ECE385
DIGITAL SYSTEMS LABORATORY
Introduction to NIOS II and Platform Designer (formerly Qsys)

Abstract and Goals

The goal of this lab is creating a NIOS II based system on the Altera Cyclone IV device. The NIOS II is an IP based 32-bit CPU which can programmed using a high-level language (in this class, we'll be using C). A typical use case scenario is to have the NIOS II be the system controller and handle tasks which do not need to be high performance (for example, user interface, data input and output) while an accelerator peripheral in the FPGA logic (designed using SystemVerilog) handles the high-performance operations.

The following will give you a walkthrough of the Platform Designer tool which is used to instantiate IP blocks (including the NIOS II). We will set up a minimal NIOS II device with an SDRAM (Synchronous Dynamic RAM) controller and a PIO (Parallel I/O) block to blink some LEDs using a C program running on the NIOS II to confirm it is working. You will then be asked to write a program which reads 8-bit numbers from the switches on the DE2 board and sums into an accumulator, displaying the output using the green LEDs via the NIOS II. This will involve instantiating another PIO block to read data from the switches and modifying the C program to input data, add, and display the data.

Goals (to demonstrate to your TA):
The TA will test the following functionality:
The green LEDs should always display the value of the accumulator in binary and the accumulator should be 0 on startup (all LEDs off). The accumulator should overflow at 255+1 to 0. (e.g. 255 + 1 → 0, 255 + 2 → 1, etc.)
Pressing the second to the left pushbutton at any time clears the accumulator to 0 and updates the display accordingly (turns all the LEDs off)
Pressing the left most pushbutton loads the number represented by the switches into the CPU, adding it to the accumulator. The 8 right-most switches are read as an 8-bit, unsigned, binary number with up being 1, down being 0.
Push buttons should only react once to a single actuation.

Be prepared to give answers to any of the italicized questions in this document from your TA when demoing. This is to ensure that you try to research what the settings do instead of simply trying to "make the picture look like your screen".

Hints:
Unit test the input and output. The output should already work, but make sure you can turn on and off every segment. If you have problems, check the schematic for the DE2, and make sure you are toggling the correct pins.
For this, and the rest of the class, you may use the C standard libraries (stdlib.h) or the C++ equivalents, this can save you a lot of work when coding in C.

<u>Set up the System Combining the FPGA with the Nios II Processor:</u>

Create a New Project:

- Start Quartus Prime.
- From the ***File*** menu select ***New Project Wizard***. Click *Next* to go through the intro screen, if it appears.
- The window in Figure 1 will appear. Fill in the fields from figure 1 (make sure there are no spaces in any of your entries). The program will ask you if it should create the specified directory if it does not exist; choose *yes*.
- Select ***Next*** on page 2 without adding any files.
- On page 3, select ***Cyclone IV E*** for the device family, make sure the second option under Target device is selected, and chose ***EP4CE115F29C7*** in the available devices list. See Figure 2.

  Click ***Next*** on page 4. Select *ModelSim-Altera* as the simulation tool name, and *SystemVerilog HDL* as the simulation format. See Figure 3. Click ***Finish*** on page 5.

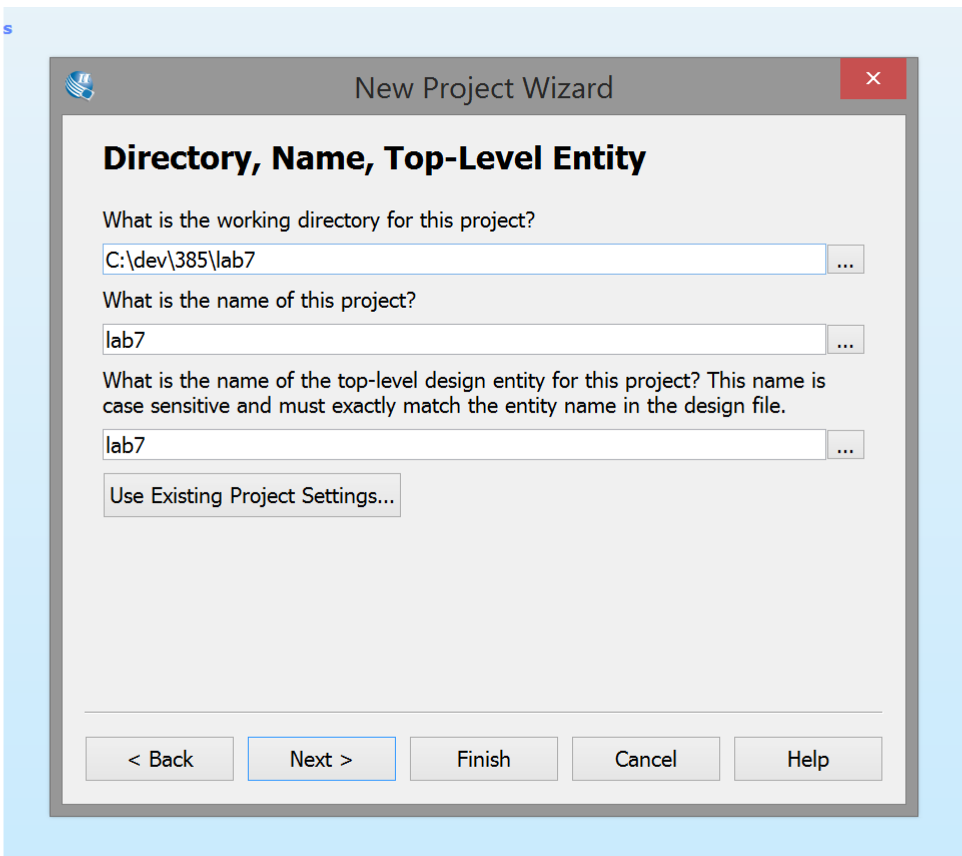


Figure 1

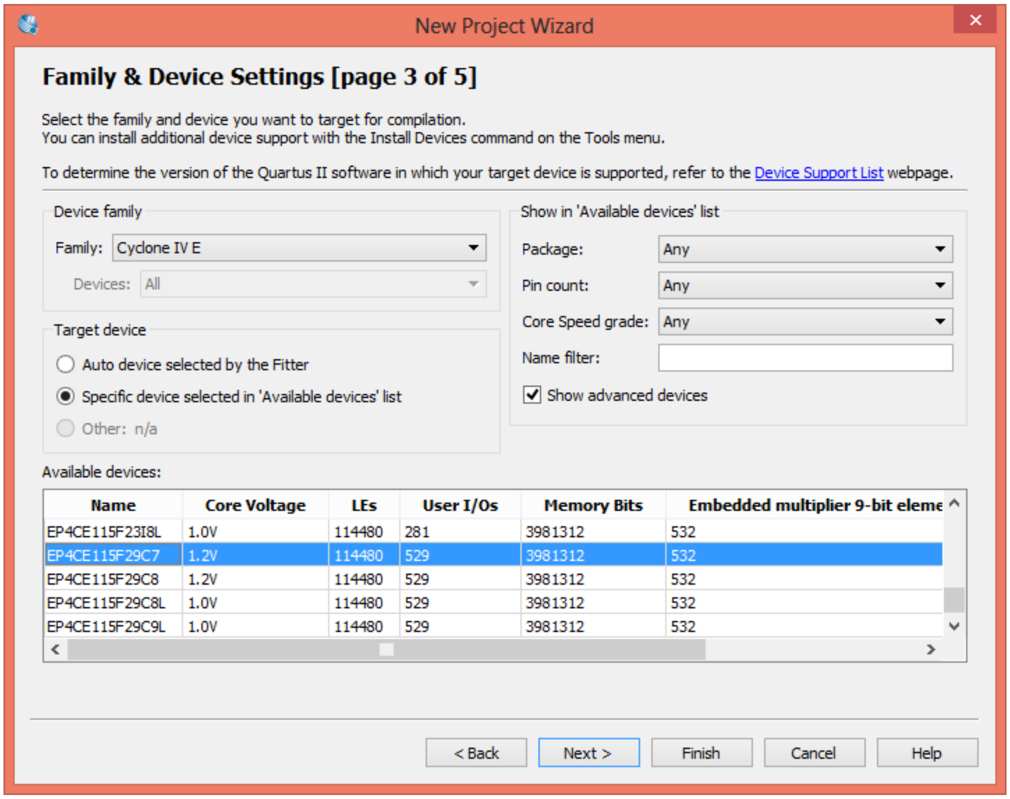


Figure 2

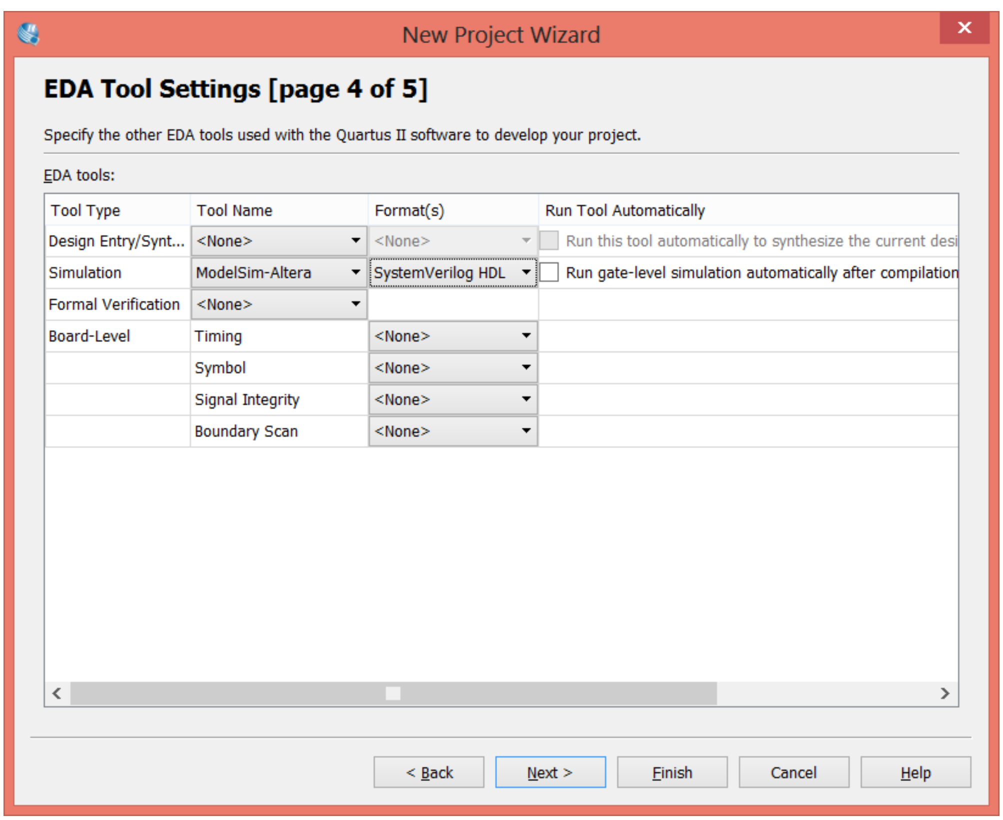


Figure 3

INQ. 4

Create the SoC with Qsys:

Now we can set up the SoC with Platform Designer (formerly Qsys). In the ***Tools*** menu, select ***Platform Designer*** to launch Platform Designer. Immediately save the Platform Designer file as **lab7_soc.qsys**. This is the name of the hardware block which will contain the CPU and the supporting hardware (peripherals, memory, etc). A window shown in Figure 4 should pop up. Here, you can see a predefined clock signal. If needed, the clock frequency can be modified in the ***Clock Settings*** tab, as shown in Figure 5. In this lab, we will work with the default 50 MHz clock. If the *Clocks* tab is not available by default, then you can find it by selecting ***View*** > ***Clocks*** from the main menu.

