

Space Code Design Systems Inc.

Lab 1

Architecture Exploration and HW/SW Partitioning

1. Objective

The objective of this tutorial is to familiarize with platform-based design approach (i.e., modeling, partitioning, and refinement) using SpaceStudio graphical environment to design, simulate and profile a system across different levels of abstraction. You will be working with the dataflow sequence shown in Figure 1. It is a JPEG Decoder combined with contour, facial and eye detection in post-processing. The decompressed images are faces.

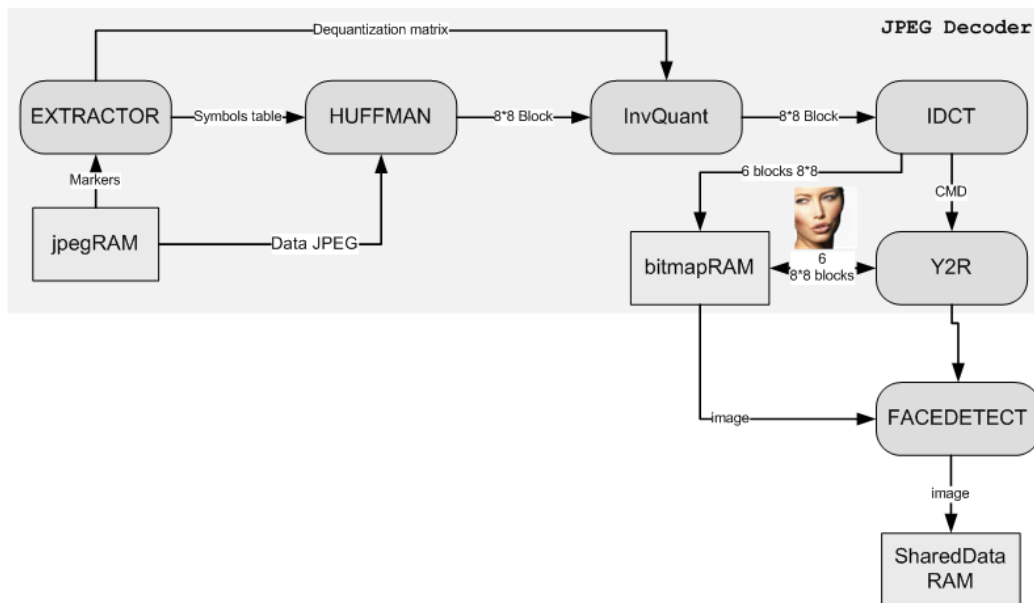


Figure 1 Data flow for JPEG Decoding

2. Target Processors

You will be working with a μ Blaze (MicroBlaze) which is a 32-bit RISC processor that is proprietary to Xilinx. The μ Blaze connects to two of two bus types: AMBA AXI, and LMB. The AXI bus serves to communicate with the peripherals and the other modules (e.g., over 2 to 3 cycles), while the LMB (Local Memory Bus) bus serves to read instructions and data directly from BRAM memory (e.g., in 1 cycle). Finally, you will see how a single task (module) runs on one or several processors, perhaps with the initial RTOS (μ C/OS II) replaced by Unity (a bare-metal pseudo OS), to see how that can be accelerated.

3. A Few Remarks Before Starting

This tutorial focuses on using SpaceStudio's Elix and Simtek technology for architectural exploration. The hardware/software co-synthesis for implementation on FPGA (e.g., Xilinx) conducted by SpaceStudio's GenX module does not play a part in this tutorial.

All of the modules for the JPEG application have been pre-characterized for timing (wait¹) using a high level synthesis (HLS) tool named Cynthesizer from Forte Design Systems. Thus when one of these modules is mapped to hardware, SpaceStudio activates the timing models derived from that characterization. When we map the same module to software, SpaceStudio automatically deactivates the timing annotations, since it is the software cross-compiler (via the scheduling of instructions) that determines the different delays (what is commonly called the process of linearization).

It is possible that when you try an initial solution using multiple processors (e.g., sections 4.8), the speed-up versus a uniprocessor solution is smaller than expected. That is simply because the achievable speed-up is limited by the algorithm's sequential fraction, and by communication and synchronization delays between processors. To achieve higher speedups, it might be necessary to make modifications to the algorithm to take further advantage of parallel computation, which this tutorial does not do.


4. Tutorial

In this tutorial, you will become familiar with the SpaceStudio tool while exploring a JPEG Decoder at different levels of abstraction and with different hardware and software architectures.

4.1. Opening the project

To get started, you must first obtain a copy of the reference (initial) project. Once you have obtained the reference project, expand the zip file in a directory on a path that contains only unaccented characters without spaces. This directory will be denoted by the token **%PROJECT_ROOT%**.

¹ In SpaceStudio the *wait()* statement has been replaced by the *computeFor()* statement

To open the project, you must next launch SpaceStudio. SpaceStudio can be accessed in the Start menu. Once you have launched SpaceStudio, click on  to open the existing project. Browse to the file *JPEGDecoder.wsp* that is located in **%PROJECT_ROOT%**.

