Students' Guide to

Practicals on Systems-on-Chip Design

Isuru Nawinne
Faculty of Engineering - University of Peradeniya

Students' Guide to Practicals on Systems-on-Chip Design

By Isuru Nawinne

ISBN 978-624-92913-3-1

Downloadable ebook and supplementary material available at

https://cepdnaclk.github.io/soc-design

Publisher:

Dr. Isuru Nawinne,
Department of Computer Engineering, Faculty of Engineering,
University of Peradeniya,
Peradeniya 20400,
Sri Lanka.
isurunawinne@eng.pdn.ac.lk https://people.ce.pdn.ac.lk/staff/academic/isuru-nawinne/

Content

Preface

The rapid evolution of digital systems has brought hardware and software closer than ever. Modern computing platforms, from embedded controllers to cloud-scale accelerators, frequently integrate reconfigurable logic, high-performance processors, and diverse peripherals into compact, efficient, and intelligent systems. At the heart of this transformation lies the *system-on-chip* (SoC) paradigm and the practical ability to realize such systems using *field-programmable gate arrays* (FPGAs).

This book aims to introduce students to the fundamental concepts of SoC design and FPGA-based prototyping using industrial-grade tools. It bridges theory and practice, focusing not only on what SoC and FPGA technologies are, but how they are used in real engineering workflows. Beginning with the building blocks of reconfigurable logic, the book guides learners through the process of designing, analyzing, synthesizing, testing, and deploying complete digital systems on FPGA hardware.

The material grew out of a set of lectures and laboratory sessions delivered to undergraduate engineering students. Real systems, real tools, and real hardware experiences shape the content throughout. By working through the examples and exercises, students develop an integrated understanding of hardware architecture, digital logic, embedded systems, and tool-driven engineering processes.

Whether you are a student encountering SoC concepts for the first time, a practitioner seeking a concise introduction, or an educator looking for structured material for project-based learning, this book provides a practical and accessible starting point.

-

Isuru Nawinne
Senior Lecturer in Computer Engineering

Introduction

This book follows a hands-on, application-driven learning approach. Readers are encouraged not only to understand the principles of SoCs and FPGAs, but to *build*, *test*, *and experiment* with real designs on actual hardware. The learning experience is structured around several complementary methods:

Conceptual Foundations

Each chapter develops essential theoretical knowledge:

- digital logic and reconfigurable fabrics,
- SoC architectures and components,
- hardware–software interaction,
- system-level design flows.

These concepts prepare readers for the design tasks performed later in the book.

Guided Walkthrough

The first practical provides step-by-step guidance through the development environment and toolchain. Students learn how to:

- create FPGA projects,
- configure SoC components,
- perform pin planning,
- compile and program hardware,
- execute software on soft processors.

This walkthrough helps readers build confidence before advancing to more complex systems.

Progressive, Realistic Design Work

Later practicals introduce progressively advanced concepts that integrate multiple hardware and software modules. These design work mirror industry workflows and encourage students to think critically about:

- modular design,
- resource optimization,
- timing constraints,
- communication and interfacing,
- hardware/software co-design.

Experimentation and Inquiry

Readers are encouraged to modify parameters, alter architectures, and test alternative designs. Through iterative experimentation, students solidify their understanding and develop engineering intuition.

Reflection and Analysis

Each practical includes checkpoints and discussion prompts to help learners analyze their results, interpret performance behavior, and understand the implications of design decisions.

Tools Used

The examples in this book use a professional, industry-proven toolchain for FPGA-based SoC design. All tools are widely used in academic and commercial environments:

- **Intel (Altera) FPGAs** - Cyclone series FPGA devices (Terasic DE2 Development Kit)
- **Quartus Prime Web Edition** - for designing, analysis and synthesis
- **Qsys (Platform Designer)** - for building and integrating SoC components
- **Nios II Software Build Tools** - for building and running software on softprocessors

These tools support the complete development workflow, from writing embedded programs to deploying them on the FPGA.

Field Programmable Gate Arrays (FPGAs)

An FPGA is an integrated circuit designed to be configured by the user. In essence, it is a hardware device whose internal logic can be reprogrammed after manufacturing.

When we speak about programmable hardware, we must recognize that programmability exists at different levels. A modern computer processor is also programmable, but at a much higher level. You write software programs, compile them, and the processor executes those instructions. Although the hardware is fixed, the behavior of the system changes based on the software running on it.

Traditional integrated circuits, however, are not programmable. They are designed for a specific purpose, and once fabricated, that purpose is fixed. You design a logic circuit composed of gates, define its inputs and outputs, and specify the function it must perform. After that, the design moves through the typical stages of hardware production: design, synthesis, fabrication, packaging, and finally deployment.

During synthesis, the design described at the register-transfer level or gate level is transformed into a transistor-level design. In conventional integrated circuit design, the output of the synthesis process is a layout of transistors, typically implemented using lithography on a silicon wafer. You end up with a physical chip containing transistors and wiring that implement the logic exactly as designed.

This fabrication process is expensive. Producing a single custom chip can cost thousands of dollars, even before packaging. Once fabricated, the chip is packaged, for example, in a dual in-line package or any of several modern surface-mount formats. The package protects the chip and provides pins or pads that allow it to be mounted on a circuit board. Once fabricated and packaged, the chip performs exactly the function it was designed for, and nothing else.

FPGAs, however, follow a different approach. Instead of designing a circuit, synthesizing it to transistors, and sending it to a foundry, you begin with a pre-manufactured, pre-packaged FPGA device already mounted on a board. This device contains a large array of configurable hardware building blocks. You design your logic circuit in the usual way, but instead of synthesizing it down to transistors, you synthesize it to the primitive elements that exist inside the FPGA.

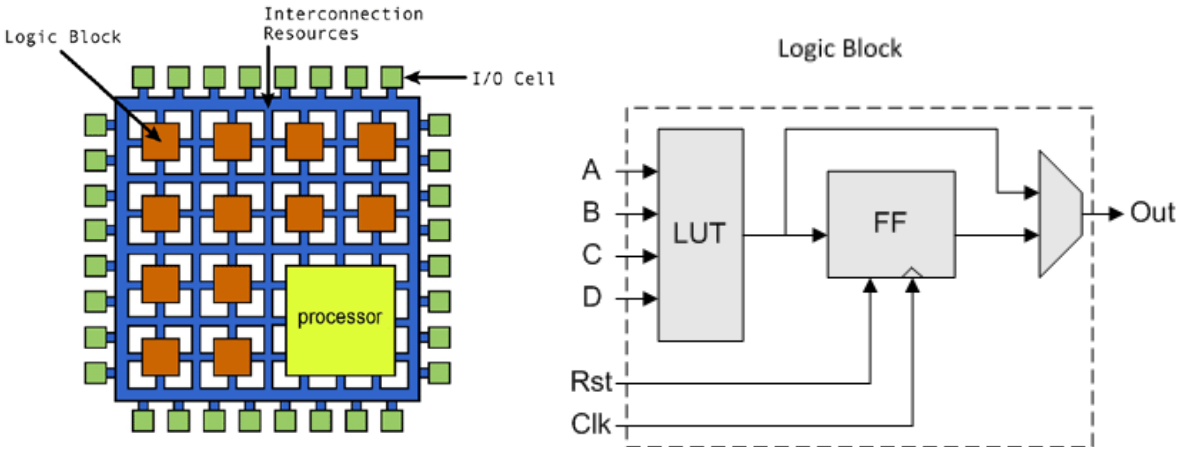
These primitive elements, often referred to as logic blocks or logic elements, are the basic computational units of the FPGA. The synthesis tool maps your gate-level design onto

these logic elements, producing a configuration file that, when loaded onto the FPGA, causes the device to behave as the logic circuit you designed. This configuration process is what we commonly refer to as “programming” the FPGA. While the term is borrowed from software development, the underlying concept is quite different: instead of executing software instructions, the FPGA’s hardware itself is being reconfigured.

The key advantage of this approach is flexibility. Once a logic circuit is implemented on the FPGA, it functions as intended. But if a different functionality is needed, the FPGA can be reconfigured with a different design using the very same physical device. In other words, the hardware can be rewritten. This makes FPGAs extremely valuable, not only for final products where hardware upgrades may be necessary, but also for prototyping, where designers need to test logic circuits at the hardware level before committing to fabrication.

This flexibility is also why FPGAs occupy a unique space between traditional integrated circuits and programmable processors. In terms of raw performance, a dedicated integrated circuit will generally outperform an FPGA, because fixed hardware can be optimized more aggressively. However, FPGAs offer considerably higher performance than general-purpose processors for many digital logic tasks, because the logic can run in true parallel hardware rather than through sequential software instructions.

Looking inside an FPGA reveals a structure known as the **reconfigurable logic fabric**. This fabric consists of a large mesh of logic blocks interconnected through a programmable routing network. The entire structure resembles a woven fabric of computational elements. At the edges of this fabric are input/output cells that provide the interface between the FPGA and external circuitry.



Each logic block can be configured to implement different logic functions. By combining many such blocks and customizing their interconnections, one can construct very sophisticated circuits, including complete computer processors. A processor designed, synthesized, and mapped onto these logic blocks can then execute software programs just like a processor built as a traditional integrated circuit.

The name field-programmable comes from the fact that these chips can be configured even after deployment, in the field. Unlike fixed-function chips, whose behavior is permanently set at fabrication, an FPGA can be reprogrammed at any time to implement new logic or updated functionality. This capability enables hardware upgrades in deployed products just as easily as software updates, and it allows rapid iteration in research and development.

The configurability of FPGA logic blocks allows them to host a wide range of digital circuits from very simple ones to highly complex systems. Just as a processor can run many different software programs, an FPGA can be configured to implement many different logic circuits by arranging and combining its logic blocks in various ways.

In the early days of programmable logic hardware, these logic blocks were relatively simple. They were often implemented as basic gates such as NAND or XOR. Modern FPGAs, however, contain far more sophisticated units. These are commonly referred to as logic blocks or logic cells, and one of their key components is the lookup table (LUT). A LUT can be thought of as a small truth table: it accepts several inputs and produces an output based on stored logic values. Different FPGA manufacturers offer LUTs with varying numbers of inputs, but the concept is the same.

Along with the LUT, each logic block typically includes a flip-flop to hold state, as well as reset and clock signals to support synchronous circuit behavior. The LUT is what makes the FPGA configurable. By programming the contents of the LUT, one can implement different truth tables and therefore different logic functions. If a single LUT is insufficient for a given design, multiple logic blocks can be combined to implement larger, more complex circuits. These blocks are then interconnected appropriately using the FPGA's routing network to realize the desired design.

This is the fundamental background behind how FPGAs operate and why they are so versatile. They are widely used for hardware upgrades in deployed products, where the underlying logic can be updated long after installation. They are also invaluable for prototyping, enabling designers to test logic circuits at hardware level early in the development process. In addition, FPGAs are used in high-performance applications.

Although they allow programmability like general-purpose processors, FPGAs operate much closer to the hardware level and therefore avoid many of the performance costs associated with software layers. As a result, FPGAs can deliver higher performance than a typical processor for certain classes of applications.

A general-purpose computer system includes a CPU, memory, input/output interfaces, and buses. Above the hardware lies firmware or device drivers, which allow the operating system to communicate with the hardware securely. The operating system provides resource management and supports multiple programs running concurrently. Application software sits at the top, implementing the algorithms required by the user. The performance of any algorithm depends on all these layers: hardware, firmware, the operating system, and the application software itself. The processor may spend only a fraction of its time executing the specific application code, creating performance bottlenecks.