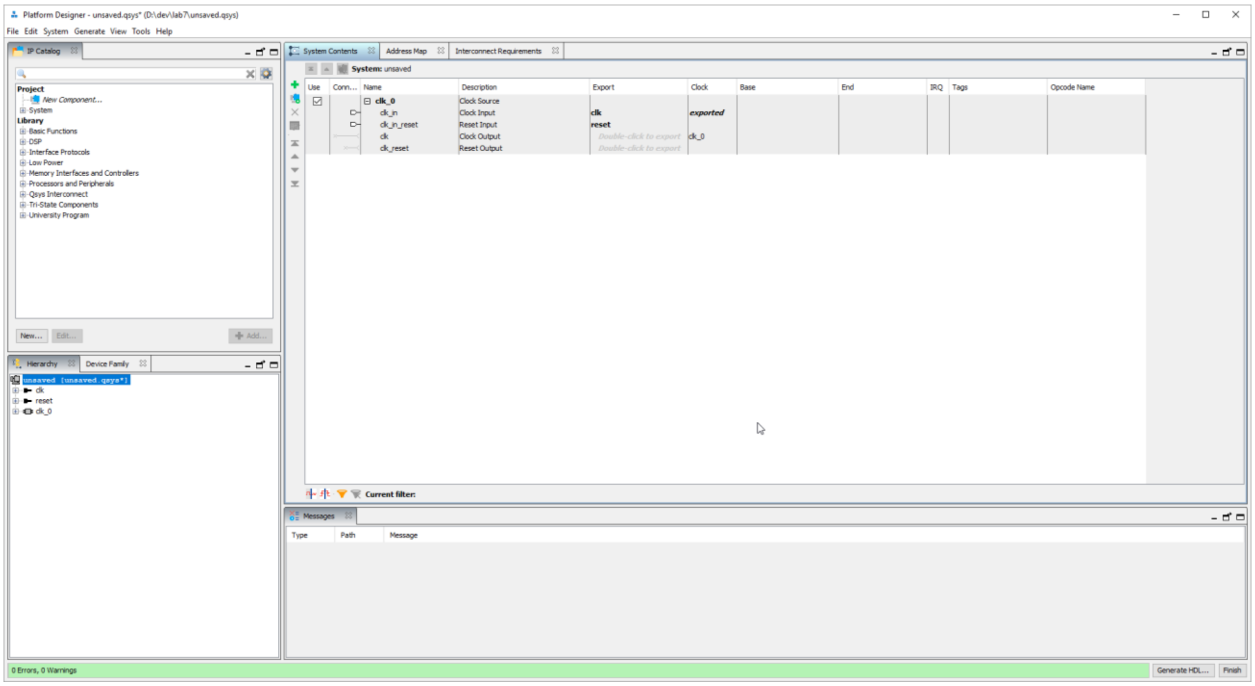


Figure 4

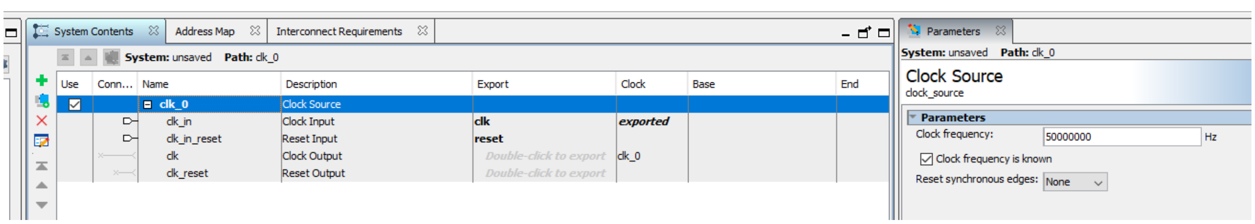


Figure 5

Next, specify the processor on the left side of the Qsys window by selecting ***Processors and Peripherals*** > ***Embedded Processors*** > ***Nios II Processor*** and clicking ***Add***. A window should pop up, as shown in Figure 6. In this window, select ***Nios II/e***, which is the economy version of the processor. Note that error messages regarding reset and exception vectors are shown on the bottom. This is because we haven't specified memory components in the system. Ignore the messages for now as we will provide the necessary information later. Click ***Finish***. Now we have included the Nios II processor, as shown in Figure 7. *What are the differences between the Nios II/e and Nios II/f CPUs?* e: economy; f: fast. The fast version has more features than economy version, but it is not free.

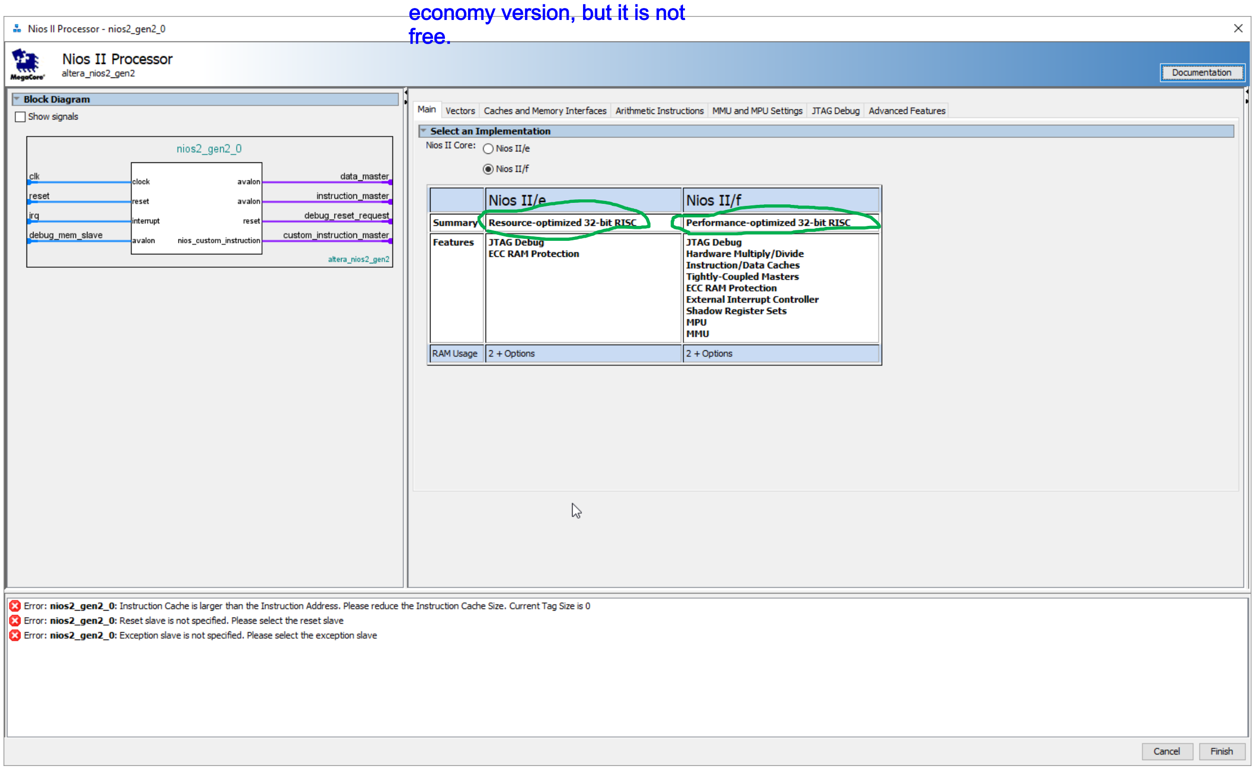


Figure 6

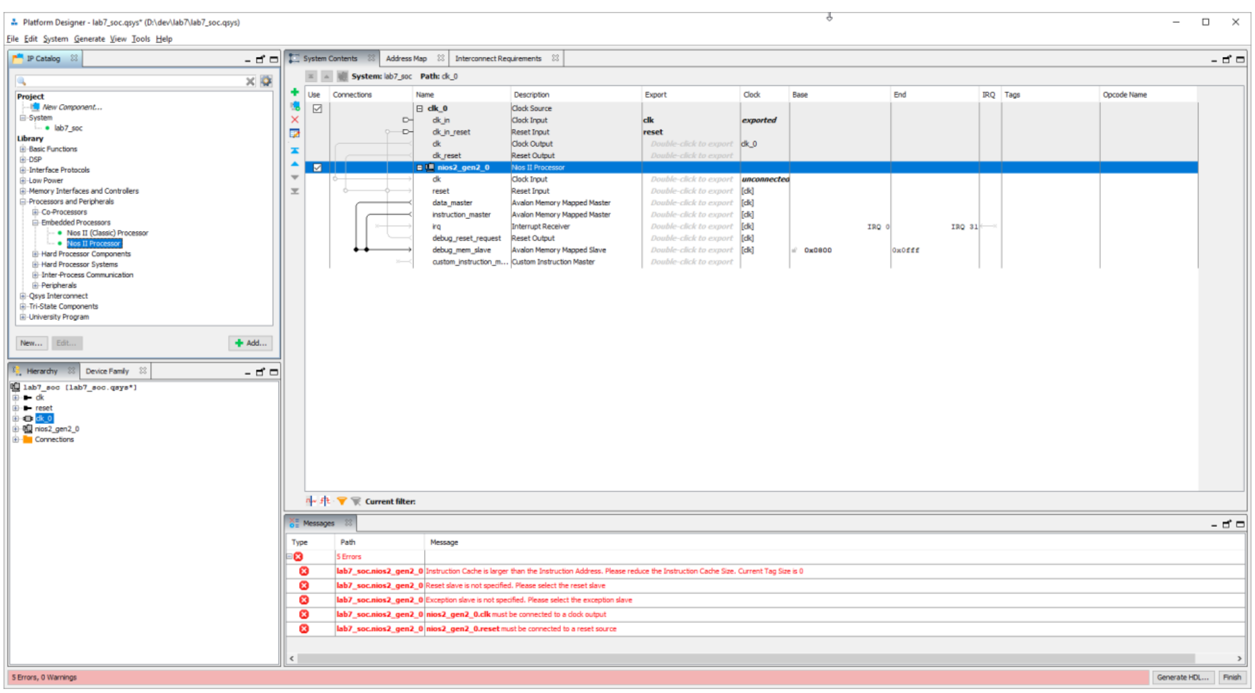


Figure 7

INQ. 6

Next, we're going to instantiate an on-chip memory block. On the left side of the Qsys window, select ***Basic Functions > On Chip Memory > On-Chip Memory (RAM or ROM) Intel FPGA IP***, and click ***Add***. A window as shown in Figure 8 should pop up. Choose Memory Type to be ***RAM (Writable)*** and Total memory size to be 16 bytes. For most designs, we will save valuable on-chip memory and instead execute NIOS II programs from the DRAM, however we are instantiating a small on-chip RAM as a placeholder block (which you may use for your final project if you decide to use on-chip memory) *What advantage might on-chip memory have for program execution?* Note that you can get the datasheet for any IP block by clicking on documentation (top right). Click ***Finish***.

decrease latency for critical/ frequently used data.

