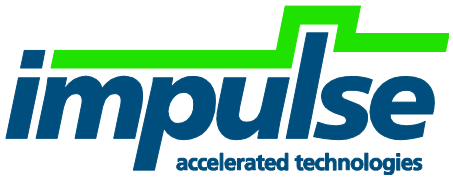


CoDeveloper Platform Support Package

Pico Computing M501 PSP User Guide – Linux Version 1.1.1

Impulse Accelerated Technologies, Inc.

www.ImpulseAccelerated.com

1.0 Table of Contents

2.0 Table Of Figures

3.0 Revision History

Date	Version	Description	Author
11/3/2012	1.0	Initial Creation	Shaumil Dave
11/10/2012	1.1	Added Memtest description	Shaumil Dave
2/5/2013	1.1.1	Added no-EOS support to limitations, EOS has never been supported	Ed Trexel

4.0 Overview

This user guide covers the CoDeveloper Platform Support Package (PSP) for the Pico Computing M501 module (referred hereon simply as “M501”). Highlights for this PSP include:

- Automatic creation of a complete ready-to-build Xilinx ISE project upon exporting hardware for creating FPGA binary ‘.bit’ file referred to as the “bitfile”. The bitfile may be built under Windows or Linux and either via the ISE GUI or from command line using a build script.
- Application executable built via Makefile on a Linux platform.
- Loading of the FPGA on the M501 via host CPU over PCIe.

4.1. Hardware Block Diagram

4.1.1. Pico Computing Block Diagram

Below is a diagram of the M50X platform:

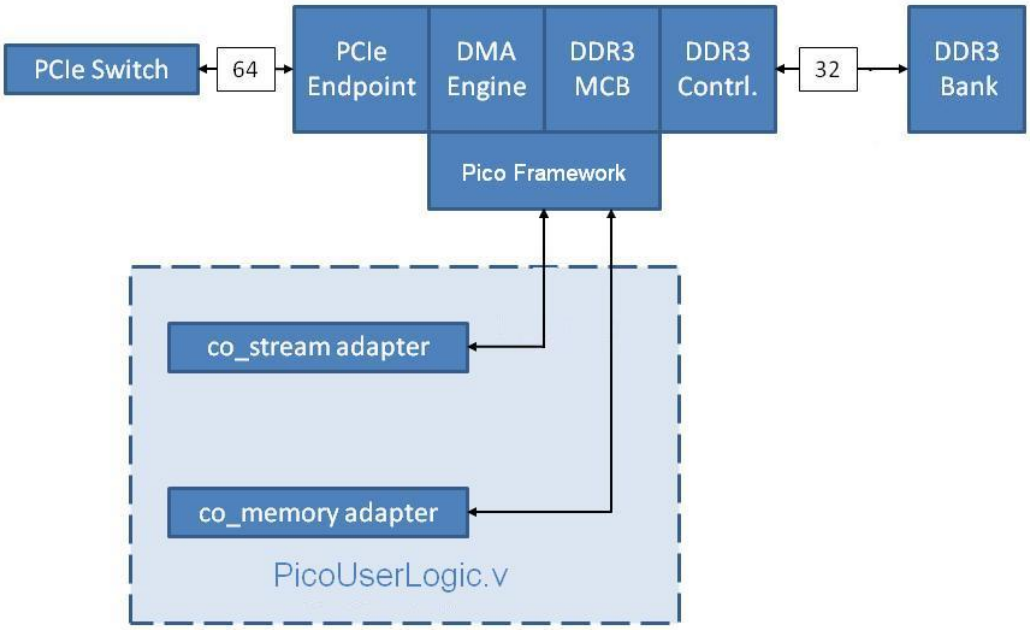


Figure 1 – Firmware Architecture (Pico M50X Series Platform Support Package Users Guide)

The only elements of the architecture the user has control of are those contained inside the Verilog module “PicoUserLogic.v”. All of the logic generated by Impulse CoDeveloper and any external HDL modules must be instantiated inside this top level user module

5.0 Before Getting Started: Read This First

Before getting started, please ensure that you have obtained and installed all the necessary software tools, additional files, and hardware as described below.

5.1. Hardware limitations:

1. co_streams require all data transfers to be a multiple of 128-bits/16bytes
2. co_streams to/from host do not support End-Of-Stream (EOS). Using co_stream_close() on a stream to/from host will either cause extra data being sent to the host or the hardware process to lockup.
3. co_memory (not used in Passthrough example) use limitations:
    a. The memory interface requires that memory transfers begin at an address with 256-byte alignment
    b. The memory interface assumes that the transfer size in bytes is a multiple of 16

5.2. Required Software Tools:

- Impulse CoDeveloper v3.70.e.6 or newer for the Development System.
- NOTE: The use of floating point for Virtex-6 and newer devices requires the use of Xilinx's v5.0 CORE Generator cores which is supported via a patch to CoDeveloper made available via the 'XilinxFPv5BetaPatch' link under the supplied Pico PSP link.
- Xilinx ISE 13.4 for Windows (exactly, not newer) for the Development System.
- Windows 7 for Impulse CoDeveloper and Xilinx ISE 13.4 (the Development System).
- Ubuntu (Debian) 12.04 LTS for the Host System (With EX500 PCIe card and either M501 or M503 FPGA modules)
- Pico Linux installer package version 5.0.6.1 or newer for the Host System:
    - Go to http://picocomputing.com/support/software
    - Select *Linux Installer (M-501, M-503, M-505)*
        - Linux_5.0.6.1_all.deb

5.3. Additional Required Files

The following Examples files are not included with the installation of the software tools and are required for development using this PSP. They include the CoDeveloper project file (*.icProj), associated design files (*_hw.c, *_sw.c, *.h), and data files (*.dat) used as stimulus. A link to download the M5xx PSP and Examples files should have already been provided, if not please email support@impulsec.com to request one.

Examples
- **M5XX**
  - **Passthrough**
    - filter_in.dat
    - passthrough.icProj
    - passthrough_hw.c
    - passthrough_sw.c
    - passthrough.h

CoDeveloper will use the project file, “passthrough.icProj”, to generate HDL as well as a software application to load the FPGA image.

5.4. Required Hardware

The following hardware is required for development using this PSP:

- EX500 x16 full length full height PCIe FPGA Development Board.
- M501 or M503 FPGA module.
- Host System with Ubuntu 12.04 LTS 64-bit OS based development PC for running all tools – Recommended: 100GB disk space available for tools installation and 12GB RAM.
- Host System that has an Intel Motherboard with x58 chipset and available x16 (physical) Gen 2 PCIe express slot. Consideration for your PCIe video card must be given if it is a x16 video card. The motherboard must accommodate (not a shared resource) independent x16 Gen 2 PCIe slots.
- Available 12 volt PCIe power supply for the M501 FPGA development board in the Host System. Consideration for your video card must be given if it also requires a separate 12 volt power connection.
- Development System for installation of CoDeveloper and Xilinx ISE 13.4.
  - Impulse CoDeveloper and M50x PSP: Windows only (32 or 64-bit)
  - Xilinx ISE 13.4: Windows 64-bit or Linux 64-bit

6.0 Host System Setup (Linux)

6.1. Install Pico M501 driver

After the Pico M501 and EX-xxx carrier are installed, the user must install the drivers.

1. After downloading the Pico Linux installer package (Linux_*.deb), follow the "Linux_PicoGettingStarted.pdf" to install Linux software, firmware, and the driver for the M50x.

6.2. Running ISE on Linux

6.2.1. Install Xilinx ISE 13.4

1. Please see vendor supplied documentation for installation.
2. Please see vendor supplied documentation for licensing.

6.3. Running ISE on Windows

6.3.1. Copy Pico Installer source from Host PC to Development PC

If Xilinx ISE will be run on Windows, prepare to copy the directory "/usr/src/picocomputing-5.0.6.1" from the Linux Host PC to the Windows Development system. For example, save the directory to a USB flash drive.

7.0 Development System Setup (Windows)

7.1. Install Impulse CoDeveloper v3.70.e.6 or newer

NOTE: The use of floating point for Virtex-6 and newer devices requires the use of Xilinx's v5.0 CORE Generator cores which is supported via a patch to CoDeveloper made available via the 'XilinxFPv5BetaPatch' link under the supplied Pico PSP link

1. Add the Pico M5XX PSP to the CoDeveloper installation.
    a. Copy the supplied "**Architectures**" directory to "**Impulse\CoDeveloper3**".

    Architectures

    Pico

    pico_m501_linux_vhdl.xml

    pico_m503_linux_vhdl.xml

2. Copy the supplied "**Examples**" directory to a working directory on the development PC for access to the pre-built example files.
3. Download the latest version 3.x and installation note
    a. **http://www.impulseaccelerated.com/ReleaseFiles/**
    b. (optional) Installation of floating point support is via unzipping the .zip file (password: impulsefpv5beta) into 'Impulse' after each CoDeveloper installation. Please see enclosed README file for specific notes on the patch.

7.2. Running ISE on Windows

7.2.1. Install Xilinx ISE 13.4

1. Please see vendor supplied documentation for installation.
2. Please see vendor supplied documentation for licensing.

7.2.2. Copy the Linux Pico Installer

1. If Xilinx ISE is to be run on Windows: From the Host PC, copy "/usr/src/picocomputing-5.0.6.1" to "c:\usr\src\picocomputing-5.0.6.1"

7.2.3. Configure Environment Variables

1. Add environment variable PICOBASE = "c:\usr\src\picocomputing-5.0.6.1"
2. Add environment variable XILINX = "c:\Xilinx\13.4\ISE_DS\ISE"
3. Add environment variable XILINX_BASE = "c:\Xilinx\13.4"

8.0 Passthrough Example and Tutorial

"Passthrough" is provided as an example that may be used for quickly creating user applications and for the purpose of a tutorial showing the steps involved to go from an Impulse C application in CoDeveloper all the way through to a Xilinx ISE 13.4-compiled FPGA binary and target application executable. The base files required for recreating the example using this tutorial are provided within the Impulse supplied examples which needs to be copied to a working directory on the development PC in order to run the tutorial.

NOTE: Ensure there are no spaces (' ') in the directory path chosen to avoid potential path issues with any of the tools.

The Passthrough example's hardware process is "Passthrough()", located in the source file "Passthrough_hw.c". It performs the following operations:

1) Read value from co_stream "input_stream"
2) Write value to co_stream to "output_stream"

   NOTE: The hardware code runs continuously

8.1. Prerequisites

The tutorial in this Platform Support Package assumes that you have read and understand the introductory sections of the CoDeveloper User's Guide, installed with CoDeveloper and accessed from the Help menu. In particular, you should take the time to go through the tutorials provided with CoDeveloper so you have a good understanding of the front-end design flow including both desktop software simulation and hardware compilation.

8.2. CoDeveloper Project Files

The Passthrough example CoDeveloper project is made up of the following files:

- Passthrough.icProj – CoDeveloper project file
- Passthrough_hw.c – Source code for hardware process
- Passthrough_sw.c – Source code for software processes
- Passthrough.h – Header file that defines the width of the stream

When you define the width of the steam, you must make the changes in the header file as well as the in the passthrough_hw.c file. The default example defines the steam to be the maximum width of 64 bit data bus.

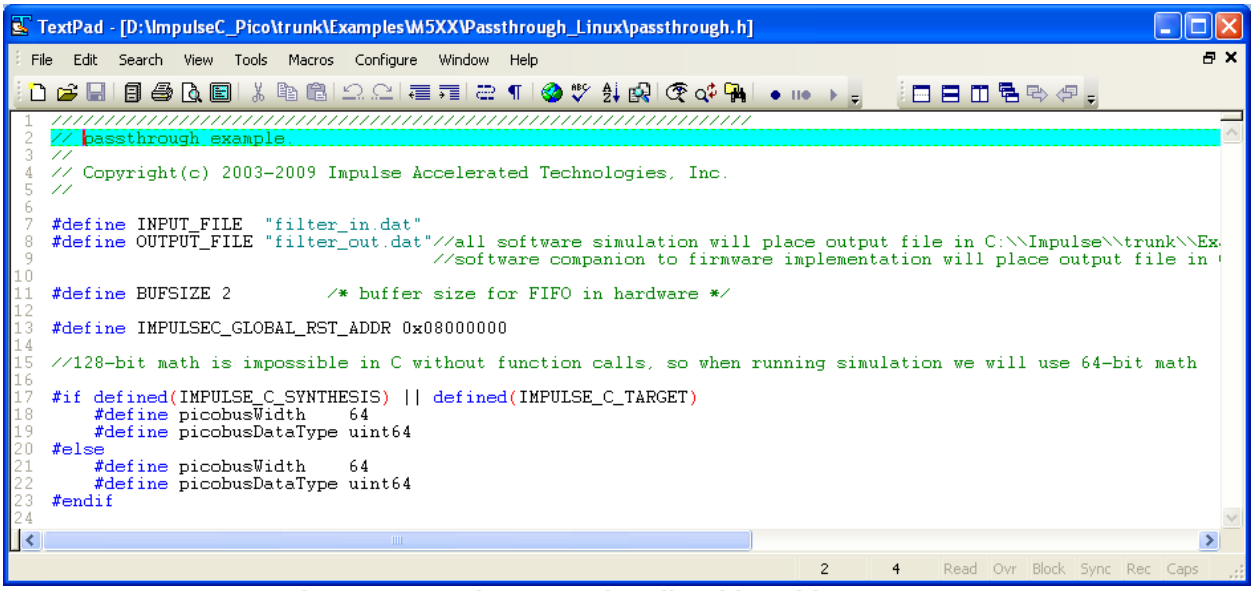
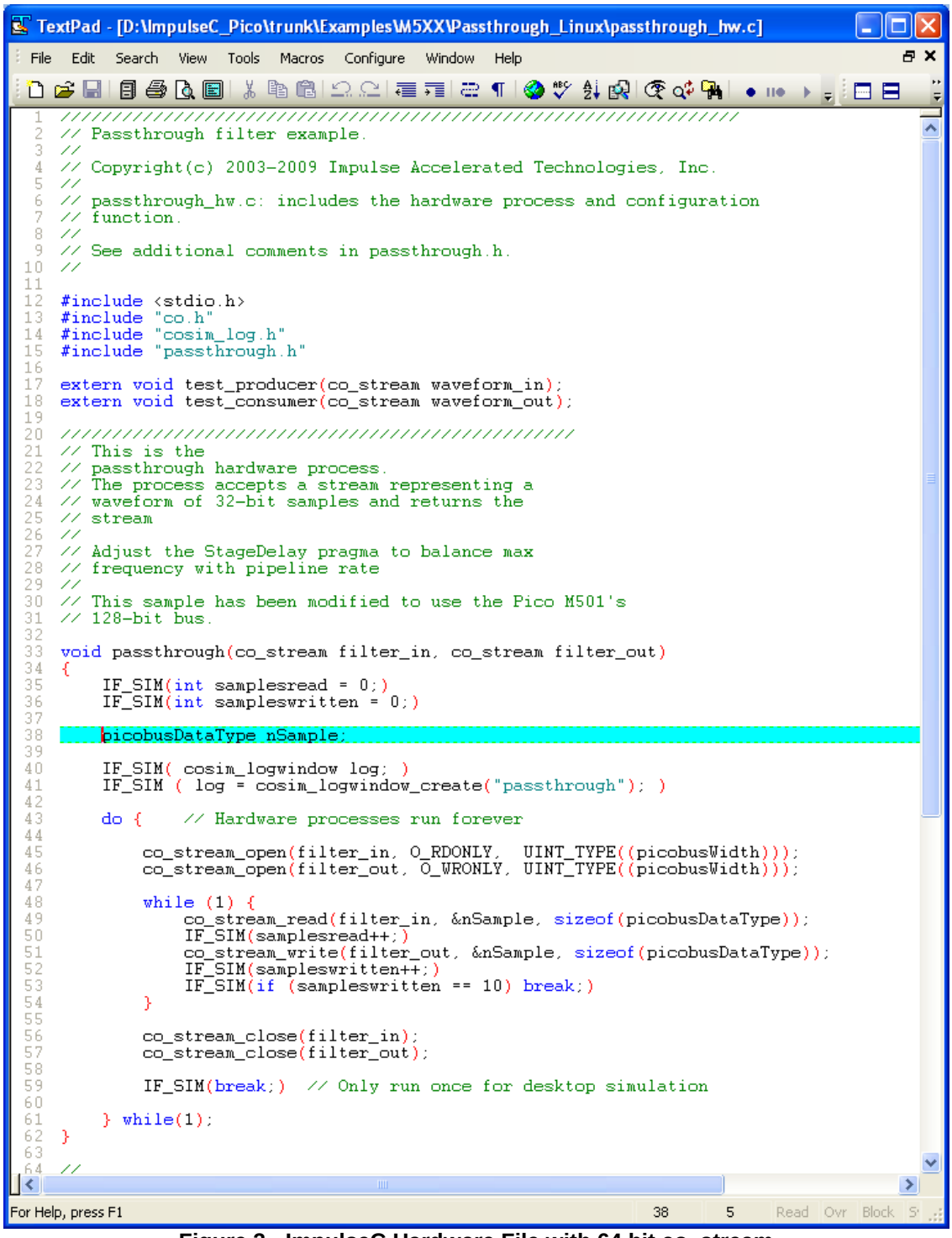


Figure 2 - Impulse C Header File with 64 bit co_stream




```
1  ////////////////////////////////////////////////////////////////////////
2  // Passthrough filter example.
3  //
4  // Copyright(c) 2003-2009 Impulse Accelerated Technologies, Inc.
5  //
6  // passthrough_hw.c: includes the hardware process and configuration
7  // function.
8  //
9  // See additional comments in passthrough.h.
10 //
11
12 #include <stdio.h>
13 #include "co.h"
14 #include "cosim_log.h"
15 #include "passthrough.h"
16
17 extern void test_producer(co_stream waveform_in);
18 extern void test_consumer(co_stream waveform_out);
19
20 ////////////////////////////////////////////////
21 // This is the
22 // passthrough hardware process.
23 // The process accepts a stream representing a
24 // waveform of 32-bit samples and returns the
25 // stream
26 //
27 // Adjust the StageDelay pragma to balance max
28 // frequency with pipeline rate
29 //
30 // This sample has been modified to use the Pico M501's
31 // 128-bit bus.
32
33 void passthrough(co_stream filter_in, co_stream filter_out)
34 {
35     IF_SIM(int samplesread = 0;)
36     IF_SIM(int sampleswritten = 0;)
37
38     picobusDataType nSample;
39
40     IF_SIM( cosim_logwindow log; )
41     IF_SIM ( log = cosim_logwindow_create("passthrough"); )
42
43     do {    // Hardware processes run forever
44
45         co_stream_open(filter_in, O_RDONLY,  UINT_TYPE((picobusWidth)));
46         co_stream_open(filter_out, O_WRONLY, UINT_TYPE((picobusWidth)));
47
48         while (1) {
49             co_stream_read(filter_in, &nSample, sizeof(picobusDataType));
50             IF_SIM(samplesread++;)
51             co_stream_write(filter_out, &nSample, sizeof(picobusDataType));
52             IF_SIM(sampleswritten++;)
53             IF_SIM(if (sampleswritten == 10) break;)
54         }
55
56         co_stream_close(filter_in);
57         co_stream_close(filter_out);
58
59         IF_SIM(break;)  // Only run once for desktop simulation
60
61     } while(1);
62 }
63
64 //
```

Figure 3 - ImpulseC Hardware File with 64 bit co_stream

8.3. Opening Project

Open the CoDeveloper project file ‘Passthrough.icProj’ by selecting and pressing ‘Enter’ or by double-clicking it:

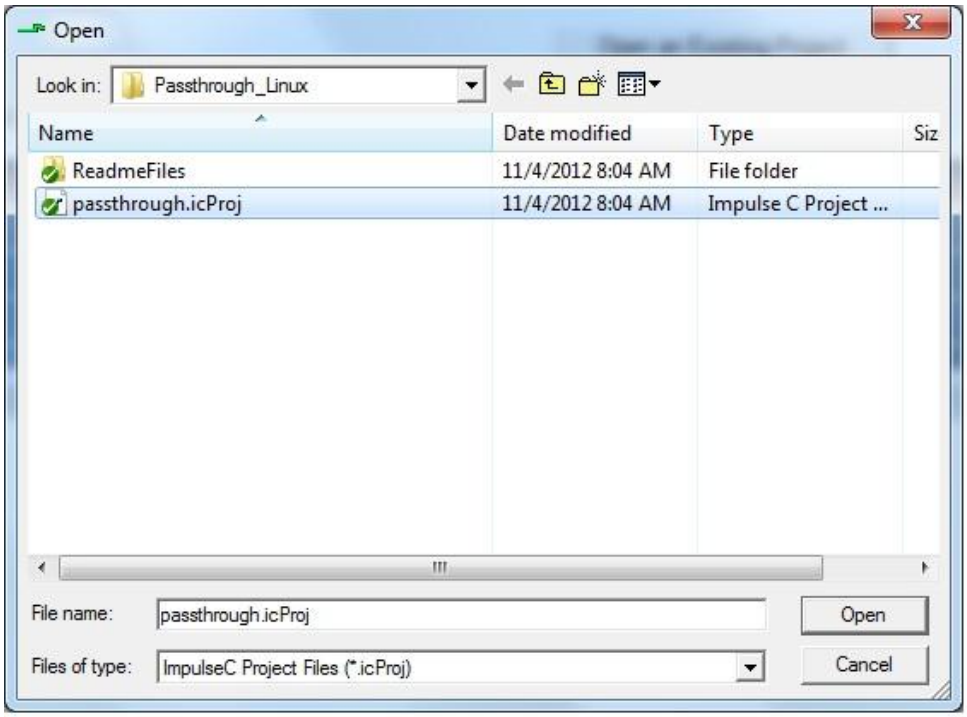


Figure 4 - Opening a project in CoDeveloper

8.4. Building Desktop Simulation Executable

Build the desktop software simulation executable via the “Project” menu:

