

Lab 4: Implementation of a Memory-Mapped Slave Peripheral

Introduction

This lab introduces a lot of new material, and begins the re-integration of assembly language programming experience from ECE243 or a similar course. First, you will learn a brand new tool: the Intel Platform Designer (formerly Qsys) Tool. You will use it to generate a hardware system containing a Nios II processor and other peripherals. If you learnt about the ARM CPU in ECE243, do not worry. The Nios II CPU you will use in this course is very similar to the ARM CPU.

Systems created using the Qsys tool create Verilog code that can be synthesized for an FPGA. When running this on a real FPGA, the Monitor program is used to communicate with the CPU on the FPGA. However, for this year, we will only be limited to simulating such systems.

The ultimate goal of this lab will be create a full system, comprising of a Nios II CPU, some memory and a floating-point multiplier unit, all connected together using the Avalon interconnect (or bus). Code running on the Nios II CPU will be used to control the floating point unit and perform calculations. Figure 1 shows a block diagram of the final system.

NOTE: The “Intel Platform Designer” tool was previously called the “Qsys System Integration”. For convenience, we will still use “Qsys” to refer to this tool throughout the document.

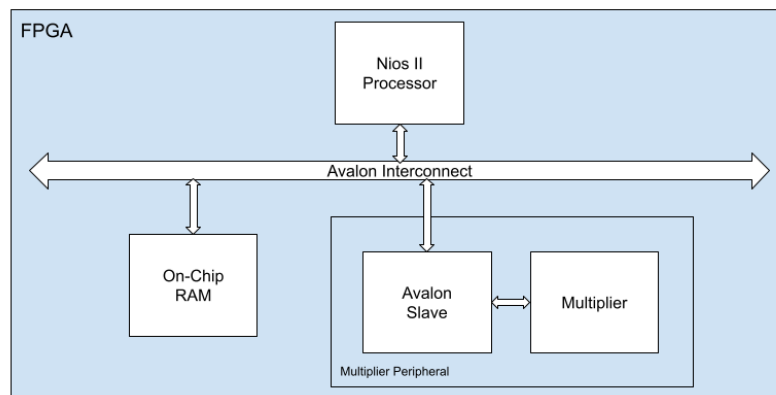


Figure 1: Top-level hardware block diagram for this lab

Part 1: Basic Qsys System

In this Part, you will be creating a simple system using Qsys, which uses the NIOS II CPU to read some switches and write that value to some LEDs. This is to familiarize you with using Qsys for a simple task. To start, open the Quartus project given to you in the **part1** folder of the lab starter kit. This project includes one top level module for the DE1-SoC board. Launch Qsys by selecting **Tools > Platform Designer**, as shown in Figure 2. Then we will need to create the following systems and peripherals:

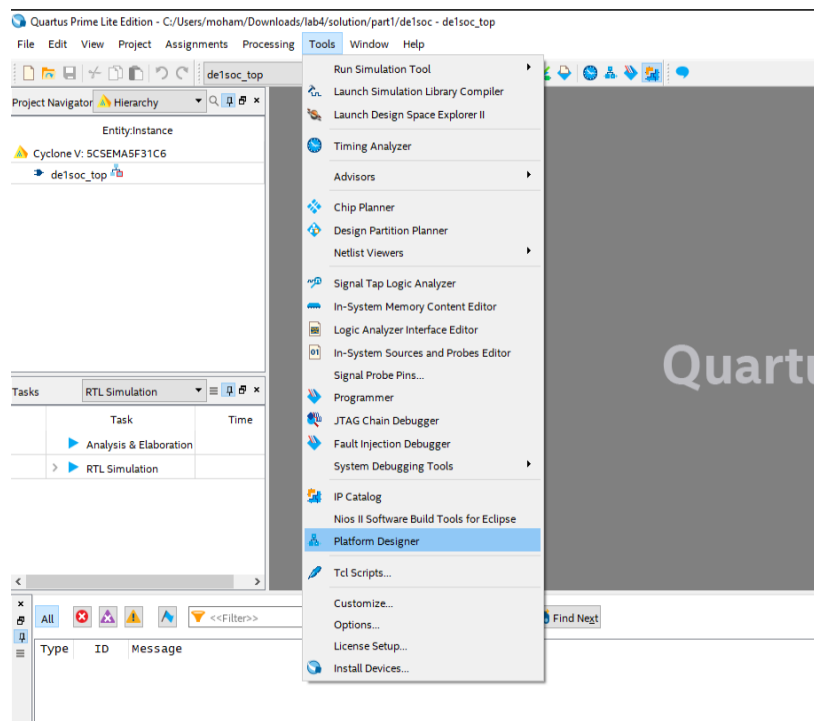


Figure 2: Launching Qsys

- A Nios II/e Processor.
- A 16KB On-Chip Memory to hold program code and data.
- A Parallel IO Peripheral, configured as inputs, to read 8 switches, namely `SW[7:0]`.
- Another Parallel IO, configured as output, to drive the red LEDs, namely `LEDR[7:0]`.

First, we will need to instantiate the Nios II/e processor. Start by clicking the green plus button to add a new instance, type the phrase `Nios` in the search bar, then select the **Nios II Processor**, as shown in Figure 3. In the configuration window that appears after, make sure to pick the type of Nios II implementation as **Nios II/e**, as shown in Figure 4, before clicking **Finish**.

As any other processor, our Nios II requires a memory to hold its instructions and data. For this purpose, we will create an on-chip RAM. Instantiate the RAM just like you did the processor, then configure it as shown in Figure 5. The RAM size is **16KB**, i.e., 16,384 bytes. Notice that the RAM only has one slave interface which will be shared by both the instruction and data accesses from the processor. Based on what was taught in class, can you determine whether this CPU uses the Von Neumann or Harvard architectures? Once we have the processor and the RAM, we need to make two instances of a component called Parallel I/O (PIO). This component acts as a bridge between the processor system bus, and an external connection. We will use one instance to connect to the switches, and the other to connect to the LEDs. Create and instantiate both PIOs, make sure to configure the switches PIO as **Input**, and the LEDs PIO as **output**. Right click the components you instantiated and select **rename** to give them a proper name. Then under the **Export** column give them exactly the same name as shown in Figure 6, this name will later become an input or output port for the Nios system.

