Lab Manual

Laboratory

Synthesis of Digital Systems

Prof. Dr.-Ing. Ulf Schlichtmann

April 8, 2014

Institute for Electronic Design Automation
Technische Universität München
Arcisstr. 21, 80333 München
Tel.: (089) 289 23666

Synthesis of Digital Systems - Laboratory

Institute for Electronic Design Automation
Technische Universität München

2013-2014 – Munish Jassi

Contents

List of Figures

1. Checklist for Completion of the Laboratory

1. Evaluation

   - Report: This laboratory consists of four parts. Each lab has a set of questions in the last *Questionnaire* section. At the end of each lab, you must submit a report with the answers to those questions, according to respective deadlines.
   - Project code: Please insert your name and account as comments at the beginning of the source files. For submission, you must zip the lab assignment and upload it on the server. Only the code inside these directories will be evaluated. For evaluation, you must guarantee that the submitted project code can be directly imported into the Xilinx tools and tested.
   - Written exam: 90 min. 1/3 of the points for the final exams will be related to Lab content.

2. Grade

   This laboratory makes 50% of the Synthesis of Digital Systems (SDS) course. 25% of the final grade will be on code submission and reports, 1/2 part for code and other 1/2 part for the reports. Final written exam will be 75% of the grade, 1/3 of the points will be related to Lab content.

3. The working environment and tools

   For login at the institute the user name is `your_lrz_account` (e.g. `ne42zup`) while the password is the same as that of the mytum email box. For remote login, the server domain name is `sdsX.regent.e-technik.tu-muenchen.de` where `X` is a number between `1` and `9`. Eventually you need to install the vpn client as described here: http://www.lrz.de/services/netz/mobil/vpn/. If you use `ssh`, the full command is:

   ```
   ssh -X your_lrz_account@sdsX.regent.e-technik.tu-muenchen.de
   ```

   The lab working directory is:

   ```
   /usr/local/labs/SDS/current/your_lrz_account
   ```

   Code Submission must be present in respective directories (LabA, LabB, LabC, and LabD) under,

   ```
   /usr/local/labs/SDS/current/<your_lrz_account>/Lab<X>
   ```

4. Materials

   - Link to information on the ZedBoard: `http://www.zedboard.com`.
   - Schedule, lab manual, project material and lecture notes are or will be available on Moodle.

5. Contact

- In case you encounter problems while doing the laboratory, please write an email (always with your lrz login on title) to the following address:

  eda-sds-support@lists.lrz-muenchen.de

  Please attach the relevant files that your question refers to. Always use this email as the first option in case that you want to get an answer to the problem that you can not solve after long-time thinking by yourself.

- Or you can come to the tutor hour on Wednesday, 16:45-18:15, in room 2909.

  Tutor: Benjamin Bodes
  benjamin.bodes@tum.de

- Or contact the lab supervisor:

  M.Sc. Munish Jassi
  Room 2909
  Tel.: (089) - 289 23651
  E-Mail: munish.jassi@tum.de

2. Hello World

This is bootstrap to the Xilinx tool chain for ZedBoard FPGA programming, while executing already build "Hello World" project on ZedBoard.

Steps:

1. Copy *HelloWorld* project from <project docs>/HelloWorld to <work dir>/. (*$cp -r <project docs>/HelloWorld <work dir>/.*)
2. Load enviroment settings on terminal (*$module load xilinx/ise/14.7*).
3. Start PlanAhead, *$planAhead* and open *HelloWorld* project by pointing it to *HelloWorld.ppr* inside <HelloWorld>directory. This will open *PlanAhead*
4. Double-click *system_i - system (system.xmp)* in the *Project Manager* to open *XPS*. Play around with different panels on *XPS* and close *XPS*
5. Connect *ZedBoard's micro USB ports J14 and J17* to *desktop*. Turn On the power switch (SW8).
6. Open new terminal, and start *minicom* as *$minicom -D /dev/ttyACM0*.
7. Select *File>Export >Export Hardware for SDK...* and check both *Export Hardware and Launch SDK* to start SDK. Play around with different panels on SDK.
8. On SDK window, click *Program FPGA* or select *Xilinx Tools>Program FPGA* to program hardware on FPGA.
9. On SDK window, click *Run* or select *Run>Run* to program software on FPGA.
10. After running the software "Hello World" message will appear on the *minicom terminal*.

3. Introduction

3.1. Overview

According to the ITRS 2011 [1], System-on-Chip (SoC) design involves 54% reuse of Intellectual Property (IP) blocks in the current generation of chips and it is estimated to reach 90% by 2020. This is a major increase in reuse of IP blocks that is expected for the next generations of SoCs.

In the typical design methodology scenario, SoC design starts with compiling requirements of a given application which includes operating conditions, expected inputs and outputs, target tasks to achieve, type and amount of data processing and available technology related information. Once the requirements are specified, these are translated to system specifications parameterized by chip area, power dissipation, performances (clock frequencies, and latencies) etc.

Designing the system architecture is the most critical step of the chip design. Decisions on the desired CPU configuration, communication bus protocol, memory types, I/O interface peripherals, hardware acceleration units and inclusion of other hardware (HW) components are taken at very early stages of the design cycle. Any recall of the decisions taken at this stage, after the design implementation, can cause serious financial losses and missing product time lines. Various EDA tools are used to estimate the system parameters for the desired final product. These tools either do extensive power and performance simulations at clock-cycle level accuracy on the system architecture models, which further use Instruction Set Simulator(ISS) for the CPU models, or they estimate the parameters by extrapolating the numbers available from various individual hardware Intellectual Property (IP) modules used in the design. At this stage of the flow, different system architectures are taken into consideration. These architectures differ their usage of set of hardware components, and/or bus systems for data communication.

Application profiling is an important step which guide the system designer to decide about the inclusion of certain hardware accelerators (HA) in the architecture. When the desired application is compiled and executed on the CPU, some specific sub-tasks of the application can take up a considerable portion of the CPU processing. It can occupy a significant CPU bandwidth and require excessive processing power. These sub-tasks are generally some repetitive operations, data transfers from I/O or memory, data filtering operations, computationally intensive operations etc. It is beneficial to execute these sub-tasks on dedicated hardware. Typically a system design contains multiple hardware accelerator IP blocks, dedicated for specific sub-tasks of the desired application. HAs are the hardware implementation of the software (SW) sub-tasks. With HAs in the design, CPU can offload computationally intensive tasks to HA, and CPU itself can process the rest of the application. Data intensive applications, like video processing, deal extensively with the repeated processing of input data. It is a usual practice to have dedicated hardware modules for data transfers and/or data processing. CPU controls the HA by writing to its control registers which control the behavior of the respective HA. Data flow synchronization is maintained by interrupts from HA and interrupt service routines (ISR) executed by CPU. To benefit from HAs in the system, SW application execution must transfer the control seamlessly between CPU and HAs. If the system runs an Operating Systems (OS), this seamless exchange is maintained by the OS itself (resource allocation). SW driver functions make the OS aware of hardware resources of the system. These driver functions give low-level read-write access to the control and data

registers. Driver functions initialize the HAs, allocate the resources for processing data when the HAs are free, and after finishing the processing make them available for other processing requests. In non-OS based systems, drivers are embedded within the application itself. These drivers reset and initialize the HAs at the beginning of execution of application. The operations are generally transferred between CPU and HA using ISR calls via interrupt handlers. While executing ISRs, CPU transfers the control by reading and writing to the control registers of the HA. The control is handed back to the CPU when HA finishes the operations. Figure 3.2 shows the architecture of a typical SoC system.

Data communication among the various components of a system is handled by the bus-system. A bus-system connects various Masters and Slaves of the system. It follows the defined protocol for data transfer, handles the conflict of multiple requests from various Masters, and allows the Master to the access to the Slave peripherals and Slave registers. It is common to have multiple bus-systems in a SoC. CPU accesses fast peripherals and memories via the fast bus-system, while slower peripherals and I/O interfaces are attached to the slower bus-system. Multiple bus-systems help to lower the overall power consumption of SoCs.

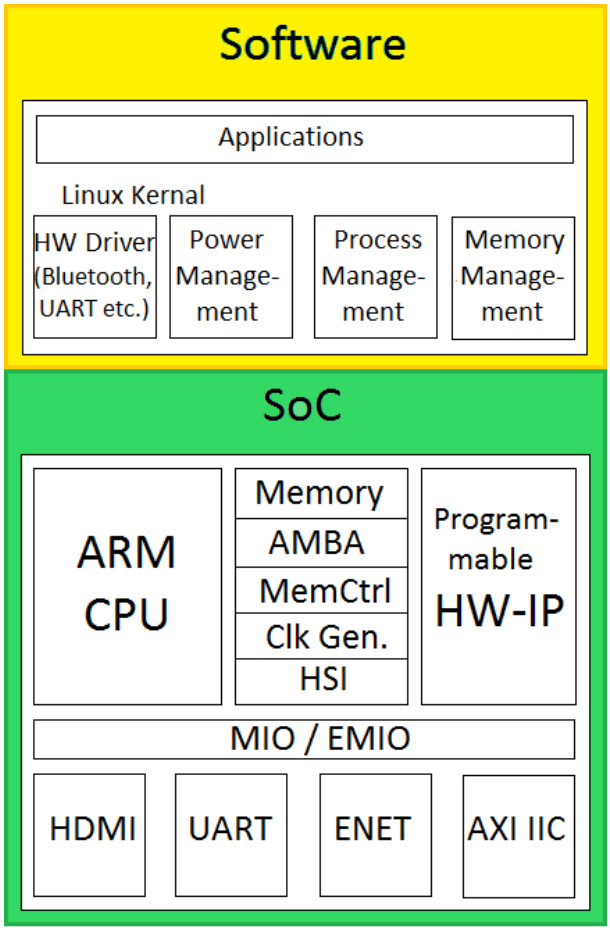

Figure 3.1.: Architecture of typical SoC System, including HW and OS stack

3.2. Objectives of this Laboratory

The objective of this laboratory is to introduce you to the essential aspects of High Level Synthesis (HLS) and FPGA prototyping. The First part of the laboratory will introduce you to FPGA and get you started on working with ZedBoard FPGA kit from Digilent. We will use the software tool chain from Xilinx to implement HW and SW projects and finally burn it on FPGA for system prototyping. Here we will configure the Programmable System (PS) part of the zynq7000 FPGA chip to initialize the build-in peripherals, clocks, memory controller and add additional hardware to the Programmable Logic (PL) of the FPGA to build an LED controller system. On this hardware we will develop the bare metal software code (no operating system) for the application using hardware driver functions. In the second part of this laboratory we will introduce HLS using the industrial tool chain from Xilinx. Here we will use the Vivado_HLS tool to do HLS of a digital filter and apply various optimizations to achieve the design goals for the digital filter.

The last part of this laboratory will be the implementation of a video processing SoC. In this part we will read video input from a digital camera into the FPGA, do filtering operation on the input video (gray scale conversion), and display it on the monitor. We will design two implementations of this system, first with pure software version (without hardware acceleration) and the second implementation with hardware acceleration of filter operation sub-tasks. Finally, we will compare the performances of these two systems.

At the end of this laboratory you should be able to understand the principles of FPGA system prototyping, high level synthesis methodologies and the associated industrial tool chains.

3.3. High Level Synthesis (HLS)

In the previous section we gave a brief overview of typical SoC systems and associated SW-HW partitioning. IP-based system design leads to the aspect of Model-Based Design (MBD). MBD uses an IP-library consisting of multiple IP modules as input to design the system architectures. These IP modules can either be supplied by the IP-supplier or can be the design knowledge of the design team from previous works. When designing the system architectures, components of IP-library are taken as building blocks. In MBD, low level design description is abstracted to high level abstraction. This reduces the system complexity, and iterations related to system architectural changes can be quickly incorporated into the design.

