in AsAP2 chip, so a total of 200 cycles is required. Since the counter starts from zero, the maximum value is 199 counter values. For every 4 chunks and the extra 1 chunk, The state machine must go through MEMREAD and MEMDEC once. Therefore two more cycles have been reduced from 199 counter values resulting in 197 maximum counter values. Finally the values and intervals set for different signals are based on AsAP2 design. Figure 4.5 displays these signals at different counter values.

4.2.1.2 Simulation

The prog_arb is an instance of *AsAp_programmer* module. The prog_arb instance has been simulated for correct operation using three verilog modules from the AsAP2 verilog code so far. The spi_slave.v, cfg_unpack.v, and *cfg_glue.v* have been used to check the input and unpack behavioral

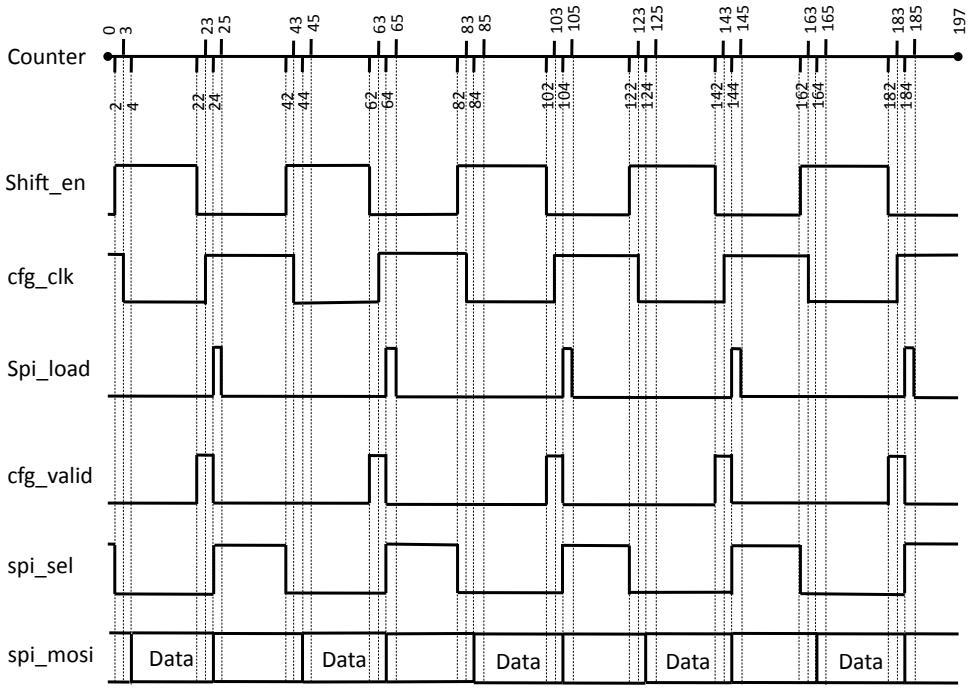


Figure 4.5: The *SPI* and *config* signal behavior at different counter values in the SENDDATA state.

functionality of the code.

In this simulation, all of the stages and the final results of the program were verified.

4.2.2 Input and Output Logic

The "input and output group" logic used in this context are not necessarily the direction of the signal, instead the term "input and output group" to a collection of signals responsible ore sending and receiving data. For example, the signals that are called AsAP2 input group contain 16 data signals, one clock, and one valid signal entering the AsAP2 chip, in addition to a request signal exiting the AsAP2 chip. These signals are all entering or exiting the FPGA FMC connector in the LVDS standard format, but the length of the signals are not identical in each group. Therefore all of these signals must be delayed to ensure the data arrive simultaneously at the receiver.

Due to different clock domains between the AsAP2 chip and the FPGA board clock, dual clock FIFOs are used to synchronize the transferred data between the two regions. These FIFOs are designed with overflow locations. When the request signal has gone low, data in transit can occupy these locations. At any time, multiple data values can be transmitted on the same wire. By the time the transmitter finds out that the request signal has gone low many data values may get

transmitted to the receiver, and this may result in data loss without any spare space in the FIFOs, so enough space is required to store those values.

4.2.2.1 Input Logic and Programming

Input logic group is managed in two different cases:

1. Static Delay Logic with Dynamic Input Ports

   The static delay logic (*asap_input_buff* instance) is implemented in the *input_buffers.v* file, and it mostly contains the buffers and the delay block.

   The static delay tab values are as follows, they get loaded into the delay block by setting the switch 0 to calibration mode and switch 1 to static calibration mode then pushing the center push button.

```
in_data_delay_tap_const = {in_data[15](21 taps),
        in_data[14](20 taps), in_data[13](18 taps),
        in_data[12](18 taps), in_data[11](17 taps),
        in_data[10](16 taps), in_data[9](15 taps),
        in_data[8](13 taps), in_data[7](11 taps),
        in_data[6](10 taps), in_data[5](9 taps),
        in_data[4](12 taps), in_data[3](10 taps),
        in_data[2](9 taps), in_data[1](8 taps),
        in_data[0](6 taps)}
in_request_delay_tap_const ={in_request(14 taps)}
in_valid_delay_tap_const = {in_valid(13 taps)}
```

2. FIFO Logic

   The FIFO (*asap_in_arb* instance) is interfaced in *input_arbiter.v* file. The dual clock FIFO consists of 512 locations, each with a data width of 16 bits.

4.2.2.2 Output Logic and Programming

Output logic group is managed in two different cases:

1. Static Delay Logic with Dynamic Input Ports

2. FIFO logic

The static delay logic (*asap_output_buff* instance) is implemented in the *output_buffers.v* file, and it mostly contains the buffers and the delay block.

The static delay tab values are as follow. They get loaded into delay block by setting the switch 0 to calibration and switch 1 to static calibration then pushing the center push button.

```
out_data_delay_tap_const = {out_data[15](19 taps),
        out_data[14](17 taps), out_data[13](18 taps),
        out_data[12](16 taps), out_data[11](16 taps),
        out_data[10](15 taps), out_data[9](18 taps),
        out_data[8](12 taps), out_data[7](13 taps),
        out_data[6](11 taps), out_data[5](11 taps),
        out_data[4](12 taps), out_data[3](11 taps),
        out_data[2](8 taps), out_data[1](9 taps),
        out_data[0](7 taps)}
out_request_delay_tap_const = {out_request(15 taps)}
out_valid_delay_tap_const = {out_valid(13 taps)}
```

The FIFO (*asap_out_arb* instance) is interfaced in the *output_arbiter.v* file. The dual clock FIFO consists of 512 locations, each with a data width of 16 bits.

4.2.3 Dynamic Delay (Not Verified)

The dynamic delay logic is used to dynamically increase the delay taps on input or output. This functionality starts when switch 0 is set to calibration, and switch 1 is set to dynamic calibration. This dynamic delay logic (*dynamic_delay_arb* instance) is implemented in the *dynamic_delay_arbiter.v* file.

The input group dynamic taps setting starts when switch 2 is set to input calibration and the center push button is pushed. The output group dynamic taps setting starts when switch 2 is set to output calibration and the center push button is pushed.

The logic results the same for both the input and output group using different programs running on AsAP2 chip.

The dynamic logic block starts after the start command is given. The programming unit should start sending the run command to AsAP2 afterward using the programming module. Then this module monitors the outputs of AsAP2 while looking at all data signals and the valid signal while having the request signal high. As soon as the first high bit arrives this logic starts increasing

the delay taps on that signal until all the signals go high or 16 samples are collected after the arrival of the first high value signal. An example using four signals is provided below.

```
Example: Sampled data on each output signal
signal 1 : 00001110000111 (delay tap increase of 5)
signal 2 : 00000011110000 (delay tap increase of 3)
signal 3 : 00000111100000 (delay tap increase of 4)
signal 4 : 00000000011110 (delay tap increase of 0)
```

In order to have this logic set the taps on the output group of AsAP2, AsAP2 gets a program to generate a constant output of 0xFFFF. The program has an offset feature to move the output processor producing the output stream from (12, 0) to (12, 11) as shown on the right side of Figure 2.1 on page 10.

In order to have this logic set the delay taps of the input group, a different program should be loaded into the AsAP2 chip. This program streams the inputs coming to the AsAP2 chip to the corresponding output of the AsAP2 chip as a straight pipe through. This pipe through can get offset to assign different input delay values for different input processors. For example, the pipe can input on processor (0,6) and pipe the data to processor (12,6) by passing through all the intermediate processors and then arriving at the FPGA board and the dynamic delay logic.

For a complete output delay tap calculation two different inputs should be fed into the AsAP2 chip in two different runs. The first run inputs the following stream

```
0
0
0
0
0
0
0
65535  // all 1's just to match
0
0
0
0
0
```

```
0
0
```

This input sets the delay on all the inputs except the valid signal. This is done this way, so the input of the AsAP2 chip accepts all the combinations of the ones even if they arrive earlier or later than the valid signal for the actual group of ones. After the first run is done, the second input should be sent to the AsAP2 input to set the delay on the valid signal, and since the taps roll over after 32 taps the valid signal gets its new value after this run. This run may be repeated if the delays are not set in the first run. The input to the second run is as follows:

```
65535  /* a 0xFFFF just to match */
0
0
0
0
0
0
0
0
0
0
0
0
0
0
```

In this case, all the signals arrive after the valid signal arrives, otherwise the valid signal gets delayed until it rolls over and becomes the shortest delay. It must be noted that the valid signal should roll over 31 taps for this to work, and that is the main reason this may run multiple times. The first input delays the valid signal the most, so it rolls over before any other signal.

4.2.4 Temporary Arbitration

The temporary arbitration logic (*tmp_arb* instance) is written temporarily in the verilog file called *temporary _arbiter.v* only to connect the PCIe to the AsAP2 inputs and outputs. This

code must be modified with the real arbitration interconnect. This can be replaced with an all-to-all crossbar, or any other interconnection design.

4.3 Future Work

The work done on the FPGA verilog code has not been completely tested on the FPGA board and requires further improvement. The additional interfaces such as SATA, DDR3, SFP, and the UART must be implemented in addition to the current arbiters. All these modules should be connected using an interconnect technique with a central control system such as Micro Blaze running a higher-level program.

Chapter 5

PCIe Bring up and Host Data Interface

One of the important interfaces required for this project is the PCIe interface. The Xilinx FPGA board provides a simple interface for PCIe on the FPGA side, but for Direct Memory Access (DMA) on the host system and high-speed data transfer, a more advance design is required.

Xillybus [43] already designed this interface for previous versions of Xilinx FPGA boards, but it hadn't been tested for the FPGA boards with the Virtex 7 chip.

5.1 Bring up and Hardware

Xillybus provided the code and test benches while they needed someone to test the code on a Virtex 7 FPGA board since they didn't have a Virtex7 FPGA board available for testing purposes. A computer case large enough to fit the FPGA board had to be found as well. A computer meeting the requirements was purchased. The Fedora operating system that the Xillybus software was tested on was installed on the computer in order to eliminate potential software problems. This version was Fedora 19.x86_64 with 3.13.9 linux kernel version.

Installing the FPGA board in the computer requires close attention to make sure all connections on PCIe are connecting since in some cases this connection does not connect due to a poor design of the metal brackets on the FPGA board after getting screwed into the computer case. In order to prevent this issue, either the case screw shouldn't be tightened completely, or an extension PCIe cable should be used.

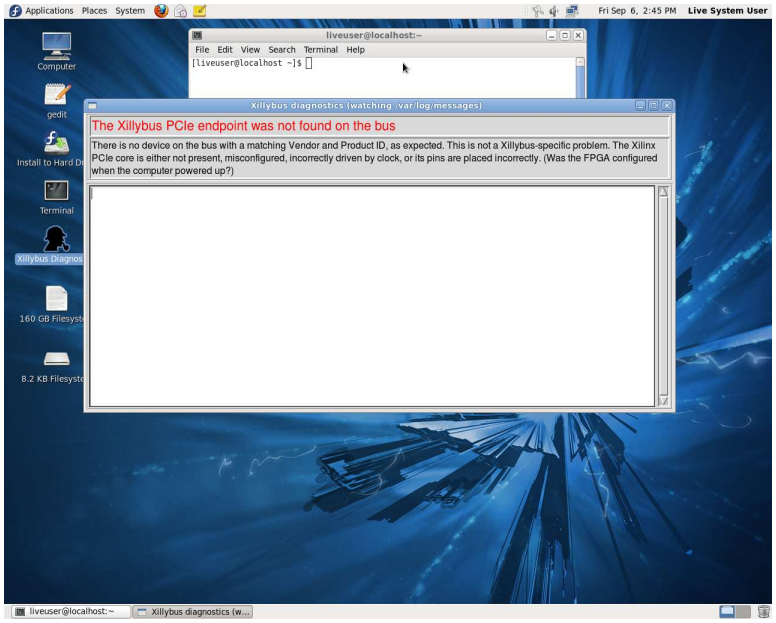


Figure 5.1: Xillybus error message when the FPGA board is incorrectly installed

When the Xillybus code is run and the FPGA board is incorrectly installed, the message shown in Figure 5.1 is displayed.

5.2 Xillybus IP Core

The Xillybus IP core only supports the ISE environment. Figure 5.2 from the *xillybus_getting_started _xilinx.pdf* [44] document provides a schematic view of this environment. As shown on this figure, there are two FIFOs available on the FPGA side of this core, one for input to the FPGA board, and the other for the output from the FPGA board. There are two sets of these kind of FIFOs, one with 32-bit data width and the other with an 8 bit data width. In this project implementation the 8-bit FIFO is replaced with a 8 to 16-bit and vice versa data width dual clock FIFO in order to interface the 16-bit data interface of the AsAP2 chip with the 8-bit interface of the Xillybus code. The 32-bit FIFO interface used for reprogramming is also replaced with a dual clock FIFO with a 32 to 128-bit data width and a 128 to 32-bit data width interface in this project since the memory interface is required to be more than 64 bits while the Xillybus Interface is only 32 bits.

The 32-bit interface flips the order of the bytes transferred between the host and the FPGA board in the 32-bit packet interface in a way that a 32-bit hex value of 0xDEAD_BEEF arrives to

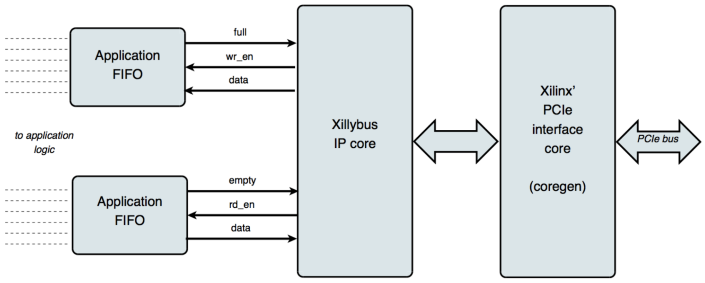


Figure 5.2: The schematic view of Xillybus internal design

the other side as 0xEFBE_ADDE. This change happens in both transfers from the FPGA board to the host and the vice versa.

In the event the design doesn't meet timing based on "Xillybus Getting Started Xilinx" PDF document, the "placer cost table figure" (seed generator) must be modified until the design meets timing. This is due to an incorrect initial guess done by ISE as described in this document. This document also explains that "placer cost table figure" is available in the Process Properties of MAP option when viewed in Advanced View. The number in this field can change to any other number that hasn't been tested previously.

After programming the FPGA board with Xillybus code, the system with the PCIe connection must be restarted to have the Xillybus drivers installed.

5.3 Linux

After loading the Xillybus IP core onto the FPGA board that is installed in the host computer and restarting the host computer, the drivers are loaded and the PCIe connection can be detected. The following command can be used to check the PCIe connection (More details are available in "Getting Started with Xillybus on a Linux Host" [45] PDF files):

```
$ lspci -v
```

After running this command the list of all the available PCI connection is displayed. A message similar to the following inside this list shows a connection to the Xilinx FPGA board through PCIe.

```
02:00.0 Unassigned class [ff00]: Xilinx Corporation Device ebeb (rev 08)
        Subsystem: Xilinx Corporation Device ebeb
        Flags: bus master, fast devsel, latency 0, IRQ 47
```

```
Memory at f3200000 (64-bit, non-prefetchable) [size=128]
Capabilities: <access denied>
Kernel driver in use: xillybus_pcie
```

After the connection has been made, the following files get added to the /dev/ directory of the system:

1. xillybus_mem_8 : Reading from and writing to xillybus_mem_8 is used to directly map between the two memories, but is not currently used in this project since other programs have provided all the required functionality.

2. Xillybus_read_32 :The *xillybus_read_32* file is used to read from the 32 bit Xillybus PCIe FPGA FIFO

3. Xillybus_read_8 : The *xillybus_read_8* file is used to read from the 8 bit Xillybus PCIe FPGA FIFO

4. Xillybus_write_32 : The *xillybus_write_32* is used to write to the 32 bit Xillybus PCIe FPGA FIFO

5. Xillybus_write_8 : The *xillybus_write_8* is used to write to the 8 bit Xillybus PCIe FPGA FIFO

In order to test a simple connection while a loop back program is loaded in the FPGA board, the following two commands can be used to read what has been typed in one terminal. The commands should be run in separate terminals

In Terminal 1:

```
$ cat /dev/xillybus_read_8
```

In Terminal 2:

```
$ cat > /dev/xillybus_write_8
```

When using these commands, the first terminal displays what has been entered into the second terminal after pressing “Enter” on the keyboard. For more advanced test programs, both, “Getting started with Xillybus on a Linux host” [45] and, “Xillybus host application programming guide for Linux” [46] can be found on Xillybus website.

Chapter 6

Host Computer Programming Chain

The system in charge of programming all the FPGA boards as well as sending the data to the system interface Picoblade is considered as the host computer. This system should be able to program the FPGA board and generate the required binary data to program the AsAP2 chip. This system should also send and receive data to and from the AsAP2 to be processed.

The host system requires being big enough to be able to fit the FPGA board connecting to the host computer through PCIe while the AsAP2 daughter card is connected to the FPGA board using the FMC connector. This is important since the FPGA board and the AsAP2 daughter card did not fit in many computer systems. This section requires a previous knowledge about AsAP2 programming, available in the AsAP2 manual.

6.1 Host Design Interface

There are two main design interfaces implemented on the host system.

The first interface describes the host programming and the programs used to convert a simple assembly code to a binary or FPGA readable format.

The second interface describes the host input and output interfaces to generate the desired binary data inputs from the human visible format or convert the binary outputs to human visible format to be evaluated.

Most of the programs written on the host system are in Perl and C programming language

while there are parts that haven't been modified, but considered as a part of the design interface and are written in Python.

6.2 Host Programming Interface

In the old interface, the assembly code gets processed through a python code named *assem.py* and gets converted to ASCII binary format then these outputs are fed to a C program named aprog to get converted to binary format and get loaded to the FPGA board using aprog.

The new interface can be broken into three different categories of JTAG, PCIe, and UART, but since the UART has not been implemented this part is not covered in this chapter. The first method described uses the JTAG to load the AsAP2 programming code into the FPGA BRAM. The second method uses the PCIe connection to send the AsAP2 program to the BRAM on the FPGA board. The second method is mostly used after the testing phase of the FPGA program is done and is only used by the system interface Picoblade.

6.2.1 First Host Programming Interface Method

The first programming method is to generate and send the AsAP2 programming code using the JTAG. In order to do this the AsAP2 programming code should go through the interface shown in Figure 6.1. This interface assumes that the code has been written in the pseudo assembly format used in the simulators that is in `.cpp` extension format. This program gets converted to unoptimized assembly using a converter program. A scheduler program can run on this code to add all the required optimizations to this program. After optimization, reducing the unrequired DMEM allocations using a program called *null_remover* can further optimize the resulted code by register renaming. The output of the null_remover can be converted back to the simulator format with `.cpp` extension to be simulated again, or it can also be given to the assembler to get converted to an intermediate binary in ASCII format with a `.dat` extension. Then a program called aprog converts this code to two files: *asap.coe* initializes the FPGA program memory, and *run.coe* initializes the run memory (hold the run instruction for each processor to start the clock on the processors) on the FPGA board. Finally these files get used to initialize the FPGA board BRAM memory module as a part of verilog code to be synthesized, implemented, and converted to a bit stream with .bit extension to be programmed to the FPGA board.

The next four subsections describe each one of these programs in more details.

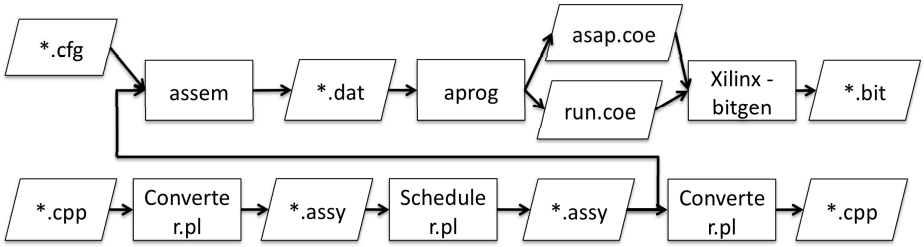


Figure 6.1: Schematic view of the first host programming interface running on the host system.

6.2.1.1 Programming Converter

The programming converter converts the programs from the simulator format to an unscheduled format with .assy extension. It can also convert the programs back from the assembly format to the simulator format and convert the DMEM variables in a .assy file to original variable name used in simulator if a conversion table is given. The assembly output of the converter should be scheduled to ensure correct functionality.

The converter code is written in Perl. The converter can be used using the two methods described below.

A. The following are the steps used to convert the `.cpp` format to .assy format

A.1. The following is the format used to invoke the converter:

```
converter.pl [-h] [-i] -2a [-s <source_file>]
                    [-o <destination_file>] [-c <config_file>]
                    [-p <processor_number>]
                    [-t <output_table_file>]

<source_file>  : The file containing C++ source to Brent
                 simulator code.
<destination_file> : The file containing the converted assembly
                 code.
<config_file> : The file containing the simulator's initial
                 block.
<processor_number> : It is the processor number to be set
                  for this file ex: 0,0.
<output_table_file> : The file containing the convertion table.
```

```
-h      : This help.
-i      : In order to display some info regarding the program,
          and these info get stored in info_log.
```

The Processor number follows the format shown in the next example.

Example: 3,2

There are no spaces between the numbers and the comma. If -p is not given, the default value is 0,0. The -i tag is used to create a "`info_log`" to output the number of used DMEMs in the program

A.2. Check the resulted code and add subtract from the code if required.

A.3. Move the '`Start_Initialization`' and '`End_Initialization`' code to the config file if the config file doesn't get specified with -c when running the converter.

A.4. The -t flag specifies the name of the file that saves all the DMEM renaming correspondences.

B. The following are the steps used to convert the .assy format back to `.cpp` format.

B.1. The following is the format used to invoke the converter:

```
converter.pl  [-h] [-i] -2c [-s <source_file>]
              [-o <destination_file>] [-t <input_table_file>]

<source_file>  : The file containing assembly source file.
<destination_file> : The file containing converted simulator
                     code.
<input_table_file> : The file containing the convertion table.

-h          : This help.
-i          : In order to display some info regarding the
              program, and these info get stored in
              info_log.
```

The input table given using -t must follow the following format:

dmem <dmem_address> -> <new_variable_name>

This format is kept if the -t flag with the corresponding *output_table_file* was selected when the assembly code is generated. The DMEMs in comments also turn into the old variables. this is desired since in simulator there is no concept of DMEM variables.

The conversion from simulator to assembly is possible for the following cases:

1. The assembly/simulator instructions such as `rptb`, `rpt`, `move`, ...
2. All functions/subfunctions become *begin <given_processor_number>* ()
3. The `end` gets inserted automatically before the next function starts.
4. All $\#defines$ gets replaced in the code with their corresponding values.
5. All /* */ comments (multi line comments) become // comments (single line comments)
6. All *_numbers* become #numbers ex: _3 —> #3
7. Most C++ instructions become comments in case the assembly code inside them should be recognized and removed. However, the switch/case instructions should manually be removed since these constructs aren't recognized.
8. All output instructions become `#output`
9. Simple calculations get simplified

   Example: $2 * 3$ —> 6
10. The start of all hex numbers get converted from $0x$ to h
11. Negative hex numbers get converted to their corresponding two's complement hex number in assembly.

The conversion from assembly to simulator has been verified with simple DMEM replacements. The following summarizes these conversion capabilities:

1. The converter can remove `end` and `begin` lines and uncomment any 'function' or 'subfunction' in the code.
2. The converter can reinsert `Start_Initialization` and `End_Initialization` from another file back into the simulator file.
3. The converter calculates the amount of data memory used and outputs this data into the "`info_log`" file.

4. The converter can convert the converted DMEMs variables back to their original variable names.

6.2.1.2 Scheduler

There was a scheduler written in Perl with limited functionality for AsAP2 assembly code. This program wasn't complete. This code was improved upon and debugged, so it could be added to the AsAP2 tool chain. Currently this code is still in Perl, but with additional features such as forwarding.

Since AsAP2 is a pipelined system, all instructions aren't executed in a single clock cycle. In order to prevent data and control hazards, No OPeration (`NOP`) instructions must be explicitly included in the assembly code. Due to inefficiency of using `NOP` instructions, AsAP2 architecture can also use forwarding logic to reduce the number of the `NOP` instructions in its architecture. The forwarding has been implemented using forwarding registers referred to as bypass registers (written in assembly code as `regbp[1-3]`). The scheduler explicitly includes these bypass registers as source locations in the assembly code to reduce `NOP` instructions.

In order to use the scheduler outside the *makefile* tool chain, the following steps should be taken:

1. Write a program ignoring all `NOP`s and `regbp`s. The program can be transferred from the simulator format to assembly as described before.

2. Run the scheduler with the command below. The *instruction_src.h* and *instruction_dest.h* should be located in your current directory with sched.pl and the `.assy` file in order to schedule the program.

```
sched.pl [-h] [-t] [-nn] [-nc] [-ni] [-c] [-na] [-sf <sources file>]
         [-df <destinations file>] [-s <input assembly file>]
         [-o <output assembly file>]

<sources file> : Architecture dependent file defining source
                 operands and pipe stages for instructions.
<destinations file> : Architecture dependent file defining
                 destination operands and pipe stages for
                 instructions.
<input assembly file> : The file containing unscheduled code.
```

```
<output assembly file> : The file containing scheduled code.

-h      : Scheduler shows the help and instruction flag definition.
-t      : Test mode (for debugging) scheduler shows extra testing
          outputs.
-c      : Scheduler displays comments similar to -t, but it displays
          different information.
-na     : With this flag, scheduler doesn't consider the DMEM
        dependencies between address generators (ag or agpi) and
        address pointers (aptr) with each other or any DMEM
        addresses. The programmer must make sure there are no
        dependencies, and if there are, programmer should manually
        add NOPs after running the scheduler.
-nn     : Scheduler doesn't add null to all unused DMEM operands.
        The null insertion mostly happens after forwarding is added
        to the code, and the DMEM is not used in the code again to
        reduce the DMEM usage, but this flag disables this feature
        for faster scheduling run time.
-nc     : Scheduler doesn't show user comments in the scheduled
        output code. These comments are in the body of the code
        written by the programmer, and with -nc these comments get
        removed from the body of the output code after scheduling.
-ni     : Scheduler doesn't create the warning logs. This flag
        forces the scheduler to surpass its outputs. This flag is
        mostly used for testing purposes.
```

The following is an example on how to run the scheduler code.

```
$ perl5.16 sched.pl -sf instruction_src.h -df instruction_dest.h
          -s test.assy -o test_out.assy
```

Or just simply

```
$ sched.pl -s test.assy -o test_out.assy
(If instruction_src.h and instruction_dest.h are in the same
```

```
directory as sched.pl)
```

Also "sched.pl -h" can be typed for help and usage information.

3. Double check all the `NOP`s and add or subtract them as desired.

It is worth noting that all the instructions that are not supposed to be modified are inside `#pragma notouch` blocks, so the scheduler doesn't touch them.