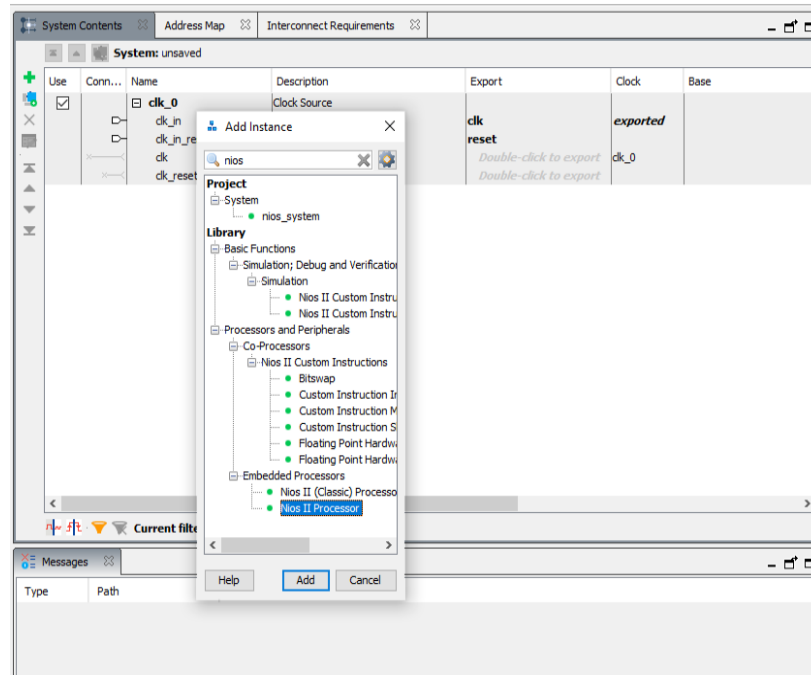


Figure 3: Instantiation of Qsys components

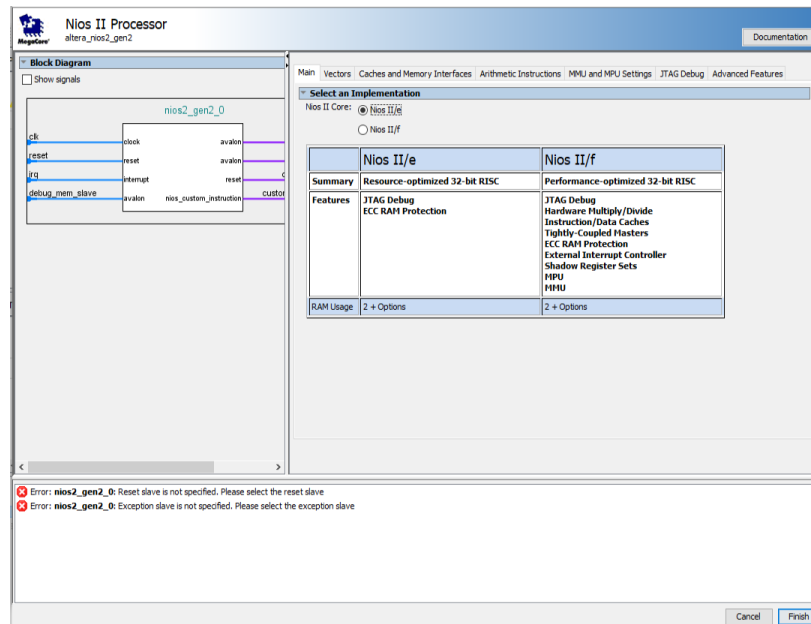


Figure 4: Configuring Nios II

Figure 5: Configuring On-Chip RAM

NOTE: Make sure to use the names and export names as ‘switches’ and ‘leds’. Otherwise, they will not be recognized by the automarker.

After the entire system has been instantiated, you should now connect all the components together properly. You can do so by clicking the dot points on the left, as shown in Figure 7.

NOTE: Be extra careful with this step. The connections must exactly match those in the figure, or your system will not work as expected. If that happens, you will no choice but to start all over.

By now, you should still have errors in your system. Those are caused by the reset vector and the exception vector of the Nios system not being assigned to any memory or address space, as well as the address map of the entire system not being properly assigned. First, open the configuration of the Nios II processor, under the **Vectors** tab configure the **Reset Vector Memory** to `onchip_memory2.0.s1`, then do the same for **Exception Vector Memory**. Once done, select **System > Assign Base Addresses** to assign non-conflicting addresses for all your peripherals. By now you should have no errors, and your address map should look like Figure 8.

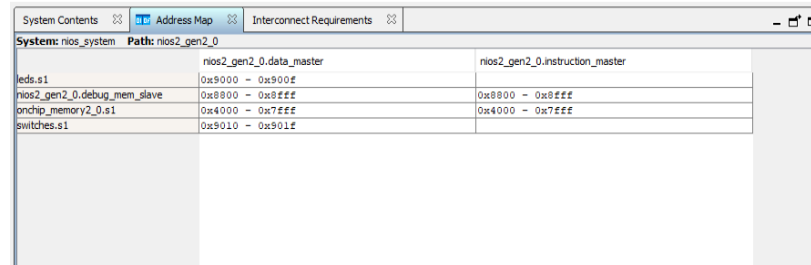
Now that we are done, save your system, make sure to use the name `nios_system` and use the default location inside your project directory. Then generate the HDL files by selecting **Generate > Generate HDL**. Disable the HDL file generation for synthesis, only enable it for simulation, as shown in Figure 9. Next select

Name	Description	Export	Clock
clk_0	Clock Source		
clk_in	Clock Input	clk	exported
clk_in_reset	Reset Input	reset	
clk	Clock Output	<i>Double-click to export</i>	clk_0
clk_reset	Reset Output	<i>Double-click to export</i>	
nios2_gen2_0	Nios II Processor		
clk	Clock Input	<i>Double-click to export</i>	clk_0
reset	Reset Input	<i>Double-click to export</i>	[clk]
data_master	Avalon Memory Mapped Master	<i>Double-click to export</i>	[clk]
instruction_master	Avalon Memory Mapped Master	<i>Double-click to export</i>	[clk]
irq	Interrupt Receiver	<i>Double-click to export</i>	[clk]
debug_reset_request	Reset Output	<i>Double-click to export</i>	[clk]
debug_mem_slave	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk]
custom_instruction_m...	Custom Instruction Master	<i>Double-click to export</i>	
switches	PIO (Parallel I/O) Intel FPGA IP		
clk	Clock Input	<i>Double-click to export</i>	clk_0
reset	Reset Input	<i>Double-click to export</i>	[clk]
s1	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk]
external_connection	Conduit	switches	