In HW-SW co-design, certain software tasks of an application are implemented by HW. These tasks are essentially the computational or communicational bottlenecks of the design. Once these bottlenecks are found by doing application profiling, the next step is to convert the corresponding SW description into their equivalent HW implementation. The first straightforward way is to implement the HW equivalent manually from the functional description available as SW. But the manual implementation is time consuming and error prone, requires HW implementation expertise, additional effort for HW verification and documentation. Another disadvantage is that excessive time and effort for implementation mean system designers can only do limited number of iterations over multiple HW implementations.

High Level Synthesis (HLS) is a powerful way of translating SW function description to its HW implementation. Automation of SW description to HW helps overcome the disadvantages of manual implementation. HLS based HW generation probably doesn't give the best performances compared to manual implementation. But current state of the art HLS tools give the designer enough tuning modes to allow multiple optimizations and generate sub-optimal implementations

which could be taken through final manual optimizations. HLS tools give a faster way to generate new HW IPs and integrate them to iterate over multiple design architectures.

In this laboratory, we will be working with the industrial HLS tool from Xilinx (Vivado_HLS) which translates the C++ equivalent description of a SW function to a synthesizable HW for FPGA. There are inbuilt limitations of C++ which prevents the synthesizability of pure C++ code (eg. dynamic memory allocation, floating data types, recursive functions, etc.). There are many data types and concepts in C++ which prevent synthesizability. Xilinx provides a set of special data types (in the form of libraries), which makes the SW descriptions synthesizable. Once the SW functions are translated to HDL description (VHDL/Verilog), those are added to the IP-library of the HW components of Vivado/PlanAhead (Xilinx tool for SoC prototyping). XPS (Xilinx Platform Studio) is a tool which is used to configure the CPU cores and the peripheral HW components to design the complete system. All the components of the system are integrated in XPS. This system description is then exported into SDK (Xilinx Software Development Kit), where the software application can be developed for the given hardware. This hardware and software is finally exported to the FPGA using SDK (or iMPACT) for system prototyping.

Questions:

1. Describe briefly about Programmable System (PS) and Programmable Logic (PL) for Zynq chipset.
2. What is High Level Synthesis (HLS) and Model-Based Design (MBD)? What are their advantages?
3. What is the advantage of using hardware accelerators (HW) in SoC designing?
4. What is the key difference between application development for operating system and without operating system?
5. What are the inbuild limitations of C++ which prevents its hardware synthesizability?

3.4. ZedBoard and Xilinx Tool Chain

ZedBoard is a FPGA development board from Digilent Technologies. ZedBoard is centered around the zynq7z020 series of the Zynq FPGA from Xilinx. Zynq is Xilinx's latest series of FPGA chipsets which consist of powerful processing power of ARM dual core processors (ARM Cortex-A9) and programmable logic matrix equivalent to 1.3M (for z7020) to 6.6M (for z7100) instances. The key features provided by ZedBoard are listed below:

- Processor
  - ZynqT M -7000 AP SoC XC7Z020-CLG484-1
- Memory
  - 512 MB DDR3
  - 256 Mb Quad-SPI Flash
  - 4 GB SD card
- Communication
  - Onboard USB-JTAG Programming

  - 10/100/1000 Mbps Ethernet
  - USB OTG 2.0 and USB-UART
- Expansion connectors
  - FMC-LPC connector (68 single-ended or 34 differential I/Os)
  - 5 Pmod compatible headers (2x6)
- Clocking
  - 33.33333 MHz clock source for PS
  - 100 MHz oscillator for PL
- Display
  - HDMI output supporting 1080p60 with 16-bit
  - VGA output (12-bit resolution color)
  - 128x32 OLED display
- General Purpose I/O
  - 8 user LEDs
  - 7 push buttons
  - 8 DIP switches

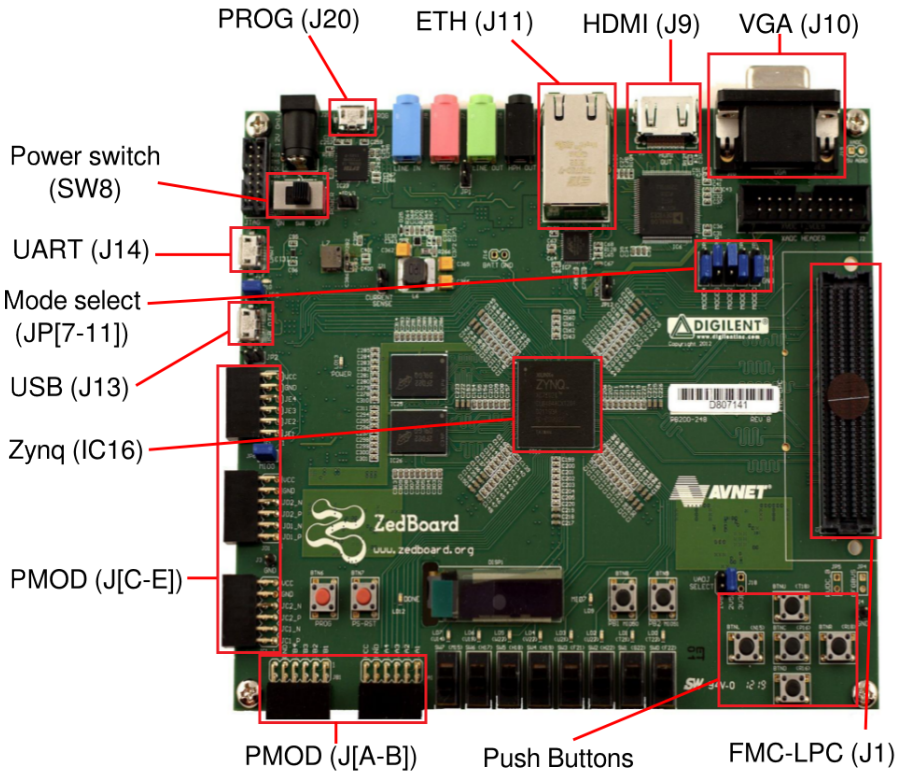

Figure 3.2.: ZedBoard - Zynq Evaluation and Development board

ZedBoard works with the Xilinx's toolchain, which basically contains three SW tools:

- Vivado/PlanAhead,
- XPS (Xilinx Platform Studio),
- SDK (Software Development Kit).

Vivado/PlanAhead is the central tool for creating and managing the project by including the required IP components and constraints. XPS can be invoked internally from Vivado/PlanAhead or independently. XPS is for programming and configuring Processing System (PS) CPU cores and instantiating additional hardware in the Programmable Logic (PL). Clock generators, interrupts interfaces, memory controller, external peripherals and other hardware related configurations are also done in XPS. Once hardware is defined in XPS, top-level netlist and bitstream for FPGA is generated in Vivado/PlanAhead. The generated bitstream is further exported to SDK, which is used to implement the software to be executed on the generated hardware. SDK is used to develop the C/C++ applications for the hardware and it has a cross-compiler which cross-compiles the software application for ARM processors. It can also be used to compile SW applications for Linux, when the system is to run Linux operating system, and applications are to be run on top of OS. The cross-compiled SW with hardware bitstream is burnt on FPGA using SDK itself or iMPACT tool. This way the ZedBoard is programmed with the desired hardware on the FPGA fabric and the software is put onto the FPGA memory as instructions for execution. ZedBoard, combined with the associated software tool chain from Xilinx, provides us with a powerful kit for system prototyping.

3.5. Vivado-HLS

Vivado-HLS is a high level synthesis tool from Xilinx. It translates the C/C++ based functional description to synthesizable HDL (VHDL/Verilog) code, which can further be synthesized w.r.t. a given technology. Xilinx provides its own library of data types to enable synthesizability of the functions. Vivado-HLS supports multiple directives for design optimization, like hardware pipelining, loop unrolling, various I/O interfaces, memory interfaces, etc.. These directives guide the HDL code generation and tool tries to optimize the performance of the hardware core while obeying those directives. These directives are manually controlled by the designer, who can iteratively add new directives after analyzing the results of a given iteration. The tool provides a debugging environment as well as analysis perspective for analyzing the results of HDL generation in term of area utilization (or hardware resources and LUTs) and performance bottlenecks. We will look into the debugging and analysis perspective in more detail later while working on the labs.

3.6. Video Processing on ZedBoard

Video processing is an application domain of engineering, where video inputs captured by camera modules are processed for the given application requirements. These applications can vary across diverse range, like security applications, face/motion/object detection, image filtering, driver assistance, embedded video processing etc.

ZedBoard provides a powerful platform to do video processing prototyping on FPGA. In addition to Zynq chipset, it contains peripherals like HDMI and VGA drivers, Pmod, UART, and USB serial interfaces. The processing power of ARM cores can be used to run operating system and/or processing video data from the camera inputs. The flexibility of the programmable fabric gives

designer an option to add dedicated HA to improve the performance for the application. Especially for video processing applications, HAs can give big performance improvements because it involves a lot of repetitive processing on image data. Using the available peripherals on ZedBoard it is possible to prototype a real-time video processing system, and Xilinx's tool chain gives a tool set to debug and analyze the application software. Using add-on hardware module (chipscope) available in IP-library from Xilinx, it is also possible to analyze the real-time activities on the system buses and signals.

In this laboratory we will set up a video processing system on ZedBoard and implement a video processing application on it. This video processing application will later be accelerated by using dedicated HA for performing specific computationally intensive filtering tasks.

Questions:

6. What is an FPGA and how is it different from ASIC? What are the building blocks for an FPGA?
7. How can HW especially be helpful for video processing applications?

3.7. Laboratory Tasks

In Lab A, we will implement an application to control the intensity of the LED on ZedBoard. Using the keyboard inputs through UART we will make it turn-on/off, and control the intensity of LEDs.

Tasks:

- Set up project on Xilinx tool chain for Zynq FPGA.
- Configure the ARM Cortex-A9 CPU cores for the system design.
- Configure the clock generator, memory controller and peripherals.
- Add required hardware IPs to the programmable logic for LED control.
- Synthesize the design and generate bitstream for FPGA programming.
- Develop application software for the desired application.

In Lab B, we will implement an Discrete Cosine Transform (DCT) filter function using HLS on its software description. DCT is used in number of applications related to image and signal processing, data compression, and numerical solutions to mathematical methods. Figure 3.3 shows the design flow of the DCT application. At first step, it reads the input data and at the end of DCT it writes the DCT coefficients as outputs. DCT has DCT_2D as sub-function, and DCT_2D has DCT_1D as sub-function. For this lab we will perform optimizations to achieve the design target of performing DCT operations on each set of inputs within every 300 clock cycles. We start with non-optimal HLS and progressively add optimization directives to achieve the design target.

Tasks:

- Set up Vivado-HLS project for DCT operations.
- Generate the default setting HDL implementation of DCT function.
- Analyze the implementation outputs and find the design bottlenecks.

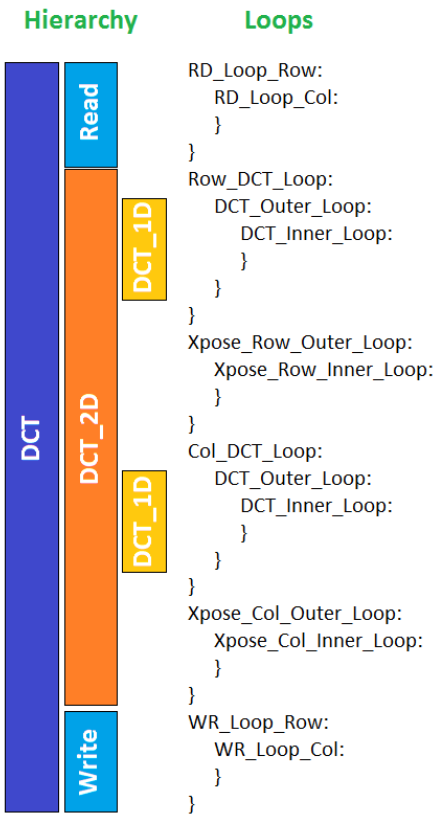

Figure 3.3.: An overview of DCT design flow.

- Progressively add optimization directives to configurations, to optimize for hardware pipelining, loop unrolling, I/O and dataflow optimizations.
- Achieve the performance of the design target with minimum area utilization.
- Package the final IP module as deliverable for Xilinx IP-library.