With an FPGA, however, one can bypass these layers entirely. Instead of relying on generic hardware, the designer can take an application's algorithm and build an application-specific logic circuit tailored to that exact computation. The FPGA fabric is then configured to implement this logic directly. Although this hardware is not at the transistor level like a custom integrated circuit, the resulting performance is still significantly higher than what a general-purpose system can provide. This is one major way FPGAs enhance performance.

The second performance advantage comes from parallelism. While the illustration might show a small mesh of logic blocks, real FPGAs contain thousands or even millions of these elements, enabling very large designs on a single device. In the hardware community, a reusable circuit is often referred to as an intellectual property (IP) block. If an application can be parallelized, multiple instances of an IP block can be placed on the FPGA and operated simultaneously. This resembles having many processor cores, but at a much finer granularity, hundreds of small hardware circuits operating in parallel on distributed data.

Just as GPUs use a SIMD (single-instruction multiple-data) model for parallel processing, an FPGA can host multiple copies of identical circuits or heterogeneous blocks performing different tasks, all running concurrently. Multiple FPGAs can be connected together to create massive parallel hardware systems capable of hosting very large designs.

One commercial example of FPGA-accelerated performance is high-frequency trading in financial markets. In stock markets and foreign exchange markets, trading decisions must be made in extremely short time frames, often in fractions of a millisecond. Human

traders cannot react at such speeds, but machines can. FPGAs are used to analyze market data at very high rates and make rapid buy or sell decisions. The FPGA receives real-time market data through fast network connections and performs risk analysis and decision-making logic at hardware speed. A traditional programmable computer then manages the broader system, while the FPGA handles the critical high-speed computations that provide a competitive advantage. Many firms employ such FPGA-based systems to achieve significant financial gains.

This example illustrates how FPGAs are integrated into modern high-performance applications: by combining their reconfigurable hardware capabilities with traditional processors, systems can achieve both flexibility and extreme speed. FPGAs therefore play a central role in environments where rapid decision-making and parallel data processing are essential.

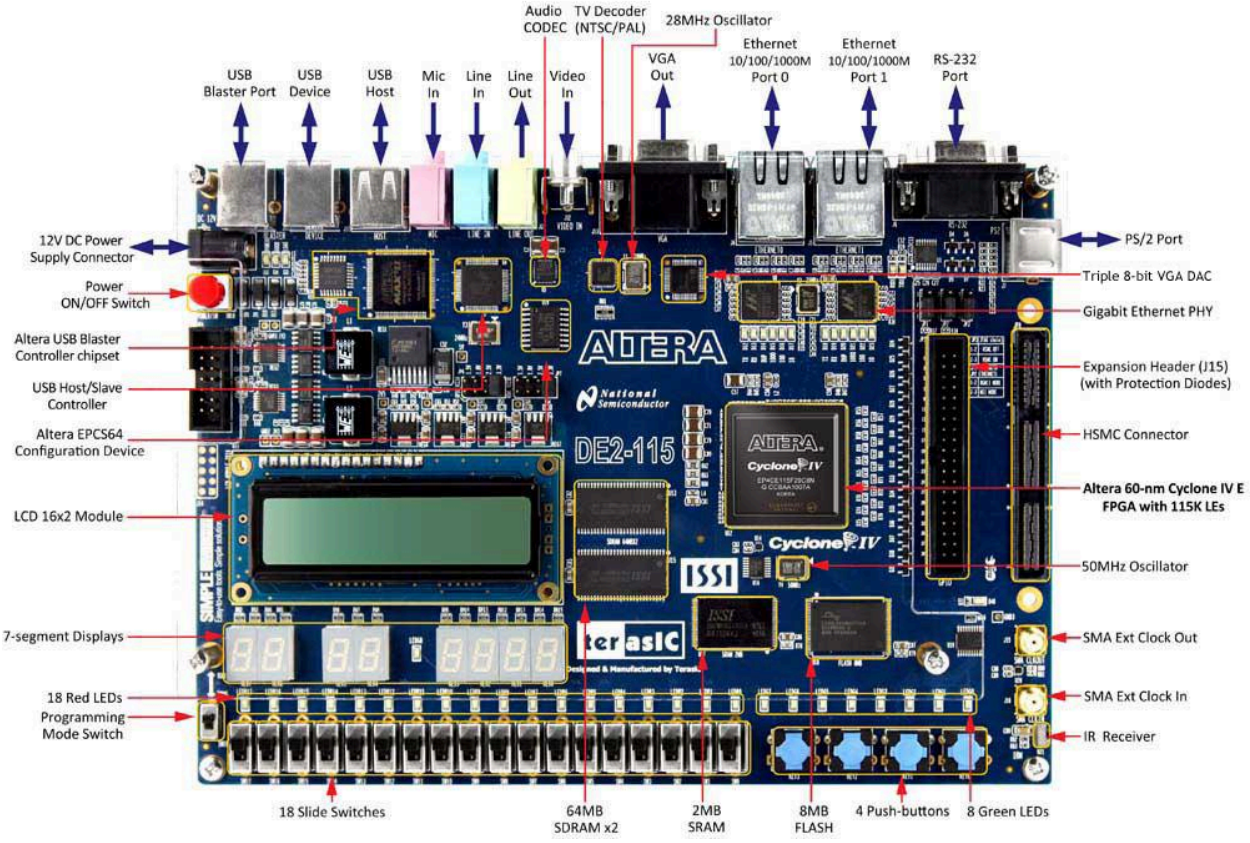
The FPGA typically works alongside a traditional processor, with the FPGA handling the high-performance portions of the workload while the processor manages overall coordination. This hybrid approach is used in many real-world systems. For example, major search engines such as Microsoft Bing and even Google use FPGAs to accelerate large-scale data-processing tasks like page ranking, where vast amounts of information must be analyzed very quickly.

Cloud and distributed services such as Microsoft Azure also make extensive use of FPGAs. Large data centers must handle enormous numbers of requests from around the world, and FPGAs provide highly parallel, high-throughput computation that complements conventional servers. More recently, hybrid computing has emerged, where a single chip integrates both CPU cores and FPGA fabric. Part of the chip functions as a processor, while another part serves as a reconfigurable FPGA region. These hybrid processors are now commercially available and are targeted primarily at cloud and data-center workloads, where both flexibility and performance are essential.

The increasing interest in FPGAs is reflected in large industry shifts. Intel's acquisition of Altera several years ago, valued at approximately 16.7 billion USD, was a major strategic move that positioned Intel to incorporate FPGA technology directly into its server-grade and data-center products. Although FPGAs are not a new technology (they have existed since the 1980s) their relevance has grown again as traditional CPU performance gains have begun to encounter physical and architectural limits. To overcome these bottlenecks and continue improving performance for large-scale applications, FPGAs have re-emerged as critical components in modern computing.

The FPGA Evaluation Board

For practical work, we use a Cyclone IV FPGA on an evaluation board designed for education and experimentation. This board includes a range of peripheral components hard-wired to the FPGA, including USB interfaces, Ethernet, buttons, switches, LEDs, and general-purpose I/O. Each of these components is mapped to specific FPGA pins, and the manual provides detailed information about these connections. By routing your FPGA design appropriately, you can use these external devices as part of your experiments.



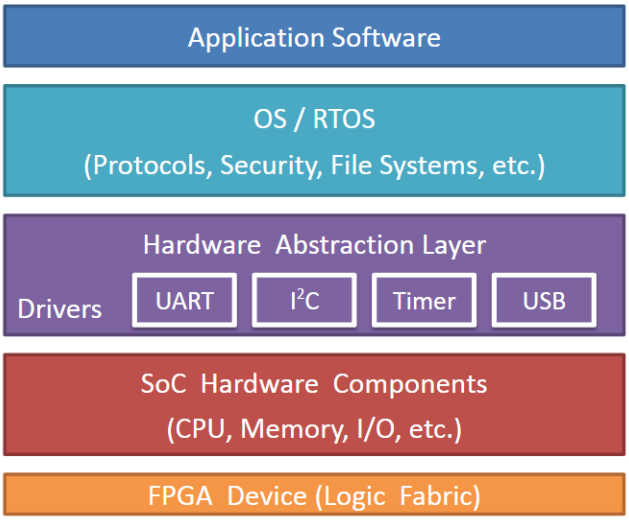
Systems-on-Chip (SoCs)

A system on chip (SoC) integrates all major components of a computing system onto a single chip: processors, memory, timers, communication interfaces, and application-specific hardware. You have previously worked with microcontrollers, which also integrate multiple components into a single package. However, microcontrollers are generally designed for lower-level sensing and control tasks and typically contain simpler processors. SoCs, by contrast, are aimed at more computation-intensive applications and often include high-performance CPUs, GPUs, memory controllers, and specialized peripherals.

The advantage of placing all components on the same chip is that communication between them becomes fast, efficient, and low-power. If these elements were spread across multiple chips on a circuit board, communication would require more time and energy, and the system as a whole would be larger and more costly. By integrating everything, SoCs achieve higher performance, lower power consumption, smaller physical size, and reduced overall cost.

Combining SoC Concepts with FPGA Technology

When SoC concepts are combined with FPGA technology, the FPGA provides the reconfigurable logic fabric, while soft components such as a processor, memory blocks, timers, and communication units are implemented within that fabric. The processor can run software programs, and the rest of the system benefits from FPGA configurability. This results in a flexible SoC that can be modified, upgraded, prototyped, or adapted quickly.



In this combined system, the overall software stack resembles traditional computing, but with important differences. At the top are the application programs. Beneath them lies the operating system, which may be a lightweight embedded OS, a real-time operating system, or even a version of Linux, depending on the system's requirements. Below the OS is the hardware abstraction layer (HAL), which consists of device drivers and low-level support code. In FPGA-based SoCs, this layer must be generated specifically for the hardware configuration you create, because the hardware itself is designed and customized by you.

At the lowest level are the hardware components of the SoC—CPU, memory, caches, analog-to-digital converters, counters, and other peripherals—implemented using the FPGA's logic fabric. The logic fabric is configured to host the SoC, the HAL interfaces with that hardware, the operating system provides services above the HAL, and application software runs at the top. This forms a complete FPGA-based SoC environment.

Design Process for FPGA-Based SoCs

Designing a system on chip for an FPGA follows several key steps:

1. **SoC Design**
   You first design the components of the SoC—CPU, memory, peripherals—using hardware description languages such as Verilog or VHDL, or using block diagrams, schematics, or state machines.

2. **Analysis and Elaboration**
   The design is checked for syntax and semantic errors. The compiler analyzes the structure of the system to ensure correctness.

3. **Pin Planning**
   Inputs and outputs of the SoC are mapped to physical FPGA pins. These pins provide the electrical connections from the FPGA chip to external devices on the evaluation board.

4. **Technology Mapping, Optimization, and Routing**
   The compiler maps your gate-level SoC design onto the FPGA's configurable logic blocks. It then optimizes resource usage and connects the logic blocks through the routing network to form the complete circuit.

5. **Assembler Generation**
   If the SoC includes a soft processor, the toolchain generates an assembler and other support tools. Since the CPU architecture itself can be customized, the assembler must be built to match the specific configuration of your processor.

6. **Timing Analysis**
   The system is evaluated to ensure that all timing constraints are met, including the maximum achievable clock frequency.

7. **FPGA Programming**
   Once the design is complete, the configuration bitstream is downloaded to the FPGA. The FPGA logic fabric then behaves as the designed hardware system.

Software Development for the SoC

From the software perspective, additional steps are required:

- **Generate HAL and Device Drivers**
  The hardware abstraction layer is produced based on the specific set of peripherals included in the SoC.

- **Define Memory Layout**
  Developers must specify how memory is organized, including address spaces and permissions, especially if multiple CPUs or applications are involved.

- **Generate Compiler Support**
  Tools for compiling application software are configured to match the SoC architecture.

