



MIDDLE EAST TECHNICAL UNIVERSITY



ELECTRICAL AND ELECTRONICS ENGINEERING
DEPARTMENT

EE446

COMPUTER ARCHITECTURE II

Laboratory Manual

Course Instructors:

Prof. Dr. Ece GÜRAN SCHMIDT

Laboratory Assistants:

Doğ u Erkan ARKADAŞ

Bariş TIRYAKI

March 2023

Contents

1	Laboratory Regulations	3
1.1	Rules	3
1.2	Cheating	4
1.3	Remarks and Evaluation	4
2	General Information about Laboratory	5
2.1	Experimental Setup	5
3	Quartus Software	6
4	Simulation Software	7
4.1	Simulation using cocotb	7
5	Some Useful How-to Items	8
5.1	How to create a Verilog module with parameters	8
5.2	How to initialize the Memory on FPGA	8
5.3	Problems with running Cocotb	9
5.3.1	Path Issues	9
5.3.2	"I give up" error	9
5.4	Making your Code Compile Faster	9
5.4.1	Avoiding Latches	9

Course Instructors

Name	e-mail	Room
Prof. Dr. Ece Gran Schmidt	eguran@metu.edu.tr	A-402

Laboratory Assistants

Name	e-mail	Room
Doęu Erkan Arkadař	arkadas@metu.edu.tr	A-404
Barıř Tiryaki	btiryaki@metu.edu.tr	A-405

1 Laboratory Regulations

This laboratory is a very important part of *EE446 - Computer Architecture II* course to understand the concepts in the computer architecture lectures thoroughly. By attending experiments and completing all the work, the key concepts and most of the abstract parts of the lectures can be grasped very easily. Thus, it is important to know the regulating rules of this laboratory for both a better understanding and for your grades.

There are rules for the regulation of *EE446 Laboratory*. These rules are strict, and by taking *EE446 - Computer Architecture II* course, you will be considered that you have understood and accepted all the rules stated below.

1.1 Rules

The rules for EE446 - Computer Architecture II Laboratory are given below. Please read thoroughly:

1. The manual of an experiment will be available at least a week before the corresponding experiment.
2. A preliminary work to be detailed in each experiment manual has to be prepared for each experiment.
3. The preliminary works will be collected last Sunday before the corresponding laboratory week through the ODTUCLASS.
4. Preliminary work is a crucial part of the laboratory work in preparing for the experiment and understanding the concepts to be covered in the corresponding experiment. Thus, **You will NOT be allowed** to attend the relevant laboratory session without any preliminary work. Partially done preliminary work reports are acceptable if at least you got a passing grade of 50%.
5. **Cheating and plagiarism will result in zero grades, whereas disciplinary actions may also be taken.** Please read the subsection 1.2 for detailed information about cheating.
6. Experiments are to be performed individually.
7. Grading for each laboratory work will be 40% preliminary work and 60% laboratory performance.
8. Talking and sharing information between students during a laboratory session is strictly forbidden. Repeated offenses of this rule will make your lab performance grade 0.
9. No extra time will be given to the latecomers.
10. To leave the laboratory room during a session, you must get permission from the laboratory assistant.
11. Transfer between laboratory sessions (e.g. from *Group 1* to *Group 5*) will NOT be allowed unless you have a valid excuse (medical report, etc.).
12. Your codes have to be well commented. The codes lacking any comments or overly laden with comments will not be evaluated. Please read the subsection 1.3 for detailed information about coding and commenting.
13. Students with officially documented legal excuses will be allowed to take make-ups. Only academic permissions (given by the University), signed confirmation from the instructors for any exam clashes, and METU Health and Counseling Center (MEDIKO) will be considered valid documents for the right to take make-ups. You may take at most three make-ups even if you have more than three officially documented legal excuses.

Since preliminary work spans several weeks unless you have a valid excuse for the said weeks, you will not be allowed to submit new preliminary work

14. Students who do not attend or get zero grades from **3 or more** experiments, excluding the first one, will get **N/A** from the EE446 course. These students also won't be allowed to participate in the term project. Please note that insufficient preparation of the preliminary work of an experiment or plagiarism is equivalent to not attending the corresponding experiment.

15. **Be gentle with the FPGAs as they are expensive. Each FPGA costs around 364£ before shipping and tax.** You will lose performance grades if you handle FPGAs roughly during the laboratory sessions or do not put them in their boxes properly at the end of the sessions.
16. Bringing materials outside of those provided to the laboratory (written codes, etc.) are strictly prohibited

1.2 Cheating

You are considered to graduate and become an engineer in 1 or 2 years. Hence, you are expected to act according to the professionalism required as a METU graduate.

Copying work from any other resource (web page, your friend's report, older resources you have found, etc.) during preliminary work or sharing information or code files during sessions is considered cheating. Automated tools such as ChatGPT are allowed if the output is changed sufficiently to be different from other students; otherwise, it will count as plagiarism.

Helping your friends, studying together, or any form of cooperation is encouraged -since it fosters your relationships with others and helps you learn the topic better- but YOU do your own work. Creating only one report/code is not studying together or is not cooperation and will NOT be accepted.

1.3 Remarks and Evaluation

Your laboratory grade comprises your preliminary work grade and laboratory session performance grade. Preliminary work requires the implementation of different computer architectures, all of which are used to experiment during the laboratory sessions. The performance grade is based on the functionality of your implementations, the comprehensiveness of your knowledge of the related laboratory topic, and how much help you get from your TAs. TAs will help you as if they are your laboratory partner; however, any big help you receive (such as a TA writing code for you or fixing major issues in your design) will result in a percentage reduction in your laboratory performance grade. If a big part of the design is not implemented in your preliminary work, you will most likely get ZERO performance grades from the corresponding step, as implementing the design during the laboratory session is nearly impossible.

Implementations without comments will not be considered valid implementations of the tasks, and the corresponding task will not be evaluated. You are expected to write down explanatory comments in your code. That does not mean you should write an explanation next to each code line. What is required is an explanation of the functionality of the code blocks and the functionality of the representative code lines where necessary.

You are always expected to do proper test benching of your code in software before embedding it into an FPGA. **For this laboratory, test benches of all the designs (except for the first laboratory and the term project) will be given to you.** You are also expected to practice your implementations with the FPGAs before attending your laboratory session so that major bugs that may cost too much time to fix can be eliminated and you are on the safer side to complete the experimental work within the required time slot. You can practice and work on your implementations by using the laboratory at EA-407, which is open to access 24/7.

2 General Information about Laboratory

As it is mentioned before, this laboratory is a very important part of *EE446 - Computer Architecture II* course. The laboratory work helps you understand most of the concepts given in the computer architecture lectures.

The experiments will be carried out in Microprocessor Laboratory at EA-409 which is located on the 4th floor of A Block of our department. EA-407 laboratory will be open to access on the weekdays during working hours (09:00-17:00) for you to practice. Hence, you may test and debug your preliminary work prior to your laboratory session.



Figure 1: Microprocessor and Computer Architecture Laboratory, A Building, Room 407

There are different sessions of laboratory and you are assigned to one of the sessions to perform your experiments.

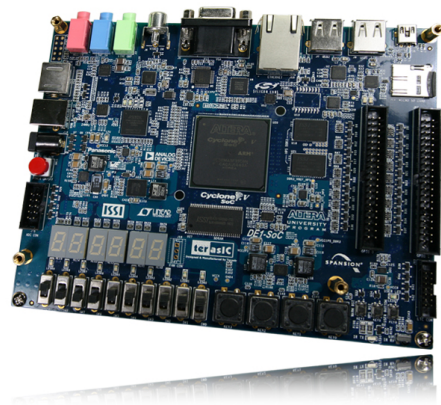
There will be a total of 4 experiments and 1 course project. These experiments and the project are based on constructively practicing the design of computers via Verilog hardware description language and will be on the following subjects:

- Fundamental modules for computer design
- Single cycle computer
- Multi-cycle computer
- Pipelined computer
- Project: Simplified RISC-V Computer

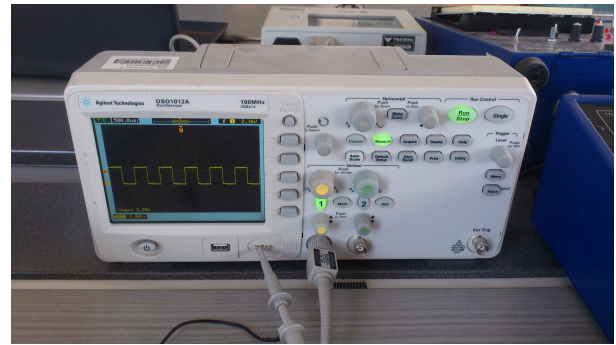
2.1 Experimental Setup

The experimental setup of EE446 Laboratory is composed of the following items:

- A notebook computer
 - You may login as a student, which does not require any password
 - Computer has Quartus and other materials you may need during the lab/ when practicing
- DE1-SoC Development Kit - A board containing Altera Cyclone V 5CSEMA5F31C6N FPGA
 - DE1-SoC is connected to the notebook via a USB data cable.
 - DE1-SoC is programmed via Quartus installed on the notebook (to be explained in Section 3).



(a) DE1-SoC Board



(b) Digital Oscilloscope

Figure 2: Devices used in the laboratory

- For simulation purposes, cocotb is installed on the notebooks (to be explained in Section 4.1).
- Digital Oscilloscope - Oscilloscope for debugging purposes

3 Quartus Software

Throughout the labs of this semester, we will be using Quartus Prime Lite Edition to be able to program the DE1-SoC FPGA board. This requires very little know-how in creating a project, adding files, compiling, and programming with the compiled board. To learn how to do this you should read sections 4-6 of [Quartus Prime Introduction](#) Using Verilog Designs documents. DE1-SOC board pin assignments and project creation will be done using the vendor supplied DE1-SoC System Builder program which you should read about in [DE1-SoC user manual](#) l section 4. These are very light readings which should only take about 30 minutes of your time, read them before attending laboratory sessions.

4 Simulation Software

This section introduces simulation software to be used throughout the laboratory work to perform behavioral simulation for the Verilog design codes and embed the designs to the FPGA, respectively. Basics for the cocotb are covered in this section. For more general usage tutorials of the cocotb, one can refer to the tutorials available on the EE446 ODTUClass course page.

4.1 Simulation using cocotb

This part only concisely covers the basics of cocotb. To get a more detailed look into cocotb with installation guides, please check the cocotb document in ODTUCLASS or official [cocotb documents](#).

Cocotb is simply an interface between a Verilog simulator (Icarus Verilog for this course) and a Python script you will write (or one that will be given to you) as a test bench. Cocotb will show your Verilog design as an object in the Python script. Thus, every signal in your Verilog design will be a variable of the Verilog object. You can use anything available in Python to verify your design. To better understand this, please check the supplied test bench examples on ODTUCLASS. Since Cocotb is fully on Python, unlike ModelSim, no Verilog knowledge is required to test any design on the bench.

To run your test bench, open the Anaconda Prompt and change the directory to match the location of the test files. If your "User" folder or the folder test-bench is in has a space in its PATH, cocotb will give an error so be mindful of that. Use **make** command to run the test bench with the default simulator (specified in the makefile) or **make sim=icarus** to simulate with Icarus Verilog explicitly.

5 Some Useful How-to Items

In this section, some tips and tricks on using Quartus software and Verilog programming are to be presented. Those can be helpful for your preliminary work tasks.

5.1 How to create a Verilog module with parameters

There are many cases where designing a module should be generalized. For example, a registered design can be used for different data widths. It is cumbersome to implement the same design to support different data widths. To overcome this, some parts of the module can be parametrized so that modules with different properties of the same design can be instantiated. For the register example, one can make the data width a parameter so that by varying the parameter, registers of different data width can be created from the same design.

In Verilog, a module can be easily parameterized. You will just add a parameter block before defining the arguments of the module. An example of a module with parameters is given below:

```
// an example module with parameters
// the module is to split a data bus into two

module bus_split #(parameter W=8, S=4) (bus_in , split_out0 , split_out1);

    input wire [(W - 1):0] bus_in;

    output wire [(W - S - 1):0] split_out1;
    output wire [(S - 1):0] split_out0;

    assign bus_out0 = bus_in [(S - 1):0];
    assign bus_out1 = bus_in [(W - 1):S];

endmodule
```

Note that, the default values of the parameters should be provided so that the module can be instantiated with the default parameters if no parameter setting is performed. To create a module with a specified parameter, the values of the parameters should be supplied to the module in the order they are defined:

```
// assume you want to split the content of the instruction register
// say the name of the output of the instruction register is reg_out

...

wire [3:0] instr;          // most significant 4 bits are for instruction
wire [11:0] oprnd;        // the rest is for operand

bus_split #(16, 12) split_instr(.bus_in(reg_out), .split_out0(oprnd),
    .split_out1(instr));

...
```

5.2 How to initialize the Memory on FPGA

To initialize the memory on the FPGA you need to use "readmemh" command inside an initial block as in below. One thing you need to make certain of is that the hex file should have elements for each memory location.

For example: If you have a memory with 256 locations each containing a byte hex file should have 256 bytes specified in it.

```

initial begin
    $readmemh("instructions.hex", mem, 0);
end

```

5.3 Problems with running Cocotb

5.3.1 Path Issues

Cocotb does not like Turkish characters (as do most software libraries). If you have a Turkish character in your files or in your file path (either the test folder or the folder Icarus/Python is installed) you will get a "Unicode Error" when trying to run tests.

5.3.2 "I give up" error

This is usually because one or more components of your environment are not set up correctly.

The best way to solve this is to go to ODTUClass or the cocotb website and reinstall everything by following the steps. We highly suggest using a version of Anaconda as it handles most of the environment for you.

Also, older Icarus versions can cause this error, so delete the old version and install the latest version (should be v12+).

5.4 Making your Code Compile Faster

5.4.1 Avoiding Latches

For your codes to compile faster, you need to avoid any undesirable latches.

For your combinational circuits to not produce latches, you must specify every signal's value in each possible case.

For example:

If you write a mux like this:

```

always @(*) begin
    case(select)
        2'b00: output_value = input_0;
        2'b01: output_value = input_1;
        2'b10: output_value = input_2;
    endcase
end

```

Since 1 case is empty, the synthesizer assumes that case means storing the previous value. A combinational circuit that stores a value results in a latch you can see in Figure 3

The same thing can happen if you have multiple signals in your case statements. Even if some of the signals are "don't care" in some cases, you should always assign a value to every single signal in every case.

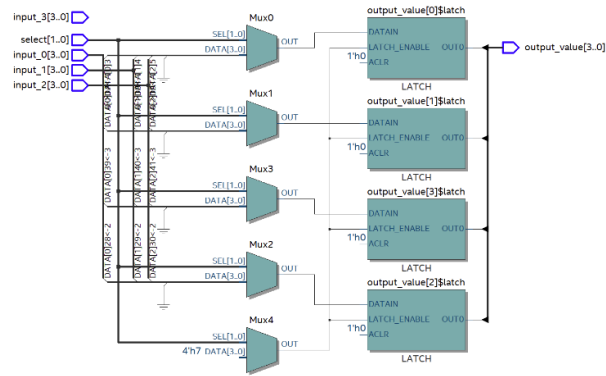
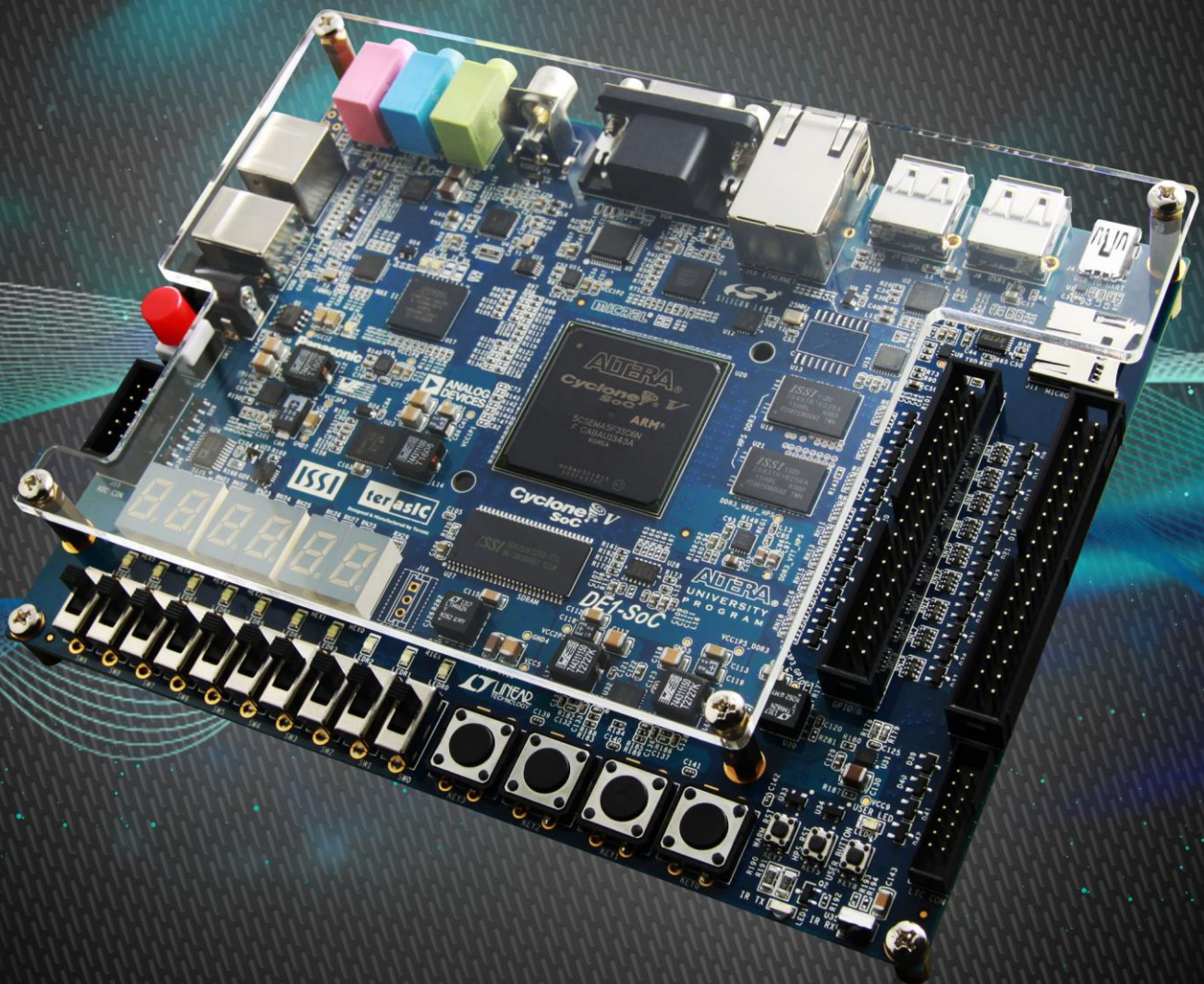


Figure 3: Example of a Latch in Synthesis

DE1-SOC

USER MANUAL



Chapter 1	DE1-SoC Development Kit.....	4
1.1	Package Contents	4
1.2	DE1-SoC System CD	5
1.3	Getting Help	5
Chapter 2	Introduction of the DE1-SoC Board	6
2.1	Layout and Components.....	6
2.2	Block Diagram of the DE1-SoC Board.....	8
Chapter 3	Using the DE1-SoC Board	12
3.1	Settings of FPGA Configuration Mode	12
3.2	Configuration of Cyclone V SoC FPGA on DE1-SoC.....	13
3.3	Board Status Elements.....	19
3.4	Board Reset Elements	20
3.5	Clock Circuitry	21
3.6	Peripherals Connected to the FPGA.....	23
3.6.1	User Push-buttons, Switches and LEDs	23
3.6.2	7-segment Displays	26
3.6.3	2x20 GPIO Expansion Headers.....	28
3.6.4	24-bit Audio CODEC	30
3.6.5	I2C Multiplexer	31
3.6.6	VGA	32
3.6.7	TV Decoder	35
3.6.8	IR Receiver.....	37
3.6.9	IR Emitter LED	37

3.6.10	SDRAM Memory	38
3.6.11	PS/2 Serial Port.....	40
3.6.12	A/D Converter and 2x5 Header	42
3.7	Peripherals Connected to Hard Processor System (HPS).....	43
3.7.1	User Push-buttons and LEDs.....	43
3.7.2	Gigabit Ethernet.....	44
3.7.3	UART	45
3.7.4	DDR3 Memory	46
3.7.5	Micro SD Card Socket.....	48
3.7.6	2-port USB Host	49
3.7.7	G-sensor.....	50
3.7.8	LTC Connector	51
Chapter 4	DE1-SoC System Builder	53
4.1	Introduction	53
4.2	Design Flow	53
4.3	Using DE1-SoC System Builder	54
Chapter 5	Examples For FPGA	60
5.1	DE1-SoC Factory Configuration.....	60
5.2	Audio Recording and Playing	61
5.3	Karaoke Machine	64
5.4	SDRAM Test in Nios II.....	66
5.5	SDRAM Test in Verilog	69
5.6	TV Box Demonstration	71
5.7	PS/2 Mouse Demonstration.....	73
5.8	IR Emitter LED and Receiver Demonstration	76
5.9	ADC Reading	82
Chapter 6	Examples for HPS SoC	87

6.1 Hello Program	87
6.2 Users LED and KEY	89
6.3 I2C Interfaced G-sensor	95
6.4 I2C MUX Test	98
Chapter 7 Examples for using both HPS SoC and FGPA.....	101
7.1 HPS Control LED and HEX.....	101
7.2 DE1-SoC Control Panel	105
7.3 DE1-SoC Linux Frame Buffer Project	105
Chapter 8 Programming the EPCS Device	107
8.1 Before Programming Begins	107
8.2 Convert .SOF File to .JIC File.....	107
8.3 Write JIC File into the EPCS Device	112
8.4 Erase the EPCS Device	114
8.5 Nios II Boot from EPCS Device in Quartus II v16.0.....	115
Chapter 9 Appendix	116
9.1 Revision History.....	116
9.2 Copyright Statement.....	116

Chapter 1

DE1-SoC *Development Kit*

The DE1-SoC Development Kit presents a robust hardware design platform built around the Altera System-on-Chip (SoC) FPGA, which combines the latest dual-core Cortex-A9 embedded cores with industry-leading programmable logic for ultimate design flexibility. Users can now leverage the power of tremendous re-configurability paired with a high-performance, low-power processor system. Altera's SoC integrates an ARM-based hard processor system (HPS) consisting of processor, peripherals and memory interfaces tied seamlessly with the FPGA fabric using a high-bandwidth interconnect backbone. The DE1-SoC development board is equipped with high-speed DDR3 memory, video and audio capabilities, Ethernet networking, and much more that promise many exciting applications.

The DE1-SoC Development Kit contains all the tools needed to use the board in conjunction with a computer that runs the Microsoft Windows XP or later.

1.1 Package Contents

Figure 1-1 shows a photograph of the DE1-SoC package.

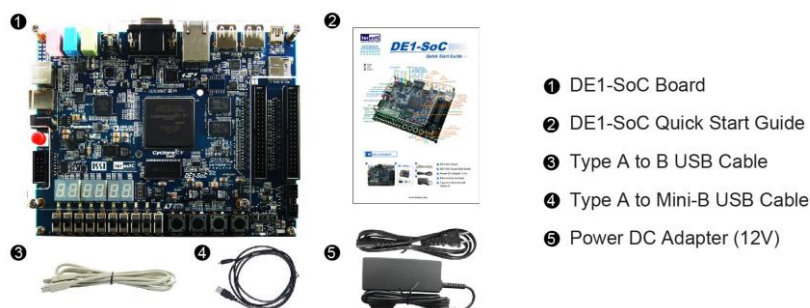


Figure 1-1 The DE1-SoC package contents

The DE1-SoC package includes:

- The DE1-SoC development board
- DE1-SoC Quick Start Guide
- USB cable (Type A to B) for FPGA programming and control
- USB cable (Type A to Mini-B) for UART control
- 12V DC power adapter

1.2 DE1-SoC System CD

The DE1-SoC System CD contains all the documents and supporting materials associated with DE1-SoC, including the user manual, system builder, reference designs, and device datasheets. Users can download this system CD from the link: <http://cd-de1-soc.terasic.com>.

1.3 Getting Help

Here are the addresses where you can get help if you encounter any problems:

- Altera Corporation
- 101 Innovation Drive San Jose, California, 95134 USA

Email: university@altera.com

- Terasic Technologies
- 9F., No.176, Sec.2, Gongdao 5th Rd, East Dist, Hsinchu City, 30070. Taiwan

Email: support@terasic.com

Tel.: +886-3-575-0880

Website: de1-soc.terasic.com

Chapter 2

Introduction of the DE1-SoC Board

This chapter provides an introduction to the features and design characteristics of the board.

2.1 Layout and Components

Figure 2-1 shows a photograph of the board. It depicts the layout of the board and indicates the location of the connectors and key components.

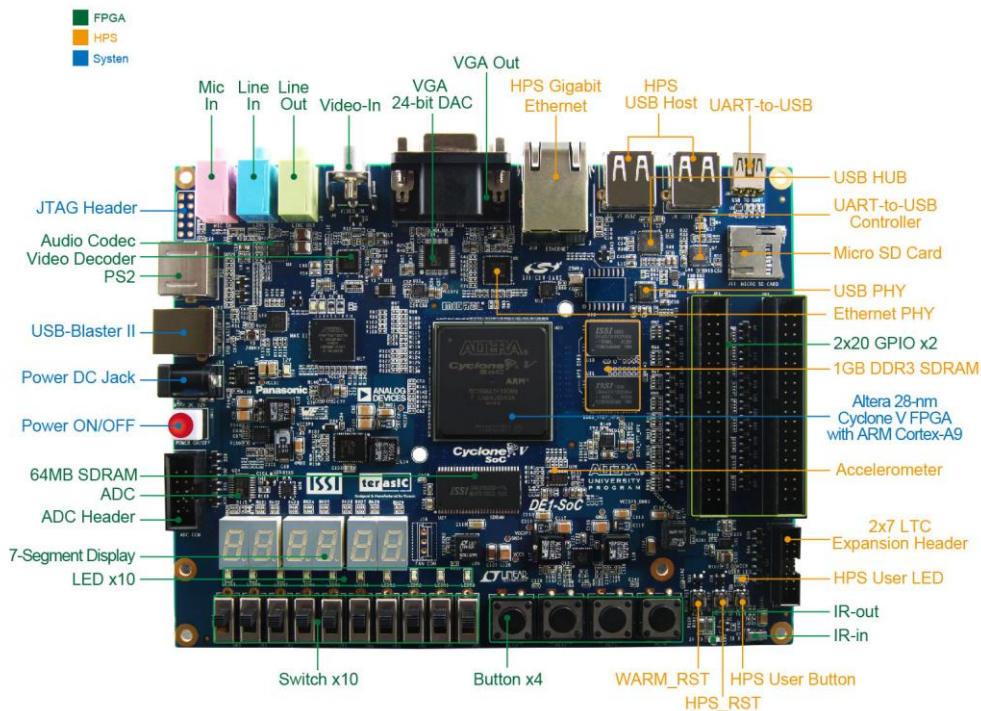


Figure 2-1 DE1-SoC development board (top view)

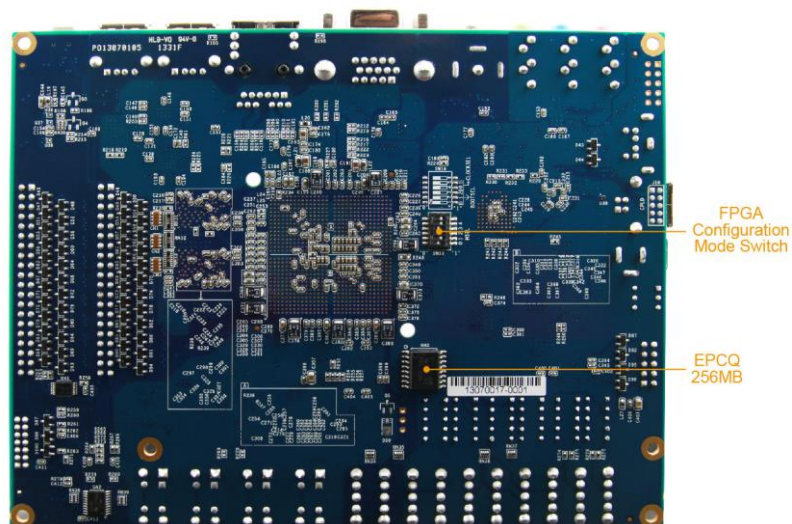


Figure 2-2 De1-SoC development board (bottom view)

The DE1-SoC board has many features that allow users to implement a wide range of designed circuits, from simple circuits to various multimedia projects.

The following hardware is provided on the board:

■ **FPGA**

- Altera Cyclone® V SE 5CSEMA5F31C6N device
- Altera serial configuration device – EPCS128
- USB-Blaster II onboard for programming; JTAG Mode
- 64MB SDRAM (16-bit data bus)
- 4 push-buttons
- 10 slide switches
- 10 red user LEDs
- Six 7-segment displays
- Four 50MHz clock sources from the clock generator
- 24-bit CD-quality audio CODEC with line-in, line-out, and microphone-in jacks
- VGA DAC (8-bit high-speed triple DACs) with VGA-out connector
- TV decoder (NTSC/PAL/SECAM) and TV-in connector
- PS/2 mouse/keyboard connector
- IR receiver and IR emitter
- Two 40-pin expansion header with diode protection
- A/D converter, 4-pin SPI interface with FPGA

■ **HPS (Hard Processor System)**

- 800MHz Dual-core ARM Cortex-A9 MPCore processor
- 1GB DDR3 SDRAM (32-bit data bus)
- 1 Gigabit Ethernet PHY with RJ45 connector
- 2-port USB Host, normal Type-A USB connector
- Micro SD card socket
- Accelerometer (I2C interface + interrupt)
- UART to USB, USB Mini-B connector
- Warm reset button and cold reset button
- One user button and one user LED
- LTC 2x7 expansion header

2.2 Block Diagram of the DE1-SoC Board

Figure 2-3 is the block diagram of the board. All the connections are established through the Cyclone V SoC FPGA device to provide maximum flexibility for users. Users can configure the FPGA to implement any system design.

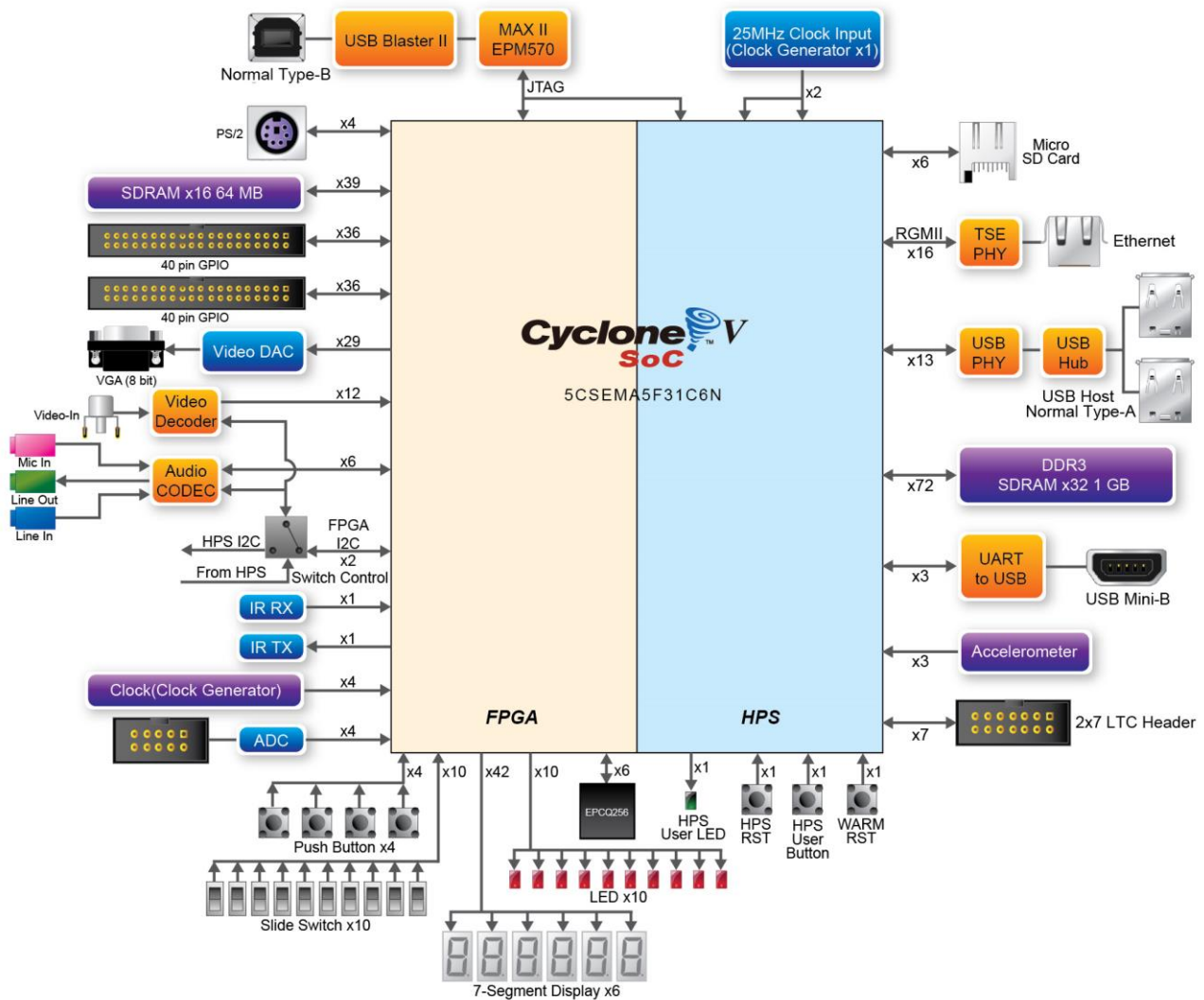


Figure 2-3 Block diagram of DE1-SoC

Detailed information about **Figure 2-3** are listed below.

FPGA Device

- Cyclone V SoC 5CSEMA5F31 Device
- Dual-core ARM Cortex-A9 (HPS)
- 85K programmable logic elements
- 4,450 Kbits embedded memory
- 6 fractional PLLs
- 2 hard memory controllers

Configuration and Debug

- Quad serial configuration device – EPCS128 on FPGA
- Onboard USB-Blaster II (normal type B USB connector)

Memory Device

- 64MB (32Mx16) SDRAM on FPGA
- 1GB (2x256Mx16) DDR3 SDRAM on HPS
- Micro SD card socket on HPS

Communication

- Two port USB 2.0 Host (ULPI interface with USB type A connector)
- UART to USB (USB Mini-B connector)
- 10/100/1000 Ethernet
- PS/2 mouse/keyboard
- IR emitter/receiver
- I2C multiplexer

Connectors

- Two 40-pin expansion headers
- One 10-pin ADC input header
- One LTC connector (one Serial Peripheral Interface (SPI) Master ,one I2C and one GPIO interface)

Display

- 24-bit VGA DAC

Audio

- 24-bit CODEC, Line-in, Line-out, and microphone-in jacks

Video Input

- TV decoder (NTSC/PAL/SECAM) and TV-in connector

ADC

- Interface: SPI
- Fast throughput rate: 500 KSPS
- Channel number: 8
- Resolution: 12-bit
- Analog input range : 0 ~ 4.096

Switches, Buttons, and Indicators

- 5 user Keys (FPGA x4, HPS x1)
- 10 user switches (FPGA x10)
- 11 user LEDs (FPGA x10, HPS x 1)
- 2 HPS reset buttons (HPS_RESET_n and HPS_WARM_RST_n)
- Six 7-segment displays

Sensors

- G-Sensor on HPS

Power

- 12V DC input

Chapter 3

Using the DE1-SoC Board

This chapter provides an instruction to use the board and describes the peripherals.

3.1 Settings of FPGA Configuration Mode

When the DE1-SoC board is powered on, the FPGA can be configured from EPCS or HPS. The MSEL[4:0] pins are used to select the configuration scheme. It is implemented as a 6-pin DIP switch **SW10** on the DE1-SoC board, as shown in **Figure 3-1**.

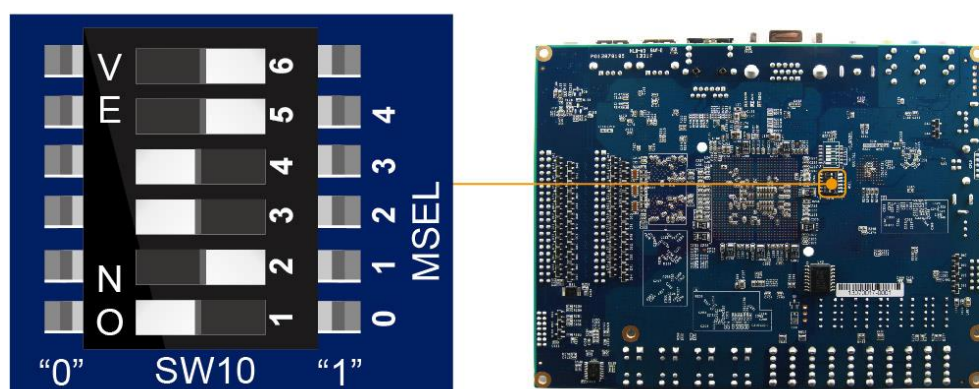


Figure 3-1 DIP switch (SW10) setting of Active Serial (AS) mode at the back of DE1-SoC board

Table 3-1 shows the relation between MSEL[4:0] and DIP switch (SW10).

Table 3-1 FPGA Configuration Mode Switch (SW10)

Board Reference	Signal Name	Description	Default
SW10.1	MSEL0	Use these pins to set the FPGA Configuration scheme	ON ("0")
SW10.2	MSEL1		OFF ("1")
SW10.3	MSEL2		ON ("0")
SW10.4	MSEL3		ON ("0")
SW10.5	MSEL4		OFF ("1")
SW10.6	N/A	N/A	N/A

Figure 3-1 shows MSEL[4:0] setting of AS mode, which is also the default setting on DE1-SoC. When the board is powered on, the FPGA is configured from EPCS, which is pre-programmed with the default code. If developers wish to reconfigure FPGA from an application software running on Linux, the MSEL[4:0] needs to be set to "01010" before the programming process begins. If developers using the "Linux Console with frame buffer" or "Linux LXDE Desktop" SD Card image, the MSEL[4:0] needs to be set to "00000" before the board is powered on.

Table 3-2 MSEL Pin Settings for FPGA Configure of DE1-SoC

MSEL[4:0]	Configure Scheme	Description
10010	AS	FPGA configured from EPCS (default)
01010	FPPx32	FPGA configured from HPS software: Linux
00000	FPPx16	FPGA configured from HPS software: U-Boot, with image stored on the SD card, like LXDE Desktop or console Linux with frame buffer edition.

3.2 Configuration of Cyclone V SoC FPGA on DE1-SoC

There are two types of programming method supported by DE1-SoC:

1. JTAG programming: It is named after the IEEE standards Joint Test Action Group.

The configuration bit stream is downloaded directly into the Cyclone V SoC FPGA. The FPGA will retain its current status as long as the power keeps applying to the board; the configuration information will be lost when the power is off.

2. AS programming: The other programming method is Active Serial configuration.

The configuration bit stream is downloaded into the quad serial configuration device (EPCS128), which provides non-volatile storage for the bit stream. The information is retained within EPCS128

even if the DE1-SoC board is turned off. When the board is powered on, the configuration data in the EPCS128 device is automatically loaded into the Cyclone V SoC FPGA.

■ JTAG Chain on DE1-SoC Board

The FPGA device can be configured through JTAG interface on DE1-SoC board, but the JTAG chain must form a closed loop, which allows Quartus II programmer to detect FPGA device.

Figure 3-2 illustrates the JTAG chain on DE1-SoC board.

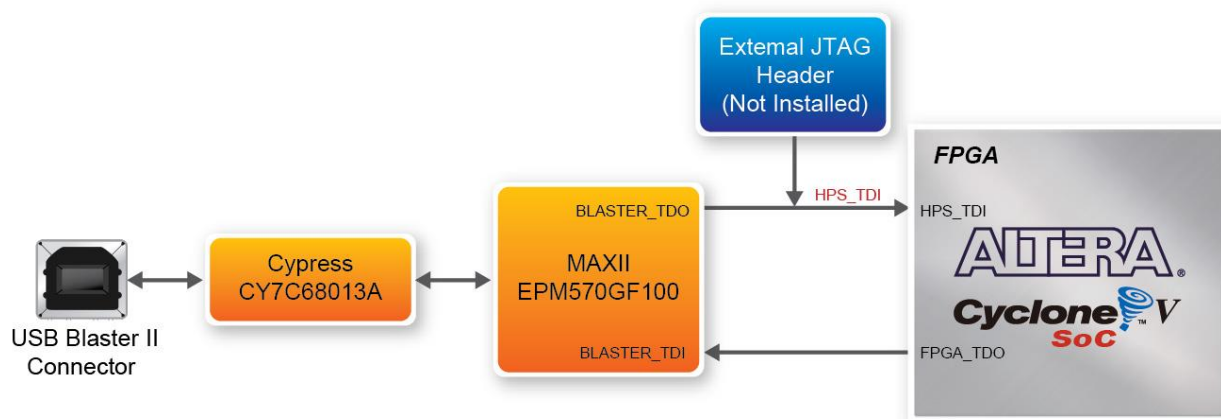


Figure 3-2 Path of the JTAG chain

■ Configure the FPGA in JTAG Mode

There are two devices (FPGA and HPS) on the JTAG chain. The following shows how the FPGA is programmed in JTAG mode step by step.

1. Open the Quartus II programmer and click “Auto Detect”, as circled in **Figure 3-3**

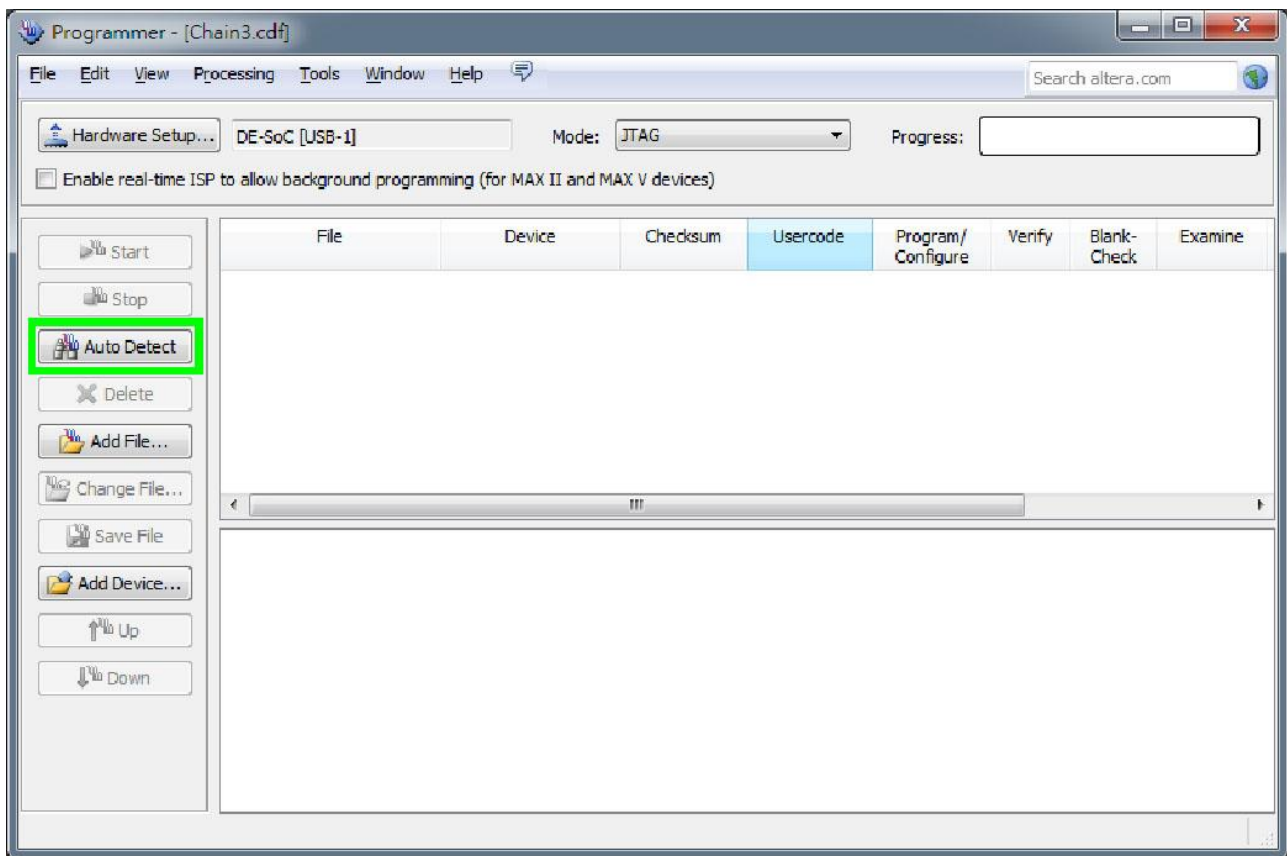


Figure 3-3 Detect FPGA device in JTAG mode

2. Select detected device associated with the board, as circled in **Figure 3-4**.

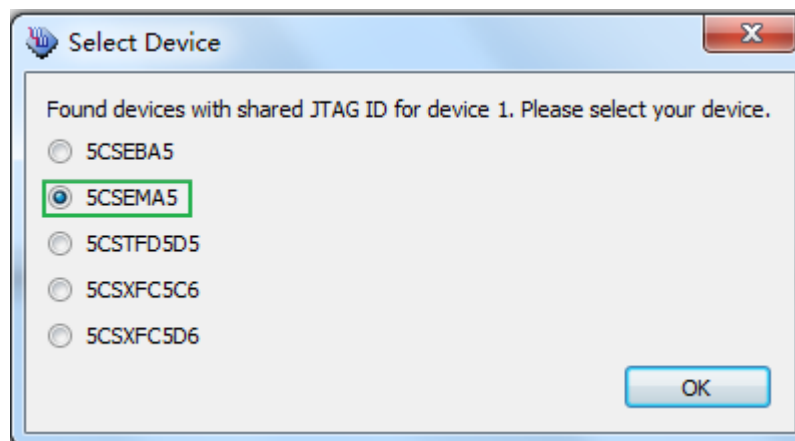


Figure 3-4 Select 5CSEMA5 device

3. Both FPGA and HPS are detected, as shown in **Figure 3-5**.

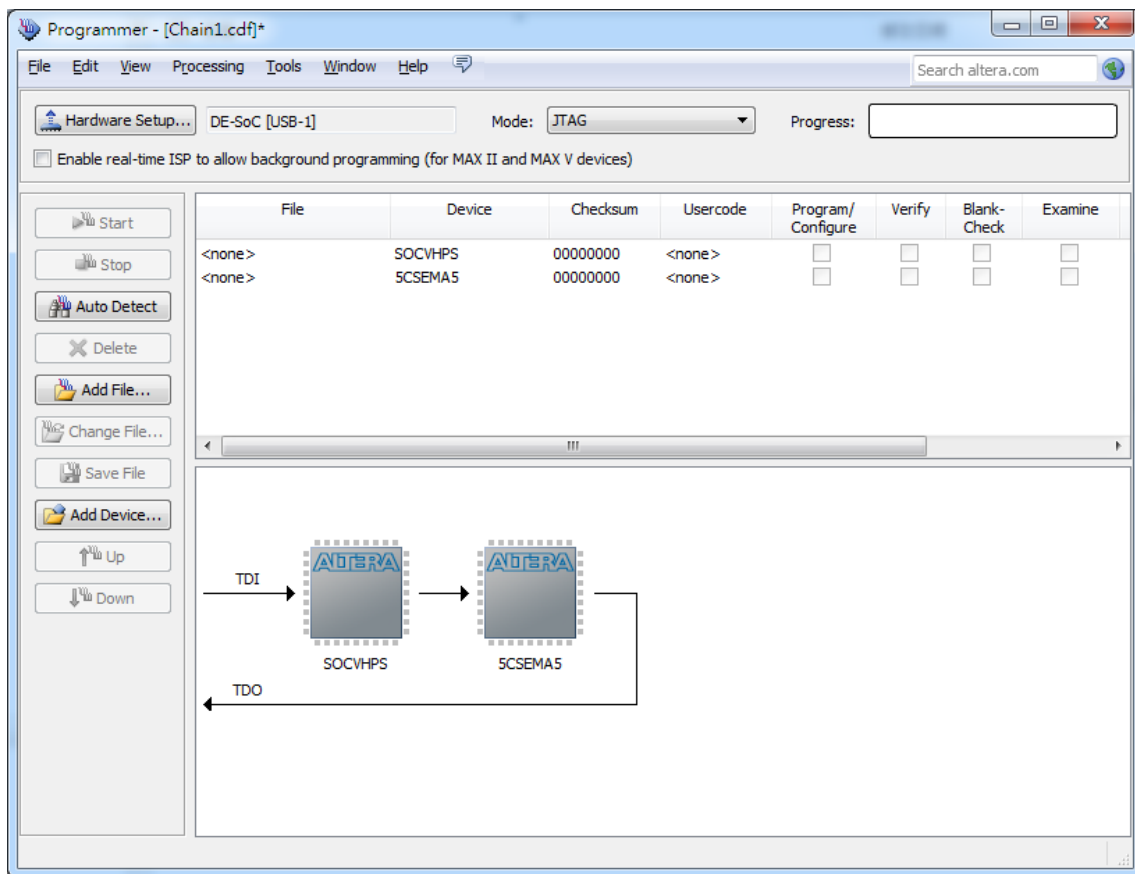


Figure 3-5 FPGA and HPS detected in Quartus programmer

4. Right click on the FPGA device and open the .sof file to be programmed, as highlighted in **Figure 3-6**.

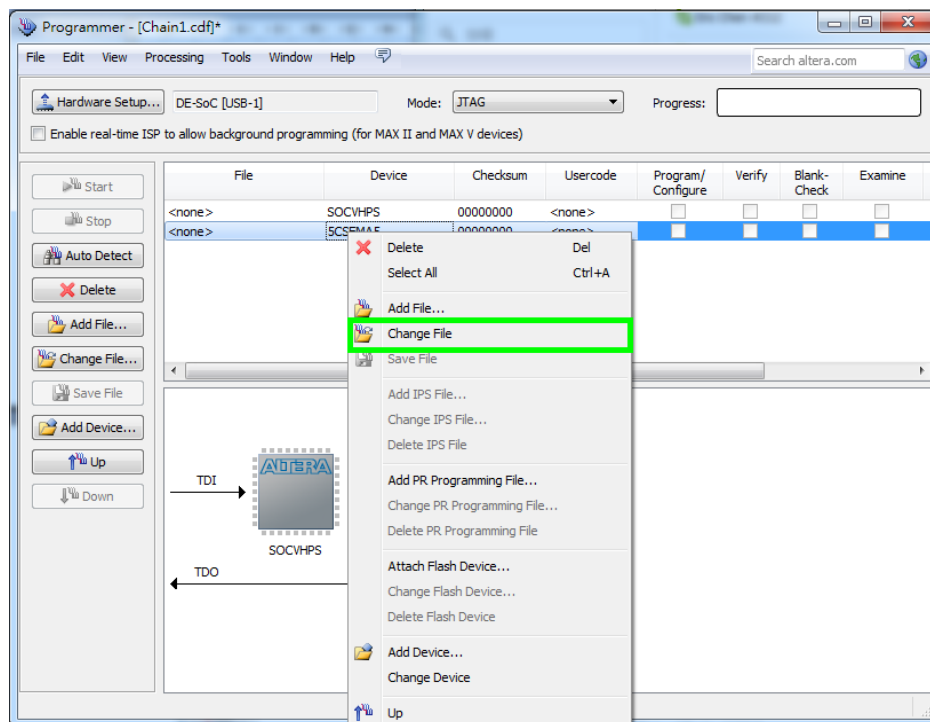


Figure 3-6 Open the .sof file to be programmed into the FPGA device

5. Select the .sof file to be programmed, as shown in **Figure 3-7**.

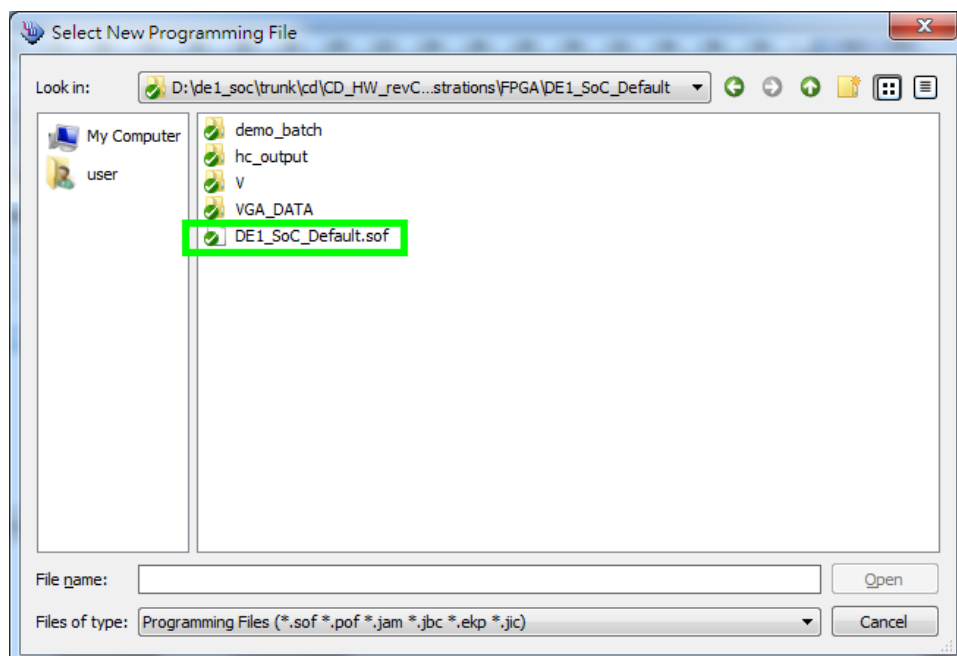


Figure 3-7 Select the .sof file to be programmed into the FPGA device

6. Click “Program/Configure” check box and then click “Start” button to download the .sof file into the FPGA device, as shown in **Figure 3-8**.

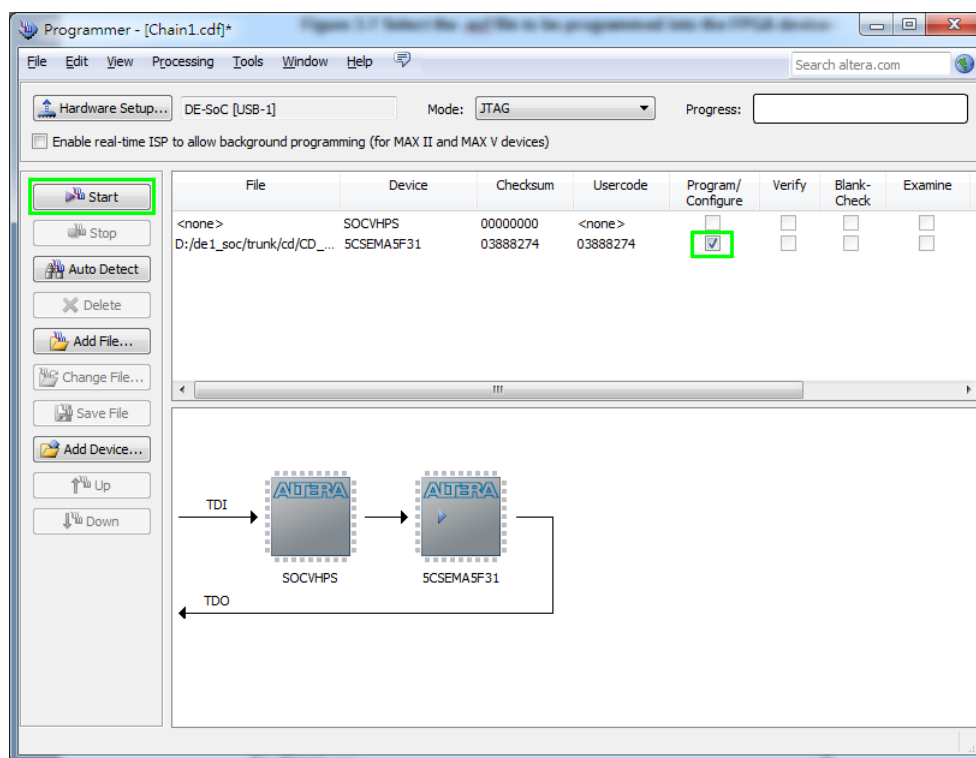


Figure 3-8 Program .sof file into the FPGA device

■ Configure the FPGA in AS Mode

- The DE1-SoC board uses a quad serial configuration device (EPCS128) to store configuration data for the Cyclone V SoC FPGA. This configuration data is automatically loaded from the quad serial configuration device chip into the FPGA when the board is powered up.
- Users need to use Serial Flash Loader (SFL) to program the quad serial configuration device via JTAG interface. The FPGA-based SFL is a soft intellectual property (IP) core within the FPGA that bridge the JTAG and Flash interfaces. The SFL Megafunction is available in Quartus II. **Figure 3-9** shows the programming method when adopting SFL solution.
- Please refer to Chapter 9: Steps of Programming the Quad Serial Configuration Device for the basic programming instruction on the serial configuration device.

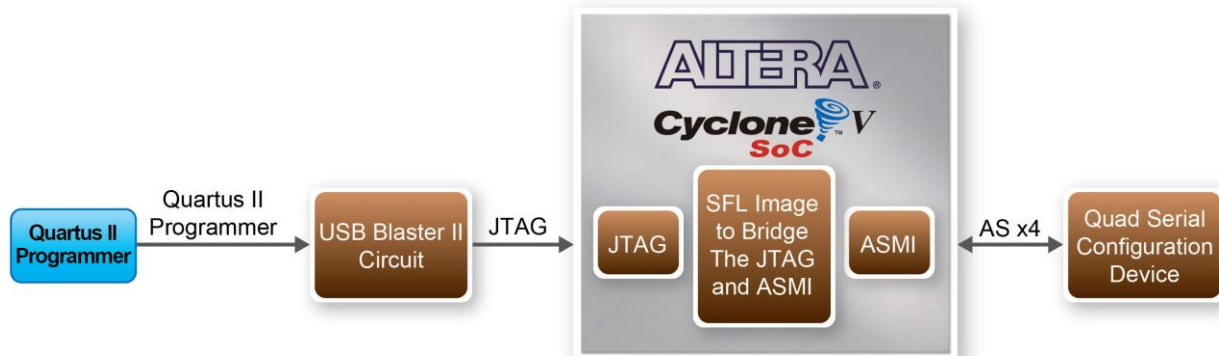


Figure 3-9 Programming a quad serial configuration device with SFL solution

3.3 Board Status Elements

In addition to the 10 LEDs that FPGA device can control, there are 5 indicators which can indicate the board status (See Figure 3-10), please refer the details in [Table 3-3](#)

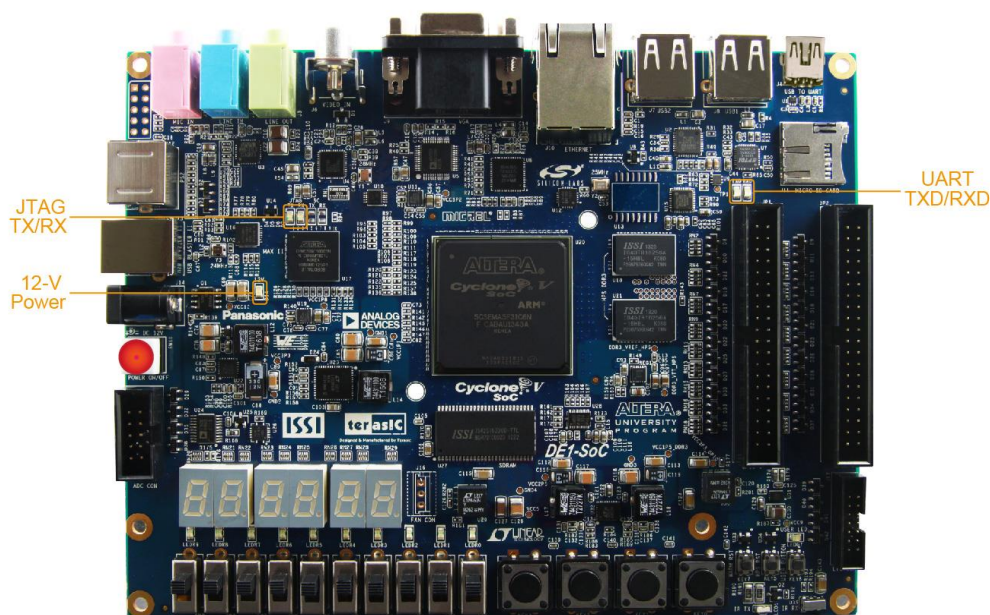


Figure 3-10 LED Indicators on DE1-SoC

Table 3-3 LED Indicators

Board Reference	LED Name	Description
D14	12-V Power	Illuminate when 12V power is active.
TXD	UART TXD	Illuminate when data is transferred from FT232R to USB Host.
RXD	UART RXD	Illuminate when data is transferred from USB Host to FT232R.
D5	JTAG_RX	Reserved
D4	JTAG_TX	

3.4 Board Reset Elements

There are two HPS reset buttons on DE1-SoC, HPS (cold) reset and HPS warm reset, as shown in **Figure 3-11**. **Table 3-4** describes the purpose of these two HPS reset buttons. **Figure 3-12** is the reset tree for DE1-SoC.