Figure 6: Exporting PIOs

Use	Connections	Name	Description	Export	Clock
<input checked="" type="checkbox"/>		clk_0	Clock Source		
		clk_in	Clock Input	clk	exported
		clk_in_reset	Reset Input	reset	
		clk	Clock Output	<i>Double-click to export</i>	clk_0
		clk_reset	Reset Output	<i>Double-click to export</i>	
<input checked="" type="checkbox"/>		nios2_gen2_0	Nios II Processor		
		clk	Clock Input	<i>Double-click to export</i>	clk_0
		reset	Reset Input	<i>Double-click to export</i>	[clk]
		data_master	Avalon Memory Mapped Master	<i>Double-click to export</i>	[clk]
		instruction_master	Avalon Memory Mapped Master	<i>Double-click to export</i>	[clk]
		irq	Interrupt Receiver	<i>Double-click to export</i>	[clk]
		debug_reset_request	Reset Output	<i>Double-click to export</i>	[clk]
		debug_mem_slave	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk]
		custom_instruction_m...	Custom Instruction Master	<i>Double-click to export</i>	
<input checked="" type="checkbox"/>		switches	PIO (Parallel I/O) Intel FPGA IP		
		clk	Clock Input	<i>Double-click to export</i>	clk_0
		reset	Reset Input	<i>Double-click to export</i>	[clk]
		s1	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk]
		external_connection	Conduit	switches	
<input checked="" type="checkbox"/>		leds	PIO (Parallel I/O) Intel FPGA IP		
		clk	Clock Input	<i>Double-click to export</i>	clk_0
		reset	Reset Input	<i>Double-click to export</i>	[clk]
		s1	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk]
		external_connection	Conduit	leds	
<input checked="" type="checkbox"/>		onchip_memory2_0	On-Chip Memory (RAM or ROM) Intel ...		
		clk1	Clock Input	<i>Double-click to export</i>	clk_0
		s1	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk1]
		reset1	Reset Input	<i>Double-click to export</i>	[clk1]

Figure 7: Connecting everything



System: nios_system	Path: nios2_gen2_0		
	nios2_gen2_0.data_master		nios2_gen2_0.instruction_master
leds.s1	0x9000 - 0x900f		
nios2_gen2_0.debug_mem_slave	0x8800 - 0x8fff		0x8800 - 0x8fff
onchip_memory2_0.s1	0x4000 - 0x7fff		0x4000 - 0x7fff
switches.s1	0x9010 - 0x901f		

Figure 8: Address Map of Nios system

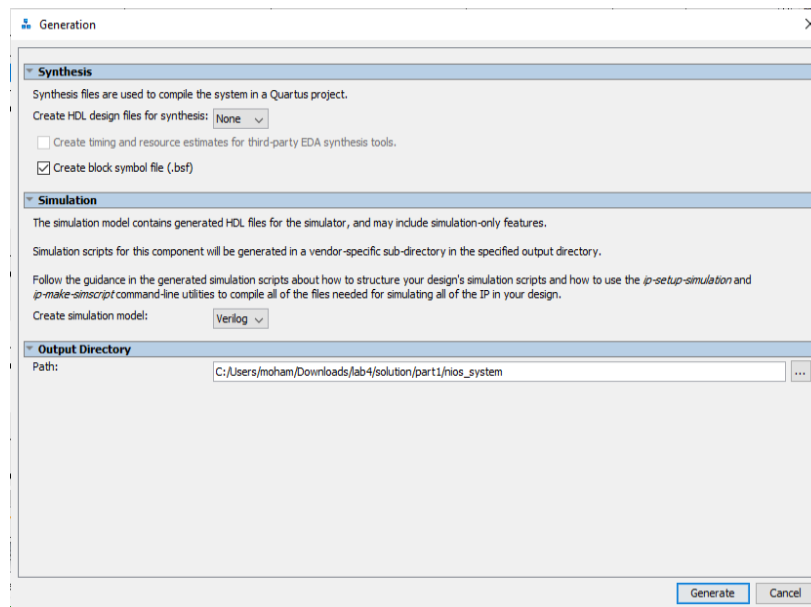


Figure 9: Generate HDL files

Generate > Generate Testbench System and generate the testbenches as shown in Figure 10. Exit the Platform Designer and add the qsys generated `nios_system.sip` to the Quartus project.

NOTE: Make sure to use the name 'nios_system' and save it in the default location. Otherwise your design will not work with the automarker.

Once you're done creating your design, you will proceed with creating a software project that runs on our Qsys generated system, which we can then simulate and verify. From your Quartus window, select **Tools > Nios II Software Build Tools for Eclipse**. If this is the first time you launch the Nios II SDK, it will ask you for the workspace. If you are running on your own machine, anywhere should be fine for this.

NOTE: If you are using an ECF machine, the entire Quartus project, Nios SDK project, and the Nios SDK workspace have to be on the C: drive. Unfortunately, the C: drive is local to each machine and is wiped every 24 hours. So if you use ECF, try to connect to the same machine each time and make sure to copy your work to the W: drive frequently. The W: drive is a network drive that will not be wiped and is shared between all ECF machines. Unfortu-

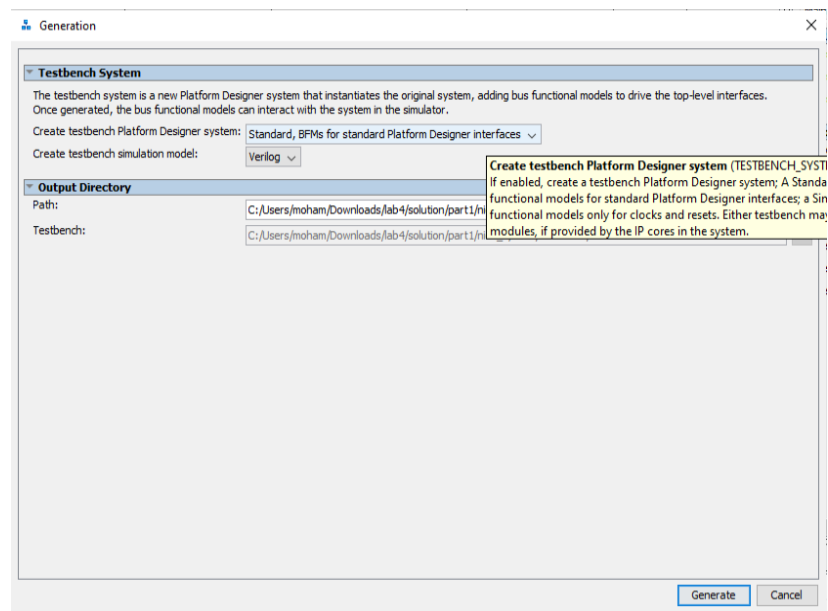


Figure 10: Generate Testbenches

nately, we can't use W: directly because of permission limitations.