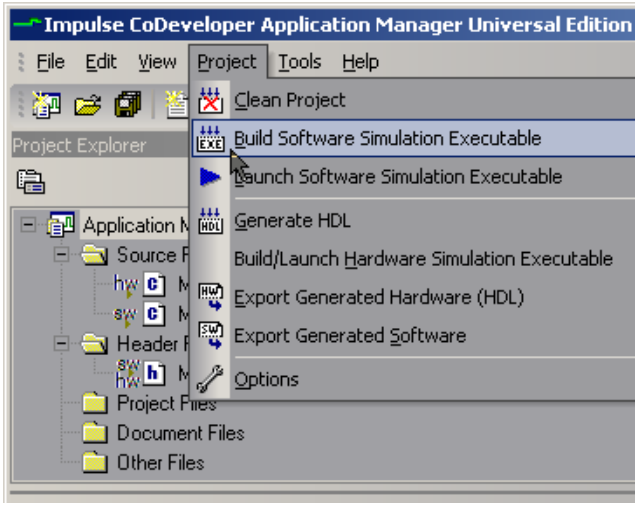


Figure 5 - Build Simulation Desktop in CoDeveloper using pull-down menu

Or via toolbar:

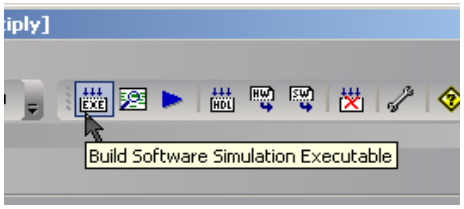


Figure 6 - Build Simulation Desktop in CoDeveloper using toolbar icon

Note the compiler output in the CoDeveloper IDE “Build” window:

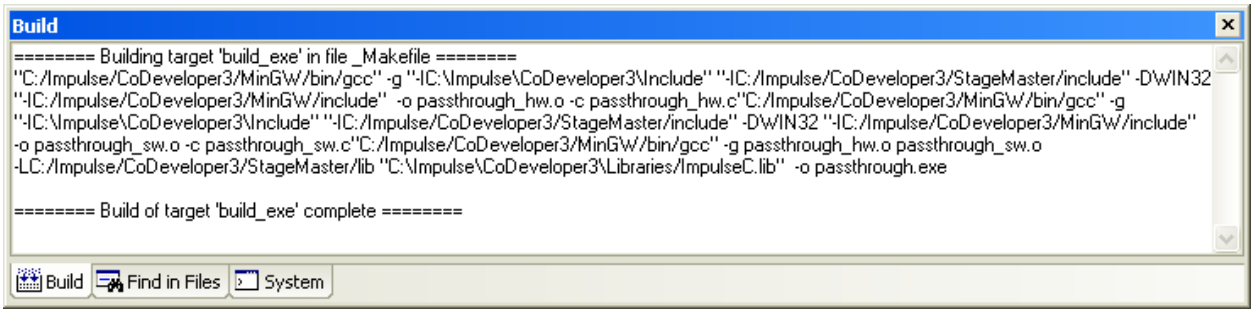


Figure 7 - Output within the CoDeveloper IDE build window

8.5. Running Desktop Simulation Executable

Launch the desktop software simulation executable via “Project” menu:

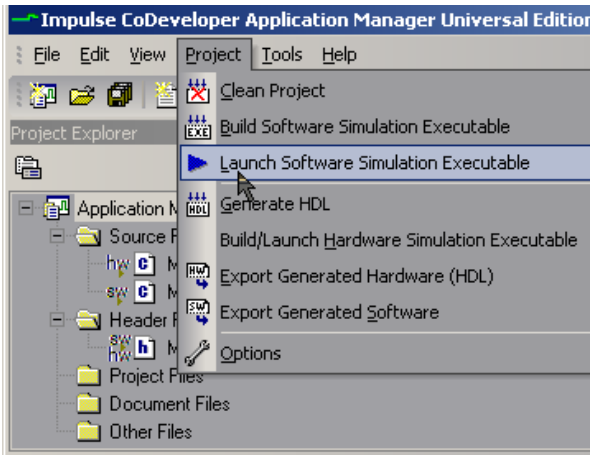


Figure 8 - Launch software simulation window using pull-down menu

Or via toolbar:

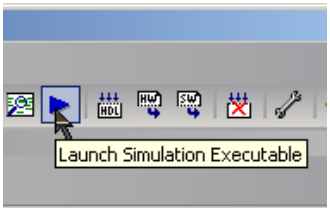


Figure 9 - Launch software simulation using toolbar icon

A command window will pop up in which the desktop simulation executable runs. Press “Enter” to exit:

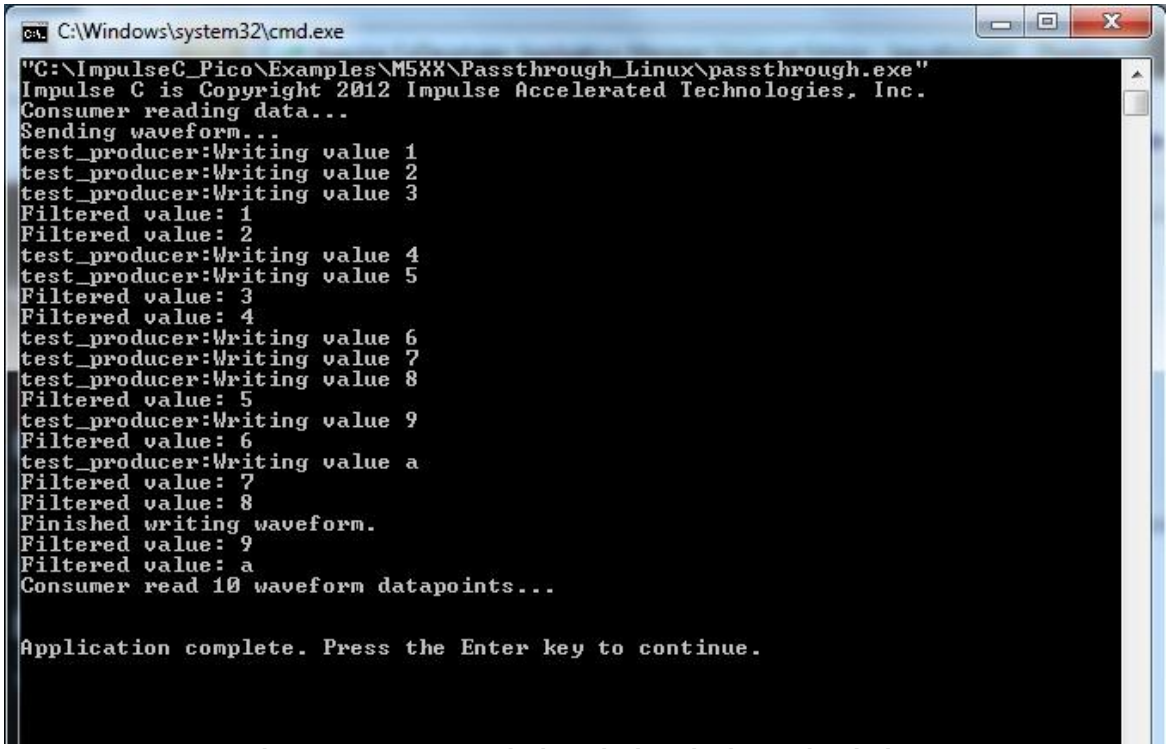


Figure 10 - Pop-up window during desktop simulation

8.6. Project Setup Before Hardware/Software Generation and Export

Settings within the CoDeveloper IDE necessary for generating and exporting both hardware and software using this PSP are summarized below:

- Platform Support Package: “Pico M-501 Linux (VHDL)”
- Hardware export directory: <user hardware export directory>
- Software export directory: <user software export directory>
- Unsupported settings include:
    - Generate dual clocks (must be unchecked)
    - Active-low reset (must be unchecked)
    - Include floating point library (must be unchecked)

An example of these settings as it appears in the Passthrough example:

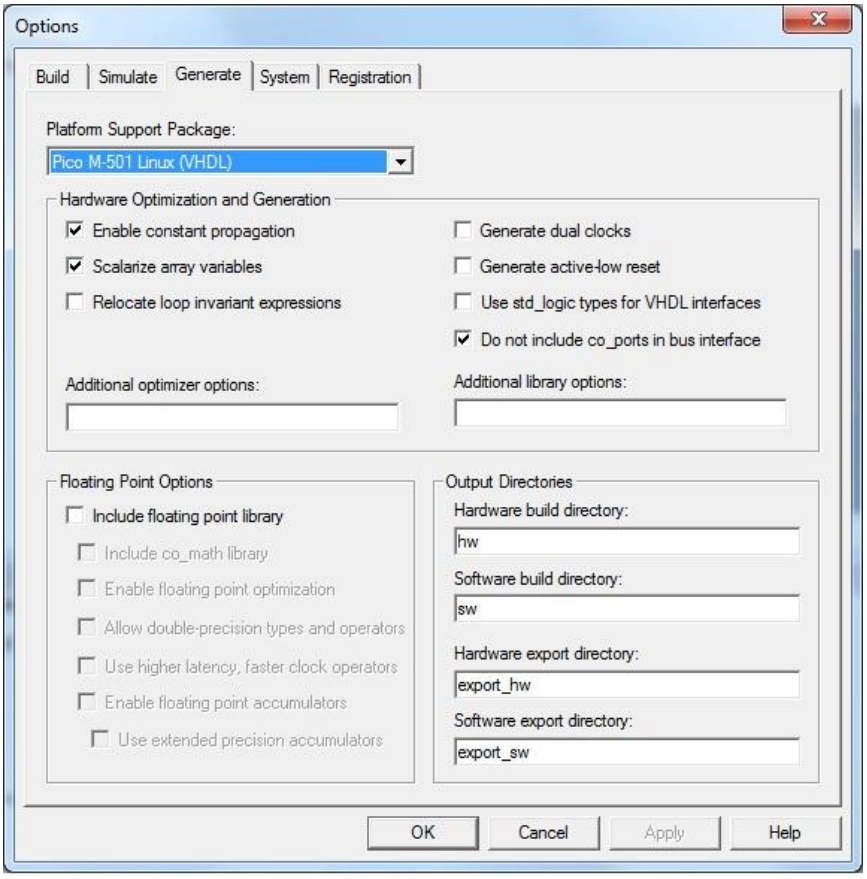


Figure 11 - Project setup to pick Platform Support Package

8.7. Generating Hardware

Generate hardware via “Project” menu:

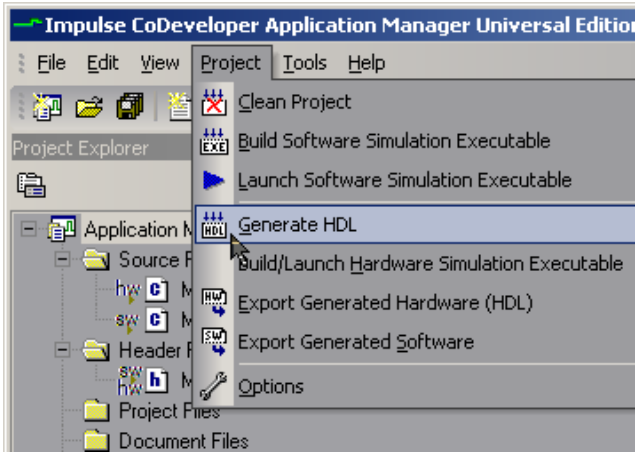


Figure 12 - Generate HDL using pull-down menu

Or via toolbar:

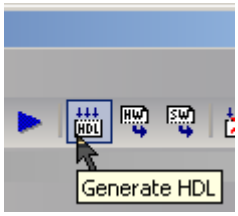


Figure 13 - Generate HDL using toolbar icon

Final results will appear in the directory specified during project setup in “Hardware build directory”. Note the final output in the CoDeveloper IDE’s “Build” window:

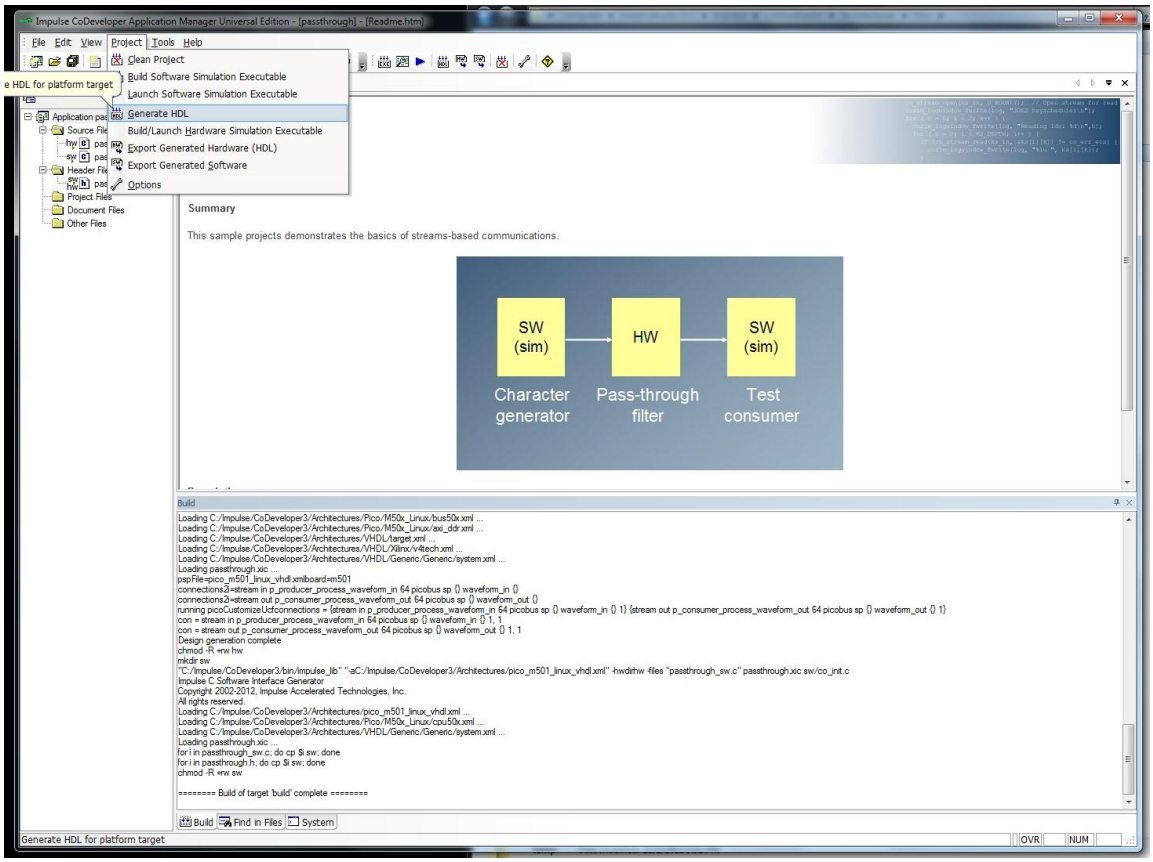


Figure 14 - Build window output

8.8. Exporting Hardware

Export hardware via “Project” menu:

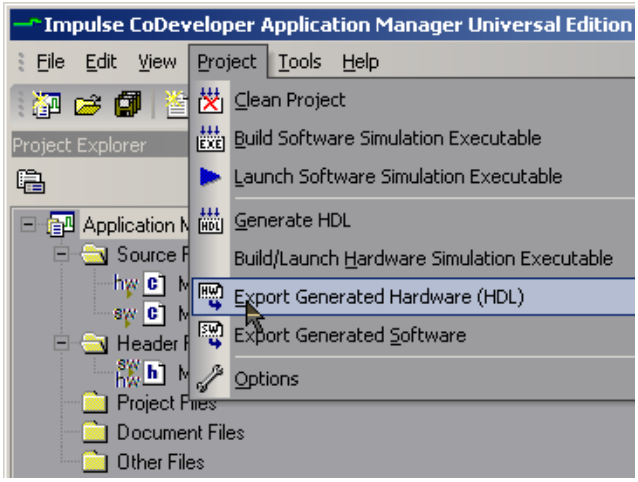


Figure 15 - Export Generated Hardware (HDL) using pull-down menu

Or via toolbar:

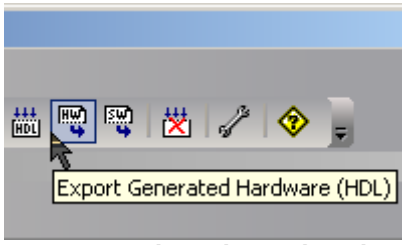


Figure 16 - Export Generated Hardware (HDL) using toolbar icon

Final results will appear in the directory specified during project setup in “Hardware export directory”. Note the final output in the CoDeveloper IDE’s “Build” window:

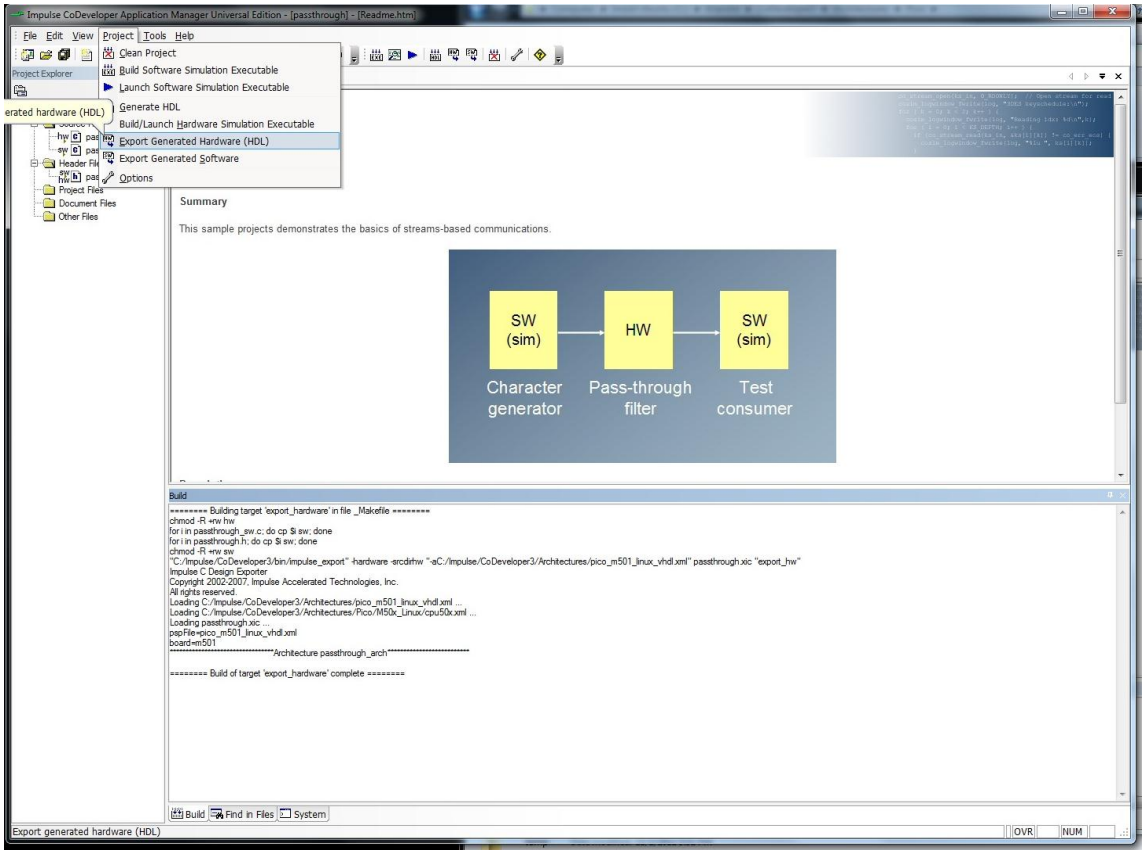


Figure 17 - Build window output

8.9. Compiling FPGA in Xilinx ISE 13.4

After exporting hardware, under the specified hardware export directory will be a directory structure that includes all necessary files for building the FPGA binary.

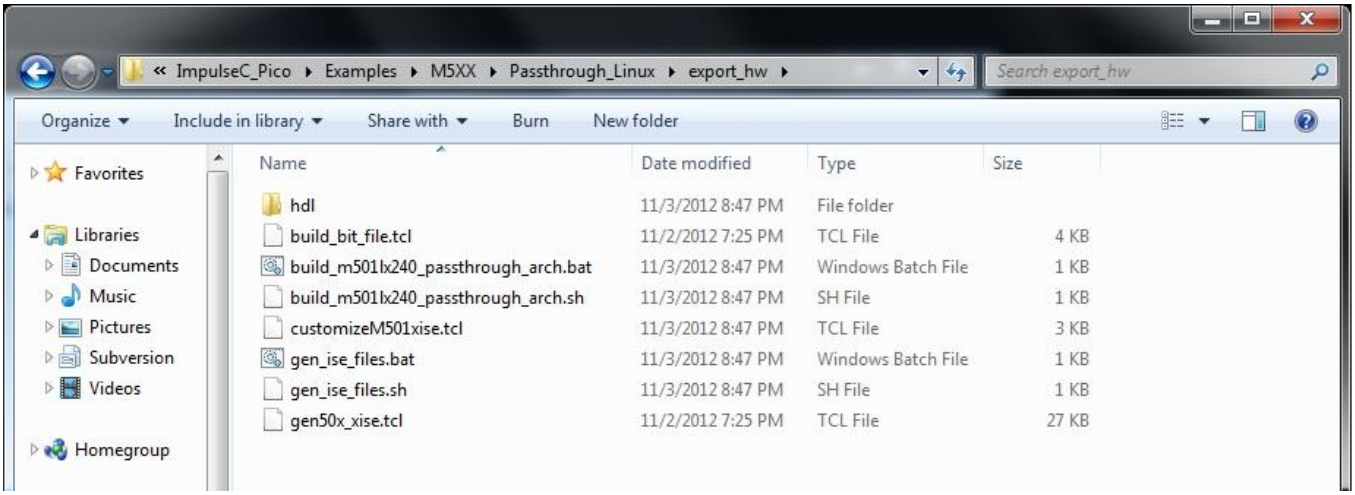


Figure 18 - Compiling FPGA in exported ISE directory structure

At this point either Windows or Linux may be used to produce the bitfile running Xilinx ISE either through the GUI or from command line. Windows and Linux both use the following similar steps. Choose the one that suits your needs.

