

CO503 : Practical 1 - System-on-Chip (SoC) Design

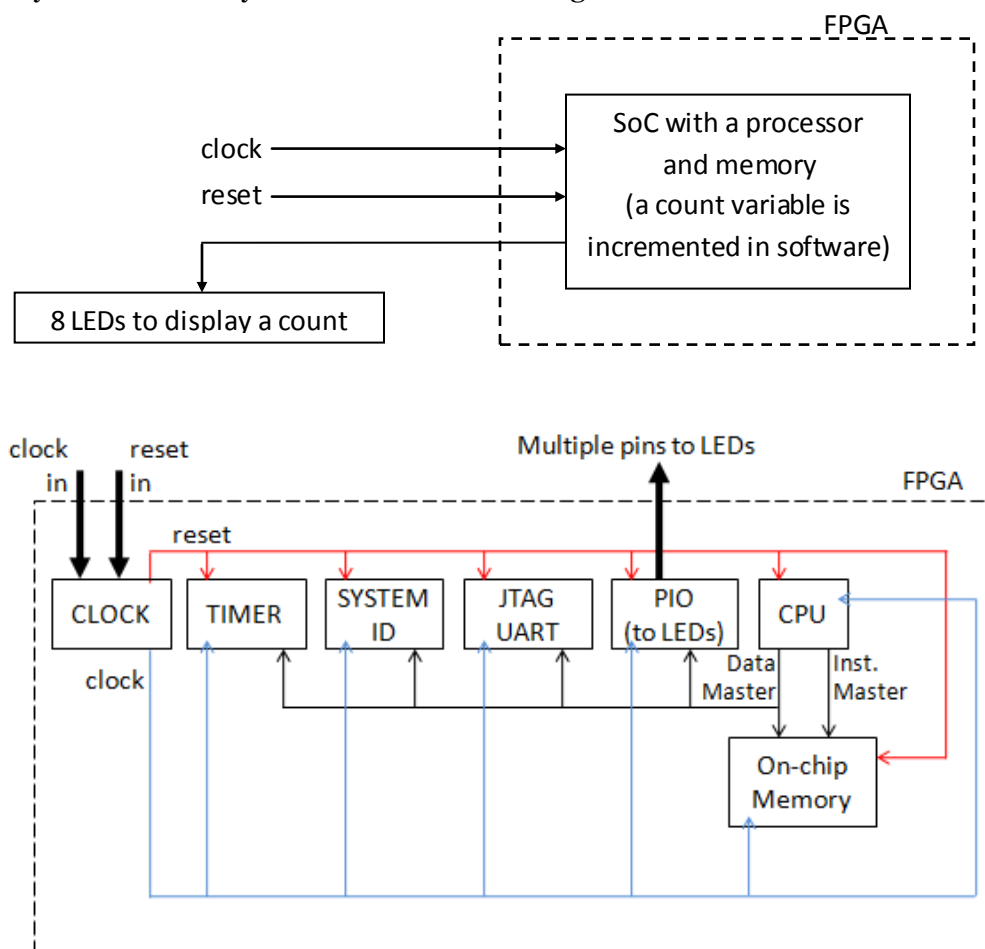
In this practical, you will use FPGA design tools to create your first System-on-Chip. The practical consists of two parts. In Part 1, you will learn how to create hardware and software for a simple SoC to control an LED counter. In Part 2 of the practical, you should build a SoC for JPEG image compression.

Learning Objectives:

- Designing and synthesizing System-on-Chip hardware.
- Co-design and development of embedded hardware and software.
- 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 guide you to build 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.

Directory, Name, Top-Level Entity [page 1 of 5]

What is the working directory for this project?

D:\CO503\Lab1

What is the name of this project?

FirstSoc

What is the name of the top-level design entity for this project? This name is case sensitive and must exactly match the entity name in the design file.

TopLevel

Use Existing Project Settings...

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

Family & Device Settings [page 3 of 5]

Select the family and device you want to target for compilation.