Once the SDK has launched, select **File > New > Nios II Application and BSP from template**, you will then specify the **SOPC information file name** as the one located in your Quartus project directory, this is a descriptor file for your Qsys-generated system, allowing the SDK to create projects based on it. After that, name your project to **leds_sw**. Keep the default project location and select the **Blank Project** template. Your configuration should match Figure 12. Click finish.

NOTE: Make sure to use the name 'leds_sw' and save it in the default location. Otherwise your design will not work with the automarker.

You will notice that two projects get created, **leds_sw** and **leds_sw.bsp**. The first is the actual project we are going to use for our code, while the second is the Board Support Package (BSP) which includes an abstraction layer hiding some of the hardware specifics for you. In the BSP you can find a header file called **system.h** which includes macros and definitions for the peripherals in our Qsys-generated system, along with their addresses in the memory map.

Right click on the **leds_sw** project and select **New > Source File** to add a new source file, you can name it **main.cpp**. Populate the file with the necessary code to continuously monitor the value of the switches and display it on the LEDs. When you are done, build your project by selecting **Project > Build all**, fix any issues you might have, and then select **Run > Run As > Nios II Modelsim**, as shown in Figure 13. This will convert your compiled program to a hex file, install it in the on chip RAM, and launch modelsim to simulate the entire Nios II system. Your top level design should only have four signals, **clk**, **reset**, **switches**, and **LEDs**. Add these to your waveform and proceed to verify your design and make sure that changing the value of the switches results in a similar change to the value of the LEDs.

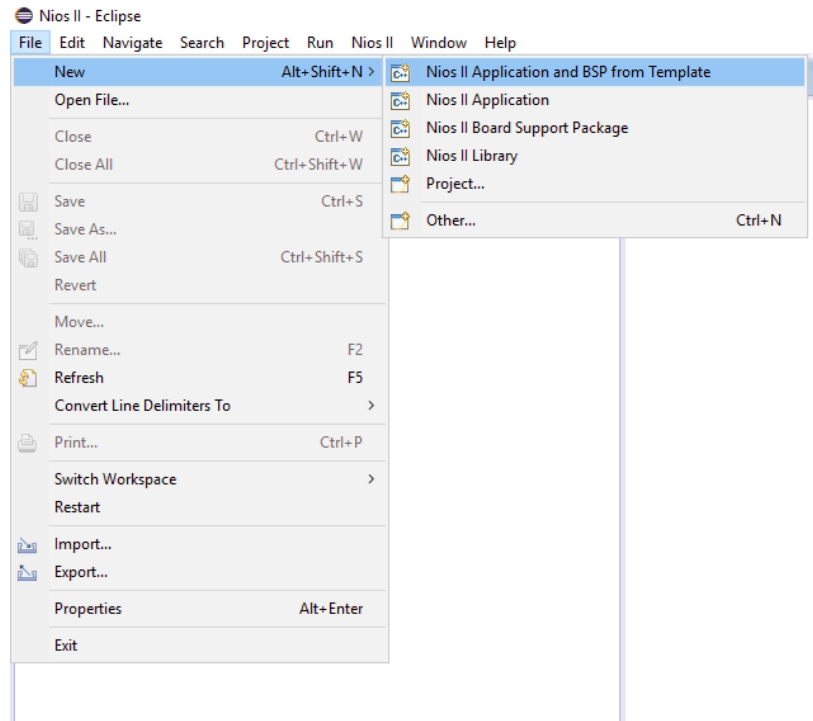


Figure 11: SDK configuration for the newly created project

Part 2: FP Multiplier Qsys Component

Next, we want to integrate a more useful component than the simple switches and LEDs you did in Part I. In preparation for that, in Part II, you will turn a Floating Point (FP) Multiplier into a Qsys Component. Later, in part III, you will add this custom Qsys component to your system like you did in Part I.

Overview

The FP Multiplier component will consist of two modules: An avalon slave controller and the multiplication circuitry. A block diagram of the peripheral is provided in Figure 14. For simplicity, you can simply use an Intel generated floating point core instead of the one you made in Lab 3. This core will automatically support all the corner cases such as Inf, NaN, overflow, underflow etc for you. Normally, you will instantiate this block like any other module. For example, the RAM that you use in ECE241 was created in the same manner. However, to interface with the avalon bus, the multiplier will need an avalon slave controller circuit to translate the FP multiplier signals (e.g., input, result) into Avalon bus signals (e.g., read, write, waitrequest etc). Your multiplier will receive two inputs to be multiplier from the ASC and send the result to the ASC. The ASC will then send that data over the Avalon bus to your NIOS CPU.

The “Avalon interconnect” signals will connect to the rest of the Qsys-generated system using the Avalon-MM protocol, which your ASC will need to understand. Via these signals, an Avalon Master (such as the Nios II) can communicate with your peripheral. These signals include a clock and synchronous reset.

