



Center for Embedded Computer Systems
University of California, Irvine

Embedded System Environment (Front End)

ESE Version 2.0 evaluation

Tutorial

Daniel D. Gajski, Samar Abdi, Gunar Schirner, Han-su Cho, Yonghyun
Hwang, Lochi Yu, Ines Viskic and Quoc-Viet Dang

Technical Report CECS-08-15

December 12, 2008

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
(949) 824-8919

{gajski, sabdi, hschirne, hscho, yonghyuh, lochi.yu, iviskic,
qpdang}@uci.edu

<http://www.cecs.uci.edu/~ese>



Center for Embedded Computer Systems
University of California, Irvine

**Embedded System Environment (Front End): ESE Version 2.0 evaluation;
Tutorial**

Daniel D. Gajski, Samar Abdi, Gunar Schirner, Han-su Cho, Yonghyun Hwang, Lochi Yu, Ines Viskic and Quoc-Viet Dang

Technical Report CECS-08-15

December 12, 2008

.

Center for Embedded Computer Systems

University of California, Irvine

Irvine, CA 92697-2620, USA

(949) 824-8919

.

{gajski, sabdi, hschirne, hscho, yonghyuh, lochi.yu, iviskic, qpdang}@uci.edu

<http://www.cecs.uci.edu/~ese>

Copyright © 2008 CECS, UC Irvine

Table of Contents

1. Introduction.....	1
1.1. Motivation.....	1
1.2. Embedded System Environment.....	2
1.3. ESE Front End Design Flow.....	3
1.4. Design Example.....	4
1.4.1. JPEG Encoder.....	4
1.4.2. MP3 Audio Decoder.....	4
2. Multi-Processor System Design with ESE.....	7
2.1. ESE Startup and Settings.....	8
2.1.1. Environment Setup.....	9
2.1.2. ESE Demonstration Setup.....	10
2.1.3. Launching ESE.....	11
2.1.4. ESE GUI.....	12
2.1.5. Editing Database Preferences.....	13
2.1.6. Select Database File.....	14
2.2. Platform Creation.....	15
2.2.1. Open Partial Design.....	16
2.2.2. View Partial Design.....	17
2.2.3. Add Processing Element.....	18
2.2.4. View PE Properties.....	19
2.2.5. Assign New Name to PE.....	20
2.2.6. Add Port to PE.....	20
2.2.7. Connect PE to Bus.....	22
2.3. Mapping Application to Platform.....	23
2.3.1. Add Application Process.....	24
2.3.2. Assign Name to New Process.....	25
2.3.3. Add C Source File.....	26
2.3.4. Select C Source File.....	27
2.3.5. Add Process Ports.....	28
2.3.6. View Application Channels.....	30
2.3.7. Add New Application Channel.....	31
2.3.8. Channel Wizard.....	32
2.3.9. View New Channel Communication.....	34
2.4. Generating Functional and Timed TLMs.....	35
2.4.1. Generate Functional TLM.....	36
2.4.2. Simulate Functional TLM.....	37
2.4.3. View Functional Simulation Results.....	38
2.4.4. Generate Timed TLM.....	39

2.4.5. Simulate Timed TLM.....	40
2.4.6. View Timed Simulation	41
2.5. TLM Performance Estimation	42
2.5.1. View Performance Estimates	43
2.5.2. PE, Process and Function Level Estimates	44
2.5.3. View Communication Estimates	45
2.5.4. Bus and Channel Level Estimates.....	46
3. Heterogeneous System Design with ESE	47
3.1. ESE Startup and Settings	48
3.1.1. Environment Setup.....	49
3.1.2. ESE Demonstration Setup.....	50
3.1.3. Launching ESE	51
3.1.4. ESE GUI	52
3.1.5. Editing Database Preferences	53
3.1.6. Select Database File	54
3.2. Platform Creation.....	55
3.2.1. Open Partial Design	56
3.2.2. View Partial Design	57
3.2.3. Add Processing Element	58
3.2.4. View PE Properties	59
3.2.5. Assign New Name to PE.....	60
3.2.6. Add Port to PE	60
3.2.7. Connect PE to Bus	62
3.3. Mapping Application to Platform	63
3.3.1. Add Application Process.....	64
3.3.2. Assign Name to New Process	65
3.3.3. Add C Source File.....	66
3.3.4. Select C Source File.....	67
3.3.5. Add Process Ports	68
3.3.6. View Application Channels	70
3.3.7. Add New Application Channel	71
3.3.8. Channel Wizard.....	72
3.3.9. View New Channel Communication.....	73
3.4. Generating Functional and Timed TLMs.....	74
3.4.1. Generate Functional TLM.....	75
3.4.2. Simulate Functional TLM.....	76
3.4.3. View Functional Simulation Results.....	77
3.4.4. Generate Timed TLM	78
3.4.5. Simulate Timed TLM.....	79

3.4.6. View Timed Simulation	80
3.5. TLM Performance Estimation	81
3.5.1. View Performance Estimates	82
3.5.2. PE, Process and Function Level Estimates	83
3.5.3. View Communication Estimates	84
3.5.4. Bus and Channel Level Estimates	85
4. Multi-threaded System Design with ESE	87
4.1. ESE Startup and Settings	88
4.1.1. Environment Setup	89
4.1.2. ESE Demonstration Setup	90
4.1.3. Launching ESE	91
4.1.4. ESE GUI	92
4.1.5. Editing Database Preferences	93
4.1.6. Select Database File	94
4.2. Platform Creation	95
4.2.1. Open Partial Design	96
4.2.2. View Partial Design	97
4.2.3. Add Processing Element	98
4.2.4. View PE Properties	99
4.2.5. Assign New Name to PE	100
4.2.6. Add Port to PE	100
4.2.7. Connect PE to Bus	102
4.3. Mapping Application to Platform	103
4.3.1. Add Application Process	104
4.3.2. Assign Name to New Process	105
4.3.3. Add C Source File	106
4.3.4. Select C Source File	107
4.3.5. Add Process Ports	108
4.3.6. View Application Channels	110
4.3.7. Add New Application Channel	111
4.3.8. Channel Wizard for Inter-Process Communication	112
4.3.9. Channel Wizard for Intra-Process Communication	114
4.3.10. View New Channel Communication	115
4.3.11. Add RTOS	116
4.4. Generating Functional and Timed TLMs	117
4.4.1. Generate Functional TLM	118
4.4.2. Simulate Functional TLM	119
4.4.3. View Functional Simulation Results	120
4.4.4. Generate Timed TLM	121

4.4.5. Simulate Timed TLM.....	122
4.4.6. View Timed Simulation	123
4.5. TLM Performance Estimation	124
4.5.1. View Performance Estimates	125
4.5.2. PE, Process and Function Level Estimates	126
4.5.3. View Communication Estimates	127
4.5.4. Bus and Channel Level Estimates.....	128
5. Conclusion	129
References	131

Chapter 1. Introduction

The basic purpose of this tutorial is to guide a user through our Embedded System Environment (ESE) Front End. ESE helps designers to take C/C++ application processes and graphical platform capture and automatically produce Transaction Level Models (TLMs) for functional verification and performance estimation. Extensive information about ESE and its projected impact on embedded system design processes is available on our website at <http://www.cecs.uci.edu/~ese>

The tutorial demonstrates ESE Front End being used for TLM generation using the JPEG encoder and MP3 decoder applications. Three platforms are used for this purpose. The first platform consists of five microprocessors connected via a shared bus and communicating each other using a memory architecture. This platform is representative of a multi-processor design where all components are programmable. The second platform demonstrates usage of ESE for heterogeneous system design with one microprocessor and four HW accelerators. The HW Intellectual Properties (IPs) have a proprietary bus protocol which requires a protocol convertor between the processor bus and IP bus. The last platform has two multi-threaded microprocessors to which several processes are mapped, thus it needs a Real-Time Operating System (RTOS) model to control the execution of the processes in a microprocessor. The design examples show the versatility of ESE, which is a huge benefit over manually written virtual platforms.

The tutorial gives a step by step illustration of using ESE Front End. Screenshots of the Graphical User Interface (GUI) are presented to aid the user in using the various features of ESE. Please note that, depending on your specific version of ESE and your system settings, the screen shots shown in this document may be slightly different from the actual display on your screen. The screenshots at each design step are supplemented with brief observations about the specific ESE feature. This would help the designer to gain an insight into the design process instead of merely following the demonstration steps. We wind up the tutorial with a conclusion and references.

1.1. Motivation

The rise in complexity of modern design has forced system designers to move to higher levels of abstraction above Register Transfer Level (RTL) and traditional cycle accurate design. Therefore, models such as TLMs that provide manyfold speedup over RTL simulation are being used. However, in order for TLMs to be synthesizable to Hardware (HW) and Software (SW) implementation, they must follow well defined semantics. These semantics are currently missing in the industry and TLM standards. Moreover,

enforcing semantics is not easy with manual modeling.

Secondly, embedded application developers come from a variety of different engineering backgrounds and are not necessarily adept at electronic design. Model automation tools are needed for such developers so that they do not need to learn modeling languages such as SystemC.

Thirdly, businesses that use external suppliers for their embedded system designs need unambiguous executable specifications for design hand-off. An even better proposition would be to build pre-silicon board prototypes in house. This would reduce the chances for mis-communication in requirement specification and lead to a more robust design process. Consequently, tools are required that take abstract applications and platforms and quickly produce fast TLMs and board prototypes.

It is with these challenges in mind that we have come up with ESE that takes off the drudgery of manual modeling from system designers. It enables non-experts to create system models and generate board prototypes using a convenient graphical interface.

1.2. Embedded System Environment

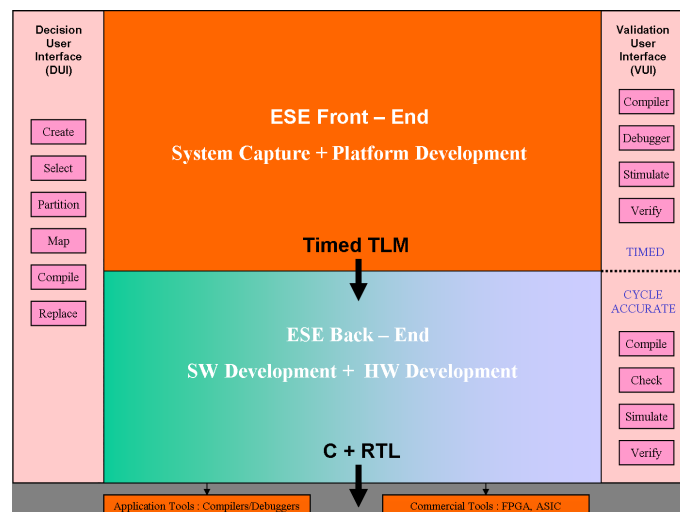


Figure 1-1. ES Environment

ESE consists of a Front-End and a Back-End supported by two interfaces as shown in

Section 1.2 *Embedded System Environment* (page 2). The Front-End consists of System Capture, which is a GUI for capturing the definition of the platform architecture and product application code. Platform Development tool generates timed TLMs of the platform architecture executing the product application defined by the capture tool. These timed TLMs provide reliable performance metrics and are used for early exploration of design choices. In the Back-End, the HW Development component is used to generate cycle-accurate or RTL description of the HW components which can be further refined by commercially available tools for Application-Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA) manufacturing. SW Development generate firmware necessary to run communication and application SW on the platform. Validation User Interface is used to debug and validate developed SW and HW. Decision User Interface is used by the designer, to estimate the quality metrics and make decisions such as component selection, task scheduling, mapping of SW functions to HW components and others.

1.3. ESE Front End Design Flow

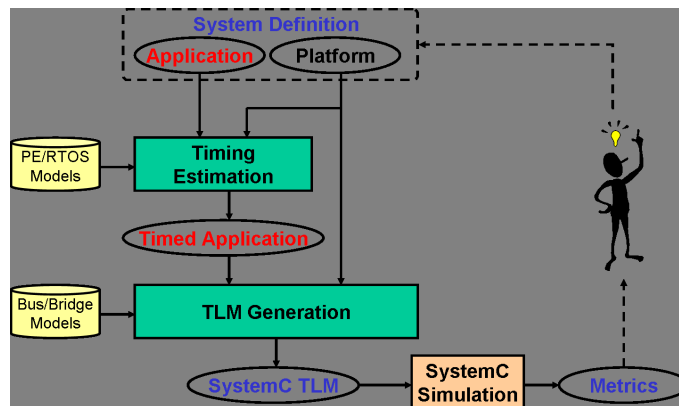


Figure 1-2. ES Environment

The inputs to ESE Front-End are the system definition consisting of a platform and application code. A library of processing elements, buses, bridges and RTOS is provided in ESE to develop such a platform. The retargetable timing estimation tool in ESE is used to annotate timing to the application code based on the mapping of application code on the platform components. The timed application and platform are input to the

TLM generator tool that uses the bus and bridge models to generate a SystemC TLM. This SystemC TLM can be simulated by any commercial or freely available SystemC simulator to provide the performance metrics. The designer can use the metrics to make to application code and/or the platform in order to optimize the system for a particular metric.

1.4. Design Example

To demonstrate the usefulness of ESE, two applications were chosen, JPEG encoder and MP3 decoder. JPEG encoder is used to demonstrate multiprocessor system design and system design including RTOS with ESE. MP3 decoder is used to demonstrate heterogeneous system design with ESE.

1.4.1. JPEG Encoder

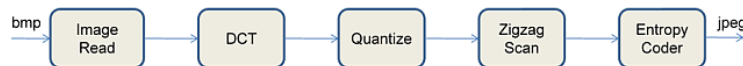


Figure 1-3. JPEG Encoder

Figure 1-3 shows the block diagram of JPEG encoder. It takes a BMP as an input and outputs an encoded JPEG file. In general, JPEG encoder consists of five processes. First, it partitions the image into 8x8 blocks of pixels and the blocks are applied to a 2-dimensional DCT. Next, the transform matrix is normalized by an 8x8 quantization matrix and the quantized DCT coefficients form a matrix. The elements of the matrix are ordered in a zigzag scan. Then, an entropy coder combined with a run-length coding of the zeros generates an efficient representation of the quantized coefficients to be transmitted or stored. The C model is used to create test benches with golden JPEG output files. These test benches are used later to verify the ESE generated TLMs.

1.4.2. MP3 Audio Decoder

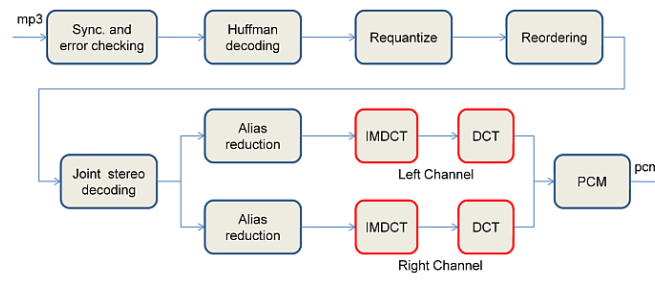


Figure 1-4. MP3 Decoder

MP3 decoder first reads a codeword via synchronous and error checker. Next, huffman decoder translates the codeword to several symbols using variable length decoding algorithm and sends it to next stages for requantizing and reordering. Then, the decoded frequency line is sent to alias reduction and IMDCT. Finally, DCT produces the output samples. The block diagram in Figure 1-4 shows the IMDCT and DCT transforms that are applied during the stereo decoding on the left and right channels of the MP3 input. These function blocks are the most time consuming part of the decoding and are hence ideal for implementation using custom HW for faster decoding. The C model is also used to create test benches with golden PCM output files. These test benches are used later to verify the ESE generated TLMs.

Chapter 2. Multi-Processor System Design with ESE

This section deals with design of JPEG encoder on a platform consisting of five MicroBlaze processors. The JPEG application code is available as a C model. The JPEG encoder has five processes and each process is mapped to a unique processor, thus the processes can be executed concurrently. The communication between the processes can take place through pairs of various channels such as process-to-process (or point-to-point) message passing channel, shared memory channel and First-In-First-Out (FIFO) channel. In this Chapter, all the channels in the JPEG encoder are via the FIFO channels. ESE provides well defined communication APIs for this purpose. The encoded output is shown graphically during the TLM simulation of the JPEG encoder.

The chapter starts by explaining the set up for ESE. It then shows, using screenshots, how the platform is created. To speed up the demonstration, and to emphasize on the features, we start with an existing partial platform that is upgraded with additional processors and a bus. Then we show the application mapping on the platform, followed by TLM generation, simulation and performance estimation. Thus, we present the core capabilities of the ESE Front-End tools in easy platform design & upgrade, model generation, validation and estimation.

2.1. ESE Startup and Settings

Before starting the demonstration, please ensure that you have the ESE software installed in the right location at `"/home/ease/local."` Also make sure that you have an `"/home/ease/local"` directory containing the SystemC 2.2.0 libraries and Simple Direct-media Layer (SDL) libraries that are needed for simulation of generated TLMs. Also make sure that you have GCC version 3.4 or higher because it is needed to correctly compile the generated TLMs. The demonstration shown here assumes the user to have a bourne shell. For C shell, the user may call the `".csh"` version of the setup scripts. Alternately, just use `"sh"` to create a new bourne shell and follow the tutorial directions.

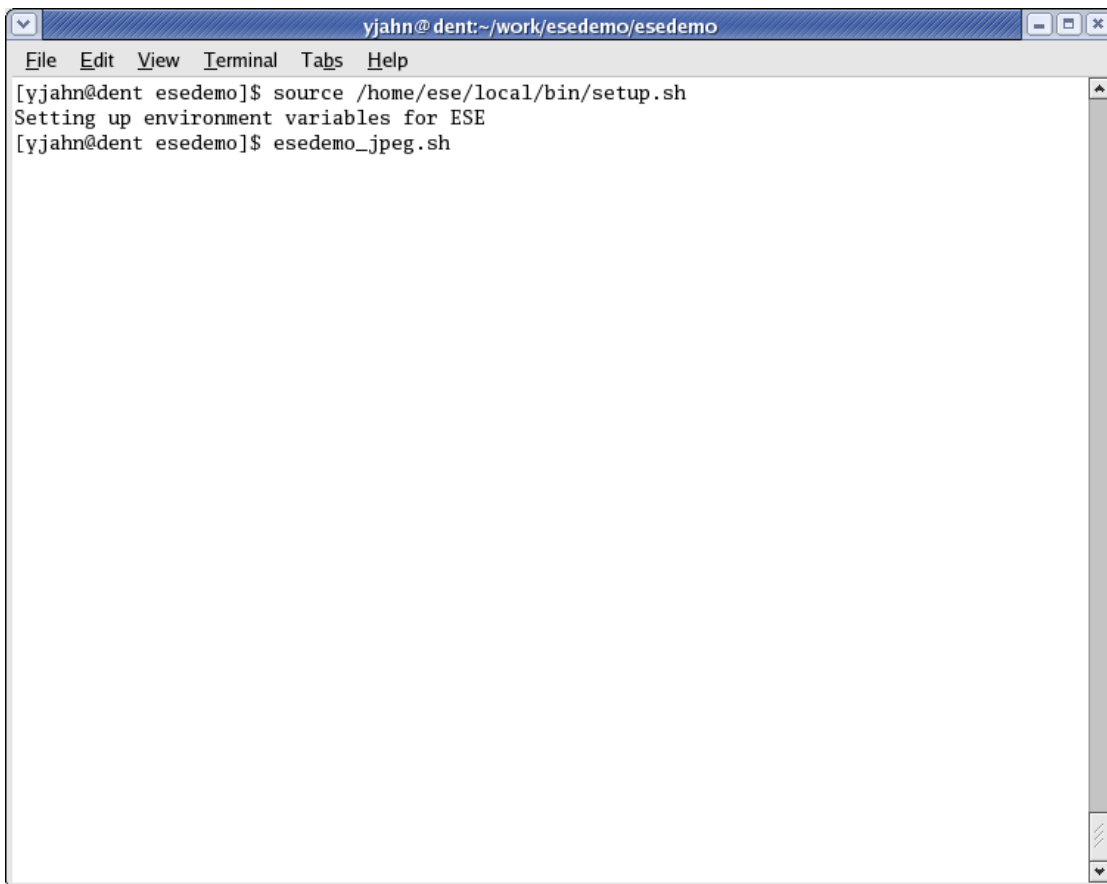
2.1.1. Environment Setup

A screenshot of a terminal window. The title bar at the top reads "yjahn@dent:~/work/esedemo/esedemo". Below the title bar is a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The main area of the terminal shows a command prompt "[yjahn@dent esedemo]" followed by the command "source /home/ease/local/bin/setup.sh". The command has been executed, and the prompt is now on a new line.

```
yjahn@dent:~/work/esedemo/esedemo
File Edit View Terminal Tabs Help
[yjahn@dent esedemo]$ source /home/ease/local/bin/setup.sh
```

We start by setting up the environment variables to access ESE binaries. This is provided by the "setup.sh" script in your installation. Typically, the installation path would be "/home/ease/local." The script is in the "bin" directory in the installation. The script modifies your PATH environmental variable to include path to ESE as well as the LD_LIBRARY_PATH variable to access the shared libraries that ESE depends on. Run the command "source /home/ease/local/bin/setup.sh" and create a new local directory for the demo.

2.1.2. ESE Demonstration Setup

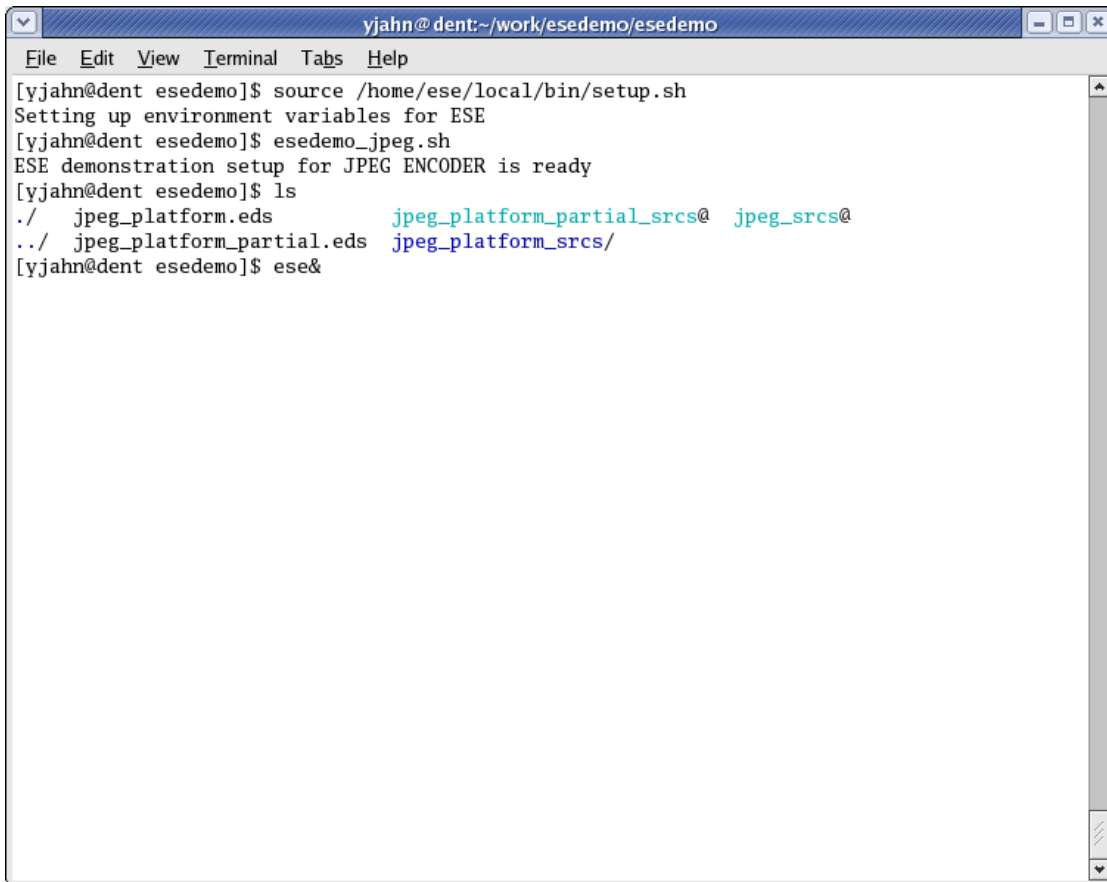


The screenshot shows a terminal window titled "yjahn@dent:~/work/esedemo/esedemo". The window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The terminal content shows the following commands and output:

```
[yjahn@dent esedemo]$ source /home/e/e/local/bin/setup.sh
Setting up environment variables for ESE
[yjahn@dent esedemo]$ esedemo_peg.sh
```

Once the environmental variables have been set, the user is ready to launch ESE and create his or her design. For the purposes of this tutorial, we will start with a partial design to quickly demonstrate the key capabilities of the toolset. We have created a shell script called "esedemo_mpd.sh" that prepares a partial design to start the demo for the JPEG encoder. At this point, run the "esedemo_mpd.sh" script after changing into the local directory created for the demo.

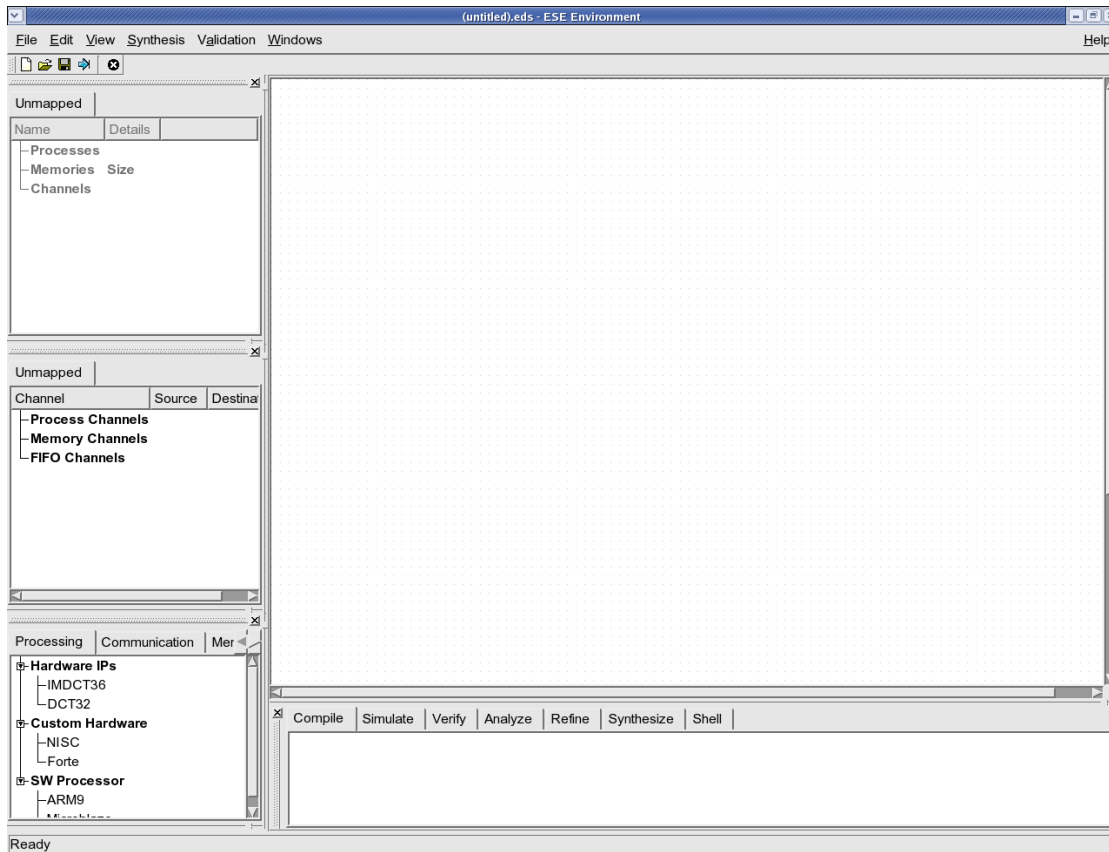
2.1.3. Launching ESE



```
yjahn@dent:~/work/esedemo/esedemo
File Edit View Terminal Tabs Help
[yjahn@dent esedemo]$ source /home/e/e/local/bin/setup.sh
Setting up environment variables for ESE
[yjahn@dent esedemo]$ esedemo_jpeg.sh
ESE demonstration setup for JPEG ENCODER is ready
[yjahn@dent esedemo]$ ls
./      jpeg_platform.eds          jpeg_platform_partial_srcs@  jpeg_srcs@
../     jpeg_platform_partial.eds  jpeg_platform_srcs/
[yjahn@dent esedemo]$ ese&
```

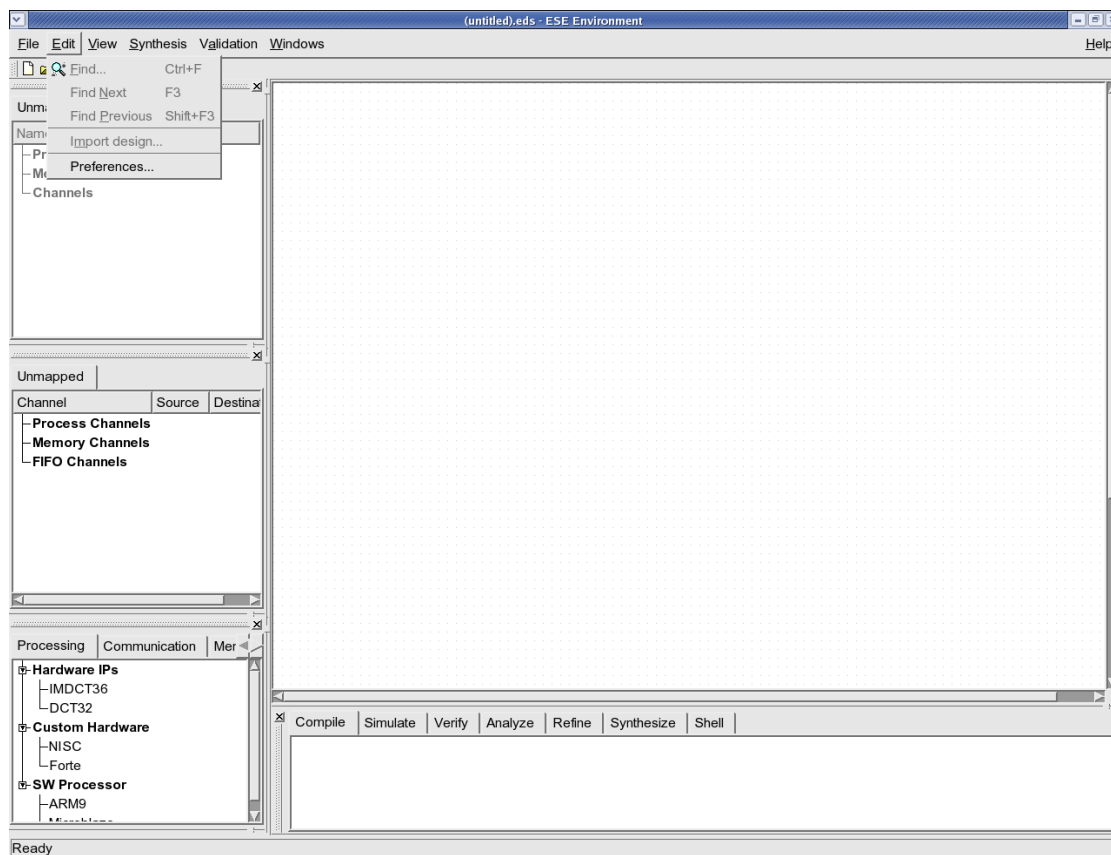
After running the "esedemo_mpd.sh" script, you will notice several files in the working directory. Some of these files will have a ".eds" extension. They are the ESE design files for the JPEG encoder design that we will be using for this demo. You may also see links to source directories. These point to the C code for the processes of the JPEG application. To launch the ESE GUI, simply run "ese" from your shell.

2.1.4. ESE GUI



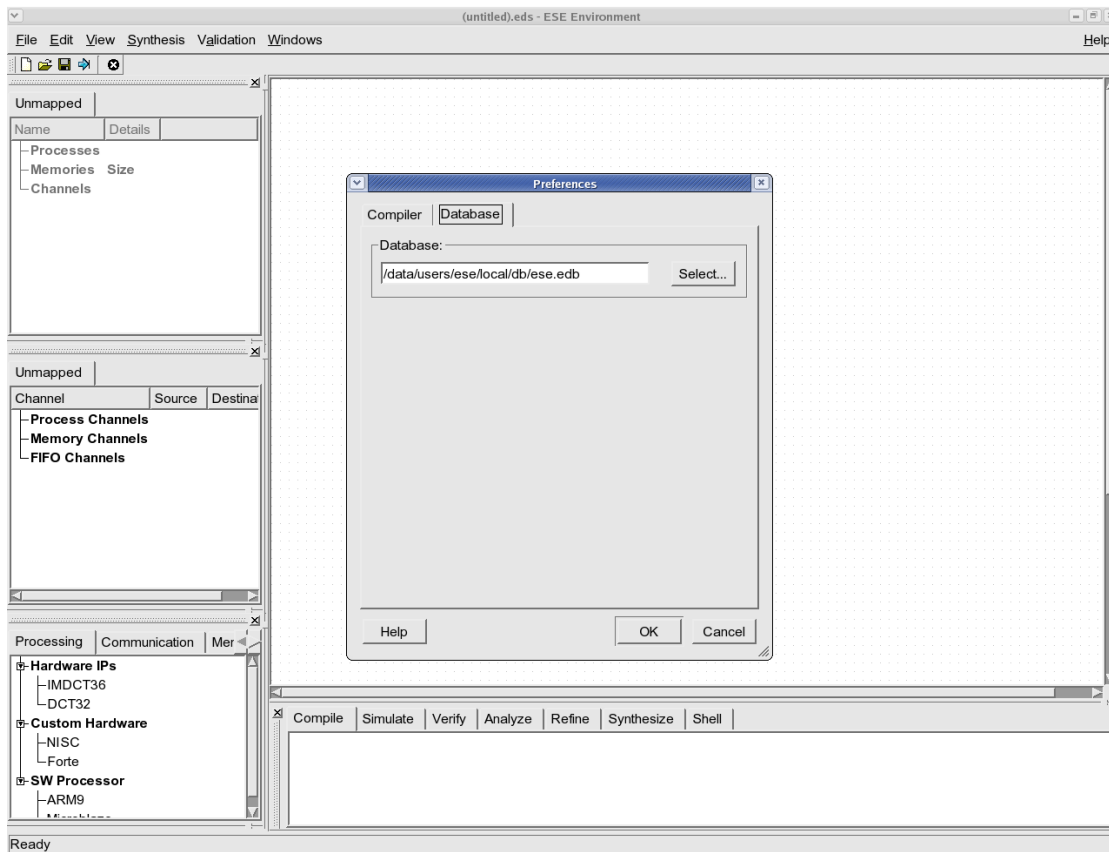
The ESE GUI should now appear as shown in the screenshot. The GUI has several menu items that we shall explore over this tutorial. It is divided into five windows. The top left window is the "PE" window. It organizes the various application processes mapped to the Processing Elements (PEs) in the design. The mid-left window is the "Channel" window that organizes the various channels used for communication between the application processes. The tabs represent the physical communication links in the platform. The bottom left window is the "Database" window that organizes the PE, Communication Element (CE), memory and RTOS model. The top right window is the "Platform Canvas" on which the platform architecture is edited graphically. The bottom right window is the "Logging" window that logs the messages from various ESE tools.

2.1.5. Editing Database Preferences



Before creating a new design, we must ensure that the components needed for our JPEG platform are accessible by the GUI. To do so, we edit the database preferences by selecting **Edit**→**Preferences** from the menu bar.

2.1.6. Select Database File

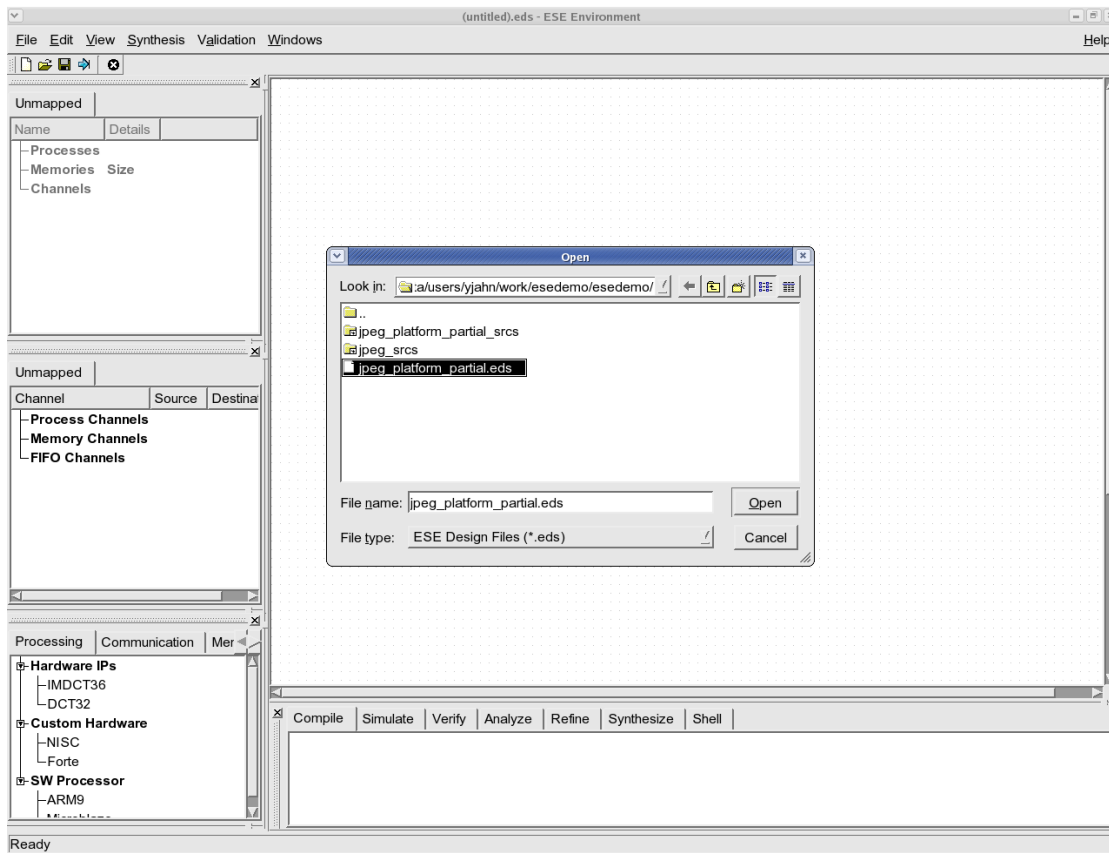


In the **Preferences** dialog, select the tab for **Database**. This will allow the user to browse for the database file that has a ".edb" extension. The database file needed for the JPEG demonstration already comes with the ESE installation. Typically, this file will be called "ease.edb" and will be located at "/data/users/ease/local/db/ease.edb." If the selection is not already there, please browse for the file and press **OK**. All the elements should now be visible in the database window, if they weren't already.

2.2. Platform Creation

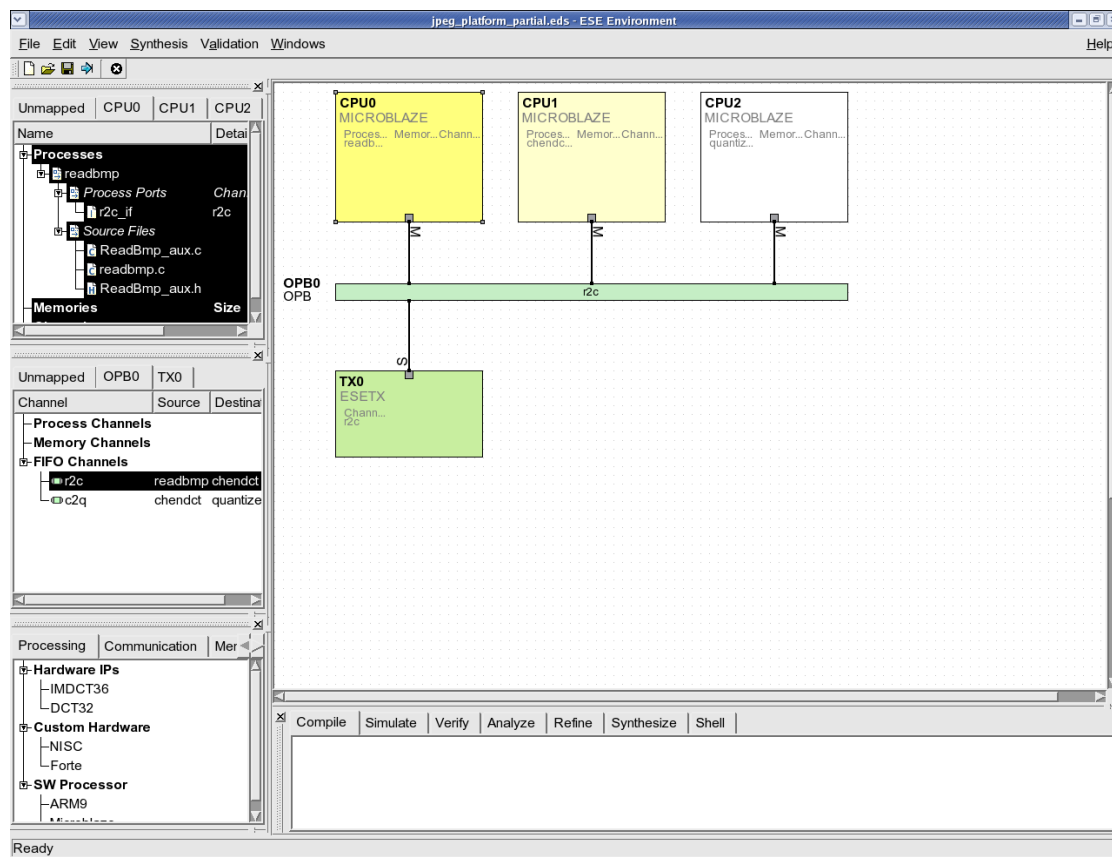
We will start by loading the multi-processor design of the JPEG encoder into ESE. As mentioned earlier, we will start with a partial platform consisting of three Microblaze processors. Each processor carries the application code for each process in the JPEG encoder. Two Microblaze processors for "zigzag" and "huffencode" processes will be added to the platform. In this section, we will show how to use the database and platform editor canvas to upgrade a multi-processor platform in ESE.

2.2.1. Open Partial Design



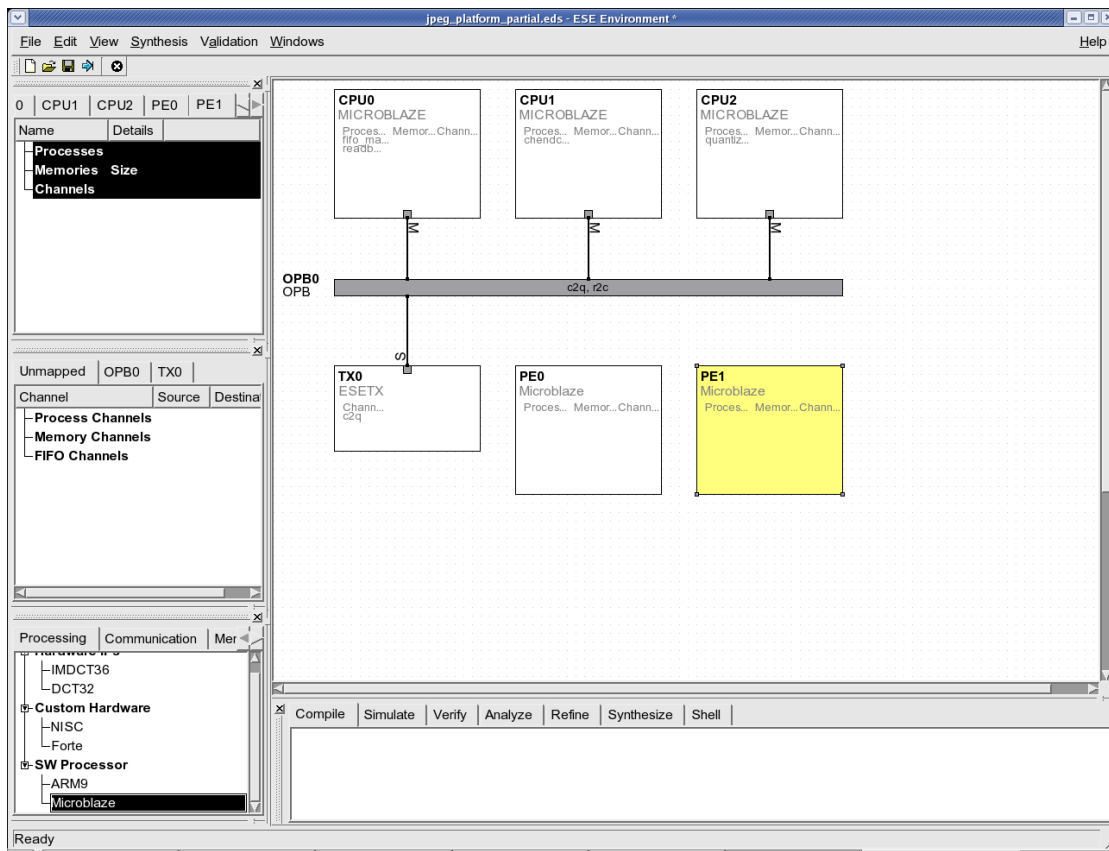
We begin by adding the already created partial design. The ESE designs are stored in XML based files with the extension ".eds." Select **File**→**Open** from the menu bar. Browse into the demo working directory and select "jpeg_mpd_platform_partial.eds." This is the design with the partial multi-processor design example. Press **Open** to open the design.

