Getting Started with OpenCL on the ZYNQ

Bo Joel Svensson
bo.joel.svensson@gmail.com

Rakesh Tripathi
rakesht@chalmers.se

Guide version	0.5
Last edited	October 12, 2018
Board compatibility	zynqberry
Tested Vivado versions	2015.4, 2018.2

Disclaimer

1 Introduction

This document attempts to provide a complete walk through of the entire OpenCL HLS work flow using Xilinx Vivado. The Board we target in this version (0.5) of the document is the Trenz ZynqBerry. In future versions we will try to highlight the parts that differ when using a ZedBoard.

This document is work in progress and new versions will be posted as we refine the procedure and gain a deeper understanding of all the details.

All feedback, hints, tips, corrections and explanations of details we are vague upon, would be greatly appreciated and acknowledged in future revisions.

1.1 Initial setup for the zynqberry

The procedure outlined in this document has been tested against a ZynqBerry board [1]. The version of this board that we use is the 128MB variant "TE0726-02" which has now been superseded by the "TE0726-02M" variant with more memory. While we do not have access to an "TE0726-02M" board and we cannot test, we assume the procedure explained in this document will apply with minimal changes.

In order to make the description of the procedure as complete as possible we are not basing this guide on an existing (vendor supplied) example Vivado project. Only one piece of data is taken from a reference design provided by Trenz, the so called "board files". We copy the board files available in the "test board" reference design [2]. Under linux these board files are copied

[1] http://www.trenz-electronic.de/products/fpga-boards/trenz-electronic/te0726-zynq.html

[2] http://www.trenz-electronic.de/download/d0/Trenz_Electronic/d1/TE0726/d2/Reference%20Designs/d3/2015.4/d4/test_board.html

to directory Xilinx/Vivado/2015.4/data/boards/board_files of your Xilinx vivado install tree. If these board files have been added correctly it will be possible to select the ZynqBerry as a target for Vivado.

1.2 Guide structure

This guide is split into three parts that goes through: first writing a simple OpenCL program and synthesizing it using Vivado HLS, second designing a system (in Vivado) that interfaces the hardware generated by HLS in step 1 with the processing system and the memory system in the Zynq chip, finally we show how to develop software (in the SDK) for the processing system that starts computations in the OpenCL generated hardware.

2 Part 1: Vivado HLS and OpenCL

In this section we develop an OpenCL program for vector addition (`vadd`). This `vadd` computation is given pointers to three vectors (arrays), two inputs and one output, and performs element wise addition of the inputs into the output.

2.1 Creating a Vivado HLS project

Start Vivado_hls and create project using the following steps:

- step1: Create a project and name it "vadd_OpenCL", see figure 1.
- step2: Now you are asked to provide a name for the top level function. This is the function that specifies the interface to the generated hardware. Name the top level function (vadd). This step is outlined in figure 2.
- step3: We are not adding a testbench file. See figure 2.
- step4: Now it is time to configure the solution details. We can leave the solution name unchanged ("solution1") and then we select the device:
  - Family: Zynq
  - Package: clg225
  - Speed grade: -1

  There should now be just one available selection "xc7z010clg225-1". See figure 3.

Now project configuration is done and we can hit finish and enter into the development environment. This should look like the left part of figure 4.

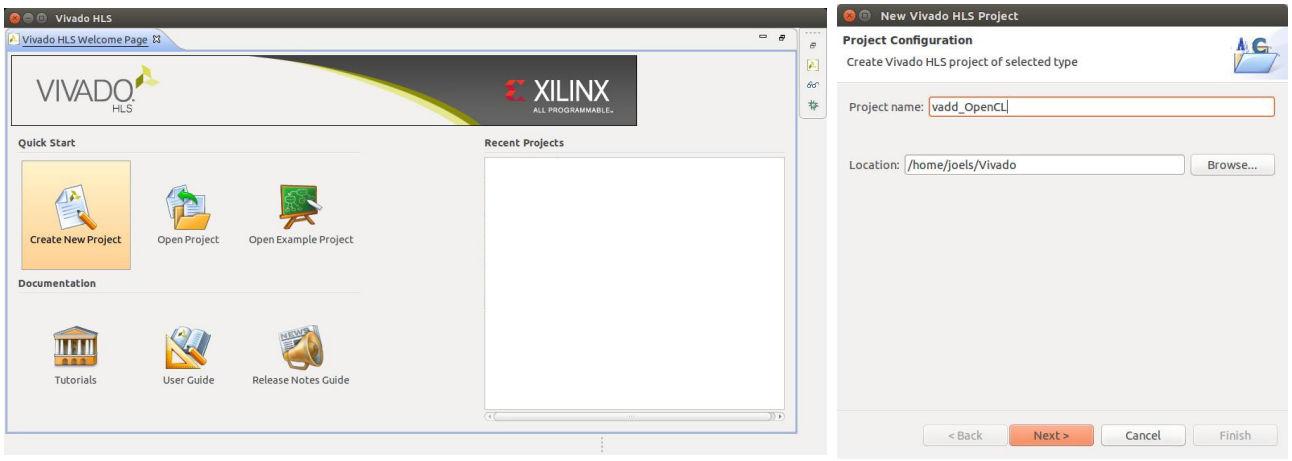


Figure 1: Project creation and naming

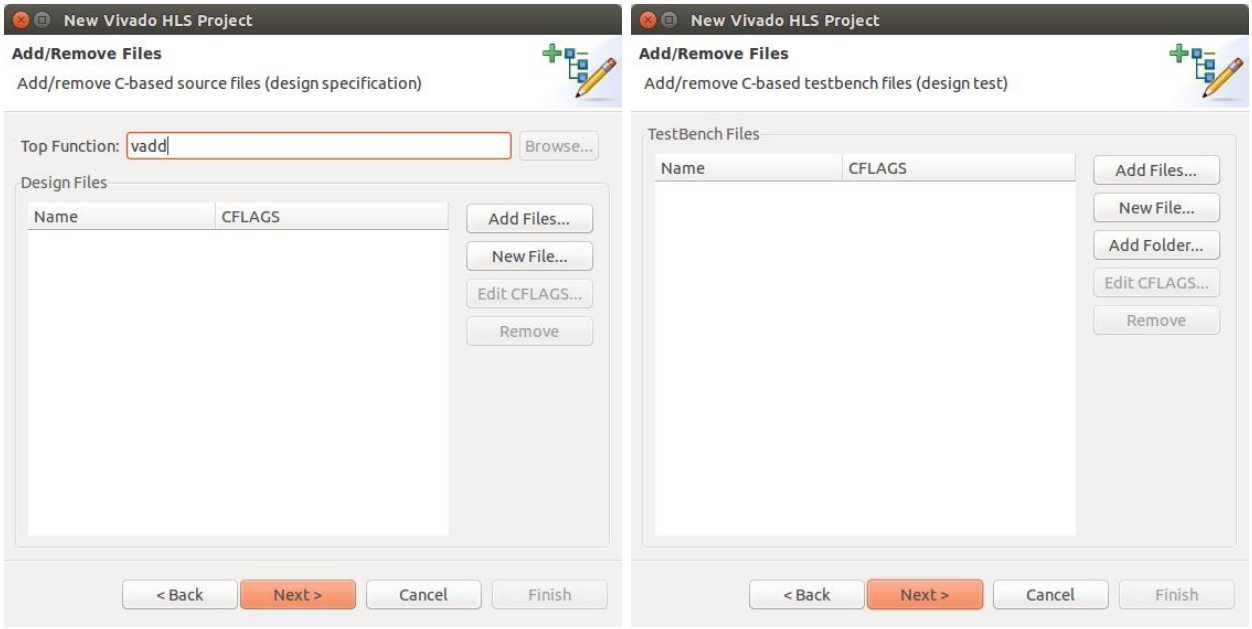


Figure 2: Identification of the *Top* function and addition of testbench file. We are not providing any testbench.

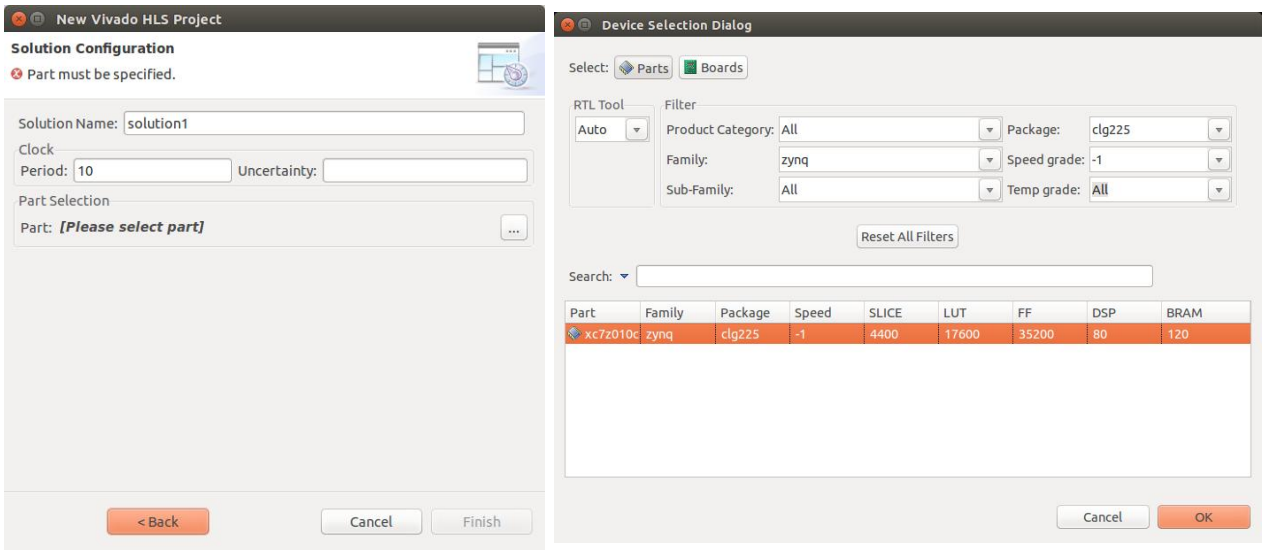


Figure 3: Solution configuration. Select the "xc7z010clg225-1" device.

2.2 Writing a simple OpenCL kernel

The example kernel used in this guide is very simple and is outlined in total below. Add a new source file to the project. Right click "source" in the *Explorer* under "vadd_OpenCL" and select "New file". Name the file "vadd.cl". The file extension is important ".cl".

```
#include <clc.h>

__kernel void __attribute__ ((reqd_work_group_size(128,1,1)))
  vadd( __global int *a, __global int *b, __global int *c) {

    int i = get_global_id(0);

    c[i] = a[i] + b[i];
}
```

After creating the new file type in the OpenCL code as above.