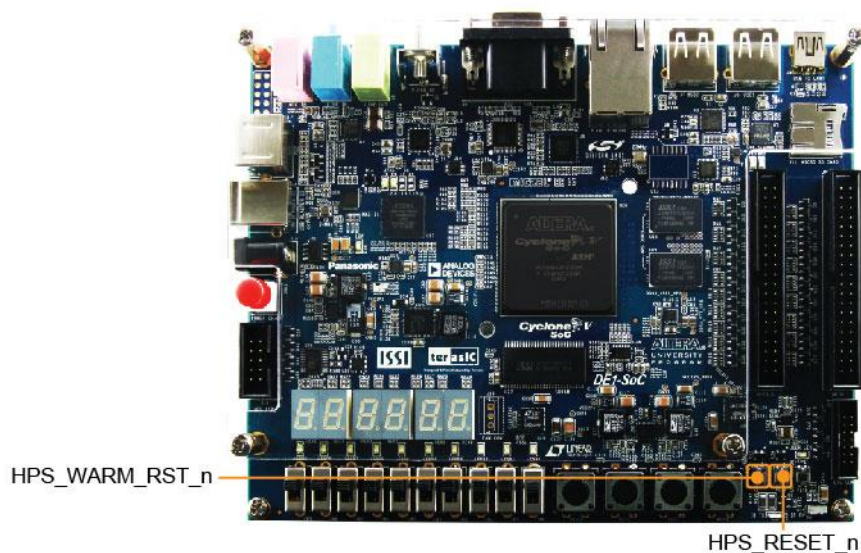


Figure 3-11 HPS cold reset and warm reset buttons on DE1-SoC

Table 3-4 Description of Two HPS Reset Buttons on DE1-SoC

Board Reference	Signal Name	Description
KEY5	HPS_RESET_N	Cold reset to the HPS, Ethernet PHY and USB host device. Active low input which resets all HPS logics that can be reset.
KEY7	HPS_WARM_RST_N	Warm reset to the HPS block. Active low input affects the system reset domain for debug purpose.

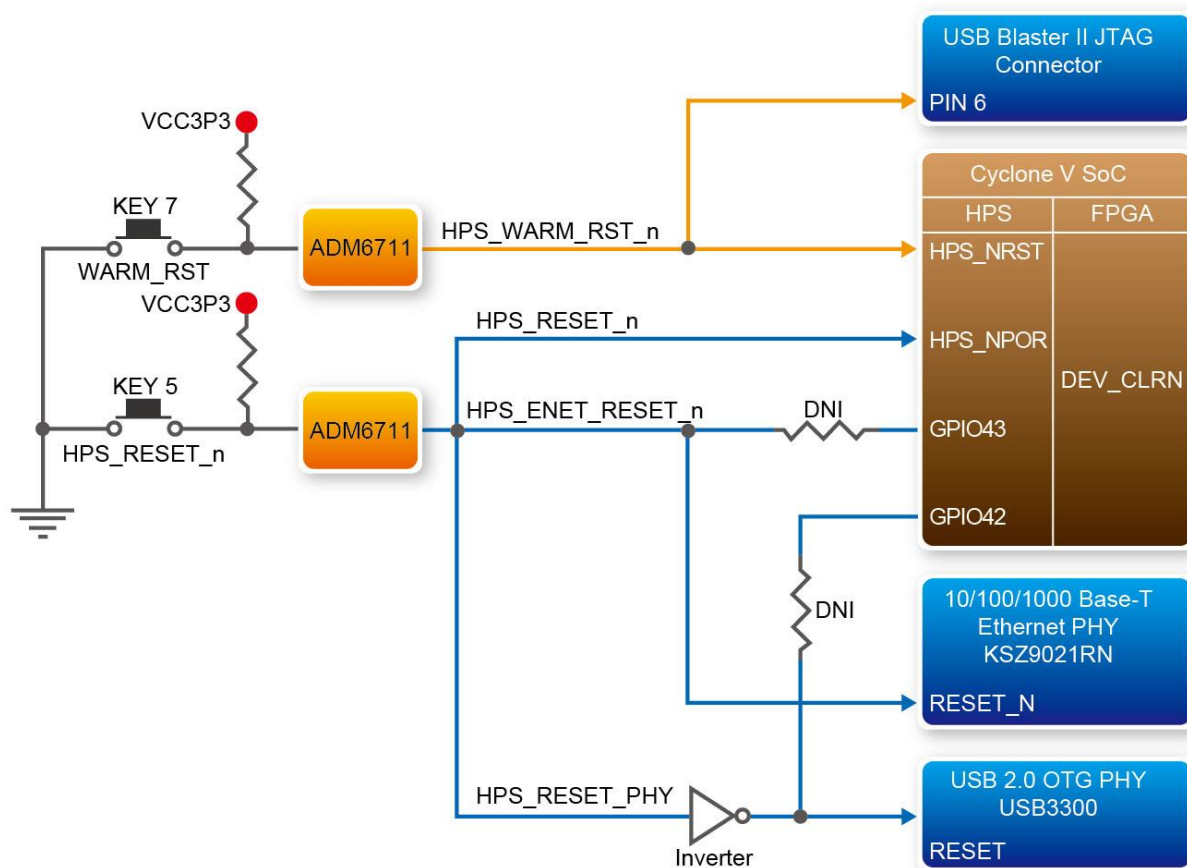


Figure 3-12 HPS reset tree on DE1-SoC board

3.5 Clock Circuitry

Figure 3-13 shows the default frequency of all external clocks to the Cyclone V SoC FPGA. A clock generator is used to distribute clock signals with low jitter. The four 50MHz clock signals connected to the FPGA are used as clock sources for user logic. One 25MHz clock signal is connected to two HPS clock inputs, and the other one is connected to the clock input of Gigabit

Ethernet Transceiver. Two 24MHz clock signals are connected to the clock inputs of USB Host/OTG PHY and USB hub controller. The associated pin assignment for clock inputs to FPGA I/O pins is listed in **Table 3-5**.

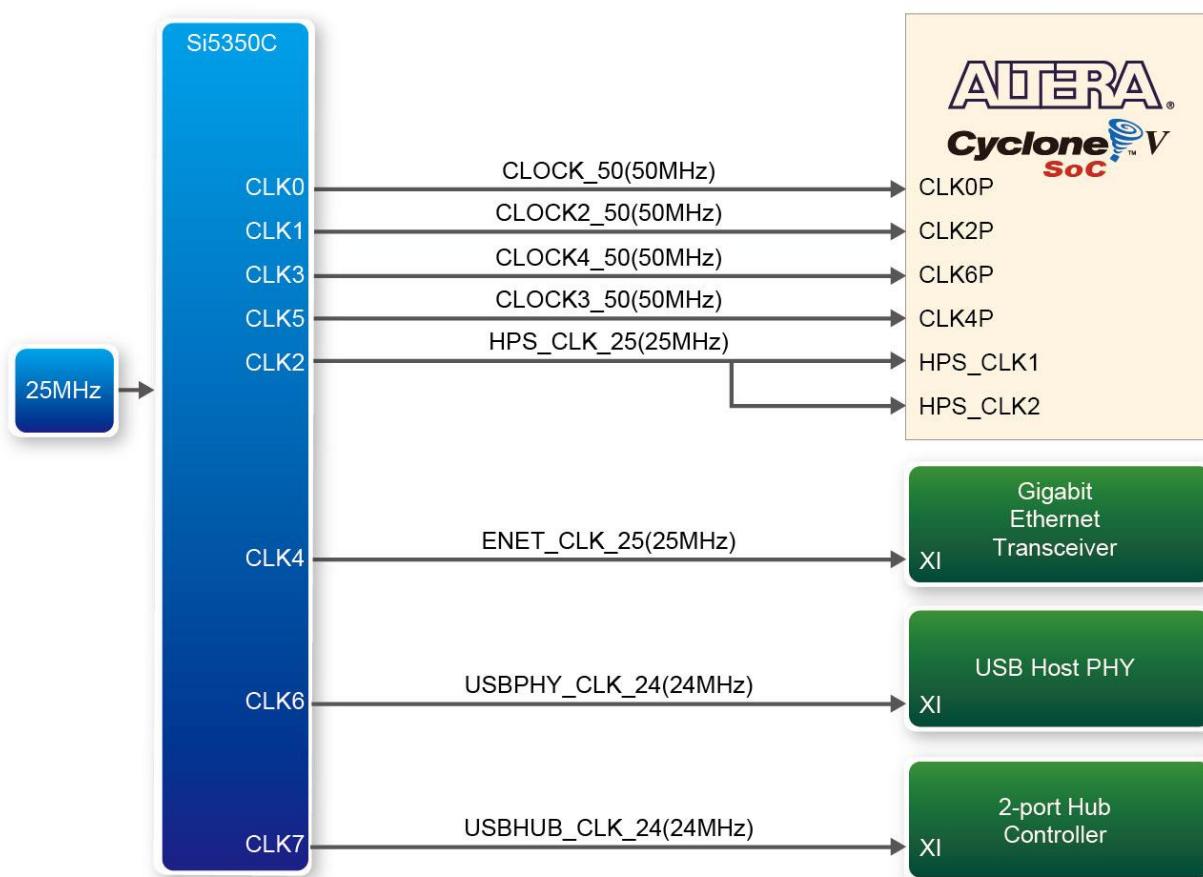


Figure 3-13 Block diagram of the clock distribution on DE1-SoC

Table 3-5 Pin Assignment of Clock Inputs

Signal Name	FPGA Pin No.	Description	I/O Standard
CLOCK_50	PIN_AF14	50 MHz clock input	3.3V
CLOCK2_50	PIN_AA16	50 MHz clock input	3.3V
CLOCK3_50	PIN_Y26	50 MHz clock input	3.3V
CLOCK4_50	PIN_K14	50 MHz clock input	3.3V
HPS_CLOCK1_25	PIN_D25	25 MHz clock input	3.3V
HPS_CLOCK2_25	PIN_F25	25 MHz clock input	3.3V

3.6 Peripherals Connected to the FPGA

This section describes the interfaces connected to the FPGA. Users can control or monitor different interfaces with user logic from the FPGA.

3.6.1 User Push-buttons, Switches and LEDs

The board has four push-buttons connected to the FPGA, as shown in **Figure 3-14** Connections between the push-buttons and the Cyclone V SoC FPGA. Schmitt trigger circuit is implemented and act as switch debounce in **Figure 3-15** for the push-buttons connected. The four push-buttons named KEY0, KEY1, KEY2, and KEY3 coming out of the Schmitt trigger device are connected directly to the Cyclone V SoC FPGA. The push-button generates a low logic level or high logic level when it is pressed or not, respectively. Since the push-buttons are debounced, they can be used as clock or reset inputs in a circuit.

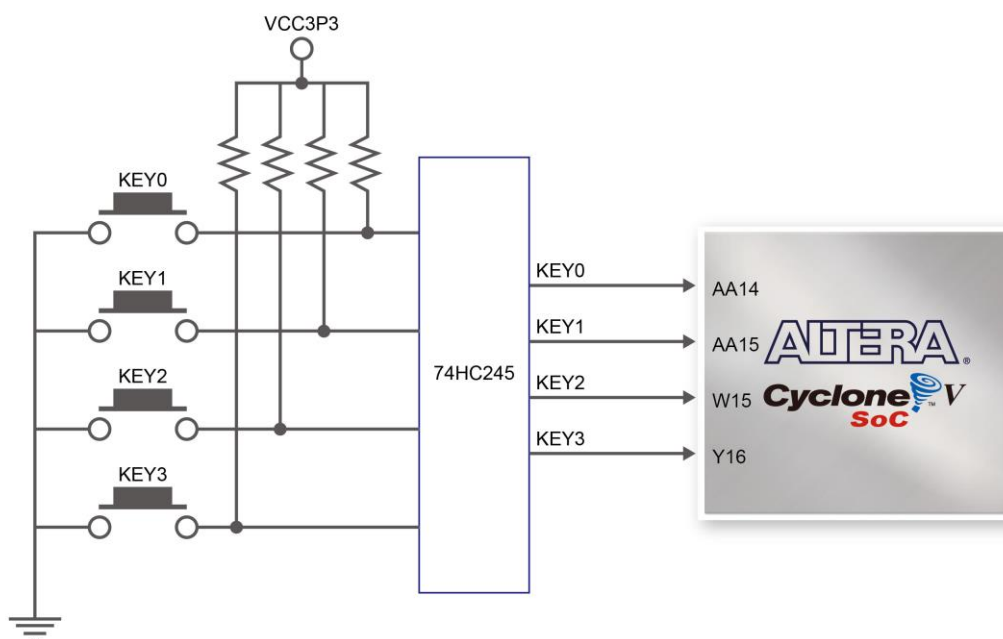


Figure 3-14 Connections between the push-buttons and the Cyclone V SoC FPGA

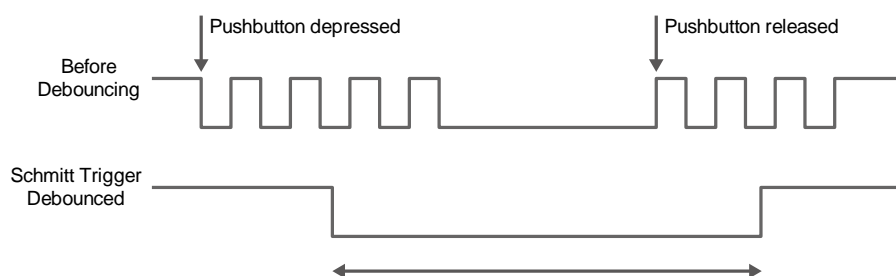


Figure 3-15 Switch debouncing

There are ten slide switches connected to the FPGA, as shown in **Figure 3-16**. These switches are not debounced and to be used as level-sensitive data inputs to a circuit. Each switch is connected directly and individually to the FPGA. When the switch is set to the DOWN position (towards the edge of the board), it generates a low logic level to the FPGA. When the switch is set to the UP position, a high logic level is generated to the FPGA.

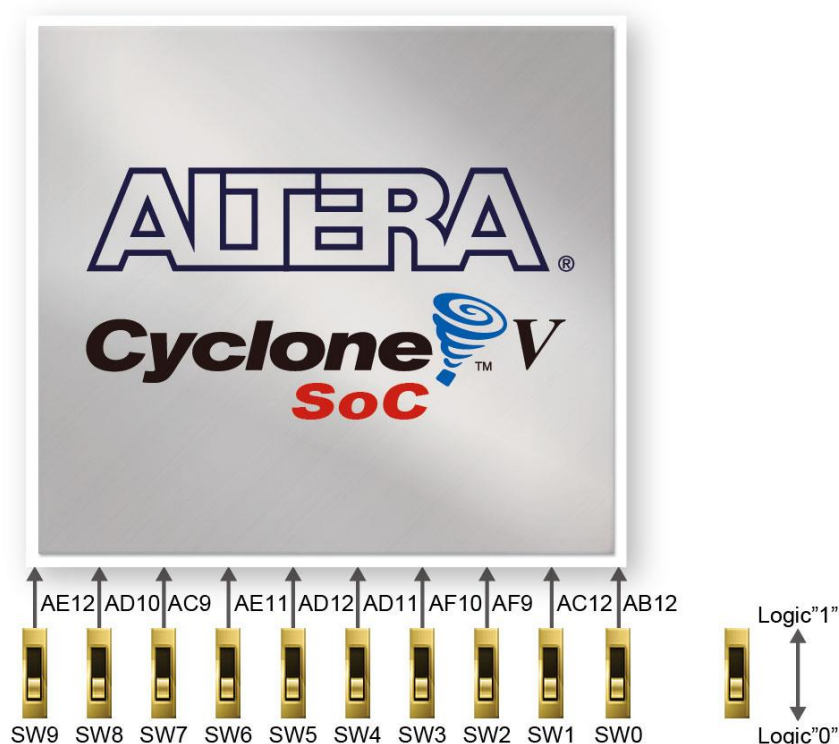


Figure 3-16 Connections between the slide switches and the Cyclone V SoC FPGA

There are also ten user-controllable LEDs connected to the FPGA. Each LED is driven directly and individually by the Cyclone V SoC FPGA; driving its associated pin to a high logic level or low

level to turn the LED on or off, respectively. **Figure 3-17** shows the connections between LEDs and Cyclone V SoC FPGA. **Table 3-6**, **Table 3-7** and **Table 3-8** list the pin assignment of user push-buttons, switches, and LEDs.



Figure 3-17 Connections between the LEDs and the Cyclone V SoC FPGA

Table 3-6 Pin Assignment of Slide Switches

Signal Name	FPGA Pin No.	Description	I/O Standard
SW[0]	PIN_AB12	Slide Switch[0]	3.3V
SW[1]	PIN_AC12	Slide Switch[1]	3.3V
SW[2]	PIN_AF9	Slide Switch[2]	3.3V
SW[3]	PIN_AF10	Slide Switch[3]	3.3V
SW[4]	PIN_AD11	Slide Switch[4]	3.3V
SW[5]	PIN_AD12	Slide Switch[5]	3.3V
SW[6]	PIN_AE11	Slide Switch[6]	3.3V
SW[7]	PIN_AC9	Slide Switch[7]	3.3V
SW[8]	PIN_AD10	Slide Switch[8]	3.3V
SW[9]	PIN_AE12	Slide Switch[9]	3.3V

Table 3-7 Pin Assignment of Push-buttons

Signal Name	FPGA Pin No.	Description	I/O Standard
KEY[0]	PIN_AA14	Push-button[0]	3.3V
KEY[1]	PIN_AA15	Push-button[1]	3.3V
KEY[2]	PIN_W15	Push-button[2]	3.3V
KEY[3]	PIN_Y16	Push-button[3]	3.3V

Table 3-8 Pin Assignment of LEDs

Signal Name	FPGA Pin No.	Description	I/O Standard
LEDR[0]	PIN_V16	LED [0]	3.3V
LEDR[1]	PIN_W16	LED [1]	3.3V
LEDR[2]	PIN_V17	LED [2]	3.3V
LEDR[3]	PIN_V18	LED [3]	3.3V
LEDR[4]	PIN_W17	LED [4]	3.3V
LEDR[5]	PIN_W19	LED [5]	3.3V
LEDR[6]	PIN_Y19	LED [6]	3.3V
LEDR[7]	PIN_W20	LED [7]	3.3V
LEDR[8]	PIN_W21	LED [8]	3.3V
LEDR[9]	PIN_Y21	LED [9]	3.3V

3.6.2 7-segment Displays

The DE1-SoC board has six 7-segment displays. These displays are paired to display numbers in various sizes. **Figure 3-18** shows the connection of seven segments (common anode) to pins on Cyclone V SoC FPGA. The segment can be turned on or off by applying a low logic level or high logic level from the FPGA, respectively.

Each segment in a display is indexed from 0 to 6, with corresponding positions given in **Figure 3-18**. **Table 3-9** shows the pin assignment of FPGA to the 7-segment displays.

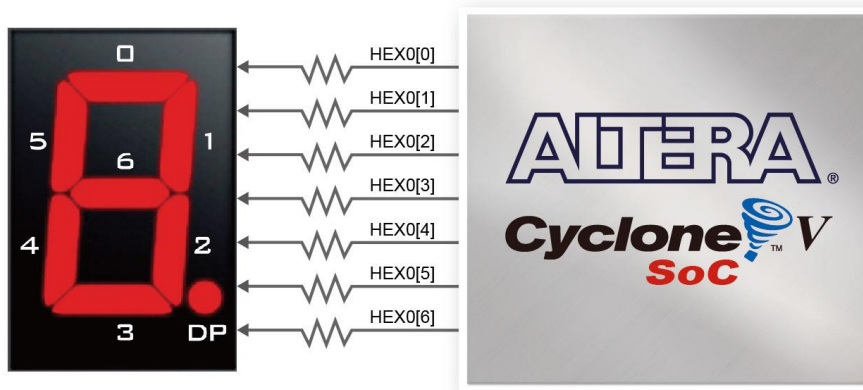


Figure 3-18 Connections between the 7-segment display HEX0 and the Cyclone V SoC FPGA

Table 3-9 Pin Assignment of 7-segment Displays

<i>Signal Name</i>	<i>FPGA Pin No.</i>	<i>Description</i>	<i>I/O Standard</i>
HEX0[0]	PIN_AE26	Seven Segment Digit 0[0]	3.3V
HEX0[1]	PIN_AE27	Seven Segment Digit 0[1]	3.3V
HEX0[2]	PIN_AE28	Seven Segment Digit 0[2]	3.3V
HEX0[3]	PIN_AG27	Seven Segment Digit 0[3]	3.3V
HEX0[4]	PIN_AF28	Seven Segment Digit 0[4]	3.3V
HEX0[5]	PIN_AG28	Seven Segment Digit 0[5]	3.3V
HEX0[6]	PIN_AH28	Seven Segment Digit 0[6]	3.3V
HEX1[0]	PIN_AJ29	Seven Segment Digit 1[0]	3.3V
HEX1[1]	PIN_AH29	Seven Segment Digit 1[1]	3.3V
HEX1[2]	PIN_AH30	Seven Segment Digit 1[2]	3.3V
HEX1[3]	PIN_AG30	Seven Segment Digit 1[3]	3.3V
HEX1[4]	PIN_AF29	Seven Segment Digit 1[4]	3.3V
HEX1[5]	PIN_AF30	Seven Segment Digit 1[5]	3.3V
HEX1[6]	PIN_AD27	Seven Segment Digit 1[6]	3.3V
HEX2[0]	PIN_AB23	Seven Segment Digit 2[0]	3.3V
HEX2[1]	PIN_AE29	Seven Segment Digit 2[1]	3.3V
HEX2[2]	PIN_AD29	Seven Segment Digit 2[2]	3.3V
HEX2[3]	PIN_AC28	Seven Segment Digit 2[3]	3.3V
HEX2[4]	PIN_AD30	Seven Segment Digit 2[4]	3.3V
HEX2[5]	PIN_AC29	Seven Segment Digit 2[5]	3.3V
HEX2[6]	PIN_AC30	Seven Segment Digit 2[6]	3.3V
HEX3[0]	PIN_AD26	Seven Segment Digit 3[0]	3.3V
HEX3[1]	PIN_AC27	Seven Segment Digit 3[1]	3.3V
HEX3[2]	PIN_AD25	Seven Segment Digit 3[2]	3.3V
HEX3[3]	PIN_AC25	Seven Segment Digit 3[3]	3.3V
HEX3[4]	PIN_AB28	Seven Segment Digit 3[4]	3.3V
HEX3[5]	PIN_AB25	Seven Segment Digit 3[5]	3.3V
HEX3[6]	PIN_AB22	Seven Segment Digit 3[6]	3.3V
HEX4[0]	PIN_AA24	Seven Segment Digit 4[0]	3.3V
HEX4[1]	PIN_Y23	Seven Segment Digit 4[1]	3.3V
HEX4[2]	PIN_Y24	Seven Segment Digit 4[2]	3.3V
HEX4[3]	PIN_W22	Seven Segment Digit 4[3]	3.3V
HEX4[4]	PIN_W24	Seven Segment Digit 4[4]	3.3V
HEX4[5]	PIN_V23	Seven Segment Digit 4[5]	3.3V
HEX4[6]	PIN_W25	Seven Segment Digit 4[6]	3.3V
HEX5[0]	PIN_V25	Seven Segment Digit 5[0]	3.3V
HEX5[1]	PIN_AA28	Seven Segment Digit 5[1]	3.3V
HEX5[2]	PIN_Y27	Seven Segment Digit 5[2]	3.3V
HEX5[3]	PIN_AB27	Seven Segment Digit 5[3]	3.3V
HEX5[4]	PIN_AB26	Seven Segment Digit 5[4]	3.3V
HEX5[5]	PIN_AA26	Seven Segment Digit 5[5]	3.3V
HEX5[6]	PIN_AA25	Seven Segment Digit 5[6]	3.3V

3.6.3 2x20 GPIO Expansion Headers

The board has two 40-pin expansion headers. Each header has 36 user pins connected directly to the Cyclone V SoC FPGA. It also comes with DC +5V (VCC5), DC +3.3V (VCC3P3), and two GND pins. The maximum power consumption allowed for a daughter card connected to one or two GPIO ports is shown in [Table 3-10](#).

Table 3-10 Voltage and Max. Current Limit of Expansion Header(s)

Supplied Voltage	Max. Current Limit
5V	1A
3.3V	1.5A

Each pin on the expansion headers is connected to two diodes and a resistor for protection against high or low voltage level. [Figure 3-19](#) shows the protection circuitry applied to all 2x36 data pins. [Table 3-11](#) shows the pin assignment of two GPIO headers.

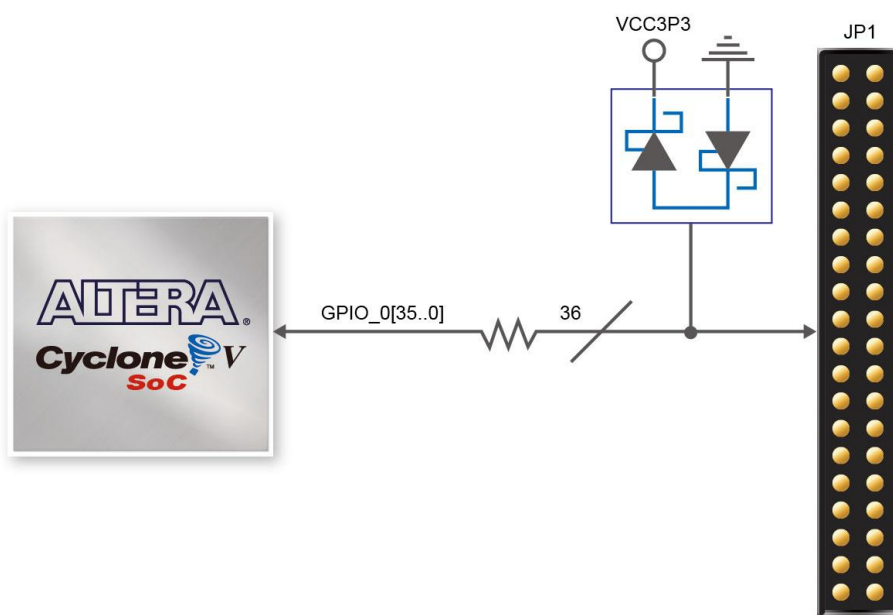


Figure 3-19 Connections between the GPIO header and Cyclone V SoC FPGA

Table 3-11 Pin Assignment of Expansion Headers

Signal Name	FPGA Pin No.	Description	I/O Standard
GPIO_0[0]	PIN_AC18	GPIO Connection 0[0]	3.3V
GPIO_0 [1]	PIN_Y17	GPIO Connection 0[1]	3.3V
GPIO_0 [2]	PIN_AD17	GPIO Connection 0[2]	3.3V
GPIO_0 [3]	PIN_Y18	GPIO Connection 0[3]	3.3V

GPIO_0 [4]	PIN_AK16	GPIO Connection 0[4]	3.3V
GPIO_0 [5]	PIN_AK18	GPIO Connection 0[5]	3.3V
GPIO_0 [6]	PIN_AK19	GPIO Connection 0[6]	3.3V
GPIO_0 [7]	PIN_AJ19	GPIO Connection 0[7]	3.3V
GPIO_0 [8]	PIN_AJ17	GPIO Connection 0[8]	3.3V
GPIO_0 [9]	PIN_AJ16	GPIO Connection 0[9]	3.3V
GPIO_0 [10]	PIN_AH18	GPIO Connection 0[10]	3.3V
GPIO_0 [11]	PIN_AH17	GPIO Connection 0[11]	3.3V
GPIO_0 [12]	PIN_AG16	GPIO Connection 0[12]	3.3V
GPIO_0 [13]	PIN_AE16	GPIO Connection 0[13]	3.3V
GPIO_0 [14]	PIN_AF16	GPIO Connection 0[14]	3.3V
GPIO_0 [15]	PIN_AG17	GPIO Connection 0[15]	3.3V
GPIO_0 [16]	PIN_AA18	GPIO Connection 0[16]	3.3V
GPIO_0 [17]	PIN_AA19	GPIO Connection 0[17]	3.3V
GPIO_0 [18]	PIN_AE17	GPIO Connection 0[18]	3.3V
GPIO_0 [19]	PIN_AC20	GPIO Connection 0[19]	3.3V
GPIO_0 [20]	PIN_AH19	GPIO Connection 0[20]	3.3V
GPIO_0 [21]	PIN_AJ20	GPIO Connection 0[21]	3.3V
GPIO_0 [22]	PIN_AH20	GPIO Connection 0[22]	3.3V
GPIO_0 [23]	PIN_AK21	GPIO Connection 0[23]	3.3V
GPIO_0 [24]	PIN_AD19	GPIO Connection 0[24]	3.3V
GPIO_0 [25]	PIN_AD20	GPIO Connection 0[25]	3.3V
GPIO_0 [26]	PIN_AE18	GPIO Connection 0[26]	3.3V
GPIO_0 [27]	PIN_AE19	GPIO Connection 0[27]	3.3V
GPIO_0 [28]	PIN_AF20	GPIO Connection 0[28]	3.3V
GPIO_0 [29]	PIN_AF21	GPIO Connection 0[29]	3.3V
GPIO_0 [30]	PIN_AF19	GPIO Connection 0[30]	3.3V
GPIO_0 [31]	PIN_AG21	GPIO Connection 0[31]	3.3V
GPIO_0 [32]	PIN_AF18	GPIO Connection 0[32]	3.3V
GPIO_0 [33]	PIN_AG20	GPIO Connection 0[33]	3.3V
GPIO_0 [34]	PIN_AG18	GPIO Connection 0[34]	3.3V
GPIO_0 [35]	PIN_AJ21	GPIO Connection 0[35]	3.3V
GPIO_1[0]	PIN_AB17	GPIO Connection 1[0]	3.3V
GPIO_1[1]	PIN_AA21	GPIO Connection 1[1]	3.3V
GPIO_1 [2]	PIN_AB21	GPIO Connection 1[2]	3.3V
GPIO_1 [3]	PIN_AC23	GPIO Connection 1[3]	3.3V
GPIO_1 [4]	PIN_AD24	GPIO Connection 1[4]	3.3V
GPIO_1 [5]	PIN_AE23	GPIO Connection 1[5]	3.3V
GPIO_1 [6]	PIN_AE24	GPIO Connection 1[6]	3.3V
GPIO_1 [7]	PIN_AF25	GPIO Connection 1[7]	3.3V
GPIO_1 [8]	PIN_AF26	GPIO Connection 1[8]	3.3V
GPIO_1 [9]	PIN_AG25	GPIO Connection 1[9]	3.3V
GPIO_1[10]	PIN_AG26	GPIO Connection 1[10]	3.3V
GPIO_1 [11]	PIN_AH24	GPIO Connection 1[11]	3.3V

GPIO_1 [12]	PIN_AH27	GPIO Connection 1[12]	3.3V
GPIO_1 [13]	PIN_AJ27	GPIO Connection 1[13]	3.3V
GPIO_1 [14]	PIN_AK29	GPIO Connection 1[14]	3.3V
GPIO_1 [15]	PIN_AK28	GPIO Connection 1[15]	3.3V
GPIO_1 [16]	PIN_AK27	GPIO Connection 1[16]	3.3V
GPIO_1 [17]	PIN_AJ26	GPIO Connection 1[17]	3.3V
GPIO_1 [18]	PIN_AK26	GPIO Connection 1[18]	3.3V
GPIO_1 [19]	PIN_AH25	GPIO Connection 1[19]	3.3V
GPIO_1 [20]	PIN_AJ25	GPIO Connection 1[20]	3.3V
GPIO_1 [21]	PIN_AJ24	GPIO Connection 1[21]	3.3V
GPIO_1 [22]	PIN_AK24	GPIO Connection 1[22]	3.3V
GPIO_1 [23]	PIN_AG23	GPIO Connection 1[23]	3.3V
GPIO_1 [24]	PIN_AK23	GPIO Connection 1[24]	3.3V
GPIO_1 [25]	PIN_AH23	GPIO Connection 1[25]	3.3V
GPIO_1 [26]	PIN_AK22	GPIO Connection 1[26]	3.3V
GPIO_1 [27]	PIN_AJ22	GPIO Connection 1[27]	3.3V
GPIO_1 [28]	PIN_AH22	GPIO Connection 1[28]	3.3V
GPIO_1 [29]	PIN_AG22	GPIO Connection 1[29]	3.3V
GPIO_1 [30]	PIN_AF24	GPIO Connection 1[30]	3.3V
GPIO_1 [31]	PIN_AF23	GPIO Connection 1[31]	3.3V
GPIO_1 [32]	PIN_AE22	GPIO Connection 1[32]	3.3V
GPIO_1 [33]	PIN_AD21	GPIO Connection 1[33]	3.3V
GPIO_1 [34]	PIN_AA20	GPIO Connection 1[34]	3.3V
GPIO_1 [35]	PIN_AC22	GPIO Connection 1[35]	3.3V

3.6.4 24-bit Audio CODEC

The DE1-SoC board offers high-quality 24-bit audio via the Wolfson WM8731 audio CODEC (Encoder/Decoder). This chip supports microphone-in, line-in, and line-out ports, with adjustable sample rate from 8 kHz to 96 kHz. The WM8731 is controlled via serial I2C bus, which is connected to HPS or Cyclone V SoC FPGA through an I2C multiplexer. The connection of the audio circuitry to the FPGA is shown in [Figure 3-20](#), and the associated pin assignment to the FPGA is listed in [Table 3-12](#). More information about the WM8731 codec is available in its datasheet, which can be found on the manufacturer's website, or in the directory \DE1_SOC_datasheets\Audio CODEC of DE1-SoC System CD.

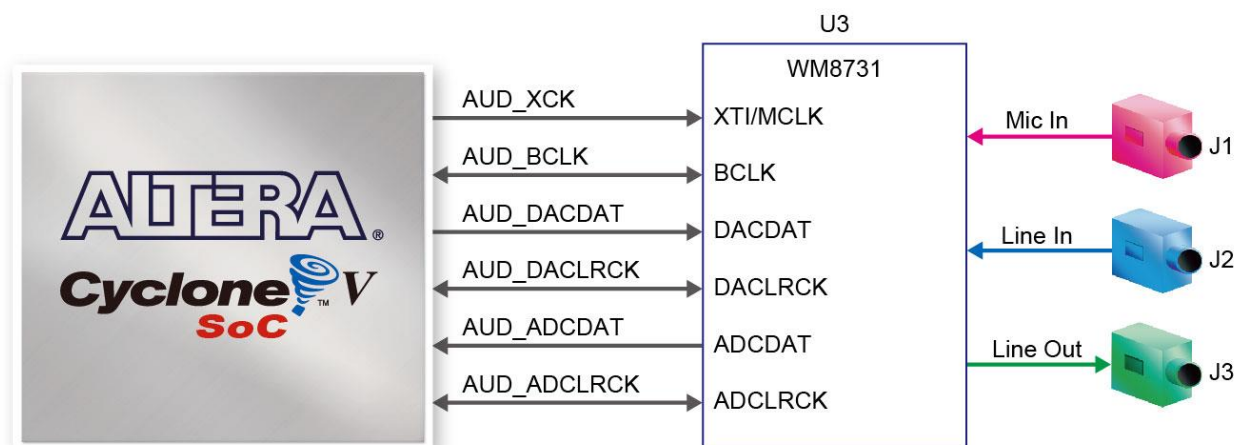


Figure 3-20 Connections between the FPGA and audio CODEC

Table 3-12 Pin Assignment of Audio CODEC

Signal Name	FPGA Pin No.	Description	I/O Standard
AUD_ADCLK	PIN_K8	Audio CODEC ADC LR Clock	3.3V
AUD_ADC DAT	PIN_K7	Audio CODEC ADC Data	3.3V
AUD_DACLK	PIN_H8	Audio CODEC DAC LR Clock	3.3V
AUD_DAC DAT	PIN_J7	Audio CODEC DAC Data	3.3V
AUD_XCK	PIN_G7	Audio CODEC Chip Clock	3.3V
AUD_BCLK	PIN_H7	Audio CODEC Bit-stream Clock	3.3V
I2C_SCLK	PIN_J12 or PIN_E23	I2C Clock	3.3V
I2C_SDAT	PIN_K12 or PIN_C24	I2C Data	3.3V

3.6.5 I2C Multiplexer

The DE1-SoC board implements an I2C multiplexer for HPS to access the I2C bus originally owned by FPGA. **Figure 3-21** shows the connection of I2C multiplexer to the FPGA and HPS. HPS can access Audio CODEC and TV Decoder if and only if the HPS_I2C_CONTROL signal is set to high. The pin assignment of I2C bus is listed in **Table 3-13**.

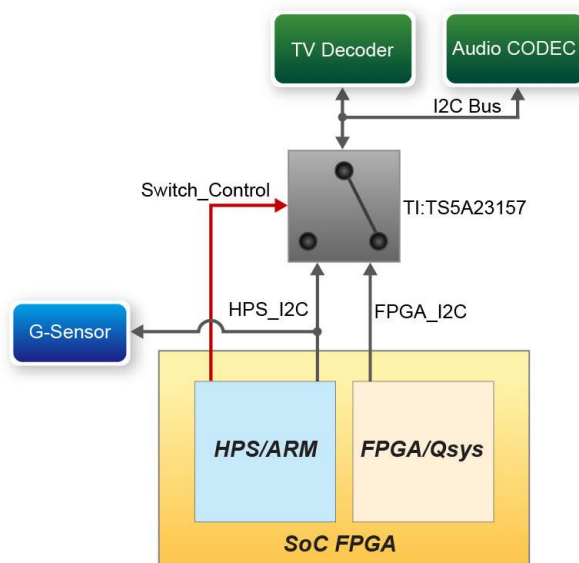


Figure 3-21 Control mechanism for the I2C multiplexer

Table 3-13 Pin Assignment of I2C Bus

Signal Name	FPGA Pin No.	Description	I/O Standard
FPGA_I2C_SCLK	PIN_J12	FPGA I2C Clock	3.3V
FPGA_I2C_SDAT	PIN_K12	FPGA I2C Data	3.3V
HPS_I2C1_SCLK	PIN_E23	I2C Clock of the first HPS I2C concontroller	3.3V
HPS_I2C1_SDAT	PIN_C24	I2C Data of the first HPS I2C concontroller	3.3V
HPS_I2C2_SCLK	PIN_H23	I2C Clock of the second HPS I2C concontroller	3.3V
HPS_I2C2_SDAT	PIN_A25	I2C Data of the second HPS I2C concontroller	3.3V

3.6.6 VGA

The DE1-SoC board has a 15-pin D-SUB connector populated for VGA output. The VGA synchronization signals are generated directly from the Cyclone V SoC FPGA, and the Analog Devices ADV7123 triple 10-bit high-speed video DAC (only the higher 8-bits are used) transforms signals from digital to analog to represent three fundamental colors (red, green, and blue). It can support up to SXGA standard (1280*1024) with signals transmitted at 100MHz. **Figure 3-22** shows the signals connected between the FPGA and VGA.

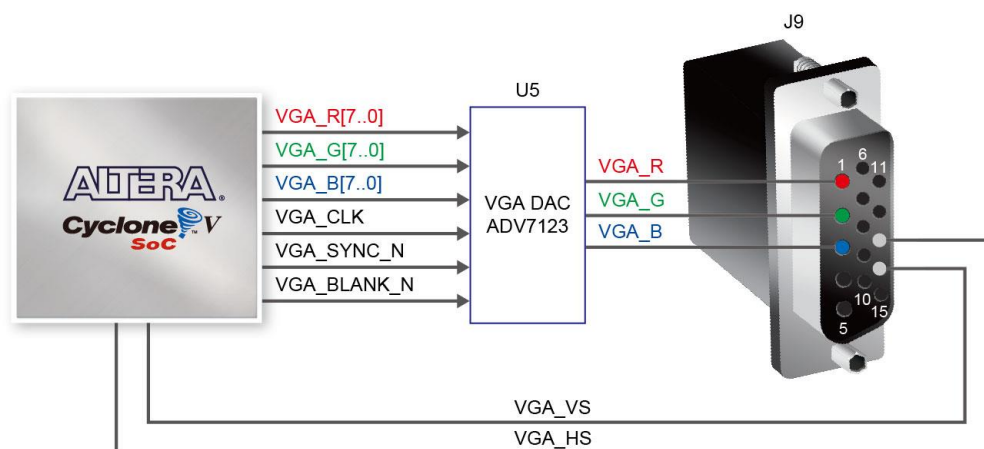


Figure 3-22 Connections between the FPGA and VGA

The timing specification for VGA synchronization and RGB (red, green, blue) data can be easily found on website nowadays. **Figure 3-22** illustrates the basic timing requirements for each row (horizontal) displayed on a VGA monitor. An active-low pulse of specific duration is applied to the horizontal synchronization (hsync) input of the monitor, which signifies the end of one row of data and the start of the next. The data (RGB) output to the monitor must be off (driven to 0 V) for a time period called the back porch (b) after the hsync pulse occurs, which is followed by the display interval (c). During the data display interval the RGB data drives each pixel in turn across the row being displayed. Finally, there is a time period called the front porch (d) where the RGB signals must again be off before the next hsync pulse can occur. The timing of vertical synchronization (vsync) is similar to the one shown in **Figure 3-23**, except that a vsync pulse signifies the end of one frame and the start of the next, and the data refers to the set of rows in the frame (horizontal timing). **Table 3-14** and **Table 3-15** show different resolutions and durations of time period a, b, c, and d for both horizontal and vertical timing.

More information about the ADV7123 video DAC is available in its datasheet, which can be found on the manufacturer's website, or in the directory \Datasheets\VIDEO DAC of DE1-SoC System CD. The pin assignment between the Cyclone V SoC FPGA and the ADV7123 is listed in **Table 3-16**.

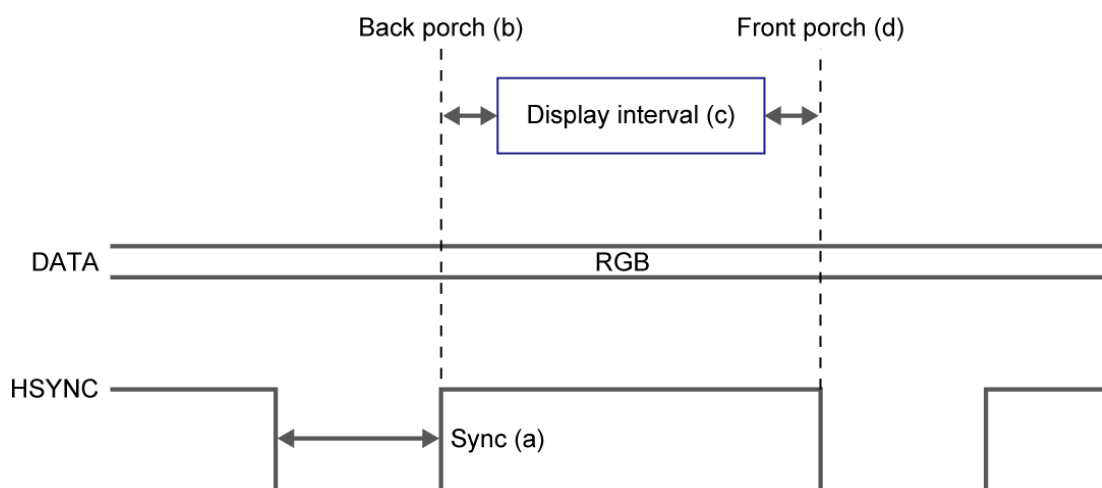


Figure 3-23 VGA horizontal timing specification

Table 3-14 VGA Horizontal Timing Specification

VGA mode		Horizontal Timing Spec				
Configuration	Resolution(HxV)	a(us)	b(us)	c(us)	d(us)	Pixel clock(MHz)
VGA(60Hz)	640x480	3.8	1.9	25.4	0.6	25
VGA(85Hz)	640x480	1.6	2.2	17.8	1.6	36
SVGA(60Hz)	800x600	3.2	2.2	20	1	40
SVGA(75Hz)	800x600	1.6	3.2	16.2	0.3	49
SVGA(85Hz)	800x600	1.1	2.7	14.2	0.6	56
XGA(60Hz)	1024x768	2.1	2.5	15.8	0.4	65
XGA(70Hz)	1024x768	1.8	1.9	13.7	0.3	75
XGA(85Hz)	1024x768	1.0	2.2	10.8	0.5	95
1280x1024(60Hz)	1280x1024	1.0	2.3	11.9	0.4	108

Table 3-15 VGA Vertical Timing Specification

VGA mode		Vertical Timing Spec				
Configuration	Resolution(HxV)	a(lines)	b(lines)	c(lines)	d(lines)	Pixel clock(MHz)
VGA(60Hz)	640x480	2	33	480	10	25
VGA(85Hz)	640x480	3	25	480	1	36
SVGA(60Hz)	800x600	4	23	600	1	40
SVGA(75Hz)	800x600	3	21	600	1	49
SVGA(85Hz)	800x600	3	27	600	1	56
XGA(60Hz)	1024x768	6	29	768	3	65
XGA(70Hz)	1024x768	6	29	768	3	75
XGA(85Hz)	1024x768	3	36	768	1	95
1280x1024(60Hz)	1280x1024	3	38	1024	1	108

Table 3-16 Pin Assignment of VGA

<i>Signal Name</i>	<i>FPGA Pin No.</i>	<i>Description</i>	<i>I/O Standard</i>
VGA_R[0]	PIN_A13	VGA Red[0]	3.3V
VGA_R[1]	PIN_C13	VGA Red[1]	3.3V
VGA_R[2]	PIN_E13	VGA Red[2]	3.3V
VGA_R[3]	PIN_B12	VGA Red[3]	3.3V
VGA_R[4]	PIN_C12	VGA Red[4]	3.3V
VGA_R[5]	PIN_D12	VGA Red[5]	3.3V
VGA_R[6]	PIN_E12	VGA Red[6]	3.3V
VGA_R[7]	PIN_F13	VGA Red[7]	3.3V
VGA_G[0]	PIN_J9	VGA Green[0]	3.3V
VGA_G[1]	PIN_J10	VGA Green[1]	3.3V
VGA_G[2]	PIN_H12	VGA Green[2]	3.3V
VGA_G[3]	PIN_G10	VGA Green[3]	3.3V
VGA_G[4]	PIN_G11	VGA Green[4]	3.3V
VGA_G[5]	PIN_G12	VGA Green[5]	3.3V
VGA_G[6]	PIN_F11	VGA Green[6]	3.3V
VGA_G[7]	PIN_E11	VGA Green[7]	3.3V
VGA_B[0]	PIN_B13	VGA Blue[0]	3.3V
VGA_B[1]	PIN_G13	VGA Blue[1]	3.3V
VGA_B[2]	PIN_H13	VGA Blue[2]	3.3V
VGA_B[3]	PIN_F14	VGA Blue[3]	3.3V
VGA_B[4]	PIN_H14	VGA Blue[4]	3.3V
VGA_B[5]	PIN_F15	VGA Blue[5]	3.3V
VGA_B[6]	PIN_G15	VGA Blue[6]	3.3V
VGA_B[7]	PIN_J14	VGA Blue[7]	3.3V
VGA_CLK	PIN_A11	VGA Clock	3.3V
VGA_BLANK_N	PIN_F10	VGA BLANK	3.3V
VGA_HS	PIN_B11	VGA H_SYNC	3.3V
VGA_VS	PIN_D11	VGA V_SYNC	3.3V
VGA_SYNC_N	PIN_C10	VGA SYNC	3.3V

3.6.7 TV Decoder

The DE1-SoC board is equipped with an Analog Device ADV7180 TV decoder chip. The ADV7180 is an integrated video decoder which automatically detects and converts a standard analog baseband television signals (NTSC, PAL, and SECAM) into 4:2:2 component video data, which is compatible with the 8-bit ITU-R BT.656 interface standard. The ADV7180 is compatible with wide range of video devices, including DVD players, tape-based sources, broadcast sources, and security/surveillance cameras.

The registers in the TV decoder can be accessed and set through serial I2C bus by the Cyclone V SoC FPGA or HPS. Note that the I2C address W/R of the TV decoder (U4) is 0x40/0x41. The pin assignment of TV decoder is listed in **Table 3-17**. More information about the ADV7180 is available on the manufacturer's website, or in the directory \DE1_SOC_datasheets\Video Decoder of DE1-SoC System CD.

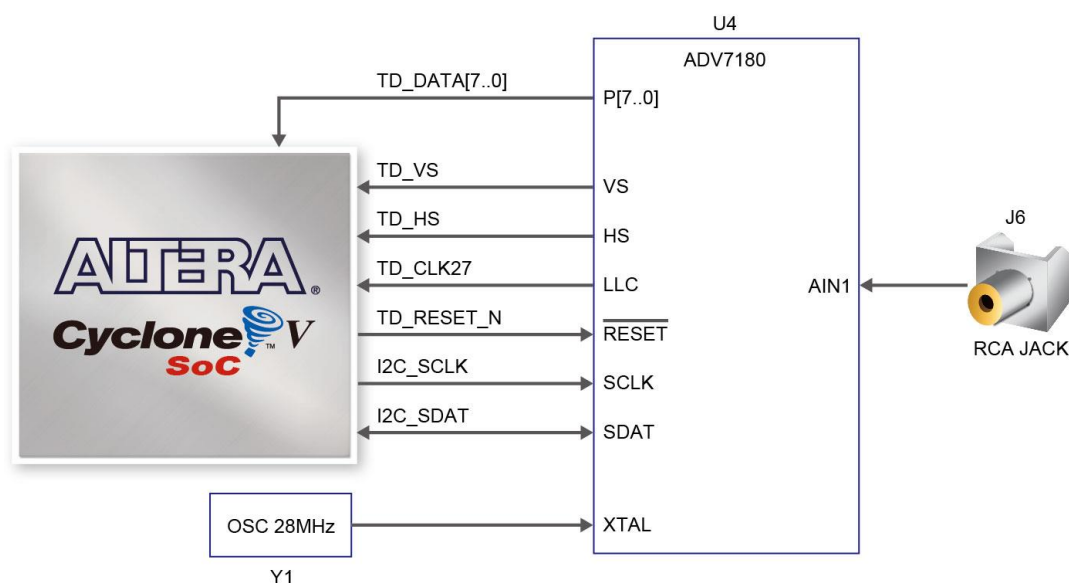


Figure 3-24 Connections between the FPGA and TV Decoder

Table 3-17 Pin Assignment of TV Decoder

Signal Name	FPGA Pin No.	Description	I/O Standard
TD_DATA [0]	PIN_D2	TV Decoder Data[0]	3.3V
TD_DATA [1]	PIN_B1	TV Decoder Data[1]	3.3V
TD_DATA [2]	PIN_E2	TV Decoder Data[2]	3.3V
TD_DATA [3]	PIN_B2	TV Decoder Data[3]	3.3V
TD_DATA [4]	PIN_D1	TV Decoder Data[4]	3.3V
TD_DATA [5]	PIN_E1	TV Decoder Data[5]	3.3V
TD_DATA [6]	PIN_C2	TV Decoder Data[6]	3.3V
TD_DATA [7]	PIN_B3	TV Decoder Data[7]	3.3V
TD_HS	PIN_A5	TV Decoder H_SYNC	3.3V
TD_VS	PIN_A3	TV Decoder V_SYNC	3.3V
TD_CLK27	PIN_H15	TV Decoder Clock Input.	3.3V
TD_RESET_N	PIN_F6	TV Decoder Reset	3.3V
I2C_SCLK	PIN_J12 or PIN_E23	I2C Clock	3.3V
I2C_SDAT	PIN_K12 or PIN_C24	I2C Data	3.3V

3.6.8 IR Receiver

The board comes with an infrared remote-control receiver module (model: IRM-V538/TR1), whose datasheet is provided in the directory \Datasheets\ IR Receiver and Emitter of DE1-SoC system CD. The remote control, which is optional and can be ordered from the website, has an encoding chip (uPD6121G) built-in for generating infrared signals. **Figure 3-25** shows the connection of IR receiver to the FPGA. **Table 3-18** shows the pin assignment of IR receiver to the FPGA.

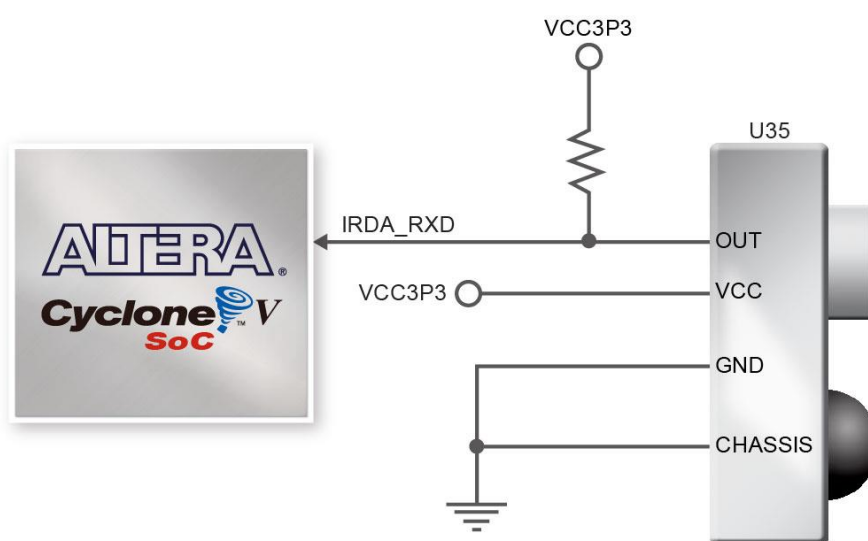


Figure 3-25 Connection between the FPGA and IR Receiver

Table 3-18 Pin Assignment of IR Receiver

Signal Name	FPGA Pin No.	Description	I/O Standard
IRDA_RXD	PIN_ AA30	IR Receiver	3.3V

3.6.9 IR Emitter LED

The board has an IR emitter LED for IR communication, which is widely used for operating television device wirelessly from a short line-of-sight distance. It can also be used to communicate with other systems by matching this IR emitter LED with another IR receiver on the other side. **Figure 3-26** shows the connection of IR emitter LED to the FPGA. **Table 3-19** shows the pin assignment of IR emitter LED to the FPGA.

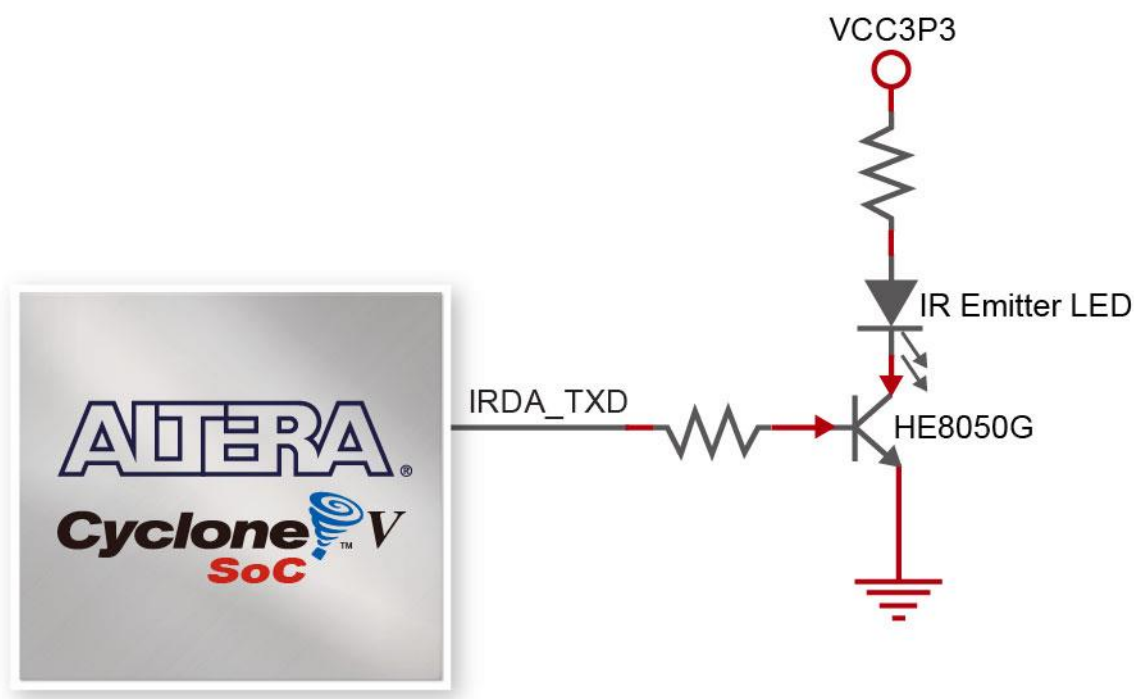


Figure 3-26 Connection between the FPGA and IR emitter LED

Table 3-19 Pin Assignment of IR Emitter LED

Signal Name	FPGA Pin No.	Description	I/O Standard
IRDA_TXD	PIN_ AB30	IR Emitter	3.3V

3.6.10 SDRAM Memory

The board features 64MB of SDRAM with a single 64MB (32Mx16) SDRAM chip. The chip consists of 16-bit data line, control line, and address line connected to the FPGA. This chip uses the 3.3V LVCMOS signaling standard. Connections between the FPGA and SDRAM are shown in [Figure 3-27](#), and the pin assignment is listed in [Table 3-20](#).

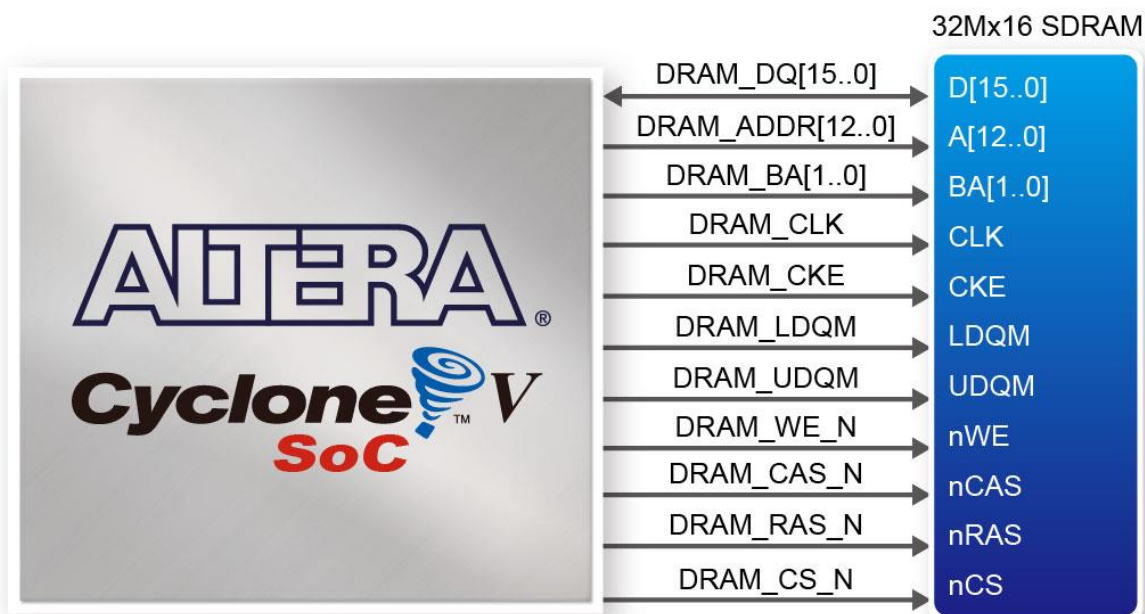


Figure 3-27 Connections between the FPGA and SDRAM

Table 3-20 Pin Assignment of SDRAM

Signal Name	FPGA Pin No.	Description	I/O Standard
DRAM_ADDR[0]	PIN_AK14	SDRAM Address[0]	3.3V
DRAM_ADDR[1]	PIN_AH14	SDRAM Address[1]	3.3V
DRAM_ADDR[2]	PIN_AG15	SDRAM Address[2]	3.3V
DRAM_ADDR[3]	PIN_AE14	SDRAM Address[3]	3.3V
DRAM_ADDR[4]	PIN_AB15	SDRAM Address[4]	3.3V
DRAM_ADDR[5]	PIN_AC14	SDRAM Address[5]	3.3V
DRAM_ADDR[6]	PIN_AD14	SDRAM Address[6]	3.3V
DRAM_ADDR[7]	PIN_AF15	SDRAM Address[7]	3.3V
DRAM_ADDR[8]	PIN_AH15	SDRAM Address[8]	3.3V
DRAM_ADDR[9]	PIN_AG13	SDRAM Address[9]	3.3V
DRAM_ADDR[10]	PIN_AG12	SDRAM Address[10]	3.3V
DRAM_ADDR[11]	PIN_AH13	SDRAM Address[11]	3.3V
DRAM_ADDR[12]	PIN_AJ14	SDRAM Address[12]	3.3V
DRAM_DQ[0]	PIN_AK6	SDRAM Data[0]	3.3V
DRAM_DQ[1]	PIN_AJ7	SDRAM Data[1]	3.3V
DRAM_DQ[2]	PIN_AK7	SDRAM Data[2]	3.3V
DRAM_DQ[3]	PIN_AK8	SDRAM Data[3]	3.3V
DRAM_DQ[4]	PIN_AK9	SDRAM Data[4]	3.3V
DRAM_DQ[5]	PIN_AG10	SDRAM Data[5]	3.3V

DRAM_DQ[6]	PIN_AK11	SDRAM Data[6]	3.3V
DRAM_DQ[7]	PIN_AJ11	SDRAM Data[7]	3.3V
DRAM_DQ[8]	PIN_AH10	SDRAM Data[8]	3.3V
DRAM_DQ[9]	PIN_AJ10	SDRAM Data[9]	3.3V
DRAM_DQ[10]	PIN_AJ9	SDRAM Data[10]	3.3V
DRAM_DQ[11]	PIN_AH9	SDRAM Data[11]	3.3V
DRAM_DQ[12]	PIN_AH8	SDRAM Data[12]	3.3V
DRAM_DQ[13]	PIN_AH7	SDRAM Data[13]	3.3V
DRAM_DQ[14]	PIN_AJ6	SDRAM Data[14]	3.3V
DRAM_DQ[15]	PIN_AJ5	SDRAM Data[15]	3.3V
DRAM_BA[0]	PIN_AF13	SDRAM Bank Address[0]	3.3V
DRAM_BA[1]	PIN_AJ12	SDRAM Bank Address[1]	3.3V
DRAM_LDQM	PIN_AB13	SDRAM byte Data Mask[0]	3.3V
DRAM_UDQM	PIN_AK12	SDRAM byte Data Mask[1]	3.3V
DRAM_RAS_N	PIN_AE13	SDRAM Row Address Strobe	3.3V
DRAM_CAS_N	PIN_AF11	SDRAM Column Address Strobe	3.3V
DRAM_CKE	PIN_AK13	SDRAM Clock Enable	3.3V
DRAM_CLK	PIN_AH12	SDRAM Clock	3.3V
DRAM_WE_N	PIN_AA13	SDRAM Write Enable	3.3V
DRAM_CS_N	PIN_AG11	SDRAM Chip Select	3.3V

3.6.11 PS/2 Serial Port

The DE1-SoC board comes with a standard PS/2 interface and a connector for a PS/2 keyboard or mouse. **Figure 3-28** shows the connection of PS/2 circuit to the FPGA. Users can use the PS/2 keyboard and mouse on the DE1-SoC board simultaneously by a PS/2 Y-Cable, as shown in **Figure 3-29**. Instructions on how to use PS/2 mouse and/or keyboard can be found on various educational websites. The pin assignment associated to this interface is shown in **Table 3-21**.