- **Select and Integrate an Operating System (Optional)**
  Embedded operating systems such as MicroC/OS-II or embedded Linux may be used if the application requires more advanced features.

- **Develop and Deploy Application Software**
  Applications are written in C or assembly and loaded onto the SoC executing on the FPGA. These programs use the HAL to interact with hardware.

Laboratory Work Overview

In the practical series which unfolds in the upcoming sections, we begin by building and testing a simple SoC on an FPGA. The first practical exercise is a guided walkthrough to become familiar with the development tools., followed by an exercise which involves implementing a more complex SoC capable of performing JPEG encoding. The second practical dives into application-specific optimizations in SoCs, using custom instructions. In the third practical, we tackle multiprocessor systems-on-chip (MPSoCs) and interprocessor communication. The fourth and final practical deals with combining all the above concepts into a real world application, an optimized MPSoC for efficient JPEG encoding.

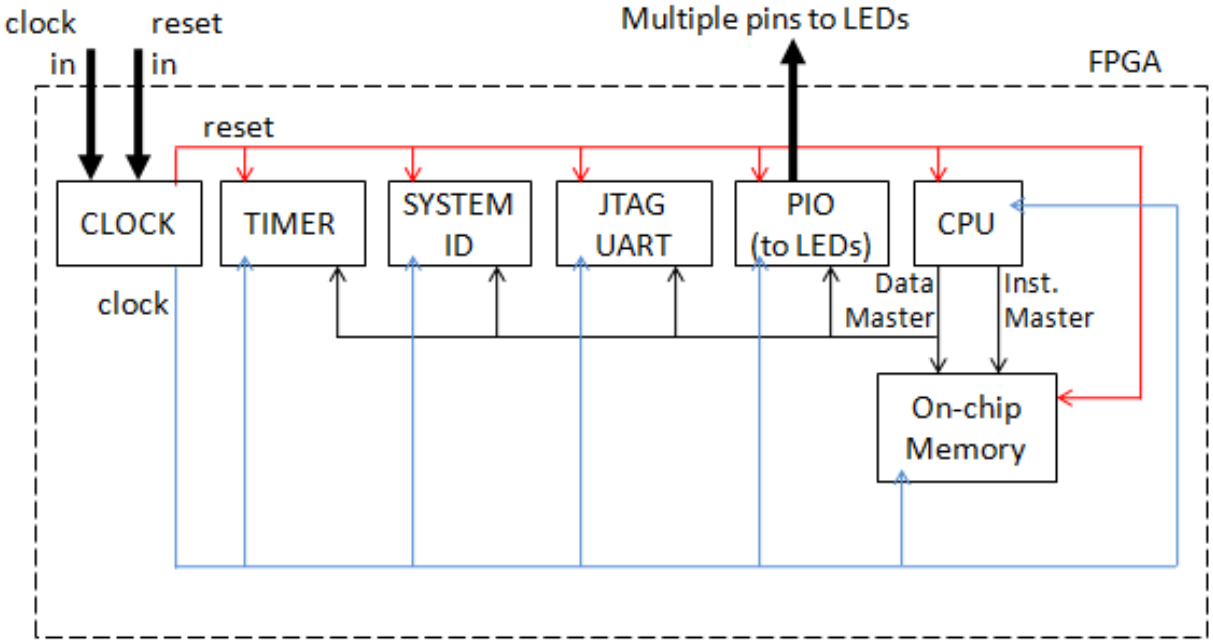
Practical 1

Getting Started with SoC Design

In this practical, you will use FPGA design tools to create your first System-on-Chip. The practical consists of two parts. In Part 1, you will learn how to create hardware and software for a simple SoC to control an LED counter. In Part 2 of the practical, you will build a SoC for JPEG image compression. These exercises are aimed at helping you get familiar with designing and synthesizing System-on-Chip hardware, co-design and development of embedded hardware and software, and get hands-on experience with FPGA-based design tools.

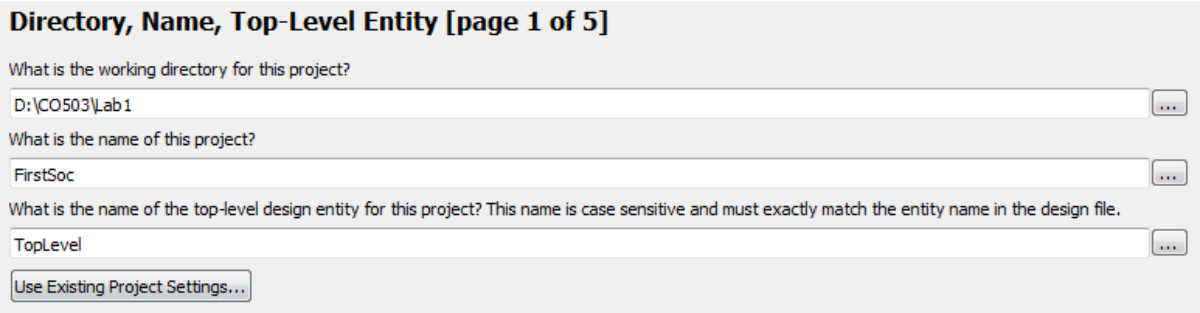
Part 1: First SoC – LED Counter

Your task is to create a simple SoC which can display an increasing count on a set of LEDs. An overview of your hardware system is shown in the diagrams below.

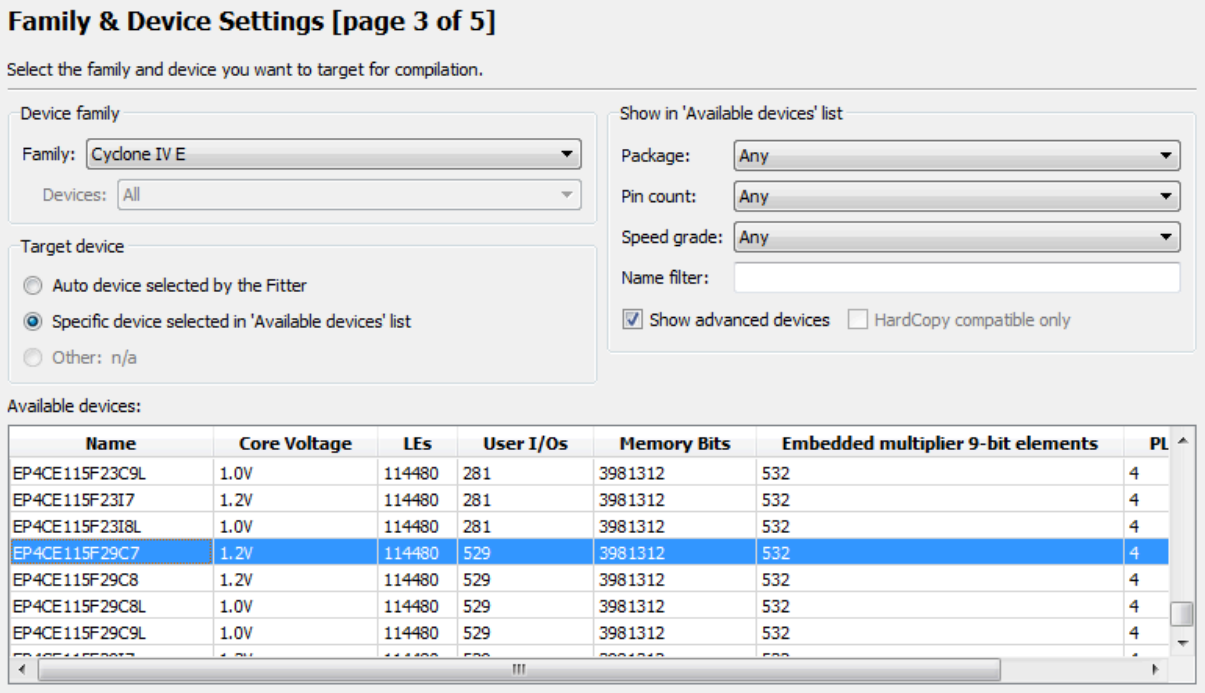


The following steps will walk you through building your first SoC on FPGA.

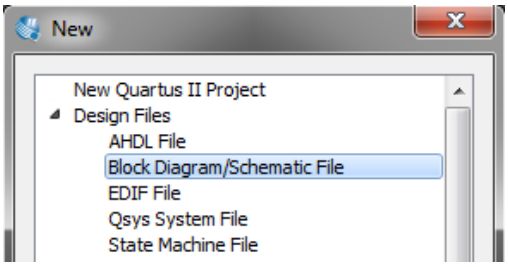
1. Launch *"Quartus II 12.1 (32.bit)"* software. Click on *"New Project Wizard"*. At the Introduction dialog, select *"Next"*. At page 1, name your project as *FirstSoC* and create a working directory for the project as shown below. Name the top-level entity as *TopLevel*. **Do not use spaces** in the directory or file names.



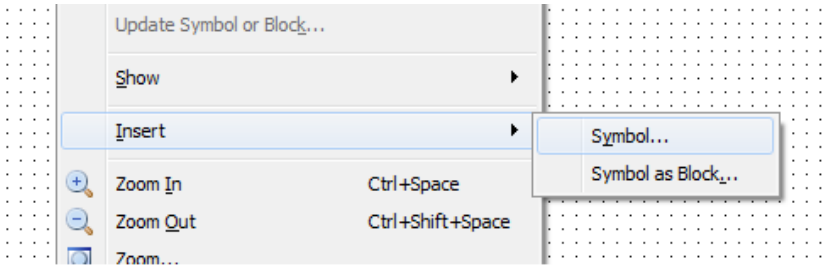
Skip to page 3. Select the device family *"Cyclone IV E"* and pick the device *"EP4CE115F29C7N"*.



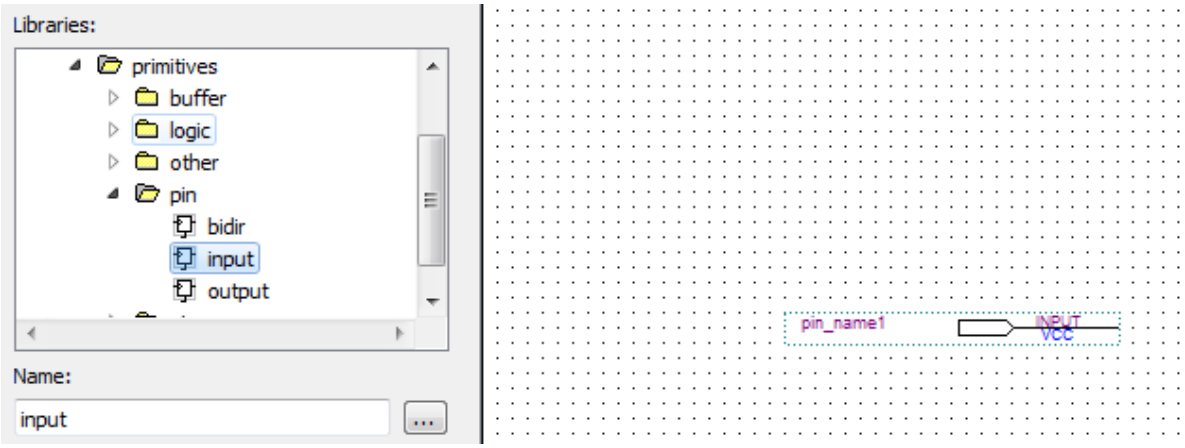
2. We're about to create the top-level design file for this project. Design files can be made in one of many ways like VHDL, Verilog, schematic, state machine, etc.. We will use a schematic (or block diagram). Create a new Block Diagram File (BDF) from the File menu (*File->New->BDF*).