Device family

Family: Cyclone IV E

Devices: 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

Speed grade: Any

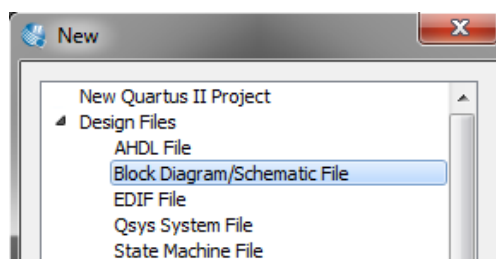
Name filter:

☒ Show advanced devices ☐ HardCopy compatible only

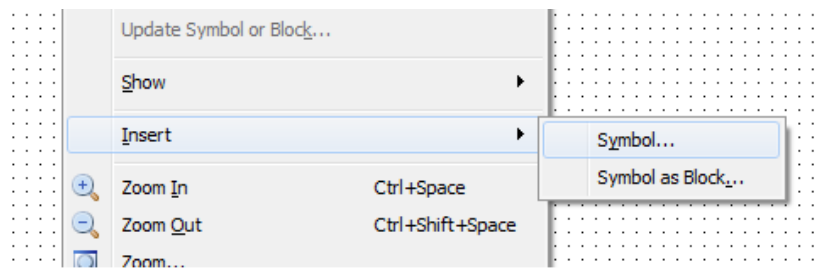
Available devices:

Name	Core Voltage	LEs	User I/Os	Memory Bits	Embedded multiplier 9-bit elements	PL
EP4CE115F23C9L	1.0V	114480	281	3981312	532	4
EP4CE115F23I7	1.2V	114480	281	3981312	532	4
EP4CE115F23I8L	1.0V	114480	281	3981312	532	4
EP4CE115F29C7	1.2V	114480	529	3981312	532	4
EP4CE115F29C8	1.2V	114480	529	3981312	532	4
EP4CE115F29C8L	1.0V	114480	529	3981312	532	4
EP4CE115F29C9L	1.0V	114480	529	3981312	532	4

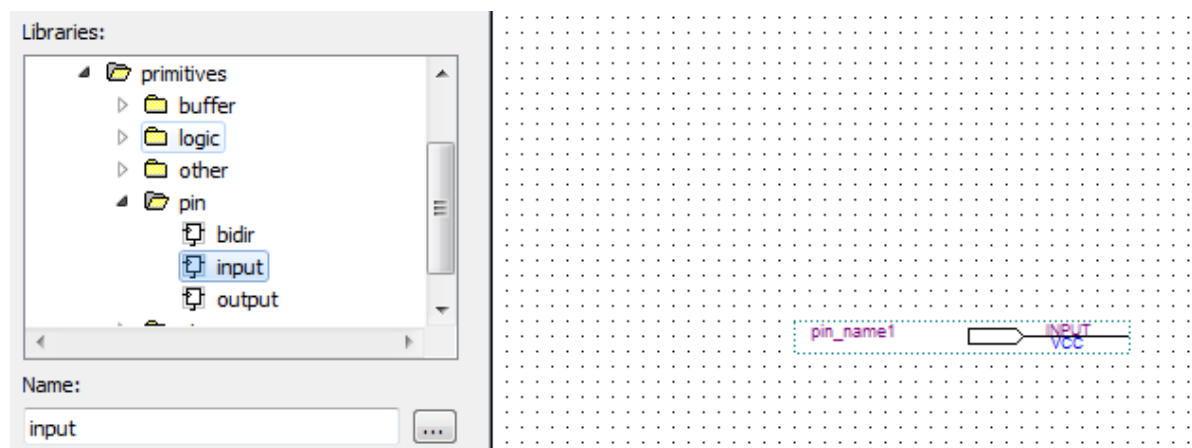
2. We're about to create the top-level design file for this project. Design files can be made in one of many ways like VHDL, Verilog, schematic, state machine, etc.. We will use a schematic (or block diagram). Create a new Block Diagram File (BDF) from the File menu (*File->New->BDF*).