Windows:

1. Generate the ISE project files by running “gen_ise_files.bat”
2. Build the bit file using Xilinx ISE either by:
    a. Running the generated “build_m50*.bat” file
    b. Launching the Xilinx ISE GUI opening the newly built “m50*.xise” file

Linux:

3. Generate the ISE project files by running “gen_ise_files.sh”
4. Build the bit file using Xilinx ISE either by:
    a. Running the generated “build_m50*.sh” file
    b. Launching the Xilinx ISE GUI opening the newly built “m50*.xise” file

These methods will generate the necessary bitfile to be loaded on the Host System and are outlined in the sections that follow.

8.9.1. Building bitfile under Windows

8.9.1.1. Generate Xilinx ISE Project Files

First generate the ISE project by executing “gen_ise_files.bat”. This process will create an ISE project file as well as copy all the necessary HDL files and CORE Generator components from the Pico installation needed by ISE to generate the FPGA bitfile.

1. Open a command window
2. Change directories to “export_hw”
3. Execute “gen_ise_files.bat” as shown in Figure 19.
4. Verify that the files were created successfully by viewing the end of log file, as shown in Figure 20. “Successful!” should be the last line in the log file (use “type gen50x_xise.log” to display log).

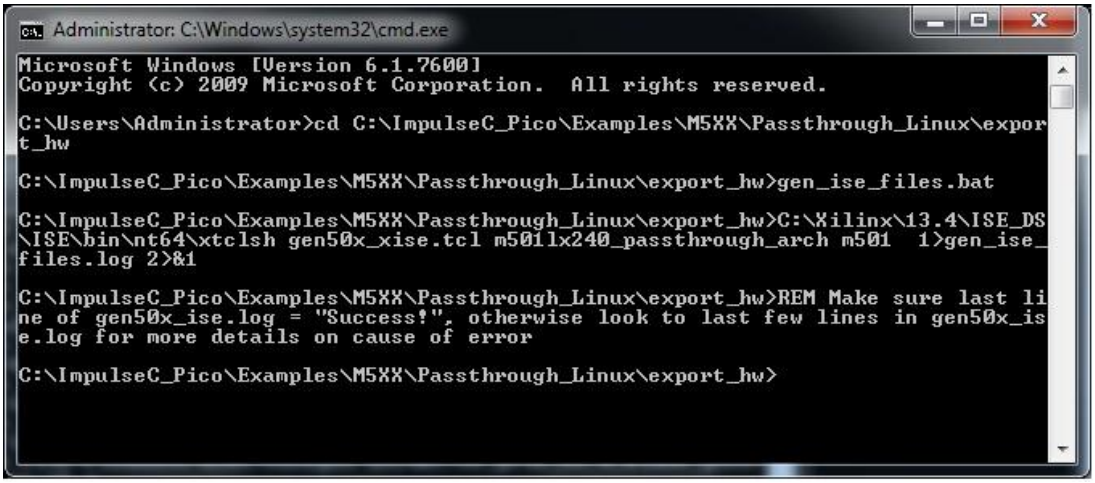


Figure 19 - Generate ISE project files

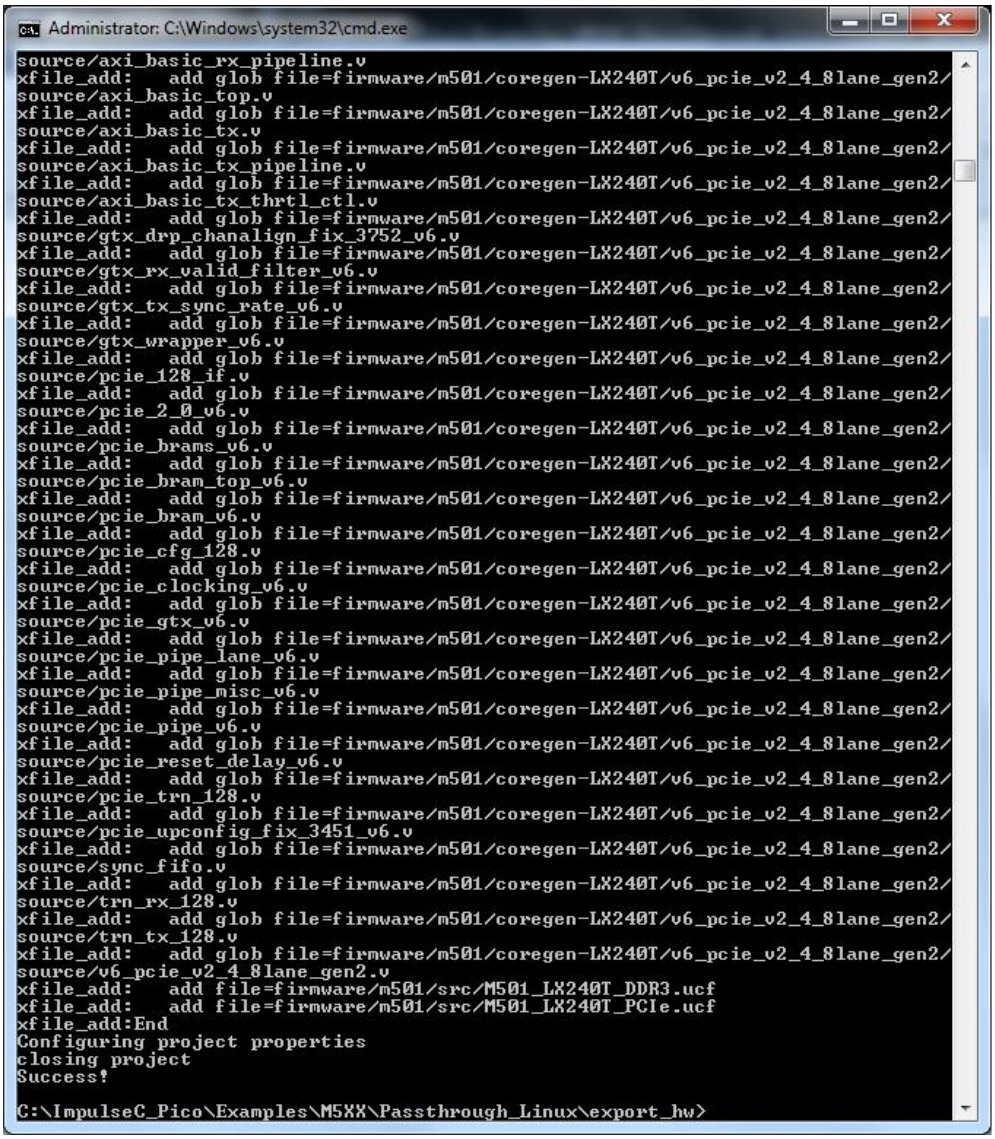


Figure 20 - Expected gen50x_xise.log report

8.9.1.2. Compiling FPGA in Xilinx ISE GUI

Launch the Xilinx ISE GUI and open the ISE project located in the **export_hw** directory. Select Pico_Toplevel in the *Hierarchy* window and double-click “generate programming file” in the *Processes* window.

NOTE: Xilinx ISE will ask the user to regenerate two fifos (a one time process). The following figures illustrate initial project launch, regenerate the two corgen fifos, and final output with timing score equal to zero.

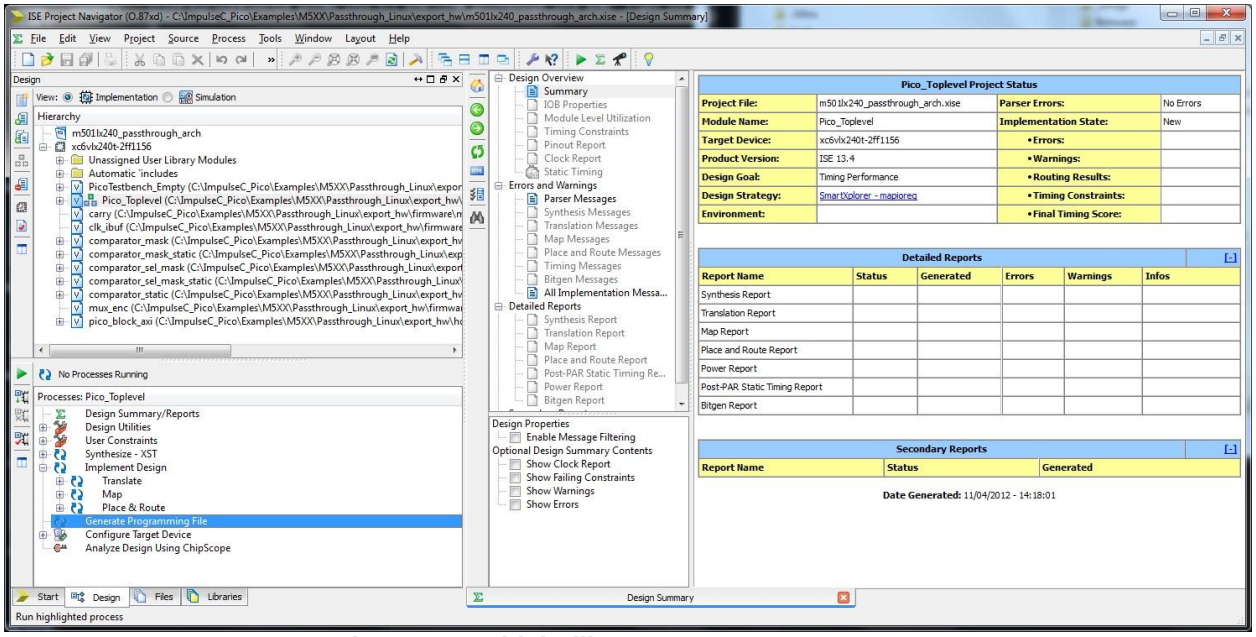


Figure 21 - Initial Xilinx ISE GUI screen

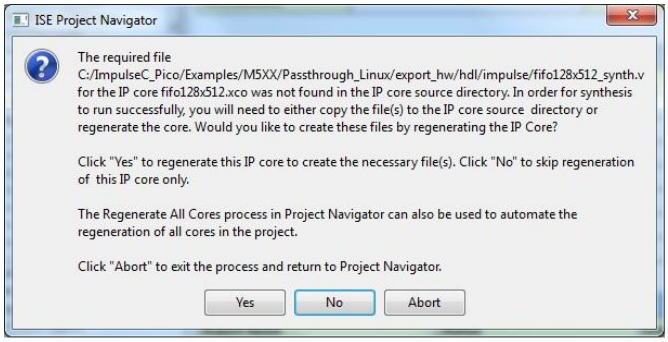


Figure 22 - Xilinx ISE regenerate IP core fifo128x512

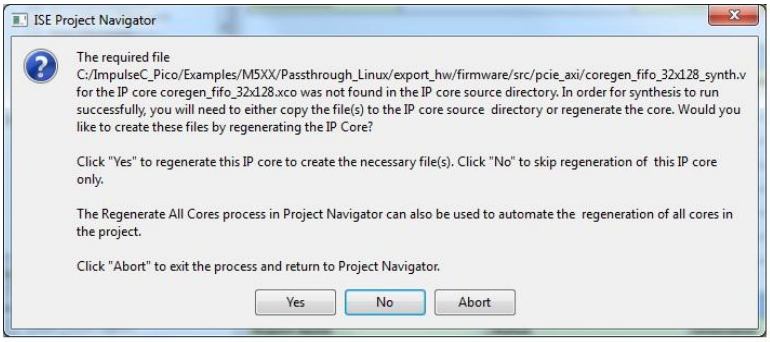


Figure 23 - Xilinx ISE regenerate IP Core coregen_fifo_32x128

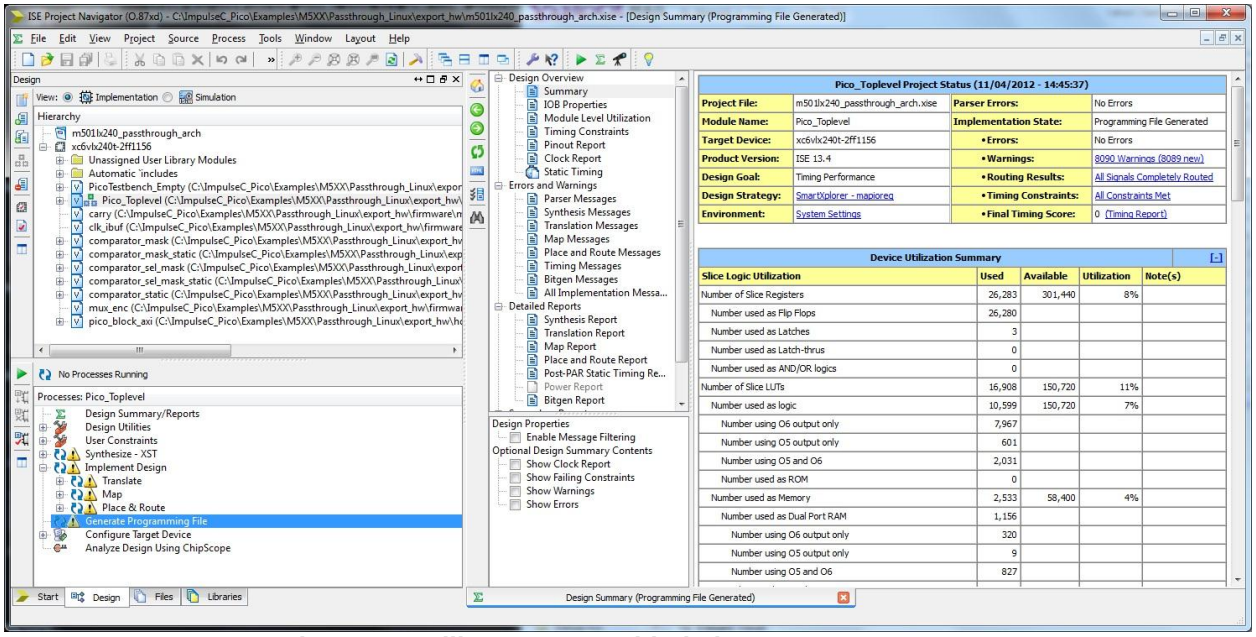


Figure 24 - Xilinx ISE 13.4 with timing score = 0

Once ISE 13.4 has completed, a bitfile “pico_toplevel.bit” will be present in the **export_hw** directory.

Note: The bitfile will need to be renamed to “Pico_Toplevel.bit” when copying to Linux to run with the software in the example.

8.9.1.3. Compiling FPGA in Xilinx ISE using Command Line

In the top directory there will be the batch file “build_passthrough_arch.bat” used to automatically run Xilinx ISE to create the necessary .bit file used to program the M501 FPGA.

1. Open a command window
2. Execute “build_m501lx240_passthrough_arch.bat” as shown in Figure 25.

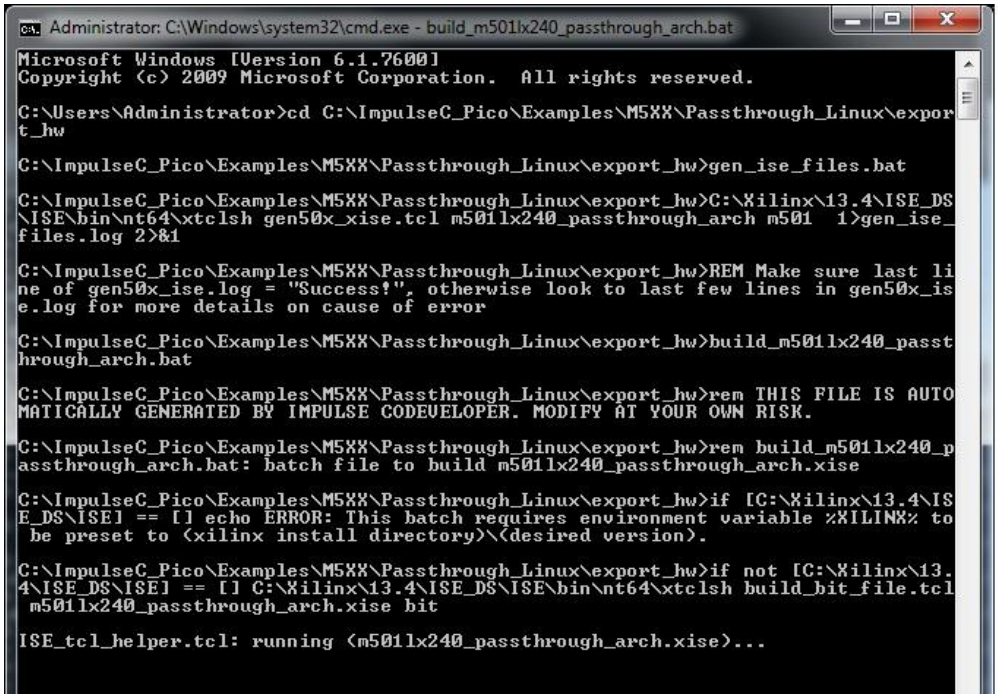
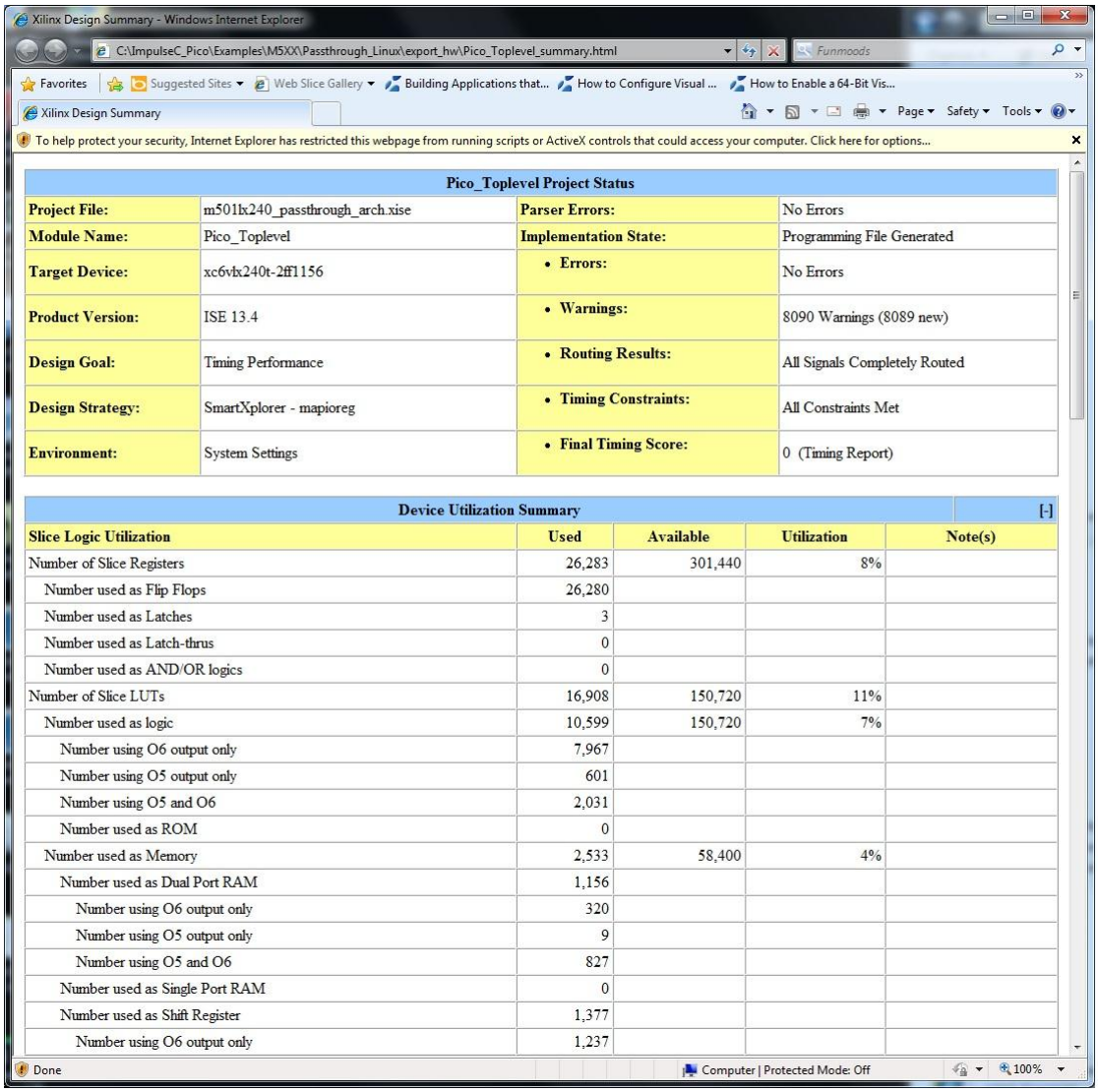


Figure 25 - Build bitfile in ISE 13.4

A command window will appear showing the FPGA build process (primarily made up of many, many info and warning messages). Compile time will vary by machine depending upon project size. When completed successfully, something similar to the following will appear and a bitfile “pico_toplevel.bit” will be present in the **export_hw** directory.

Note: The bitfile will need to be renamed to “Pico_Toplevel.bit” when copying to Linux to run with the software in the example.



Pico_Toplevel Project Status

Project File:	m501lx240_passthrough_arch.xise	**Parser Errors:**	No Errors
Module Name:	Pico_Toplevel	**Implementation State:**	Programming File Generated
Target Device:	xc6vlx240t-2ff1156	• **Errors:**	No Errors
Product Version:	ISE 13.4	• **Warnings:**	8090 Warnings (8089 new)
Design Goal:	Timing Performance	• **Routing Results:**	All Signals Completely Routed
Design Strategy:	SmartXplorer - mapioreg	• **Timing Constraints:**	All Constraints Met
Environment:	System Settings	• **Final Timing Score:**	0 (Timing Report)

Device Utilization Summary [-]

Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	26,283	301,440	8%	
Number used as Flip Flops	26,280			
Number used as Latches	3			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	16,908	150,720	11%	
Number used as logic	10,599	150,720	7%	
Number using O6 output only	7,967			
Number using O5 output only	601			
Number using O5 and O6	2,031			
Number used as ROM	0			
Number used as Memory	2,533	58,400	4%	
Number used as Dual Port RAM	1,156			
Number using O6 output only	320			
Number using O5 output only	9			
Number using O5 and O6	827			
Number used as Single Port RAM	0			
Number used as Shift Register	1,377			
Number using O6 output only	1,237			

Figure 26 – Xilinx ISE 13.4 compile log file – timing score

8.9.2. Building bitfile under Linux

8.9.2.1. Generate Xilinx ISE Project Files

First generate the ISE project by executing “gen_ise_files.sh”. This process will create an ISE project file as well as copy all the necessary HDL files and CORE Generator components from the Pico installation needed by ISE to generate the FPGA bitfile.