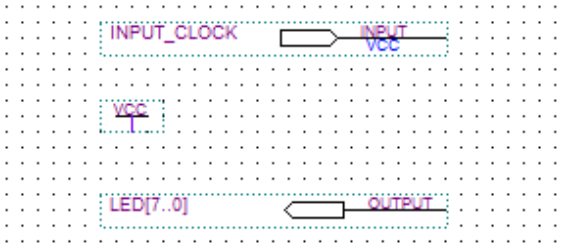
Next, we should add some I/O pins to help connect our SoC to components outside the FPGA (such as the LEDs). Right click on the dotted area in the BDF and select *Insert->Symbol*.



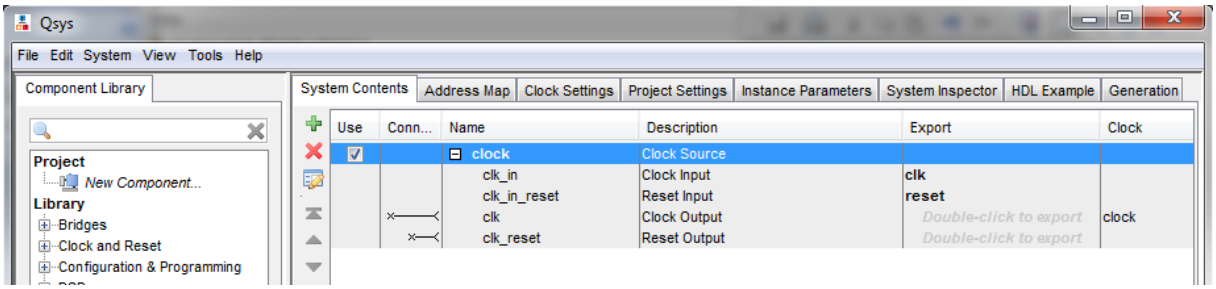
Under *Libraries->primitives->pin*, find the input pin symbol and click OK. Left-click to place the symbol on the BDF. Right click on the newly added *pin* symbol to open the Properties dialog. Rename the pin as *INPUT_CLOCK*.



Next, add a *Vcc* symbol (*Libraries->primitives->other*). Add an output pin (*Libraries->primitives->pin*) and name it *LED[7..0]*. Note that this last symbol now represents a set of eight individual pins.



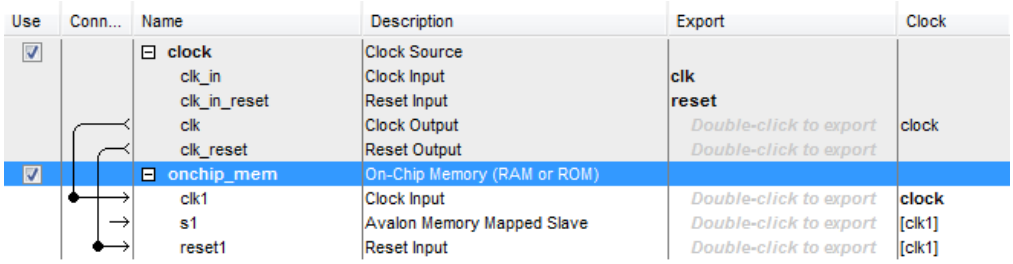
3. Now we're going to create the main component of our project, the System-on-Chip. For this, we will use the Qsys tool (*Tools->Qsys*). In the starting window of Qsys, you will see a Clock Source component called *"clk_0"*. Right click on this and rename it as *clock*.



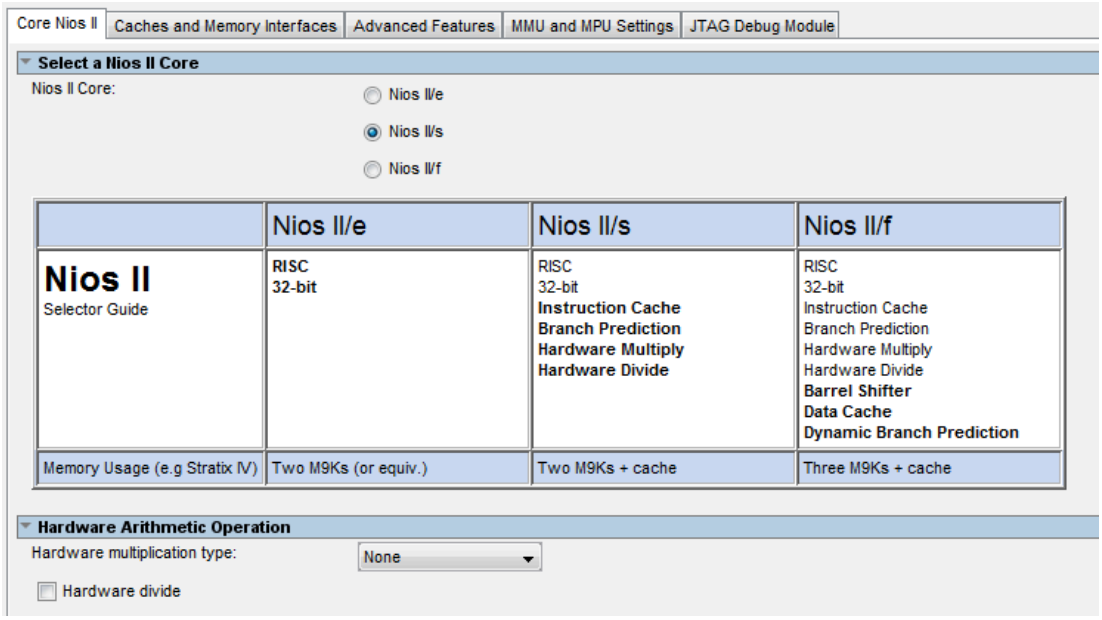
We need a memory for our SoC. We can either use standard memory controllers (like DDR3) to interface external SDRAM chips on the board, or implement memories on the FPGA itself (On-Chip Memory). Our system needs only a small amount of memory, so we will use an on-chip memory component.

Select the On-Chip Memory from the component library (*Library->Memories->On-Chip*), and click *"Add"*. Set the total memory size as *256KB (262144B)*, and click *"Finish"*. Rename the newly added memory component as *onchip_mem*. You will see some errors displayed in the *Messages* section at the bottom, don't worry, ignore them for the moment.

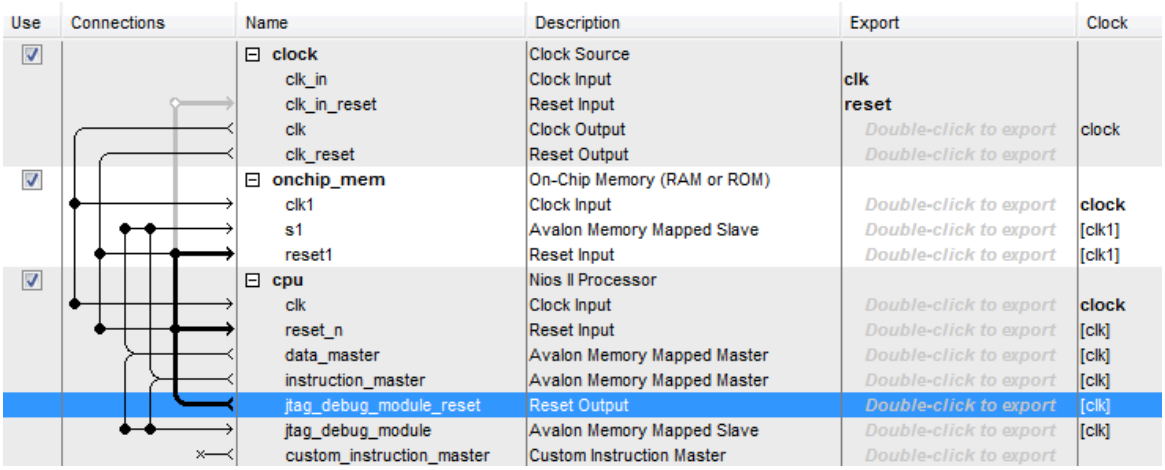
Next we should supply the *clock* and *reset* input signals to the memory. Connect both signals by clicking on the empty circles in front of *clk1* and *reset1* inputs of the memory component. Clicking will fill the circle (in black colour) to indicate the connection between the *clock* component and *onchip_mem* component has been made.



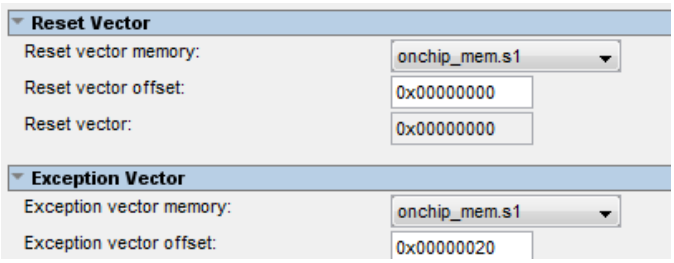
As the CPU in our SoC, we are going to use a *Nios II* RISC processor (*Library->Embedded Processors*). When you click *"Add"*, you will be taken to the processor configuration dialog shown below. The *Nios II* processor has three variants (*e*, *s* and *f*), study the differences between them. Select *Nios II/s* variant, set the *Hardware multiplication type* to *"None" and* click *"Finish"*.



Rename the new processor as *cpu*. Supply the clock and reset signals from the *clock* component to *cpu*. Connect both *data_master* and *instruction_master* ports of *cpu* to the *s1* port of *onchip_mem*. Make sure the *instruction_master* port of *cpu* is connected to the *jtag_debug_module* port. Then supply the *jtag_debug_module_reset* signal from *cpu* to *reset* inputs of both *onchip_mem* and *cpu*.



Now our processor and memory are coupled. We still need to tell the *cpu* where exactly in the memory it needs to look for program code, when starting execution (reset) or when interrupts/exceptions occur. To do this, double click on *cpu* to bring up the configuration dialog again. Select *"onchip_mem.s1"* for both *Reset vector memory* and *Exception vector memory*.



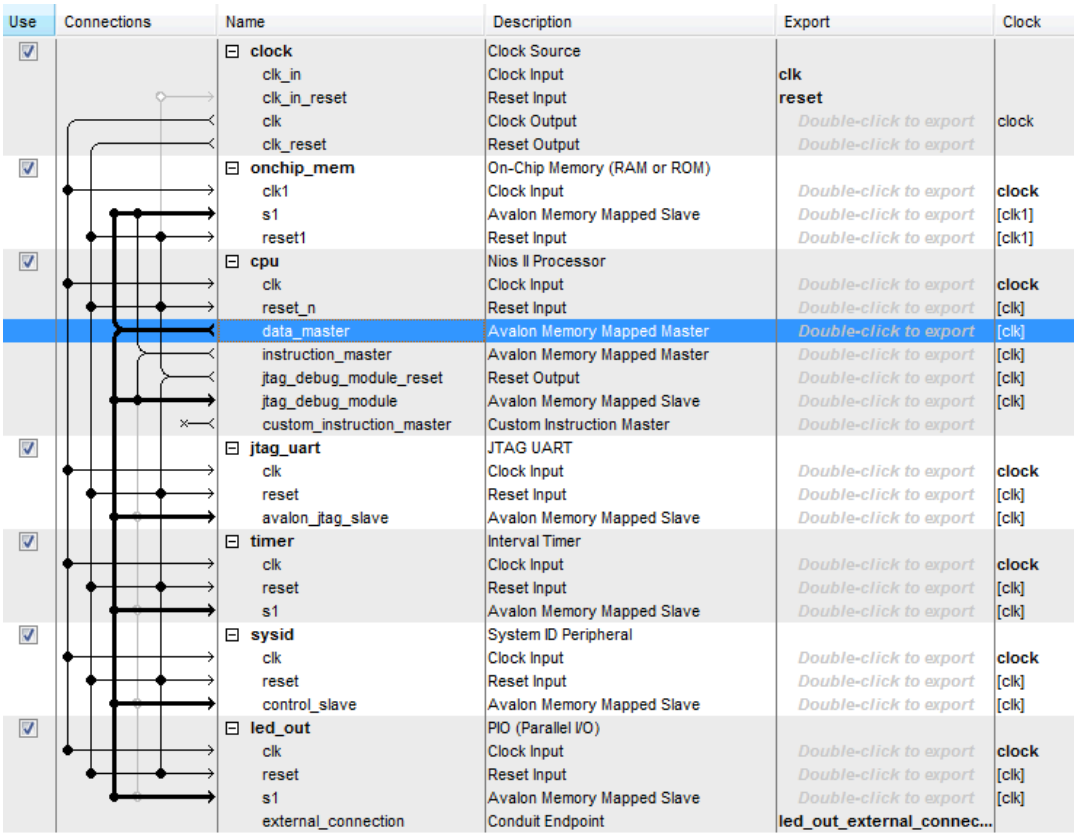
The JTAG UART provides a convenient way to communicate with our CPU through the USB-Blaster cable. Add a *JTAG UART* from the component library (*Library->Interface Protocols->Serial)*. Click *"Finish"* to keep the default settings. Rename the new device as *jtag_uart*. Supply the clock and reset signals from the *clock* component, and the *jtag_debug_module_reset* signal from *cpu* to *jtag_uart*. Connect the *data_master* port of *cpu* to the *avalon_jtag_slave* port of *jtag_uart*.