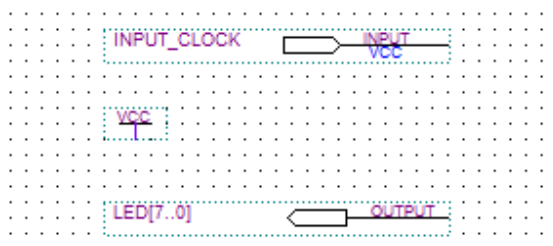
Next, we should add some I/O pins to help connect our SoC to components outside the FPGA (such as the LEDs). Right click on the dotted area in the BDF and select *Insert->Symbol*.



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*.

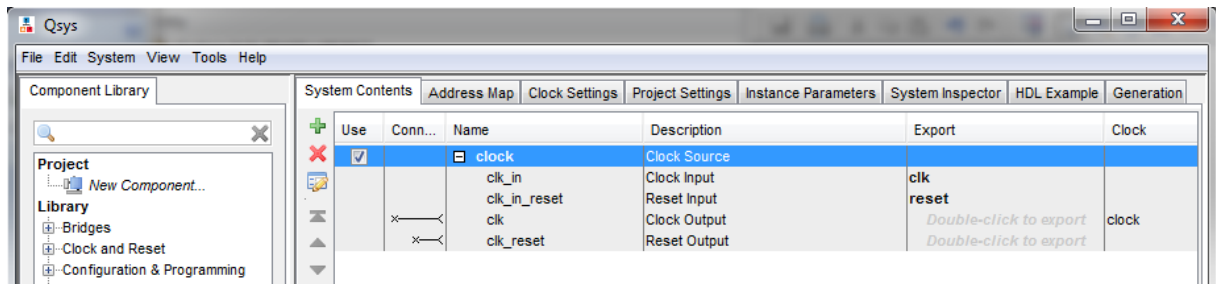


Next, add a *Vcc* symbol (*Libraries->primitives->other*). Add an output pin (*Libraries->primitives->pin*) and name it *LED[7..0]*. Note that this last symbol now represents a set of eight individual pins.



Save your BDF as *TopLevel.bdf*.

- 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_o*". 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.

Use	Conn...	Name	Description	Export	Clock
<input checked="" type="checkbox"/>		clock	Clock Source		
		clk_in	Clock Input	clk	
		clk_in_reset	Reset Input	reset	
		clk	Clock Output	Double-click to export	clock
		clk_reset	Reset Output	Double-click to export	
<input checked="" type="checkbox"/>		onchip_mem	On-Chip Memory (RAM or ROM)		
		clk1	Clock Input	Double-click to export	clock
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk1]
		reset1	Reset Input	Double-click to export	[clk1]

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".

Core Nios II Caches and Memory Interfaces Advanced Features MMU and MPU Settings JTAG Debug Module

Select a Nios II Core

Nios II Core:

☐ Nios II/e

☒ Nios II/s

☐ Nios II/f

	Nios II/e	Nios II/s	Nios II/f
Nios II Selector Guide	RISC 32-bit	RISC 32-bit Instruction Cache Branch Prediction Hardware Multiply Hardware Divide	RISC 32-bit Instruction Cache Branch Prediction Hardware Multiply Hardware Divide Barrel Shifter Data Cache Dynamic Branch Prediction
Memory Usage (e.g. Stratix IV)	Two M9Ks (or equiv.)	Two M9Ks + cache	Three M9Ks + cache

Hardware Arithmetic Operation

Hardware multiplication type:

☐ Hardware divide

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*.

Use	Connections	Name	Description	Export	Clock
<input checked="" type="checkbox"/>		<input type="checkbox"/> clock	Clock Source		
		clk_in	Clock Input	clk	
		clk_in_reset	Reset Input	reset	
		clk	Clock Output	Double-click to export	clock
		clk_reset	Reset Output	Double-click to export	
<input checked="" type="checkbox"/>		<input type="checkbox"/> onchip_mem	On-Chip Memory (RAM or ROM)		
		clk1	Clock Input	Double-click to export	clock
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk1]
		reset1	Reset Input	Double-click to export	[clk1]
<input checked="" type="checkbox"/>		<input type="checkbox"/> cpu	Nios II Processor		
		clk	Clock Input	Double-click to export	clock
		reset_n	Reset Input	Double-click to export	[clk]
		data_master	Avalon Memory Mapped Master	Double-click to export	[clk]
		instruction_master	Avalon Memory Mapped Master	Double-click to export	[clk]
		jtag_debug_module_reset	Reset Output	Double-click to export	[clk]
		jtag_debug_module	Avalon Memory Mapped Slave	Double-click to export	[clk]
		custom_instruction_master	Custom Instruction Master	Double-click to export	

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*.