Note: If users connect only one PS/2 equipment, the PS/2 signals connected to the FPGA I/O should be "PS2_CLK" and "PS2_DAT".

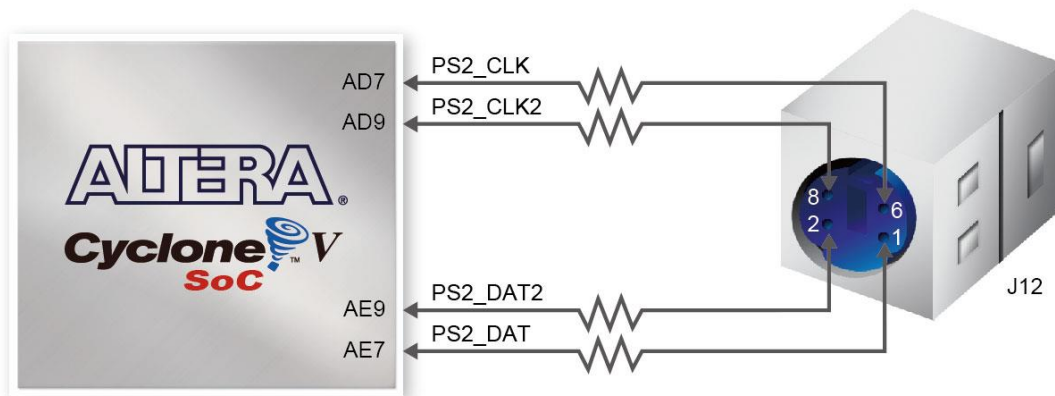


Figure 3-28 Connections between the FPGA and PS/2



Figure 3-29 Y-Cable for using keyboard and mouse simultaneously

Table 3-21 Pin Assignment of PS/2

Signal Name	FPGA Pin No.	Description	I/O Standard
PS2_CLK	PIN_AD7	PS/2 Clock	3.3V
PS2_DAT	PIN_AE7	PS/2 Data	3.3V
PS2_CLK2	PIN_AD9	PS/2 Clock (reserved for second PS/2 device)	3.3V
PS2_DAT2	PIN_AE9	PS/2 Data (reserved for second PS/2 device)	3.3V

3.6.12 A/D Converter and 2x5 Header

The DE1-SoC has an analog-to-digital converter (LTC2308), which features low noise, eight-channel CMOS 12-bit. This ADC offers conversion throughput rate up to 500KSPS. The analog input range for all input channels can be 0 V to 4.096V. The internal conversion clock allows the external serial output data clock (SCLK) to operate at any frequency up to 40MHz. It can be configured to accept eight input signals at inputs ADC_IN0 through ADC_IN7. These eight input signals are connected to a 2x5 header, as shown in **Figure 3-30**.

More information about the A/D converter chip is available in its datasheet. It can be found on manufacturer's website or in the directory \datasheet of De1-SoC system CD.

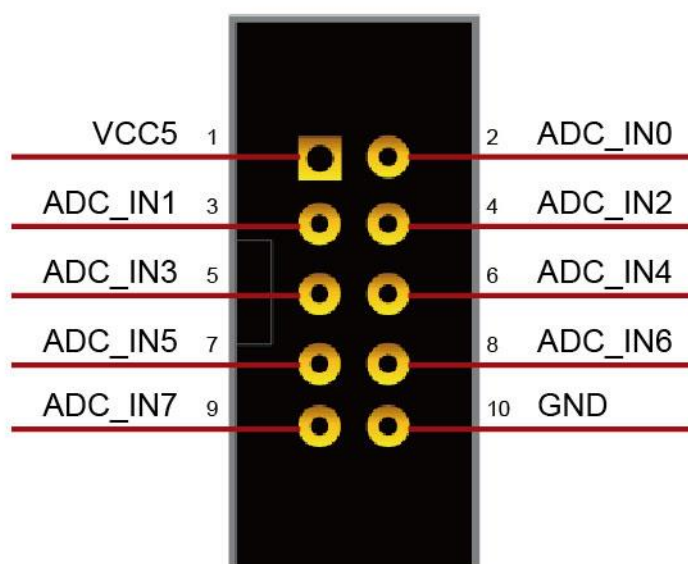


Figure 3-30 Signals of the 2x5 Header

Figure 3-31 shows the connections between the FPGA, 2x5 header, and the A/D converter.

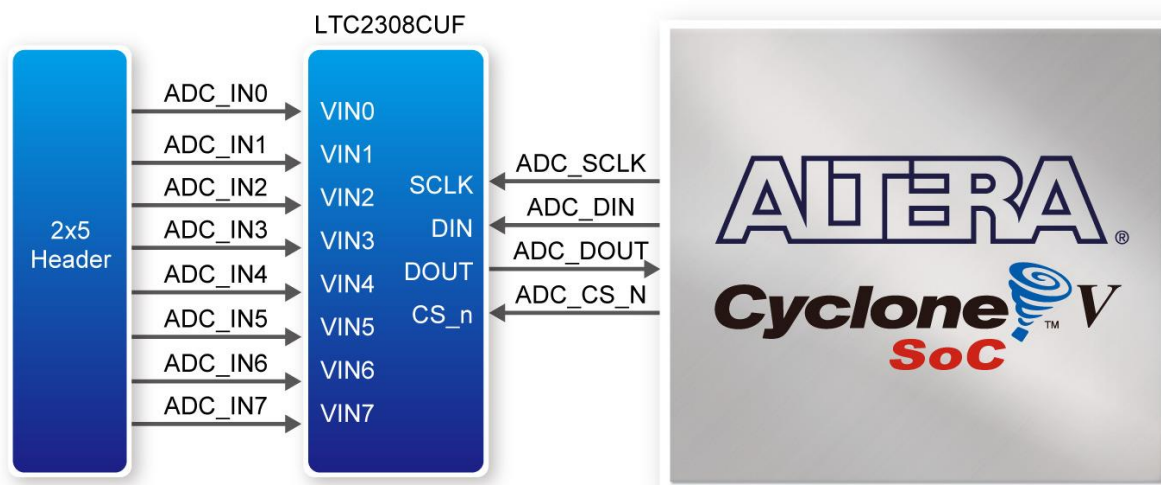


Figure 3-31 Connections between the FPGA, 2x5 header, and the A/D converter

Table 3-22 Pin Assignment of ADC

Signal Name	FPGA Pin No.	Description	I/O Standard
ADC_CS_N	PIN_AJ4	Chip select	3.3V
ADC_DOUT	PIN_AK3	Digital data input	3.3V
ADC_DIN	PIN_AK4	Digital data output	3.3V
ADC_SCLK	PIN_AK2	Digital clock input	3.3V

3.7 Peripherals Connected to Hard Processor System (HPS)

This section introduces the interfaces connected to the HPS section of the Cyclone V SoC FPGA. Users can access these interfaces via the HPS processor.

3.7.1 User Push-buttons and LEDs

Similar to the FPGA, the HPS also has its set of switches, buttons, LEDs, and other interfaces connected exclusively. Users can control these interfaces to monitor the status of HPS.

Table 3-23 gives the pin assignment of all the LEDs, switches, and push-buttons.

Table 3-23 Pin Assignment of LEDs, Switches and Push-buttons

Signal Name	HPS GPIO	Register/bit	Function
HPS_KEY	GPIO54	GPIO1[25]	I/O
HPS_LED	GPIO53	GPIO1[24]	I/O

3.7.2 Gigabit Ethernet

The board supports Gigabit Ethernet transfer by an external Micrel KSZ9021RN PHY chip and HPS Ethernet MAC function. The KSZ9021RN chip with integrated 10/100/1000 Mbps Gigabit Ethernet transceiver also supports RGMII MAC interface. **Figure 3-32** shows the connections between the HPS, Gigabit Ethernet PHY, and RJ-45 connector.

The pin assignment associated to Gigabit Ethernet interface is listed in **Table 3-24**. More information about the KSZ9021RN PHY chip and its datasheet, as well as the application notes, which are available on the manufacturer's website.

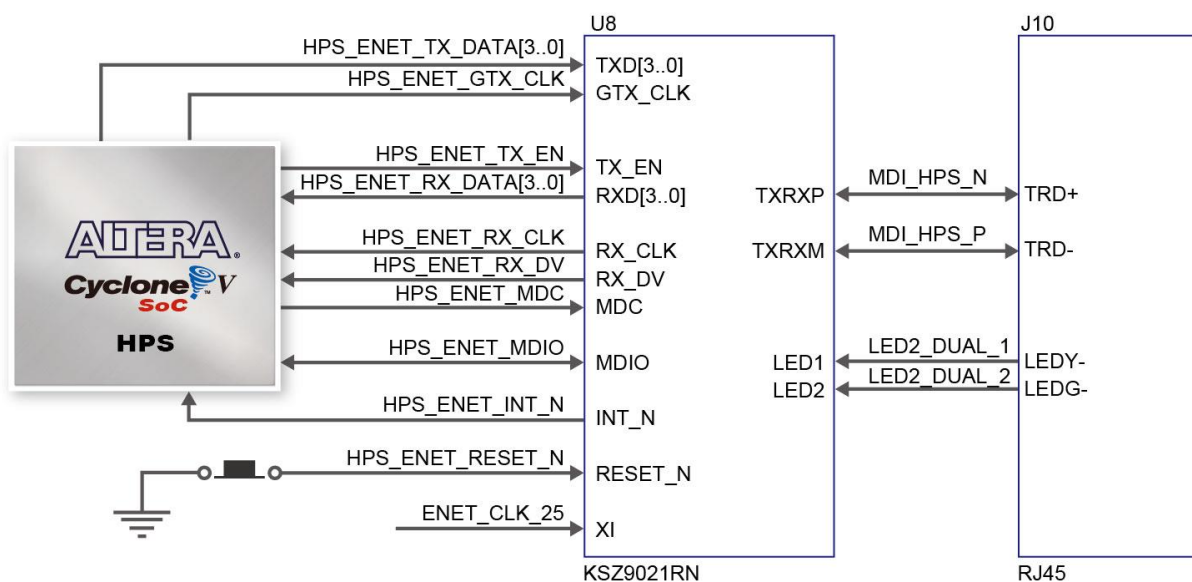


Figure 3-32 Connections between the HPS and Gigabit Ethernet

Table 3-24 Pin Assignment of Gigabit Ethernet PHY

<i>Signal Name</i>	<i>FPGA Pin No.</i>	<i>Description</i>	<i>I/O Standard</i>
HPS_ENET_TX_EN	PIN_A20	GMII and MII transmit enable	3.3V
HPS_ENET_TX_DATA[0]	PIN_F20	MIl transmit data[0]	3.3V
HPS_ENET_TX_DATA[1]	PIN_J19	MIl transmit data[1]	3.3V
HPS_ENET_TX_DATA[2]	PIN_F21	MIl transmit data[2]	3.3V
HPS_ENET_TX_DATA[3]	PIN_F19	MIl transmit data[3]	3.3V
HPS_ENET_RX_DV	PIN_K17	GMII and MII receive data valid	3.3V
HPS_ENET_RX_DATA[0]	PIN_A21	GMII and MII receive data[0]	3.3V
HPS_ENET_RX_DATA[1]	PIN_B20	GMII and MII receive data[1]	3.3V
HPS_ENET_RX_DATA[2]	PIN_B18	GMII and MII receive data[2]	3.3V
HPS_ENET_RX_DATA[3]	PIN_D21	GMII and MII receive data[3]	3.3V
HPS_ENET_RX_CLK	PIN_G20	GMII and MII receive clock	3.3V
HPS_ENET_RESET_N	PIN_E18	Hardware Reset Signal	3.3V
HPS_ENET_MDIO	PIN_E21	Management Data	3.3V
HPS_ENET_MDC	PIN_B21	Management Data Clock Reference	3.3V
HPS_ENET_INT_N	PIN_C19	Interrupt Open Drain Output	3.3V
HPS_ENET_GTX_CLK	PIN_H19	GMII Transmit Clock	3.3V

There are two LEDs, green LED (LEDG) and yellow LED (LEDY), which represent the status of Ethernet PHY (KSZ9021RNI). The LED control signals are connected to the LEDs on the RJ45 connector. The state and definition of LEDG and LEDY are listed in [Table 3-25](#). For instance, the connection from board to Gigabit Ethernet is established once the LEDG lights on.

Table 3-25 State and Definition of LED Mode Pins

<i>LED (State)</i>		<i>LED (Definition)</i>		<i>Link /Activity</i>
<i>LEDG</i>	<i>LEDY</i>	<i>LEDG</i>	<i>LEDY</i>	
H	H	OFF	OFF	Link off
L	H	ON	OFF	1000 Link / No Activity
Toggle	H	Blinking	OFF	1000 Link / Activity (RX, TX)
H	L	OFF	ON	100 Link / No Activity
H	Toggle	OFF	Blinking	100 Link / Activity (RX, TX)
L	L	ON	ON	10 Link/ No Activity
Toggle	Toggle	Blinking	Blinking	10 Link / Activity (RX, TX)

3.7.3 UART

The board has one UART interface connected for communication with the HPS. This interface doesn't support HW flow control signals. The physical interface is implemented by UART-USB onboard bridge from a FT232R chip to the host with an USB Mini-B connector. More information about the chip is available on the manufacturer's website, or in the directory \Datasheets\UART TO

USB of DE1-SoC system CD. **Figure 3-33** shows the connections between the HPS, FT232R chip, and the USB Mini-B connector. **Table 3-26** lists the pin assignment of UART interface connected to the HPS.

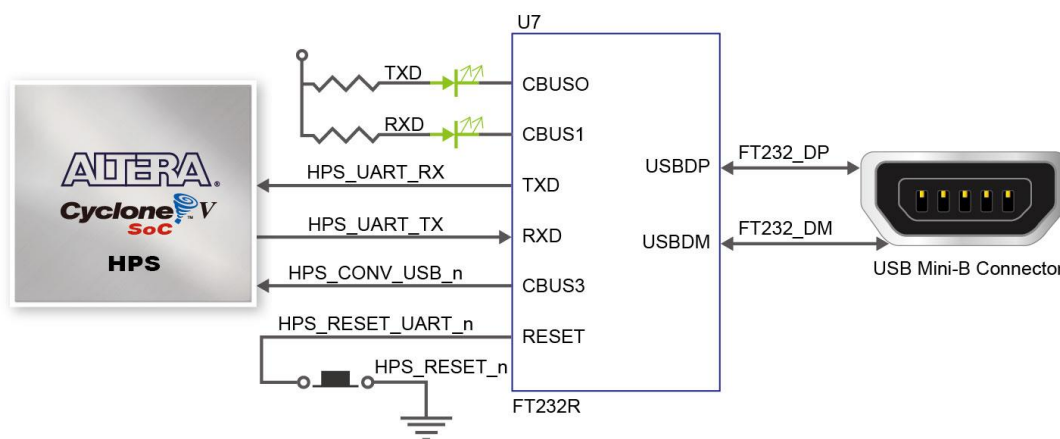


Figure 3-33 Connections between the HPS and FT232R Chip

Table 3-26 Pin Assignment of UART Interface

Signal Name	FPGA Pin No.	Description	I/O Standard
HPS_UART_RX	PIN_B25	HPS UART Receiver	3.3V
HPS_UART_TX	PIN_C25	HPS UART Transmitter	3.3V
HPS_CONV_USB_N	PIN_B15	Reserve	3.3V

3.7.4 DDR3 Memory

The board supports 1GB of DDR3 SDRAM comprising of two x16 bit DDR3 devices on HPS side. The signals are connected to the dedicated Hard Memory Controller for HPS I/O banks and the target speed is 400 MHz. **Table 3-27** lists the pin assignment of DDR3 and its description with I/O standard.

Table 3-27 Pin Assignment of DDR3 Memory

Signal Name	FPGA Pin No.	Description	I/O Standard
HPS_DDR3_A[0]	PIN_F26	HPS DDR3 Address[0]	SSTL-15 Class I
HPS_DDR3_A[1]	PIN_G30	HPS DDR3 Address[1]	SSTL-15 Class I
HPS_DDR3_A[2]	PIN_F28	HPS DDR3 Address[2]	SSTL-15 Class I
HPS_DDR3_A[3]	PIN_F30	HPS DDR3 Address[3]	SSTL-15 Class I
HPS_DDR3_A[4]	PIN_J25	HPS DDR3 Address[4]	SSTL-15 Class I
HPS_DDR3_A[5]	PIN_J27	HPS DDR3 Address[5]	SSTL-15 Class I
HPS_DDR3_A[6]	PIN_F29	HPS DDR3 Address[6]	SSTL-15 Class I

HPS_DDR3_A[7]	PIN_E28	HPS DDR3 Address[7]	SSTL-15 Class I
HPS_DDR3_A[8]	PIN_H27	HPS DDR3 Address[8]	SSTL-15 Class I
HPS_DDR3_A[9]	PIN_G26	HPS DDR3 Address[9]	SSTL-15 Class I
HPS_DDR3_A[10]	PIN_D29	HPS DDR3 Address[10]	SSTL-15 Class I
HPS_DDR3_A[11]	PIN_C30	HPS DDR3 Address[11]	SSTL-15 Class I
HPS_DDR3_A[12]	PIN_B30	HPS DDR3 Address[12]	SSTL-15 Class I
HPS_DDR3_A[13]	PIN_C29	HPS DDR3 Address[13]	SSTL-15 Class I
HPS_DDR3_A[14]	PIN_H25	HPS DDR3 Address[14]	SSTL-15 Class I
HPS_DDR3_BA[0]	PIN_E29	HPS DDR3 Bank Address[0]	SSTL-15 Class I
HPS_DDR3_BA[1]	PIN_J24	HPS DDR3 Bank Address[1]	SSTL-15 Class I
HPS_DDR3_BA[2]	PIN_J23	HPS DDR3 Bank Address[2]	SSTL-15 Class I
HPS_DDR3_CAS_n	PIN_E27	DDR3 Column Address Strobe	SSTL-15 Class I
HPS_DDR3_CKE	PIN_L29	HPS DDR3 Clock Enable	SSTL-15 Class I
HPS_DDR3_CK_n	PIN_L23	HPS DDR3 Clock	Differential 1.5-V SSTL Class I
HPS_DDR3_CK_p	PIN_M23	HPS DDR3 Clock p	Differential 1.5-V SSTL Class I
HPS_DDR3_CS_n	PIN_H24	HPS DDR3 Chip Select	SSTL-15 Class I
HPS_DDR3_DM[0]	PIN_K28	HPS DDR3 Data Mask[0]	SSTL-15 Class I
HPS_DDR3_DM[1]	PIN_M28	HPS DDR3 Data Mask[1]	SSTL-15 Class I
HPS_DDR3_DM[2]	PIN_R28	HPS DDR3 Data Mask[2]	SSTL-15 Class I
HPS_DDR3_DM[3]	PIN_W30	HPS DDR3 Data Mask[3]	SSTL-15 Class I
HPS_DDR3_DQ[0]	PIN_K23	HPS DDR3 Data[0]	SSTL-15 Class I
HPS_DDR3_DQ[1]	PIN_K22	HPS DDR3 Data[1]	SSTL-15 Class I
HPS_DDR3_DQ[2]	PIN_H30	HPS DDR3 Data[2]	SSTL-15 Class I
HPS_DDR3_DQ[3]	PIN_G28	HPS DDR3 Data[3]	SSTL-15 Class I
HPS_DDR3_DQ[4]	PIN_L25	HPS DDR3 Data[4]	SSTL-15 Class I
HPS_DDR3_DQ[5]	PIN_L24	HPS DDR3 Data[5]	SSTL-15 Class I
HPS_DDR3_DQ[6]	PIN_J30	HPS DDR3 Data[6]	SSTL-15 Class I
HPS_DDR3_DQ[7]	PIN_J29	HPS DDR3 Data[7]	SSTL-15 Class I
HPS_DDR3_DQ[8]	PIN_K26	HPS DDR3 Data[8]	SSTL-15 Class I
HPS_DDR3_DQ[9]	PIN_L26	HPS DDR3 Data[9]	SSTL-15 Class I
HPS_DDR3_DQ[10]	PIN_K29	HPS DDR3 Data[10]	SSTL-15 Class I
HPS_DDR3_DQ[11]	PIN_K27	HPS DDR3 Data[11]	SSTL-15 Class I
HPS_DDR3_DQ[12]	PIN_M26	HPS DDR3 Data[12]	SSTL-15 Class I
HPS_DDR3_DQ[13]	PIN_M27	HPS DDR3 Data[13]	SSTL-15 Class I
HPS_DDR3_DQ[14]	PIN_L28	HPS DDR3 Data[14]	SSTL-15 Class I
HPS_DDR3_DQ[15]	PIN_M30	HPS DDR3 Data[15]	SSTL-15 Class I
HPS_DDR3_DQ[16]	PIN_U26	HPS DDR3 Data[16]	SSTL-15 Class I
HPS_DDR3_DQ[17]	PIN_T26	HPS DDR3 Data[17]	SSTL-15 Class I
HPS_DDR3_DQ[18]	PIN_N29	HPS DDR3 Data[18]	SSTL-15 Class I
HPS_DDR3_DQ[19]	PIN_N28	HPS DDR3 Data[19]	SSTL-15 Class I
HPS_DDR3_DQ[20]	PIN_P26	HPS DDR3 Data[20]	SSTL-15 Class I
HPS_DDR3_DQ[21]	PIN_P27	HPS DDR3 Data[21]	SSTL-15 Class I
HPS_DDR3_DQ[22]	PIN_N27	HPS DDR3 Data[22]	SSTL-15 Class I
HPS_DDR3_DQ[23]	PIN_R29	HPS DDR3 Data[23]	SSTL-15 Class I

HPS_DDR3_DQ[24]	PIN_P24	HPS DDR3 Data[24]	SSTL-15 Class I
HPS_DDR3_DQ[25]	PIN_P25	HPS DDR3 Data[25]	SSTL-15 Class I
HPS_DDR3_DQ[26]	PIN_T29	HPS DDR3 Data[26]	SSTL-15 Class I
HPS_DDR3_DQ[27]	PIN_T28	HPS DDR3 Data[27]	SSTL-15 Class I
HPS_DDR3_DQ[28]	PIN_R27	HPS DDR3 Data[28]	SSTL-15 Class I
HPS_DDR3_DQ[29]	PIN_R26	HPS DDR3 Data[29]	SSTL-15 Class I
HPS_DDR3_DQ[30]	PIN_V30	HPS DDR3 Data[30]	SSTL-15 Class I
HPS_DDR3_DQ[31]	PIN_W29	HPS DDR3 Data[31]	SSTL-15 Class I
HPS_DDR3_DQS_n[0]	PIN_M19	HPS DDR3 Data Strobe n[0]	Differential 1.5-V SSTL Class I
HPS_DDR3_DQS_n[1]	PIN_N24	HPS DDR3 Data Strobe n[1]	Differential 1.5-V SSTL Class I
HPS_DDR3_DQS_n[2]	PIN_R18	HPS DDR3 Data Strobe n[2]	Differential 1.5-V SSTL Class I
HPS_DDR3_DQS_n[3]	PIN_R21	HPS DDR3 Data Strobe n[3]	Differential 1.5-V SSTL Class I
HPS_DDR3_DQS_p[0]	PIN_N18	HPS DDR3 Data Strobe p[0]	Differential 1.5-V SSTL Class I
HPS_DDR3_DQS_p[1]	PIN_N25	HPS DDR3 Data Strobe p[1]	Differential 1.5-V SSTL Class I
HPS_DDR3_DQS_p[2]	PIN_R19	HPS DDR3 Data Strobe p[2]	Differential 1.5-V SSTL Class I
HPS_DDR3_DQS_p[3]	PIN_R22	HPS DDR3 Data Strobe p[3]	Differential 1.5-V SSTL Class I
HPS_DDR3_ODT	PIN_H28	HPS DDR3 On-die Termination	SSTL-15 Class I
HPS_DDR3_RAS_n	PIN_D30	DDR3 Row Address Strobe	SSTL-15 Class I
HPS_DDR3_RESET_n	PIN_P30	HPS DDR3 Reset	SSTL-15 Class I
HPS_DDR3_WE_n	PIN_C28	HPS DDR3 Write Enable	SSTL-15 Class I
HPS_DDR3_RZQ	PIN_D27	External reference ball for output drive calibration	1.5 V

3.7.5 Micro SD Card Socket

The board supports Micro SD card interface with x4 data lines. It serves not only an external storage for the HPS, but also an alternative boot option for DE1-SoC board. **Figure 3-34** shows signals connected between the HPS and Micro SD card socket.

Table 3-28 lists the pin assignment of Micro SD card socket to the HPS.

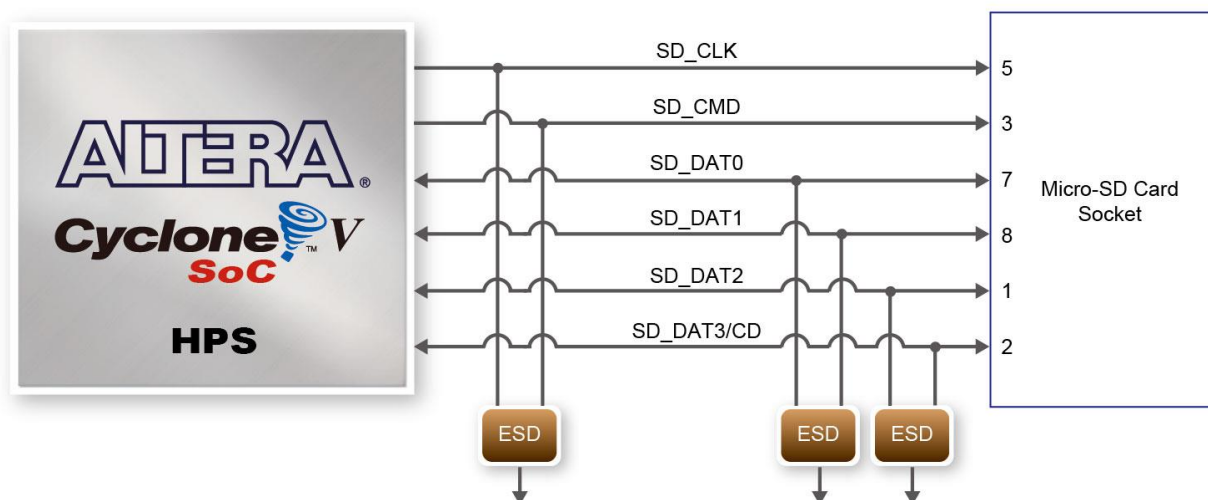


Figure 3-34 Connections between the FPGA and SD card socket

Table 3-28 Pin Assignment of Micro SD Card Socket

Signal Name	FPGA Pin No.	Description	I/O Standard
HPS_SD_CLK	PIN_A16	HPS SD Clock	3.3V
HPS_SD_CMD	PIN_F18	HPS SD Command Line	3.3V
HPS_SD_DATA[0]	PIN_G18	HPS SD Data[0]	3.3V
HPS_SD_DATA[1]	PIN_C17	HPS SD Data[1]	3.3V
HPS_SD_DATA[2]	PIN_D17	HPS SD Data[2]	3.3V
HPS_SD_DATA[3]	PIN_B16	HPS SD Data[3]	3.3V

3.7.6 2-port USB Host

The board has two USB 2.0 type-A ports with a SMSC USB3300 controller and a 2-port hub controller. The SMSC USB3300 device in 32-pin QFN package interfaces with the SMSC USB2512B hub controller. This device supports UTMI+ Low Pin Interface (ULPI), which communicates with the USB 2.0 controller in HPS. The PHY operates in Host mode by connecting the ID pin of USB3300 to ground. When operating in Host mode, the device is powered by the two USB type-A ports. **Figure 3-35** shows the connections of USB PTG PHY to the HPS. **Table 3-29** lists the pin assignment of USBOTG PHY to the HPS.

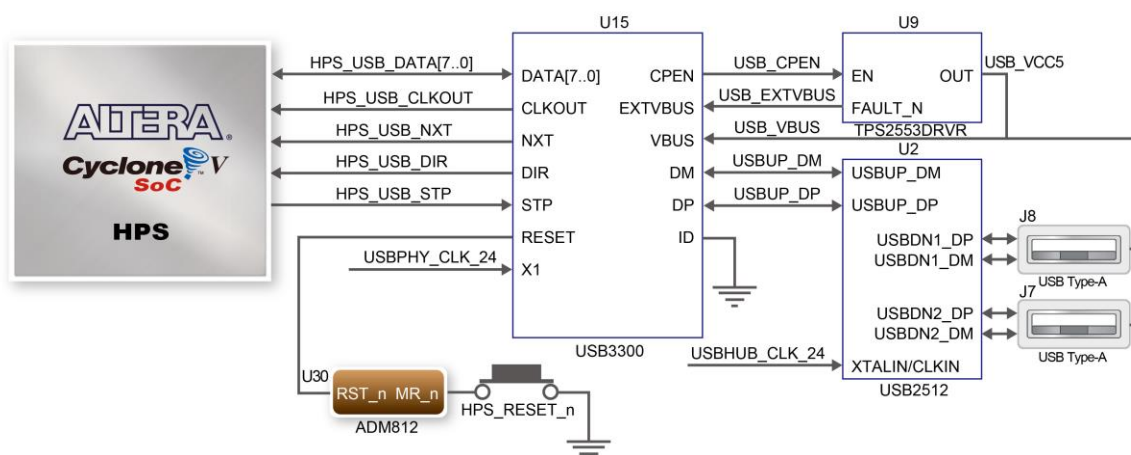


Figure 3-35 Connections between the HPS and USB OTG PHY

Table 3-29 Pin Assignment of USB OTG PHY

Signal Name	FPGA Pin No.	Description	I/O Standard
HPS_USB_CLKOUT	PIN_N16	60MHz Reference Clock Output	3.3V
HPS_USB_DATA[0]	PIN_E16	HPS USB_DATA[0]	3.3V
HPS_USB_DATA[1]	PIN_G16	HPS USB_DATA[1]	3.3V
HPS_USB_DATA[2]	PIN_D16	HPS USB_DATA[2]	3.3V
HPS_USB_DATA[3]	PIN_D14	HPS USB_DATA[3]	3.3V
HPS_USB_DATA[4]	PIN_A15	HPS USB_DATA[4]	3.3V
HPS_USB_DATA[5]	PIN_C14	HPS USB_DATA[5]	3.3V
HPS_USB_DATA[6]	PIN_D15	HPS USB_DATA[6]	3.3V
HPS_USB_DATA[7]	PIN_M17	HPS USB_DATA[7]	3.3V
HPS_USB_DIR	PIN_E14	Direction of the Data Bus	3.3V
HPS_USB_NXT	PIN_A14	Throttle the Data	3.3V
HPS_USB_RESET	PIN_G17	HPS USB PHY Reset	3.3V
HPS_USB_STP	PIN_C15	Stop Data Stream on the Bus	3.3V

3.7.7 G-sensor

The board comes with a digital accelerometer sensor module (ADXL345), commonly known as G-sensor. This G-sensor is a small, thin, ultralow power assumption 3-axis accelerometer with high-resolution measurement. Digitalized output is formatted as 16-bit in two's complement and can be accessed through I2C interface. The I2C address of G-sensor is 0xA6/0xA7. More information about this chip can be found in its datasheet, which is available on manufacturer's website or in the directory \Datasheet folder of DE1-SoC system CD. **Figure 3-36** shows the connections between the HPS and G-sensor. **Table 3-30** lists the pin assignment of G-sensor to the HPS.

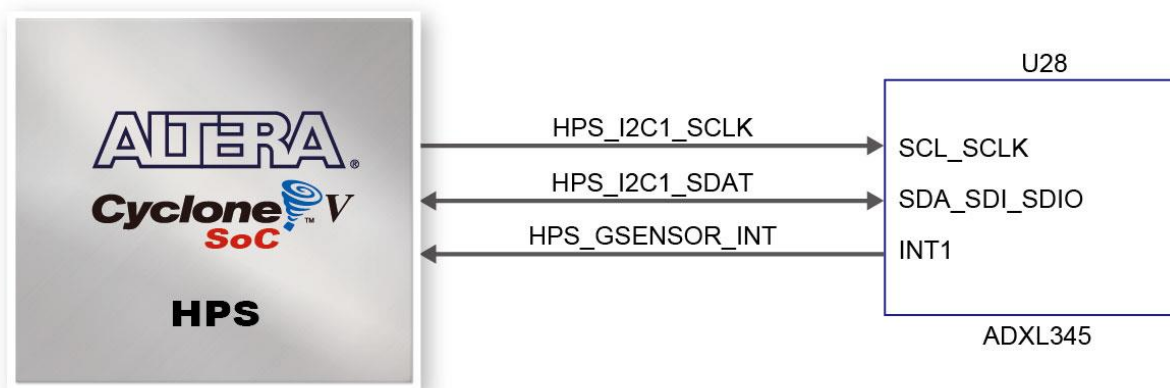


Figure 3-36 Connections between Cyclone V SoC FPGA and G-Sensor

Table 3-30 Pin Assignment of G-senor

Signal Name	FPGA Pin No.	Description	I/O Standard
HPS_GSENSOR_INT	PIN_B22	HPS GSENSOR Interrupt Output	3.3V
HPS_I2C1_SCLK	PIN_E23	HPS I2C Clock (share bus with LTC)	3.3V
HPS_I2C1_SDAT	PIN_C24	HPS I2C Data (share bus)	3.3V

3.7.8 LTC Connector

The board has a 14-pin header, which is originally used to communicate with various daughter cards from Linear Technology. It is connected to the SPI Master and I2C ports of HPS. The communication with these two protocols is bi-directional. The 14-pin header can also be used for GPIO, SPI, or I2C based communication with the HPS. Connections between the HPS and LTC connector are shown in [Figure 3-37](#), and the pin assignment of LTC connector is listed in [Table 3-31](#).

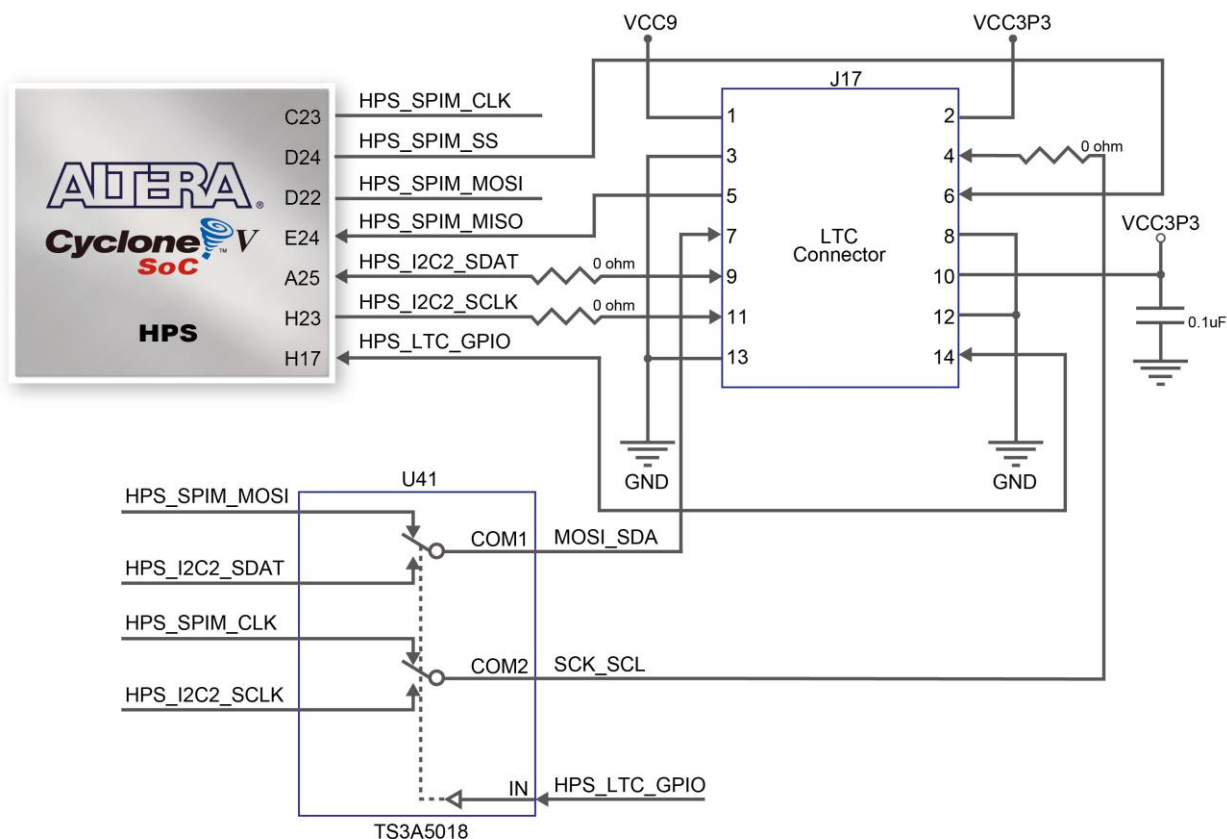


Figure 3-37 Connections between the HPS and LTC connector

Table 3-31 Pin Assignment of LTC Connector

Signal Name	FPGA Pin No.	Description	I/O Standard
HPS_LTC_GPIO	PIN_H17	HPS LTC GPIO	3.3V
HPS_I2C2_SCLK	PIN_H23	HPS I2C2 Clock (share bus with G-Sensor)	3.3V
HPS_I2C2_SDAT	PIN_A25	HPS I2C2 Data (share bus with G-Sensor)	3.3V
HPS_SPIM_CLK	PIN_C23	SPI Clock	3.3V
HPS_SPIM_MISO	PIN_E24	SPI Master Input/Slave Output	3.3V
HPS_SPIM_MOSI	PIN_D22	SPI Master Output /Slave Input	3.3V
HPS_SPIM_SS	PIN_D24	SPI Slave Select	3.3V

Chapter 4

DE1-SoC System Builder

This chapter describes how users can create a custom design project with the tool named DE1-SoC System Builder.

4.1 Introduction

The DE1-SoC System Builder is a Windows-based utility. It is designed to help users create a Quartus II project for DE1-SoC within minutes. The generated Quartus II project files include:

- Quartus II project file (.qpf)
- Quartus II setting file (.qsf)
- Top-level design file (.v)
- Synopsis design constraints file (.sdc)
- Pin assignment document (.htm)

The above files generated by the DE1-SoC System Builder can also prevent occurrence of situations that are prone to compilation error when users manually edit the top-level design file or place pin assignment. The common mistakes that users encounter are:

- Board is damaged due to incorrect bank voltage setting or pin assignment.
- Board is malfunctioned because of wrong device chosen, declaration of pin location or direction is incorrect or forgotten.
- Performance degradation due to improper pin assignment.

4.2 Design Flow

This section provides an introduction to the design flow of building a Quartus II project for DE1-SoC under the DE1-SoC System Builder. The design flow is illustrated in **Figure 4-1**.

The DE1-SoC System Builder will generate two major files, a top-level design file (.v) and a Quartus II setting file (.qsf) after users launch the DE1-SoC System Builder and create a new project according to their design requirements

The top-level design file contains a top-level Verilog HDL wrapper for users to add their own design/logic. The Quartus II setting file contains information such as FPGA device type, top-level pin assignment, and the I/O standard for each user-defined I/O pin.

Finally, the Quartus II programmer is used to download .sof file to the development board via JTAG interface.

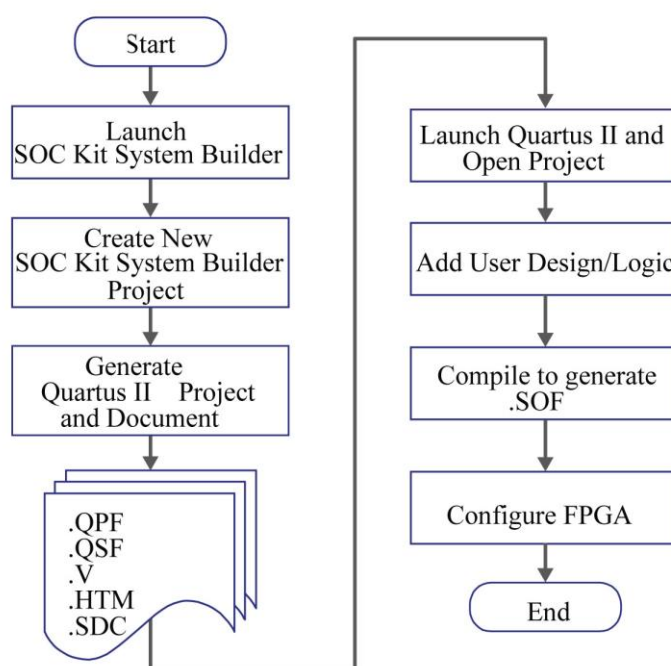


Figure 4-1 Design flow of building a project from the beginning to the end

4.3 Using DE1-SoC System Builder

This section provides the procedures in details on how to use the DE1-SoC System Builder.

■ Install and Launch the DE1-SoC System Builder

The DE1-SoC System Builder is located in the directory: “Tools\SystemBuilder” of the DE1-SoC System CD. Users can copy the entire folder to a host computer without installing the utility. A window will pop up, as shown in **Figure 4-2**, after executing the DE1-SoC SystemBuilder.exe on the host computer.

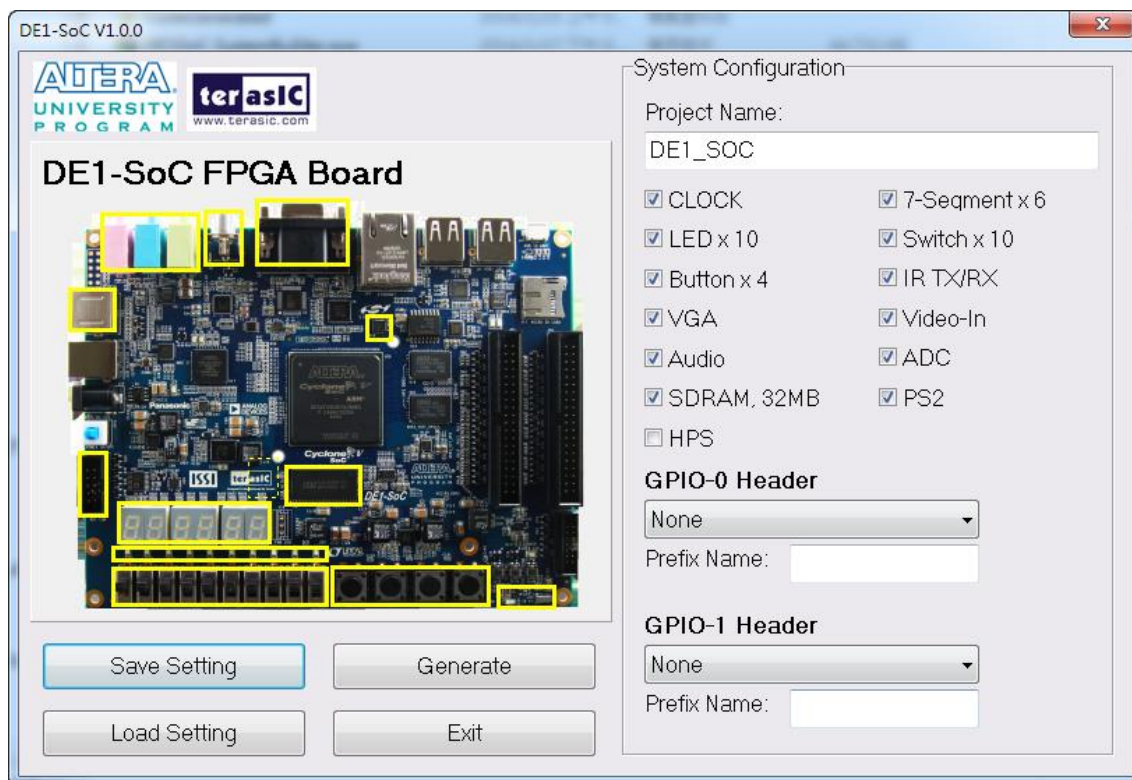


Figure 4-2 The GUI of DE1-SoC System Builder

■ Enter Project Name

Enter the project name in the circled area, as shown in **Figure 4-3**.

The project name typed in will be assigned automatically as the name of your top-level design entity.

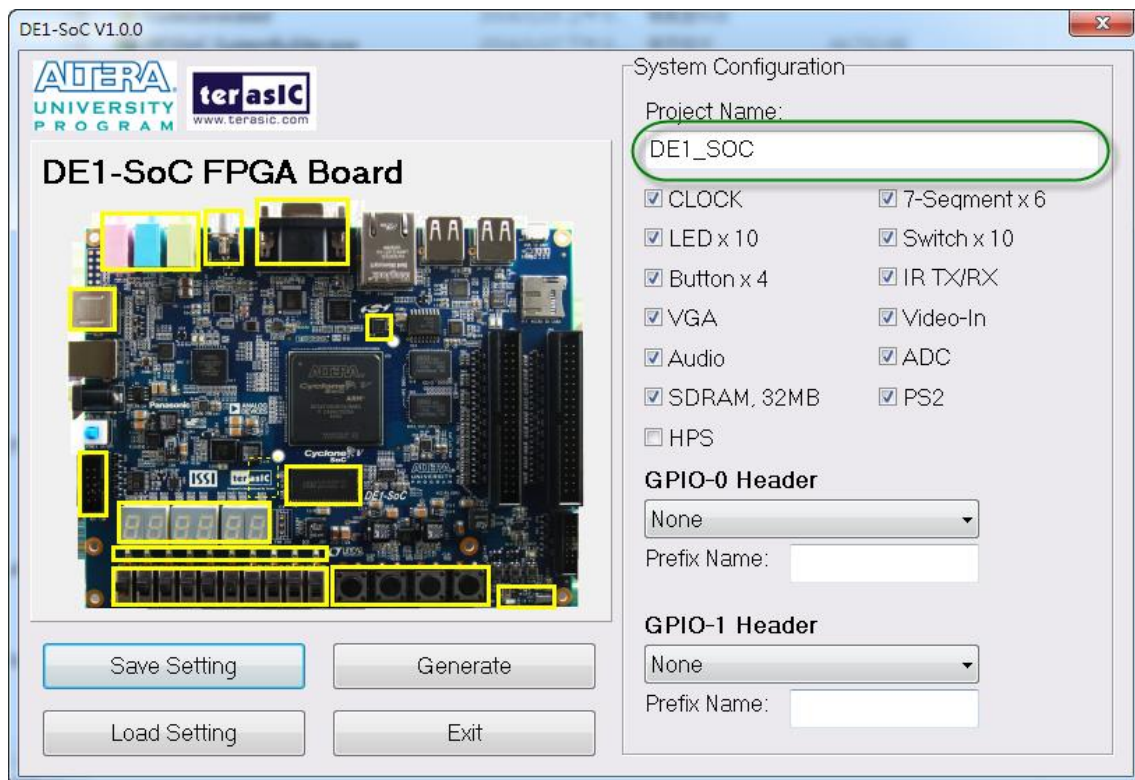


Figure 4-3 Enter the project name

■ System Configuration

Users are given the flexibility in the System Configuration to include their choice of components in the project, as shown in **Figure 4-4**. Each component onboard is listed and users can enable or disable one or more components at will. If a component is enabled, the DE1-SoC System Builder will automatically generate its associated pin assignment, including the pin name, pin location, pin direction, and I/O standard.

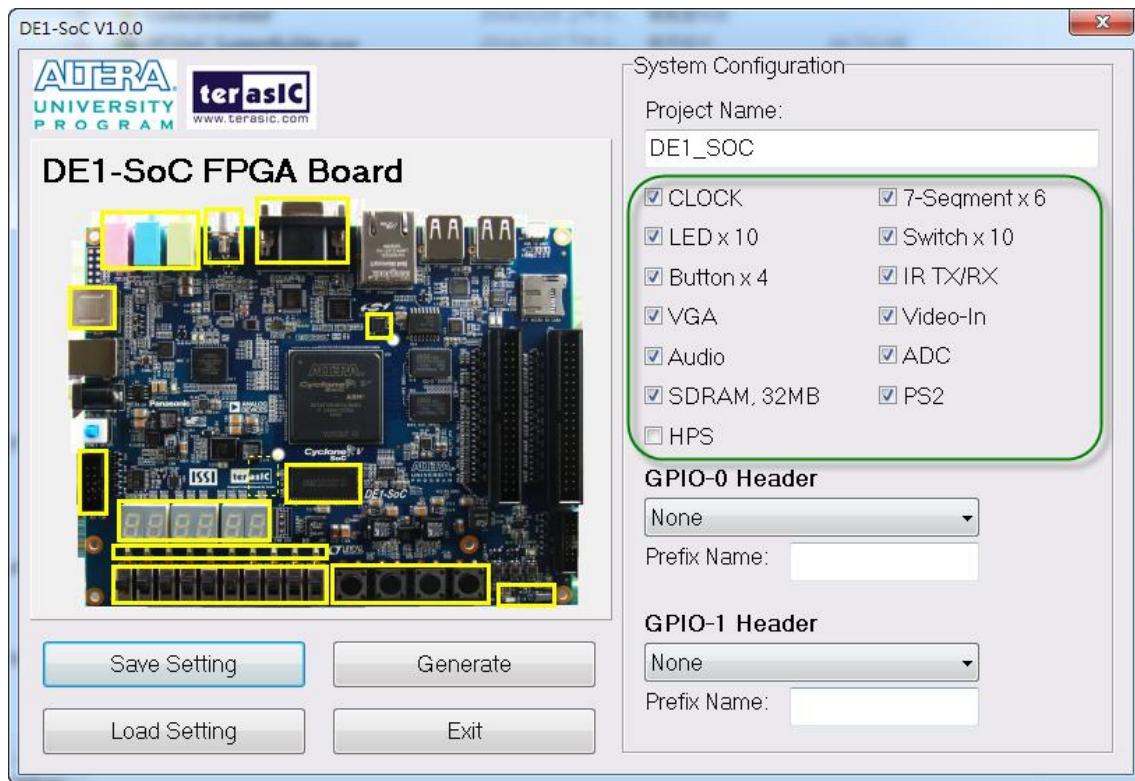


Figure 4-4 System configuration group

■ GPIO Expansion

If users connect any Terasic GPIO-based daughter card to the GPIO connector(s) on DE1-SoC, the DE1-SoC System Builder can generate a project that include the corresponding module, as shown in **Figure 4-5**. It will also generate the associated pin assignment automatically, including pin name, pin location, pin direction, and I/O standard.

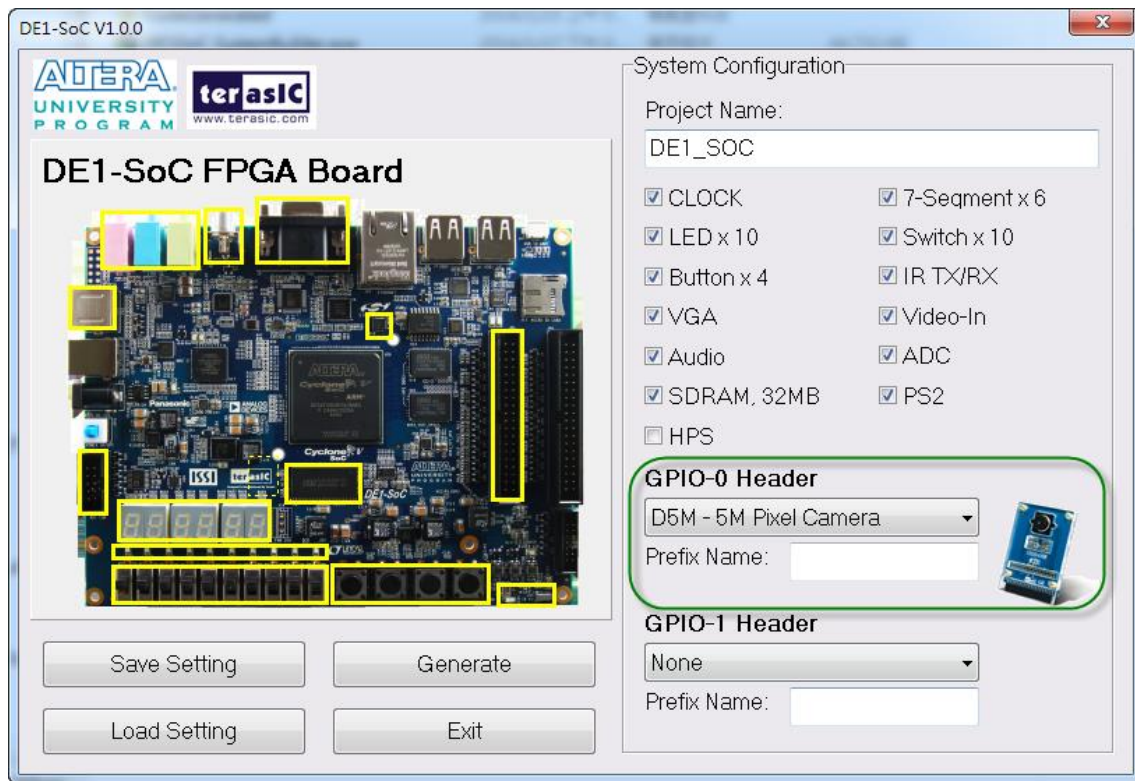


Figure 4-5 GPIO expansion group

The “Prefix Name” is an optional feature that denote the pin name of the daughter card assigned in your design. Users may leave this field blank.

■ Project Setting Management

The DE1-SoC System Builder also provides the option to load a setting or save users’ current board configuration in .cfg file, as shown in **Figure 4-6**.

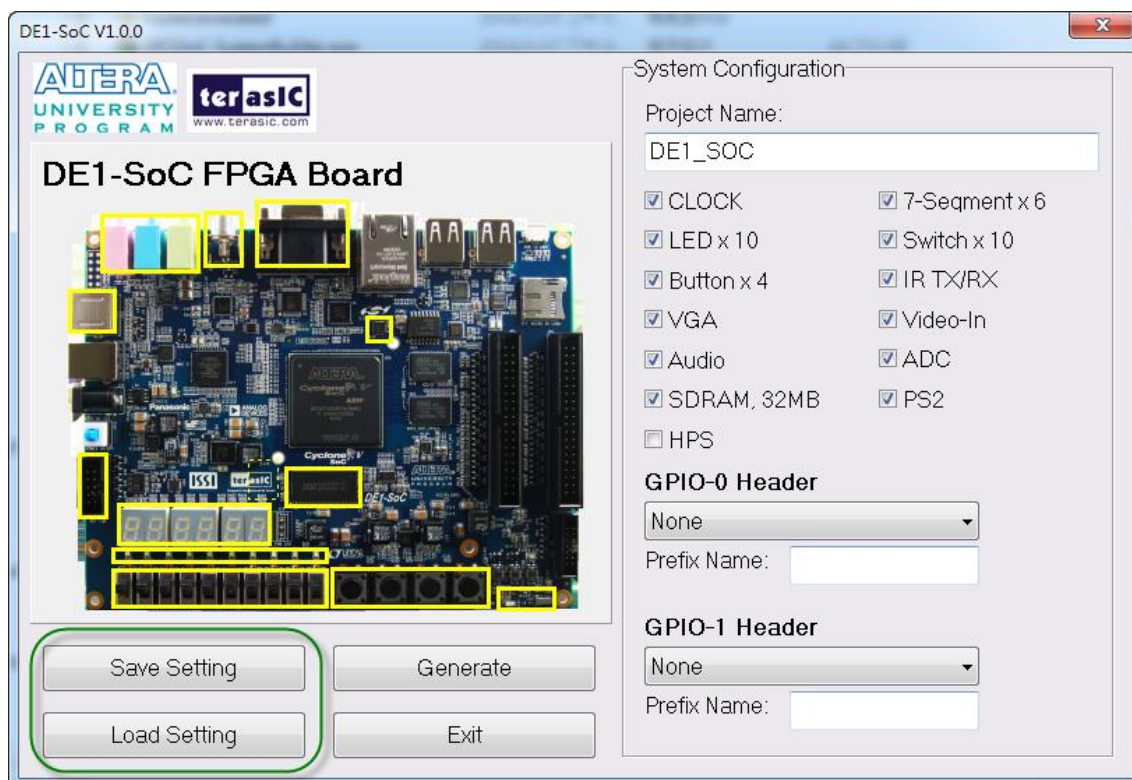


Figure 4-6 Project Settings

■ Project Generation

When users press the *Generate* button, the DE1-SoC System Builder will generate the corresponding Quartus II files and documents, as listed in **Table 4-1**:

Table 4-1 Files generated by the DE1-SoC System Builder

No.	Filename	Description
1	<Project name>.v	Top level Verilog HDL file for Quartus II
2	<Project name>.qpf	Quartus II Project File
3	<Project name>.qsf	Quartus II Setting File
4	<Project name>.sdc	Synopsis Design Constraints file for Quartus II
5	<Project name>.htm	Pin Assignment Document

Users can add custom logic into the project in Quartus II and compile the project to generate the SRAM Object File (.sof).

Chapter 5

Examples For FPGA

This chapter provides examples of advanced designs implemented by RTL or Qsys on the DE1-SoC board. These reference designs cover the features of peripherals connected to the FPGA, such as audio, SDRAM, and IR receiver. All the associated files can be found in the directory \Demonstrations\FPGA of DE1-SoC System CD.

■ Installation of Demonstrations

To install the demonstrations on your computer:

Copy the folder Demonstrations to a local directory of your choice. It is important to make sure the path to your local directory contains NO space. Otherwise it will lead to error in Nios II. **Note** Quartus II v16.0 or later is required for all DE1-SoC demonstrations to support Cyclone V SoC device.

5.1 DE1-SoC Factory Configuration

The DE1-SoC board has a default configuration bit-stream pre-programmed, which demonstrates some of the basic features onboard. The setup required for this demonstration and the location of its files are shown below.

■ Demonstration Setup, File Locations, and Instructions

- Project directory: DE1_SoC_Default
- Bitstream used: DE1_SoC_Default.sof or DE1_SoC_Default.jic
- Power on the DE1-SoC board with the USB cable connected to the USB-Blaster II port. If necessary (that is, if the default factory configuration is not currently stored in the EPCS device), download the bit stream to the board via JTAG interface.
- You should now be able to observe the 7-segment displays are showing a sequence of characters, and the red LEDs are blinking.

- If the VGA D-SUB connector is connected to a VGA display, it would show a color picture.
- If the stereo line-out jack is connected to a speaker and KEY[1] is pressed, a 1 kHz humming sound will come out of the line-out port .
- For the ease of execution, a demo_batch folder is provided in the project. It is able to not only load the bit stream into the FPGA in command line, but also program or erase .jic file to the EPCS by executing the test.bat file shown in **Figure 5-1**.

If users want to program a new design into the EPCS device, the easiest method is to copy the new .sof file into the demo_batch folder and execute the test.bat. Option “2” will convert the .sof to .jic and option “3” will program .jic file into the EPCS device.

```
*****
Please choose your operation
"1" for programming .sof to FPGA.
"2" for converting .sof to .jic
"3" for programming .jic to EPCS.
"4" for erasing .jic from EPCS.
"5" for EXIT batch.
*****
Please enter your choise: [1,2,3,4,5]?_
```

Figure 5-1 Command line of the batch file to program the FPGA and EPCS device

5.2 Audio Recording and Playing

This demonstration shows how to implement an audio recorder and player on DE1-SoC board with the built-in audio CODEC chip. It is developed based on Qsys and Eclipse. **Figure 5-2** shows the buttons and slide switches used to interact this demonstration onboard. Users can configure this audio system through two push-buttons and four slide switches:

- SW0 is used to specify the recording source to be Line-in or MIC-In.
- SW1, SW2, and SW3 are used to specify the recording sample rate such as 96K, 48K, 44.1K, 32K, or 8K.
- **Table 5-1** and **Table 5-2** summarize the usage of slide switches for configuring the audio recorder and player.