Our SoC needs a timer device for precise time calculations (this is required when preparing software applications). Add an *Interval Timer* from the library (*Library->Peripherals->Microcontroller Peripherals)*. Select *"Full Featured"* from the *Presets* list and click *"Finish"*. Rename the new device as *timer*. Supply the clock and reset

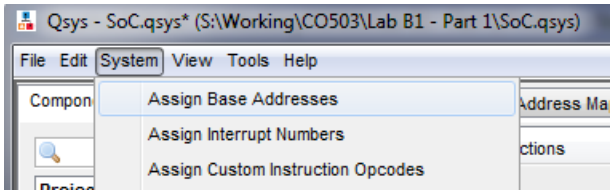
signals from the *clock* component, and the *jtag_debug_module_reset* signal from *cpu* to *timer*. Connect the *data_master* port of *cpu* to the *s1* port of *timer*.

To prevent accidentally downloading software compiled for a different SoC, we will add a *System ID peripheral* from the library (*Library->Peripherals->Debug and Performance)*. Leave the *32-bit System ID* as "*0x00000000*" and click *"Finish"* (In real systems, a unique ID could be provided). Rename the new device as *sysid*. Supply the clock and reset signals from the *clock* component, and the *jtag_debug_module_reset* signal from *cpu* to *sysid*. Connect the *data_master* port of *cpu* to the *control_slave* port of *sysid*.

Driving the 8 LED pins we created earlier (in the top level file) requires a parallel I/O interface for the *cpu*. Go to *Library->Peripherals->Microcontroller Peripherals* and add a *PIO* device. Select the *Width* as *"8"* bits and the *Direction* as *"Output"*, then click *"Finish"*. Rename the new *PIO* as *led_out*. Supply the clock and reset signals from the *clock* component, and the *jtag_debug_module_reset* signal from *cpu* to *led_out*. Connect the *data_master* port of *cpu* to the *s1* port of *led_out*. In the *external_connection* row, double-click on the *Export* column to export *led_out* pins to outside.



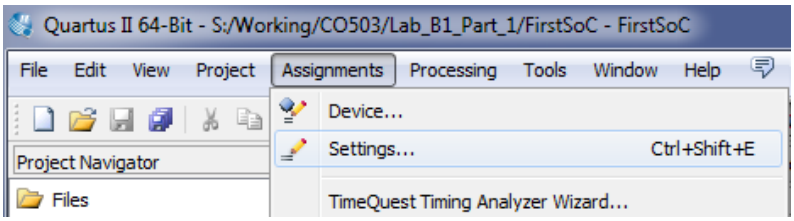
At this point, you may see a number of error messages. Most of the errors are due to overlapping address ranges . Nios II processors access peripheral devices through memory mapped IO, hence use unique address ranges for each attached peripheral device. To ensure that each device in our system uses a unique address range, select *Assign Base Addresses* from the *System* menu. You will see that the *Base* and *End* address on most devices in your system are now changed, so that no overlap occurs. **Note the new base address** of the *led_out* PIO device: `0x_____` you will need to use this value later on.

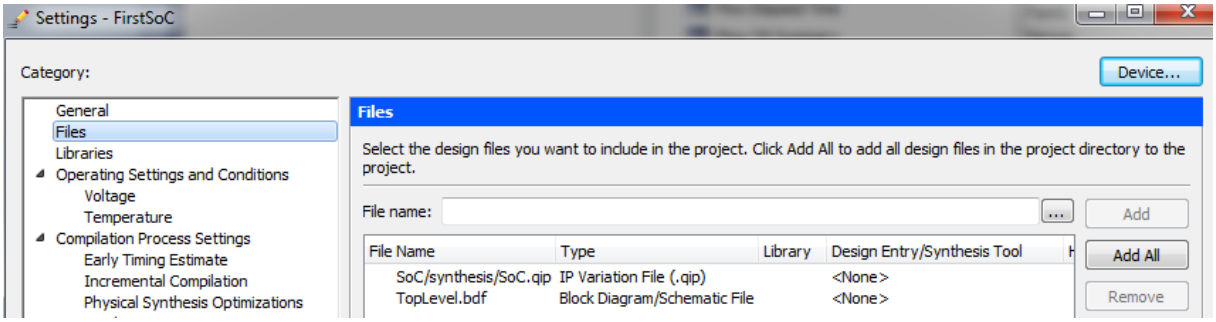


The *timer* and *jtag_uart* in our system send interrupt requests (IRQs) to the *cpu*. We must assign priorities to these interrupts. In the *IRQ* column, click on the empty circles in the rows of each device and assign a number (lower number means higher priority). Assign *"1"* for the *timer* and *"16"* for the *jtag_uart*. Qsys also provides an *Assign Interrupt Numbers* command which automatically connects *IRQ* signals. However, assigning IRQs effectively requires an understanding of how software responds to them. Because Qsys does not know the software behaviour, it cannot make educated guesses about the best IRQ assignment.

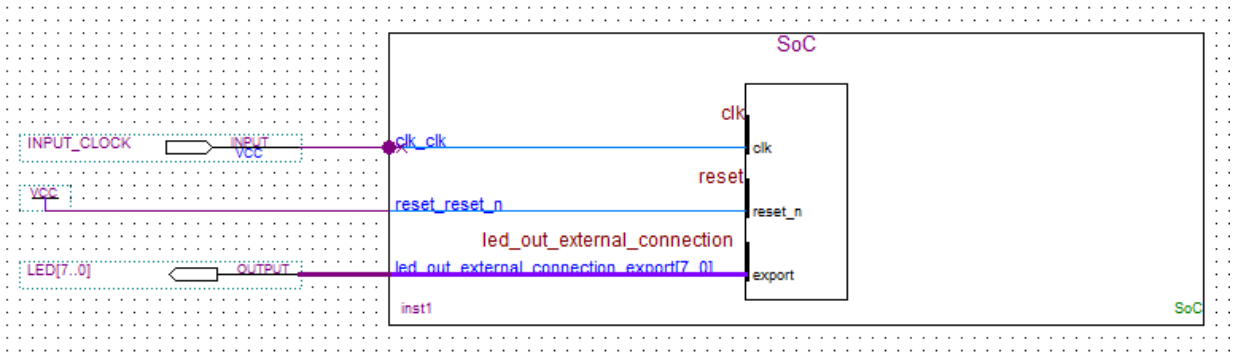
Now we're ready to generate our SoC. First, save the system under the name *SoC*. Then go to the *Clock Settings* tab and make sure the clock frequency matches the oscillator on the board, and go to the *Project Settings* tab and make sure the FPGA device ID is correct. Go to the *Generation* tab. Leave the simulation models as *"None"*, as we are going to deploy the SoC in actual FPGA hardware. Click on the "Generate" button. Close Qsys and return to Quartus II.

4. In order for Quartus to link the newly created *SoC* with our project *FirstSoC*, we must add the Quartus IP file of our *SoC* to the project. On the *Assignments* menu, click *Settings*. The Settings dialog will appear. Under *Category*, select *Files*. Next to *File name*, click the browse (...) button. In the *Files of type* list, select *Script Files (*.tcl, *.sdc, *.qip)*. Browse to locate "*<project directory>/SoC/synthesis/SoC.qip*" and click *Open* to select the file. Click "*Add*" to include the *SoC.qip* file in the project. Click *"OK"* to close the Settings dialog box.





5. We should now add the symbol for the SoC we created into our top level diagram. Open the *TopLevel.bdf* file, right click on the dotted area and select *Insert->Symbol*. You should see a new library called *Project*. Select the *SoC* block from the *Project* library and place it on the top level diagram . Next, we connect the previously added pins to the new SoC block. Connect the *INPUT_CLOCK* pin to the clock input of the SoC, and the *VCC* pin to the reset signal of the SoC. Finally, connect the set of pins *LED[7..0]* to *led_out_external_connection_export[7..0]*.

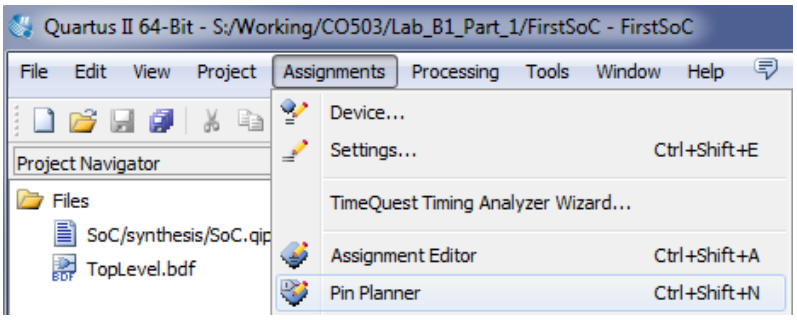


Save the *TopLevel.bdf* file.

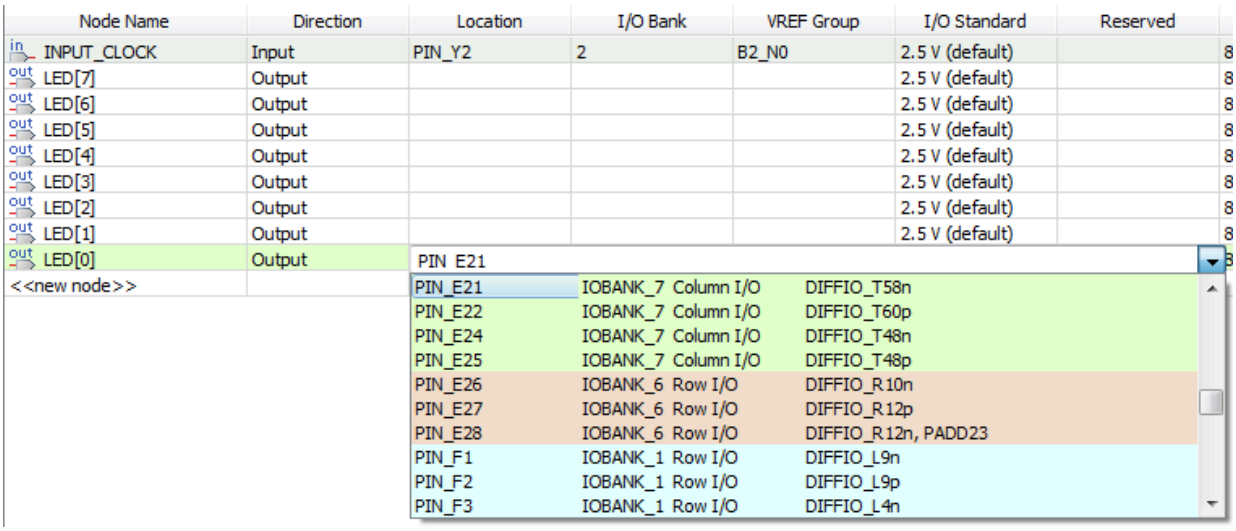
6. Now that our hardware design is complete, we should ask Quartus to perform a preliminary analysis on it and identify any errors. Start the analysis and elaboration process (*Processing->Start->Start Analysis and Elaboration*). This may take a few minutes to finish.