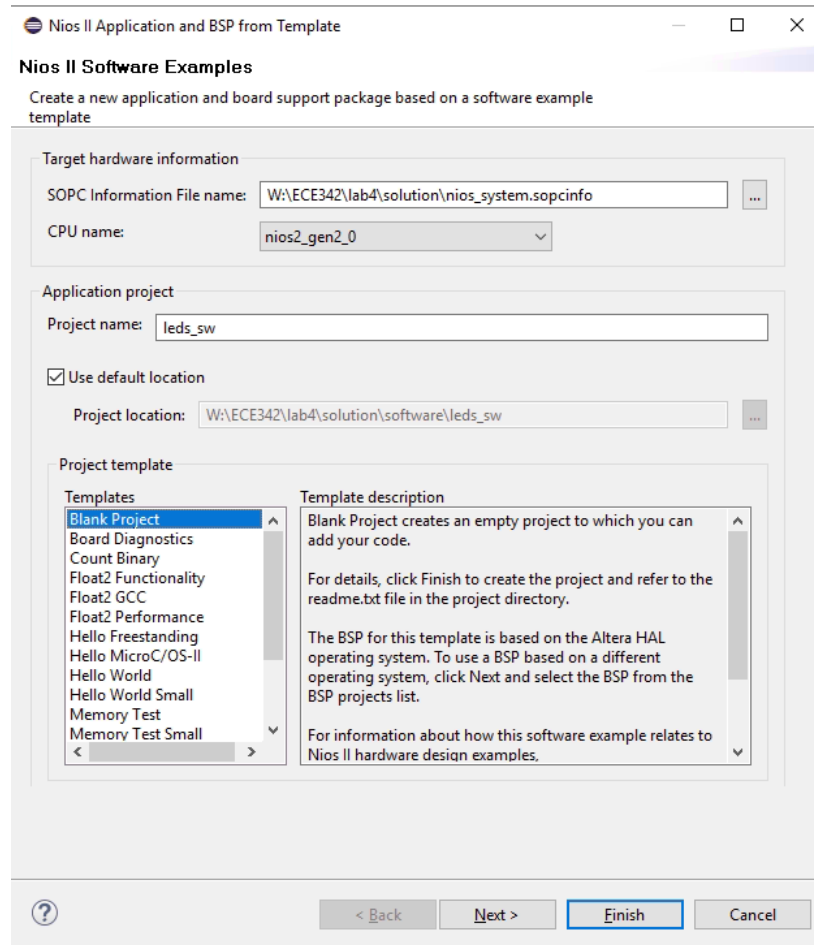


Figure 12: SDK configuration for the newly created project

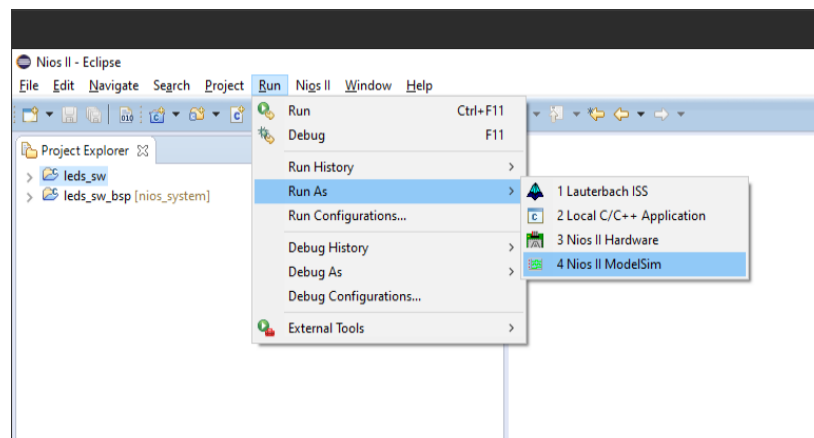


Figure 13: Running the Nios II simulation in modelsim

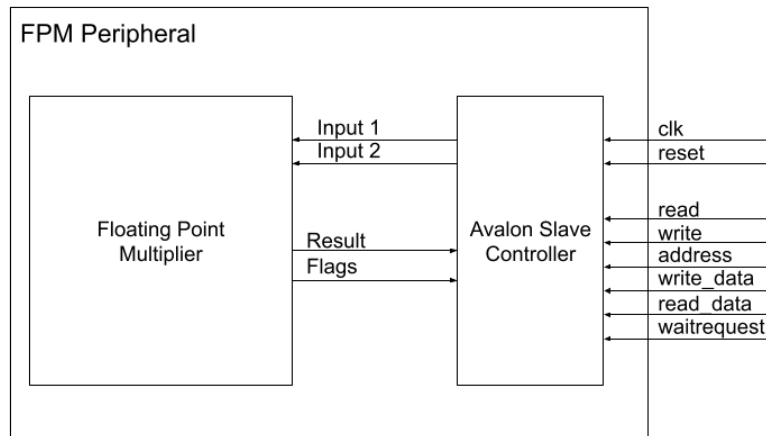


Figure 14: The FP Multiplier peripheral diagram

Master Byte Address	Slave Offset Address	Bits			
		31	30	3 2 1 0
0x7000	000	Operand 1			
0x7004	001	Operand 2			
0x7008	010	Status			
0x700C	011	Result			
0x7010	100	Status			

Figure 15: The FP multiplier memory-mapped registers

Register Map

Recall that for a system using memory-mapped I/O, all reads and writes either to memory or peripherals look identical. So within your Qsys system, all communications between the processor and peripherals look like reads and writes but to addresses well outside the actual range of system RAM. By reading from or writing to a range of addresses associated with a peripheral, code running on the Nios II processor can access different functions of that peripheral via its set of exposed memory-mapped registers.

The memory-mapped register interface of the FP Multiplier is shown in Figure 15, using an **example** base address of 0x7000. Through this interface the processor can set the two multiplication operands in the first two registers, respectively. It can also read back the result from the fourth register, and check the necessary flags in the fifth register. The flags occupy only the four least significant bits of their register, while the rest is unused, these are ordered as followed, from bit 3 down to bit 0: Overflow, Underflow, Zero, and NaN. These correspond exactly to the four conditions you checked in lab 3. Whereas in Lab 3, you had different outputs for each of these, in this case, the 4 outputs are all concatenated into a single 4-bit ‘status’ register. But the behavior remains the same.

The multiplier works as follows; The two operands must first be written into the first two registers. Then, writing 1 to the least significant bit of the start register triggers the start of the multiplication. This bit

also serves as the busy flag indicating whether the multiplier has finished computing and can receive new inputs (The multiplier is busy as long as this bit is still 1). Once the start bit is written, the ASC sends the operands to the multiplier, keeps the start/busy bit high, and raises the avalon waitrequest signal, indicating that it is unable to receive any more requests. Upon completing the multiplication, it saves the result in the result register and the flags in the status register. On a subsequent read to these locations, it issues these values back to the processor.

Figure 15 also shows the master byte address for each register. This master byte address is the address that the avalon master (i.e., the NIOS CPU) sees for the peripheral. Any code running on the NIOS cpu must read/write to this location to access this peripheral's register(s). This is because to Avalon masters, 1 unit of address always represents 1 byte, regardless of the actual physical size of the register. The slave offset address is the address that the peripheral sees when it is being accessed. It is a relative, rather than absolute address, from the peripheral's assigned base address within the system. Its units are defined by the peripheral, and can be set to a size most convenient for it. In our case, that means 1 address is 1 entire 32-bit register. Only 3 bits are needed, since there are only 5 such registers.

So, for example, if Nios II wants to write to the start register, it will present an address of 0x7008 to the Avalon interconnect. Since this address is within the range of our peripheral, the interconnect will know to direct the write to us and not some other device. The interconnect then strips off most of the bits, and translates the rest into the slave offset address, 010 in binary. The ASC module within the peripheral will then see its `write` signal go high along with the slave address being the value 010. The exact details of how a read or write arrives at the ASC, and what signals it sees, is covered in the next section.