Reset Vector

Reset vector memory:

Reset vector offset:

Reset vector:

Exception Vector

Exception vector memory:


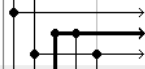

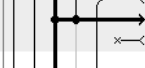


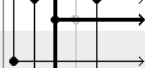
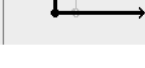
Exception vector offset:

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*.

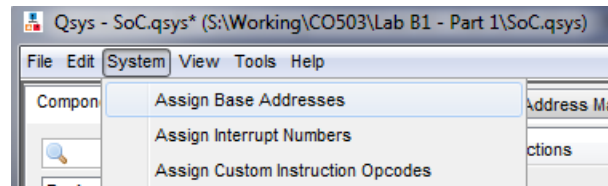
Our SoC needs a timer device for precise time calculations (this is required when preparing software applications). Add an *Interval Timer* from the library (*Library->Peripherals->Microcontroller Peripherals*). Select "Full Featured" from the Presets list and click "Finish". Rename the new device as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

Use	Connections	Name	Description	Export	Clock
<input checked="" type="checkbox"/>		<div>clock</div> <div>clk_in</div> <div>clk_in_reset</div> <div>clk</div> <div>clk_reset</div>	<div>Clock Source</div> <div>Clock Input</div> <div>Reset Input</div> <div>Clock Output</div> <div>Reset Output</div>	<div>clk</div> <div>reset</div> <div>Double-click to export</div> <div>Double-click to export</div>	clock
<input checked="" type="checkbox"/>		<div>onchip_mem</div> <div>clk1</div> <div>s1</div> <div>reset1</div>	<div>On-Chip Memory (RAM or ROM)</div> <div>Clock Input</div> <div>Avalon Memory Mapped Slave</div> <div>Reset Input</div>	<div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div>	<div>clock</div> <div>[clk1]</div> <div>[clk1]</div>
<input checked="" type="checkbox"/>		<div>cpu</div> <div>clk</div> <div>reset_n</div>	<div>Nios II Processor</div> <div>Clock Input</div> <div>Reset Input</div>	<div>Double-click to export</div> <div>Double-click to export</div>	<div>clock</div> <div>[clk]</div>
<input checked="" type="checkbox"/>		<div>data_master</div> <div>instruction_master</div> <div>jtag_debug_module_reset</div> <div>jtag_debug_module</div> <div>custom_instruction_master</div>	<div>Avalon Memory Mapped Master</div> <div>Avalon Memory Mapped Master</div> <div>Reset Output</div> <div>Avalon Memory Mapped Slave</div> <div>Custom Instruction Master</div>	<div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div>	<div>[clk]</div> <div>[clk]</div> <div>[clk]</div> <div>[clk]</div>
<input checked="" type="checkbox"/>		<div>jtag_uart</div> <div>clk</div> <div>reset</div> <div>avalon_jtag_slave</div>	<div>JTAG UART</div> <div>Clock Input</div> <div>Reset Input</div> <div>Avalon Memory Mapped Slave</div>	<div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div>	<div>clock</div> <div>[clk]</div> <div>[clk]</div>
<input checked="" type="checkbox"/>		<div>timer</div> <div>clk</div> <div>reset</div> <div>s1</div>	<div>Interval Timer</div> <div>Clock Input</div> <div>Reset Input</div> <div>Avalon Memory Mapped Slave</div>	<div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div>	<div>clock</div> <div>[clk]</div> <div>[clk]</div>
<input checked="" type="checkbox"/>		<div>sysid</div> <div>clk</div> <div>reset</div> <div>control_slave</div>	<div>System ID Peripheral</div> <div>Clock Input</div> <div>Reset Input</div> <div>Avalon Memory Mapped Slave</div>	<div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div>	<div>clock</div> <div>[clk]</div> <div>[clk]</div>
<input checked="" type="checkbox"/>		<div>led_out</div> <div>clk</div> <div>reset</div> <div>s1</div>	<div>PIO (Parallel I/O)</div> <div>Clock Input</div> <div>Reset Input</div> <div>Avalon Memory Mapped Slave</div>	<div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div>	<div>clock</div> <div>[clk]</div> <div>[clk]</div>
		external_connection	Conduit Endpoint	led_out_external_conne...	