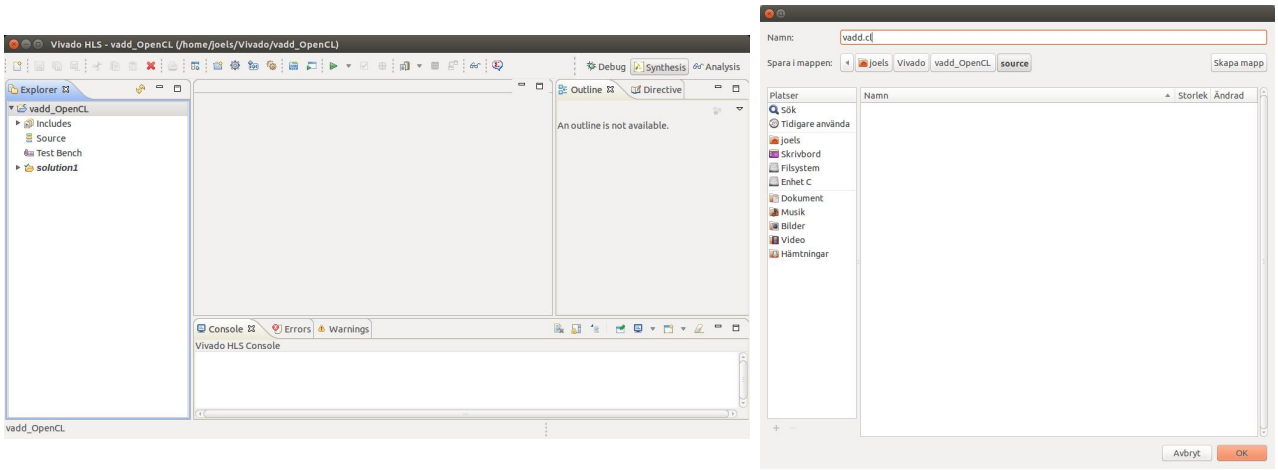
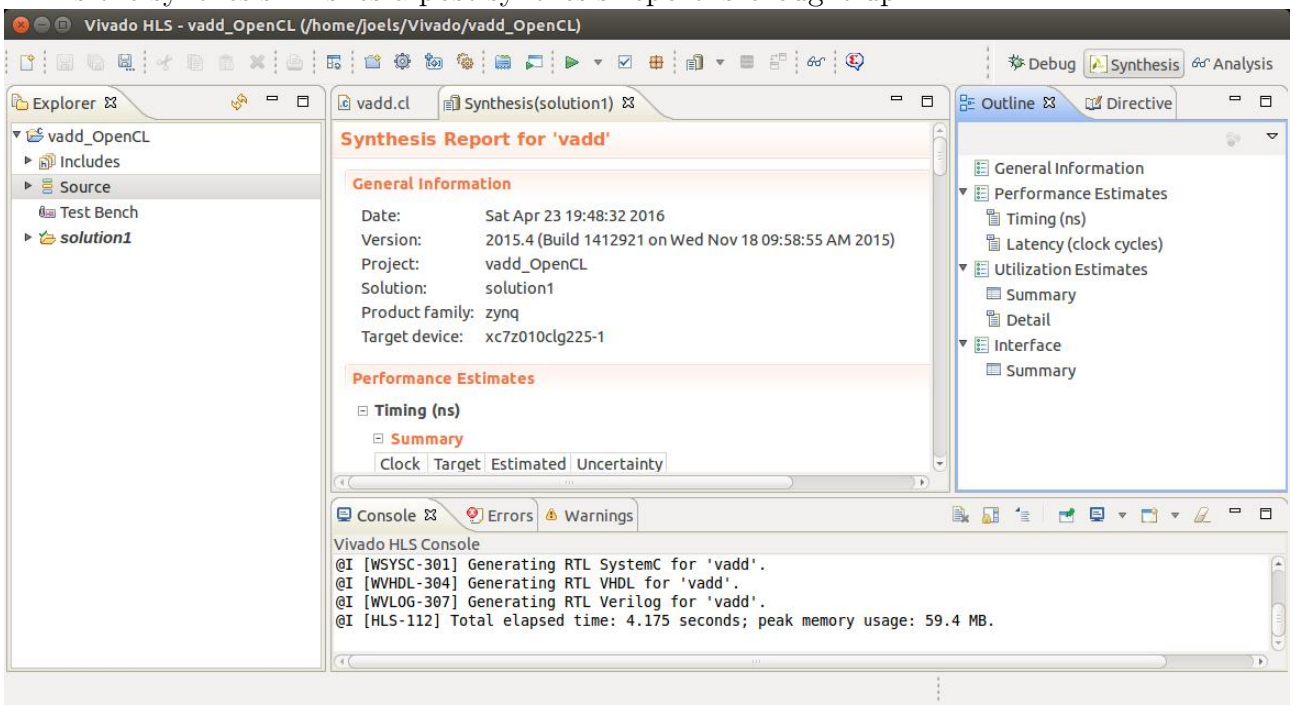


Figure 4: IDE view and source file creation.

2.3 Synthesize the OpenCL code

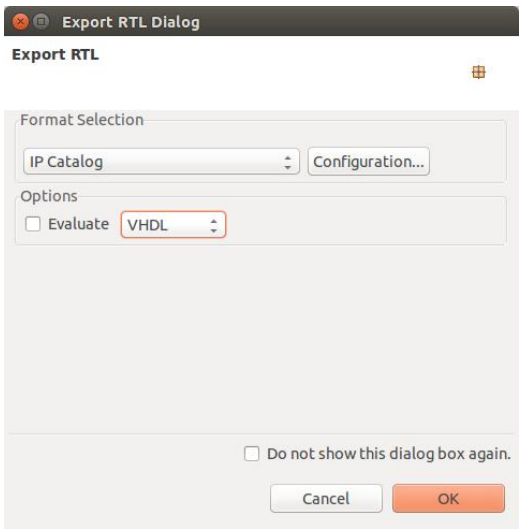
After writing the OpenCL, synthesis and exporting the IP remains in order to conclude the part of the work that takes place in vivado_hls. If the code has been entered correctly this should go through synthesis without problems. Hit the green "synthesis" button in the toolbar.

As the synthesis finishes a post synthesis report is brought up.



Now export the generated hardware description into the IP catalog. This step makes our vadd hardware unit available for use in Vivado. Click the "Export RTL" button in the toolbar.

The choices we make in the "Export RTL" dialog are shown in figure 5. We choose "IP Catalog" and VHDL as the desired language. One can also provide identification details using "Configuration" button but we leave these settings unchanged.



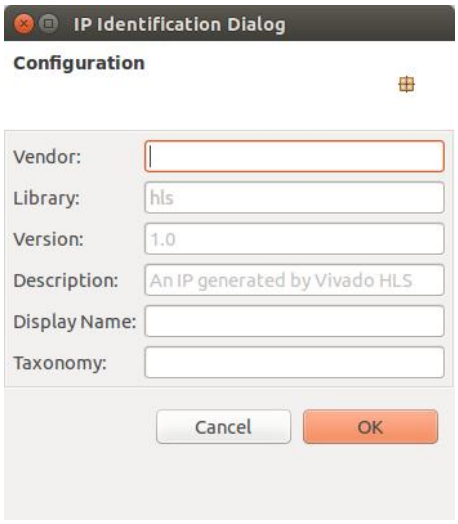


Figure 5: The "Export RTL" dialog.

We are now done with vivado_hls and will start up Vivado.

2.4 Programming Interface Generated by HLS

After synthesizing hardware from the OpenCL code new directories appeared called "impl" and "syn" containing VHDL code. Within one of these files we find information that is important to keep in mind later when writing the software that interfaces with the generated hardware, this information will be used in section 4.1. The information we seek, is located in the "vadd_control_s_axi.vhd" file and shows the layout of the memory mapped interface for communication with the vadd hardware unit:

```
-- ------------------------Address Info-------------------
-- 0x00 : Control signals
--        bit 0  - ap_start (Read/Write/COH)
--        bit 1  - ap_done (Read/COR)
--        bit 2  - ap_idle (Read)
--        bit 3  - ap_ready (Read)
--        bit 7  - auto_restart (Read/Write)
--        others - reserved
-- 0x04 : Global Interrupt Enable Register
--        bit 0  - Global Interrupt Enable (Read/Write)
--        others - reserved
-- 0x08 : IP Interrupt Enable Register (Read/Write)
--        bit 0  - Channel 0 (ap_done)
--        bit 1  - Channel 1 (ap_ready)
--        others - reserved
-- 0x0c : IP Interrupt Status Register (Read/TOW)
--        bit 0  - Channel 0 (ap_done)
--        bit 1  - Channel 1 (ap_ready)
--        others - reserved
-- 0x10 : Data signal of group_id_x
--        bit 31~0 - group_id_x[31:0] (Read/Write)
-- 0x14 : reserved
-- 0x18 : Data signal of group_id_y
--        bit 31~0 - group_id_y[31:0] (Read/Write)
-- 0x1c : reserved
-- 0x20 : Data signal of group_id_z
--        bit 31~0 - group_id_z[31:0] (Read/Write)
-- 0x24 : reserved
-- 0x28 : Data signal of global_offset_x
--        bit 31~0 - global_offset_x[31:0] (Read/Write)
-- 0x2c : reserved
-- 0x30 : Data signal of global_offset_y
--        bit 31~0 - global_offset_y[31:0] (Read/Write)
-- 0x34 : reserved
-- 0x38 : Data signal of global_offset_z
--        bit 31~0 - global_offset_z[31:0] (Read/Write)
-- 0x3c : reserved
-- 0x40 : Data signal of a
--        bit 31~0 - a[31:0] (Read/Write)
-- 0x44 : reserved
-- 0x48 : Data signal of b
--        bit 31~0 - b[31:0] (Read/Write)
-- 0x4c : reserved
-- 0x50 : Data signal of c
--        bit 31~0 - c[31:0] (Read/Write)
-- 0x54 : reserved
-- (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)
```

This gives us the "offsets" from some base address to where each of the register used by the vadd hardware is located. Later, in vivado, a complementary step will provide us with the

base address, see section 3.3.

The directly important pieces of information here is the control register, the group_id registers and the a,b and c data registers.

- **Control**: using this register we can start computations in the vadd hardware unit and also poll for the done signal.
- **Group id**: group_id_x, group_id_y, group_id_z specifies a three dimensional work-group id. Since the OpenCL kernel we use is meant for one dimensional "NDRanges" only group_id_x is of importance. This value (group_id_x) is changed between invocations of vadd if the data we operate upon is larger than what can be computed by one workgroup instance (the only valid value for the others is zero).
- **Argument pointers**: pointer to memory where the vadd hardware can fetch and store data should be written to the a,b,c register.

3 Part 2: Vivado

This section presents step by step instructions on how to integrate the OpenCL kernel IP-block designed earlier into a Zynq base system.

3.1 Creating a Vivado project

Begin by starting Vivado. This presents you with the view shown in figure 6. Select "Create New Project" and click "Next".

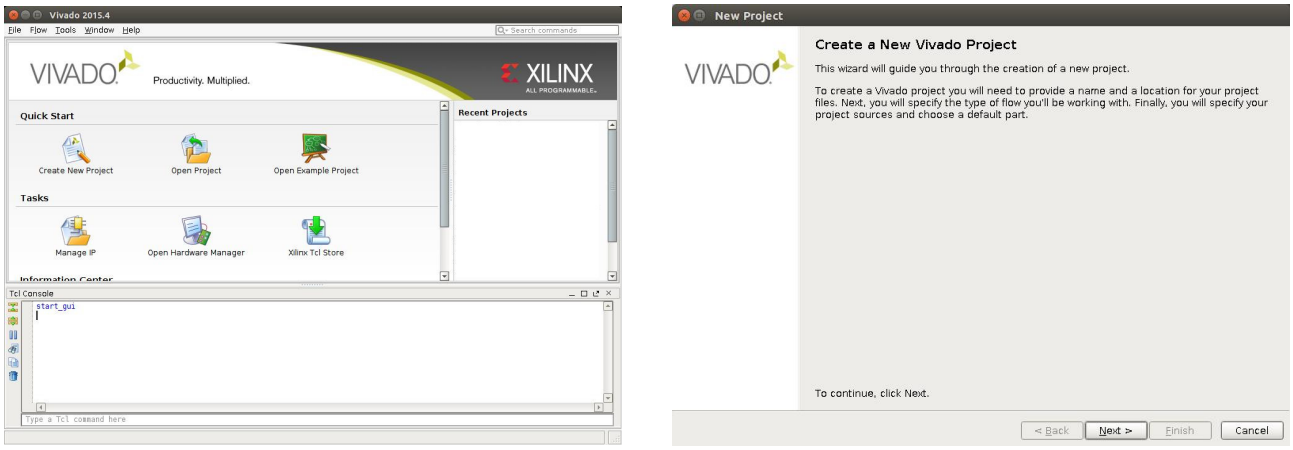


Figure 6: Vivado project creation wizard.

Choose a name and location for the project, in this case "ZynqOpenCL" and a directory called "Vivado". Click "Next". In the next window select "RTL Project" and check "Do not specify sources..". This part of the procedure is shown in figure 7.

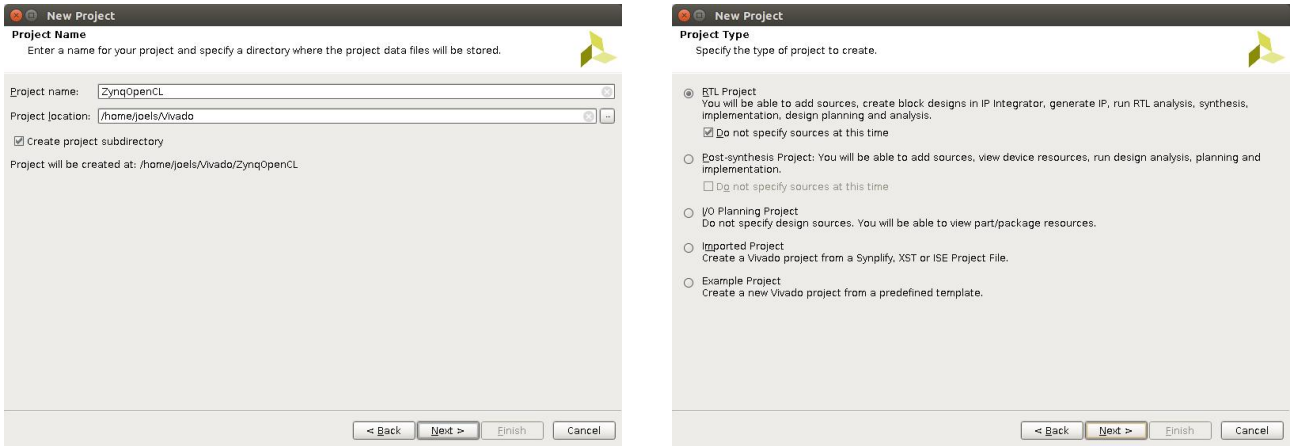


Figure 7: Project name and project type.