7. After an error-free analysis, the next step is to map the inputs/outputs of our hardware design to the outside world (external components on the board, outside the FPGA). Since we use an evaluation board, the FPGA device's pins are already hardwired to these external components. So, all we have to do is map the inputs/outputs of our design to specific pins of the FPGA device.

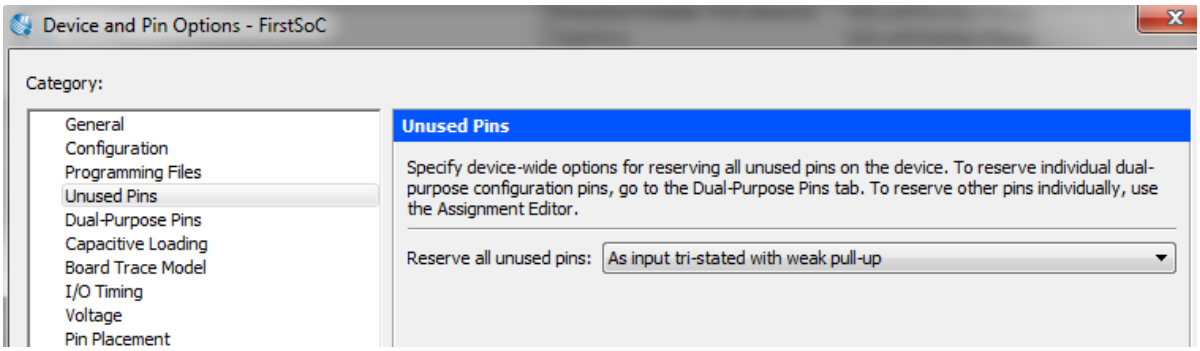
   In the *Assignments* menu, click on the *Pin Planner*. This will bring up a vivid diagram, with the top view pin layout of our FPGA device.



   At the bottom, you will see a list of I/O pins in our design (clock input and LED outputs). At each row, double click on the *Location* column to select an appropriate pin from the list. Lists of various pin numbers and their corresponding hardwired components can be found in the user manual of the FPGA development kit.



Select pins for all inputs and outputs in the list. Close the *Pin Planner* window when done. On the *Assignments* menu, click *Device*. The Device dialog will appear. Click the *"Device and Pin Options"* button. In the *Unused Pins* page, set *Reserve all unused pins* as *"input tri-stated with weak pull-up"*. With this setting, all unused I/O pins on the FPGA enter a high-impedance state after power-up.



Click *"OK"* to close the Device and Pin Options dialog. Click *"OK"* to close the Device dialog.

8. It's time to compile our hardware design. The compiler will perform the number of tasks: 1) analysing the design; 2) synthesizing; 3) fitting (placing and routing); 4) generating assembler; and 5) analysing the timing. Select *Processing->Start Compilation* or click the  button to start the compilation. At the end, you should see a summary report like below.

Flow Summary

Flow Status	Successful - Wed Aug 10 09:03:48 2016
Quartus II 64-Bit Version	12.1 Build 177 11/07/2012 SJ Full Version
Revision Name	SoC
Top-level Entity Name	SoC
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	2,424 / 114,480 (2 %)
Total combinational functions	2,258 / 114,480 (2 %)
Dedicated logic registers	1,293 / 114,480 (1 %)
Total registers	1293
Total pins	9 / 529 (2 %)
Total virtual pins	0
Total memory bits	2,125,888 / 3,981,312 (53 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

The report shows various resource usages for our hardware design, from the available FPGA resources.

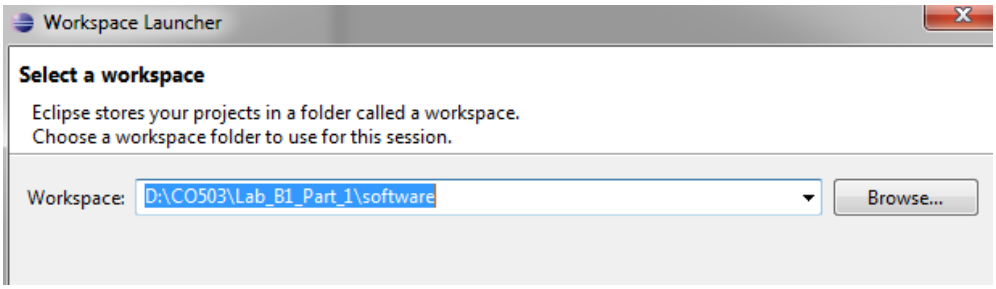
9. Expand *TimeQuest Timing Analyzer* and click on *" Multicorner Timing Analysis Summary "* from the table of contents. This shows the timing performances of the clock signals. Any negative slack values indicate the paths for the clock are too slow. Prior to compilation, you can manually set design constraints through an SDC file, which will force the fitting algorithm to try alternate options to satisfy the constraints.

Multicorner Timing Analysis Summary

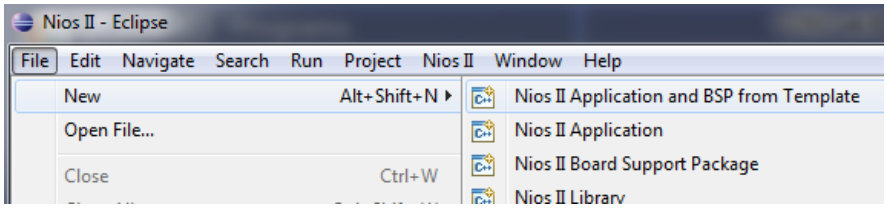
	Clock	Setup	Hold	Recovery	Removal
1	Worst-case Slack	46.231	0.179	48.122	0.483
1	altera_reserved_tck	46.231	0.179	48.122	0.483
2	Design-wide TNS	0.0	0.0	0.0	0.0
1	altera_reserved_tck	0.000	0.000	0.000	0.000

10. Finally, we are about to download our design onto the FPGA device. Make sure your DE2-115 board is powered and its USB Blaster port is connected to the computer. Start the Programmer (*Tools->Programmer*). The programmer should automatically detect the FPGA device and the bitstream (.sof file) to be downloaded. Click *"Start"* to begin downloading.

11. Now we're about to create a software application for our SoC. Open *Nios II Software Build Tools*. As the workspace folder, create a new folder named *software* inside your project directory.



Create a new software project (*File->New->Nios II Application and BSP from Template*).

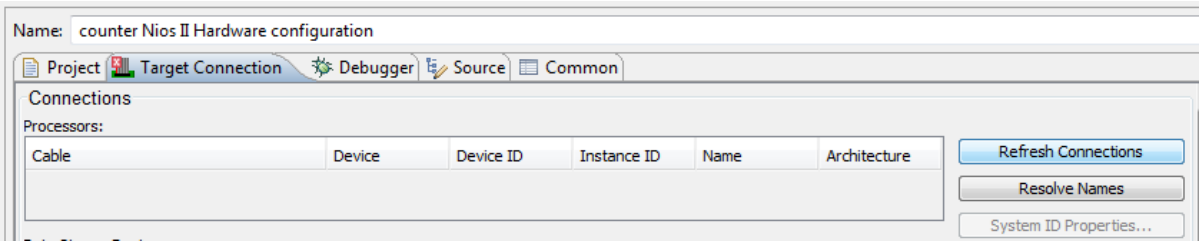


Select the target hardware by browsing for *"SoC.sopcinfo"* file in the project directory. Choose *cpu* as the CPU name from the dropdown list. Name the project as *counter*. Select *"Blank Project"* as the template and click *"Finish"*. You will see two new projects are created: *counter* is the application program; *counter_bsp* is the auto generated board support package (a tiny OS). Right click on the *counter_bsp* project, go to *Nios II->BSP Editor*. Select *timestamp_timer* from the list and set the value to *"timer"*. Click "Generate" button and close the window. Save the files if prompted. Right click on the *counter_bsp* project again and go to *Nios II->Generate BSP*.

Now we will prepare the application program. Right click on the *counter* project and import the *counter.c* file provided to you (*Right Click->Import->General->File System->Browse for the file*). Study the imported code. Insert the base address of the *led_out* PIO device (which you noted earlier), at the correct place in the code. Save the file, right click on the *counter* project again and build it.

What do you think the statement IOWR_8DIRECT(LED_BASE,OFFSET,count); does?

12. Finally, it's time to run the app. Right click the *counter* project, go to *Run As->Nios II Hardware*. A *Run Configurations* window may pop up. Go to the *Target Connections* tab and click the *"Refresh Connections"* button, then *"Finish"*.



The application should now be downloaded onto the FPGA, and you should see the count value change on LEDs. Try changing the code. Any *"printf"* statements should write the output to the JTAG interface in our SoC (*jtag_uart* is stdout), which will then be displayed in the host console (these settings may be changed in the BSP)

If you encounter any errors when trying to run the software:

- Verify that you haven't missed anything at previous steps. Common mistakes are: incorrect connections in the SoC; wiring in the schematic diagram not properly connected; pin mapping not complete.
- If everything is in order, you may need to reduce the clock speed of the SoC using a PLL. You may check the next part of the practical to see how this can be done.

Part 2: JPEG Encoder SoC

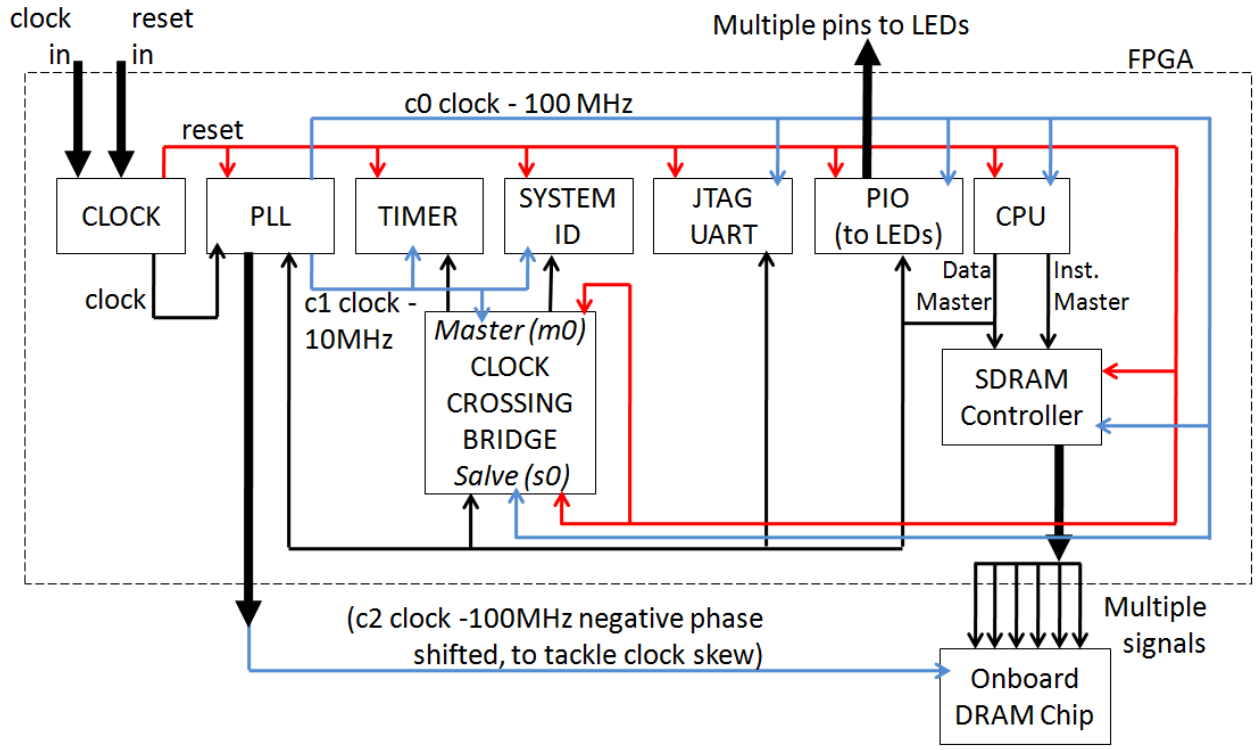
Now that you're familiar with the process, let's create a new SoC for JPEG image encoding using onboard SDRAM as the main memory (instead of on-chip memory). Input and output files are accessed from the host computer through the JTAG USB cable. LEDs on the board should indicate the processing status in a suitable way (*encoding*, *finished, waiting, etc.*).