2.2.2. View Partial Design



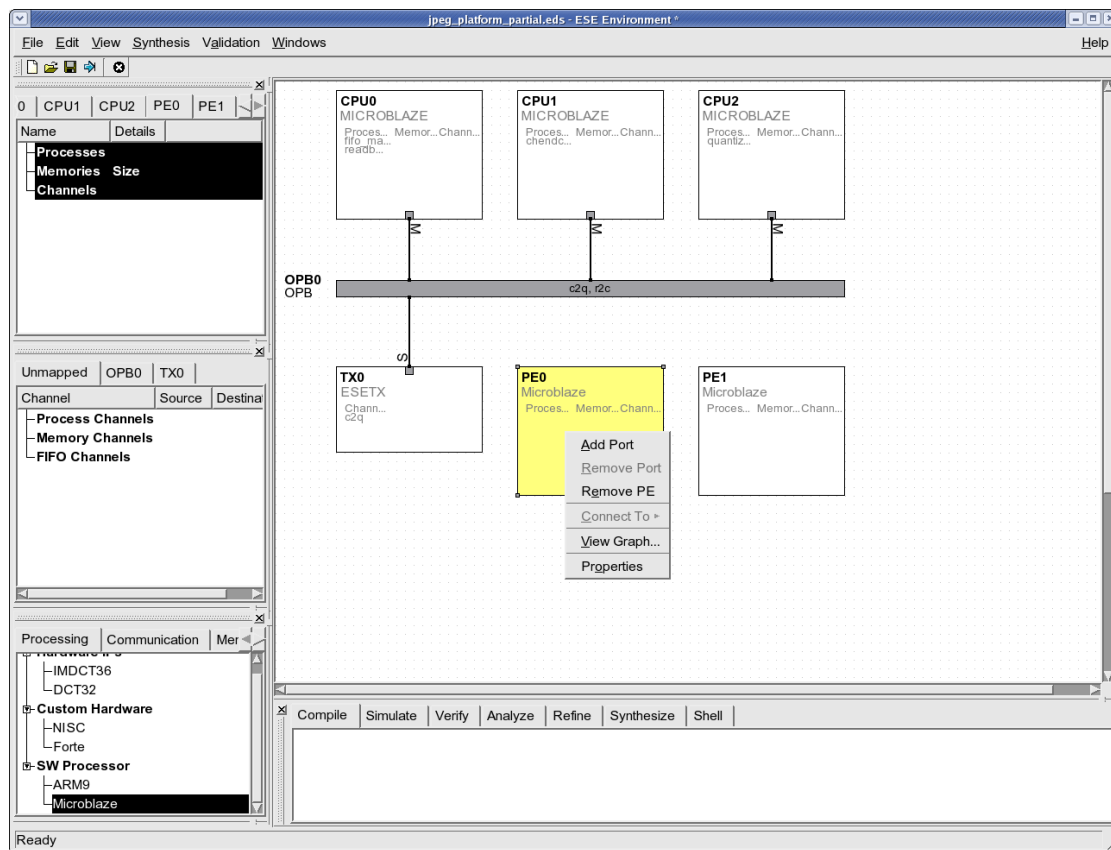
The partial platform will appear in the canvas as shown in the above screenshot. We can see three Microblaze processors CPU0 , CPU1 and CPU2 in the platform. These processors are connected via the Open Peripheral Bus (OPB). There are two FIFO channels in this partial design. Each process has its own process port and the process port is connected through the FIFO channel. For example, as shown in PE window, CPU0 has a process named "readbmp" and the process has a process port named "r2c_if" which is for sending data from "readbmp" to "chendct". And the process port is connected to a FIFO channel named "r2c" as shown in Channel window. Note that these processors are both connected as "Masters" as indicated by an "M" at the connecting port. Since bus masters cannot communicate directly over the bus, we provide a transducer (Tx0) which consists of a FIFO controller and FIFO memories. It acts as a shared memory for data transfer between CPU0, CPU1 and CPU2.

2.2.3. Add Processing Element



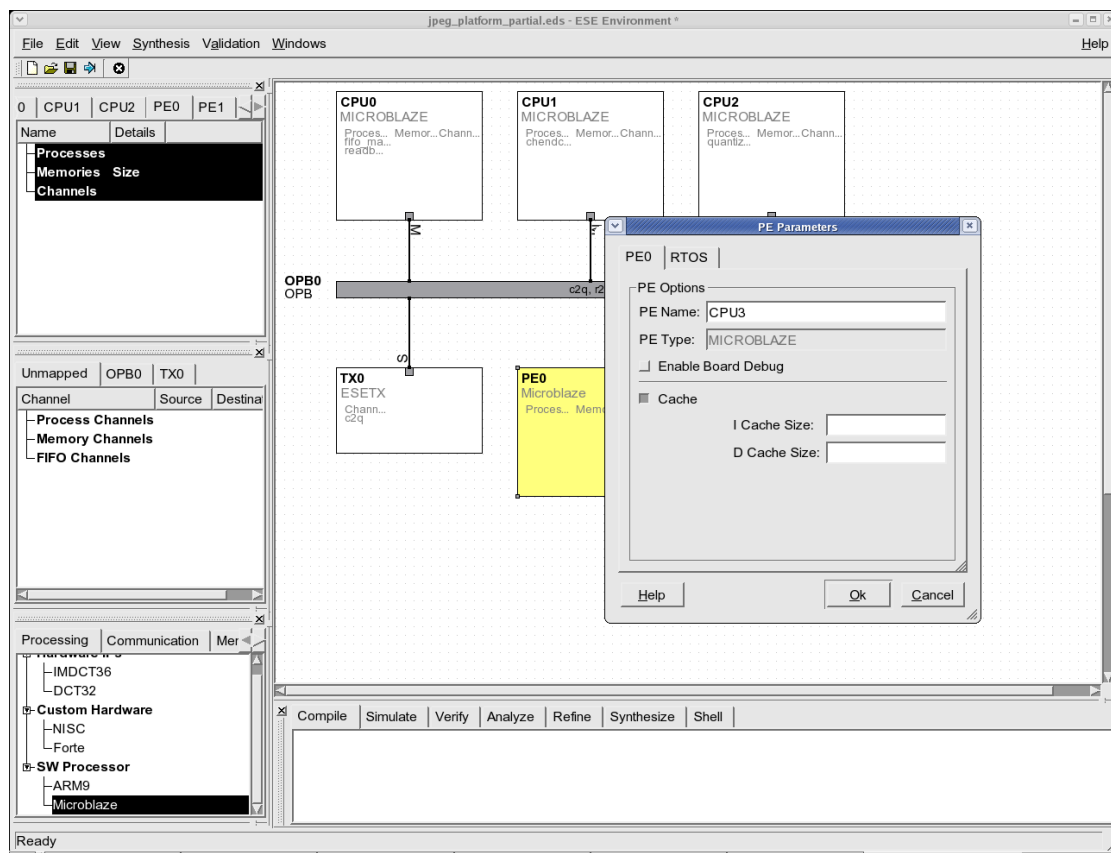
Adding a new PE to the platform is very easy. Browse the database under the Processing tab and select Microblaze. Now drag and drop the selection into the platform canvas. The new PE of type "Microblaze" will be added to the platform!. We need to add two new PEs for the JPEG encoder.

2.2.4. View PE Properties



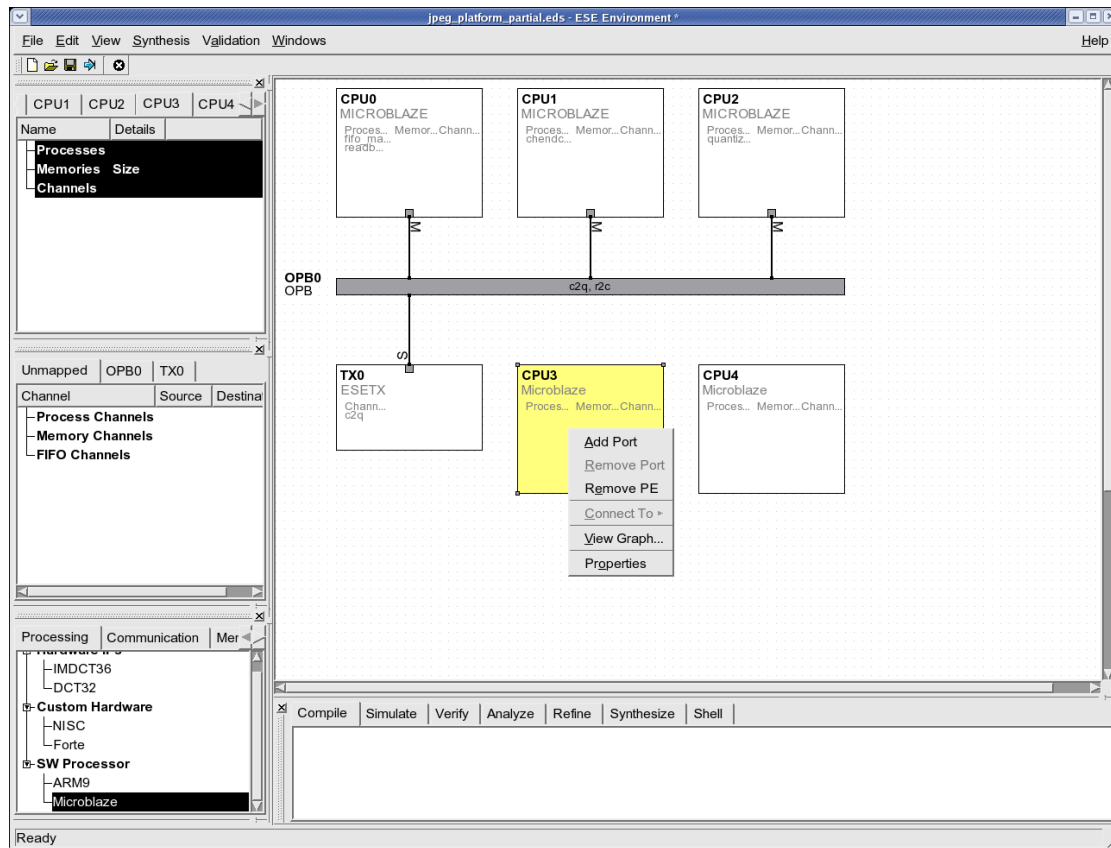
After the drag-drop, the user will find two new PEs called PE0 and PE1 in the platform. These are the PEs that will host the "zigzag" and "huffencode" processes in the design. We start by providing an appropriate names to the new PEs to be consistent with the rest of the design. To do so, right click on the PE0 box and the PE1 box and select Properties.

2.2.5. Assign New Name to PE



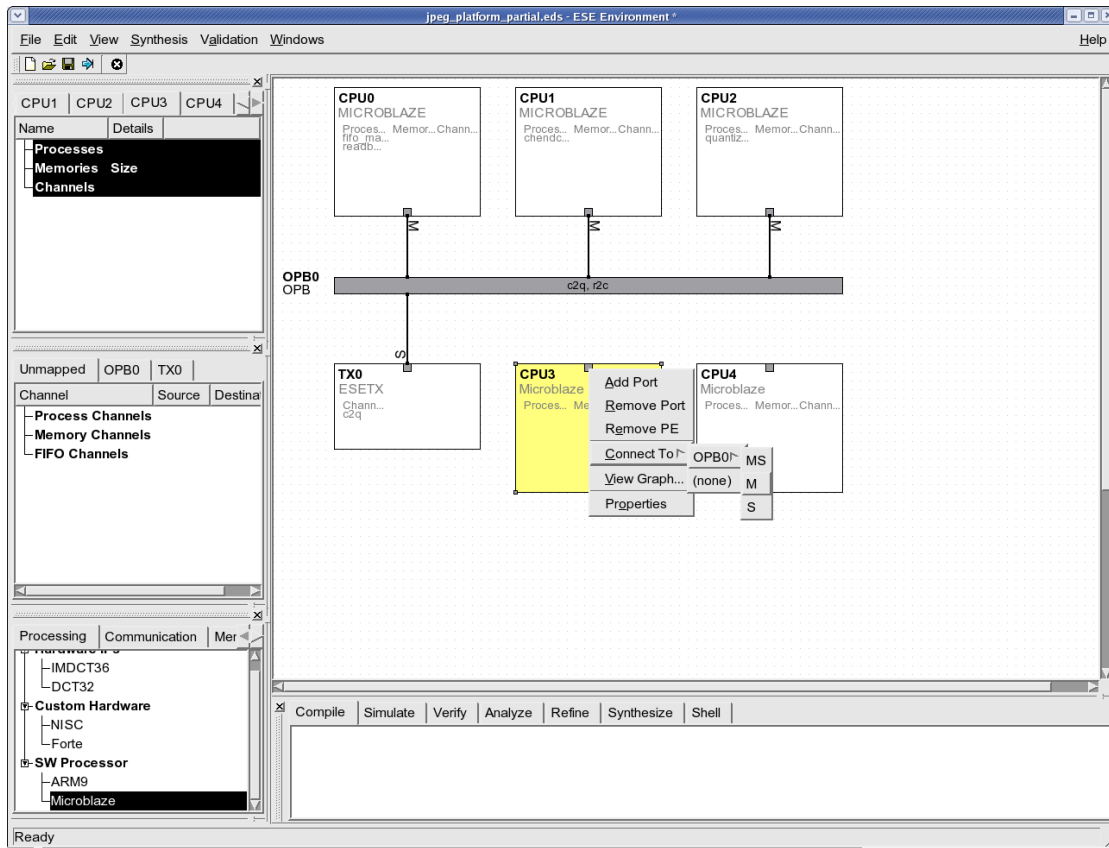
In the properties dialog, change the PE name of the PE0 to "CPU3" and that of the PE1 to "CPU4" to be consistent with the other PE names.

2.2.6. Add Port to PE



The new PEs, CPU3 and CPU4 are not yet connected to the rest of the design. Since the application processes meant to execute on these PEs will need communication with processes on other processors, we must physically connect CPU3 and CPU4 to the shared OPB bus in the platform. For this physical connection, each port is required for CPU3 and CPU4, respectively. To add the port, simply right-click on the CPU3 and CPU4 box and select **Add Port**.

2.2.7. Connect PE to Bus

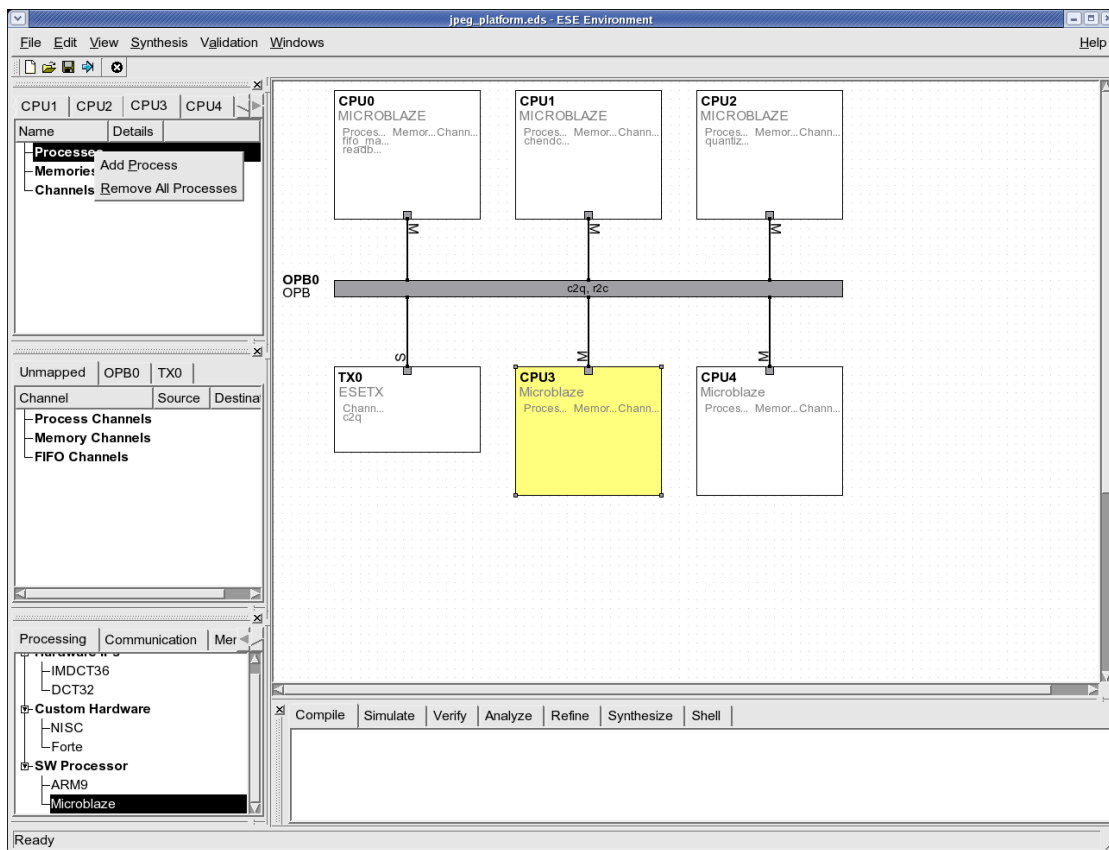


The created port must be connected to the OPB bus to be able to communicate with the rest of the system. Note that CPU3 and CPU4 are Microblaze cores. This means that they can only connect to the OPB bus as a Master. To connect CPU3 and CPU4, right-click on the port and select **Connect To** → **OPB0** → **M** from the menu choice. This will create the bus connection and complete the platform design step. Next, we will look at application input and its mapping to the created platform.

2.3. Mapping Application to Platform

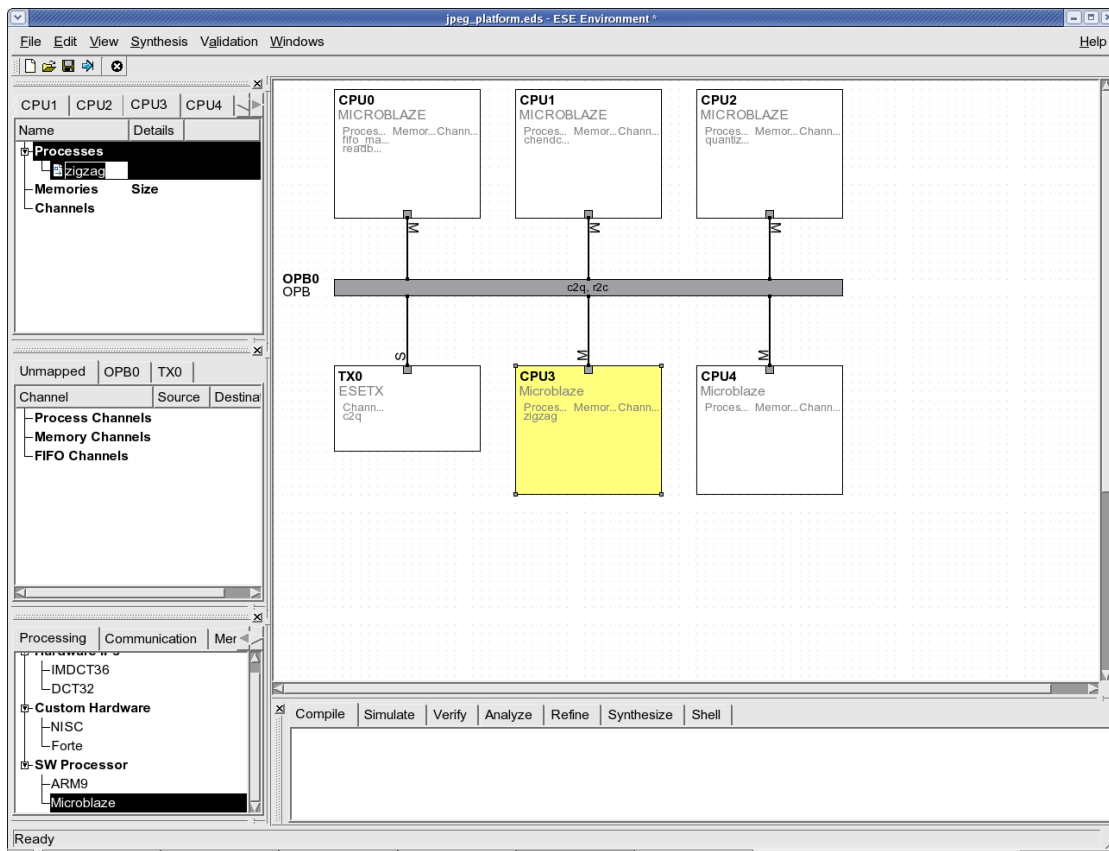
The application input model for ESE is C/C++ processes communicating through FIFO channels. Since most legacy application is written in C, this is an advantage over other forms of input styles or languages. For communication, the user does not need to write any SystemC channel code. ESE provides very simple APIs for inter-process communication as we will see in this section.

2.3.1. Add Application Process



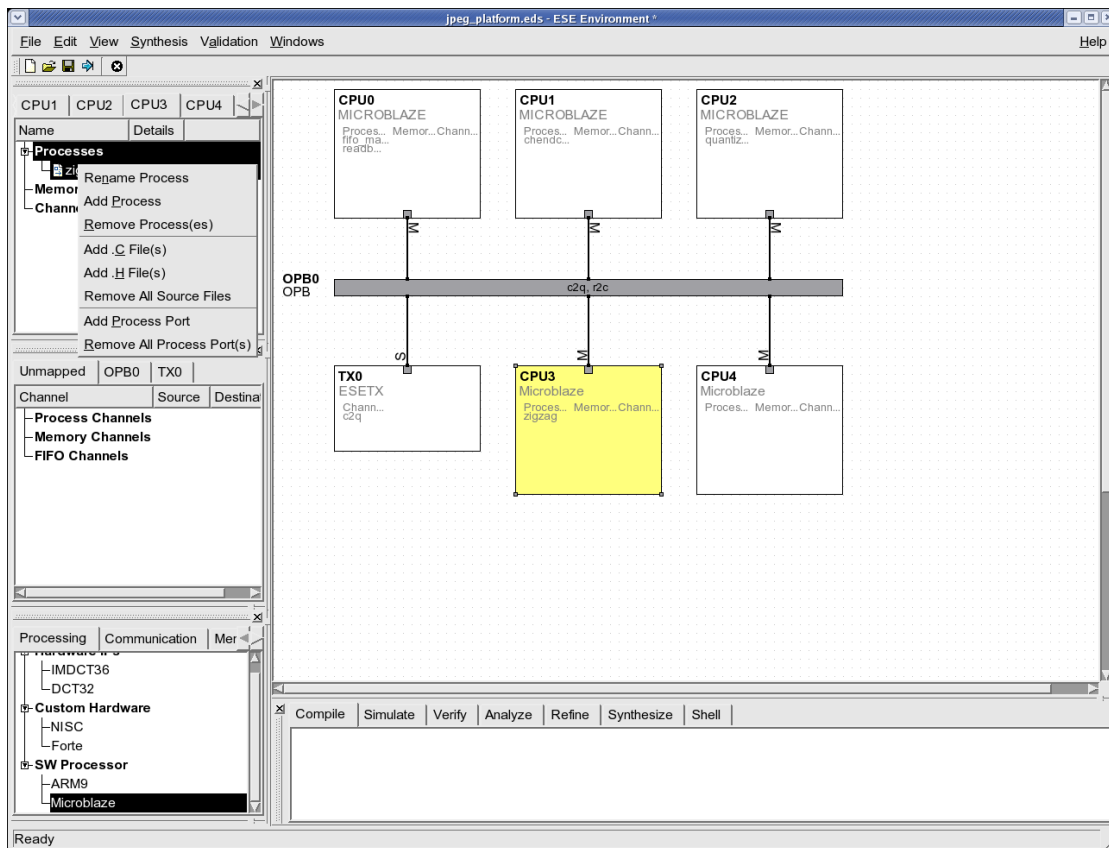
The PE window on the top left corner organizes the processes mapped to the various PEs in the design. In general, several processes may be added for execution on a PE where RTOS should be involved. The platform which has such multi-threaded processors will be demonstrated in Chapter 4. In this section, we assume that there is only one process per PE. To add a new process executing on CPU3, change to the CPU3 tab. Then right-click and select **Add Process**. This will create a new process with a default name. The same goes for CPU4.

2.3.2. Assign Name to New Process



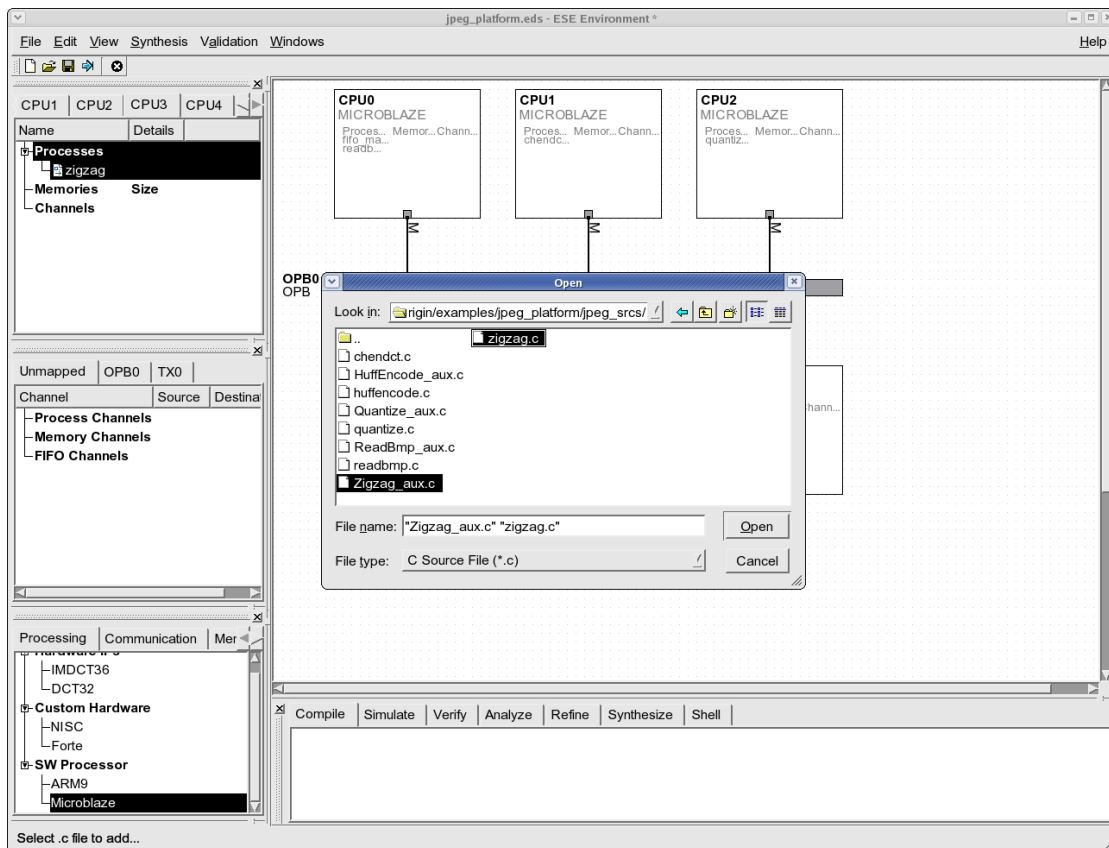
Change the name of the new process to "zigzag". This is the process for the zigzag scan in the JPEG encoder application. Please ensure that the process is named correctly since there exist references to it in the existing partial design. If the process is not named as suggested, the generated models will not compile. The name of the new process for the CPU4 is "huffencode".

2.3.3. Add C Source File



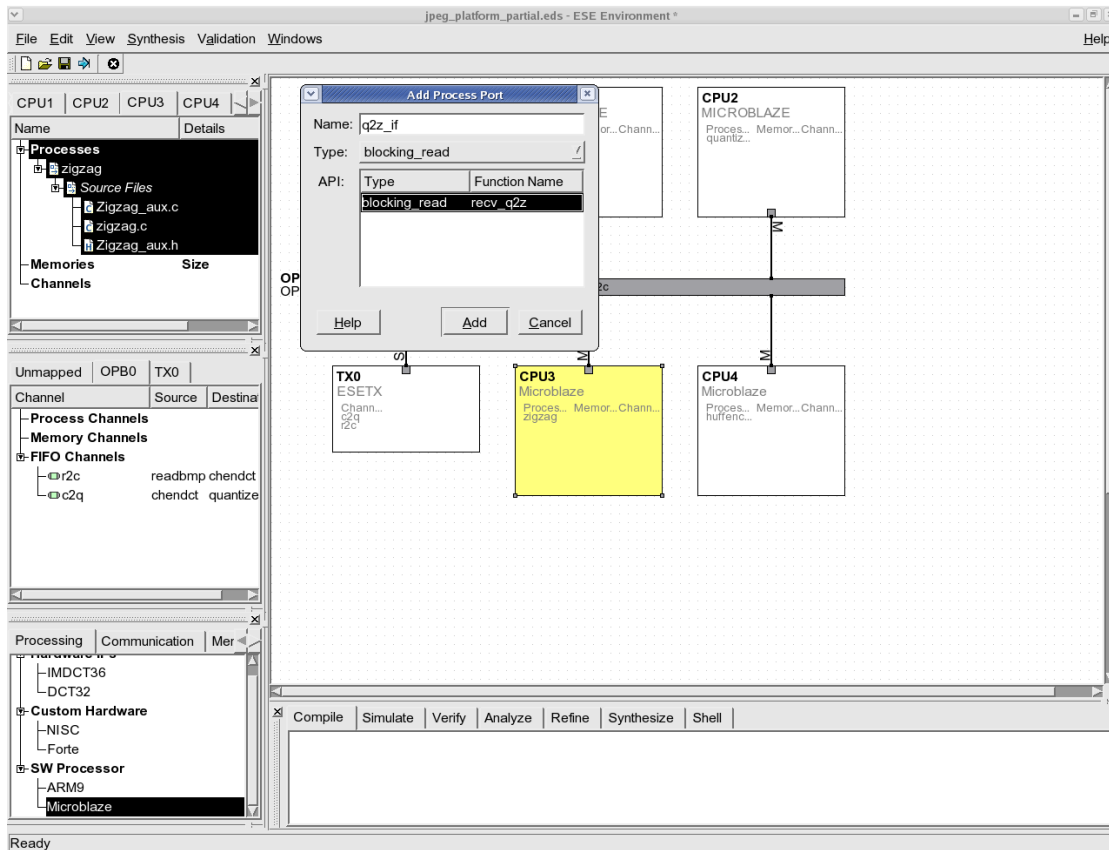
The process added in the last step is only symbolic. The user must associate the actual C/C++ code with it for the models to be functionally correct. In this case, we add C code by right-clicking on the process name in the PE window and selecting **Add .C File** for adding ".c" files. And we can also add ".h" files by selecting **Add .H File**. This will open the file browser.

2.3.4. Select C Source File



Go to the demo directory and follow the symbolic link to "jpeg_srcs". For the "zigzag" process, select two ".c" files, "zigzag.c" and "Zigzag_aux.c", and one ".h" file, "Zigzag_aux.h", and then click Open. In the same way, for the "huffencode" process on CPU4, select three files, "HuffEncode_aux.c", "huffencode.c", and "HuffEncode_aux.h". The files will be added under the new process in the PE window.

2.3.5. Add Process Ports

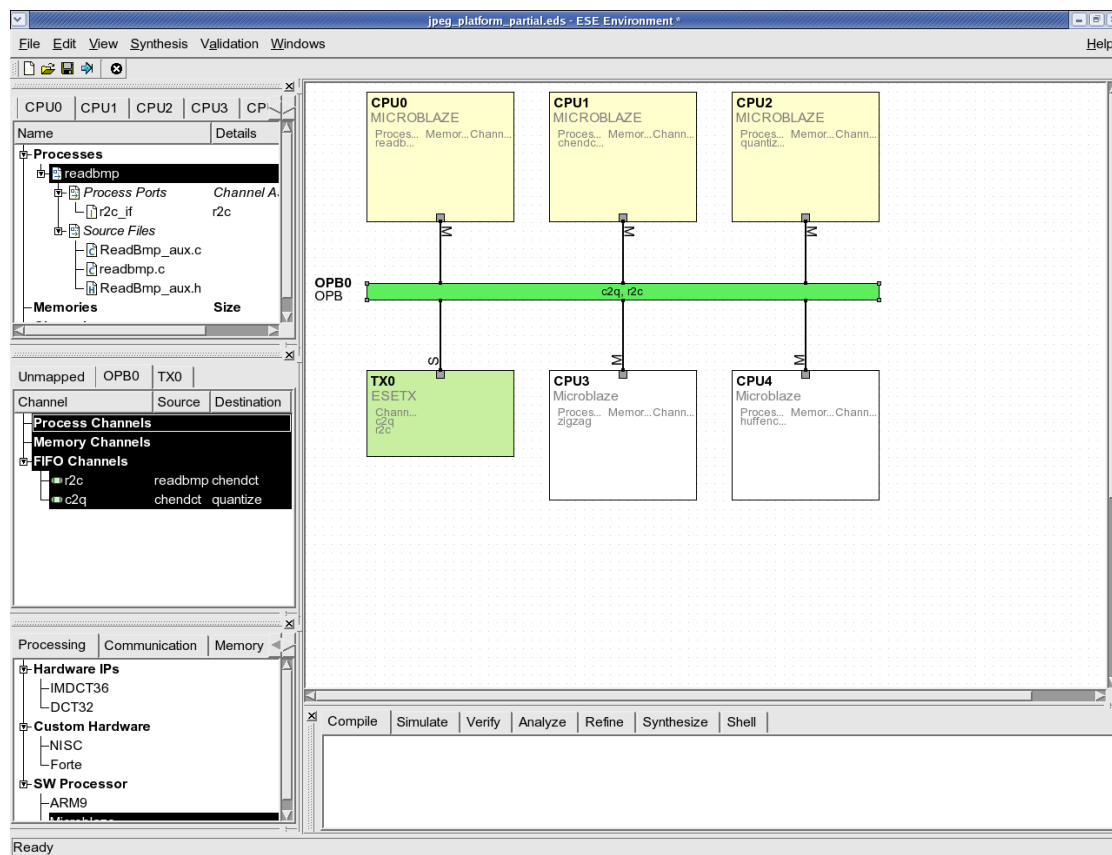


After, the C code for the process is added, we need to add the application level communication to the design. First of all, we need to add the process port for each process, which will be connected to a channel for data transfer to another process. To add the process port for the new process, click on the new process and select the **Add Process Port**. This will open the window to add the process port. We can create any name for the process port and select the type of it. There are ten possible types. We can categorize them into three kinds. The "Send", "Receive", and "Send/Receive" are used for double handshake channels. The "Read", "Write", and "Read/Write" are used for shared memory. And the others are for the FIFO channels. Finally, we need to assign its function name to be what is actually used in C code. Please ensure that the function name is the same as that used in C code. If the name is not correct, the generated models will not compile.

The "zigzag" process has two process ports. One is for receiving data from "quantize" process and the other is for sending data to "huffencode" process. Assign the process port name to be "q2z_if" for the former and "z2h_if" for the latter. Since we are using FIFO

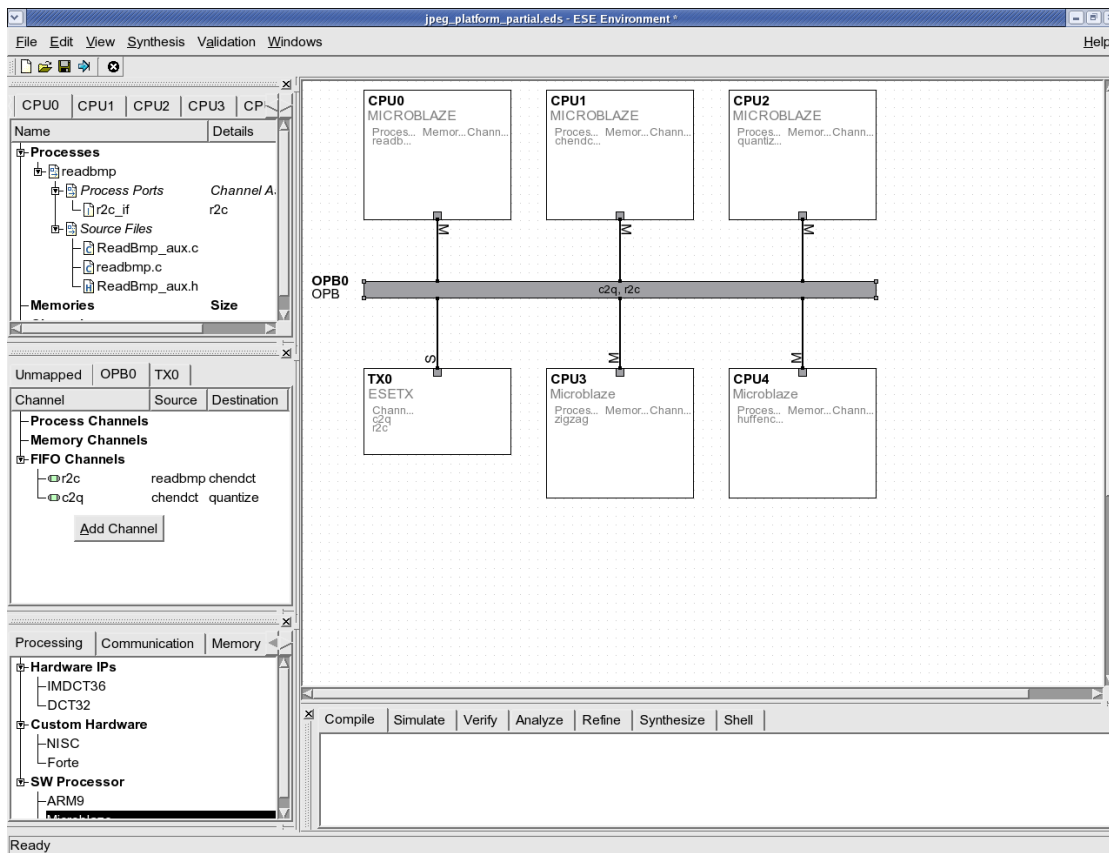
channels, select the type to be "blocking_read" for the former and "blocking_write" for the latter, respectively. Also, assign the function name to be "recv_q2z" and "send_z2h", respectively. The "huffencode" process on CPU4 has only one process port which is for receiveing data from "zigzag". Its process port name is "z2h_if" and its function name is "recv_z2h". Please add all the process ports for all the new processes using the given names.

2.3.6. View Application Channels



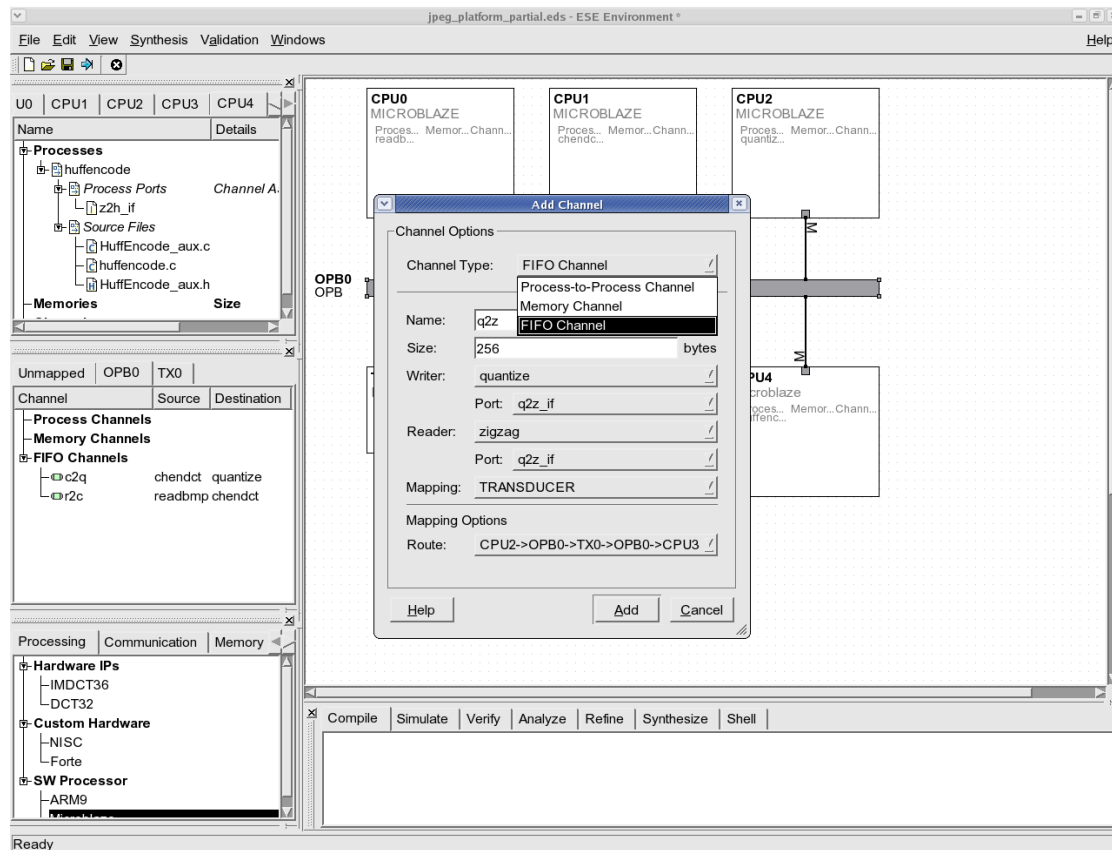
After, the process port for the process is added, we need to create the channels for the communication between processes. To view the existing channels, click on the Tx0 tab. This will display the existing FIFO channels between processes in CPU0, CPU1 and CPU2, including the source and destination names as well as the route used to implement the channel in the communication platform. All the channels in ESE can be uni-directional or bi-directional channels. If the user clicks on a PE in the platform canvas, all the channels originating or terminating at the PE will be selected. All other PEs that the clicked PE communicates with will be highlighted in light yellow. All physical connections, including buses and transducers used by the PE for communication will be highlighted in green.

2.3.7. Add New Application Channel



Please click on CPU3 and CPU4 and see the Channel window. We can know that they are currently not connected at the application level to any other PE. Since we need communication between the "quantize" process in CPU2 and the "zigzag" process in CPU3, we will add the application level channels, by right-clicking in the channel window and selecting Add Channel. This will pop up the channel wizard for adding application level channels.