Example: The original input is as follows:

```
begin 0, 0 (chipout east)
      #output east
      add dmem 1 dmem 2 dmem 3 // nop3
      #pragma notouch
            add dmem3 dmem 4 dmem 1 // nop3 Hazard
            sub dmem 6 dmem 7 dmem 3
            add dmem 5 #2 #3
      #endpragma
      add dmem 6 dmem 3 dmem 1 // nop3
      sub dmem 3 dmem 6 dmem 1
end
```

The output produced by scheduler is as follows:

```
begin 0, 0 (chipout east)
#output east
add dmem 1 dmem 2 dmem 3 nop3  // nop3
#pragma notouch
add dmem 3 dmem 4 dmem 1 //nop3 Hazard
sub dmem 6 dmem 7 dmem 3
add dmem 5 #2 #3
#endpragma
add dmem 6 dmem 3 dmem 1 nop3  // nop3
sub dmem 3 dmem 6 dmem 1
end
```

The section between `#pragma notouch` and `#endpragma` doesn't get touched by the scheduler. As it is shown in the previous example, there is a Hazard inside the no touch area that hasn't

been fixed, but all the other dependencies outside of the no touch area has been consider. Even the dependencies from the outside of no touch block to the inside of no touch block are considered. The notouch section cannot be a part of the `rptb` block - however, it is okay for an entire `rptb` block to be included in a no touch section. It is also okay to branch in or out of a no touch section.

In order to run the scheduler code using the makefile the following command is used:

```
$ make schedule TEST=test\_name ARG="-flag1 -flag2 ..."
```

The ARG is optional, and test_name is the name of directory that the *test_name.assy* is located in. This usage is shown in the following examples:

```
Example1: The following can be used to get the help output of the
scheduler:
$ make schedule ARG="-h"
```

```
Example2: The following can be used to run scheduler with -na flag on
test1.assy inside test1 directory:
$ make schedule TEST=test1 ARG="-na"
```

Each time a file gets scheduled using *make*, a backup of the original input file gets created in the same directory with the naming convention as follows:

<file name>.orig#.assy

Where the # represent a number. The make command can store up to 100 backups. In order to return to previous generated versions of the scheduled file the following command can be used:

```
$ make undo_schedule TEST=testname VER=version_number
```

The version_number can be a number between 0 and 100, and all the future versions after that version get deleted. The selected version replaces the scheduled version.

The following is a list of data hazards the scheduler can correctly handle, each have been verified:

DMEM -- DMEM ---> 3

DCMEM -- DCMEM ---> 3

DMEM -- APTR ---> 4

DMEM -- DAG ---> 4

MAC -- ACC ---> 1

ACC -- OBUF ---> 1

BR (POST) ---> 0

BRC -- CONDITIONAL (PRE) ---> 2

RPT (POST) ---> 3

RPTB (POST) ---> 3

OBUF DIR (POST) ---> 1

MIN/MAX -- DCMEM 24 ---> 3

MIN/MAX -- BRMS1,2 ---> 2

BR(BF) -- PCPTR ---> 3

Appendix C on page 126 has a complete list of all the fixes to the previous version of the scheduler.

The list below shows a list of possible future improvements on the scheduler:

1. The scheduler schedules the code based on standard conditional mode. The dependency of the extreme conditional mode must be implemented. The following displays an example of the issue when using the extreme mode:

   ```
   movi mask hffff cxa
   sub dmem 1 #3 #2 cxs // nop3
   add dmem 2 dmem 1 #1 cxt //nop2
   sub dmem 2 dmem 1 #1 cxf // nop3
   move obuf dmem 2
   ```

   It must output the following:

   ```
   movi mask hffff cxa
   sub dmem 1 #3 #2 nop3 cxs // nop3
   add dmem 2 dmem 1 #1 nop2 cxt //nop2
   sub dmem 2 dmem 1 #1 nop3 cxf // nop3
   move obuf dmem 2
   ```

   However, the scheduler outputs the following:

   ```
   movi mask hffff cxa
   sub dmem 1 #3 #2 nop3 cxs // nop3
   add dmem 2 dmem 1 #1 cxt //nop2 <- error not enough nop in extreme
   ```

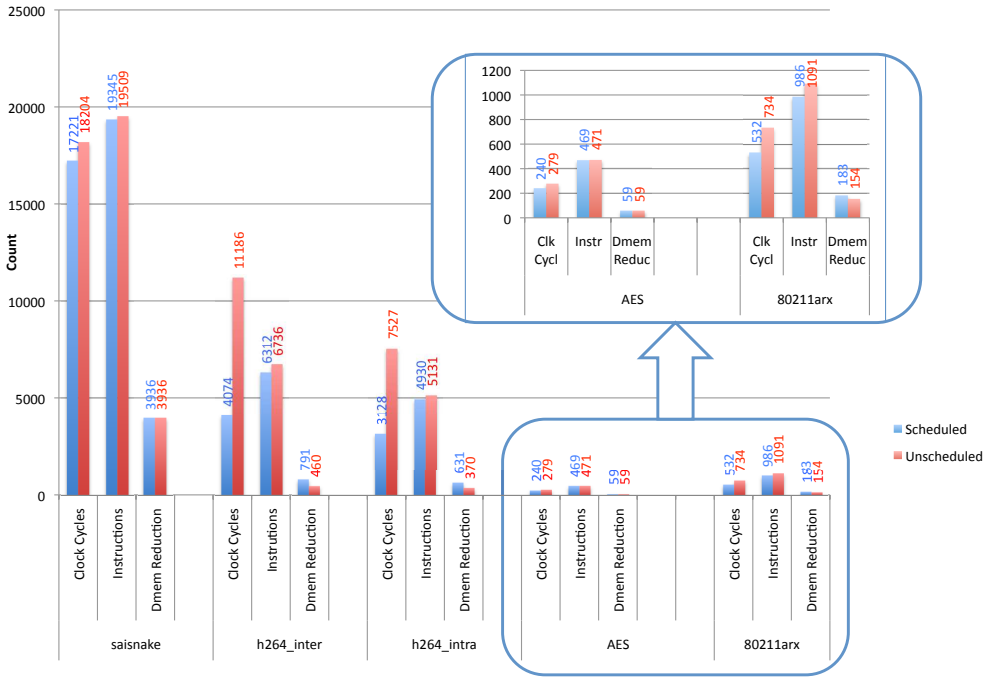


Figure 6.2: Program Performance Effects with Scheduler on different programs

```
                    // mode
sub dmem 2 dmem 1 #1 nop3 cxf // nop3
move obuf dmem 2
```

2. The `ibuf` and `ibufnap` ordering is linear, and they don't switch positions, however the `ibufnaps` can switch order between two `ibufs`. This can be improved by having the first `ibuf` switch with the following `ibufnaps` as long as the first access to the buffer comes as `ibuf` and the rest as `ibufnap`.

3. Optimize across simple blocks by moving blocks around can improve the optimization of the code.

4. Add the capability of finding the `pcptr` destination and calculating the dependencies.

5. Add the support for `mode/cxa` (conditional execution) for an immediate operand calculation in `subc`, `subch`, `subcs` (when operands are equal), `macl, mac, and mach`. Also, adding the support for other values other than immediate operand calculation such as DMEM operands.

6. Add notouch pragma capability to only a part of the `rptb` block.

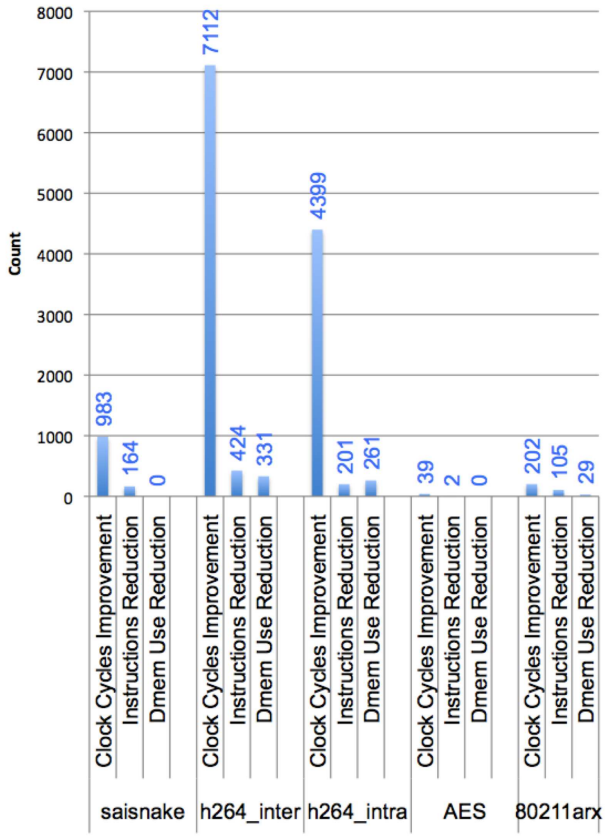


Figure 6.3: Application improvements after using scheduler on previously unscheduled code.

In order to test the functionality of the scheduler a small Perl program called *nop_counter.pl* was written to generate the number of `NOP` instructions per processor, the total instruction count per processor, the number of `null`s inserted per processor, and the total lines for the entire assembly program. This program can get information in both simulator and assembly format based on the extension of the file, where `.assy` is used for assembly and `.cpp` is used for the simulator input files.

In order to use the nop_counter the following command structure should be followed:

```
nop_counter.pl [-wi] [-s <input file>] [-o <output file>]

<input file>      : Input file assembly or simulator format.
<output file>     : File for printing output to.

-wi : This flag is only used for files with the .cpp extension. With
this flag nop_counter counts the number of NOPs and instructions inside
```

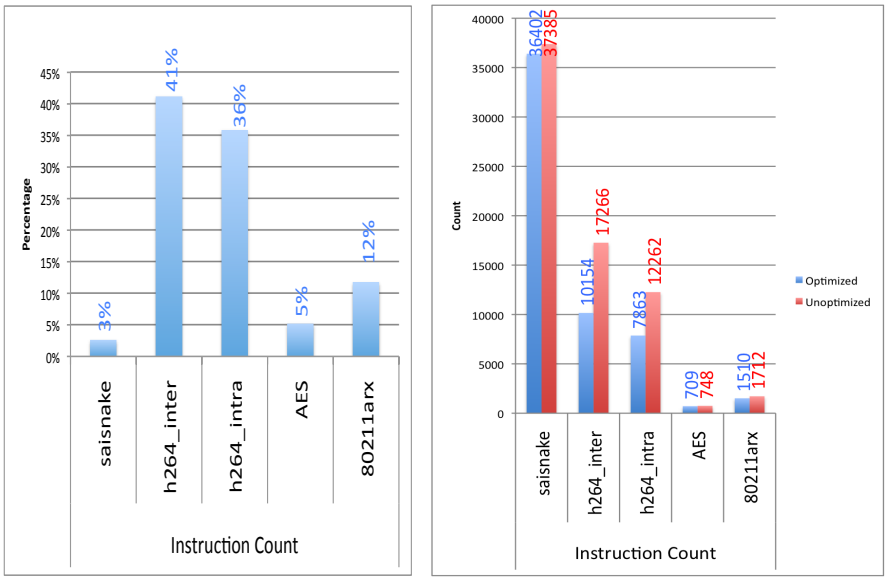


Figure 6.4: Instruction count for scheduled and unscheduled code on right and percentage of instruction improvement on left

the initial block (instructions between 'Start_Initialization' and 'End_Initialization' code) as well as the rest of the code. Without this flag, nop_counter doesn't consider the information inside the initial block.

Figure 6.2 displays the different parameter values before and after using the scheduler on some existing assembly programs. This information is gathered using the nop_counter.

Figure 6.3 displays the difference between the scheduled and unscheduled version of different code.

Finally, Figure 6.4 displays the percentage improvement for the total number of instructions per program.

6.2.1.3 Assembler

The assembler has not been updated, and the output of this file is kept the same in multiple `.dat` files. The output of these files is binary in ASCII format, so that they are human readable. However, they still must get converted to Xilinx BRAM ASCII initializing format to be programmed to the FPGA board after getting ordered using the aprog described next.

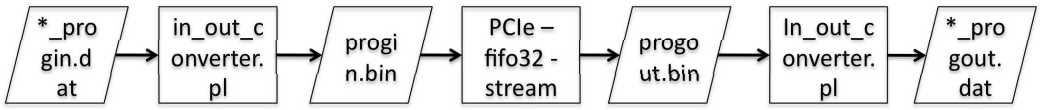


Figure 6.5: Schematic view of the second host programming interface running on the host system

6.2.1.4 AsAP2 programmer (aprog)

The aprog is a C program, and it was written to convert the `.dat` files outputted from the assembler to binary format. There have only been some minor changes to this program to produce the output in coe format (The format used to initialize the FPGA BRAM) instead of binary format. These changes are such as addition of the procedure of *pairlist_save_coe* in *pairlist.c* file and addition of the coe file type to *filelist.c* and *main.c* file. In order to run this program using the makefile to generate the coe files the following command is used:

```
$ make TEST=<test_directory_name> coefile
```

This command first runs a script called *makecoe.sh* to set the arguments for aprog and then it runs aprog to generate both the asap.coe and run.coe files.

The run.coe should be loaded in the run memory of the FPGA board, and asap.coe should be loaded in the program memory of the FPGA board as the initializing data. After the bit files get generated these data get loaded to the FPGA board using the JTAG connection.

6.2.2 Second Host Programming Interface Method

This interface is described in Figure 6.5 using a flow diagram. This interface is used to read from a generated `.dat` file in a specific format and does not contain a higher level programming capability for human readability or programming since inputs are in binary character format.

The input `.dat` files for programming contain instructions that are 128 bits long. Each instruction is stored in hex and has the following format:

```
[127: 127]  This bit specifies whether this instruction is a read or
                write (read == low [zero] / write == high [one]).
[126: 126] This specifies which memory is being written to
                (run_memory == high [one] /program_memory == low [zero]).
[125: 79] These bits are always set low [zero].
[78: 64] These are the address bits for the memories. It requires
```

```
                attention that the run memory can only hold up to 256
                instructions while the program memory can hold up to
                30K instructions.
[63: 0]  These are the data bits containing the instruction when writing
                a new instruction and zeros when reading an instruction
                from either of the memories.
```

Below is an example of code used to read from address 0 to 5 of the run memory:

```
h4000000000000000_0000000000000000
h4000000000000001_0000000000000000
h4000000000000010_0000000000000000
h4000000000000011_0000000000000000
h4000000000000100_0000000000000000
h4000000000000101_0000000000000000
```

The "h" character at the beginning of each line specifies these values are hex numbers without "h" numbers get considered as decimal values.

After the program has been written in the above format, it should get converted to binary format. The *in_out_converter* described in the next section does this conversion.

6.2.2.1 Input and Output Programming Converter

To convert the input program file from ASCII to a binary to be sent to the FPGA board over PCIe and to convert the BRAM reads from the FPGA board to ASCII, *in_out_converter.pl* is used. This program is written in Perl. The following is how this program can be used:

1. The following command converts the output from the programming and run memory reads from the FPGA over PCIe to their corresponding 128-bit hex in ASCII representation:

```
in_out_converter.pl [-h] [-2d32 -s <input bin file>
                    -o <output dat file>]
 <input bin file>  : The file containing .bin source code
 <output dat file> : The file containing converted .dat code in 128
                     bit chunks
 -h                : This flag shows help and usage of the program
```

2. The following command converts the files containing the input data or programs in the form of binary, hex, or decimal characters to their corresponding binary format to be sent through PCIe to the FPGA board.

```
in_out_converter.pl [-h] [-2b -s <input dat file>
                         -o <output bin file>]
<input dat file>  : The input file containing .dat source file.
<output bin file> : The output file containing converted .bin code.
-h          : This flag shows help and usage of the program.
```

3. The following command converts the output from data out explained in Section 6.3.1 to the corresponding 16 bit hex character representation:

```
in_out_converter.pl [-h] [-2d -s <input bin file>
                         -o <output dat file>]
<input bin file>  : The file containing .bin source code to be
                    converter.
<output dat file> : The file containing converted .dat code in
                    16 bit chunks.
-h          : This flag shows the help and the usage of the program.
```

In order to use the makefile to run this code the following commands can be used:

1. The following command can be used to convert the input from hexadecimal values stored in ASCII format to binary.

```
$ make TEST=<test_directory_name> binprogin
```

2. The following command can be used to convert the output from binary to .dat with hexadecimal values stored in ASCII format.

```
$ make TEST=<test_directory_name> binprogout
```

6.2.2.2 Stream Run

In order to send the binary program commands to the PCIe an existing program by Xillybus (The PCIe IP core provider) called *streamwrite.c* that has slightly been modified can be used to send the binary program values to the 32 bit PCIe FIFO explained in Chapter 5 on page 56. The following is the command to run this program:

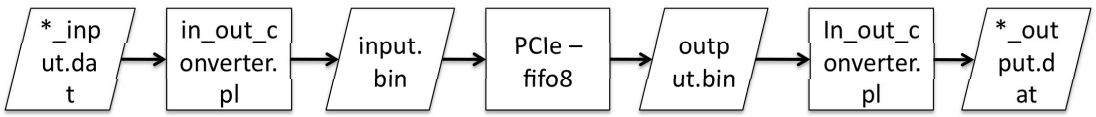


Figure 6.6: Schematic displaying the data interface in and out of the FPGA board using the PCIe connection

```
$ ./streamwrite /dev/xillybus_write_32 progin.bin
```

To receive the binary pogram output from the PCIe, an existing program by Xillybus called *streamread.c* can be used. The following is the command used to read from the 32-bit PCIe FIFO explained more in Chapter 5 on page 56.

```
$ ./streamread /dev/xillybus_read_32 > progout.bin
```

The *streamread* program must run before *streamwrite* to get correct results. Also to make things easier a makefile command can run these two programs at the same time in the correct order described below.

1. The following command can be used when the *progin.bin* has already been generated.

   ```
   $ make streamrun
   ```

2. The following command can be used to convert the `.dat` *progin* file to binary format first, and then run both streamread and streamwrite afterward.

   ```
   $ make TEST=<test_directory_name> streamrun
   ```

6.3 Host Data Interface

The Data input and output between the FPGA board and the system interface Picoblade uses the PCIe connection. In order to convert and send the input data and receive and convert the output data to hexadecimal values stored in ASCII format from the FPGA board, certain steps are required, which are displayed in Figure 6.6.

In order to send the data to the FPGA board, first the `.dat` file is converted to a binary format, then this converted data is sent to the FPGA board. Similarly, the output data from the PCIe is stored as a binary file, and then it gets converted to a `.dat` format. The `.dat` file is in ASCII format with hexadecimal values.

6.3.1 Input and Output Data Converter

For the input and output data converter in_out_converter.pl (described in Section 6.2.2.1) is used. The -2d flag is used to convert to a `.dat` file and the -2b flag is used to convert to a `.bin` file.

A makefile command can also be used to make things easier to type (shorter commands). The following are the makefile commands used to convert between data `.bin` and `.dat` files:

1. The following command is used to convert the input data from ASCII to binary:

   ```
   $ make TEST=<test_directory_name> bininput
   ```

2. The following command is used to convert the output data from binary to `.dat` in ASCII.

   ```
   $ make TEST=<test_directory_name> binoutput
   ```

6.3.2 FIFO Run

In order to send and receive the binary data through PCIe to and from the 8 bit FIFO on the FPGA, a multi-threaded program called "*fifo*" by Xillybus but modified was used. This program can be started using the following command:

```
$ ./fifo <memory allocation> input.bin > output.bin
```

A makefile was created and the following commands can be used to run the fifo program with the memory allocation of 128 MB of memory.

1. The following command can be used if the *input.bin* has already been generated:

   ```
   $ make fiforun
   ```

2. The following command can first be used to convert the `.dat` input file to binary format, and then the fifo program can be run afterwards:

   ```
   $ make TEST=<test_directory_name> fiforun
   ```

To increase the maximum locked memory size allocated for PCIe DMA to a higher value than 512 MB, edit the */etc/security/ directory/limit.conf* file. The maximum soft limit of the locked memory size can be changed using the command below. This soft limit can only be increased up to the hard limit explained previously which is currently at 512 MB. These limits are only for allocation purposes only.

```
$ ulimit -Sl <desired value less than the hard limit in bytes>
```

6.4 Future Work

The second programming method was implemented using only basic hex values in ASCII format. It is possible to improve this programming method by allowing a more convenient input format, such as AsAP2 Assembly with some extra commands to describe whether a write or a read to either BRAMs are being requested.

Chapter 7

Battery Powered Supply for AsAP2

Research was carried out to determine whether an on or off-chip DC-to-DC converter could be combined with a battery source to provide sufficient power for Integrated Circuits [47]. Methods for providing various voltages using bulk and boost converters were also researched [48]. Chips have been designed to use trench capacitors [49] to increase the efficiency while others use interleaved methods [50] to reduce the area of the DC-to-DC converters while increasing the efficiency. There has been research for hybrid methods to use a combination of linear and switching convertors [51] to reduce the area and increase the efficiency on the DC-to-DC converters. All these have shown that the use of a DC-to-DC converter on die is not possible since the AsAP2 chip is 32.75 mm^2 on 65 nm technology [10], and a high efficiency converter is about 0.0042 A/mm^2 [49]. This converter on a 65 nm technology requires an area of 235.10 mm^2/A, or 1410.54 $mm^2$for AsAP2 running at full speed and dissipating 6 A that is 43× bigger than AsAP2 chip. A battery-powered system to power up the AsAP2 board has been implemented to demonstrate the portability of the design.

This chapter describes a completed project to show the future capabilities of this design. Prior to this project, the original AsAP2 daughter card and the FPGA board were running on five different power supply devices, resulting in a large and immobile system. The AsAP2 chip is a low power chip that can switch to lower voltage levels at a wide rage of clock frequencies, so it can run using simple batteries.

An off-chip DC-to-DC converter was considered to convert and regulate the output of two AA or AAA batteries while AsAP2 chip runs at 1 V and 200 MHz to demonstrate low power

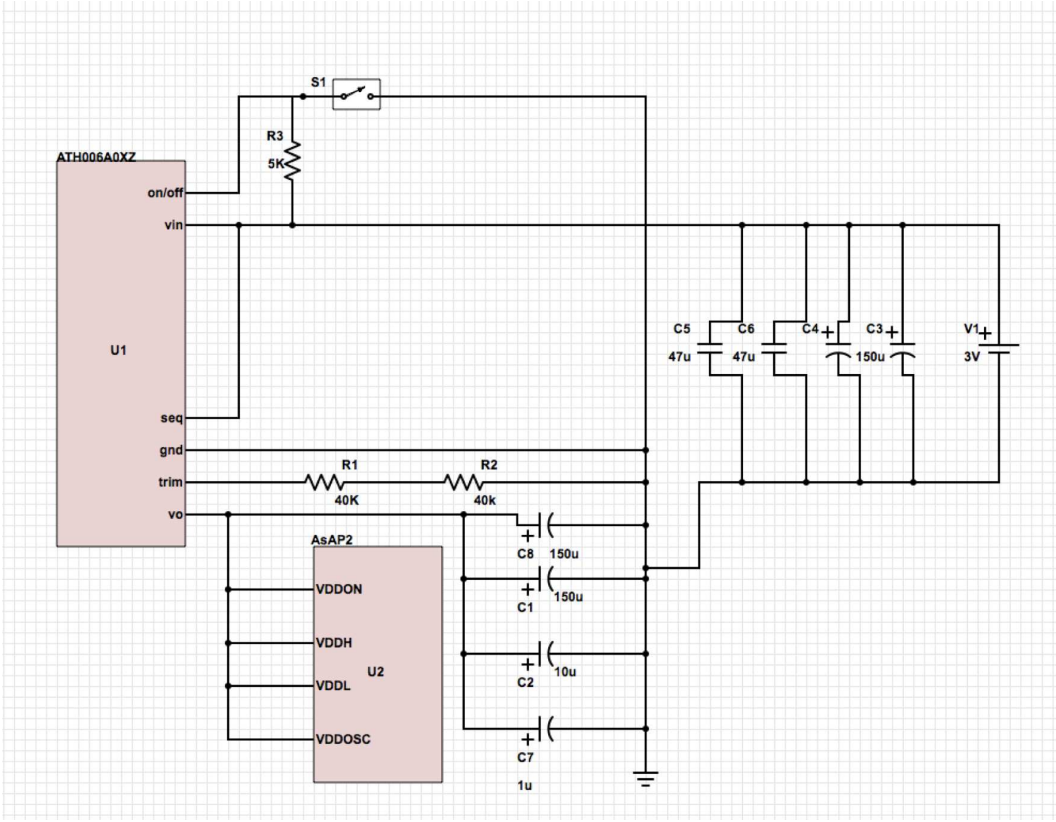


Figure 7.1: Schematic view of the battery powered demo.

dissipation of the AsAP2 chip.

7.1 Schematics and Components

The schematic view of the voltage converter design can be viewed on Figure 7.1. The requirement for this design is to produce a 1 V output using minimal number of AA or AAA batteries. In order to achieve this goal a buck converter was used to convert a 3 V (two 1.5 V batteries in series) input voltage to a 1 V output voltage. The goal is to provide enough current to all the power inputs for the AsAP2 chip except for the *VDDIO* that is sourced by a power supply. For this purpose Austin Microlynx II SIP [52] power module buck converter was used for power conversion from 2.4 V – 3 V to a 1 V output while supporting a current up to 6 A. The main reason for having a range input is that the converter operates at 2.4 V to 5.5 V, but since two AA or AAA batteries in series generate can output a maximum of 3 V, and this voltage drops over time, the input range gets limited to 2.4 V to 3 V.

The Austin Microlynx II SIP converter has the following 6 pins:

1. ON/OFF: This pin is used with negative logic to turn the DC-to-DC converter on and off. When this pin is connected to ground (Low), the DC-to-DC converter turns on, and when this pin is connected to *VIN* (High), the DC-to-DC converter turns off. In order to implement this, a 5 KΩ pull up resistor (R1 on Figure 7.1) connects this pin to *VIN* while a physical switch pulls this node to ground to turn the device on.

2. VIN: This pin is used to provide the input to the DC-to-DC converter. This pin is connected to the positive side of the AA or AAA batteries in series (3 V input source). For the purpose of input decoupling, two 150 µF (C3 and C4 on the schematic) and two 47 µF (C5 and C6 on the schematic) capacitors are used connecting *VIN* to ground.

3. SEQUENCE: This pin is used for the case that the DC-to-DC converter should turn on while following the ramp of another device. This feature is not being used for this demo, so it has been connected to *VIN*.

4. GND: This pin gets connected to ground.

5. TRIM: The output voltage gets set based on the value of the resistor connecting this pin to ground. The output voltage value and the ground-connecting resistor follow Equation 7.1.

$$Rtrim = \left[\frac{21070}{vo - 0.7525} - 5110\right] \ \Omega \tag{7.1}$$

   The required resistance for 1 V output is 80 KΩ, so two 40 KΩ resistors are used (R1 and R2 as is shown on Figure 7.1).

6. VO: This pin gets connected to the output decoupling capacitors and the AsAP2 input power pins. The output decoupling capacitors used are two 150 µF (C1 and C8 on schematic), one 10 µF (C2 on schematic), and one 1 µF (C7 on schematic) capacitor.

Figure 7.2 displays the result after soldering the parts for both the AA and AAA designs.

The DC-to-DC converter turns off when the input voltage is below 2.4 V that limits the run time of this design the values discussed next.

7.2 Prediction and Results

Before connecting the designed circuit to the AsAP2 daughter board, a simple calculation was done to calculate the battery run time of the design at 1 V output when AsAP2 chip is running

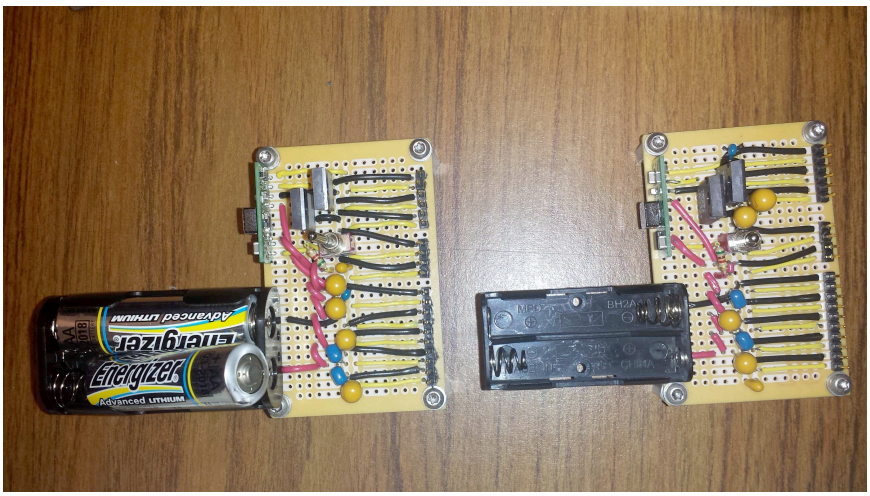


Figure 7.2: AA battery powered DC-DC converter (left) and AAA battery powered DC-DC converter (right)

at 200 MHz. For this purpose some current measurements where done to measure the current usage of the AsAP2 chip at 1 V and 200 MHz clock frequency. The following current measurement values are based on the measurement collected while running a sorting program on the AsAP2 chip when all supplies are running at 1 V except for *VDDIO* running at 2.5 V:

VDDIO = 2.5V, I = 0.395A

VDDON = 1V, I = 0.025A

VDDOSC = 1V, I = 0.008A

VDDH = 1V, I = 0.115A

VDDL = 1V, I = 0.000A

In this measurement the batteries only provide energy for *VDDON*, *VDDOSC*, and *VDDH*, so the total current required is 0.148A.