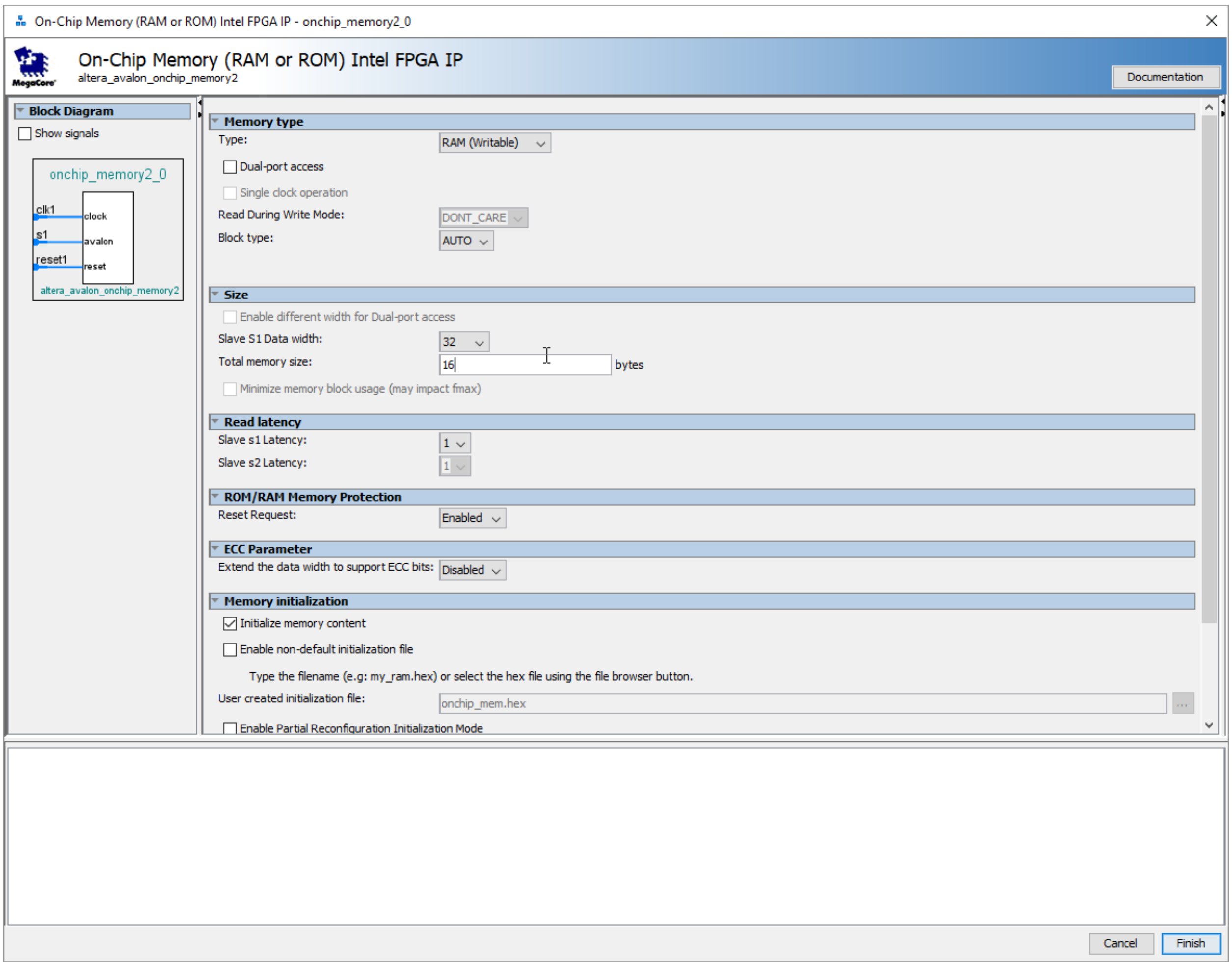


Figure 8

Next, we establish the connections between the Nios II processor and the on-chip memory. In the ***Connections*** column of the Qsys window, click on the empty circles to make a connection. Make sure you connect the NIOS II and on-chip memory clock and reset to the external clock and reset. *Note the bus connections coming from the NIOS II; is it a Von Neumann, "pure Harvard", or "modified Harvard" machine and why?*

modified Harvard: allows the contents of the instruction memory to be accessed as data. Since the instruction and data are connected as the same bus.

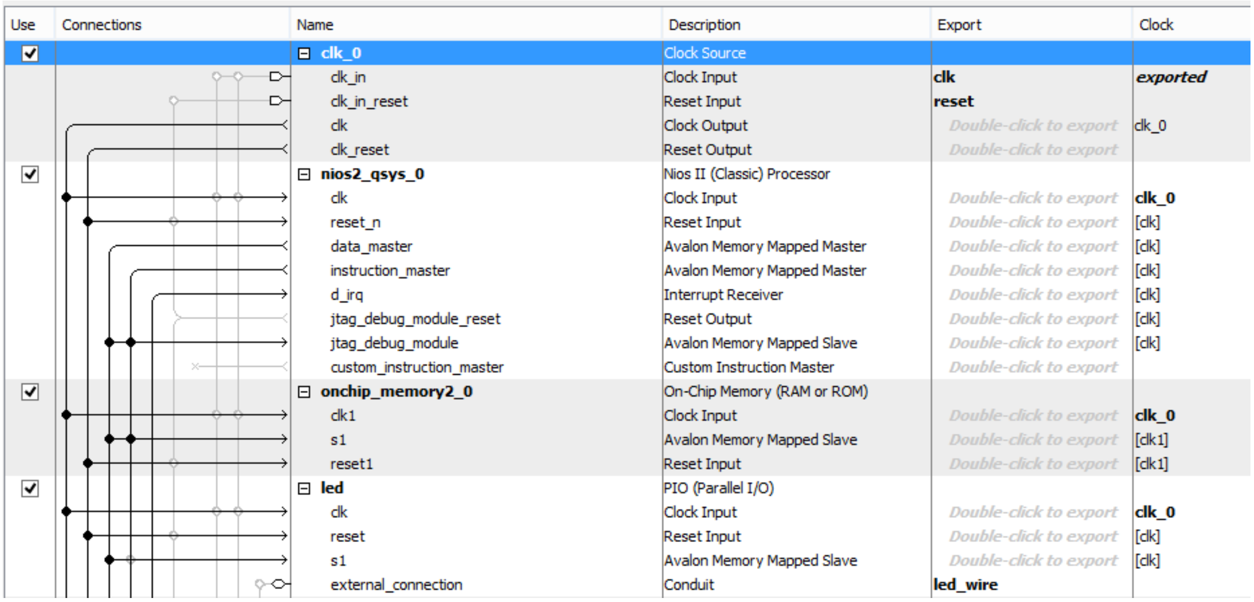


Figure 9

Next, we specify the input parallel I/O interface. This will be used to drive the LEDs and tell us that the system is working correctly. On the left side of the Qsys window, select ***Processors and Peripherals*** > ***Peripherals*** > ***PIO (Parallel I/O) Intel FPGA IP*** and click ***Add***. Specify the width of the port to be *8* bits. Choose the direction of the port to be ***Output***. Click ***Finish***.

Rename the **PIO** block **led**, so we can keep track of what it is for. Create all the bus connections as shown below between the PIO (led) peripheral and the NIOS II through the Avalon bus. Make the required connections as shown in Figure 9. Note that while the on-chip memory needs access to both the data and program bus, the led peripheral only needs access to the data bus. *Why might this be the case?*

because the led dont need to get the program. And we want to ensure that the data on the instruction bus would not affect the performance of the led.

Since the on-chip memory has limited storage capacity, we will use the off-chip SDRAM to store the software program. SDRAM cannot be interfaced to the bus directly, as it has a complex row/column addressing scheme and requires constant refreshing to retain data. We will use a SDRAM controller IP core to interface the SDRAM to the Avalon bus. *Why does SDRAM require constant refreshing?*

Dram uses a single transistor and a single capacitor to contain a bit, represented as a charge on the capacitor. The transistor is turned on to charge the capacitor, but the capacitor leaks off over a time typically measured in milliseconds. It must therefore be regularly read and re-written to ensure that the data is not lost. This is what is done by refresh. The data does not need to leave the chip, so quite a large row can be refreshed in one cycle. But every row must be refreshed with the refresh cycle time.

You will need to determine the following parameters to instantiate the SDRAM controller. Refer to the DE2-115 schematic and the ISSI IS42S16320D SDRAM (note there are two on the board) datasheet (PDF). Make sure you are looking at the correct part of the datasheet (the two chips each provide 16 bits)

SDRAM parameter	Short name	Parameter value (fill in from datasheet)
Data Width	[width]	32
# of Rows	[nrows]	13
# of Columns	[ncols]	10
# of Chip Selects	[ncs]	1
# of Banks	[nbanks]	4

INQ. 8

Note that there are two 32M*16 chips, so the total amount of memory should be 1Gbit (128 Mbytes), *make sure this is consistent with your above numbers; you will need to justify how you came up with 1 Gbit to your TA.* 2*32M*16/8 = 128Mbytes

On the left side of the Qsys window, select ***Memory Interfaces and Controllers*** > ***SDRAM*** > ***SDRAM Controller Intel FPGA IP*** and click ***Add***. A window should pop up, as shown in Figure 14. On the ***Memory Profile*** tab, set the ***Data Width*** to be ***[width]*** bits, ***Address Width*** to be ***[nrows]*** rows and **[ncols]** columns. Make sure **[ncs]** and **[nbanks]** are correct as well. In general, we would have to also research all the SDRAM timings from the datasheet; however this has been done for you. On the ***Timing*** tab, enter the numbers according to Figure 10. *What is the maximum theoretical transfer rate to the SDRAM according to the timings given?* Click ***Finish***. Rename the component as *sdram*. about 2 MByte/s

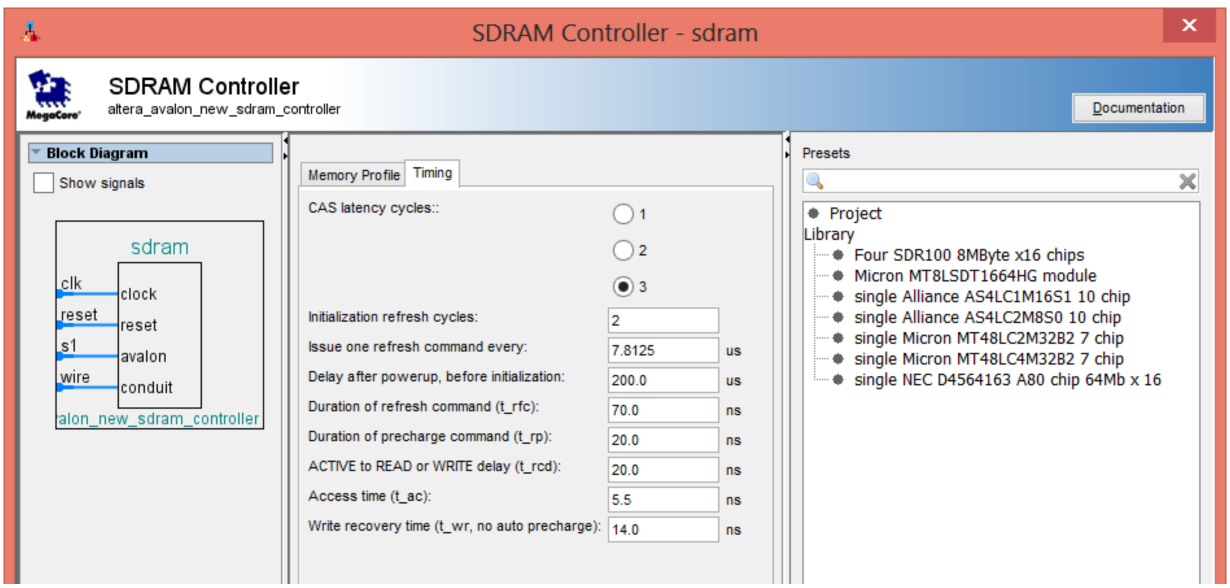


Figure 10

Next, we add a PLL component to provide the required clock signal for the SDRAM chip. This is because the SDRAM requires precise timings, and the PLL allows us to compensate for clock skew due to the board layout. The SDRAM also cannot be run too slowly (below 50 MHz). *Why might this be the case?* A block diagram for how we will use the SDRAM controller is shown in Figure 11.