NOTE: The base address of 0x7000 is arbitrary. You can choose a different one in Qsys and change it accordingly throughout this Part.

Avalon Slave Read/Write Transfers

This section describes the “Slave signals” portion of Figure 14, their purpose and timing. This will allow you to design your ASC.

The shortest duration of Avalon read/write transfers is one cycle. The address and control signals are held constant for the duration of only one cycle, so the slave must respond within that cycle by either presenting valid data (read transfer) or capturing data (write transfer). In this lab all of your transfers will be one cycle, with the exception that any reads or writes issued while the waitrequest signal is asserted will be of variable latency.

Figure 16 shows the examples of one-cycle transfers with no waitrequest signal ($CC[n] = n$ -th clock cycle)

- CC [1] The master asserts address and read on the rising edge of the clock. In the same cycle the slave decodes the signals from the master and presents valid readdata.
- CC [2] The master captures readdata on the rising edge of the clock and deasserts the address and control signals marking the end of the transfer.
- CC [4] The master asserts address, write, and writedata on the rising edge of the clock. The master signals are held constant and the slave decodes them.
- CC [5] The slave captures writedata on the rising edge of the clock. The master deasserts the address, writedata and control signals marking the end of the transfer.

Figure 17 shows the examples of variable duration transfers. Below is the explanation for the write transfer that is 3 cycles in duration. The slave prolongs the transfer for 2 extra cycles by inserting 2 wait-states using the waitrequest signal.

- CC [1] The master asserts address, write and writedata on the rising edge of the clock. In the same cycle the slave decodes the signals from the master and asserts waitrequest and thereby stalls the transfer.

- CC [2] The master samples waitrequest on the rising edge of the clock. Thus, this cycle becomes a wait-state (or wait cycle), hence the signals address, write and writedata remain constant. In this cycle, CC (2), the slave keeps the waitrequest high, thereby inserting the second wait-state.
- CC [3] The slave deasserts waitrequest on the rising edge of the clock.
- CC [4] The slave captures writedata on the rising edge of the clock. The master samples deasserted waitrequest and ends the transfer by deasserting all signals.

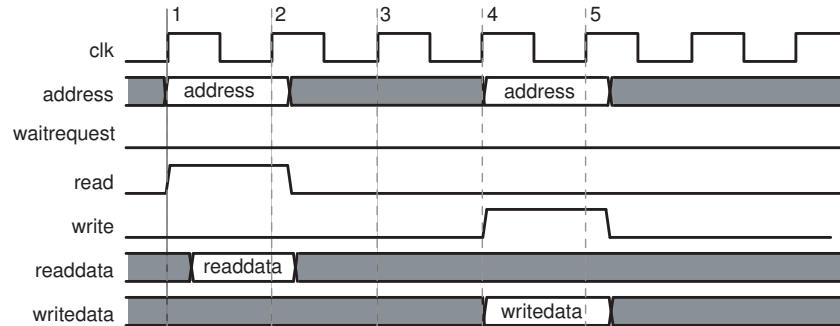


Figure 16: Avalon slave timing diagrams for read/write transfers (without `waitrequest`) - 1 cycle duration

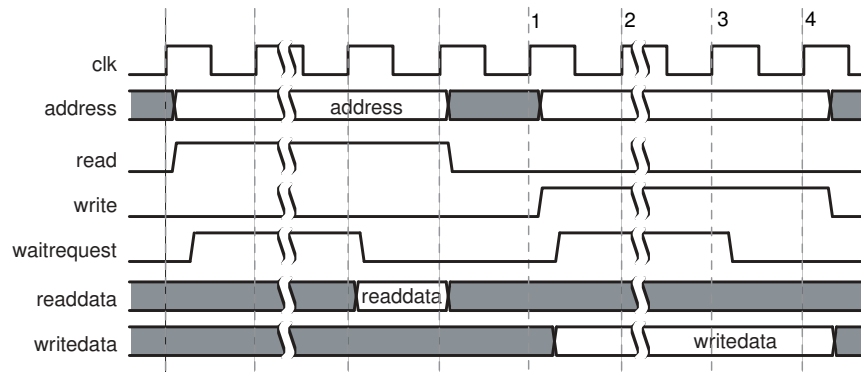


Figure 17: Avalon slave timing diagrams for read/write transfers (with `waitrequest`) - variable duration

Creating the Component

In a NEW project created from the `part3` folder of our starter kit, use the IP Catalog in your Quartus window to search for an IP called `ALT_FPMULT`. Once found, double click the IP name, this will ask your for the IP name, give it a proper name and a MegaWizard window will launch right after, allowing you to customize the multiplier before instantiating it. In the first tab, make sure the multiplier is Single Precision, then click next and check all the optional signals to include them in our multiplier. Click finish to generate our multiplier, and add the generated `.qip` file to the project. The configuration for the multiplier is shown in Figures 18 and 19.

Once you have the multiplier created and added to your project, you will have to create two more modules: the Avalon Slave Controller, and a top-level component module that instantiates and connects everything