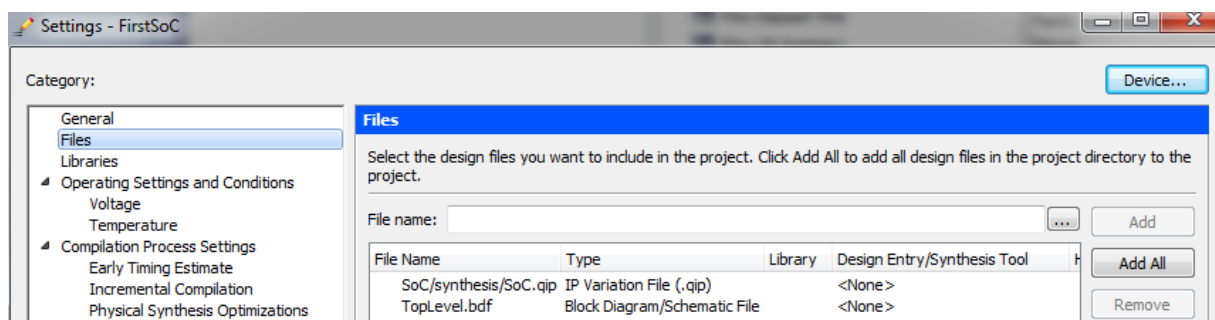
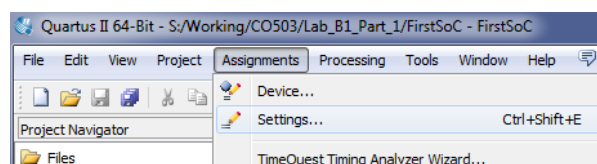
At this point, you may see a number of error messages. Most of the errors are due to overlapping address ranges. Nios II processors access peripheral devices through memory mapped IO, hence use unique address ranges for each attached peripheral device. To ensure that each device in our system uses a 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: $0 \times \underline{\hspace{2cm}}$ you will need to use this value later on.



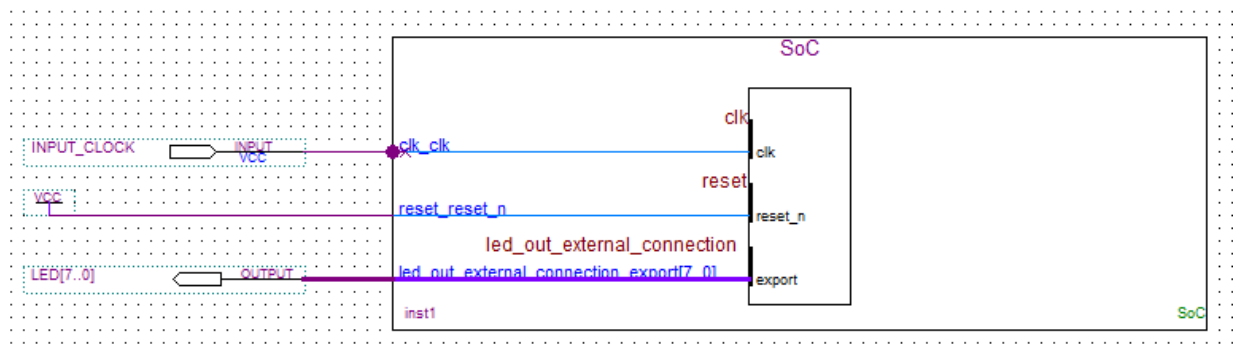
The *timer* and *jtag_uart* in our system send interrupt requests (IRQs) to the *cpu*. We must assign priorities to these interrupts. In the *IRQ* column, click on the empty circles in the rows of each device and assign a number (lower number means higher priority). Assign "1" for the *timer* and "16" for the *jtag_uart*. Qsys also provides an *Assign Interrupt Numbers* command which automatically connects *IRQ* signal. 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 *SoC.qip* file in the project. Click "OK" to close the Settings dialog box.