According to the AA [53] and AAA [54] Energizer batteries datasheet, A AAA battery at 21 degree Celsius drops from 1.5 V to 1.2 V in 1 hour, assuming it is sourcing a constant current of 250 mA. Again, according to the AAA datasheet the battery drops from 1.5 V to 1.2 V in 6 hours, assuming it is sourcing 100 mA of current. Based on this information, it can be inferred that a AAA battery sourcing 150mA takes between 1 to 6 hours to drop from 1.5 V to 1.2 V. Additionally, it can be concluded that two AAA batteries connected in series sourcing 150 mA takes between 1 to 6 hours to drop from 3 V to 2.4 V. According to the datasheet, a AA battery at 21 degree Celsius drops from 1.5 V to 1.2 V in 3 hour, assuming it is sourcing a constant current of 250 mA. Again, according to the AA datasheet the battery drops from 1.5 V to 1.2 V in 15 hours, assuming

it is sourcing 100 mA of current. Based on this information, it can be inferred that a AA battery sourcing 150mA takes between 3 to 15 hours to drop from 1.5 V to 1.2 V. Additionally, it can be concluded that two AA batteries connected in series sourcing 150 mA takes between 3 to 15 hours to drop from 3 V to 2.4 V.

The results calculated for a AAA battery were confirmed by connecting the battery powered circuit to the power inputs of AsAP2 and running the same sorting algorithm used for current measurements at 200 MHz clock frequency for more than one hour at a 1 V output voltage. This measurement confirmed the lower bound of the service time for a AAA battery demo to be between 1 and 6 hours.

Chapter 8

Conclusion

These documents covered certain aspects of the AsAP2 optical interconnect interface by mostly focusing on the AsAP2 programming and interconnects. This document first describes the complete architecture. Then it focuses on the AsAP2 chip interface. It describes how a new daughter card was designed, and how the FPGA interfaces all the components with each other. Then it describes the host connection and programming interfaces. The PCIe connection and a simple battery powered demo are explained last.

The design has not been finalized since the daughter card has not fully been tested for functionality, and also the verilog code for programming and AsAP2 inputs and outputs have not been completely tested. The second method of programming still requires more work as well. However, many steps in this project have been finalized and completed such as the design of the daughter card and the PCIe interface.

8.1 Future Work

The steps required to finish this project are to implement all of the other interfaces such as optical interconnect, DDR3 interface, SATA interface, UART interface, and a central interconnect using a Micro Blaze in addition to testing and verifying the current stage of the completed work.

In the further future this project can be converted to a design that doesn't require a power hungry FPGA board. The setup could be performed using a daughter card connected to a Solid State Drive (SSD) while everything is running on batteries at very low power as shown in Figure 8.1. This design can have the DRAM memories, the optical modules, and the administrator unit all on the same PCB connecting to other PCBs in an optical network.

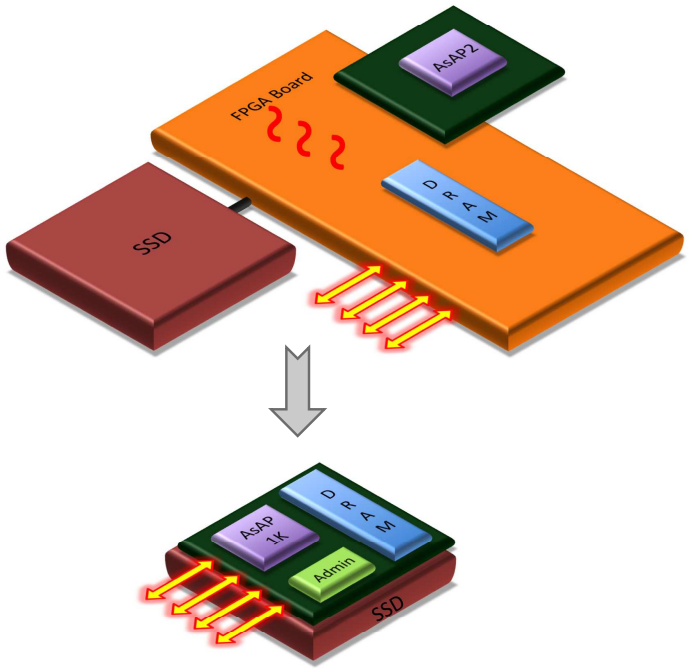


Figure 8.1: The future view of the designed project

Appendix A

Daughter Card Signal Layers Gerber Files

The list of Gerber views are as follow:

- BoardOutline.gdo — Board outline for reference
- DrillDrawingThrough.gdo — Drill drawing
- DrillDrawingThrough_holes — Through hole drill drawing
- EtchLayer1Top.gdo — Top signal layer
- EtchLayer2.gdo — Signal 2
- EtchLayer3.gdo — Signal 3
- EtchLayer4.gdo — Signal 4
- EtchLayer5.gdo — Signal 5
- EtchLayer6.gdo — Signal 6
- EtchLayer7.gdo — Signal 7
- EtchLayer8.gdo — Signal 8
- EtchLayer9.gdo — Signal 9
- EtchLayer10.gdo — Signal 10

- EtchLayer11.gdo — Signal 11
- EtchLayer12Bottom.gdo — Bottom signal layer
- GeneratedSilkscreenBottom.gdo — Bottom silkscreen
- GeneratedSilkscreenTop.gdo — Top silkscreen
- SoldermaskBottom.gdo — Bottom soldermask
- SoldermaskTop.gdo — Top soldermask
- SolderPasteBottom.gdo — Bottom paste mask
- SolderPasteTop.gdo — Top paste mask
- ThruHoleNonPlated — Non plated drills
- ThruHolePlated — Plated drills

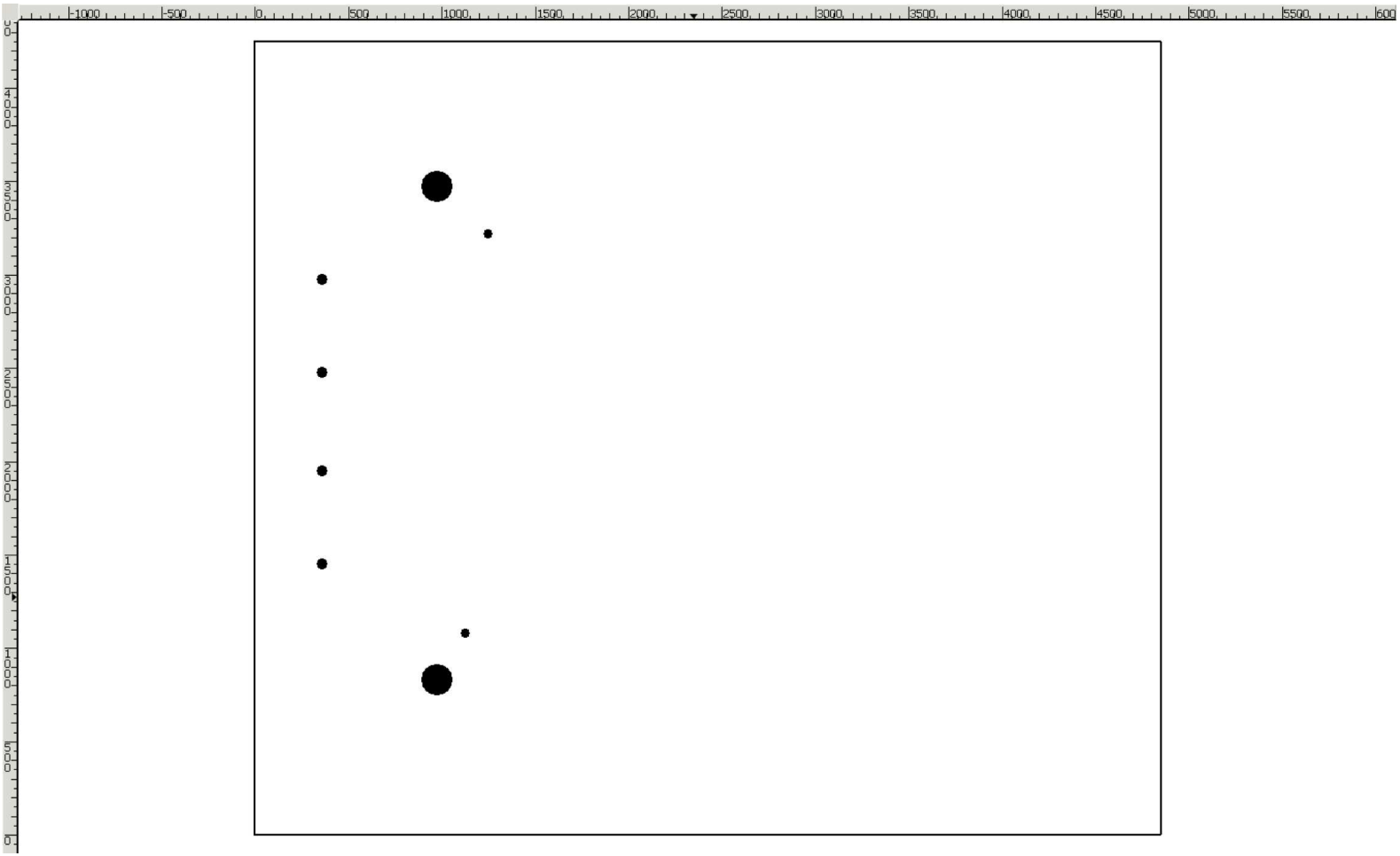
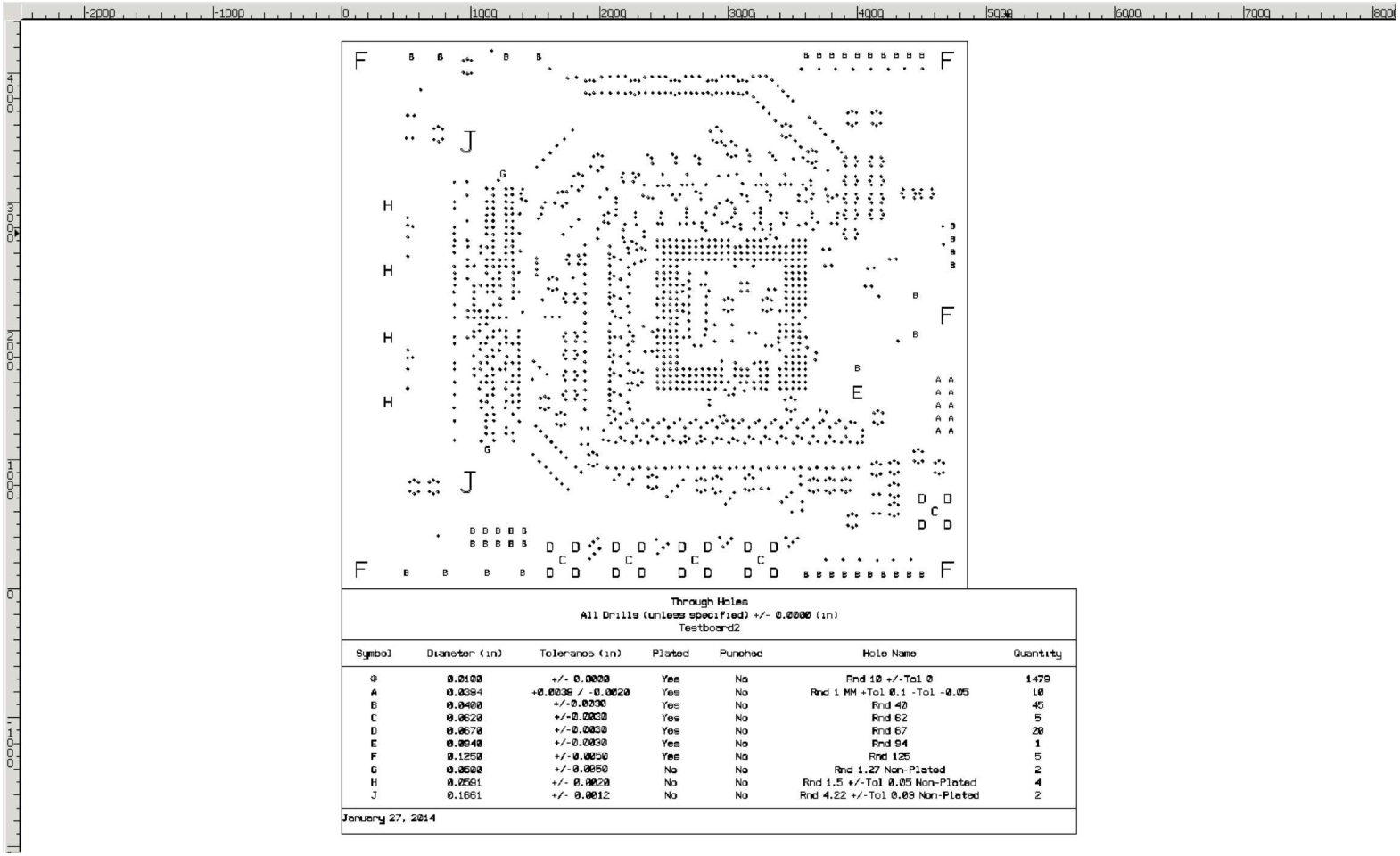


Figure A.1: Board outline for reference



Through Holes
All Drills (unless specified) +/- 0.0000 (in)
Testboard2

Symbol	Diameter (in)	Tolerance (in)	Plated	Punched	Hole Name	Quantity
@	0.0100	+/- 0.0000	Yes	No	Rnd 10 +/-Tol 0	1479
A	0.0394	+0.0038 / -0.0020	Yes	No	Rnd 1 MM +Tol 0.1 -Tol -0.05	10
B	0.0400	+/-0.0030	Yes	No	Rnd 40	45
C	0.0620	+/-0.0030	Yes	No	Rnd 62	5
D	0.0670	+/-0.0030	Yes	No	Rnd 67	20
E	0.0940	+/-0.0030	Yes	No	Rnd 94	1
F	0.1250	+/-0.0050	Yes	No	Rnd 125	5
G	0.0500	+/-0.0050	No	No	Rnd 1.27 Non-Plated	2
H	0.0591	+/- 0.0020	No	No	Rnd 1.5 +/-Tol 0.05 Non-Plated	4
J	0.1661	+/- 0.0012	No	No	Rnd 4.22 +/-Tol 0.03 Non-Plated	2