1. Copy the **export_hw** directory to a location on your Host System that has the Linux operating system. (ie Pico/Passthrough/export_hw).
2. Open a command console
3. Change directories to “export_hw”
4. Execute “source ./gen_ise_files.sh” as shown in Figure 27.
5. Verify that the files were created successfully by viewing the end of log file, as shown in Figure 28. “Successful!” should be the last line in the log file (use “cat gen50x_xise.log” to display log).

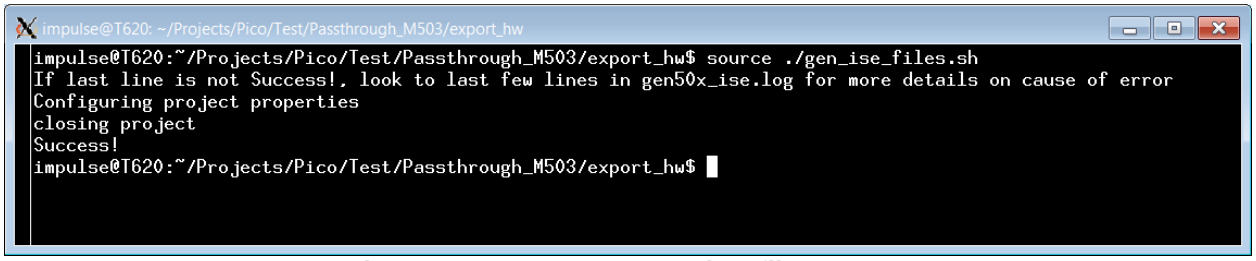


Figure 27 - Generate ISE project files

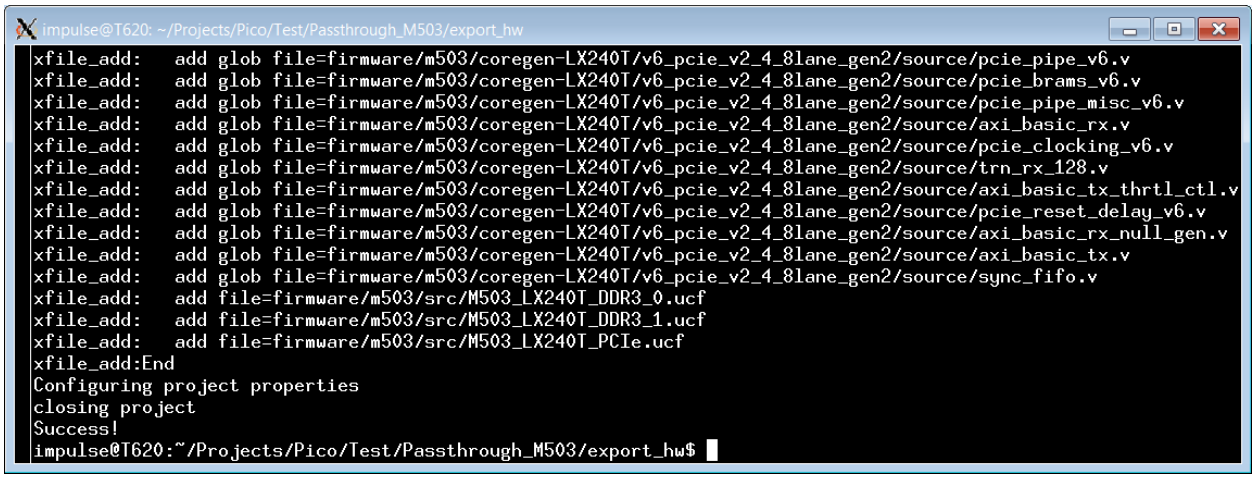


Figure 28 - Expected gen50x_xise.log report

8.9.2.2. Compiling FPGA in Xilinx ISE GUI

1. If you already haven't done so, launch or verify that the license manager is operational and that your license is valid. Contact Xilinx for more information on licensing related to a Linux installation.
2. Open a terminal window and set environment variables by executing the "settings64.sh" script as well as the following variables:
    a. source /opt/Xilinx/13.4/ISE_DS/setting64.sh
    b. export PICOBASE=/usr/src/picocomputing-5.0.6.1
    c. export XILINX=/opt/Xilinx/13.4/ISE_DS/ISE
3. Go to the **export_hw** directory and begin executing the shell scripts to generate the Xilinx ISE project.
    a. cd /Pico/Passthrough/export_hw
    b. source gen_ise_files.sh
4. Launch the Xilinx ISE 13.4 GUI, select the project, regenerate coregent files, and generate the bitfile.
    a. /opt/Xilinx/13.4/ISE_DS/ISE/bin/lin64/ise &

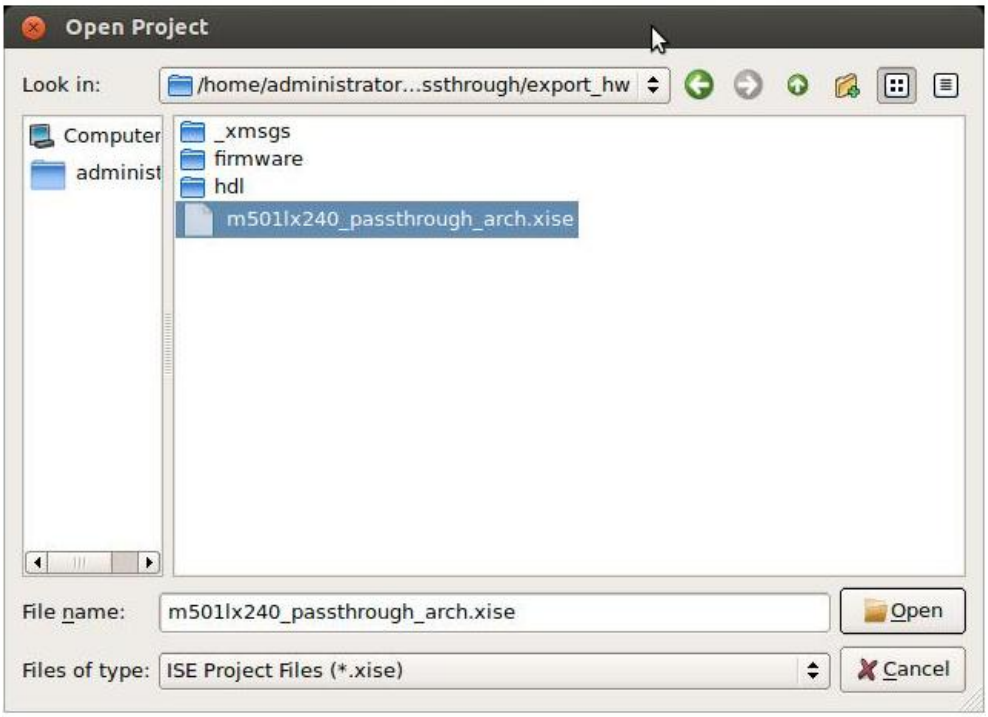


Figure 29 - Select the ISE project

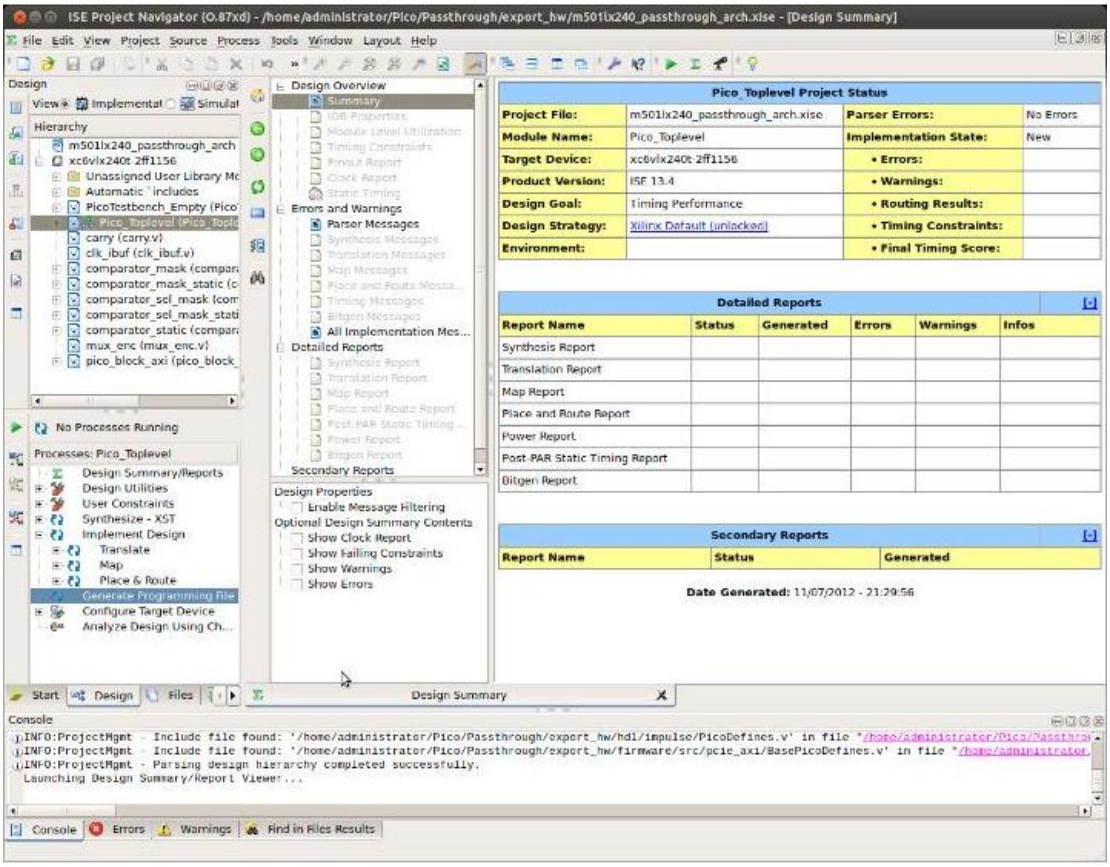


Figure 30 - Initial project status after loading project

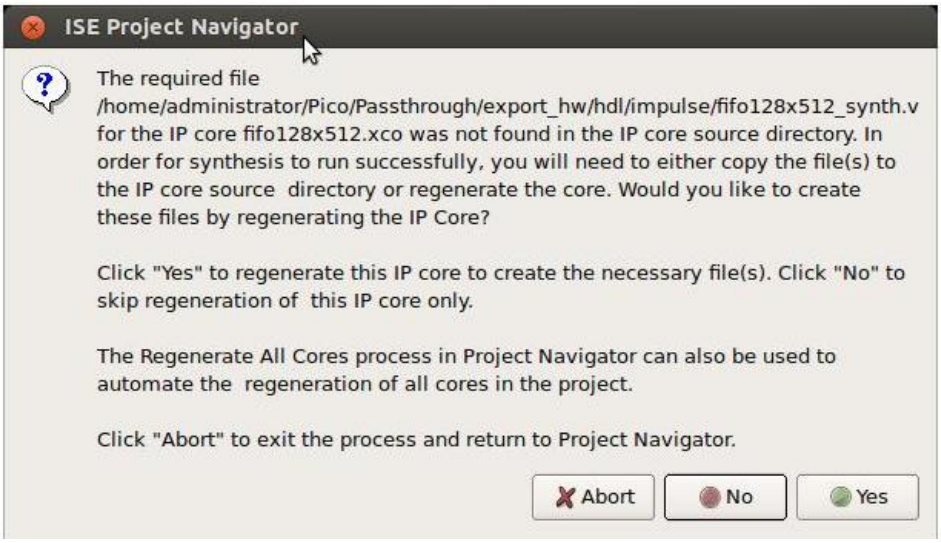


Figure 31 - Regenerate fifo128x512

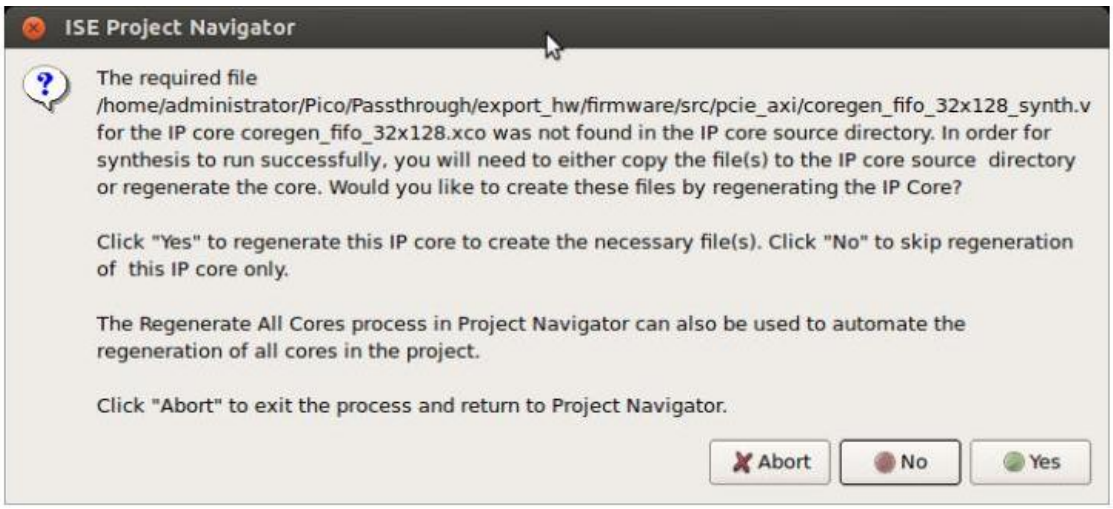


Figure 32 - Regenerate coregen_fifo_32x128

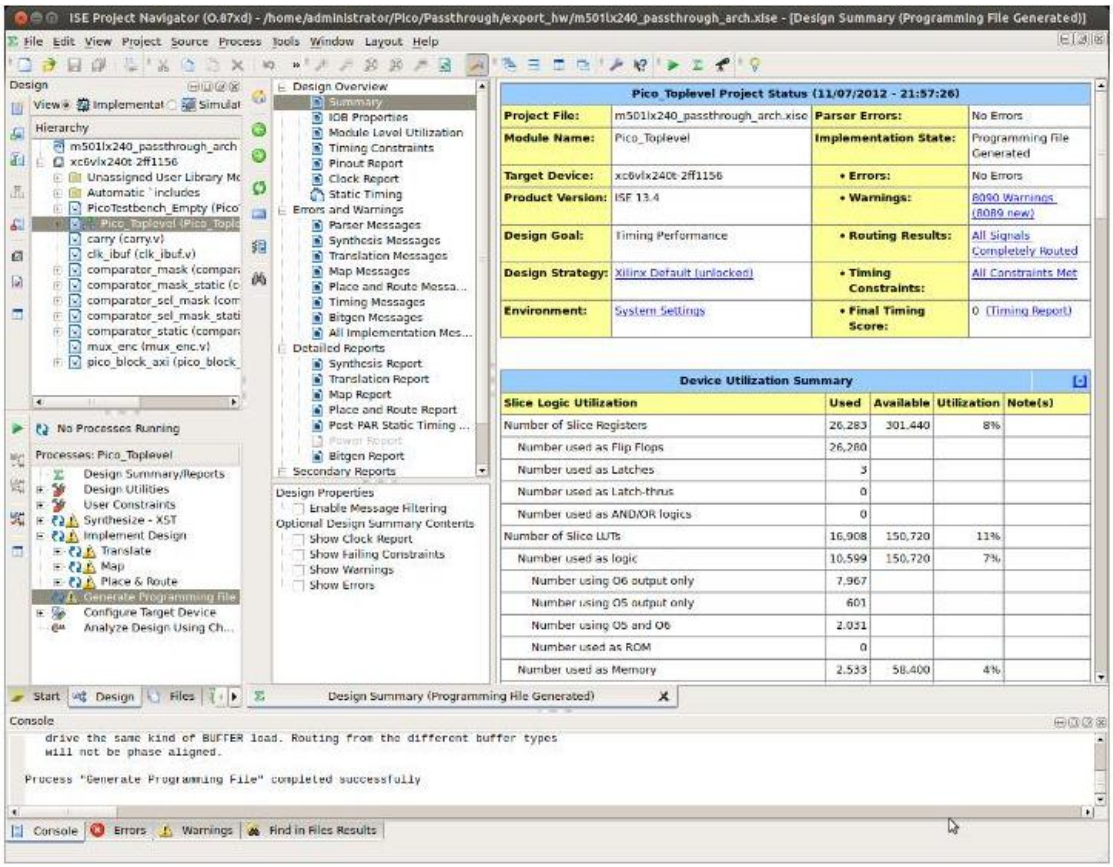


Figure 33 - Place and Route and bitfile generation with timing score equal to zero

8.9.2.3. Compiling FPGA in Xilinx ISE using Command Line

1. Copy the **export_hw** directory to a location on your Host System that has the Ubuntu operating system. (ie Pico/Passthrough/export_hw).

2. If you already haven't done so, launch or verify that the license manager is operational and that your license is valid. Contact Xilinx for more information on licensing related to a Linux installation.
3. Open a terminal window and set environment variables by executing the "settings64.sh" script as well as the following variables:
    c. cd /opt/Xilinx/13.4/ISE_DS
    d. sudo ./setting64.sh
    e. export PICOBASE=/usr/src/picocomputing-5.0.6.1
    f. export XILINX=/opt/Xilinx/13.4/ISE_DS/IS
4. Go to the **export_hw** directory and begin executing the shell scripts to generate then build the FPGA bitfile.
    g. cd /Pico/Passthrough/export_hw
    h. source gen_ise_files.sh
    i. source build_m501lx240_passthrough_arch.sh
5. Verify that the bitfile was generated and that the timing score equals zero.
    j. ll *.bit (should see Pico_Toplevel.bit)
    k. grep –i score *.par

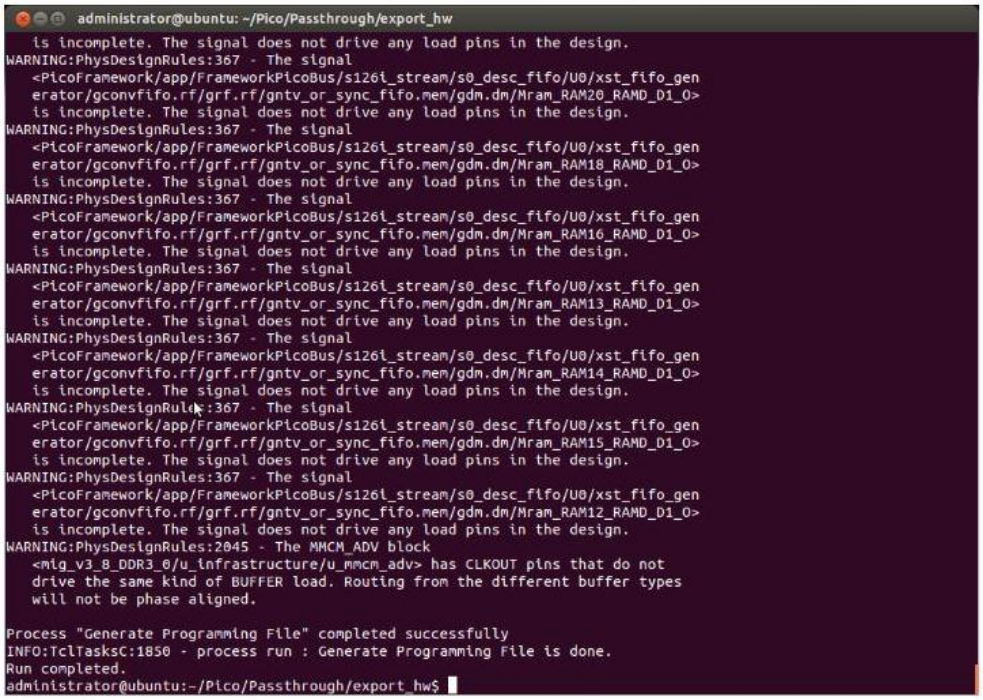


Figure 34 - Ubuntu Xilinx ISE 13.4 Command Line output

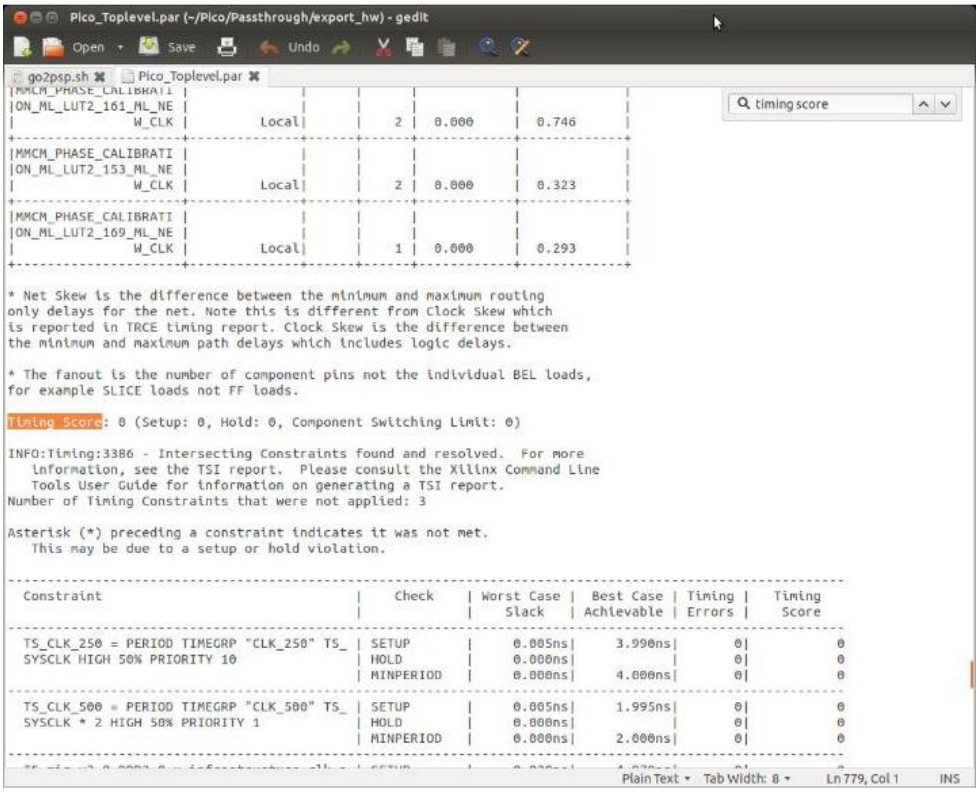


Figure 35 - Xilinx ISE 13.4 results file with timing score equal to zero

8.10. Exporting Software

The software application to be run on the host computer (with the M501 installed with it's drivers) can be exported in CoDeveloper.