The clock for the SDRAM chip (SDRAM clock) must be driven at the same frequency as the clock for the Avalon-MM interface on the SDRAM controller (controller clock). As in all synchronous designs, you must ensure that address, data, and control signals at the SDRAM pins are stable when a clock edge arrives

?

CAS Latency (CL) : CAS Latency (Column Access Strobe Latency), also known as “Access Time,” is the most important memory parameter and is the first of the series of numbers. It is the delay time between the moment a memory controller tells the memory module to access a particular memory column on a RAM memory module, and the moment the data from given array location is available on the module's output pins. In DDR SDRAM it is specified in clock cycles, while in asynchronous DRAM it is specified in nanoseconds.

RAS to CAS Delay (tRCD) : tRCD stands for row address to column address delay time. Inside the memory, the process of accessing the stored data is accomplished by first activating the row then the column where it is located. tRCD is the time required between the memory controller asserting a row address strobe (RAS), and then asserting a column address strobe (CAS) during the subsequent read or write command. The lesser this time, the better it is, as the data will be read sooner.

RAS Precharge (tRP) : Whenever a new row is to be activated for the purpose of accessing a data bit, a command called “Precharge” needs to be issued to close the already activated row. RAS Precharge time, tRP is the number of clock cycles needed to terminate access to an open row of memory, and open access to the next row.

Active to Precharge Delay (tRAS) : After an “Active” command is issued, another “Precharge” command cannot be issued until tRAS has elapsed. So, tRAS is the minimum number of clock cycles needed to access a certain row of data in the memory between the data request (Active) and the Precharge command. Basically, this parameter limits when the memory can start reading (or writing) a different row.

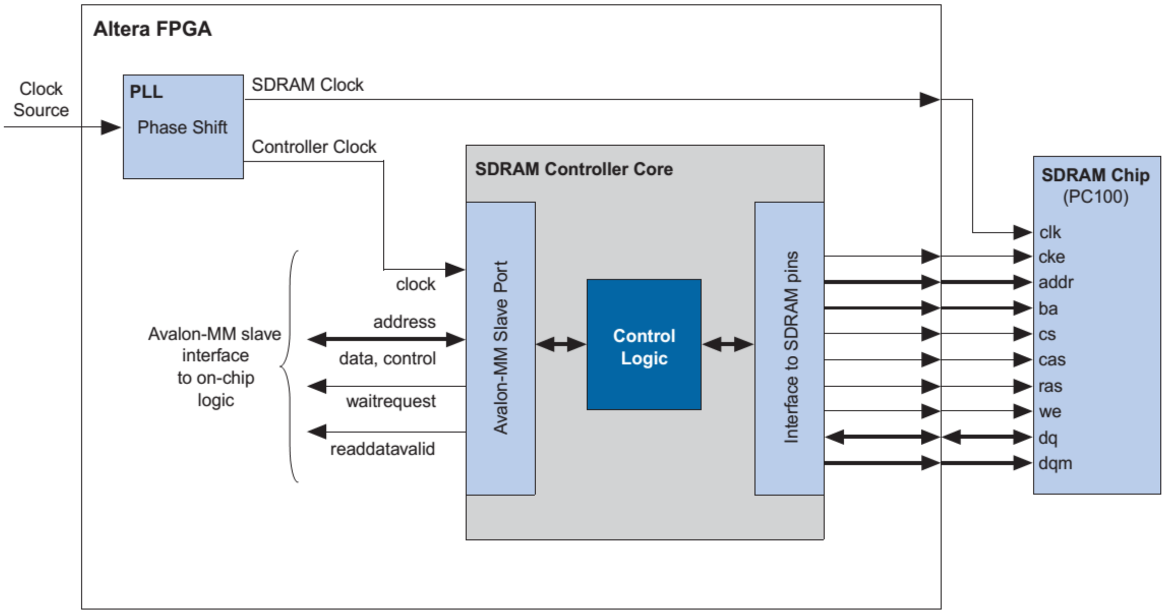


Figure 11

On the left side of the Qsys window, select ***Basic Functions*** > ***Clocks; PLLs and Resets*** > ***PLL*** > ***ALTPLL Intel FPGA IP*** and click ***Add***. A window should pop up as shown in Figure 12. Choose device speed to be ***7***, and the frequency of the ***inclk0*** input (input clock) to be ***50*** MHz. Click ***Next***.

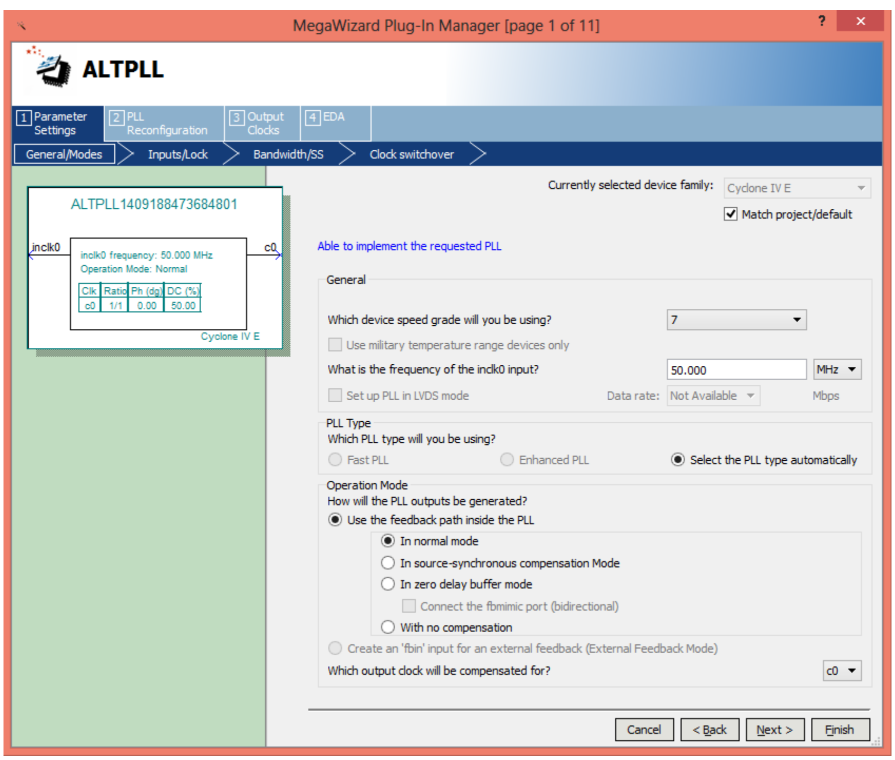


Figure 12

On page 2 (Inputs/Lock), deselect all the checked boxes because we do not need these additional ports (e.g. locked status) in this lab, as shown in Figure 17. Skip Bandwidth/SS and Clock Switchover and the entire [2] PLL Reconfiguration tab by clicking ***Next*** until you get to **[3] Output Clocks**.

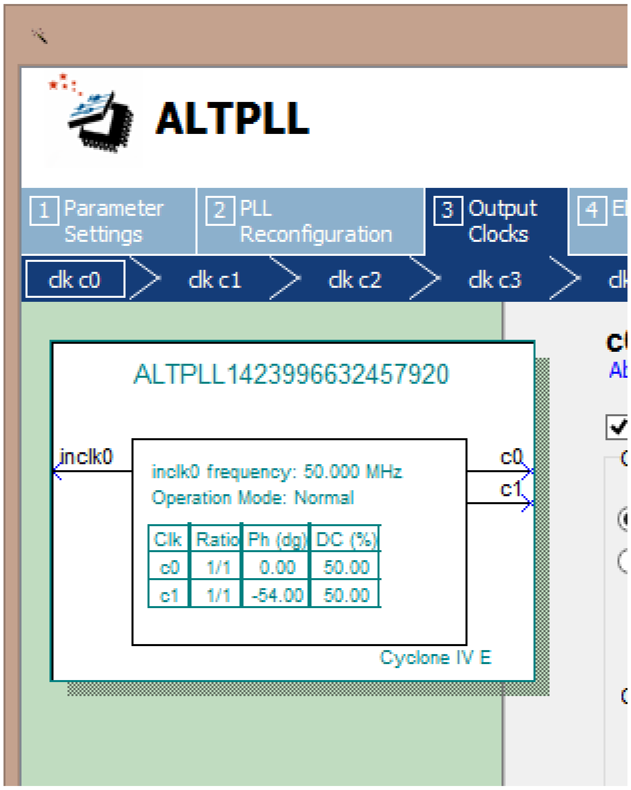


Figure 13

On page **[3] Output Clocks** for **clk c0**, make sure the actual clock frequency is 50 MHz, and set the clock phase shift to be ***0ns***. On the left, observe that the *Ph (dg)* (phase in degrees) is now *0*. This is the clock which goes to the SDRAM controller, and should be synchronous to the rest of the design, this is the *controller clock*, as shown in Figure 11.

You must now make a second clock, which goes out to the SDRAM chip itself, as recommended by Figure 11. Make another output by clicking ***clk c1***, and verify it has the same settings, except that the phase shift should be **-3ns**. This puts the clock going out to the SDRAM chip (***clk c1***) 3ns behind of the controller clock (***clk c0***). *Why do we need to do this? Hint, check Altera Embedded Peripheral IP datasheet under SDRAM controller.* The configuration for ***clk c1*** should look like Figure 14. Now complete the rest of the wizard by clicking ***Finish*** and the ***Finish*** again. Name your new module *sdram_pll*, to remind us what this is for. Verify that there are indeed two ports as shown in Figure 13, one with 0 degrees shift, going out to the controller, one with -54 degrees shift, going out to the SDRAM chip.