In Lab C we will implement the video processing platform on ZedBoard. Here we will set up the camera interface to camera device and interface to the display monitor via HDMI port. The camera is connected via Pmod ports to the ZedBoard and the camera interface module reads the data from camera into the FPGA. Received video data will be handled by camera interface module, which is a custom module in PL to receive and buffer the pixel data in accordance to the image format. It converts the input data into word width format, synchronizes the start of frame and start of line using camera input signals and generates the corresponding interrupts for the CPU. CPU writes camera data to the DDR memory, which is further displayed on the display monitor via HDMI port of ZedBoard. On this hardware system we will then implement a simple software application of gray scale conversion of the input video stream.

Tasks:

- Set up the HDMI hardware peripheral and display monitor to display the output video.
- Set up the video input interface to the ZedBoard and the corresponding camera interface modules.
- Hardware system generation for the complete video processing system.

- Implement software application to receive video input and display on monitor.
- Add software-based gray scale filtering operation on input video stream.
- Implement the SW-HW system on ZedBoard.

Last lab Lab D is about designing a complete video processing system. Tasks for this lab will be to do hardware acceleration of the SW application which we implemented in the Lab C. We will generate the new HA following the HLS steps in Lab B, and export a new PCore for Xilinx design enviroment. We will use the set up from the Lab C and add the generated HW IP to the system. We will make appropriate software changes to use the new HW into the application processing. Figure 3.4 shows the block diagram of the system set up.

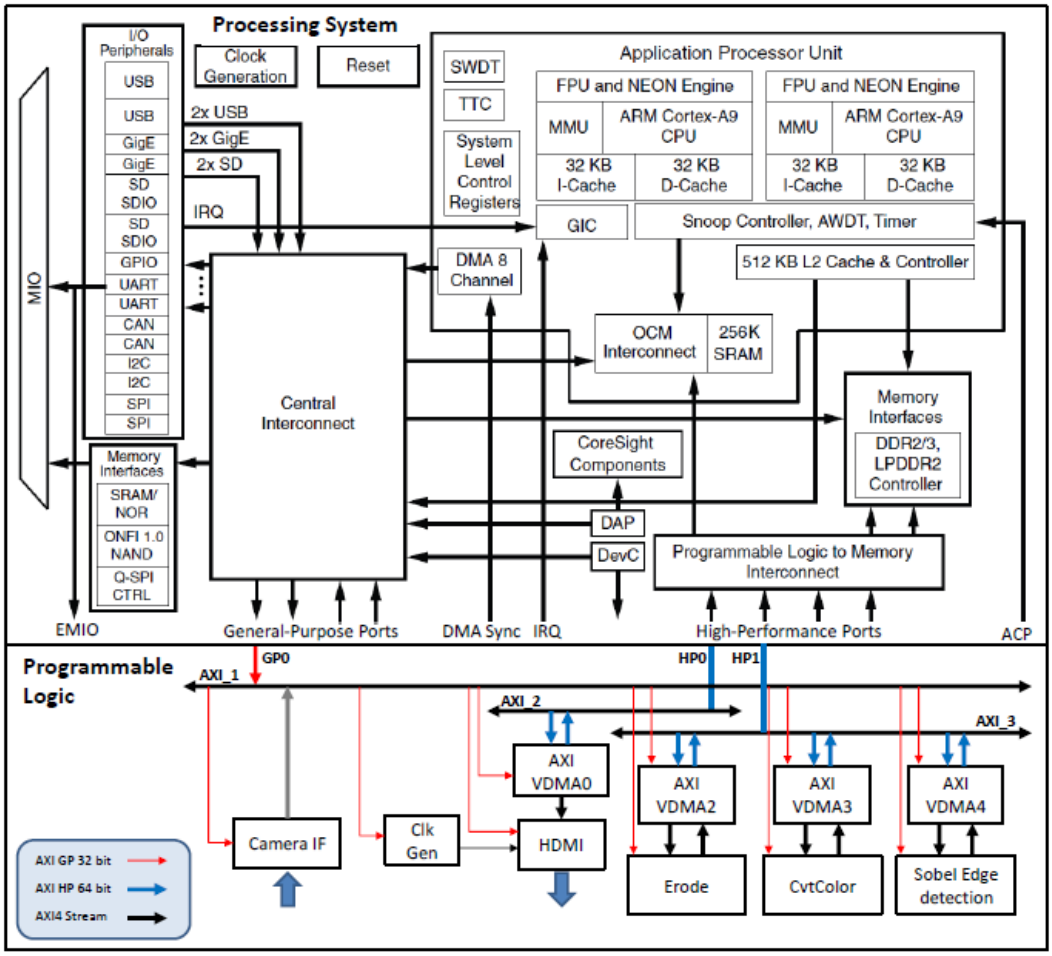

Figure 3.4.: Block diagram of the system set up on Zynq.

Tasks:

- HLS of Gray scale algorithm as hardware accelerator.
- Synthesize the VDMA (Video Direct Memory Access) core for HW for offloading the CPU.
- Configure the VDMA core for frame synchronization and streaming interface.
- Implement the new video processing system, including hardware gray scale filter.
- Modify the existing SW code to include HW for video processing.

- Do application profiling and performance analysis for the hardware accelerated system.

Questions:

8. Give five hardware features for ARM Cortex-A9 used in Zynq chipset.
9. What are the possible interfaces from PS to PL on zynq chipsets?
10. Give a brief description of the building blocks of *Zynq's* Application Processing Unit? (ref. Figure 3.4)

4. Lab A: Getting Started with ZedBoard

In this introduction lab we will learn the principles of FPGA (ZedBoard in our case) and use associated software tools, by following step by step approach to develop a simple LED application. This will require three tool chains to work with: Vivado/PlanAhead, XPS, and SDK. In this laboratory please use only the specified versions of different tools, as the execution scripts and settings are developed specific to the tool versions. All the laboratories use some of the default tool settings, which varies across different versions. **We will use the following versions of the tools:**

- **PlanAhead, XPS and SDK: 14.7 version**
- **Vivado and Vivado_HLS: 2013.3 release**

Please take care of that these builds are quite recent and can have some bugs. During the implementation if you find some unexpected error messages, please save all your files in the session and restart the tool to see if those errors still persist.

We will be using these pointers to the directories in the lab manual,

- **<work dir>** : Working directory provided to you for lab-work.
  `/usr/local/labs/SDS/current/your_lrz_account`
- **<project docs>** : Directory provided with initial set of materials for labs.
  `/usr/local/labs/SDS/current/your_lrz_account/project_documents`
- **<$dir >** : Path to directory *"dir"*.
- **<project name>** : Name of the active session's project.

Here are the steps we will be following:

1. Vivado/PlanAhead environmental set up and creation the project.
2. XPS for adding embedded source.
3. XPS to configure the system peripherals.
4. XPS to configure the memory and clocks for the system.
5. Add User Constraint File (UCF) for PL.
6. Exporting hardware to SDK and test application development.
7. ZedBoard hardware set up and memory/peripheral tests.
8. Adding and setting the hardware for PL.
9. Software application development for LED project.
10. ZedBoard hardware set up and FPGA programming.

Tasks:

4.1. Vivado/PlanAhead Environmental set up and Creation of the Project

1. $mkdir LabA, will create a new working direction in your working area <work dir >.
2. $source /usr/local/tools/xilinx/ise/14.7/ISE_DS/settings64.sh or $module load xilinx/ise/14.7 on Terminal will load the environmental settings for this version. Set these enviromental settings on the terminal everytime before executing any Xilinx tool, if those settings are not already sourced.
3. Set system language to en_US.utf8, *$export LANG en_US.utf8*.
4. $planAhead, command invokes the GUI for PlanAhead.
5. Click *File>New Project*, to create new project.
6. Click *Next* on *Create a New PlanAhead Project* dialog.
7. In the next tab give Project name: *"LED_project"*, keep Project location to the current directory, check *Create project subdirectory* and click *Next*.
8. Next tab, select *RTL Project*, check *Do not specify sources at this time* and click *Next*.
9. Next tab is to specify the working board for the project. To filter down the available boards, use the filters as, Family:*Zynq-7000*, Sub-Family: *Zynq-7000*, Package: *clg484*, Speed grade: *-1*. Then, select *xc7z020clg484-1* and click *Next*.
10. Next tab shows the summary which should look like in Figure 4.1. Click *Finish* to finish create the project.

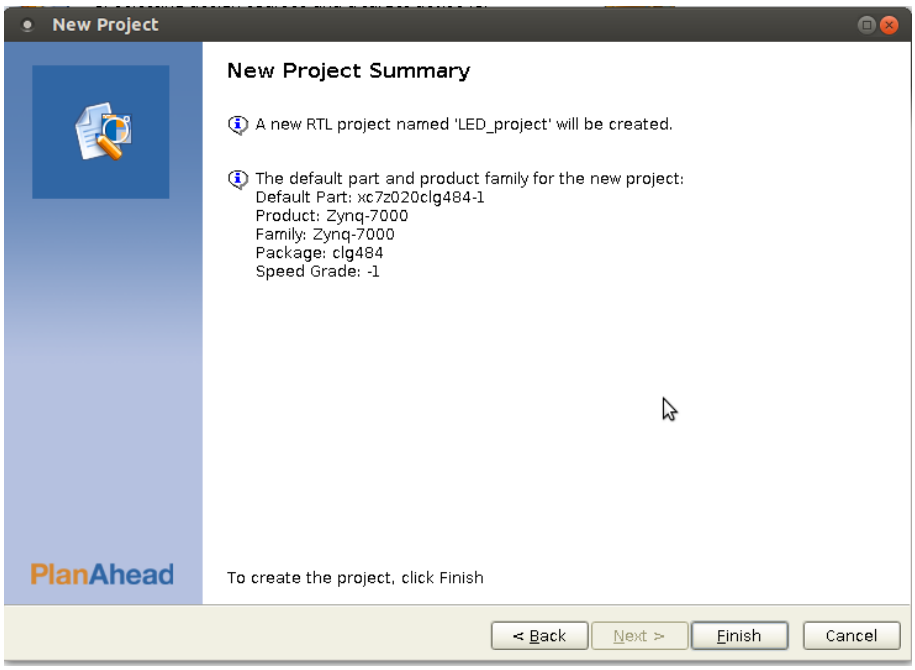

Figure 4.1.: Summary of Project.

Questions:

1. Describe the roles of *PlanAhead, XPS, and SDK tools* in Xilinx based system design flow?

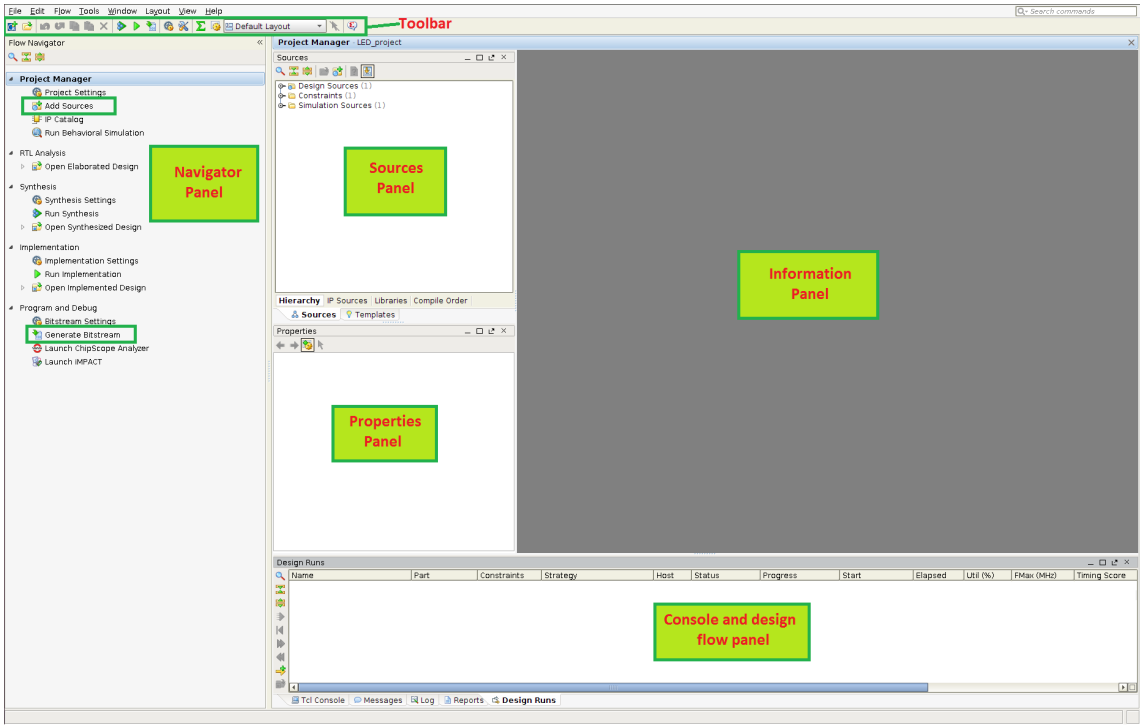

Figure 4.2.: PlanAhead window and its different panels.