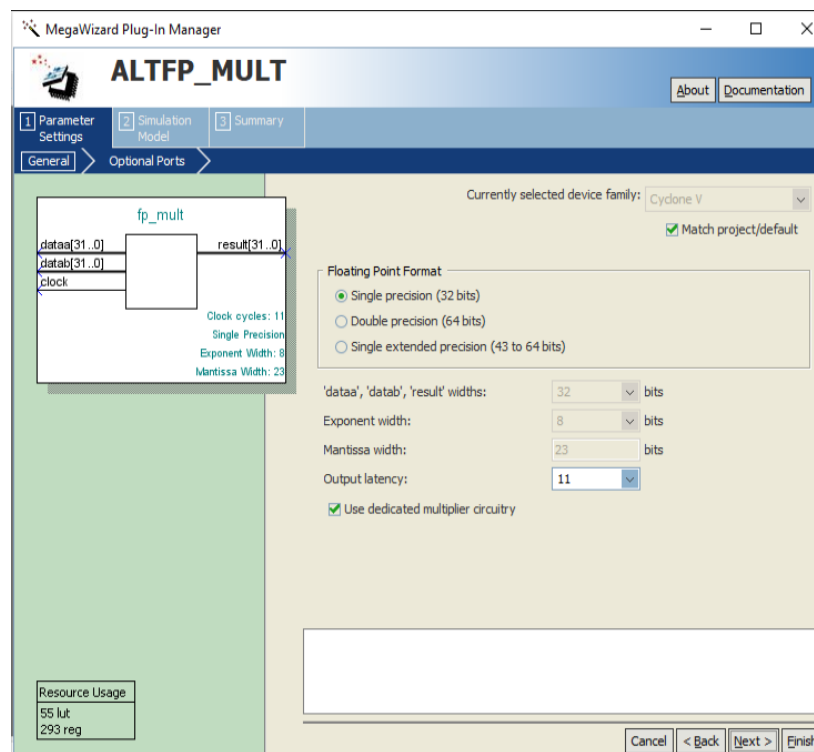


Figure 18: FP Multiplier configuration

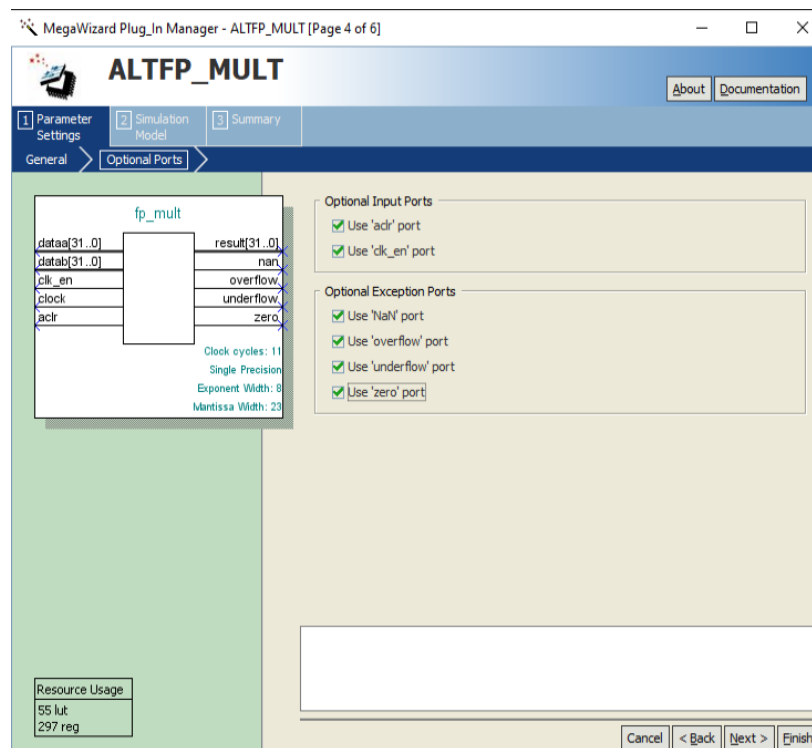


Figure 19: FP Multiplier extra signals

in Figure 14. Skeleton code for these modules are given to you as part of the starter kit, you are required to keep the names of modules, instances, and registers already defined there. When all three modules are complete, a Qsys component can be generated for the multiplier.

NOTE: The names of the modules, registers, and instances must not be modified, so the automarker can recognize them

Open the platform designer by selecting **Tools > Platform Designer**. When the platform designer opens, select **File > New Component** which will prompt us with a wizard to create a Qsys component for our multiplier. Fill out the fields as shown in Figure 20, make sure the name exactly matches the figure so the automarker can recognize it correctly. Switch to the Files tab of the wizard. Under **Synthesis Files** select **Add Files** and add the `avalon_slave.v`, `avalon_fp_mult.v`, `fp_mult.v`, and `fp_mult.qip`. Then click **Analyze Synthesis Files** and make sure there are no errors. Next, under **Simulation Files** select **Copy From Synthesis Files** and make sure the files configuration matches that of Figure 21. In the signals tab, specify the associated clock and reset as shown in Figure 22 and click **Finish**.

Part 3: Nios II Connected Multiplier

Once the multiplier is created as a Qsys component, start creating your new Nios II system using it. Namely, it should include the following:

- A Nios II/e Processor.
- A 16KB On-Chip Memory to hold program code and data.