January 27, 2014

Figure A.2: Drill drawing

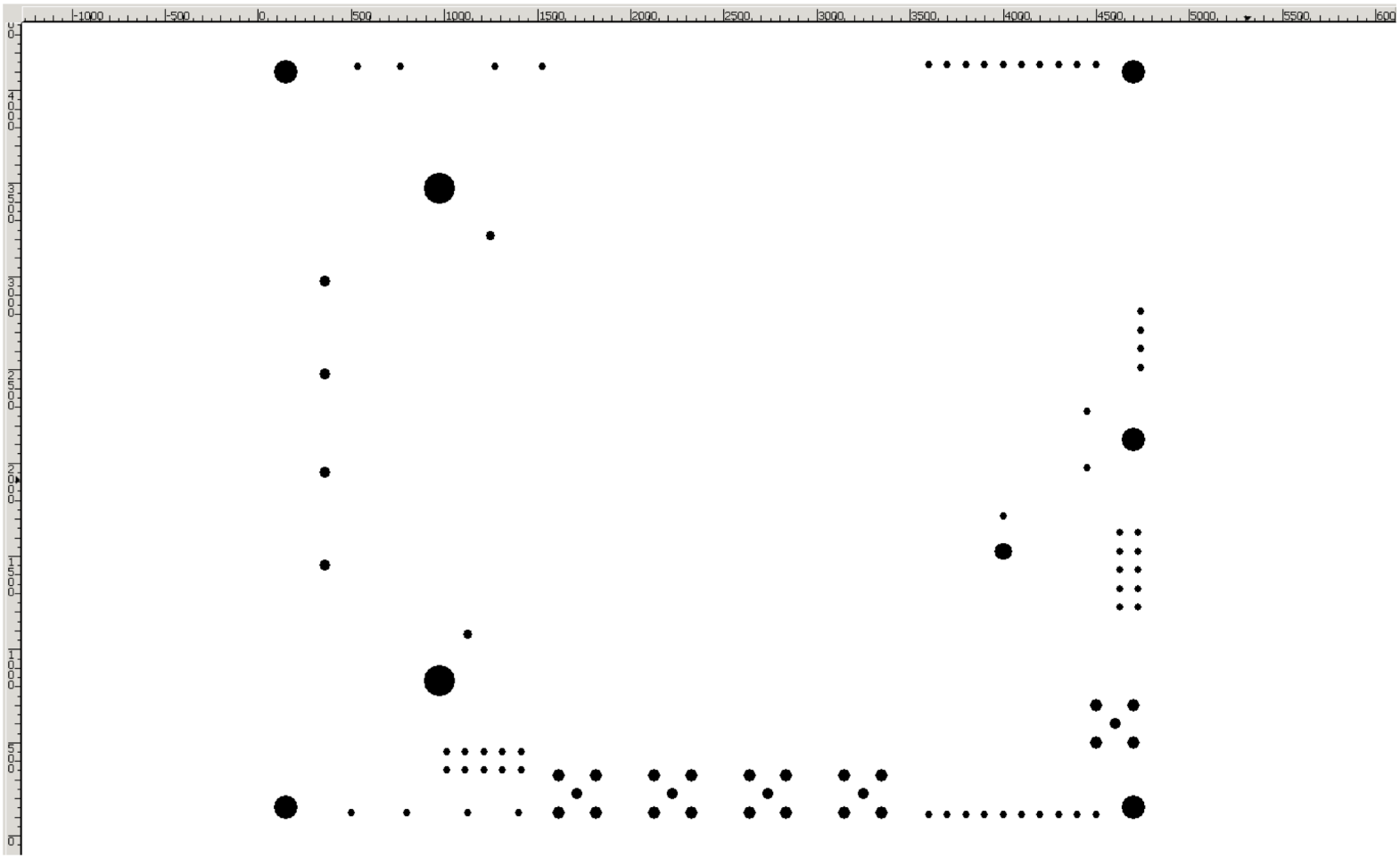


Figure A.3: Through hole drill drawing

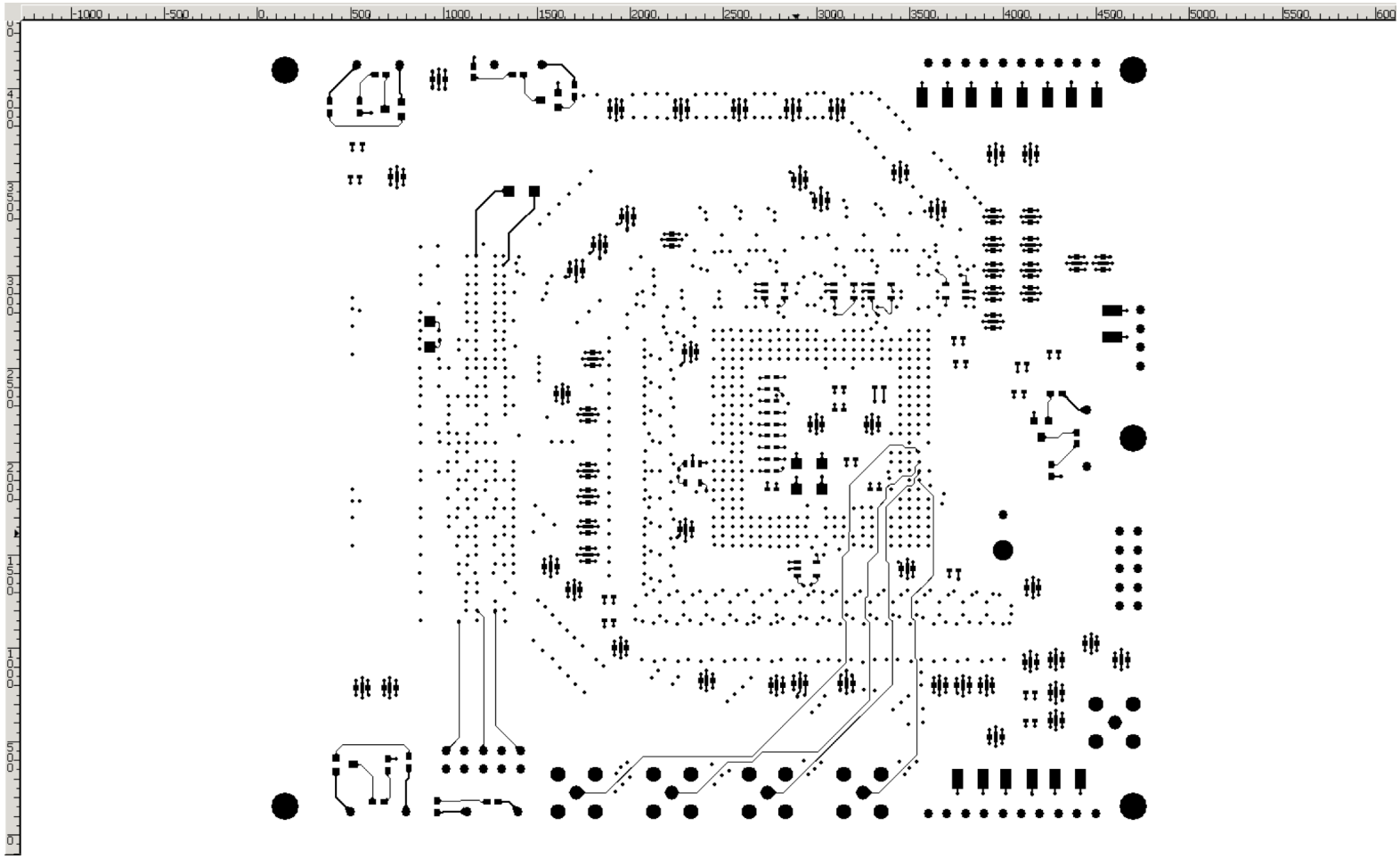


Figure A.4: Top signal layer

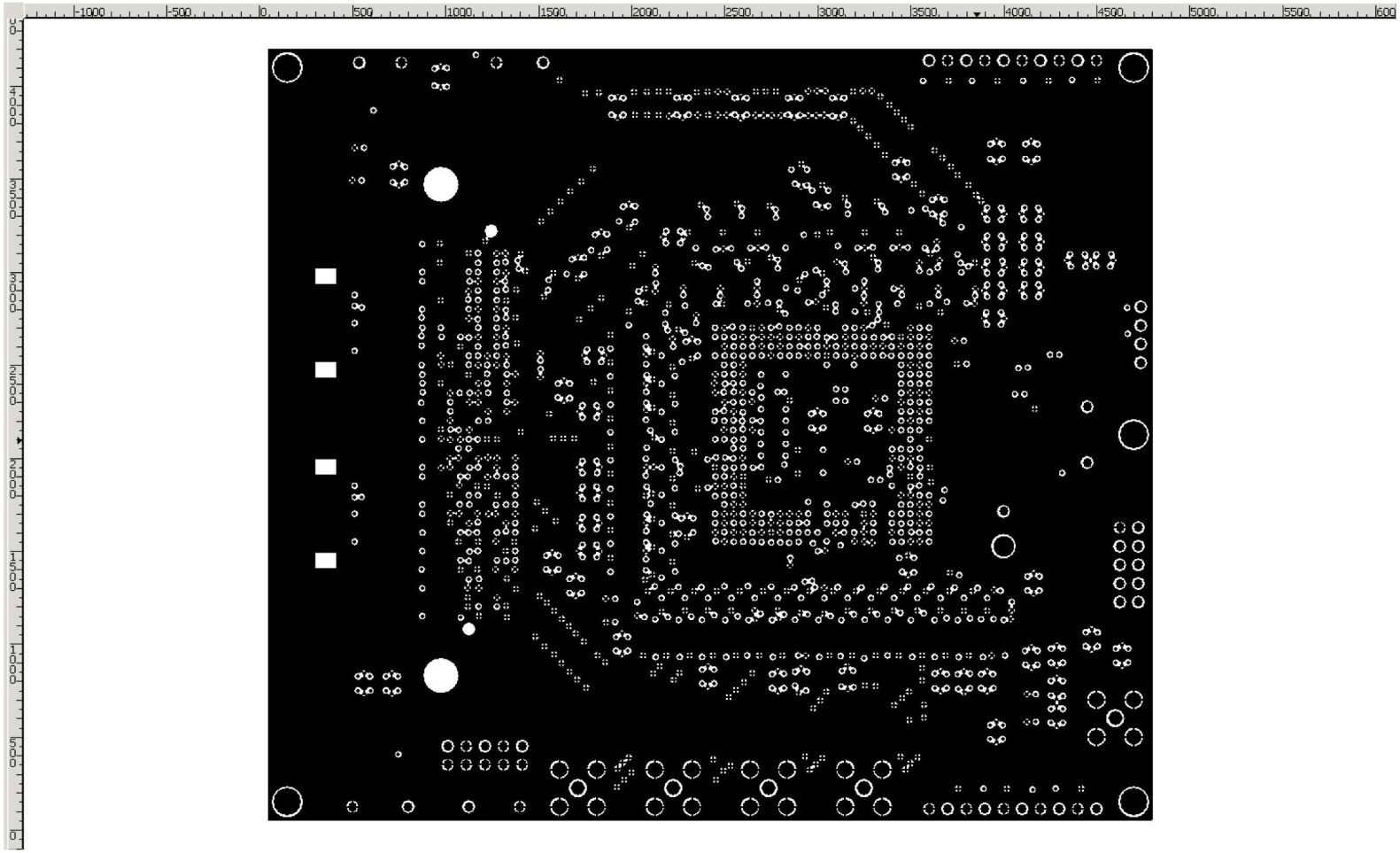


Figure A.5: Signal 2

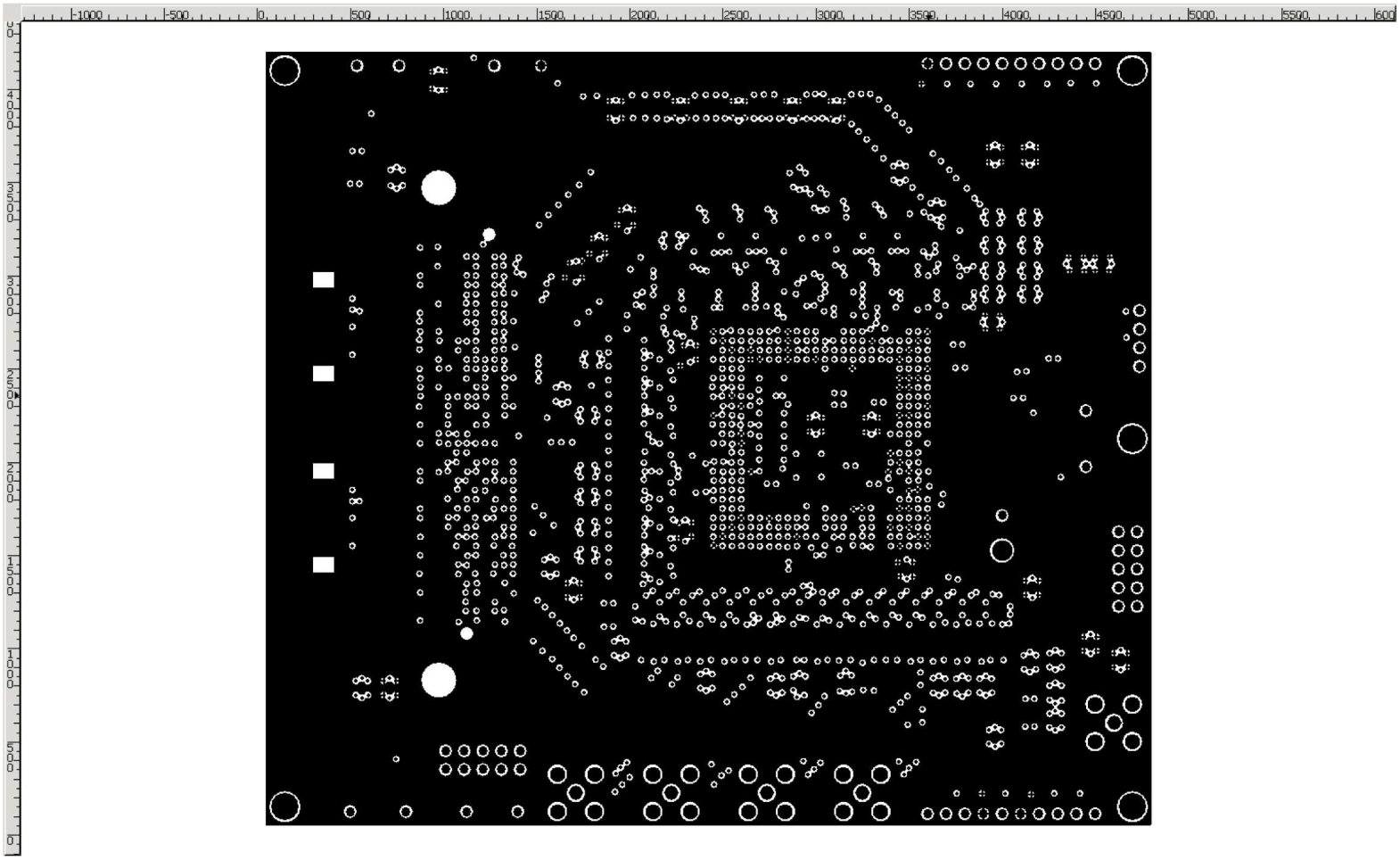


Figure A.6: Signal 3

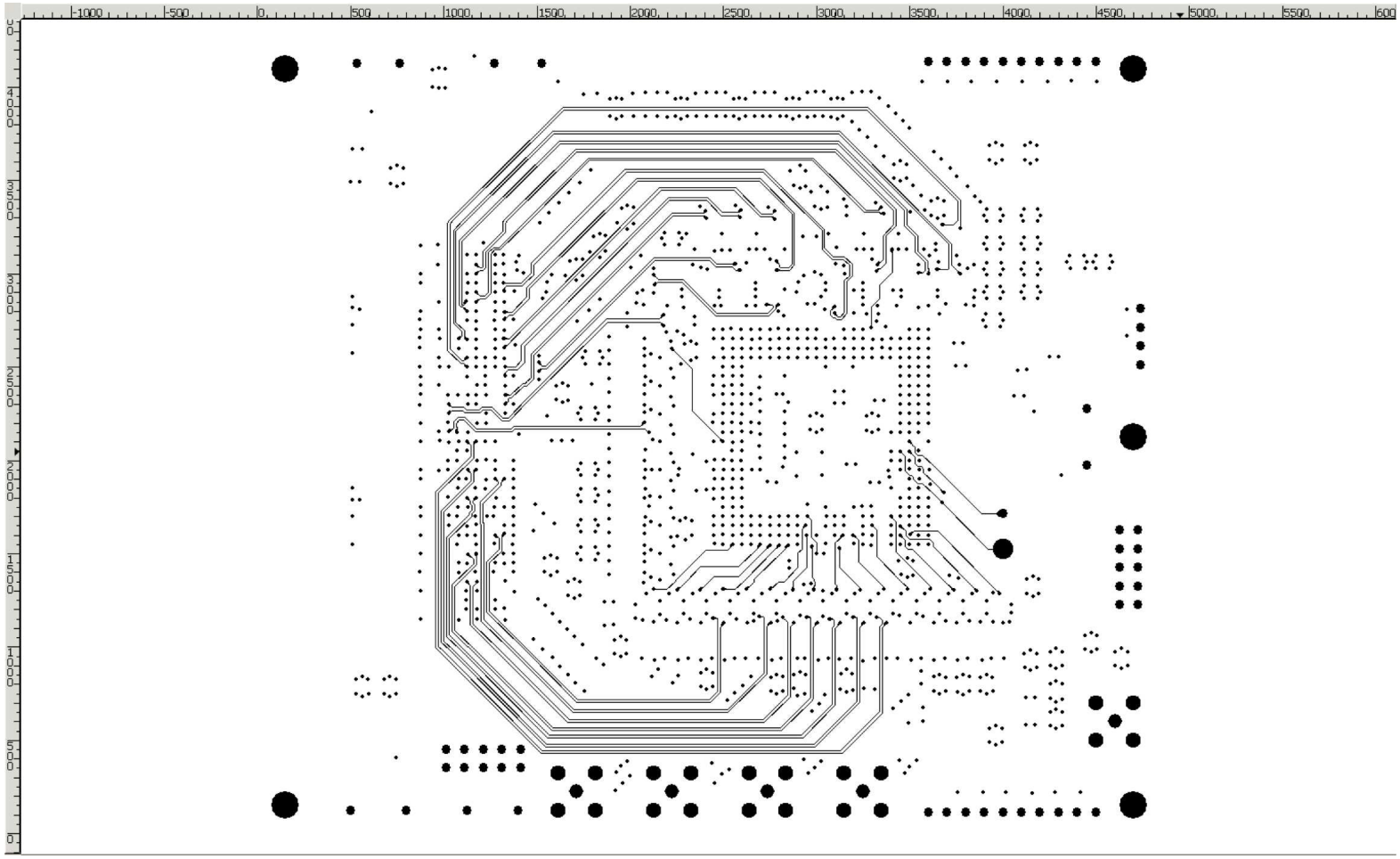


Figure A.7: Signal 4

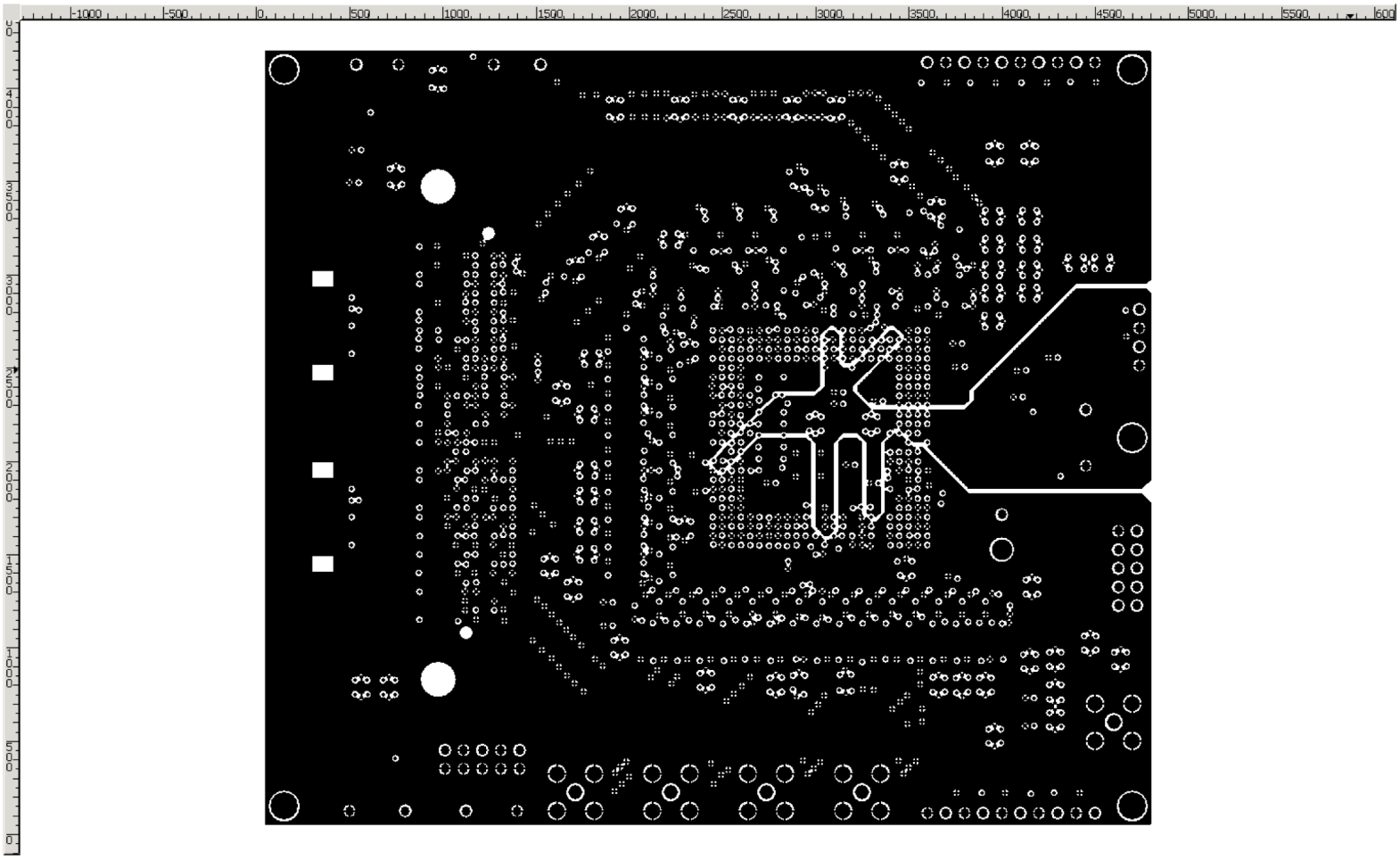


Figure A.8: Signal 5

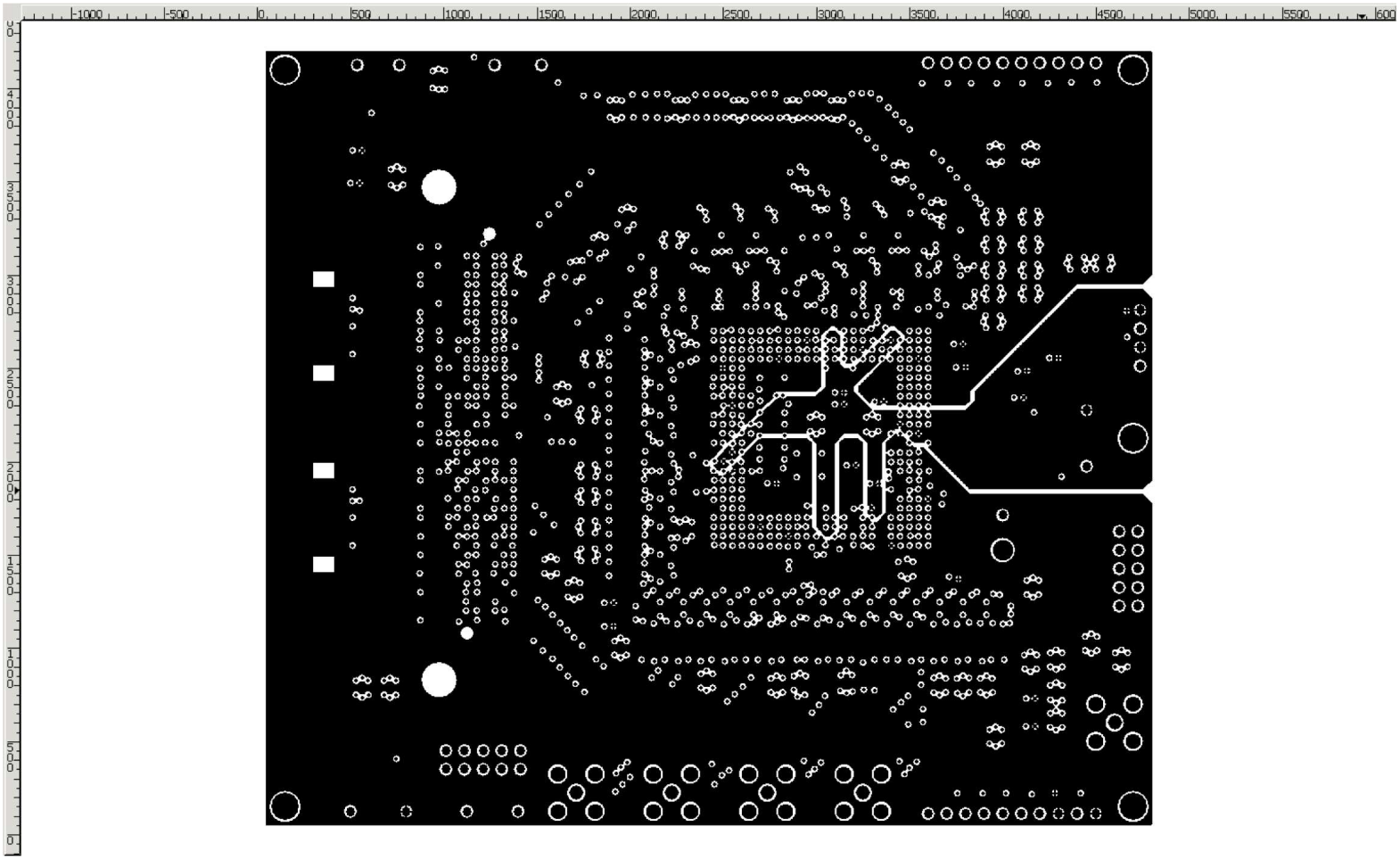


Figure A.9: Signal 6

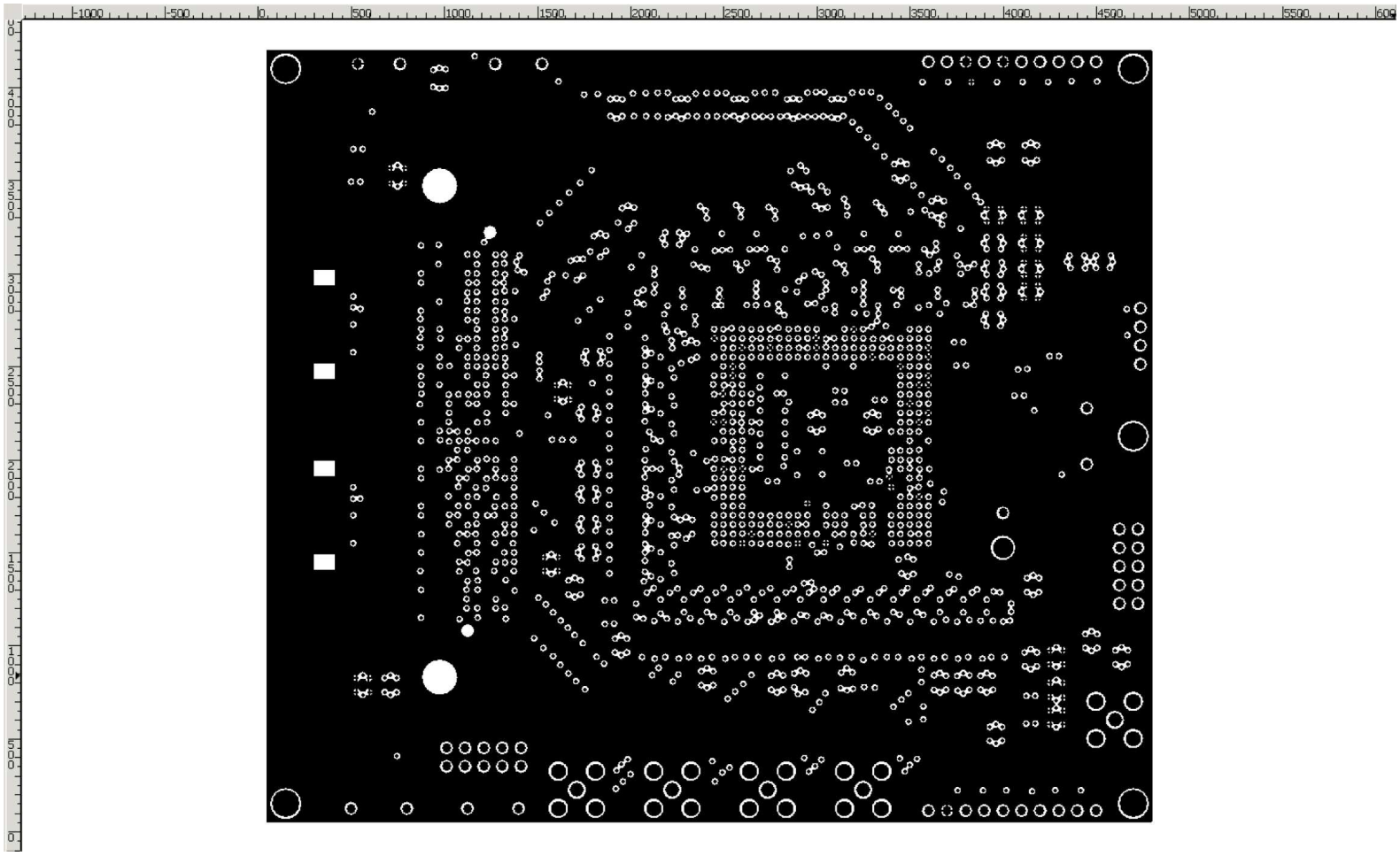