Now it is time to select the "Default Part" to use as target platform. Click "Boards". If the steps in section 1.1 has been performed there should be an option "ZYNQ-7 TE0726-02" for the zynqberry board. Select the suitable board then click "Next" and then "Finish'

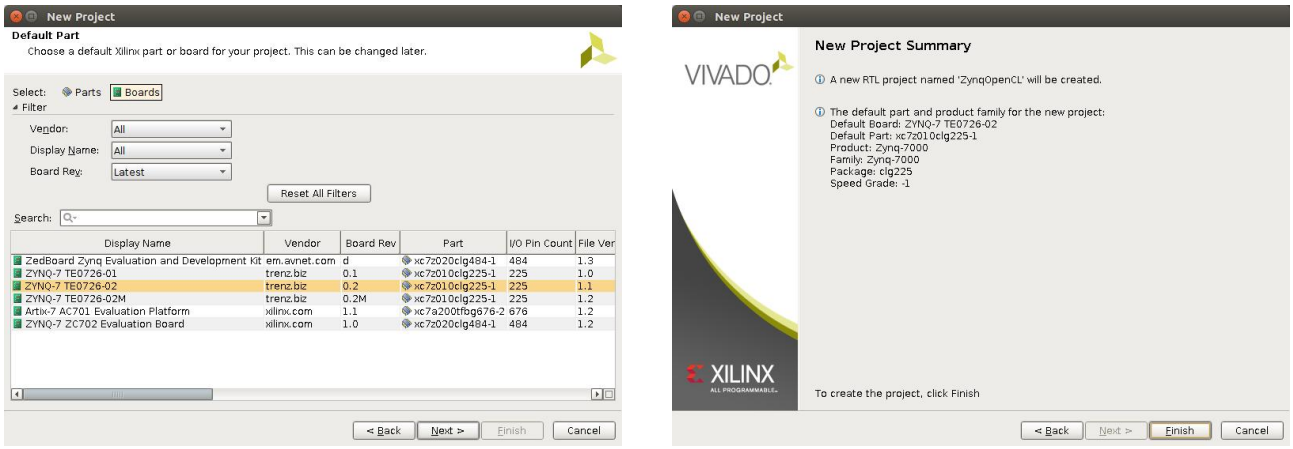


Figure 8: Project configuration wizard board selection.

This concludes the project configuration procedure.

3.2 Designing the system

Now we have entered Vivado and are presented with a “Project Manager” view, a “Project Summary” and the “Flow Navigator”. In the Flow Navigator select “Create Block Design”. The default name “design_1” is fine and we can keep it and just hit “OK”. This should bring up a “Block Diagram View” as is shown in figure 9.

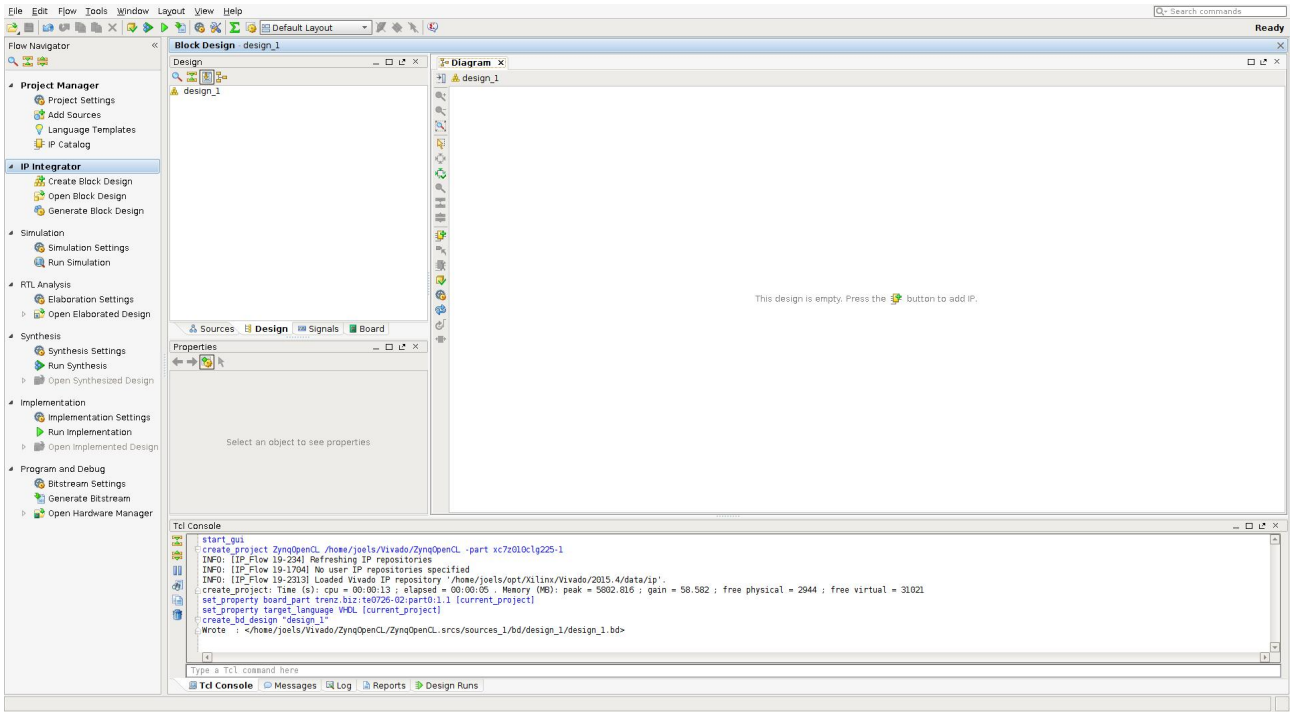
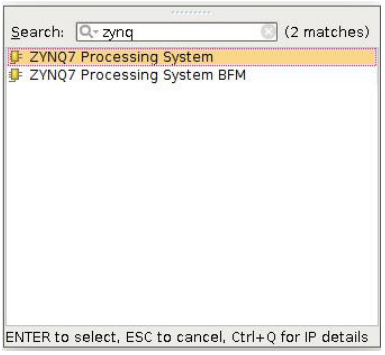


Figure 9: Vivado Block Diagram view.

In the Diagram view “design_1” we click the add IP button. The shape of this button should now be shown in the middle of the diagram view but can later be found in the vertical toolbar next to the diagram view. Click “add IP” and enter “zynq” into the search field. Select the “ZYNQ7 Processing System”.



The diagram view should now contain a Zynq processing system as shown in figure 10. Not that there is a "Run Block Automation" link within the block diagram at this point. Hit this link and mark "All Automation" and then click Ok. The block automation dialog is shown in figure 11.

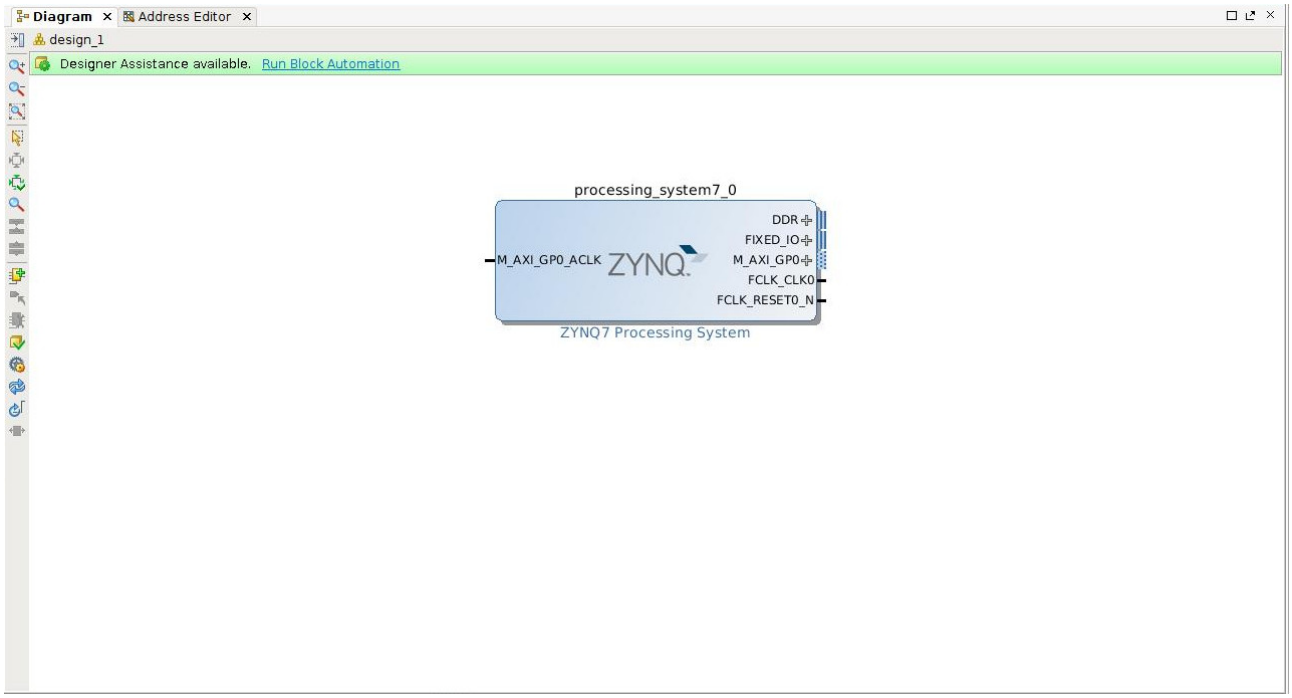


Figure 10: Vivado Block Diagram view with ZYNQ processing system.

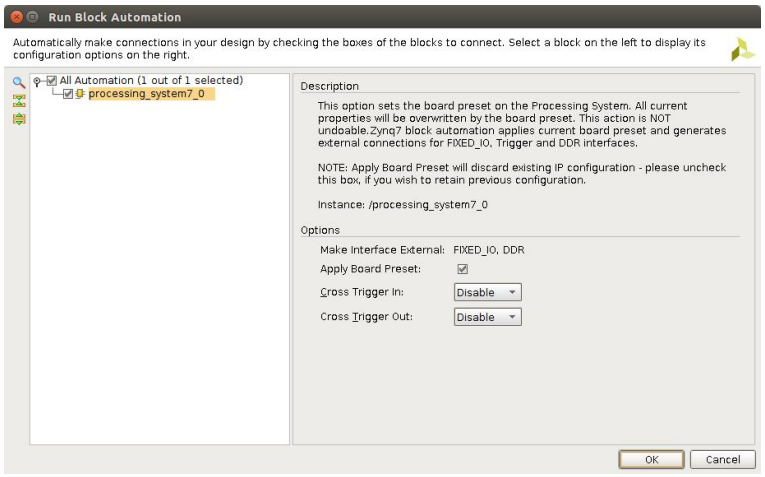


Figure 11: Block automation dialog for the Processing System.

After allowing the block automation for the processing system to apply the default settings, the block diagram should look as in figure 12.

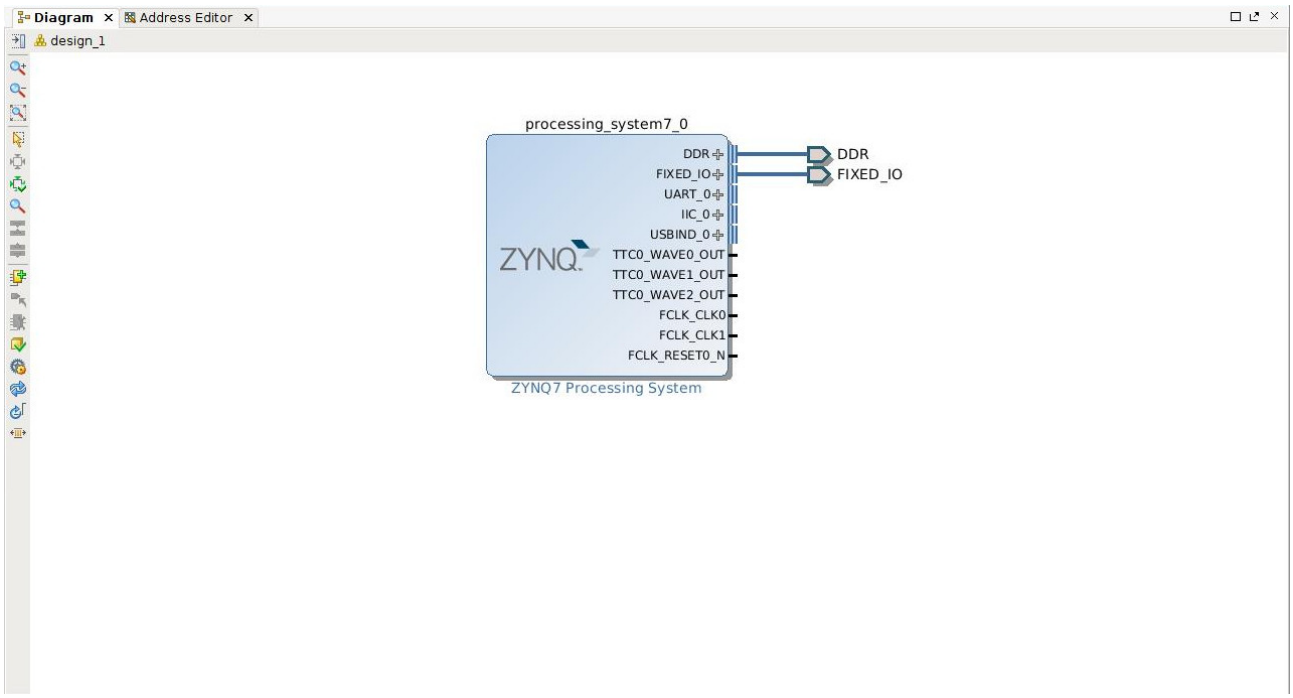


Figure 12: Block automation dialog for the Processing System.