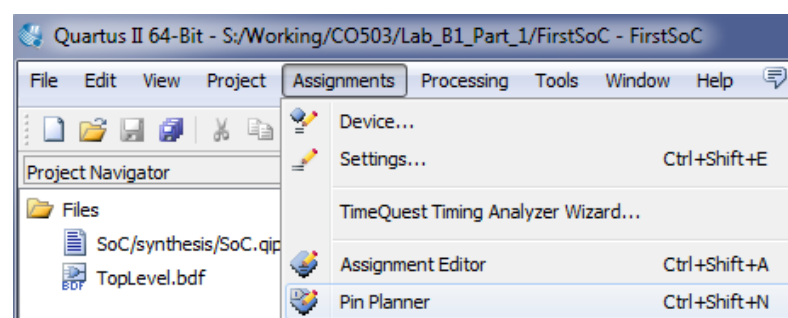
- We should now add the symbol for the SoC we created in to our top level diagram. Open the *TopLevel.bdf* file, right click on the dotted area and select *Insert->Symbol*. You should see a new library called *Project*. Select *SoC* block from the *Project* library and place it on the top level diagram. 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.

- 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.
- 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.

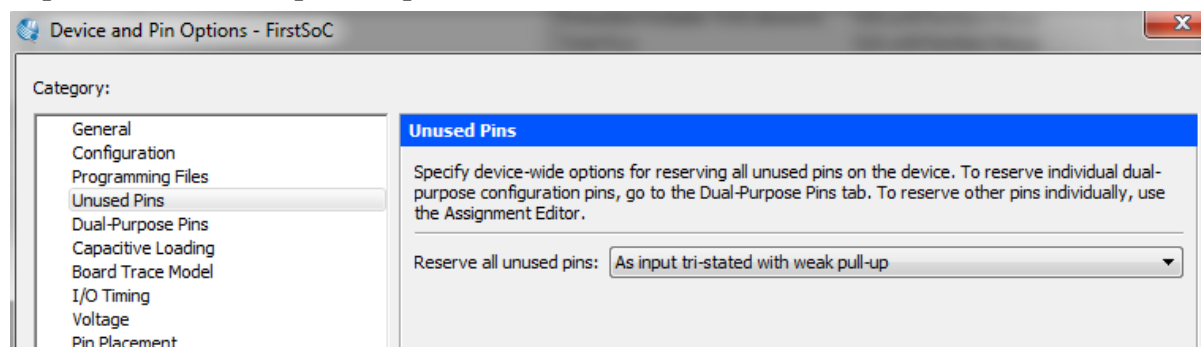
In the *Assignments* menu, click on the *Pin Planner*. This will bring up a vivid diagram, with the top view pin layout of our FPGA device.



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 Lab Board (available in the course web page).

Node Name	Direction	Location	I/O Bank	VREF Group	I/O Standard	Reserved	
in INPUT_CLOCK	Input	PIN_Y2	2	B2_N0	2.5 V (default)		8
out LED[7]	Output				2.5 V (default)		8
out LED[6]	Output				2.5 V (default)		8
out LED[5]	Output				2.5 V (default)		8
out LED[4]	Output				2.5 V (default)		8
out LED[3]	Output				2.5 V (default)		8
out LED[2]	Output				2.5 V (default)		8
out LED[1]	Output				2.5 V (default)		8
out LED[0]	Output	PIN_E21			2.5 V (default)		8
<<new node>>		PIN_E21	IOBANK_7 Column I/O	DIFFIO_T58n			8
		PIN_E22	IOBANK_7 Column I/O	DIFFIO_T60p			
		PIN_E24	IOBANK_7 Column I/O	DIFFIO_T48n			
		PIN_E25	IOBANK_7 Column I/O	DIFFIO_T48p			
		PIN_E26	IOBANK_6 Row I/O	DIFFIO_R10n			
		PIN_E27	IOBANK_6 Row I/O	DIFFIO_R12p			
		PIN_E28	IOBANK_6 Row I/O	DIFFIO_R12n, PADD23			
		PIN_F1	IOBANK_1 Row I/O	DIFFIO_L9n			
		PIN_F2	IOBANK_1 Row I/O	DIFFIO_L9p			
		PIN_F3	IOBANK_1 Row I/O	DIFFIO_L4n			

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 "*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.