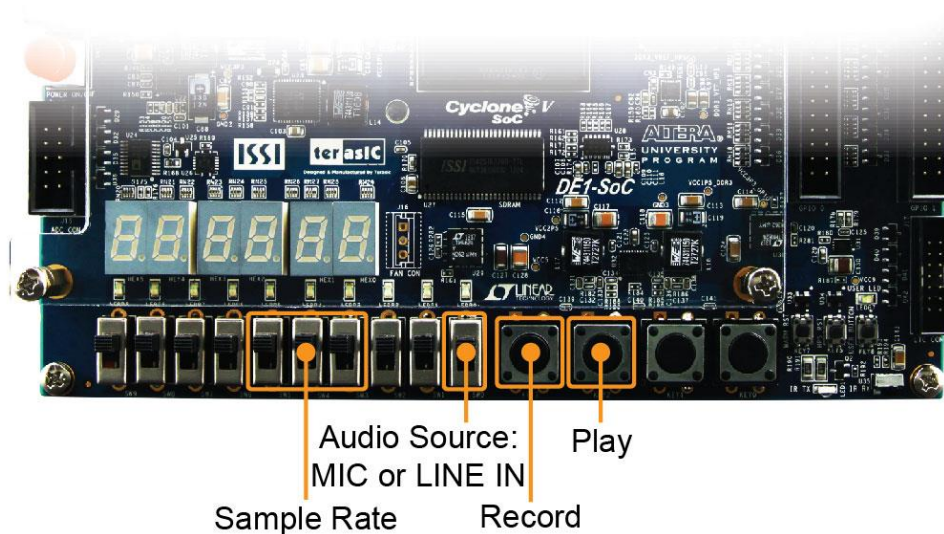


Figure 5-2 Buttons and switches for the audio recorder and player

Figure 5-3 shows the block diagram of audio recorder and player design. There are hardware and software parts in the block diagram. The software part stores the Nios II program in the on-chip memory. The software part is built under Eclipse in C programming language. The hardware part is built under Qsys in Quartus II. The hardware part includes all the other blocks such as the “AUDIO Controller”, which is a user-defined Qsys component and it is designed to send audio data to the audio chip or receive audio data from the audio chip.

The audio chip is programmed through I2C protocol, which is implemented in C code. The I2C pins from the audio chip are connected to Qsys system interconnect fabric through PIO controllers. The audio chip is configured in master mode in this demonstration. The audio interface is configured as 16-bit I2S mode. 18.432MHz clock generated by the PLL is connected to the MCLK/XTI pin of the audio chip through the audio controller.

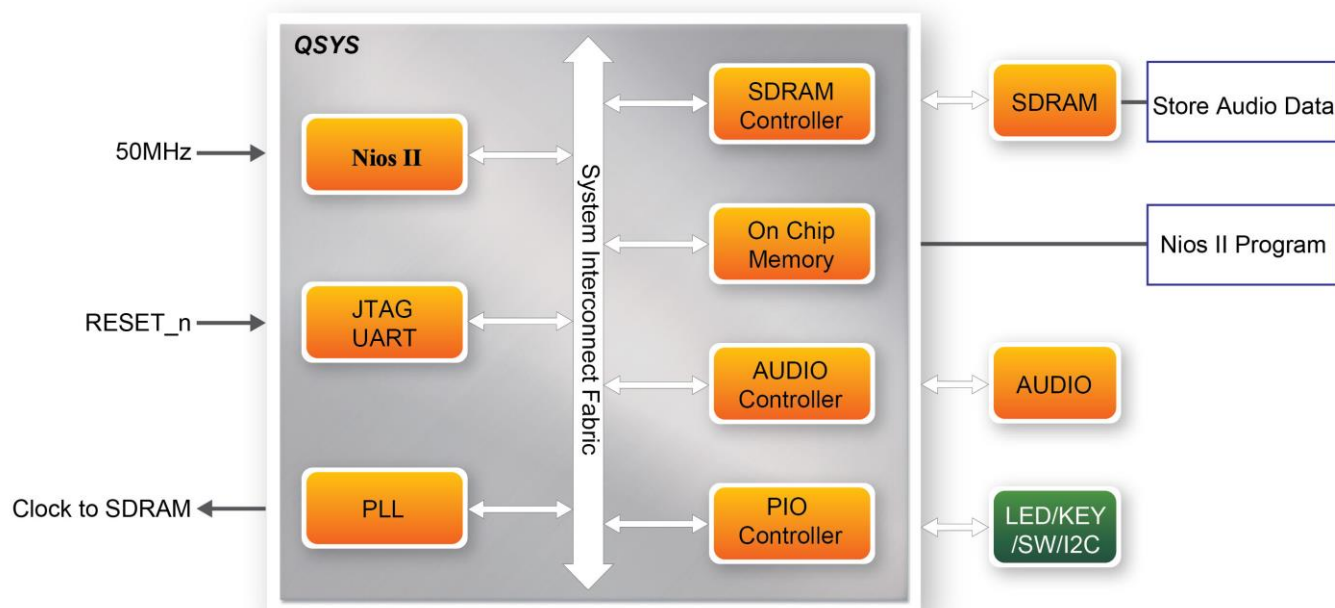


Figure 5-3 Block diagram of the audio recorder and player

■ Demonstration Setup, File Locations, and Instructions

- Hardware project directory: DE1_SoC_Audio
- Bitstream used: DE1_SoC_Audio.sof
- Software project directory: DE1_SoC_Audio\software
- Connect an audio source to the Line-in port
- Connect a Microphone to the MIC-in port
- Connect a speaker or headset to the Line-out port
- Load the bitstream into the FPGA. (note *1)
- Load the software execution file into the FPGA. (note *1)
- Configure the audio with SW0, as shown in [Table 5-1](#).
- Press KEY3 to start/stop audio recording (note *2)
- Press KEY2 to start/stop audio playing (note *3)

Table 5-1 Slide switches usage for audio source

Slide Switches	0 – DOWN Position	1 – UP Position
SW0	Audio is from MIC-in	Audio is from Line-in

Table 5-2 Settings of switches for the sample rate of audio recorder and player

SW5 (0 – DOWN; 1- UP)	SW4 (0 – DOWN; 1-UP)	SW3 (0 – DOWN; 1-UP)	Sample Rate
0	0	0	96K
0	0	1	48K
0	1	0	44.1K
0	1	1	32K
1	0	0	8K
Unlisted combination			96K



Note:

- (1). Execute `DE1_SoC_Audio \demo_batch\ DE1-SoC_Audio.bat` to download .sof and .elf files.
- (2). Recording process will stop if the audio buffer is full.
- (3). Playing process will stop if the audio data is played completely.

5.3 Karaoke Machine

This demonstration uses the microphone-in, line-in, and line-out ports on DE1-SoC to create a Karaoke machine. The WM8731 CODEC is configured in master mode. The audio CODEC generates AD/DA serial bit clock (BCK) and the left/right channel clock (LRCK) automatically. The I2C interface is used to configure the audio CODEC, as shown in **Figure 5-4**. The sample rate and gain of the CODEC are set in a similar manner, and the data input from the line-in port is then mixed with the microphone-in port. The result is sent out to the line-out port.

The sample rate is set to 48 kHz in this demonstration. The gain of the audio CODEC is reconfigured via I2C bus by pressing the pushbutton KEY0, cycling within ten predefined gain values (volume levels) provided by the device.

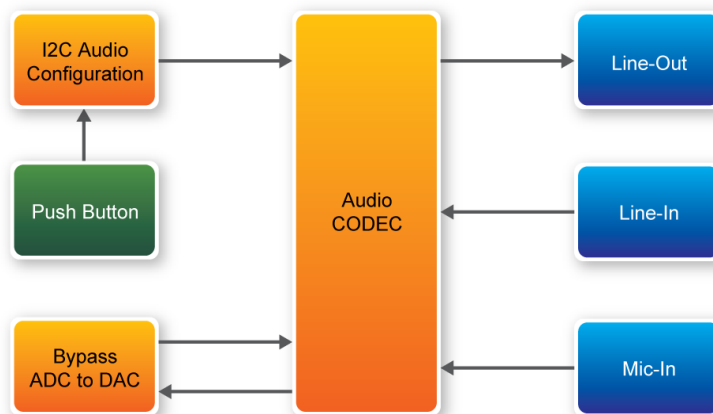


Figure 5-4 Block diagram of the Karaoke machine demonstration

■ Demonstration Setup, File Locations, and Instructions

- Project directory: DE1_SOC_i2sound
- Bitstream used: DE1_SOC_i2sound.sof
- Connect a microphone to the microphone-in port (pink color)
- Connect the audio output of a music player, such as a MP3 player or computer, to the line-in port (blue color)
- Connect a headset/speaker to the line-out port (green color)
- Load the bitstream into the FPGA by executing the batch file 'DE1_SOC_i2sound' in the directory DE1_SOC_i2sound\demo_batch
- Users should be able to hear a mixture of microphone sound and the sound from the music player
- Press KEY0 to adjust the volume; it cycles between volume level 0 to 9

Figure 5-5 illustrates the setup for this demonstration.

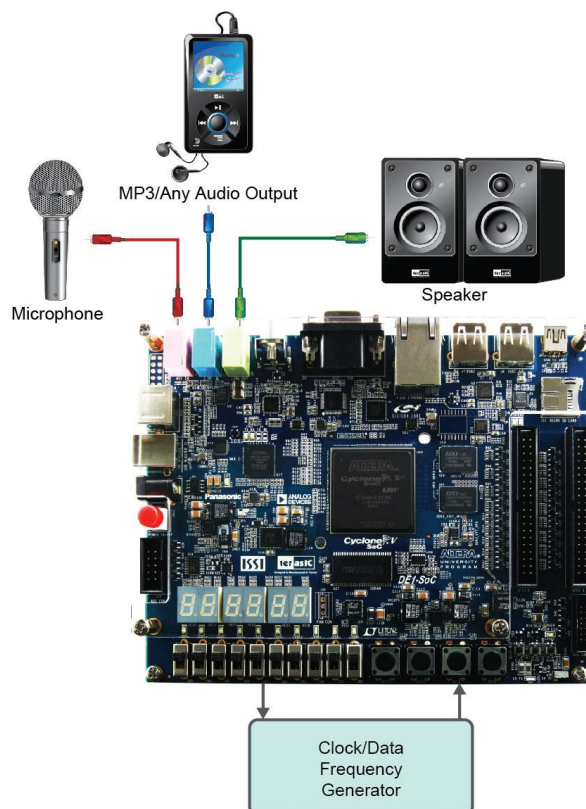


Figure 5-5 Setup for the Karaoke machine

5.4 SDRAM Test in Nios II

There are many applications use SDRAM as a temporary storage. Both hardware and software designs are provided to illustrate how to perform memory access in Qsys in this demonstration. It also shows how Altera's SDRAM controller IP accesses SDRAM and how the Nios II processor reads and writes the SDRAM for hardware verification. The SDRAM controller handles complex aspects of accessing SDRAM such as initializing the memory device, managing SDRAM banks, and keeping the devices refreshed at certain interval.

■ System Block Diagram

Figure 5-6 shows the system block diagram of this demonstration. The system requires a 50 MHz clock input from the board. The SDRAM controller is configured as a 64MB controller. The working frequency of the SDRAM controller is 100MHz, and the Nios II program is running on the on-chip memory.

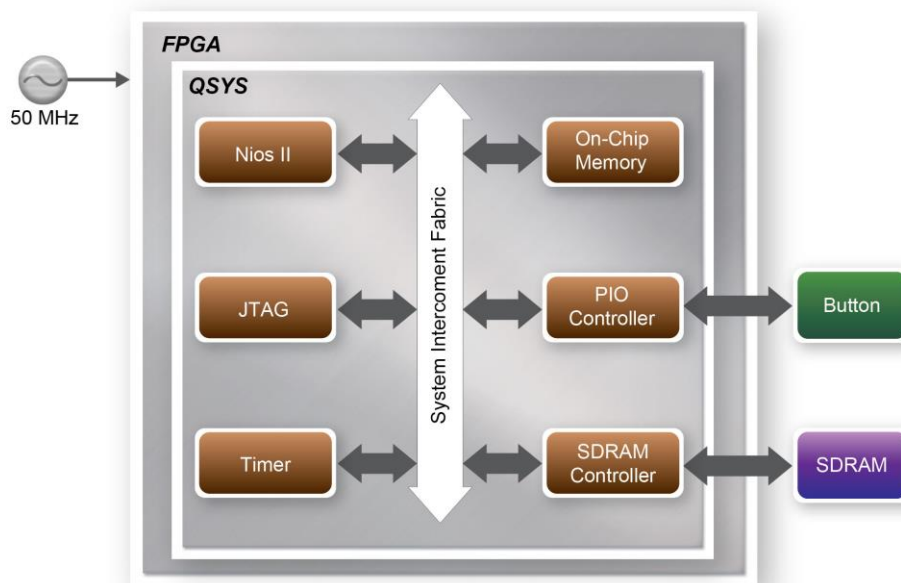


Figure 5-6 Block diagram of the SDRAM test in Nios II

The system flow is controlled by a program running in Nios II. The Nios II program writes test patterns into the entire 64MB of SDRAM first before calling the Nios II system function, `alt_dcache_flush_all`, to make sure all the data are written to the SDRAM. It then reads data from the SDRAM for data verification. The program will show the progress in nios-terminal when writing/reading data to/from the SDRAM. When the verification process reaches 100%, the result will be displayed in nios-terminal.

■ Design Tools

- Quartus II v16.0
- Nios II Eclipse v16.0

■ Demonstration Source Code

- Quartus project directory: `DE1_SoC_SDRAM_Nios_Test`
- Nios II Eclipse directory: `DE1_SoC_SDRAM_Nios_Test \Software`

■ Nios II Project Compilation

- Click “Clean” from the “Project” menu of Nios II Eclipse before compiling the reference design in Nios II Eclipse.

■ Demonstration Batch File

The files are located in the directory \DE1_SoC_SDRAM_Nios_Test \demo_batch.

The folder includes the following files:

- Batch file for USB-Blaster II : DE1_SoC_SDRAM_Nios_Test.bat and DE1_SoC_SDRAM_Nios_Test_bashrc
- FPGA configuration file : DE1_SoC_SDRAM_Nios_Test.sof
- Nios II program: DE1_SoC_SDRAM_Nios_Test.elf

■ Demonstration Setup

- Quartus II v16.0 and Nios II v16.0 must be pre-installed on the host PC.
- Power on the DE1-SoC board.
- Connect the DE1-SoC board (J13) to the host PC with a USB cable and install the USB-Blaster driver if necessary.
- Execute the demo batch file “DE1_SoC_SDRAM_Nios_Test.bat” from the directory DE1_SoC_SDRAM_Nios_Test\demo_batch
- After the program is downloaded and executed successfully, a prompt message will be displayed in nios2-terminal.
- Press any button (**KEY3~KEY0**) to start the SDRAM verification process. Press **KEY0** to run the test continuously.
- The program will display the test progress and result, as shown in **Figure 5-7**.

```

Altera Nios II EDS 13.0 [gcc4]
Info: Quartus II 32-bit Programmer was successful. 0 errors, 0 warnings
Info: Peak virtual memory: 192 megabytes
Info: Processing ended: Wed Sep 04 15:22:21 2013
Info: Elapsed time: 00:00:06
Info: Total CPU time (on all processors): 00:00:02
Using cable "DE-SoC [USB-1]", device 1, instance 0x00
Resetting and pausing target processor: OK
Initializing CPU cache (if present)
OK
Downloaded 61KB in 0.1s
Verified OK
Starting processor at address 0x200201B4
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "DE-SoC [USB-1]", device 1, instance 0
nios2-terminal: (Use the IDE stop button or Ctrl-C to terminate)

===== SDRAM Test! Size=64MB (CPU Clock:100000000) =====
=====
Press any KEY to start test [KEY0 for continued test]
=====> SDRAM Testing, Iteration: 1
write...
10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
read/verify...
10% 20% 30% 40% 50%

```

Figure 5-7 Display of progress and result for the SDRAM test in Nios II

5.5 SDRAM Test in Verilog

DE1-SoC system CD offers another SDRAM test with its test code written in Verilog HDL. The memory size of the SDRAM bank tested is still 64MB.

■ Function Block Diagram

Figure 5-8 shows the function block diagram of this demonstration. The SDRAM controller uses 50 MHz as a reference clock and generates 100 MHz as the memory clock.

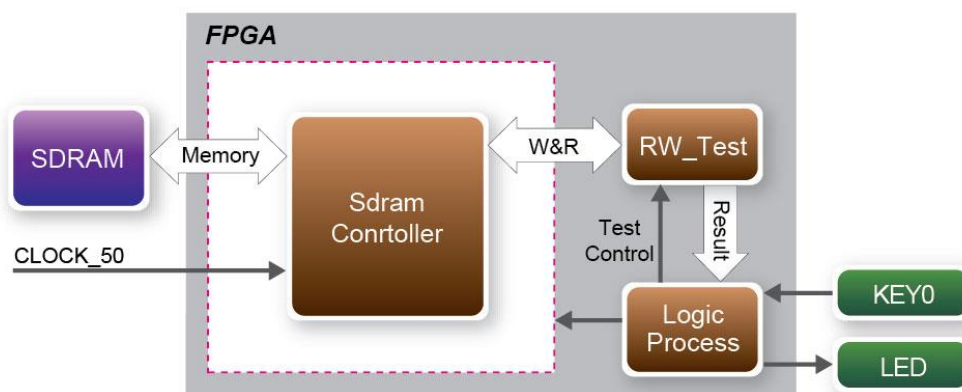


Figure 5-8 Block diagram of the SDRAM test in Verilog

RW_test module writes the entire memory with a test sequence first before comparing the data read back with the regenerated test sequence, which is same as the data written to the memory. KEY0 triggers test control signals for the SDRAM, and the LEDs will indicate the test result according to [Table 5-3](#).

■ Design Tools

- Quartus II v16.0

■ Demonstration Source Code

- Project directory: DE1_SoC_SDRAM_RTL_Test
- Bitstream used: DE1_SoC_SDRAM_RTL_Test.sof

■ Demonstration Batch File

Demo batch file folder: \DE1_SoC_SDRAM_RTL_Test\demo_batch

The directory includes the following files:

- Batch file: DE1_SoC_SDRAM_RTL_Test.bat
- FPGA configuration file: DE1_SoC_SDRAM_RTL_Test.sof

■ Demonstration Setup

- Quartus II v16.0 must be pre-installed to the host PC.
- Connect the DE1-SoC board (J13) to the host PC with a USB cable and install the USB-Blaster II driver if necessary
- Power on the DE1_SoC board.
- Execute the demo batch file “DE1_SoC_SDRAM_RTL_Test.bat” from the directory \DE1_SoC_SDRAM_RTL_Test\demo_batch.
- Press **KEY0** on the DE1_SoC board to start the verification process. When **KEY0** is pressed, the **LEDR** [2:0] should turn on. When **KEY0** is then released, **LEDR1** and **LEDR2** should start blinking.
- After approximately 8 seconds, **LEDR1** should stop blinking and stay ON to indicate the test is PASS. [Table 5-3](#) lists the status of **LED** indicators.
- If **LEDR2** is not blinking, it means 50MHz clock source is not working.
- If **LEDR1** failed to remain ON after approximately 8 seconds, the SDRAM test is NG.
- Press **KEY0** again to repeat the SDRAM test.

Table 5-3 Status of LED Indicators

Name	Description
LEDRO	Reset
LEDRI	ON if the test is PASS after releasing KEY0
LEDRI	Blinks

5.6 TV Box Demonstration

This demonstration turns DE1-SoC board into a TV box by playing video and audio from a DVD player using the VGA output, audio CODEC and the TV decoder on the DE1-SoC board. **Figure 5-9** shows the block diagram of the design. There are two major blocks in the system called I2C_AV_Config and TV_to_VGA. The TV_to_VGA block consists of the ITU-R 656 Decoder, SDRAM Frame Buffer, YUV422 to YUV444, YCbCr to RGB, and VGA Controller. The figure also shows the TV decoder (ADV7180) and the VGA DAC (ADV7123) chip used.

The register values of the TV decoder are used to configure the TV decoder via the I2C_AV_Config block, which uses the I2C protocol to communicate with the TV decoder. The TV decoder will be unstable for a time period upon power up, and the Lock Detector block is responsible for detecting this instability.

The ITU-R 656 Decoder block extracts YcrCb 4:2:2 (YUV 4:2:2) video signals from the ITU-R 656 data stream sent from the TV decoder. It also generates a data valid control signal, which indicates the valid period of data output. De-interlacing needs to be performed on the data source because the video signal for the TV decoder is interlaced. The SDRAM Frame Buffer and a field selection multiplexer (MUX), which is controlled by the VGA Controller, are used to perform the de-interlacing operation. The VGA Controller also generates data request and odd/even selection signals to the SDRAM Frame Buffer and field selection multiplexer (MUX). The YUV422 to YUV444 block converts the selected YcrCb 4:2:2 (YUV 4:2:2) video data to the YcrCb 4:4:4 (YUV 4:4:4) video data format.

Finally, the YcrCb_to_RGB block converts the YcrCb data into RGB data output. The VGA Controller block generates standard VGA synchronous signals VGA_HS and VGA_VS to enable the display on a VGA monitor.

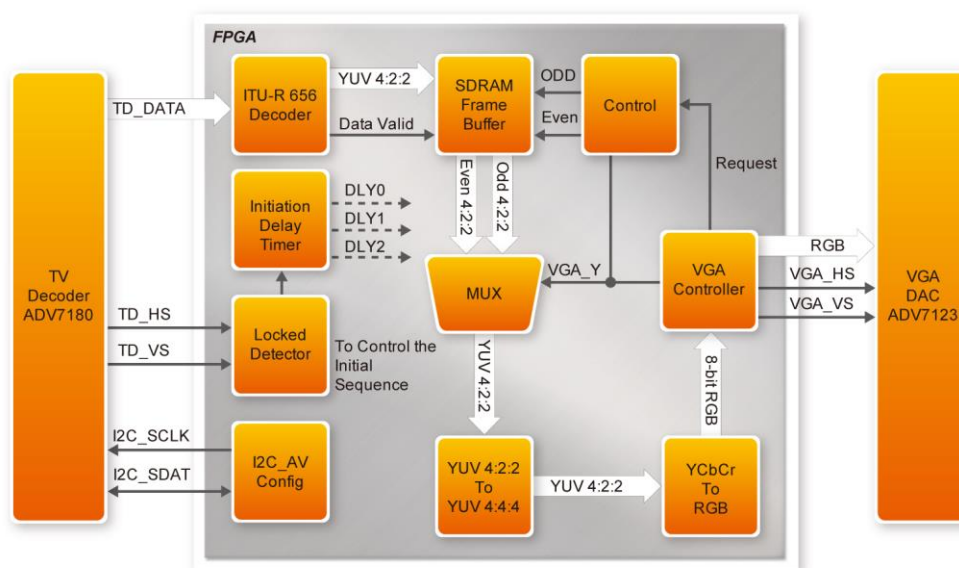


Figure 5-9 Block diagram of the TV box demonstration

Demonstration Source Code

- Project directory: DE1_SoC_TV
- Bitstream used: DE1_SoC_TV.sof

Demonstration Batch File

Demo batch directory: \DE1_SoC_TV \demo_batch

The folder includes the following files:

- Batch file: DE1_SoC_TV.bat
- FPGA configuration file : DE1_SoC_TV.sof

Demonstration Setup, File Locations, and Instructions

- Connect a DVD player's composite video output (yellow plug) to the Video-in RCA jack (J6) on the DE1-SoC board, as shown in **Figure 5-10**. The DVD player has to be configured to provide:
 - NTSC output
 - 60Hz refresh rate
 - 4:3 aspect ratio
 - Non-progressive video

- Connect the VGA output of the DE1-SoC board to a VGA monitor.
- Connect the audio output of the DVD player to the line-in port of the DE1-SoC board and connect a speaker to the line-out port. If the audio output jacks from the DVD player are RCA type, an adaptor is needed to convert to the mini-stereo plug supported on the DE1-SoC board.
- Load the bitstream into the FPGA by executing the batch file 'DE1_SoC_TV.bat' from the directory \DE1_SoC_TV\demo_batch\. Press KEY0 on the DE1-SoC board to reset the demonstration.

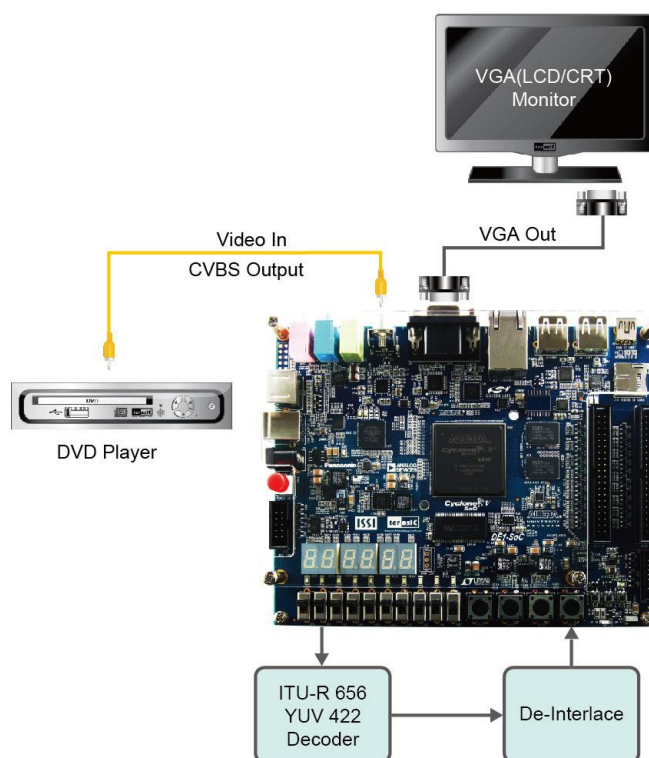


Figure 5-10 Setup for the TV box demonstration

5.7 PS/2 Mouse Demonstration

A simply PS/2 controller coded in Verilog HDL is provided to demonstrate bi-directional communication with a PS/2 mouse. A comprehensive PS/2 controller can be developed based on it and more sophisticated functions can be implemented such as setting the sampling rate or resolution, which needs to transfer two data bytes at once.

More information about the PS/2 protocol can be found on various websites.

■ Introduction

PS/2 protocol uses two wires for bi-directional communication. One is the clock line and the other one is the data line. The PS/2 controller always has total control over the transmission line, but it is the PS/2 device which generates the clock signal during data transmission.

■ Data Transmission from Device to the Controller

After the PS/2 mouse receives an enabling signal at stream mode, it will start sending out displacement data, which consists of 33 bits. The frame data is cut into three sections and each of them contains a start bit (always zero), eight data bits (with LSB first), one parity check bit (odd check), and one stop bit (always one).

The PS/2 controller samples the data line at the falling edge of the PS/2 clock signal. This is implemented by a shift register, which consists of 33 bits.

easily be implemented using a shift register of 33 bits, but be cautious with the clock domain crossing problem.

■ Data Transmission from the Controller to Device

When the PS/2 controller wants to transmit data to device, it first pulls the clock line low for more than one clock cycle to inhibit the current transmission process or to indicate the start of a new transmission process, which is usually called as inhibit state. It then pulls low the data line before releasing the clock line. This is called the request state. The rising edge on the clock line formed by the release action can also be used to indicate the sample time point as for a 'start bit'. The device will detect this succession and generates a clock sequence in less than 10ms time. The transmit data consists of 12bits, one start bit (as explained before), eight data bits, one parity check bit (odd check), one stop bit (always one), and one acknowledge bit (always zero). After sending out the parity check bit, the controller should release the data line, and the device will detect any state change on the data line in the next clock cycle. If there's no change on the data line for one clock cycle, the device will pull low the data line again as an acknowledgement which means that the data is correctly received.

After the power on cycle of the PS/2 mouse, it enters into stream mode automatically and disable data transmit unless an enabling instruction is received. **Figure 5-11** shows the waveform while communication happening on two lines.

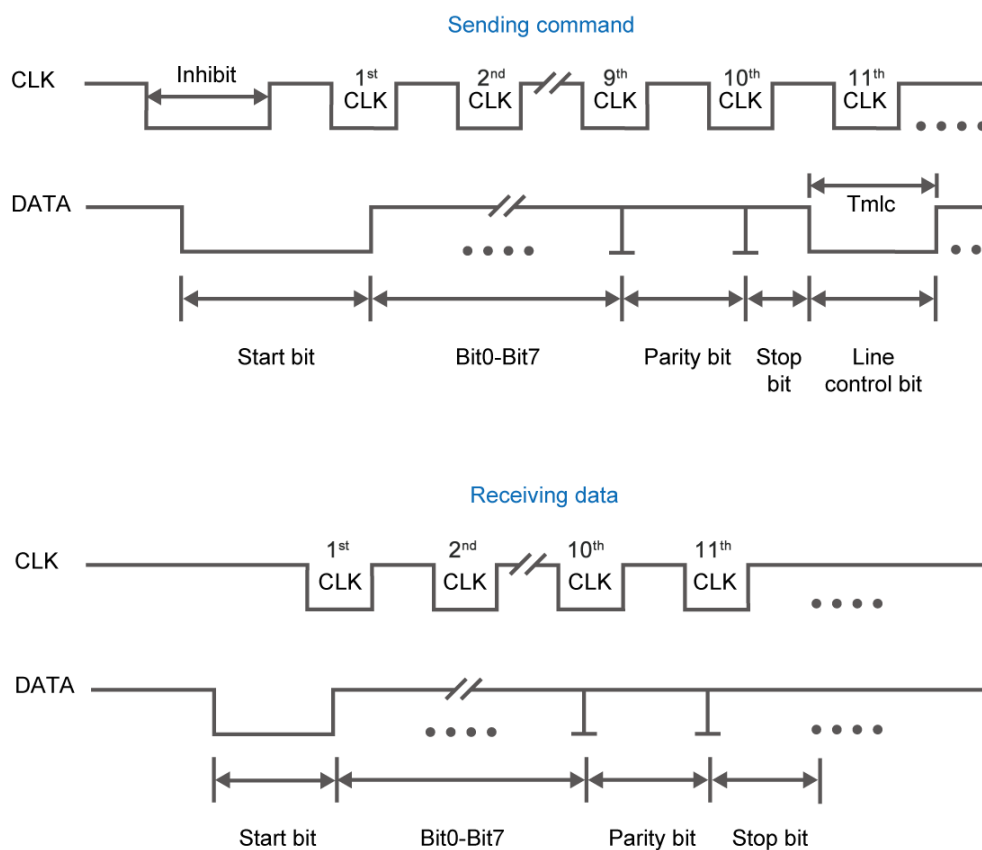


Figure 5-11 Waveform of clock and data signals during data transmission

Demonstration Source Code

- Project directory: DE1_SoC_PS2_DEMO
- Bitstream used: DE1_SoC_PS2_DEMO.sof

Demonstration Batch File

Demo batch file directoy: \DE1_SoC_PS2_DEMO \demo_batch

The folder includes the following files:

- Batch file: DE1_SoC_PS2_DEMO.bat
- FPGA configuration file : DE1_SoC_PS2_DEMO.sof

Demonstration Setup, File Locations, and Instructions

- Load the bitstream into the FPGA by executing `\DE1_SoC_PS2_DEMO\demo_batch\DE1_SoC_PS2_DEMO.bat`
- Plug in the PS/2 mouse
- Press KEY[0] to enable data transfer
- Press KEY[1] to clear the display data cache
- The 7-segment display should change when the PS/2 mouse moves. The LEDR[2:0] will blink according to **Table 5-4** when the left-button, right-button, and/or middle-button is pressed.

Table 5-4 Description of 7-segment Display and LED Indicators

<i>Indicator Name</i>	<i>Description</i>
LEDR[0]	Left button press indicator
LEDR[1]	Right button press indicator
LEDR[2]	Middle button press indicator
HEX0	Low byte of X displacement
HEX1	High byte of X displacement
HEX2	Low byte of Y displacement
HEX3	High byte of Y displacement

5.8 IR Emitter LED and Receiver Demonstration

DE1-SoC system CD has an example of using the IR Emitter LED and IR receiver. This demonstration is coded in Verilog HDL.

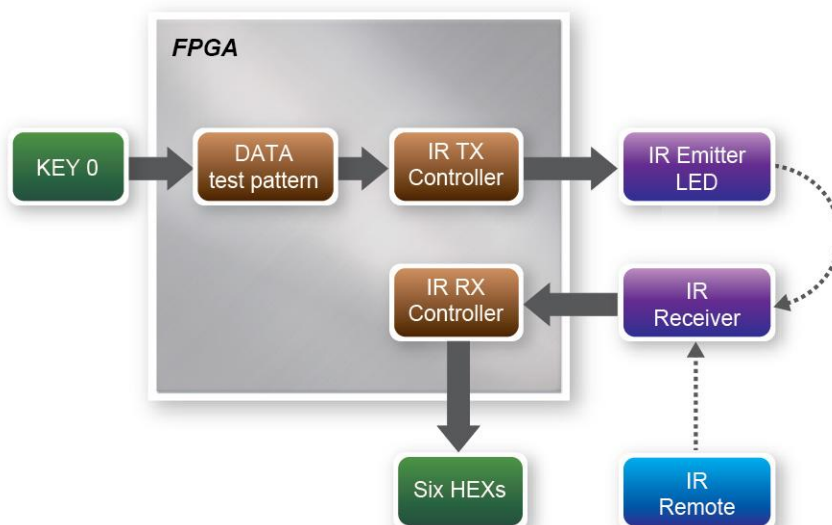


Figure 5-12 Block diagram of the IR emitter LED and receiver demonstration

Figure 5-12 shows the block diagram of the design. It implements a IR TX Controller and a IR RX Controller. When KEY0 is pressed, data test pattern generator will generate data to the IR TX Controller continuously. When IR TX Controller is active, it will format the data to be compatible with NEC IR transmission protocol and send it out through the IR emitter LED. The IR receiver will decode the received data and display it on the six HEXs. Users can also use a remote control to send data to the IR Receiver. The main function of IR TX /RX controller and IR remote control in this demonstration is described in the following sections.

■ IR TX Controller

Users can input 8-bit address and 8-bit command into the IR TX Controller. The IR TX Controller will encode the address and command first before sending it out according to NEC IR transmission protocol through the IR emitter LED. The input clock of IR TX Controller should be 50MHz.

The NEC IR transmission protocol uses pulse distance to encode the message bits. Each pulse burst is 562.5μs in length with a carrier frequency of 38kHz (26.3μs).

Figure 5-13 shows the duration of logical “1” and “0”. Logical bits are transmitted as follows:

- Logical '0' – a 562.5μs pulse burst followed by a 562.5μs space with a total transmit time of 1.125ms

- Logical '1' – a 562.5µs pulse burst followed by a 1.6875ms space with a total transmit time of 2.25ms

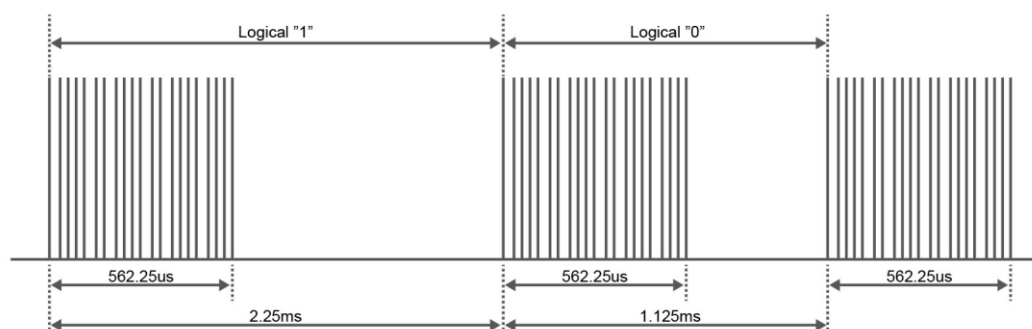


Figure 5-13 Duration of logical “1” and logical “0”

Figure 5-14 shows a frame of the protocol. Protocol sends a lead code first, which is a 9ms leading pulse burst, followed by a 4.5ms window. The second inversed data is sent to verify the accuracy of the information received. A final 562.5µs pulse burst is sent to signify the end of message transmission. Because the data is sent in pair (original and inverted) according to the protocol, the overall transmission time is constant.

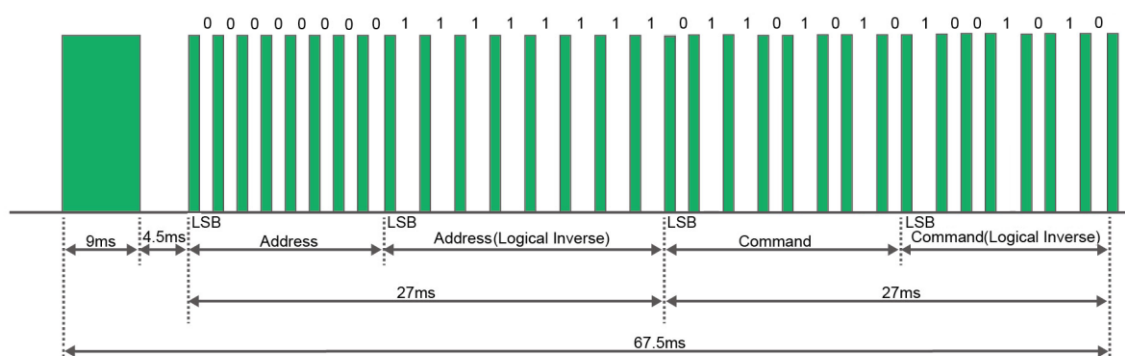


Figure 5-14 Typical frame of NEC protocol

Note: The signal received by IR Receiver is inverted. For instance, if IR TX Controller sends a lead code 9 ms high and then 4.5 ms low, IR Receiver will receive a 9 ms low and then 4.5 ms high lead code.

■ IR Remote

When a key on the remote control shown in **Figure 5-15** is pressed, the remote control will emit a standard frame, as shown in **Table 5-5**. The beginning of the frame is the lead code, which represents the start bit, followed by the key-related information. The last bit end code represents the end of the frame. The value of this frame is completely inverted at the receiving end.



Figure 5-15 The remote control used in this demonstration

Table 5-5 Key Code Information for Each Key on the Remote Control

Key	Key Code	Key	Key Code	Key	Key Code	Key	Key Code
	0x0F		0x13		0x10		0x12
	0x01		0x02		0x03		0x1A
	0x04		0x05		0x06		0x1E
	0x07		0x08		0x09		0x1B
	0x11		0x00		0x17		0x1F
	0x16		0x14		0x18		0x0C

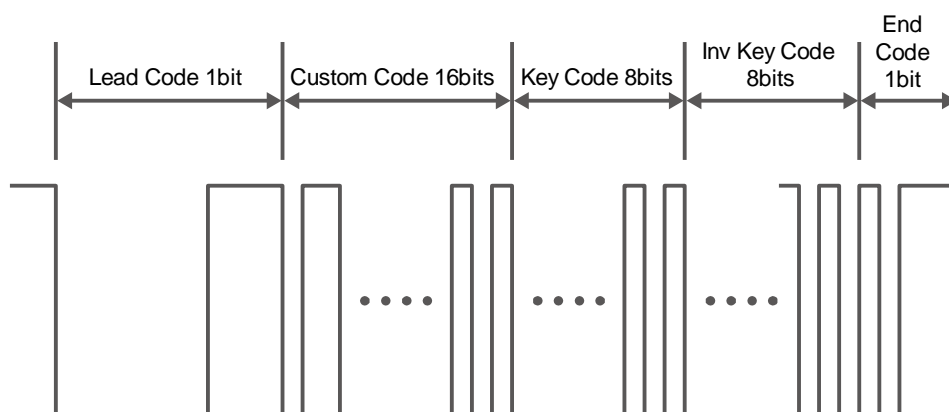


Figure 5-16 The transmitting frame of the IR remote control

■ IR RX Controller

The following demonstration shows how to implement the IP of IR receiver controller in the FPGA. **Figure 5-17** shows the modules used in this demo, including Code Detector, State Machine, and Shift Register. At the beginning the IR receiver demodulates the signal inputs to the Code Detector. The Code Detector will check the Lead Code and feedback the examination result to the State Machine.

The State Machine block will change the state from IDLE to GUIDANCE once the Lead Code is detected. If the Code Detector detects the Custom Code status, the current state will change from GUIDANCE to DATAREAD state. The Code Detector will also save the receiving data and output to the Shift Register and display on the 7-segment. **Figure 5-18** shows the state shift diagram of State Machine block. The input clock should be 50MHz.

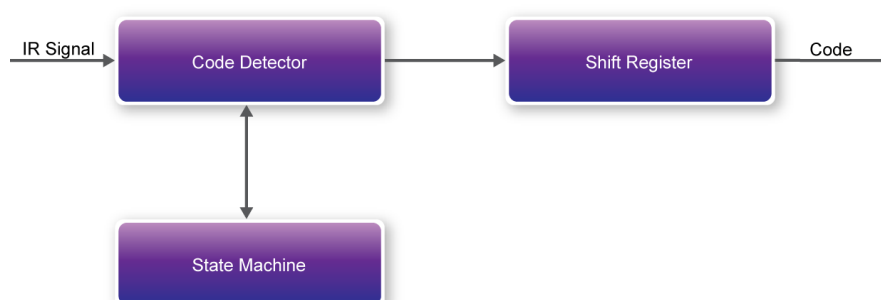


Figure 5-17 Modules in the IR Receiver controller

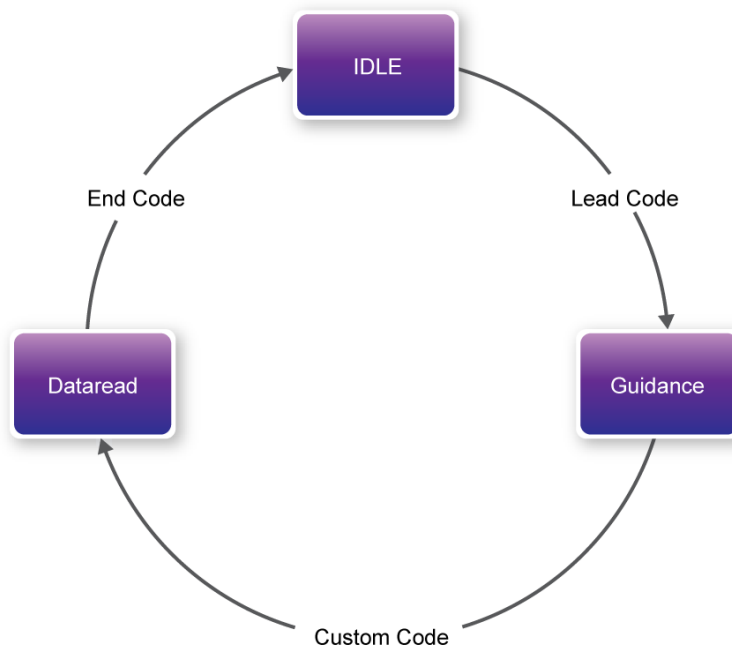


Figure 5-18 State shift diagram of State Machine block

Demonstration Source Code

- Project directory: DE1_SoC_IR
- Bitstream used: DE1_SOC_IR.sof

Demonstration Batch File

Demo batch file directory: DE1_SoC_IR \demo_batch

The folder includes the following files:

- Batch file: DE1_SoC_IR.bat
- FPGA configuration file : DE1_SOC_IR.sof

Demonstration Setup, File Locations, and Instructions

- Load the bitstream into the FPGA by executing DE1_SoC_IR \demo_batch\ DE1_SoC_IR.bat
- Keep pressing KEY[0] to enable the pattern to be sent out continuously by the IR TX Controller.
- Observe the six HEXs according to **Table 5-6**
- Release KEY[0] to stop the IR TX.
- Point the IR receiver with the remote control and press any button

- Observe the six HEXs according to **Table 5-6**

Table 5-6 Detailed Information of the Indicators

<i>Indicator Name</i>	<i>Description</i>
HEX5	Inversed high byte of DATA(Key Code)
HEX4	Inversed low byte of DATA(Key Code)
HEX3	High byte of ADDRESS(Custom Code)
HEX2	Low byte of ADDRESS(Custom Code)
HEX1	High byte of DATA(Key Code)
HEX0	Low byte of DATA (Key Code)

5.9 ADC Reading

This demonstration illustrates steps to evaluate the performance of the 8-channel 12-bit A/D Converter LTC2308. The DC 5.0V on the 2x5 header is used to drive the analog signals by a trimmer potentiometer. The voltage should be adjusted within the range between 0 and 4.096V. The 12-bit voltage measurement is displayed on the NIOS II console. **Figure 5-19** shows the block diagram of this demonstration.

The default full-scale of ADC is 0~4.096V.

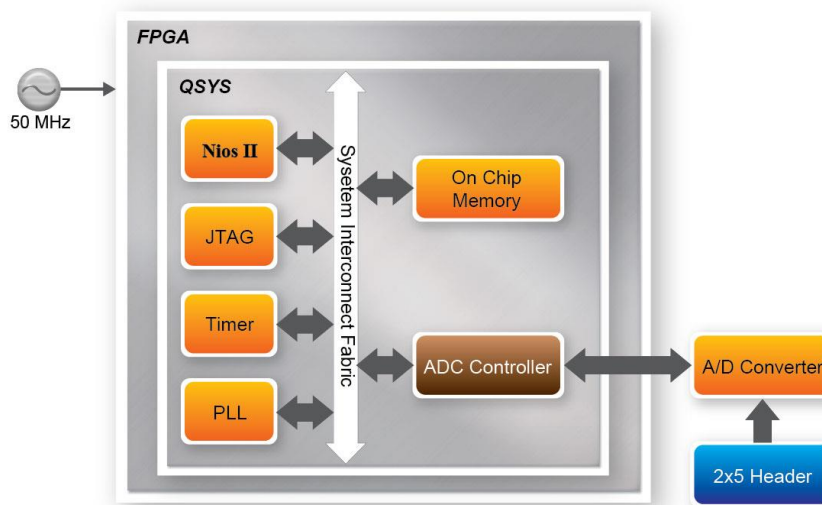


Figure 5-19 Block diagram of ADC reading

Figure 5-20 depicts the pin arrangement of the 2x5 header. This header is the input source of ADC convertor in this demonstration. Users can connect a trimmer to the specified ADC channel (ADC_IN0 ~ ADC_IN7) that provides voltage to the ADC convert. The FPGA will read the associated register in the convertor via serial interface and translates it to voltage value to be displayed on the Nios II console.

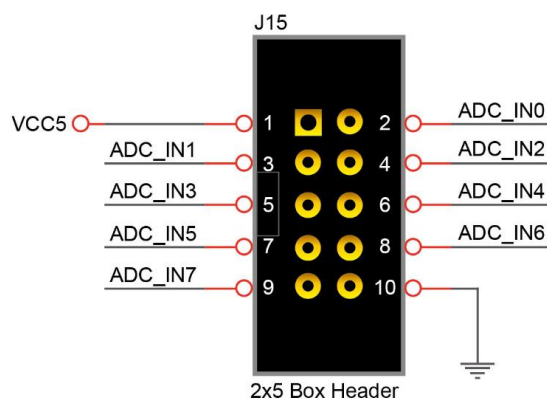


Figure 5-20 Pin distribution of the 2x5 Header for the ADC

The LTC2308 is a low noise, 500ksps, 8-channel, 12-bit ADC with an SPI/MICROWIRE compatible serial interface. The internal conversion clock allows the external serial output data clock (SCK) to operate at any frequency up to 40MHz. In this demonstration, we realized the SPI protocol in Verilog, and packet it into Avalon MM slave IP so that it can be connected to Qsys.

Figure 5-21 is SPI timing specification of LTC2308.

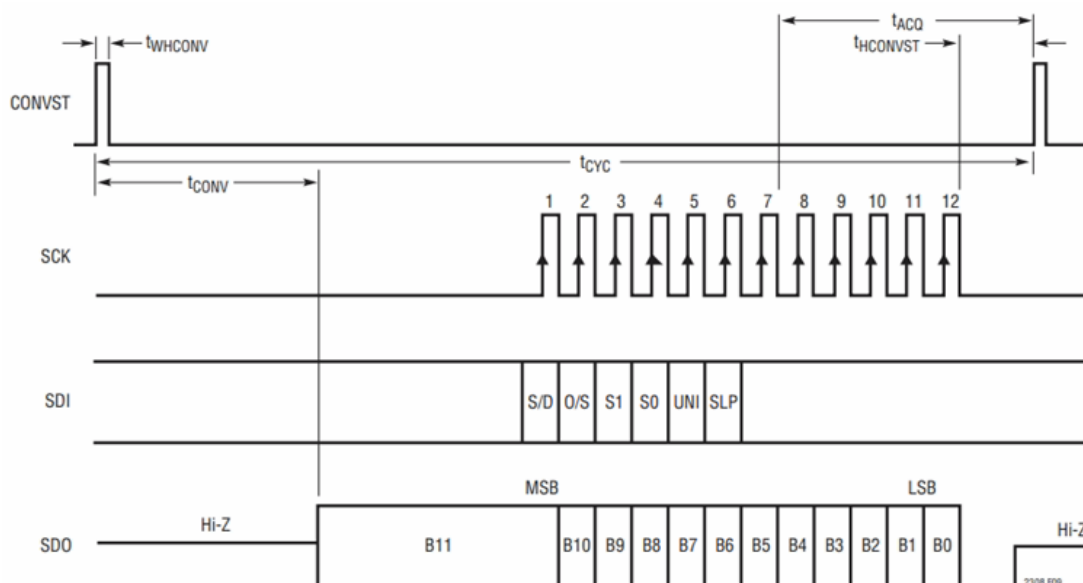


Figure 5-21 LTC2308 Timing with a Short CONVST Pulse

Important: Users should pay more attention to the impedance matching between the input source and the ADC circuit. If the source impedance of the driving circuit is low, the ADC inputs can be driven directly. Otherwise, more acquisition time should be allowed for a source with higher impedance.

To modify acquisition time t_{ACQ} , user can change the $t_{HCONVST}$ macro value in `adc_ltc2308.v`. When **SCK** is set to 40MHz, it means 25ns per unit. The default $t_{HCONVST}$ is set to 320, achieving a 100KHz f_{sample} . Thus adding more $t_{HCONVST}$ time (by increasing $t_{HCONVST}$ macro value) will lower the sample rate of the ADC Converter.

```
`define tHCONVST      320
```

Figure 5-22 shows the example MUX configurations of ADC. In this demonstration, it is configured as 8 signal-end channel in the verilog code. User can change **SW[2:0]** to measure the corresponding channel. The default reference voltage is 4.096V.

The formula of the sample voltage is:

$$\text{Sample Voltage} = \text{ADC Data} / \text{full scale Data} * \text{Reference Voltage.}$$

In this demonstration, full scale is $2^{12} = 4096$. Reference Voltage is 4.096V. Thus

$$\text{ADC Value} = \text{ADC data} / 4096 * 4.096 = \text{ADC data} / 1000$$

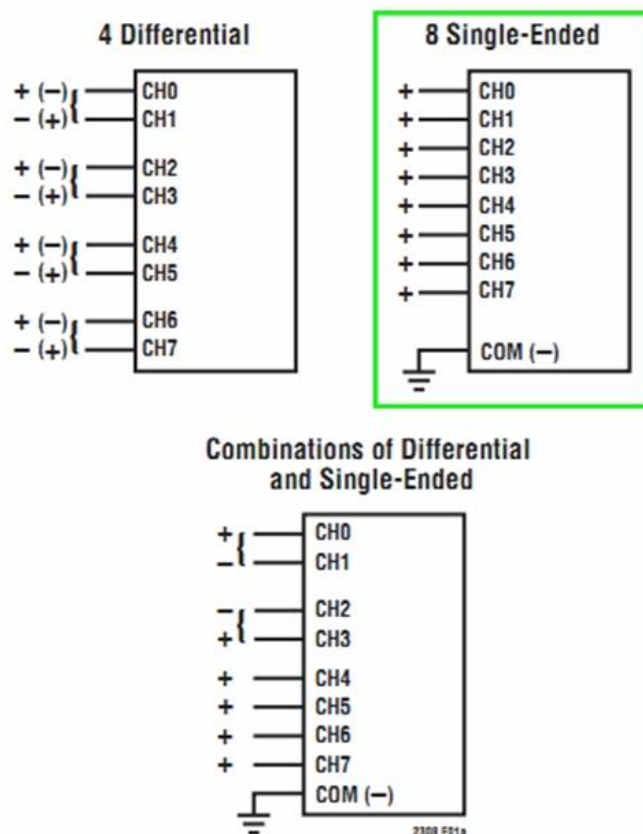


Figure 5-22 Example MUX Configurations

■ System Requirements

The following items are required for this demonstration.

- DE1-SoC board x1
- Trimmer Potentiometer x1
- Wire Strip x3

■ Demonstration File Locations

- Hardware project directory: DE1_SoC_ADC
- Bitstream used: DE1_SoC_ADC.sof
- Software project directory: DE1_SoC_ADC software
- Demo batch file : DE1_SoC_ADC\demo_batch\ DE1_SoC_ADC.bat

■ Demonstration Setup and Instructions

- Connect the trimmer to corresponding ADC channel on the 2x5 header, as shown in **Figure 5-23**, as well as the +5V and GND signals. The setup shown above is connected to ADC channel 0.
- Execute the demo batch file DE1_SoC_ADC.bat to load the bitstream and software execution file to the FPGA.
- The Nios II console will display the voltage of the specified channel voltage result information.
- Provide any input voltage to other ADC channels and set SW[2:0] to the corresponding channel if user want to measure other channels

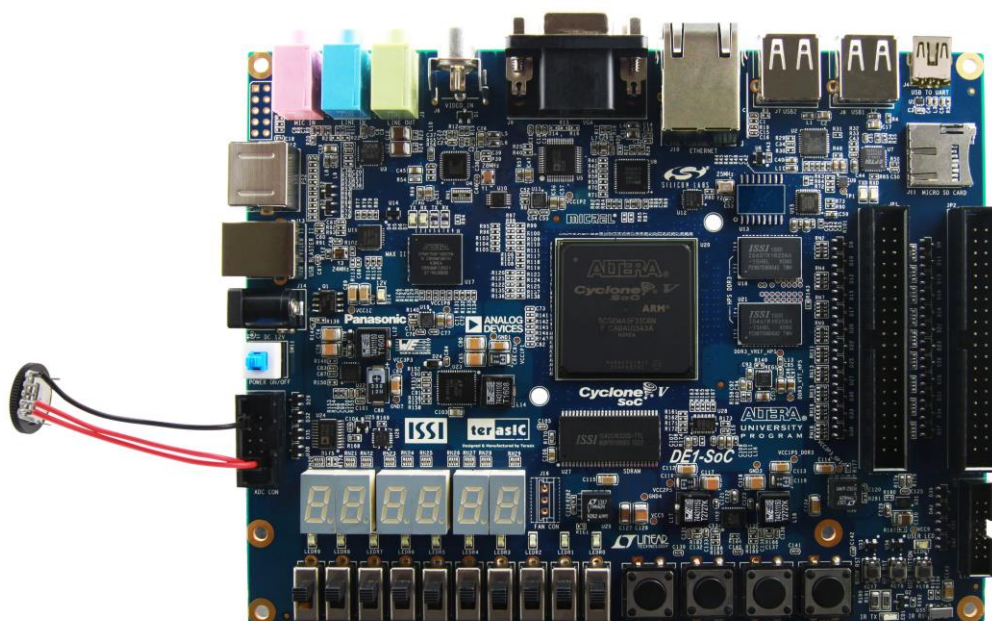


Figure 5-23 Hardware setup for the ADC reading demonstration

Chapter 6

Examples for HPS

SoC

This chapter provides several C-code examples based on the Altera SoC Linux built by Yocto project. These examples demonstrate major features connected to HPS interface on DE1-SoC board such as users LED/KEY, I2C interfaced G-sensor, and I2C MUX. All the associated files can be found in the directory *Demonstrations/SOC* of the DE1-SoC System CD. Please refer to Chapter 5 "**Running Linux on the DE1-SoC board**" from the *DE1-SoC_Getting_Started_Guide.pdf* to run Linux on DE1-SoC board.

■ Installation of the Demonstrations

To install the demonstrations on the host computer:

Copy the directory *Demonstrations* into a local directory of your choice. **Altera SoC EDS v16.0 is required for users to compile the c-code project.**

6.1 Hello Program

This demonstration shows how to develop first HPS program with Altera SoC EDS tool. Please refer to *My_First_HPS.pdf* from the system CD for more details.

The major procedures to develop and build HPS project are:

- Install Altera SoC EDS on the host PC.
- Create program .c/.h files with a generic text editor
- Create a "Makefile" with a generic text editor
- Build the project under Altera SoC EDS

■ Program File

The main program for the Hello World demonstration is:

```
#include <stdio.h>

int main(int argc, char **argv) {

    printf("Hello World!\r\n");

    return( 0 );
}
```

■ Makefile

A Makefile is required to compile a project. The Makefile used for this demo is:

```
#
TARGET = my_first_hps

#
CROSS_COMPILE = arm-linux-gnueabi-
CFLAGS = -g -Wall -I $(SOCEDS_DEST_ROOT)/ip/altera/hps/altera_hps/hwlib/include
LDFLAGS = -g -Wall
CC = $(CROSS_COMPILE)gcc
ARCH= arm

build: $(TARGET)

$(TARGET): main.o
    $(CC) $(LDFLAGS) $^ -o $@

%.o : %.c
    $(CC) $(CFLAGS) -c $< -o $@

.PHONY: clean
clean:
    rm -f $(TARGET) *.a *.o *
```

■ Compile

Please launch Altera SoC EDS Command Shell to compile a project by executing

C:\altera\16.0\embedded\Embedded_Command_Shell.bat

The "cd" command can change the current directory to where the Hello World project is located.

The "make" command will build the project. The executable file "**my_first_hps**" will be generated after the compiling process is successful. The "clean all" command removes all temporary files.

■ Demonstration Source Code

- Build tool: Altera SoC EDS v16.0
- Project directory: \Demonstration\SoC\my_first_hps
- Binary file: my_first_hps
- Build command: make ("**make clean**" to remove all temporary files)
- Execute command: ./my_first_hps

■ Demonstration Setup

- Connect a USB cable to the USB-to-UART connector (J4) on the DE1-SoC board and the host PC.
- Copy the demo file "**my_first_hps**" into a microSD card under the **"/home/root"** folder in Linux.
- Insert the booting microSD card into the DE1-SoC board.
- Power on the DE1-SoC board.
- Launch PuTTY and establish connection to the UART port of Putty. Type "**root**" to login Altera Yocto Linux.
- Type **"/my_first_hps"** in the UART terminal of PuTTY to start the program, and the "Hello World!" message will be displayed in the terminal.

```
root@socfpga:~# ./my_first_hps
Hello World!
root@socfpga:~#
```

6.2 Users LED and KEY

This demonstration shows how to control the users LED and KEY by accessing the register of GPIO controller through the memory-mapped device driver. The memory-mapped device driver allows developer to access the system physical memory.

■ Function Block Diagram

Figure 6-1 shows the function block diagram of this demonstration. The users LED and KEY are connected to the **GPIO1** controller in HPS. The behavior of GPIO controller is controlled by the register in GPIO controller. The registers can be accessed by application software through the memory-mapped device driver, which is built into Altera SoC Linux.

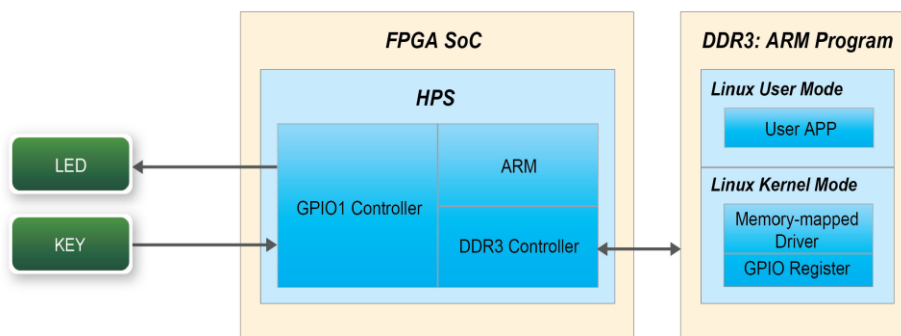


Figure 6-1 Block diagram of GPIO demonstration

■ Block Diagram of GPIO Interface

The HPS provides three general-purpose I/O (GPIO) interface modules. **Figure 6-2** shows the block diagram of GPIO Interface. GPIO[28..0] is controlled by the GPIO0 controller and GPIO[57..29] is controlled by the GPIO1 controller. GPIO[70..58] and input-only GPI[13..0] are controlled by the GPIO2 controller.

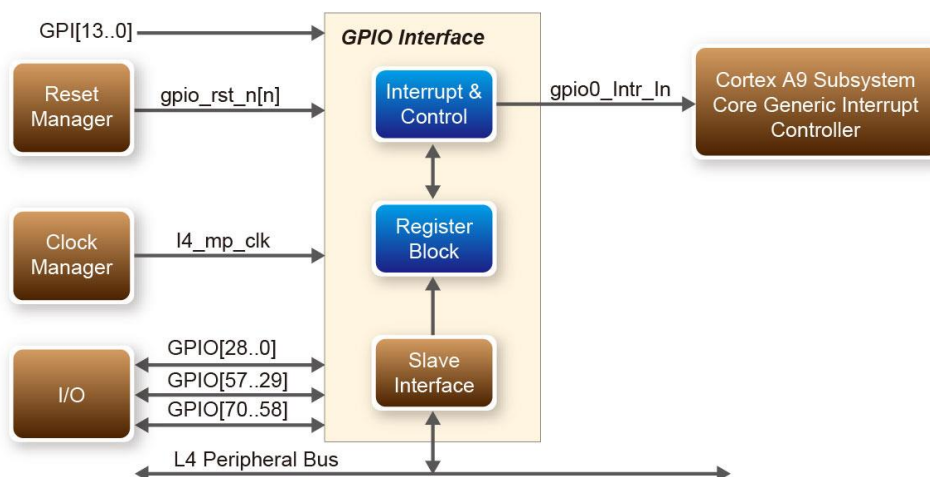


Figure 6-2 Block diagram of GPIO Interface

■ GPIO Register Block

The behavior of I/O pin is controlled by the registers in the register block. There are three 32-bit registers in the GPIO controller used in this demonstration. The registers are:

- **gpio_swporta_dr**: write output data to output I/O pin
- **gpio_swporta_ddr**: configure the direction of I/O pin
- **gpio_ext_porta**: read input data of I/O input pin

The **gpio_swporta_ddr** configures the LED pin as output pin and drives it high or low by writing data to the **gpio_swporta_dr** register. The first bit (least significant bit) of **gpio_swporta_dr** controls the direction of first IO pin in the associated GPIO controller and the second bit controls the direction of second IO pin in the associated GPIO controller and so on. The value "1" in the register bit indicates the I/O direction is output, and the value "0" in the register bit indicates the I/O direction is input.