4.2. XPS for Adding Embedded Source

1. To add embedded source, in the Navigator panel (on left side) click *Add Sources* (Figure 4.2).
2. In the new window, select *Add or Create Embedded Sources* and click *Next*.
3. In next tab, click *Create Sub-Design....* In the pop-up window rename the *Module name* to *"system"*, and click *OK*. After this you should see a new entry in the table with Id as *'1'* and name as *"system.xmp"* and location as <LabA>/LED_project/LED_project.srcs/sources_1. Directory *LED_project.srcs* is according to Xilinx's project directory structure, contains the information about all the sources associated with the project.
4. Click *Finish*, this will automatically invoke XPS tool. It will ask to start with *"BSB wizard"*, click *No*. XPS will prompt to add Processing System7 instance to the system. Click *No*, because we will add it manually later in this lab. Now you should see the XPS window as shown in Figure 4.3.
5. The IP-Catalog panel (on left side) shows the list of IPs which are available to use for the system design. In the IP-Catalog panel, expand *EDK Install* and further *Processor* to see the list of available Processing IPs. Double click *Processing System version 4.03.a*. Click *Yes* to the pop-up to add the new processing_system7 IP.
6. You should now see new processing_system7 IP in the *System Assembly View*. In this view, there are four tabs available at the top: *Zynq, Bus Interfaces, Ports, Addresses*. The *Zynq* tab shows the configuration of the PS. The gray color on I/O Peripherals means that none of the IO peripherals are configured and are in their default state (Figure 4.4)

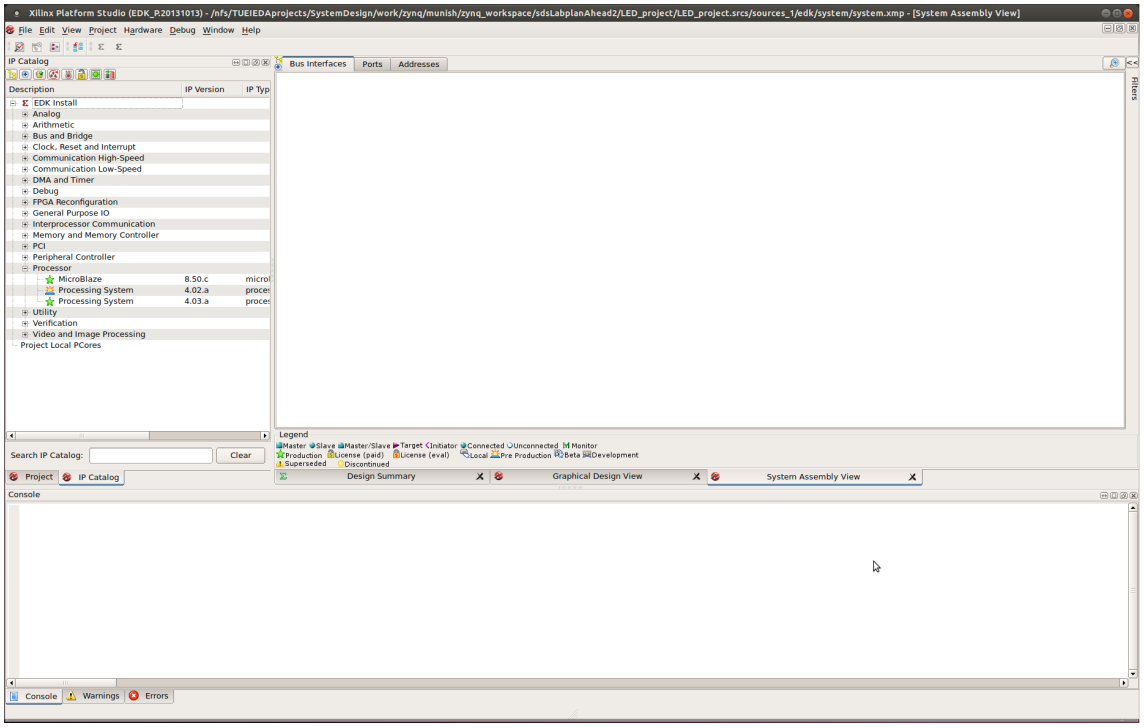

Figure 4.3.: XPS window.

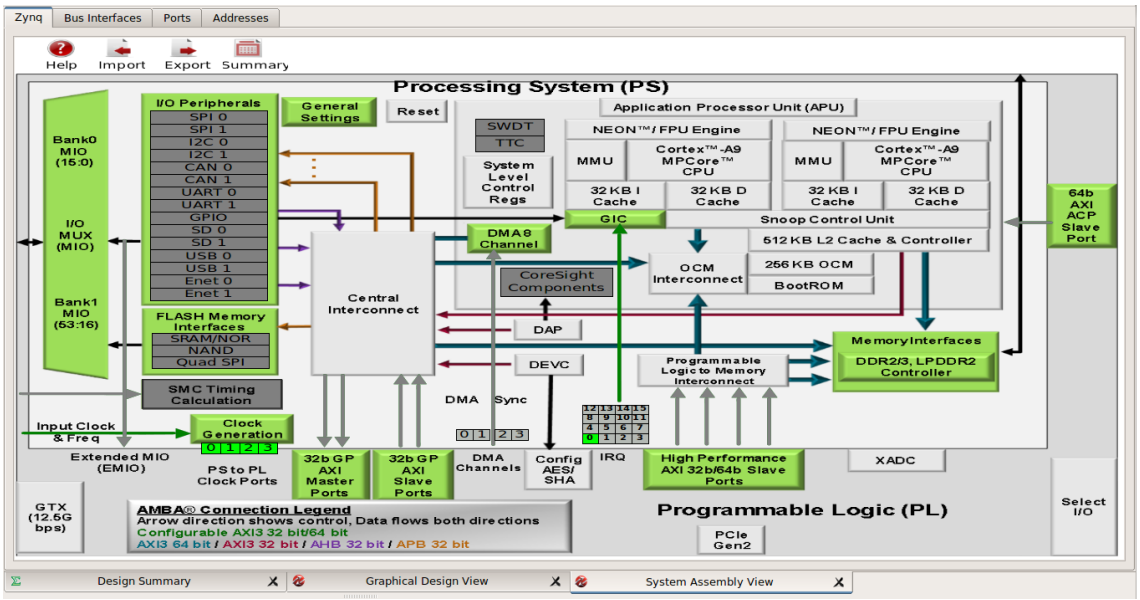

Figure 4.4.: XPS with Zynq tab

4.3. XPS to Configure the System Peripherals

1. Click on the *I/O Peripherals* box in the *zynq* tab. Then, the *Zynq PS MIO Configurations* window will pop-up. Check *"Show I/O Standard Options"*. Set the *MIO Configuration* (on right top): Bank 0 IP Voltage: *LVCMOS 3.3V* and Bank 1 IO Voltage: *LVCMOS 1.8V*.

Click *Yes* to confirm the changes.

2. In the *Zynq PS Configuration panel* (on left side), various desired peripherals can be selected and mapped to different MIO (Multiplexed IO) or EMIO (Extended MIO) ports. Select *UART1* and map its *IO* to MIO 48..49. Expand *UART1* and uncheck *Modem Signals*. Now you will be able to see MIO[48-49] connections to *UART1* in the *MIO Configuration table* (on right side).
3. Select *Quad SPI Flash* and map it to MIO 1..6. Quad SPI from Spansion is Zynq's on-board Flash. Then expand it to verify *Feedback Clk* is selected and mapped to MIO 8.
4. Next select *USB 0* and map it to MIO 28..39.
5. Select *Enet 0* (Ethernet peripheral) and map it to MIO 16..27. Then select *MDIO* under *Enet 0* and map it to MIO 52..53.
6. Next peripheral will be *SD 0*. Select *SD 0* and map it to MIO 40..45, expand it and connect *CD* to MIO 47, *WP* to MIO 46.
7. Till now we have used 43 MIOs and rest 11 we will need for PL (Pmod, LED, Push Buttons (PB)).
8. Finally select *GPIO*, this will connect rest of MIOs to GPIOs. Please note the directions for MIOs for GPIO. Direction for MIO 7 is *Out*, so we will use this to connect to LED. Set the directions for MIO 50-51 to *In*, which we will use to connect PBs. At this point Zynq PS MIO Configurations window should look like Figure 4.5.
9. Click *Close* to exist this window.

Questions:

2. What controls are provided in *XPS System Assembly View* with different tabs: *Zynq, Bus Interfaces, Ports, Addresses*?
3. What are MIOs and EMIOs? What are the differences between them? {ref. to Zynq user manual}
4. Which I/O peripherals are supported by PS?

4.4. XPS to Configure the Memory and Clocks for the System

1. From XPS window, click on *Clock Generation* box to open the *Clock Wizard*. Click "+" icon (on left top) to expand the the clock table.
2. Verify these clock frequencies in the table,
   - Input frequency (MHz) : 33.333333
   - CPU requested frequency (MHz) : 666.666666
   - DDR requested frequency (MHz) : 533.333333
   - QSPI IO clock is enabled and set to 200MHz.
   - ENET0 IO clock is enabled and set to 1000Mbps.
   - SDIO IO clock is enabled and set to 50MHz.
   - UART IO clock is enabled and fixed at 100MHz.