Figure A.10: Signal 7

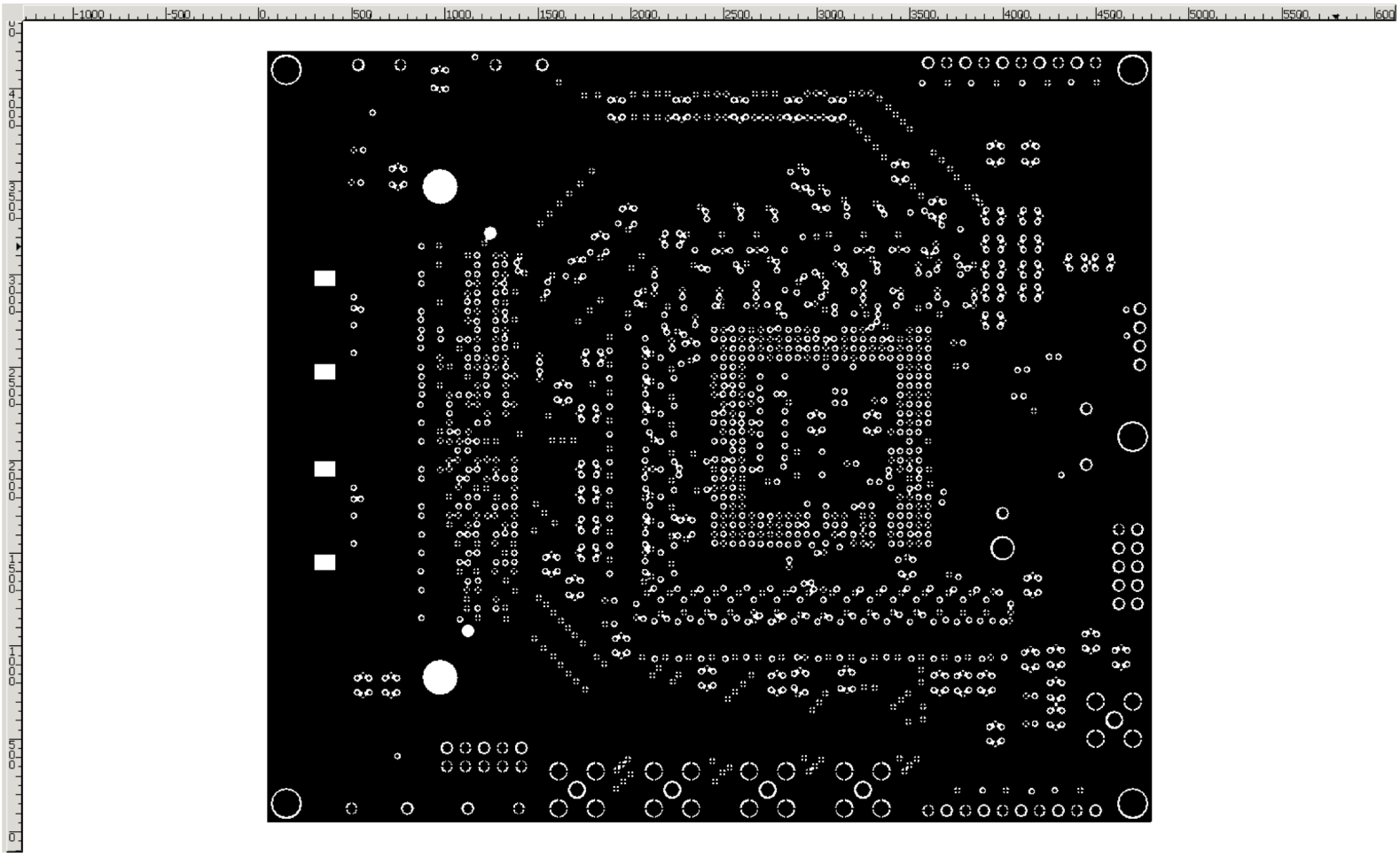


Figure A.11: Signal 8

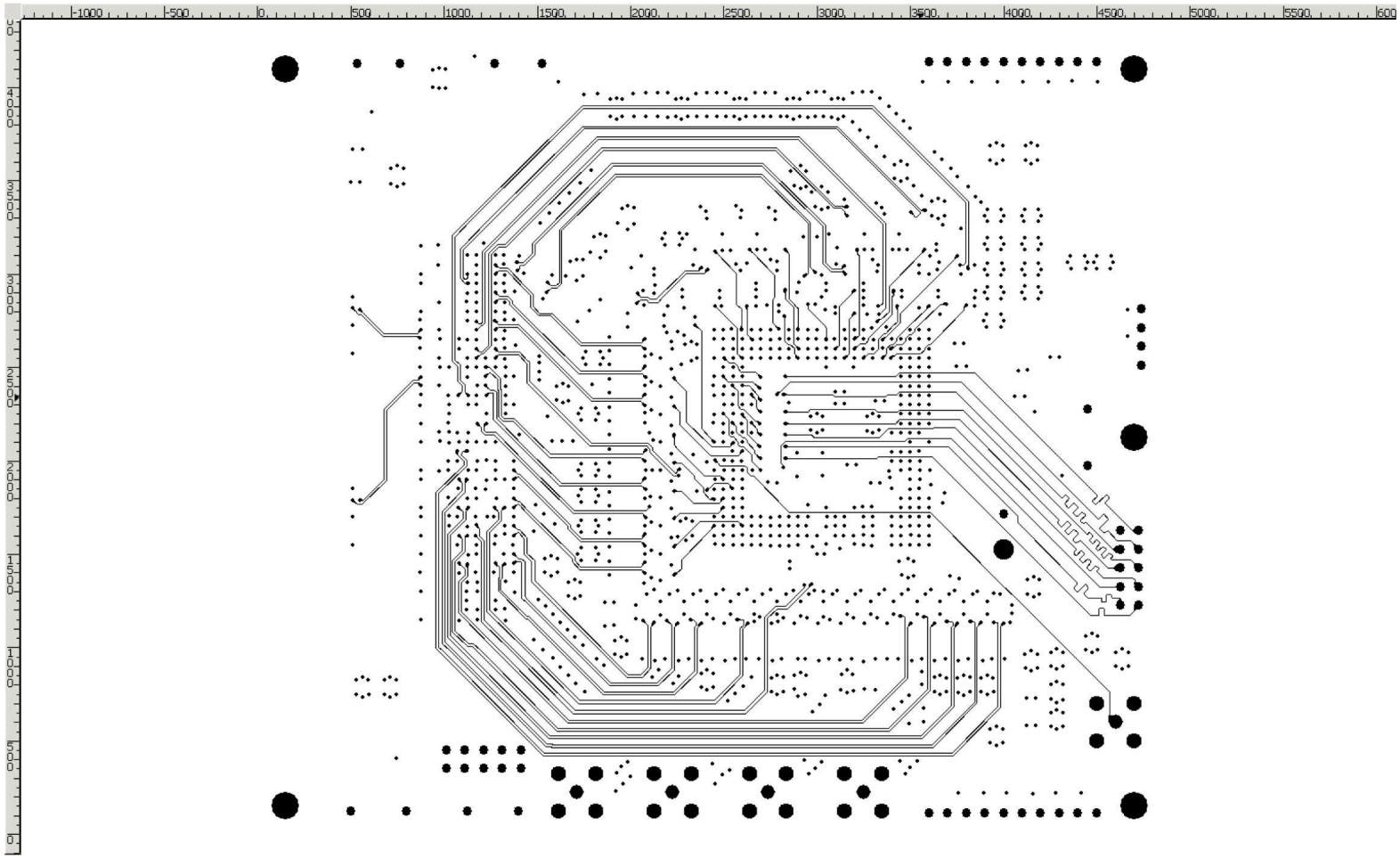


Figure A.12: Signal 9

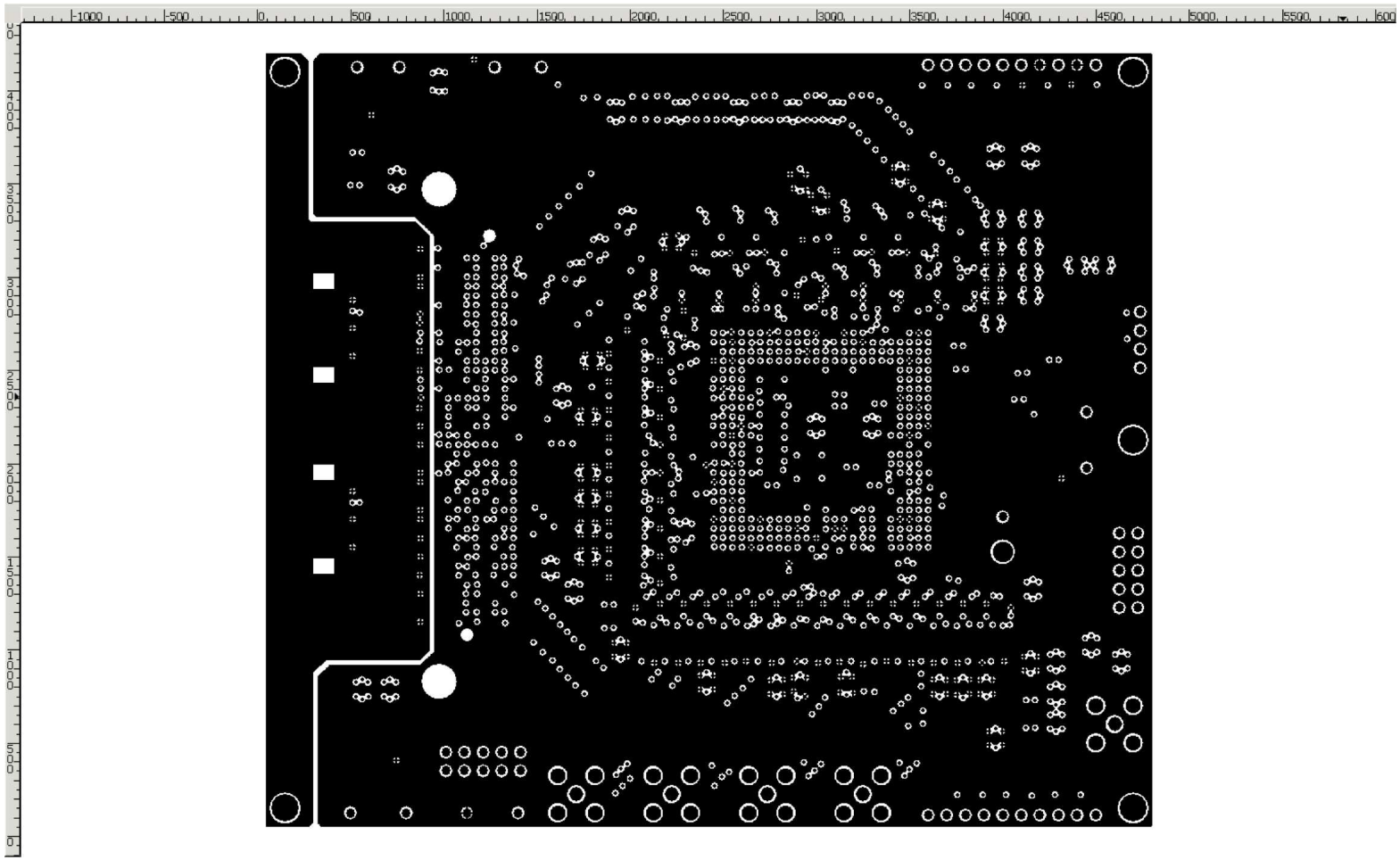


Figure A.13: Signal 10

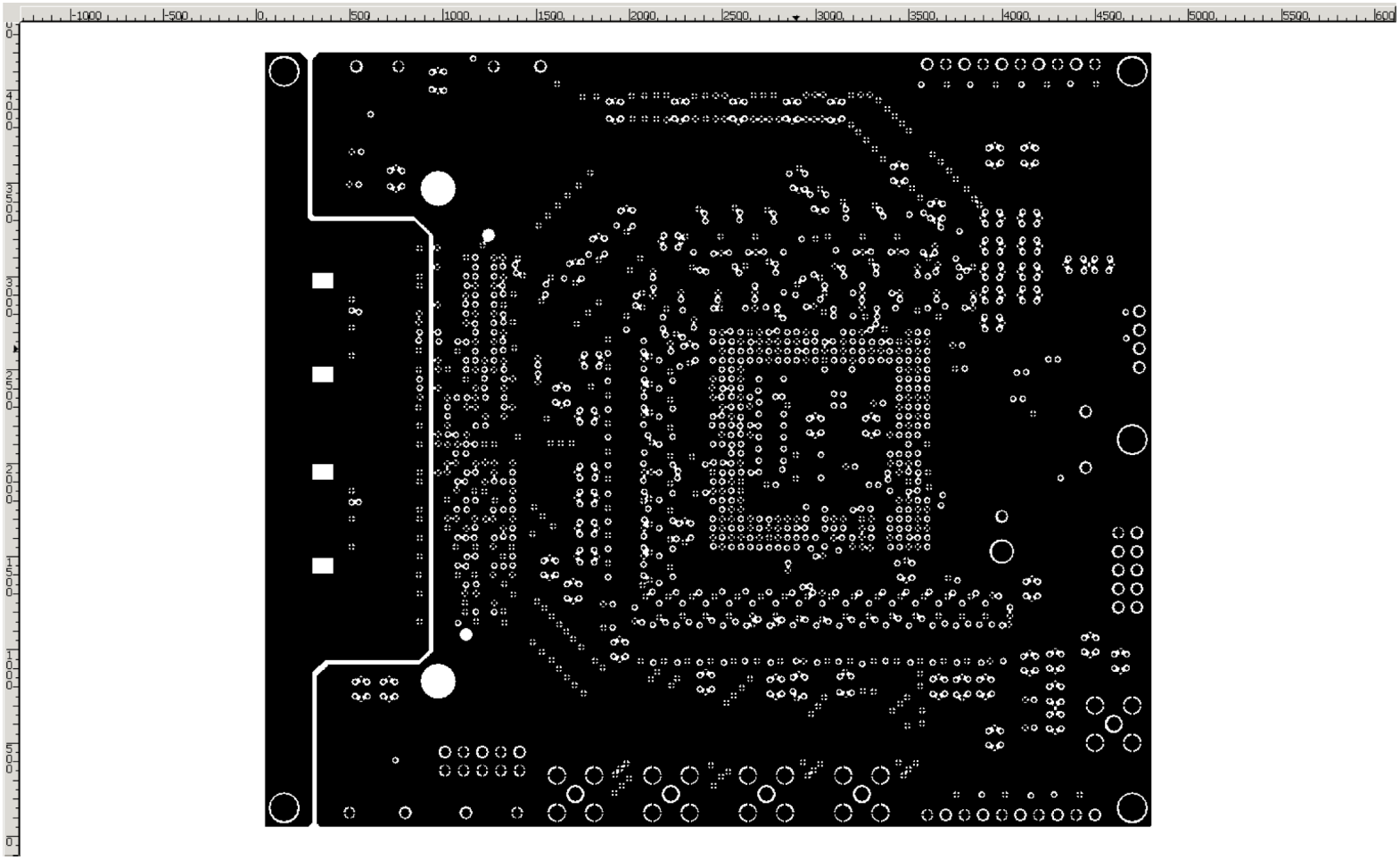


Figure A.14: Signal 11

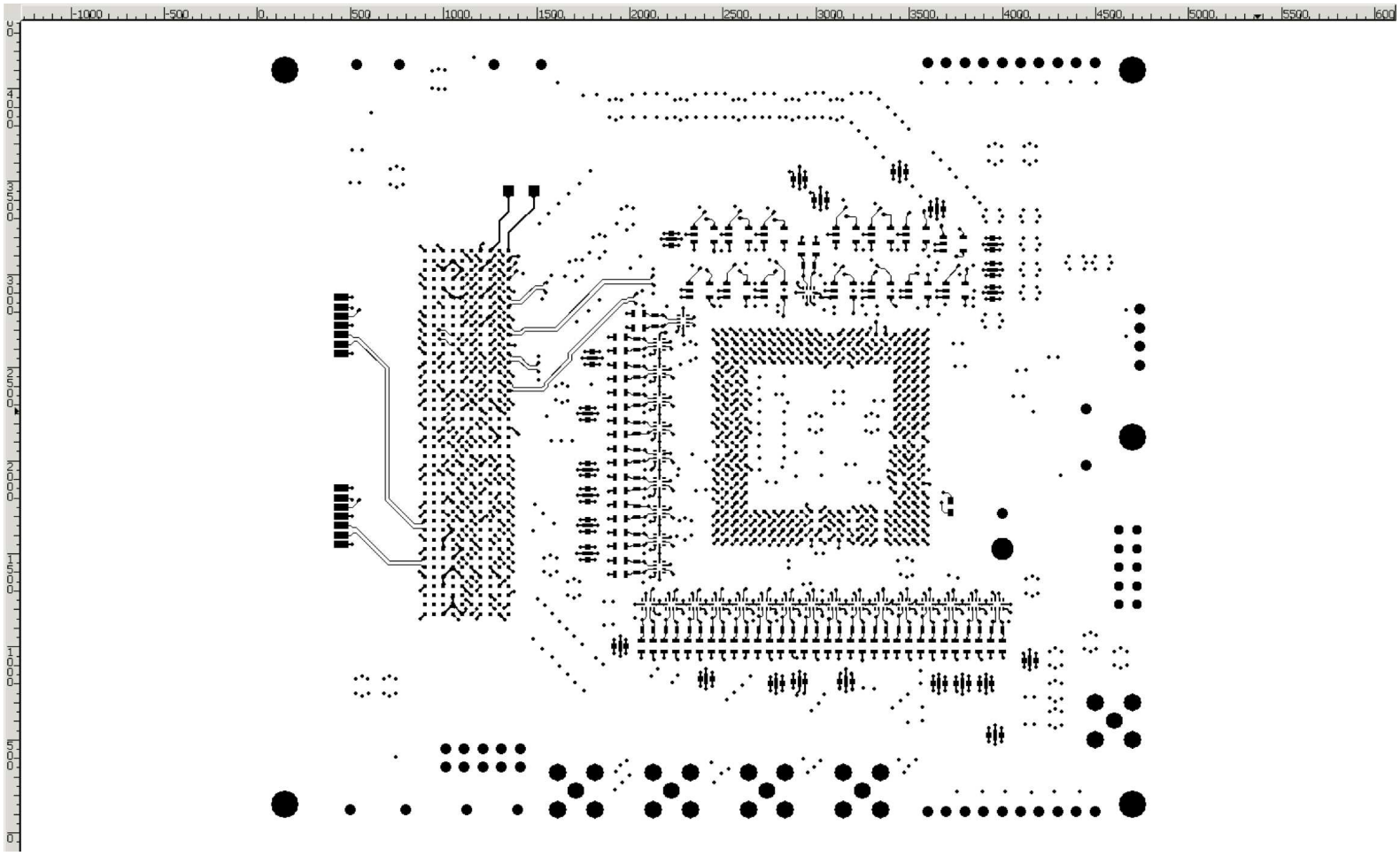


Figure A.15: Bottom signal layer

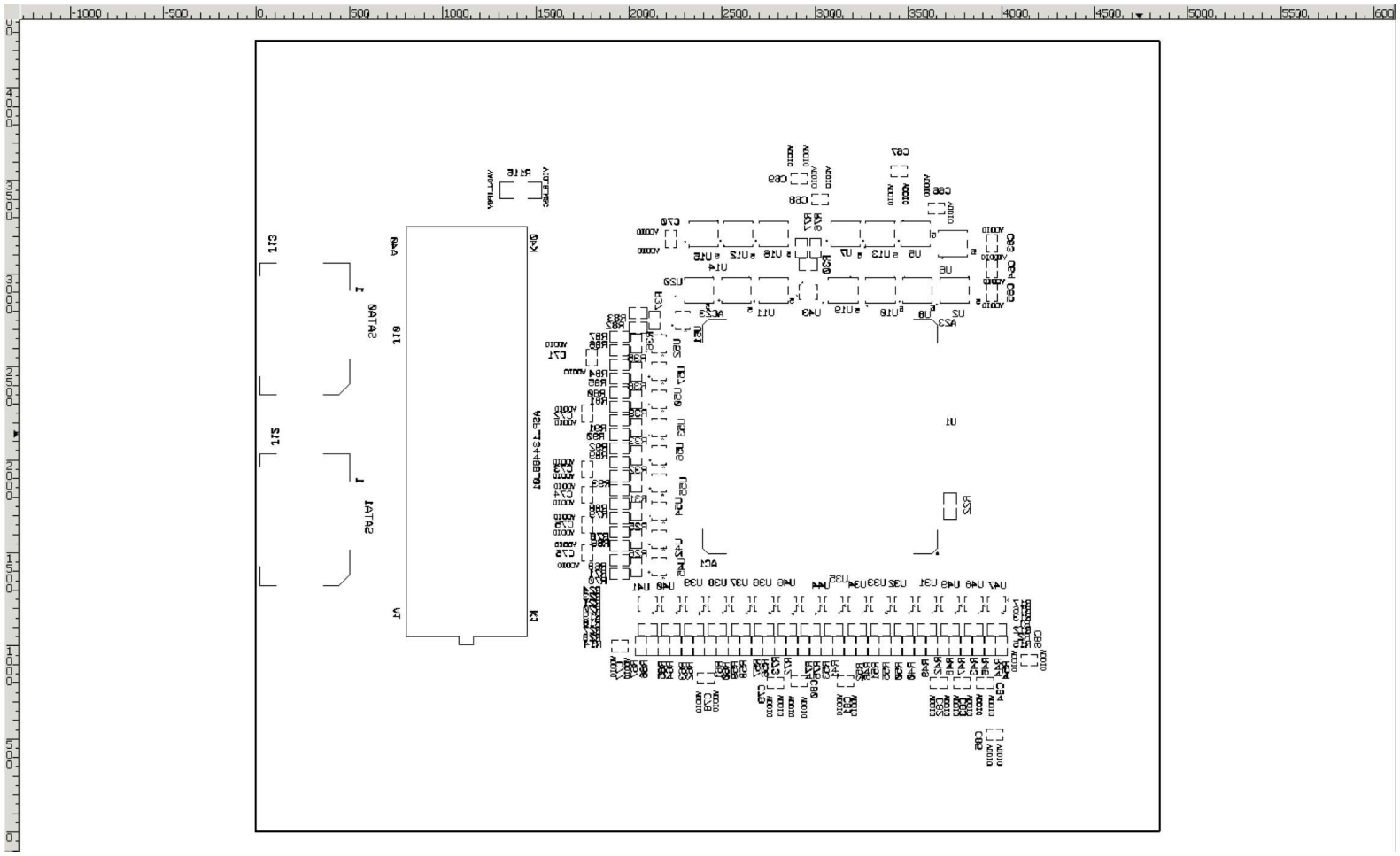


Figure A.16: Bottom silkscreen

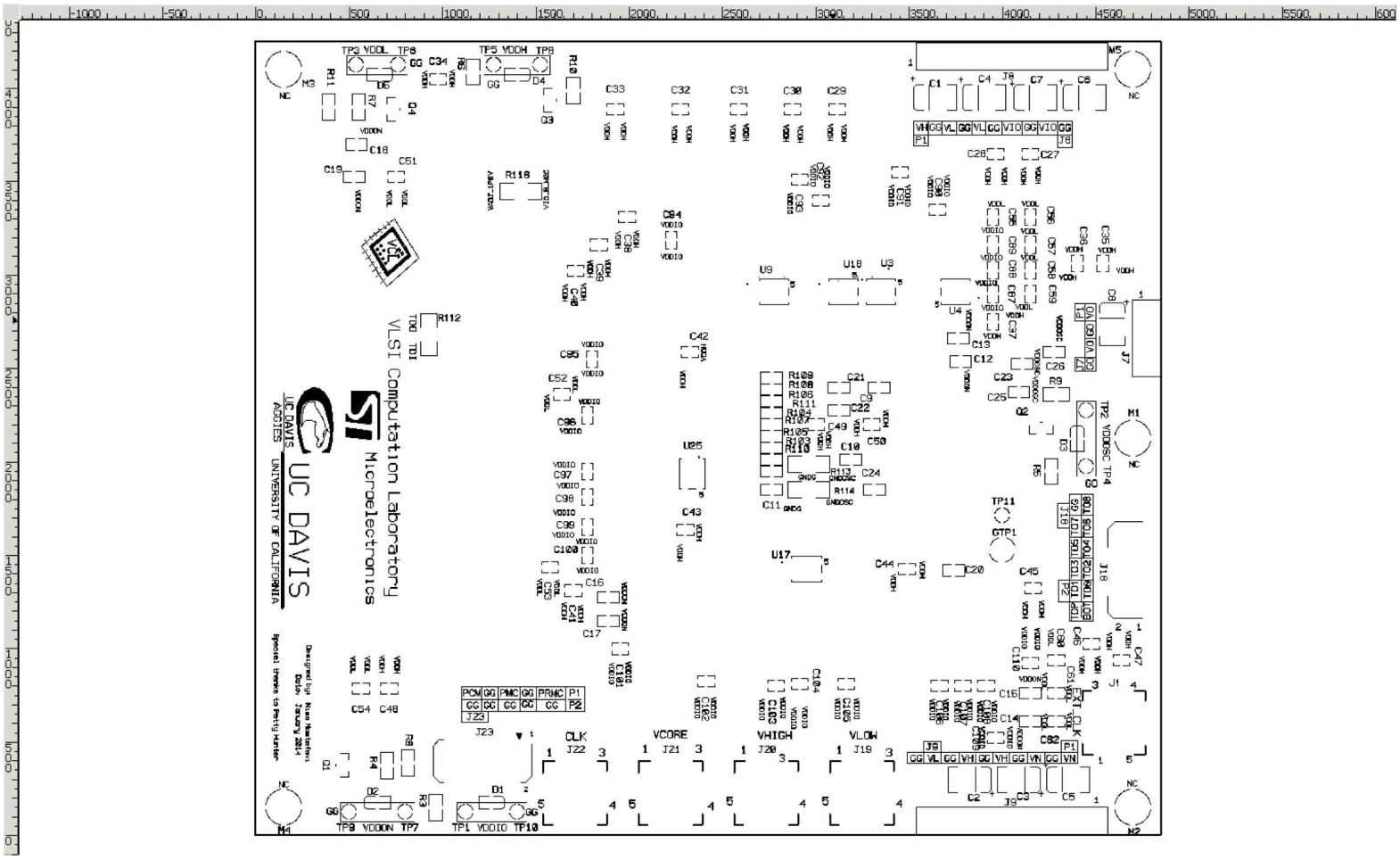


Figure A.17: Top silkscreen

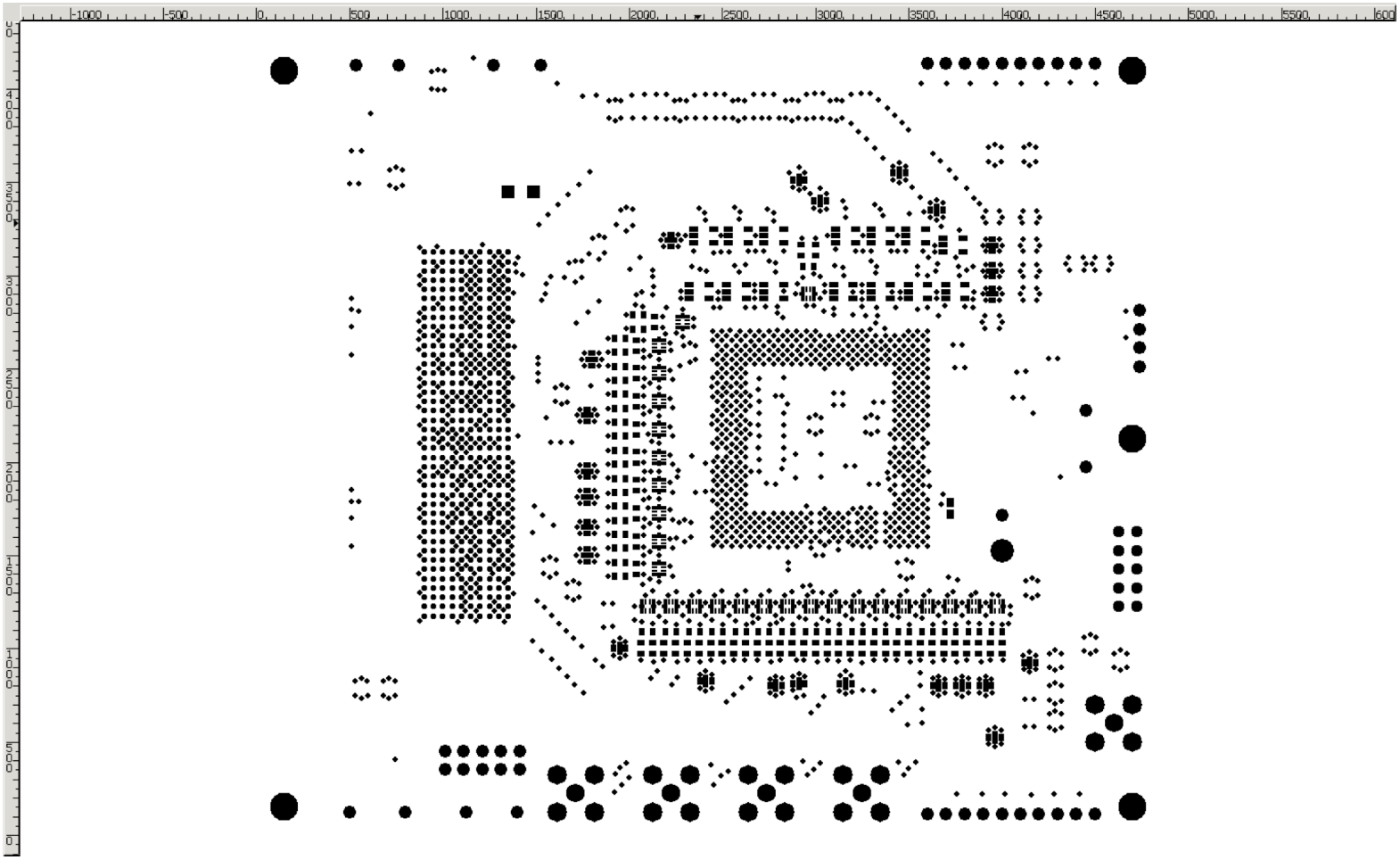


Figure A.18: Bottom soldermask