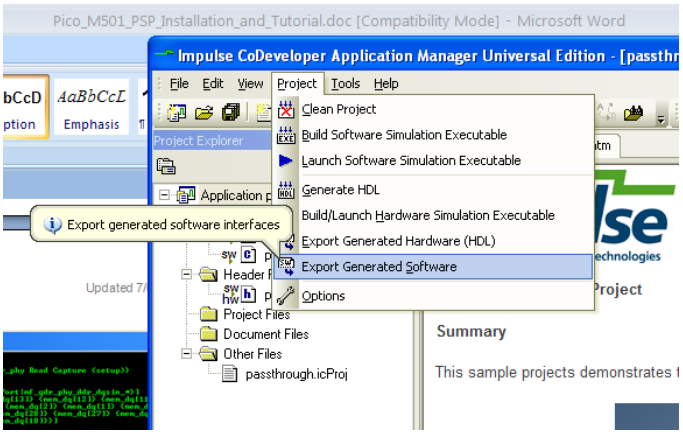


Figure 36 - Export Generated Software

Once completed without error in the Build window, it should be noted that the software code will be written to a newly created directory. The user can modify the target directory name. In this example, export_sw contains the exported software files.

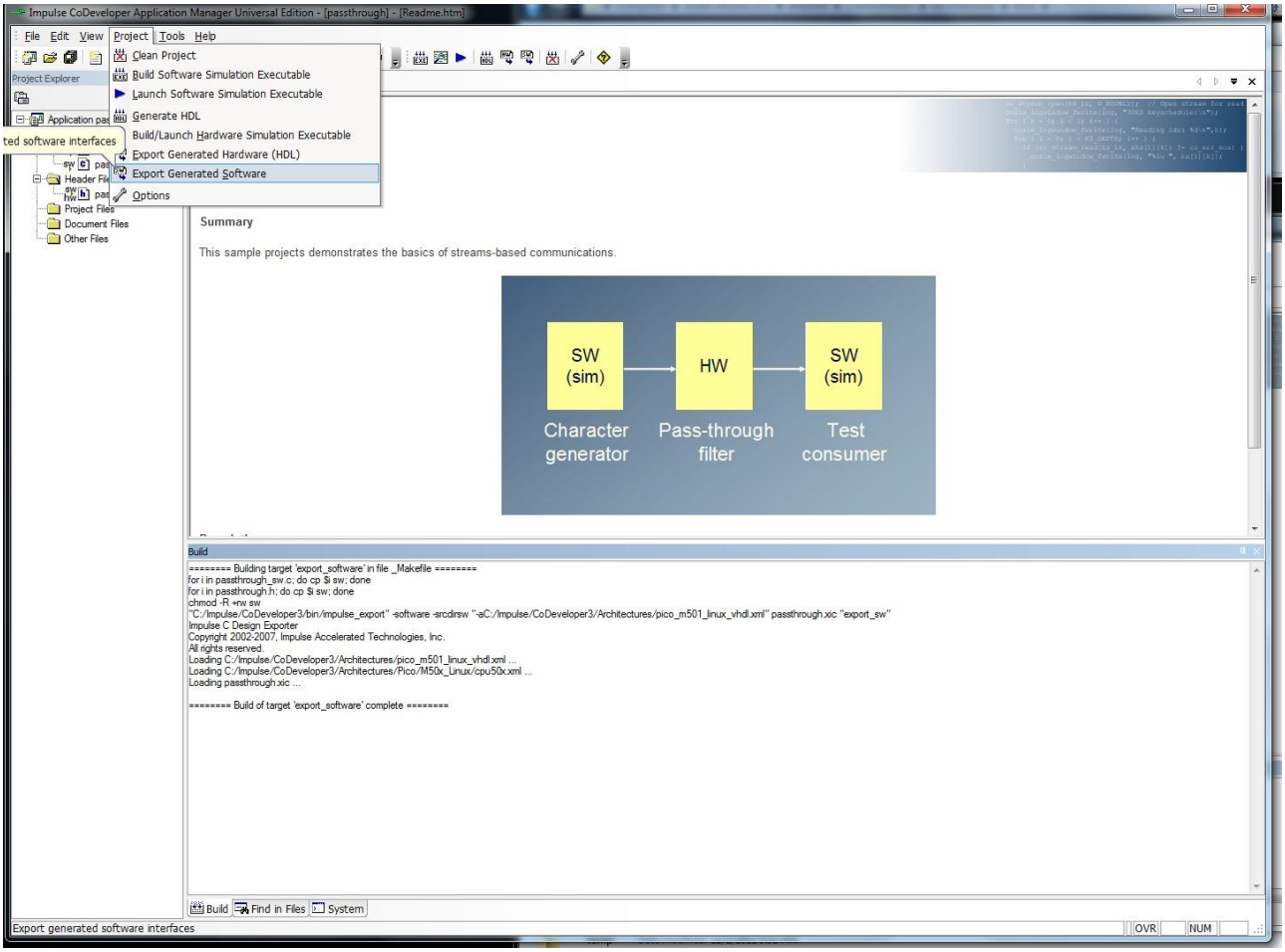


Figure 37 - Build window output

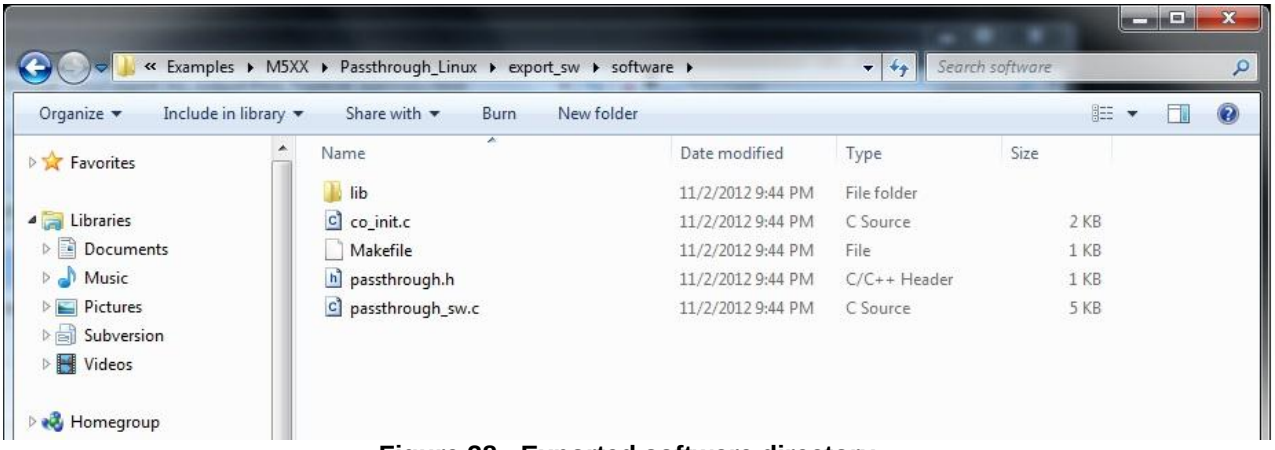


Figure 38 - Exported software directory

8.11. Programming the FPGA

The compiled software application in the “export_sw” directory will program the FPGA on the M501. Please continue to the next section.

8.12. Running Target Executable on the Host System

The Host System will need the following files:

1. Compiled bitfile output from ISE 13.4
2. Input data file
3. Application software source code
4. Makefile to compile the software

Please copy the following files and directory over to the Host System (figure 30). Create a folder (ie Pico) and copy the following to that folder.

1. **export_hw** directory which includes “pico_toplevel.bit” bitfile
2. **export_sw** directory which includes the Makefile and software application source code
3. filter_in.dat file which will be used to input stimulus to the FPGA via the software application.
4. Once the files are copied, please copy filter_in.dat to the **export_sw** directory. The application software will expect the input data file to be present in that directory.

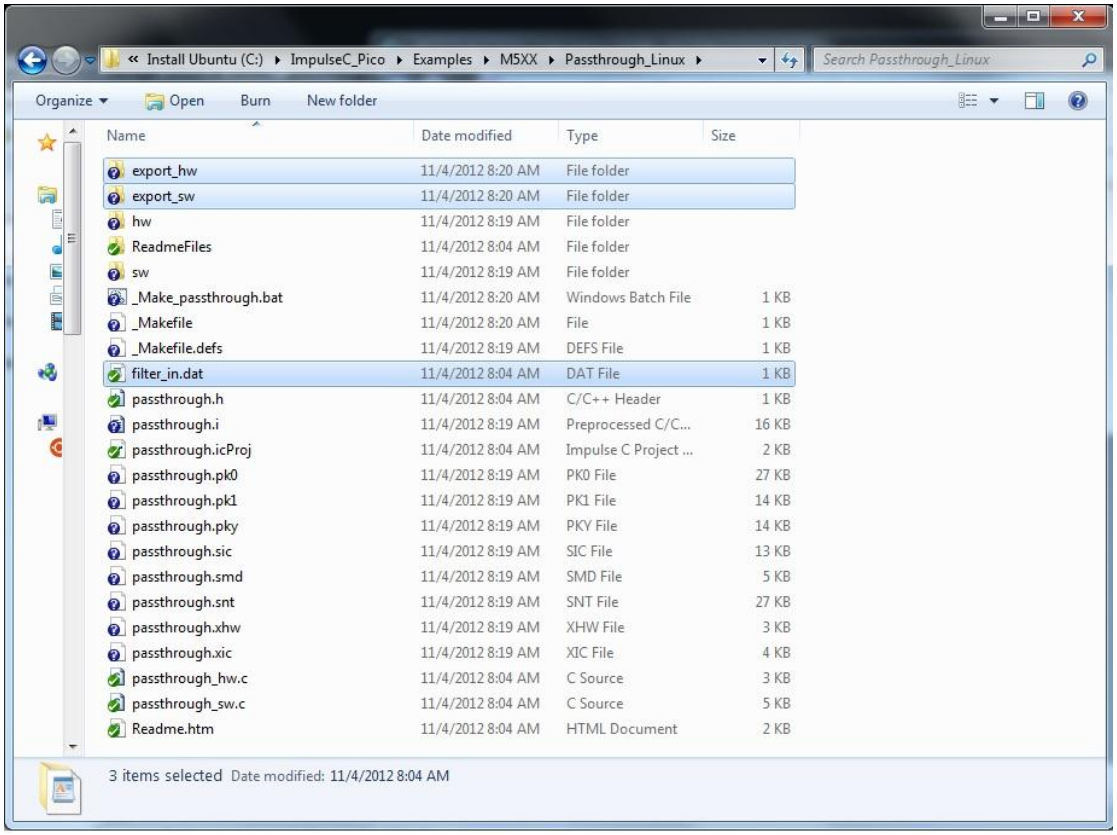


Figure 39 - Files & Directories to be copied to the Host System

The software application will load the FPGA every time it is executed and report the results upon completion. Please note that the process of loading the FPGA may take up to 20 seconds due to the driver re-start sequence after the FPGA bitfile has been transferred to the card.

1. Open a terminal window.
2. Navigate to the location of your copied files and directories:

   (ie. cd Pico/export_sw/software).
3. Run “make”
4. Execute the generated application: “./passthrough_arch”

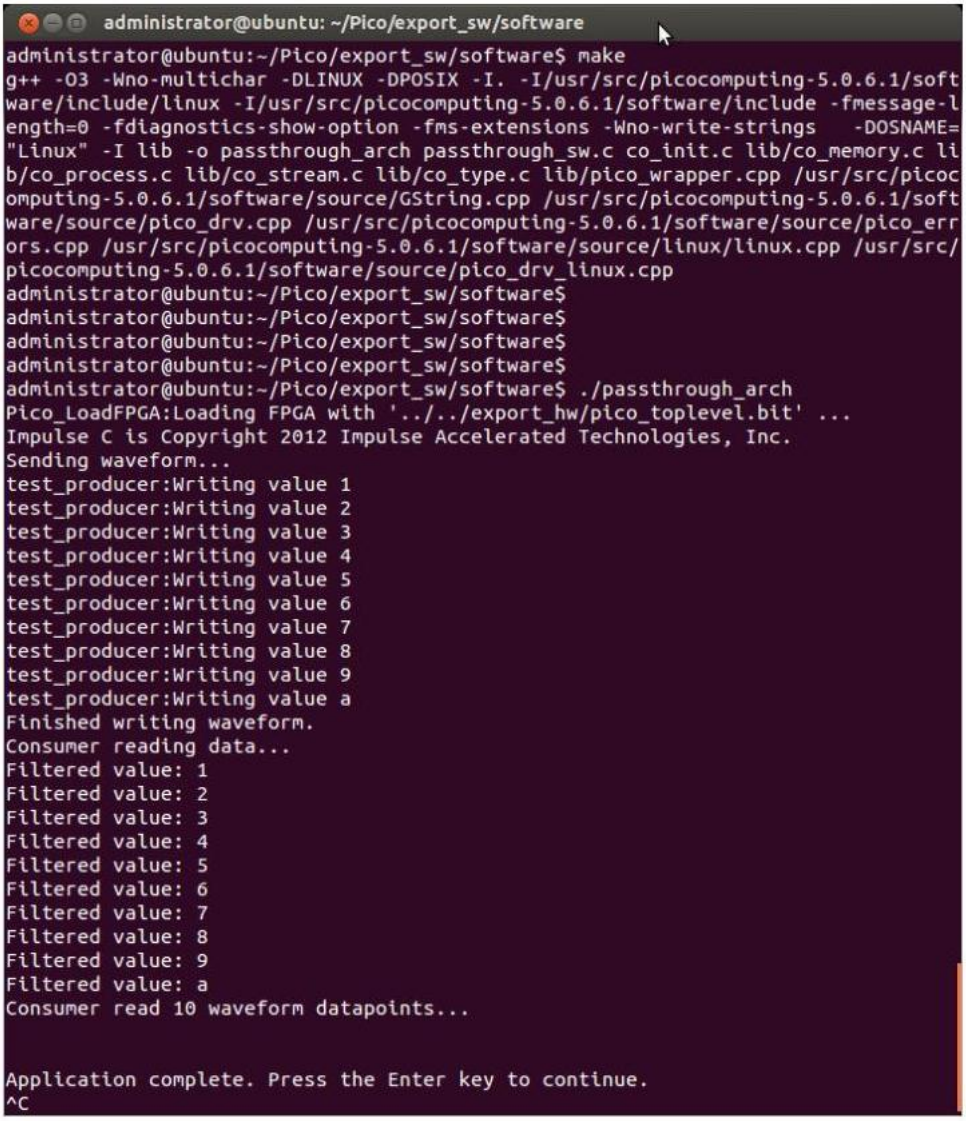


Figure 40 - Exported SW executed on target platform

9.0 Memtest Example and Tutorial

9.1. Prerequisites

The tutorial in this Platform Support Package assumes that you have read and understand the introductory sections of the CoDeveloper User's Guide, installed with CoDeveloper and accessed from the Help menu. In particular, you should take the time to go through the tutorials provided with CoDeveloper so you have a good understanding of the front-end design flow including both desktop software simulation and hardware compilation.

9.2. Memtest Example Description

The Memtest example is a CoDeveloper project that includes three source files. A brief description of the files and their functions is outlined in the following sections. This example targets existing external DDR3 memory available on the M501 FPGA module.

1. memtest.h
2. memtest_hw.c
3. memtest_sc.c

The Memtest example allows a stream to write to DDR3 memory. The Impulse User module will read from DDR3, increment the data by one, and write it back out to a stream. The software application generates the data and checks the data returned from the FPGA.

9.2.1. Overview of memtest.h

The figure below is a screen shot of header file. It defines the stream widths (STREAMWIDTH) as 128 bits. It also defines the size of the target memory (MEMORY_SIZE) that will be used by the co_memory API calls.

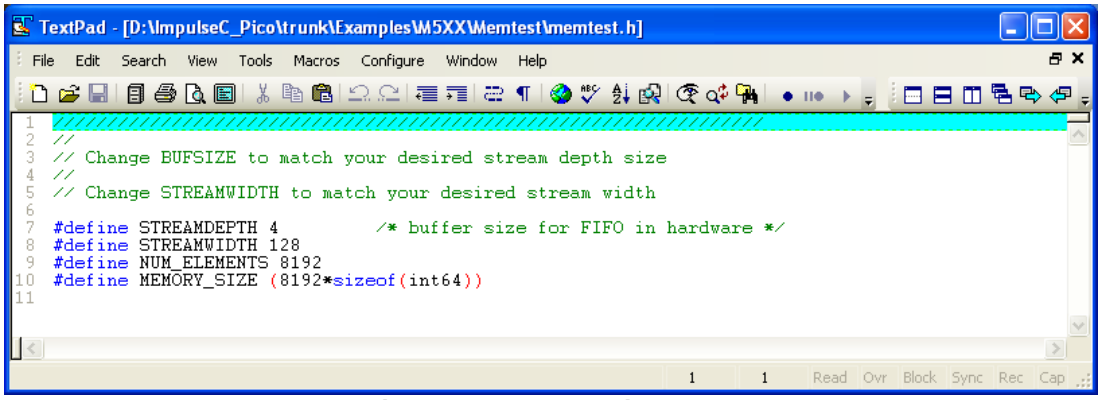


Figure 41 - memtest.h

9.2.2. Overview of memtest_hw.c

The memtest_hw.c file is the user defined function that accesses external DDR3 memory. The figure below shows the user defined function, ***pass***, that will be implemented on the FPGA. On line 18, the ports are defined as input stream ***idata***, output stream ***odata***, and memory port ***mem***.

In this example, the function ***pass*** will receive, on stream ***idata***, the number of words to read from external DDR3. The “do” loop on line 29 indicates that this hardware process run continuously. Input and output streams, **idata** and **odata**, are open as shown on lines 32 & 33. Line 36 is an Impulse API to read in data from the stream. Line 39 shows the Impulse API to read a block of data from external memory. The “for” loop on line 41 operates on each word as it is received from external memory and is also incremented by one. Once all data had been read and incremented, it is written back to external memory as shown on line 47. The data is also written back out on the stream, **odata**, as shown on line 49. Once all data and processing is complete, the input and output streams are closed.

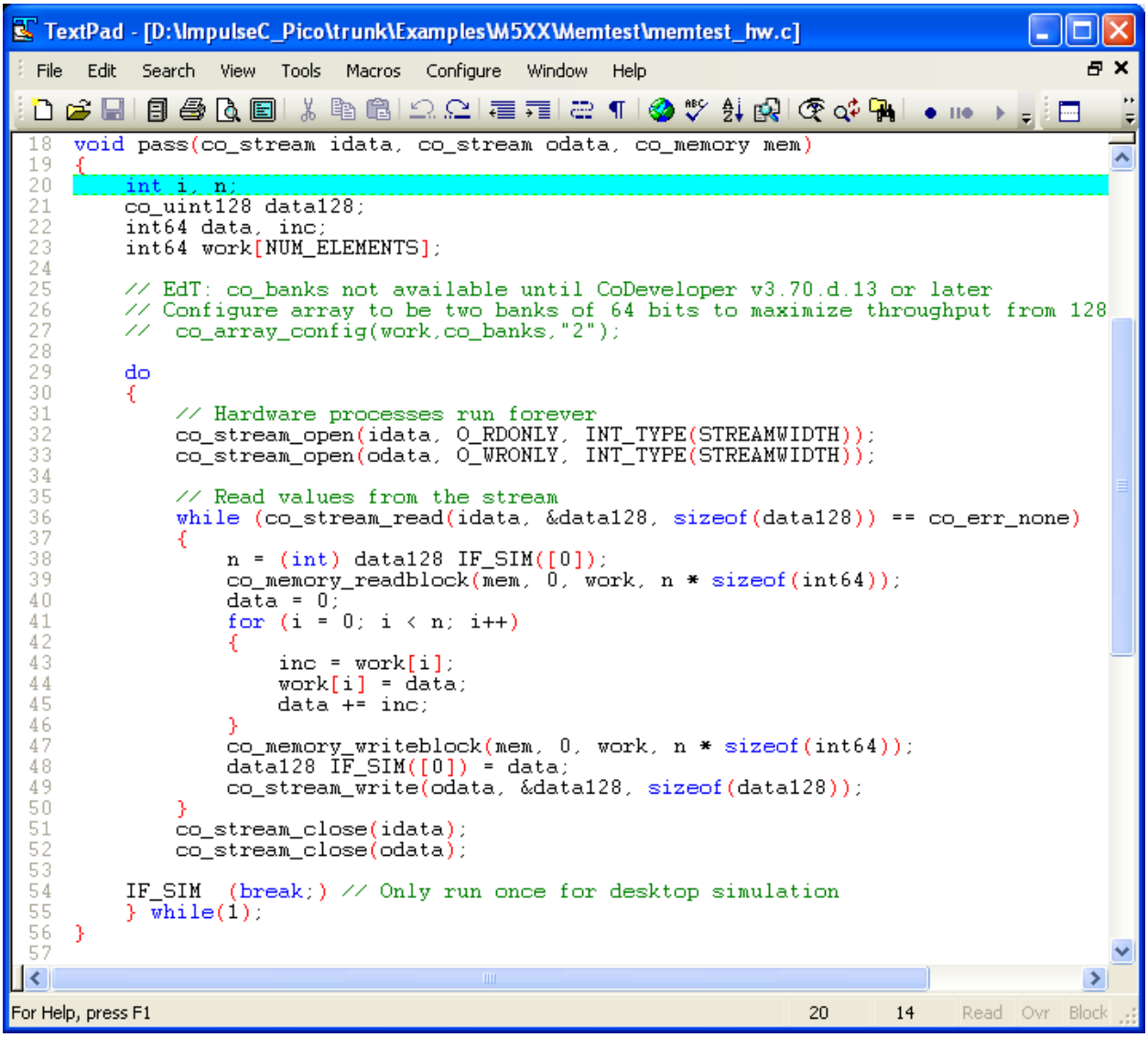


Figure 42 - User defined funtion in memtest_hw.c

The configuration process defines the processes and ports that can be accessed by the user defined function **pass**. Line 58 is the configuration **config_memtest**. Line 60 thru 62 defined the available ports while line 64 & 65 define processes. Lines 67 shows the Impulse defined co_memory with type “mc0” and size MEMORY_SIZE. Lines 68 & 69 shows the Impulse defined co_streams of widths STREAMWIDTH and depth STREAMDEPTH (both defined in the header file). The configuration statement also defines the testbench process (line 71) as well as the user process (line 77). Line 86 is necessary when using co_memory.