The first bit of **gpio_swporta_dr** register controls the output value of first I/O pin in the associated GPIO controller, and the second bit controls the output value of second I/O pin in the associated GPIO controller and so on. The value "1" in the register bit indicates the output value is high, and the value "0" indicates the output value is low.

The status of KEY can be queried by reading the value of **gpio_ext_porta** register. The first bit represents the input status of first IO pin in the associated GPIO controller, and the second bit represents the input status of second IO pin in the associated GPIO controller and so on. The value "1" in the register bit indicates the input state is high, and the value "0" indicates the input state is low.

■ GPIO Register Address Mapping

The registers of HPS peripherals are mapped to HPS base address space 0xFC000000 with 64KB size. The registers of the GPIO1 controller are mapped to the base address 0xFF708000 with 4KB size, and the registers of the GPIO2 controller are mapped to the base address 0xFF70A000 with 4KB size, as shown in **Figure 6-3**.

HPS

Identifier: HPS
Access: R/W
Description: Address map for the HHP HPS system-domain

Title	Identifier	Offset
Reserved		0x0
QSPI Flash Controller Module	QSPIREGS	0xFF705000
Register		0xFF705100
Manager Module	FPC	0xFF706000
ACP ID Mapper Registers	ACPIDMAP	0xFF707000
GPIO Module	GPIO0	0xFF708000
Reserved		0xFF708080
GPIO Module	GPIO1	0xFF709000
Reserved		0xFF709080
GPIO Module	GPIO2	0xFF70A000
Reserved		0xFF70A080
L3 Cache	EGCS	0xFF800000
		0xFF880000
AND Controller Module Data (AXI Slave)	NANDC	
EMAC Module	EMAC1	0xFF702000

Figure 6-3 GPIO address map

■ Software API

Developers can use the following software API to access the register of GPIO controller.

- open: open memory mapped device driver
- mmap: map physical memory to user space
- alt_read_word: read a value from a specified register
- alt_write_word: write a value into a specified register
- munmap: clean up memory mapping
- close: close device driver.

Developers can also use the following MACRO to access the register

- alt_setbits_word: set specified bit value to one for a specified register
- alt_clrbits_word: set specified bit value to zero for a specified register

The program must include the following header files to use the above API to access the registers of GPIO controller.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
```

```
#include <sys/mman.h>
#include "hwlib.h"
#include "socal/socal.h"
#include "socal/hps.h"
#include "socal/alt_gpio.h"
```

■ LED and KEY Control

Figure 6-4 shows the HPS users LED and KEY pin assignment for the DE1_SoC board. The LED is connected to HPS_GPIO53 and the KEY is connected to HPS_GPIO54. They are controlled by the GPIO1 controller, which also controls HPS_GPIO29 ~ HPS_GPIO57.

HPS_GPIO54	G21	HPS_KEY
HPS_GPIO53	A24	HPS_LED

Figure 6-4 Pin assignment of LED and KEY

Figure 6-5 shows the **gpio_swporta_ddr** register of the GPIO1 controller. The bit-0 controls the pin direction of HPS_GPIO29. The bit-24 controls the pin direction of HPS_GPIO53, which connects to HPS_LED, the bit-25 controls the pin direction of HPS_GPIO54, which connects to HPS_KEY and so on. The pin direction of HPS_LED and HPS_KEY are controlled by the bit-24 and bit-25 in the **gpio_swporta_ddr** register of the GPIO1 controller, respectively. Similarly, the output status of HPS_LED is controlled by the bit-24 in the **gpio_swporta_dr** register of the GPIO1 controller. The status of KEY can be queried by reading the value of the bit-24 in the **gpio_ext_porta** register of the GPIO1 controller.

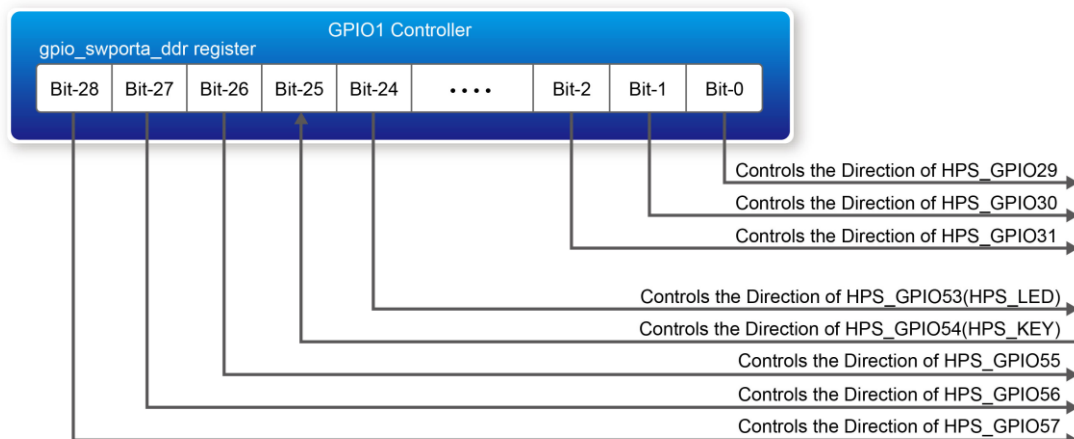


Figure 6-5 gpio_swporta_ddr register in the GPIO1 controller

The following mask is defined in the demo code to control LED and KEY direction and LED's output value.

```
#define USER_IO_DIR      (0x01000000)

#define BIT_LED           (0x01000000)

#define BUTTON_MASK      (0x02000000)
```

The following statement is used to configure the LED associated pins as output pins.

```
alt_setbits_word( ( virtual_base +
( ( uint32_t )( ALT_GPIO1_SWPORTA_DDR_ADDR ) &
( uint32_t )( HW_REGS_MASK ) ) ), USER_IO_DIR );
```

The following statement is used to turn on the LED.

```
alt_setbits_word( ( virtual_base +
( ( uint32_t )( ALT_GPIO1_SWPORTA_DR_ADDR ) &
( uint32_t )( HW_REGS_MASK ) ) ), BIT_LED );
```

The following statement is used to read the content of **gpio_ext_porta** register. The bit mask is used to check the status of the key.

```
alt_read_word( ( virtual_base +
( ( uint32_t )( ALT_GPIO1_EXT_PORTA_ADDR ) &
( uint32_t )( HW_REGS_MASK ) ) );
```

■ Demonstration Source Code

- Build tool: Altera SoC EDS V16.0
- Project directory: \Demonstration\SoC\hps_gpio
- Binary file: hps_gpio
- Build command: make ('make clean' to remove all temporal files)
- Execute command: ./hps_gpio

■ Demonstration Setup

- Connect a USB cable to the USB-to-UART connector (J4) on the DE1-SoC board and the host PC.
- Copy the executable file "**hps_gpio**" into the microSD card under the **"/home/root"** folder in Linux.
- Insert the booting micro SD card into the DE1-SoC board.
- Power on the DE1-SoC board.
- Launch PuTTY and establish connection to the UART port of Putty. Type **"root"** to login Altera Yocto Linux.
- Type **"./hps_gpio "** in the UART terminal of PuTTY to start the program.

```
root@socfpga:~# ./hps_gpio
led test
the led flash 2 times
user key test
press key to control led
```

- HPS_LED will flash twice and users can control the user LED with push-button.
- Press HPS_KEY to light up HPS_LED.
- Press "CTRL + C" to terminate the application.

6.3 I2C Interfaced G-sensor

This demonstration shows how to control the G-sensor by accessing its registers through the built-in I2C kernel driver in [Altera Soc Yocto Powered Embedded Linux](#).

■ Function Block Diagram

Figure 6-6 shows the function block diagram of this demonstration. The G-sensor on the DE1_SoC board is connected to the **I2C0** controller in HPS. The G-Sensor I2C 7-bit device address is 0x53. The system I2C bus driver is used to access the register files in the G-sensor. The G-sensor interrupt

signal is connected to the PIO controller. This demonstration uses polling method to read the register data.

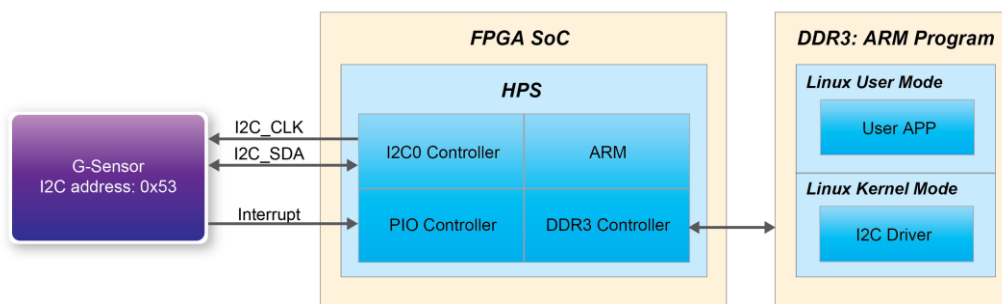


Figure 6-6 Block diagram of the G-sensor demonstration

■ I2C Driver

The procedures to read a register value from G-sensor register files by the existing I2C bus driver in the system are:

1. Open I2C bus driver "/dev/i2c-0": `file = open("/dev/i2c-0", O_RDWR);`
2. Specify G-sensor's I2C address 0x53: `ioctl(file, I2C_SLAVE, 0x53);`
3. Specify desired register index in g-sensor: `write(file, &Addr8, sizeof(unsigned char));`
4. Read one-byte register value: `read(file, &Data8, sizeof(unsigned char));`

The G-sensor I2C bus is connected to the I2C0 controller, as shown in the [Figure 6-7](#). The driver name given is '/dev/i2c-0'.

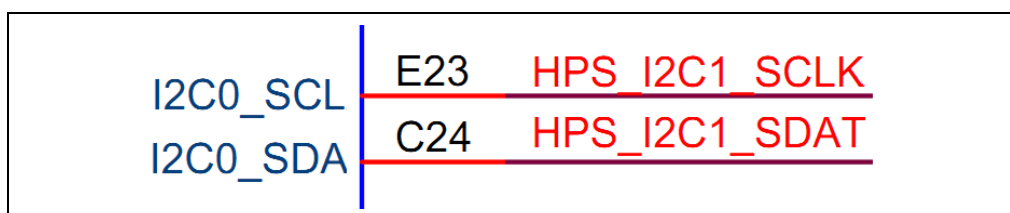


Figure 6-7 Connection of HPS I2C signals

The step 4 above can be changed to the following to write a value into a register.

`write(file, &Data8, sizeof(unsigned char));`

The step 4 above can also be changed to the following to read multiple byte values.

read(file, &szData8, sizeof(szData8)); // where szData is an array of bytes

The step 4 above can be changed to the following to write multiple byte values.

write(file, &szData8, sizeof(szData8)); // where szData is an array of bytes

■ G-sensor Control

The ADI ADXL345 provides I2C and SPI interfaces. I2C interface is selected by setting the CS pin to high on the DE1_SoC board.

The ADI ADXL345 G-sensor provides user-selectable resolution up to 13-bit $\pm 16g$. The resolution can be configured through the DATA_FORMAT(0x31) register. The data format in this demonstration is configured as:

- Full resolution mode
- $\pm 16g$ range mode
- Left-justified mode

The X/Y/Z data value can be derived from the DATA0(0x32), DATA1(0x33), DATA2(0x34), DATA3(0x35), DATA4(0x36), and DATA5(0x37) registers. The DATA0 represents the least significant byte and the DATA1 represents the most significant byte. It is recommended to perform multiple-byte read of all registers to prevent change in data between sequential registers read. The following statement reads 6 bytes of X, Y, or Z value.

read(file, szData8, sizeof(szData8)); // where szData is an array of six-bytes

■ Demonstration Source Code

- Build tool: Altera SoC EDS v16.0
- Project directory: \Demonstration\SoC\hps_gsensor
- Binary file: gsensor
- Build command: make ('make clean' to remove all temporal files)
- Execute command: ./gsensor [loop count]

■ Demonstration Setup

- Connect a USB cable to the USB-to-UART connector (J4) on the DE1-SoC board and the host PC.

- Copy the executable file "**gsensor**" into the microSD card under the **"/home/root"** folder in Linux.
- Insert the booting microSD card into the DE1-SoC board.
- Power on the DE1-SoC board.
- Launch PuTTY to establish connection to the UART port of DE1-SoC board. Type **"root"** to login Yocto Linux.
- Execute **"/gsensor"** in the UART terminal of PuTTY to start the G-sensor polling.
- The demo program will show the X, Y, and Z values in the PuTTY, as shown in **Figure 6-8**.

```
root@socfpga:~# ./gsensor
===== gsensor test =====
id=E5h
[1]X=80 mg, Y=-40 mg, Z=924 mg
[2]X=76 mg, Y=-32 mg, Z=972 mg
[3]X=76 mg, Y=-36 mg, Z=964 mg
[4]X=84 mg, Y=-36 mg, Z=976 mg
[5]X=76 mg, Y=-40 mg, Z=964 mg
[6]X=76 mg, Y=-40 mg, Z=972 mg
```

Figure 6-8 Terminal output of the G-sensor demonstration

- Press "CTRL + C" to terminate the program.

6.4 I2C MUX Test

The I2C bus on DE1-SoC is originally accessed by FPGA only. This demonstration shows how to switch the I2C multiplexer for HPS to access the I2C bus.

■ Function Block Diagram

Figure 6-9 shows the function block diagram of this demonstration. The I2C bus from both FPGA and HPS are connected to an I2C multiplexer. It is controlled by HPS_I2C_CONTROL, which is connected to the **GPIO1** controller in HPS. The HPS I2C is connected to the **I2C0** controller in HPS, as well as the G-sensor.

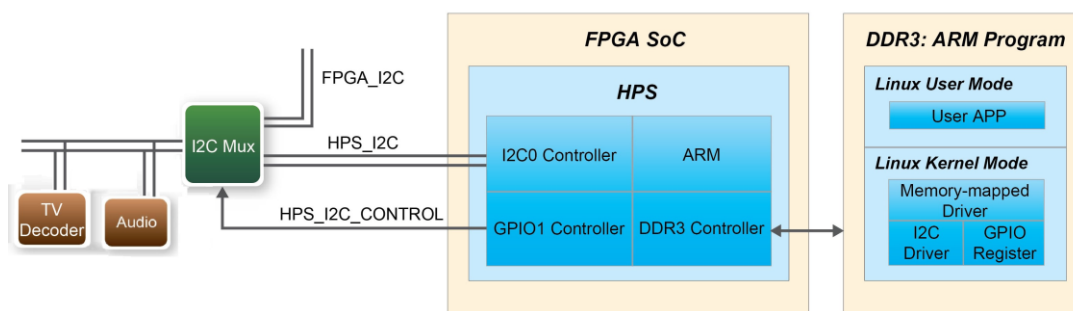


Figure 6-9 Block diagram of the I2C MUX test demonstration

■ HPS_I2C_CONTROL Control

HPS_I2C_CONTROL is connected to HPS_GPIO48, which is bit-19 of the **GPIO1** controller. Once HPS gets access to the I2C bus, it can then access Audio CODEC and TV Decoder when the HPS_I2C_CONTROL signal is set to high.

The following mask in the demo code is defined to control the direction and output value of HPS_I2C_CONTROL.

```
#define HPS_I2C_CONTROL    ( 0x00080000 )
```

The following statement is used to configure the HPS_I2C_CONTROL associated pins as output pin.

```
alt_setbits_word( ( virtual_base +
( ( uint32_t )( ALT_GPIO1_SWPORTA_DDR_ADDR ) &
( uint32_t )( HW_REGS_MASK ) ) ), HPS_I2C_CONTROL );
```

The following statement is used to set HPS_I2C_CONTROL high.

```
alt_setbits_word( ( virtual_base +
( ( uint32_t )( ALT_GPIO1_SWPORTA_DR_ADDR ) &
( uint32_t )( HW_REGS_MASK ) ) ), HPS_I2C_CONTROL );
```

The following statement is used to set HPS_I2C_CONTROL low.

```
alt_clrbits_word( ( virtual_base +
( ( uint32_t )( ALT_GPIO1_SWPORTA_DR_ADDR ) &
( uint32_t )( HW_REGS_MASK ) ) ), HPS_I2C_CONTROL );
```

■ I2C Driver

The procedures to read register value from TV Decoder by the existing I2C bus driver in the system are:

- Set HPS_I2C_CONTROL high for HPS to access I2C bus.
- Open the I2C bus driver "/dev/i2c-0": file = open("/dev/i2c-0", O_RDWR);
- Specify the I2C address 0x20 of ADV7180: ioctl(file, I2C_SLAVE, 0x20);
- Read or write registers;
- Set HPS_I2C_CONTROL low to release the I2C bus.

■ Demonstration Source Code

- Build tool: Altera SoC EDS v16.0
- Project directory: \Demonstration\SoC\ hps_i2c_switch
- Binary file: i2c_switch
- Build command: make ('make clean' to remove all temporal files)
- Execute command: ./ i2c_switch

■ Demonstration Setup

- Connect a USB cable to the USB-to-UART connector (J4) on the DE1-SoC board and host PC.
- Copy the executable file " **i2c_switch** " into the microSD card under the "/home/root" folder in Linux.
- Insert the booting microSD card into the DE1-SoC board.
- Power on the DE1-SoC board.
- Launch PuTTY to establish connection to the UART port of DE1_SoC board. Type "**root**" to login Yocto Linux.
- Execute "**./ i2c_switch** " in the UART terminal of PuTTY to start the I2C MUX test.
- The demo program will show the result in the Putty, as shown in **Figure 6-10**.

```
root@socfpga:~# ./i2c_switch
I2C BUS Switch Test
HPS owns the I2C bus!!!
Open '/dev/i2c-0' successfully.
REG[11h]=1Ch (i2c addr:20h)
HPS release the I2C bus!!!
I2C Switch Test:Success
root@socfpga:~#
```

Figure 6-10 Terminal output of the I2C MUX Test Demonstration

- Press "CTRL + C" to terminate the program.

Chapter 7

Examples for using both HPS SoC and FPGA

Although HPS and FPGA can operate independently, they are tightly coupled via a high-bandwidth system interconnect built from high-performance ARM AMBA® AXITM bus bridges. Both FPGA fabric and HPS can access to each other via these interconnect bridges. This chapter provides demonstrations on how to achieve superior performance and lower latency through these interconnect bridges when comparing to solutions containing a separate FPGA and discrete processor.

7.1 HPS Control LED and HEX

This demonstration shows how HPS controls the FPGA LED and HEX through Lightweight HPS-to-FPGA Bridge. The FPGA is configured by HPS through FPGA manager in HPS.

■ A brief view on FPGA manager

The FPGA manager in HPS configures the FPGA fabric from HPS. It also monitors the state of FPGA and drives or samples signals to or from the FPGA fabric. The application software is provided to configure FPGA through the FPGA manager. The FPGA configuration data is stored in the file with .rbf extension. The MSEL[4:0] must be set to 01010 or 01110 before executing the application software on HPS.

■ Function Block Diagram

Figure 7-1 shows the block diagram of this demonstration. The HPS uses Lightweight HPS-to-FPGA AXI Bridge to communicate with FPGA. The hardware in FPGA part is built into

Qsys. The data transferred through Lightweight HPS-to-FPGA Bridge is converted into Avalon-MM master interface. Both PIO Controller and HEX Controller work as Avalon-MM slave in the system. They control the associated pins to change the state of LED and HEX. This is similar to a system using Nios II processor to control LED and HEX.

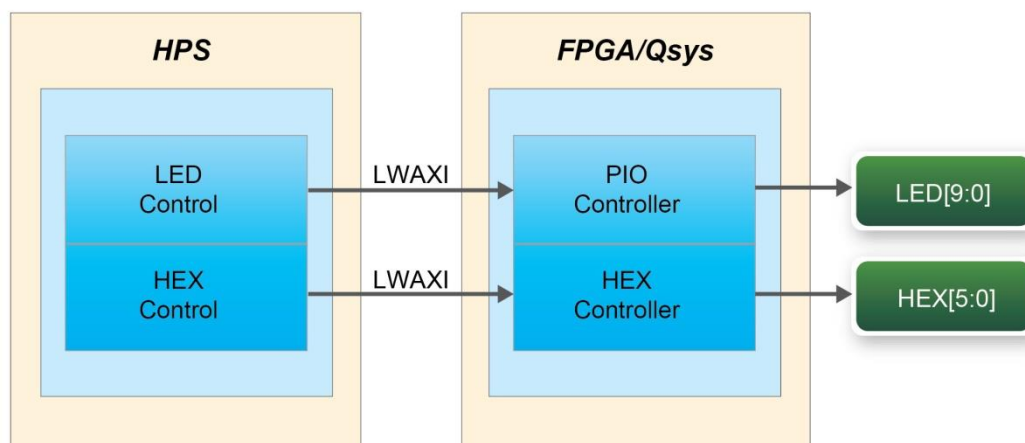


Figure 7-1 FPGA LED and HEX are controlled by HPS

■ LED and HEX control

The Lightweight HPS-to-FPGA Bridge is a peripheral of HPS. The software running on Linux cannot access the physical address of the HPS peripheral. The physical address must be mapped to the user space before the peripheral can be accessed. Alternatively, a customized device driver module can be added to the kernel. The entire CSR span of HPS is mapped to access various registers within that span. The mapping function and the macro defined below can be reused if any other peripherals whose physical address is also in this span.

```
#define HW_REGS_BASE ( ALT_STM_OFST )
#define HW_REGS_SPAN ( 0x04000000 )
#define HW_REGS_MASK ( HW_REGS_SPAN - 1 )
```

The start address of Lightweight HPS-to-FPGA Bridge after mapping can be retrieved by ALT_LWFPGASLVS_OFST, which is defined in altera_hps hardware library. The slave IP connected to the bridge can then be accessed through the base address and the register offset in these IPs. For instance, the base address of the PIO slave IP in this system is 0x0001_0040, the direction control register offset is 0x01, and the data register offset is 0x00. The following statement is used to retrieve the base address of PIO slave IP.

```
h2p_lw_led_addr=virtual_base+( ( unsigned long )( ALT_LWFPGASLVS_OFST
+ LED_PIO_BASE ) & ( unsigned long)( HW_REGS_MASK ) );
```


Considering this demonstration only needs to set the direction of PIO as output, which is the default direction of the PIO IP, the step above can be skipped. The following statement is used to set the output state of the PIO.

```
alt_write_word(h2p_lw_led_addr, Mask );
```

The Mask in the statement decides which bit in the data register of the PIO IP is high or low. The bits in data register decide the output state of the pins connected to the LEDs. The HEX controlling part is similar to the LED.

Since Linux supports multi-thread software, the software for this system creates two threads. One controls the LED and the other one controls the HEX. The system calls `pthread_create`, which is called in the main function to create a sub-thread, to complete the job. The program running in the sub-thread controls the LED flashing in a loop. The main-thread in the main function controls the digital shown on the HEX that keeps changing in a loop. The state of LED and HEX state change simultaneously when the FPGA is configured and the software is running on HPS.

■ Demonstration Source Code

- Build tool: Altera SoC EDS V16.0
- Project directory: \Demonstration\ SoC_FPGA\HPS_LED_HEX
- Quick file directory: \ Demonstration\ SoC_FPGA\HPS_LED_HEX\ quickfile
- FPGA configuration file : soc_system_dc.rbf
- Binary file: HPS_LED_HEX and hps_config_fpga
- Build app command: make ('make clean' to remove all temporal files)
- Execute app command: ./hps_config_fpga soc_system_dc.rbf and ./HPS_LED_HEX

■ Demonstration Setup

- Quartus II and Nios II must be installed on the host PC.
- The MSEL[4:0] is set to 01010 or 01110.
- Connect a USB cable to the USB-Blaster II connector (J13) on the DE1-SoC board and the host PC. Install the USB-Blaster II driver if necessary.
- Connect a USB cable to the USB-to-UART connector (J4) on the DE1-SoC board and the host PC.
- Copy the executable files "hps_config_fpga" and "HPS_LED_HEX", and the FPGA configuration file "soc_system_dc.rbf" into the microSD card under the **"/home/root"** folder in Linux.
- Insert the booting microSD card into the DE1-SoC board. Please refer to the chapter 5

"Running Linux on the DE1-SoC board" on *DE1-SoC_Getting_Started_Guide.pdf* on how to build a booting microSD card image.

- Power on the DE1-SoC board.
- Launch PuTTY to establish connection to the UART port of the DE1-SoC board. Type "**root**" to login Altera Yocto Linux.
- Execute `./hps_config_fpga soc_system_dc.rbf` in the UART terminal of PuTTY to configure the FPGA through the FPGA manager. After the configuration is successful, the message shown in **Figure 7-2** will be displayed in the terminal.

```
root@socfpga:~# ./hps_config_fpga soc_system_dc.rbf
INFO: alt_fpga_control_enable().
INFO: alt_fpga_control_enable OK.
alt_fpga_control_enable OK next config the fpga
INFO: MSEL configured correctly for FPGA image.
soc_system_dc.rbf file open success
INFO: FPGA Image binary at 0x72c1c008.
INFO: FPGA Image size is 2309848 bytes.
INFO: alt_fpga_configure() successful on the 1 of 5 retry(s).
INFO: alt_fpga_control_disable().
```

Figure 7-2 Running the application to configure the FPGA

- Execute `./HPS_LED_HEX` in the UART terminal of PuTTY to start the program.
- The message shown in **Figure 7-3** will be displayed in the terminal. The LED[9:0] will be flashing and the number on the HEX[5:0] will keep changing simultaneously.