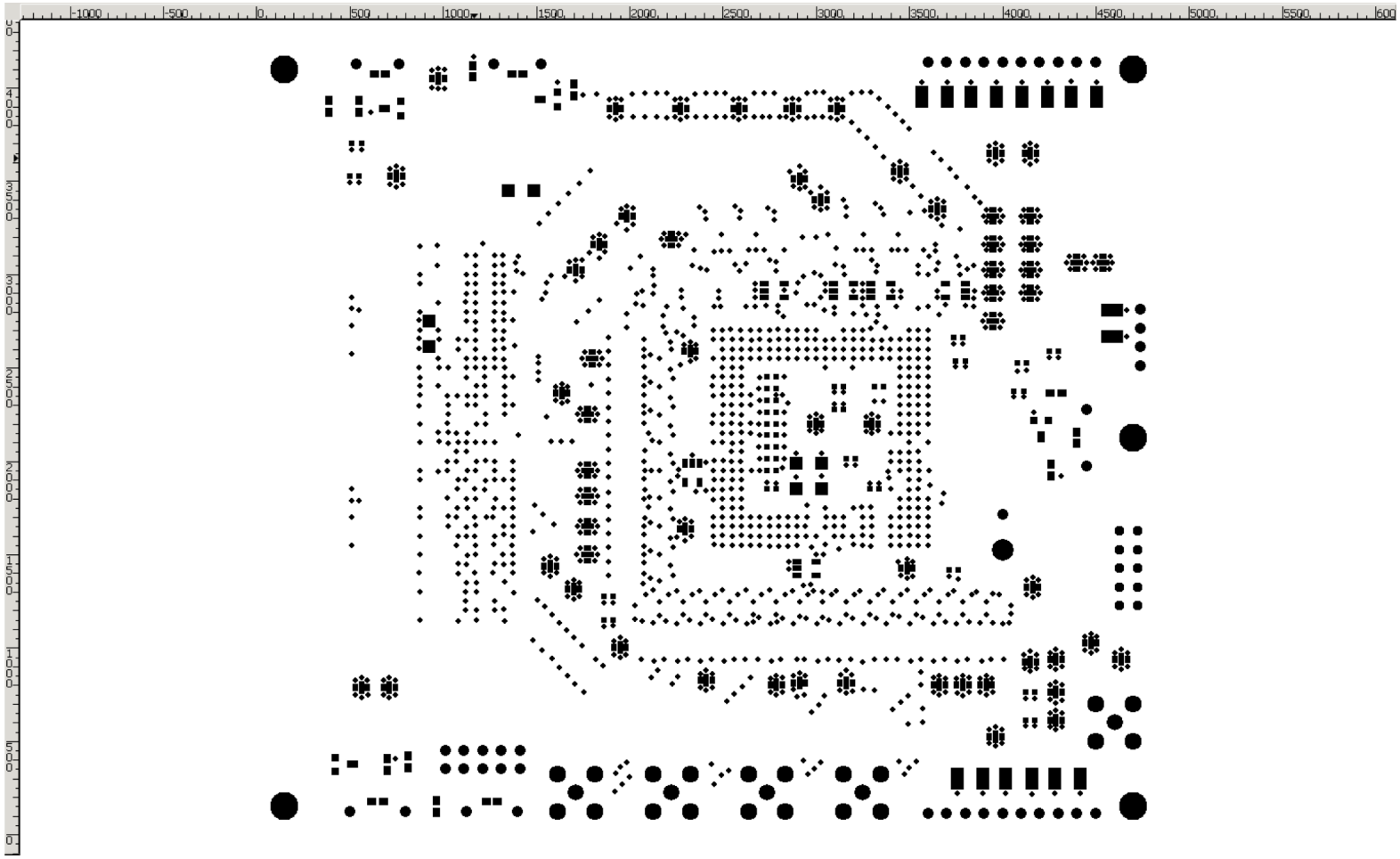


Figure A.19: Top soldermask

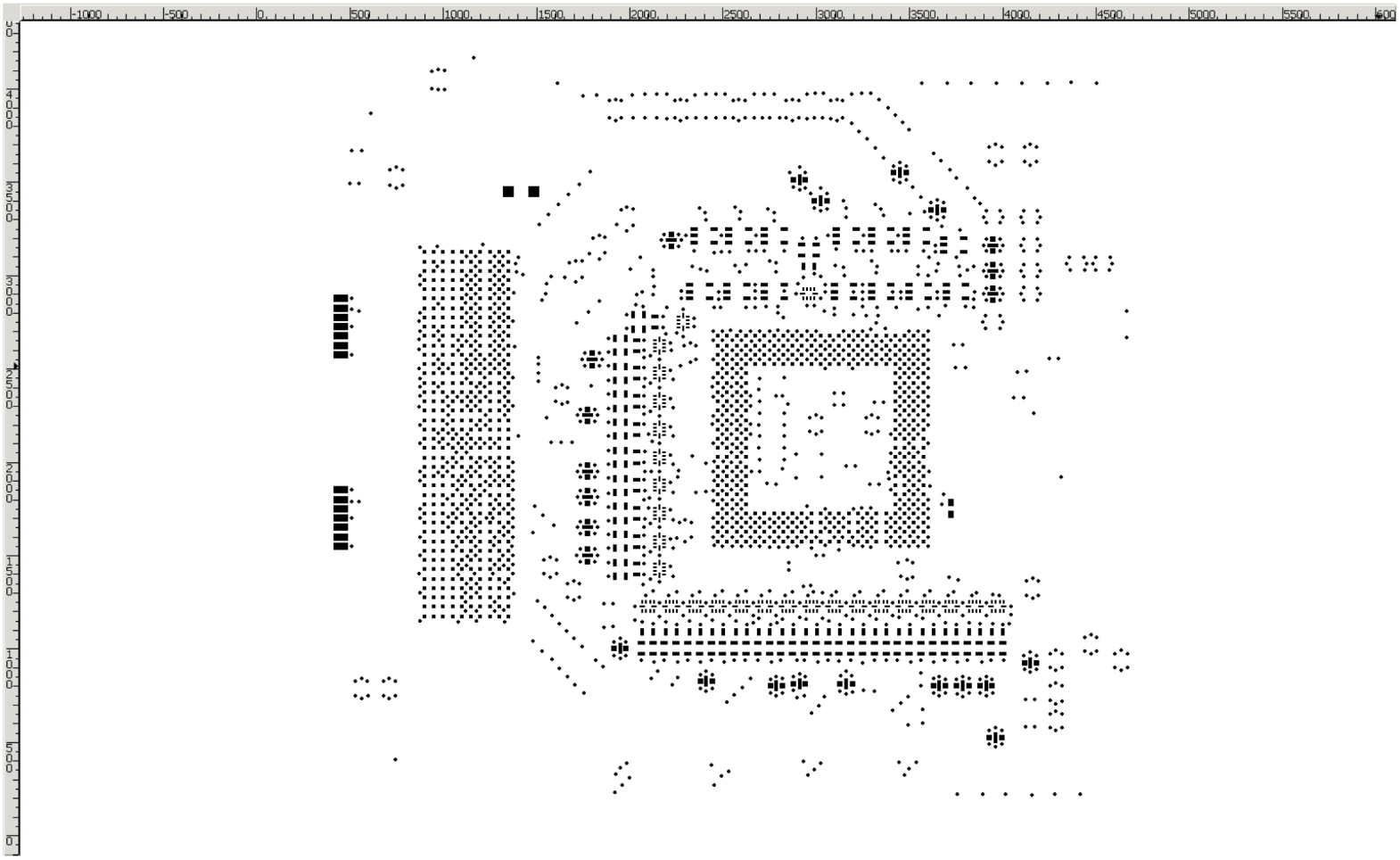


Figure A.20: Bottom paste mask

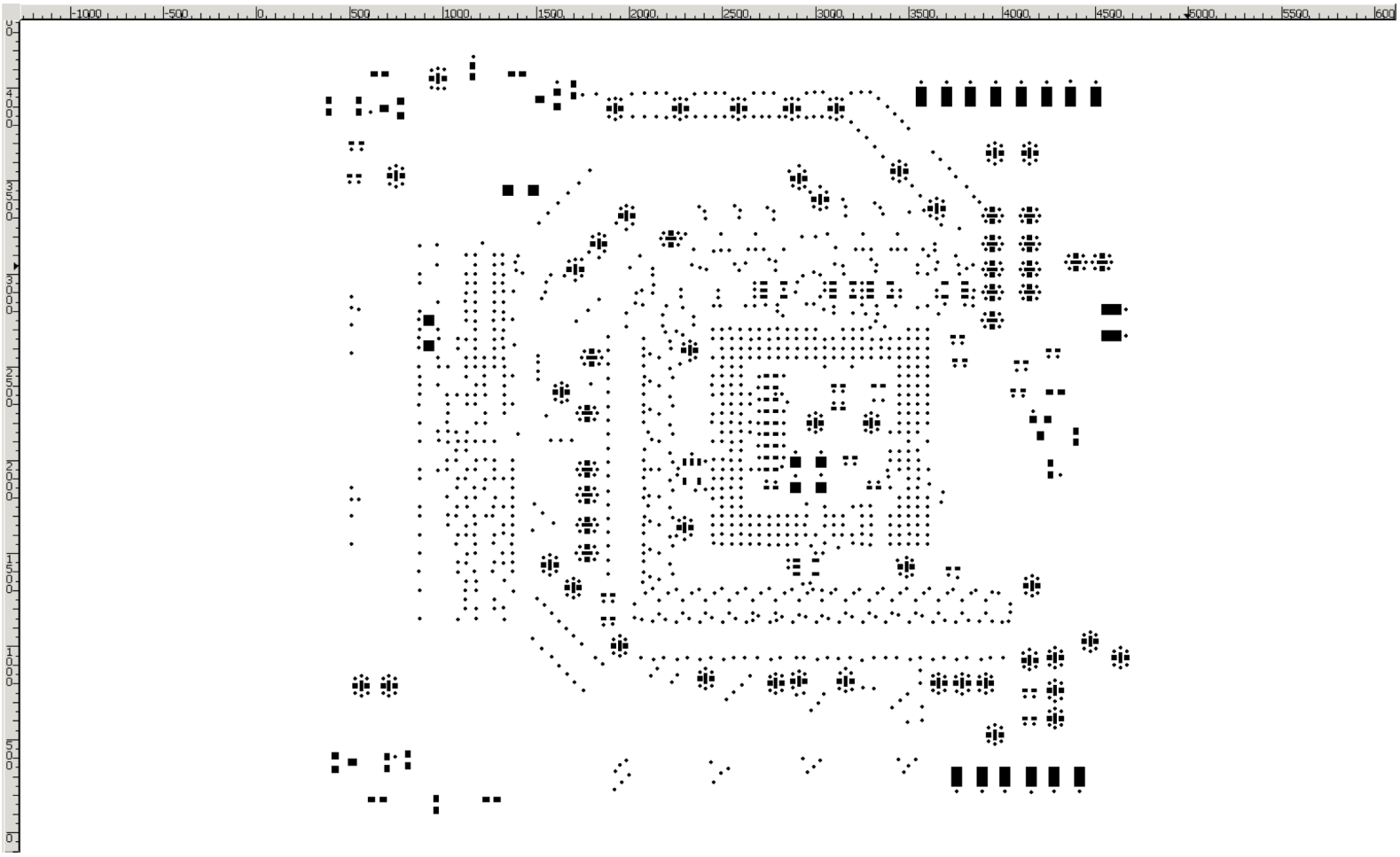


Figure A.21: Top paste mask

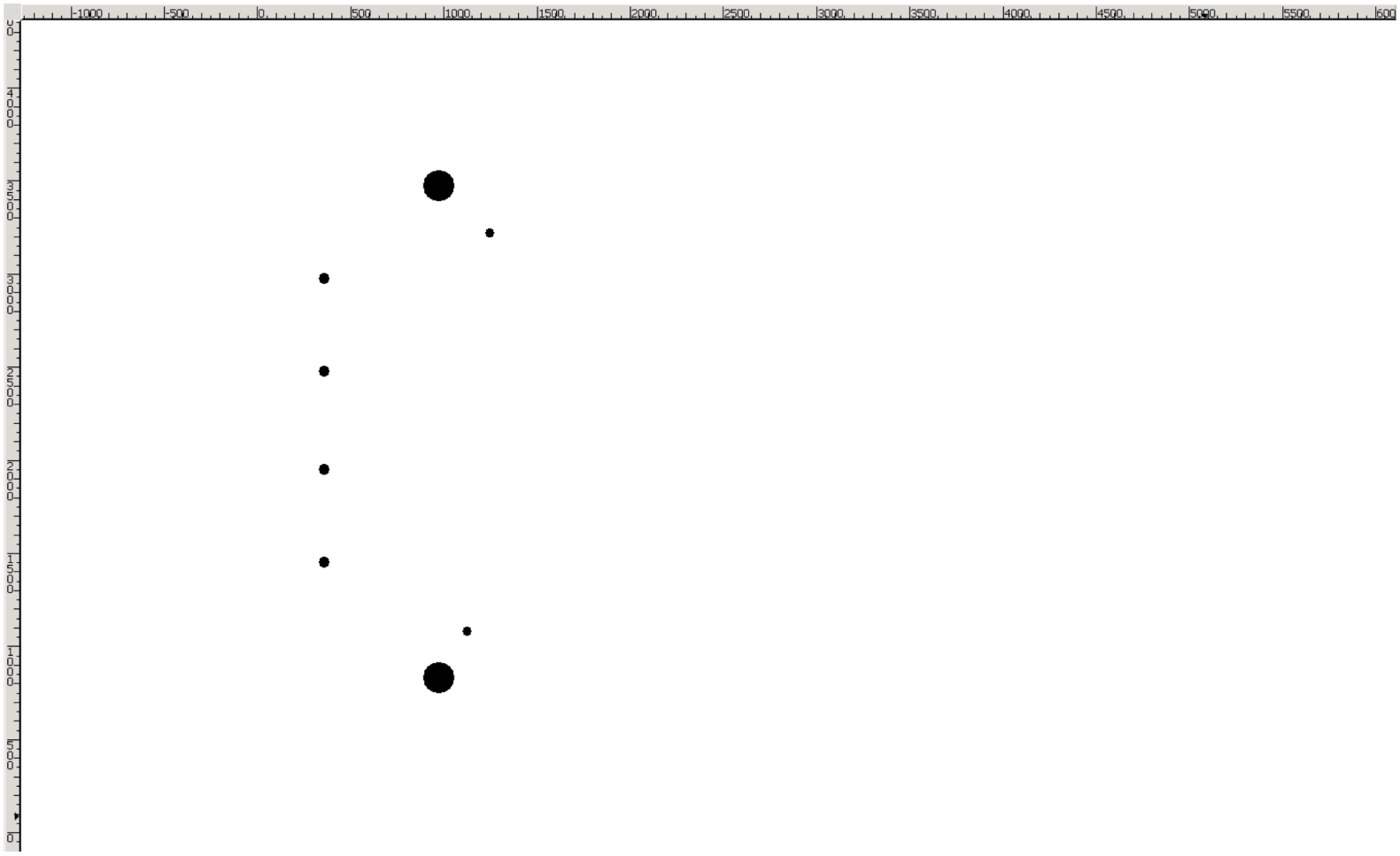


Figure A.22: Non plated drills

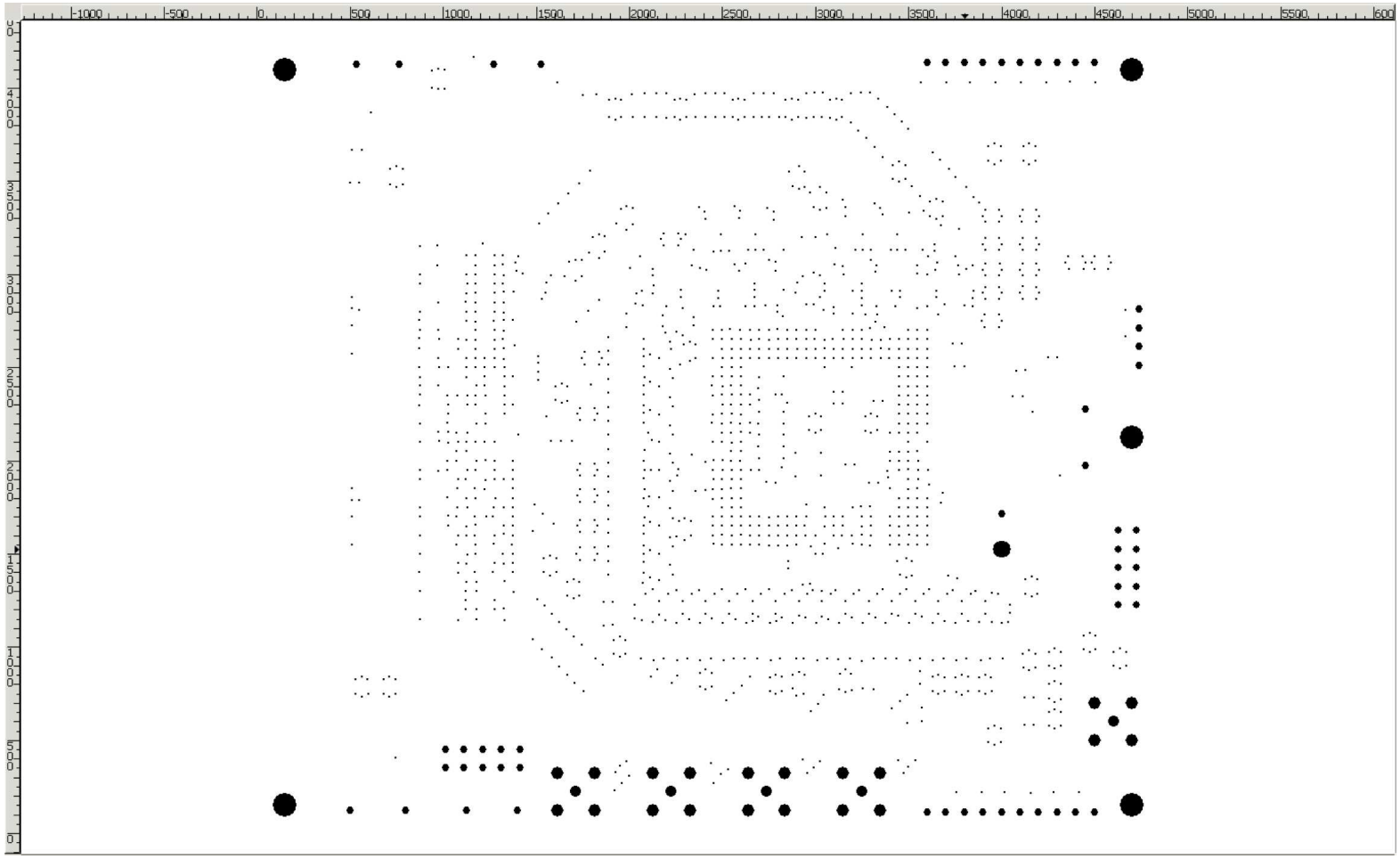


Figure A.23: Plated drills

Appendix B

Schematic View of the PCBoard Desgin

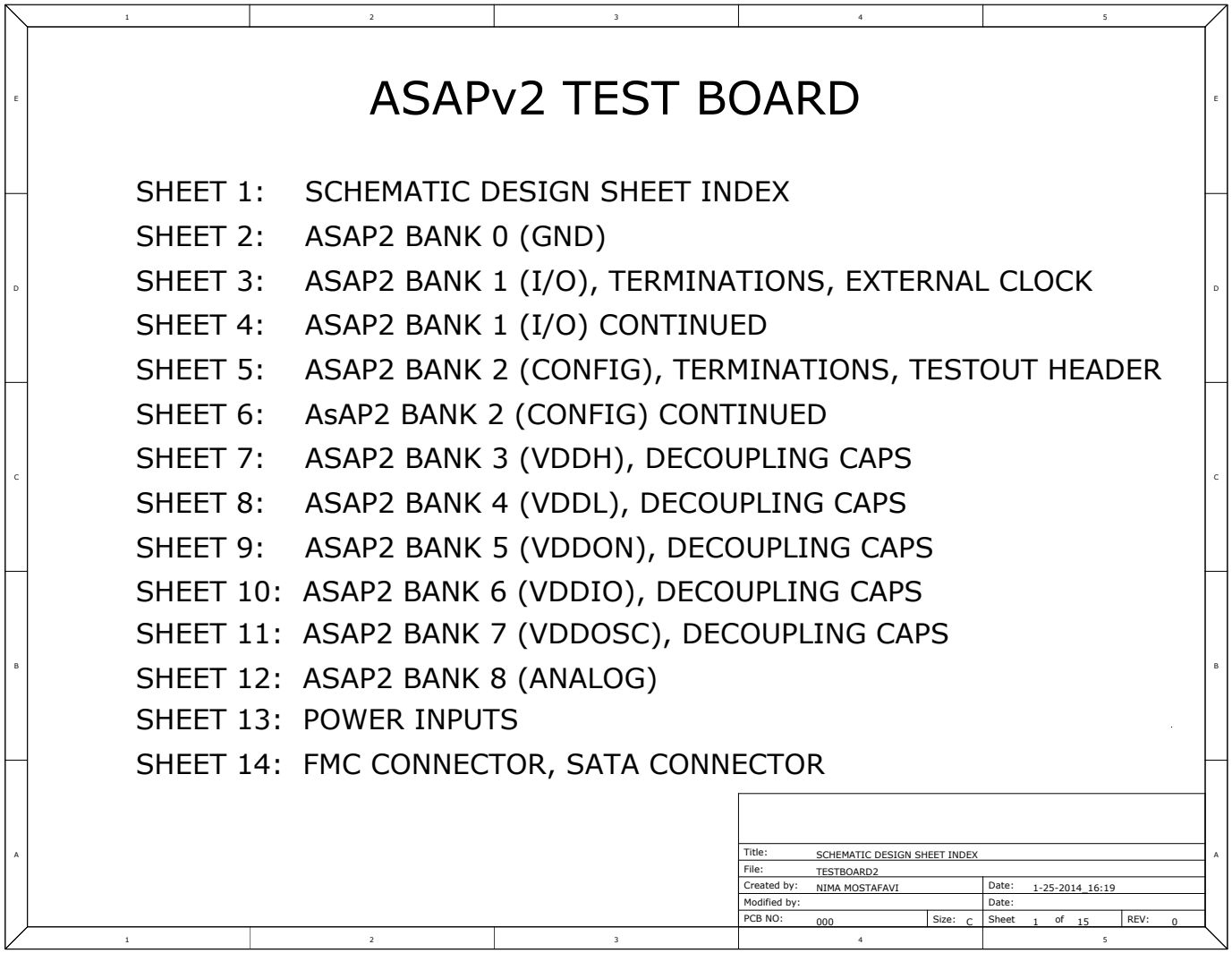


Figure B.1: Schematic design sheet index

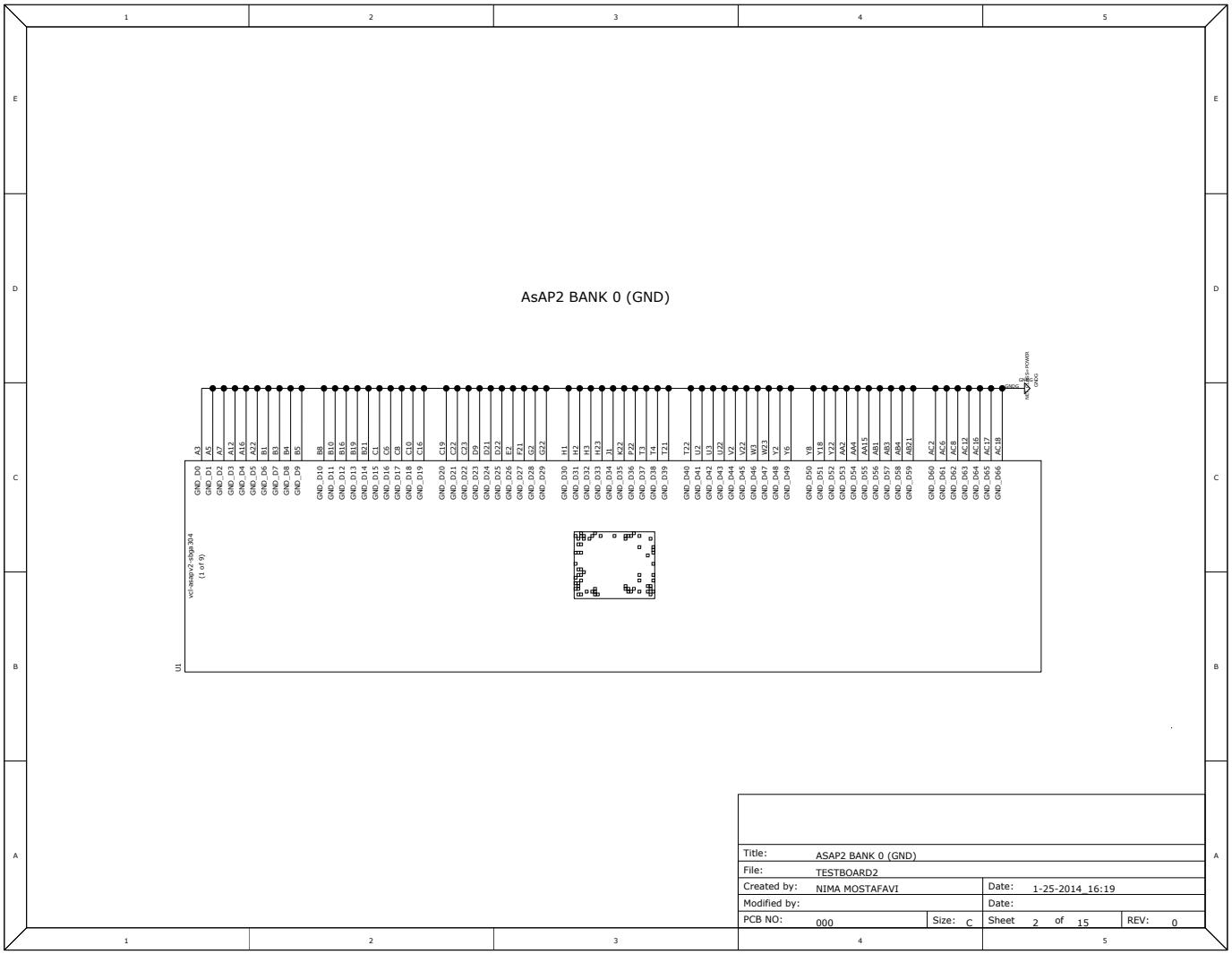


Figure B.2: AsAP2 bank 0 (GND)