Create a new *Quartus II* project called *JSoC*. Name the top-level BDF as *TopLevel.BDF*, and name the QSYS system as *SoC*. **Do not use spaces** in the directory/file names.

For this system, you should use a 10MHz clock for *timer* and *sysid*, 100MHz clock for the other SoC components and 100MHz clock with a -65 degree phase shift for the on-board DRAM chip. You can use a Phase-Locked-Loop (PLL) to generate these clocks. To facilitate communication between SoC components using different clocks, a *CLOCK CROSSING BRIDGE* should be used on the *data_master* bus.

Following are some important information you will need:

1. **Hardware block diagram**:



a. New hardware components to be used*: SDRAM Controller*, Avalon ALT*PLL*.

b. Parameters for *SDRAM Controller*: Preset = *Custom*
Chip select = *1*
Banks = *4*
Row = *13*
Column = *10*
Access time (t_ac) = *6ns*
Base Address = *0x0000_0000*

c. Parameters for Avalon ALT*PLL*: Input frequency (inclk0) = *50MHz*
No asynchronous reset input or locked output
Output clock *c0*: Requested Frequency = *100MHz*
Requested Phase shift = *0 degrees*
Output clock *c1*: Requested Frequency = *10MHz*
Requested Phase shift = *0 degrees*
Output clock *c2*: Requested Frequency = *100MHz*
Requested Phase shift = *-65 degrees*

d. Connect the *jtag_debug_module_reset* signal from *cpu* to all reset inputs of all components **except** the *clock* component.

e. Pins in BDF to interface DRAM:

SDRAM_ADDR[12..0]	*- output*
SDRAM_BA[1..0]	*- output*
SDRAM_CAS_N	*- output*
SDRAM_CKE	*- output*
SDRAM_CS_N	*- output*
SDRAM_DQ[31..0]	*- bidirectional*
SDRAM_DQM[3..0]	*- output*
SDRAM_RAS_N	*- output*
SDRAM_WE_N	*- output*
SDRAM_CLK	*- output*

(Map these pins to appropriate hardwired pins, using the information in the user manual)

2. **Software Application**:

Create a new *Nios II* application and BSP project called *JPEG_Encoder*, using a blank template and *SoC.sopcinfo* file as the target hardware. Import the provided code and sample input files into the application project. Modify the code in *jpeg_encoder.c* file to display the processing status on the LEDs.

a) Parameters for the BSP: timestamp timer = *timer* (*from the QSYS system*)
Enable *altera_hostfs* software package.
hostfs_name = */mnt/host*

b) When launching the application, use ***Debug As Nios II Hardware***, instead of *Run As*.

This design uses both on-chip and off-chip memory. The evaluation board includes an off-chip DRAM memory module. To access this memory, the FPGA design must include a DRAM controller that mediates communication between the CPU and the memory chip. The design also introduces multiple clock domains—100 MHz for the CPU and 10 MHz for components such as the timer and system ID module. Communication across clock domains requires a **clock-crossing bridge**.

A **phase-locked loop (PLL)** is also used to generate appropriately phased clocks. Since off-chip memory receives clock signals through longer copper traces on the board, clock edges arrive later than they do for on-chip components. This difference is known as **clock skew**. To compensate, a phase-shifted clock is provided to the off-chip memory so that both on-chip and off-chip components experience aligned clock edges. The phase-shift values have already been measured for the board and are provided in the lab documentation.

Practical 2

Processor Customization

In the previous practical, we examined system-on-chip (SoC) design using FPGA platforms, where a general-purpose soft processor is integrated with memory, peripherals, and custom hardware components. While this approach provides flexibility and rapid prototyping capability, it also raises an important question: *is a general-purpose processor always the most efficient choice for a given application?*

This chapter introduces the concept of processor customization - the process of adapting a processor architecture to better suit a specific application. Using FPGA-based SoC platforms, designers can go beyond fixed instruction sets and modify or extend a processor to improve performance, efficiency, and functionality. We focus on customization techniques relevant to soft processors such as Nios II, with particular emphasis on **custom instructions**.

Motivation for Processor Customization

General-purpose processors are designed to support a wide range of applications. Their instruction set architectures (ISAs) include operations that are broadly useful, but not necessarily optimal for every workload. As a result, application performance is often limited by the need to express complex operations using long sequences of basic instructions.

In contrast, **Application-Specific Instruction-set Processors (ASIPs)** are tailored to a particular class of applications. By customizing the processor architecture, designers can:

- Reduce execution time for critical code sections
- Improve energy efficiency
- Decrease code size
- Offload computation from software to hardware

FPGA-based systems provide a unique opportunity to explore this design space, as the processor itself is implemented in reconfigurable logic and can be modified without fabricating new silicon.

Types of Processor Customizations

Processor customization can take several forms, including:

- ISA extensions, such as adding new instructions
- Custom instructions, implemented as dedicated hardware units
- Datapath modifications, integrating new functional units
- Memory subsystem optimizations, including cache configurations

In this practical, we focus primarily on custom instructions, as they provide a clear and practical example of hardware–software co-design within an FPGA-based SoC.

Custom Instructions

Application software is typically written in a high-level language such as C. The compiler translates this source code into assembly instructions defined by the processor's ISA.

```
void main(){
   while(…){
   …
       if(…)
          …
   y = a * b
   }
}
```

Consider a program that repeatedly performs multiplication operations inside a loop. If the ISA lacks a multiplication instruction, the compiler must generate a sequence of simpler instructions to emulate that single operation, such as using a series of additions. This increases the instruction count. Since the multiplications happen repeatedly inside a loop, this series of additions get repeated in the assembly. This causes the execution time to significantly increase.

By extending the ISA with a new instruction (for example, a `mul` instruction), and by implementing the required hardware support to perform the multiplication, the same operation can be executed more efficiently. If such operations occur frequently in the application, the performance gains can be significant.

The Process of Including a Custom Instruction

The process of adding a custom instruction to the ISA and the microarchitecture begins with application analysis. The typical steps involve:

1. Examining the source code and generated assembly code
2. Identifying performance-critical sections, such as loops or repeated instruction sequences
3. Determining whether these sequences can be replaced by a single, specialized instruction
4. Design the hardware required to support that instruction's operation
5. Integrate the new hardware into the datapath along with control signals
6. Including system support to use the new custom instruction by software:
    - Integrate with system libraries, and use explicitly
    - Implement compiler support

Operations that involve repeated arithmetic, bitwise manipulation, or data transformations are especially good candidates. Once identified, these operations can be implemented directly in hardware and exposed to software as new instructions.

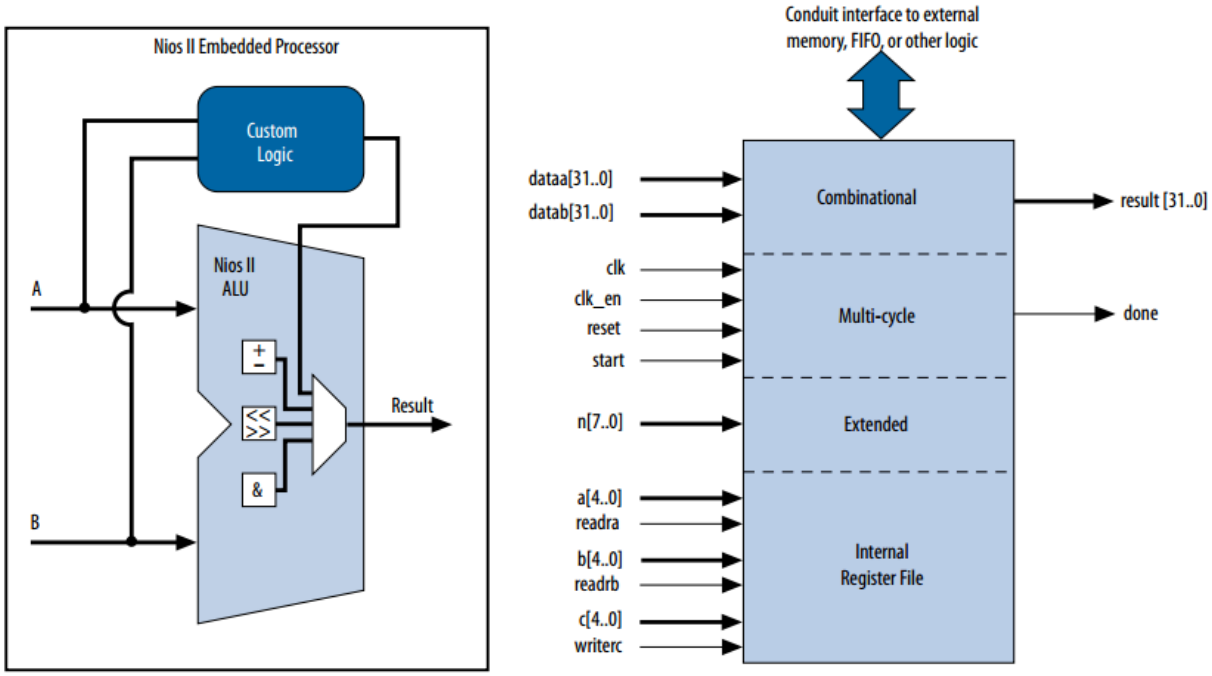
Nios II Custom Instructions

Hardware Integration

In the Nios II processor, custom instructions are implemented as separate hardware units connected directly to the processor datapath. The typical process includes:

1. Designing the custom instruction hardware using a hardware description language (HDL)
2. Adding the design as a new component in **Platform Designer (Qsys)**
3. Defining interfaces and control signals according to the Nios II custom instruction specification
4. Assigning a unique selection index for the instruction
5. Connecting the custom instruction component to the processor's `custom_instruction_master` interface

Once integrated, the custom hardware becomes part of the processor execution pipeline, allowing the instruction to be executed efficiently.



Software Integration

From the software perspective, custom instructions must be made accessible to application code. This is typically done through compiler-provided built-in functions or macros. In the Nios II environment, custom instructions can be invoked using special intrinsic functions defined in system header files.

Developers may:

- Explicitly call the custom instruction using built-in functions
- Integrate the instruction into system libraries
- Compare software-only implementations with hardware-accelerated versions

This enables systematic evaluation of performance improvements.