```
LED ON
hex show 8
hex show 9
hex show A
LED OFF
hex show B
hex show C
LED ON
hex show D
hex show E
LED OFF
hex show F
hex show 0
LED ON
hex show 1
hex show 2
LED OFF
hex show 3
hex show 4
hex show 5
LED ON
```

Figure 7-3 Running result in the terminal of PuTTY

- Press "CTRL + C" to terminate the program.

7.2 DE1-SoC Control Panel

The DE1-SoC Control Panel is a more comprehensive example. It demonstrates:

- Control HPS LED and FPGA LED/HEX
- Query the status of buttons connected to HPS and FPGA
- Configure and query G-sensor connected to HPS
- Control Video-in and VGA-out connected to FPGA
- Control IR receiver connected to FPGA

This example not only controls the peripherals of HPS and FPGA, but also shows how to implement a GUI program on Linux. [Figure 7-4](#) [OLE LINK4](#) is the screenshot of DE1-SOC Control Panel.



Figure 7-4 Screenshot of DE1-SoC Control Panel

Please refer to **DE1-SoC_Control_Panel.pdf**, which is included in the DE1-SOC System CD for more information on how to build a GUI program step by step.

7.3 DE1-SoC Linux Frame Buffer Project

The DE1-SoC Linux Frame Buffer Project is an example that a VGA monitor is utilized as a standard output interface for the Linux operating system. The Quartus II project is located at this path: Demonstrations/SOC_FPGA/DE1_SOC_Linux_FB. The soc_system.rbf file in the project is used for configuring FPGA through HPS. The .rbf file is converted from DE1_SOC_Linux_FB.sof by clicking the sof_to_rbf.bat. The project is adopted for the following demonstrations.

- DE1_SoC Linux Console with framebuffer
- DE1_SoC LXDE with Desktop
- DE1_SoC Ubuntu Desktop

The SD image file for the demonstrations above can be downloaded in the design resources for DE1-SoC at Terasic website.

These examples provide a GUI environment for further developing for the users. For example, a QT application can run on the system.



Figure 7-5 Screenshot of DE1-SoC Linux Console with framebuffer

Please refer to DE1-SoC_Getting_Started_Guide about how to get the SD images and create a boot SD card.

Chapter 8

Programming the EPCS Device

This chapter describes how to program the quad serial configuration (EPCS) device with Serial Flash Loader (SFL) function via the JTAG interface. Users can program EPCS devices with a JTAG indirect configuration (.jic) file, which is converted from a user-specified SRAM object file (.sof) in Quartus. The .sof file is generated after the project compilation is successful. The steps of converting .sof to .jic in Quartus II are listed below.

8.1 Before Programming Begins

The FPGA should be set to AS x1 mode i.e. MSEL[4..0] = “10010” to use the quad Flash as a FPGA configuration device.

8.2 Convert .SOF File to .JIC File

1. Choose **Convert Programming Files** from the File menu of Quartus II, as shown in **Figure 8-1**.

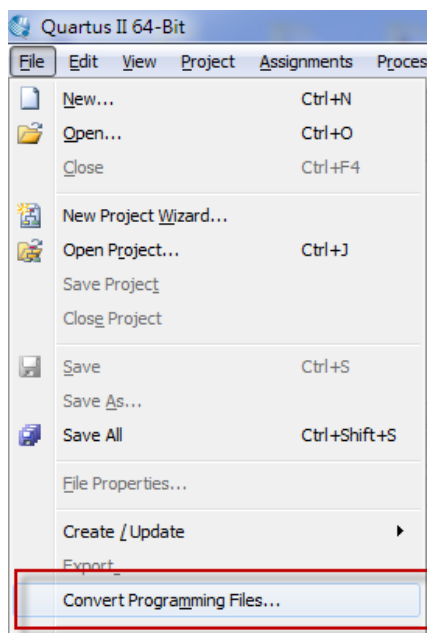


Figure 8-1 File menu of Quartus II

2. Select **JTAG Indirect Configuration File (.jic)** from the **Programming file type** field in the dialog of Convert Programming Files.
3. Choose **EPCS128** from the **Configuration device** field.
4. Choose **Active Serial** from the **Mode** field.
5. Browse to the target directory from the **File name** field and specify the name of output file.
6. Click on the **SOF data** in the section of **Input files to convert**, as shown in **Figure 8-2**.

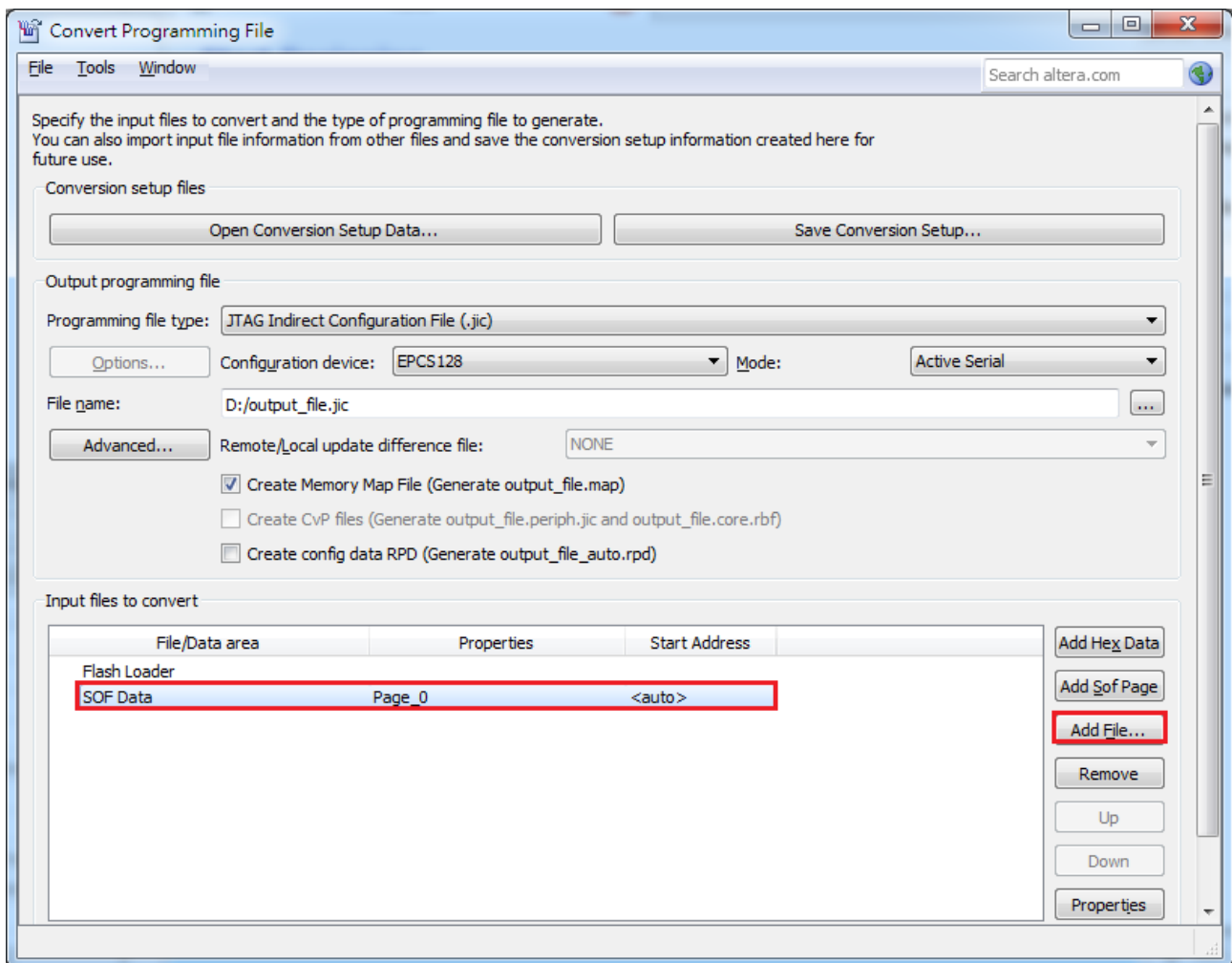


Figure 8-2 Dialog of “Convert Programming Files”

7. Click **Add File**.
8. Select the .sof to be converted to a .jic file from the Open File dialog.
9. Click **Open**.
10. Click on the **Flash Loader** and click **Add Device**, as shown in [Figure 8-3](#).
11. Click **OK** and the **Select Devices** page will appear.

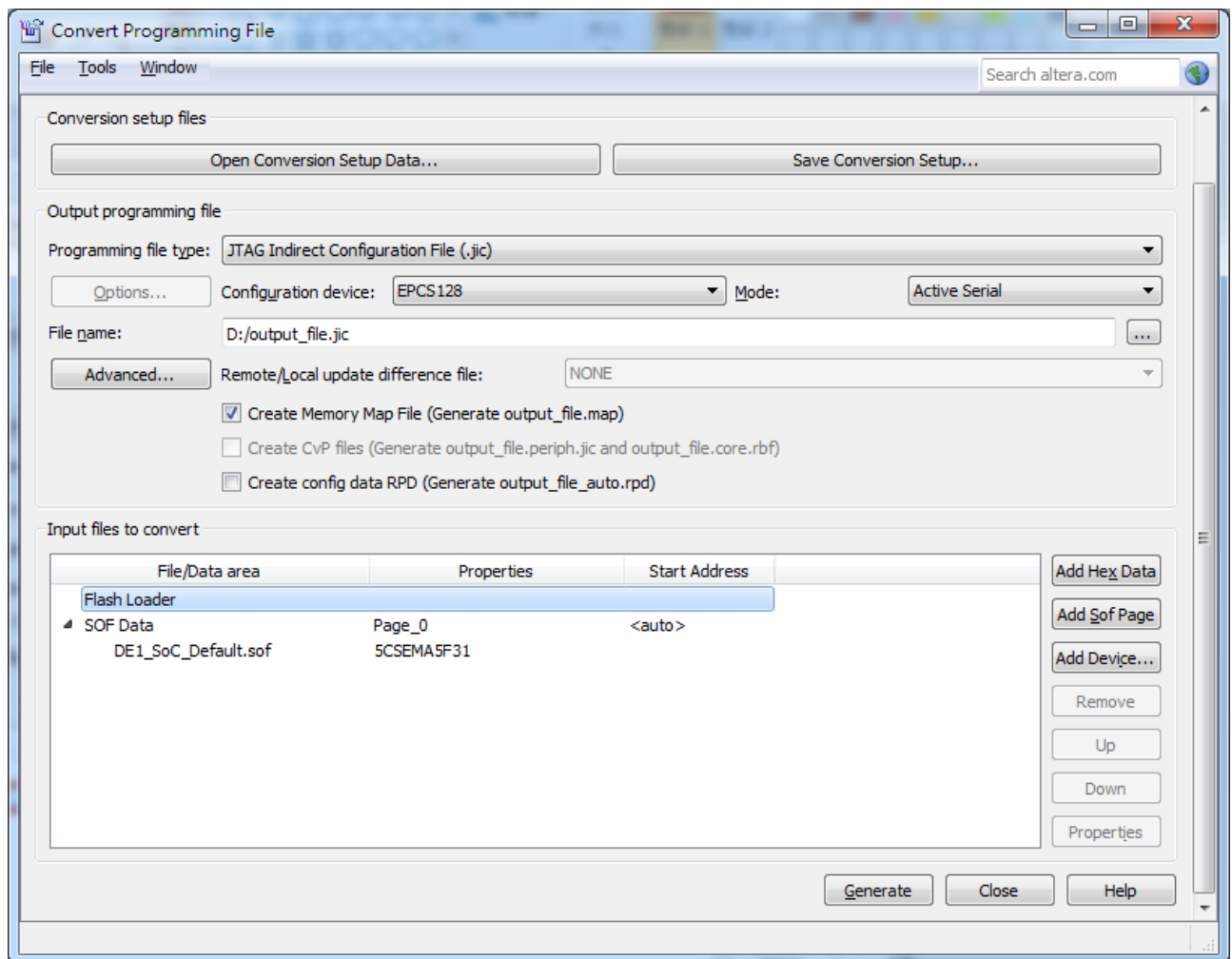


Figure 8-3 Click on the “Flash Loader”

12. Select the targeted FPGA to be programmed into the EPCS, as shown in **Figure 8-4**.
13. Click **OK** and the **Convert Programming Files** page will appear, as shown in **Figure 8-5**.
14. Click **Generate**.

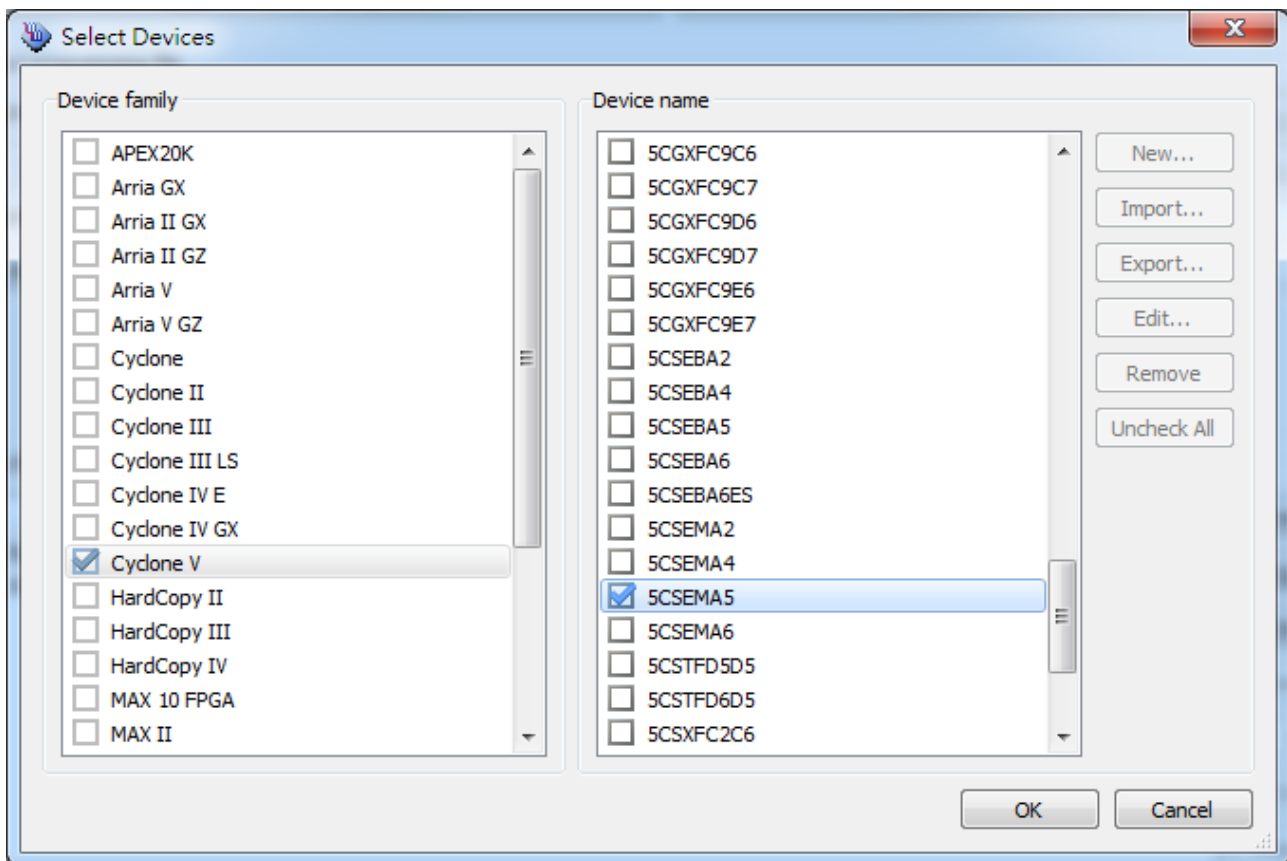


Figure 8-4 “Select Devices” page

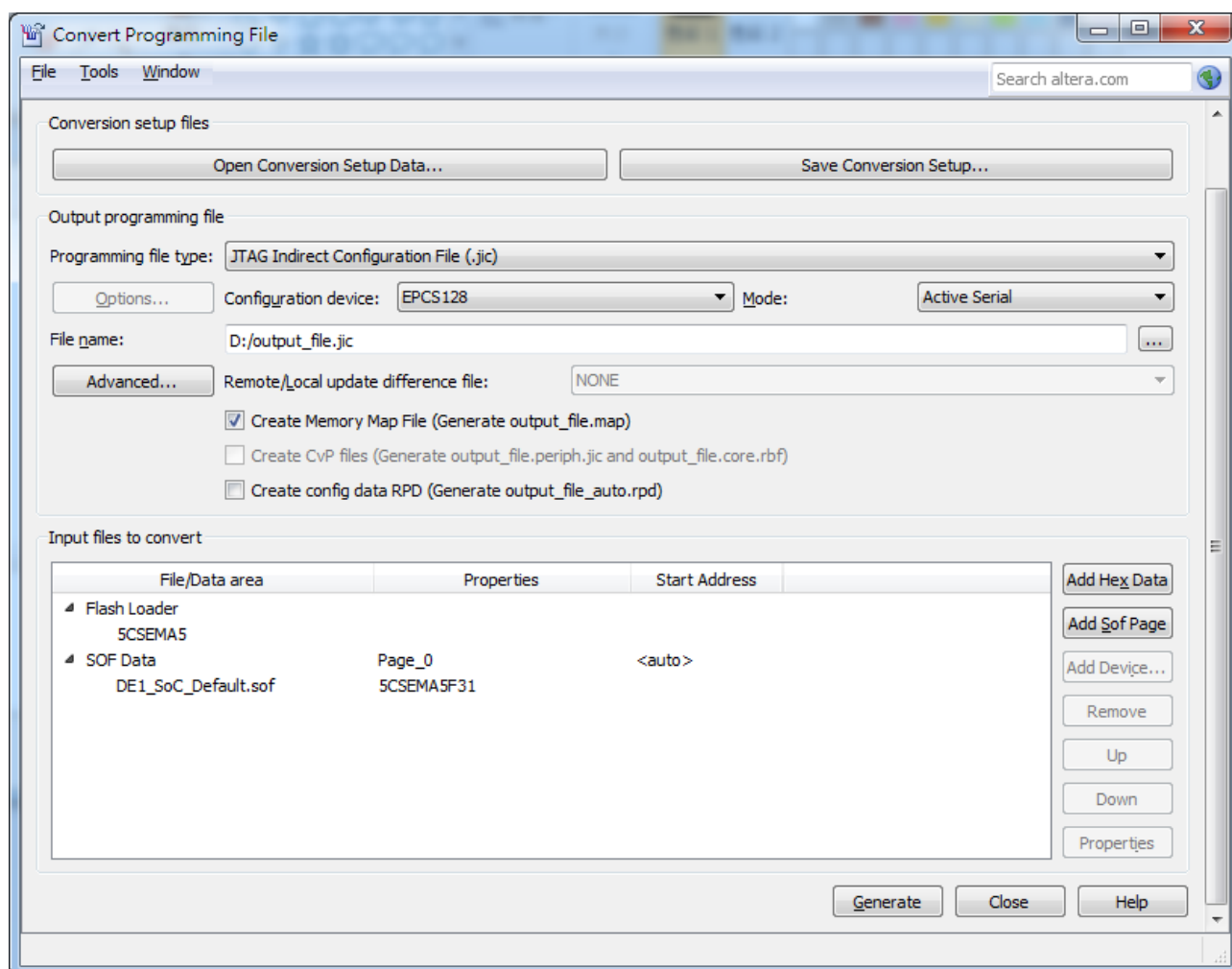


Figure 8-5 “Convert Programming Files” page after selecting the device

8.3 Write JIC File into the EPCS Device

When the conversion of SOF-to-JIC file is complete, please follow the steps below to program the EPCS device with the .jic file created in Quartus II Programmer.

1. Set MSEL[4..0] = “10010”
2. Choose **Programmer** from the Tools menu and the **Chain.cdf** window will appear.
3. Click **Auto Detect** and then select the correct device. Both FPGA device and HPS should be detected, as shown in **Figure 8-6**.

4. Double click the green rectangle region shown in **Figure 8-6** and the **Select New Programming File** page will appear. Select the .jic file to be programmed.
5. Program the EPCS device by clicking the corresponding **Program/Configure** box. A factory default SFL image will be loaded, as shown in **Figure 8-7**.
6. Click **Start** to program the EPCS device.

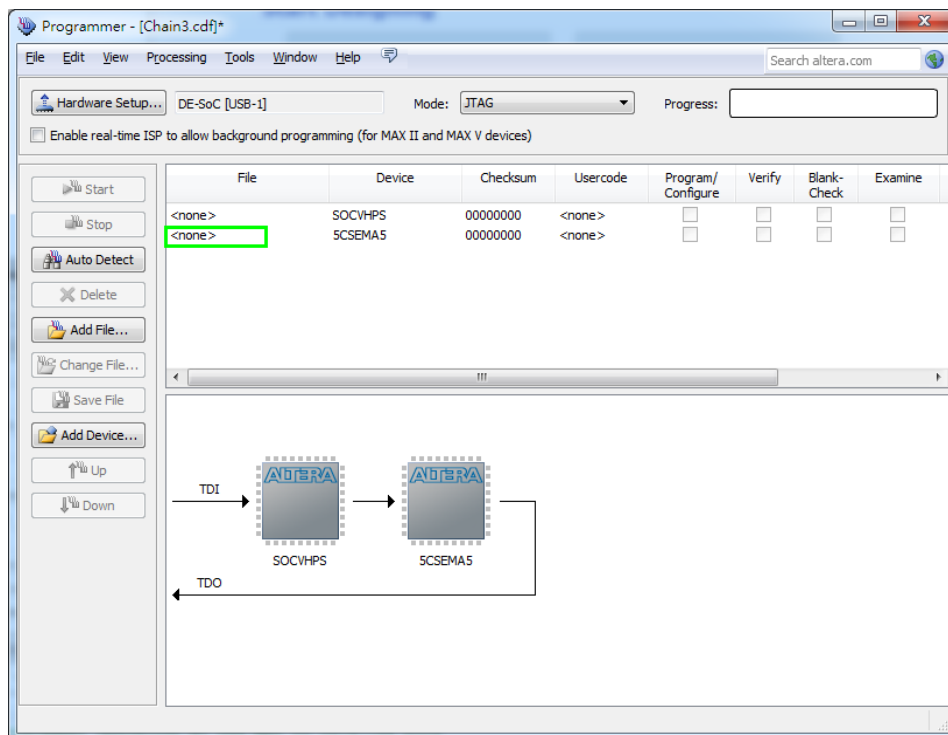


Figure 8-6 Two devices are detected in the Quartus II Programmer

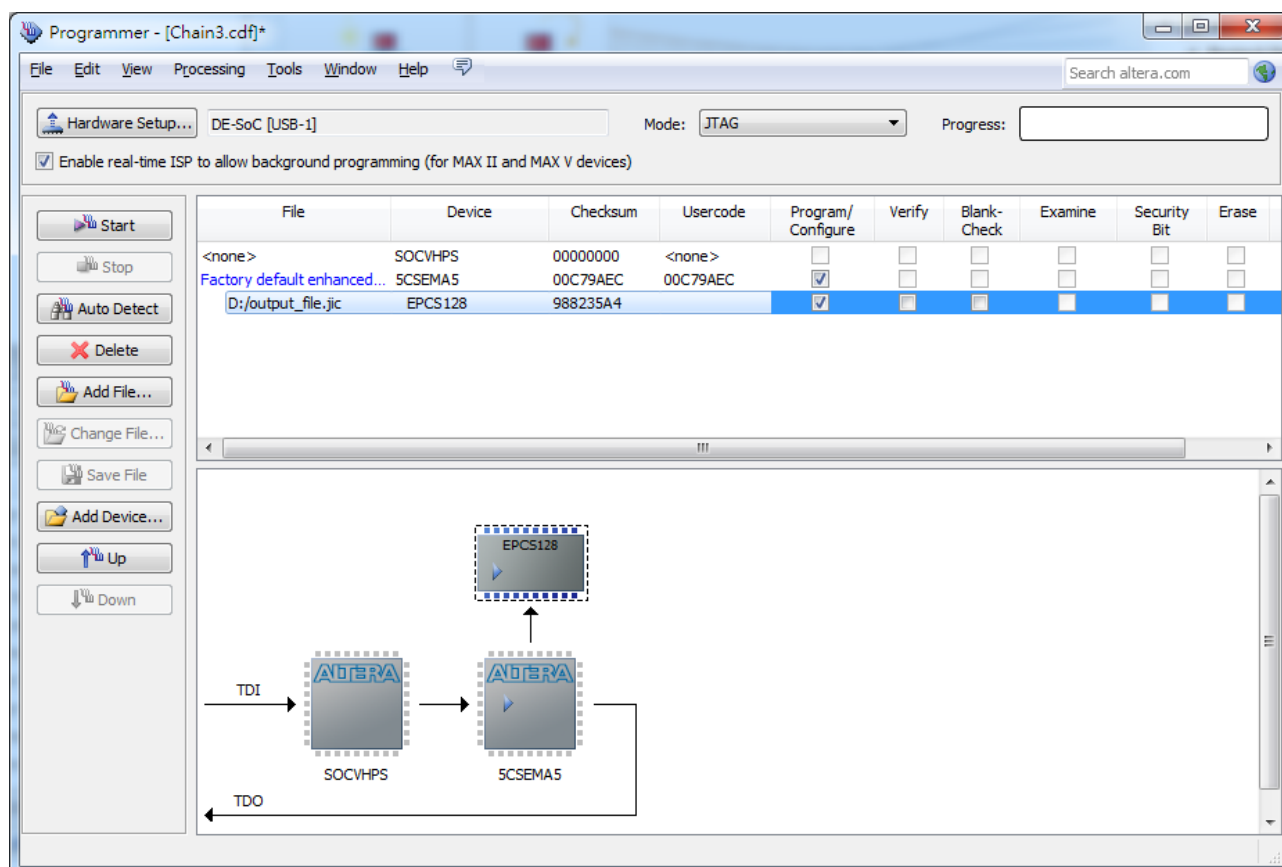


Figure 8-7 Quartus II programmer window with one .jic file

8.4 Erase the EPCS Device

The steps to erase the existing file in the EPCS device are:

1. Set MSEL[4..0] = "10010"
2. Choose **Programmer** from the **Tools** menu and the **Chain.cdf** window will appear.
3. Click **Auto Detect**, and then select correct device, both FPGA device and HPS will be detected. (See **Figure 8-6**)
4. Double click the green rectangle region shown in **Figure 8-6**, and the **Select New Programming File** page will appear. Select the correct .jic file.
5. Erase the EPCS device by clicking the corresponding **Erase** box. A factory default SFL image will be loaded, as shown in **Figure 8-8**.

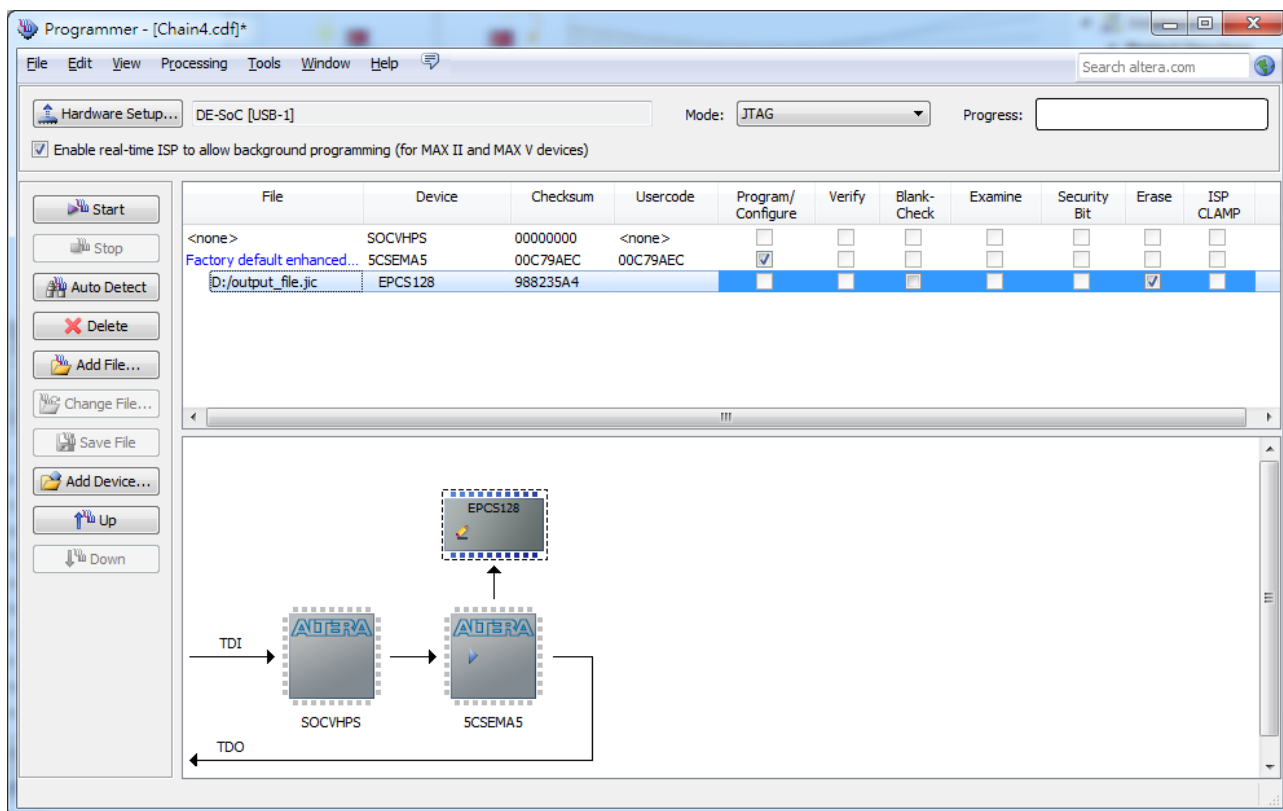


Figure 8-8 Erase the EPCS device in Quartus II Programmer

- Click **Start** to erase the EPCS device.

8.5 Nios II Boot from EPCS Device in Quartus II v16.0

There is a known problem in Quartus II software that the Quartus Programmer must be used to program the EPCS device on DE1-SoC board.

Please refer to Altera's website [here](#) with details step by step.

Chapter 9

Appendix

9.1 Revision History

<i>Version</i>	<i>Change Log</i>
V0.1	Initial Version (Preliminary)
V0.2	Add Chapter 5 and Chapter 6
V0.3	Modify Chapter 3
V0.4	Add Chapter 3 HPS
V0.5	Modify Chapter 3
V1.0	Modify Chapter 8
V1.1	Modify section 3.3
V1.2	1. Add Section 7.3 2. Modify Figure 3-2
V1.2.1	Modify Figure 3-2
V1.2.2d	Modify Figure 5-5 descriptions of remote controller
V2.0.0	Replay ADC device and modify demo description
V2.0.1	Modify EPCQ256 to EPCS128
V2.0.2	Update the remote control part and correct minor spelling
V2.0.3	Update demo for Q16.0
V2.0.4	Modify Figure 3-31

Copyright © 2016 Terasic Technologies. All rights reserved.

No matter how you found the box when you opened it, you should close it in the correct way.

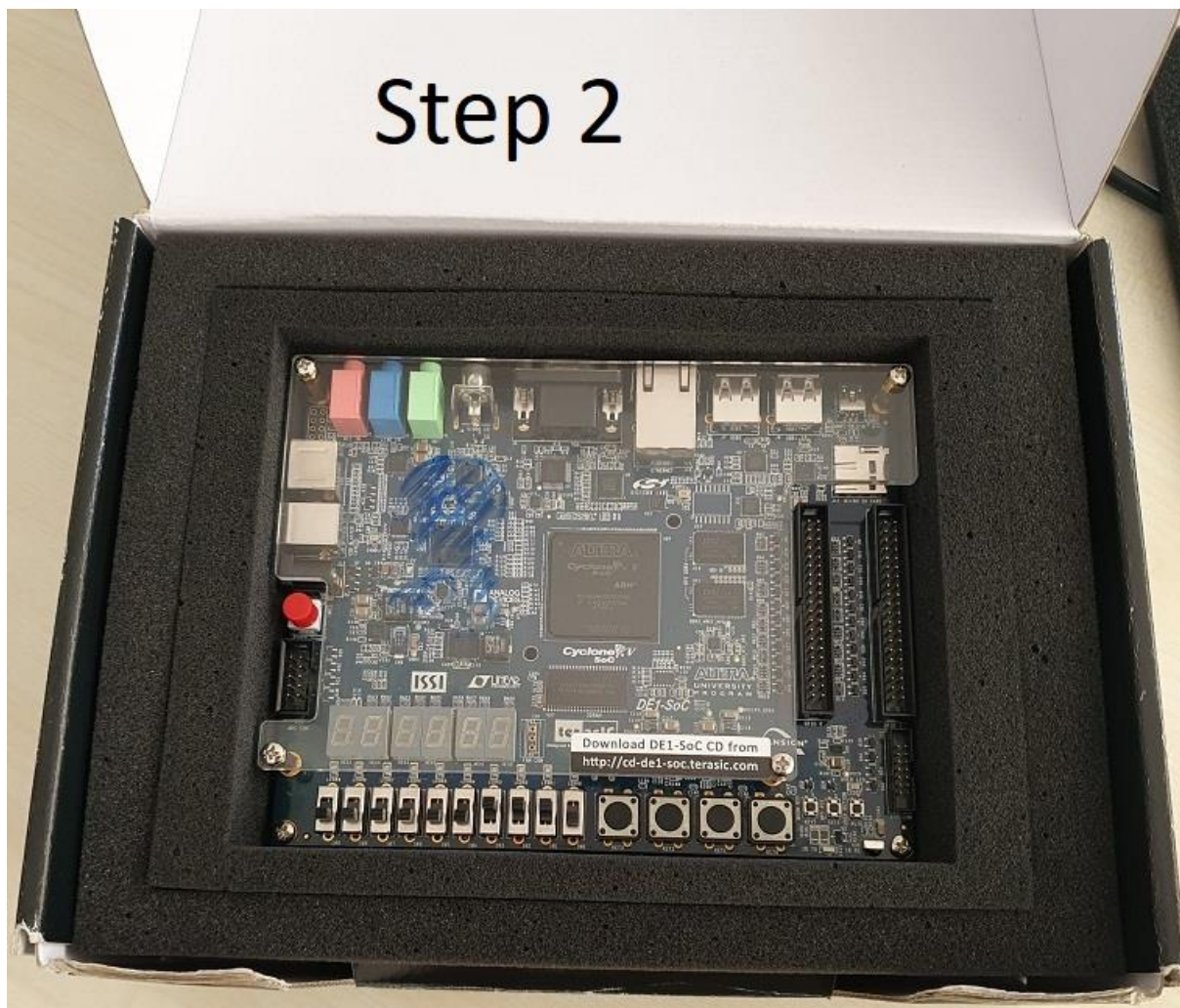
Images 1-3 show how to put FPGA back to its box correctly.

Image 4 shows how **not** to do it.

Failing to do this will reduce your performance grade



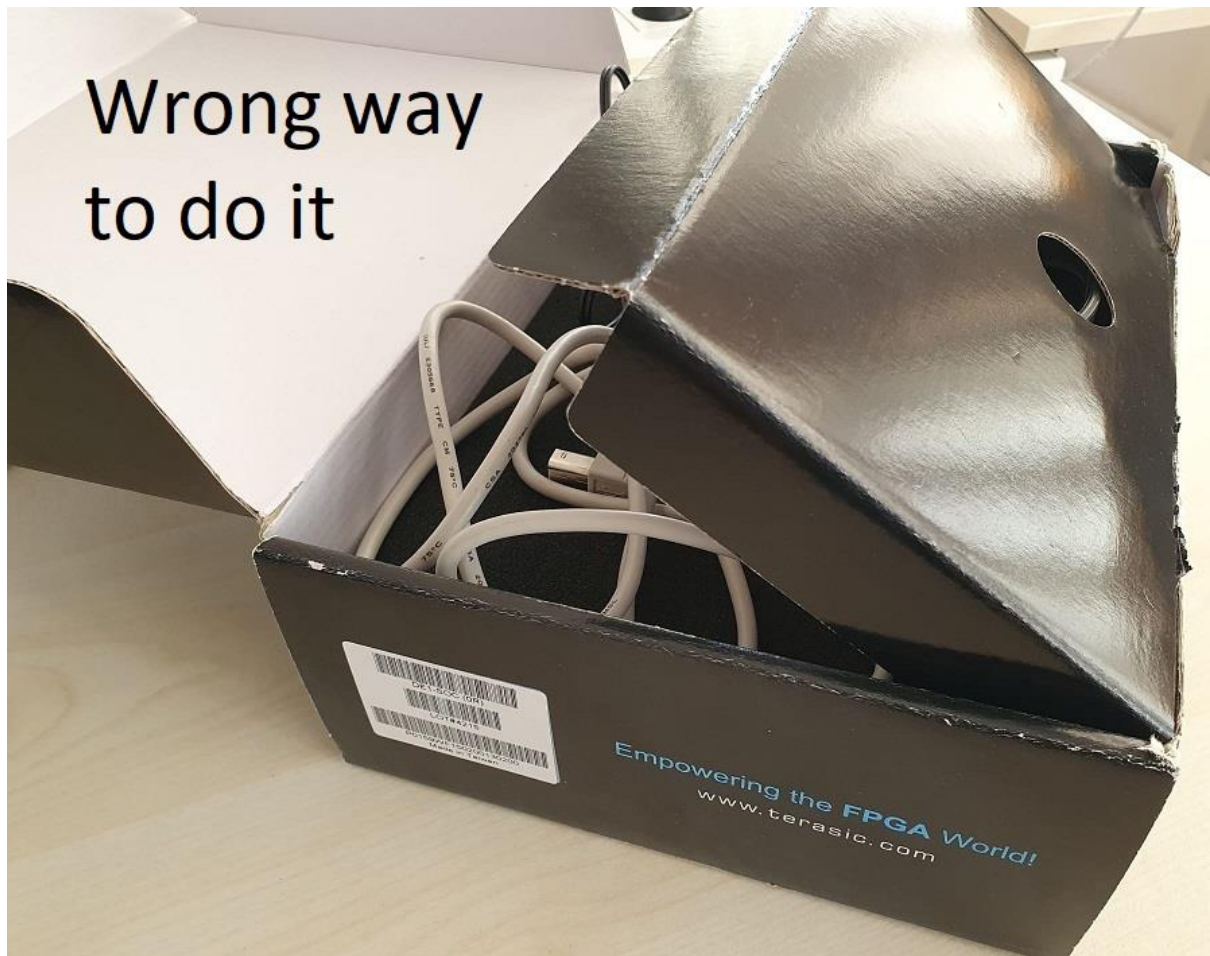
Step 2



Step 3



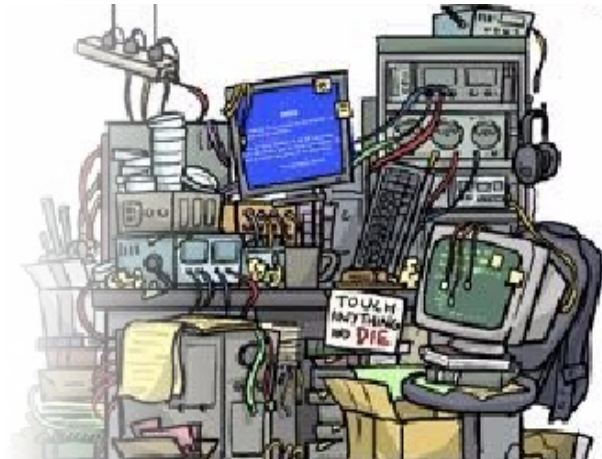
Wrong way
to do it





METU EE 446
Computer Architecture
Laboratory

Warming Up
for
Computer Design



Laboratory Work 1 - Warming Up for Computer Design

Objectives

The purpose of the first laboratory work is to construct a Verilog library composed of the fundamental modules to be used throughout the design of a computer. Moreover, simple datapath design is to be practiced through designing simple architectures from the modules in the constructed library to perform some simple tasks.

During this laboratory work, one will be familiar with designing modules with Verilog HDL. This laboratory work is a tool for getting familiar with the software Quartus and cocotb, which will be used throughout the semester. Finally, one will practice embedding the designs to a development board, DE1-SoC Board, equipped with a field programmable gate array (FPGA) and several peripheral units such as switch inputs, general purposed I/O pins, LED and 7-segment outputs, etc.

Do not worry if you can't fully complete this laboratory work. We will give you the full library for future laboratory works, and you will be required to use our codes for the supplied test benches to work.

1 Preliminary Work

To fulfill the requirements of this laboratory work, the following tasks should be performed. **Note: You should only include important parts of the code in the pdf report. The full code should be submitted separately.**

Every design here should be written in Verilog HDL and be present in your submission as a ".v" file

1.1 Reading Assignment

The laboratory manual, where the regulations and other useful information exist, is available on the ODTUClass course page. **Read that manual thoroughly.** If you feel unfamiliar with Verilog HDL programming, please refer to the corresponding lecture notes of the EE445 course, which are available on the course page. You can also find a ton of resources online for free.

1.2 Module Design with Verilog HDL (40% Credits)

For this part, you will implement fundamental modules in Verilog. These modules will then be used to construct more complex modules and computer architectures. More importantly, each module you design is to be added to your library so that you will use it in future laboratory work. Thus, consider this part of the preliminary work as building your Verilog module library.

You should submit your design codes separately from the report for each item in this part. You should also submit the corresponding cocotb test bench codes alongside the makefile for the relevant items. Additionally, remember to attach your explanation for the 2nd item of the ALU design in 1.2.4.

1.2.1 Decoder (2% Credits)

Implement a 2 to 4 and a 4 to 16 decoder.

1.2.2 Multiplexers (2% Credits)

Implement 2 to 1, 4 to 1, and 16 to 1 multiplexers, all of which have W -bit data input/outputs, where W is a module parameter.

1.2.3 Combinational Shifter (7% Credits)

Implement a W -bit combinational shifter that has three inputs: a W -bit data input, a 5-bit shift input called *shamt*, which describes the shift amount, and a 2-bit control input, where W is a module parameter specifying the data width of the input. The shifter should be able to do logical shift left, logical shift right, arithmetic shift right, and rotate right, for which the control signals are given in the Table 1

Table 1: Shifter Control Descriptions

Shifter Control [1:0]	Shifter Operation
00	LSL
01	LSR
10	ASR
11	RR

1.2.4 Arithmetic Logic Unit (ALU) (10% Credits)

1. Implement a W -bit ALU for 2's complement arithmetic, where W is a parameter specifying the data width of its operands.

The ALU has 3 data inputs, two W -bit for operands and one 1-bit for *carry*. The ALU should have 12 operations controlled by a 4-bit control input. In addition to the W -bit result output, the

ALU should have four other status output bits: Carry out (CO), overflow (OVF), negative (N), and zero (Z). Negative and zero bits are affected by all the ALU operations, whereas carry-out and overflow can only be affected by arithmetic operations. The specifications of the ALU operations and the ALU status outputs are provided in Table 2 and Table 3, respectively.

Table 2: ALU Operation Control

ALU Control [3:0]	ALU Operation	Symbol
0000	AND	$A \wedge B$
0001	EXOR	$A \oplus B$
0010	SubtractionAB	$A - B$
0011	SubtractionBA	$B - A$
0100	Addition	$A + B$
0101	Addition Carry	$A + B + carry$
0110	SubtractionAB Carry	$A - B + carry - 1$
0111	SubtractionBA Carry	$B - A + carry - 1$
1100	ORR	$A \vee B$
1101	Move	B
1110	Bit Clear	$A \wedge \neg B$
1111	Move Not	$\neg B$

Table 3: ALU Status Descriptions

Status	Description
CO	1 if there is a Carry Out from add or subtract operations; 0 for logic operations
OVF	1 if the add or subtract operation results in overflow; 0 for logic operations
Z	1 if the result is zero
N	1 if the result is negative

2. Explain your method to detect overflow.
3. Write a test bench module to test your implementation.
4. Comment the test cases in your code such that it is easily understandable which case you are testing: Addition, subtraction, AND, OR, overflow, etc.
5. Verify that your implementation is correct.
6. Provide you Makefile

1.2.5 Registers (8% Credits)

For this step, you will implement three different W -bit registers, where W is a parameter specifying the data width of the parallel input to the register and output of the register. Note that the **registers should have a clock input**, even though it is not mentioned in the following items explicitly.

1. Simple register with synchronous reset: Implement a positive edge-triggered register with parallel load and a synchronous reset. If the reset signal is 1, the content of the register is cleared at the next rising edge of the clock. If the reset signal is 0, the content of the register is loaded with the input data at the next rising edge of the clock. The specifications of the simple register with synchronous reset are provided in Table 4.

Table 4: Simple Register (A) with Reset

Reset	Operation
0	$A \leftarrow DATA$
1	$A \leftarrow 0$

2. Register with synchronous reset and write enable: Implement a positive edge triggered register with parallel load, write enable, and synchronous reset. If the reset signal is 1, the contents of the register are cleared at the next rising edge of the clock. If the reset signal is 0 and the write enable signal is 1, the contents of the register are loaded with the input data at the next rising edge of the clock. Finally, the register retains its content if the reset signal is 0 and the write enable signal is 0. The specifications of the register with synchronous reset and write enable are provided in Table 5.

Table 5: Register (A) with synchronous reset and write enable

Reset	Write Enable	Operation
0	0	Retain
0	1	$A \leftarrow DATA$
1	X	$A \leftarrow 0$

1.2.6 Memory Unit (10% Credits)

For this step, you will implement a byte-addressable memory. The module has the inputs of a clock, write enable, write data and address, and the output of read data. W is a parameter specifying the data width of write data and read data in **bytes**. The input address width is up to you. Clock and write enable is 1-bit.

Memory addressing should be combinational. Read data should change the moment the input address changes. When write-enable is given as 1, W -byte write-data input should be written to the location specified by the address input at the next positive clock edge. **You should use Little Endian convention to be consistent with ARM**

1.2.7 7-Segment Display Converter(1% Credits)

Important: Although not part of the design, you will need a 7-segment converter to show your design's operation in the lab sessions properly.

This module should take a 4-byte input and output the input as a hex number for the 7-segment display. Specifics of the conversions are up to you, but a simple case or else-if statement should be sufficient. You can see the details of the 7-segment module in Figure 1. For more info, please check the [DE1-SoC user manual](#).

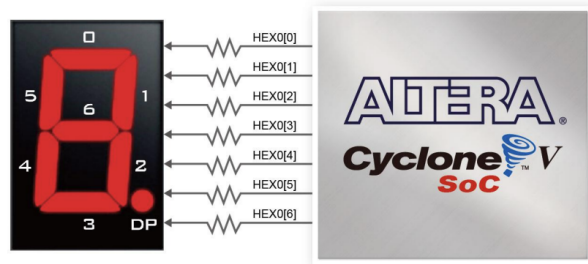


Figure 1: 7-Segment Display Signals for the DE1-SOC Board

1.3 Register File (20% Credits)

For this part, you will use your modules available from Part 1.2 to design a W -bit register file of 16 registers, where W is a parameter specifying the data width of the registers. The register file will be the central storage of the computer you will design for future laboratory work.

For the design of the register file, you will use your **decoder**, **multiplexer**, and **register** implementations. The design should be according to the desired operation of the register file.

The register file has one data input and two data outputs. The sources of the outputs and the destination of the input can be one of the 16 registers in the register file. Therefore, there should be three address inputs of width 4: one for destination select and two for source select. Finally, a control signal is required to enable write operation, and a synchronous reset signal is required to clear the contents of all registers in the register file. Note that the register file should inherently have a clock input.

The content of a register in the register file should be able to be modified without affecting the contents of the other registers. To modify a register, it should be addressed, and the write enable control of the register file module should be 1. If the write enable of the register file is 0, then the contents of the registers cannot be modified except for the reset condition. Note that the write operation is synchronous; however, the read operation should be asynchronous so that the data outputs are available as soon as their sources are addressed.

According to the aforementioned desired operation of the register file:

1. Design and sketch (on paper) a datapath for a register file design using your decoder, multiplexer and register modules. You may use additional gates wherever necessary. For your sketch, you may present your modules with boxes.
2. Implement your design in Verilog HDL.
3. Write your testbench for your implementation using cocotb. (10% Credit)
4. Provide you Makefile
5. Verify that your implementation is correct.

You should submit a pen and paper (or digital drawing) of the sketch of your design, the design code, and the corresponding test bench code with the Makefile.

1.4 Datapath Design for an Architecture (40% Credits)

In this part, you will design a datapath for an architecture so that you can perform several tasks by applying proper control signals. The architecture to be completed is provided in Figure 2. You designed the modules in subsection 1.2. There is one 8-bit register with reset and write enable, one 8-bit ALU, one 8-bit shifter, and two 8-bit multiplexers in the initial datapath. External data input is directly connected to the input of one of the MUXes.

Assuming that **the existing connections cannot be modified**, you should complete the architecture by designing a datapath so that the following tasks can be performed with the desired constraints in less than five clock cycles each:

1. 2's Complement Load: Load the register with 2's complement of the input.
2. Multiply by 10: The register will be loaded with the input times 10.
3. Duplicate the First 4-bit: Given a byte such as $x_7x_6x_5x_4x_3x_2x_1x_0$ the input make the content of the register $x_7x_6x_5x_4x_7x_6x_5x_4$

According to the aforementioned tasks with specified constraints, design a datapath with as many additional MUXes as you want.

1. Implement your design in Verilog HDL.
2. Write your testbench for your implementation using cocotb. **(20% Credit)**
3. Provide your Makefile
4. Verify that your implementations are correct.

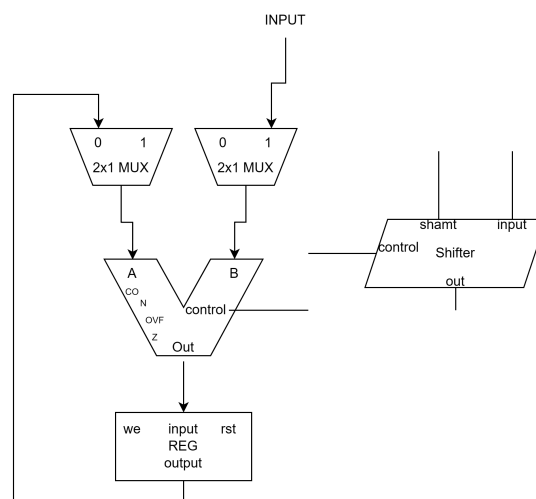


Figure 2: Architecture to which a datapath is to be designed

Considering your design, answer the following questions:

- How many control pins for the control signals does your architecture have?
- How many different control signals does your architecture use to perform the desired tasks?
- Can you reduce the number of control pins? Why not, or how?
- Write down the sequence of the control signals for all operations. How many clock cycles do these operations take?

For this part, submit pen and paper (or digital drawing) of the sketch of your datapath design and your answers to the questions.

2 Experimental Work

To upload your designs to the FPGA, you will use SystemBuilder to create a project with proper pin assignments and module initialization. DE1-SoC User manual has a short section on how the SystemBuilder works.

2.1 Register File (40% Credits)

Load the register file module designed in the Preliminary Work Part 1.3 to the DE1-SoC Board board as an 8-bit register file. There are ten switches and four push buttons on the DE1-SoC board. For the clock signal, use one of the push buttons. Debouncing exists in the push buttons of the DE1-SoC Board. Since the board does not have enough switches and buttons for other control signals, you may connect some signals to the ground or VCC. Every demonstration method is accepted as long as it is sufficient to demonstrate the full capability.

Verify the operation of the register file and demonstrate it to your lab instructor.

You must output each of the register file outputs using 7-segment display modules of the board. Use one 7-segment module for each hexadecimal digit

2.2 Datapath Design (60% Credits)

Load the custom architecture designed in the Preliminary Work Part 1.4 to the DE1-SoC Board board. Use eight hardwired connections to the ground or VCC as the data input and use the 7-Segment Display of the DE1-SoC Board board to display the content of the register. For your control signals and the clock, you may use the push buttons and switches from the DE1-SoC Board board.

Verify the operation of your design by performing the 2's Complement Load, Multiply by 10, and Duplicate the First 4-bit operations and demonstrate it to your lab instructor.

You must output the resulting signal using 7-segment display modules of the board. Use one 7-segment module for each hexadecimal digit

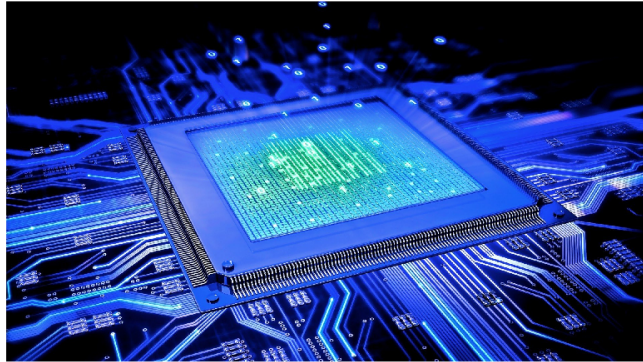
3 Parts List

DE1-SoC Board



**METU EE 446
Computer Architecture
Laboratory**

Single Cycle Processor Design



Laboratory Work 2 - Single Cycle Processor Design

Objectives

This laboratory work aims to practice the design of a 32-bit single-cycle processor. You will construct a datapath and control unit of the single-cycle processor like the one discussed in class. The designed processor will be able to execute all instructions in the given restricted instruction set.

During this laboratory work, you will further improve your hard-wired controller design skills by designing the controller unit of the single-cycle processor. Finally, you will embed your design into the FPGA of the DE1-Soc board and experiment with it.

- Instruction memory where instructions are stored
- Data memory where data is stored
- Register file
- Registers

- ALU
- Adders
- Immediate Extender
- Multiplexers
- Combinational Shifter

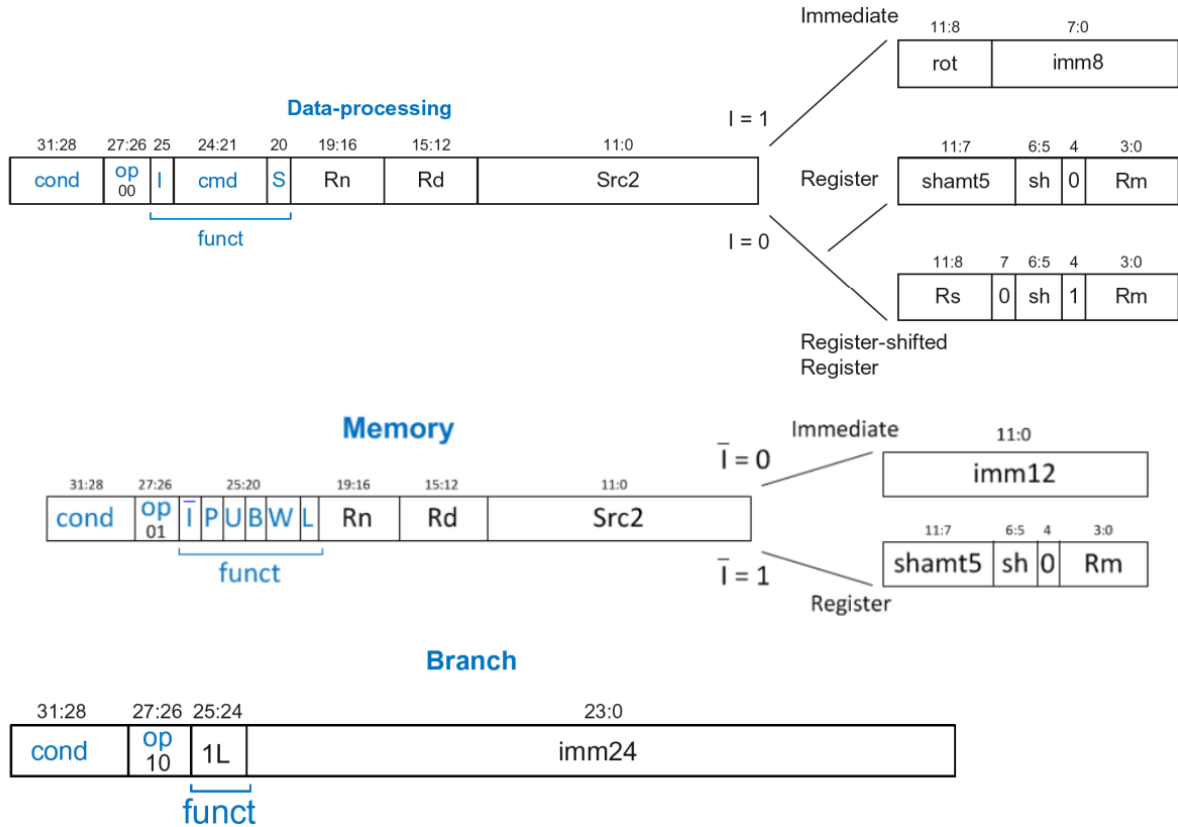


Figure 2: ARM ISA Format

Mnemonic	Name	Operation
ADD	Addition	$Rd \leftarrow Rn + (Rm \text{ sh } shamt5)$
SUB	Subtraction	$Rd \leftarrow Rn - (Rm \text{ sh } shamt5)$
AND	Bitwise And	$Rd \leftarrow Rn \& (Rm \text{ sh } shamt5)$
ORR	Bitwise Or	$Rd \leftarrow Rn (Rm \text{ sh } shamt5)$
MOV	Move to Register	$Rd \leftarrow (Rm \text{ sh } shamt5)$
MOV	Move to Register	$Rd \leftarrow (imm8 \text{ rr } rot \ll 1)$
CMP	Compare	set the flag if $(Rn - Rm = 0)$
STR	Store	$Mem[Rn + imm12] \leftarrow Rm$
LDR	Load	$Rd \leftarrow Mem[Rn + imm12]$
B	Branch	$PC \leftarrow (PC + 8) + (imm24 \ll 2)$
BL	Branch with Link	$PC \leftarrow (PC + 8) + (imm24 \ll 2), R14 \leftarrow PC + 4$
BX	Branch and Exchange	$PC \leftarrow Rm$

Table 1: ISA to be implemented

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Figure 3: ARM Condition Codes

1.2.1 Datapath Design (30% Credits)

You are given an example architecture in Figure 1. You will implement a datapath with modules in your library that have been constructed in the scope of the first laboratory work. **Use the provided files instead of the ones you wrote.** The design will extend the datapath we discussed in the lectures. A shifter needs to be added to support data processing instructions with shift. This shifter can also be used for branches, but we built-in that functionality to the extender, as in lecture notes. Some modifications in the datapath connections are also needed for BL and BX instructions. ALU has a pass-through for the second operand that you can use for MOV.

Considering the instruction set provided in Table 1, perform the following design steps:

1. (20% Credits) Using Verilog HDL, implement the Datapath using only modules and wires; no additional logic is allowed. Show the synthesized Datapaths RTL view. No I/O signal should be floating or have a constant value, as that indicates an error.
2. (10% Credits) Explain how you added the functionalities not discussed in the lecture notes. The register shifted immediate operations, both MOV operations, BL and BX.

For the operation of the computer, a certain external signal, namely **RESET**, is also necessary. Reset resets the PC register at the next positive clock edge.

1.2.2 Controller Design (40% Credits)

In this step, the controller for the single-cycle processor is to be designed. It will look like the controller in the lecture slides (Figure 4) with support for new instructions and addressing modes.

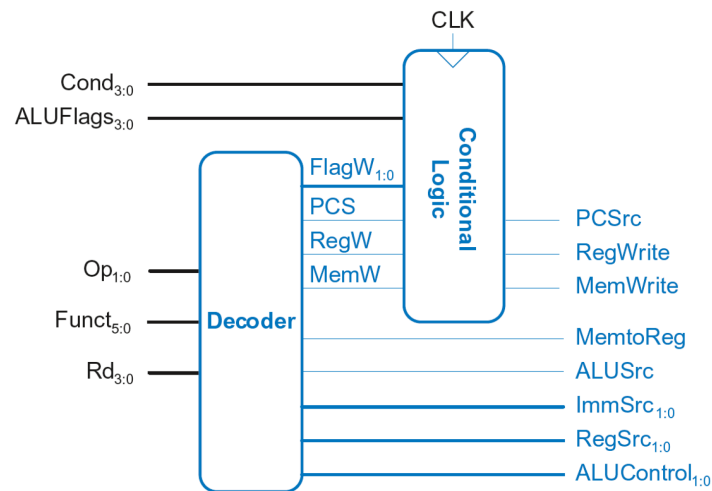


Figure 4: Controller

Perform the following steps:

1. (30% Credits) Using Verilog HDL, implement the Controller. There is no restriction, and you can write it however you like. Show the synthesized Controller RTL view. No I/O signal should be floating or have a constant value, indicating an error.
2. (10% Credits) Explain how you added the functionalities not discussed in the lecture notes. The register shifted immediate operations, both MOV operations, BL and BX.

1.2.3 Top level for Tests (10% Credits)

Use the top-level file provided in ODTUClass to assemble the controller and datapath. This will also be used to upload your design to the DE1-SoC Board. This will make:

1. Debug register select connect to the switches and debug register output connect to one of the seven segments.
2. PC register connects to one of the seven segments.

1.2.4 Testbench (20% Credits)

Now that you have completed the implementation of the single-cycle CPU, it is required to verify its operation through some light programming. You will use the supplied testbench for this. **If the computer cannot execute at least the MOV immediate instruction in the testbench, you will not be admitted to the lab.**

Don't forget to give the proper signal handles to the initialization function of the testbench class. Also, you can fill in the `log_controller` and `log_datapath` functions inside the helper library for your debugging purposes. **Do not change anything inside the TB class**

Your report must include the test bench results as a screenshot!

2 Experimental Work

To upload your designs to the FPGA, you will use SystemBuilder to create a project with proper pin assignments and module initialization. DE1-SoC User manual has a short section on how the System-

Builder works.

2.1 Single Cycle Processor (100% Credits)

Load your processor designed in the Preliminary Work Part 1.2 to the DE1-SoC board. Load the instructions to your instruction memory using \$readmemh with the provided hex file.

Your proctoring assistant will check the design and grade you depending on how many instructions the computer can successfully execute. You can get help from the proctors but any major help decreases your performance grade.

You must use the supplied top-level file that will connect all the necessary signals to the board's buttons, switches, and seven-segment displays

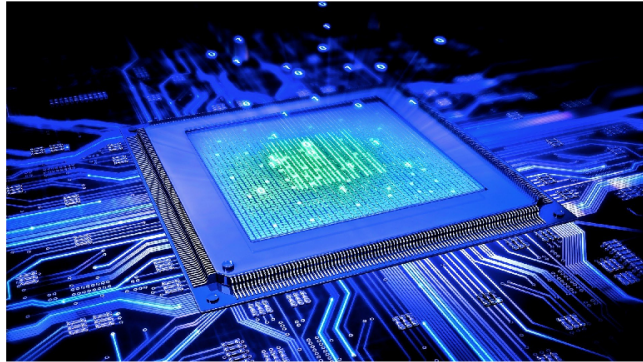
3 Parts List

DE1-SoC Board



**METU EE 446
Computer Architecture
Laboratory**

Multi Cycle Processor Design



Laboratory Work 3 - Multi Cycle Processor Design

Objectives

This laboratory work aims to practice the design of a 32-bit multi-cycle processor. You will construct a datapath and a control unit of the multi-cycle processor like the one discussed in class. The designed processor will be able to execute all instructions in the given restricted instruction set.

During this laboratory work, you will further improve your hard-wired controller design skills by designing the controller unit of the multi-cycle processor. Finally, you will embed your design into the FPGA of the DE1-SoC board and experiment with it.

1 Preliminary Work

To fulfill the requirements of this laboratory work, the following tasks should be performed.

1.1 Reading Assignment

The laboratory manual, where the regulations and other useful information exist, is available on the ODTUClass course page. Read that manual thoroughly. If you feel unfamiliar with Verilog HDL programming and multi-cycle processor design, please refer to the corresponding lecture notes of the EE445 and EE446 courses, which are available on the course page.

1.2 Multi Cycle Processor Design with Verilog HDL (100% Credits)

For this laboratory work, you will design and implement a 32-bit multi-cycle processor that executes the instruction in multiple clock cycles. First, you will design its datapath and then implement the corresponding controller.

Before starting this lab, you should be familiar with the multi-cycle implementation of the processor described in lecture slides. The multi-cycle from the lecture notes is given in Figure 2. You will implement a multi-cycle processor very similar to the one in the lecture notes with a few extra instructions you should be familiar with from the previous laboratory and some design freedom.

The processor you design will not support all ARM instructions but only a restricted set listed in Table 1. For all instructions, conditional logic of **EQ**, **NE**, and **AL** are required. Thus, you only need the "Zero" flag. Condition codes defined in ARM standards are shown in Figure 3. You will implement a shifting functionality for the second operand for data processing.

Mnemonic	Name		Operation
ADD	Addition	add Rd,Rn,Rm	$Rd \leftarrow Rn + (Rm \text{ } sh \text{ } sham5)$
SUB	Subtraction	sub Rd,Rn,Rm	$Rd \leftarrow Rn - (Rm \text{ } sh \text{ } sham5)$
AND	Bitwise And	and Rd,Rn,Rm	$Rd \leftarrow Rn \& (Rm \text{ } sh \text{ } sham5)$
ORR	Bitwise Or	orr Rd,Rn,Rm	$Rd \leftarrow Rn (Rm \text{ } sh \text{ } sham5)$
MOV	Move to Register	mov Rd,Rm	$Rd \leftarrow (Rm \text{ } sh \text{ } sham5)$
MOV	Move to Register	mov Rd,rot-imm8	$Rd \leftarrow (imm8 \text{ } rr \text{ } rot < 1)$
CMP	Compare	cmp Rd,Rn,Rm	set the flag if $(Rn - Rm = 0)$
STR	Store	str Rd,[Rn,imm12]	$Mem[Rn + imm12] \leftarrow Rd$
LDR	Load	ldr Rd,[Rn,imm12]	$Rd \leftarrow Mem[Rn + imm12]$
B	Branch	b imm24	$PC \leftarrow (PC + 8) + (imm24 < 2)$
BL	Branch with Link	bl imm24	$PC \leftarrow (PC + 8) + (imm24 < 2), R14 \leftarrow PC + 4$
BX	Branch and Exchange	bx Rm	$PC \leftarrow Rm$

Table 1: ISA to be implemented

Note: You will use the 32-bit ARM ISA format as shown in Figure 1. You can check any web resource for instructions not explained here, as we use standard ARM format.

You will need the following components to construct the datapath, all of which are given on ODTUClass
You must use the provided modules for the testbench to work

- One Instruction and Data memory (IDM)
- One Register file
- One Program Counter register
- One ALU
- One Immediate Extender
- One Combinational Shifter
- Multiplexers

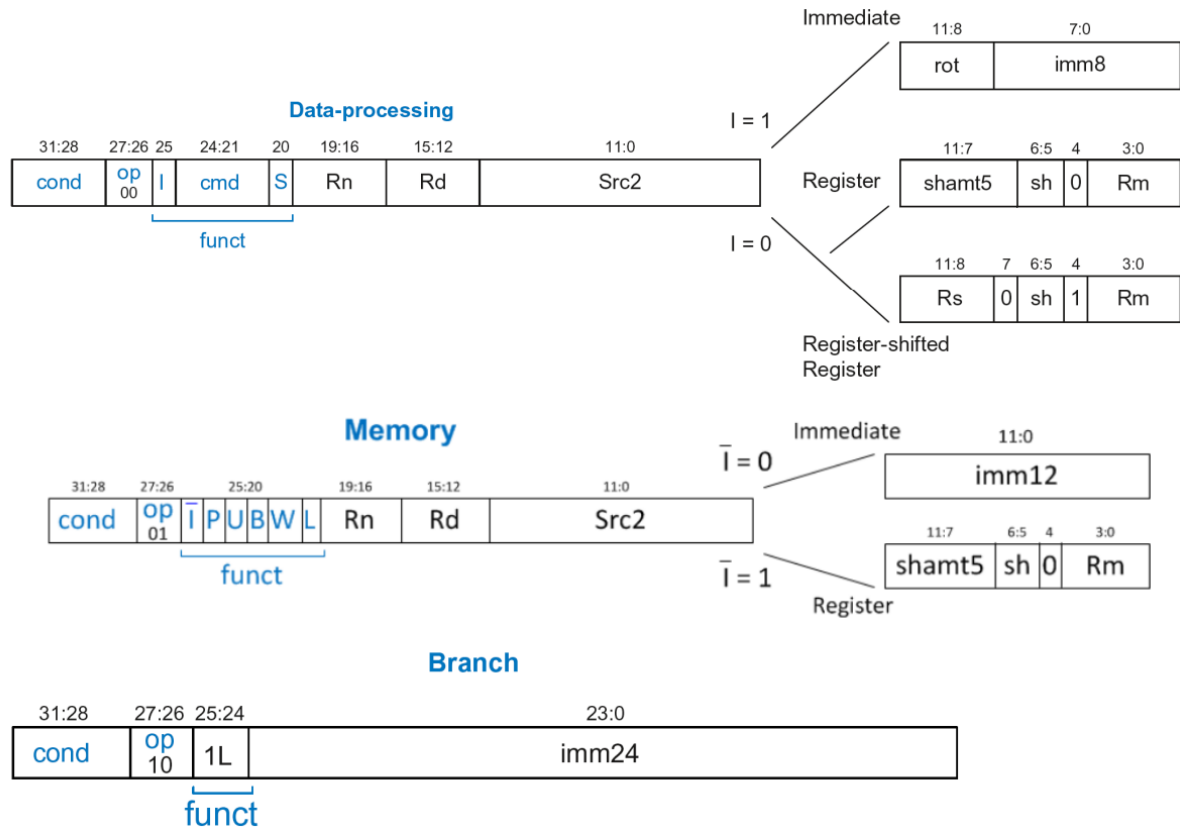


Figure 1: ARM ISA Format

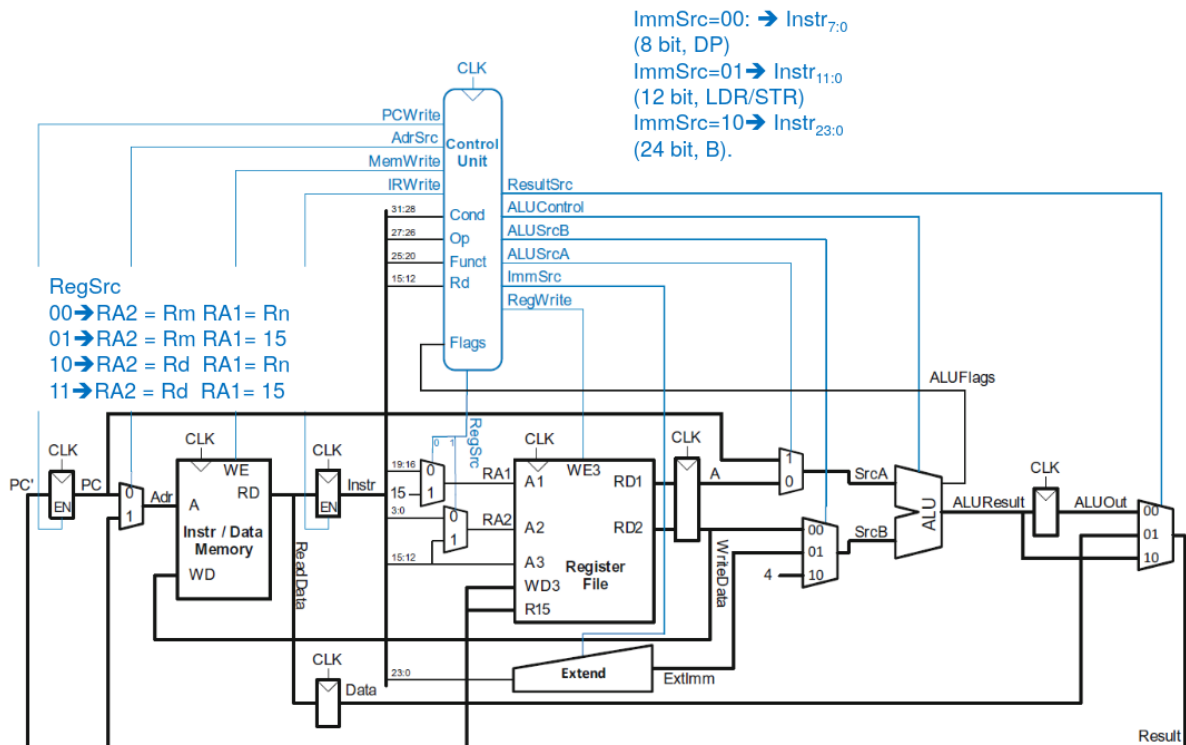


Figure 2: Multi cycle processor from the lecture notes

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Figure 3: ARM Condition Codes

1.2.1 Datapath Design (30% Credits)

In this part of the laboratory work, given your ISA, you are expected to design a full datapath to support all the instructions in the Table 1. The instructions will be stored in a unified instruction/data memory, IDM, read from and executed. You use the memory module designed in the first lab as the IDM, also given on ODTUClass.

As stated in the previous subsection, the previous ALU can be used, without modification to operations, and control signal meanings can change (which would inherently affect the control signals required for the proper operation), although there should not be any need for it. As another design limitation, **you can use only a single arithmetic logic processor, and no wired connection between the ALU and the IDM is allowed**. Besides, you may as well use other functional components and registers for temporary data storage, provided that you support your reasoning. You are not allowed to design new modules like in the previous laboratory.,

The design will extend the datapath we discussed in the lectures. A shifter needs to be added to support data processing instructions with shift. This shifter can also be used for branches, but we built in that functionality to the extender, as in lecture notes. Some modifications in the datapath connections are also needed for BL and BX instructions. ALU has a pass-through for the second operand that you can use for MOV.

Considering the instruction set provided in Table 1, perform the following design steps:

1. (20% Credits) Using Verilog HDL, implement the Datapath using only modules and wires; no additional logic is allowed. Show the synthesized Datapaths RTL view. No I/O signal should be floating or have a constant value, indicating an error.
2. (10% Credits) Explain how you added the functionalities not discussed in the lecture notes. The register shifted immediate operations, both MOV operations, BL and BX.

For the operation of the computer, a certain external signal, namely **RESET**, is also necessary. Reset resets the PC register at the next positive clock edge.

1.2.2 Controller Design (40% Credits)

The design of the control unit may be considered as designing the FSM (finite-state machine) that will generate the control sequence in the correct conditional and sequential order concerning the cycles described in the lecture notes.

- **C0: Fetch Cycle:** This is the first cycle corresponding to the operation of a single instruction. The instruction is read from the instruction/data memory to be loaded to an instruction register that holds the current one. Meanwhile, the program counter (PC) is increased to point to the next instruction.
- **C1: Decode Cycle:** Within the decode cycle, the current instruction in the instruction register is decoded to obtain the conditions and the operands.
- **C2: Execute and Branches Cycle:** In this cycle, the data is processed using the ALU or branch is taken.
- **C3: MemWrite/MemRead and ALU Writeback:** In this cycle, processed data is written back to the register file or to the memory.
- **C4: Memory Writeback:** In this cycle, data read from memory is written back to the register file.

For the supplied test bench to work, you must ensure your instruction types take the same number of cycles as in the lecture notes described in Figure 4. You are heavily encouraged to follow the lecture notes.

- Data Processing and Store instructions: 4 cycles
- Branch Instructions (BL and BX included): 3 cycles

- Load instructions: 5 cycles

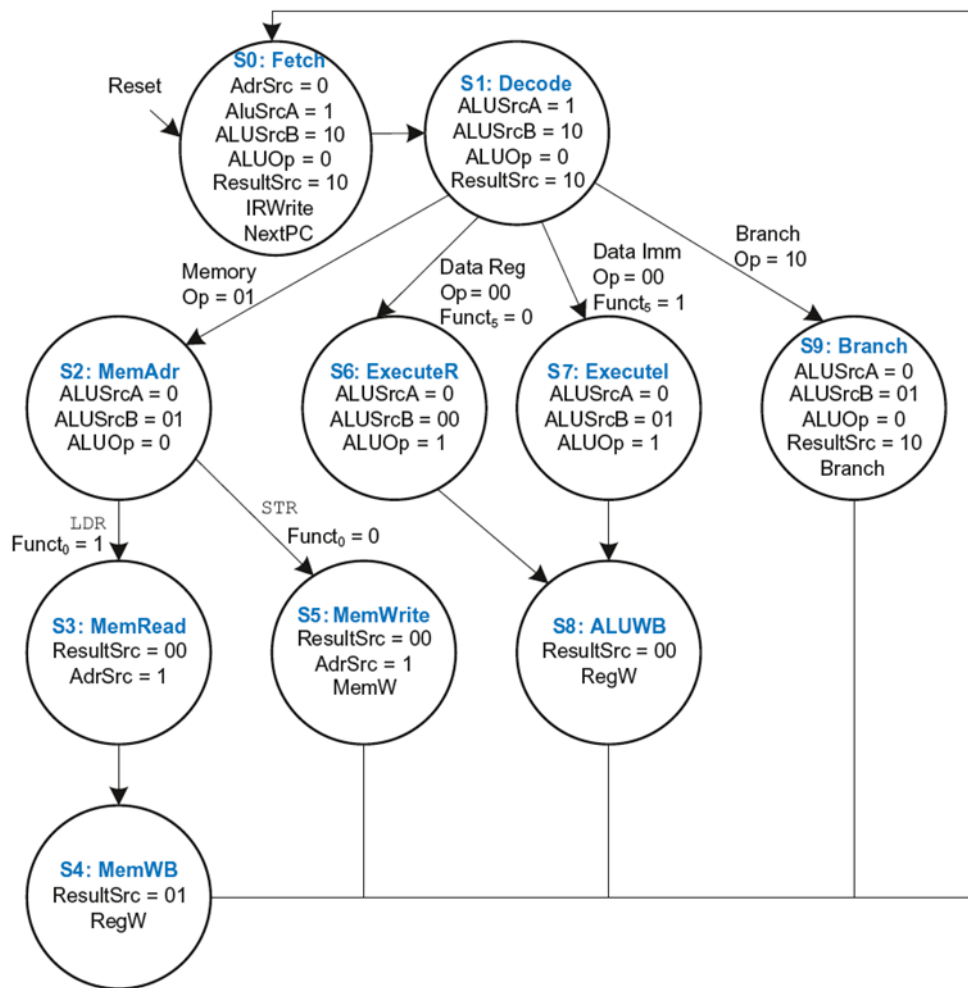
Regarding the above-given descriptions, with support for new instructions and addressing modes, you must determine which control signal in your design is to be utilized in which cycle.

For the operation of the FSM, a certain external signal, namely **RESET** is also necessary.

- **RESET(active high)**: Terminates the operation and sets the FSM to the first state at the next positive clock edge.

Perform the following steps:

1. (30% Credits) Using Verilog HDL, implement the Controller. There is no restriction, and you can write it however you like. Show the synthesized Controller's RTL view. No I/O signal should be floating or have a constant value, as that indicates an error.
2. (10% Credits) Explain how you added the functionalities not discussed in the lecture notes. The register shifted immediate operations, both MOV operations, BL and BX.



1.2.3 Top level for Tests (10% Credits)

Use the top-level file provided in ODTUClass to assemble the controller and datapath. This will also be used to upload your design to the DE1-SoC Board. This will make:

1. Debug register select connects to the switches and debug register output connects to 4 seven segments.
2. PC register connects to 2 seven segments.
3. FSM's state connects to LEDs for ease of debugging.

1.2.4 Testbench (20% Credits)

Now that you have completed the implementation of the multi-cycle CPU, it is required to verify its operation through some light programming. You will use the supplied testbench. **If the computer cannot execute at least the MOV immediate instruction in the testbench, you will not be admitted to the lab.**

Don't forget to give the proper signal handles to the initialization function of the testbench class. Also, you can fill in the log_controller and log_datapath functions inside the helper library for your debugging purposes. **Do not change anything inside the TB class**

Note that your design will fail the testbench if the number of clock cycles for each instruction does not match the specification given in subsection 1.2.2. Your report must include the test bench results as a screenshot!

2 Experimental Work

To upload your designs to the FPGA, you will use SystemBuilder to create a project with proper pin assignments and module initialization. DE1-SoC User manual has a short section on how the SystemBuilder works.

2.1 Multi Cycle Processor (100% Credits)

Load your processor designed in the Preliminary Work Part 1.2 to the DE1-SoC board. Load the instructions to your instruction memory using \$readmemh with the provided hex file.

Your proctoring assistant will check the design and grade you depending on how many instructions the computer can successfully execute. You can get help from the proctors, but any major help decreases your performance grade.

You must use the supplied top-level file that will connect all the necessary signals to the board's buttons, switches, and seven-segment displays

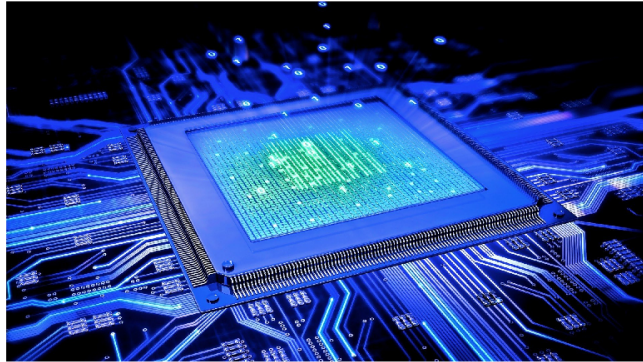
3 Parts List

DE1-SoC Board



METU EE 446
Computer Architecture
Laboratory

Pipelined Processor Design



Laboratory Work 4 - Pipelined Processor Design

Objectives

This laboratory work aims to practice the design of a 32-bit pipelined processor. You will construct a datapath and a control unit of the pipelined processor like the one discussed in class **with the hazard unit**. The designed processor will be able to execute all instructions in the instruction set.

During this laboratory work, you will improve your hard-wired controller design skills by designing the pipelined processor's controller unit, which will contain multiple stages like the datapath. Finally, you will embed your design into the FPGA of the DE1-SoC board and demonstrate your design.

1 Preliminary Work

To fulfill the requirements of this laboratory work, the following tasks should be performed.

1.1 Reading Assignment

The laboratory manual, where the regulations and other useful information exist, is available on the ODTUClass course page. Read that manual thoroughly. If you feel unfamiliar with pipelined CPU architecture, please refer to the corresponding **lecture notes of EE446** course.

1.2 Pipelined Processor Design with Verilog HDL (100% Credits)

For this laboratory work, you will design and implement a 32-bit pipelined processor that executes the instruction in multiple clock cycles but differs from a multi-cycle because it has an IPC of one. First, you will design its datapath and then implement the corresponding controller.

You will implement a pipelined processor very similar to the one in the lecture notes, with a few extra instructions you should be familiar with from the previous laboratories.

The processor you design will not support all ARM instructions but only a restricted set listed in Table 1. For all instructions, conditional logic of **EQ**, **NE**, and **AL** are required. Conditions codes defined in ARM standards are shown in Figure 3. You will implement a shifting functionality for the second operand for data processing.

Mnemonic	Name		Operation
ADD	Addition	add Rd,Rn,Rm	$Rd \leftarrow Rn + (Rm \text{ } sh \text{ } shamt5)$
SUB	Subtraction	sub Rd,Rn,Rm	$Rd \leftarrow Rn - (Rm \text{ } sh \text{ } shamt5)$
AND	Bitwise And	and Rd,Rn,Rm	$Rd \leftarrow Rn \& (Rm \text{ } sh \text{ } shamt5)$
ORR	Bitwise Or	orr Rd,Rn,Rm	$Rd \leftarrow Rn (Rm \text{ } sh \text{ } shamt5)$
MOV	Move to Register	mov Rd,Rm	$Rd \leftarrow (Rm \text{ } sh \text{ } shamt5)$
MOV	Move to Register	mov Rd,rot-imm8	$Rd \leftarrow (imm8 \text{ } rr \text{ } rot << 1)$
CMP	Compare	cmp Rd,Rn,Rm	set the flag if $(Rn - Rm = 0)$
STR	Store	str Rd,[Rn,imm12]	$Mem[Rn + imm12] \leftarrow Rd$
LDR	Load	ldr Rd,[Rn,imm12]	$Rd \leftarrow Mem[Rn + imm12]$
B	Branch	b imm24	$PC \leftarrow (PC + 8) + (imm24 << 2)$
BL	Branch with Link	bl imm24	$PC \leftarrow (PC + 8) + (imm24 << 2), R14 \leftarrow PC + 4$
BX	Branch and Exchange	bx Rm	$PC \leftarrow Rm$

Table 1: ISA to be implemented

Note: For this lab, you will use the 32-bit ARM ISA format as shown in Figure 1. You can check any resource from the web for any instructions not explained here, as we use standard ARM format.

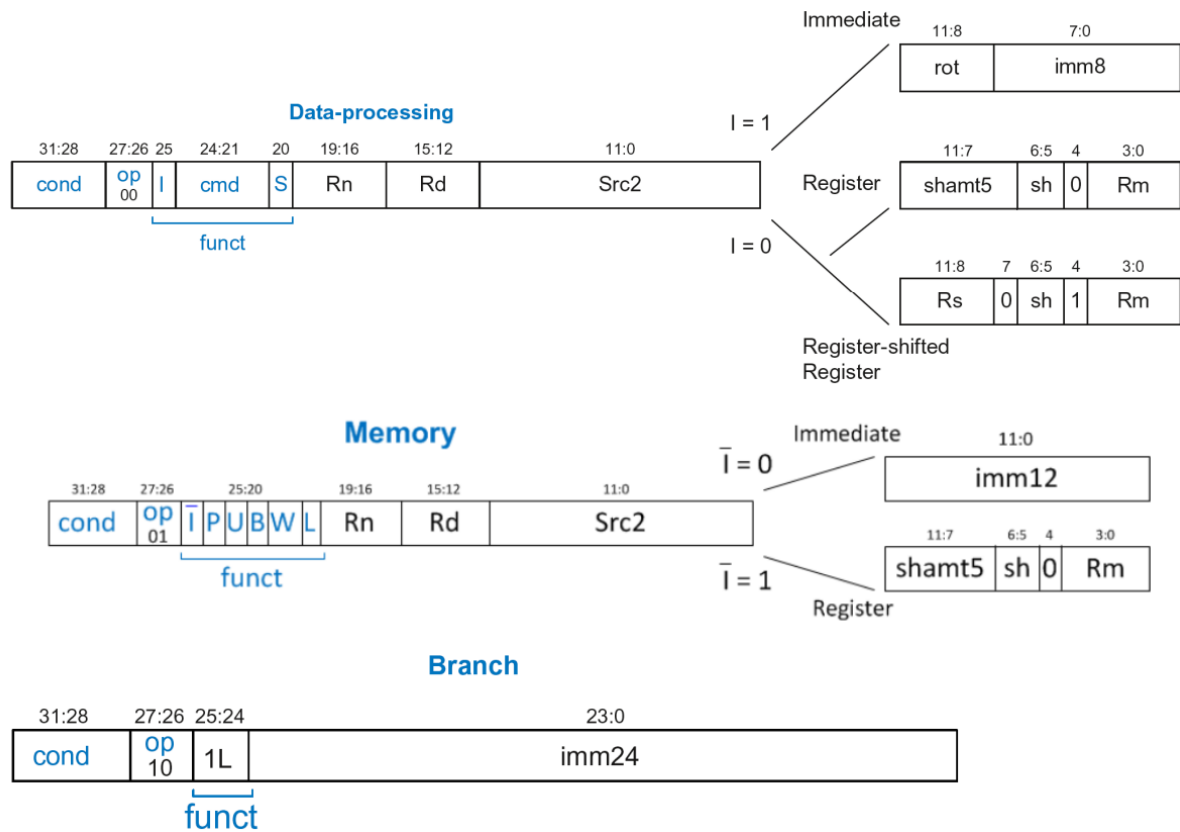


Figure 1: ARM ISA Format

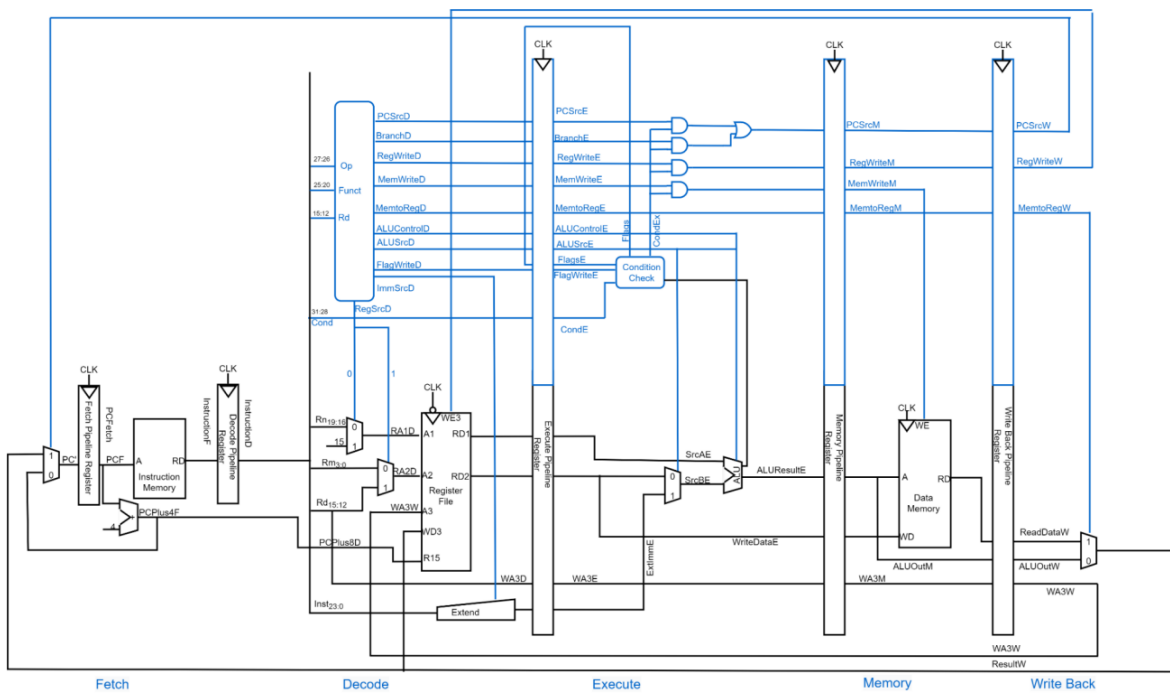


Figure 2: Pipelined processor from the lecture notes

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Figure 3: ARM Condition Codes

1.2.1 Datapath Design (20% Credits)

In this part of the laboratory work, given your ISA, you are expected to design a full datapath that would support all the instructions included in Figure 1. Using pipelined processor implementation, you will use five stages for your datapath as in lecture notes: Fetch, Decode, Execute, Memory, and Writeback. 5 Stages are important because the test bench won't work with different stages.

You will need the following components to construct the datapath, all of which are given on ODTUClass
You must use the provided modules for the testbench to work

- Instruction Memory
- Data Memory
- Register file
- Program Counter register
- One ALU
- One Immediate Extender
- Multiplexers
- One Combinational Shifter
- Interim registers for pipelined operation
- Adders

The design will extend the datapath we discussed in the lectures. A shifter needs to be added to support data processing instructions with shift. This shifter can also be used for branches, but we built that functionality into the extender, as in lecture notes. Some modifications in the datapath connections are also needed for BL and BX instructions. ALU has a pass-through for the second operand that you can use for MOV.

Give your reasoning in the report for the changes you made to the datapath in the lecture notes (including how you used shifter and implemented the BL/BX instruction). You can change the datapath as much as you want if you give proper reasoning. As a rule of thumb, try not to needlessly forward data between stages and use the inter-stage registers as much as possible to ensure the critical path is small.

Considering the instruction set provided in Table 1, perform the following design steps:

1. (15% Credits) Using Verilog HDL, implement the Datapath using only modules and wires; no additional logic is allowed. Show the synthesized Datapaths RTL view. No I/O signal should be floating or have a constant value, as that indicates an error.
2. (5% Credits) Explain how you added the functionalities not discussed in the lecture notes. The register shifted immediate operations, both MOV operations, BL and BX.

1.2.2 Controller Design (30% Credits)

You are going to design a controller for the datapath you have designed. The pipelined computer controller will be very similar to the single-cycle controller but will forward the controller signals through the pipeline stages.

The design will extend the controller we discussed in the lectures. The shifter controller will have additional signals, and BL instruction will require you to design a new set of control signals for it. Make sure everything is consistent with your datapath.

Give your reasoning in the report for the changes you made to the controller in the lecture notes (including how you used shifter and implemented the BL instruction). As with the datapath, you can change the controller as much as you want if you give proper reasoning.

For the correct operation of the computer, you will need a **RESET** signal that terminates the operation and sets the PC to the very first slot in the instruction/data memory (active high) at the next positive clock edge. **Don't forget to reset interim registers.**

Perform the following steps:

1. (20% Credits) Using Verilog HDL, implement the Controller. There is no restriction, and you can write it however you like. Show the synthesized Controller's RTL view. No I/O signal should be floating or have a constant value, as that indicates an error.
2. (10% Credits) Explain how you added the functionalities not discussed in the lecture notes. The register shifted immediate operations, both MOV operations, BL and BX.

1.2.3 Hazard Unit (25%)

The hazard unit implemented for this project will handle two hazard types. Most of the design will be consistent with the implementation you have studied in the lectures. As a reminder, the list of the terms for hazard handling is given.

- Flush: Clearing a stage register so that the result of that stage is discarded
 - Stall: Holding the value of a stage register so that a bubble can be introduced
 - Forward: Sending the calculated value to a previous stage
1. (20% Credits) Using Verilog HDL, implement the Hazard Unit. There is no restriction, and you can write it however you like. Show the synthesized Hazard Unit's RTL view. No I/O signal should be floating or have a constant value, indicating an error.
 2. (10% Credits) Explain how you handled the functionalities not discussed in the lecture notes. The register shifted immediate operations, both MOV operations, BL and BX.

Data Hazard Handling (15%) Data hazards happen when an instruction tries to read a register that a previous instruction has not yet written back. There can be multiple methods to handle this hazard type, even as simple as constant stalling. However, you can see that this implementation method will decrease the efficiency of your design. Hence you are required to implement your hazard unit such that:

- Hazards caused by data operations must be handled by forwarding such that no cycle is wasted.
- Hazards caused by memory operations can use a minimal amount of stalling, which is one cycle.

Control Hazard Handling (10%) Control hazards happen when the decision of what instruction to fetch next has not been made by the time the fetch takes place.

Branch operations and other operations that write to the PC (MOV R15, BX, B, BL, etc. with their conditional variants) will forward the new PC value to the fetch cycle and flush the wrong stages **when the branch is taken**. This should be implemented with a minimal amount of flushing while considering the critical path. See the lecture notes for more detailed explanations.

1.2.4 Top level for Tests (5% Credits)

Use the top-level file provided in ODTUClass to assemble the controller and datapath. This will also be used to upload your design to the DE1-SoC Board. This will make:

1. Debug register select connect to the switches and debug register output connect to one of the seven segments.
2. PC register connects to one of the seven segments.
3. You can use LEDs to connect to various hazard signals.

1.2.5 Testbench (20% Credits)

Now that you have completed the implementation of the pipelined CPU, it is required to verify its operation through some light programming. You will use the supplied testbench for this.

Don't forget to give the proper signal handles to the initialization function of the testbench class. Also, you can fill in the log_controller and log_datapath functions inside the helper library for your debugging purposes. **Do not change anything inside the TB class**

Your report must include the test bench results as a screenshot! You should submit the testbench files as well

2 Experimental Work

To upload your designs to the FPGA, you will use SystemBuilder to create a project with proper pin assignments and module initialization. DE1-SoC User manual has a short section on how the SystemBuilder works.

2.1 Pipelined Processor (100% Credits)

Load your processor designed in the Preliminary Work to the DE1-SoC board. Load the instructions to your instruction memory using \$readmemh with the provided hex file.

Your proctoring assistant will check the design and grade you depending on how many instructions the computer can successfully execute. You can get help from the proctors, but any significant help decreases your performance grade.

You must use the supplied top-level file that will connect all the necessary signals to the board's buttons, switches, and seven-segment displays

3 Parts List

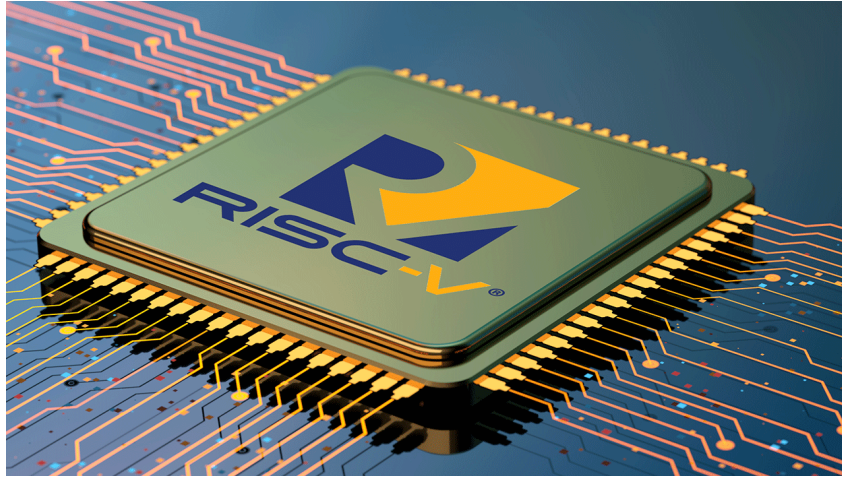
DE1-SoC Board



METU EE446

Computer Architecture

Laboratory



Laboratory Project - Single Cycle RISC-V Processor

Objectives

This project aims to explore a novel, license-free, open-source instruction set architecture that is becoming increasingly popular in the industry. You will construct the datapath and the control unit of a single-cycle 32-bit RISC-V processor. For this project, the instruction set has been extended by one more instruction. The designed processor will be able to execute all instructions in the **extended** instruction set. Finally, you will embed your design into the FPGA of the DE1-SoC board and demonstrate your design.

This project will be done in groups of 2 students unless you choose to do the project by yourself. You can choose your partner. Each partner is expected to contribute in equal amounts. If the work is divided too unevenly, the student who did more will be generously graded, while the student who did less will be penalized. The most uneven work division acceptable is 60-40. If you wish to do the project in a group but cannot find a partner, we will match you with another student (if possible).

The project needs to be done by groups individually. **In other words, inter-group cooperation will be considered as cheating and further action will be taken as explained in the EE446 Laboratory Manual.**

Contents

1	Introduction	3
1.1	Reading Assignment	3
1.2	Comparison with ARM	3
2	Project Preliminary Work (20%)	4
2.1	ISA to be implemented	4
2.1.1	Extra instruction	4
2.2	Datapath (25%)	4
2.2.1	Datapath Design	4
2.2.2	Datapath Implementation	4
2.3	Controller (25%)	4
2.3.1	Controller Design	4
2.3.2	Controller Implementation	5
2.4	Top Level (5%)	5
2.5	Testbench (45%)	5
2.6	Important Considerations	5
3	Project Demonstration (80%)	6
A	Useful Materials	7

1 Introduction

RISC-V is an innovative instruction-set architecture (ISA) that was initially developed to support research and education in computer architecture at UC Berkeley. Its design aspirations have since expanded, aiming to become a standard, free, and open architecture for industry implementations. Characterized by its flexibility and generality, RISC-V is not tailored to specific microarchitectural styles or technologies, making it suitable for a wide range of hardware implementations, from custom chips to multicore processors. The architecture includes a modular structure with a base integer ISA and optional extensions supporting 32-bit and 64-bit address spaces. RISC-V is designed to facilitate efficient implementations and is fully virtualizable, simplifying hypervisor development. This open and versatile ISA holds significant potential to influence academic research and industrial applications broadly.

1.1 Reading Assignment

In this project, a part of RISC-V ISA will be implemented. To get familiar with it, read the RISC-V specification given in the link below. Chapters 2, 24 and 25 are of most interest. Note that this project manual doesn't contain the low-level details needed to implement the processor fully. Therefore, you'll need to seek additional information from relevant parts of the RISC-V specification.

[*RISC-V Specification 20191213.pdf*](#)

Additionally, Chapter 7.3 of Harris & Harris Risc-v book contains an implementation of part of the ISA, which you can consider:

Sarah Harris, David Harris - Digital Design and Computer Architecture RISC-V Edition (2021)

Since RISC-V is an open standard, many materials are freely available online.

As usual, taking any code without citing its origin or taking large sections of code from any source is Plagiarism and will result in a 0x0 (zero) grade, whereas disciplinary action may be taken.

1.2 Comparison with ARM

ARM 32-bit had 16 architectural registers, with PC=R15. On the other hand, RISC-V has 32 registers (x0 to x31), and PC is not one of these registers. Also, register0 is hardwired to zero value.

ARM had conditional instructions, but RISC-V only has conditional branches. Therefore, ALU flags don't need to be saved to a register. RISC-V does not have shifted operands, but it has shift instructions.

RISC-V has six types of instructions as seen in Figure 1. Instructions can be read from the rs1/rs2 registers and written to the rd register. Instruction encodings may look complicated, especially the immediate encodings. However, these encodings reduce the number of multiplexers and are efficient to implement in hardware.

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7				rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]						rs1		funct3		rd		opcode		I-type
imm[11:5]				rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12 10:5]				rs2		rs1		funct3		imm[4:1 11]		opcode		B-type
imm[31:12]										rd		opcode		U-type
imm[20 10:1 11 19:12]										rd		opcode		J-type

Figure 1: RISC-V instruction types

2 Project Preliminary Work (20%)

2.1 ISA to be implemented

RISC-V consists of base ISA and extensions. In this project, the Unprivileged RV32I Base Integer Instruction Set will be implemented. Additionally, one new instruction will be added as an extension.

FENCE, ECALL, EBREAK, and HINT instructions will not be implemented, as they are irrelevant.

The list of instructions to be implemented is given in Table 1. For more information about the instructions, refer to the RISC-V specification.

Arithmetic instructions:	ADD[I], SUB
Logic instructions:	AND[I], OR[I], XOR[I]
Shift instructions:	SLL[I], SRL[I], SRA[I]
Set if less than:	SLT[I][U]
Conditional branch:	BEQ, BNE, BLT[U], BGE[U]
Unconditional jump:	JAL, JALR (Return-address stack push/pop functionality will not be implemented)
Load:	LW, LH[U], LB[U]
Store:	SW, SH, SB
Others:	LUI, AUIPC
Extra instruction:	XORID

Table 1: List of Instructions

2.1.1 Extra instruction

The extra instruction is XORID. It will take the xor of rs1 with an embedded constant and write the result to rd. It has the following format:

$$\text{XORID } rd, rs1 \quad rd \leftarrow rs1 \oplus (studentId1 \oplus studentId2)$$

Instruction encoding details are given below:

- opcode[6:0]=0001011
- I type instruction, but immediate value will not be used
- funct3[2:0]=100

2.2 Datapath (25%)

2.2.1 Datapath Design

Design a datapath that will support all of the instructions listed above. In your report, explain your design with appropriate visuals. It can be based on the datapath shown in Figure 2.

2.2.2 Datapath Implementation

Implement your design in Verilog HDL. Datapath must consist of submodules but should not contain "always blocks" or direct logic. You can use the modules from laboratories or create your own.

Datapath needs to have a synchronous reset.

Show the synthesized RTL view of your datapath in your report. Explain how each instruction type is executed in your datapath.

2.3 Controller (25%)

2.3.1 Controller Design

Design a controller that matches your datapath. In your report, explain your controller with appropriate visuals. Explain any submodules the controller has.

2.3.2 Controller Implementation

Implement your design in Verilog HDL. The controller can be written in a single module, or it can have submodules.

Show the synthesized RTL view of your controller in your report. Explain how each instruction type is executed in your controller.

2.4 Top Level (5%)

In this part, the controller and datapath are assembled in a top module. The top module needs to have a synchronous reset.

For debugging purposes, connect five switches to the register files debug port select. Connect debug register output and PC register output to 7-segment displays as in previous labs.

2.5 Testbench (45%)

It is required to verify your Computer's operation by writing a testbench.

- The testbench should read the instructions from a hex file like the HDL computer.
- It should execute all the instructions by itself and then compare its register file and PC values to the HDL design.
- The testbench should be able to execute arbitrary RISC-V code consisting of the given RV32I instructions.
- A proper testbench is automated, so it should indicate when something fails in the design without needing any manual work.

Your testbench should be very similar in form to the supplied testbenches on ODTUClass. You can use the single-cycle ARM computer testbench that was provided and modify it appropriately. For debugging purposes, you can use prints/logs in your testbench.

Explain how you implemented your testbench and its details in your report. You'll need to write RISC-V program(s) that cover all instructions with their special cases. Explain your programs and how they cover all instructions and special cases.

2.6 Important Considerations

Your codes should be clean, easily readable, and understandable. Check your indentations so that they are obvious where a block ends. Put comments when and where needed.

Since you have almost graduated, we expect a more professional report for the project compared to some of your laboratory reports. Your report should be proofread and structured well, and the texts in it should not be pictures of handwriting.

If you used some online/offline tools to compile or assemble RISC-V codes, specify them in your report.

You can lose grades if your code/report is unprofessional, hard to read, etc.

Deliverables:

- PDF Report
- Python testbench codes
- HDL codes

3 Project Demonstration (80%)

In the demonstration, you will be given a code segment for your computer; you will first use the code in your testbench and demonstrate it to your teachers. Then, load your processor designed in Part 2 to the DE1-SoC board with the given instructions.

Additionally, the teachers will ask questions about the parts each student has contributed. You should be very familiar with your work to be able to answer any questions. If a student cannot answer the questions to the satisfaction of the teachers, you will lose grades.

You should also bring your own test codes incase the computer cannot execute the given instructions

A Useful Materials

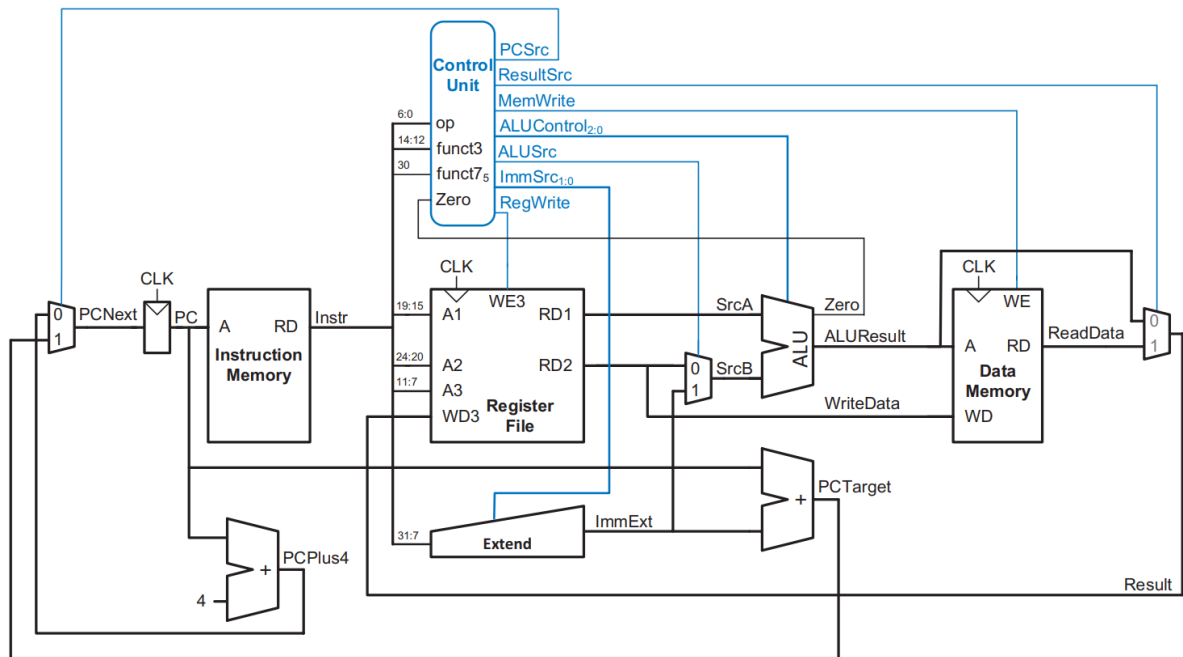


Figure 2: RISC-V Single Cycle Computer as Described in Harris&Harris



MIDDLE EAST TECHNICAL UNIVERSITY

**ELECTRICAL-ELECTRONICS ENGINEERING
DEPARTMENT**

COCOTB for EE445-446 Verilog Simulations

Contents

1	Introduction	1
2	What is cocotb	1
3	Advantages of Cocotb	1
4	Installation of Cocotb and Icarus Verilog	1
5	Introduction to Cocotb	2
5.1	How Does Cocotb Work?	2
5.2	How to Write Test-benches	3
5.3	Running The Test-bench Examples	3
5.4	Where to Learn Further About Cocotb?	4

1 Introduction

In the scope of the E446 laboratory, you will develop Verilog codes and embed them in the DE0 Nano board. An important part of developing your codes is the design of test benches to verify your designs. Up until now ModelSim simulator in tandem with Verilog was used for the test benches but this semester you can also use cocotb.

2 What is cocotb

Cocotb is a COroutine based COsimulation TestBench environment for verifying VHDL and SystemVerilog RTL using Python. Cocotb is completely free and hosted on github. Cocotb works with a plethora of HDL simulators on Linux, Windows, and macOS.

3 Advantages of Cocotb

With cocotb, Verilog will only be used for your hardware design. All the verification will be done using python which has several advantages over Verilog such as:

- Writing in Python is **fast** and **easy**.
- Interfacing with any other language or program with Python is **simple**.
- Python has **a lot of libraries** for you to use.
- Python is a much more **flexible** language compared to Verilog

4 Installation of Cocotb and Icarus Verilog

Cocotb is regularly updated, meaning the installation steps given in this part might get outdated. The best way of installing cocotb is following the installation section of the [cocotb documents](#).

Installation process for **Windows** as of writing this manual is as follows:

1. Install the latest miniconda version from [conda.io documents](#), you do not need to add miniconda to PATH but select to register it as default. This will give you a new package management system and enviroment with Anaconda Prompt terminal. (**You can also install the full version of Anaconda instead of miniconda**)
2. Open the newly installed Anaconda Prompt and use the following line to install a compiler (GCC or Clang) and GNU Make:

```
conda install -c msys2 m2-base m2-make
```

3. From the Anaconda Prompt install cocotb with the following line (You may need to install Visual Studio C++ 2014 redistributable if you don't have it):

```
pip install cocotb
```

4. Now you need a verilog simulator for cocotb to use. For this course we will use iverilog (Icarus Verilog), you can download iverilog from [Icarus Verilog for Windows](#) site. Install iverilog with add to PATH option selected. If you forgot to select the "add iverilog to PATH" option you can manually add the bin folder of iverilog to PATH.
5. (Optional) Using Anaconda Prompt install pytest with the following line:

```
pip install pytest
```

This will make the errors shown much more detailed, you can check exactly what it does at [pytest site](#).

Note: For pytest to actually do its job, your test-bench files must have the name format of test_*.py or *_test.py

Execute the **cocotb-config** command in the Anaconda prompt to check if cocotb installed correctly.

You can try running the examples provided by the cocotb or for the EE446 course to confirm the cocotb and Icarus Verilog is fully functional.

5 Introduction to Cocotb

Because the cocotb gets frequently updated. Its module, methods and variables can change over time. Thus, you should read through the [cocotb quickstart guide](#) and "How-to Guides" section of the [cocotb documents](#), as that will give you the most up-to-date knowledge on how to write your test benches.

5.1 How Does Cocotb Work?

In the most basic sense, cocotb is an interface between Python and an HDL simulator. Cocotb transfers every signal in your Verilog code to the python script as the variables of an object representing your Verilog design. It executes your python code until no thread is left anything to do after which it switches to the HDL simulator, changes the signal values you changed in the python and advances the simulation time. A fundamental example is driving the clock. You can check Figure 1 for a simple flowchart of cocotb's operation.

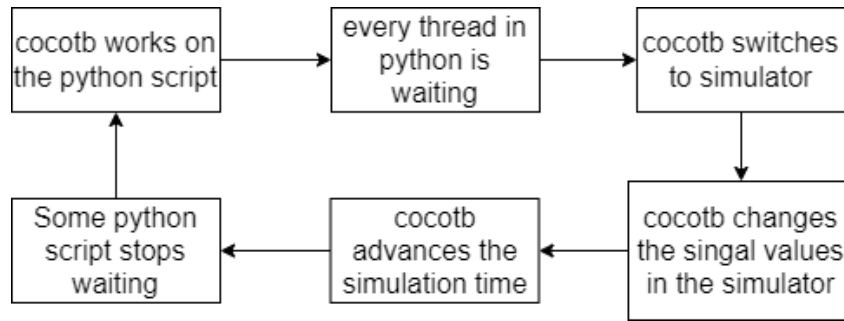


Figure 1: Simplified Workflow of cocotb

5.2 How to Write Test-benches

You should first read [cocotb quickstart guide](#) if you did not read it until now. All of the code will be written in Python. Cocotb modules will be imported just like any other library modules and be called in the code. Cocotb will call the functions marked as tests and give your design as an argument to the function. Thus, all your signals from the Verilog design can be accessed just like variables of an object.

For synchronous design, the first thing you need to do is create a clock, which will be done with the Clock module of cocotb. After this point, you can either verify your design by manually changing the signals and checking the outputs (just as you would do in ModelSim) or you can fully automate the process. Anything doable with Python is possible so there is no form your test-bench must take.

One thing to note is the usage of 'makefile'. As explained in [cocotb quickstart guide](#) your makefile is simple text that will contain the name of the Verilog source codes you want to include, the top-level module for your Verilog design, the Simulator that will be used, and the Python test file that contains the test-benches.

For reference please check the supplied test benches on Odtuclass. You can also use them as templates for your test benches.

5.3 Running The Test-bench Examples

Test-benches discussed in this section will be provided to you through Odtuclass. The Verilog module that will be used for the test benches is the signed magnitude adder/subtractor you learned in the EE445 lectures. Please be aware that you need to read at least the [cocotb quickstart guide](#) to understand the code.

In the AU_cocotb_test.py file provided to you, there are 3 test-benches, one to show failure, another for a basic test bench, and the last one for an advanced test bench.

To run the tests go to the directory where the test file (AU_cocotb_test.py) resides using the Anaconda Prompt. You can modify the following lines and change the directory in the Anaconda prompt.

```
cd C:\Users\...\AU\tests
```

Then simply type 'make' to the Anaconda Prompt and hit enter. If everything is correct you should see the result shown in Figure 2. You are highly encouraged to read through the test-bench codes, they are well commented

```
1000.00ns INFO au_fail_test failed
Traceback (most recent call last):
  File "C:\Users\erkan\cocotb_work\practice\AU\tests\AU_cocotb_test.py", line 54, in au_fail_test
    assert dut.Q.value == A - B
  File "C:\users\erkan\miniconda3\lib\site-packages\cocotb\binary.py", line 501, in __eq__
    return self.value == other
  File "C:\users\erkan\miniconda3\lib\site-packages\cocotb\binary.py", line 345, in value
    return self.integer
  File "C:\users\erkan\miniconda3\lib\site-packages\cocotb\binary.py", line 336, in integer
    return self._convert_from(self._str)
  File "C:\users\erkan\miniconda3\lib\site-packages\cocotb\binary.py", line 231, in _convert_from_u
nsigned
    return int(x.translate(_resolve_table), 2)
  File "C:\users\erkan\miniconda3\lib\site-packages\cocotb\binary.py", line 77, in __missing__
    return self.resolve_x(key)
  File "C:\users\erkan\miniconda3\lib\site-packages\cocotb\binary.py", line 59, in resolve_error
    raise ValueError("Unresolvable bit in binary string: '{}'.format(chr(key)))
ValueError: Unresolvable bit in binary string: 'x'

1000.00ns INFO running au_basic_test (2/3)
72000.00ns INFO au_basic_test passed
72000.00ns INFO running au_advanced_test (3/3)
1073000.00ns INFO au_advanced_test passed
1073000.00ns INFO
*****
** TEST                                STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
*****
** AU_cocotb_test.au_fail_test         FAIL      1000.00        0.00    1023250.55 **
** AU_cocotb_test.au_basic_test        PASS      71000.00        0.00   36347563.04 **
** AU_cocotb_test.au_advanced_test     PASS     1001000.00        0.01   68336045.57 **
*****
** TESTS=3 PASS=2 FAIL=1 SKIP=0         1073000.00        0.04   28172951.84 **
```

Figure 2: Result of a Successful Demo

5.4 Where to Learn Further About Cocotb?

Since cocotb is a novel environment there is no good way of learning it apart from testing things on your own and reading the documents. You can find all methods, variables, functions, modules, etc. explained with their source codes in the [cocotb documents](#).



Quartus® Prime Introduction Using Verilog Designs

For Quartus® Prime 18.0

Contents

1	Introduction	2
2	Background	3
3	Getting Started	4
3.1	Quartus® Prime Online Help	6
4	Starting a New Project	6
5	Design Entry Using Verilog Code	13
5.1	Using the Quartus® Prime Text Editor	14
5.1.1	Using Verilog Templates	16
5.2	Adding Design Files to a Project	16
6	Compiling the Designed Circuit	18
6.1	Errors	19
7	Pin Assignment	22
8	Programming and Configuring the FPGA Device	26
8.1	JTAG® Programming for the DE0-CV, DE0-Nano, DE10-Lite, and DE2-115 Boards	26
8.2	JTAG® Programming for the DE0-Nano-SoC, DE1-SoC Board, DE10-Nano, and DE10-Standard	28
9	Simulating the Designed Circuit	30
9.1	Performing the Simulation	34
9.1.1	Functional Simulation	34
9.1.2	Timing Simulation	35
10	Testing the Designed Circuit	36

1 Introduction

This tutorial presents an introduction to the Quartus® Prime CAD system. It gives a general overview of a typical CAD flow for designing circuits that are implemented by using FPGA devices, and shows how this flow is realized in the Quartus Prime software. The design process is illustrated by giving step-by-step instructions for using the Quartus Prime software to implement a very simple circuit in an Intel® FPGA device.

The Quartus Prime system includes full support for all of the popular methods of entering a description of the desired circuit into a CAD system. This tutorial makes use of the Verilog design entry method, in which the user specifies the desired circuit in the Verilog hardware description language. Three versions of this tutorial are available; one uses the Verilog hardware description language, another uses the VHDL hardware description language, and the third is based on defining the desired circuit in the form of a schematic diagram.

The last step in the design process involves configuring the designed circuit in an actual FPGA device. To show how this is done, it is assumed that the user has access to the Intel DE-series Development and Education board connected to a computer that has Quartus Prime software installed. A reader who does not have access to the DE-series board will still find the tutorial useful to learn how the FPGA programming and configuration task is performed.

The screen captures in the tutorial were obtained using the Quartus Prime version 18.0 Standard Edition; other versions of the software may be slightly different.

2 Background

Computer Aided Design (CAD) software makes it easy to implement a desired logic circuit by using a programmable logic device, such as a Field-Programmable Gate Array (FPGA) chip. A typical FPGA CAD flow is illustrated in Figure 1.

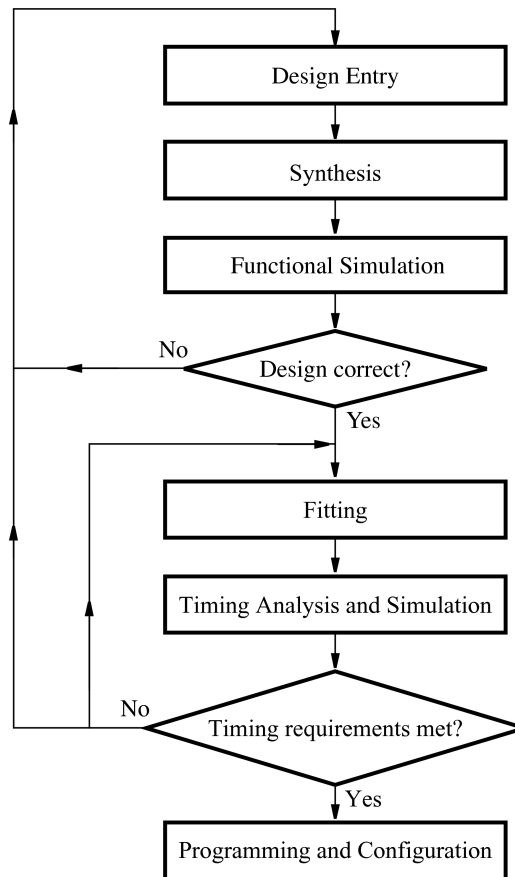


Figure 1. Typical CAD flow.

The CAD flow involves the following steps:

- **Design Entry** – the desired circuit is specified either by means of a schematic diagram, or by using a hardware description language, such as Verilog or VHDL
- **Synthesis** – the entered design is synthesized into a circuit that consists of the logic elements (LEs) provided in the FPGA chip
- **Functional Simulation** – the synthesized circuit is tested to verify its functional correctness; this simulation does not take into account any timing issues

- **Fitting** – the CAD Fitter tool determines the placement of the LEs defined in the netlist into the LEs in an actual FPGA chip; it also chooses routing wires in the chip to make the required connections between specific LEs
- **Timing Analysis** – propagation delays along the various paths in the fitted circuit are analyzed to provide an indication of the expected performance of the circuit
- **Timing Simulation** – the fitted circuit is tested to verify both its functional correctness and timing
- **Programming and Configuration** – the designed circuit is implemented in a physical FPGA chip by programming the configuration switches that configure the LEs and establish the required wiring connections

This tutorial introduces the basic features of the Quartus Prime software. It shows how the software can be used to design and implement a circuit specified by using the Verilog hardware description language. It makes use of the graphical user interface to invoke the Quartus Prime commands. Doing this tutorial, the reader will learn about:

- Creating a project
- Design entry using Verilog code
- Synthesizing a circuit specified in Verilog code
- Fitting a synthesized circuit into an Intel FPGA
- Assigning the circuit inputs and outputs to specific pins on the FPGA
- Simulating the designed circuit
- Programming and configuring the FPGA chip on Intel's DE-series board

3 Getting Started

Each logic circuit, or subcircuit, being designed with Quartus Prime software is called a *project*. The software works on one project at a time and keeps all information for that project in a single directory (folder) in the file system. To begin a new logic circuit design, the first step is to create a directory to hold its files. To hold the design files for this tutorial, we will use a directory *introtutorial*. The running example for this tutorial is a simple circuit for two-way light control.

Start the Quartus Prime software. You should see a display similar to the one in Figure 2. This display consists of several windows that provide access to all the features of Quartus Prime software, which the user selects with the computer mouse. Most of the commands provided by Quartus Prime software can be accessed by using a set of menus that are located below the title bar. For example, in Figure 2 clicking the left mouse button on the menu named File opens the menu shown in Figure 3. Clicking the left mouse button on the entry Exit exits from Quartus Prime software. In general, whenever the mouse is used to select something, the *left* button is used. Hence we will not normally specify which button to press. In the few cases when it is necessary to use the *right* mouse button, it will be specified explicitly.

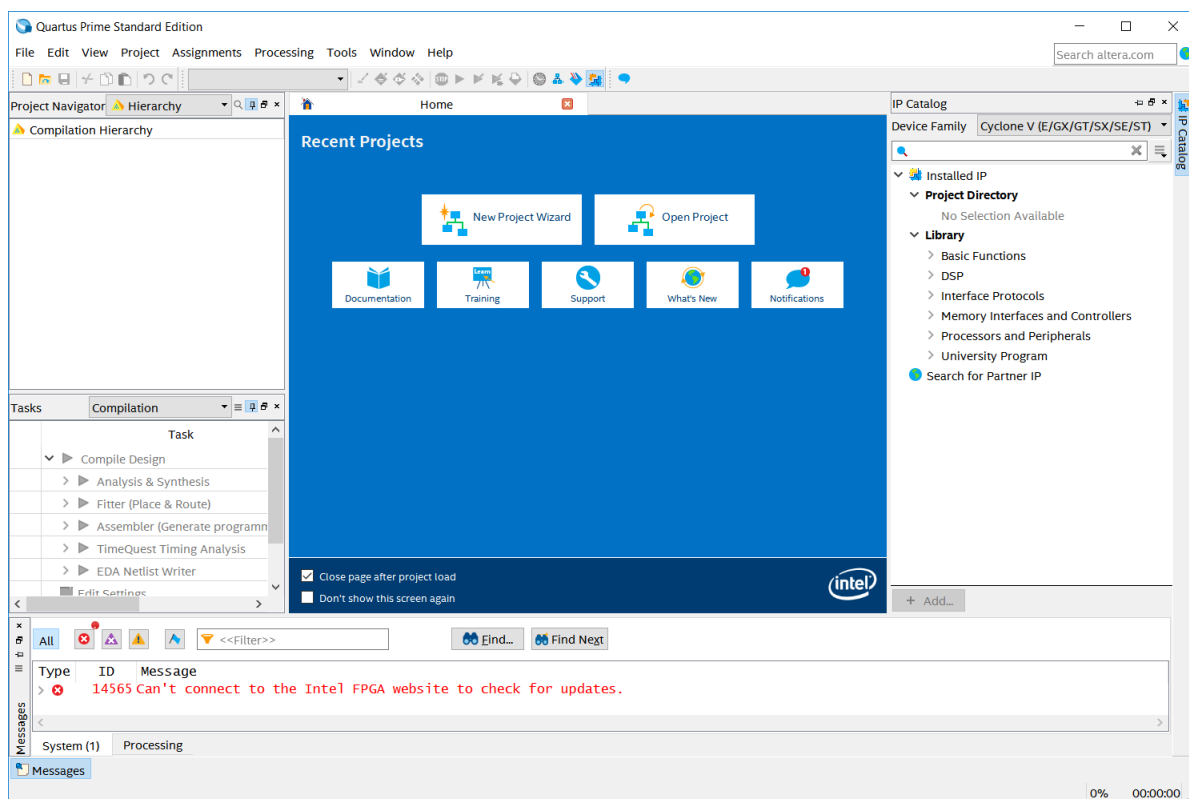


Figure 2. The main Quartus Prime display.

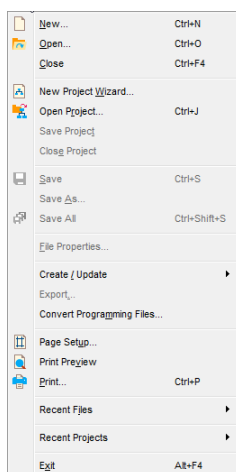


Figure 3. An example of the File menu.

For some commands it is necessary to access two or more menus in sequence. We use the convention **Menu1 > Menu2 > Item** to indicate that to select the desired command the user should first click the left mouse button on **Menu1**, then within this menu click on **Menu2**, and then within **Menu2** click on **Item**. For example, **File > Exit** uses the mouse to exit from the system. Many commands can be invoked by clicking on an icon displayed in one of the toolbars. To see the command associated with an icon, position the mouse over the icon and the command name will be shown in the status bar at the bottom of the screen.

3.1 Quartus® Prime Online Help

Quartus Prime software provides comprehensive online documentation that answers many of the questions that may arise when using the software. The documentation is accessed from the **Help** menu. To get some idea of the extent of documentation provided, it is worthwhile for the reader to browse through the **Help** menu.

The user can quickly search through the Help topics by using the search box in the top right corner of the main Quartus display. Another method, context-sensitive help, is provided for quickly finding documentation for specific topics. While using most applications, pressing the **F1** function key on the keyboard opens a Help display that shows the commands available for the application.

4 Starting a New Project

To start working on a new design we first have to define a new *design project*. Quartus Prime software makes the designer's task easy by providing support in the form of a *wizard*. Create a new project as follows:

1. Select **File > New Project Wizard** and click **Next** to reach the window in Figure 4, which asks for the name and directory of the project.
2. Set the working directory to be *introtutorial*; of course, you can use some other directory name of your choice if you prefer. The project must have a name, which is usually the same as the top-level design entity that will be included in the project. Choose *light* as the name for both the project and the top-level entity, as shown in Figure 4. Press **Next**. Since we have not yet created the directory *introtutorial*, Quartus Prime software displays the pop-up box in Figure 5 asking if it should create the desired directory. Click **Yes**, which leads to the window in Figure 6.

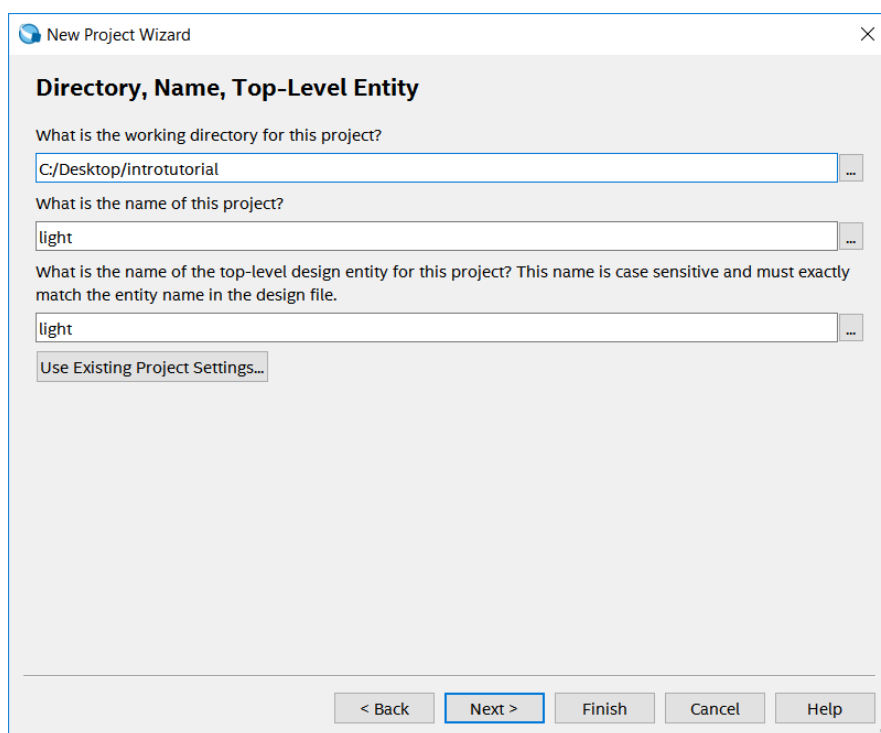


Figure 4. Creation of a new project.

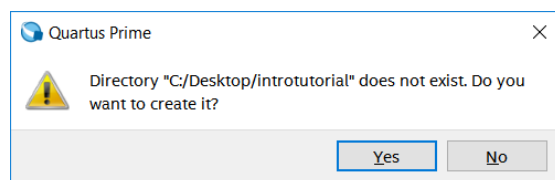


Figure 5. Quartus Prime software can create a new directory for the project.

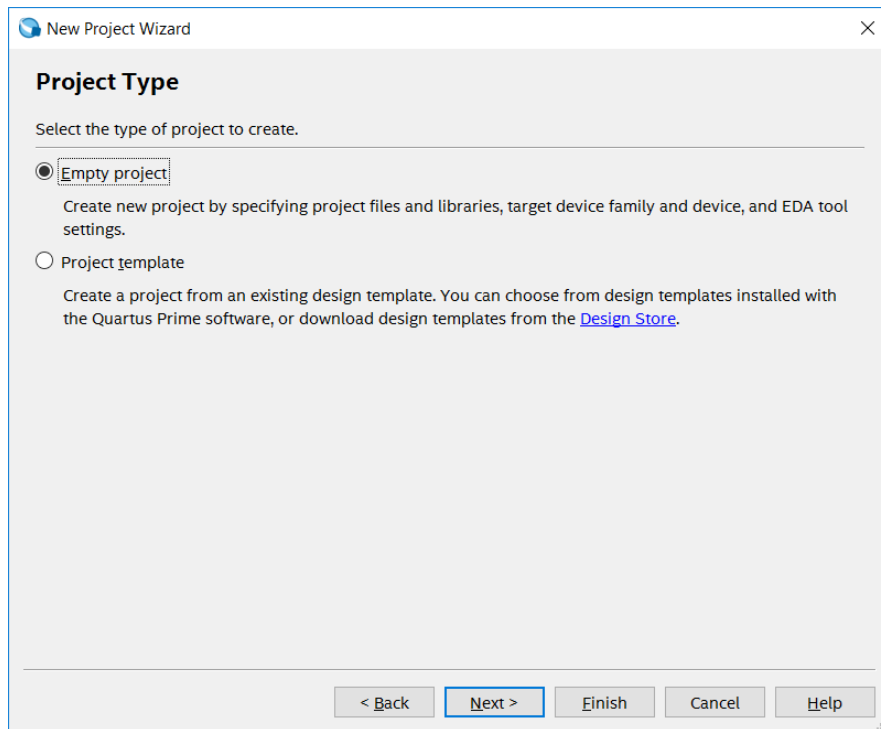


Figure 6. Choosing the project type.

3. The Project Type window, shown in Figure 6, allows you to choose from the Empty project and the Project template options. For this tutorial, choose Empty project as we will be creating a project from scratch, and press Next which leads to the window in Figure 7.

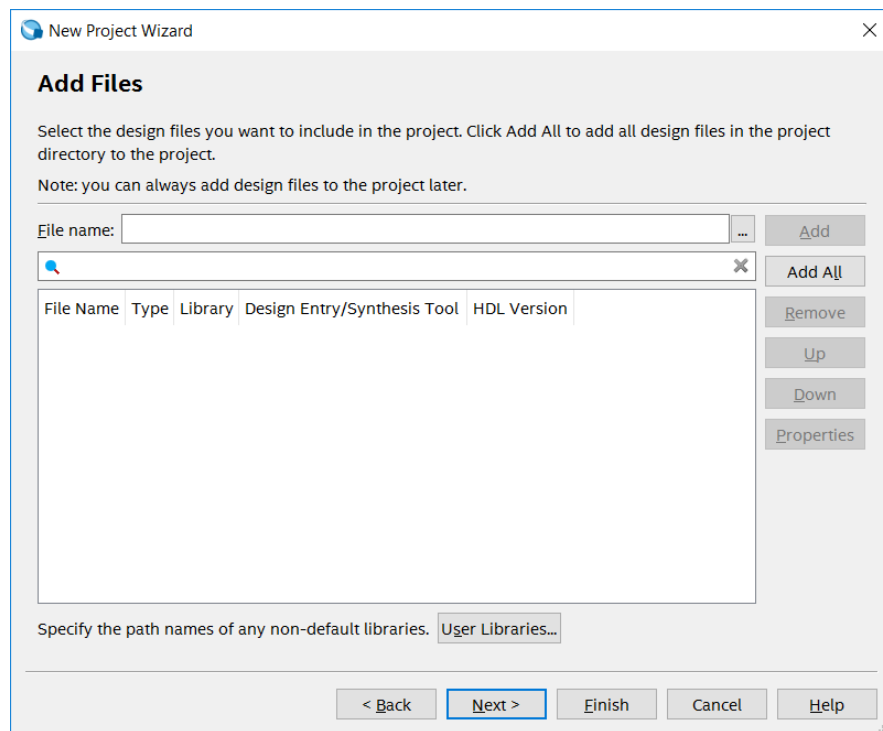


Figure 7. The wizard can include user-specified design files.

4. The wizard makes it easy to specify which existing files (if any) should be included in the project. Assuming that we do not have any existing files, click **Next**, which leads to the window in Figure 8.

Family, Device & Board Settings

Device Board

Select the family and device you want to target for compilation.
You can install additional device support with the Install Devices command on the Tools menu.

To determine the version of the Quartus Prime software in which your target device is supported, refer to the [Device Support List](#) webpage.

Device family

Family: Cyclone V (E/GX/GT/SX/SE/ST)

Device: All

Target device

☐ Auto device selected by the Fitter

☒ Specific device selected in 'Available devices' list

☐ Other: n/a

Show in 'Available devices' list

Package: Any

Pin count: Any

Core speed grade: Any

Name filter:

☒ Show advanced devices

Available devices:

Name	Core Voltage	ALMs	Total I/Os	GPIOs	GXB Channel PMA	GXB Channel PCS
5CSEMA4U23C8	1.1V	15880	314	314	0	0
5CSEMA4U23I7	1.1V	15880	314	314	0	0
5CSEMA5F31A7	1.1V	32070	457	457	0	0
5CSEMA5F31C6	1.1V	32070	457	457	0	0
5CSEMA5F31C7	1.1V	32070	457	457	0	0
5CSEMA5F31C8	1.1V	32070	457	457	0	0
5CSEMA5F31I7	1.1V	32070	457	457	0	0
5CSEMA5U23A7	1.1V	32070	314	314	0	0
5CSEMA5U23C6	1.1V	32070	314	314	0	0
5CSEMA5U23C7	1.1V	32070	314	314	0	0

< Back Next > Finish Cancel Help

Figure 8. Choose the device family and a specific device.

- We have to specify the type of device in which the designed circuit will be implemented. Choose the Cyclone® series device family for your DE-series board. We can let Quartus Prime software select a specific device in the family, or we can choose the device explicitly. We will take the latter approach. From the list of available devices, choose the appropriate device name for your DE-series board. A list of devices names on DE-series boards can be found in Table 1. Press Next, which opens the window in Figure 9.

Board	Device Name
DE0-CV	Cyclone V 5CEBA4F23C7
DE0-Nano	Cyclone IVE EP4CE22F17C6
DE0-Nano-SoC	Cyclone V SoC 5CSEMA4U23C6
DE1-SoC	Cyclone V SoC 5CSEMA5F31C6
DE2-115	Cyclone IVE EP4CE115F29C7
DE10-Lite	Max 10 10M50DAF484C7G
DE10-Standard	Cyclone V SoC 5CSXFC6D6F31C6
DE10-Nano	Cyclone V SE 5CSEBA6U2317

Table 1. DE-series FPGA device names

EDA Tool Settings

Specify the other EDA tools used with the Quartus Prime software to develop your project.

EDA tools:

Tool Type	Tool Name	Format(s)	Run Tool Automatically
Design Entry/S...	<None>	<None>	<input type="checkbox"/> Run this tool automatically to synthesize the current design
Simulation	<None>	<None>	<input type="checkbox"/> Run gate-level simulation automatically after compilation
Board-Level	Timing	<None>	
	Symbol	<None>	
	Signal Integrity	<None>	
	Boundary Scan	<None>	

< Back Next > Finish Cancel Help

Figure 9. Other EDA tools can be specified.

- The user can specify any third-party tools that should be used. A commonly used term for CAD software for electronic circuits is *EDA tools*, where the acronym stands for Electronic Design Automation. This term is used in Quartus Prime messages that refer to third-party tools, which are the tools developed and marketed by companies other than Intel. Since we will rely solely on Quartus Prime tools, we will not choose any other tools. Press Next.

7. A summary of the chosen settings appears in the screen shown in Figure 10. Press Finish, which returns to the main Quartus Prime window, but with *light* specified as the new project, in the title bar, as indicated in Figure 11.

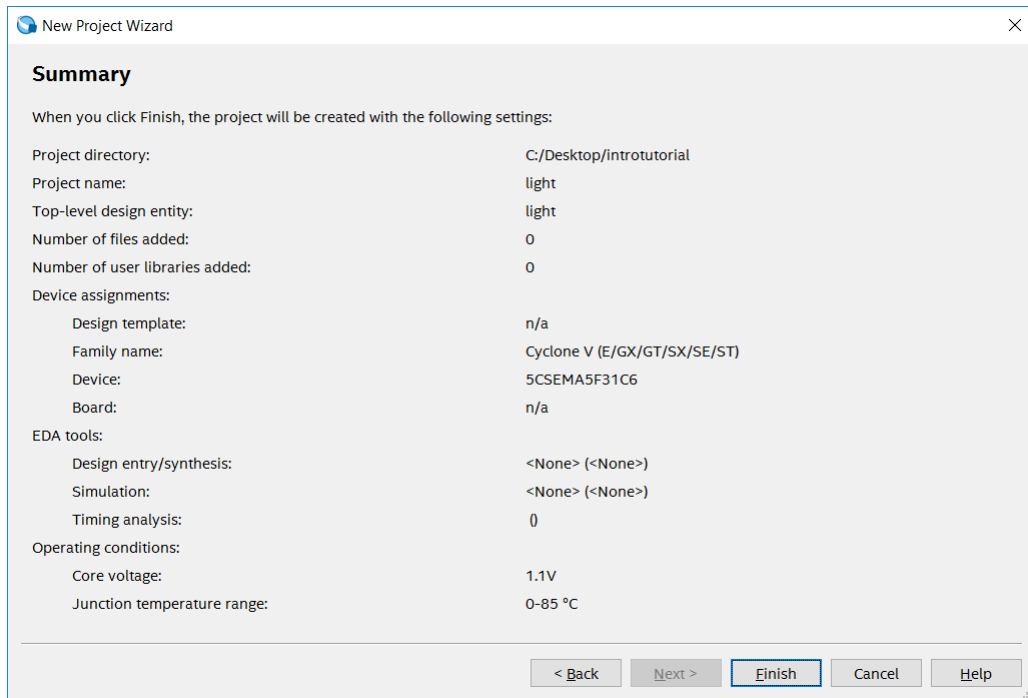


Figure 10. Summary of project settings.

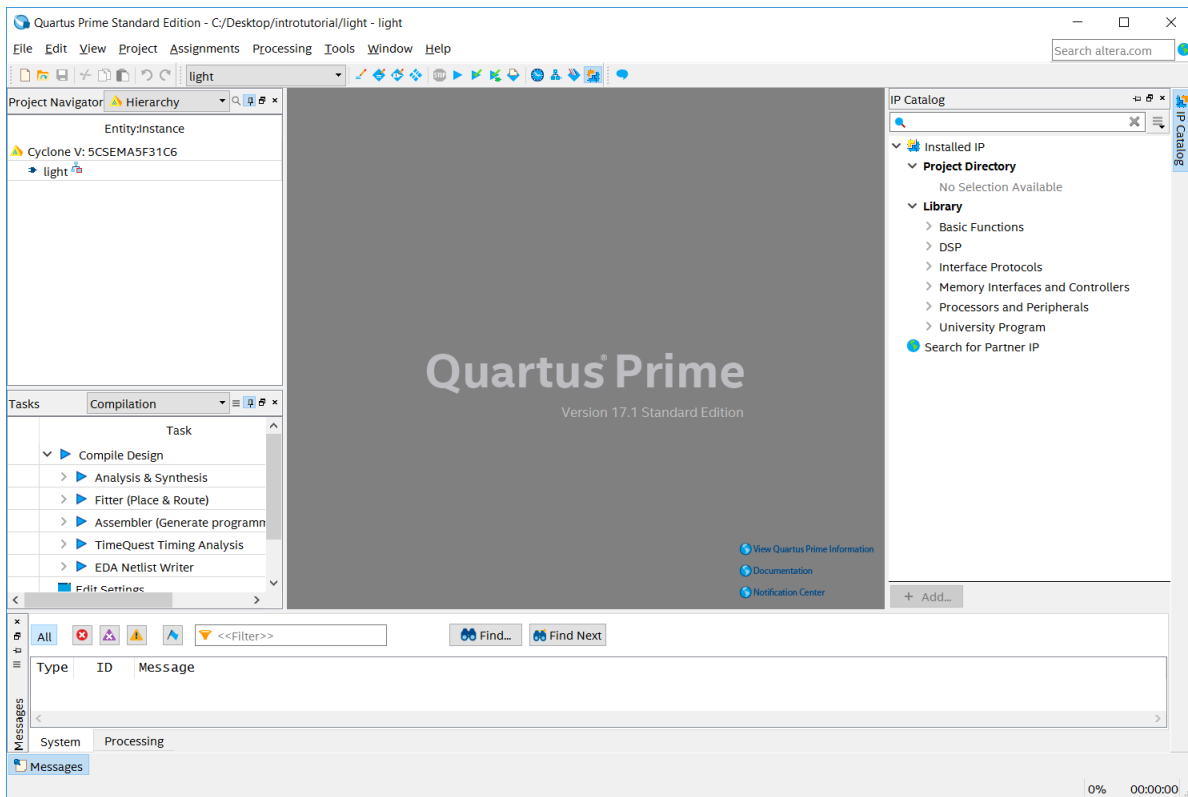


Figure 11. The Quartus Prime window for a created project.

5 Design Entry Using Verilog Code

As a design example, we will use the two-way light controller circuit shown in Figure 12. The circuit can be used to control a single light from either of the two switches, x_1 and x_2 , where a closed switch corresponds to the logic value 1. The truth table for the circuit is also given in the figure. Note that this is just the Exclusive-OR function of the inputs x_1 and x_2 , but we will specify it using the gates shown.

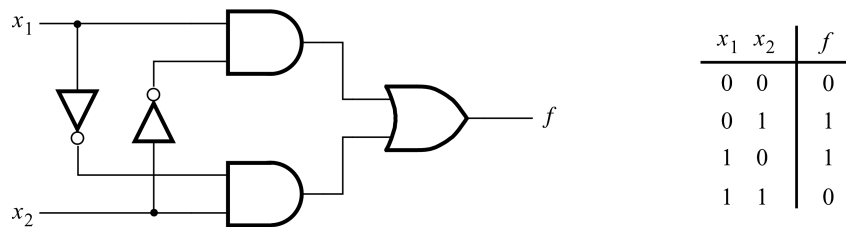


Figure 12. The light controller circuit.

The required circuit is described by the Verilog code in Figure 13. Note that the Verilog module is called *light* to match the name given in Figure 4, which was specified when the project was created. This code can be typed into a file by using any text editor that stores ASCII files, or by using the Quartus Prime text editing facilities. While the file can be given any name, it is a common designers' practice to use the same name as the name of the top-level Verilog module. The file name must include the extension *v*, which indicates a Verilog file. So, we will use the name *light.v*.