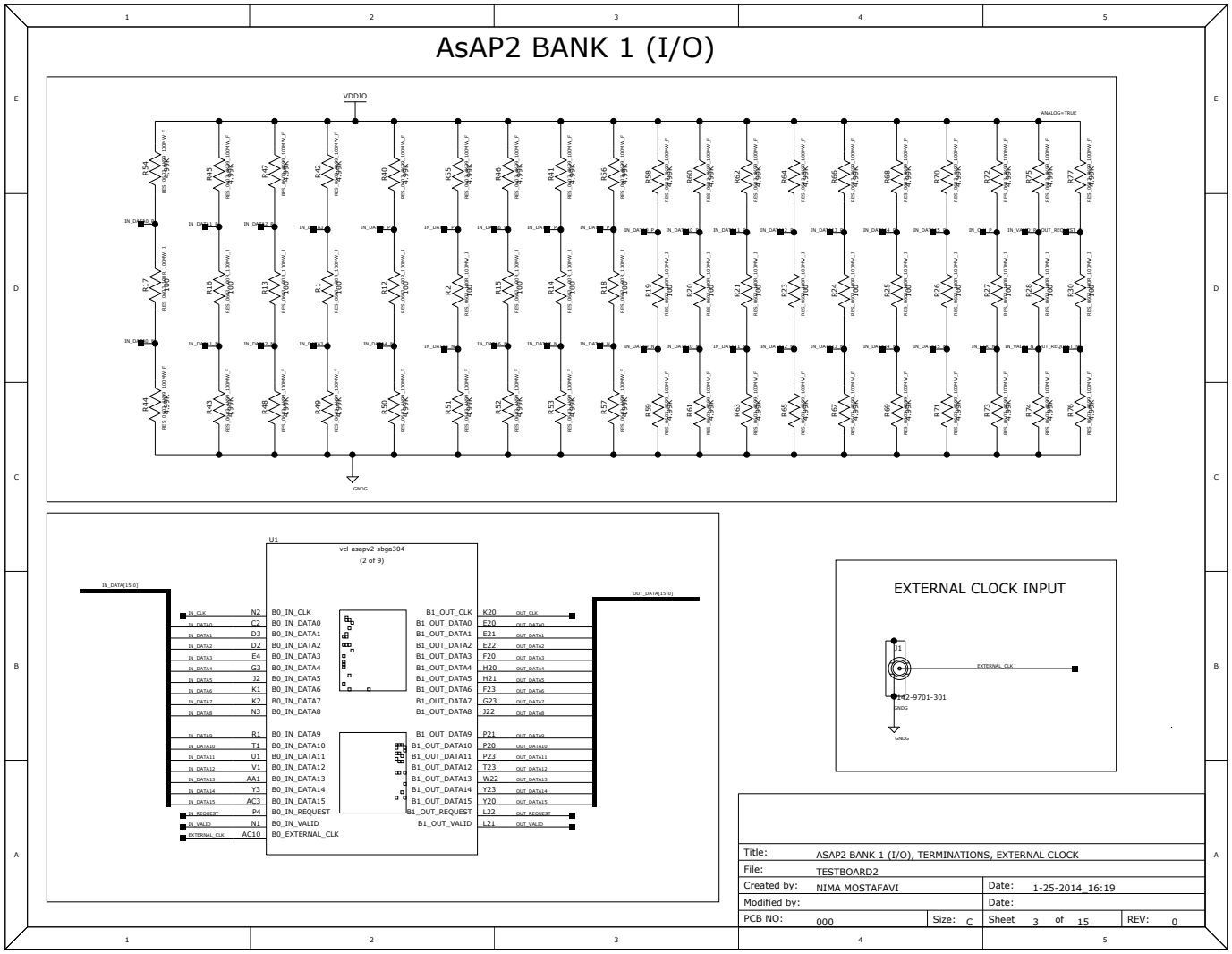


Figure B.3: AsAP2 bank 1 (I/O), terminations, external clock

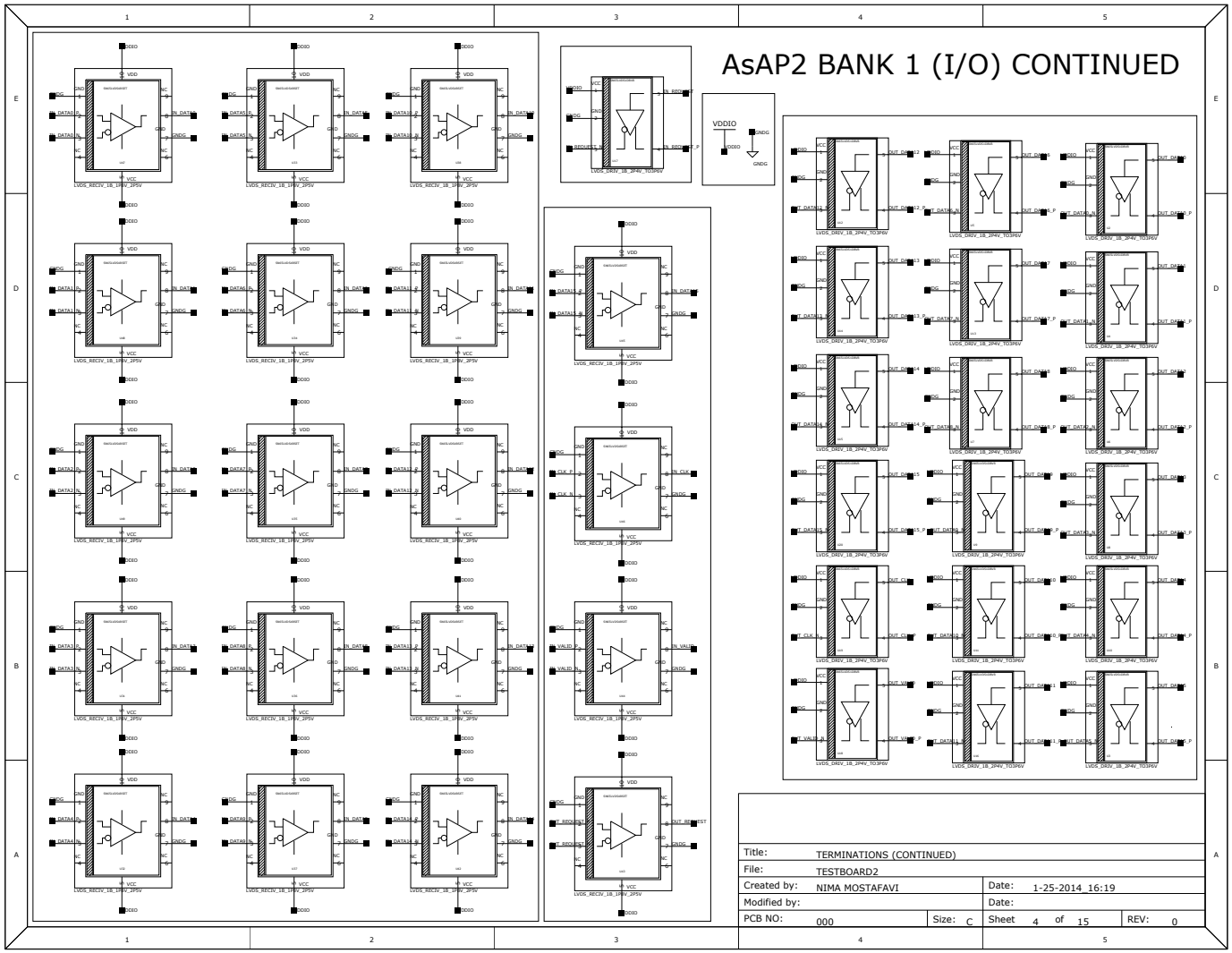


Figure B.4: AsAP2 bank 1 (I/O) continued

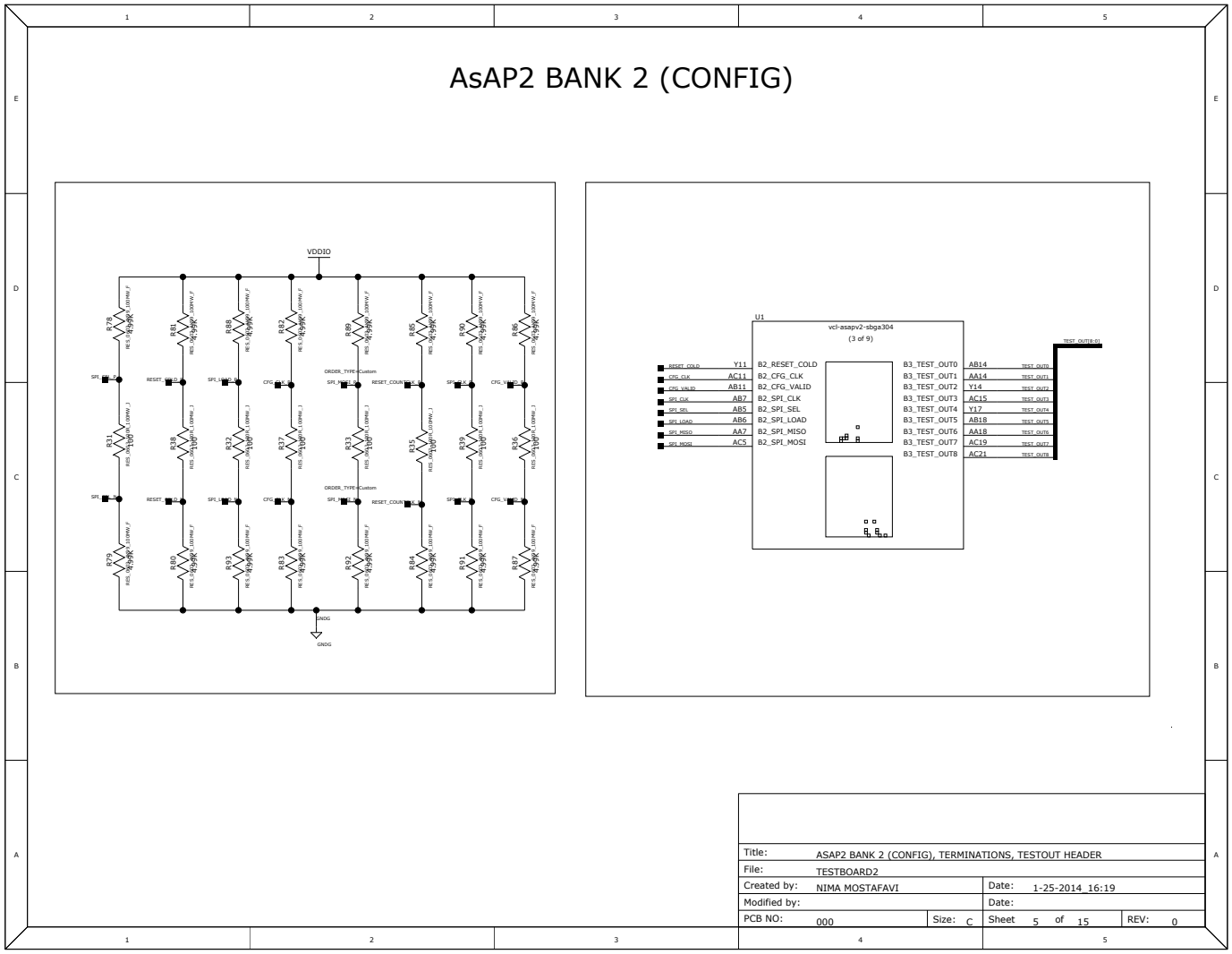


Figure B.5: AsAP2 bank 2 (*config*), terminations, *testout* header

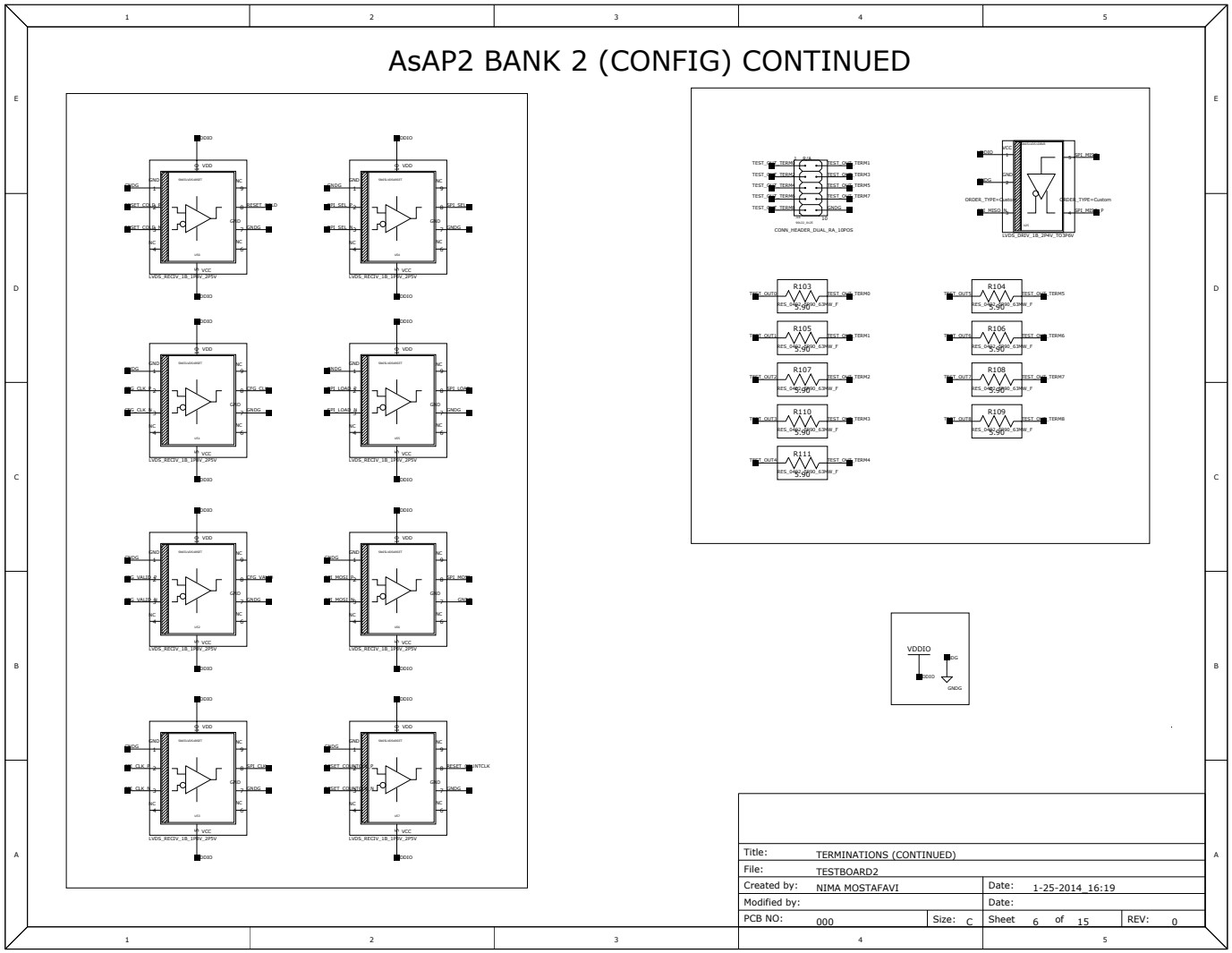


Figure B.6: AsAP2 bank 2 (*config*) continued

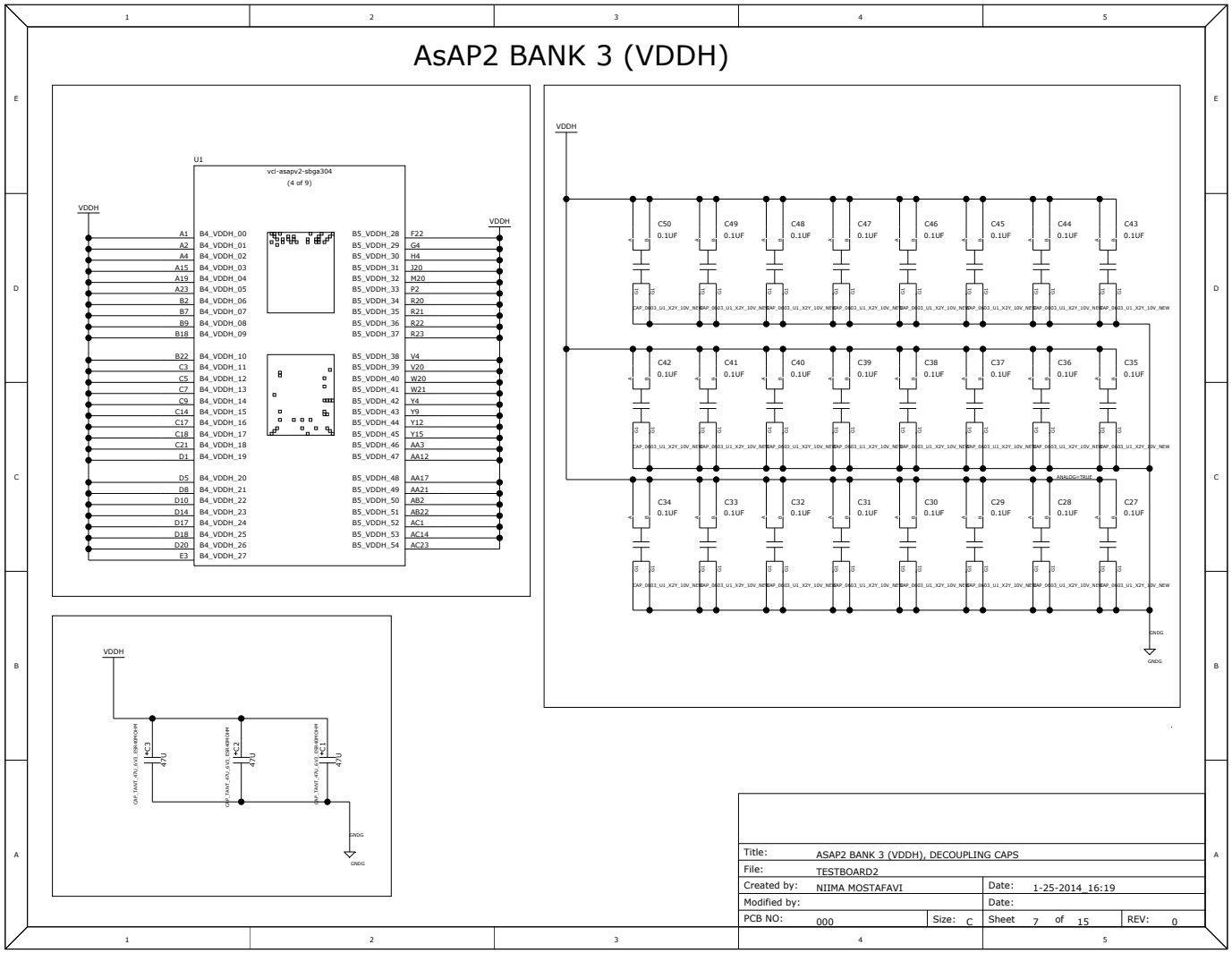


Figure B.7: AsAP2 bank 3 (VDDH), decoupling capacitors

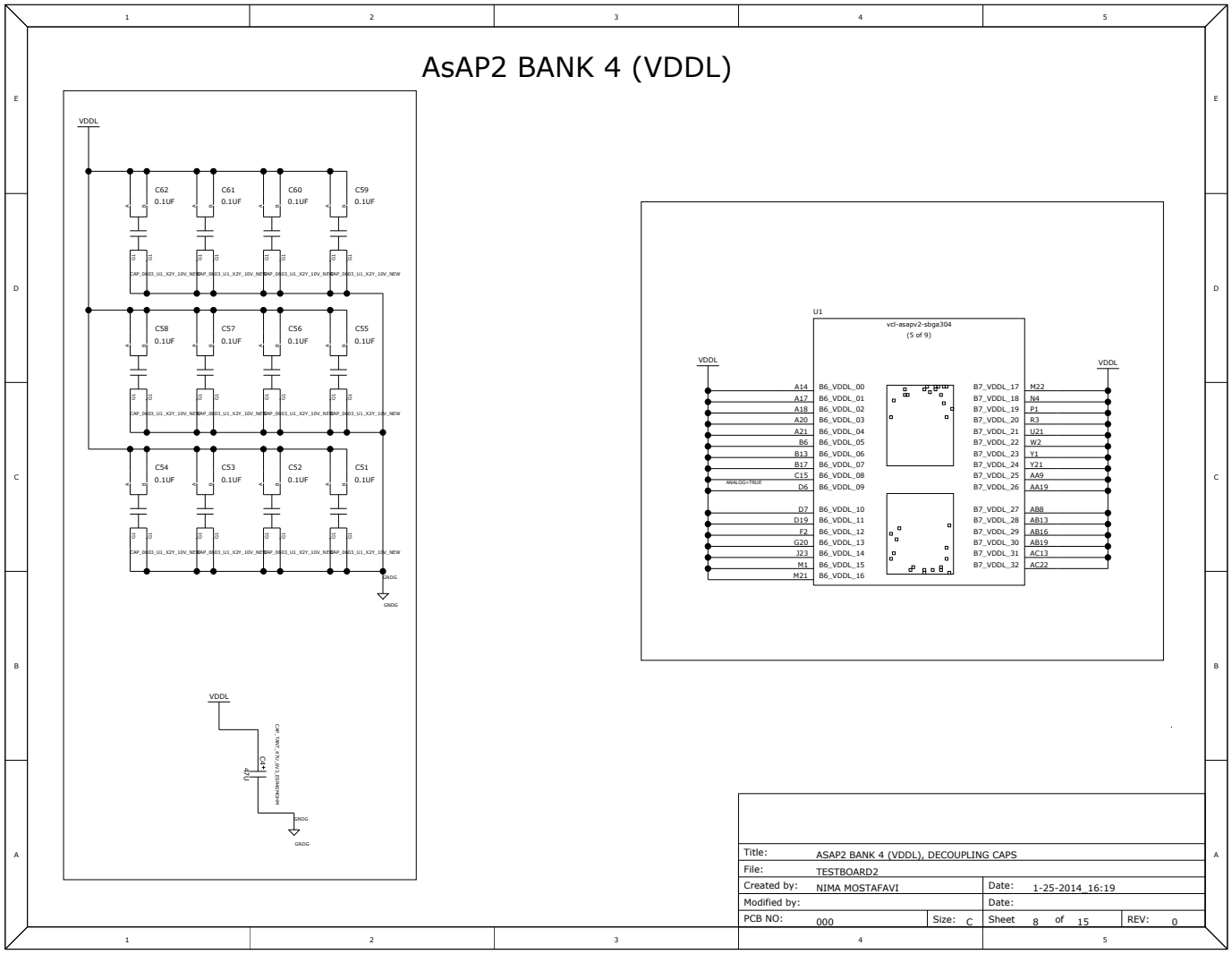


Figure B.8: [AsAP2 bank 4 (VDDL), decoupling capacitors

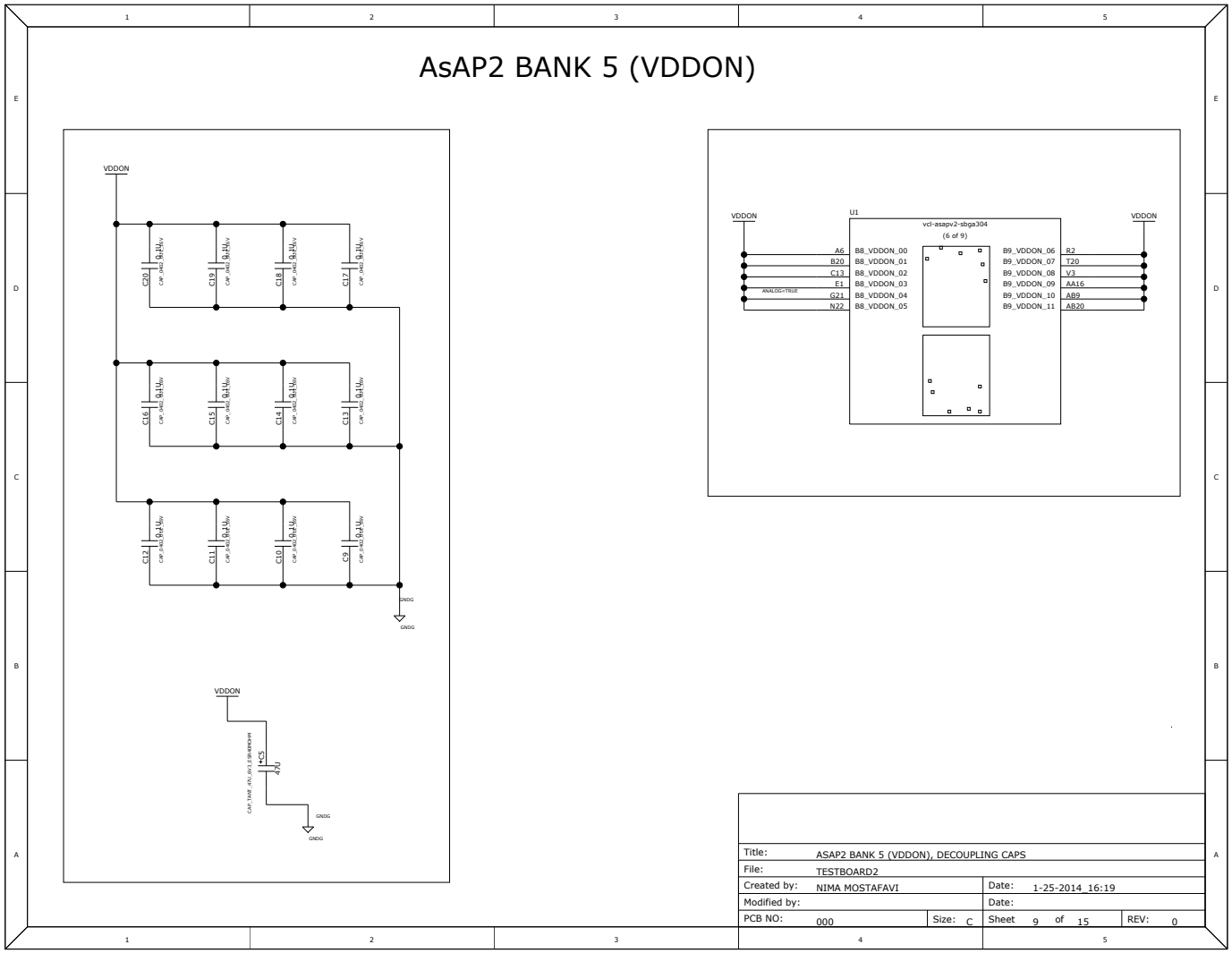


Figure B.9: AsAP2 bank 5 (VDDON), decoupling capacitors

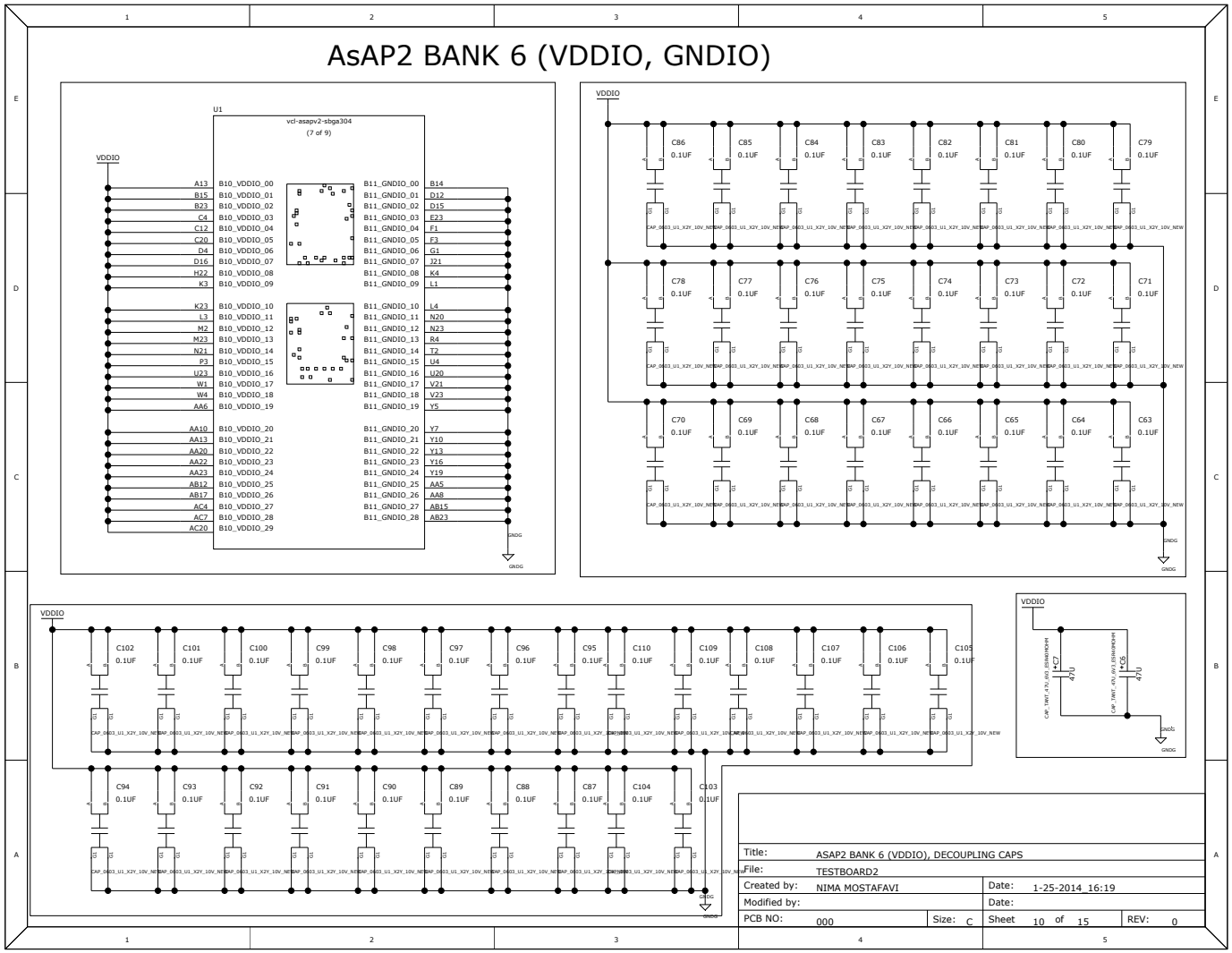


Figure B.10: AsAP2 bank 6 (VDDIO), decoupling capacitors

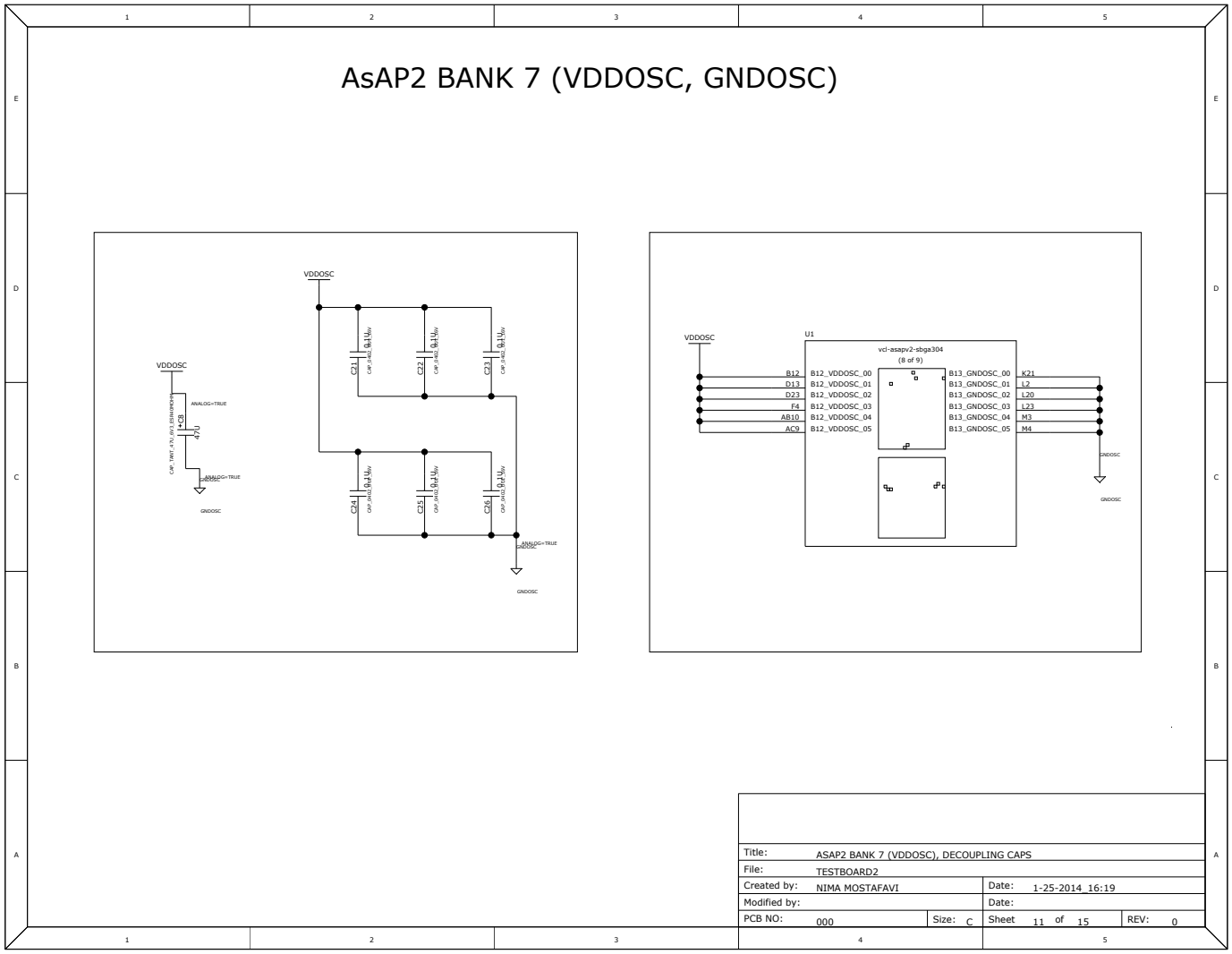


Figure B.11: AsAP2 bank 7 (VDDOSC), decoupling capacitors

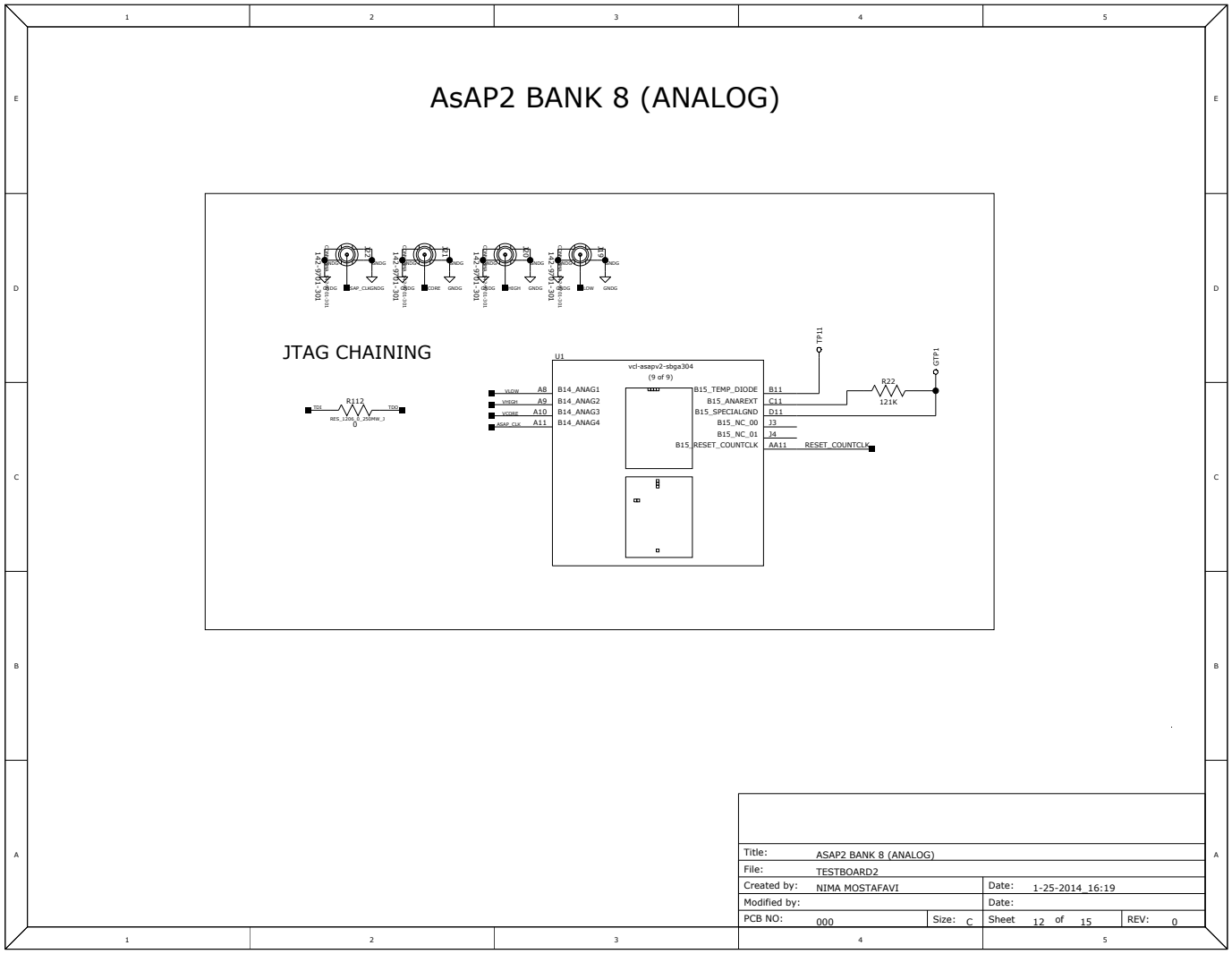


Figure B.12: AsAP2 bank 8 (analog)