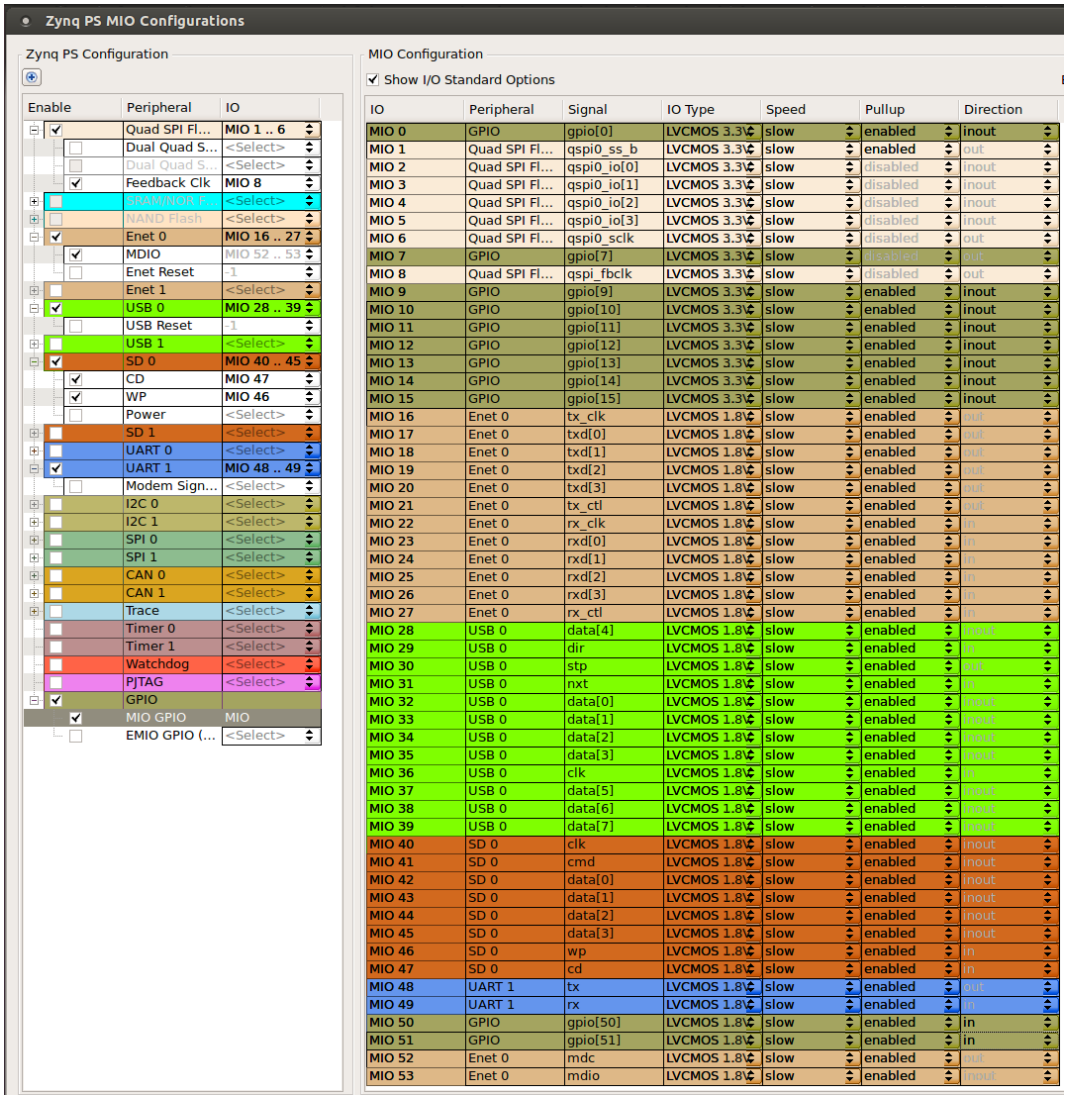

Figure 4.5.: Final Peripheral IO overview

- PS can generate four clock frequencies for PL, which are set in *PL Fabric Clocks* section. Set these frequencies for the PL IO clocks,
  - FCLK_CLK0: 100MHz
  - FCLK_CLK1: 150MHz
  - FCLK_CLK2: 50MHz
  - FCLK_CLK3: 25MHz

3. Now the *Clock Wizard* should look like in Figure 4.6.
4. Click *Validate Clocks*, and then click *OK* to confirm the changes and return to XPS window.
5. Now on XPS window, click *Memory Interfaces box* (on right), and the *PS7 DDR Configuration window* for DDR Controller configuration will pop-up.
6. Check *Enable DDR Controller* on the top left of the window.

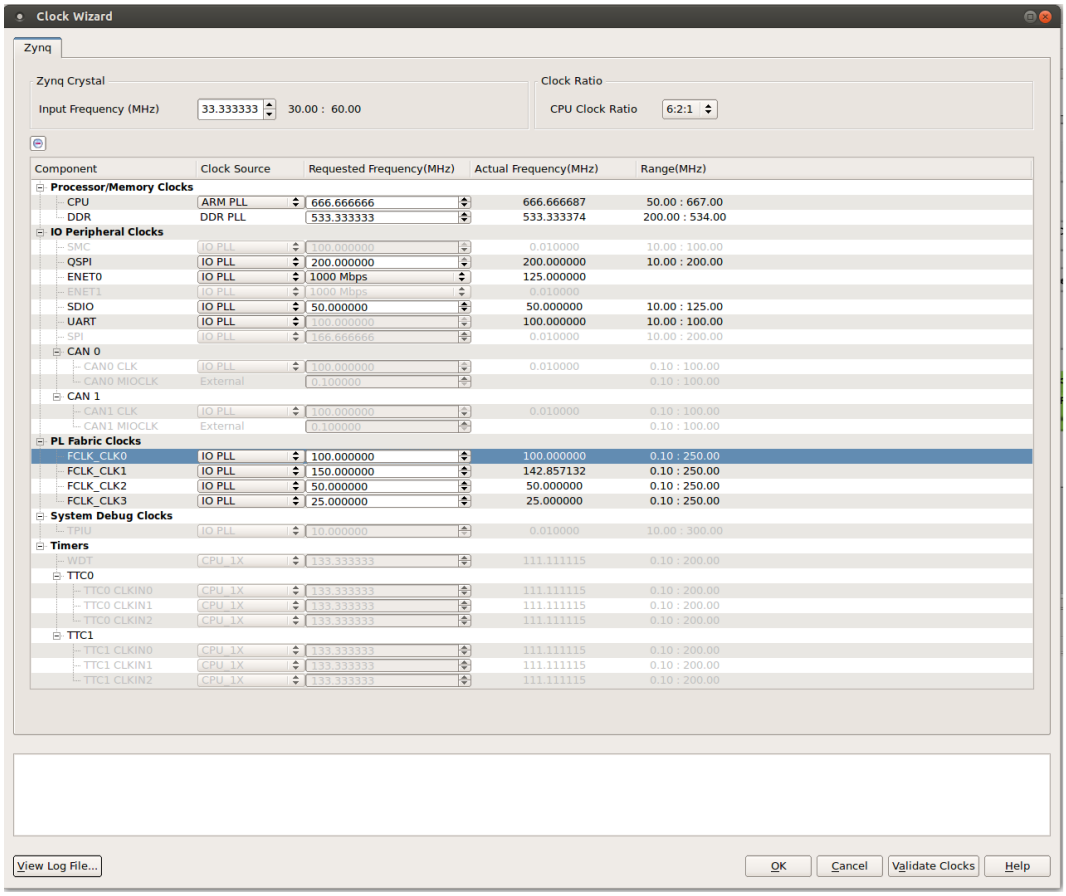

Figure 4.6.: Final Clock Wizard overview

7. Confirm the following settings for DDR Controller Configuration (on top-left),
   - Memory Type: DDR3.
   - Memory Part: MT41J128M16 HA-15E. With this Memory Part Configuration (on right) will turn gray.
   - Effective DRAM Bus Width: 32 Bit.
   - Operating Freq (MHz): 533.333333.
   - Internal Vref: checked.
   - Operating Temperature (C): Normal (0-85).
8. Confirm the following Training/Board Detail settings,
   - Check *Write leveling, Read gate, and Read data eye.*
   - Click *Expand To Calculate Delay.* A dialog box will appear to confirm the recalculation of current delay values, click *Yes* to confirm.
   - The delay Calculation tool would already have compiled the *Package Length (mils)*, but *Length (mm)* column needs to be filled according to Figure 4.7. These lengths are the lengths of connections on the PCB, which are further used to compute the latencies of the DDR signals. More details about the computation of these lengths are available on the ZedBoard's Hardware User manual.

- Click *OK* to return to XPS.

9. Clocks and DDR Controllers are now configured. Close XPS, File>Exit.
10. Now back on PlanAhead, in *Sources* panel under Design Sources, right-click *system (system.xmp)* and select *Create Top HDL*. This will create *system_stub.v* as top-level design module.

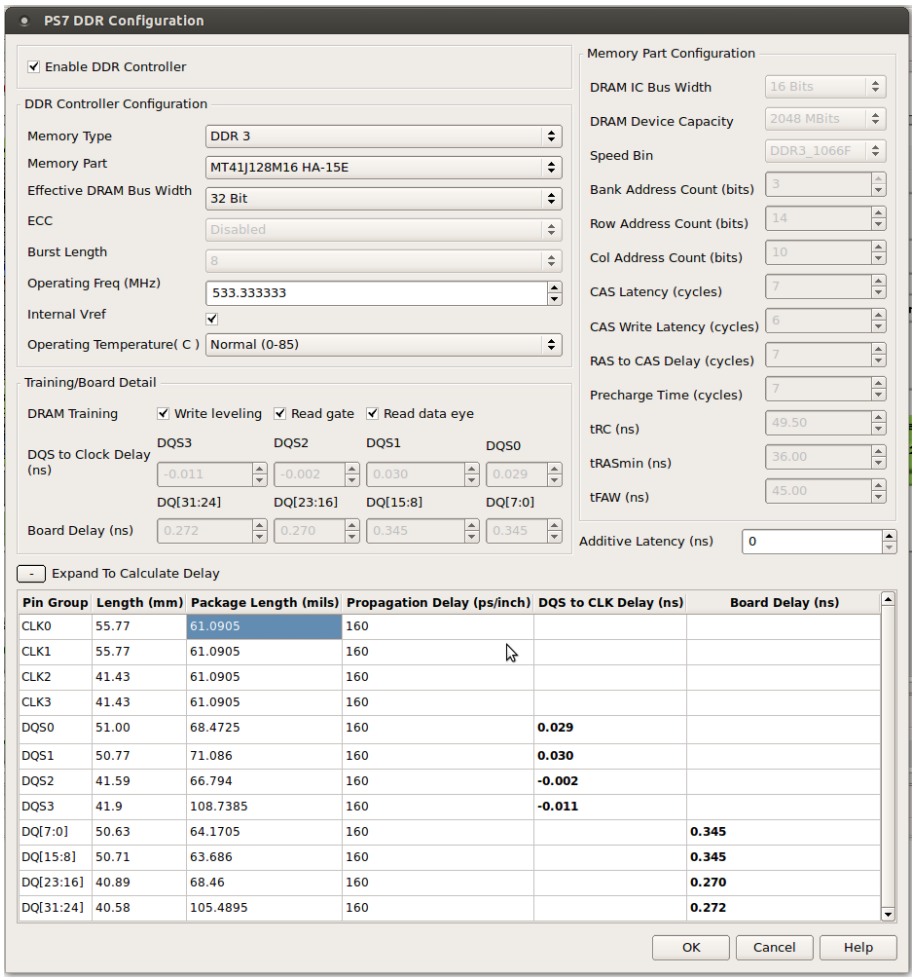

Figure 4.7.: DDR3 Configuration Wizard for PCB connection lengths

4.5. Add User Constraint File (UCF) for PL

When we start adding components in the PL, we need to define the pin mapping for HDL for modules to the pin locations on FPGA and the timing information for PL, this is defined by using User Constraint File (UCF). A UCF file is added as constraints source in the design.

1. In PlanAhead, click *Add Sources* in Navigator panel (on left). In the next window select *Add or Create Constraints* and click *Next*.
2. Click *Add Files...*, select *<project docs >/LabA/zedboard_UCF.ucf* file and click *OK*. Confirm that *Copy Constraints into Project* is checked. Click *Finish* to close the window.

3. The UCF file is now added under Constraints in Sources panel.
4. Now, generate the bitstream by clicking *Generate Bitstream* in the Navigator panel. Click *Yes* to confirm the synthesis and implementation.
5. Bit generation should finish without any Error and some 53 warnings. Click *OK* to continue the Bitgen.
6. After Bitstream Generation completed, select *View Reports* and click *OK* to view the reports. This will present set of synthesis reports for review.

Questions:

5. In *System Assembly View*, what are the functions of following configuration modules: Clock Generation, and Memory Interfaces?
6. What does the *Length* column in *DDR3 Configuration Wizard* (Fig 4.7) represent?
7. What is a UCF? What are its usage for FPGA?