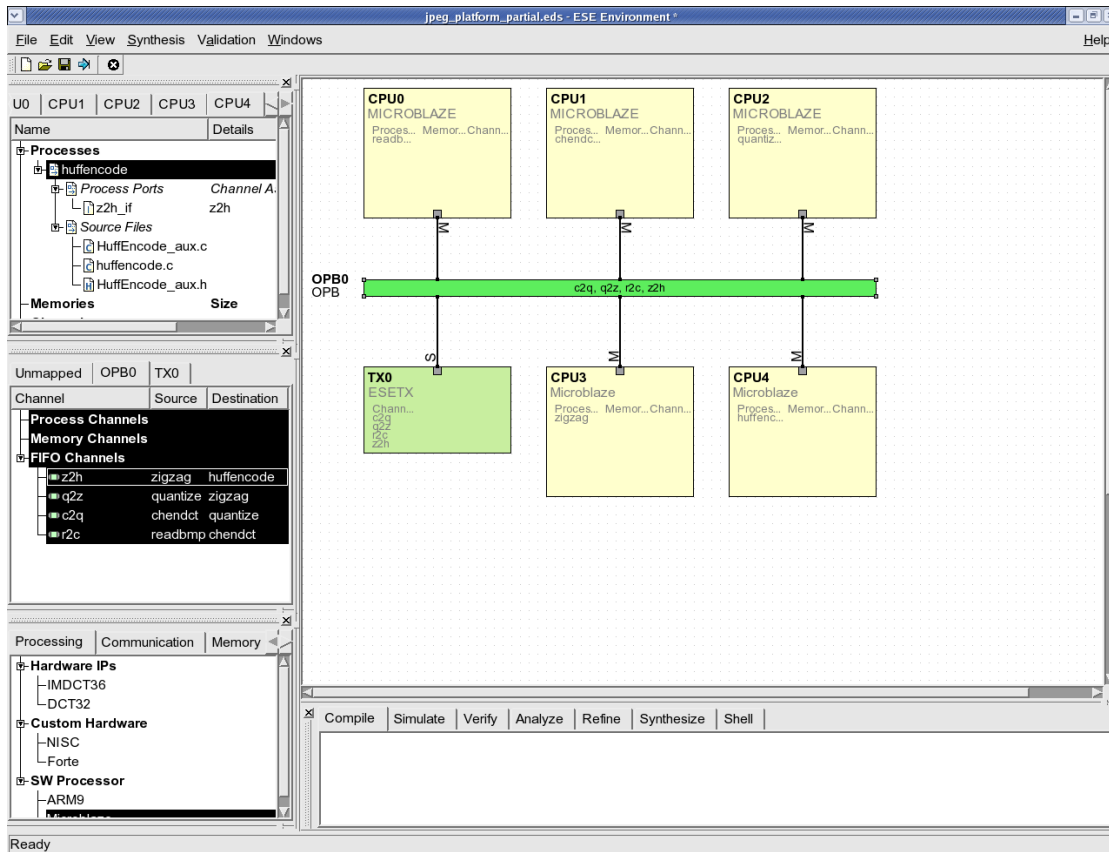
2.3.8. Channel Wizard



In the channel wizard dialog, we first need to select the channel type. Choose "FIFO Channel" since we are using FIFOs. Then, assign the channel name to be "q2z" for consistency with existing channels and also assign the FIFO size to be "256" bytes since the processes send/receive an 64-array integer data each other. Next, since the process will send data in one way from "quantize" to "zigzag", select "Unidirectional" using the pull down menu. Then, use the pull down menu to select the first communicating process as "quantize" and also use the next pull down menu to select the process port as "q2z_if". In the same way, select the other communicating process as "zigzag" and select the process port as "q2z_if". Next, select the mapping to be "TRANSDUCER" since we are using a transducer for the inter-process communication. Once the communicating processes and process ports are decided, ESE automatically filters all the possible physical routes on the platform that can implement the channels. For this example, it shows that there is only one route for each direction that goes over the OPB bus from the sender PE to the transducer Tx0 and back to the receiver PE on the OPB bus. The route goes through the transducer because all PEs in the platform are connected as masters, which does not allow direct communication. The slave interface of Tx0, thus makes the routing possible.

Click **Add** to add the channel. The same goes for the channel for the communication from "zigzag" to "huffencode". Please create the channel for its name to be "z2h" in the same way.

2.3.9. View New Channel Communication



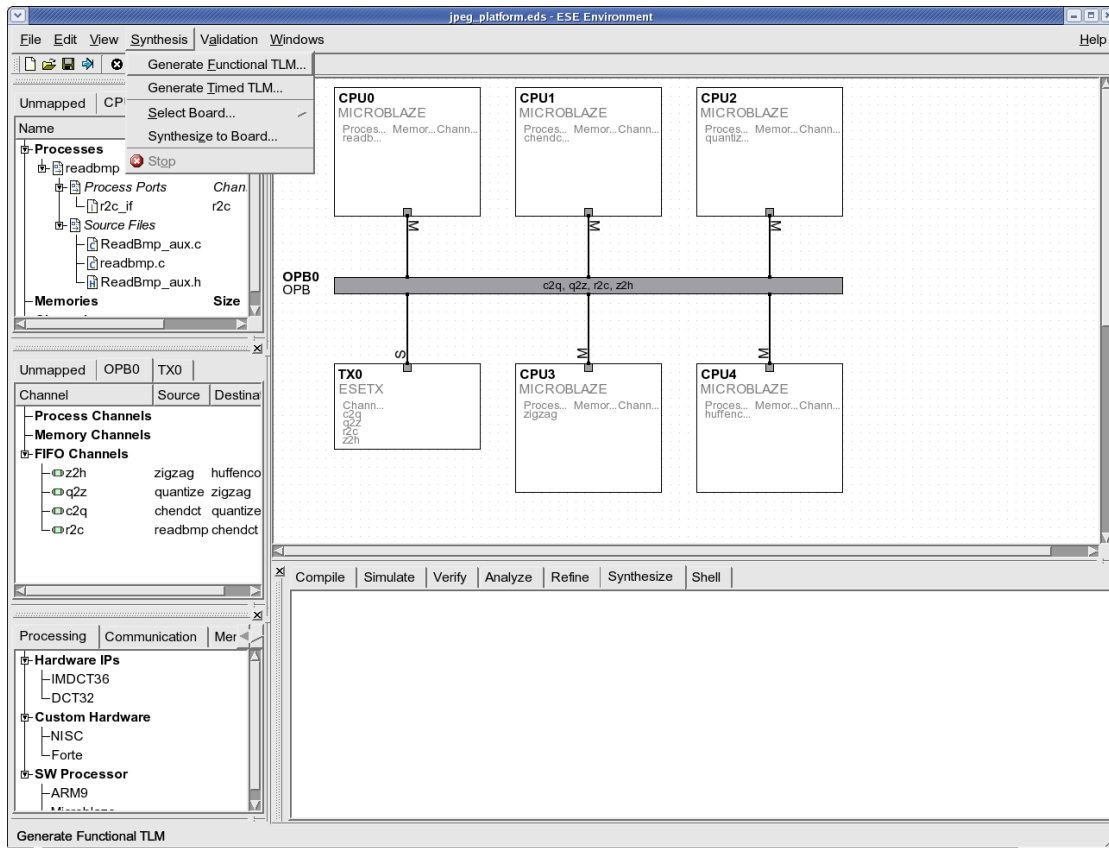
The newly created channels will now be visible in the channel window under the Tx0 tab. Once the channels are selected, the communicating PEs will be highlighted. This shows that the new PEs, CPU3 and CPU4 are now "connected" with the rest of the system on an application level.

2.4. Generating Functional and Timed TLMs

The previous steps complete the platform and application input that is necessary for generation of TLMs. We will show generation of two types of SystemC TLMs. The first one is called the "Functional TLM" because it is used for the validation of design functionality only. It is completely un-timed and simulates the design based on causal dependency only. A universal bus channel is used to model the system bus and the mapping of channels on the bus.

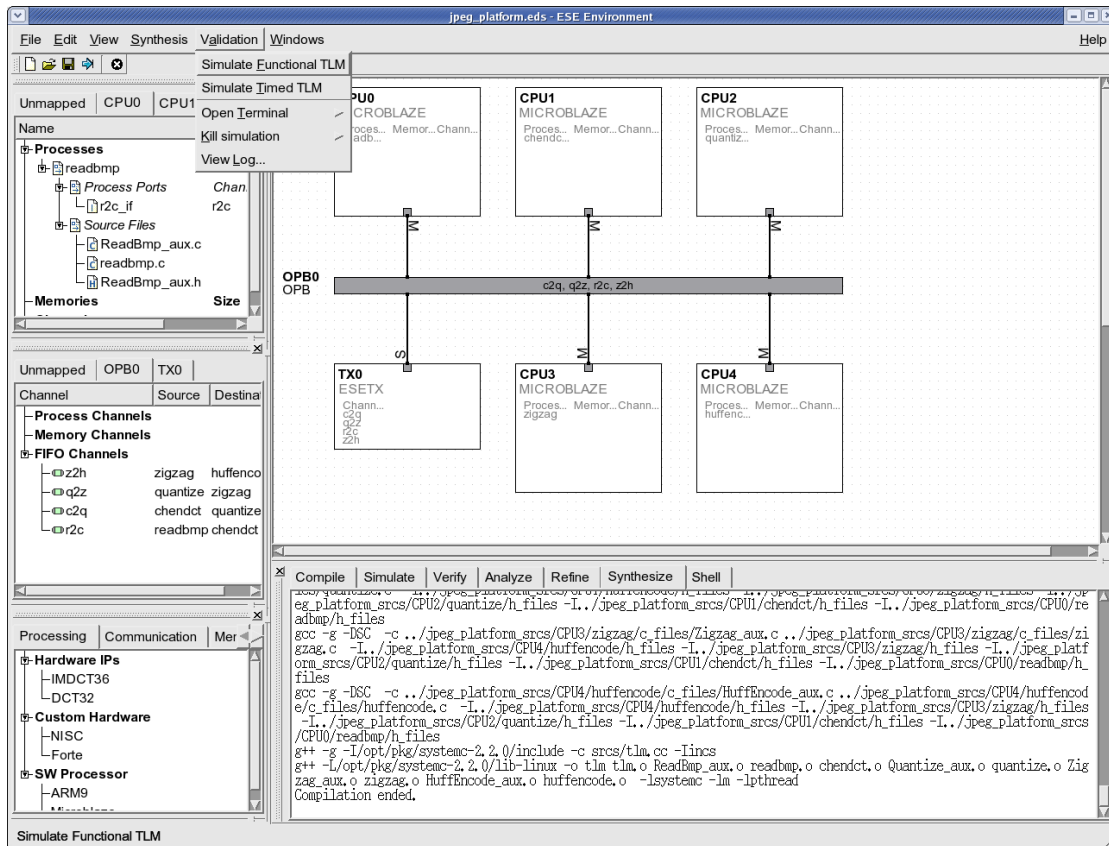
The second TLM is called the "Timed TLM" and is used for performance estimation of the design. It relies on timing data models of PEs and Buses that are available in the ESE database. The data models are used by our estimation and annotation technique to apply "wait" statements in the application C code. The technique is retargetable and applicable to processors as well as HW IPs. A retargetable bus timing annotation modifies the bus channel to apply "wait" statements for inter-PE communication.

2.4.1. Generate Functional TLM



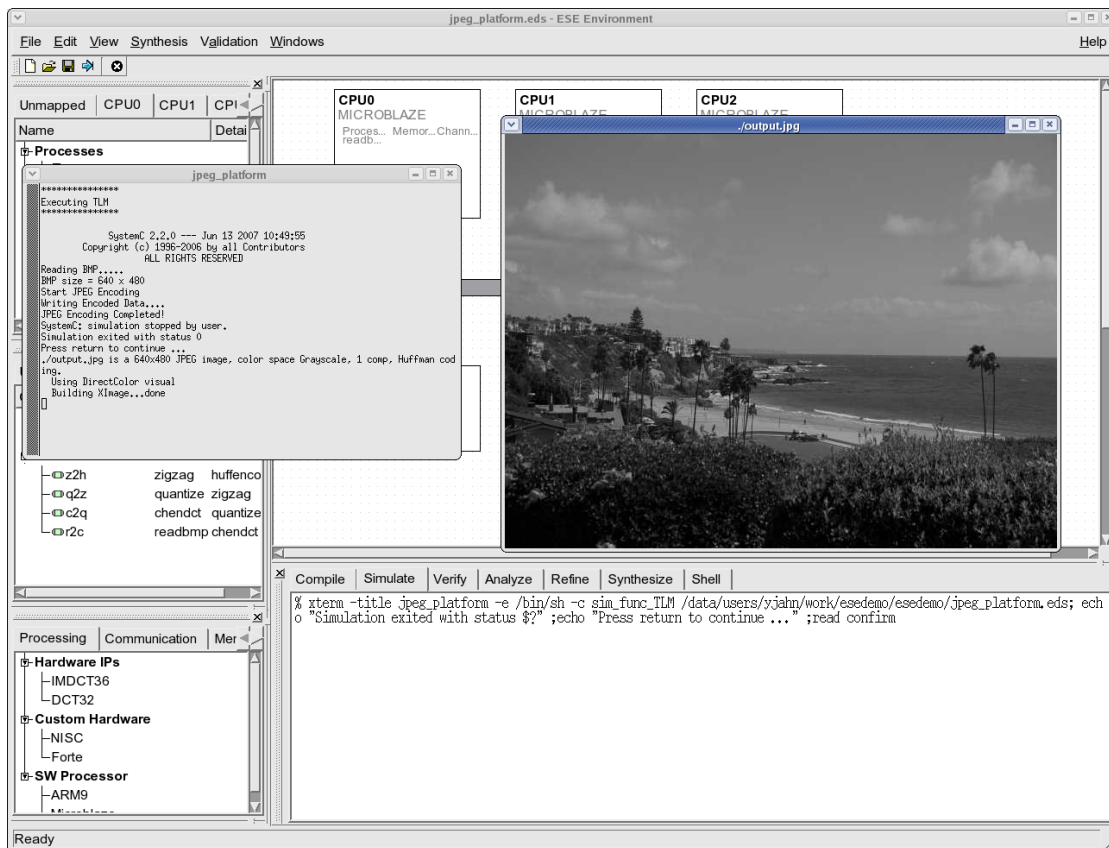
After the platform and application input is complete, the functional TLM can be generated automatically by selecting **Synthesis**→**Generate Functional TLM** from the menu bar. This will generate the SystemC code needed for platform modeling, including PEs, buses and transducers. The generated code is then compiled natively along with the C application code and linked to the SystemC libraries to produce a single binary. This process can be viewed in the log window.

2.4.2. Simulate Functional TLM



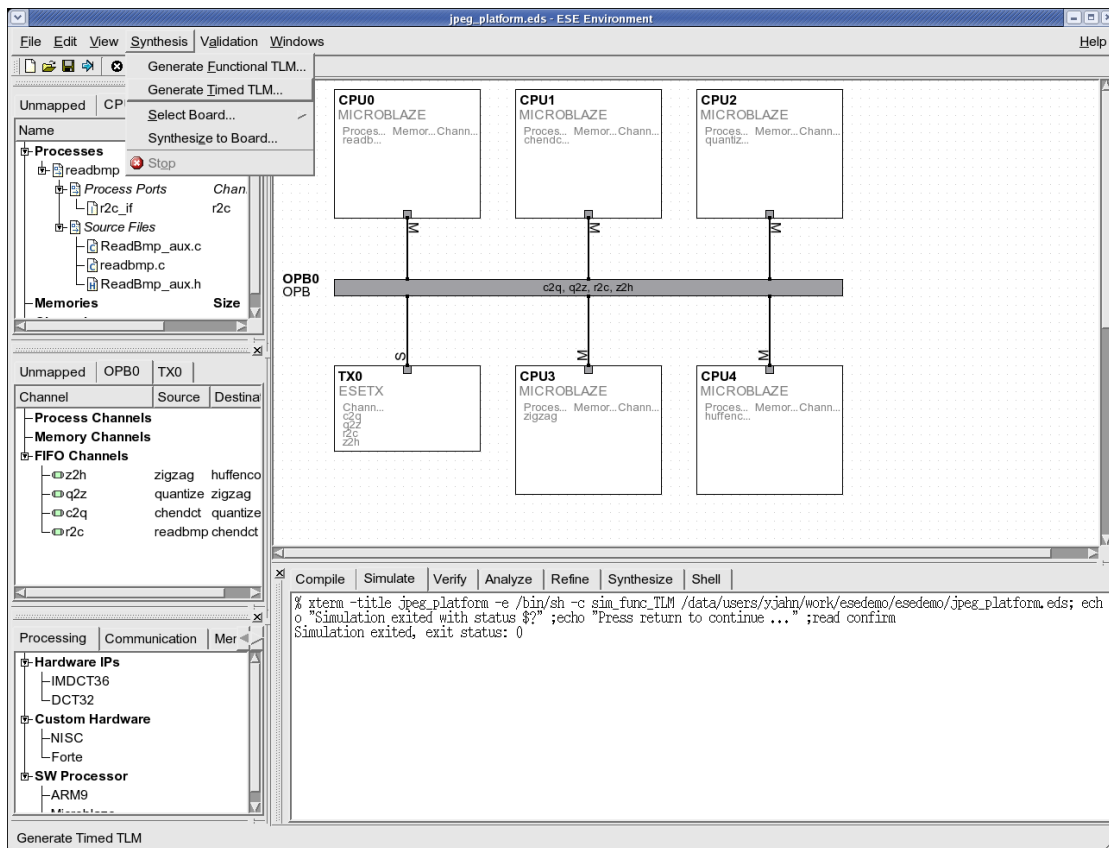
Once the compilation has completed, the generated TLM can be executed from the GUI by selecting Validation—>Simulate Functional TLM from the menu bar.

2.4.3. View Functional Simulation Results



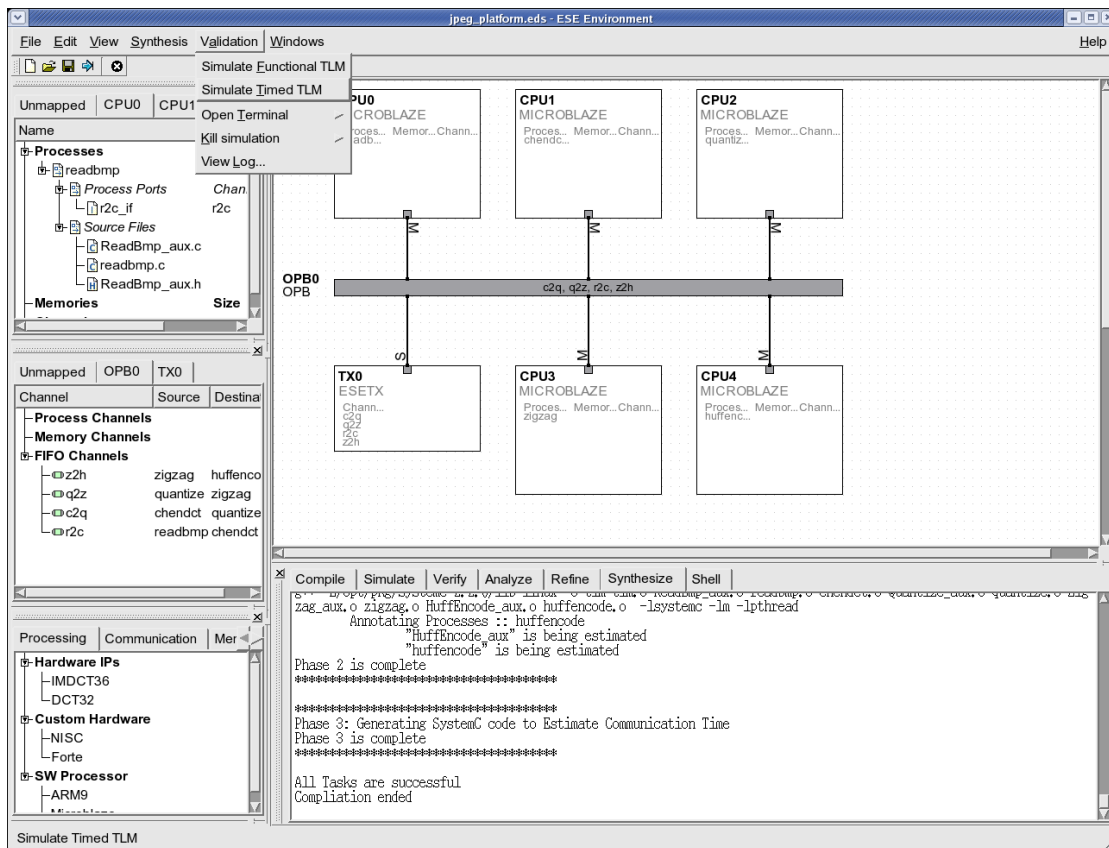
The simulation pops up a terminal that shows the picture size of BMP input that has been encoded. The JPEG encoder we are using deals with BMP inputs of 640x480 size. An additional window shows the picture of the encoded JPEG which is the output of the simulation. The pop up windows can now be killed simply by pressing "Enter" in the simulation logging terminal.

2.4.4. Generate Timed TLM



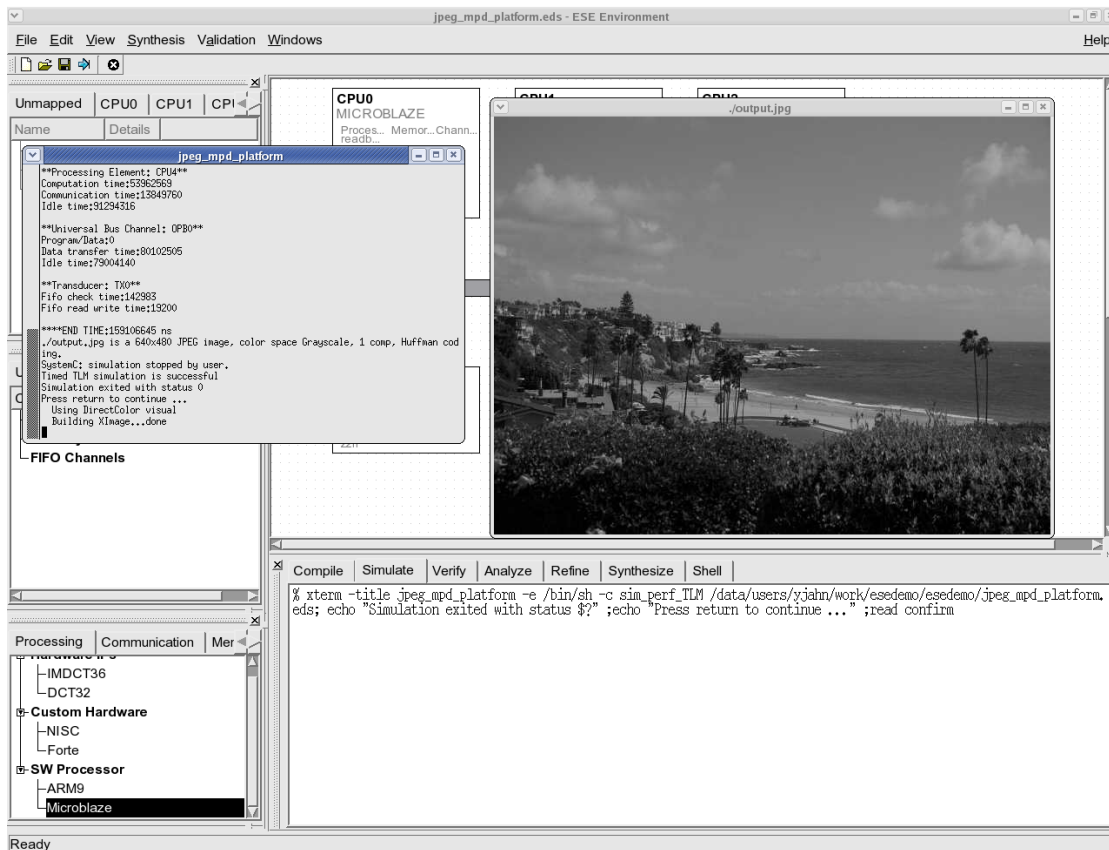
Similar to the functional TLM generation, the Timed TLM can be generated automatically by selecting **Synthesis** → **Generate Timed TLM** from the menu bar. The bus channels generated for timed TLM will include timing for synchronization, arbitration and data transfer. The timing parameters are imported into the TLM from the bus data model. For the computation part, we use a retargetable source level timing estimation technique that utilizes the PE data models. Naturally, the timed TLM generation and compilation is significantly slower than functional TLM generation, but still in the order of seconds.

2.4.5. Simulate Timed TLM



To simulate the generated timed TLM, simply select Validation→Simulate Timed TLM from the menu bar, after the TLM compilation has ended.

2.4.6. View Timed Simulation



The timed TLM simulation looks very similar to the functional TLM simulation except for one marked difference. Notice that timed simulation is significantly slower than functional TLM simulation. This is natural since we are simulation a lot more "wait" statements that are annotated to the application codes. However, our results show that this is still several orders of magnitude faster than RTL simulation for the same design.

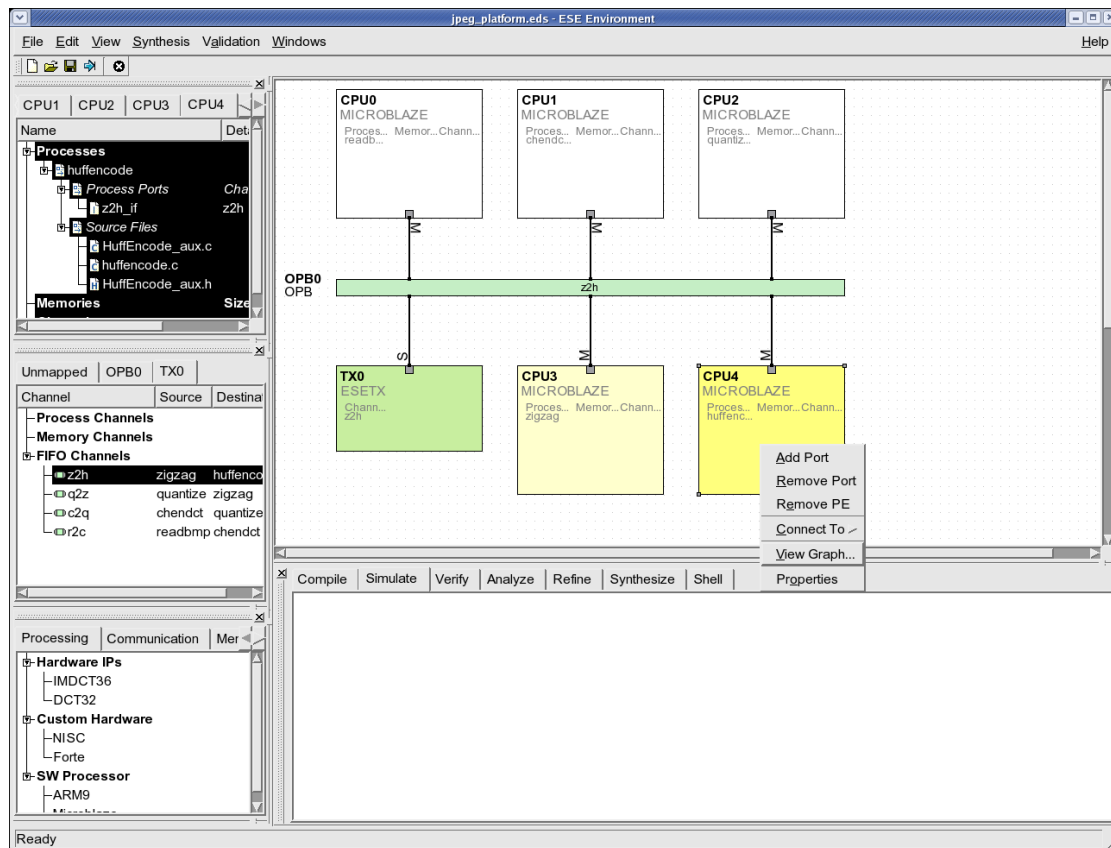
When the timed simulation ends, several statistical data are dumped in the simulation logging terminal. These are the estimated cycles for CPU computation and communication, bus congestion estimates and so on. However, all these estimated performance statistics can be viewed graphically as shown in next section.

2.5. TLM Performance Estimation

The timed TLM produces several statistical data that is gathered during simulation. Since the source annotation is fine grained, the TLM produces results for cycles used for invocation of each function in the application code. Computation and communication cycles for each PE can be viewed using pie charts. The distribution of cycles for each function amongst its sub-functions can be browsed recursively. Similarly, the distribution of inter-PE bus traffic over inter-process channels can also be viewed graphically.

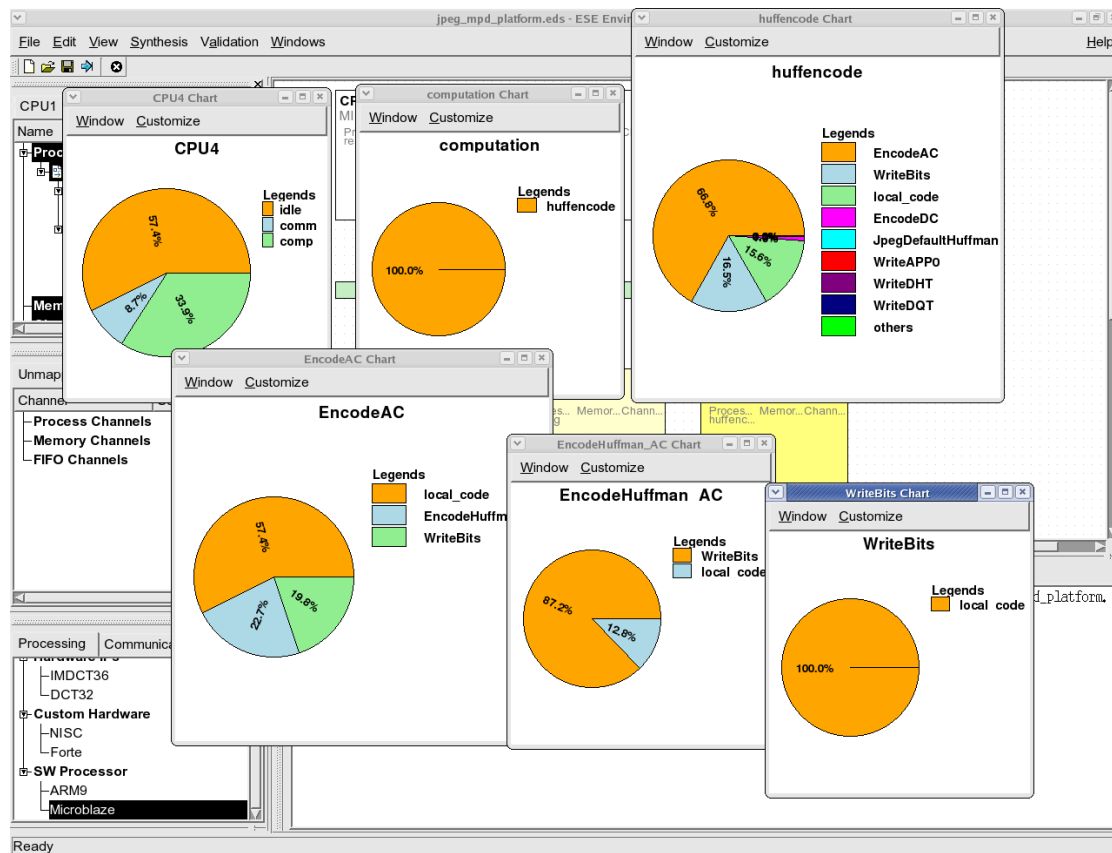
The performance estimates are useful for early platform and mapping evaluation. Since the timed TLMs are generated automatically, and TLM simulation is very fast, early design space exploration becomes feasible. Users may explore platforms manually or plug in their exploration algorithms for system level design optimization.

2.5.1. View Performance Estimates



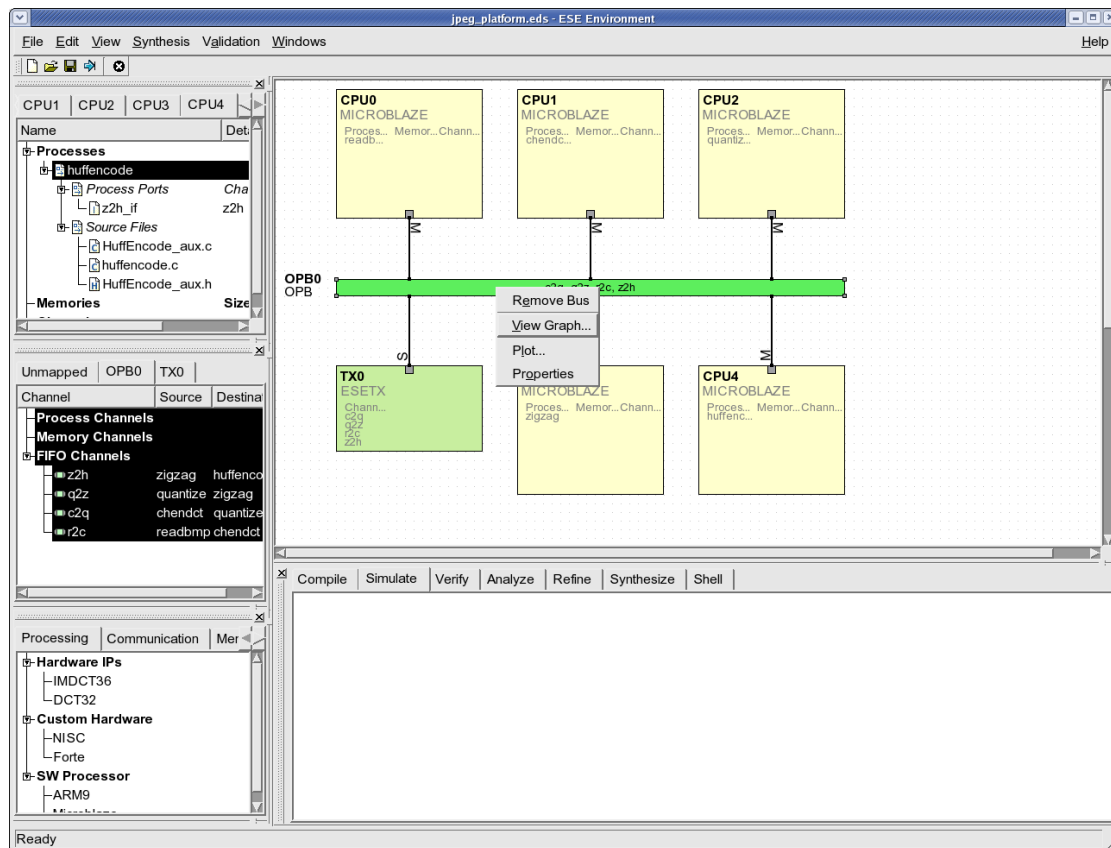
To view the PE performance statistics, right-click on the PE in the platform canvas and select View Graph. In this case, we will select the CPU4 Microblaze processor.

2.5.2. PE, Process and Function Level Estimates



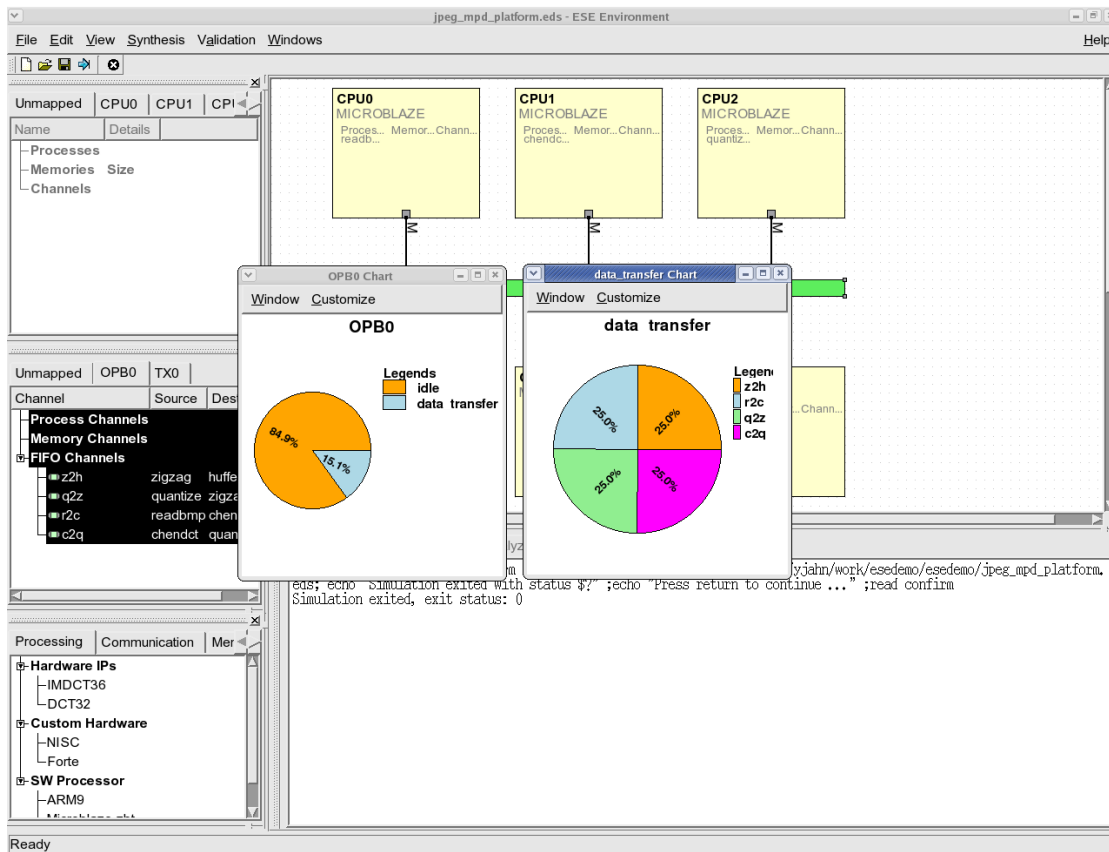
The first pie-chart window divides the total execution time into computation, communication and idle cycles. Double-clicking on the computation part of the pie chart pops up the distribution of computation across different processes in the design. In this case, we have only 1 process "huffencode" mapped to CPU4, hence it is 100%. Double-clicking on the process in the pie chart produces the distribution of computation across the top level functions in the process. These function(s) call lower level functions and so on. Double-clicking on a function produces the pie chart for the distribution of cycles amongst the sub-function invocations. Using this viewing feature, the user may go down to any level in the function call hierarchy. If the pie chart appears too small, please increase the window size to enlarge the chart.

2.5.3. View Communication Estimates



To view the bus communication statistics, right-click on the bus in the platform canvas and select View Graph. In this case, we will select the only OPB bus.

2.5.4. Bus and Channel Level Estimates



The top level pie-chart for the bus shows the distribution of bus cycles in idle, program/data access and inter-PE data transfer phases. Double-clicking on the "data-transfer" part of the pie-chart produces the distribution of communication traffic amongst the various application channels in the design.

Chapter 3. Heterogeneous System Design with ESE

This section deals with design of MP3 decoder on a heterogeneous platform consisting of one MicroBlaze processor and four HW accelerators. There are two buses in the platform. The Microblaze processor uses its compatible OPB. The hardware accelerators on the other hand, were manually designed and use their own proprietary Double Handshake (DH) bus. Since the two bus protocols are incompatible, a transducer that acts as a buffer and protocol converter, interfaces between the buses.

The MP3 application code is available as a C model. This C model was divided into five processes, by separating the left/right channel DCT32 and IMDCT36 transforms into separate processes. These new processes run concurrently to the main MP3 thread since they are data independent. In this Chapter, the communication between the processes takes place through pairs of message passing channels. ESE provides well defined communication APIs for this purpose. The testbench includes an MP3 file that is decoded into a corresponding PCM file by the design. The decoded output is shown graphically during the TLM simulation of the MP3 decoder.

The chapter starts by explaining the set up for ESE. It then shows, using screenshots, how the platform is created. To speed up the demonstration, and to emphasize on the features, we start with an existing partial platform that is upgraded with an additional processor. Then we show the application mapping on the platform, followed by TLM generation, simulation and performance estimation. Thus, we present the core capabilities of the ESE Front-End tools in easy platform design & upgrade, model generation, validation and estimation.

3.1. ESE Startup and Settings

Before starting the demonstration, please ensure that you have the ESE software installed in the right location at `"/home/ease/local."` Also make sure that you have an `"/home/ease/local/pkg"` directory containing the SystemC 2.2.0 libraries and SDL libraries that are needed for simulation of generated TLMs. Also make sure that you have GCC version 3.4 or higher because it is needed to correctly compile the generated TLMs. The demonstration shown here assumes the user to have a bourne shell. For C shell, the user may call the `".csh"` version of the setup scripts. Alternately, just use `"sh"` to create a new bourne shell and follow the tutorial directions.

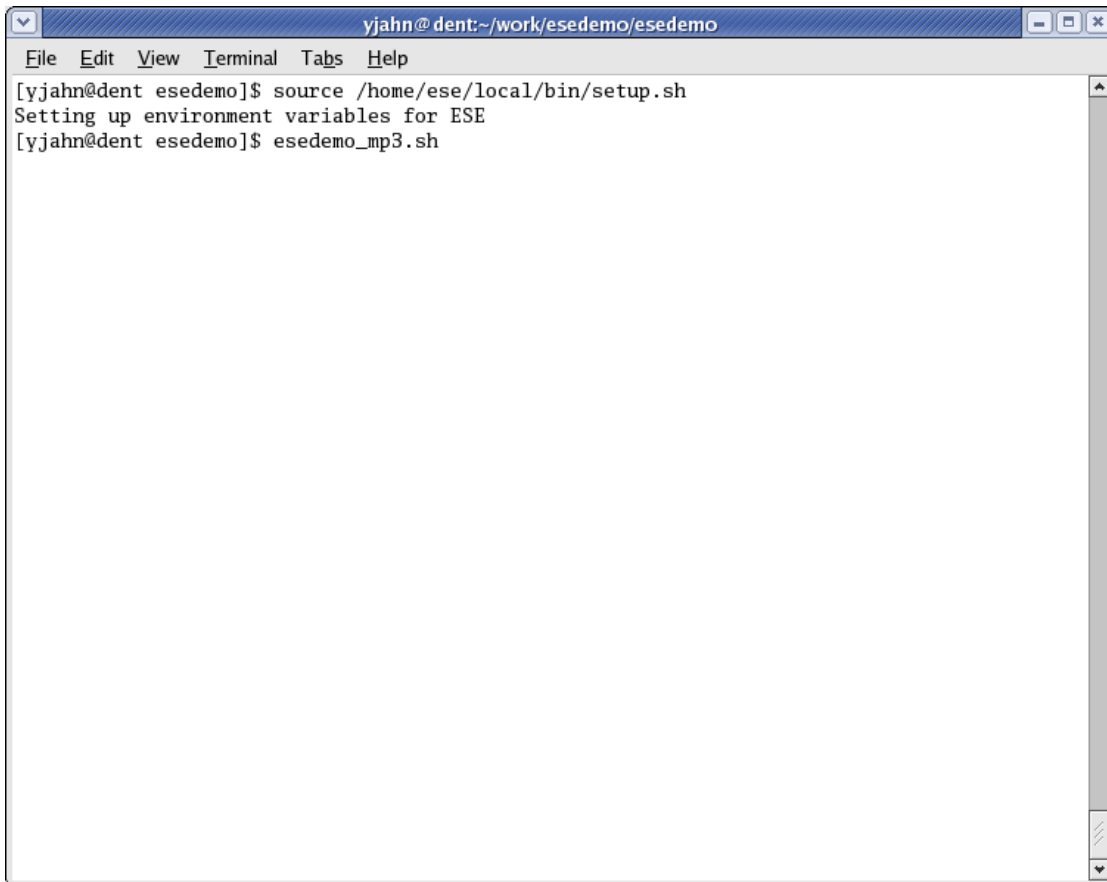
3.1.1. Environment Setup

A terminal window titled "yjahn@dent:~/work/esedemo/esedemo" with a menu bar (File, Edit, View, Terminal, Tabs, Help). The command prompt shows "[yjahn@dent esedemo]\$ source /home/e/e/local/bin/setup.sh". The terminal area is mostly empty, indicating the script has been executed without visible output or error messages.

```
yjahn@dent:~/work/esedemo/esedemo
File Edit View Terminal Tabs Help
[yjahn@dent esedemo]$ source /home/e/e/local/bin/setup.sh
```

We start by setting up the environment variables to access ESE binaries. This is provided by the "setup.sh" script in your installation. Typically, the installation path would be "/home/e/e/local." The script is in the "bin" directory in the installation. The script modifies your PATH environmental variable to include path to ESE as well as the LD_LIBRARY_PATH variable to access the shared libraries that ESE depends on. Run the command "source /home/e/e/local/bin/setup.sh" and create a new local directory for the demo.