Exercise: Custom Instruction for CRC Computation

A practical example of processor customization is the implementation of a cyclic redundancy check (CRC) computation as a custom instruction. CRC algorithms involve repeated XOR and shift operations, which can be inefficient when implemented purely in software.

Three approaches will be compared:

1. A software-only implementation using iterative modulo-2 division
2. An optimized software version using lookup tables
3. A hardware-accelerated version using a custom instruction

In the hardware-based approach, the CRC computation is performed in parallel within the custom instruction unit, significantly reducing execution time. Performance measurements can be obtained using a **high-resolution timer** integrated into the SoC.

Steps: Hardware

1) Create the hardware unit for the custom instruction (*HDL*)
2) Add as a New Component to the library in Qsys
   a. give a name “CRC_CUSTOM”
   b. provide the HDL files and denote the top-level, then analyze
   c. set up the interfaces and signals (*guide!*)
3) Add an instance of CRC_CUSTOM and name it “crc”
4) Assign a selection index base value (0) (*what is this?*)
5) Connect “crc” to the *custom_instruction_master* port of the CPU
6) Add a second timer (at *us* scale) and name it “high_resolution_timer”
7) Generate and compile

Steps: Software

1) *system.h -> custom x, n, A*

__built_in_custom_ini(x, n, A)

2) Study the provided source code

   a. CRC algorithm needs to calculate the remainder of a division

   b. method 1: iterative modulo 2 division in S/W

   c. method 2: using a look-up table (optimized)

   d. method 3: using the custom instruction
   (XOR and shift in H/W, in parallel)

3) Assign the “high_resolution_timer” for time-stamping (BSP)

4) Build and run. Compare the performance of the three methods

Practical 3

Multiprocessor Systems-on-Chip

This practical introduces **multiprocessor systems-on-chip (MPSoCs)** using FPGA-based design tools. You will design, implement, and evaluate a system that integrates multiple processor cores on a single FPGA fabric, focusing on *inter-processor communication* mechanisms.

A fundamental challenge in MPSoC design is enabling efficient data exchange between processors. One widely used approach is **shared memory**, where processors communicate by reading from and writing to a common memory region. Another approach is to use **dedicated hardware communication mechanisms**, such as FIFO queues, which can offer improved performance and determinism.

In this practical, you will explore both approaches in two stages:

- **Part 1** introduces a shared-memory-based producer–consumer MPSoC.
- **Part 2** replaces the shared-memory software mechanism with a dedicated hardware FIFO, allowing you to compare the two communication strategies.

Part 1: Producer-Consumer Applications on a Shared Memory Multiprocessor

In the first part of the practical, you will design a simple MPSoC consisting of **two processor cores** that communicate through shared memory. The application follows the classic *producer–consumer* model, where one processor generates data items (tokens) and the other processes them.

Hardware Design

Using **Platform Designer (Qsys)**, create a new system containing two Nios II processors, named `cpu0` and `cpu1`. The system must satisfy the following requirements:

- Each processor must have its **own timer** and **own JTAG UART** for independent timing and debugging.
- Each processor must use a **dedicated on-chip memory** as its *instruction memory*.
- Both processors must share a **separate on-chip memory device** that serves as *shared data memory*.
- A **single 50 MHz clock** may be used to drive all components in the system.

Before proceeding further, discuss with your peers how the shared data memory should be **partitioned**. The memory must include:

- A private data region for `cpu0`
- A private data region for `cpu1`
- A shared region accessible by both processors for inter-processor communication

Decide on a clear **memory partitioning scheme** and document it before continuing with the software development.

Software Design

Once the hardware system has been completed and compiled, create **two separate software projects**, one targeting each processor:

- The first project runs the **producer application** on `cpu0`.
- The second project runs the **consumer application** on `cpu1`.

To enforce the chosen memory partitioning scheme, you must modify the **linker script** for each project using the BSP Editor. This ensures that private data and shared data are placed in the correct memory regions.

You are provided with sample producer and consumer applications. Study this code carefully to understand the expected behavior. The producer generates tokens and writes

them into a communication buffer, while the consumer reads and processes incoming tokens.

Your task is to implement the communication mechanism using **shared memory**. A skeleton implementation of a **software FIFO queue** is provided for this purpose. Carefully study the skeleton code and complete it so that the FIFO data structure resides entirely within the shared memory partition.

Alternatively, you may design your own software FIFO implementation, provided that:

- The FIFO is implemented exclusively within the shared data memory region
- Both processors access the FIFO in a safe and consistent manner

This part of the practical highlights the challenges of shared-memory communication, including synchronization, memory organization, and software overhead.

Part 2: Hardware FIFO-based Communication

In the second part of the practical, you will redesign the MPSoC to use a **dedicated hardware FIFO** for inter-processor communication.

Create a new MPSoC that is structurally similar to the system developed in Part 1. However, instead of implementing a software FIFO in shared memory, include an **on-chip hardware FIFO memory component** using Platform Designer.

The producer and consumer applications should be modified to use this hardware FIFO for data transfer. Refer to the provided datasheet for the on-chip FIFO component to understand its interface, configuration options, and usage.

Conduct a detailed comparison between the two systems you've created. By comparing this hardware FIFO implementation with the shared-memory approach from Part 1, you will observe how hardware-supported communication can:

- Reduce software complexity
- Improve communication performance
- Provide more predictable behavior

Practical 4

JPEG MPSoC Case Study

In this practical, you will design and implement a **multiprocessor system-on-chip (MPSoC)** for **JPEG image encoding** using FPGA design tools. This exercise serves as a capstone practical that integrates the concepts and skills developed throughout the previous practicals, including system-on-chip design, multiprocessor architectures, inter-processor communication, hardware FIFOs, and performance analysis.

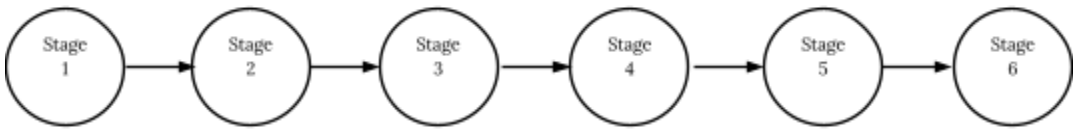
In addition to applying previously learned techniques, this practical requires independent technical research. You will study an existing pipelined JPEG encoder architecture described in a provided research paper and adapt its design principles to an FPGA-based implementation using Nios II soft processors.

The primary objective is not only to build a functional JPEG encoder, but also to understand how **pipeline parallelism**, **synchronization**, and **processor-level optimizations** affect system throughput.

Part 1: Pipelined JPEG MPSoC

Application Decomposition

JPEG encoding can be decomposed into a sequence of well-defined processing stages. In this practical, the JPEG encoder is divided into **six distinct stages**, each operating on one image macro-block:



Stage 1: Input and Color Space Conversion
Read a macro-block from the raw image and convert pixel values from RGB format to YCbCr.

Stage 2: Level Shifting
Adjust pixel values to center them around zero in preparation for frequency-domain processing.

Stage 3: Discrete Cosine Transform (DCT)
Perform vertical and horizontal DCT operations to convert spatial-domain data into the frequency domain.

Stage 4: Quantization
Quantize DCT coefficients using the JPEG quantization tables.

Stage 5: Huffman Encoding
Apply entropy encoding to compress the quantized coefficients.

Stage 6: Output Generation
Write the encoded macro-block to the output JPEG image.

These stages form a **processing pipeline**. A macro-block flows sequentially through all six stages, and in steady-state operation each stage processes a different macro-block concurrently. This pipelined execution model enables significantly higher throughput compared to a single-processor sequential implementation.

Pipeline Architecture

Each pipeline stage is implemented using a **dedicated Nios II processor core**. Communication between consecutive stages is achieved using **hardware FIFO queues**.

Each processor reads data from its input FIFO, performs the computation required for its stage, and writes the result to its output FIFO. The FIFO queues provide buffering and synchronization between stages, allowing them to operate at different speeds without requiring tight coupling.

You are provided with a reference implementation of a pipelined JPEG encoder that uses **Tensilica Xtensa embedded processors** and hardware FIFO queues. Refer to the accompanying research paper to understand:

- The mapping of JPEG stages to processors
- The structure and role of FIFO-based inter-stage communication
- Performance considerations in pipelined multimedia processing systems

Your task is to implement an equivalent architecture on FPGA hardware using **Nios II processors** and **on-chip hardware FIFOs**.

Synchronization and Performance Measurement

In practice, different JPEG stages have different computational workloads. As a result, some stages execute faster than others. The overall throughput of the pipeline is therefore limited by the **slowest stage**.

When a stage completes its computation faster than adjacent stages, it must either:

- Stall if its output FIFO is full, or
- Wait if its input FIFO is empty

Correct use of FIFO status signals is essential to ensure proper synchronization and correct system behavior.

Once the system is operating correctly, measure the **steady-state throughput** of the pipeline. Specifically, determine:

- How frequently encoded macro-blocks are produced
- The execution time consumed by each pipeline stage

These measurements establish a baseline for performance analysis and optimization in Part 2.

Part 2: Improving Performance

The second part of this practical focuses on **performance optimization**. Your objective is to improve the **throughput** of the JPEG encoder—that is, to increase the rate at which encoded macro-blocks are produced during steady-state operation.

Several optimization strategies may be explored, individually or in combination.

1. Processor Customization

Customize the Nios II processors to better match the computational characteristics of the JPEG workload. Possible techniques include:

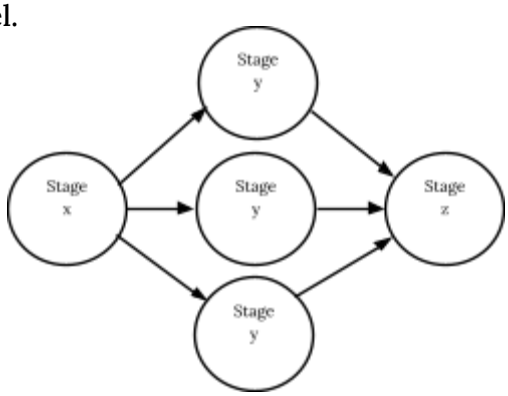
- Adding **custom instructions** for frequently used or computationally expensive operations
- Enabling **instruction and data caches**
- Configuring cache sizes and policies to suit program access patterns

2. Pipeline Extension

If a particular stage dominates execution time, consider **dividing it into multiple smaller stages**. Increasing pipeline depth can reduce the critical stage latency and improve overall throughput.

3. Superscalar Pipeline Stages

For stages that are significantly slower than others, introduce **parallelism within a stage**. This can be achieved by using multiple processors for the same stage, each operating on a portion of the macro-block. For example, four processors may each process one quarter of a macro-block in parallel.



This approach increases hardware resource usage but can significantly improve performance when the workload is highly parallelizable.

Research and Analysis

The provided research paper includes an analysis of performance bottlenecks and optimization techniques for pipelined JPEG encoders. You are expected to:

- Study the paper carefully
- Apply relevant techniques to your own design
- Explore additional academic literature for further optimization ideas

Your final implementation should demonstrate not only functional correctness, but also a clear and well-reasoned application of performance engineering principles.

Learning Outcomes

Upon completing this final practical, you should be able to:

- Design and implement a pipelined MPSoC for a real multimedia application
- Map application stages to processor cores in a pipeline architecture
- Use hardware FIFO queues for efficient inter-processor communication
- Analyze system throughput and identify performance bottlenecks
- Apply processor customization and parallelism to improve performance
- Integrate research insights into practical FPGA-based system design

Students' Guide to Practicals on Systems-on-Chip Design

By Isuru Nawinne