4.6. Exporting Hardware to SDK and Test Application Development

1. In PlanAhead, select File>Export>Export Hardware for SDK... to export the Hardware for SDK.
2. Check *Export Hardware* and *Launch SDK* in the dialog pop-up and click *OK*.
3. At this point, PlanAhead will export the hardware specification of our project to SDK. It will export five files: *system.xml, ps7_init.h, ps7_init.c, ps7_init.tcl and ps7_init.html*. These files will be visible in the Project explorer panel (on left side) of the SDK. *system.xml* file contains the hardware specification of the design, and *ps7_init.h and ps7_init.c* files contain the initialization code for the Zynq chipset.
4. On the terminal verify that files in this locations have been updated to current time, <LabA>/ LED_project/LED_project.sdk/SDK/SDK_Export/system_hw_platform.
5. In SDK, Xilinx Board Support Package (BSP), is created by File >New >Board Support Package.
6. Verify the default settings as (Figure 4.8),
   - Project name: ”standalone_bsp_0”
   - Use default location: Checked
   - Hardware Platform: system_hw_platform
   - CPU: ps7_cortexa9_0
   - Board Support Package OS: Standalone
7. Click *Finish*, this will open *BSP Settings* dialog. In Overview, you will see the available support libraries. But for this lab we will not use any of them, so keep all unchecked. Click *OK*.
8. This will generate standalone_bsp_0 in the Project Explorer, and *auto build* should automatically build the project without any error.

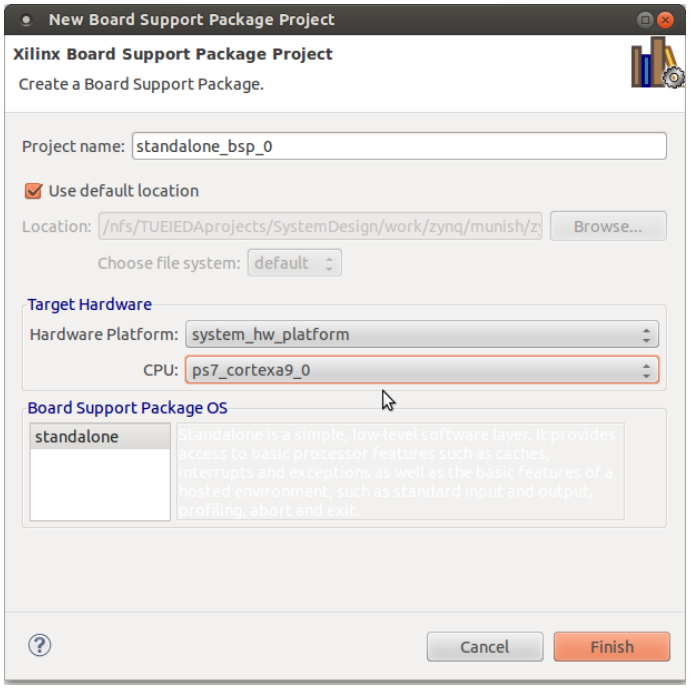

Figure 4.8.: Xilinx BSP project set up

9. To create and run test applications, in Xilinx SDK, select File >New >Application Project. In the new window, set Project name: *"memoryTest"*, and for Board Support Package select *Use existing: standalone_bsp_0*. Verify these default settings, Hardware Platform: *system_hw_platform*, Processor: *ps7_cortexa9_0*. Click *Next*.
10. Now select *Memory Tests* in the *Available Templates* column and click *Finish*.

4.7. ZedBoard Hardware set up and Memory/Peripheral Tests

1. ZedBoard has various configuration modes as shown in Figure 4.9. For this project we need to set ZedBoard to configure via JTAG. In Figure 1.2, Jumpers (JP7-JP11) are located on top-right corner of the ZedBoard. Connect JP[7-11] to GND, by setting the following jumper settings on the ZedBoard,

   JP7 [MODE0] – GND
   JP8 [MODE1] – GND
   JP9 [MODE2] – GND
   JP10 [MODE3] – GND
   JP11 [MODE4] – GND

2. Now, connect two microUSB-to-USB cables from ZedBoard *UART (J14)* and *PROG(J17)* ports to USB ports on PC.
3. Connect the power supply cable to ZedBoard *Power port (J20)*.
4. Switch on the ZedBoard power by turning ON switch *SW8*. Now, the green LED (LD13) should glow.

Xilinx TRM→	MIO[6] Boot_Mode[4]	MIO[5] Boot_Mode[2]	MIO[4] Boot_Mode[1]	MIO[3] Boot_Mode[0]	MIO[2] Boot_Mode[3]
JTAG Mode					
Cascaded JTAG					0
Independent JTAG					1
Boot Devices					
JTAG		0	0	0	
Quad-SPI		1	0	0	
SD Card		1	1	0	
PLL Mode					
PLL Used	0				
PLL Bypassed	1				
Bank Voltages					
MIO Bank 500			3.3V		
MIO Bank 501			1.8V		

Figure 4.9.: Configuration modes for ZedBoard.

5. On the PC terminal run *$lsusb*. The list should contain these two serial devices,
   Bus 006 Device 002: ID 04b4:0008 Cypress Semiconductor Corp.
   Bus 002 Device 006: ID 0403:6014 Future Technology Devices International, Ltd FT232H Single HS USB-UART/FIFO IC

6. We will use *minicom* as serial interface to ZedBoard. Type $minicom -s on the terminal. On minicom window goto *Serial port set up*, press enter. Make the following settings,
   A- Serial Device : /dev/ttyACM0
   E- Bps/Par/Bits : 115200 8N1

   Press *Esc*, select *Exit* and press *enter*. The *minicom* terminal should now show Port as */dev/ttyACM0*.

7. In Xilinx SDK, click *Program FPGA* button on the toolbar (or Xilinx Tools>Program FPGA). In the popup verify the *system_stub.bit* is from the current project (otherwise browse to the file in the current project) and click *Program*.

8. In Project Explorer tab right-click on memoryTest then select *Run As>Run Configurations...* . Double-click *Xilinx C/C++ application (GDB)* (on left) to create a new GDB run configuration, *memoryTest Debug* (Figure 4.10). Make sure that "Path to initialization TCL file" in the *Device Initialization Tab* points to the TCL file in the current active project. Then, click *Run*.

9. With successful Memory test, results should be as shown on the *minicom* terminal in Figure 4.11

10. Now, follow the previous section to set up test application for Peripheral Tests as peripheralTests and run the test application on ZedBoard. The successful results should look as in Figure 4.12.

Questions:

8. Which files are exported automatically from PlanAhead to SDK on hardware export? Give the roles for each of those files?

9. What do the Software applications for memory and peripheral tests do? (ref. to the software source code)

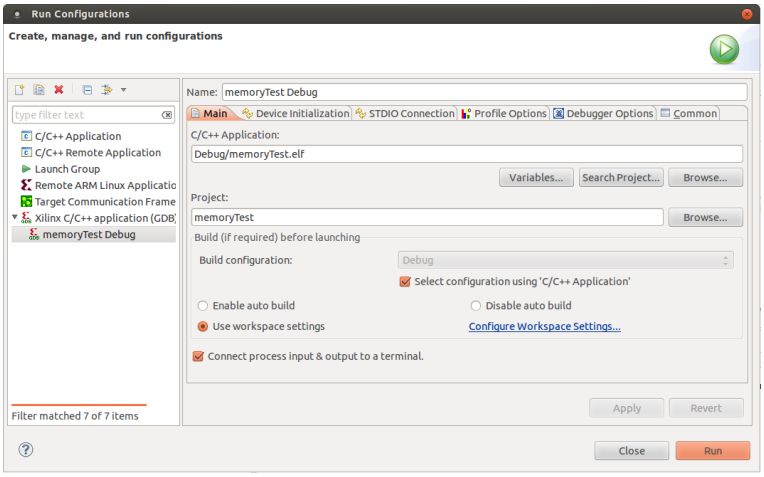

Figure 4.10.: GDB run configuration

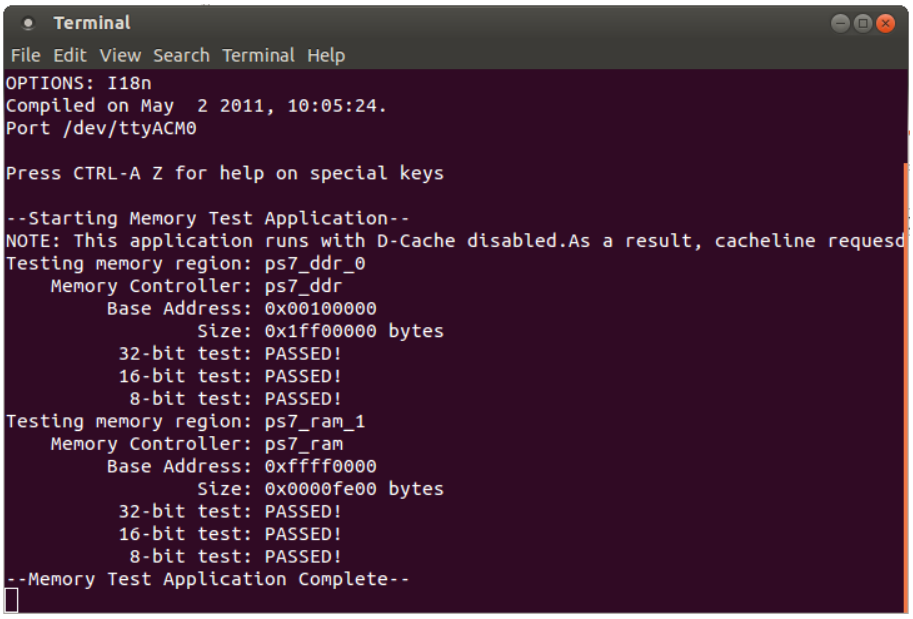

Figure 4.11.: *minicom* terminal output on successful memory Test

10. Which are three booting configurations available on ZedBoard?

4.8. Adding and Setting the Hardware for PL

1. For this part of the lab, we will return back to PlanAhead. So close SDK (if it is still open) and return to the PlanAhead.
2. Open XPS by double-clicking *system_i – system (system.xmp)*.
3. In XPS, select *Bus Interfaces* tab. From the *IP Catalog panel* (on left) drag *AXI General Purpose IO* under *General Purpose IO* to the *Bus Interfaces* panel. Click *Yes* to confirm adding axi_gpio_0.

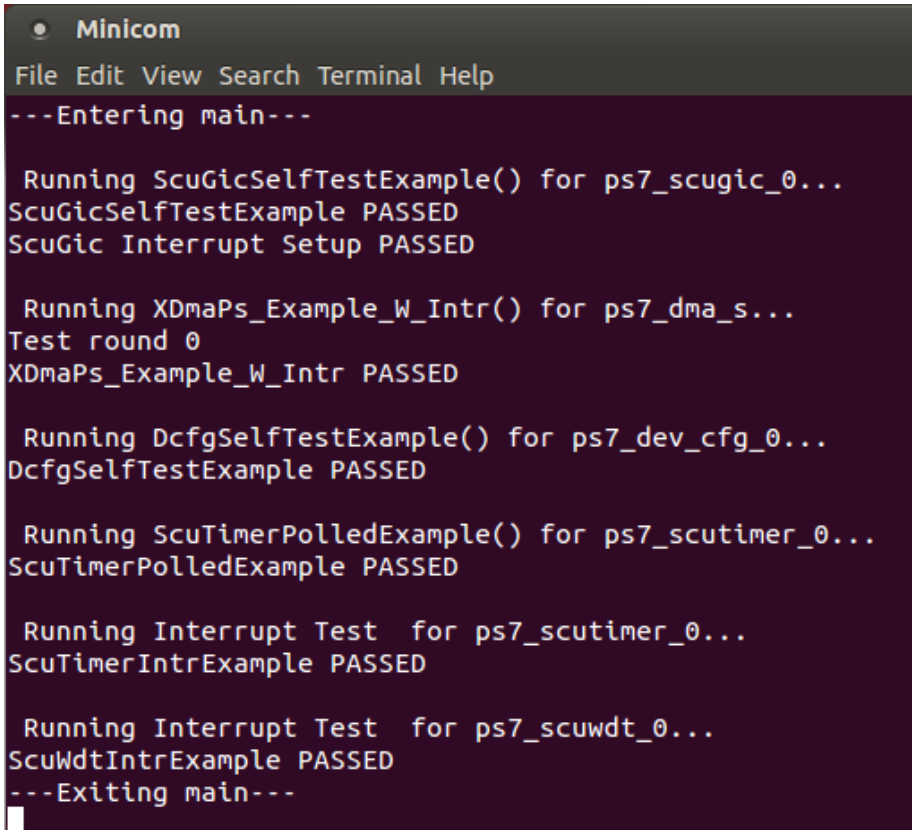

Figure 4.12.: *minicom* terminal output on successful peripheral tests

4. In the XPS Core Config window expand Channel 1. Verify that the *GPIO Data Channel Width* is set to 32, and keep other settings default. Click *OK*.
5. In Instantiate and Connect IP dialog box, select *Select processor instance* to *processing_system7_0* and click *OK*.
6. This will add axi_gpio_0 peripheral in the Bus Interfaces panel. Expand it to see the bus connections. Now, you will see connection from *M_AXI_GP0* to *S_AXI* of *axi_gpio_0* via *axi_interconnect_1* (Figure 4.13).
7. Now we will make the connections for GPIO ports. Switch to the *Ports* tab and expand all ports.
8. In the *Ports* panel, go to *axi_gpio_0* peripherals. Disconnect *GPIO_IO* port by right-clicking on the *Connected Port (External Ports:: axi_gpio_0_GPIO_IO_pin)* and selecting *Delete Connection*.
9. Since the data is output only to PL, make the *GPIO_IO_O* port *External* by right-clicking on it and selecting *Make External*.
10. Now scroll up to *External Ports* on *Ports* tab, and rename *axi_pgio_0_GPIO_IO_O_pin* to *LED_DutyCycle*, by clicking on its Name.
11. In the *Addresses* tab verify that *axi_gpio_0* has the *Base Address* 0x41200000.
12. To provide clocks for PL, in the *Ports* tab, select *FCLK_CLK0* under *processing_system7_0 peripheral* and make it also External.