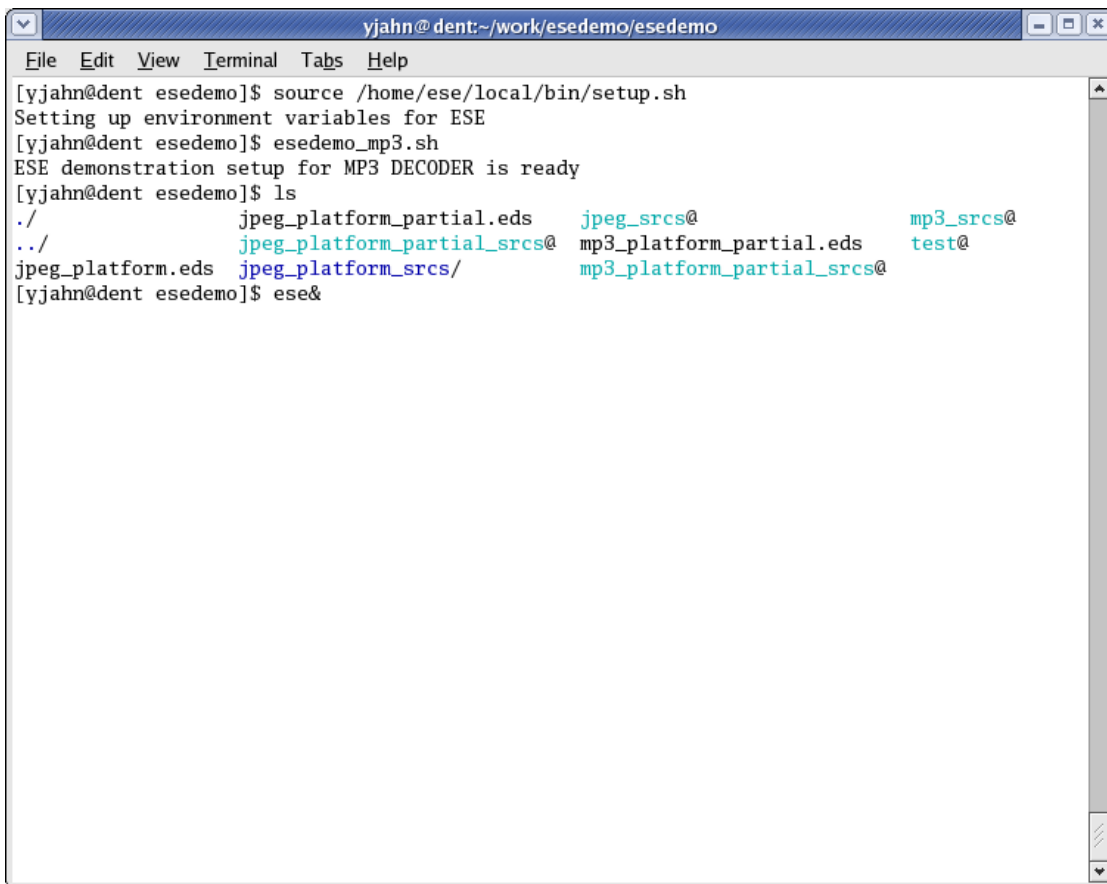
3.1.2. ESE Demonstration Setup



```
yjahn@dent:~/work/esedemo/esedemo
File Edit View Terminal Tabs Help
[yjahn@dent esedemo]$ source /home/e/e/local/bin/setup.sh
Setting up environment variables for ESE
[yjahn@dent esedemo]$ esedemo_mp3.sh
```

Once the environmental variables have been set, the user is ready to launch ESE and create his or her design. For the purposes of this tutorial, we will start with a partial design to quickly demonstrate the key capabilities of the toolset. We have created a shell script called "esedemo_hsd.sh" that prepares a partial design to start the demo for the MP3 decoder. At this point, run the "esedemo_hsd.sh" script after changing into the local directory created for the demo.

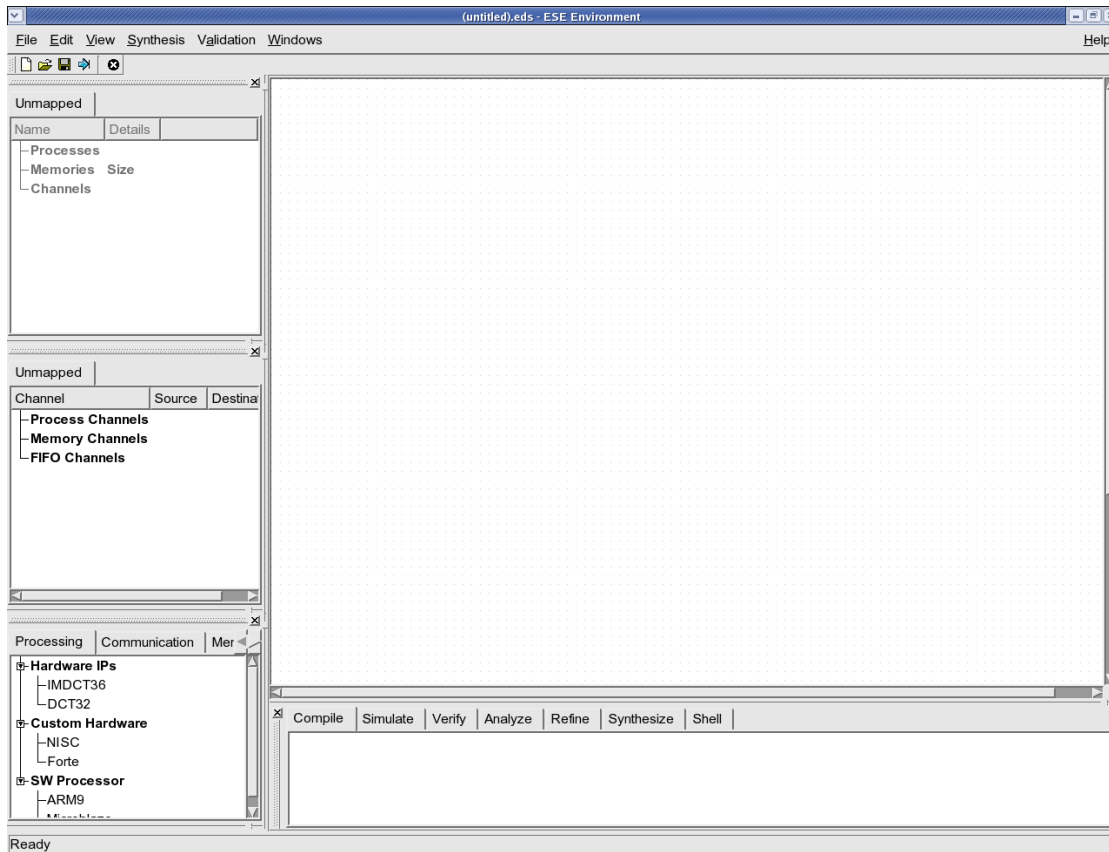
3.1.3. Launching ESE



```
yjahn@dent:~/work/esedemo/esedemo
File Edit View Terminal Tabs Help
[yjahn@dent esedemo]$ source /home/e/e/local/bin/setup.sh
Setting up environment variables for ESE
[yjahn@dent esedemo]$ esedemo_mp3.sh
ESE demonstration setup for MP3 DECODER is ready
[yjahn@dent esedemo]$ ls
./                jpeg_platform_partial.eds  jpeg_srcs@          mp3_srcs@
../              jpeg_platform_partial_srcs@ mp3_platform_partial.eds test@
jpeg_platform.eds jpeg_platform_srcs/        mp3_platform_partial_srcs@
[yjahn@dent esedemo]$ ese&
```

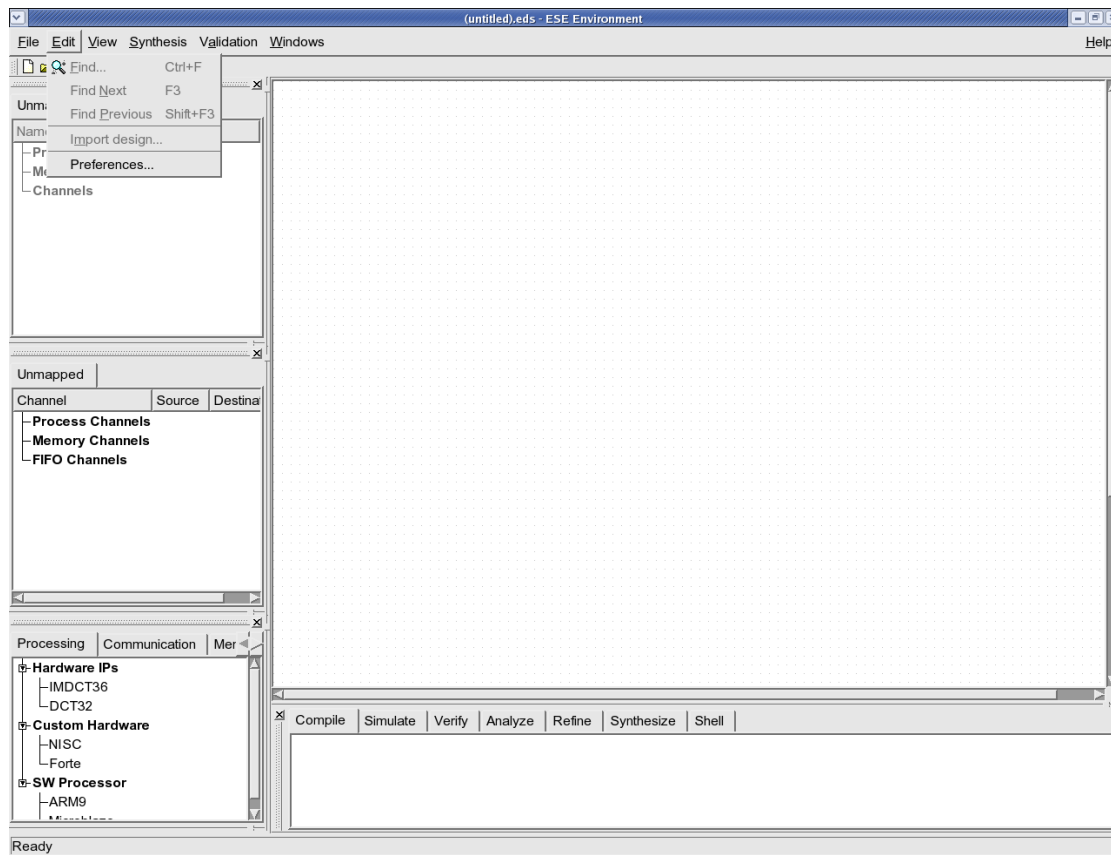
After running the "esedemo_hsd.sh" script, you will notice several files in the working directory. Some of these files will have a ".eds" extension. They are the ESE design files for the MP3 decoder designs that we will be using for this demo. You may also see links to source directories. These point to the C code for the processes of the MP3 application. To launch the ESE GUI, simply run "ese" from your shell.

3.1.4. ESE GUI



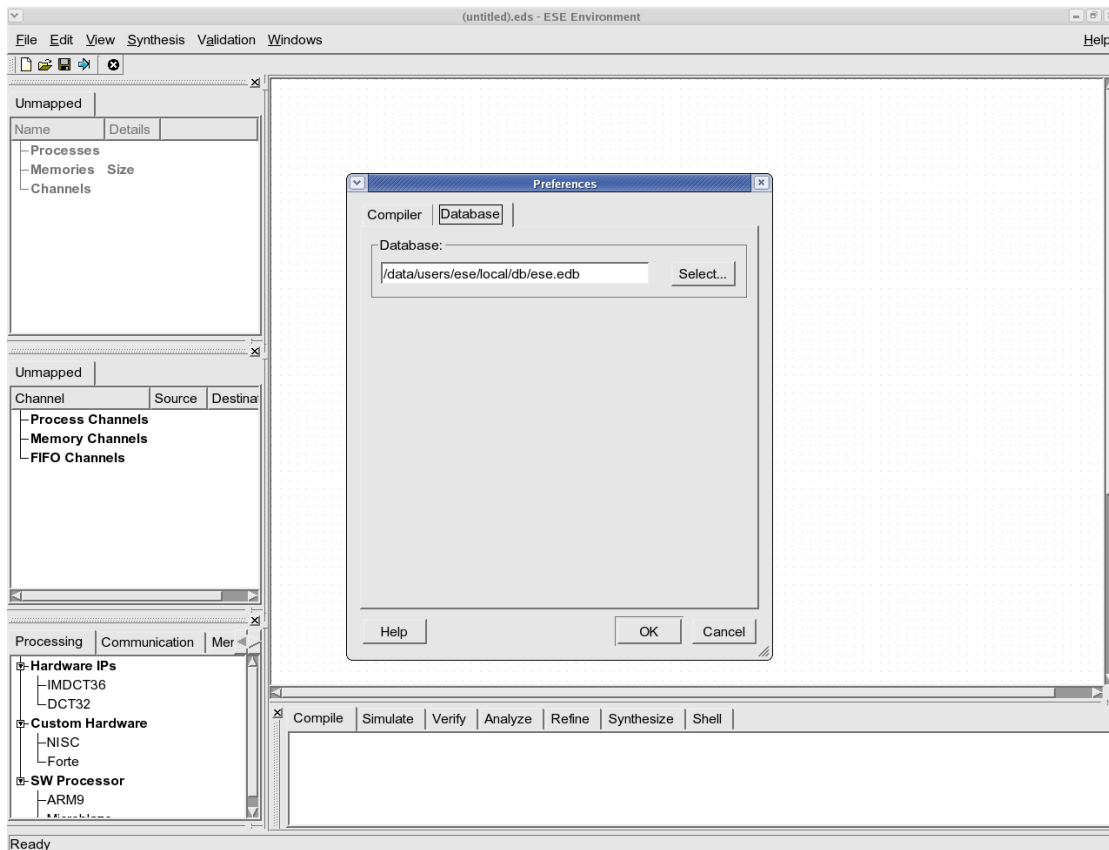
The ESE GUI should now appear as shown in the screenshot. The GUI has several menu items that we shall explore over this tutorial. It is divided into five windows. The top left window is the "PE" window. It organizes the various application processes mapped to PEs in the design. The mid-left window is the "Channel" window that organizes the various channels used for communication between the application processes. The tabs represent the physical communication links in the platform. The bottom left window is the "Database" window that organizes the PE, CE, memory and RTOS model. The top right window is the "Platform Canvas" on which the platform architecture is edited graphically. The bottom right window is the "Logging" window that logs the messages from various ESE tools.

3.1.5. Editing Database Preferences



Before creating a new design, we must ensure that the components needed for our MP3 platform are accessible by the GUI. To do so, we edit the database preferences by selecting **Edit**→**Preferences** from the menu bar.

3.1.6. Select Database File

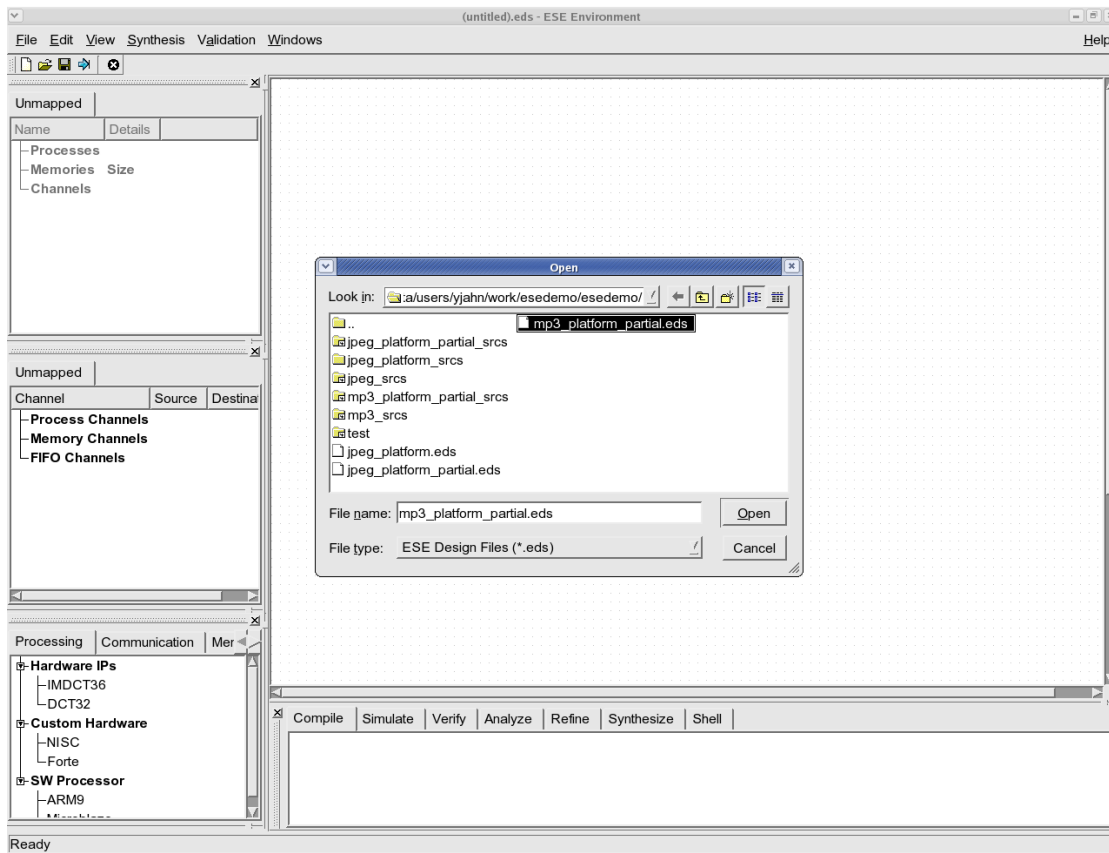


In the **Preferences** dialog, select the tab for **Database**. This will allow the user to browse for the database file that has a ".edb" extension. The database file needed for the MP3 demonstration already comes with the ESE installation. Typically, this file will be called "ease.edb" and will be located at "/data/users/ease/local/db/ease.edb." If the selection is not already there, please browse for the file and press **OK**. The PEs, buses and transducers should now be visible in the database window, if they weren't already.

3.2. Platform Creation

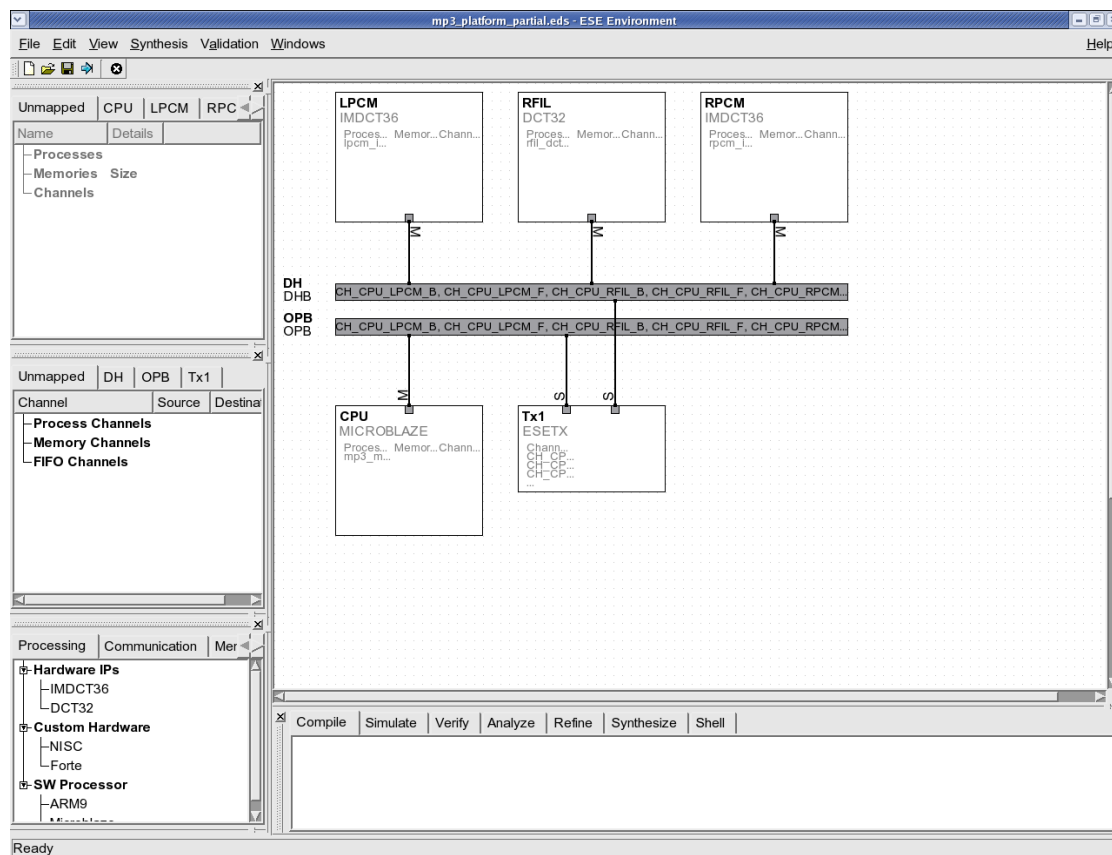
We will start by loading the heterogeneous system design of the MP3 decoder into ESE. As mentioned earlier, we will start with a partial platform consisting of one Microblaze processor and three HW accelerator PEs. The Microblaze processor carries the application code for all of the decoder, except the filter processes, which are the most computationally intensive parts of the application. A new HW PE, customized for DCT32 function, will be added to the platform. In this section we will show how to use the database and platform editor canvas to upgrade a heterogeneous platform in ESE.

3.2.1. Open Partial Design



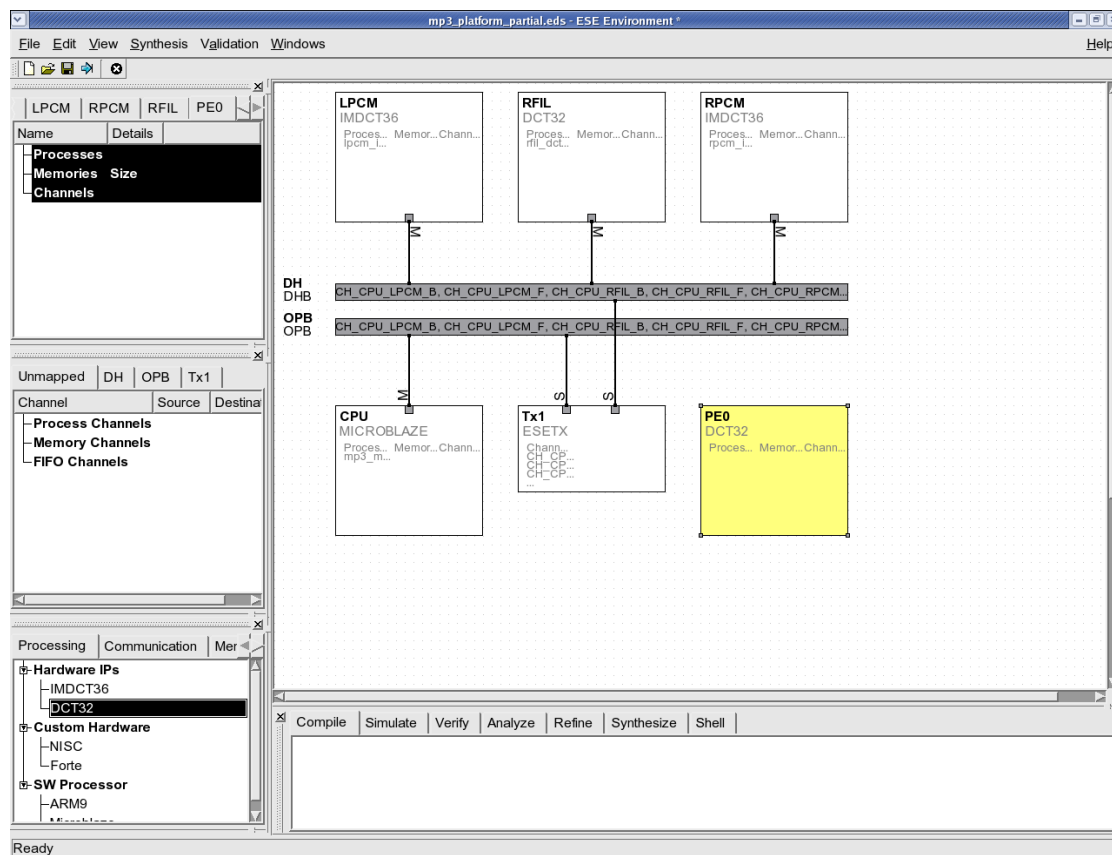
We begin by adding the already created partial design. The ESE designs are stored in XML based files with the extension ".eds." Select **File**→**Open** from the menu bar. Browse into the demo working directory and select "mp3_hsd_platform_partial.eds." This is the design with the partial heterogeneous system design example. Press **Open** to open the design.

3.2.2. View Partial Design



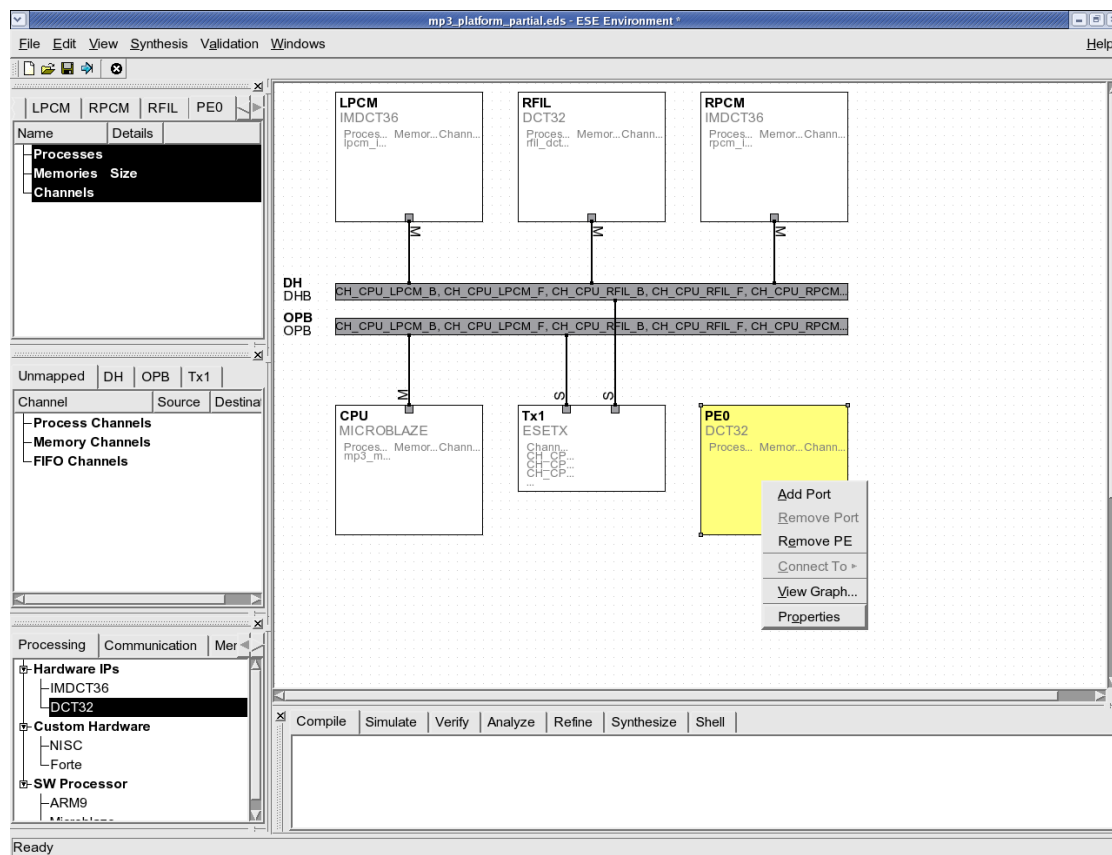
The partial platform will appear in the canvas as shown in the above screenshot. We can see one Microblaze processor **CPU** and three HW filters in the platform. Filters **LPCM** and **RPCM** are for the left and right channel IMDCT36 transforms. Filter **RFIL** perform the right channel DCT32 transform. The HW accelerator for the left channel DCT32 transform is missing and will be added during this demonstration. The CPU connects to the OPB, while the filters connect to the Double Handshake Bus (DHB). Also note that all PEs are connected to their respective buses as "Masters" as indicated by an "M" at the connecting port. Since the DHB and OPB protocols are incompatible, we provide a transducer (Tx1) that acts as a buffer and protocol converter for communication between the CPU and the HW filters.

3.2.3. Add Processing Element



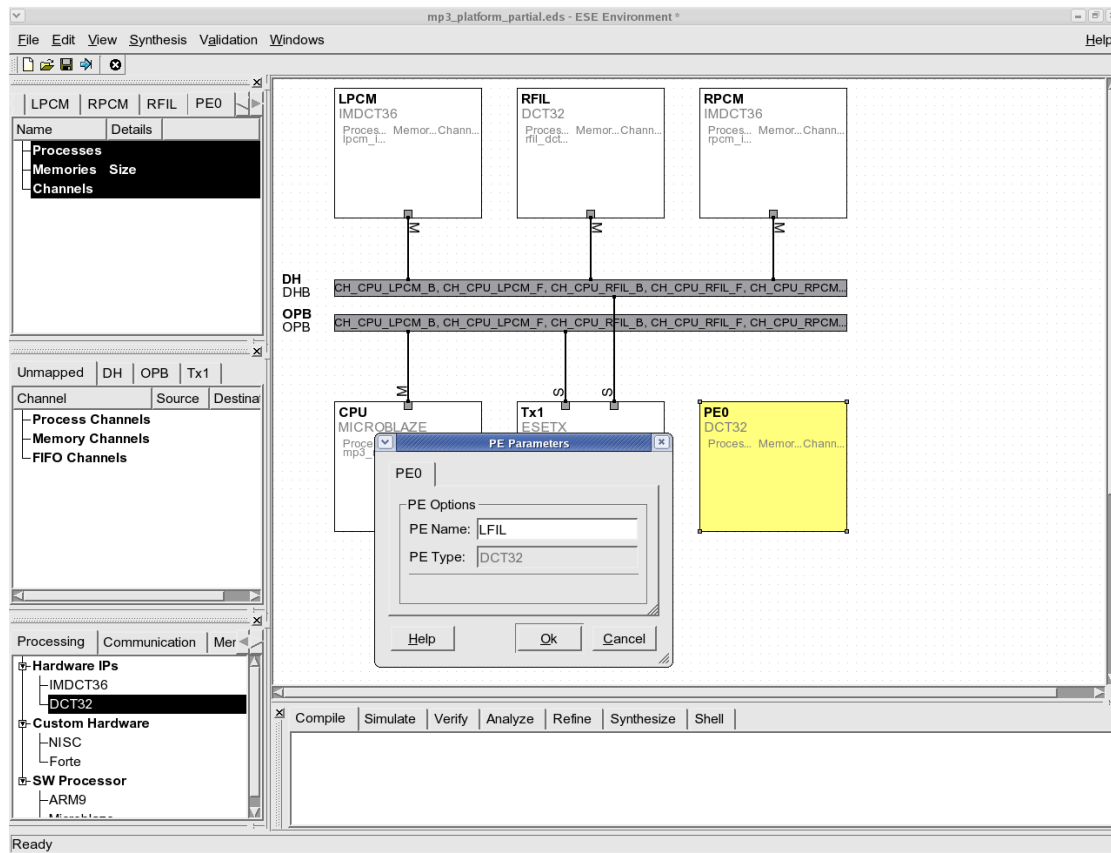
Adding a new PE to the platform is very easy. Browse the database under the Processing tab and select DCT32. Now drag and drop the selection into the central platform canvas. The new PE of type "DCT32" will be added to the platform!

3.2.4. View PE Properties



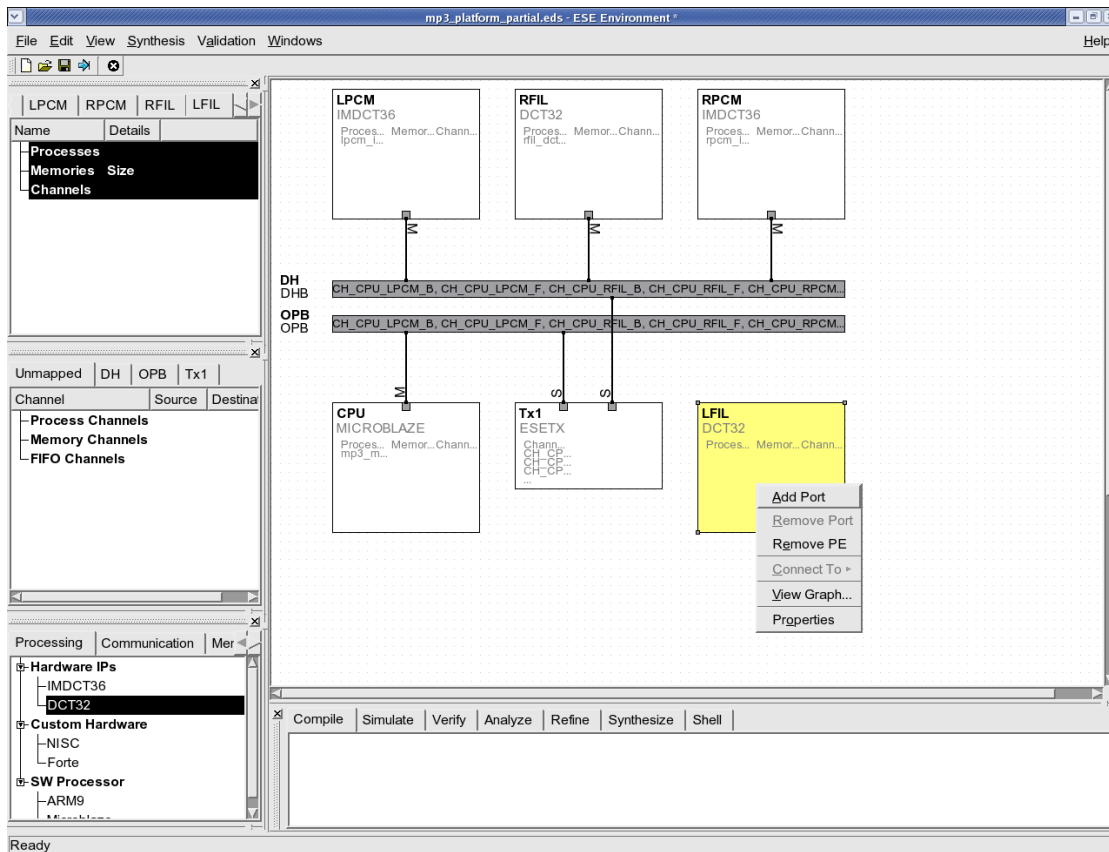
After the drag-drop, the user will find a new PE called PE0 in the platform. This is the PE that will host the missing DCT32 filter process in the design. We start by providing an appropriate name to the new PE to be consistent with the rest of the design. To do so, right click on the PE0 box and select **Properties**.

3.2.5. Assign New Name to PE



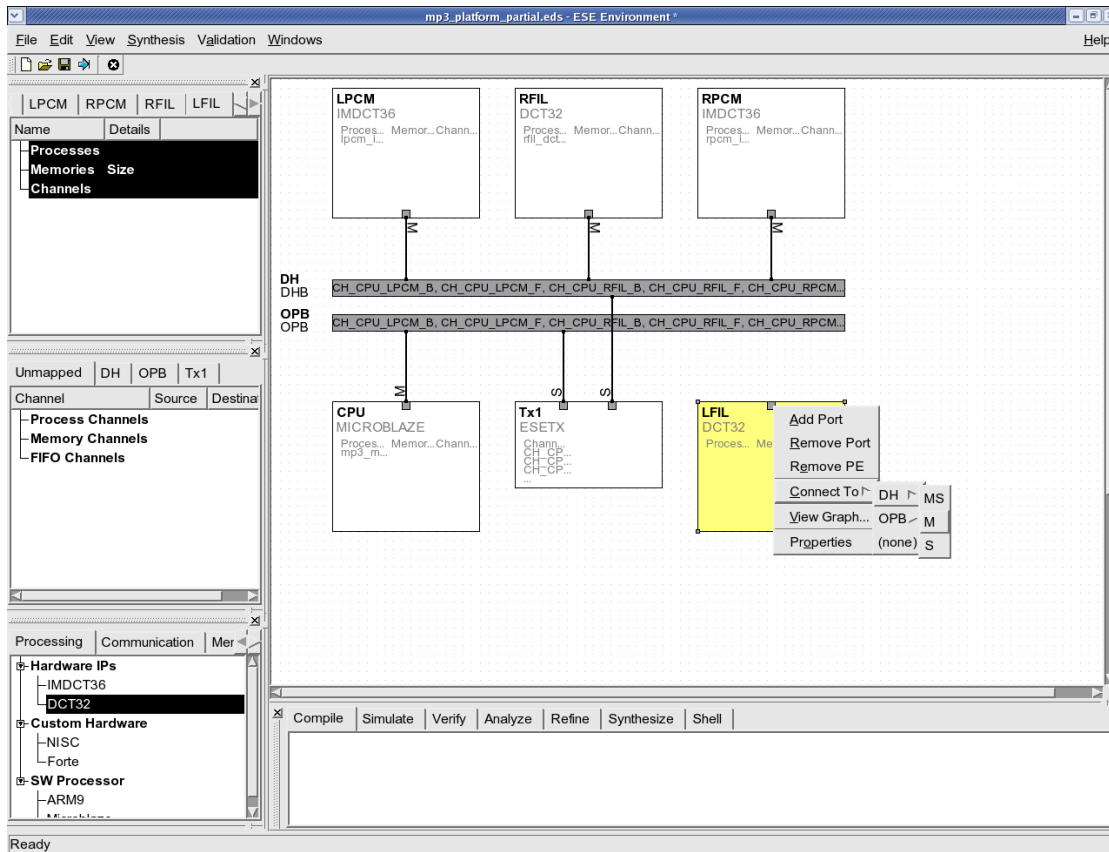
In the properties dialog, change the PE name to "LFIL" to be consistent with the other HW filter names.

3.2.6. Add Port to PE



The new PE, LFIL is not yet connected to the rest of the design. Since the application process meant to execute on this PE will need communication with processes on CPU, we must physically connect LFIL to the compatible DHB bus in the platform. For this physical connection, a port is required for LFIL. To add the port, simply right-click on the LFIL box and select Add Port.

3.2.7. Connect PE to Bus

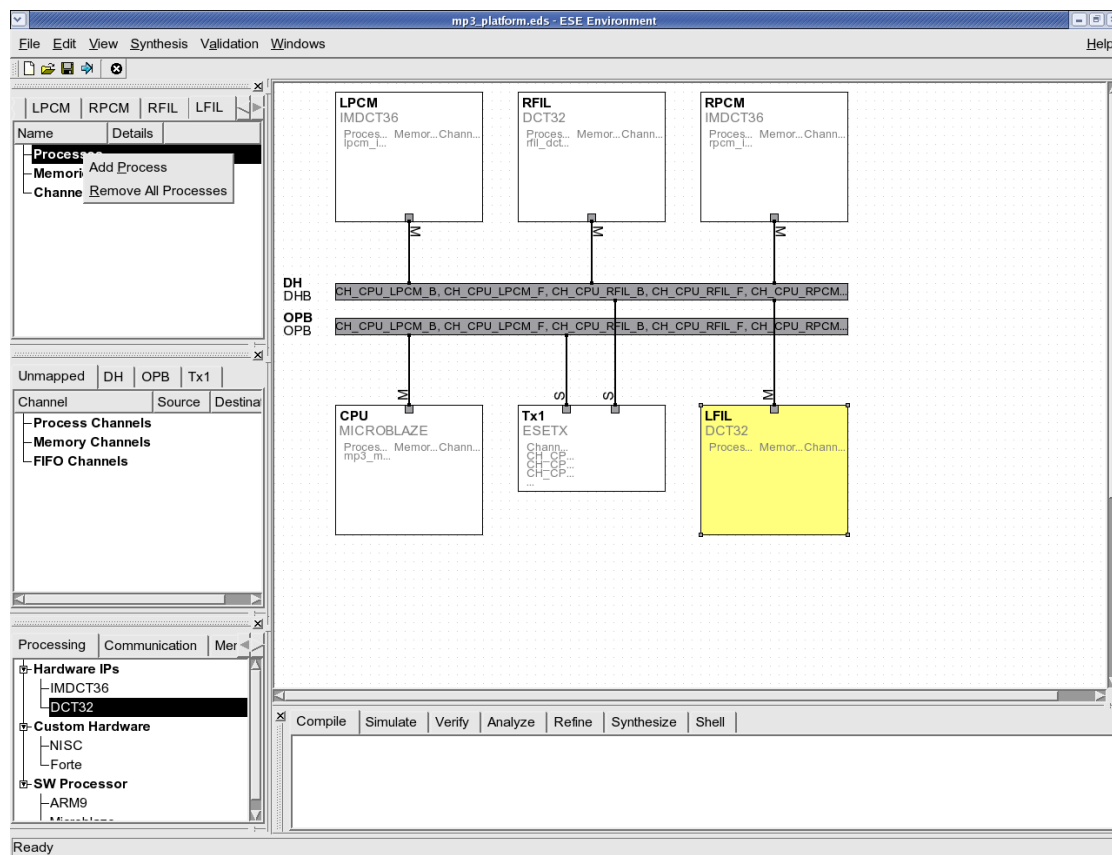


The created port must be connected to the DH bus for LFIL to be able to communicate with the rest of the system. LFIL connects to DH as a Master like other HW filters do. To connect LFIL, right-click on the port and select **Connect To**→**DH**→**M** from the menu choice. This will create the bus connection and complete the platform design step. Next, we will look at application input and its mapping to the created platform.

3.3. Mapping Application to Platform

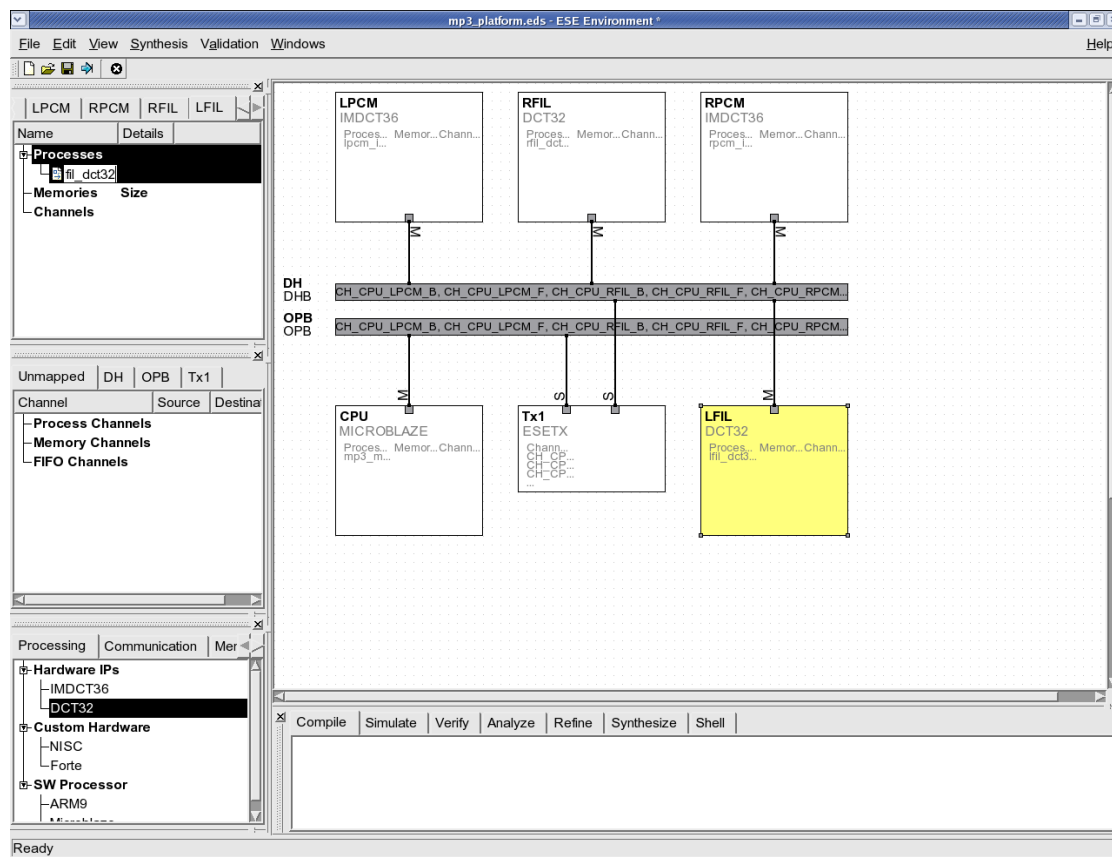
The application input model for ESE is C/C++ processes communicating through either synchronized double handshake channels or shared variables. Since most legacy application is written in C, this is an advantage over other forms of input styles or languages. For communication, the user does not need to write any SystemC channel code. ESE provides very simple APIs for inter-process communication as we will see in this section.