Now it is time to add the vadd IP block to the design but before doing that we need to point out to Vivado where that IP can be found. Find the "IP settings" button in the toolbar within the Diagram view and click it. Then click the "Repository Manager" tab and the plus (+) symbol. Find the "impl" directory of the vadd_OpenCL vivado_hls project and click select. This process is outlined in figure 13.

Clicking "Select" should bring up the "Add Repository" dialog. Just click "OK" and then we are back to the list of IP Repositories, but now augmented with our recently added IP. Click "OK".

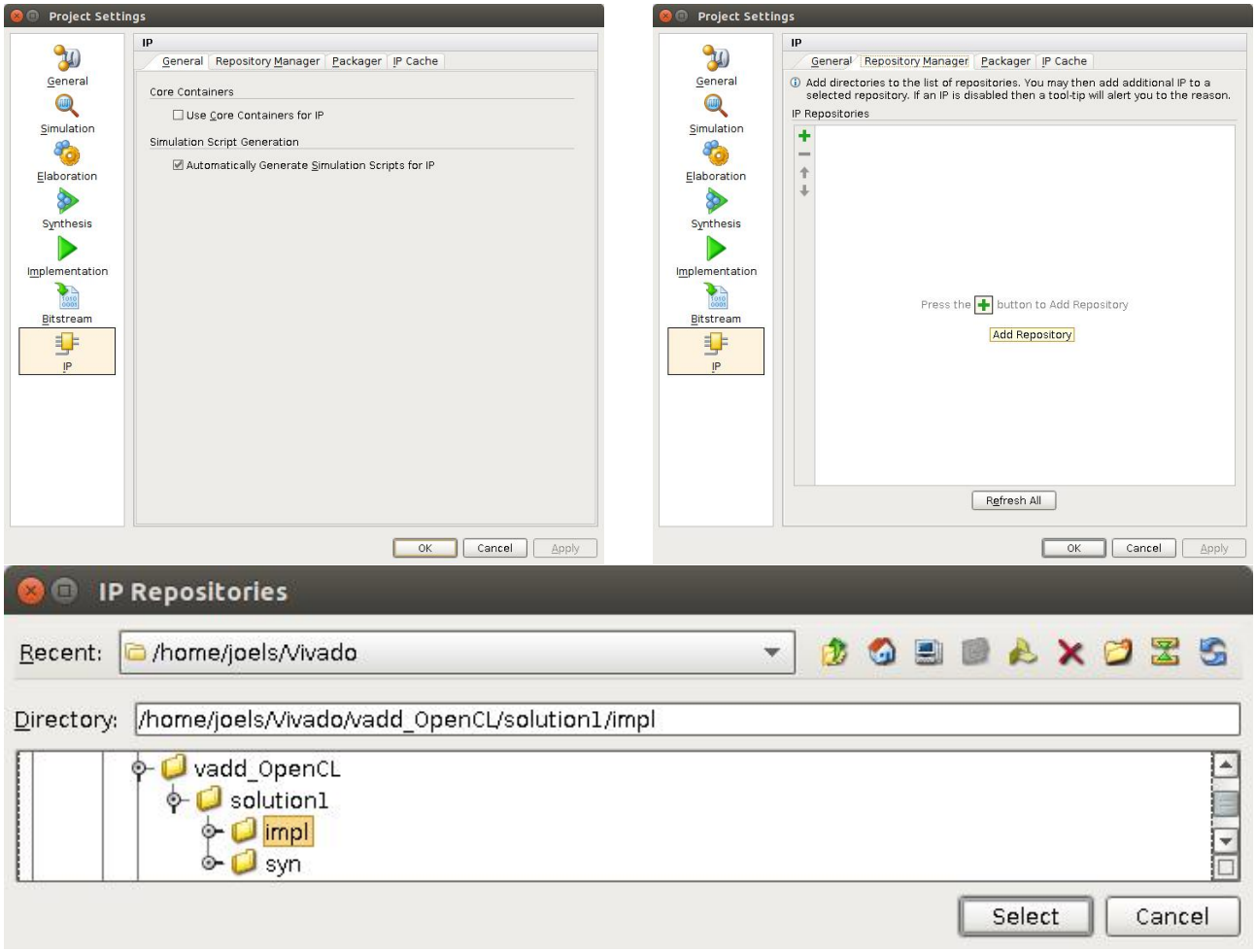


Figure 13: Outline of how to add our custom vadd IP to the IP repository.

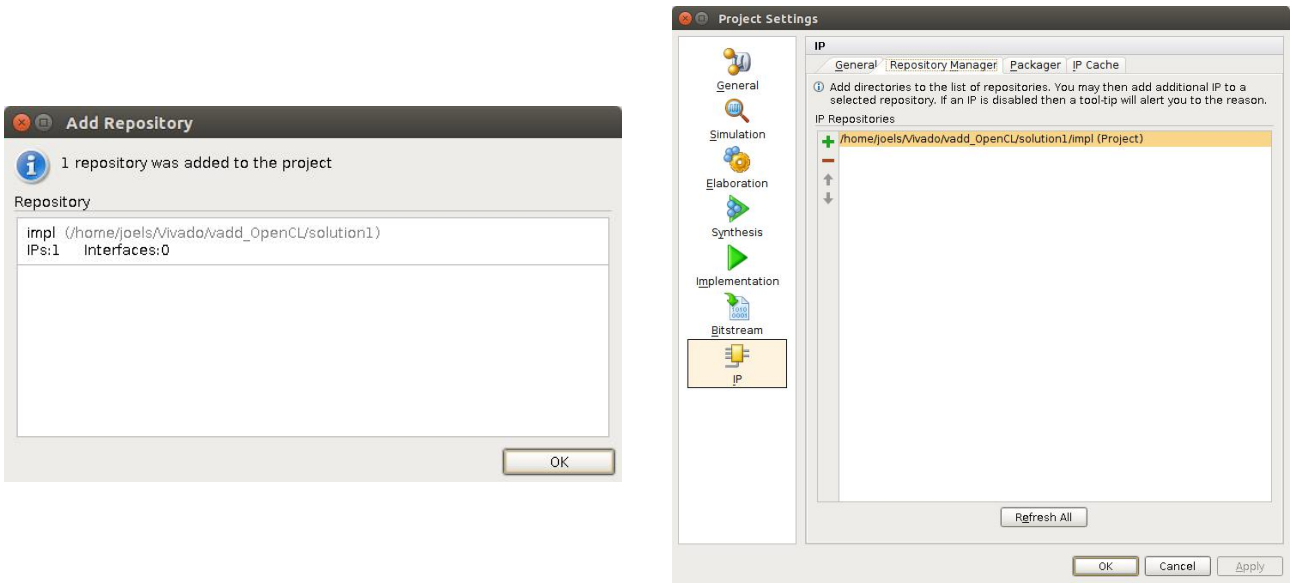


Figure 14: Outline of IP repository configuration continues.

With the IP repository configured we can add the vadd IP to the design. Click the "Add IP" button in the toolbar in the Diagram view. Type "vadd" in the search field and the "Vadd" IP should appear in the list window. Select it.

After adding the Vadd IP the block Diagram should look as in figure 15. Now we need to connect the Vadd unit to the processing system but in order to that we need to go into the processing system block and reconfigure it. If you look at the Vadd_0 unit it has a "s_axi_control" interface and a "m_axi_gmem" interface. These interfaces needs to be connected (as well as the clock and reset). Luckily much of this connecting can be done for us automatically, if only we configure the processing system block correctly.

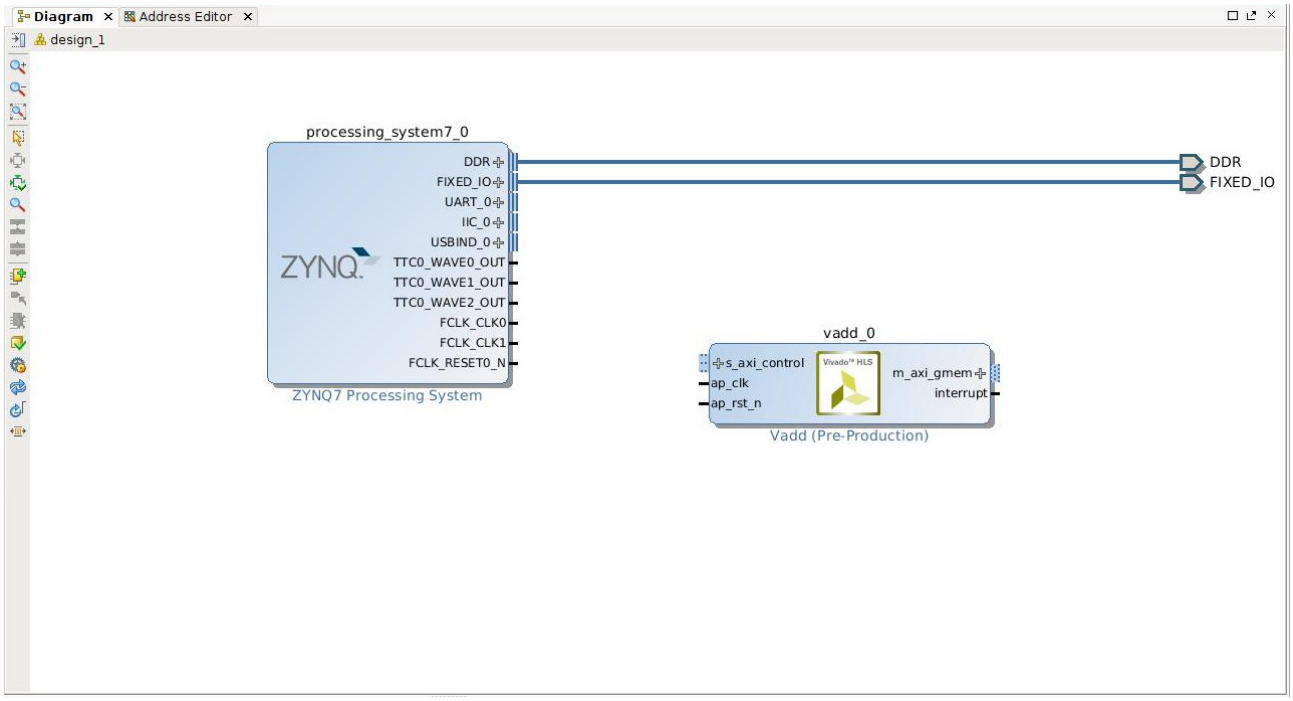
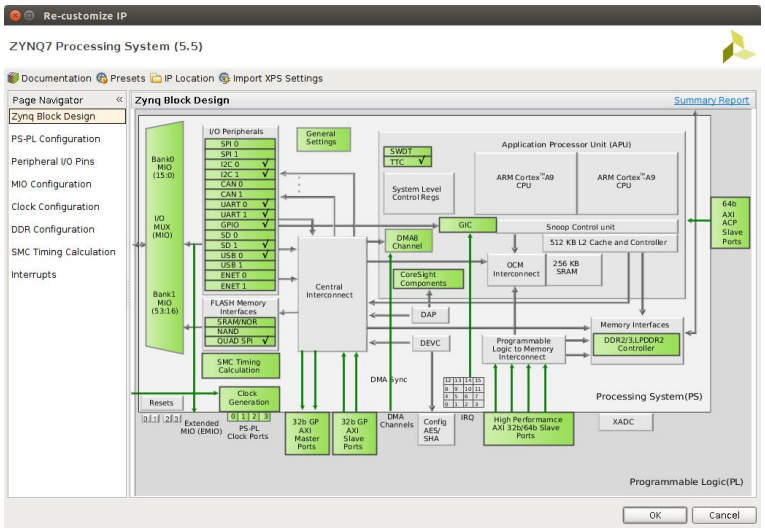


Figure 15: Outline of IP repository configuration continues.