```
module light (x1, x2, f);  
    input x1, x2;  
    output f;  
    assign f = (x1 & ~x2) | (~x1 & x2);  
endmodule
```

Figure 13. Verilog code for the circuit in Figure 11.

5.1 Using the Quartus® Prime Text Editor

This section shows how to use the Quartus Prime Text Editor. You can skip this section if you prefer to use some other text editor to create the Verilog source code file, which we will name *light.v*.

Select File > New to get the window in Figure 14, choose Verilog HDL File, and click OK. This opens the Text Editor window. The first step is to specify a name for the file that will be created. Select File > Save As to open the pop-up box depicted in Figure 15. In the box labeled Save as type choose Verilog HDL File. In the box labeled File name type *light*. Put a checkmark in the box Add file to current project. Click Save, which puts the file into the directory *introtutorial* and leads to the Text Editor window shown in Figure 16. Enter the Verilog code in Figure 13 into the Text Editor and save the file by typing File > Save, or by typing the shortcut Ctrl-s.

Most of the commands available in the Text Editor are self-explanatory. Text is entered at the *insertion point*, which is indicated by a thin vertical line. The insertion point can be moved either by using the keyboard arrow keys or by using the mouse. Two features of the Text Editor are especially convenient for typing Verilog code. First, the editor can display different types of Verilog statements in different colors, which is the default choice. Second, the editor can automatically indent the text on a new line so that it matches the previous line. Such options can be controlled by the settings in Tools > Options > Text Editor.

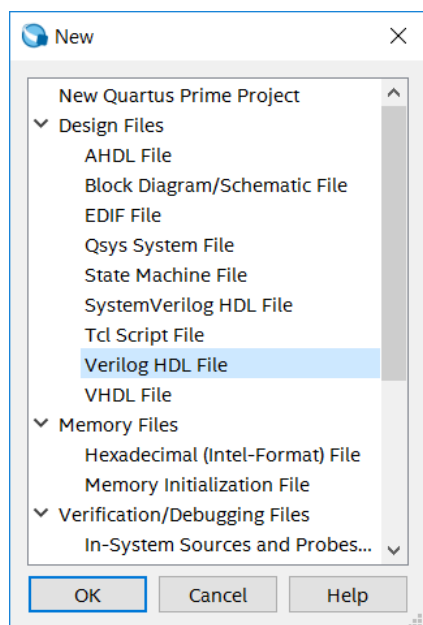


Figure 14. Choose to prepare a Verilog file.

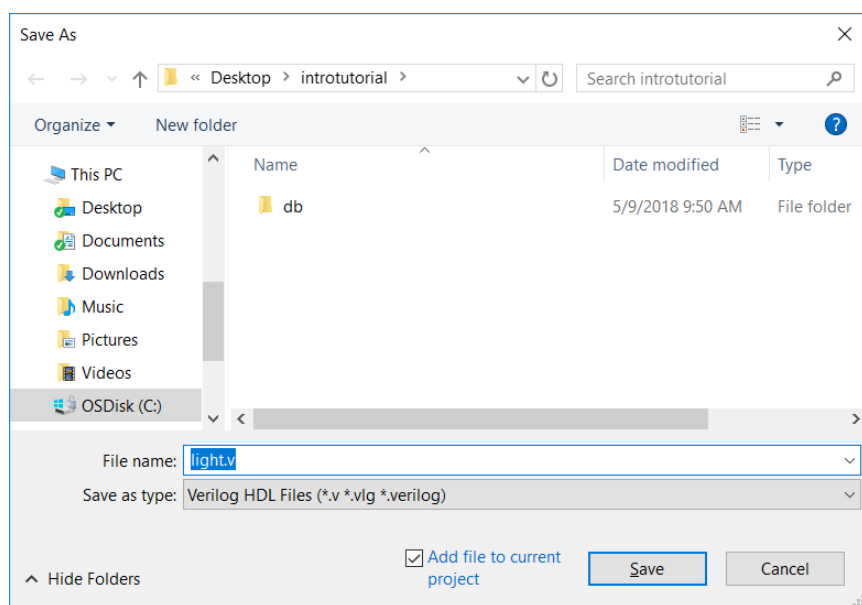


Figure 15. Name the file.

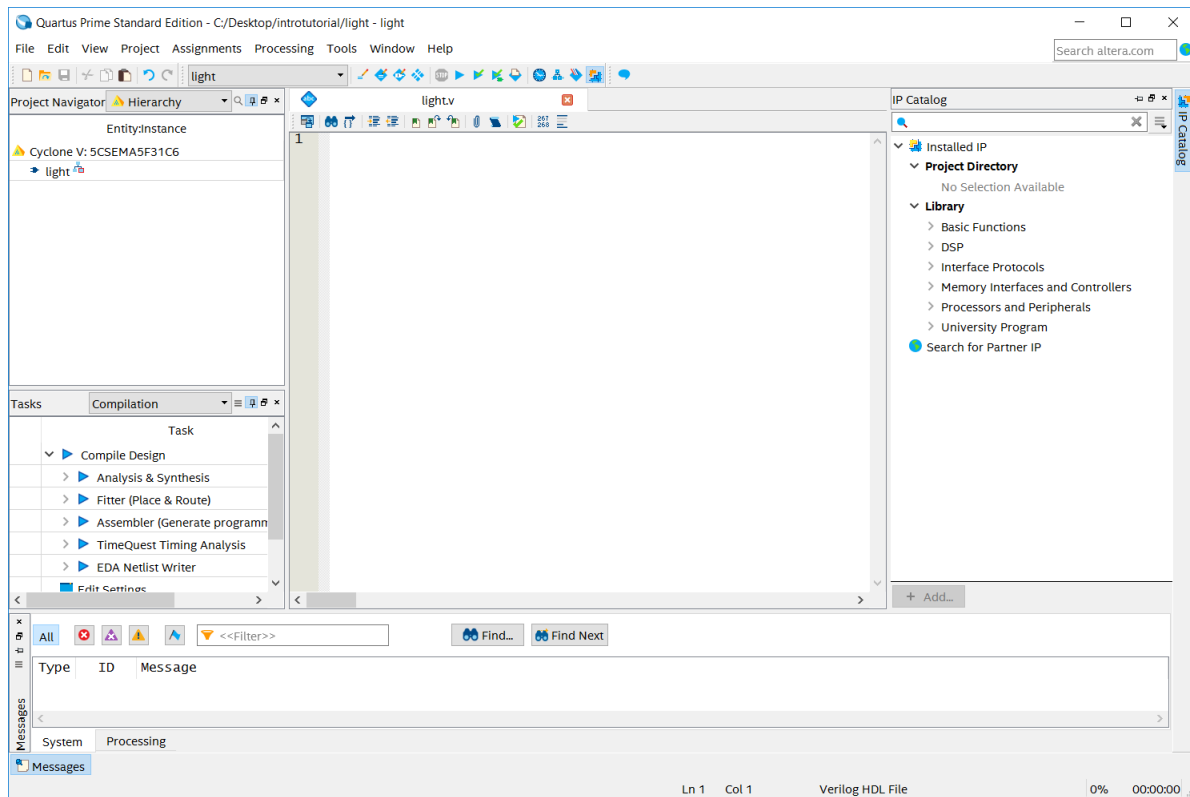


Figure 16. Text Editor window.

5.1.1 Using Verilog Templates

The syntax of Verilog code is sometimes difficult for a designer to remember. To help with this issue, the Text Editor provides a collection of *Verilog templates*. The templates provide examples of various types of Verilog statements, such as a **module** declaration, an **always** block, and assignment statements. It is worthwhile to browse through the templates by selecting **Edit > Insert Template > Verilog HDL** to become familiar with this resource.

5.2 Adding Design Files to a Project

As we indicated when discussing Figure 7, you can tell Quartus Prime software which design files it should use as part of the current project. To see the list of files already included in the *light* project, select **Assignments > Settings**, which leads to the window in Figure 17. As indicated on the left side of the figure, click on the item **Files**. An alternative way of making this selection is to choose **Project > Add/Remove Files in Project**.

If you used the Quartus Prime Text Editor to create the file and checked the box labeled **Add file to current project**, as described in Section 5.1, then the *light.v* file is already a part of the project and will be listed in the window in Figure 17. Otherwise, the file must be added to the project. So, if you did not use the Quartus Prime Text Editor, then place a copy of the file *light.v*, which you created using some other text editor, into the directory *introtutorial*. To add this file to the project, click on the ... button next to the box labeled **File name** in Figure 17 to get the pop-up

window in Figure 18. Select the *light.v* file and click Open. The selected file is now indicated in the File name box in Figure 17. Click Add then OK to include the *light.v* file in the project. We should mention that in many cases the Quartus Prime software is able to automatically find the right files to use for each entity referenced in Verilog code, even if the file has not been explicitly added to the project. However, for complex projects that involve many files it is a good design practice to specifically add the needed files to the project, as described above.

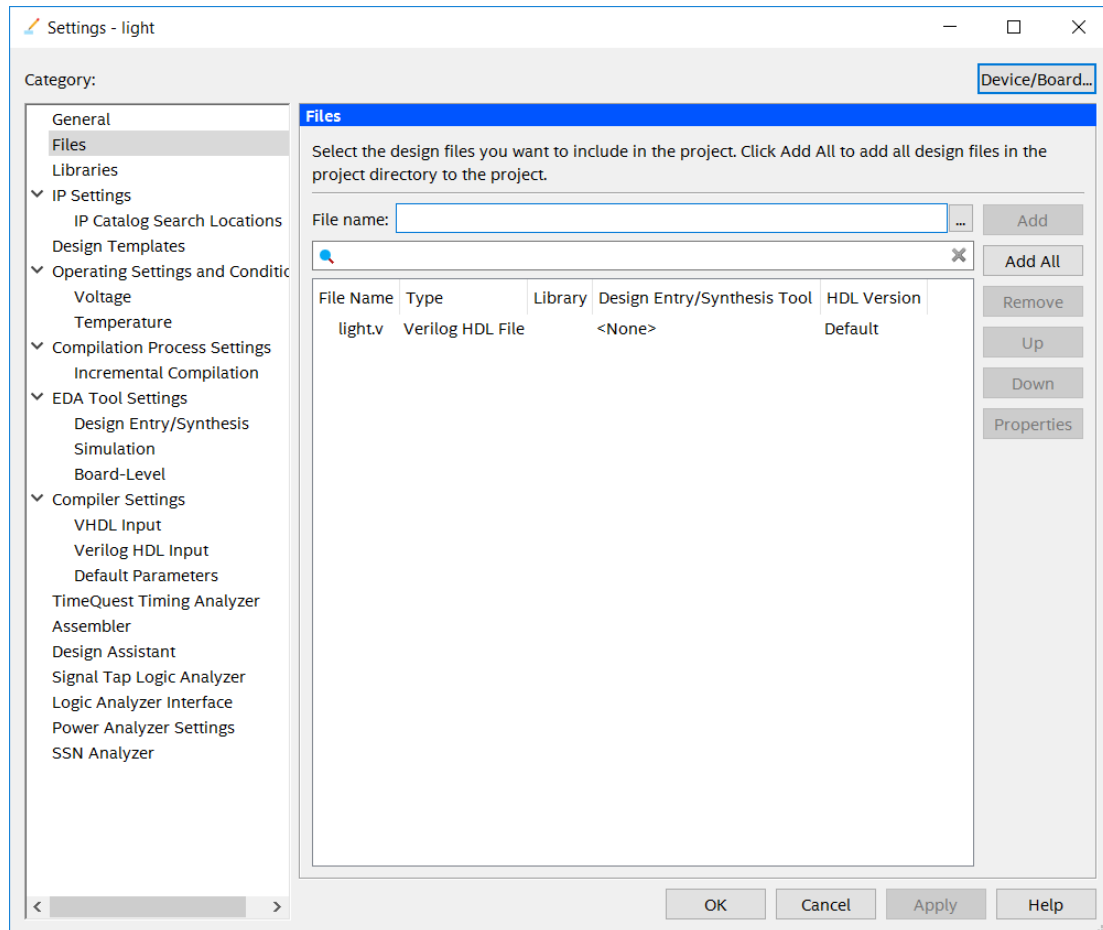


Figure 17. Settings window.

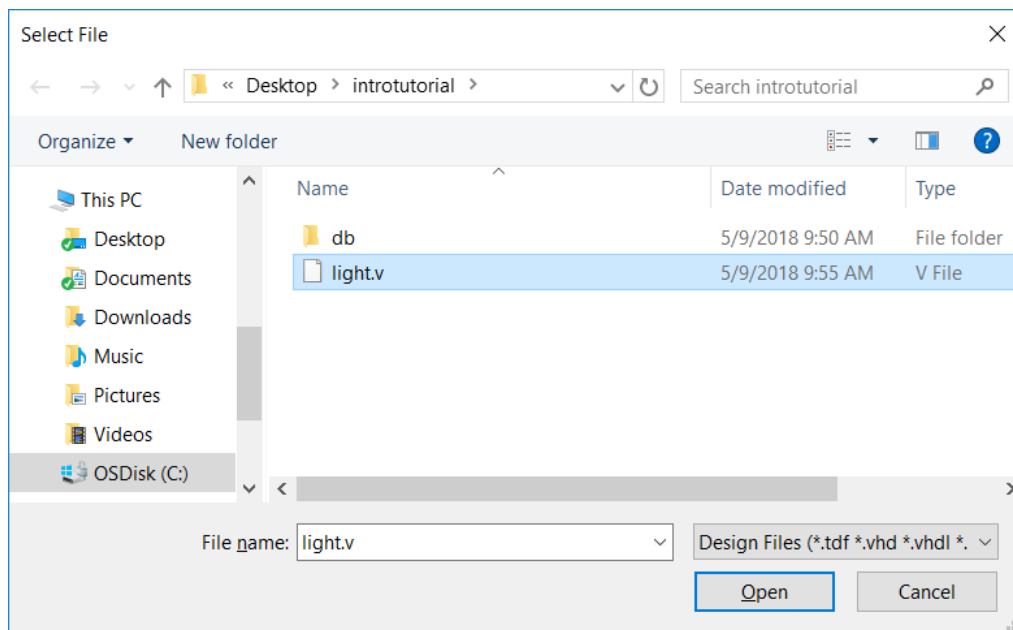




Figure 18. Select the file.

6 Compiling the Designed Circuit

The Verilog code in the file *light.v* is processed by several Quartus Prime tools that analyze the code, synthesize the circuit, and generate an implementation of it for the target chip. These tools are controlled by the application program called the *Compiler*.

Run the Compiler by selecting Processing > Start Compilation, or by clicking on the toolbar icon  that looks like a blue triangle. Your project must be saved before compiling. As the compilation moves through various stages, its progress is reported in a window on the left side of the Quartus Prime display. In the message window, at the bottom of the figure, various messages are displayed throughout the compilation process. In case of errors, there will be appropriate messages given.

When the compilation is finished, a compilation report is produced. A tab showing this report is opened automatically, as seen in Figure 21. The tab can be closed in the normal way, and it can be opened at any time either by selecting Processing > Compilation Report or by clicking on the icon . The report includes a number of sections listed on the left side. Figure 21 displays the Compiler Flow Summary section, which indicates that only one logic element and three pins are needed to implement this tiny circuit on the selected FPGA chip.

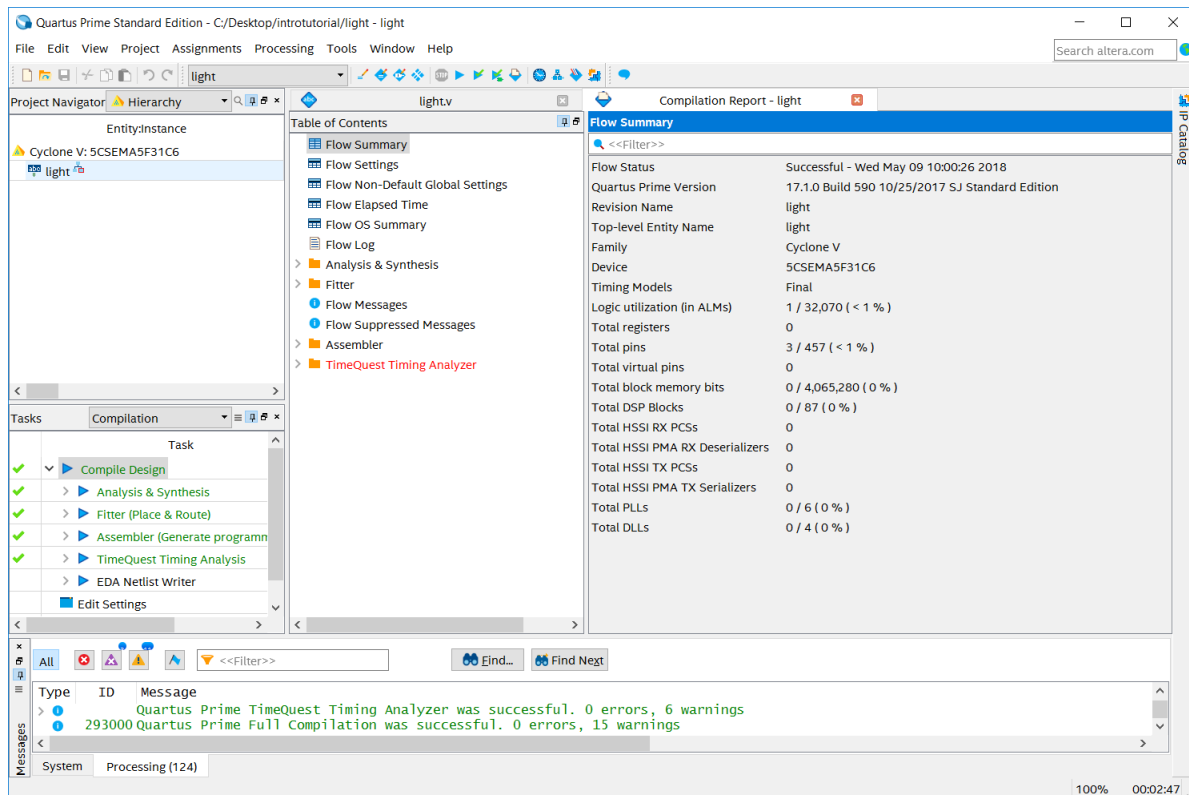



Figure 19. Display after a successful compilation.

6.1 Errors

Quartus Prime software displays messages produced during compilation in the Messages window. If the Verilog design file is correct, one of the messages will state that the compilation was successful and that there are no errors.

If the Compiler does not report zero errors, then there is at least one mistake in the Verilog code. In this case a message corresponding to each error found will be displayed in the Messages window. Double-clicking on an error message will highlight the offending statement in the Verilog code in the Text Editor window. Similarly, the Compiler may display some warning messages. Their details can be explored in the same way as in the case of error messages. The user can obtain more information about a specific error or warning message by selecting the message and pressing the F1 function key.

To see the effect of an error, open the file *light.v*. Remove the semicolon in the **assign** statement, illustrating a typographical error that is easily made. Compile the erroneous design file by clicking on the  icon. A pop-up box will ask if the changes made to the *light.v* file should be saved; click **Yes**. After trying to compile the circuit, Quartus Prime software will display error messages in the Messages window, and show that the compilation failed in the **Analysis & Synthesis** stage of the compilation process. The compilation report summary, given in Figure 20, confirms the failed result. In the Table of Contents panel, expand the **Analysis & Synthesis** part of the report and then select **Messages** to have the messages displayed as shown in Figure 21. The Compilation Report can

be displayed as a separate window as in Figure 21 by right-clicking its tab and selecting **Detach Window**, and can be reattached by clicking **Window > Attach Window**. Double-click on the first error message. Quartus Prime software responds by opening the *light.v* file and highlighting the statement which is affected by the error, as shown in Figure 22. Correct the error and recompile the design.

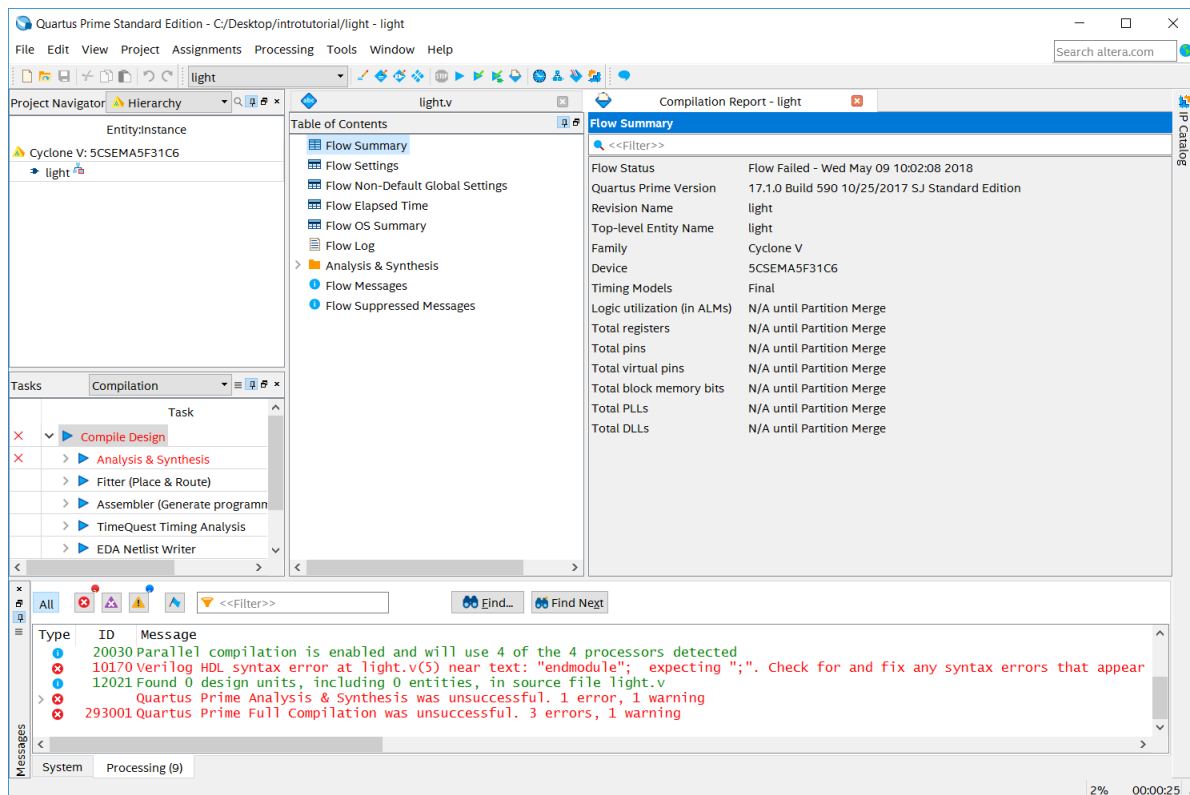


Figure 20. Compilation report for the failed design.

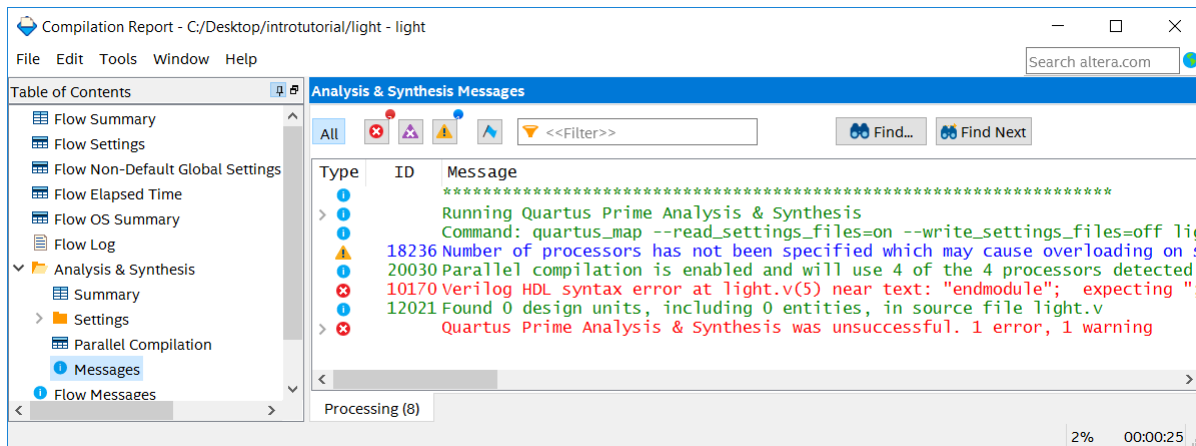


Figure 21. Error messages.

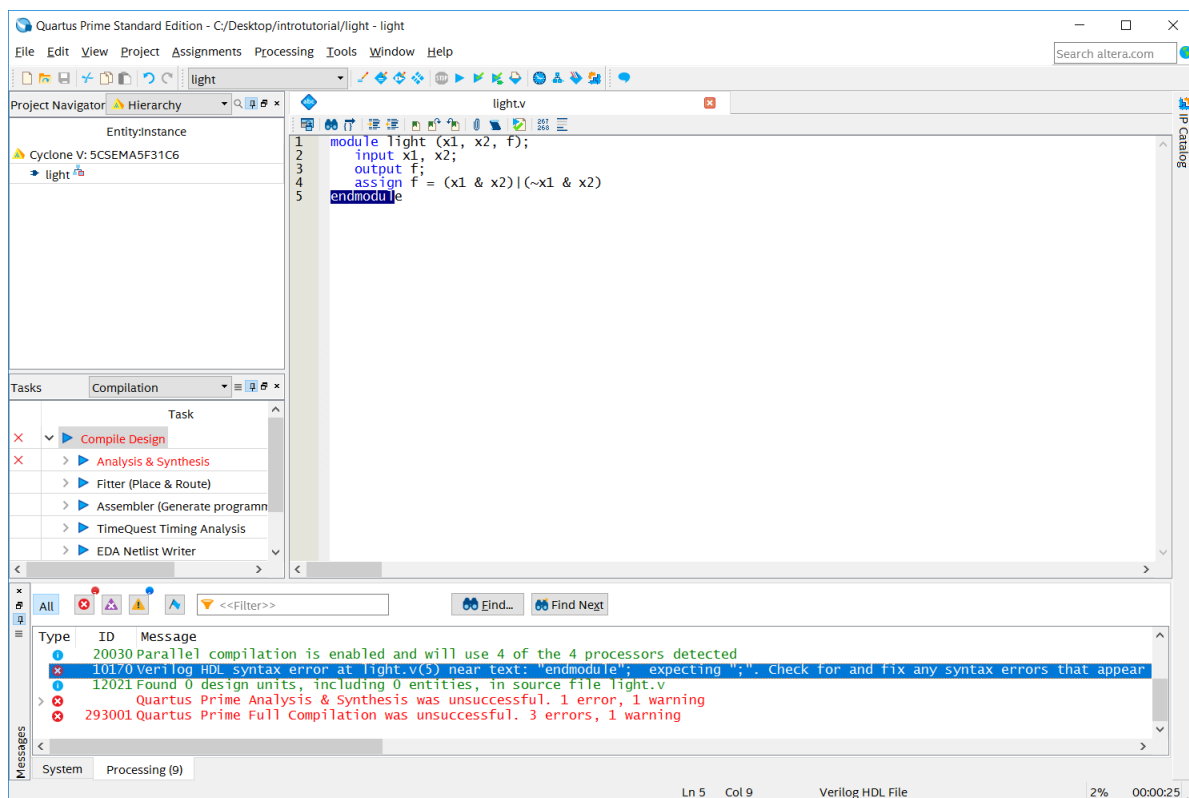


Figure 22. Identifying the location of the error.

7 Pin Assignment

During the compilation above, the Quartus Prime Compiler was free to choose any pins on the selected FPGA to serve as inputs and outputs. However, the DE-series board has hardwired connections between the FPGA pins and the other components on the board. We will use two toggle switches, labeled SW_0 and SW_1 , to provide the external inputs, x_1 and x_2 , to our example circuit. These switches are connected to the FPGA pins listed in Table 2. We will connect the output f to a light-emitting diode on your DE-series board. For the DE2-115 we will use a green LED: $LEDG_0$. On the DE0-CV, DE1-SoC, DE-10 Lite and DE10-Standard we will use $LEDR_0$. On the DE0-Nano and DE0-Nano-SoC, we will use LED_0 . The FPGA pin assignment for the LEDs can also be found in Table 2.

Component	SW_0	SW_1	$LEDG_0$, LED_0 , or $LEDR_0$
DE0-CV	PIN_U13	PIN_V13	PIN_AA2
DE0-Nano	PIN_M1	PIN_T8	PIN_A1
DE0-Nano-SoC	PIN_L10	PIN_L9	PIN_W15
DE2-115	PIN_AB28	PIN_AC28	PIN_E21
DE1-SoC	PIN_AB12	PIN_AC12	PIN_V16
DE10-Lite	PIN_C10	PIN_C11	PIN_A8
DE10-Standard	PIN_AB30	PIN_AB28	PIN_AA24
DE10-Nano	PIN_Y24	PIN_W24	PIN_W15

Table 2. DE-Series Pin Assignments

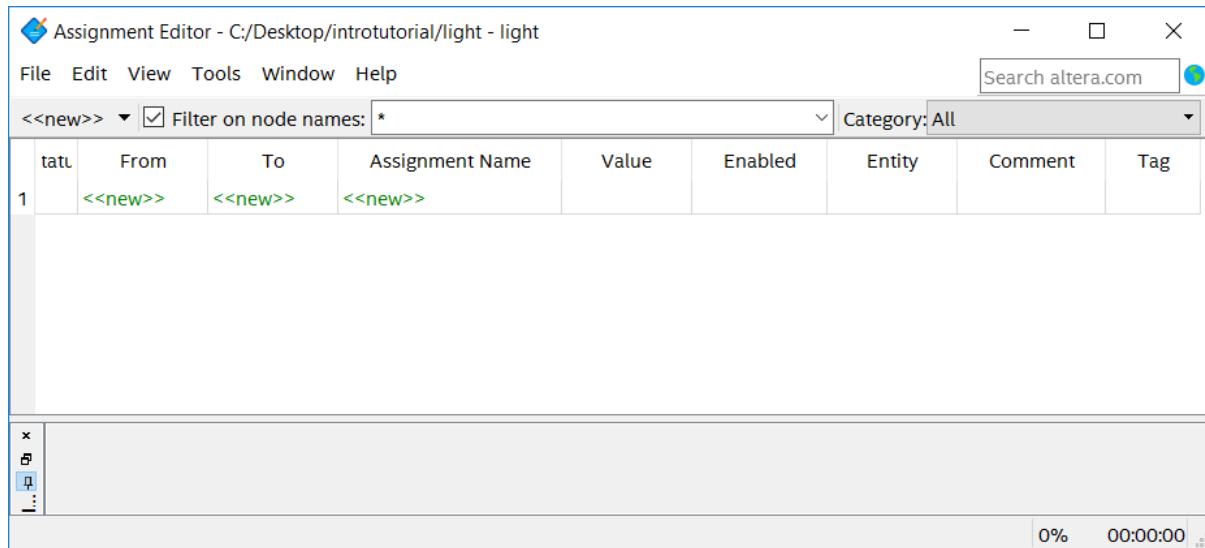





Figure 23. The Assignment Editor window.

Pin assignments are made by using the Assignment Editor. Select **Assignments > Assignment Editor** to reach the window in Figure 23 (shown here as a detached window). In the **Category** drop-down menu select **All**. Click on the **<<new>>** button located near the top left corner to make a new item appear in the table. Double click the box

under the column labeled To so that the Node Finder button  appears. Click on the button (not the drop down arrow) to reach the window in Figure 24. Click on  and  to show or hide more search options. In the Filter drop-down menu select Pins: all. Then click the List button to display the input and output pins to be assigned: f , $x1$, and $x2$. Click on $x1$ as the first pin to be assigned and click the > button; this will enter $x1$ in the Selected Nodes box. Click OK. $x1$ will now appear in the box under the column labeled To. Alternatively, the node name can be entered directly by double-clicking the box under the To column and typing in the node name.

Follow this by double-clicking on the box to the right of this new $x1$ entry, in the column labeled Assignment Name. Now, the drop-down menu in Figure 25 appears. Scroll down and select Location (Accepts wildcards/groups). Instead of scrolling down the menu to find the desired item, you can just type the first letter of the item in the Assignment Name box. In this case the desired item happens to be the first item beginning with L. Finally, double-click the box in the column labeled Value. Type the pin assignment corresponding to SW_0 for your DE-series board, as listed in Table 2.

Use the same procedure to assign input $x2$ and output f to the appropriate pins listed in Table 2. An example using a DE1-SoC board is shown in Figure 26. To save the assignments made, choose File > Save. You can also simply close the Assignment Editor window, in which case a pop-up box will ask if you want to save the changes to assignments; click Yes. Recompile the circuit, so that it will be compiled with the correct pin assignments.

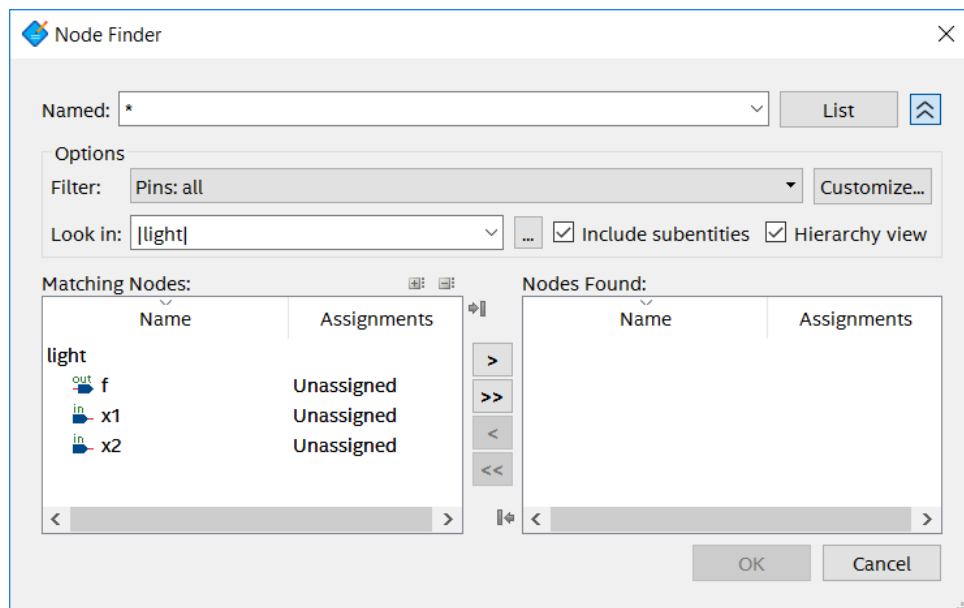


Figure 24. The Node Finder displays the input and output names.

Assignment Name	Value	Enabled	Entity	Comment	Tag
Location (Accepts wildcards/groups)					
MLAB Add Timing Constraints For Mixed-Port Feed-Through Mode Setting Don't Care (Accepts wildcards/groups)					
Manual Logic Duplication (Accepts wildcards/groups)					
Match PLL Compensation Clock (Accepts wildcards/groups)					

Figure 25. The available assignment names for a DE-series board.

Assignment Editor - C:/Desktop/introtutorial/light - light

File Edit View Tools Window Help

Search altera.com

<<new>> Filter on node names: * Category: All

	tatu	From	To	Assignment Name	Value	Enabled	Entity	Comment	Tag
1	✓		in x1	Location	PIN_AB12	Yes			
2	✓		in x2	Location	PIN_AC12	Yes			
3	✓		out f	Location	PIN_V16	Yes			
4		<<new>>	<<new>>	<<new>>					

0% 00:00:00

Figure 26. The complete assignment.

The DE-series board has fixed pin assignments. Having finished one design, the user will want to use the same pin assignment for subsequent designs. Going through the procedure described above becomes tedious if there are many pins used in the design. A useful Quartus Prime feature allows the user to both export and import the pin assignments from a special file format, rather than creating them manually using the Assignment Editor. A simple file format that can be used for this purpose is the *Quartus Settings File (QSF)* format. The format for the file for our simple project (on a DE1-SoC board) is

```
set_location_assignment PIN_AB12 -to x1
set_location_assignment PIN_AC12 -to x2
set_location_assignment PIN_V16 -to f
```

By adding lines to the file, any number of pin assignments can be created. Such *qsf* files can be imported into any design project.

If you created a pin assignment for a particular project, you can export it for use in a different project. To see how this is done, open again the Assignment Editor to reach the window in Figure 26. Select Assignments > Export

Assignment which leads to the window in Figure 27. Here, the file *light.qsf* is available for export. Click on OK. If you now look in the directory, you will see that the file *light.qsf* has been created.

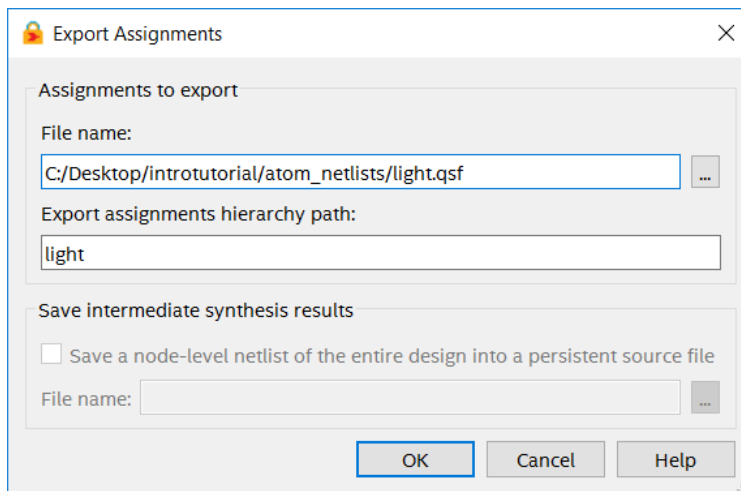


Figure 27. Exporting the pin assignment.

You can import a pin assignment by choosing **Assignments > Import Assignments**. This opens the dialogue in Figure 28 to select the file to import. Type the name of the file, including the *qsf* extension and the full path to the directory that holds the file, in the File Name box and press OK. Of course, you can also browse to find the desired file.

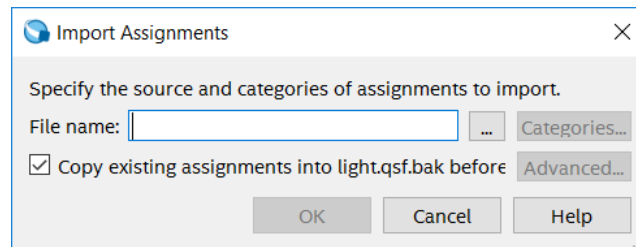


Figure 28. Importing the pin assignment.

For convenience when using large designs, all relevant pin assignments for the DE-series board are given in individual files. For example, the DE1-SoC pin assignments can be found in the *DE1_SoC.qsf* file, which is available from Intel's FPGA University Program website. This file uses the names found in the *DE1-SoC User Manual*. If we wanted to make the pin assignments for our example circuit by importing this file, then we would have to use the same names in our Block Diagram/Schematic design file; namely, *SW[0]*, *SW[1]* and *LEDG[0]* for *x1*, *x2* and *f*, respectively. Since these signals are specified in the *DE1_SoC.qsf* file as elements of vectors *SW* and *LEDG*, we must refer to them in the same way in our design file. For example, in the *DE1_SoC.qsf* file the 10 toggle switches are called *SW[9]* to *SW[0]*. In a design file they can also be referred to as a vector *SW[9..0]*.

8 Programming and Configuring the FPGA Device

The FPGA device must be programmed and configured to implement the designed circuit. The required configuration file is generated by the Quartus Prime Compiler's Assembler module. Intel's DE-series board allows the configuration to be done in two different ways, known as JTAG* and AS modes. The configuration data is transferred from the host computer (which runs the Quartus Prime software) to the board by means of a cable that connects a USB port on the host computer to the USB-Blaster connector on the board. To use this connection, it is necessary to have the USB-Blaster driver installed. If this driver is not already installed, consult the tutorial *Getting Started with Intel's DE-Series Boards* for information about installing the driver. Before using the board, make sure that the USB cable is properly connected and turn on the power supply switch on the board.

In the JTAG mode, the configuration data is loaded directly into the FPGA device. The acronym JTAG stands for Joint Test Action Group. This group defined a simple way for testing digital circuits and loading data into them, which became an IEEE* standard. If the FPGA is configured in this manner, it will retain its configuration as long as the power remains turned on. The configuration information is lost when the power is turned off. The second possibility is to use the Active Serial (AS) mode. In this case, a configuration device that includes some flash memory is used to store the configuration data. Quartus Prime software places the configuration data into the configuration device on the DE-series board. Then, this data is loaded into the FPGA upon power-up or reconfiguration. Thus, the FPGA need not be configured by the Quartus Prime software if the power is turned off and on. The choice between the two modes is made by switches on the DE-series board. Consult your manual for the location of this switch on your DE-series board. The boards should be set to JTAG mode by default. This tutorial discusses only the JTAG programming mode.

8.1 JTAG* Programming for the DE0-CV, DE0-Nano, DE10-Lite, and DE2-115 Boards

For the DE0-CV, DE0-Nano, DE10-Lite, and DE2-115 Boards, the programming and configuration task is performed as follows. If using the DE1-SoC board, then the instructions in the following section should be followed. To program the FPGA chip, the RUN/PROG switch on the board must be in the RUN position. Select **Tools > Programmer** to reach the window in Figure 29. Here it is necessary to specify the programming hardware and the mode that should be used. If not already chosen by default, select JTAG in the Mode box. Also, if the USB-Blaster is not chosen by default, press the **Hardware Setup...** button and select the USB-Blaster in the window that pops up, as shown in Figure 30.

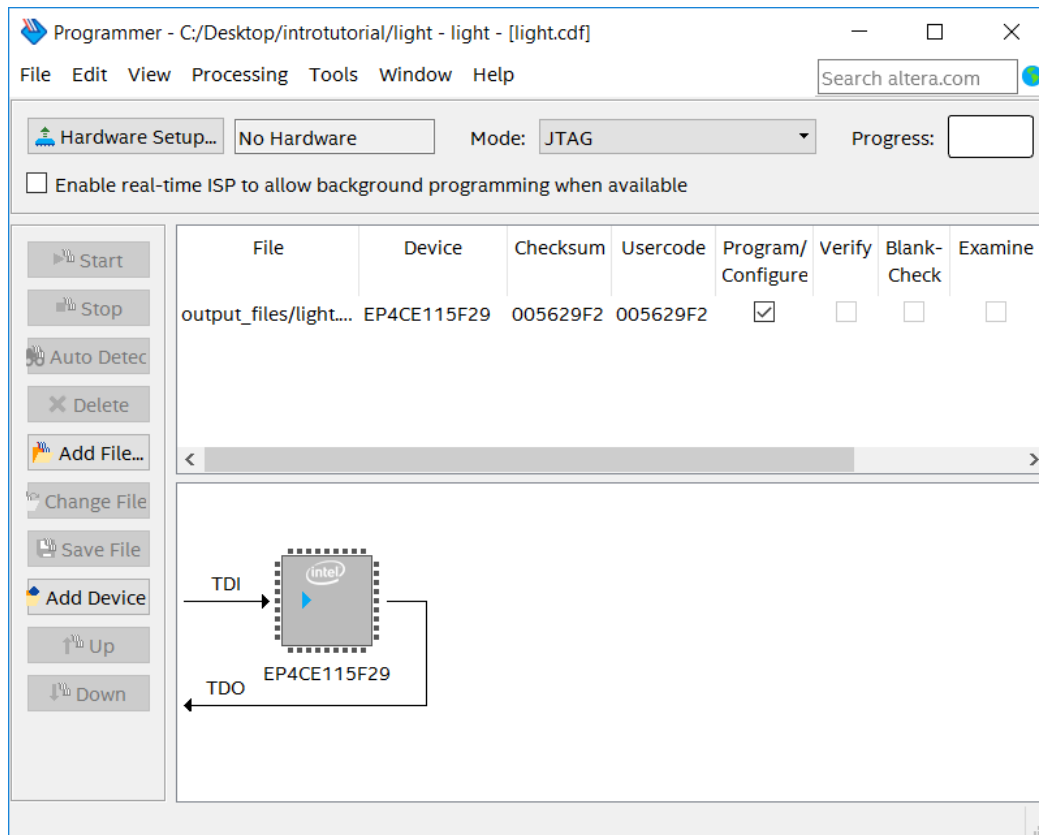


Figure 29. The Programmer window.

Observe that the configuration file *light.sof* is listed in the window in Figure 29. If the file is not already listed, then click **Add File** and select it. This is a binary file produced by the Compiler's Assembler module, which contains the data needed to configure the FPGA device. The extension *.sof* stands for SRAM Object File. Ensure the **Program/Configure** check box is ticked, as shown in Figure 29.

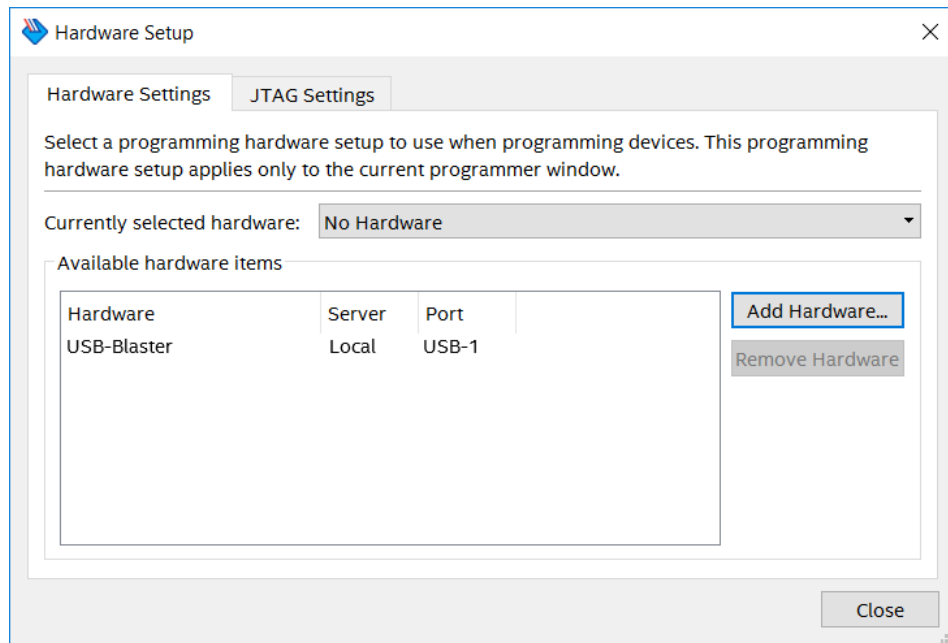


Figure 30. The Hardware Setup window.

Now, press **Start** in the window in Figure 29. An LED on the board will light up corresponding to the programming operation. If you see an error reported by Quartus Prime software indicating that programming failed, then check to ensure that the board is properly powered on.

8.2 JTAG* Programming for the DE0-Nano-SoC, DE1-SoC Board, DE10-Nano, and DE10-Standard

For the DE0-Nano-SoC, DE1-SoC Board, DE10-Nano, and DE10-Standard boards, the following steps should be used for programming. Select **Tools > Programmer** to reach the window in Figure 31 (if the SOCVHPS device is missing, it can be added through the **Add Device** menu under the *Soc Series V* family). Here it is necessary to specify the programming hardware and the mode that should be used. If not already chosen by default, select **JTAG** in the Mode box. Also, if *DE-SoC* is not chosen by default as the programming hardware, then press the **Hardware Setup...** button and select the *DE-SoC* in the window that pops up, as shown in Figure 32.

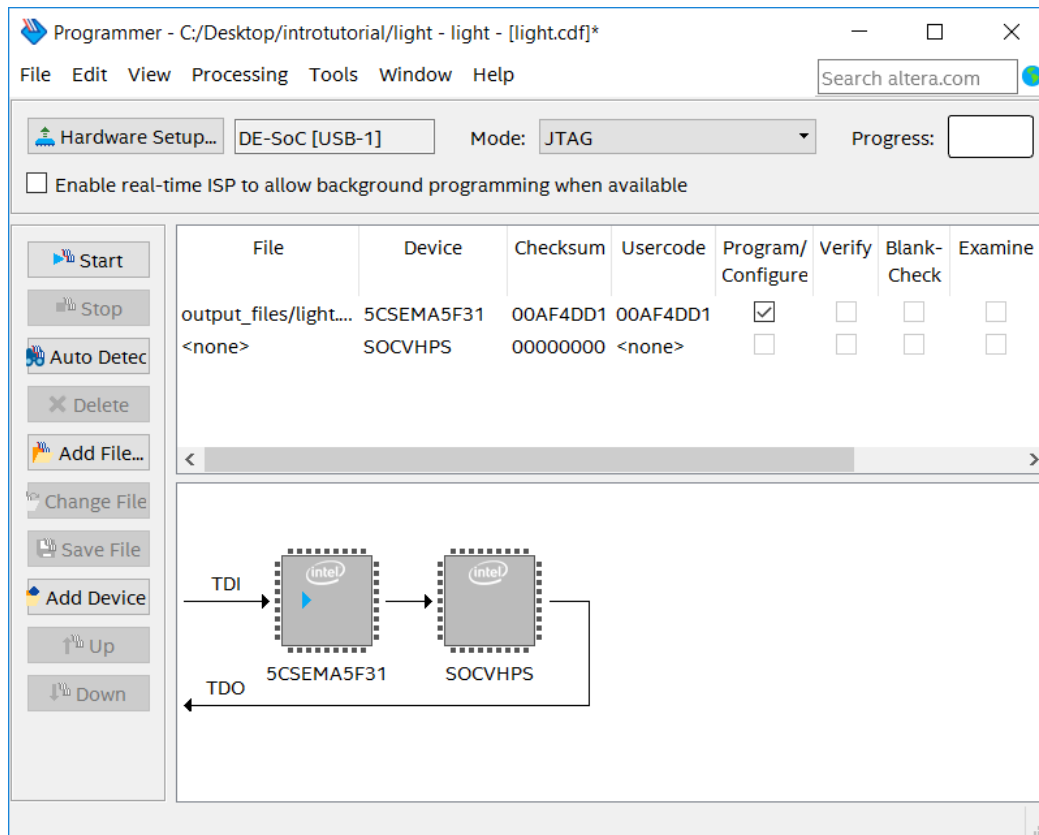


Figure 31. The Programmer window.

Observe that the configuration file *light.sof* in directory *output_files* is listed in the window in Figure 31. If the file is not already listed, then click **Add File** and select it. This is a binary file produced by the Compiler's Assembler module, which contains the data needed to configure the FPGA device. The extension *.sof* stands for SRAM Object File. Ensure the **Program/Configure** box is checked. This setting is used to select the FPGA in the Cyclone V SoC chip for programming. If the *SOCVHPS* device is not shown as in Figure 31, click **Add Device** > SoC Series V > *SOCVHPS* then click **OK**. Ensure that your device order is consistent with Figure 31 by clicking on a device and then clicking **Up** or **Down**.

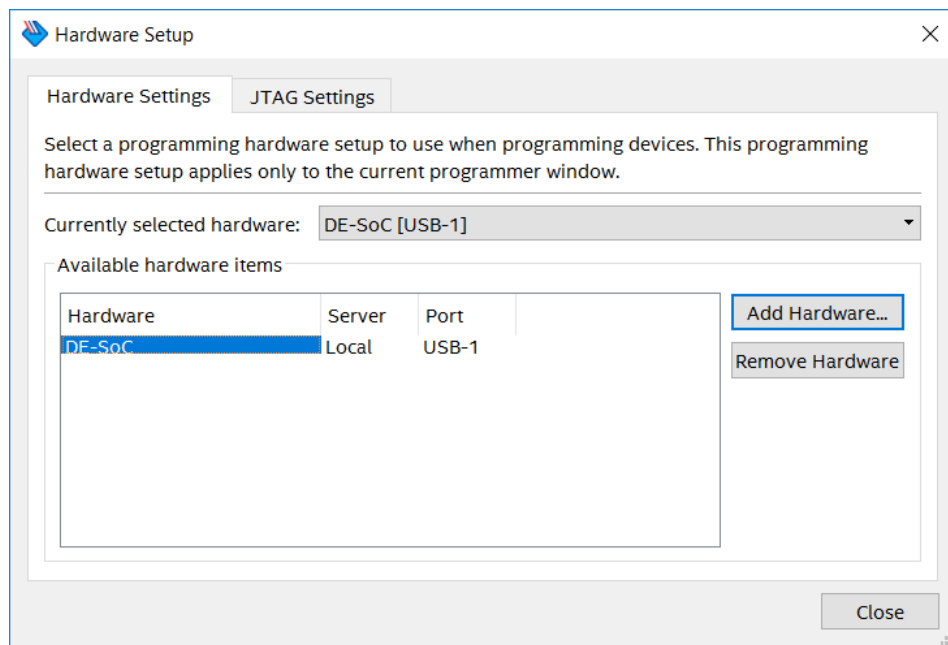


Figure 32. The Hardware Setup window.

Now, press **Start** in the Programmer. An LED on the board will light up while the FPGA device is being programmed. If you see an error reported by Quartus Prime software indicating that programming failed, then check to ensure that the board is properly powered on.

9 Simulating the Designed Circuit

Before implementing the designed circuit in the FPGA chip on the DE-series board, it is prudent to simulate it to ascertain its correctness. Quartus Prime's Simulation Waveform Editor tool can be used to simulate the behavior of a designed circuit. Before the circuit can be simulated, it is necessary to create the desired waveforms, called *test vectors*, to represent the input signals. It is also necessary to specify which outputs, as well as possible internal points in the circuit, the designer wishes to observe. The simulator applies the test vectors to a model of the implemented circuit and determines the expected response. We will use the Simulation Waveform Editor to draw the test vectors, as follows:

1. In the main Quartus Prime window, select **File > New > Verification/Debugging Files > University Program VWF** to open the Simulation Waveform Editor. Alternatively, select an existing VWF file using **File > Open** to reopen the Simulation Waveform Editor using that file.
2. The Simulation Waveform Editor window is depicted in Figure 33. Save the file under the name *light.vwf*, and then refresh the project settings by clicking **Simulation > Simulation Settings > Restore Defaults**. Set the desired simulation to run from 0 to 200 ns by selecting **Edit > Set End Time** and entering 200 ns in the

dialog box that pops up. Selecting **View > Fit in Window** displays the entire simulation range of 0 to 200 ns in the window, as shown in Figure 34. You may wish to resize the window to its maximum size.

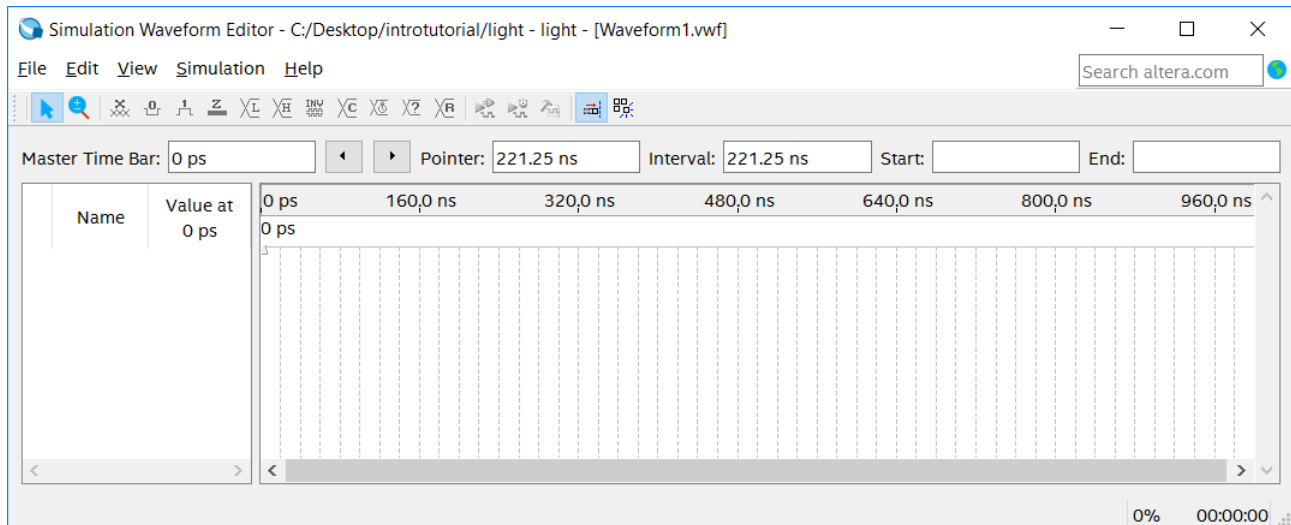


Figure 33. The Waveform Editor window.

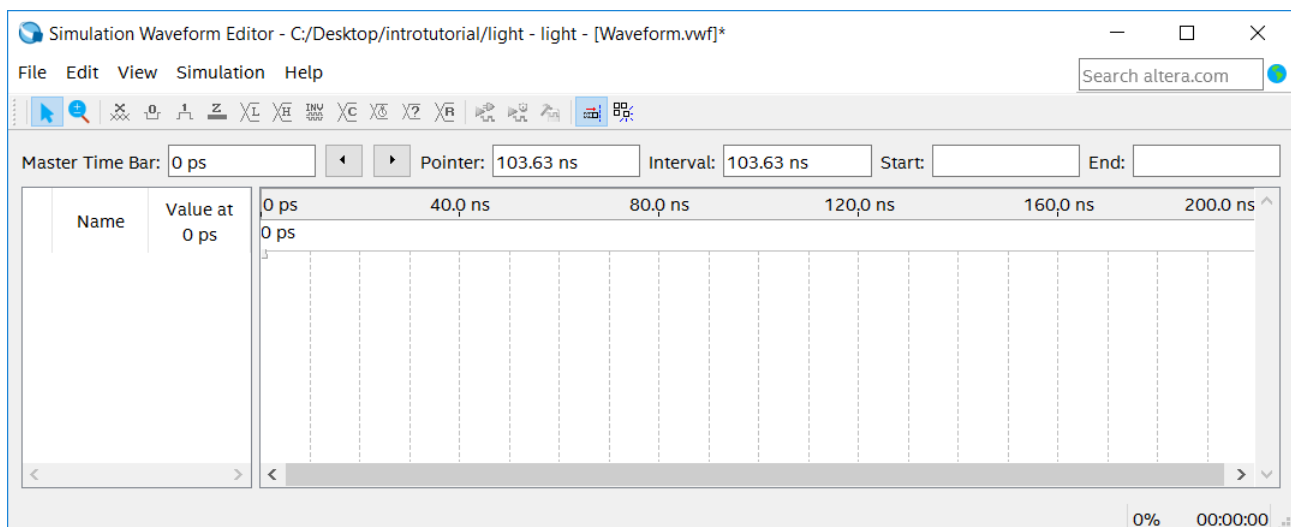


Figure 34. The augmented Waveform Editor window.

- Next, we want to include the input and output nodes of the circuit to be simulated. Click **Edit > Insert > Insert Node or Bus** to open the window in Figure 35. It is possible to type the name of a signal (pin) into the Name box, or use the Node Finder to search your project for the signals. Click on the button labeled **Node Finder** to open the window in Figure 36. The Node Finder utility has a filter used to indicate what type of

nodes are to be found. Since we are interested in input and output pins, set the filter to Pins: all. Click the List button to find the input and output nodes as indicated on the left side of the figure.

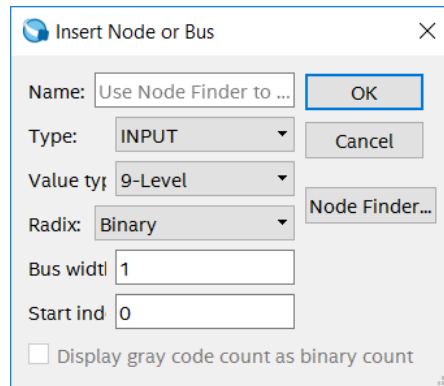


Figure 35. The Insert Node or Bus dialogue.

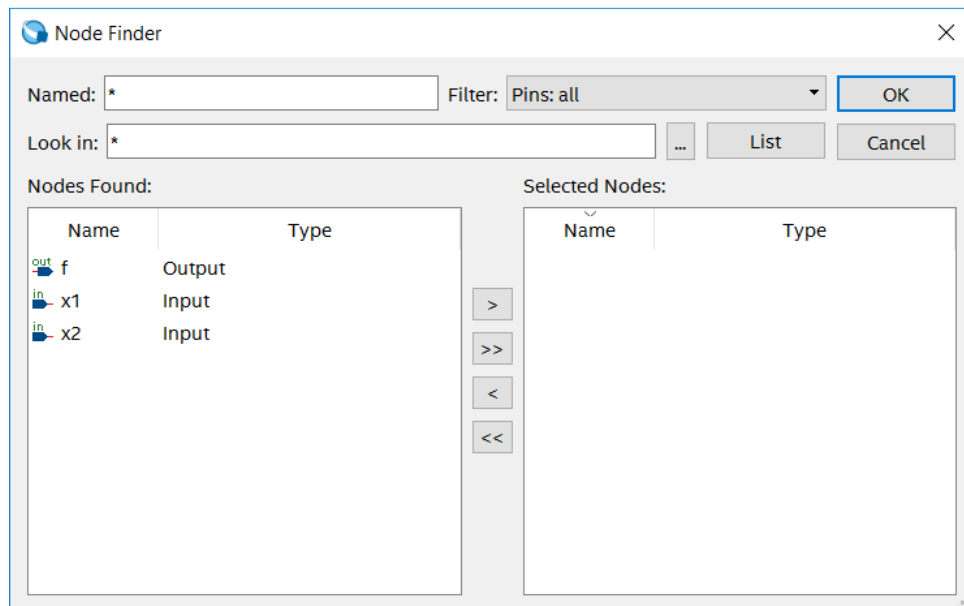


Figure 36. Selecting nodes to insert into the Waveform Editor.

Click on the *x1* signal in the Nodes Found box in Figure 36, and then click the > sign to add it to the Selected Nodes box on the right side of the figure. Do the same for *x2* and *f*. Click OK to close the Node Finder window, and then click OK in the window of Figure 35. This leaves a fully displayed Waveform Editor window, as shown in Figure 37. If you did not select the nodes in the same order as displayed in Figure 37, it is possible to rearrange them. To move a waveform up or down in the Waveform Editor window, click within the node's row (i.e. on its name, icon, or value) and drag it up or down in the Waveform Editor.

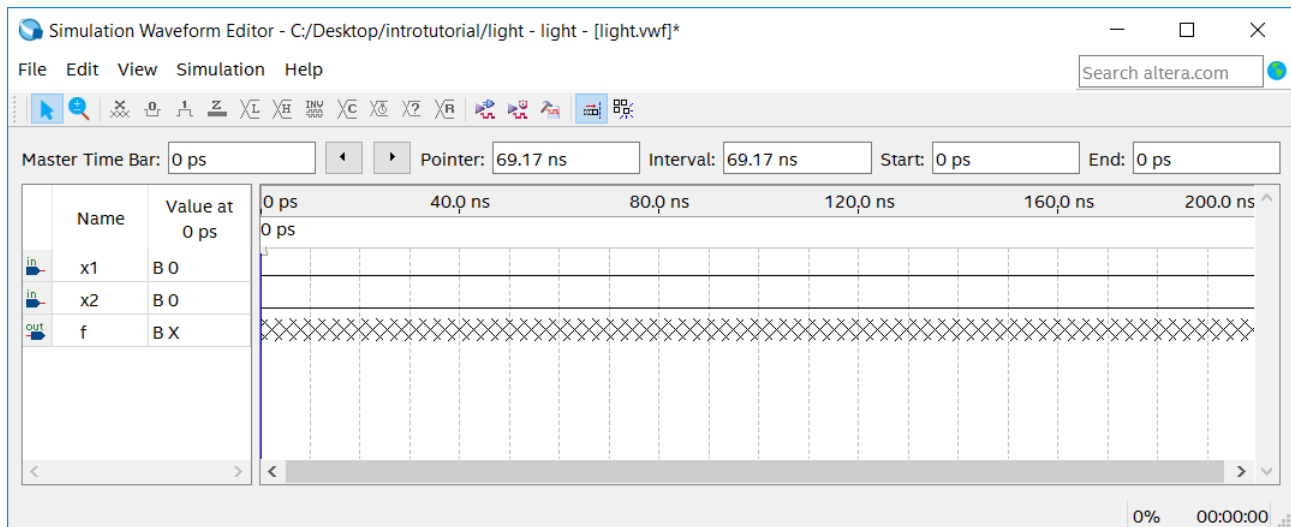



Figure 37. The nodes needed for simulation.

- We will now specify the logic values to be used for the input signals $x1$ and $x2$ during simulation. The logic values at the output f will be generated automatically by the simulator. To make it easy to draw the desired waveforms, the Waveform Editor displays (by default) vertical guidelines and provides a drawing feature that snaps on these lines (which can otherwise be invoked by choosing **Edit > Snap to Grid**). Observe also a solid vertical line, which can be moved by pointing to its top and dragging it horizontally. This reference line is used in analyzing the timing of a circuit; move it to the $time = 0$ position. The waveforms can be drawn using the Selection Tool, which is activated by selecting the icon  in the toolbar.

To simulate the behavior of a large circuit, it is necessary to apply a sufficient number of input valuations and observe the expected values of the outputs. In a large circuit the number of possible input valuations may be huge, so in practice we choose a relatively small (but representative) sample of these input valuations. However, for our tiny circuit we can simulate all four input valuations given in Figure 12. We will use four 50-ns time intervals to apply the four test vectors.

We can generate the desired input waveforms as follows. Click on the waveform for the $x1$ node. Once a waveform is selected, the editing commands in the Waveform Editor can be used to draw the desired waveforms. Commands are available for setting a selected signal to 0, 1, unknown (X), high impedance (Z), weak low (L), weak high (H), a count value (C), an arbitrary value, a random value (R), inverting its existing value (INV), or defining a clock waveform. Each command can be activated by using the **Edit > Value** command, or via the toolbar for the Waveform Editor. The Value menu can also be opened by right-clicking on a selected waveform.

Set $x1$ to 0 in the time interval 0 to 100 ns, which is probably already set by default. Next, set $x1$ to 1 in the time interval 100 to 200 ns. Do this by pressing the mouse at the start of the interval and dragging it to its end, which highlights the selected interval, and choosing the logic value 1 in the toolbar. Make $x2 = 1$ from 50 to 100 ns and also from 150 to 200 ns, which corresponds to the truth table in Figure 12. This should produce the image in Figure 38. Observe that the output f is displayed as having an unknown value at this time, which is indicated by a hashed pattern; its value will be determined during simulation. Save the file.

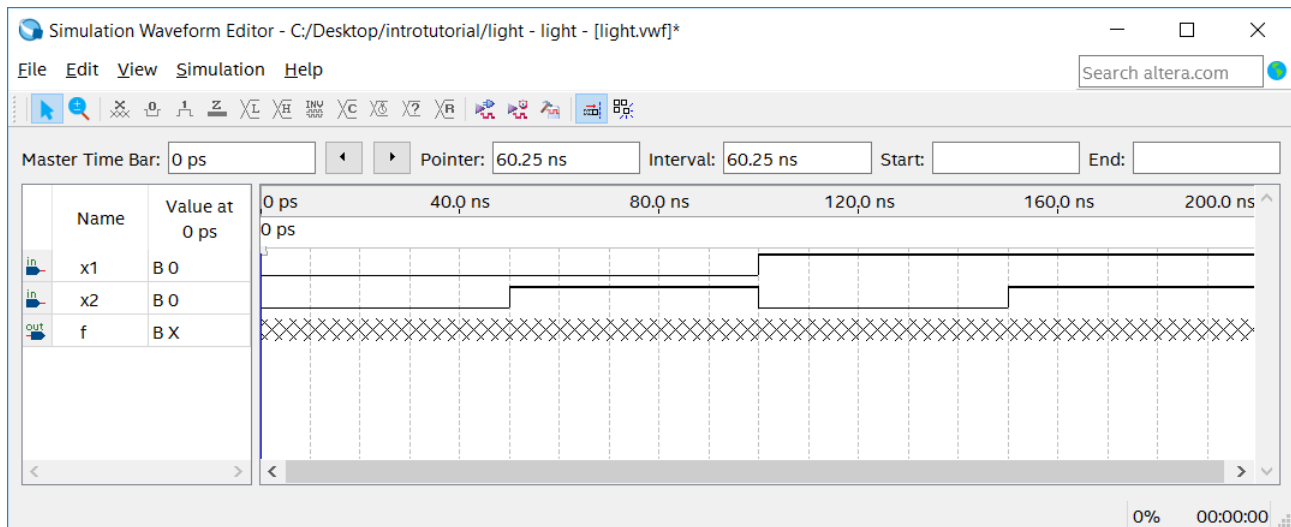




Figure 38. Setting of test values.

9.1 Performing the Simulation

A designed circuit can be simulated in two ways. The simplest way is to assume that logic elements and interconnection wires in the FPGA are perfect, thus causing no delay in propagation of signals through the circuit. This is called *functional simulation*. A more complex alternative is to take all propagation delays into account, which leads to *timing simulation*. Typically, functional simulation is used to verify the functional correctness of a circuit as it is being designed.

9.1.1 Functional Simulation

Before running a functional simulation it is necessary to run Analysis and Synthesis on your design by selecting the  icon in the main Quartus Prime window. Note that Analysis and Synthesis gets run as a part of the main compilation flow. If you compiled your design in Section 6, then it is not necessary to run Analysis and Synthesis again.

To perform the functional simulation, select **Simulation > Run Functional Simulation** or select the  icon in the Simulation Waveform Editor window. A pop-up window will show the progress of the simulation then automatically close when it is complete. At the end of the simulation, a second Waveform Editor window will open the results of the simulation as illustrated in Figure 39. Observe that the output *f* is as specified in the truth table of Figure 12.

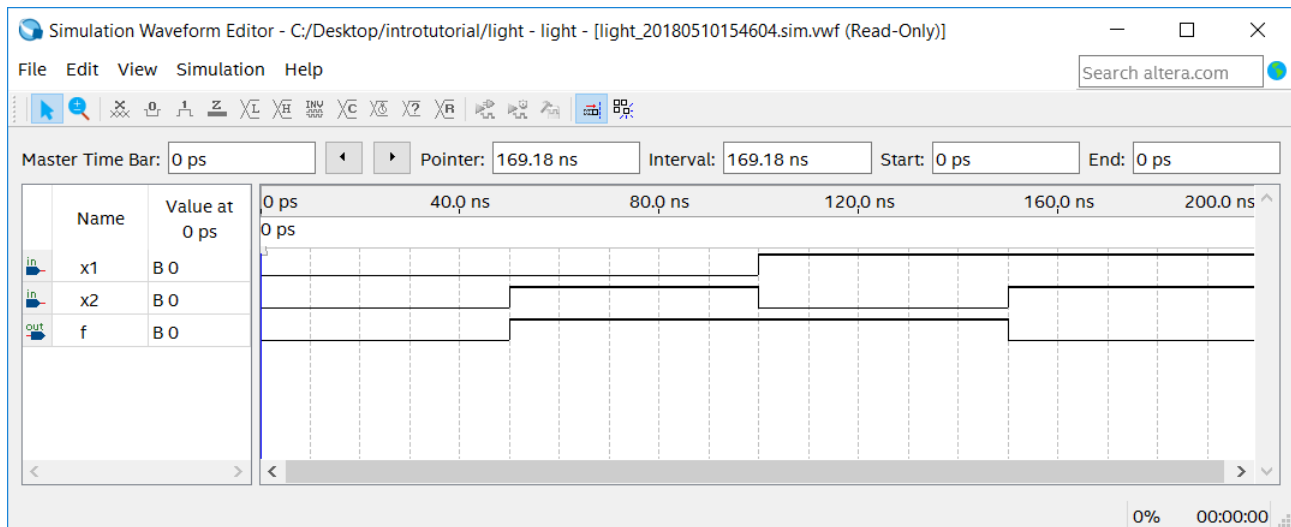




Figure 39. The result of functional simulation.

9.1.2 Timing Simulation

Having ascertained that the designed circuit is functionally correct, we should now perform the timing simulation to see how it will behave when it is actually implemented in the chosen FPGA device. Before running a timing simulation, it is necessary to compile your design by selecting the  icon in the main Quartus Prime window. Unlike functional simulations, timing simulations require the full compilation of your design, not just Analysis and Synthesis.

To perform the timing simulation, select **Simulation > Run Timing Simulation** or select the  icon in the Simulation Waveform Editor window. The simulation should produce the waveforms in Figure 40. Observe that there is a delay of about 5 ns in producing a change in the signal f from the time when the input signals, x_1 and x_2 , change their values. This delay is due to the propagation delays in the logic element and the wires in the FPGA device.

Note: timing simulations are only supported by Cyclone® IV and Stratix® IV FPGAs. If your DE-series board does not have a Cyclone IV or Stratix IV FPGA, the result of a timing simulation will be identical to the functional simulation.

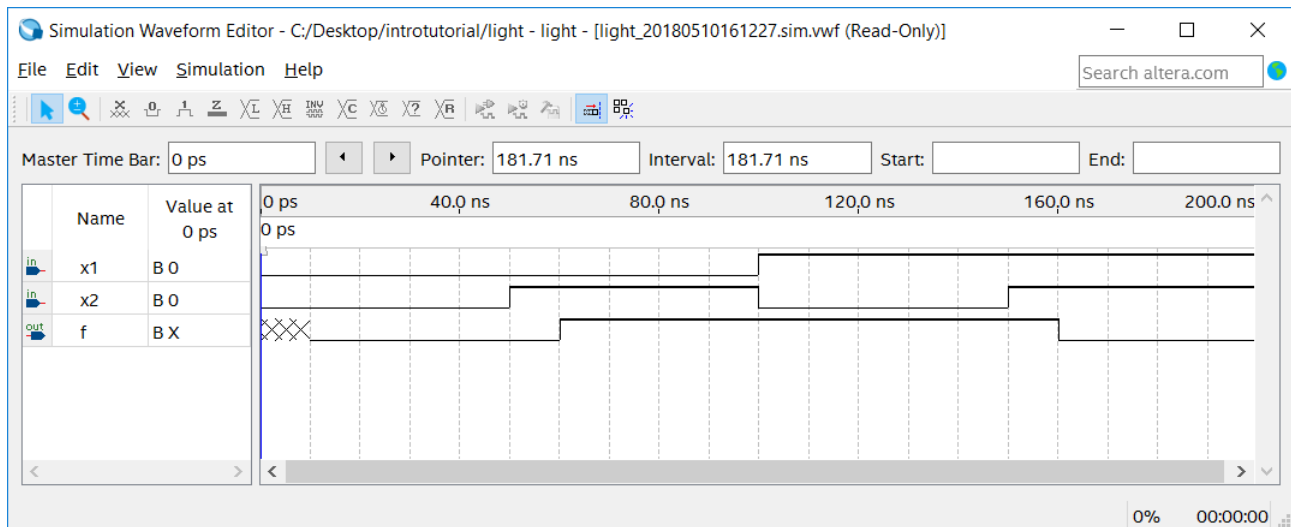


Figure 40. The result of timing simulation.

10 Testing the Designed Circuit

Having downloaded the configuration data into the FPGA device, you can now test the implemented circuit. Try all four valuations of the input variables x_1 and x_2 , by setting the corresponding states of the switches SW_1 and SW_0 . Verify that the circuit implements the truth table in Figure 12.

If you want to make changes in the designed circuit, first close the Programmer window. Then make the desired changes in the Verilog design file, compile the circuit, and program the board as explained above.

Copyright © Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Avalon, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.