3.3.1. Add Application Process



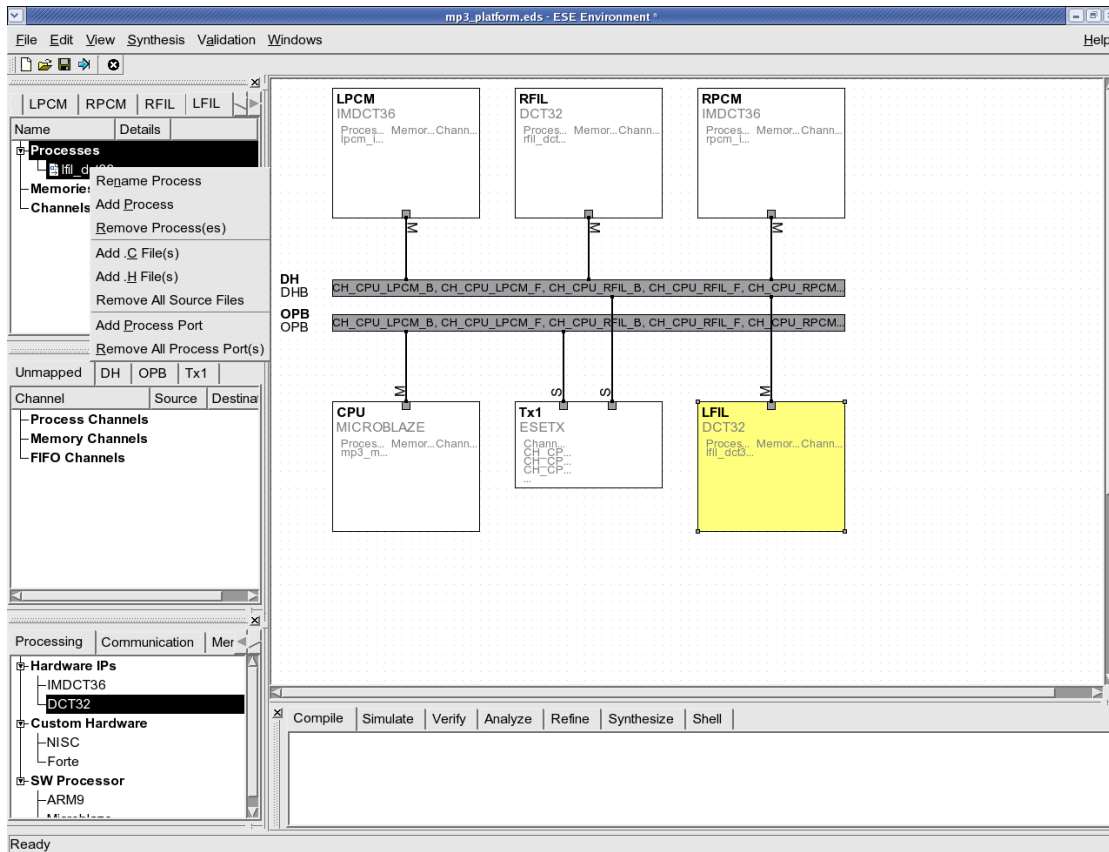
The PE window on the top left corner organizes the processes mapped to the various PEs in the design. In general, several processes may be added for execution on a PE where RTOS should be involved. The platform which has such multi-threaded processors will be demonstrated in Chapter 4. In this section, we assume that there is only 1 process per PE. To add a new process executing on LFIL, change to the **LFIL** tab. Then right-click and select **Add Process**. This will create a new process with a default name.

3.3.2. Assign Name to New Process



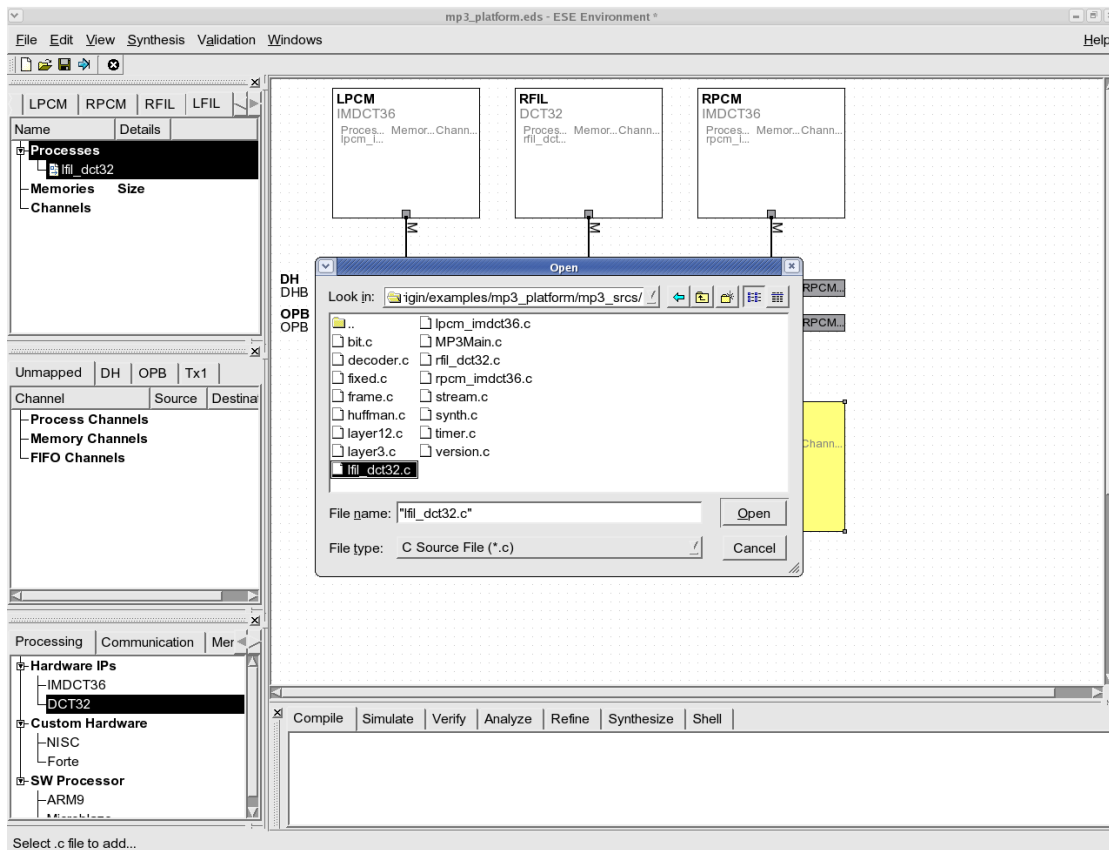
Change the name of the new process to "lfl_dct32". This is the process for the left side DCT32 transform in the MP3 stereo decoder application. Please ensure that the process is named correctly since there exist references to it in the existing partial design. If the process is not named as suggested, the generated models will not compile.

3.3.3. Add C Source File



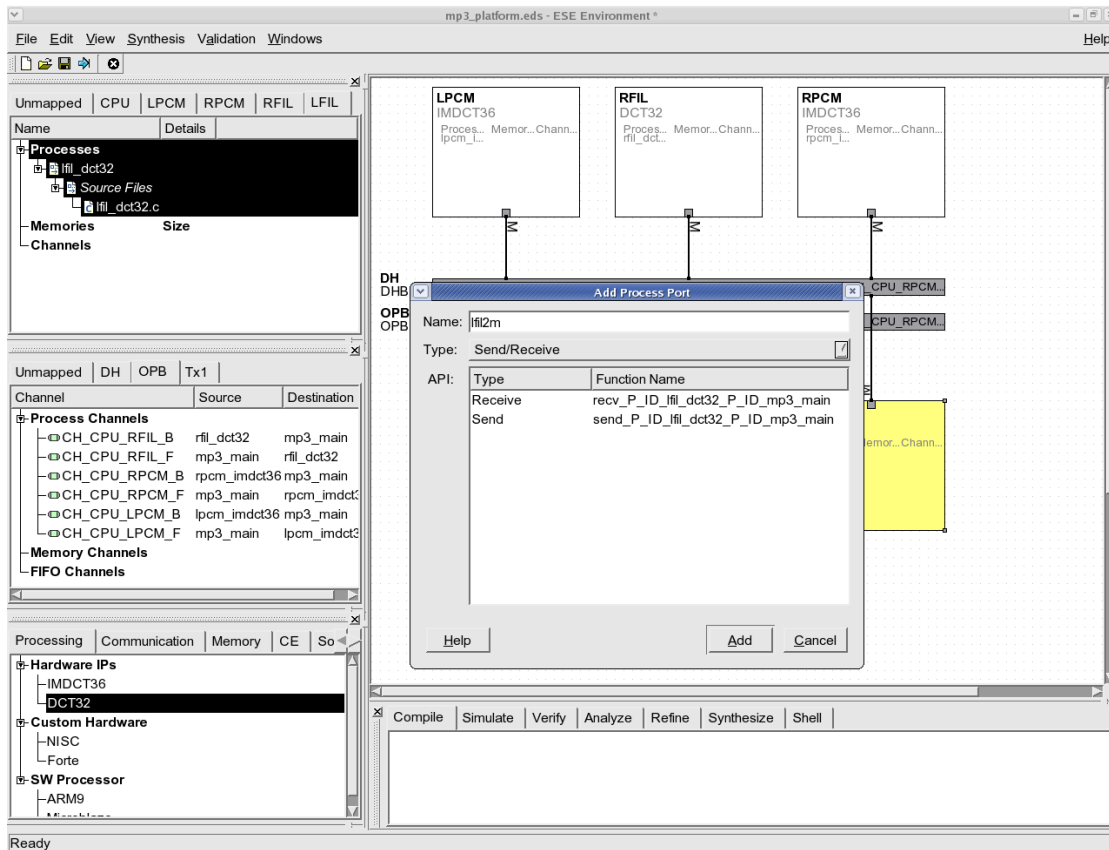
The process added in the last step is only symbolic. The user must associate the actual C/C++ code with it for the models to be functionally correct. In this case, we add C code by right-clicking on the process name in the PE window and selecting Add C File. This will open the file browser.

3.3.4. Select C Source File



Go to the demo directory and follow the symbolic link to "mp3_srcs". Select one ".c" file, "lfl_dct32.c", one ".h" file, "fixed.h", and click **Open**. The files will be added under the new "lfl_dct32" process in the PE window.

3.3.5. Add Process Ports

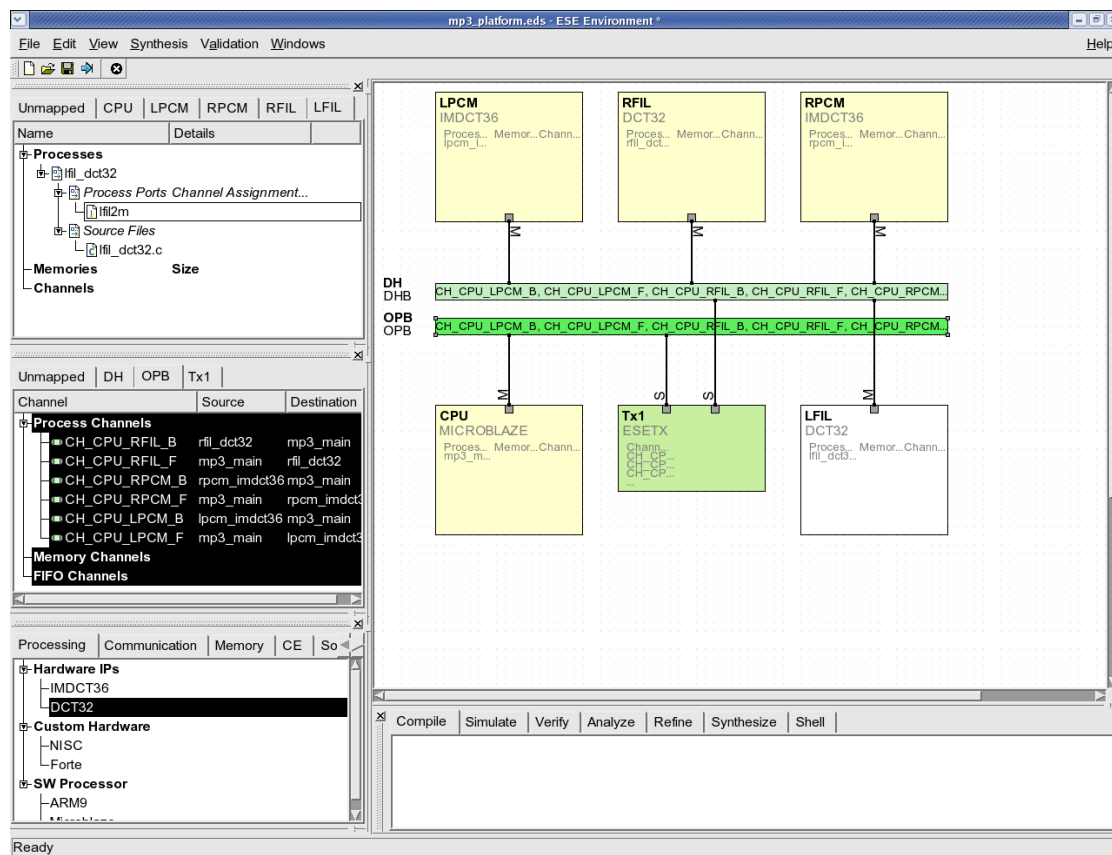


After, the C code for the process is added, we need to add the application level communication to the design. First of all, we add the process port for each process, which will be connected to a channel for data transfer to another process. To add the process port for the new process, click on the new process and select the **Add Process Port**. This will open the window to add the process port. we can create any name for the process port and select the type of it. There are ten possible types. We can categorize them into three kinds. The "Send", "Receive", and "Send/Receive" are used for double handshake channels. The "Read", "Write", and "Read/Write" are used for shared memory. And the others are for the FIFO channels. Finally, we need to assign its function name to be what is actually used in C code. Please ensure that the function name is the same as that used in C code. If the name is not correct, the generated models will not compile.

The "lfl_dct32" process needs one process port for sending/receiving data from/to "mp3_main". Assign the process port name to be "lfl2m" and select "Send/Receive" type since we are using the bi-directional double handshake channel for the communication between the "lfl_dct32" process and "mp3_main" process. Assign the

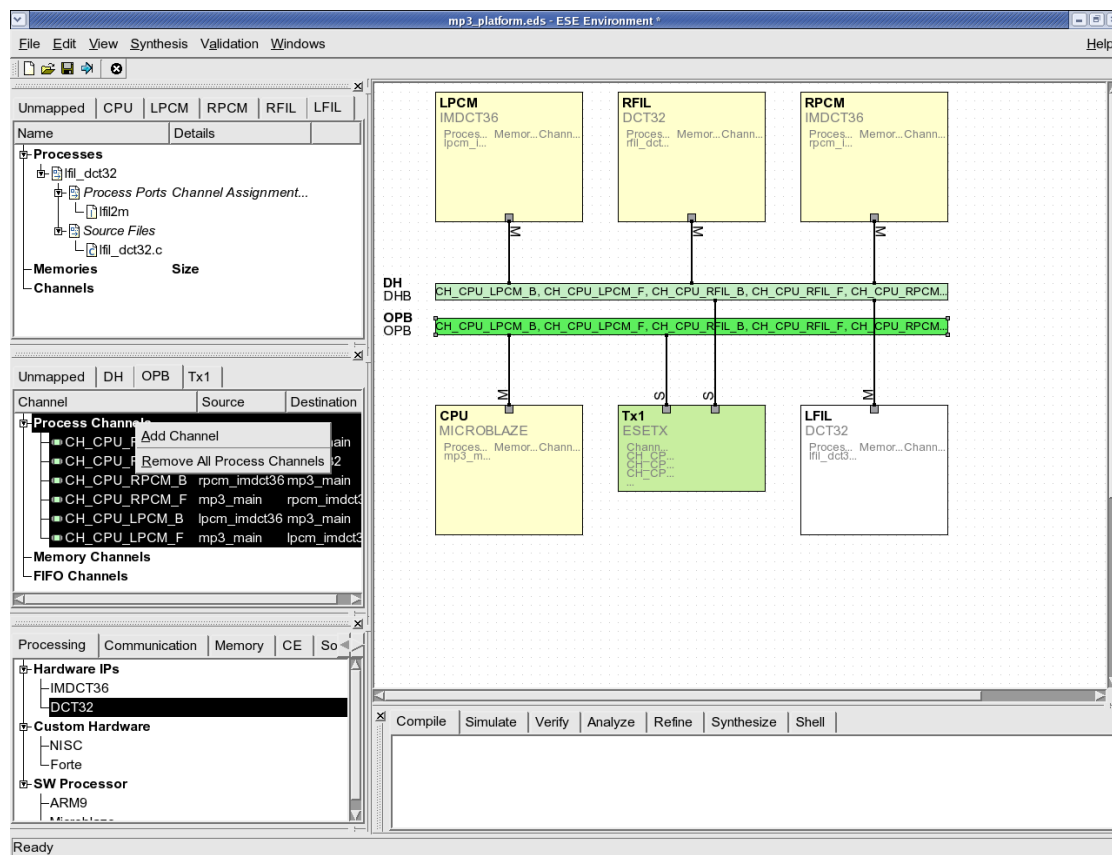
function names to be "recv_P_ID_lfil_dct32_P_ID_mp3_main" for "Receive" type and "send_P_ID_lfil_dct32_P_ID_mp3_main" for "Send" type, respectively.

3.3.6. View Application Channels



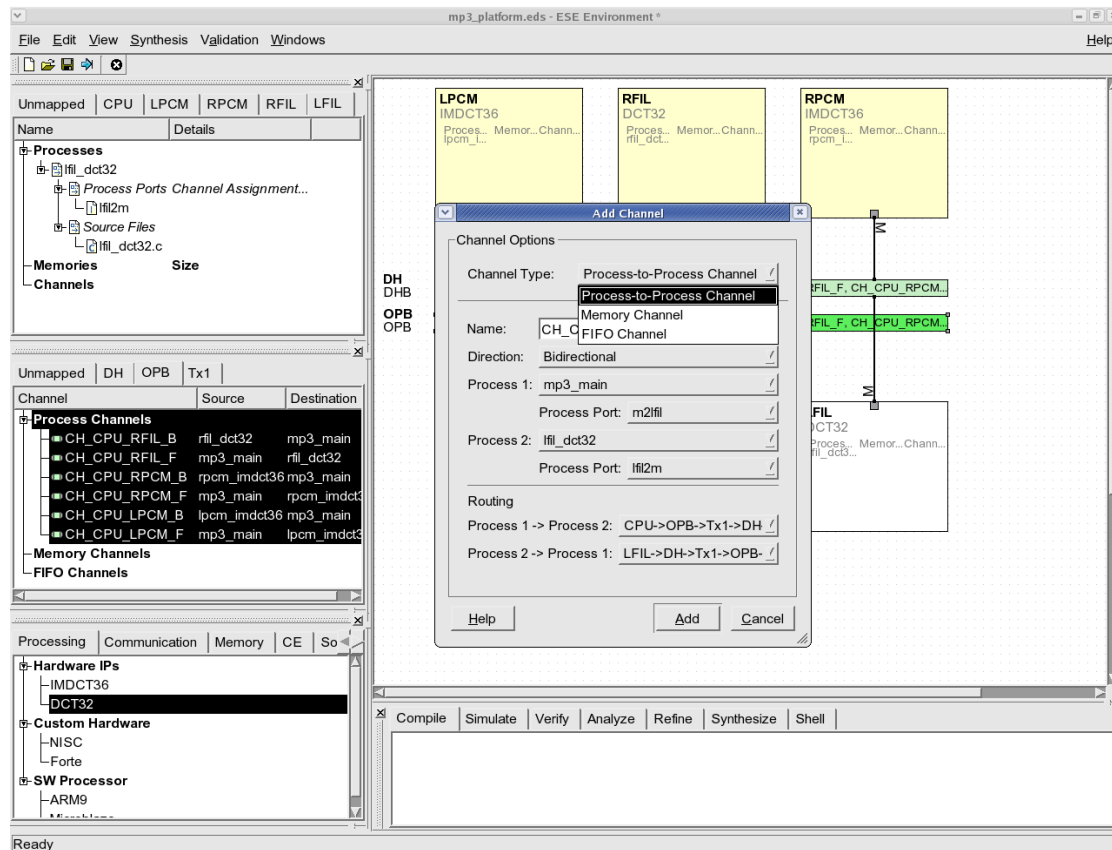
After the process port for the process is added, we need to create the channels for the communication between processes. To view the existing double handshake message passing channels in ESE, click on the **OPB** tab. This will display the existing channels between all the processes in the design, including the source and destination names as well as the route used to implement the channel in the communication platform. All the channels in ESE are uni-directional. Bi-directional channels can be added as a pair of "Forward" and "Backward" channels conveniently in the GUI.

3.3.7. Add New Application Channel



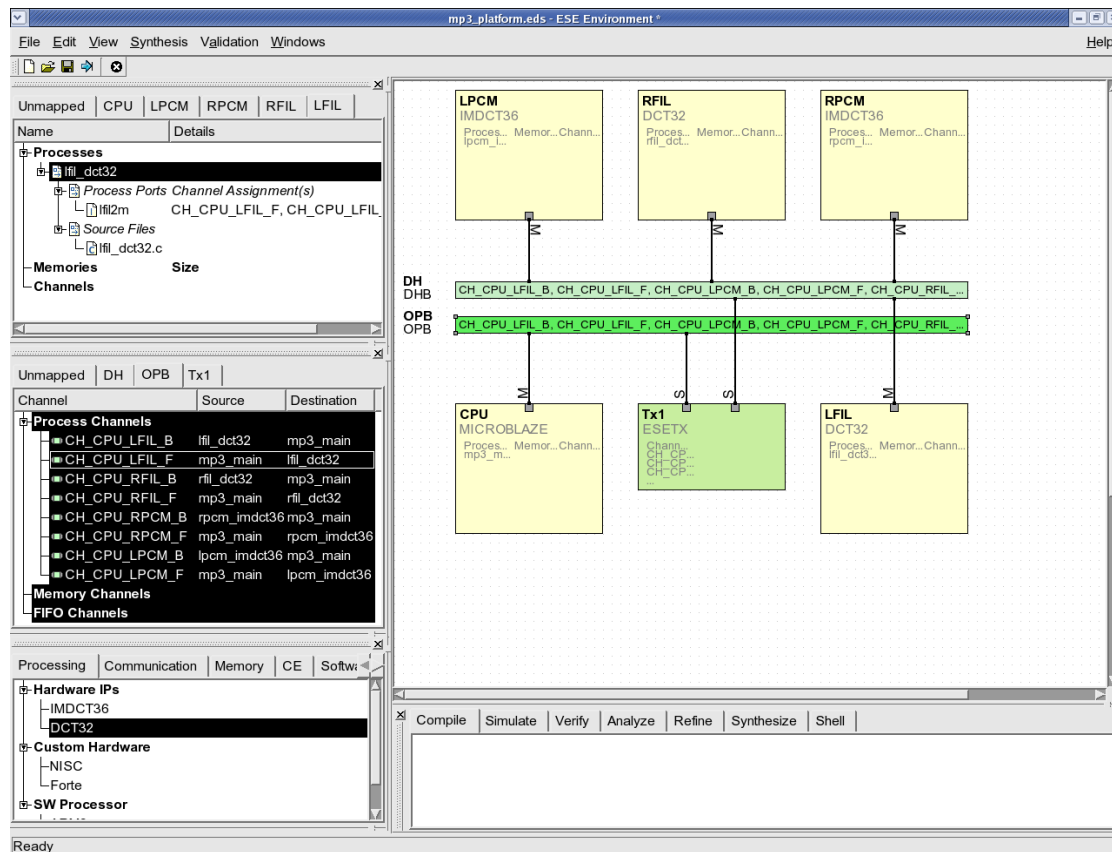
If the user clicks on a PE in the platform canvas, all the channels originating or terminating at the PE will be selected. All other PEs that the clicked PE communicates with will be highlighted in light yellow. All physical connections, including buses and transducers used by the PE for communication will be highlighted in green. Note that clicking on LFIL shows that it is not connected at the application level to any other PE. Since we need communication between the "lfil_dct32" process and the "mp3_main" process executing on CPU, we will add the application level channels, by right-clicking in the channel window and selecting Add Channel. This will pop up the channel wizard for adding application level channels.

3.3.8. Channel Wizard



In the channel wizard dialog, we first need to select the channel type. Choose "Process-to-Process Channel" since we are using message passing channels. Then, assign the name to be "CH_CPU_LFIL" for consistency with existing channels. Since, the processes will send data both ways, select a bi-directional channel type. Use the pull down menu to select the first communicating process as "mp3_main". This is the process running on CPU. Next, use the next pull down menu to select the other communicating process "lfil_dct32." Once the communicating processes are decided, ESE automatically filters all the possible physical routes on the platform that can implement the channels. For this example there is only one route from CPU to LFIL that goes over the OPB bus via the transducer Tx1 and over the DH bus to the receiver LFIL. The route goes through the transducer because CPU and LFIL are connected to different physical buses, which does not allow direct communication. Similarly, there is only one route from LFIL to CPU. Select the default routes and click Add to add the channels.

3.3.9. View New Channel Communication



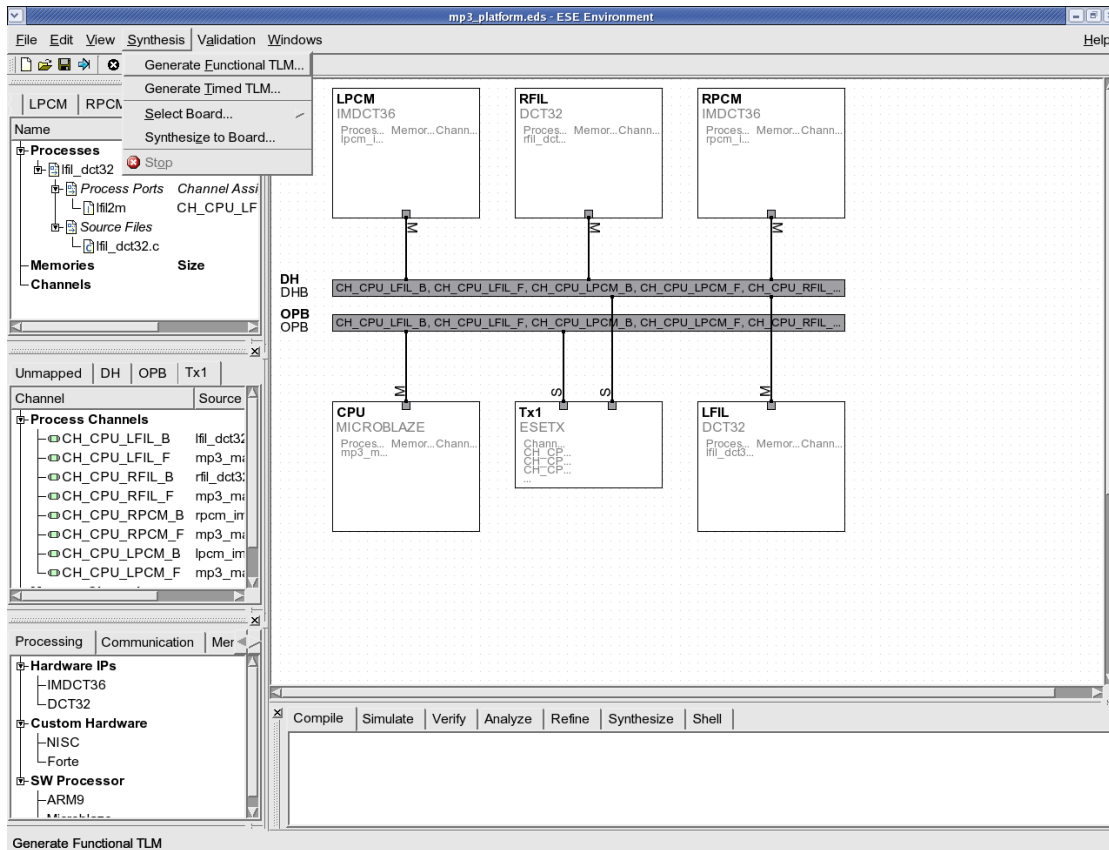
The newly created channels will now be visible in the channel window under the Tx1 tab. Note that the channel names have "_F" and "_B" appended to distinguish between forward and backward channel, respectively. The user may alternately make unidirectional channels one at a time. Once the channels are selected, the communicating PEs will be highlighted. This shows that the new PE LFIL is now "connected" with the rest of the system on an application level.

3.4. Generating Functional and Timed TLMs

The previous steps complete the platform and application input that is necessary for generation of TLMs. We will show generation of two types of SystemC TLMs. The first one is called the "Functional TLM" because it is used for the validation of design functionality only. It is completely un-timed and simulates the design based on causal dependency only. A universal bus channel is used to model the system bus and the mapping of channels on the bus.

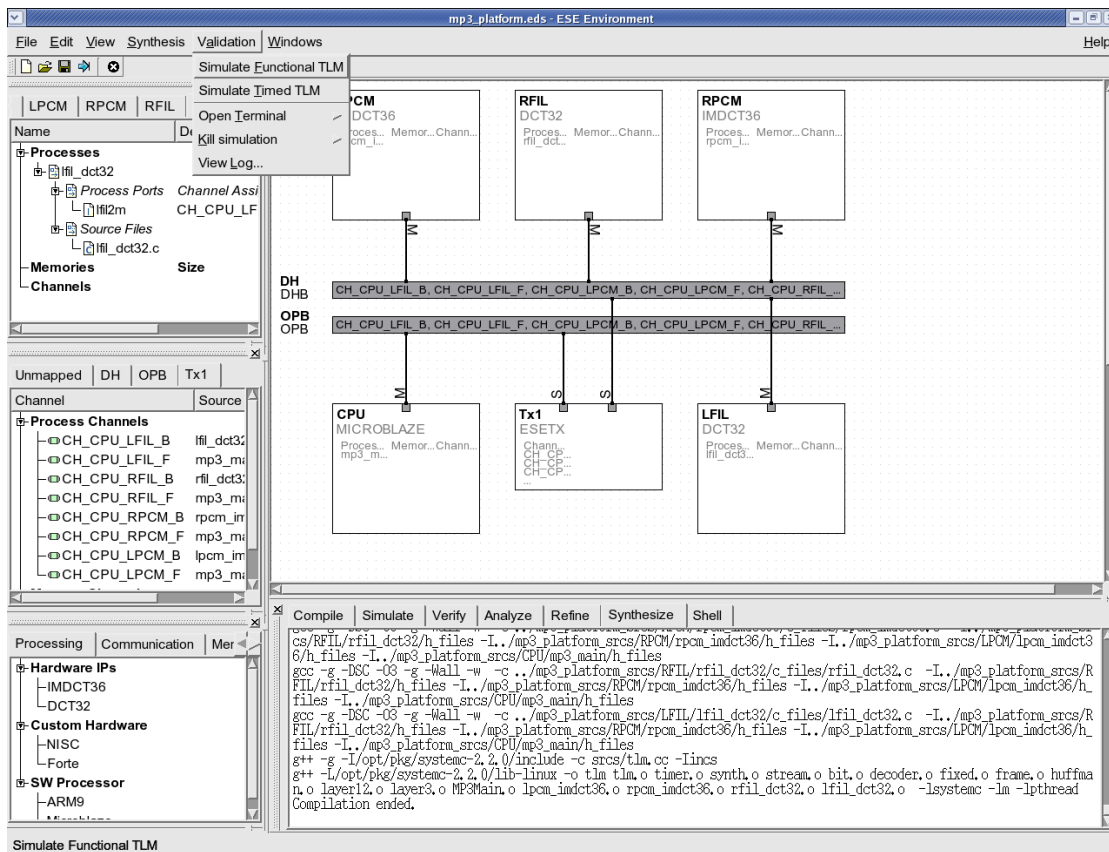
The second TLM is called the "Timed TLM" and is used for performance estimation of the design. It relies on timing data models of PEs and Buses that are available in the ESE database. The data models are used by our estimation and annotation technique to apply "wait" statements in the application C code. The technique is retargetable and applicable to processors as well as HW IPs. A retargetable bus timing annotation modifies the bus channel to apply "wait" statements for inter-PE communication.

3.4.1. Generate Functional TLM



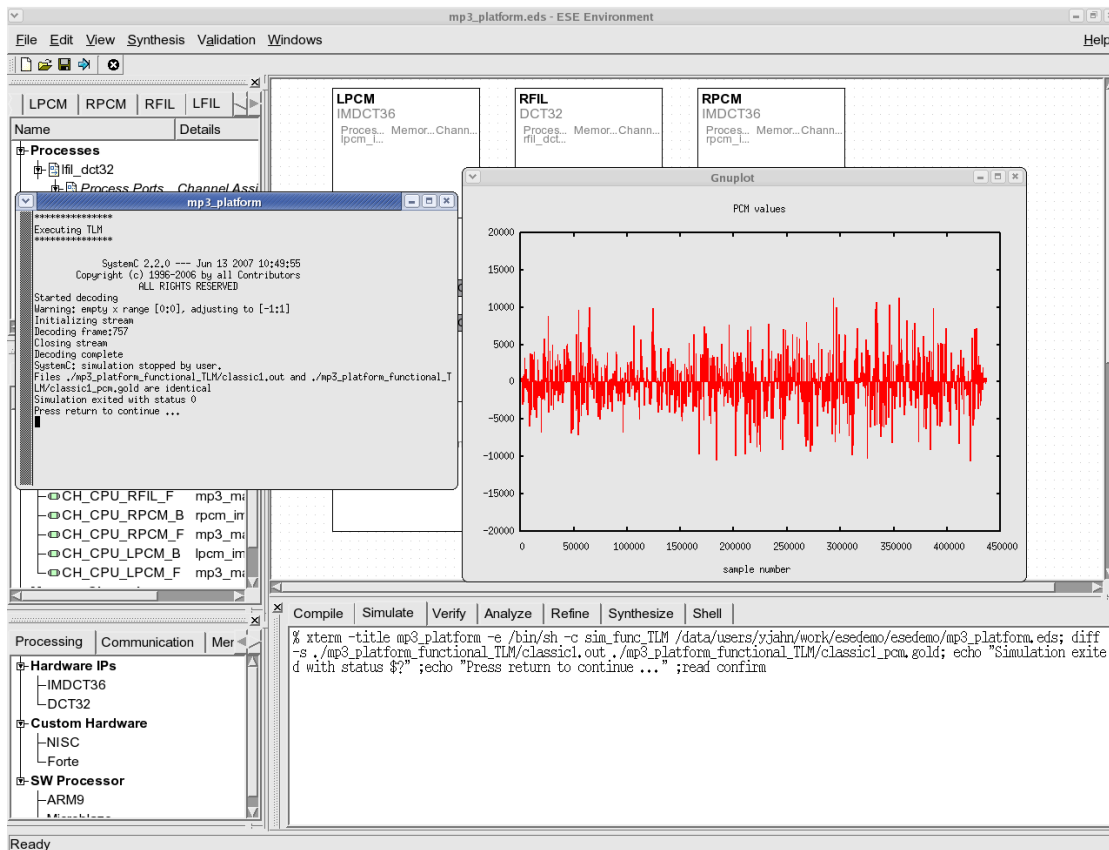
After the platform and application input is complete, the functional TLM can be generated automatically by selecting **Synthesis**→**Generate Functional TLM** from the menu bar. This will generate the SystemC code needed for platform modeling, including PEs, buses and transducers. The generated code is then compiled natively along with the C application code and linked to the SystemC libraries to produce a single binary. This process can be viewed in the log window.

3.4.2. Simulate Functional TLM



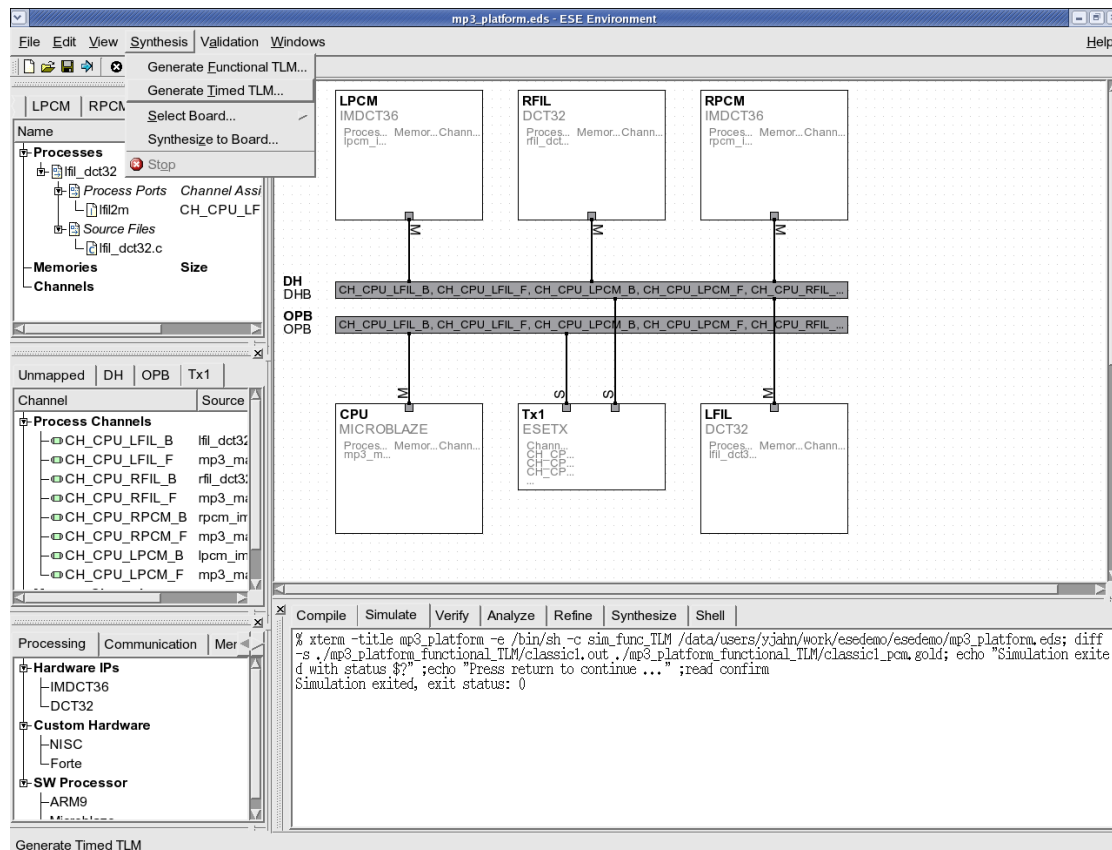
Once the compilation has completed, the generated TLM can be executed from the GUI by selecting Validation—>Simulate Functional TLM from the menu bar.

3.4.3. View Functional Simulation Results



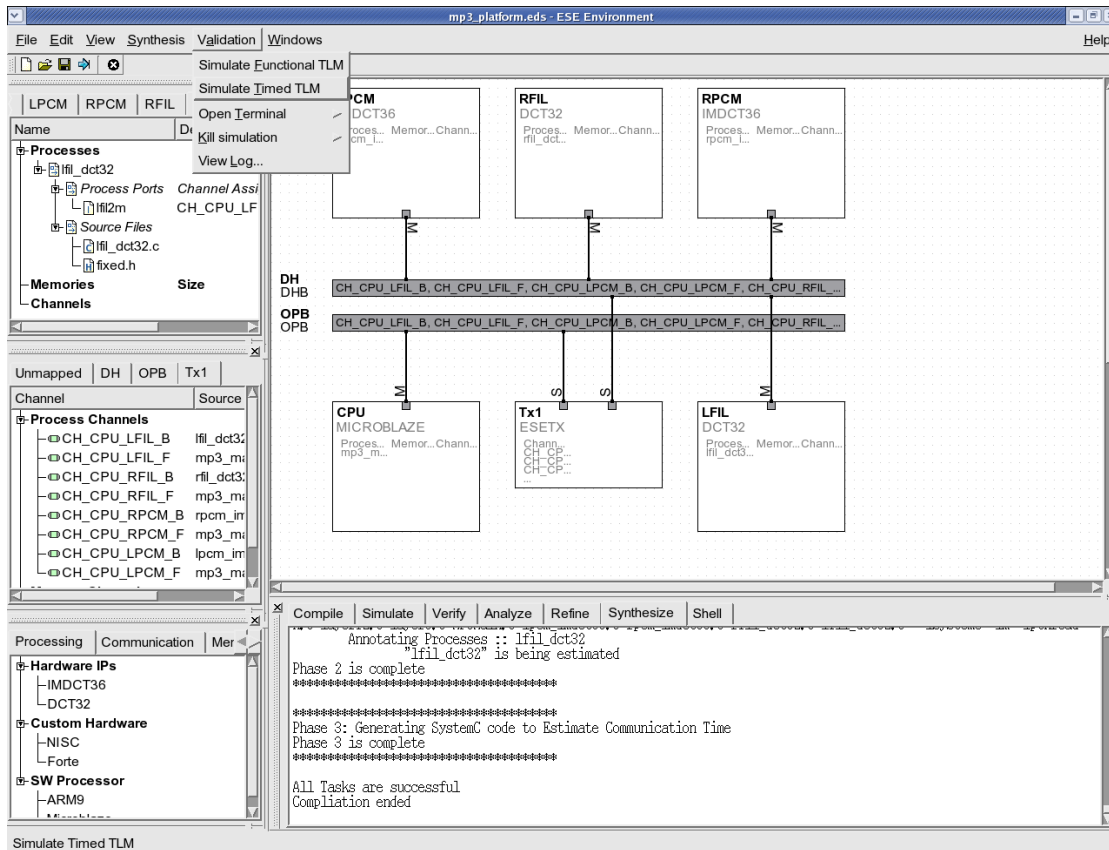
The simulation pops up a terminal that logs the number of MP3 frames that have been decoded. An additional window shows the simulation progress frame by frame. Each decoded MP3 frame produces PCM output that can be fed to the audio output. The y-axis values in the PCM output view shows the decoded values. The logging of PCM output stops once all the frames have been decoded. The pop up windows can now be killed simply by pressing "Enter" in the simulation logging terminal.