- 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.

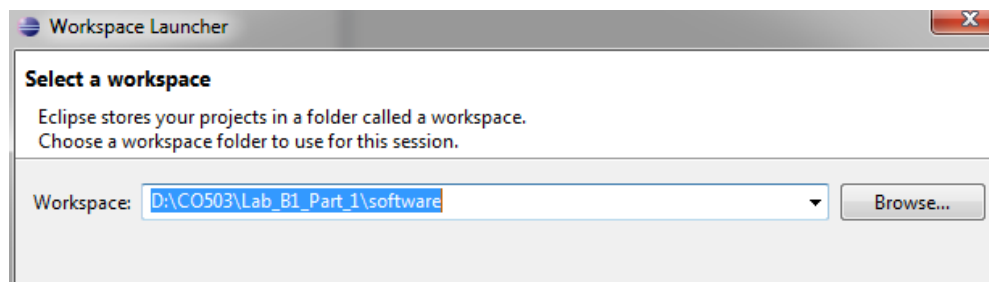
Flow Summary	
Flow Status	Successful - Wed Aug 10 09:03:48 2016
Quartus II 64-Bit Version	12.1 Build 177 11/07/2012 S3 Full Version
Revision Name	SoC
Top-level Entity Name	SoC
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	2,424 / 114,480 (2 %)
Total combinational functions	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.

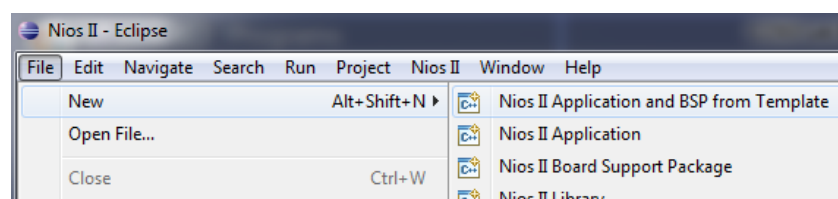
9. Expand *TimeQuest Timing Analyzer* and click on "*Multicorner Timing Analysis Summary*" from the table of contents. This shows the timing performances of the clock signals. Any negative slack values indicate the paths for the clock are too slow. Prior to compilation, you can manually set design constraints 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 on to 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 names *software* inside your project directory.



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

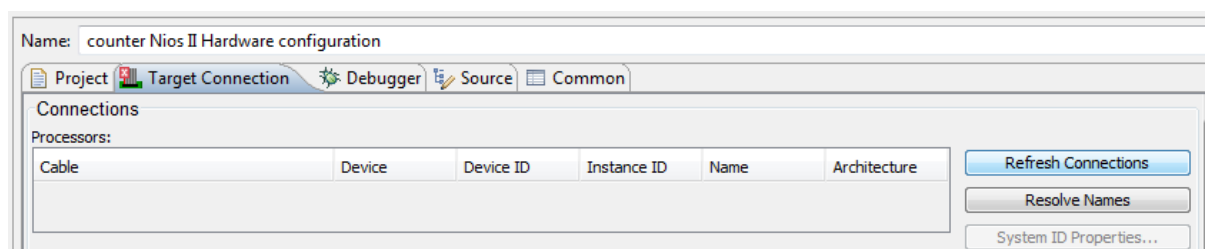


Select the target hardware by browsing for "*SoC.sopcinfo*" file in the project directory. Choose *cpu* as the CPU name from the dropdown list. Name the project as *counter*. Select "*Blank Project*" as the template and click "*Finish*". You will see two new projects are created: *counter* is the application 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 on the course web page (*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 *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. Ask an instructor for help on this.