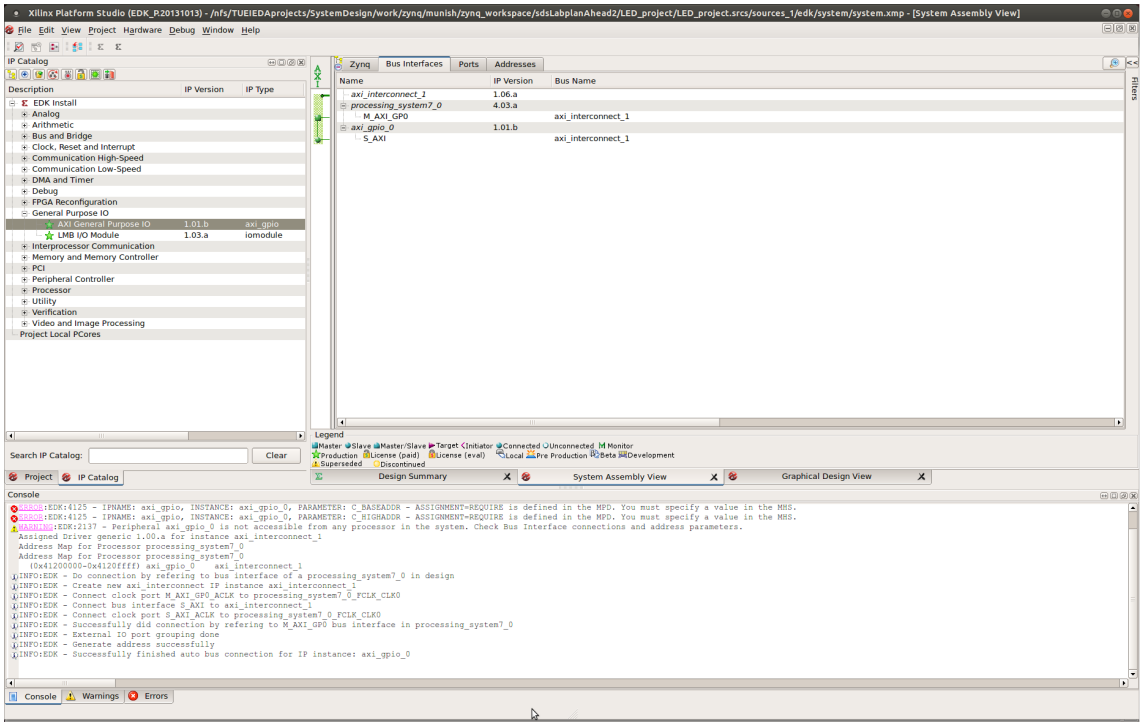

Figure 4.13.: Bus Interfaces panel showing peripherals and bus connections

13. Now go to the *Zynq* tab, and click *Clock Generation* to open the *Clock Wizard*. Set *FCLK_CLK0 and FCLK_CLK1* to *50MHz*. Click *Validate Clocks* and then *OK*.
14. Exit XPS by File>Exit.
15. Regenerate the Top HDL, by right-clicking system.xmp. In the newly generated *system_stub.v* you will see those additional external ports that we have just added, *processing_system7_0_ FCLK_CLK0_pin and LED_DutyCycle*.
16. Now, comment-out (by '//' before the line) these ports in the system_stub.v file in the *system_stub* module, as we will use these ports internally to connect to our custom module.
17. Also, comment out the corresponding output direction declarations within the *system_stub* module, and then add these two ports as wire signal declarations as shown in Figure 4.14
18. To add a hardware module to the design, on the *Navigator panel*, click *Add Sources* and select *Add or Create Design Sources*. Then *Add Files...* to select <project docs >/LED_Controller.v and click *Finish*. *LED_Controller.v* will be added under Design Sources.
19. Now, we need to instantiate LED_Controller at the top-level. For this copy the following code just before endmodule in system_stub.v file,

```
LED_Controller led1(
.Clk (processing_system7_0_FCLK_CLK0_pin),
.DutyCycle (LED_DutyCycle),
.LED_out ()
);
```

This will instantiate the LED_Controller module and will connect the *Clk* port to *processing_system7_0_FCLK_CLK0_pin*, and *DutyCycle* to *LED_DutyCycle*.

```
4
5 module system_stub
6   (
7     processing_system7_0_MIO,
8     processing_system7_0_PS_SRSTB_pin,
9     processing_system7_0_PS_CLK_pin,
10    processing_system7_0_PS_PORB_pin,
11    processing_system7_0_DDR_Clk,
12    processing_system7_0_DDR_Clk_n,
13    processing_system7_0_DDR_CKE,
14    processing_system7_0_DDR_CS_n,
15    processing_system7_0_DDR_RAS_n,
16    processing_system7_0_DDR_CAS_n,
17    processing_system7_0_DDR_WEB_pin,
18    processing_system7_0_DDR_BankAddr,
19    processing_system7_0_DDR_Addr,
20    processing_system7_0_DDR_ODT,
21    processing_system7_0_DDR_DRSTB,
22    processing_system7_0_DDR_DQ,
23    processing_system7_0_DDR_DM,
24    processing_system7_0_DDR_DQS,
25    processing_system7_0_DDR_DQS_n,
26    processing_system7_0_DDR_VRN,
27    processing_system7_0_DDR_VRP,
28    // LED_DutyCycle,
29    // processing_system7_0_FCLK_CLK0_pin,
30  );
31  inout [53:0] processing_system7_0_MIO;
32  input processing_system7_0_PS_SRSTB_pin;
33  input processing_system7_0_PS_CLK_pin;
34  input processing_system7_0_PS_PORB_pin;
35  inout processing_system7_0_DDR_Clk;
36  inout processing_system7_0_DDR_Clk_n;
37  inout processing_system7_0_DDR_CKE;
38  inout processing_system7_0_DDR_CS_n;
39  inout processing_system7_0_DDR_RAS_n;
40  inout processing_system7_0_DDR_CAS_n;
41  output processing_system7_0_DDR_WEB_pin;
42  inout [2:0] processing_system7_0_DDR_BankAddr;
43  inout [14:0] processing_system7_0_DDR_Addr;
44  inout processing_system7_0_DDR_ODT;
45  inout processing_system7_0_DDR_DRSTB;
46  inout [31:0] processing_system7_0_DDR_DQ;
47  inout [3:0] processing_system7_0_DDR_DM;
48  inout [3:0] processing_system7_0_DDR_DQS;
49  inout [3:0] processing_system7_0_DDR_DQS_n;
50  inout processing_system7_0_DDR_VRN;
51  inout processing_system7_0_DDR_VRP;
52 //  output [31:0] LED_DutyCycle;
53 //  output processing_system7_0_FCLK_CLK0_pin;
54  wire [31:0] LED_DutyCycle;
55  wire processing_system7_0_FCLK_CLK0_pin;
56
```

Figure 4.14.: Modified system_stub.v file with additional External signals

20. The *LED_out* port will be connected to LEDs (8 bit) on ZedBoard. Make a new port *LEDS* in the *system_stub* module and declare its direction in *system_stub.v*. Then connect *LEDS* to *LED_out* port of led1 instance. system_stub.v should look like as in Figure 4.15.
21. These new LEDS ports must be mapped in the UCF file to PL pins. This is done by mapping LEDS[0-7] to PL pin locations in UCF. In *zedboard_UCF.ucf* it is indicated by

```
 4
 5 module system_stub
 6   (
 7     processing_system7_0_MIO,
 8     processing_system7_0_PS_SRSTB_pin,
 9     processing_system7_0_PS_CLK_pin,
10     processing_system7_0_PS_PORB_pin,
11     processing_system7_0_DDR_Clk,
12     processing_system7_0_DDR_Clk_n,

26     processing_system7_0_DDR_VRN,
27     processing_system7_0_DDR_VRP,
28     // LED_DutyCycle,
29     // processing_system7_0_FCLK_CLK0_pin,
30     LEDS
31   );
32   inout [53:0] processing_system7_0_MIO;
33   input processing_system7_0_PS_SRSTB_pin;
34   input processing_system7_0_PS_CLK_pin;
35   input processing_system7_0_PS_PORB_pin;

47   inout [31:0] processing_system7_0_DDR_DQ;
48   inout [3:0] processing_system7_0_DDR_DM;
49   inout [3:0] processing_system7_0_DDR_DQS;
50   inout [3:0] processing_system7_0_DDR_DQS_n;
51   inout processing_system7_0_DDR_VRN;
52   inout processing_system7_0_DDR_VRP;
53   output [7:0] LEDS;
54 //  output [31:0] LED_DutyCycle;
55 //  output processing_system7_0_FCLK_CLK0_pin;
56   wire [31:0] LED_DutyCycle;

81       .processing_system7_0_DDR_VRN ( processing_system7_0_DDR_VRN ),
82       .processing_system7_0_DDR_VRP ( processing_system7_0_DDR_VRP ),
83       .LED_DutyCycle ( LED_DutyCycle ),
84       .processing_system7_0_FCLK_CLK0_pin ( processing_system7_0_FCLK_CLK0_pin )
85     );
86
87     LED_Controller led1(
88     .Clk (processing_system7_0_FCLK_CLK0_pin),
89     .DutyCycle (LED_DutyCycle),
90     .LED_out (LEDS)
91     );
92
93 endmodule
```

Figure 4.15.: Instantiation of LED_Controller and corresponding port connections.

LD0-LD7 (lines 133-140). Replace LD0-LD7 with LEDS[0-7] as in Figure 4.16.

22. Now, generate the bitstream by clicking *Generate Bitstream* in the Navigator panel. Click *Yes* to confirm the synthesis and implementation.
23. Bit generation should finish without any Error and about 53 warnings. Click *OK* to continue the Bitgen.
24. After bitstream generation completed, select *View Reports* and click *OK* to view the reports. This will show a set of synthesis reports for review.