3.4.4. Generate Timed TLM



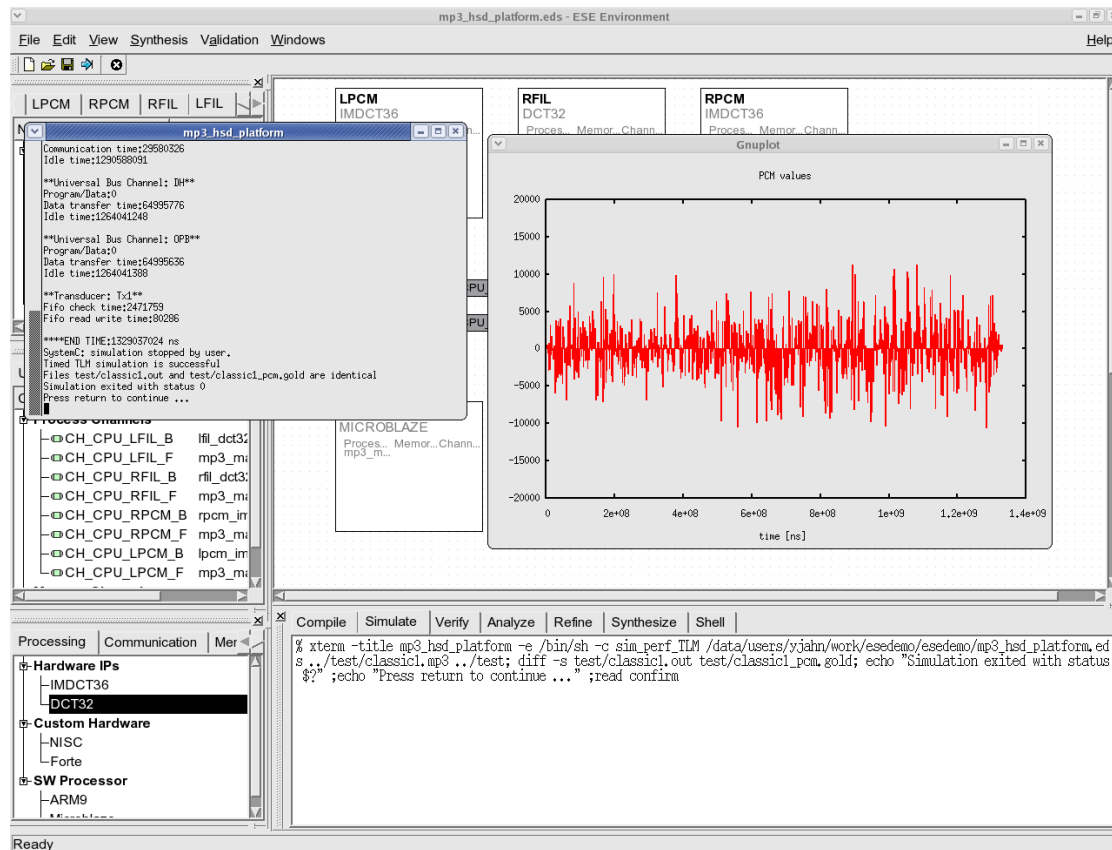
Similar to the functional TLM generation, the Timed TLM can be generated automatically by selecting **Synthesis** → **Generate Timed TLM** from the menu bar. The bus channels generated for timed TLM will include timing for synchronization, arbitration and data transfer. The timing parameters are imported into the TLM from the bus data model. For the computation part, we use a retargetable source level timing estimation technique that utilizes the PE data models. Naturally, the timed TLM generation and compilation is significantly slower than functional TLM generation, but still in the order of seconds.

3.4.5. Simulate Timed TLM



To simulate the generated timed TLM, simply select Validation→Simulate Timed TLM from the menu bar, after the TLM compilation has ended.

3.4.6. View Timed Simulation



The timed TLM simulation looks very similar to the functional TLM simulation except for some marked differences. Firstly, notice that timed simulation is significantly slower than functional TLM simulation. This is natural since we are simulation a lot more "wait" statements that are annotated to the application codes. However, our results show that this is still several orders of magnitude faster than RTL simulation for the same design. Secondly, note that the X-axis on the PCM viewing window now shows estimated cycles instead of frame numbers. This is because the estimated cycles for each frame are available at runtime as a result of our source level "wait" annotations.

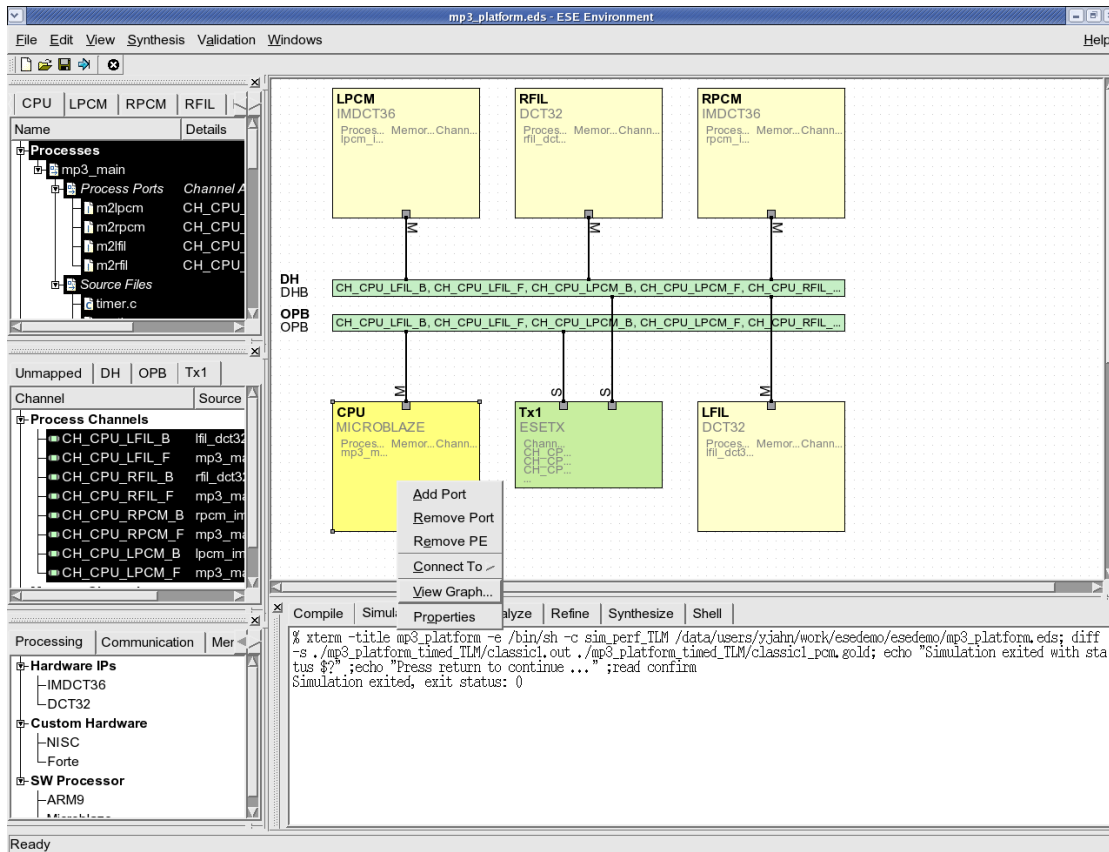
When the timed simulation ends, several statistical data are dumped in the simulation logging terminal. These are the estimated cycles for CPU computation and communication, bus congestion estimates and so on. However, all these estimated performance statistics can be viewed graphically as shown in next section.

3.5. TLM Performance Estimation

The timed TLM produces several statistical data that is gathered during simulation. Since the source annotation is fine grained, the TLM produces results for cycles used for invocation of each function in the application code. Computation and communication cycles for each PE can be viewed using pie charts. The distribution of cycles for each function amongst its sub-functions can be browsed recursively. Similarly, the distribution of inter-PE bus traffic over inter-process channels can also be viewed graphically.

The performance estimates are useful for early platform and mapping evaluation. Since the timed TLMs are generated automatically, and TLM simulation is very fast, early design space exploration becomes feasible. Users may explore platforms manually or plug in their exploration algorithms for system level design optimization.

3.5.1. View Performance Estimates



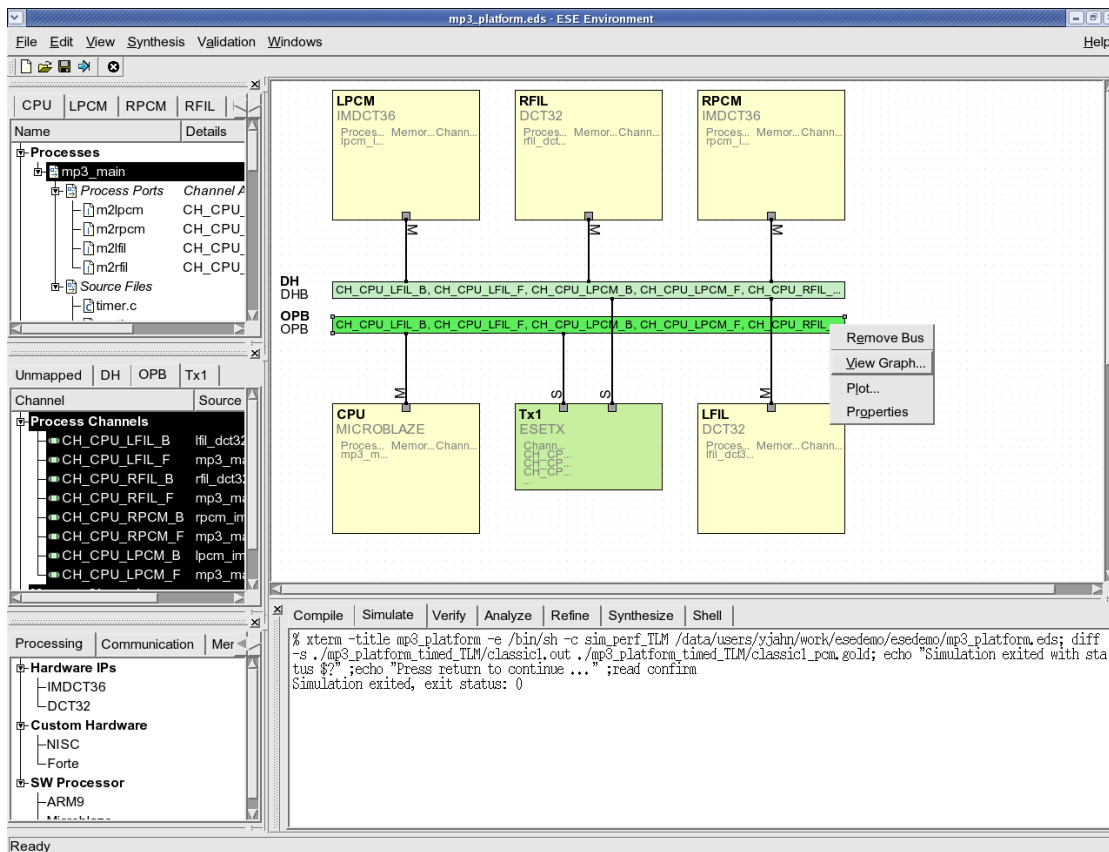
To view the PE performance statistics, right-click on the PE in the platform canvas and select View Graph. In this case, we will select the CPU Microblaze processor.

3.5.2. PE, Process and Function Level Estimates



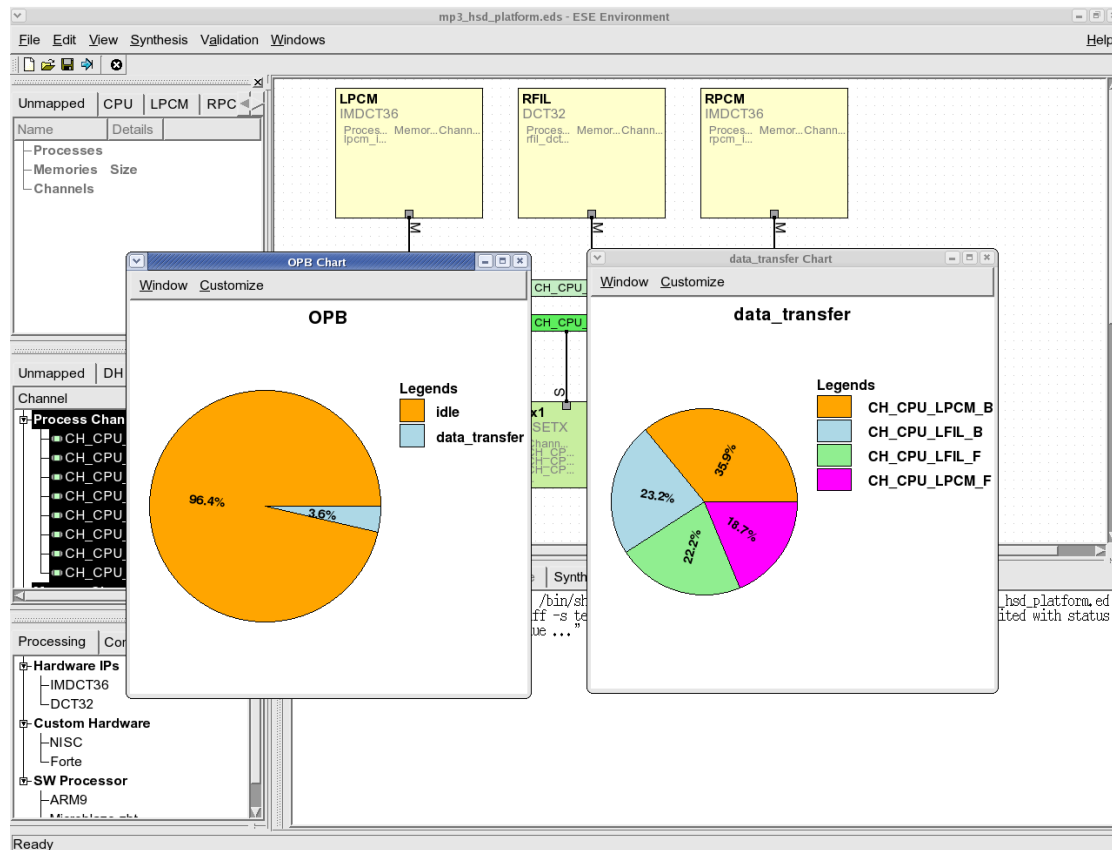
The first pie-chart window divides the total execution time into computation, communication and idle cycles. Double-clicking on the computation part of the pie chart pops up the distribution of computation across different processes in the design. In this case, we have only 1 process "mp3_main" mapped to CPU, hence it is 100%. Double-clicking on the process in the pie chart produces the distribution of computation across the top level functions in the process. These function(s) call lower level functions and so on. Double-clicking on a function produces the pie chart for the distribution of cycles amongst the sub-function invocations. Using this viewing feature, the user may go down to any level in the function call hierarchy. If the pie chart appears too small, please increase the window size to enlarge the chart.

3.5.3. View Communication Estimates



To view the bus communication statistics, right-click on the bus in the platform canvas and select View Graph. In this case, we will select the OPB bus.

3.5.4. Bus and Channel Level Estimates



The top level pie-chart for the bus shows the distribution of bus cycles in idle, program/data access and inter-PE data transfer phases. Double-clicking on the "data-transfer" part of the pie-chart produces the distribution of communication traffic amongst the various application channels in the design.

Chapter 4. Multi-threaded System Design with ESE

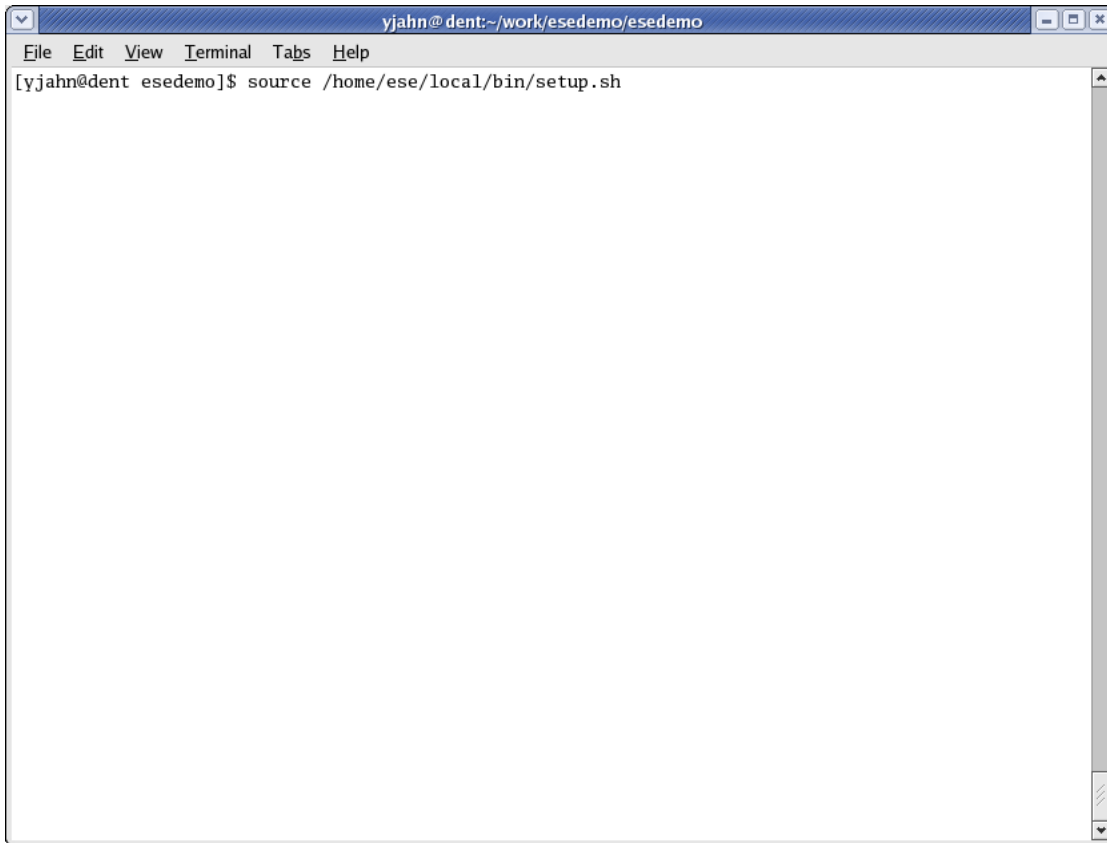
This section deals with design of JPEG encoder on a platform consisting of two multi-threaded MicroBlaze processors and one OPB. The JPEG application code is available as a C model. The JPEG encoder has five processes. Three processes are mapped into one processor and two processes are mapped into the other processor. Since they are multi-threaded in a processor, we need a RTOS model to control and schedule the execution of the processes. ESE provides two kinds of scheduling policies, Round-Robin and Priority-based scheduling. Users can select one out of the two policies. The communication between the processes can take place through pairs of various channels such as process-to-process (or point-to-point) message passing channel, shared memory channel and FIFO channel. In this section, all the channels in the JPEG encoder are via FIFO channels. ESE provides well defined communication APIs for this purpose. The encoded output is shown graphically during the TLM simulation of the JPEG encoder.

The chapter starts by explaining the set up for ESE. It then shows, using screenshots, how the platform is created. To speed up the demonstration, and to emphasize on the features, we start with an existing partial platform that is upgraded with additional processors and a bus. Then we show the application mapping on the platform, followed by TLM generation, simulation and performance estimation. Thus, we present the core capabilities of the ESE Front-End tools in easy platform design & upgrade, model generation, validation and estimation.

4.1. ESE Startup and Settings

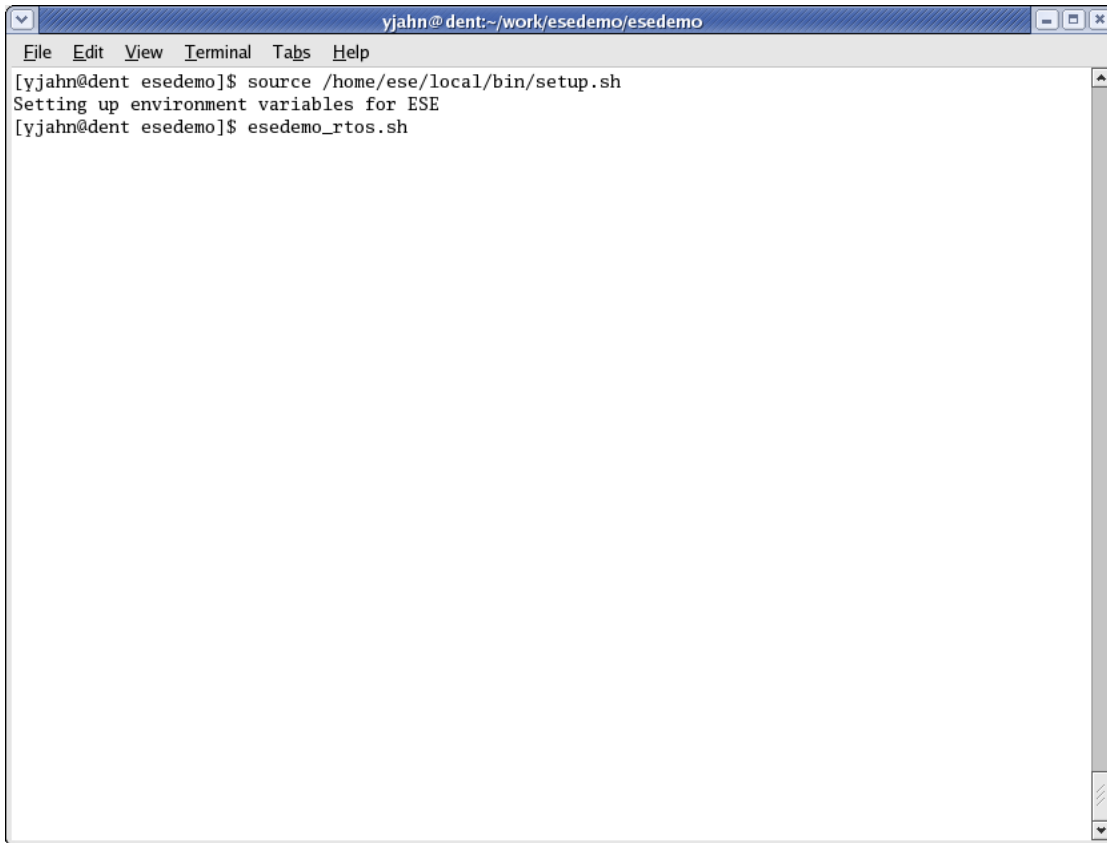
Before starting the demonstration, please ensure that you have the ESE software installed in the right location at `"/home/ease/local."` Also make sure that you have an `"/home/ease/local"` directory containing the SystemC 2.2.0 libraries and SDL libraries that are needed for simulation of generated TLMs. Also make sure that you have GCC version 3.4 or higher because it is needed to correctly compile the generated TLMs. The demonstration shown here assumes the user to have a bourne shell. For C shell, the user may call the `".csh"` version of the setup scripts. Alternately, just use `"sh"` to create a new bourne shell and follow the tutorial directions.

4.1.1. Environment Setup



We start by setting up the environment variables to access ESE binaries. This is provided by the "setup.sh" script in your installation. Typically, the installation path would be "/home/e/e/local." The script is in the "bin" directory in the installation. The script modifies your PATH environmental variable to include path to ESE as well as the LD_LIBRARY_PATH variable to access the shared libraries that ESE depends on. Run the command "source /home/e/e/local/bin/setup.sh" and create a new local directory for the demo.

4.1.2. ESE Demonstration Setup



The screenshot shows a terminal window titled "yjahn@dent:~/work/esedemo/esedemo". The window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The terminal content shows the following commands and output:

```
[yjahn@dent esedemo]$ source /home/ece/local/bin/setup.sh
Setting up environment variables for ESE
[yjahn@dent esedemo]$ esedemo_rtos.sh
```

Once the environmental variables have been set, the user is ready to launch ESE and create his or her design. For the purposes of this tutorial, we will start with a partial design to quickly demonstrate the key capabilities of the toolset. We have created a shell script called "esedemo_mtd.sh" that prepares a partial design to start the demo for the JPEG encoder. At this point, run the "esedemo_mtd.sh" script after changing into the local directory created for the demo.

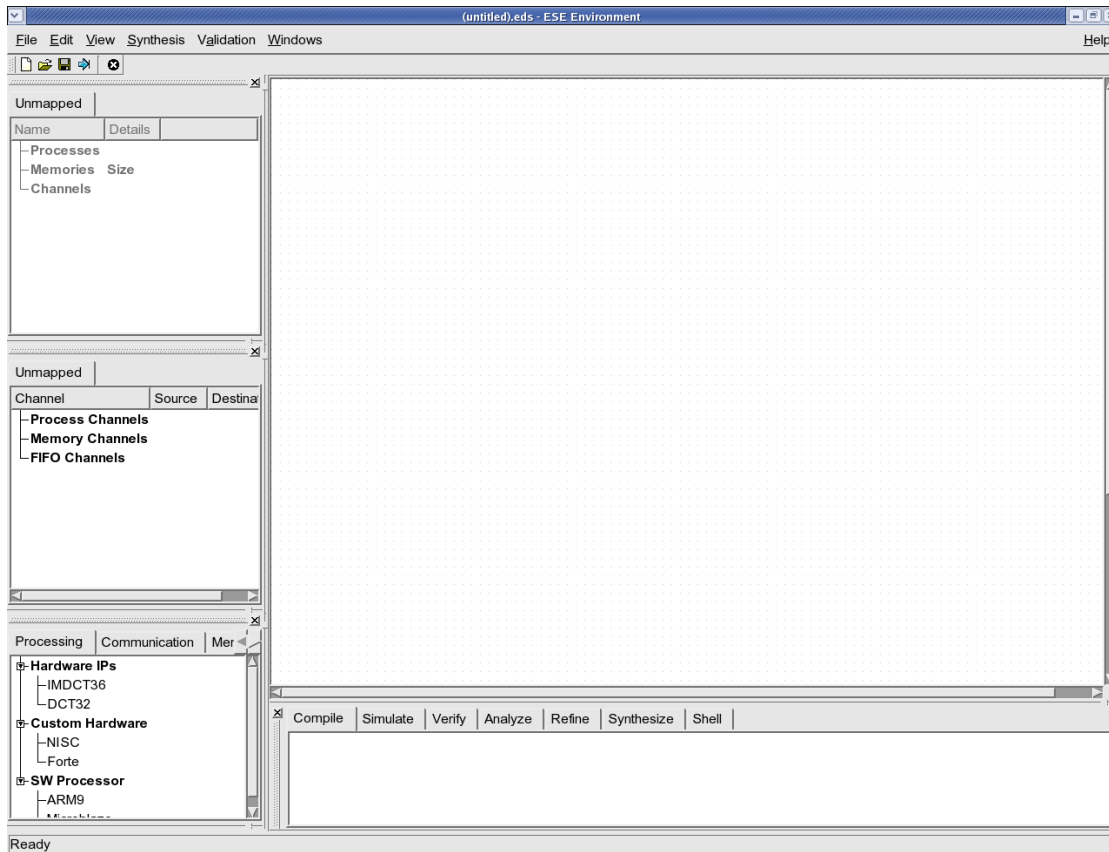
4.1.3. Launching ESE



```
yjahn@dent:~/work/esedemo/esedemo
File Edit View Terminal Tabs Help
[yjahn@dent esedemo]$ source /home/e/e/local/bin/setup.sh
Setting up environment variables for ESE
[yjahn@dent esedemo]$ esedemo_rtos.sh
ESE demonstration setup for System Design with RTOS is ready
[yjahn@dent esedemo]$ ls
./                               jpeg_platform_timed_TLM/       mp3_platform_partial_srcs@
../                              jpeg_rtos_platform_partial.eds  mp3_platform_srcs/
jpeg_platform.eds               jpeg_rtos_platform_partial_srcs@ mp3_platform_timed_TLM/
jpeg_platform_functional_TLM/   jpeg_srcs@                     mp3_srcs@
jpeg_platform_partial.eds       mp3_platform.eds               test@
jpeg_platform_partial_srcs@     mp3_platform_functional_TLM/
jpeg_platform_srcs/            mp3_platform_partial.eds
[yjahn@dent esedemo]$ ese&
```

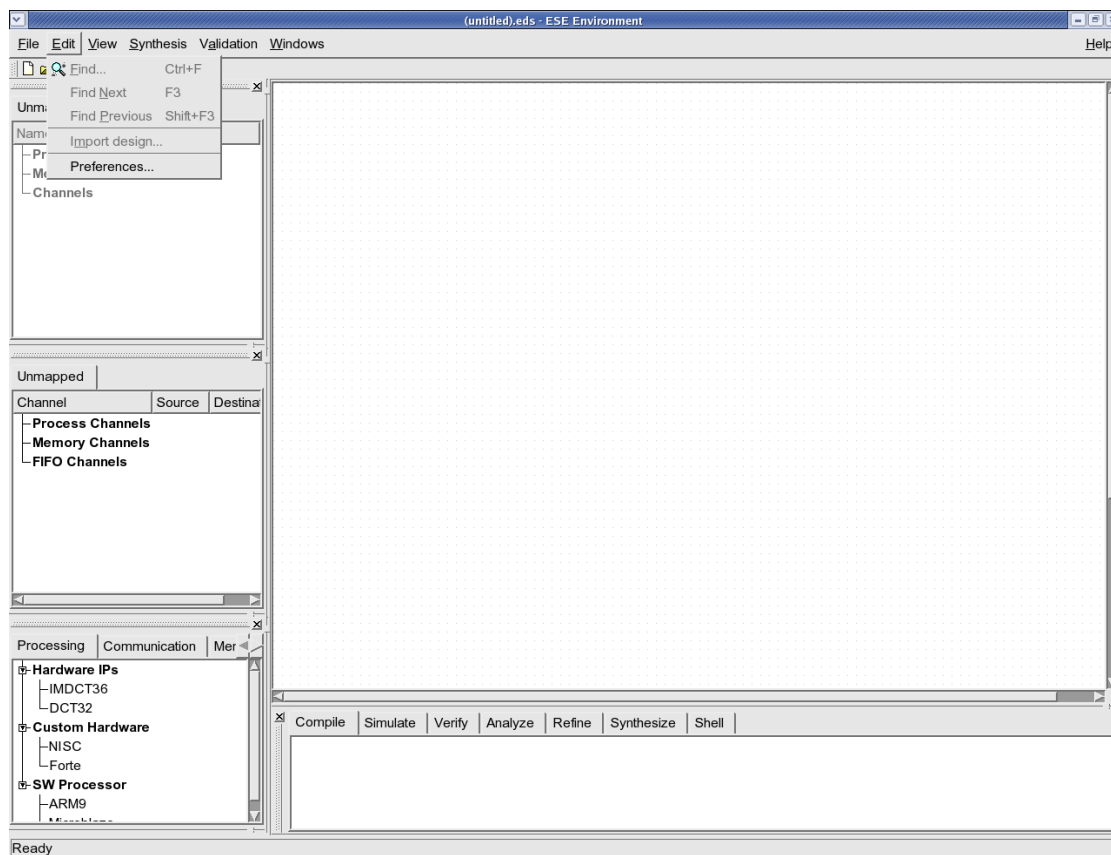
After running the "esedemo_mtd.sh" script, you will notice several files in the working directory. Some of these files will have a ".eds" extension. They are the ESE design files for the JPEG encoder design that we will be using for this demo. You may also see links to source directories. These point to the C code for the processes of the JPEG application. To launch the ESE GUI, simply run "ese" from your shell.

4.1.4. ESE GUI



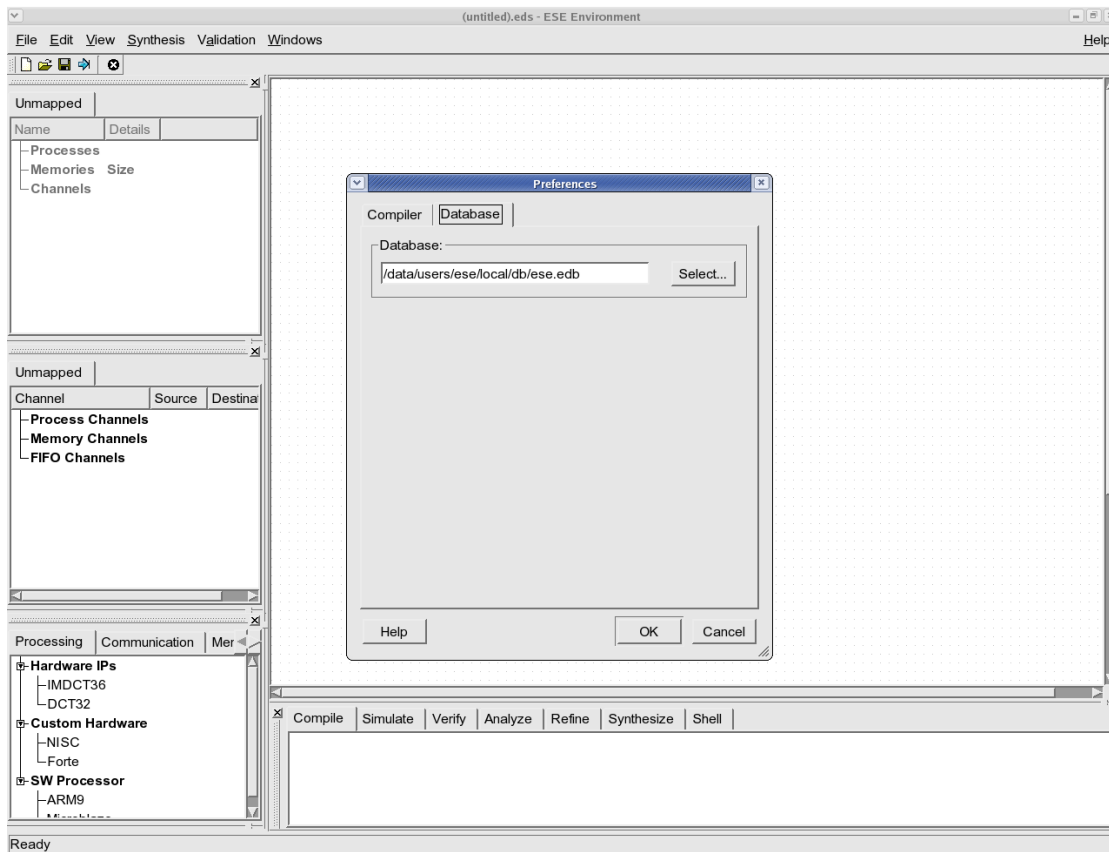
The ESE GUI should now appear as shown in the screenshot. The GUI has several menu items that we shall explore over this tutorial. It is divided into five windows. The top left window is the "PE" window. It organizes the various application processes mapped to PEs in the design. The mid-left window is the "Channel" window that organizes the various channels used for communication between the application processes. The tabs represent the physical communication links in the platform. The bottom left window is the "Database" window that organizes the PE, CE, memory and RTOS model. The top right window is the "Platform Canvas" on which the platform architecture is edited graphically. The bottom right window is the "Logging" window that logs the messages from various ESE tools.

4.1.5. Editing Database Preferences



Before creating a new design, we must ensure that the components needed for our JPEG platform are accessible by the GUI. To do so, we edit the database preferences by selecting **Edit**→**Preferences** from the menu bar.

4.1.6. Select Database File

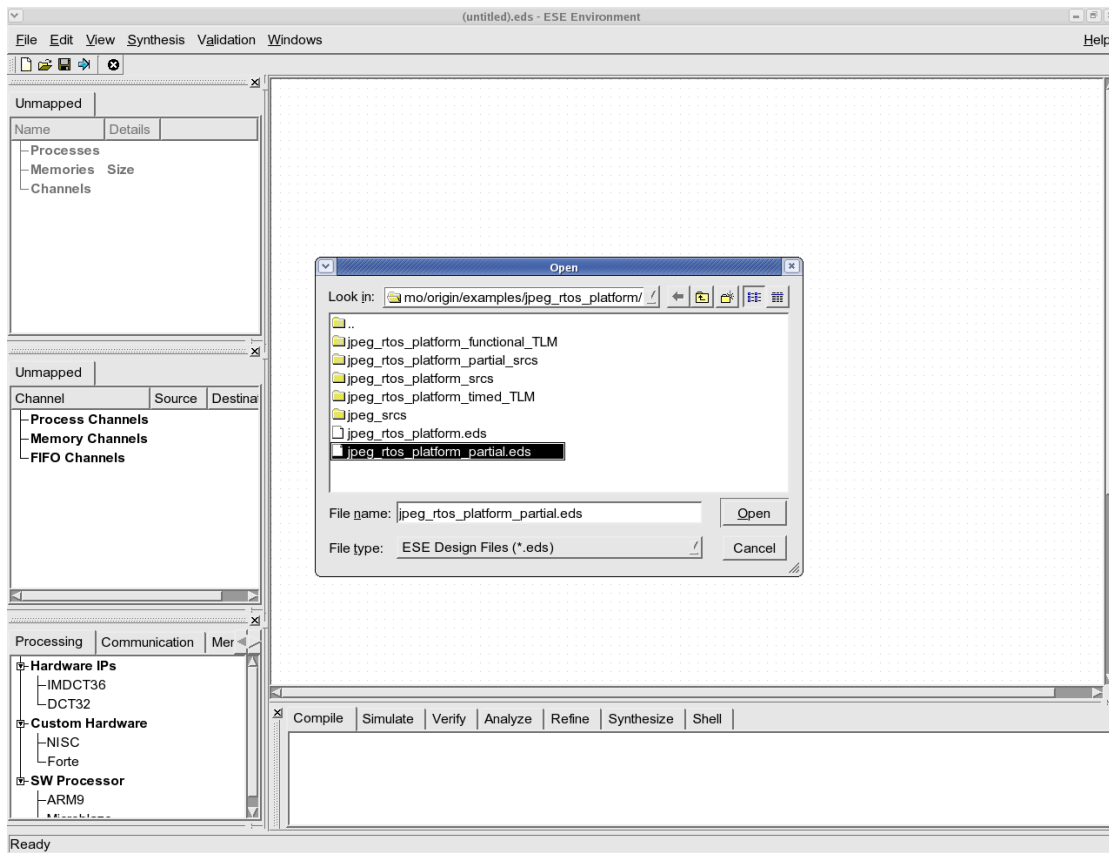


In the **Preferences** dialog, select the tab for **Database**. This will allow the user to browse for the database file that has a ".edb" extension. The database file needed for the JPEG demonstration already comes with the ESE installation. Typically, this file will be called "ease.edb" and will be located at "/data/users/ease/local/db/ease.edb." If the selection is not already there, please browse for the file and press **OK**. All the elements should now be visible in the database window, if they weren't already.

4.2. Platform Creation

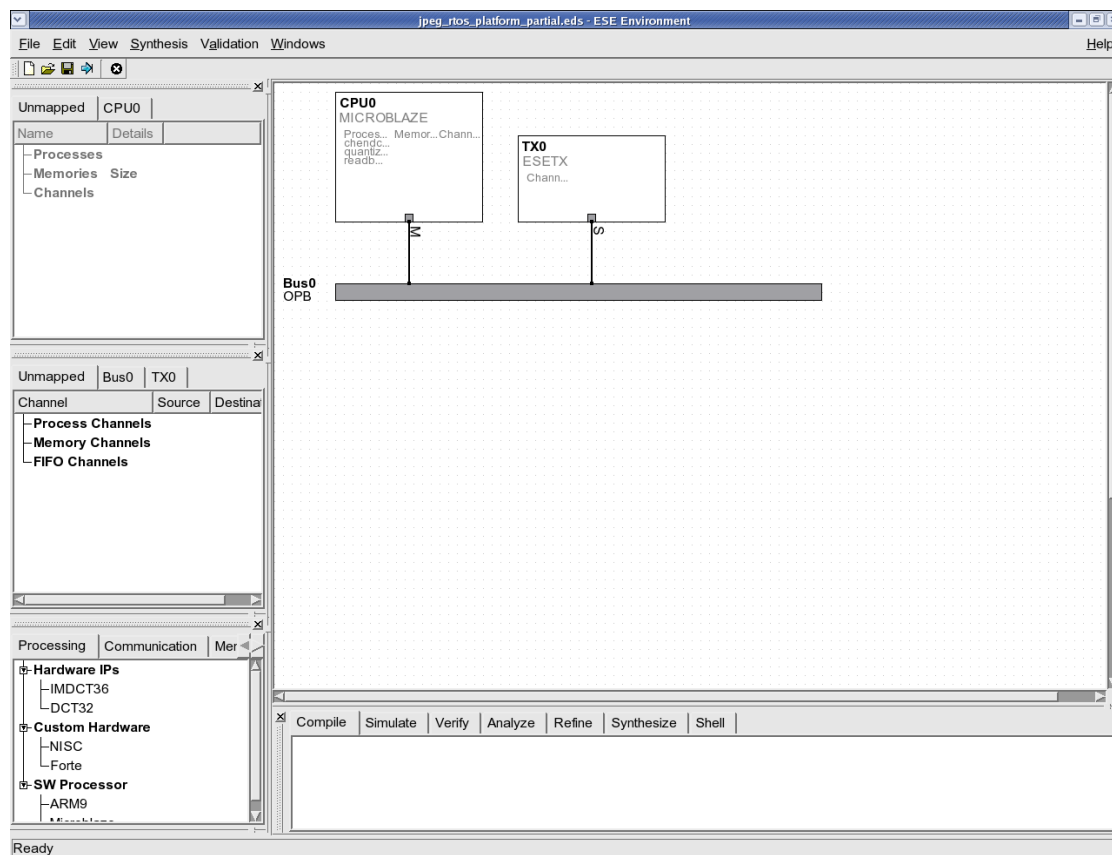
We will start by loading the design of the JPEG encoder into ESE. As mentioned earlier, we will start with a partial platform consisting of one multi-threaded Microblaze processor and one OPB. The processor carries the application code for three processes in the JPEG encoder. One Microblaze processor for "zigzag" and "huffencode" processes will be added to the platform and then the two processes will be multi-threaded by adding a RTOS model. In this section, we will show how to use the database and platform editor canvas and how to upgrade a platform in ESE.

4.2.1. Open Partial Design



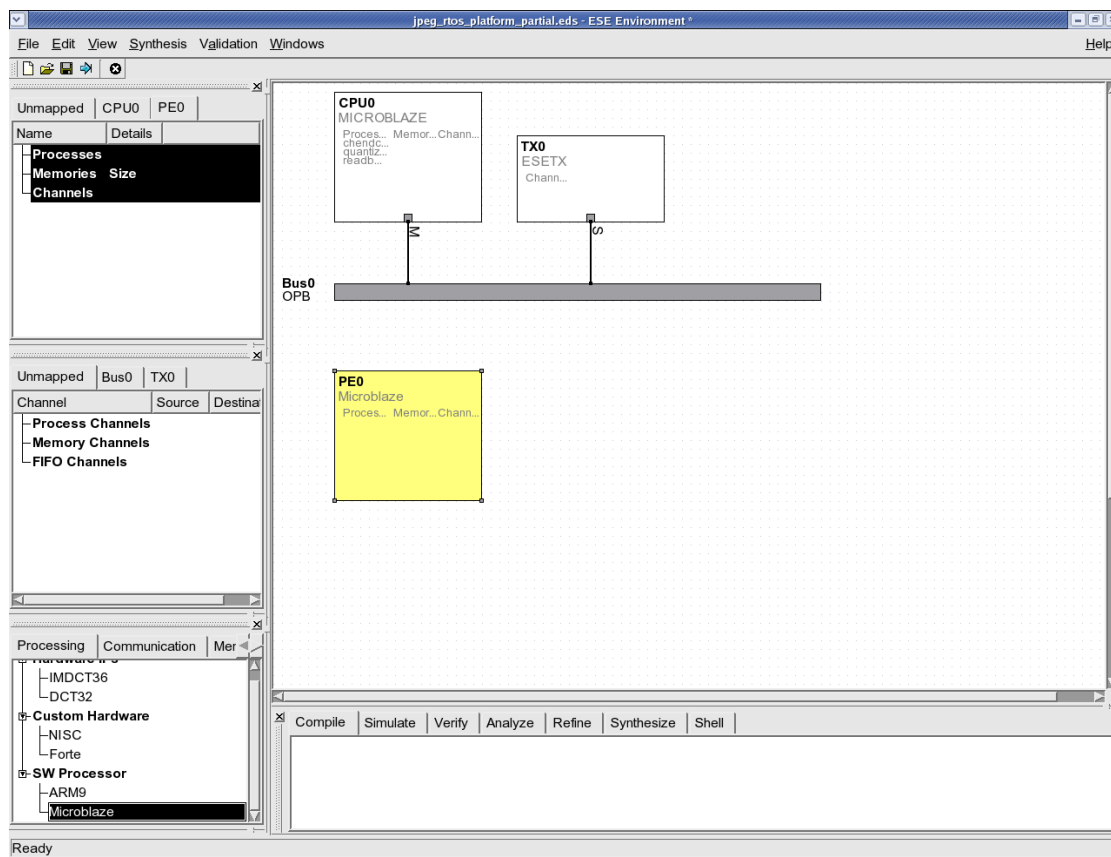
We begin by adding the already created partial design. The ESE designs are stored in XML based files with the extension ".eds." Select **File**→**Open** from the menu bar. Browse into the demo working directory and select "jpeg_mtd_platform_partial.eds." This is the design with the partial design example including RTOS. Press **Open** to open the design.