If you use a PLL, you must tune the PLL to introduce a clock phase shift so that SDRAM clock edges arrive after synchronous signals have stabilized

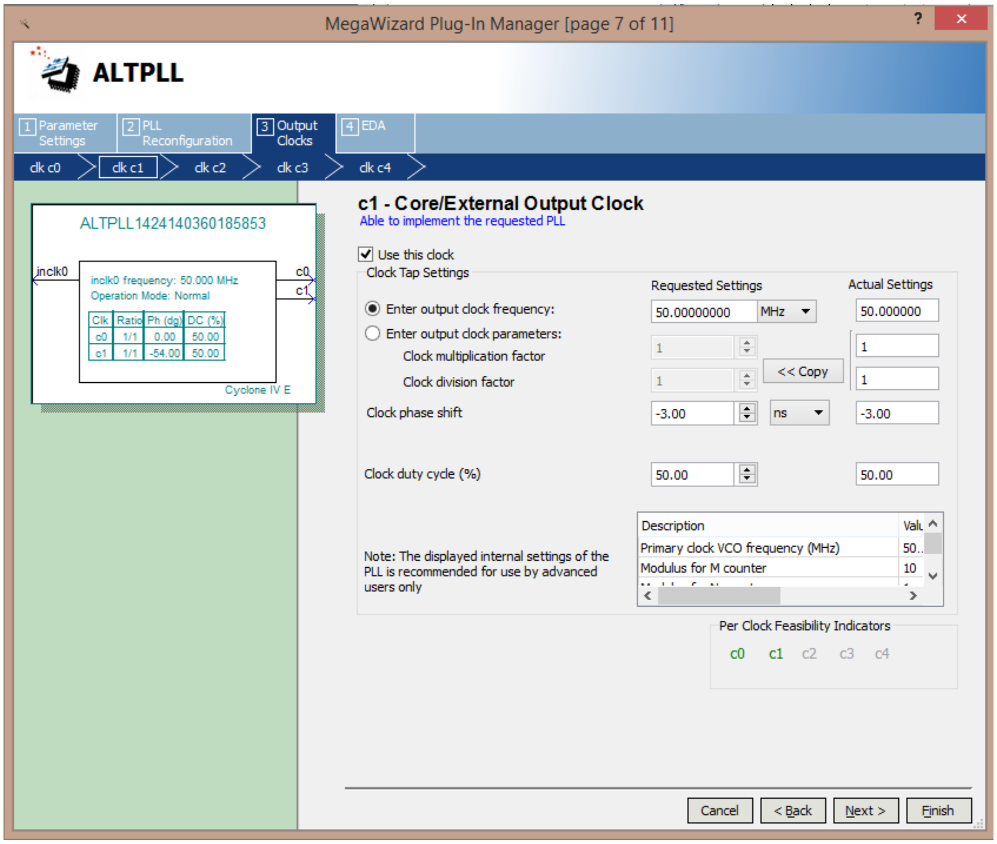


Figure 14

Next, we add a system ID checker to ensure the compatibility between hardware and software. This module gives us a serial number (we'll assign it to 0), which the software loader checks against when we start the software. This prevents us from loading software onto an FPGA which has an incompatible NIOS II configuration (or an FPGA without a NIOS II at all). For example, if we had added a new NIOS II peripheral and forgot the regenerate/reprogram the FPGA, this block would prevent us from trying to load software from the old configuration onto our incompatible NIOS II. On the left side of the Qsys window, select ***Basic Functions > Simulation; Debug and Verification > System ID Peripheral Intel FPGA IP*** and click ***Add***. Accept the default settings and click ***Finish***, as per Figure 15.

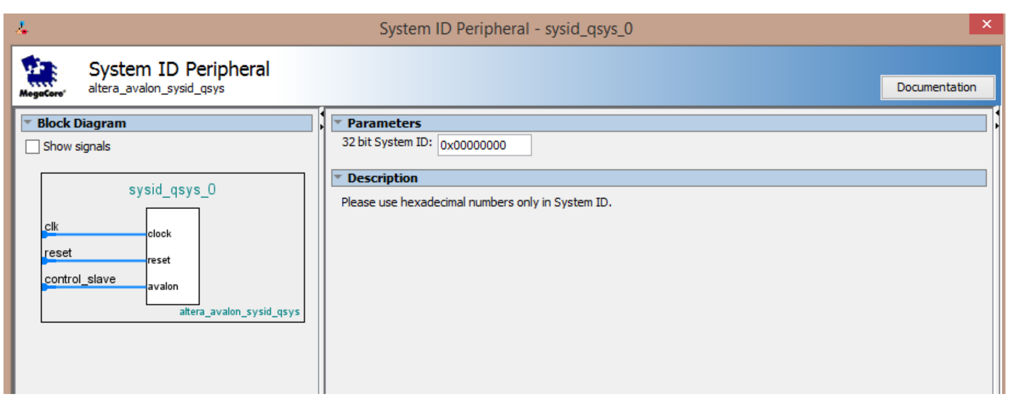


Figure 15

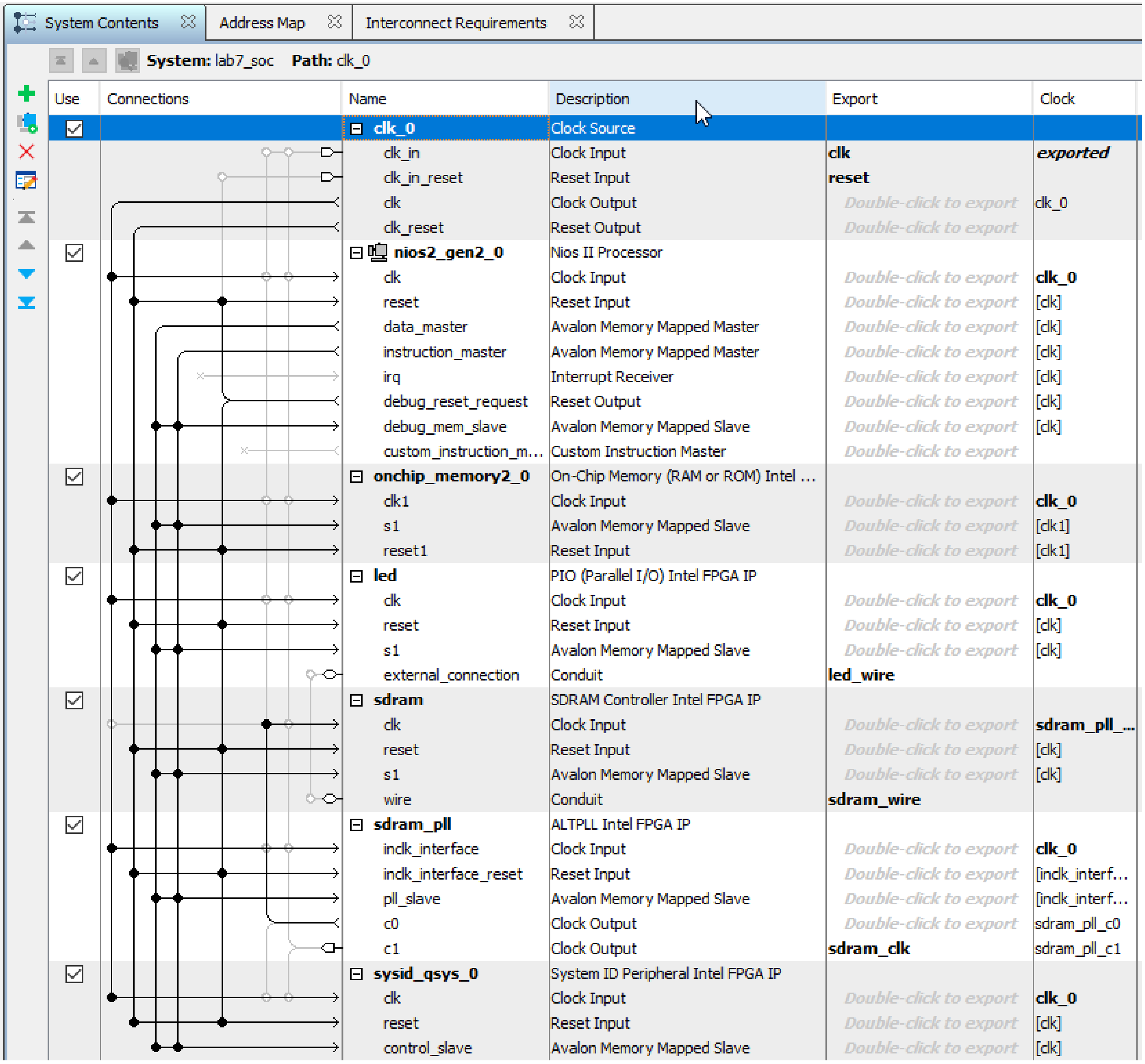


Figure 16

Next, we make the necessary connections for the newly added components, as shown in Figure 16. **Note that the *sdram* is connected to the *sdram_pll_c0* instead of the main clock**, as per Figure 11. Next, enable external access to the clock, reset, PIO, SDRAM, and the other clock out from the PLL (to the physical SDRAM chip). Do this by clicking on the ***Double-click to export*** in the ***Export*** column. Name the PIO(led) port ***led_wire***. Also, click on the ***Double-click to export*** on the right of the *sdram* wire. Type in the name ***sdram_wire***. This "breaks out" the connections to the rest of the world for the SDRAM controller (to the SDRAM chips, which are external to the FPGA itself), the PIO block, the reset, and the second port of the PLL. Export the second output of the PLL (***sdram_pll_c1)*** as ***sdram*_clk**. When you are done, make sure following connections are exported:

sdram_wire	***led_wire***	***reset***	***clk***	***sdram_clk***

These are the connections your SystemVerilog top-level needs to connect to the appropriate external hardware.

Finally, we assign memory addresses to each component. This can be done automatically in Qsys, but we want to make sure that the on-chip memory gets the base address of 0x0000_0000. Click on the base address of on-chip memory and enter ***0x0000_0000***, and then click on the lock symbol on the left to make it fixed. On the menu, click on ***System > Assign Base Addresses*** to have Qsys automatically assign memory addresses to the other peripherals. The resulting Qsys should be similar (but maybe not identical) to Figure 16. Note that these addresses will be used in the software program later in this tutorial.

After we have the memory set up, we can assign the reset and exception vectors for the Nios II processor. Right click on *nios2_qsys_0* and click ***Edit***. For both reset and exception vectors, choose *sdram.s1*, as shown in Figure 17. Do not change default settings for offsets. Click ***Finish***. *What address does the NIOS II start execution from? Why do we do this step after assigning the addresses?*

0x00, because it can only be done after all the addresses are assigned. otherwise the exception vector might be wrong

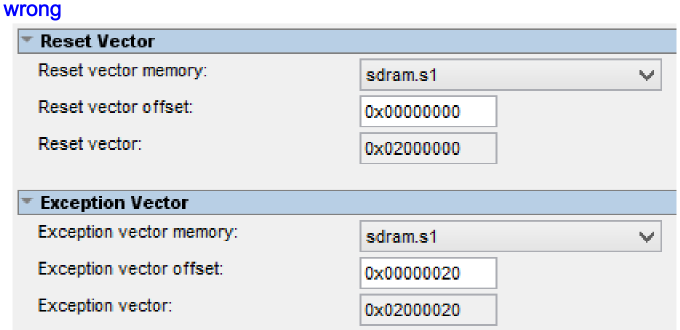


Figure 17

Now, we have specified all the necessary components. Save the system as lab7_soc. Then, go to the ***Generate > Generate HDL*** from the main menu. Specify the settings as shown in Figure 18, and then click ***Generate***.