Double click on the ZYNQ7 Processing system in the in the diagram view. This should bring up a view of the internals of the processing system as below:



There are two interfaces that needs to be configured inside the processing system. the “32b GP AXI Master Ports” and the “32b GP AXI Slave Ports”. Double click on the Master ports and configure according to the left side of figure 16. Then do the same for the Slave ports and the right side of figure 16.

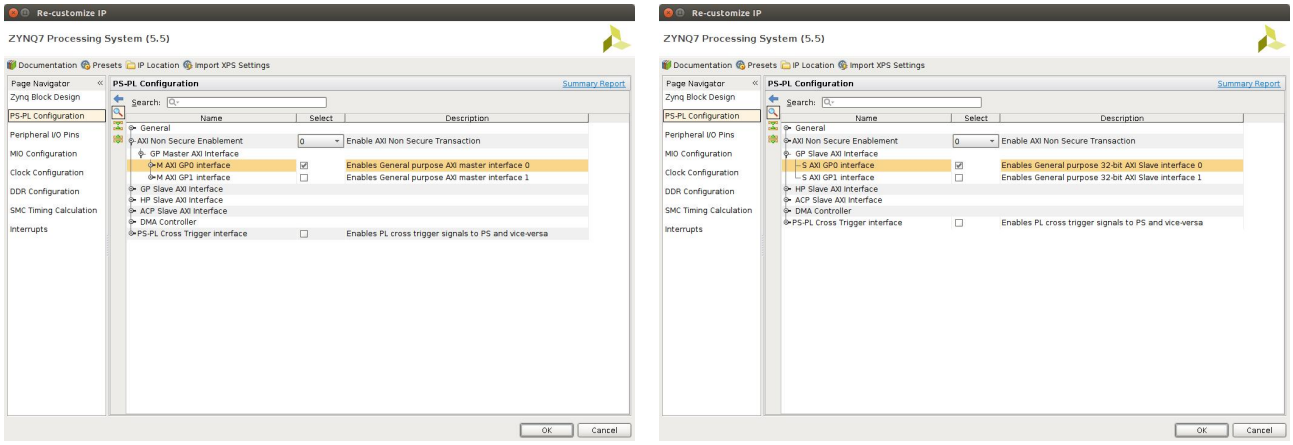


Figure 16: Configuration of the AXI Master/Slave ports.

When this configuration of the processing system is completed the ZYNQ Processing System in the diagram view should show the newly added interfaces. Compare to figure 17.

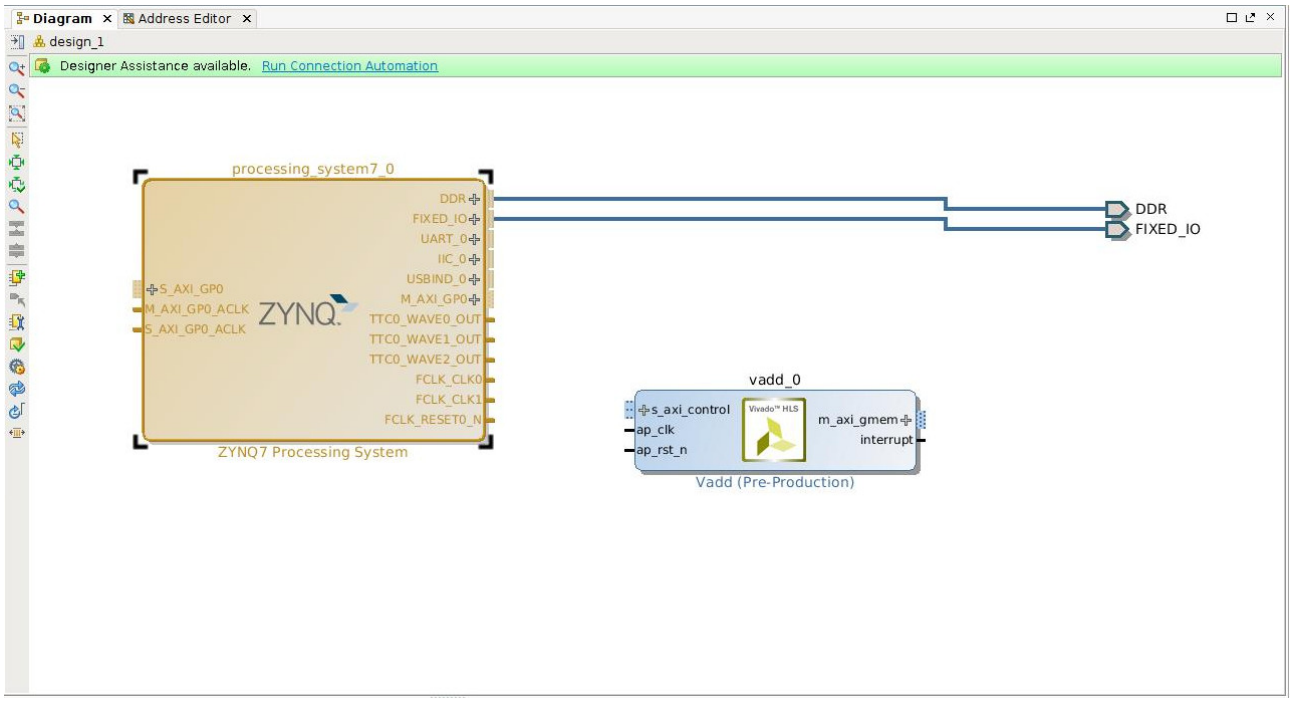
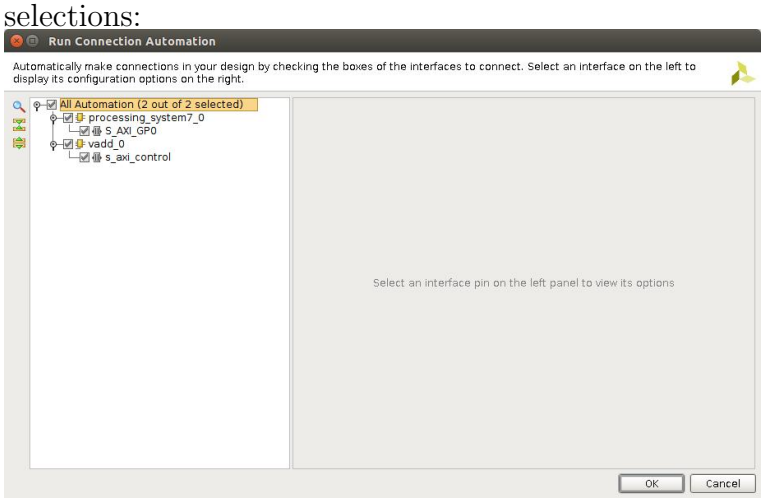


Figure 17: Block diagram view after configuration of the processing system.

Most important here is to note that this change makes the "Run Connection Automation" link to appear at the top of the block diagram view. Click this link and make the following selections:



At this point the diagram view should look similar to figure 18. Notice how two AXI interconnects have been automatically added to the design and connects to the processing system and to the vadd unit. One of these interfaces connect to the control port on the vadd unit and is used to program the vadd unit control registers. The other interface is used by the vadd unit for memory accesses.

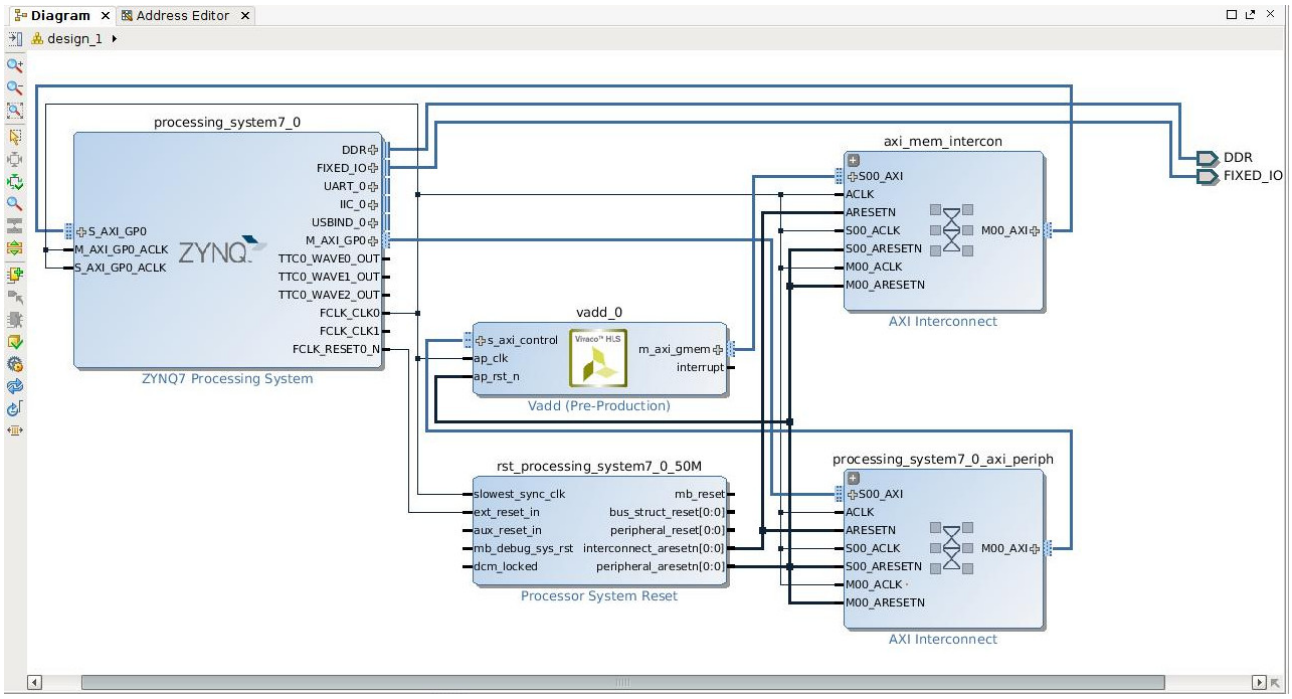
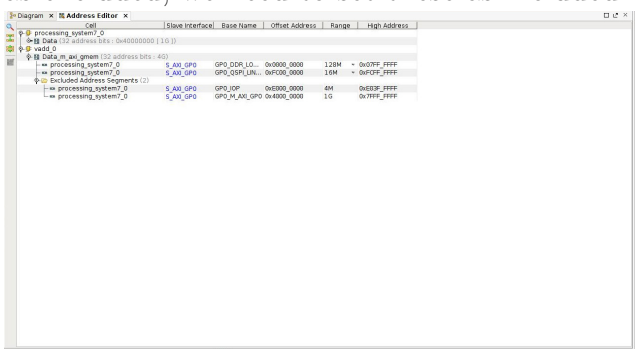
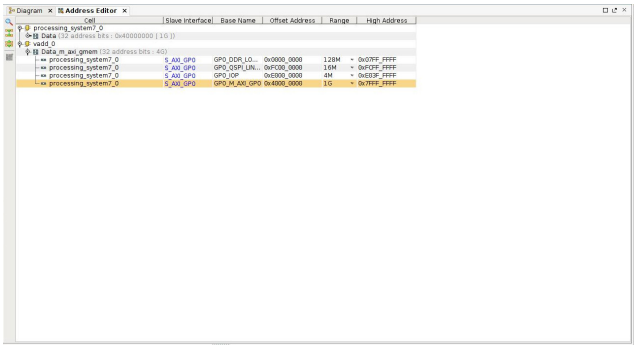


Figure 18: Block diagram view after configuration of the processing system.

At this stage the design is mostly complete. We do, however, need to go into the "Address editor" and do some small tweaks. Within the Address editor some address ranges are listed as excluded, we need to set these as included.