4.2.2. View Partial Design



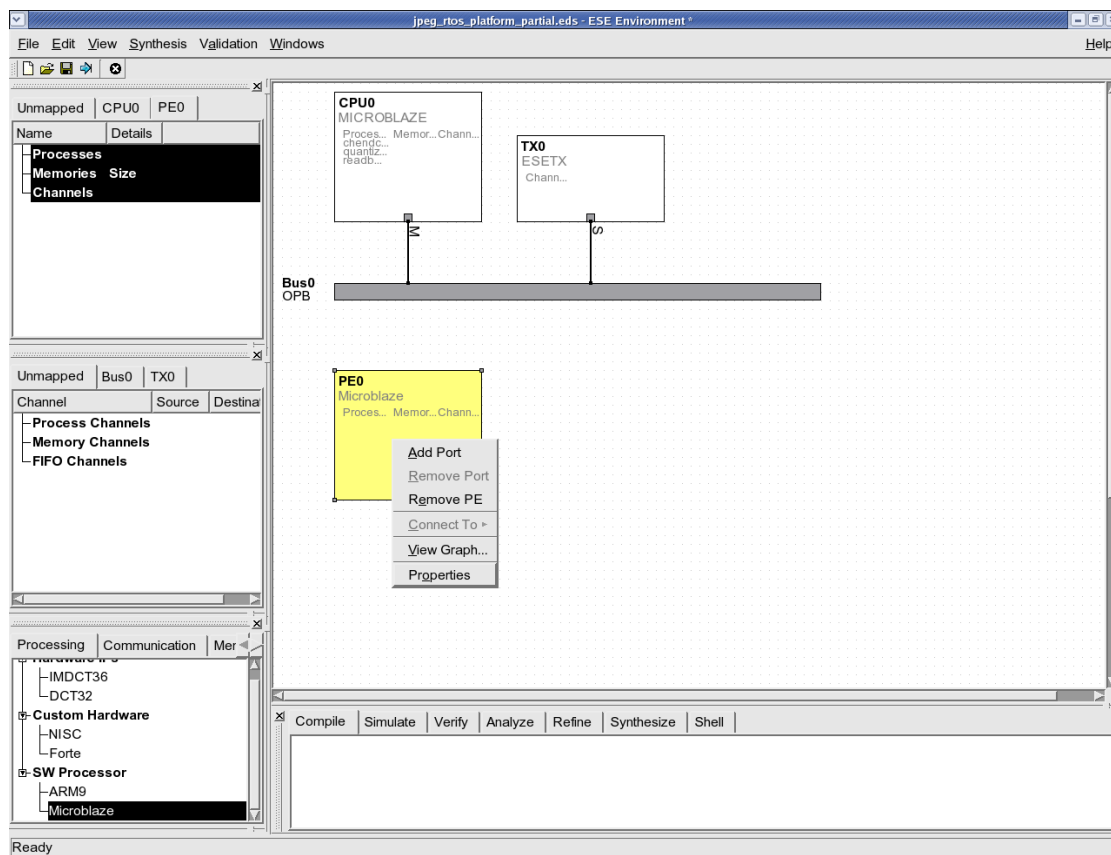
The partial platform will appear in the canvas as shown in the above screenshot. We can see one multi-threaded Microblaze processors **CPU0** in the platform. The processor is connected via the Open Peripheral Bus (OPB). There are two local FIFO channels in this partial design. Each process has its own process port and the process port is connected through the FIFO channel. For example, as shown in PE window, CPU0 has three processes, "readbmp", "chendct" and "huffencode". Among them, the "readbmp" process has a process port which is for sending data from "readbmp" to "chendct". And the process port is connected to a local FIFO channel named "r2c" as shown at the bottom in PE window. Since this channel is for the intra-process communication in a processor, the channel is located to a local memory and is shown in the PE window not the Channel window which only shows the channels for the inter-process communication. Note that the processor is both connected as "Master" as indicated by an "M" at the connecting port. Since bus master cannot communicate directly over the bus, we provide a transducer (Tx0) which consists of a FIFO controller and FIFO memories. It acts as a shared memory for data transfer between CPU0 and CPU1.

4.2.3. Add Processing Element



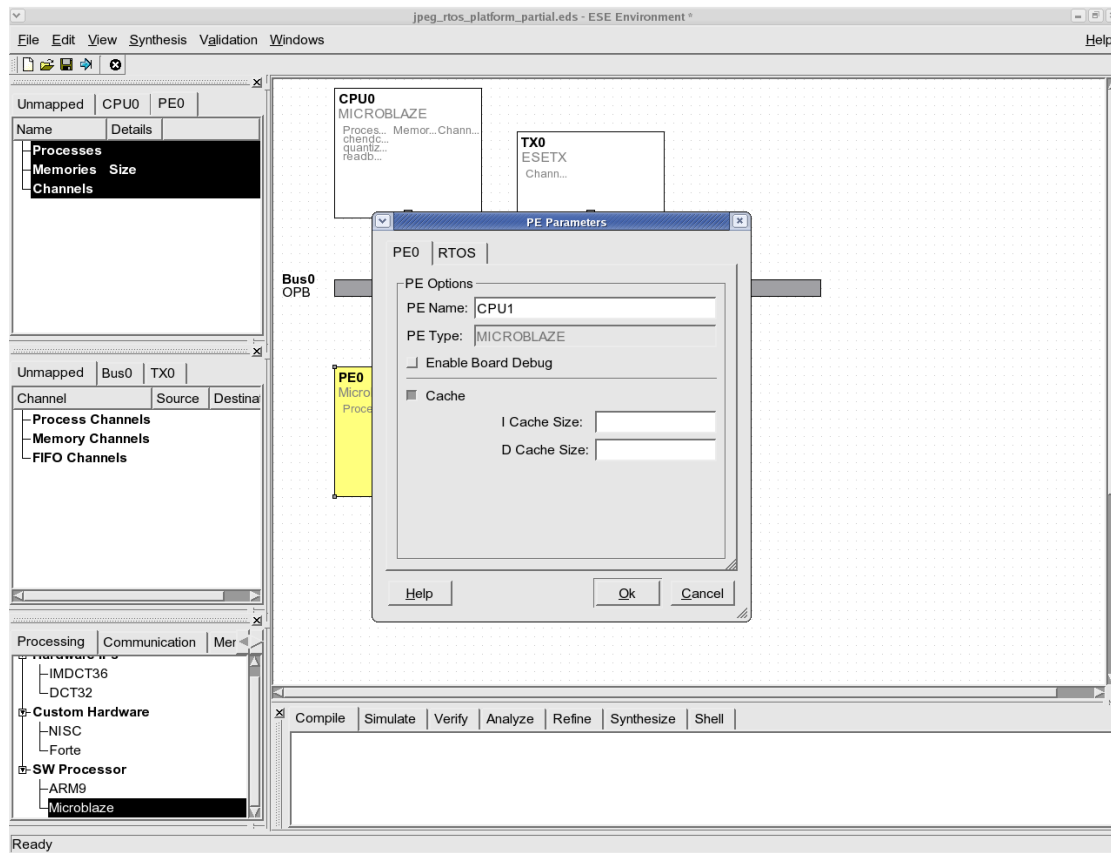
Adding a new PE to the platform is very easy. Browse the database under the Processing tab and select Microblaze. Now drag and drop the selection into the platform canvas. The new PE of type "Microblaze" will be added to the platform!.

4.2.4. View PE Properties



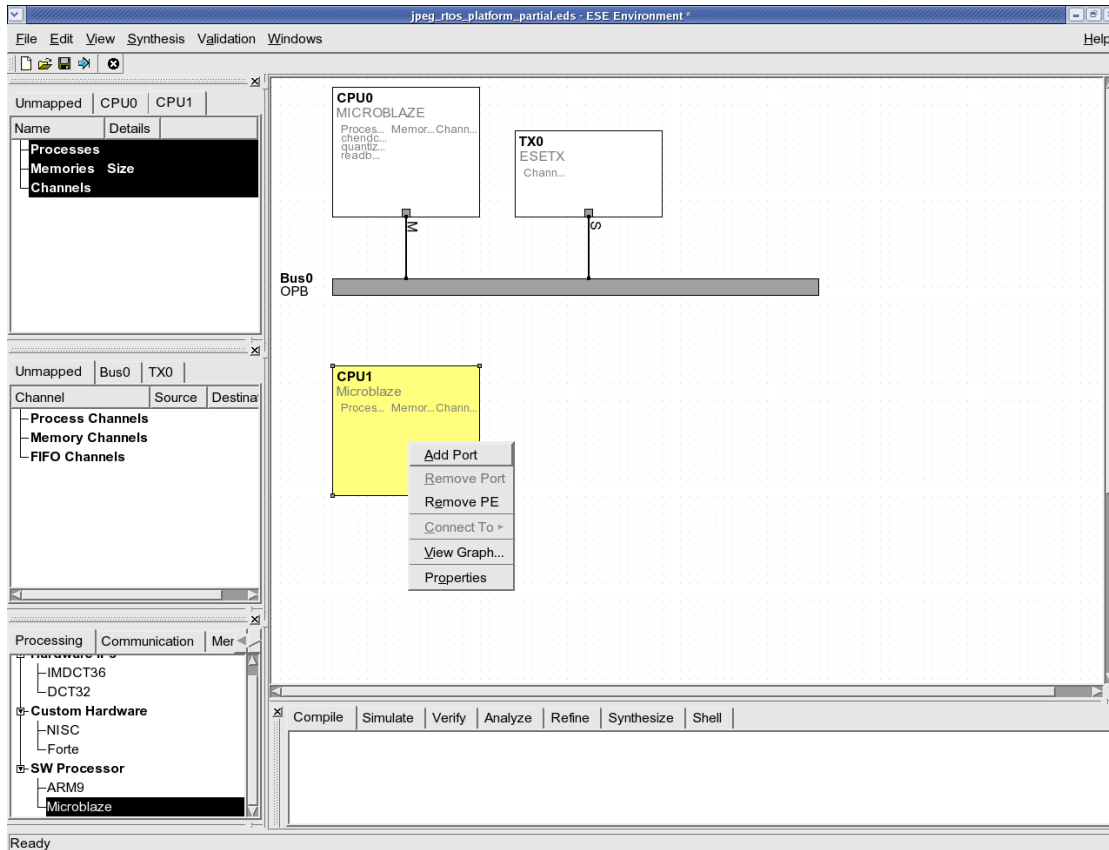
After the drag-drop, the user will find the new PE called **PE0** in the platform. This is the PE that will host the "zigzag" and "huffencode" processes in the design. We start by providing an appropriate name to the new PE to be consistent with the rest of the design. To do so, right click on the PE0 box and select **Properties**.

4.2.5. Assign New Name to PE



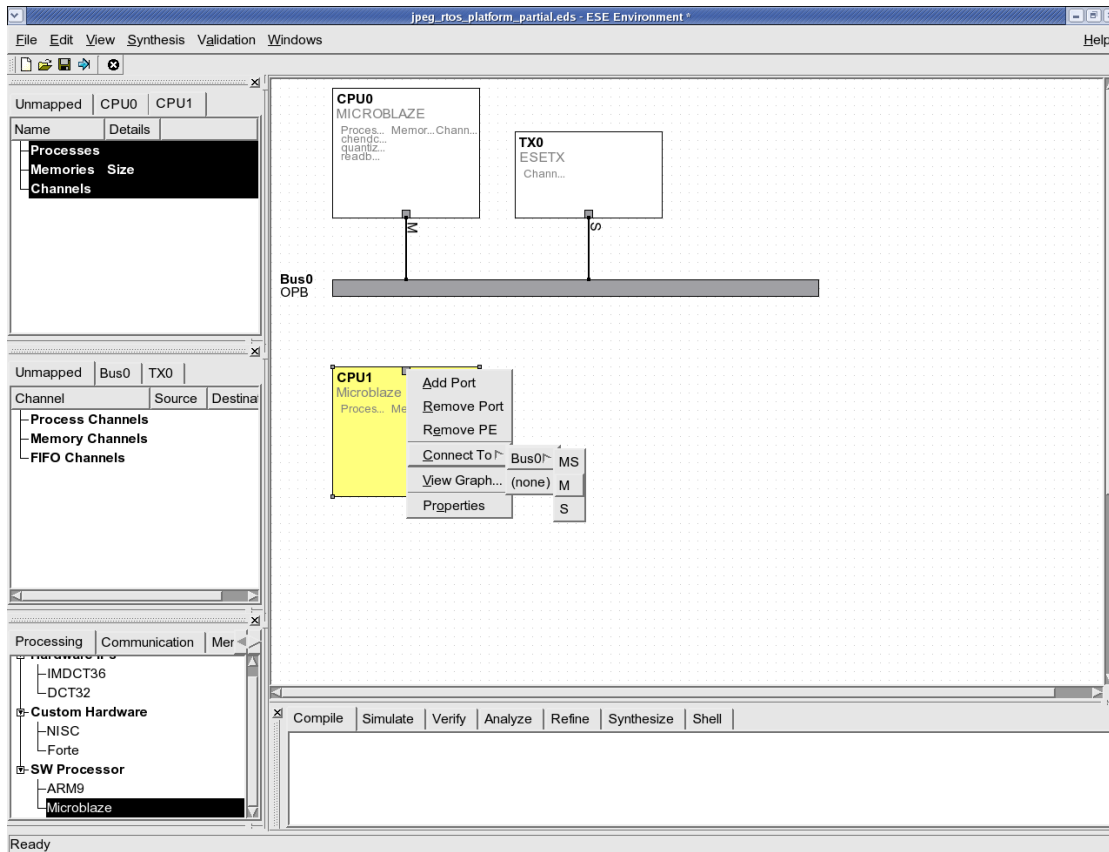
In the properties dialog, change the PE name of the PE0 to "CPU1" to be consistent with the other PE names.

4.2.6. Add Port to PE



The new PEs, CPU1 is not yet connected to the rest of the design. Since the application processes meant to execute on the PE will need communication with processes on other processor, we must physically connect CPU1 to the shared OPB bus in the platform. For this physical connection, a port is required for CPU1. To add the port, simply right-click on the CPU1 box and select **Add Port**.

4.2.7. Connect PE to Bus

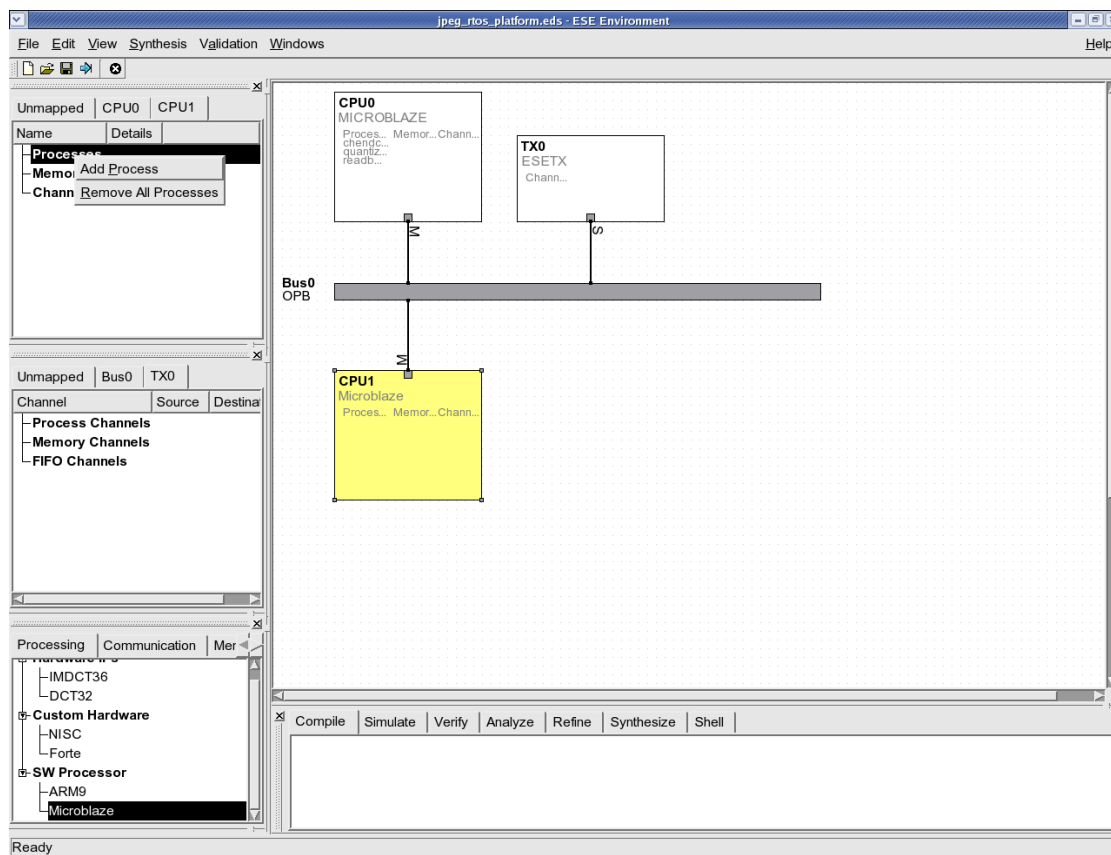


The created port must be connected to the OPB bus to be able to communicate with the rest of the system. Note that CPU1 is Microblaze core. This means that it can only connect to the OPB bus as a Master. To connect CPU1, right-click on the port and select **Connect To** → **OPB0** → **M** from the menu choice. This will create the bus connection and complete the platform design step. Next, we will look at application input and its mapping to the created platform.

4.3. Mapping Application to Platform

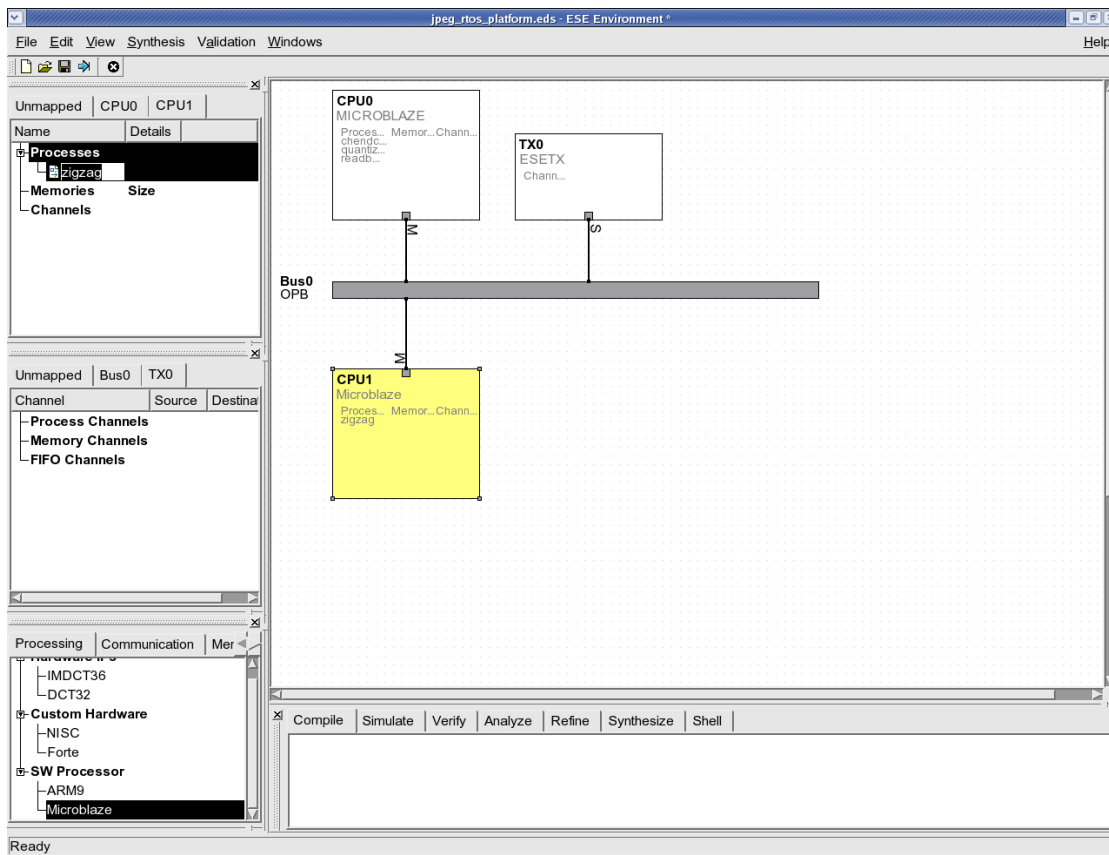
The application input model for ESE is C/C++ processes communicating through FIFO channels. Since most legacy application is written in C, this is an advantage over other forms of input styles or languages. For communication, the user does not need to write any SystemC channel code. ESE provides very simple APIs for inter-process communication as we will see in this section.

4.3.1. Add Application Process



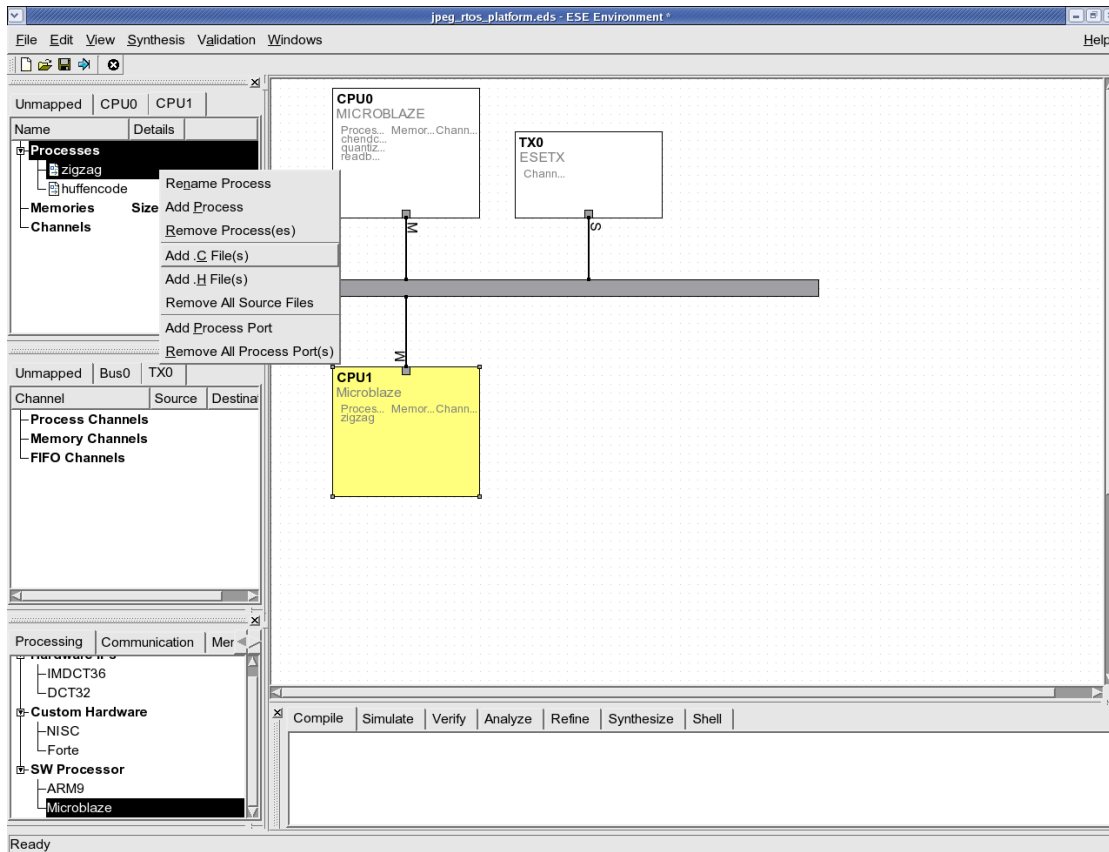
The PE window on the top left corner organizes the processes mapped to the various PEs in the design. To add a new process executing on CPU1, change to the **CPU1** tab. Then right-click and select **Add Process**. This will create a new process with a default name.

4.3.2. Assign Name to New Process



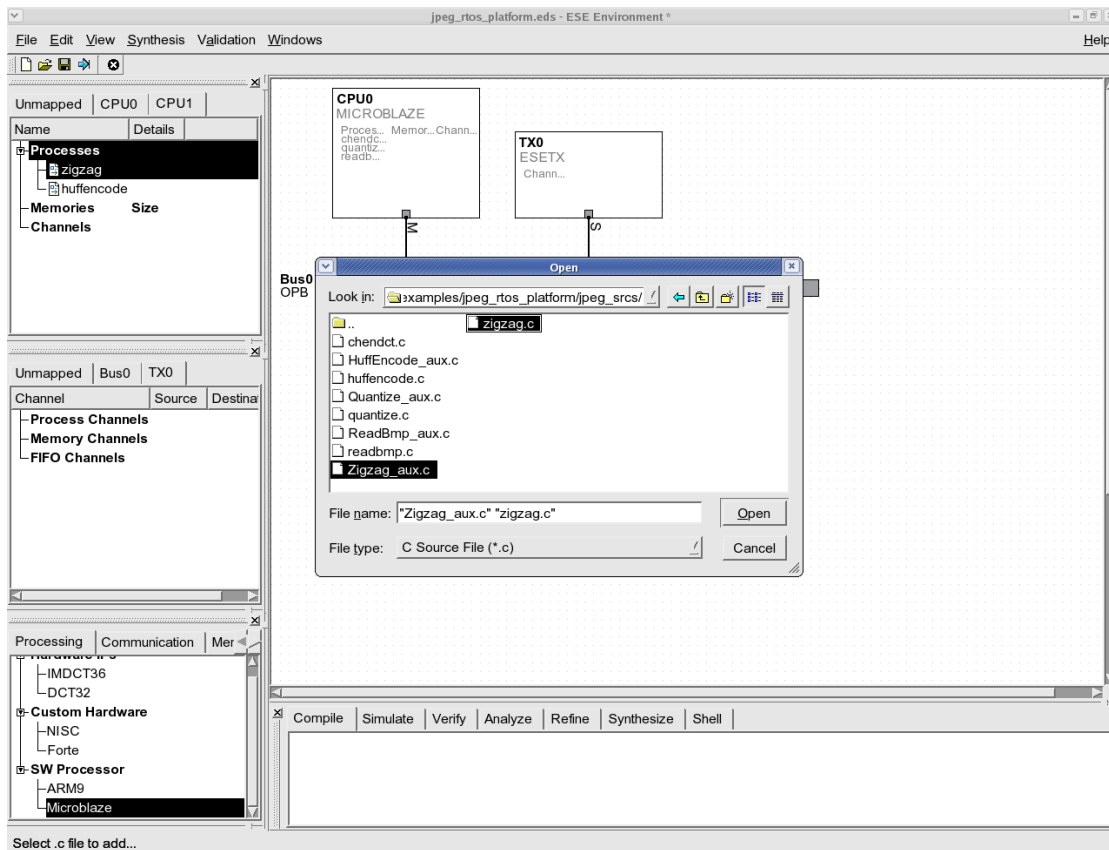
Change the name of the new process to "zigzag". This is the process for the zigzag scan in the JPEG encoder application. Please ensure that the process is named correctly since there exist references to it in the existing partial design. If the process is not named as suggested, the generated models will not compile. Create one more process for its name to be "huffencoder" in the same way.

4.3.3. Add C Source File



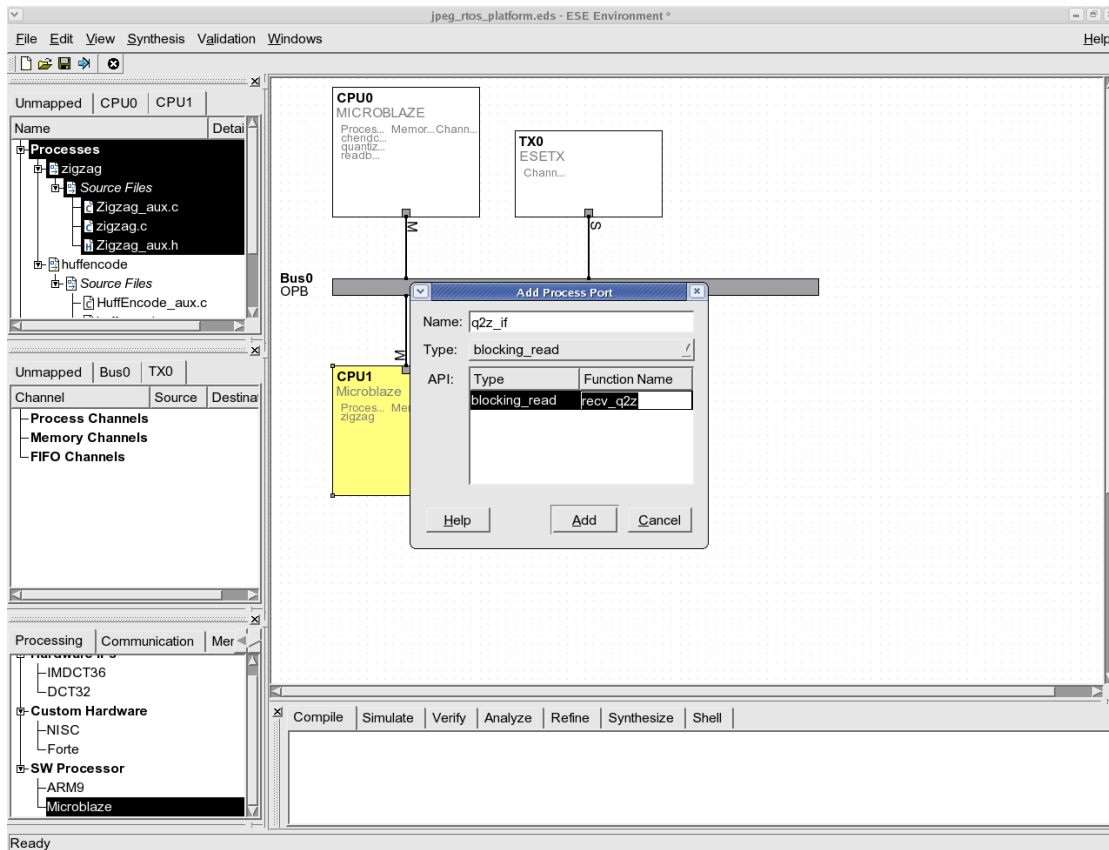
The process added in the last step is only symbolic. The user must associate the actual C/C++ code with it for the models to be functionally correct. In this case, we add C code by right-clicking on the process name in the PE window and selecting **Add .C File** for adding ".c" files. And we can also add ".h" files by selecting **Add .H File**. This will open the file browser.

4.3.4. Select C Source File



Go to the demo directory and follow the symbolic link to "jpeg_srcs". For the "zigzag" process, select two ".c" files, "zigzag.c" and "Zigzag_aux.c", and one ".h" file, "Zigzag_aux.h", and then click Open. In the same way, for the "huffencode" process, select three files, "HuffEncode_aux.c", "huffencode.c", and "HuffEncode_aux.h". The files will be added under the new process in the PE window.

4.3.5. Add Process Ports

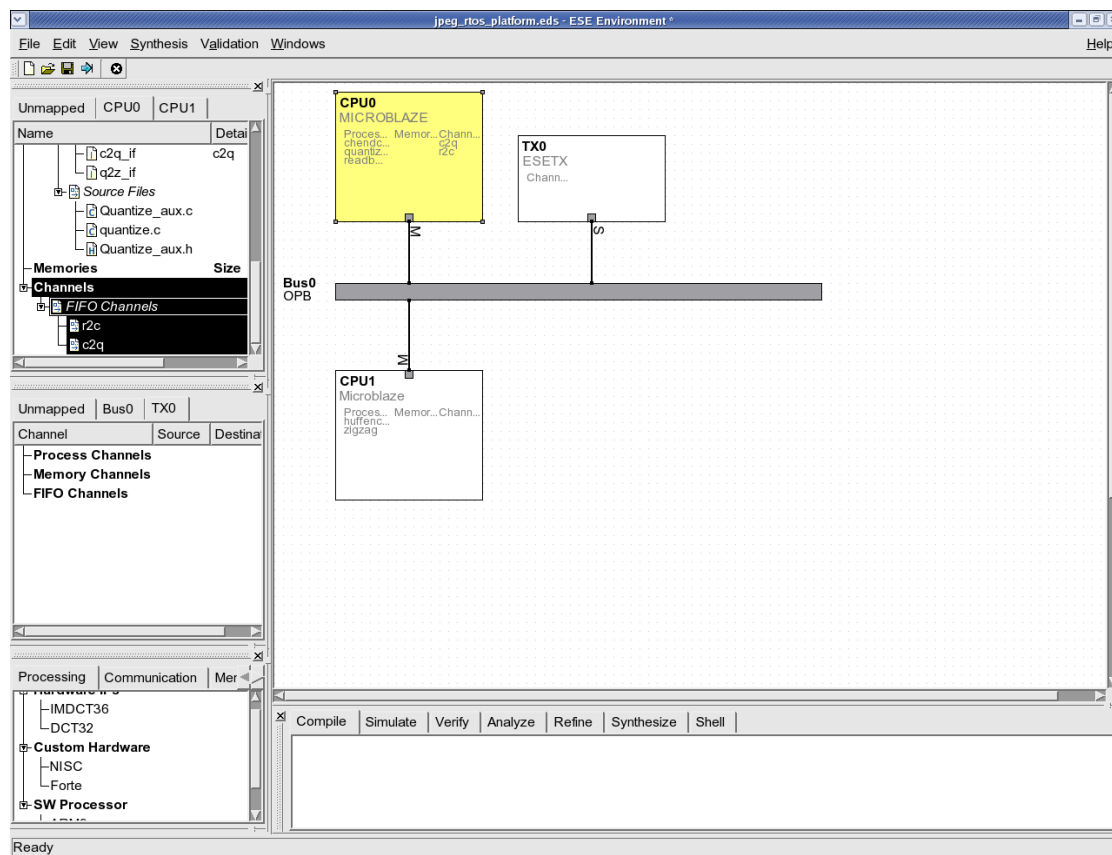


After, the C code for the process is added, we need to add the application level communication to the design. First of all, we need to add the process port for each process, which will be connected to a channel for data transfer to another process. To add the process port for the new process, click on the new process and select the **Add Process Port**. This will open the window to add the process port. We can create any name for the process port and select the type of it. There are ten possible types. We can categorize them into three kinds. The "Send", "Receive", and "Send/Receive" are used for double handshake channels. The "Read", "Write", and "Read/Write" are used for shared memory. And the others are for the FIFO channels. Finally, we need to assign its function name to be what is actually used in C code. Please ensure that the function name is the same as that used in C code. If the name is not correct, the generated models will not compile.

The "zigzag" process has two process ports. One is for receiving data from "quantize" process and the other is for sending data to "huffencode" process. Assign the process port name to be "q2z_if" for the former and "z2h_if" for the latter. Since we are us-

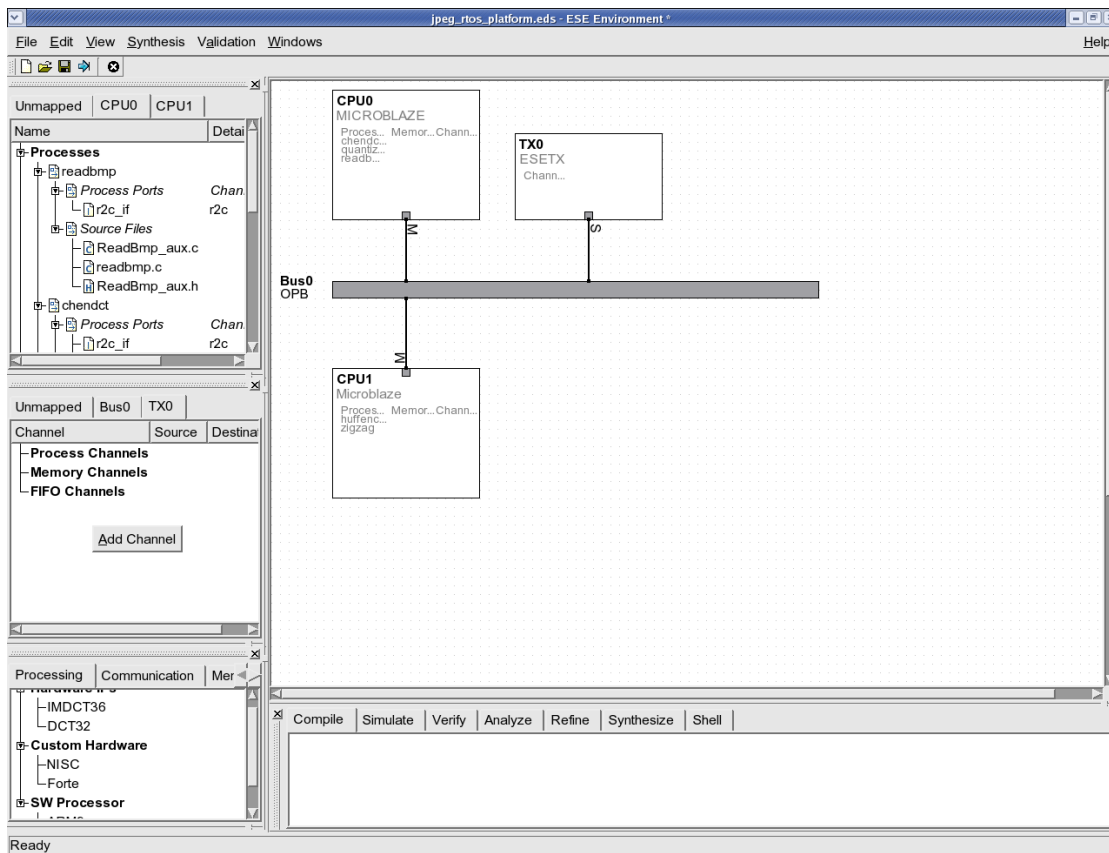
ing FIFO channels, select the type to be "blocking_read" for the former and "blocking_write" for the latter, respectively. Also, assign the function name to be "recv_q2z" and "send_z2h", respectively. The "huffencode" process has only one process port which is for receiving data from "zigzag". Its process port name is "z2h_if" and its function name is "recv_z2h". Please add all the process ports for all the new processes using the given names.

4.3.6. View Application Channels



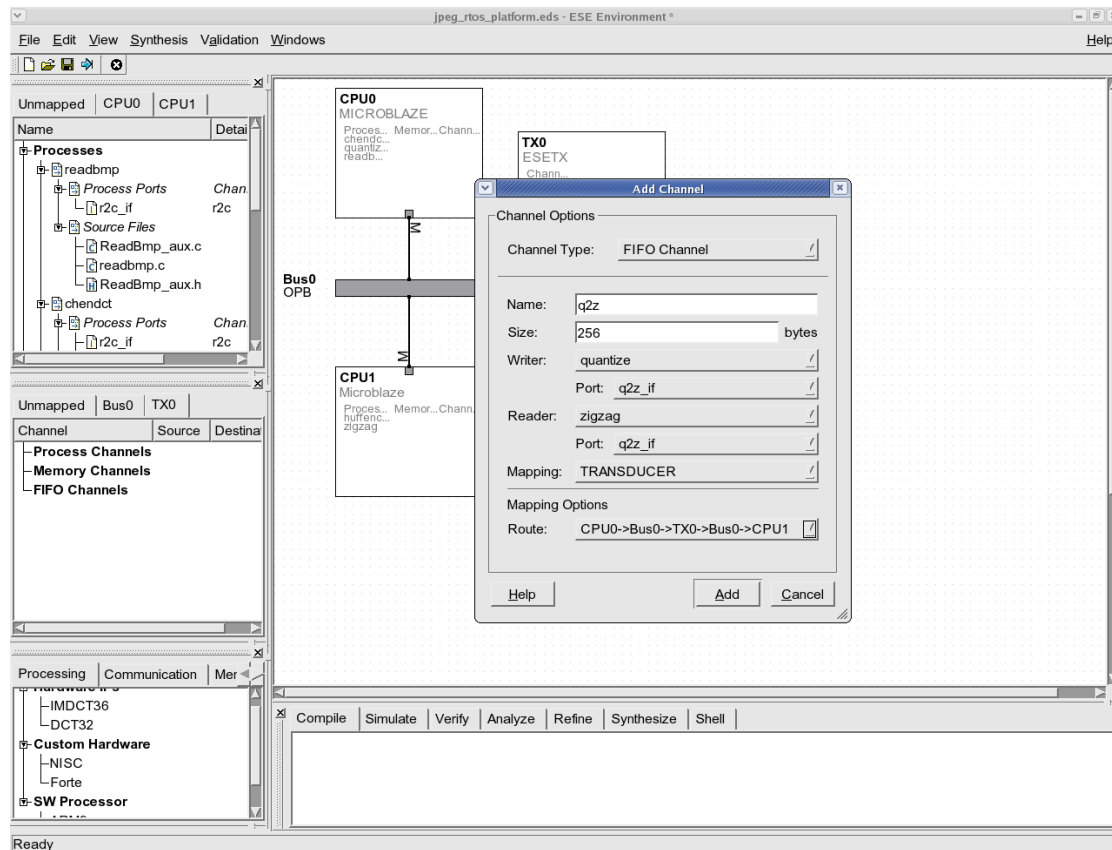
After, the process port for the process is added, we need to create the channels for the communication between processes. To view the existing channels, click on the **CPU0** tab in PE window. This will display the existing FIFO channels between processes in CPU0, including the source and destination names as well as the route used to implement the channel in the communication platform. In the partial platform, there exist only local FIFO channels for intra-process communication. All the channels in ESE can be uni-directional or bi-directional channels. If the user clicks on a PE in the platform canvas, all the channels originating or terminating at the PE will be selected. All other PEs that the clicked PE communicates with will be highlighted in light yellow. All physical connections, including buses and transducers used by the PE for communication will be highlighted in green.

4.3.7. Add New Application Channel



Please click on CPU1 and see the Channel window. We can know that they are currently not connected at the application level to any other PE. Since we need communication between the "quantize" process in CPU0 and the "zigzag" process in CPU1, we will add the application level channels, by right-clicking in the channel window and selecting Add Channel. This will pop up the channel wizard for adding application level channels.