Questions:

9. What are "External ports" in the *System Assembly View* for *Addresses*?
10. How is the intensity of the LEDs controlled in the hardware description of *LED_Controller* module?

```
126 NET HD_HSYNC      LOC = V17  | IOSTANDARD=LVCMOS33;  # "HD-HSYNC"
127 NET HD_INT        LOC = W16  | IOSTANDARD=LVCMOS33;  # "HD-INT"
128 NET HD_SCL        LOC = AA18 | IOSTANDARD=LVCMOS33;  # "HD-SCL"
129 NET HD_SDA        LOC = Y16  | IOSTANDARD=LVCMOS33;  # "HD-SDA"
130 NET HD_SPDIF      LOC = U15  | IOSTANDARD=LVCMOS33;  # "HD-SPDIF"
131 NET HD_SPDIF0     LOC = Y18  | IOSTANDARD=LVCMOS33;  # "HD-SPDIF0"
132 NET HD_VSYNC      LOC = W17  | IOSTANDARD=LVCMOS33;  # "HD-VSYNC"
133 NET LEDS[0]           LOC = T22  | IOSTANDARD=LVCMOS33;  # "LD0"
134 NET LEDS[1]           LOC = T21  | IOSTANDARD=LVCMOS33;  # "LD1"
135 NET LEDS[2]           LOC = U22  | IOSTANDARD=LVCMOS33;  # "LD2"
136 NET LEDS[3]           LOC = U21  | IOSTANDARD=LVCMOS33;  # "LD3"
137 NET LEDS[4]           LOC = V22  | IOSTANDARD=LVCMOS33;  # "LD4"
138 NET LEDS[5]           LOC = W22  | IOSTANDARD=LVCMOS33;  # "LD5"
139 NET LEDS[6]           LOC = U19  | IOSTANDARD=LVCMOS33;  # "LD6"
140 NET LEDS[7]           LOC = U14  | IOSTANDARD=LVCMOS33;  # "LD7"
141 NET NetIC16_W20   LOC = W20  | IOSTANDARD=LVCMOS33;  # "NetIC16_W20"
142 NET NetIC16_W21   LOC = W21  | IOSTANDARD=LVCMOS33;  # "NetIC16_W21"
143 NET VGA_B1        LOC = Y21  | IOSTANDARD=LVCMOS33;  # "VGA-B1"
144 NET VGA_B2        LOC = Y20  | IOSTANDARD=LVCMOS33;  # "VGA-B2"
145 NET VGA_B3        LOC = AB20 | IOSTANDARD=LVCMOS33;  # "VGA-B3"
146 NET VGA_B4        LOC = AB19 | IOSTANDARD=LVCMOS33;  # "VGA-B4"
```

Figure 4.16.: UCF configuration for LEDS ports in PL

4.9. Software Application Development for LED Project

1. As next step, from Xilinx PlanAhead export the hardware to SDK for software development, File>Export>Export Hardware for SDK... . Check *Launch SDK* and *Export Hardware*, then click *OK*.
2. Create a new software in SDK, File>New>Application Project. Name the project as *"LED_software"*. Select *OS Platform* as standalone, Language as *C* and Board Support Package as Use existing: *standalone_bsp_0*. Then click *Next*.
3. From the available packages, select *Empty Application* and click *Finish*.
4. In the Project Explorer, right-click *src* under *LED_software* and select *New File* and name it *"main.c"*.
5. Goto the following path on browser to find some example applications for GPIO drivers, file:///nfs/ tools/xilinx/ise/14.7/ISE_DS/EDK/sw/XilinxProcessorIPLib/drivers/gpio_v3_01_a/examples/index.html. Click *xgpio_example.c* to open the file. This file is also provided in the project documents directory. From this file, following code snippet is relevant to our project,

```
#include "xparameters.h" /*
#include "xgpio.h"

#define LED_CHANNEL 1
XGpio Gpio; /* The Instance of the GPIO Driver */
```

6. Add those to *main.c* file and it should now look like as in Figure 4.17.
7. Now, for the *main()* function, copy this code (similar to *xgpio_example.c*) in the *main.c*.

```
int main(void)
{
 int Status;
/* These three variables we will need later*/
```

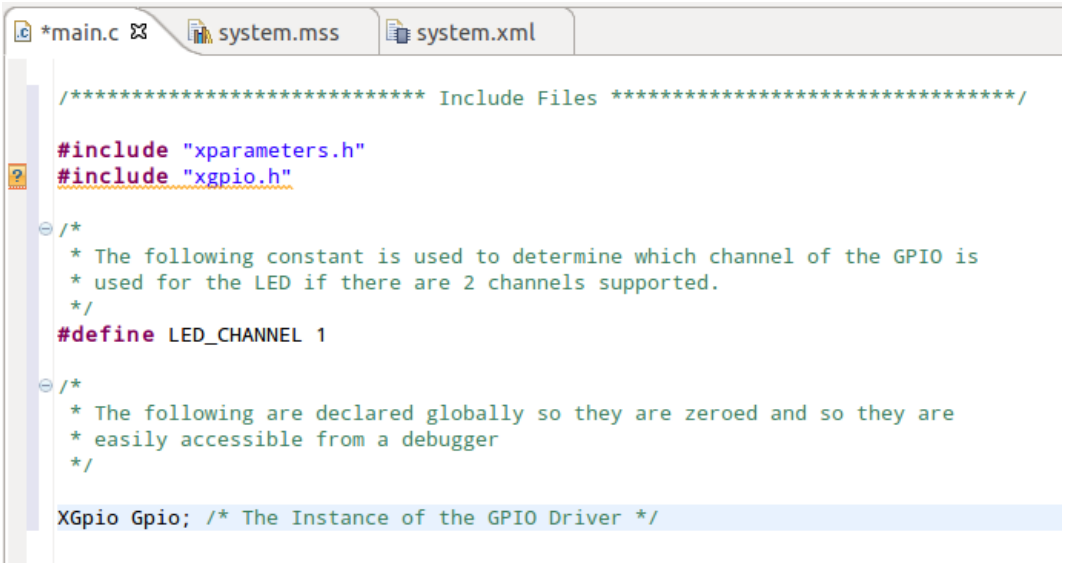

Figure 4.17.: main.c after preparation from example driver function

```
u32 value = 0;
u32 period = 0;
u32 brightness = 0;
/*
* Initialize the GPIO driver
*/
Status = XGpio_Initialize(&Gpio, GPIO_EXAMPLE_DEVICE_ID);
if (Status != XST_SUCCESS) {
return XST_FAILURE;
}
```

8. Now, use *Xgpio_DiscreteWrite* function to reset the LEDs,

```
/* To Clear LEDs */
Xgpio_DiscreteWrite(&Gpio, LED_CHANNEL, 0);
```

9. Following code snippet will read the data input from user via *UART* port to set the brightness of the LEDs.

```
while (1) {
print("Please Select Brightness between 0 and 9\n\r");
value = inbyte();
period = value - 0x30;
xil_printf("Brightness Level %d selected\n\r", period);
```

The *inbyte()* function will read data from the *UART* port. Then, the *value* variable is converted from ASCII to decimal and stored in variable *period*.

10. Brightness value is now scaled by 110,000, written to PL, close the while loop and return to close the function.

```
brightness = period * 110000;
```

```
XGpio_DiscreteWrite(&Gpio, LED_CHANNEL, brightness);
} // while loop ends

return XST_SUCCESS;

} // main() ends
```

11. Save *main.c* file and *main()* should look like in Figure 4.18
12. Here you will get an error that *Error:gpio.h No such file.* Now, right-click on *standalone_bsp_0* and select *BSP Setting.* Driver>axi_gpio: change from none to gpio. Click *OK.*

```
int main(void)
{
    int Status;
    /* These two variables we will need later*/
    u32 value = 0;
    u32 period = 0;
    u32 brightness = 0;

    /*
     * Initialize the GPIO driver
     */
    Status = XGpio_Initialize(&Gpio, GPIO_EXAMPLE_DEVICE_ID);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /* To Clear LEDs*/
    XGpio_DiscreteWrite(&Gpio, LED_CHANNEL, 0);


    while (1) {
        print("Please select the Brightness between 0 and 9\n\r");
        value = inbyte();

        period = value - 0x30;
        xil_printf("Brightness Level %d selected\n\r", period);

        brightness = period * 110000;
        XGpio_DiscreteWrite(&Gpio, LED_CHANNEL, brightness);
    } // while loop ends

    return XST_SUCCESS;
} // main() ends
```

Figure 4.18.: main() function for LED_software project.

13. The *xparameters.h* is another important file which contains are the address mapping of design module registers. To define *GPIO_EXAMPLE_DEVICE_ID*, we will add the definition of device ID from *xparameters.h* file (standalone_bsp_0 >ps7_cortexa9_0 >include >xparameters.h) within the definitions for *AXI_GPIO_0* as,

    ```
    #define GPIO_EXAMPLE_DEVICE_ID XPAR_AXI_GPIO_0_DEVICE_ID
    ```

14. Now, the LED_software project should be able to build without any error and is ready to be exported on FPGA.

4.10. ZedBoard Hardware set up and FPGA Programming

1. Please set up the ZedBoard hardware and *mincom* as we did earlier for the memory and peripheral tests. Here we will run LED_software in place of memory/peripheral test softwares.
2. Program the FPGA. LED *LD12* on ZedBoard should turn-on on successful FPGA programming.
3. On SDK, select Run>Run As>1.Launch on Hardware (GDB).
4. Now, you should be able to execute the project on the FPGA and control the LED brightness using keyboard numbers from 0-9. Figure 4.19 shows how the *minicom* output will look like.
5. Exit from *minicom*, *Ctrl+A >Z >X*, and select *Yes*.

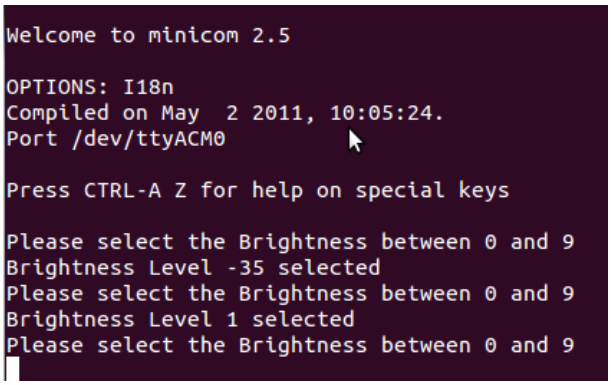

Figure 4.19.: *minicom* output on the terminal for *LED_software* project.

Questions:

11. What information does *xparameters.h* file contain?
12. How does *minicom* communicate with PS? Which peripherals are involved?
13. Trace the hierarchy of GPIO initialization function *XGpio_Initialize* and give the GPIO initilization control register values?

Bibliography

[1] ITRS-2011, *www.itrs.net*

[2] ANSI, IEEE Standards Board, IEEE Standard VHDL Language Reference Manual: IEEE Std 1076-1993, New York, 1988, ISBN 1559373768

[3] Peter J. Ashenden, Designer's Guide to Vhdl, Morgan Kaufmann Publishers, 1995, ISBN 1558602704

[4] Giovanni De Micheli, Synthesis and Optimization of Digital Circuits, McGraw-Hill Book Company, 1994, ISBN 0070163332

[5] Jayaram Bhasker, A VHDL Primer, Prentice Hall, Englewood Cliffs, 1998, ISBN 0130965758

[6] Zedboard, www.zedboard.org.

[7] Zedboard Hardware User's Guide v1.1, Aug 2012, Digilent.

[8] LogiCORE IP AXI Video Direct Memory Access v6.1, PG020, Dec 2013.

[9] Xilinx, www.xilinx.com.

[10] Vivado Design Suite Tutorial, UG871, June 2013, Xilinx.

[11] OV7670 (2006): OV7670/OV7171 CMOS VGA (640x480) CAMERACHIPT M with OmniPixel Technology, OmniVision Technologies Inc.

[12] PG, ADV7511 (2012): Low-Power HDMI Transmitter Programming Guide, Analog Devices, Inc.

[13] Zynq-7000 All Programmable SoC TRM v1.7, UG585, Feb 2014, Xilinx.

[14] Vivado Design Suite User Guide, High-Level Synthesis, UG902(v2013.2), June 19, 2013, Xilinx.

[15] Introduction to Zynq-7000^{TM} All Programmable SoC, Speedway.

[16] Umair Razzaq, Design Space Exploration for Embedded Vision Applications on Zynq FPGA using HLS, 2013, Master Thesis, EDA-TUM.