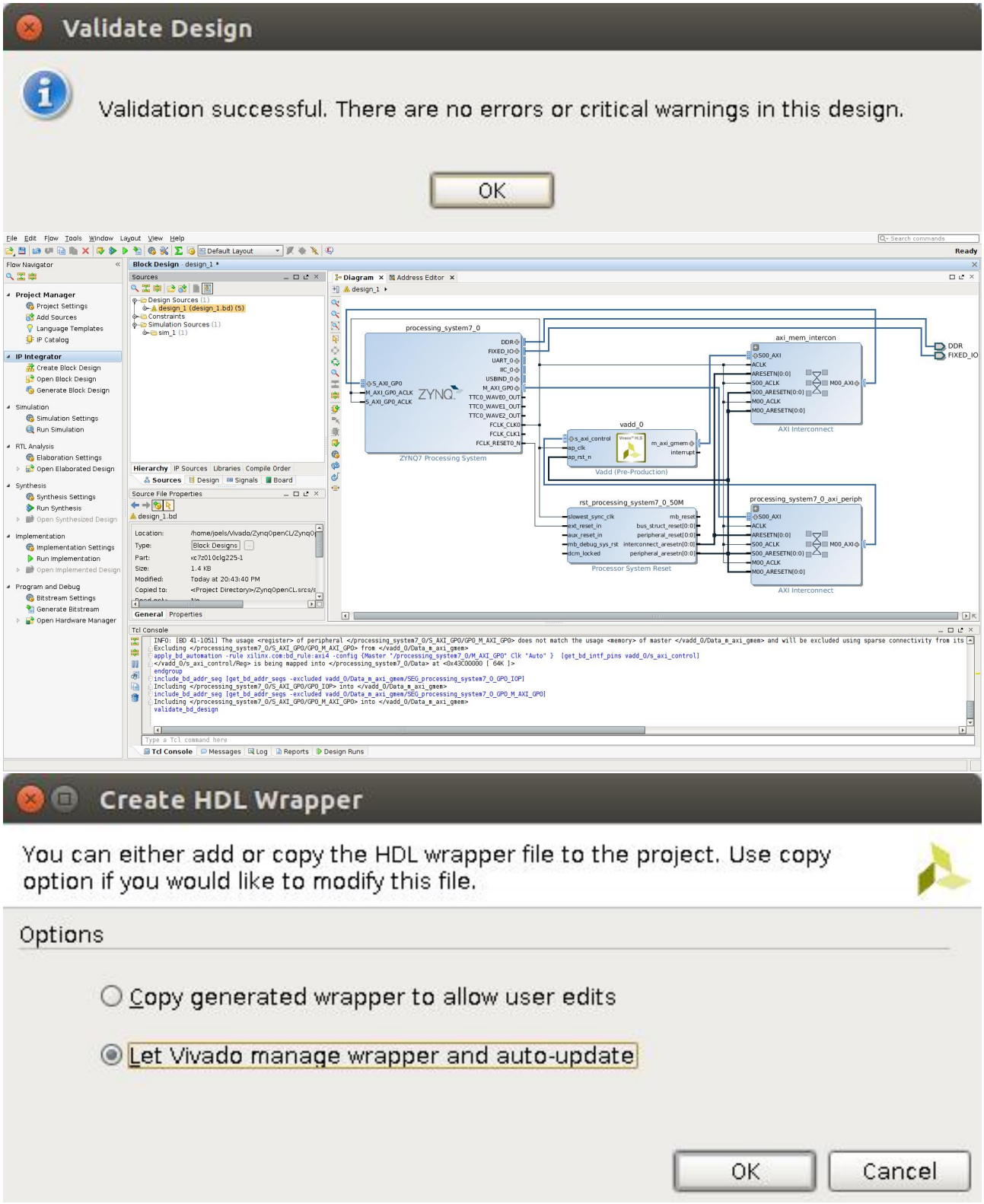


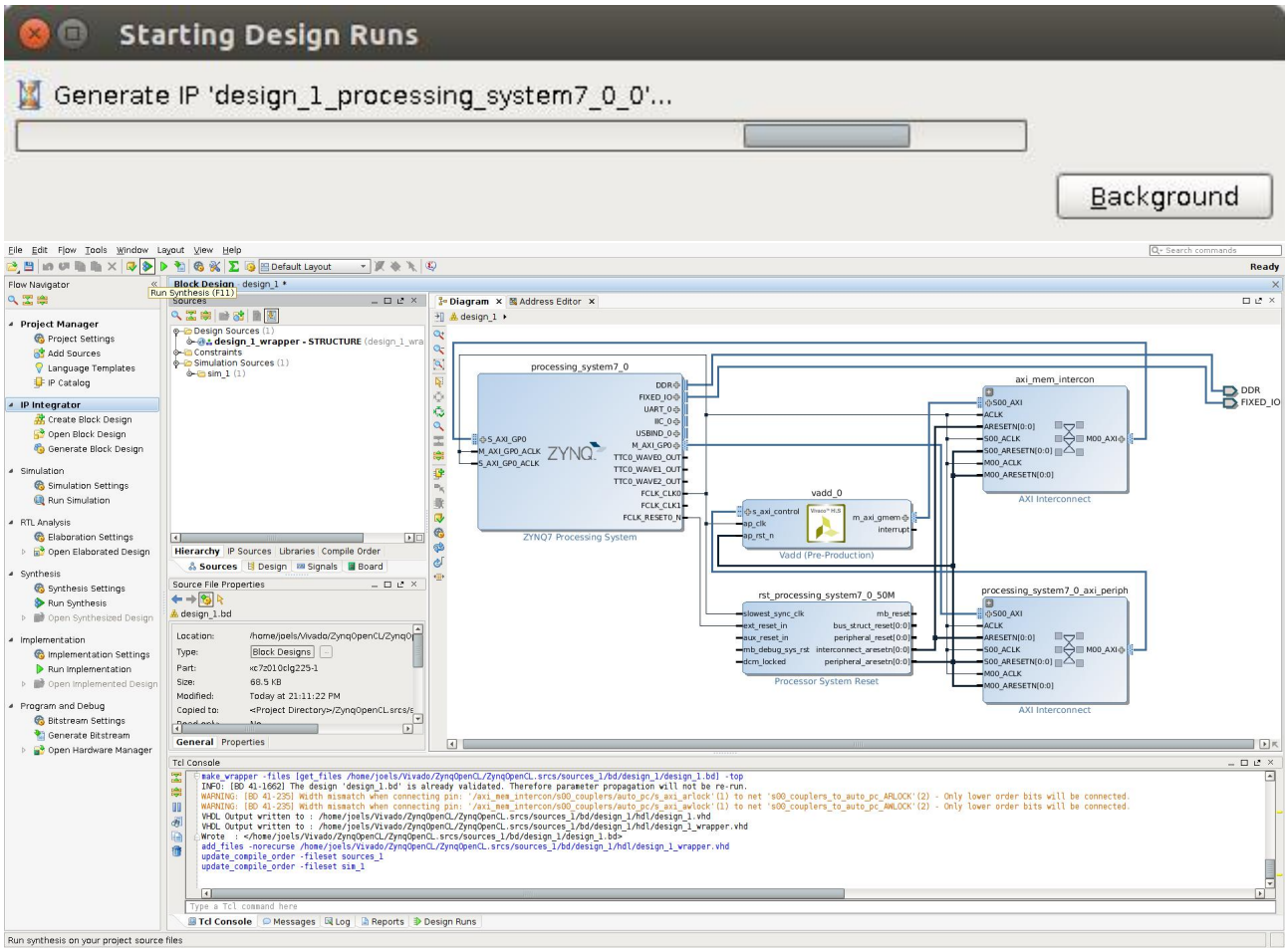


Now it is time to go through the process that in the end results in a bitstream that we can use to program the Zynq FPGA. This procedure is outlined below, in text, and following this in pictures:

- Validate the design: In the tools menu, click Validate design. This should finish with a message saying "Validation successful".
- Create HDL wrappers: This is done in the sources window by right clicking on "design_1" and selecting "Create HDL Wrappers".
- Save and run synthesis: In the flow navigator under synthesis, click run synthesis.
- Synthesis completed: In this dialog choose "Run Implementation".
- Implementation completes: Potentially reporting a great number of critical warnings. Fortunately these can be ignored. One should however check where the warnings come from. If there are warnings directly in relation to the IP you developed yourself it may be important.

- Implementation Completed dialog: Choose “Generate Bitstream”.
- After generation of bitstream one can take a look at the implemented design.





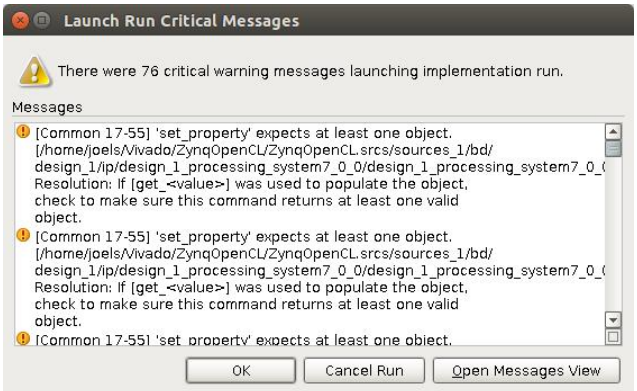


Figure 19: Synthesis completed and an example of critical warnings.

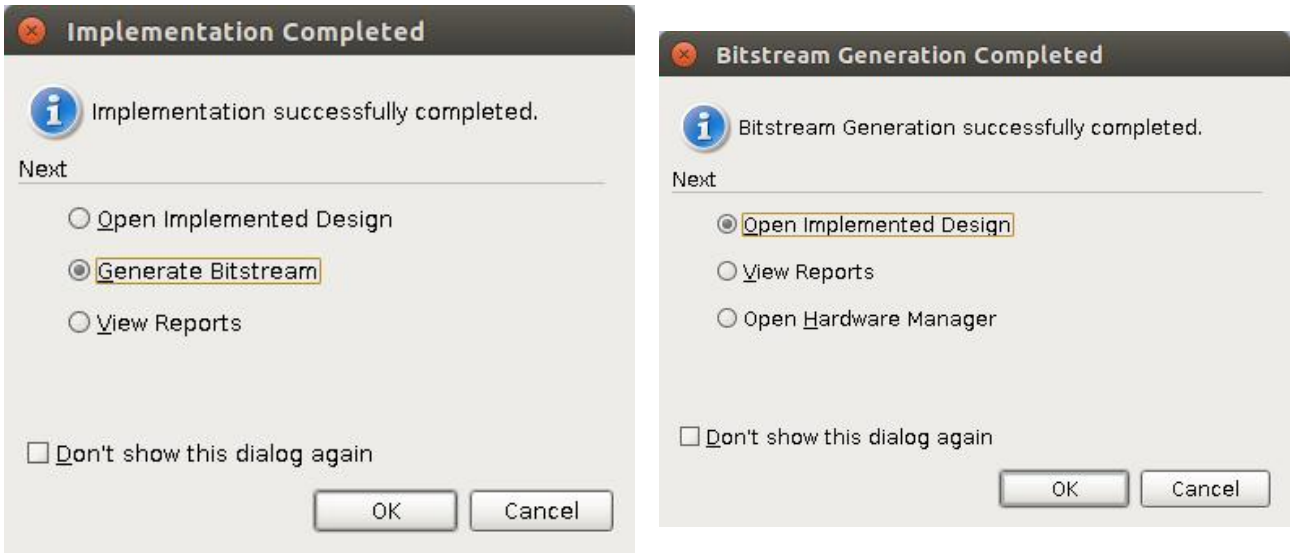
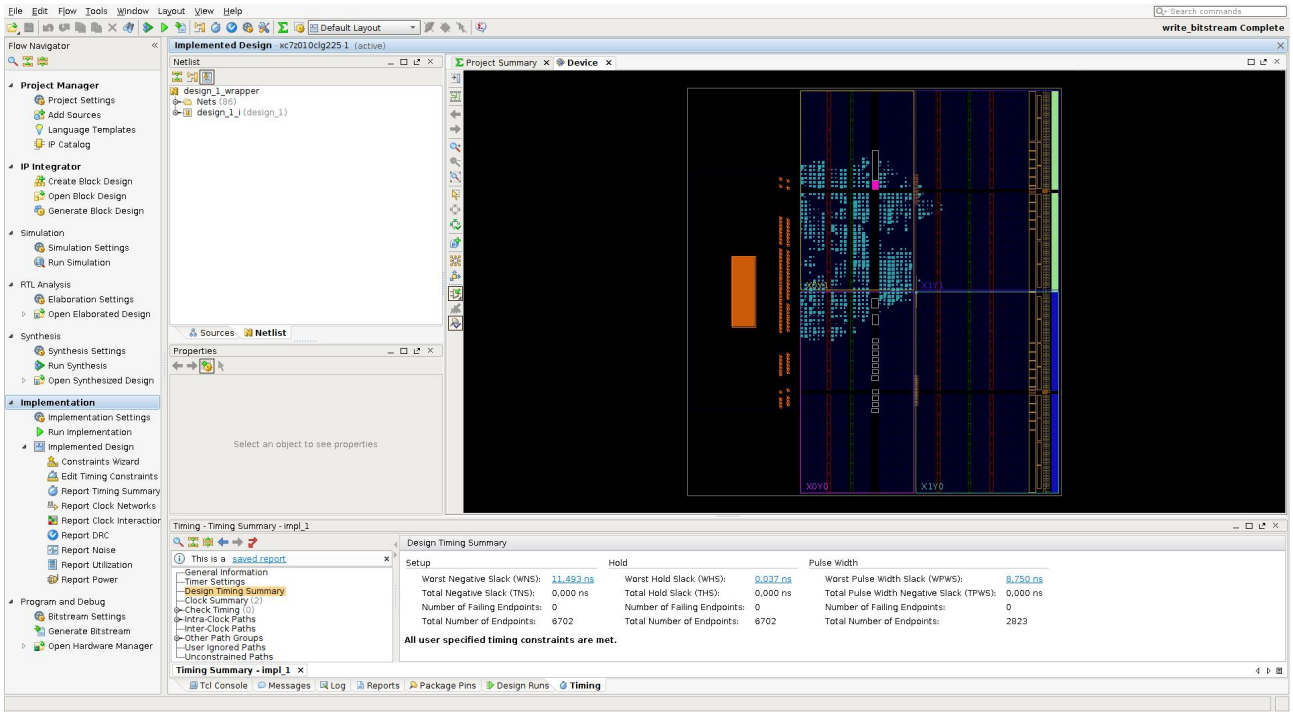


Figure 20: Implementation complete and Bitstream generation completed



At this point we have completed the part of this guide that takes place in Vivado. The next step is to go into the file menu and select “Export Hardware” (include the bitstream) and then to “Launch SDK”.