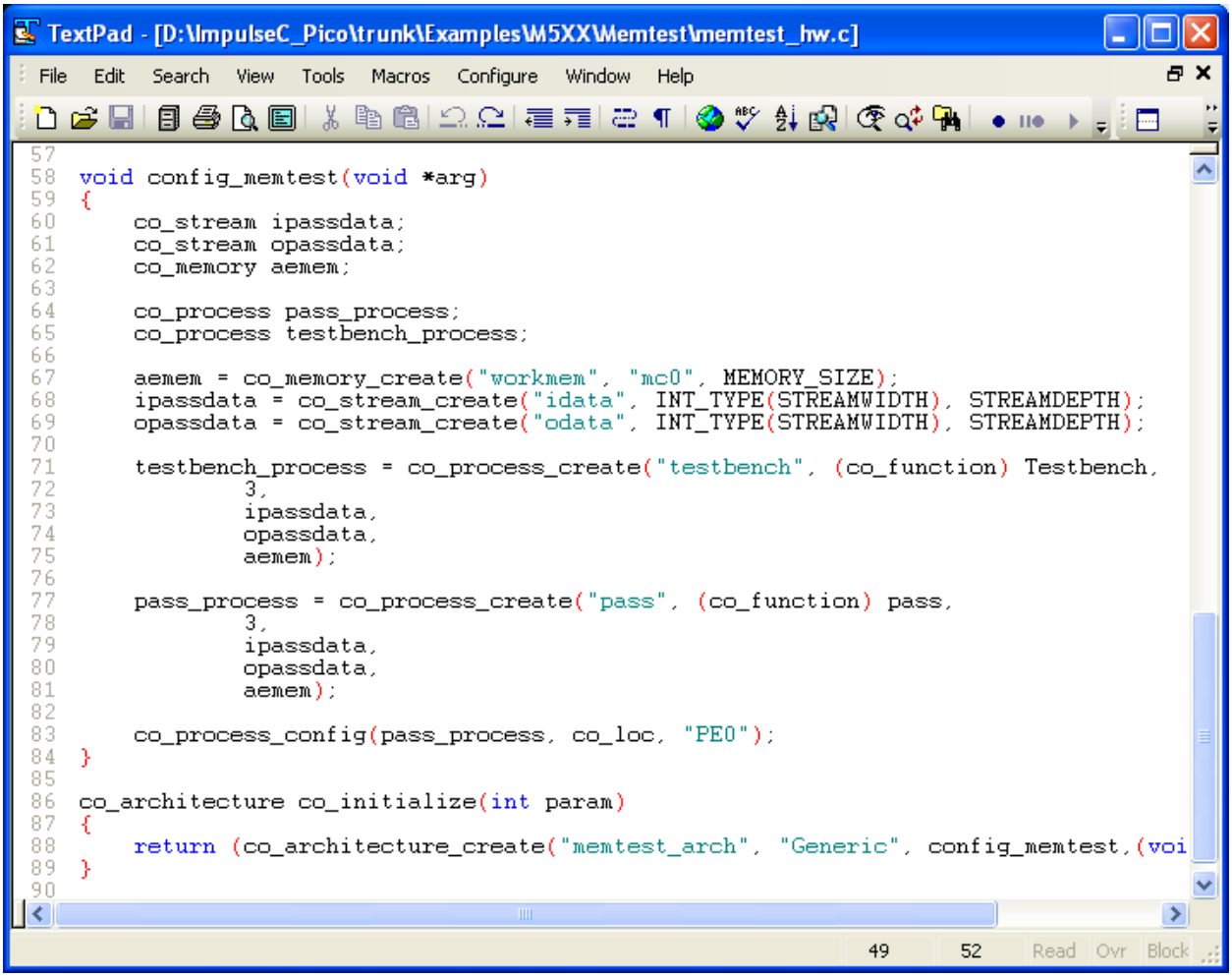


Figure 43 - Configuration in memtest_hw.c

9.2.3. Overview of memtest_sw.c

The software application provides stimulus and receives return data for post processing of the user defined functions. Line 20 defines a process “Testbench” which also has ports **ipassdata**, **opassdata** and **mem**. The port **opassdata** will send data to the user defined function. In this example, line 31 generates data to be written to external memory. Similar to the hardware process, streams must be open in order to read or write to them. Lines 38 & 39 open streams. Line 42 calls the Impulse API co_memory to write a block of data to external memory. Line 46 sends the number of bytes to be read by the hardware process while line 49 waits for data to be return by the hardware process. Line 55 reads data from external memory and that data is used to verify the data received on the input stream as shown in the “for” loop on line 58. Finally, the streams are closed as shown on lines 66 & 67.

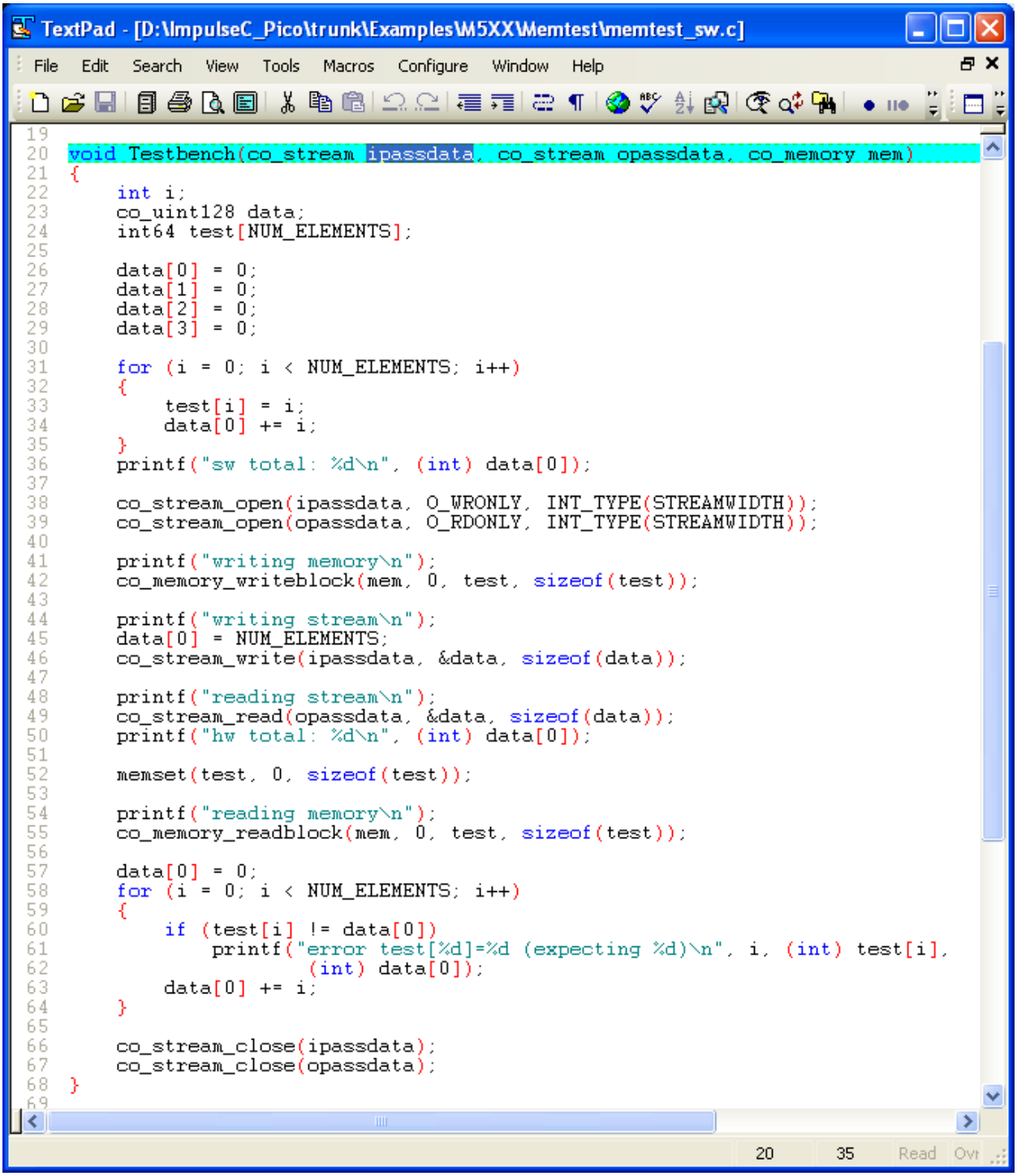


Figure 44 - User defined stimulus in memtest_sw.c

The main function is shown below. The first step for the software application is to load the FPGA bitfile using the Pico API Pico_LoadFPGA as shown on line 78. If the file does not exist or an error is encountered, then an error is reported. Otherwise the testbench is executed.

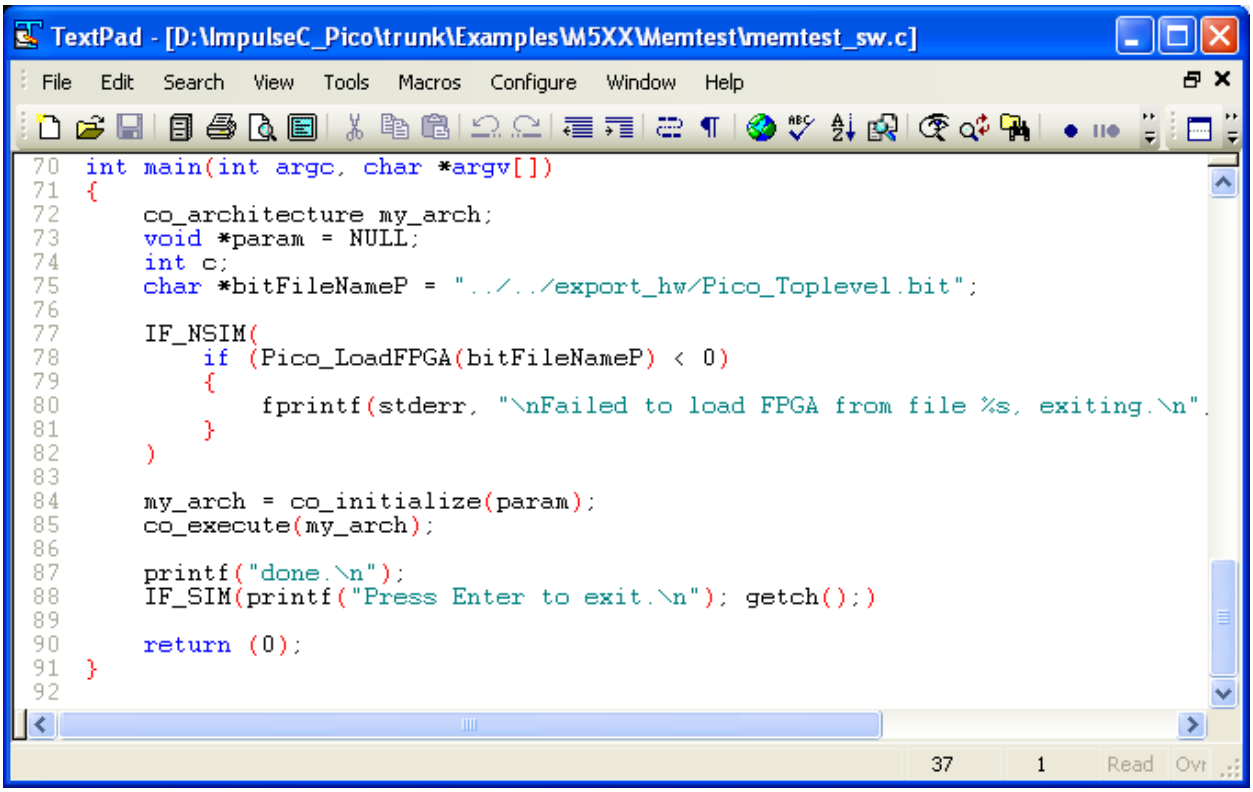


Figure 45 - Main program in memtest_sw.c

9.3. CoDeveloper Project Files

The Memtest example CoDeveloper project is made up of the following files:

- memtest.icProj – CoDeveloper project file
- memtest_hw.c – Source code for hardware process
- memtest_sw.c – Source code for software processes
- memtest.h – Header file that defines the width of the stream

When you define the width of the steam, you must make the changes in the header file as well as the in the memtest_hw.c file. The default example defines the steam to be the width of 128 bit data bus.

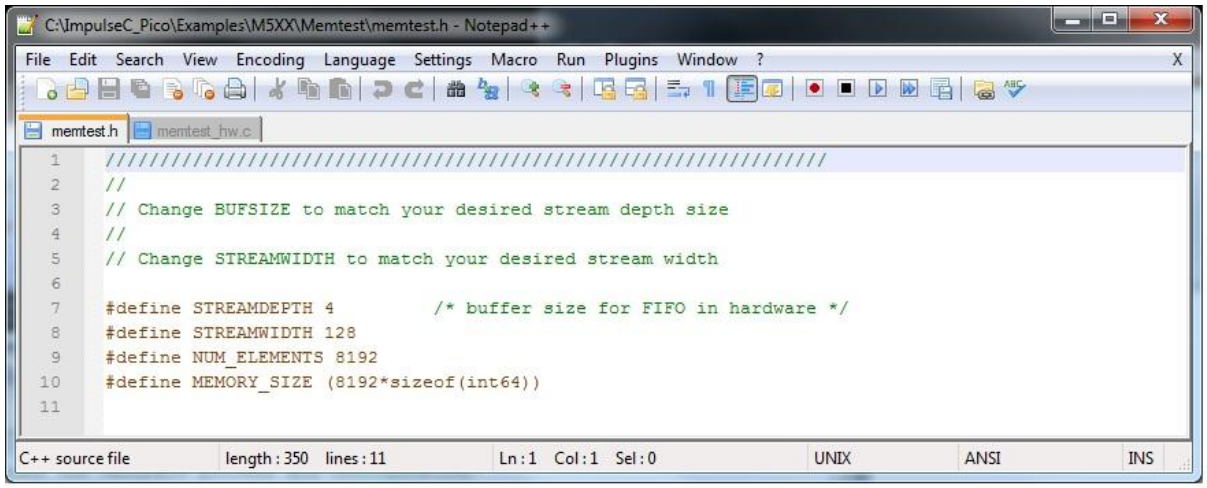
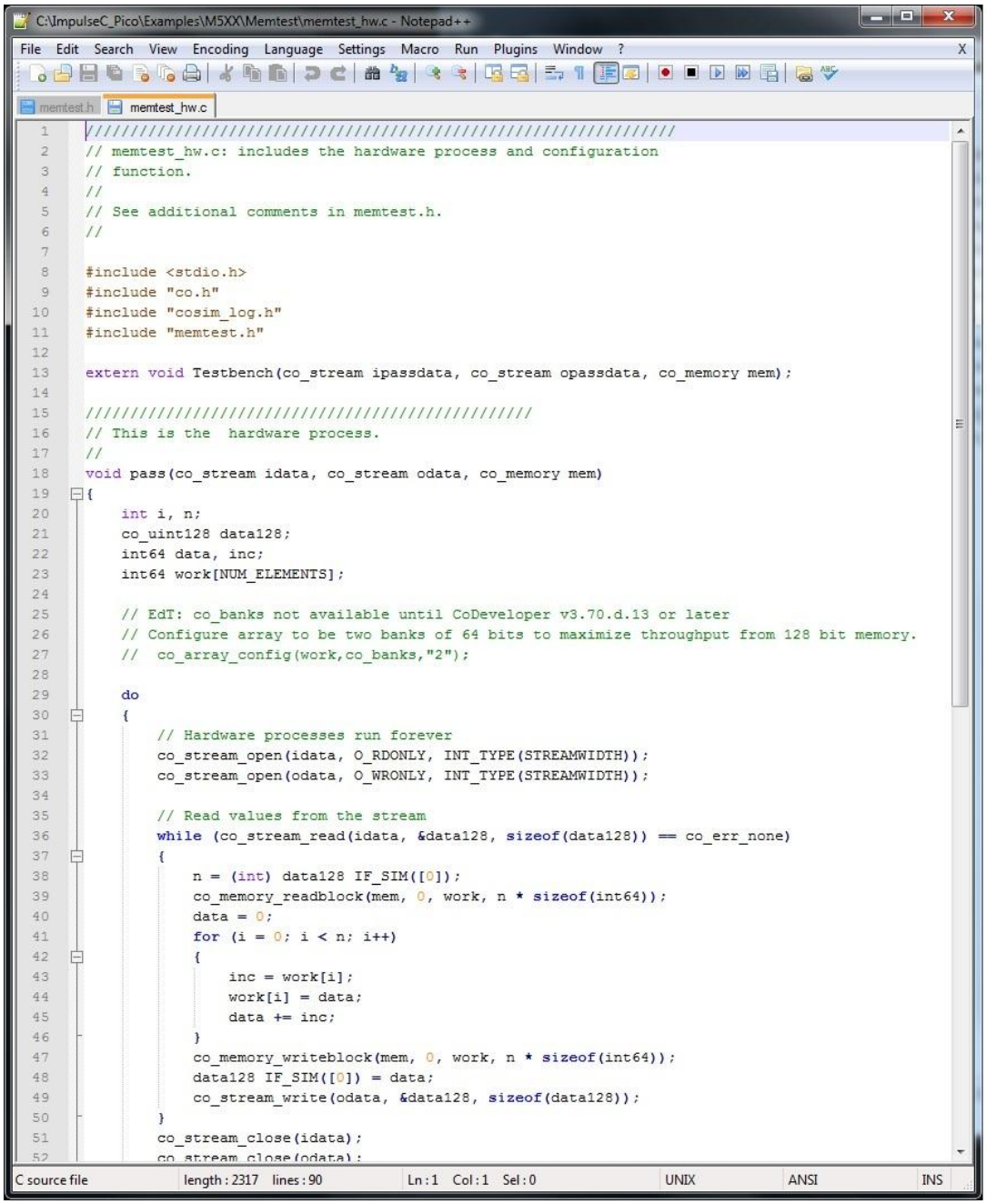


Figure 46 - Impulse C Header File




```
////////////////////////////////////////////////////////////////////
// memtest_hw.c: includes the hardware process and configuration
// function.
//
// See additional comments in memtest.h.
//

#include <stdio.h>
#include "co.h"
#include "cosim_log.h"
#include "memtest.h"

extern void Testbench(co_stream ipassdata, co_stream opassdata, co_memory mem);

////////////////////////////////////////////////
// This is the  hardware process.
//
void pass(co_stream idata, co_stream odata, co_memory mem)
{
    int i, n;
    co_uint128 data128;
    int64 data, inc;
    int64 work[NUM_ELEMENTS];

    // EdT: co_banks not available until CoDeveloper v3.70.d.13 or later
    // Configure array to be two banks of 64 bits to maximize throughput from 128 bit memory.
    //  co_array_config(work,co_banks,"2");

    do
    {
        // Hardware processes run forever
        co_stream_open(idata, O_RDONLY, INT_TYPE(STREAMWIDTH));
        co_stream_open(odata, O_WRONLY, INT_TYPE(STREAMWIDTH));

        // Read values from the stream
        while (co_stream_read(idata, &data128, sizeof(data128)) == co_err_none)
        {
            n = (int) data128 IF_SIM([0]);
            co_memory_readblock(mem, 0, work, n * sizeof(int64));
            data = 0;
            for (i = 0; i < n; i++)
            {
                inc = work[i];
                work[i] = data;
                data += inc;
            }
            co_memory_writeblock(mem, 0, work, n * sizeof(int64));
            data128 IF_SIM([0]) = data;
            co_stream_write(odata, &data128, sizeof(data128));
        }
        co_stream_close(idata);
        co_stream_close(odata);
```

Figure 47 - ImpulseC Hardware File

9.4. Opening Project

Open the CoDeveloper project file ‘Memtest.icProj’ by selecting and pressing ‘Enter’ or by double-clicking it:

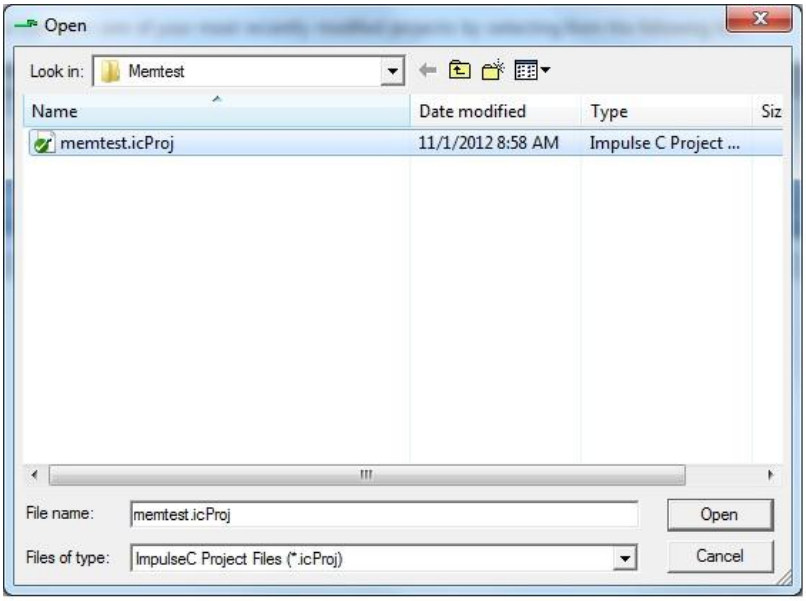


Figure 48 - Opening a project in CoDeveloper

9.5. Building Desktop Simulation Executable

Build the desktop software simulation executable via the “Project” menu:

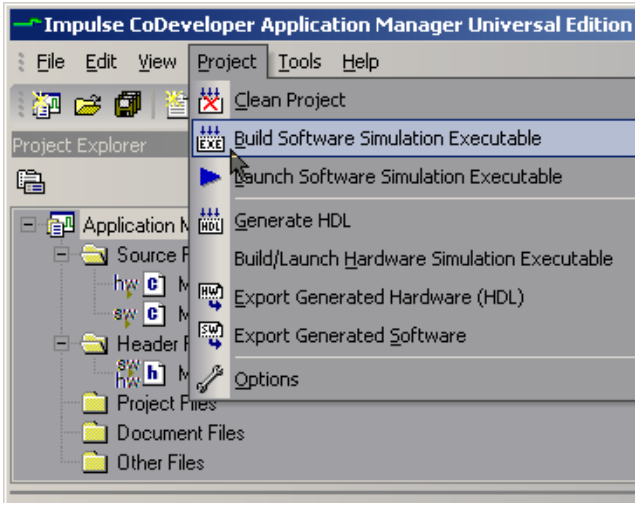


Figure 49 - Build Simulation Desktop in CoDeveloper using pull-down menu

Or via toolbar:

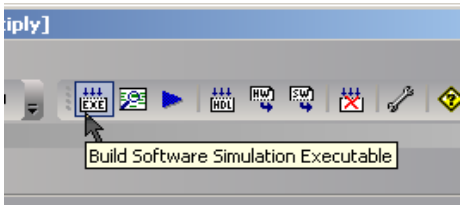


Figure 50 - Build Simulation Desktop in CoDeveloper using toolbar icon

Note the compiler output in the CoDeveloper IDE “Build” window:

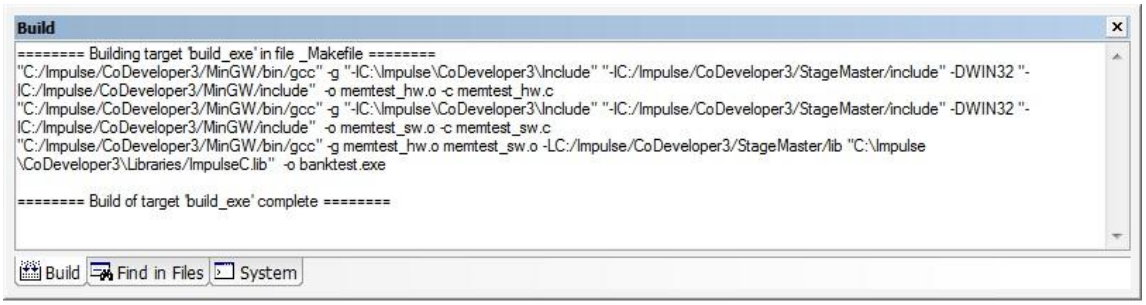


Figure 51 - Output within the CoDeveloper IDE build window

9.6. Running Desktop Simulation Executable

Launch the desktop software simulation executable via “Project” menu:

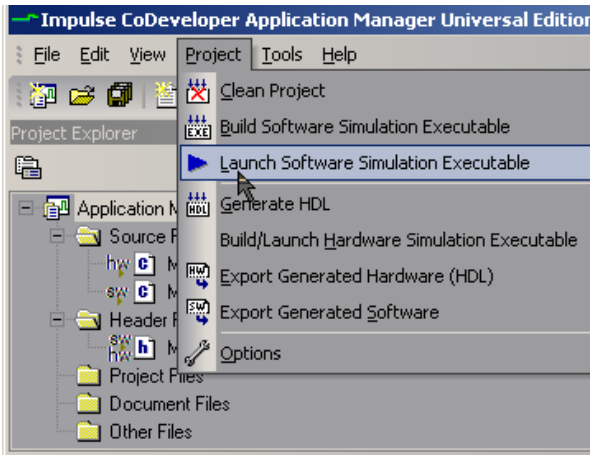


Figure 52 - Launch software simulation window using pull-down menu

Or via toolbar:

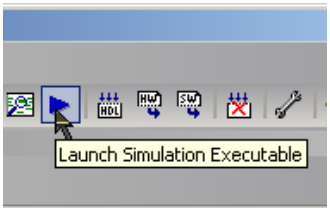


Figure 53 - Launch software simulation using toolbar icon

A command window will pop up in which the desktop simulation executable runs. Press “Enter” to exit:

Figure 54 - Pop-up window during desktop simulation

9.7. Project Setup Before Hardware/Software Generation and Export

Settings within the CoDeveloper IDE necessary for generating and exporting both hardware and software using this PSP are summarized below:

- Platform Support Package: “Pico M-501 Linux (VHDL)”
- Hardware export directory: <user hardware export directory>
- Software export directory: <user software export directory>
- Unsupported settings include:
    - Generate dual clocks (must be unchecked)
    - Active-low reset (must be unchecked)
    - Include floating point library (must be unchecked)

An example of these settings as it appears in the Memtest example:

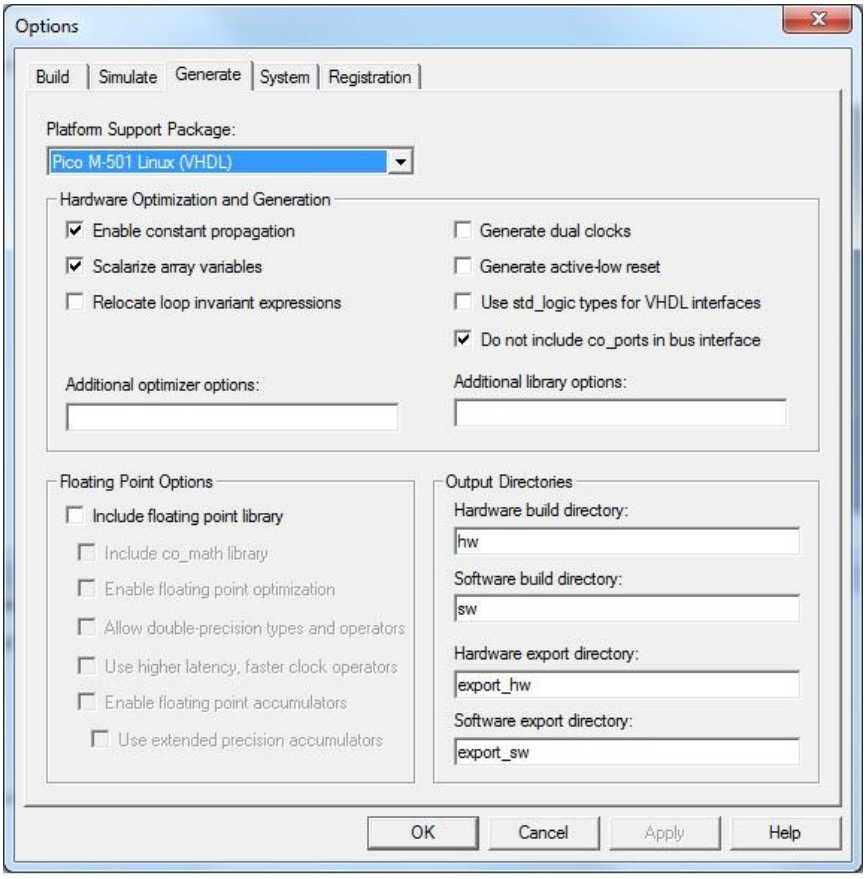


Figure 55 - Project setup to pick Platform Support Package

9.8. Generating Hardware

Generate hardware via “Project” menu:

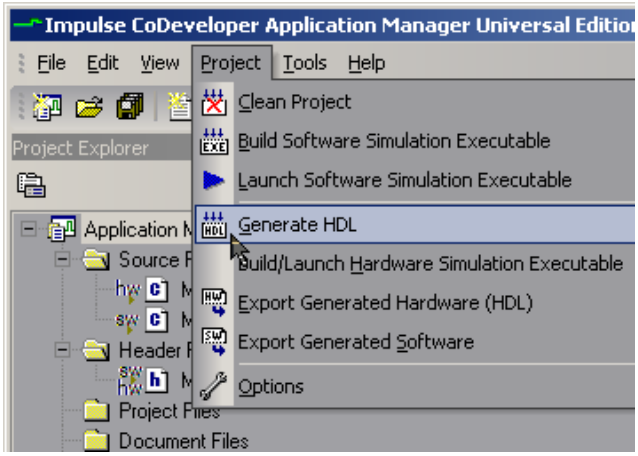


Figure 56 - Generate HDL using pull-down menu

Or via toolbar:

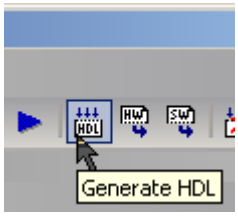


Figure 57 - Generate HDL using toolbar icon

Final results will appear in the directory specified during project setup in “Hardware build directory”. Note the final output in the CoDeveloper IDE’s “Build” window:

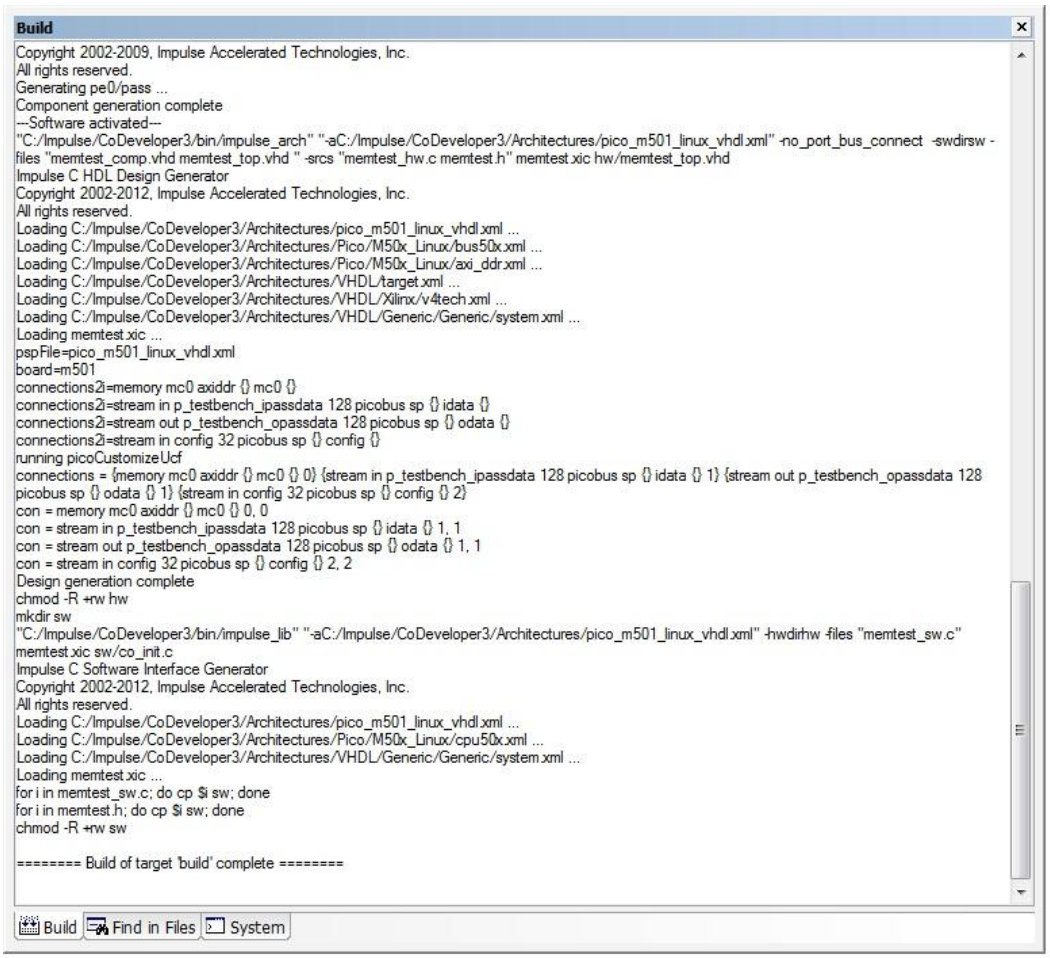


Figure 58 - Build window output

9.9. Exporting Hardware

Export hardware via “Project” menu:

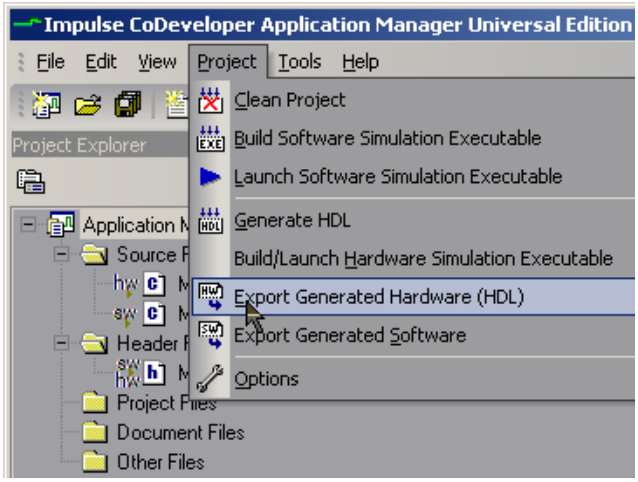


Figure 59 - Export Generated Hardware (HDL) using pull-down menu

Or via toolbar:

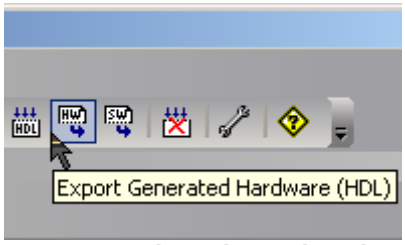


Figure 60 - Export Generated Hardware (HDL) using toolbar icon

Final results will appear in the directory specified during project setup in “Hardware export directory”. Note the final output in the CoDeveloper IDE’s “Build” window:

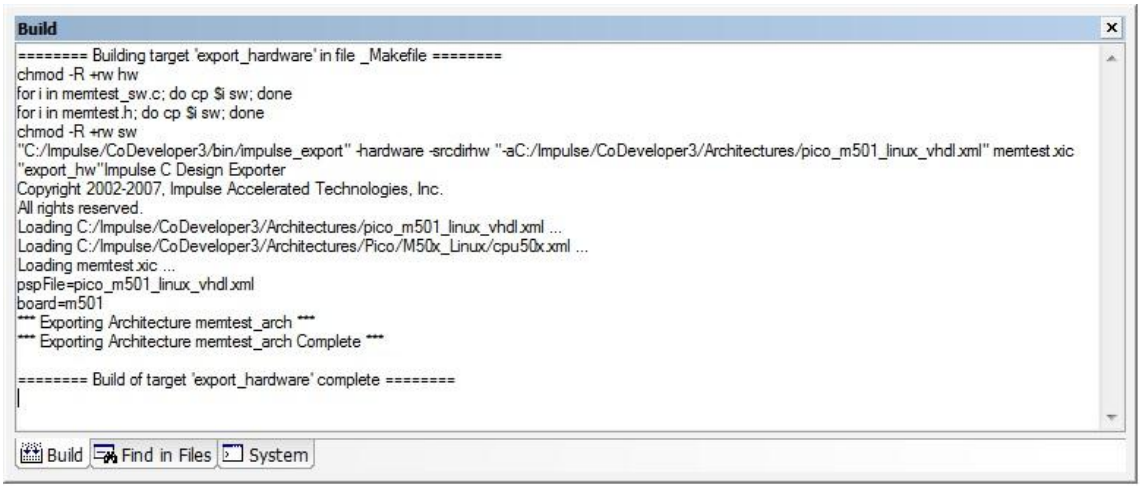


Figure 61 - Build window output

9.10. Compiling FPGA in Xilinx ISE 13.4

After exporting hardware, under the specified hardware export directory will be a directory structure that includes all necessary files for building the FPGA binary.

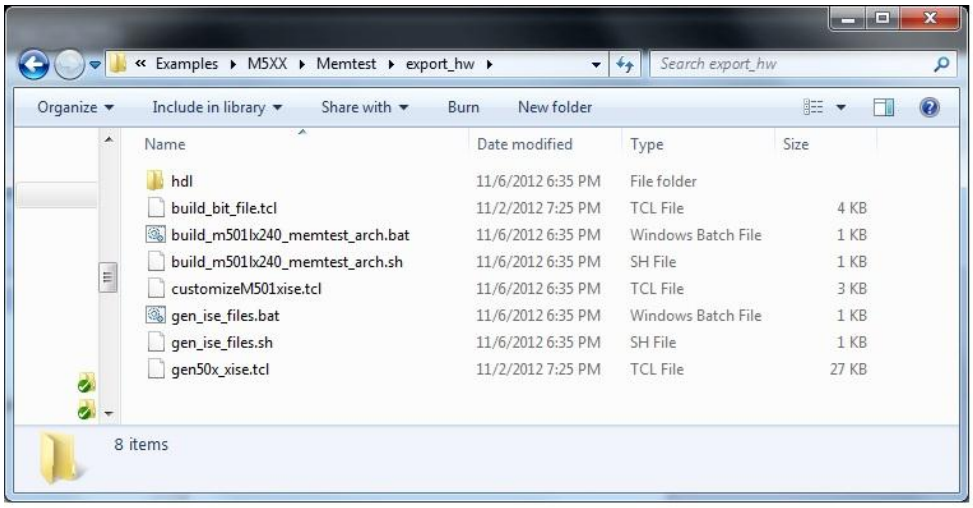


Figure 62 - Compiling FPGA in Quartus directory structure

At this point either Windows or Linux may be used to produce the bitfile running Xilinx ISE either through the GUI or from command line. Windows and Linux both use the following similar steps. Choose the one that suits your needs.

Windows:

1. Generate the ISE project files by running “gen_ise_files.bat”
2. Build the bit file using Xilinx ISE either by:
    a. Running the generated “build_m50*.bat” file
    b. Launching the Xilinx ISE GUI opening the newly built “m50*.xise” file

Linux:

1. Generate the ISE project files by running “gen_ise_files.sh”
2. Build the bit file using Xilinx ISE either by:
    a. Running the generated “build_m50*.sh” file
    b. Launching the Xilinx ISE GUI opening the newly built “m50*.xise” file

These methods will generate the necessary bitfile to be loaded on the Host System and are outlined in the sections that follow.

9.10.1. Building bitfile under Windows

9.10.1.1. Generate Xilinx ISE Project Files

First generate the ISE 13.4 project by executing “gen_ise_files.bat”. This process will create an ISE project file as well as all the necessary HDL files and corgen components used to generate the bitfile when building in ISE.

1. Open a command window
2. Change directories to “export_hw”
3. Execute “gen_ise_files.bat” as shown in Figure 58.
4. Verify that the files were created successfully by viewing the end of log file, as shown in Figure 59. “Successful!” should be the last line in the log file.

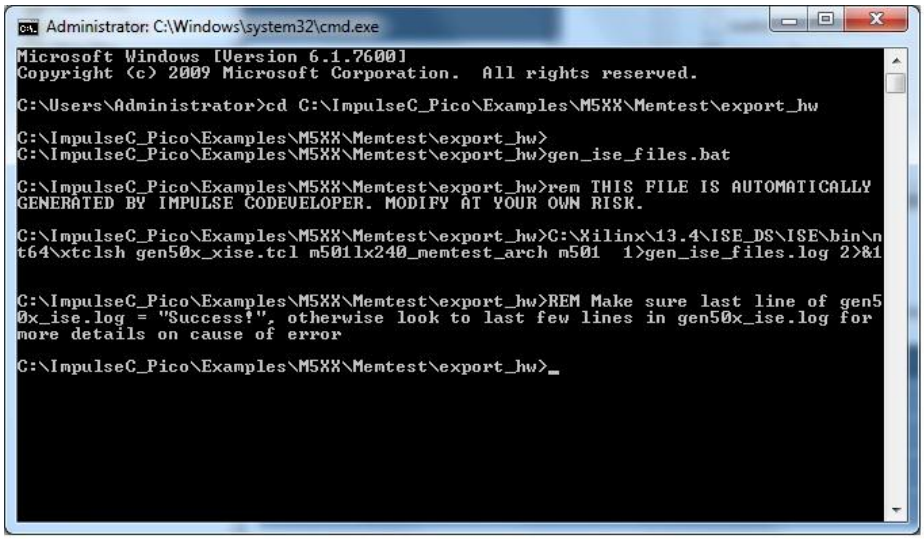


Figure 63 - Generate ISE project files

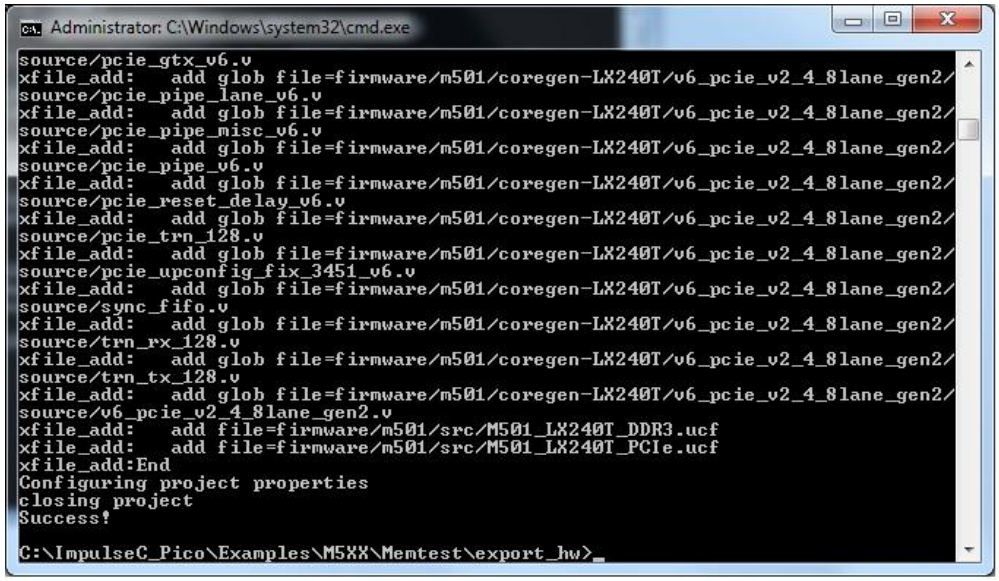


Figure 64 - Expected Gen_Ise_File log report

There are two method to produce the bitfile. The first is to launch the Xilinx ISE GUI and generate the bitfile. The second method it to execute the build script in the **export_hw** directory. Both method will generate the necessary bitfile to be loaded on the Host System.