Part 2: JPEG Encoder

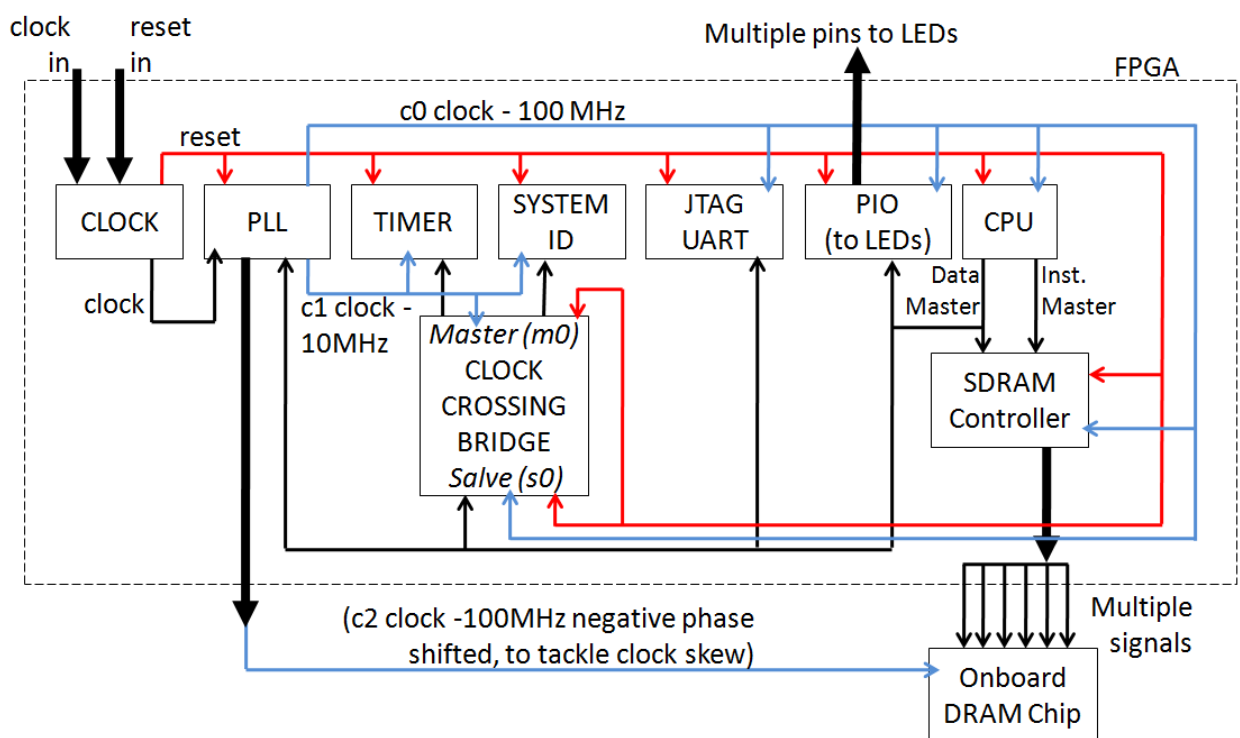
Now you should create a new SoC for JPEG image encoding, using onboard SDRAM as the main memory (instead of on-chip memory). Input and output files are accessed from the host computer through the JTAG USB cable. LEDs on the board should indicate the processing status in a suitable way (*encoding, finished,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



a) New hardware components to be used: *SDRAM Controller*, *Avalon ALTPLL*.

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 ALTPLL:
- | | |
|------------------|---|
| | Input frequency (inclko) = 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:
- | | |
|--------------------------|-----------------|
| <i>SDRAM_ADDR[12..0]</i> | - output |
| <i>SDRAM_BA[1..0]</i> | - output |
| <i>SDRAM_CAS_N</i> | - output |
| <i>SDRAM_CKE</i> | - output |
| <i>SDRAM_CS_N</i> | - output |
| <i>SDRAM_DQ[31..0]</i> | - bidirectional |
| <i>SDRAM_DQM[3..0]</i> | - output |
| <i>SDRAM_RAS_N</i> | - output |
| <i>SDRAM_WE_N</i> | - output |
| <i>SDRAM_CLK</i> | - output |

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

2. Software:

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.