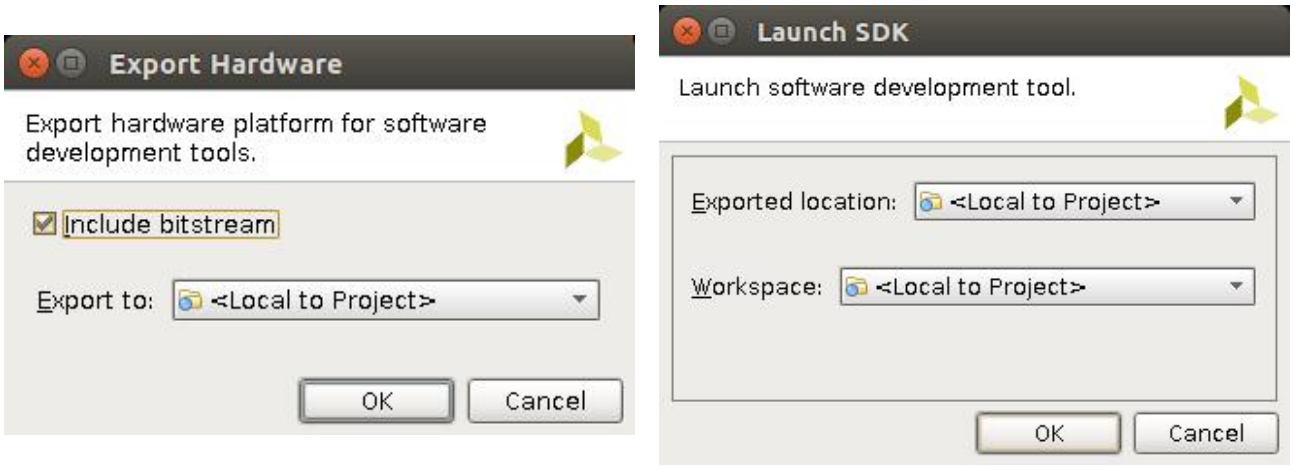
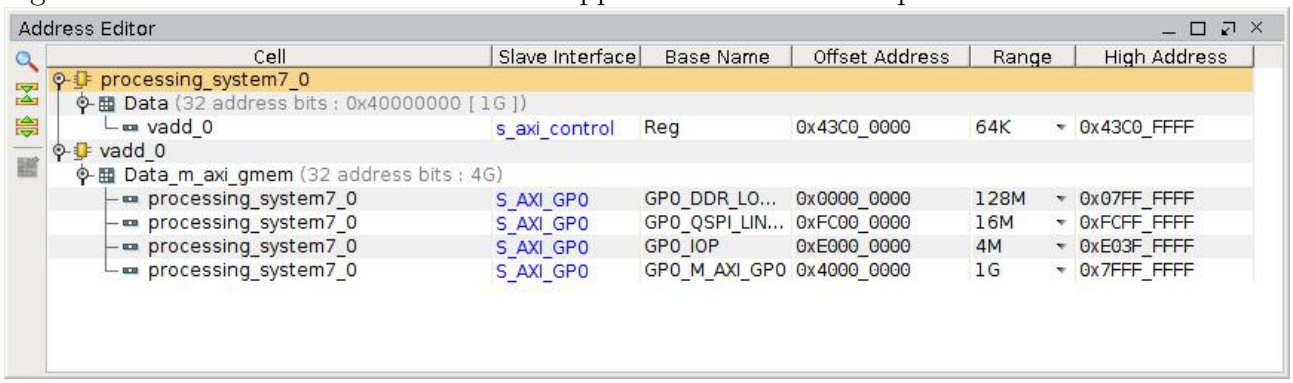


Figure 21: Export hardware and launch the SDK

3.3 Important details from the Address Editor

Before going into the part of the guide that takes place in the SDK we want to point out some details from the Address Editor. Make a note of the address mentioned for the s_axi_control. In this case this address is 0x43C00000. It is on this address and onwards that the control registers for the vadd hardware unit is mapped into the address space.



4 Part 3: Xilinx SDK

When the Xilinx SDK has launched (after launching it from the File menu in Vivado) we are presented with a view like the left part of figure 22. Here we just click the “File” menu and “New” “Application Project”. In the right part of figure 22 we name our application project “HelloOpenCL” and click next and select “hello world” then “Finish” .

Now we need to perform one key piece of configuration to the Board Support Package, the “system.mss” file of the “HelloOpenCL_bsp”. The configuration we need to change is the stdin/stdout under “Overview”, “Standalone”. Both stdin and stdout should be pointed to “ps7_uart_1” and not to uart_0 as per default.

Next we go into the Xilinx tools menu and clicks “Generate linker script”. Here we want to make the heap larger. Find the “Heap Size” box and enter for example 33554432 (for 32mb). The default setting of 1KB will not be enough for what we are going to do.

Now it is time to write the C code that talks to the vadd unit. Edit the “helloworld.c” file in the “HelloOpenCL” project as listed in figure 25.

After writing the code we can right click on the “HelloOpenCL” project in the “Project Explorer” and choose “Debug as” and “Debug Configuration”. In the debug configuration

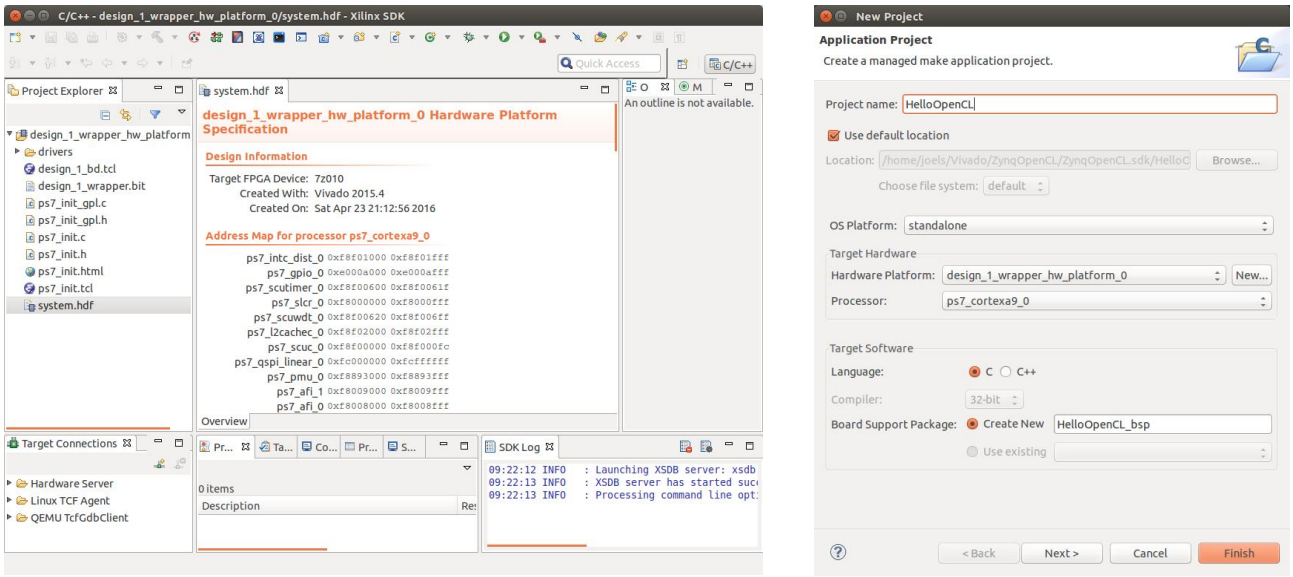


Figure 22: Left: SDK just started. Right: new application project.

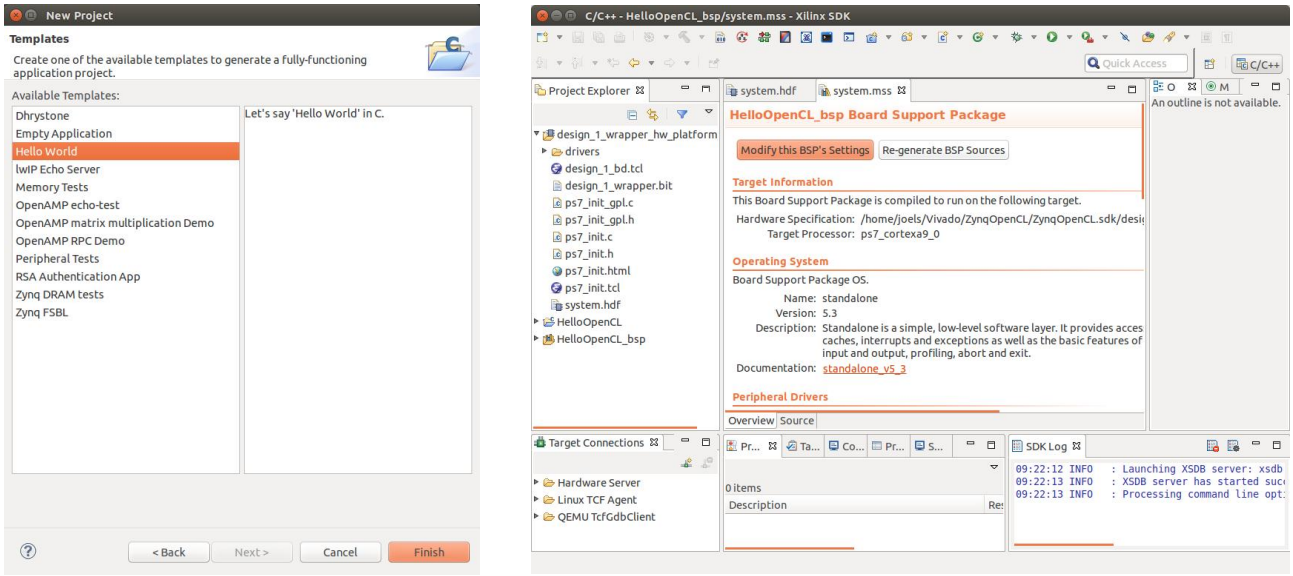


Figure 23: Board support package configuration.

select “Reset entire system” and “Program FPGA” then “Apply”.

Now to start the application in Debug mode right click on “HelloOpenCL” select “Debug as” and “Launch on hardware”. The ZynqBerry should now be connected and its LEDs will be on while the device is being programmed. The SDK will automatically enter into Debug mode and you can press “Resume” (F8) button to run. In order to see any output from the device you need to have a terminal link to it. On linux using the screen command works well: `screen /dev/ttyUSB1`. This part of the procedure is shown in figure 27.