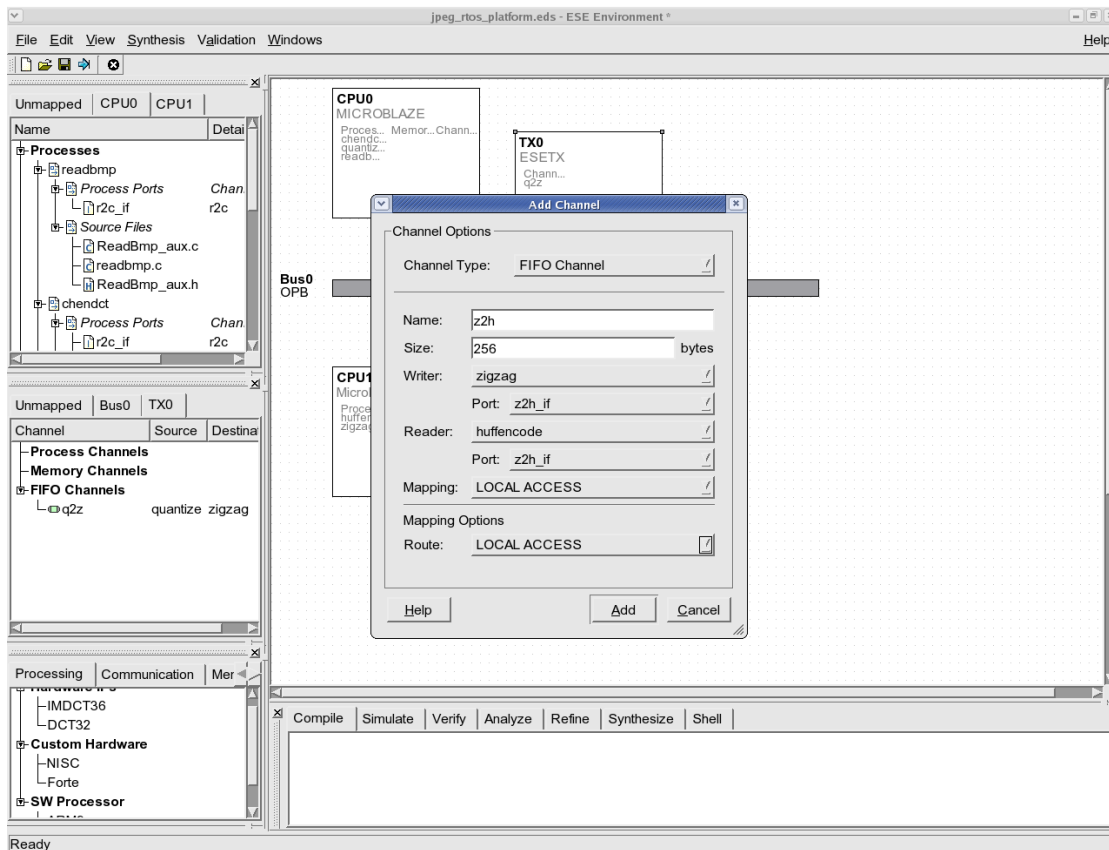
4.3.8. Channel Wizard for Inter-Process Communication



In the channel wizard dialog, we first need to select the channel type. Choose "FIFO Channel" since we are using FIFOs. Then, assign the channel name to be "q2z" for consistency with existing channels and also assign the FIFO size to be "256" bytes since the processes send/receive an 64-array integer data each other. Next, since the process will send data in one way from "quantize" to "zigzag", select "Unidirectional" using the pull down menu. Then, use the pull down menu to select the first communicating process as "quantize" and also use the next pull down menu to select the process port as "q2z_if". In the same way, select the other communicating process as "zigzag" and select the process port as "q2z_if". Next, select the mapping to be "TRANSDUCER" since we are using a transducer for the inter-process communication. Once the communicating processes and process ports are decided, ESE automatically filters all the possible physical routes on the platform that can implement the channels. For this example, it shows that there is only one route for each direction that goes over the OPB bus from the sender PE to the transducer Tx0 and back to the receiver PE on the OPB bus. The route goes through the transducer because all PEs in the platform are connected as masters, which does not allow direct communication. The slave interface of Tx0, thus makes the routing possible.

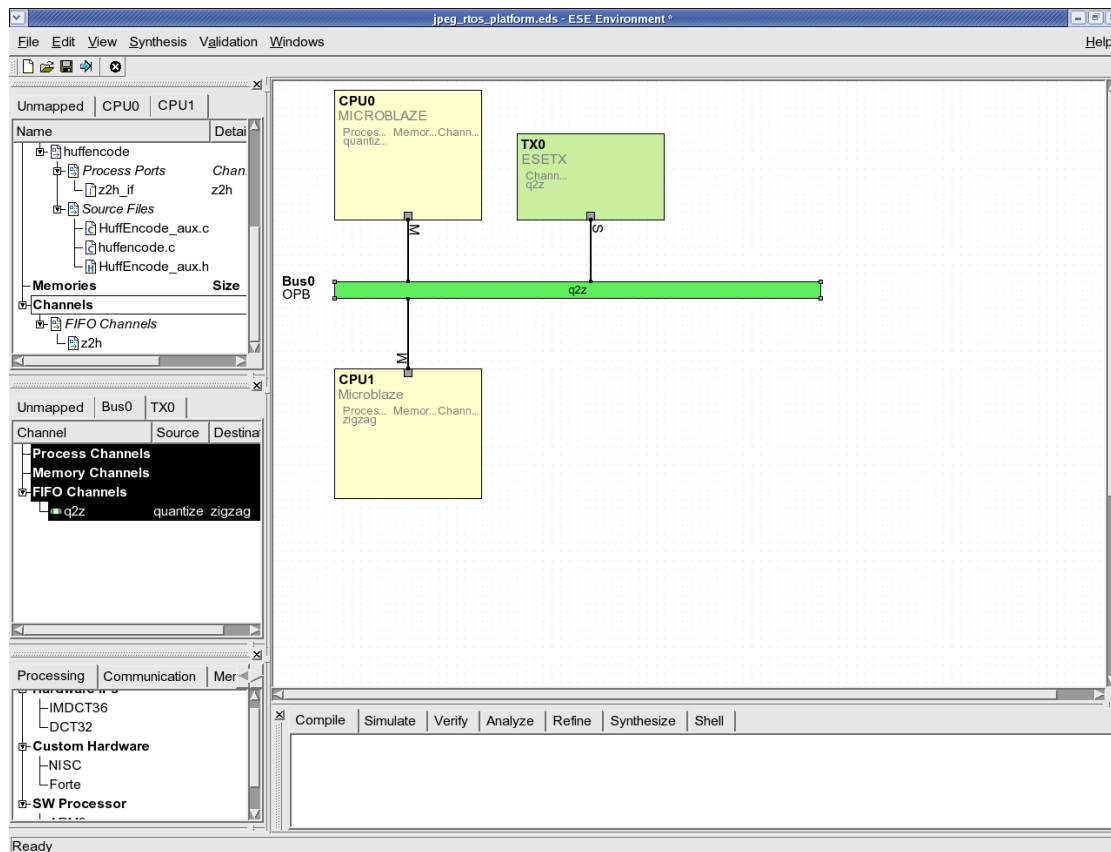
Click Add to add the channel.

4.3.9. Channel Wizard for Intra-Process Communication



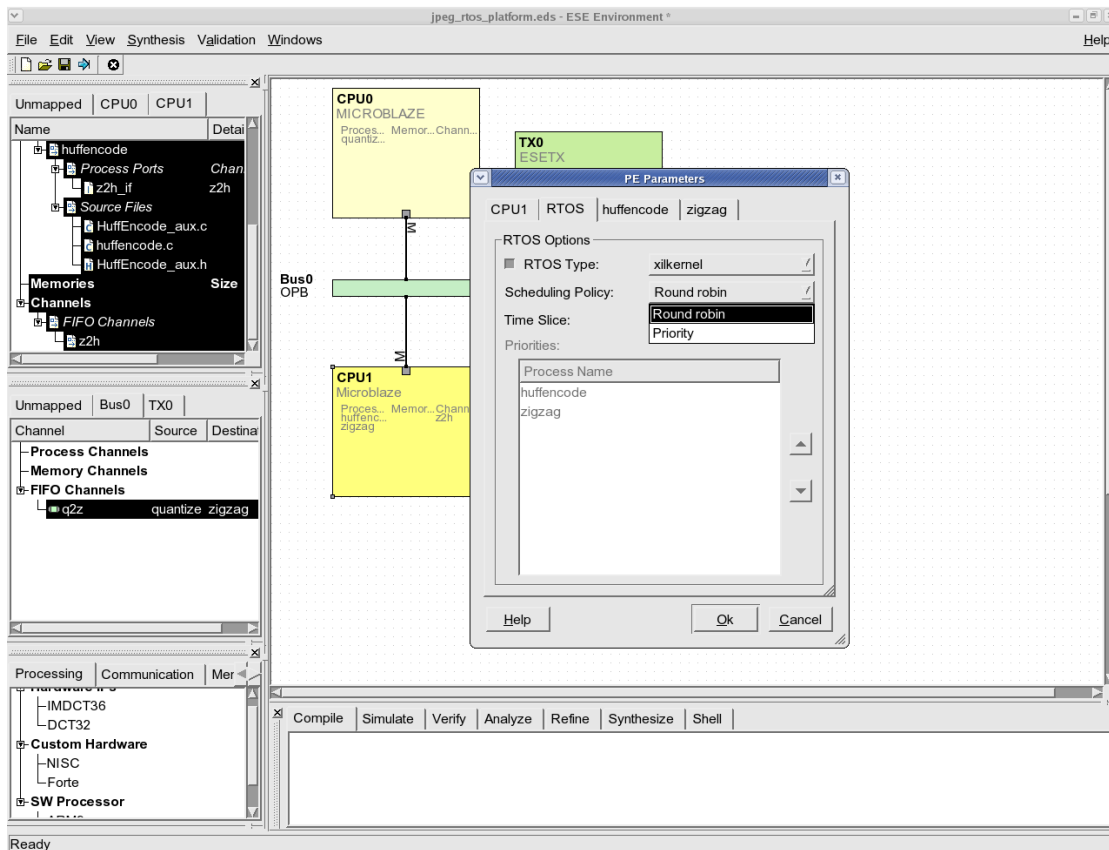
In order to add the channel for the communication between "zigzag" and "huffencode", right-click in the Channel window and select **Add Channel**. In the channel wizard dialog, assign the channel name to be "z2h". From the next step, everything is the same as the previous section except the channel mapping and route. Since it is for the intra-communication, the mapping and route will be automatically set to "LOCAL ACCESS" as shown in the above screenshot. Select the default mapping and route. Finally, click **Add** to add the channel.

4.3.10. View New Channel Communication



The newly created channels will now be visible in the channel window under the Tx0 tab. Once the channels are selected, the communicating PEs will be highlighted. This shows that the new PE, CPU1 is now "connected" with the rest of the system on an application level.

4.3.11. Add RTOS



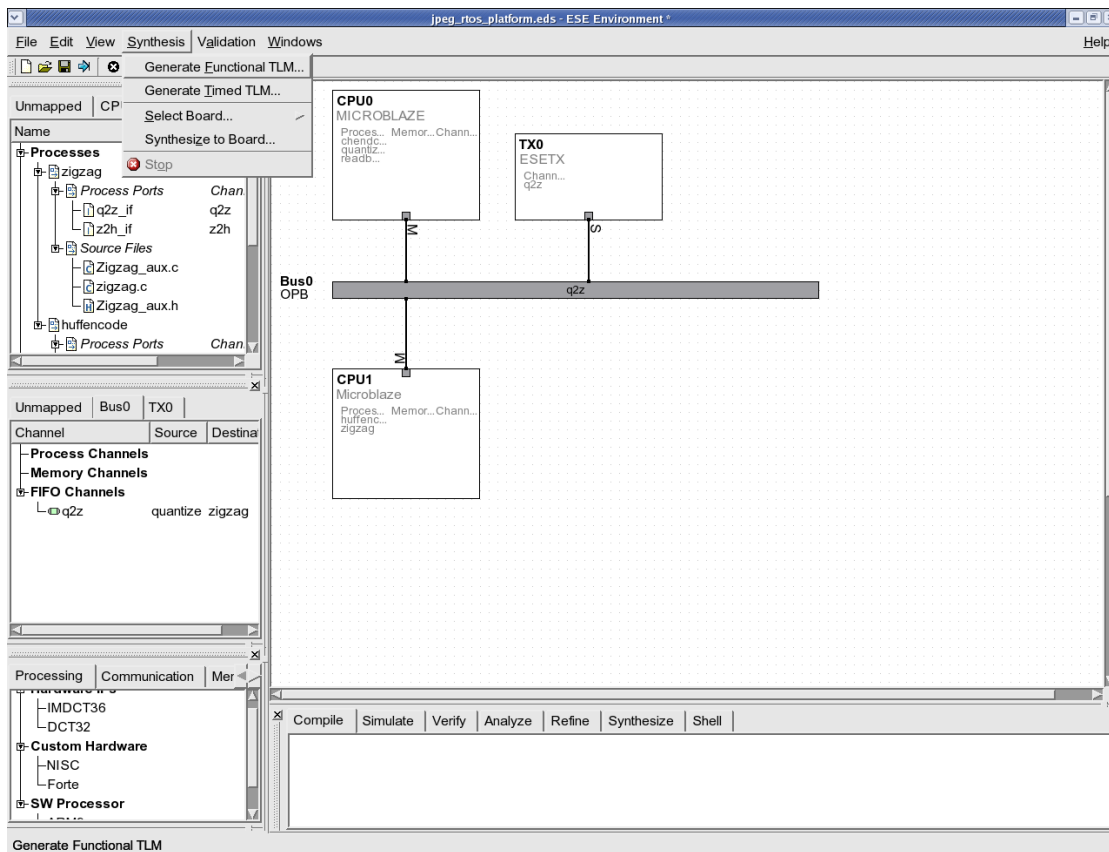
As mentioned before, since CPU1 has more than one process and the processes should be multi-threaded, we need a RTOS model to control the execution of the processes. To add the RTOS model, right-click on the PE box and select **Properties**. Then, enable RTOS by clicking the small square box. There are two scheduling policies, "Round-Robin" and "Priority". Select one from the two scheduling policies. If you select "Priority" scheduling, then the priority of the processes in the PE will be shown in order. Users can change the priority by using the arrow buttons located at the right side.

4.4. Generating Functional and Timed TLMs

The previous steps complete the platform and application input that is necessary for generation of TLMs. We will show generation of two types of SystemC TLMs. The first one is called the "Functional TLM" because it is used for the validation of design functionality only. It is completely un-timed and simulates the design based on causal dependency only. A universal bus channel is used to model the system bus and the mapping of channels on the bus.

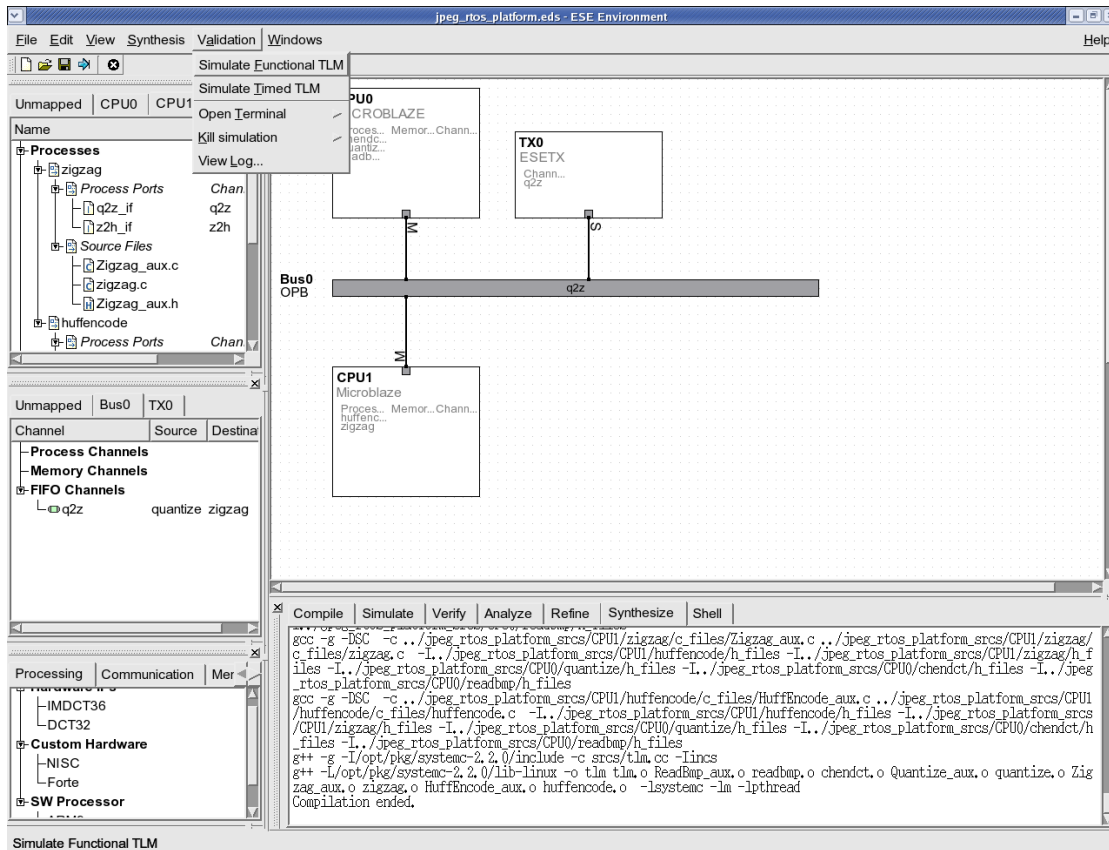
The second TLM is called the "Timed TLM" and is used for performance estimation of the design. It relies on timing data models of PEs and Buses that are available in the ESE database. The data models are used by our estimation and annotation technique to apply "wait" statements in the application C code. The technique is retargetable and applicable to processors as well as HW IPs. A retargetable bus timing annotation modifies the bus channel to apply "wait" statements for inter-PE communication.

4.4.1. Generate Functional TLM



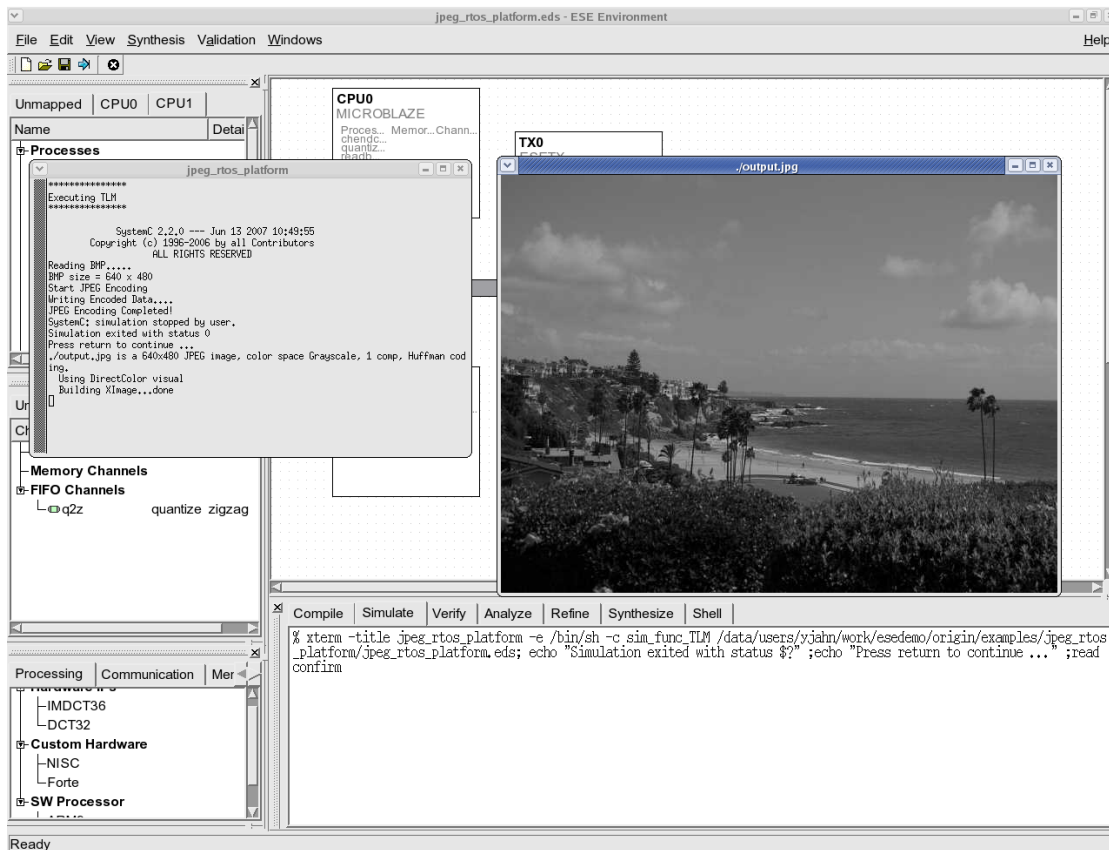
After the platform and application input is complete, the functional TLM can be generated automatically by selecting **Synthesis**→**Generate Functional TLM** from the menu bar. This will generate the SystemC code needed for platform modeling, including PEs, buses and transducers. The generated code is then compiled natively along with the C application code and linked to the SystemC libraries to produce a single binary. This process can be viewed in the log window.

4.4.2. Simulate Functional TLM



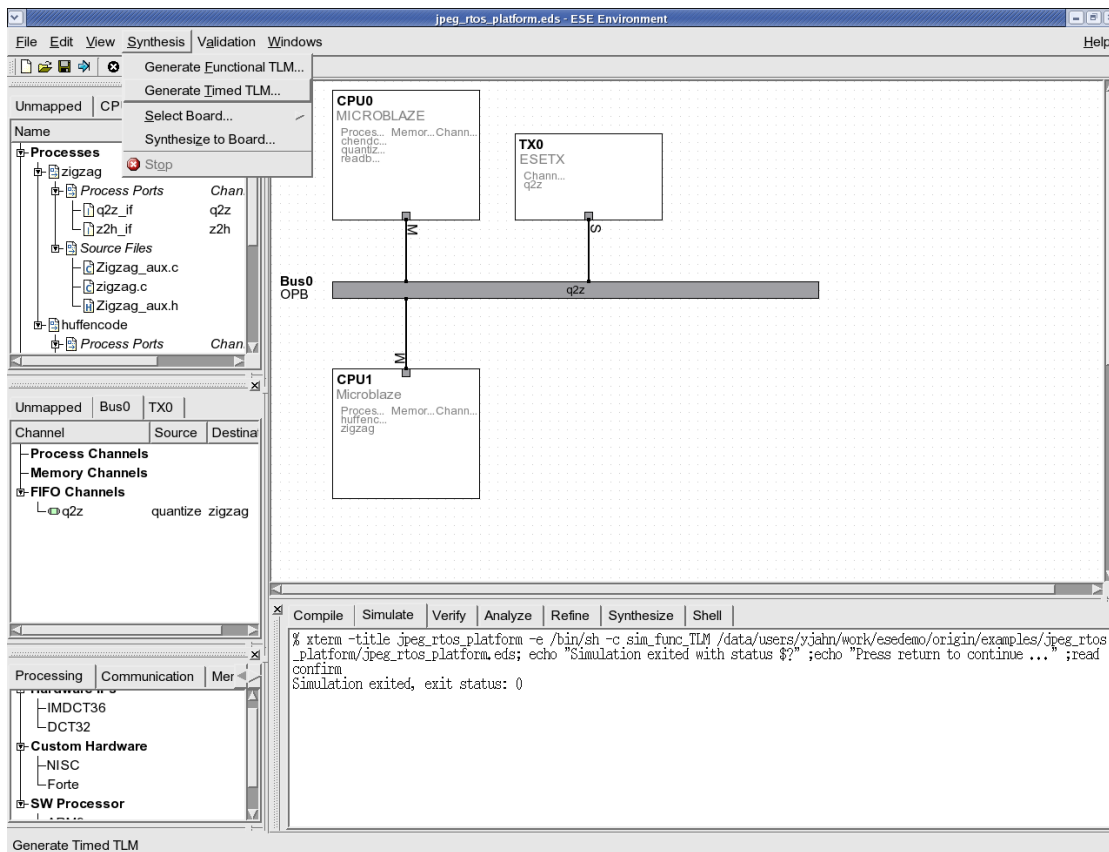
Once the compilation has completed, the generated TLM can be executed from the GUI by selecting Validation—>Simulate Functional TLM from the menu bar.

4.4.3. View Functional Simulation Results



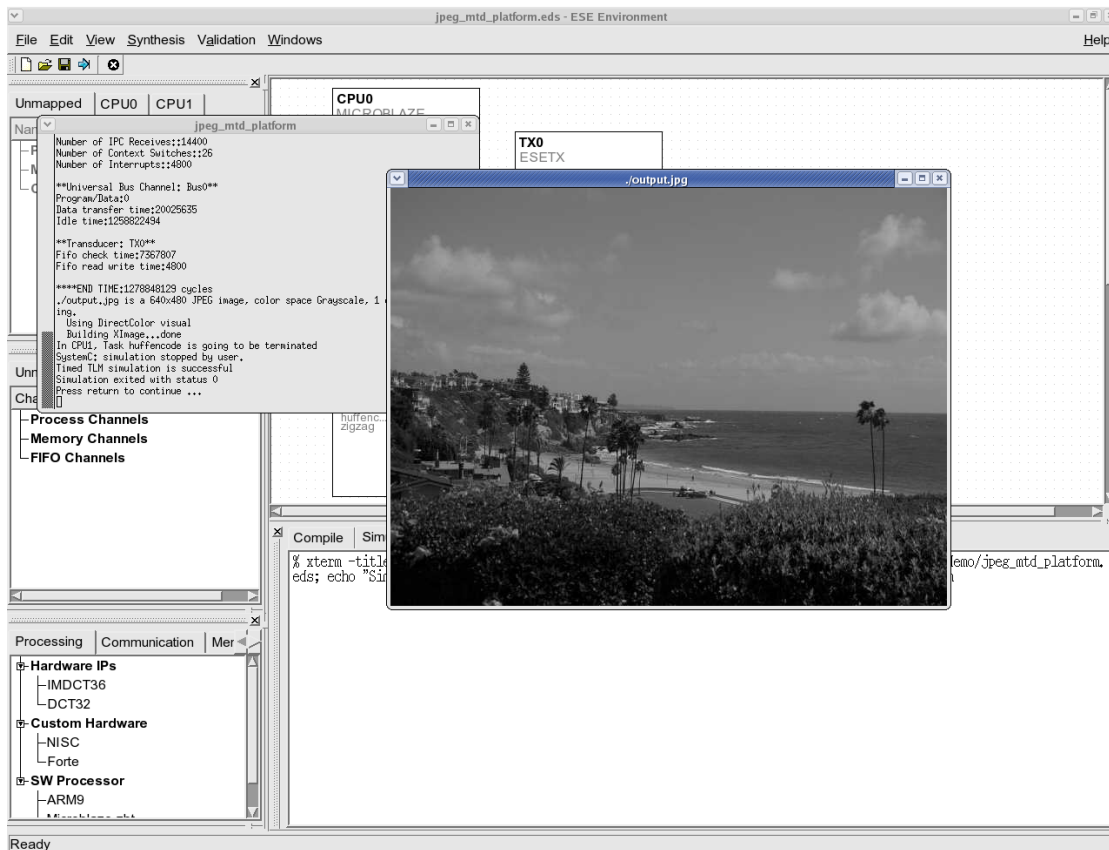
The simulation pops up a terminal that shows the picture size of BMP input that have been encoded. The JPEG encoder we are using deals with BMP inputs of 640x480 size. An additional window shows the picture of the encoded JPEG which is the output of the simulation. The pop up windows can now be killed simply by pressing "Enter" in the simulation logging terminal.

4.4.4. Generate Timed TLM



Similar to the functional TLM generation, the Timed TLM can be generated automatically by selecting **Synthesis**→**Generate Timed TLM** from the menu bar. The bus channels generated for timed TLM will include timing for synchronization, arbitration and data transfer. The timing parameters are imported into the TLM from the bus data model. For the computation part, we use a retargetable source level timing estimation technique that utilizes the PE data models. Naturally, the timed TLM generation and compilation is significantly slower than functional TLM generation, but still in the order of seconds.

4.4.6. View Timed Simulation



The timed TLM simulation looks very similar to the functional TLM simulation except for one marked difference. Notice that timed simulation is significantly slower than functional TLM simulation. This is natural since we are simulation a lot more "wait" statements that are annotated to the application codes. However, our results show that this is still several orders of magnitude faster than RTL simulation for the same design.

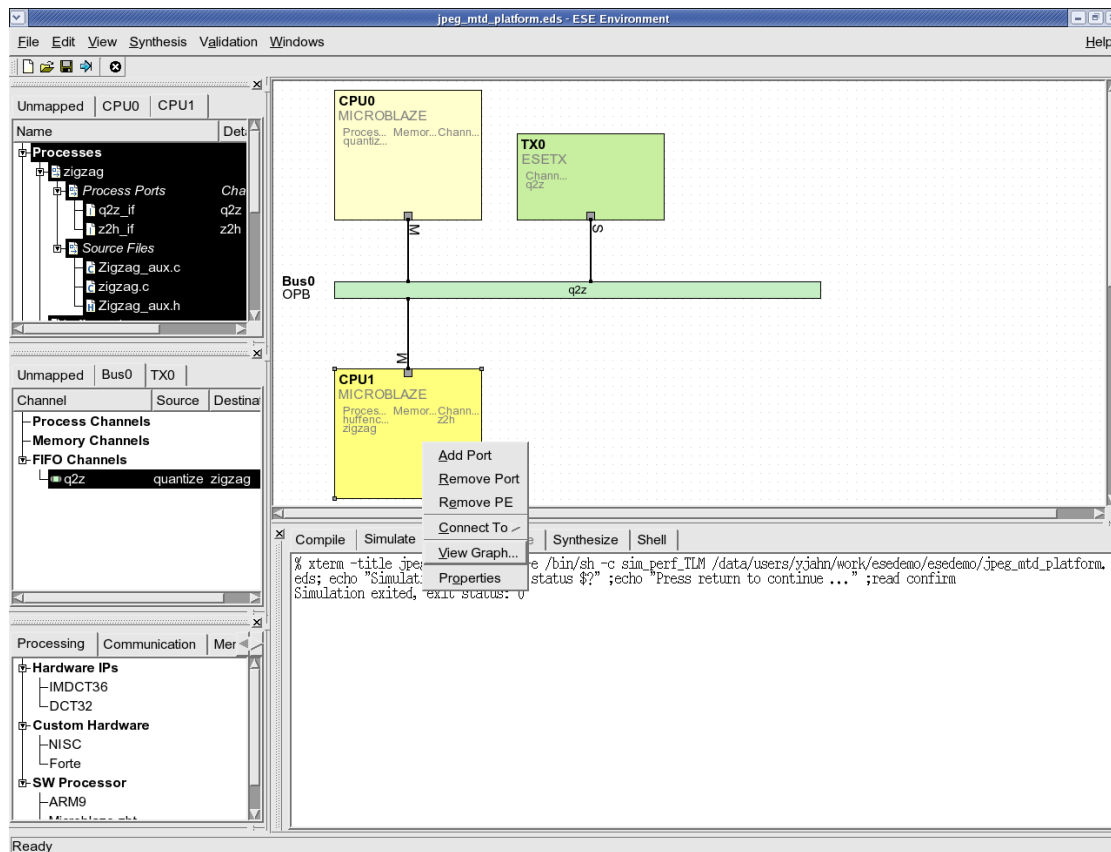
When the timed simulation ends, several statistical data are dumped in the simulation logging terminal. These are the estimated cycles for CPU computation and communication, bus congestion estimates and so on. However, all these estimated performance statistics can be viewed graphically as shown in next section.

4.5. TLM Performance Estimation

The timed TLM produces several statistical data that is gathered during simulation. Since the source annotation is fine grained, the TLM produces results for cycles used for invocation of each function in the application code. Computation and communication cycles for each PE can be viewed using pie charts. The distribution of cycles for each function amongst its sub-functions can be browsed recursively. Similarly, the distribution of inter-PE bus traffic over inter-process channels can also be viewed graphically.

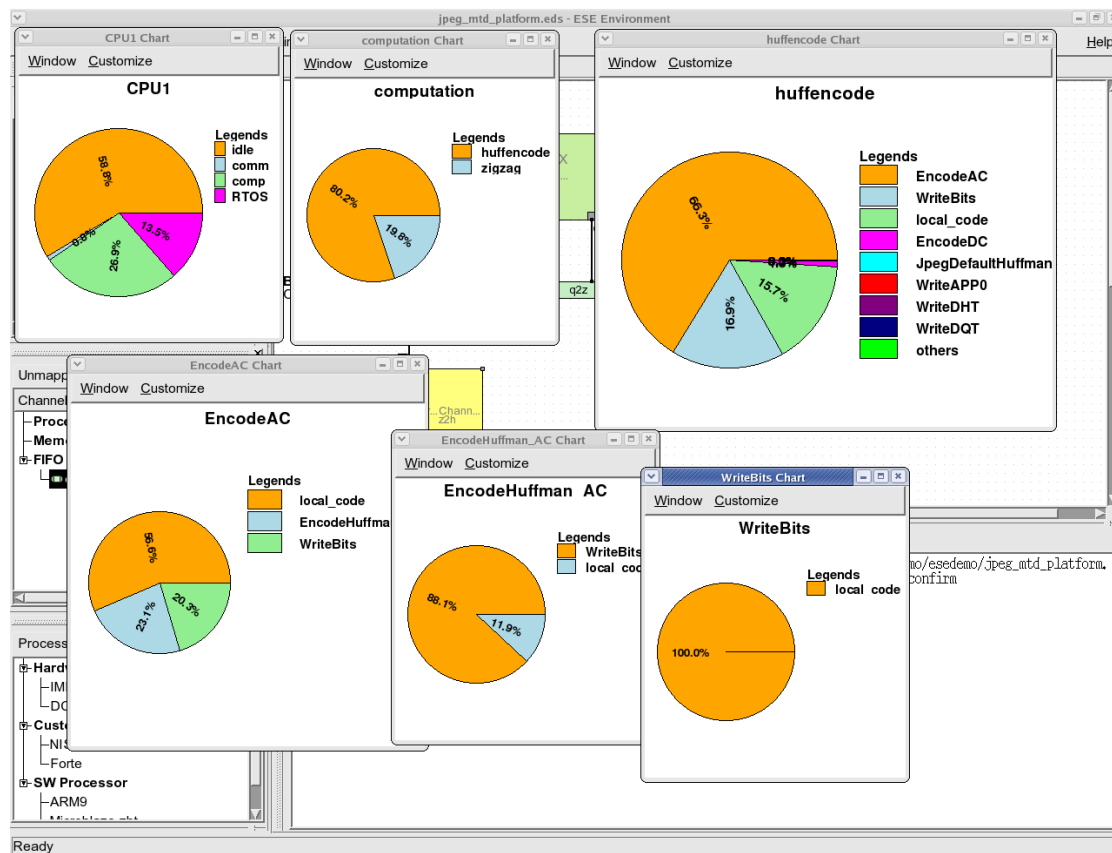
The performance estimates are useful for early platform and mapping evaluation. Since the timed TLMs are generated automatically, and TLM simulation is very fast, early design space exploration becomes feasible. Users may explore platforms manually or plug in their exploration algorithms for system level design optimization.

4.5.1. View Performance Estimates



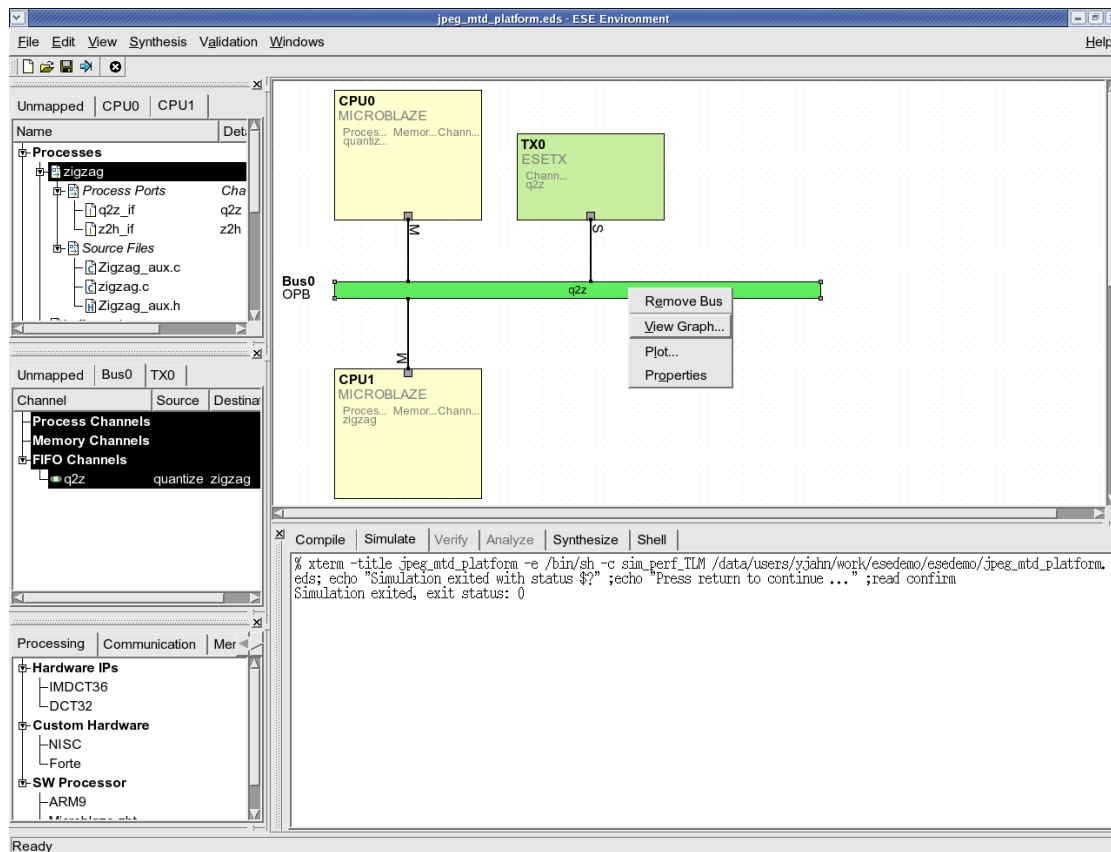
To view the PE performance statistics, right-click on the PE in the platform canvas and select View Graph. In this case, we will select the CPU1 Microblaze processor.

4.5.2. PE, Process and Function Level Estimates



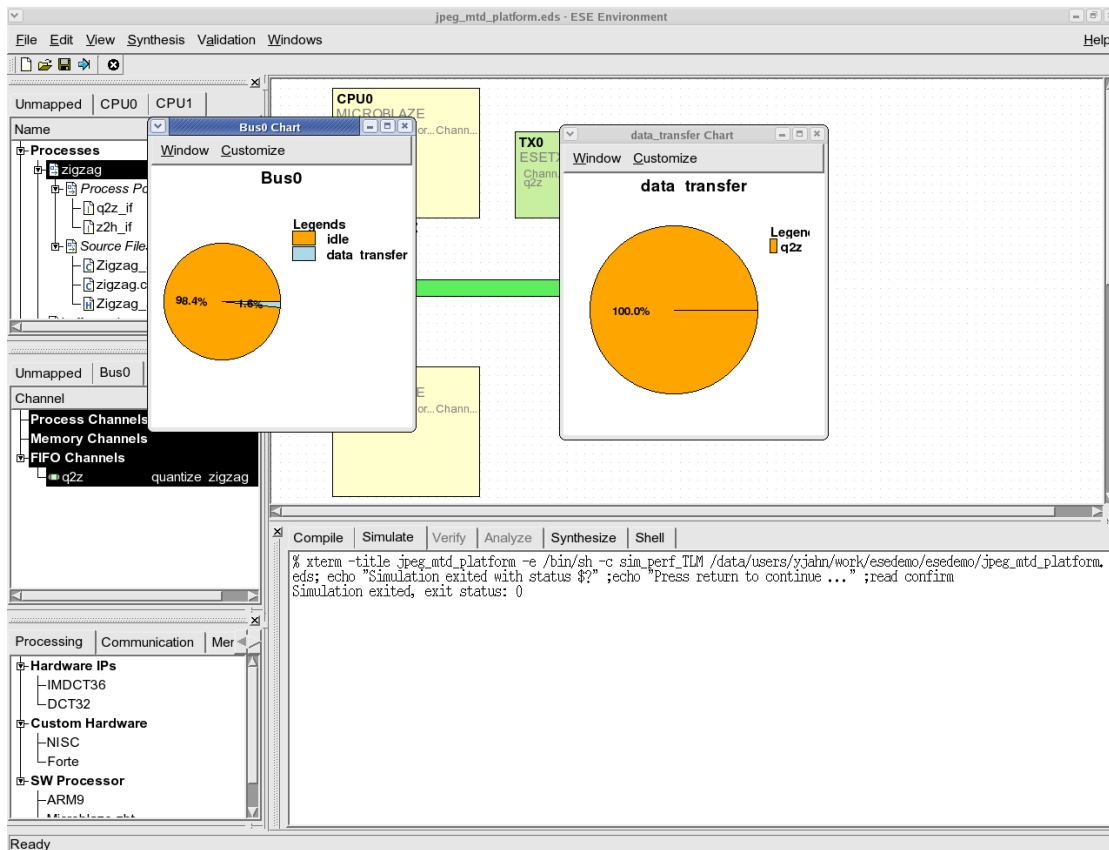
The first pie-chart window divides the total execution time into computation, communication and idle cycles. Double-clicking on the computation part of the pie chart pops up the distribution of computation across different processes in the design. In this case, we have only two processes, "zigzag" and "huffencode" mapped to CPU1. Double-clicking on the process in the pie chart produces the distribution of computation across the top level functions in the process. These function(s) call lower level functions and so on. Double-clicking on a function produces the pie chart for the distribution of cycles amongst the sub-function invocations. Using this viewing feature, the user may go down to any level in the function call hierarchy. If the pie chart appears too small, please increase the window size to enlarge the chart.

4.5.3. View Communication Estimates



To view the bus communication statistics, right-click on the bus in the platform canvas and select View Graph. In this case, we will select the only OPB bus.

4.5.4. Bus and Channel Level Estimates



The top level pie-chart for the bus shows the distribution of bus cycles in idle, program/data access and inter-PE data transfer phases. Double-clicking on the "data-transfer" part of the pie-chart produces the distribution of communication traffic amongst the various application channels in the design.

Chapter 5. Conclusion

In this tutorial we presented the ESE Front-End design methodology and tool set. ESE produces two types of TLMs; one for untimed functional verification and a timed TLM for performance estimation. The C/C++ and graphical input not only allows non-experts to create system models, but it also supports reuse of legacy code for product upgrades. The TLMs generated by ESE Front-End can be synthesized into board prototype models by ESE Back-End. This feature is not available in any commercial or academic offering.

To draw the conclusion, ESE enables embedded system developers to use the following powerful advantages that have never been available before.

1. Automatic TLM generation.

New TLMs are generated automatically from a mapping of C/C++ application to an abstract graphical platform. This means that the designer may use existing application code and map it to different platforms without having to manually modify any SystemC code.

2. Eliminates SLDL learning.

ESE *eliminates the need for system-level design languages* to be learnt by the designer. Only the knowledge of C for creating application specification is required.

3. Enables non-experts to design.

This also *enables non-experts to design* systems. There is no need for the designer to worry about design details like protocol timing diagrams, low level interfaces etc. Consequently, *software developers can design systems*.

4. Supports platforms.

ESE is great for *platform based design*. System platforms can be graphically created and modified. Pre-existing platforms can be reused and upgraded. All of these tasks are orthogonal to the application development itself.

5. Customized methodology.

ESE can also be *customized to any methodology* as per the designer's choice of components, system architecture, models and levels of abstraction.

6. Enables IP reuse.

ESE simplifies *IP reuse* to a great extent by allowing import of RTL components at system level. With C models of the IP, the designer can generate high speed TLMs for verification and performance estimation.

7. Supports interoperability.

ESE supports *interoperability with industry standard languages and tools* . The input is C/C++ which is the language of choice for embedded applications. The output is SystemC which is the de-facto system level design language. The Back-End in ESE allows generation RTL blocks from C code using third party high level synthesis tools, such as Forte. The final output of ESE Back-End is a Xilinx project that can be input to the Xilinx Embedded Development Kit (EDK) for push button FPGA prototyping.

References

- S. Abdi, J. Peng, R. Doemer, D. Shin, A. Gerstlauer, A. Gluhak, L. Cai, Q. Xie, H. Yu, P. Zhang, and D. Gajski, *System-on-Chip Environment - Tutorial*, CECS Technical Report 02-28, September 24, 2002.
- A. Gerstlauer, R. Doemer, J. Peng, and D. Gajski, *System Design: A Practical Guide with SpecC*, Kluwer Academic Publishers Inc., June, 2001.
- D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers Inc., March, 2000.
- D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*, Prentice Hall, June, 1994.
- D. Gajski, F. Vahid, S. Narayan, and J. Gong, "SpecSyn: An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software System Design", IEEE Transactions on VLSI Systems, Vol. 6, No. 1, pp. 84-100, 1998, Awarded the IEEE VLSI Transactions Best Paper Award, June 2000.
- D. Gajski, L. Ramachandran, F. Vahid, S. Narayan, and P. Fung, "100 hour design cycle : A test case", Proc. Europ. Design Automation Conf. EURO-DAC, 1994.