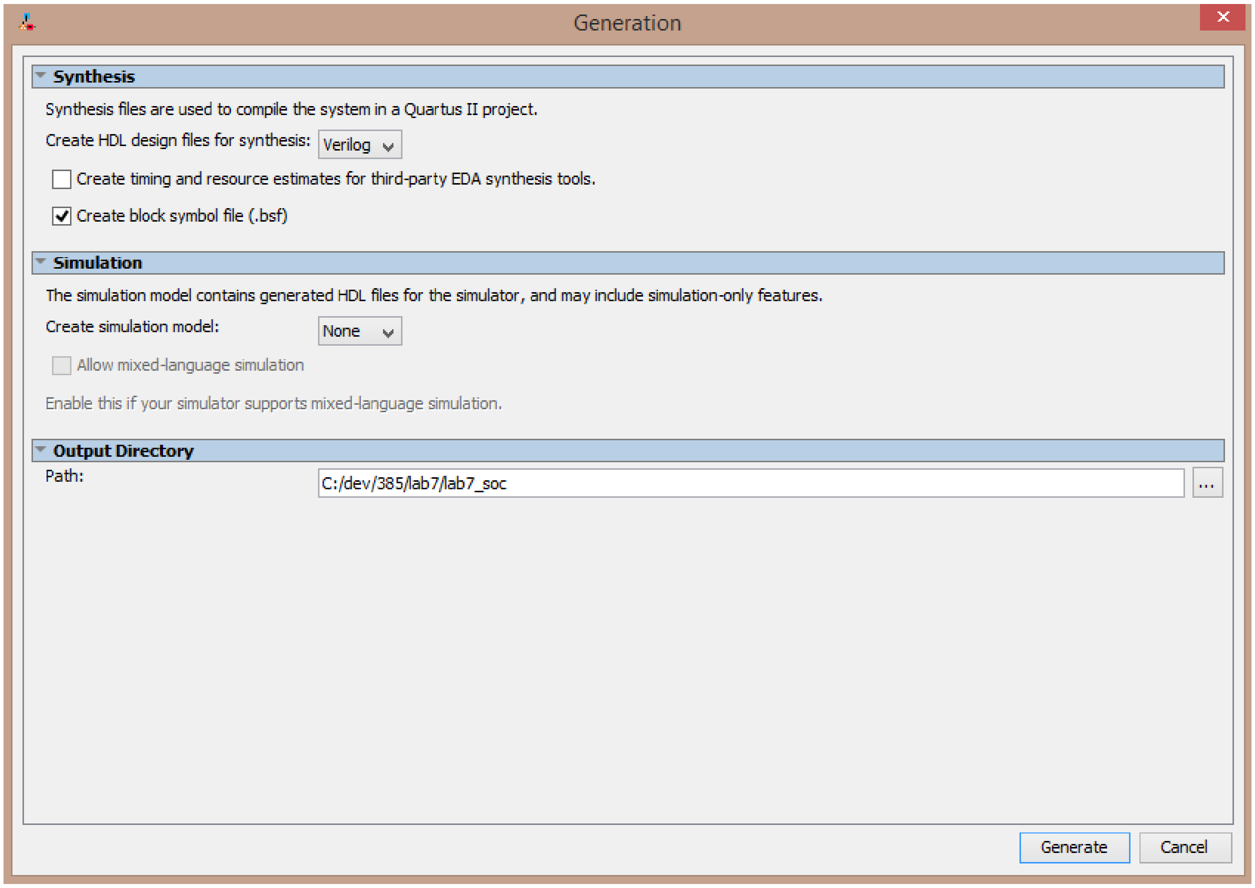


Figure 18

After generation is complete, we can find the synthesized NIOS II system-on-chip component in lab7\lab7_soc\synthesis\lab7_soc.v. It is synthesized in Verilog, but the module interface is compatible with the SystemVerilog design we are going to create in this project.

Integration of the Nios II System into a Quartus Prime Project:

In Quartus II, go to the ***Files*** tab on the left. Right click on ***Files*** and select ***Add/Remove Files in Project***. Add lab7_soc\synthesis\lab7_soc.qip into the project, which gives the required information for Quartus II to include the system we created. Then, add the lab7 top-level SystemVerilog file *lab7.sv* (given on the course website) as a new file. An instance of the ***lab7*** top level module is now created, as shown in Figure 19.

```
module lab7(     input           CLOCK_50,
                 input   [1:0]   KEY,
                 output  [7:0]   LEDG,
                 output  [12:0]  DRAM_ADDR,
                 output  [1:0]   DRAM_BA,
                 output          DRAM_CAS_N,
                 output          DRAM_CKE,
                 output          DRAM_CS_N,
                 inout   [31:0]  DRAM_DQ,
                 output   [3:0]  DRAM_DQM,
                 output          DRAM_RAS_N,
                 output          DRAM_WE_N,
                 output          DRAM_CLK

                );
```

Figure 19

Open the generated Verilog file for the Nios II system, lab7/lab7_soc/synthesis/lab7_soc.v to take a look at the SoC module you created. This will have ports and names based on what you exported from Qsys. Make sure the names are consistent with what is expected in the top-level (lab7.sv) as in Figure 20. At this point, if your names are consistent, you should be able to synthesize your design without any errors (but not necessarily implement, since we have no pin assignments yet).

```
`timescale 1 ps / 1 ps
module lab7_soc (
        input  wire        clk_clk,           //        clk.clk
        output wire [7:0]  led_wire_export,   //   led_wire.export
        input  wire        reset_reset_n,     //      reset.reset_n
        output wire        sdram_clk_clk,     //  sdram_clk.clk
        output wire [12:0] sdram_wire_addr,   // sdram_wire.addr
        output wire [1:0]  sdram_wire_ba,     //           .ba
        output wire        sdram_wire_cas_n,  //           .cas_n
        output wire        sdram_wire_cke,    //           .cke
        output wire        sdram_wire_cs_n,   //           .cs_n
        inout  wire [31:0] sdram_wire_dq,     //           .dq
        output wire [3:0]  sdram_wire_dqm,    //           .dqm
        output wire        sdram_wire_ras_n,  //           .ras_n
        output wire        sdram_wire_we_n    //           .we_n
    );
```

Figure 20

Do the necessary pin assignments as follows. (Hint: Download DE2-115.qsf from the course website. In Quartus, go to ***Assignments*** > ***Import Assignments*** to load the default DE2 pin assignment settings. Make modifications as needed. Note that these names are from our top-level SystemVerilog file, even though for this lab, our top level doesn't do more than wire the NIOS II module (lab7_soc) to the outside world.

INQ. 16

Port Name	Location	Comments
CLOCK_50	PIN_Y2	50 MHz Clock from the on-board oscillators
KEY[3]	PIN_R24	On-board push button
KEY[2]	PIN_N21	On-board push button
KEY[0]	PIN_M23	On-board push button
LEDG[7]	PIN_G21	On-board LED
LEDG[6]	PIN_G22	On-board LED
LEDG[5]	PIN_G20	On-board LED
LEDG[4]	PIN_H21	On-board LED
LEDG[3]	PIN_E24	On-board LED
LEDG[2]	PIN_E25	On-board LED
LEDG[1]	PIN_E22	On-board LED
LEDG[0]	PIN_E21	On-board LED
DRAM_ADDR[12]	PIN_Y7	On-board SDRAM
DRAM_ADDR[11]	PIN_AA5	On-board SDRAM
DRAM_ADDR[10]	PIN_R5	On-board SDRAM
DRAM_ADDR[9]	PIN_Y6	On-board SDRAM
DRAM_ADDR[8]	PIN_Y5	On-board SDRAM
DRAM_ADDR[7]	PIN_AA7	On-board SDRAM
DRAM_ADDR[6]	PIN_W7	On-board SDRAM
DRAM_ADDR[5]	PIN_W8	On-board SDRAM
DRAM_ADDR[4]	PIN_V5	On-board SDRAM
DRAM_ADDR[3]	PIN_P1	On-board SDRAM
DRAM_ADDR[2]	PIN_U8	On-board SDRAM
DRAM_ADDR[1]	PIN_V8	On-board SDRAM
DRAM_ADDR[0]	PIN_R6	On-board SDRAM
DRAM_BA[1]	PIN_R4	On-board SDRAM
DRAM_BA[0]	PIN_U7	On-board SDRAM
DRAM_CAS_N	PIN_V7	On-board SDRAM
DRAM_CKE	PIN_AA6	On-board SDRAM
DRAM_CLK	PIN_AE5	On-board SDRAM
DRAM_CS_N	PIN_T4	On-board SDRAM
DRAM_DQ[31]	PIN_U1	On-board SDRAM
DRAM_DQ[30]	PIN_U4	On-board SDRAM
DRAM_DQ[29]	PIN_T3	On-board SDRAM
DRAM_DQ[28]	PIN_R3	On-board SDRAM
DRAM_DQ[27]	PIN_R2	On-board SDRAM
DRAM_DQ[26]	PIN_R1	On-board SDRAM
DRAM_DQ[25]	PIN_R7	On-board SDRAM
DRAM_DQ[24]	PIN_U5	On-board SDRAM
DRAM_DQ[23]	PIN_L7	On-board SDRAM
DRAM_DQ[22]	PIN_M7	On-board SDRAM
DRAM_DQ[21]	PIN_M4	On-board SDRAM
DRAM_DQ[20]	PIN_N4	On-board SDRAM
DRAM_DQ[19]	PIN_N3	On-board SDRAM
DRAM_DQ[18]	PIN_P2	On-board SDRAM
DRAM_DQ[17]	PIN_L8	On-board SDRAM
DRAM_DQ[16]	PIN_M8	On-board SDRAM
DRAM_DQ[15]	PIN_AC2	On-board SDRAM
DRAM_DQ[14]	PIN_AB3	On-board SDRAM
DRAM_DQ[13]	PIN_AC1	On-board SDRAM
DRAM_DQ[12]	PIN_AB2	On-board SDRAM
DRAM_DQ[11]	PIN_AA3	On-board SDRAM

DRAM_DQ[10]	PIN_AB1	On-board SDRAM
DRAM_DQ[9]	PIN_Y4	On-board SDRAM
DRAM_DQ[8]	PIN_Y3	On-board SDRAM
DRAM_DQ[7]	PIN_U3	On-board SDRAM
DRAM_DQ[6]	PIN_V1	On-board SDRAM
DRAM_DQ[5]	PIN_V2	On-board SDRAM
DRAM_DQ[4]	PIN_V3	On-board SDRAM
DRAM_DQ[3]	PIN_W1	On-board SDRAM
DRAM_DQ[2]	PIN_V4	On-board SDRAM
DRAM_DQ[1]	PIN_W2	On-board SDRAM
DRAM_DQ[0]	PIN_W3	On-board SDRAM
DRAM_DQM[3]	PIN_N8	On-board SDRAM
DRAM_DQM[2]	PIN_K8	On-board SDRAM
DRAM_DQM[1]	PIN_W4	On-board SDRAM
DRAM_DQM[0]	PIN_U2	On-board SDRAM
DRAM_RAS_N	PIN_U6	On-board SDRAM
DRAM_WE_N	PIN_V6	On-board SDRAM

For timing analysis purposes, add the timing constraint file lab7.sdc (given on the course website). The lines in the file describe the main clock and the SDRAM clock, and the external SDRAM clock. Also, the maximum input and output pin delays are specified and checked against. You may need to modify the file to account for your different signal names. ***If you add additional inputs/outputs (which you will need to) you must add additional constraints to this file. Any unconstrained paths in your final demo will cause you to lose points, since there is no guarantee your design will work.***

Finally, compile the project. In the ***Programmer***, find the .sof file in lab7/output_files. Program the circuit onto the FPGA.

Software Setup:

In Quartus II, go to ***Tools*** > ***Nios II Software Build Tools for Eclipse*** to launch the software development environment.

The Eclipse window will show up, as in Figure 21.