9.10.1.2. Compiling FPGA in Xilinx ISE GUI

Launch the Xilinx ISE GUI and open the ISE project located in the **export_hw** directory. Select Pico_Toplevel in the *Hierarchy* window and double-click "generate programming file" in the *Processes* window.

NOTE: Xilinx ISE 13.4 will ask the user to regenerate three fifos (a one time process). The following figures illustrate initial project launch, regenerate the three corgen fifos, and final output with timing score equal to zero.

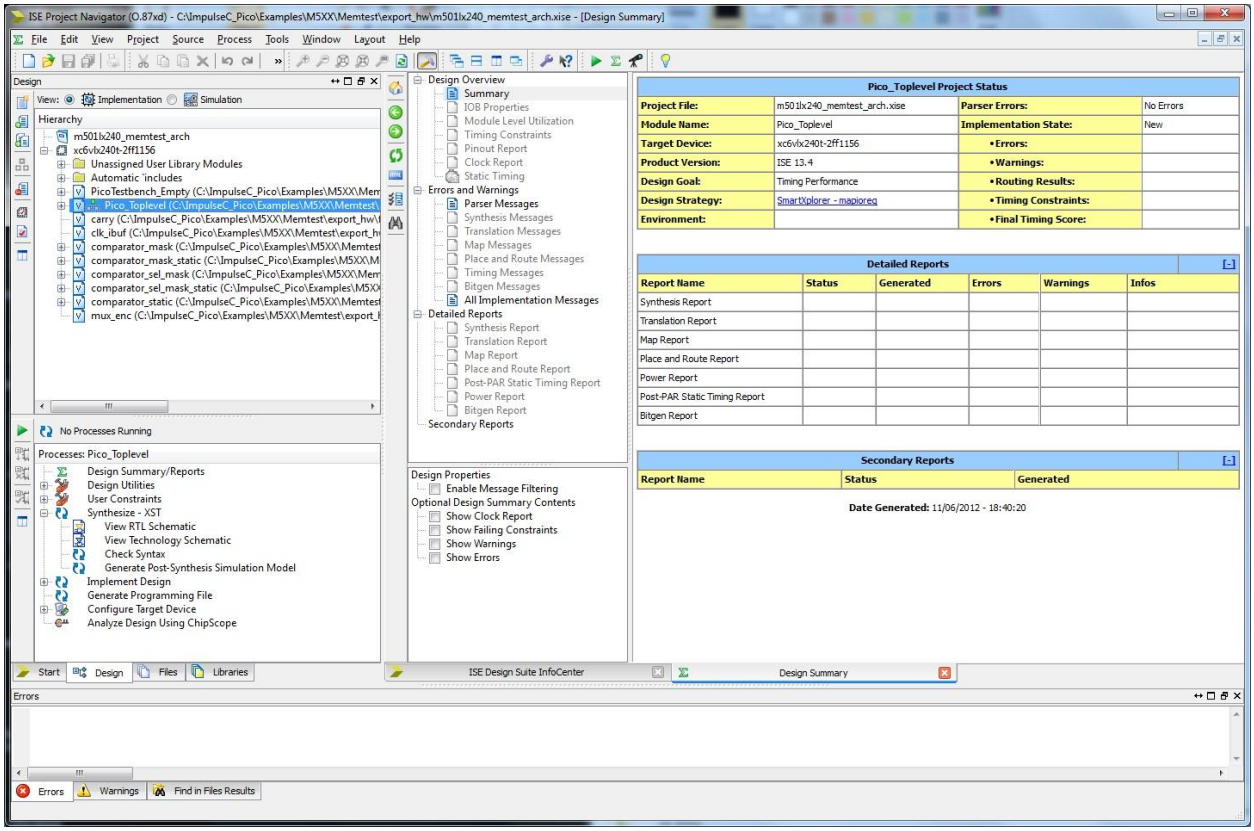


Figure 65 - Initial Xilinx ISE 13.4 GUI screen

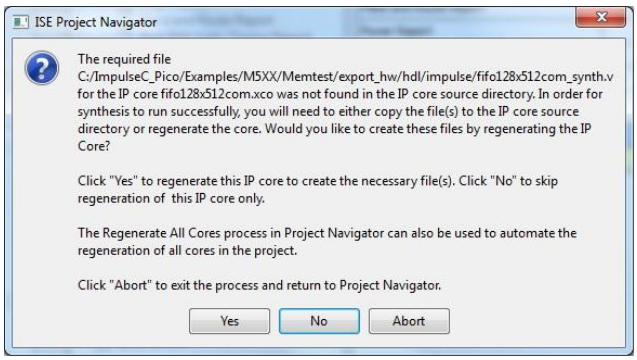


Figure 66 - Xilinx ISE 13.4 regenerate IP core fifo128x512com

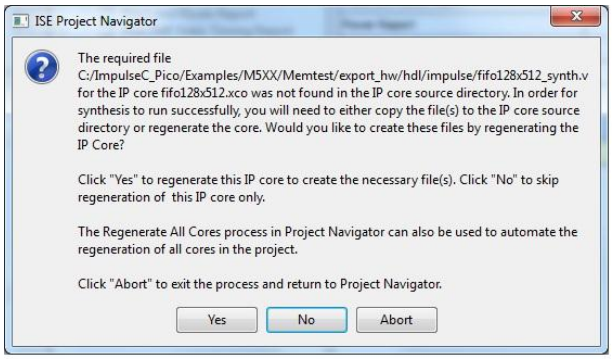


Figure 67 - Xilinx ISE 13.4 regenerate IP Core fifo128x512

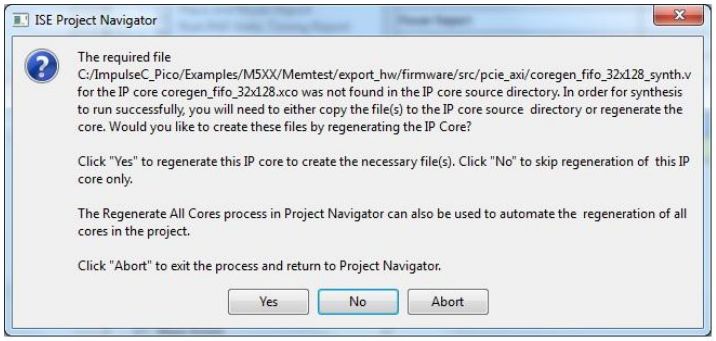


Figure 68 - Xilinx ISE 13.4 regenerate IP Core coregen_fifo_32x128

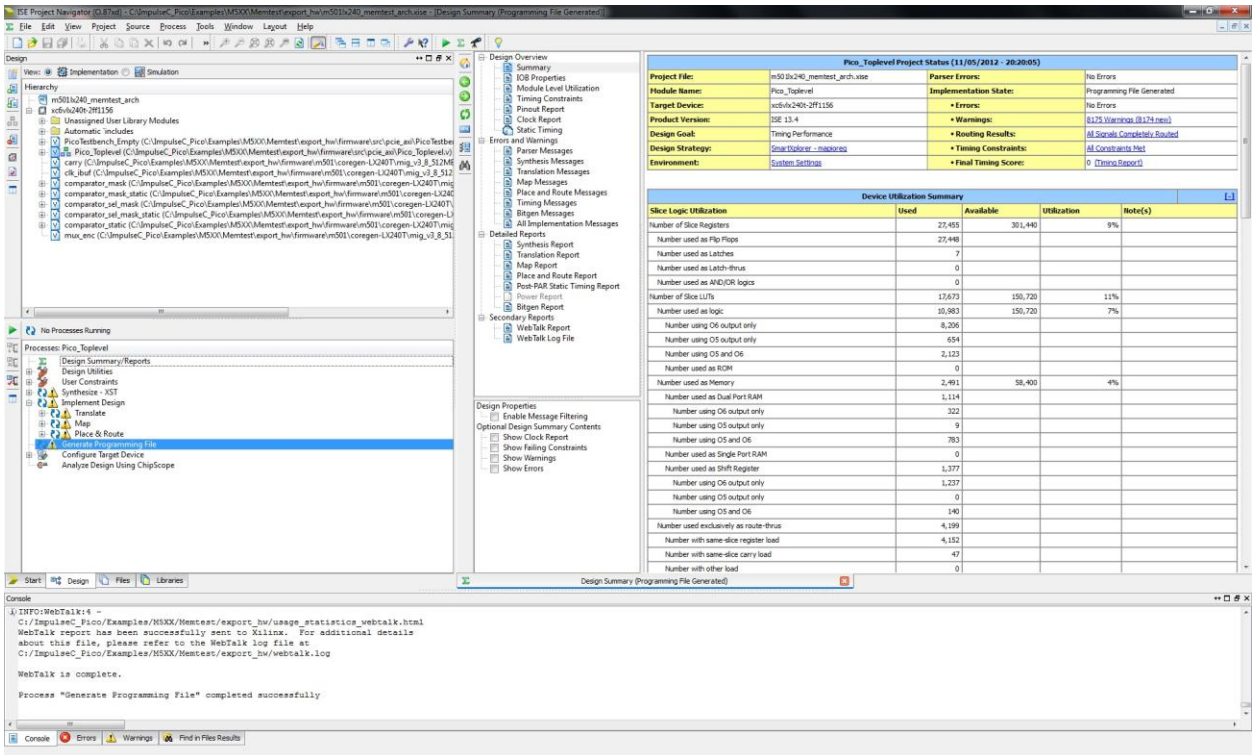


Figure 69 - Xilinx ISE 13.4 with timing score = 0

Once ISE 13.4 has completed, a bitfile “pico_toplevel.bit” will be present in the **export_hw** directory.

9.10.1.3. Compiling FPGA in Xilinx ISE using Command Line

In the top directory there will be the batch file “build_memtest_arch.bat” used to automatically run Xilinx ISE 13.4 to create the necessary .bit file used to program the M501 FPGA.

1. Open a command window
2. Execute “build_m501lx240_memtest_arch.bat”.

A command window will appear showing the FPGA build process (primarily made up of many, many info and warning messages). Compile time will vary by machine depending upon project size. When completed successfully, something similar to the following will appear and a bitfile “pico_toplevel.bit” will be present in the **export_hw** directory.

9.10.2. Building bitfile under Linux

Please follow the steps outlined in the Passthrough example, **Sections 8.9.1 – Building bitfile under Linux**, in order to build the bitfile for the Memtest example under Linux. It is left to the user, as an exercise, to successfully build the bitfile.

9.11. Exporting Software

The software application to be run on the host computer (with the M501 installed with it's drivers) can be exported in CoDeveloper.

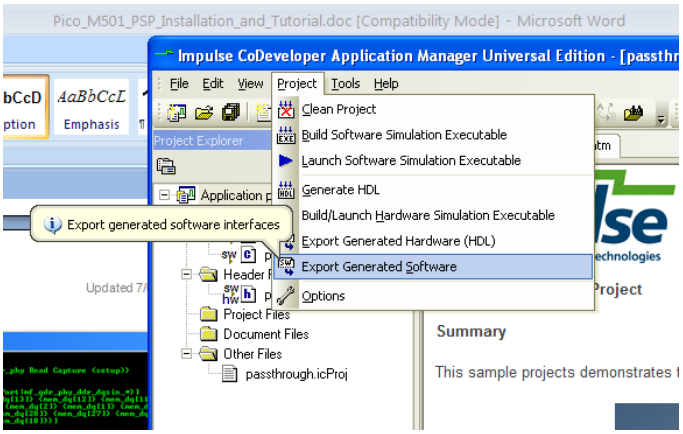


Figure 70 - Export Generated Software

Once completed without error in the Build window, it should be noted that the software code will be written to a newly created directory. The user can modify the target directory name. In this example, export_sw contains the exported software files.

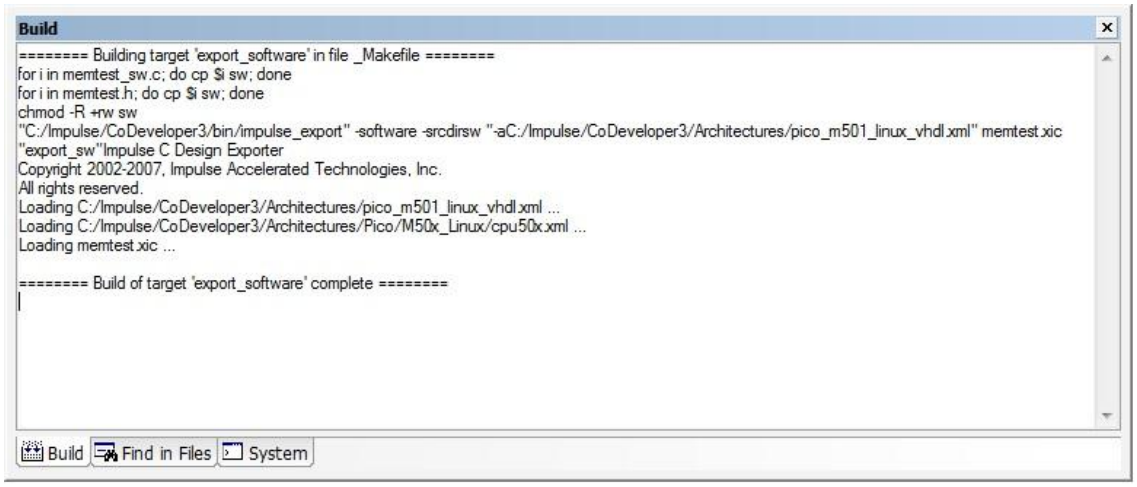


Figure 71 - Build window output

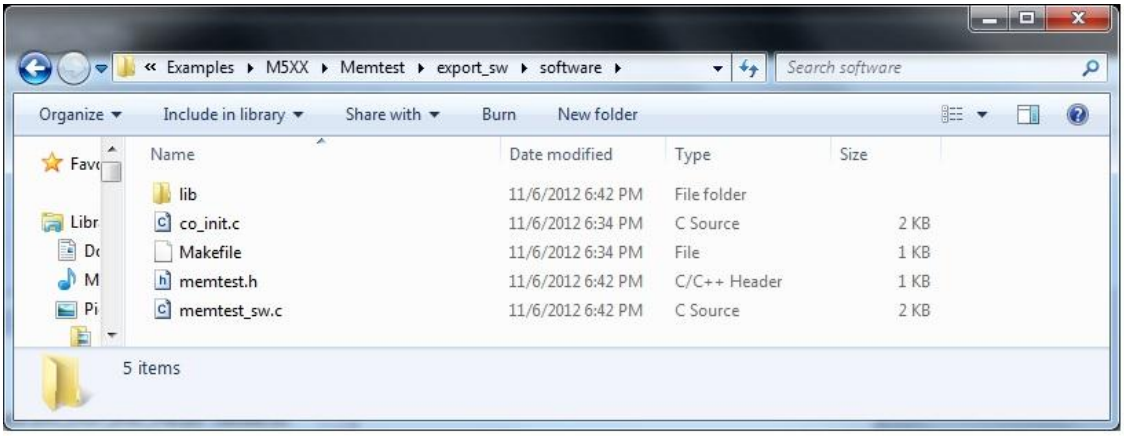


Figure 72 - Exported software directory

9.12. Programming the FPGA

The compiled software application in the “export_sw” directory will program the FPGA on the M501. Please continue to the next section.

9.13. Running Target Executable on the Host System

The Host System will need the following files:

1. Compiled bitfile output from ISE 13.4
2. Input data file
3. Application software source code
4. Makefile to compile the software

Please copy the following files and directory over to the Host System (figure 68). Create a folder (ie Pico/Memtest) and copy the following to that folder.

1. **export_hw** directory which includes “pico_toplevel.bit” bitfile
2. **export_sw** directory which includes the Makefile and software application source code

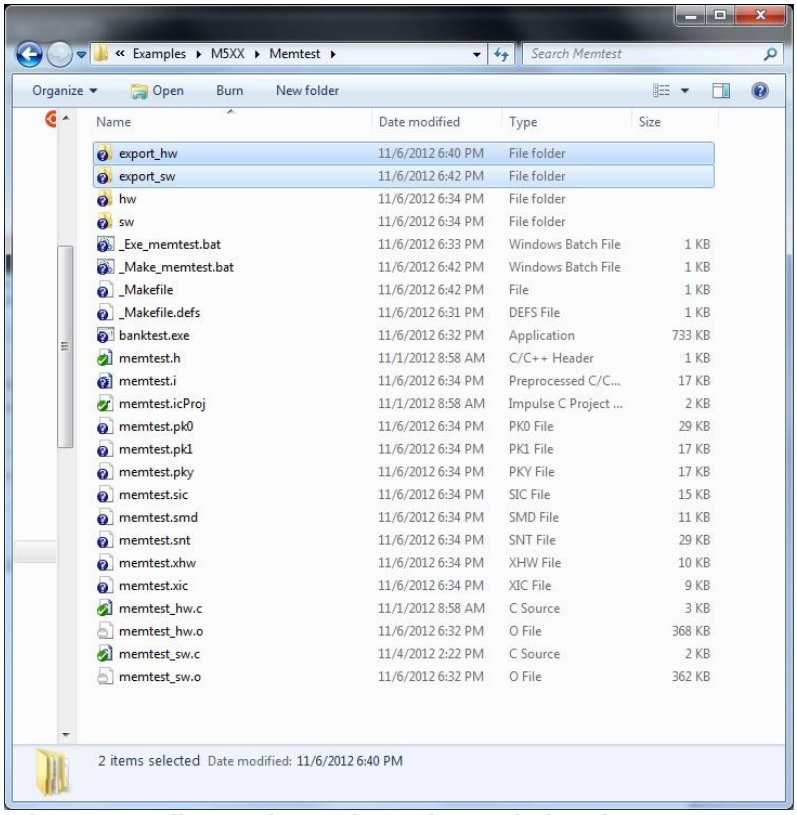


Figure 73 - Files & Directories to be copied to the Host System

The software application will load the FPGA every time it is executed and report the results upon completion. Please note that the process of loading the FPGA may take up

to 20 seconds due to the driver re-start sequence after the FPGA bitfile has been transferred to the card.

1. Open a terminal window.
2. Navigate to the location of your copied files and directories:

   (ie. cd Pico/Memtest/export_sw/software).
3. Run “make”
4. Execute the generated application: “./memtest_arch”

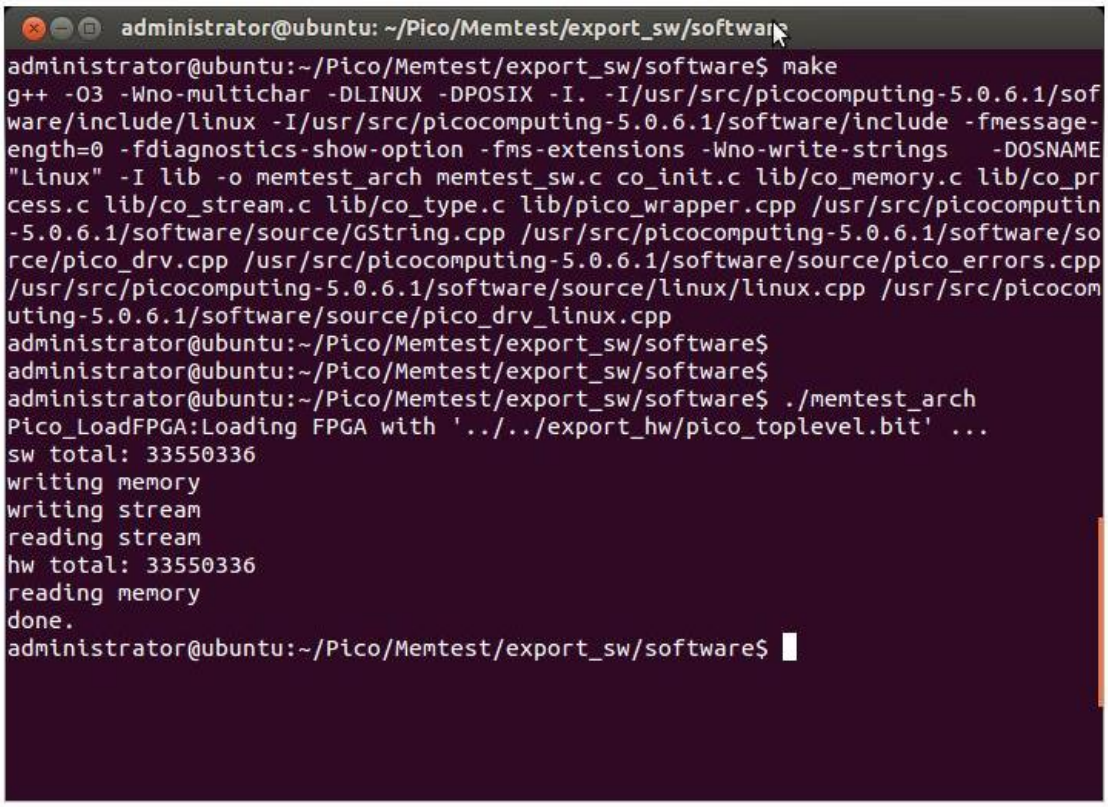


Figure 74 - Exported SW executed on target platform