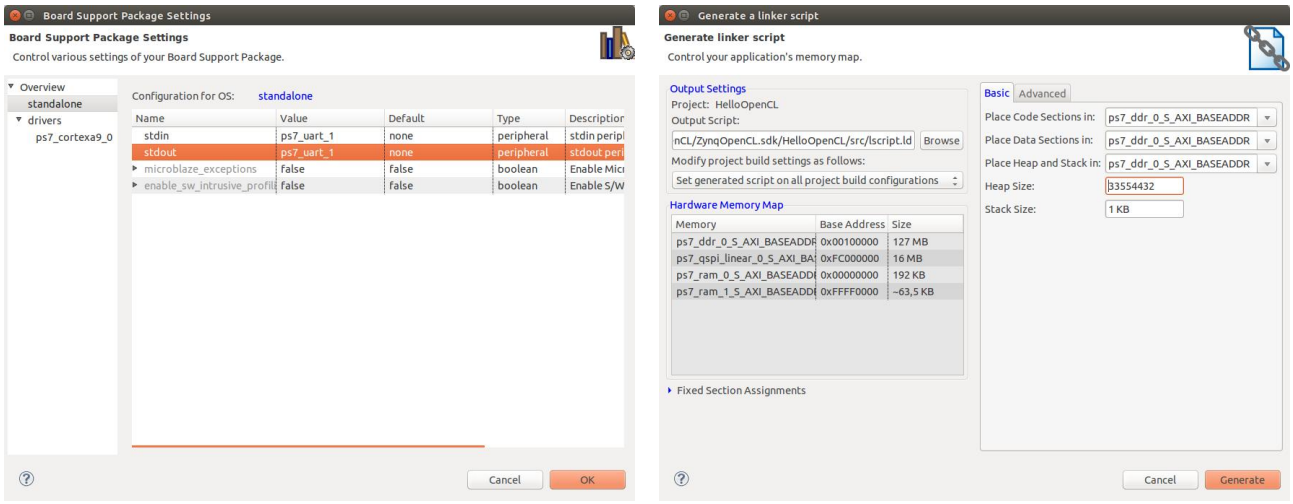


Figure 24: BSP Uart settings and the linker script

```
#include <stdlib.h>
#include "platform.h"

#include "xil_mmu.h"
#include "xil_cache.h"
#include "xil_cache_l.h"

void print(char *str);

volatile char *control = (volatile char*)0x43C00000;

volatile int *wg_x   = (volatile int*)0x43C00010;
volatile int *wg_y   = (volatile int*)0x43C00018;
volatile int *wg_z   = (volatile int*)0x43C00020;
volatile int *o_x    = (volatile int*)0x43C00028;
volatile int *o_y    = (volatile int*)0x43C00030;
volatile int *o_z    = (volatile int*)0x43C00038;

volatile int *a_addr = (volatile int*)0x43C00040;
volatile int *b_addr = (volatile int*)0x43C00048;
volatile int *c_addr = (volatile int*)0x43C00050;


#define WG_SIZE_X 128
#define WG_SIZE_Y 1
#define WG_SIZE_Z 1

int main()
{
    init_platform();
    /* more initialization */
    Xil_SetTlbAttributes(0x43c00000,0x10c06);  /* non cacheable */

    int *a;
    int *b;
    int *c;
    int i;
    int ok = 1;

    a = (int*)malloc(WG_SIZE_X *sizeof(int));
    b = (int*)malloc(WG_SIZE_X *sizeof(int));
    c = (int*)malloc(WG_SIZE_X *sizeof(int));

    print("Generating input data: \n\r");
    for (i = 0; i < WG_SIZE_X; i ++) {
            a[i] = 1;
            b[i] = 2;
            c[i] = 0;
    }
    Xil_DCacheFlush();

    *a_addr = (unsigned int)a;
    *b_addr = (unsigned int)b;
    *c_addr = (unsigned int)c;

    /* set the workgroup identity */
    *wg_y = 0;
    *wg_z = 0;
    *wg_x = 0;

    *o_x = 0;
    *o_y = 0;
    *o_z = 0;


    print("Status of control register: \n\r");
    unsigned int con = *control;
    for (i = 0; i < 8; i ++) {
            if (con & (1  << i) ) {
                    print("1");
            } else {
                    print("0");
            }
    }
    print("\n\r");

    print("Starting OpenCL kernel execution\n\r");
    *control = *control | 1;  /* start */

    /* waiting for hardware to report "done" */
    while (! ((*control) & 2));
    print("DONE!\n\r");


    Xil_DCacheInvalidate();

    for (i = 0; i < WG_SIZE_X; i ++) {
            if (c[i] != 3) ok = 0;
    }

    if (ok) {
            print("Success!\n\r");
    } else {
            print("Error: Something went wrong!\n\r");
    }

    cleanup_platform();
    return 0;
}
```

Figure 25: C code for interfacing with the OpenCL generated hardware.

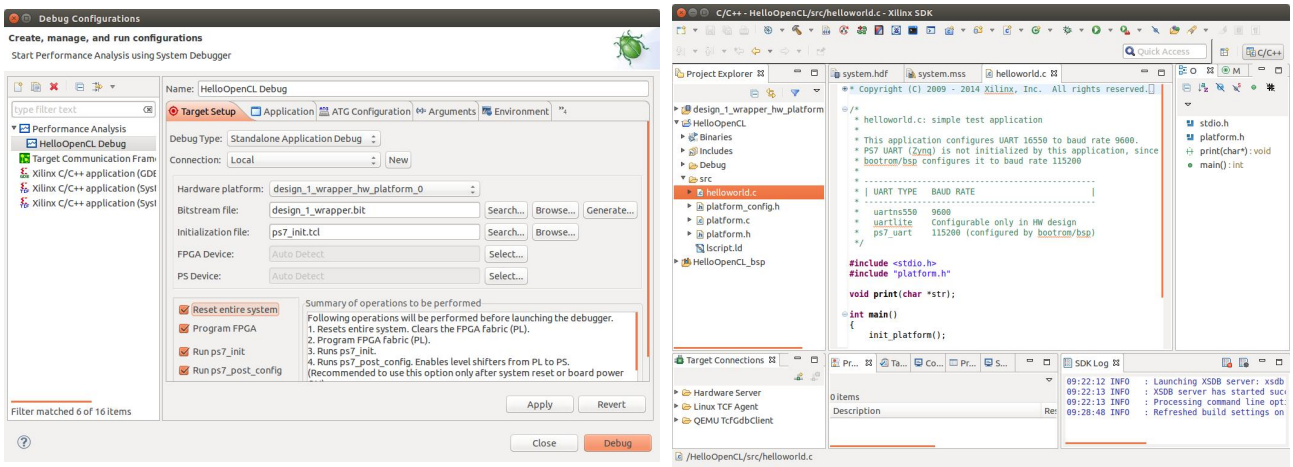


Figure 26: Debug configuration and screen interaction with the ZynqBerry.

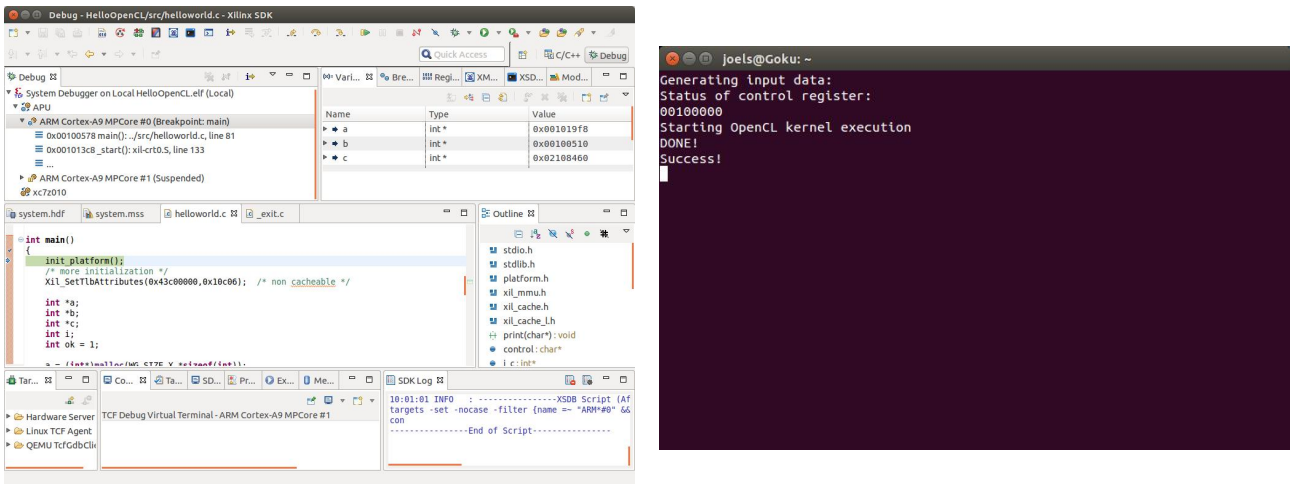


Figure 27: Debug the software and screen interaction.

4.1 C Code Walkthrough

Note that the code shown in figure 25 and gone through in this section, contains the line:

```
Xil_SetTlbAttributes(0x43c00000,0x10c06);  /* non cacheable */
```

in the hope that this would make all read and writes to control registers bypass the cache. Newer versions of the Xilinx libraries have defined a set of names (in xil_mmu.h) for these attribute bit patterns. There the following definition can be found:

```
#define NORM_NONCACHE 0x11DE2
```

So let's for now assume that the correct way to set the registers as non-cacheable is the following: and that the values used in the c code listings are incorrect:

```
Xil_SetTlbAttributes(CONTROL, NORM_NONCACHE);
```

If anyone know the details of this in depth and want to share the knowledge, please write an email to bo.joel.svensson@gmail.com.

The code for interfacing with the generated hardware is given in full in figure 25 but is here given a step by step explanation.

The code starts out by including some headers. This is just shown here for completeness.

```
#include <stdlib.h>
#include "platform.h"

#include "xil_mmu.h"
#include "xil_cache.h"
#include "xil_cache_l.h"

void print(char *str);
```

The code below, declares names for the programming registers. The base address was for this was found in section 3.3 and the offsets to each specific register is found in section 2.4.

```
volatile char *control = (volatile char*)0x43C00000;

volatile int *wg_x   = (volatile int*)0x43C00010;
volatile int *wg_y   = (volatile int*)0x43C00018;
volatile int *wg_z   = (volatile int*)0x43C00020;
volatile int *o_x    = (volatile int*)0x43C00028;
volatile int *o_y    = (volatile int*)0x43C00030;
volatile int *o_z    = (volatile int*)0x43C00038;

volatile int *a_addr = (volatile int*)0x43C00040;
volatile int *b_addr = (volatile int*)0x43C00048;
volatile int *c_addr = (volatile int*)0x43C00050;
```

The workgroup size is 128 (in the x direction). This means that each "run" of the generated hardware will perform 128 element wise additions.

```
#define WG_SIZE_X 128
#define WG_SIZE_Y 1
#define WG_SIZE_Z 1
```

This also means that the smallest amount of additions we can perform using the vadd hardware is 128 and that we can only perform multiples of 128 additions by repeatedly launching work on the vadd hardware with different workgroup identities. This restriction comes the use of the "reqd_work_group_size(128,1,1)" attribute used in the implementation of vadd in vivado_hls. This attribute can be left out resulting in a more flexible (but less efficient) hardware implementation with a more complicated interface.

The `main` function starts out by performing some standard initialization but we also add a step that marks the range of memory containing the programming registers as "non cacheable".

```
int main()
{
    init_platform();
    /* more initialization */
    Xil_SetTlbAttributes(0x43c00000,0x10c06);  /* non cacheable */
```

The following piece of code declares pointers and allocates memory for the input and output to the vadd computation. It also declares a counter variable i (used in some loops later on) and an ok status variable.

```
    int *a;
    int *b;
    int *c;
    int i;
    int ok = 1;

    a = (int*)malloc(WG_SIZE_X *sizeof(int));
    b = (int*)malloc(WG_SIZE_X *sizeof(int));
    c = (int*)malloc(WG_SIZE_X *sizeof(int));
```

Generate some input data and flush the cache to ensure that all the data we generated has been stored all the way to DRAM before launching the vadd computation.

```
    print("Generating input data: \n\r");
    for (i = 0; i < WG_SIZE_X; i ++) {
            a[i] = 1;
            b[i] = 2;
            c[i] = 0;
    }
    Xil_DCacheFlush();
```

The next step is to program the registers of the vadd unit and prepare for launching a workgroup. The workgroup id is set to (0,0,0).

```
*a_addr = (unsigned int)a;
*b_addr = (unsigned int)b;
*c_addr = (unsigned int)c;

/* set the workgroup identity */
*wg_y = 0;
*wg_z = 0;
*wg_x = 0;

*o_x = 0;
*o_y = 0;
*o_z = 0;
```

The next piece of code prints the contents of the control register. This serves no important purpose for the application but only provides a way to visually inspect that the control status (which should be "idle").

```
print("Status of control register: \n\r");
unsigned int con = *control;
for (i = 0; i < 8; i ++) {
        if (con & (1  << i) ) {
                print("1");
        } else {
                print("0");
        }
}
print("\n\r");
```

We instruct the vadd hardware to start computing by putting a one at bit position zero in the control register.

```
print("Starting OpenCL kernel execution\n\r");
*control = *control | 1;  /* start */
```

And then we wait for the hardware to report done in bit position two.

```
/* waiting for hardware to report "done" */
while (! ((*control) & 2));
print("DONE!\n\r");


Xil_DCacheInvalidate();
```

After the hardware reports to be done, we invalidate the cache of the processing system in order to ensure that we will see the fresh data that the programmable logic has computed (without any involvement of the cache hierarchy, so the changes in memory are not yet visible to the ARM cores).

After that we can check the result for correctness.

```
    for (i = 0; i < WG_SIZE_X; i ++) {
            if (c[i] != 3) ok = 0;
    }

    if (ok) {
            print("Success!\n\r");
    } else {
            print("Error: Something went wrong!\n\r");
    }
```

And we are done.

```
    cleanup_platform();
    return 0;
}
```

5 Conclusion

We hope that following this guide has allowed you to run OpenCL on a Zynq device. Please send us feedback or questions.