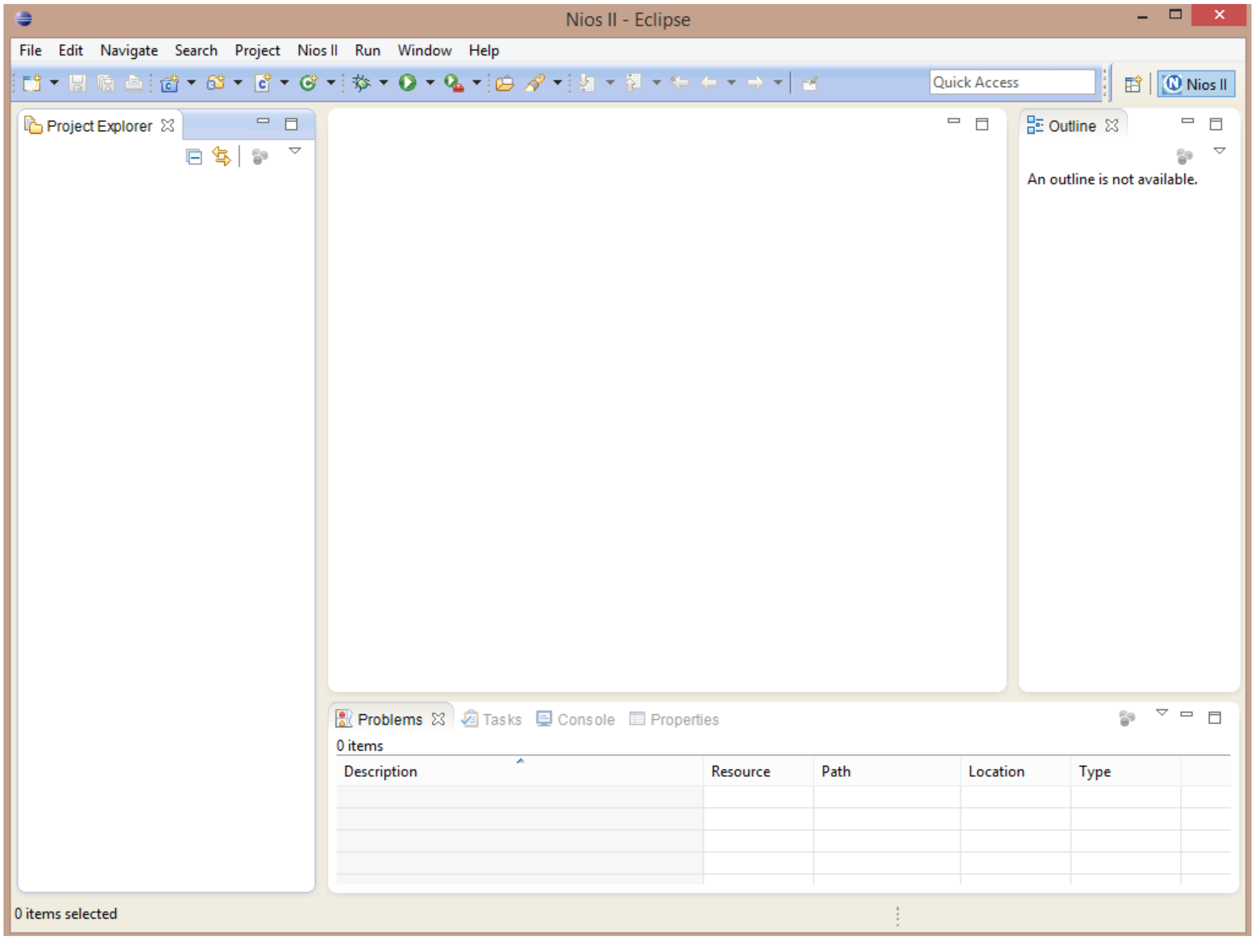


Figure 21

Eclipse works by having different “perspectives”, which represent UI layouts for different tasks common to software development. If the default perspective is not Nios II, set perspective to Nios II by going to ***Menu*** > ***Window***> ***Open Perspective*** > ***Other*** > ***Nios II***.

Create a new software program by clicking on ***File*** > ***New…*** > ***Nios II Application and BSP from Template***. A window should pop up, as shown in Figure 26. Set the SOPC Information File name to be lab7\lab7_soc.sopcinfo. The CPU name will be automatically determined. Select “Blank Project” from the available project templates on the lower left corner. Enter Project name as lab7_app. Click ***Finish***.

Two projects, *lab7_app* and *lab7_app_bsp* are generated. The bsp project contains the hardware information needed in the Makefile to compile the program. For example, the bsp contains the linker script, which tells the linker where to put various segments in memory as according to your address map. Click on the *lab7_app* project and create a new file named *main.c* and copy the contents from the *main.c* as supplied from the website, as shown in Figure 22.

main.c

```c
1  // Main.c - makes LEDG0 on DE2-115 board blink if NIOS II is set up correctly
4
5  int main()
6  {
7      int i = 0;
8      volatile unsigned int *LED_PIO = (unsigned int*)0x20; //make a pointer to acc
9
10     *LED_PIO = 0; //clear all LEDs
11     while ( (1+1) != 3) //infinite loop
12     {
13         for (i = 0; i < 100000; i++); //software delay
14         *LED_PIO |= 0x1; //set LSB
15         for (i = 0; i < 100000; i++); //software delay
16         *LED_PIO &= ~0x1; //clear LSB
17     }
18     return 1; //never gets here
19 }
20
```

Figure 22

At the beginning of the program, fill in the actual address assigned by Qsys to the assignment for pointer *LED_PIO. The address information can be found in the Qsys window. *You must be able to explain what each line of this (very short) program does to your TA. Specifically, you must be able to explain what the volatile keyword does (line 8), and how the set and clear functions work by working out an example on paper (lines 13 and 16).*

volatile Indicates that a variable can be changed by a background routine.

Right click on *lab7_app_bsp* project and select ***Nios II > Generate BSP.*** Then right click on the same project and, go to ***Nios II > BSP Editor***. A window should pop up, as shown in Figure 23. Make sure your settings are the same as shown in the figure. In particular, the *exception_stack_memory_region_name* and the *interrupt_stack_memory_region_name* should be set to *sdram*.

INQ. 20

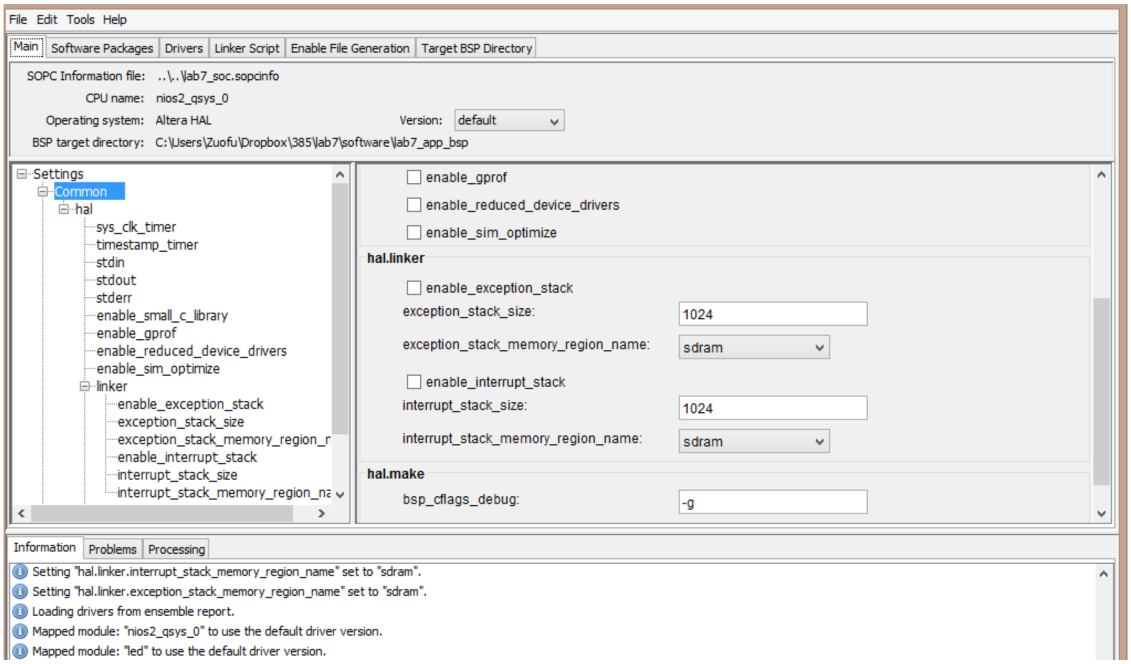


Figure 23

Finally, we have to setup the linker script. The linker script tells the linker which addresses to place the various segments of compiled code. Go to the ***Linker Script*** tab. Make sure all the linker regions and memory devices are set to *sdram*, as shown in Figure 24. Click ***Generate***.

Look at the various segment (.bss, .heap, .rodata, .rwdata, .stack, .text), what does each section mean? Give an example of C code which places data into each segment, e.g. the code:

```
const int my_constant[4] = {1, 2, 3, 4}
```

will place 1, 2, 3, 4 into the .rodata segment.

Text [edit]

The **code segment**, also known as a **text segment** or simply as **text**, is where a portion of an object file or the corresponding section of the program's virtual address space that contains executable instructions is stored and is generally read-only and fixed size.

Data [edit]

The *.data* segment contains any global or static variables which have a pre-defined value and can be modified. That is any variables that are not defined within a function (and thus can be accessed from anywhere) or are defined in a function but are defined as *static* so they retain their address across subsequent calls. Examples, in C, include:

```
int val = 3;
char string[] = "Hello World";
```

The values for these variables are initially stored within the read-only memory (typically within *.text*) and are copied into the *.data* segment during the start-up routine of the program.

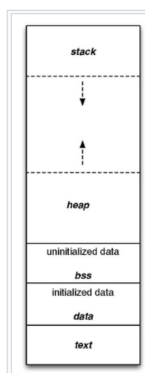
Note that in the above example, if these variables had been declared from within a function, they would default to being stored in the local stack frame.

BSS [edit]

The BSS segment, also known as *uninitialized data*, is usually adjacent to the data segment. The BSS segment contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code. For instance, a variable defined as `static int i;` would be contained in the BSS segment.

Heap [edit]

The heap area commonly begins at the end of the .bss and .data segments and grows to larger addresses from there. The heap area is managed by malloc, calloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single "heap area" is not required to fulfill the contract of malloc/calloc/realloc/free; they may also be implemented using mmap/munmap to reserve/unreserve potentially non-contiguous regions of virtual memory into the process' virtual address space). The heap area is shared by all threads, shared libraries, and dynamically loaded modules in a process.



This shows the typical layout of a simple computer's program memory with the text, various data, and stack and heap sections.

Stack [edit]

Main article: Call stack

The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. A "stack pointer" register tracks the top of the stack; it is adjusted each time a value is "pushed" onto the stack. The set of values pushed for one function call is termed a "stack frame". A stack frame consists at minimum of a return address. Automatic variables are also allocated on the stack.

The stack area traditionally adjoined the heap area and they grew towards each other; when the stack pointer met the heap pointer, free memory was exhausted. With large address spaces and virtual memory techniques they tend to be placed more freely, but they still typically grow in a converging direction. On the standard PC x86 architecture the stack grows toward address zero, meaning that more recent items, deeper in the call chain, are at numerically lower addresses and closer to the heap. On some other architectures it grows the opposite direction.