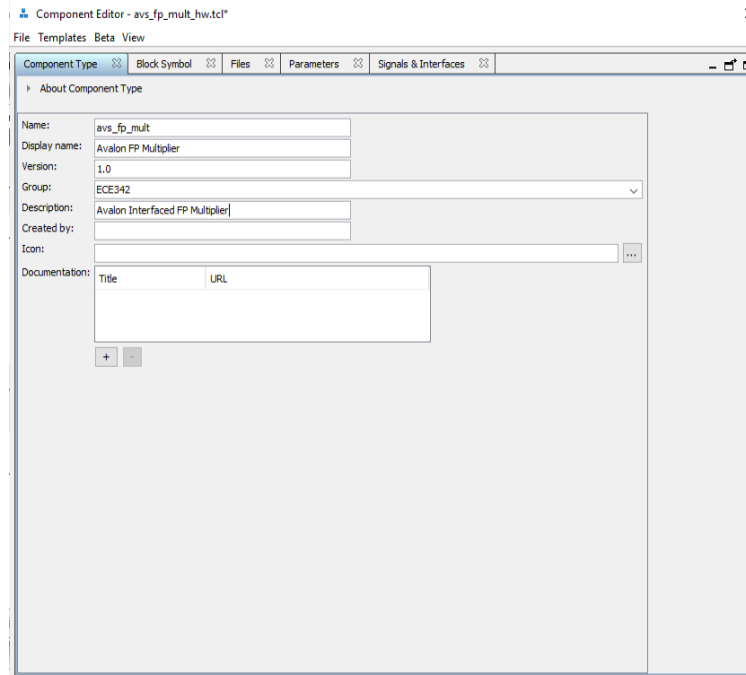


Figure 20: Name, category, and description of Qsys component

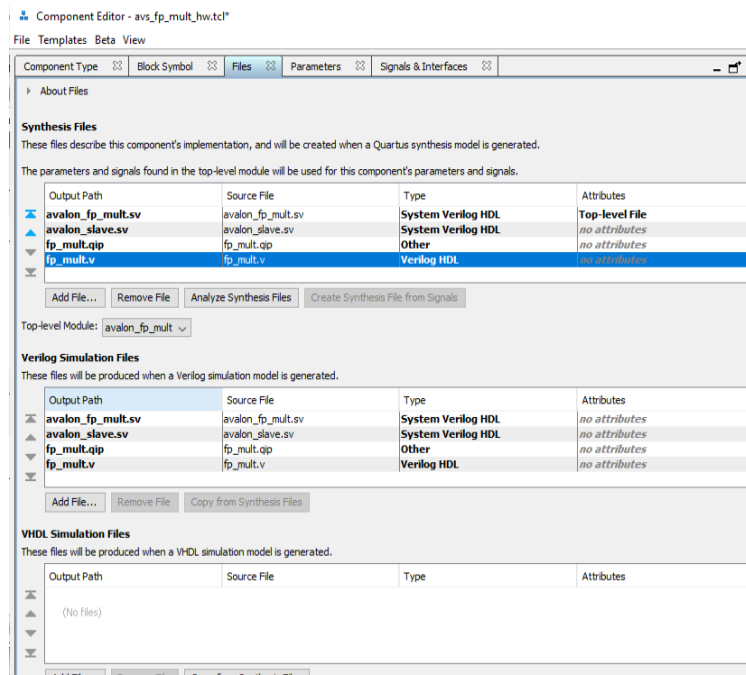


Figure 21: Synthesis and Simulation files of Qsys component

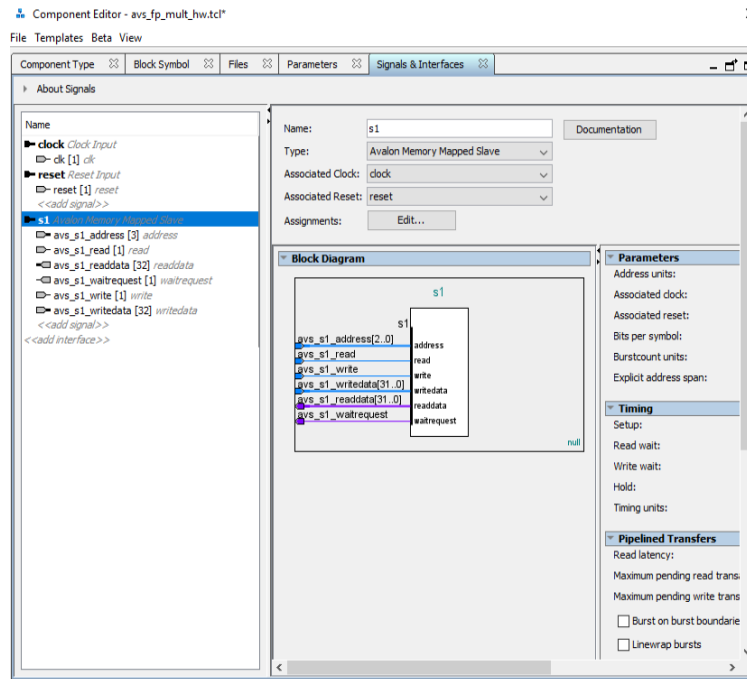


Figure 22: Signal assignment for Qsys component

- Our floating point multiplier component.

Using the same Quartus project as Part 2, follow the same procedure as Part 1. But this time, create a software project with the name 'fpmult_sw' and use it to build a program that does multiplication in two ways. The first using direct multiplication in C code, using the '*' operator. Since the NIOS II CPU we are using does not have a floating point unit built-in, this operation will be emulated using software. The second way will be using the Floating point unit you created in Part II. Doing so, you can compare the number of cycles required for each approach and see the speed-up when using dedicated FP hardware vs. doing it in software.

You will be given a skeleton C code as part of the starter kit, which you can use for creation of this program. You're expected to read the multiplication operands from specific RAM addresses (provided in the skeleton code), use them for software multiplication and then write the value back to another specific address (also provided in the skeleton code). Once this is done, proceed to use the same inputs for the multiplier we've created.

NOTE: Make sure to use the name 'fpmult_sw' and save it in the default location. Otherwise your design will not work with the automarker.

NOTE: Do not change the addresses used for the multiplication, these are the locations tracked by the automarker.

1 Submission

NOTE: The submission instructions for this lab are different from previous labs. Make sure you read these instructions carefully to avoid issues.

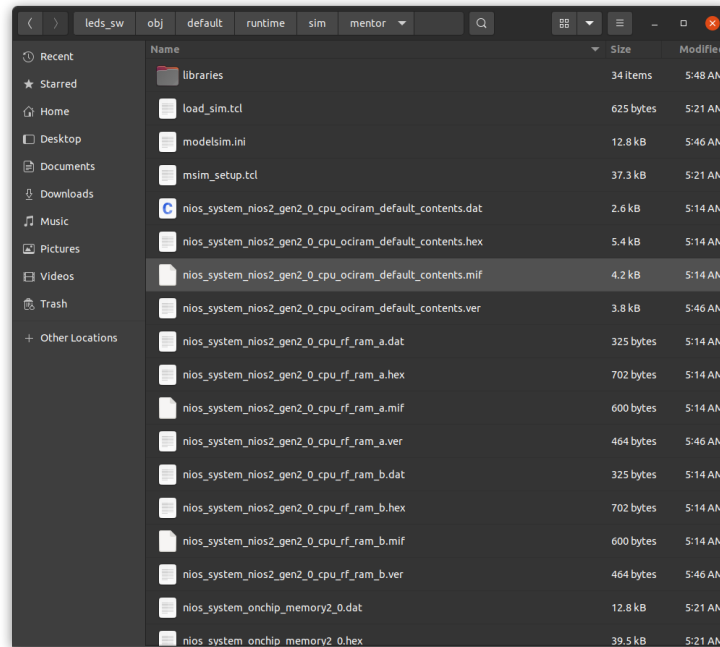


Figure 23: Some of the required files

For part 1, you will have to archive the entire folder of your Quartus project. If you have followed the instructions correctly, that folder should have the name 'part1', and should also include the Qsys Nios system and the Nios II SDK project. The archive must be named 'part1.zip' before submission.

NOTE: Your part1 folder must include the path `software/leds_sw/obj/default/runtime/sim/mentor` and that path must include files similar to Figure 23. If it does not, your submission will fail the automarker. Make sure you have followed the instructions in this document correctly, and make sure to build and test your software in modelsim.

For part 2, you must put the two modules 'avalon_slave' and 'avalon_fp_mult' in one file, named 'part2.sv', which can then be submitted.

For part 3, if you have followed the instructions correctly, the folder should have the name 'part3' and should include all the relevant components, including the files from part 2, Nios system, and the Nios II SDK project. You will also archive that folder to a file called 'part3.zip', which you can then submit.

NOTE: Your part3 folder must include the path `software/fpmult_sw/obj/default/runtime/sim/mentor` and that path must include files similar to Figure 23. If it does not, your submission will fail the automarker. Make sure you have followed the instructions in this document correctly, and make sure to build and test your software in modelsim.

You should submit part1.zip, part2.sv, and part3.zip using the instructions provided in the submission document i.e.: `submitece342s 4 part1.zip part2.sv part3.zip`