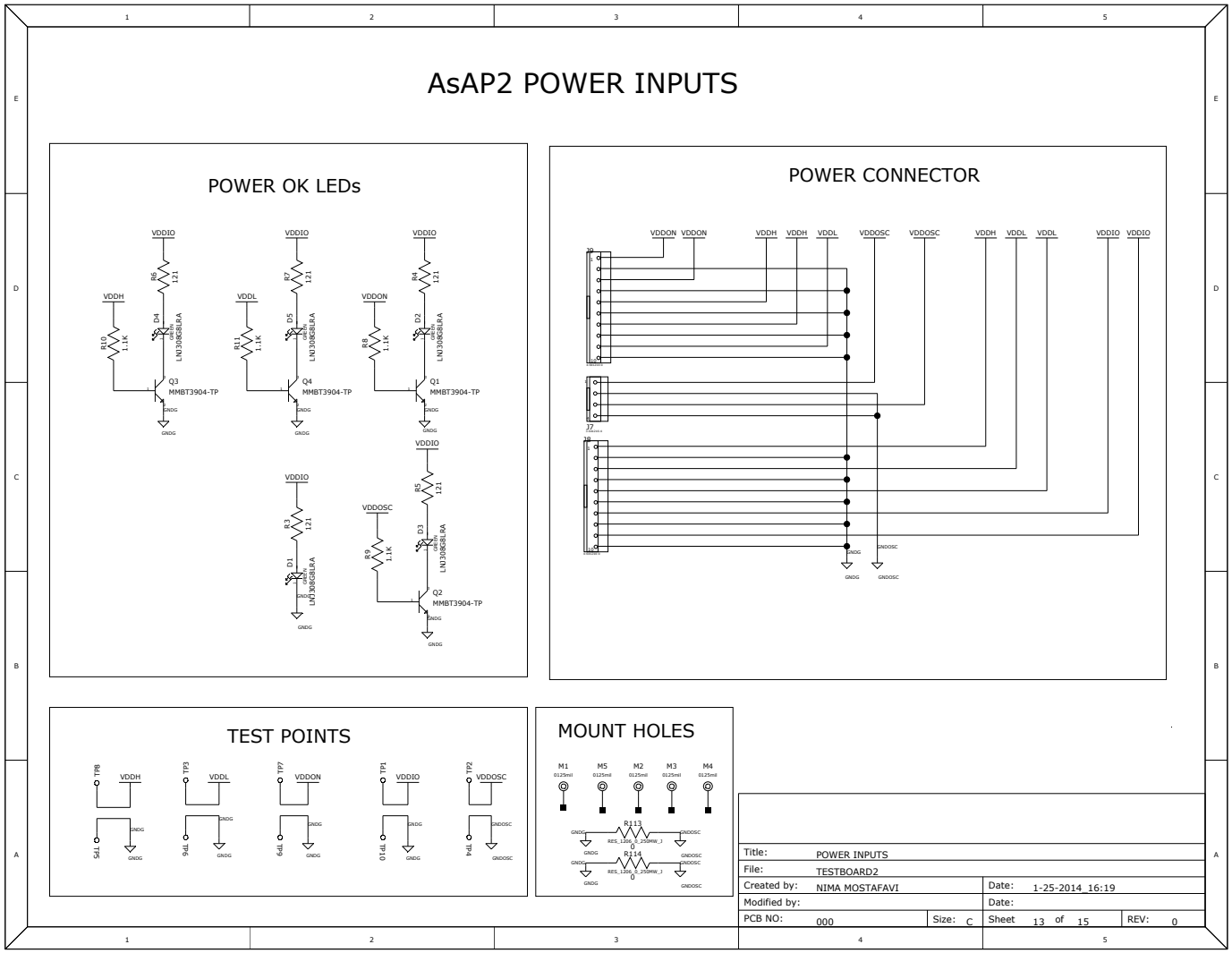


Figure B.13: Power inputs

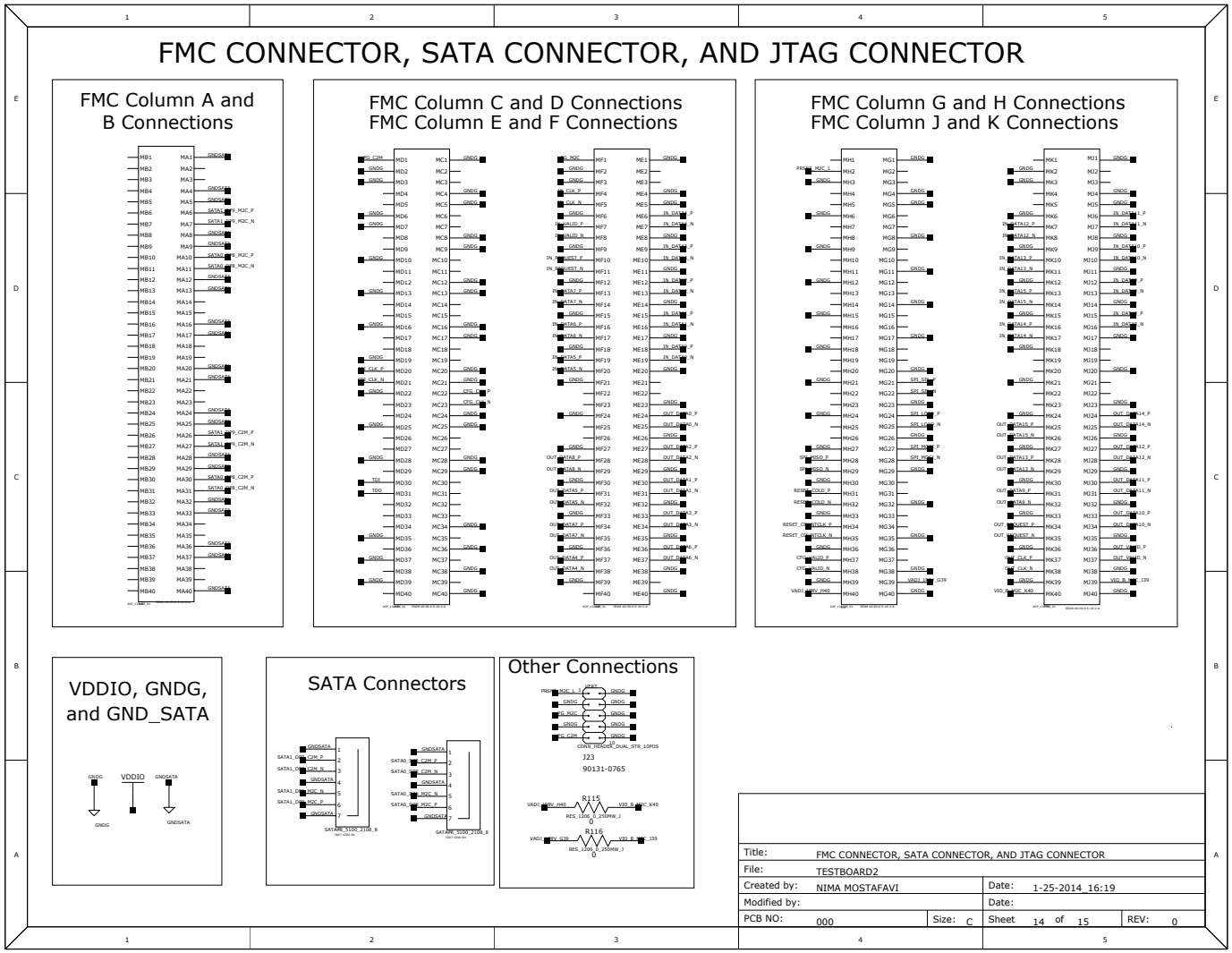


Figure B.14: FMC connector, SATA connector

Appendix C

Scheduler Code Changes and Improvements

1. The `rptb/rpt` inline `NOP3` is fixed.
2. The `rptb`, block size is fixed.
3. The `rpt/rptb` makes separate basic blocks to fix and prevent the entrance of outside of `rpt/rptb` instruction into the `rpt/rptb` block.
4. The no comment capability is added.
5. Some issues with spaces in `DMEM/DCMEM` addresses are fixed.
   Example: dmem5 —-gets converted to—-> dmem 5
6. The issue with spaces in `begin` is fixed.
   Example: begin 0 , 0 or begin 0,0 are recognizable now.
7. The `rptb`, block size increase or decrease due to insertion of `NOP`s is added.
8. The Read After Read (RAR) dependency for `ibuf` and `ibufnap` instruction is added.
9. The `rpt/rptb` dependency to the instruction after `rpt/rptb` with a new line of `NOP` is fixed.
   Example:

```
rpt 5
add dmem 2 dmem 3 dmem 4
sub dmem 5 dmem 2 dmem 2
```

becomes:

```
rpt 5 nop3
add dmem 2 dmem 3 dmem 4
nop nop2 // NOP inserted to resolve dependency
sub dmem 5 dmem 2 dmem 2
```

10. The issue with `pcptr` in branch labels is fixed.

11. The issue with label spacing between two instructions and their correct calculated dependencies is fixed.

12. The `brms1 mm.2` and `brms2 mm.2` in instruction_src.h is changed to `brms1 mm.3` and `brms2 mm.3` since they put extra `NOP` for min and max before `brms1` and `brms2`.

13. The `DCMEM 18`, `NOP` count to an instruction with `obuf` is fixed.

14. The `NOP` count between `min/max` and `brms1/2` is fixed to the correct value.

15. The capability of having labels starting with numbers is added.

16. The issue with not considering the `rpt/rptb` delay from instruction before `rpt/rptb` block to instruction inside `rpt/rptb` block is fixed.

17. The issue of in between block dependency and the dependencies with the after first instruction of the second block is fixed.

18. The `notouch` part is fixed, so it can be printed to the output. However, it still doesn't support branches into `notouch` section. The `notouch` can't be inside a `rptb` block.

19. The three `NOP` delays after a branch with `bf` and a read from `pcptr` are fixed.

20. The support of `cx_mode` is increased to more than just move immediate (`movi`). Now it supports more immediate instructions such as some immediate `add` and `sub` instructions, and also all immediate move and logical expressions. Also immediate `mult` and `macc` has been added to the supported list. Also instead of terminating the program scheduler prints a comment mentioning that it assumes extreme mode in the unsupported case.

21. The inline `NOP` removal in `notouch` section is fixed.

22. The `NOP` instruction removal from the code is added since existing `NOP` instructions caused issues after the label.

23. The bug of not being able to have `NOP` instruction in `notouch` block is fixed.

24. The issue of not being able to have `rpt/rptb` in notouch section is fixed. A complete `rpt` or `rptb` block in notouch section can be added (`notouch` section still can't exist as a part of a `rptb` block)

25. The dependencies and `NOP`s from previous block to a `notouch` block is fixed according to the `NOP`s given in the `notouch` code.

26. The capability of having a branch in `notouch` block is added.

27. The issue of having a `notouch` block at the end of a processor before the `end` is fixed.

28. The `ibuf` and `ibufnap` dependencies is improved and optimized from `ibuf` before `ibufnap`s to other `ibufnap`s and from `ibufnap`s to the next `ibuf`.

29. The issue of requiring a space between the `rptb` and the comments following this instruction is now fixed. The following example is possible on scheduler.
    Example:

    ```
    rptb 3 #3//can have comments like this.
    ```

30. The scheduler can check the 10th bit of a mask value even when the value is represented in hexadecimal or binary in addition to the decimal numbers.

31. A flag to change the dependencies between `ag/aptr` and `DMEM` on or off using -na flag is added.

32. The hexadecimal and binary numbers can be used in `rptb` as immediate block size values in addition to decimal values.

33. The capability of having `DMEM/DCMEM` as a `rptb` counter value is added.

34. The use of `regbp1` for `DMEM & DCMEM` is added

35. The use of `regbp2` for `DMEM & DCMEM` is added

36. The bug with `DCMEM` dependencies between all `DCMEM`s with the same `ag` is fixed. The scheduler would have inserted `NOP3` between `DCMEM 2` and `DCMEM 3`, but now it is fixed for within basic blocks and across basic blocks.

37. The capability of detecting existing `regbp1` & `regbp2` and make the right decisions accordingly is added.

38. The `NOP` dependencies between changing `pcptr` value and `br pcptr` is fixed. The following example shows this condition.
    Example:

    ```
    move pcptr dmem 0 // requires 3 nops
    br pcptr
    ```

39. The `regbp3` capability for `min/max` is added

40. The ability of changing the first destination to `null` when both instructions in `regbp1` equal or the first and the 3rd instructin dest in `regbp2` equal is added.

41. The issue of `NOP`s dependencies across 2 blocks is fixed.

42. The capability of putting `null` if the destination of the first instruction of `regbp 1,2` is not used in the future is added.

43. The `#output` dependency checker is added.

44. A recognition and removal of multi line (`/* */`) comments and converting them to single line (`//`) comments is added.

45. The capability of turning the user comment on and off using -nc flag is added.

46. The cases of `regbp` and conditional instructions such as `cxt` and `cxf` are fixed.

47. The issue of big `rptb` with many `NOP` instructions in them is fixed.

48. The issue of labels without any branch into them is fixed.

49. The `regbp1` recognition across blocks is added.

50. The issue of extra `NOP` between `cxt` and `cxf` instructions is fixed.

51. The `regbp2, 3` recognitions across blocks is added.

52. A warning log file to generation is added to the scheduler.

53. The issues with multiple `//` in comments is fixed.

Glossary

aprog A program written in C and part of AsAP2 programming chain that converts the ASCII machine language files to binary or BRAM innitialization format (COE) as well as grouping all the different files to two specific files called asap and run. Page(s): 13, 61, 72, 73

AsAP Asynchronous Array of simple Processors — A parallel DSP processor consisting of a 2-dimensional mesh array of simple processors operating in independent clock domains. Page(s): 1

AsAP2 The second generation of AsAP chips which also includes a few specific accelerators (FFT, Viterbi, Motion Estimation) and shared memory modules. It has a reconfigurable source synchronous network supporting long-distance interconnects for processors. Per-core DVFS is also supported for dynamic power savings. Page(s): ii, 1–7, 16, 17, 19, 21, 23, 25, 27, 29, 30, 33–36, 39, 42, 43, 45–50, 52–54, 57, 60, 61, 65, 78–84

asap.coe This is the file outputted from the aprog program used to initialize the FPGA board BRAM. This initialization holds the AsAP2 program to be programmed into the AsAP2 chip by the FPGA board. Page(s): 61, 73

AsAp_programmer This is verilog module that is used to send the program packets or run packet to AsAP2 chip using a serial connection. Page(s): 49

AsAp_programming.v This is verilog module that is used to send the program packets or run packet to AsAP2 chip using a serial connection. Page(s): 47

assem.py AsAP2 assember written in python to convert the input assembly to AsAP2 ASCII machine code. Page(s): 61

buffer A driver or a receiver that is used to redue the effects of noise on the signal by regenerate the signal. Page(s): 51, 52, 70

cfg_glue.v The code containing parts of the AsAP2 verilog code that sets the configuration values in AsAP2 chip. Page(s): 49

cfg_unpack.v The code containing parts of the AsAP2 verilog code that unpacks the address and data infromation from their packets. Page(s): 13, 49

computational This is the Picoblade that is not connected to the host computer via PCIe to interfaces the system with the outside of the system. Page(s): 3–5

dynamic_delay_arbiter.v This is verilog module that is used to automatically change the delay value on the input or output signals after the FPGA board is turned on to length match these signals. Page(s): 52

filelist.c This file is a part of aprog program defining functions acting on different input files. Page(s): 73

GALS Globally Asynchronous Locally Synchronous. A design methodology in which major design blocks are synchronous, but interface to other blocks asynchronously. Page(s): 1, 36

in_out_converter This program is used to convert between the ASCII values and binary values. Page(s): 74, 77

input_arbiter.v This is verilog module that is used to hold the dual clock FIFO and its interface for the input signals to the AsAP2 chip. Page(s): 51

input.bin This file contains the binary input values that are being sent to the input of AsAP2 chip. Page(s): 77

input_buffers.v This is verilog module that is used to have all the buffers and delay blocks related to the inputs to the AsAP2 chip signals. Page(s): 51

Macroblade Highest level of hierarchy in our designed optical networks for enterpise computing that has many Miniblades. Page(s): 2, 3

main.c This file is the top level of the aprog program. Page(s): 73

makecoe.sh This is a script file that arranges the parameters to the aprog to generate the asap.coe file. Page(s): 73

Microblade Lowest hierarchy level of the design below Miniblades that contains many Picoblades. Page(s): 2, 3

Miniblade The next level of hierarchy below Macoblade that includes many Microblades. Page(s): 2–4

nop_counter This program is part of the optimization chain that is used to output some statistics about the assembly or assembler input file. Page(s): 71, 72

null_remover Part of the AsAP2 tool chain that optimizes the AsAP2 assembly code by register renaming and extra DMEM removal for the AsAP2 simulator. Page(s): 61

output_arbiter.v This is verilog module that is used to hold the dual clock FIFO and its interface for the output signals from the AsAP2 chip. Page(s): 52

output_buffers.v This is verilog module that is used to have all the buffers and delay blocks related to the outputs from the AsAP2 chip signals. Page(s): 52

pairlist.c This file is a part of aprog program defining functions acting on the value pairs. Page(s): 73

parse.c A file written in C as a part of aprog that parses the the input files for the aprog program. Page(s): 13

Picoblade The unit size of the optical networks that contains an AsAP2 chip, optical modules, and Memory modules. Page(s): 2–5, 7, 34, 42, 45, 60, 61, 76

progin This is a binary file holding the AsAP2 programming and run instruction to be accessing the BRAM on the FPGA board. Page(s): 76

run.coe This is the file outputted from the aprog program used to initialize the FPGA board BRAM. This initialization holds the AsAP2 run command to be sent to the AsAP2 chip by the FPGA board to start the program on AsAP2. Page(s): 61, 73

spi_slave.v The code containing parts of the AsAP2 verilog code that is used to recieve the serial programming information and convert them to parallel data. Page(s): 12, 13, 49

streamread This is a C program used to receive 32 or 8 bit binary outputs from the FPGA board to the Host system using the PCIe connection. Page(s): 76

streamwrite This is a C program used to send 32 or 8 bit binary inputs to the FPGA board using the PCIe connection. Page(s): 75, 76

system interface This is the Picoblade that is connected to the host computer via PCIe to interfaces the system with the outside of the system. Page(s): 3–5, 7, 34, 42, 45, 60, 61, 76

TDI Test Data In — This is a JTAG signal goes to the FPGA boad containing the serial bit stream. Page(s): 27, 28, 43, 44

TDO Test Data Out — This is a JTAG signal returning from the FPGA board that can be used for chaining purposes. Page(s): 27, 28, 43, 44

temporary _arbiter.v This is verilog module that is used to temporarily connect the PCIe to the AsAP2 input and output data before adding other modules to the AsAP2 interface. Page(s): 54

testboard1 The original daughter card designed by the previous VCL students that was used for testing the functionality of AsAP2 chip. Page(s): 19–21

testboard2 The new designed daughter card that was fabricated after three revisions specified by v1, v2, and v3. Page(s): 17–19, 22–24, 27, 29, 39, 40

top.v The highest level of hierarchy of the verilog file programming the FPGA boad. Page(s): 47

xillybus_read_32 The file used to read from the 32 bit Xillybus PCIe FPGA FIFO. Page(s): 59

xillybus_read_8 The file used to read from the 8 bit Xillybus PCIe FPGA FIFO. Page(s): 59

xillybus_write_32 The file used to write to the 32 bit Xillybus PCIe FPGA FIFO. Page(s): 59

xillybus_write_8 The file used to write to the 8 bit Xillybus PCIe FPGA FIFO. Page(s): 59

Bibliography

[1] Bevan Baas, Zhiyi Yu, Michael Meeuwsen, Omar Sattari, Ryan Apperson, Eric Work, Jeremy Webb, Michael Lai, Daniel Gurman, Chi Chen, Jason Cheung, and Tinoosh Mohsenin. Hardware and applications of AsAP: An asynchronous array of simple processors. In *IEEE HotChips Symposium on High-Performance Chips*, August 2006.

[2] Aaron Stillmaker, Lucas Stillmaker, and Bevan Baas. Fine-grained energy-efficient sorting on a many-core processor array. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 652 –659, dec. 2012.

[3] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, T. Mohsenin, M. Singh, and B. Baas. An asynchronous array of simple processors for DSP applications. In *IEEE International Solid-State Circuits Conference (ISSCC)*, volume 49, pages 428–429, 663, February 2006.

[4] Z. Yu and B. M. Baas. Implementing tile-based chip multiprocessors with GALS clocking styles. In *IEEE International Conference of Computer Design (ICCD)*, October 2006.

[5] Z. Yu and B. M. Baas. Low-area interconnect architecture for chip multiprocessors. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2857–2860, May 2008.

[6] Bevan Baas, Zhiyi Yu, Michael Meeuwsen, Omar Sattari, Ryan Apperson, Eric Work, Jeremy Webb, Michael Lai, Tinoosh Mohsenin, Dean Truong, and Jason Cheung. AsAP: A fine-grain multi-core platform for DSP applications. *IEEE Micro*, 27(2):34–45, March 2007.

[7] Zhiyi Yu, Michael Meeuwsen, Ryan Apperson, Omar Sattari, Michael Lai, Jeremy Webb, Eric Work, Dean Truong, Tinoosh Mohsenin, and Bevan Baas. AsAP: An asynchronous array of simple processors. *IEEE Journal of Solid-State Circuits (JSSC)*, 43(3):695–705, March 2008.

[8] Zhiyi Yu, Michael Meeuwsen, Ryan Apperson, Omar Sattari, Michael Lai, Jeremy Webb, Eric Work, Tinoosh Mohsenin, and Bevan Baas. Architecture and evaluation of an asynchronous array of simple processors. *Journal of VLSI Signal Processing Systems*, 53(3):243–259, March 2008.

[9] A.T. Tran, D.N. Truong, and B.M. Baas. A GALS many-core heterogeneous DSP platform with source-synchronous on-chip interconnection network. In *Networks-on-Chip, 2009. NoCS 2009. 3rd ACM/IEEE International Symposium on*, pages 214–223, May. 2009.

[10] D. N. Truong, W. H. Cheng, T. Mohsenin, Z. Yu, A. T. Jacobson, G. Landge, M. J. Meeuwsen, A. T. Tran, Z. Xiao, E. W. Work, J. W. Webb, P. Mejia, and B. M. Baas. A 167-processor computational platform in 65 nm CMOS. *IEEE Journal of Solid-State Circuits (JSSC)*, 44(4):1130–1144, April 2009.

[11] Z. Yu and B. M. Baas. A low-area multi-link interconnect architecture for GALS chip multiprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(5):750–762, May 2010.

[12] A. T. Tran, D. N. Truong, and B. M. Baas. A reconfigurable source-synchronous on-chip network for GALS many-core platforms. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(6):897–910, Jun. 2010.

[13] A. T. Tran and B. M. Baas. RoShaQ: High-performance on-chip router with shared queues. In *IEEE International Conference on Computer Design (ICCD)*, pages 232–238, October 2011.

[14] Z. Yu and B. Baas. Performance and power analysis of globally asynchronous locally synchronous multi-processor systems. In *IEEE Computer Society Annual Symposium on VLSI*, March 2006.

[15] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, T. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, P. Mejia, A. Tran, J. Webb, E. Work, Z. Xiao, and B. Baas. A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling. In *Symposium on VLSI Circuits*, pages 22–23, June 2008.

[16] A.T. Tran, D.N. Truong, and B.M. Baas. A low-cost high-speed source-synchronous interconnection technique for GALS chip multiprocessors. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pages 996–999, May. 2009.

[17] Z. Yu and B. M. Baas. High performance, energy efficiency, and scalability with GALS chip multiprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(1):66–79, January 2009.

[18] Aaron Stillmaker, Lucas Stillmaker, Brent Bohnenstiehl, and Bevan Baas. Energy-efficient sorting on a many-core platform. In *Technology and Talent for the 21st Century (TECHCON 2013)*, sep. 2013.

[19] Bin Liu and Bevan M. Baas. A high-performance area-efficient AES cipher on a many-core platform. In *IEEE Asilomar Conference on Signals, Systems and Computers*, November 2011.

[20] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, T. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, P. Mejia, A. Tran, J. Webb, E. Work, Z. Xiao, and B. Baas. A 167-processor computational array for highly-efficient DSP and embedded application processing. In *IEEE HotChips Symposium on High-Performance Chips*, August 2008.

[21] Z. Xiao and B. Baas. A 1080p H.264/AVC baseline residual encoder for a fine-grained many-core system. *IEEE Transactions of Circuits and Systems for Video Technology*, 21(7):890–902, July 2011.

[22] Z. Xiao and B. M. Baas. A high-performance parallel CAVLC encoder on a fine-grained many-core system. In *IEEE International Conference of Computer Design (ICCD)*, October 2008.

[23] D. N. Truong and B. M. Baas. Massively parallel processor array for mid-/back-end ultrasound signal processing. In *2010 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pages 274–277, November 2010.

[24] Anh Tran, Dean Truong, and Bevan Baas. A complete full-rate 802.11a baseband reciever implemented on an array of programmable processors. In *Asilomar Conference on Signals, Systems and Computers*, October 2008.

[25] A.T. Jacobson, D.N. Truong, and B.M. Baas. The design of a reconfigurable continuous-flow mixed-radix FFT processor. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pages 1133–1136, May. 2009.

[26] W. H. Cheng and B. M. Baas. Dynamic voltage and frequency scaling circuits with two supply voltages. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1236–1239, May 2008.

[27] Z. Xiao, S. Le, and B. M. Baas. A fine-grained parallel implementation of a h.264/avc encoder on a 167-processor computational platform. In *IEEE Asilomar Conference on Signals, Systems and Computers*, November 2011.

[28] Bin Liu and Bevan M. Baas. Parallel AES encryption engines for many-core processor arrays. *Computers, IEEE Transactions on*, 62(3):536–547, march 2013.

[29] R. W. Apperson, Z. Yu, M. J. Meeuwsen, T. Mohsenin, and B. M. Baas. A scalable dual-clock FIFO for data transfers between arbitrary and haltable clock domains. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(10):1125–1134, October 2007.

[30] R. Proietti, Z. Cao, Y. Li, and S. J. Yoo. Scalable and distributed optical interconnect architecture based on awgr for hpc and data centers. In *Optical Fiber Communication Conference (pp. Th2A-59). Optical Society of America*, March 2014.

[31] Xillybus Ltd. *Xilinx Virtex-7 FPGA VC709 Connectivity Kit.*

[32] Vlsi computation lab. http://www.ece.ucdavis.edu/vcl/.

[33] David A. Patterson and ohn L. Hennessy. *Computer Organization and Design.* Morgan Kaufmann Publishers, 30 Corporate Drive, Suite 400, Burlington, MA 01803, 3rd edition, 2007.

[34] Green circuits inc. http://greencircuits.net/.

[35] Xilinx. *VC709 Evaluation Board for the Virtex-7 FPGA User Guide*, v1.3 edition, April 2014.

[36] Xilinx. *VC709 EVALUATION PLATFORM HW-V7-VC709 (XC7VX690T-FFG1761*, September 2012.

[37] Texas Instruments. *1.8-V High-Speed Differential Line Receiver*, July 2011.

[38] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital Intergrated Circuits – A Design Perspective.* Prentice-Hall, New Jersey, NJ, second edition, 2003.

[39] Xilinx. *7 Series FPGAs Clocking Resources - User Guide*, v1.9 edition, April 2014.

[40] X2Y Attenuators, LLC. *Get the Most from X2Y Capacitors with Proper Attachment Techniques*, v1 edition, february 2006. 3008.

[41] Xilinx. *7 Series FPGAs PCB Design and Pin Planning Guide*, v1.8 edition, September 2013.

[42] Mentor graphics. http://www.mentor.com.

[43] Xillybus ip cores and design services. http://xillybus.com/.

[44] Xillybus Ltd. *Getting started with the FPGA demo bundle for Xilinx*, version 2.3 edition.

[45] Xillybus Ltd. *Getting started with Xillybus on a Linux host*, version 2.1 edition.

[46] Xillybus Ltd. *Xillybus host application programming guide for Linux*, version 2.0 edition.

[47] Hanh-Phuc Le, John Crossley, Seth R. Sanders, and Elad Alon. A sub-ns response fully integrated battery-connected switched-capacitor voltage regulator delivering 0.19W/mm2 at 73% efficiency. In *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 372–373. IEEE, February 2013.

[48] M.S. Makowski and D. Maksimovic. Performance limits of switched-capacitor DC-DC converters. In *Proceedings of PESC '95 - Power Electronics Specialist Conference*, volume 2, pages 1215–1221. IEEE, 1995.

[49] Leland Chang, Robert K. Montoye, Brian L. Ji, Alan J. Weger, Kevin G. Stawiasz, and Robert H. Dennard. A fully-integrated switched-capacitor 2?1 voltage converter with regulation capability and 90% efficiency at 2.3A/mm¡sup¿2¡/sup¿. In *2010 Symposium on VLSI Circuits*, pages 55–56. IEEE, June 2010.

[50] Hanh-Phuc Le, Michael Seeman, Seth R. Sanders, Visvesh Sathe, Samuel Naffziger, and Elad Alon. A 32nm fully integrated reconfigurable switched-capacitor DC-DC converter delivering 0.55W/mm¡sup¿2¡/sup¿ at 81% efficiency. In *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*, pages 210–211. IEEE, February 2010.

[51] Selcuk Kose, Eby G. Friedman, Simon Tarn, Sally Pinzon, and Bruce McDermott. An area efficient on-chip hybrid voltage regulator. In *Thirteenth International Symposium on Quality Electronic Design (ISQED)*, pages 398–403. IEEE, March 2012.

[52] GE Critical Power. *Austin Microlynx II DC-DC Converter*.

[53] Energizer Holdings, Inc. *Energizer E91 AA Datasheet*.

[54] Energizer Holdings, Inc. *Energizer E92 AAA Datasheet*.