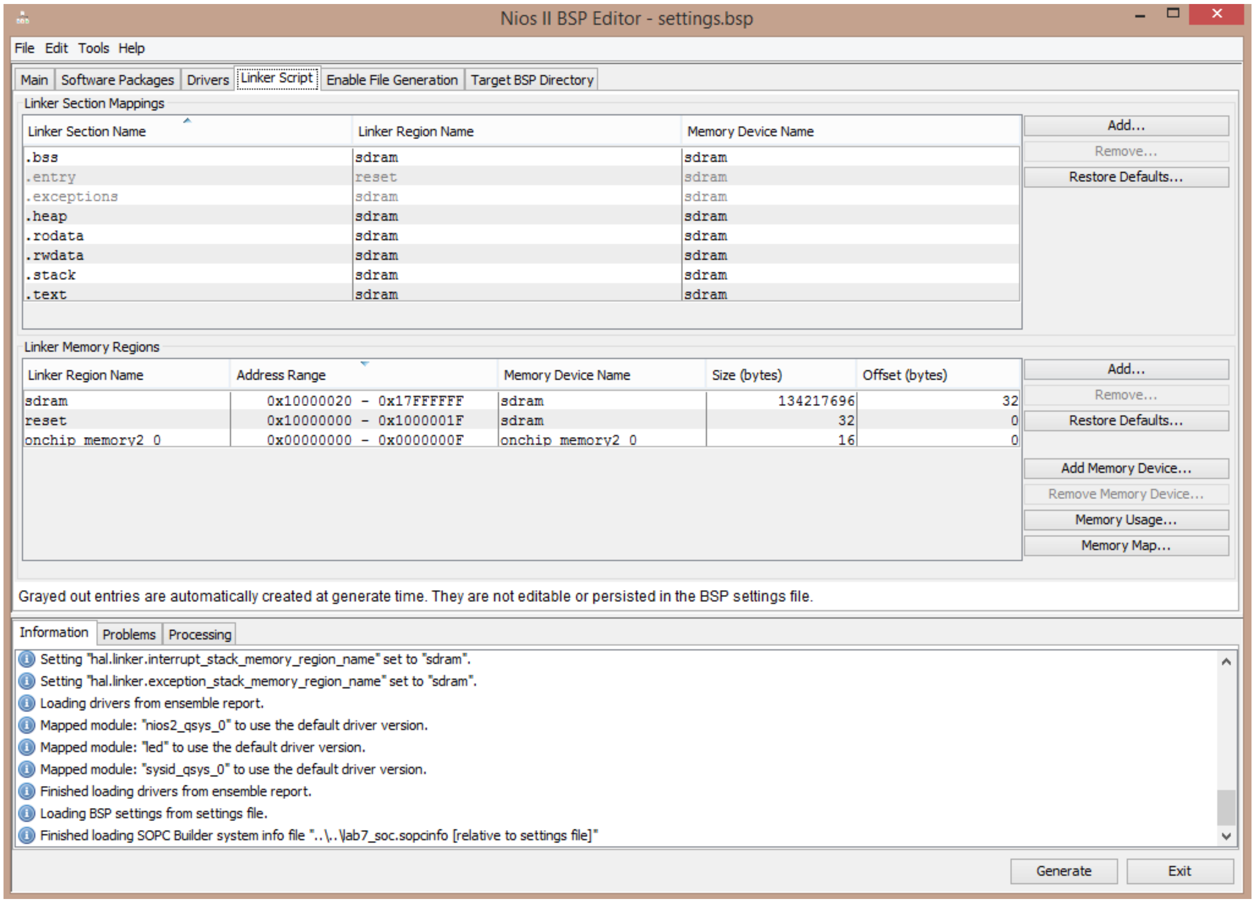


Figure 24

Now, we are ready to compile the program. Go to ***Project > Build All*** to compile the program.

> **<u>IMPORTANT</u>**:
> Whenever the hardware part is changed, it needs to be compiled in Quartus and programmed on the FPGA. If the programmer fails to load the .sof file on the FPGA, it's likely because the software is occupying the USB Blaster port. Simply stop the program or restart the FPGA board (and reprogram the FPGA) if this is the case.
>
> On the software side, make sure to right click on *lab7_app_bsp* and select ***Nios II > Generate BSP*** so the latest hardware information is included in the Makefile. Then, compile the program again (***Build All***). The System ID peripheral should report an errorif you try to load software onto an incompatible hardware platform, but in either case compatibility errors occur if you fail to maintain software-hardware consistency!

Running the program Nios II:

In Eclipse, click on ***Run*** > ***Run Configurations***… to open up a dialog window, as shown in Figure 25.

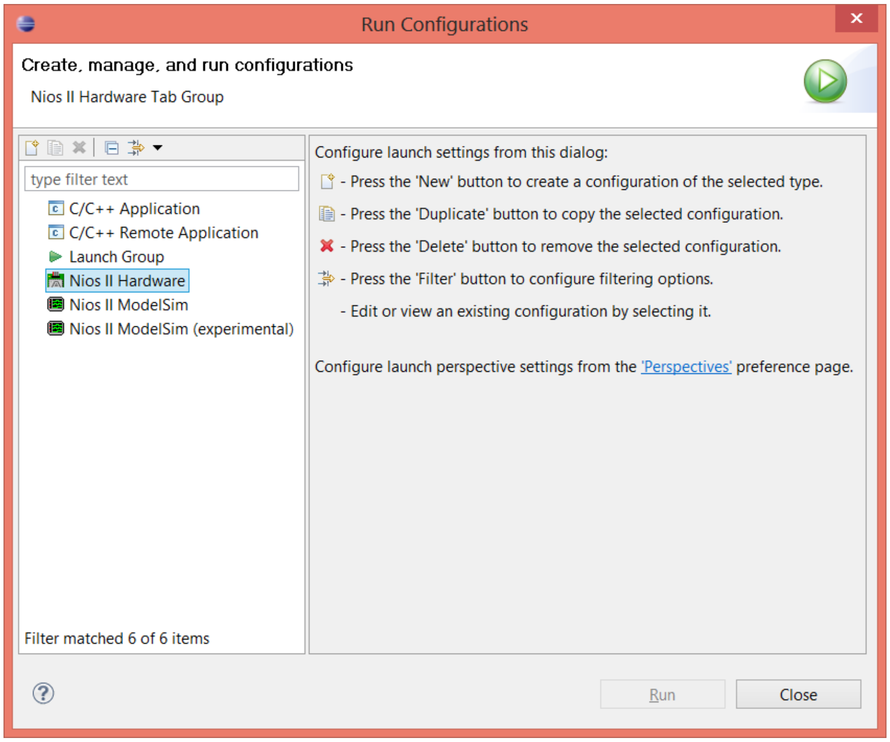


Figure 25

Right click on ***Nios II Hardware*** and select ***New***. Type in the name of the configuration as *DE2*. In the ***Project*** tab, select Project name to be *lab7_app*. The corresponding ELF file should automatically be set up. See Figure 26. The ELF file is the compiled software file that is downloaded into the system memory and run on Nios II, equivalent to an ".exe" in Windows.

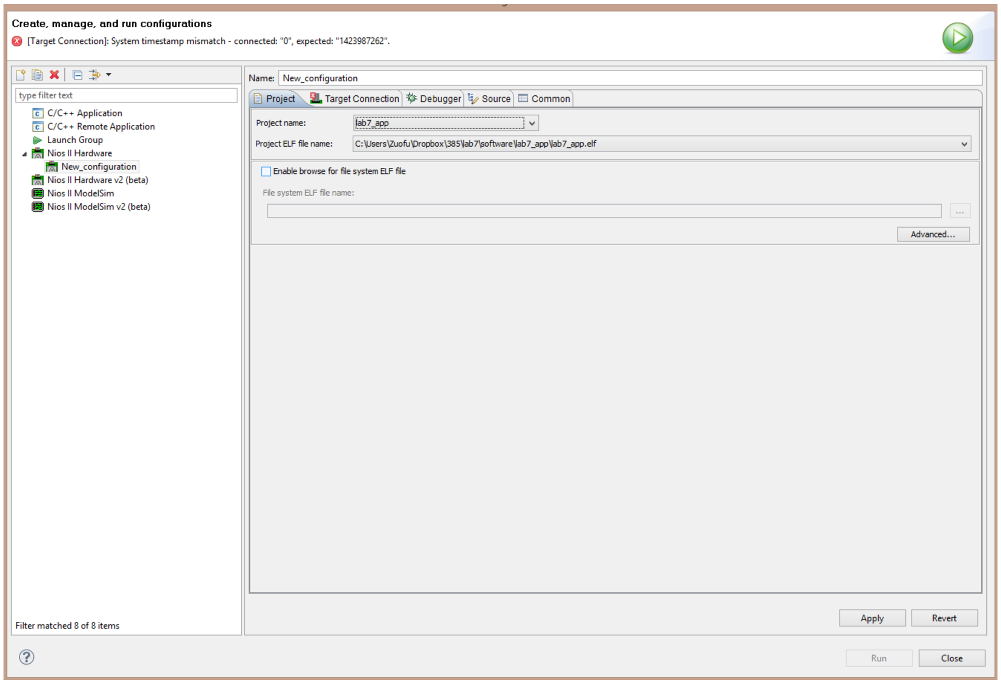


Figure 26

Go to the ***Target Connection*** tab. Click on Refresh Connections on the right. Make sure the *Processors* is listed as *USB-Blaster on localhost (in Figure 27)*. If an error message "*Connected system ID hash not found on target at expected base address*" appears, something is wrong with your NIOS II system, your computer cannot connect to the "sysid" block that you created in Qsys. Check to make the FPGA is programmed and that all the pin assignments are correct and click System ID Properties to refresh.

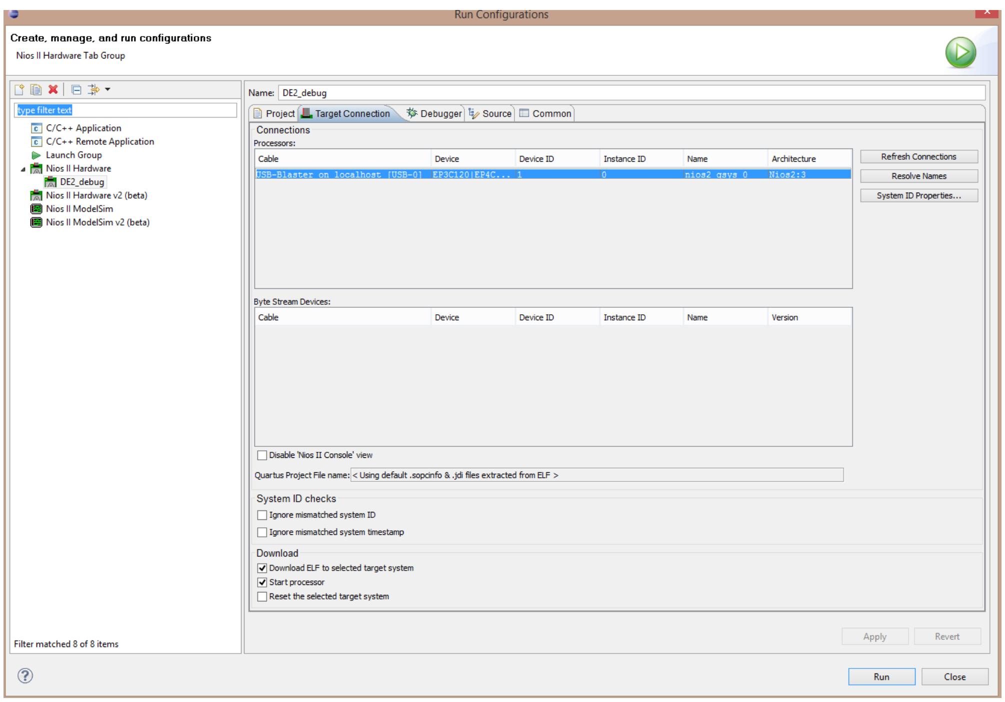


Figure 27

Click ***Run*** to run the program. Your LEDG0 should start blinking on your board as soon as the binary has been transmitted to the NIOS II CPU. Note that to do the actual lab assignment; you must add another PIO block as input corresponding to the switches. Do not forget you must export the PIO module from your SoC, modify the top-level SystemVerilog, re-assign memory addresses, assign the correct pins, and set the correct timing constraints. The following file will also be useful: Altera Embedded Peripheral IP datasheet, specifically the chapter on PIO.