4.2. Creating a functional specification with Elix

In SpaceStudio, you can create two types of design specifications, non-functional/functional specification (algorithm) using the Elix module and system specification (architecture) using the Simtek module. In this section, we will focus on functional specification using Elix. The principal characteristic of this configuration is that it performs abstract communications. All communications occur on a UTF (untimed functional) crossbar channel or a TF (timed functional) channel which simulate the bus arbitration. To create a functional specification using Elix, follow these instructions:

1. Click on **Architecture/Configuration**
2. In the drop-down menu, click on **New Configuration...**
3. Choose **Elix** for the technology, enter **validation** for the name of the configuration and then click on **OK**
4. Click again on **Architecture/Configuration**
5. In the drop-down menu, click on **Configuration Manager...**
6. In the upper table of the Instance Manager window, add all *User Blocks* (i.e., application).
7. In lower table of the Instance Manager, in the *Components* section, add the following SpaceLib library :
 - RegisterFile
 - TFChannel
8. In the *Components* section, add 3 XilinxBRAM with the following properties:

Instance	Id key	Size
jpegRAM	JPEGRAM_ID	0x100000
bitmapRAM	BITMAPRAM_ID	0x100000
sharedRAM	SHAREDGRAM_ID	0x100000

9. Change the parameter *Upload File Name* property of jpegRAM to `.././.././../././import/txt/jpegRAMInit.txt`²
10. Click on **Next**
11. In this section, the second page of the Configuration Manager called the Binding Manager, connect each peripheral to the channel *TFChannel1*. To do that, in the column labeled *TFChannel1*, click the entry for each case and select **Connected** for each **Instance**.
12. Click on **Tools**
13. In the dropdown menu, click on **Generate**
14. Next, click on **OK**

² This file contains the location of the file that will be decompressed (i.e. jpeg1.jpg at line 14).

In the System Map window, the graphical view of your system should resemble Figure 2. If that is not the case, redo the steps and be sure to connect all of the peripherals. The arrangement of the elements on the figure may differ from yours.

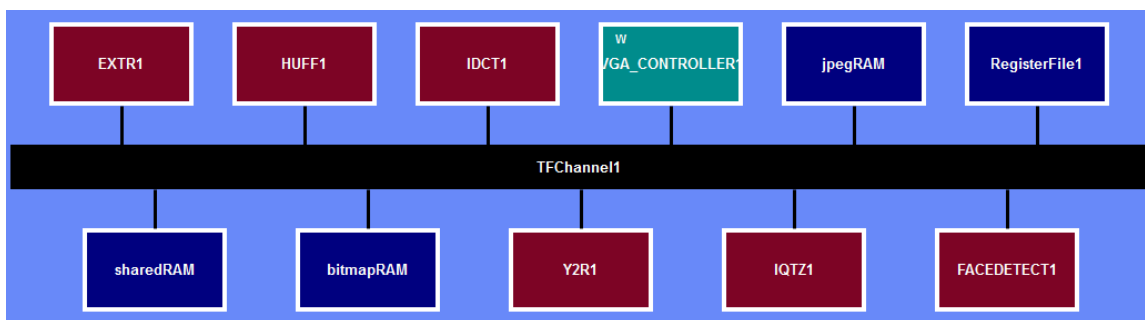




Figure 2 Functional Specification in Elix

Please note that if you make any changes to your configuration for one reason or another, you must regenerate your project according to previous steps 12 and 14, prior to compilation. The generation process is an important step which automatically generates all required Makefiles, embedded firmware, bus mappings, virtual platform, glue logic, etc.

To compile your project, you merely need to click on the  (build) icon. It is now possible to launch the simulation. In SpaceStudio, launch the simulation by pressing on the  (run) icon. To determine that the simulation is running correctly, you can check the progress messages printed in the console window, at the bottom of the GUI. The progress message output should appear similar to the following:

```
[ STARTED... ]

      SystemC 2.3.0-ASI --- Aug 29 2014 20:50:53
      Copyright (c) 1996-2012 by all Contributors,
      ALL RIGHTS RESERVED

-----
SpaceLib v2.7.0
(c) 2005-2012 Space Codesign Systems Inc. All rights reserved.
-----

SpaceLib Verbose: [BasicRAM Core Component]
File ../../../../../../import/image/jpeg1.jpg successfully loaded
into memory (0x100020 - 0x100d04) size:3301
*****

Starting simulation.

SpaceLib Verbose: [reset_manager_1]

+-----+
|  RESET @[0.000000000]
+-----+

***JPEG 1

SpaceLib Verbose: [reset_manager_1]
No more reset to come, shutting down Reset Manager
@[10000 PS]
EXTR:JPEG1
***JPEG 2
```

```
-----  
EXTRACTOR: JPEG DECODING UNDEFINED MARKER 83 - skipping JPEG
```

```
SpaceLib Verbose: [EXTR1]  
This module has requested the END OF THE SIMULATION  
Waiting for SystemC to terminate... @[2299170000 PS]
```

```
Info: /OSCI/SystemC: Simulation stopped by user.
```

```
Simulation has ended @0.00229917 s  
Simulation wall clock time: 2 seconds.
```

```
[ FINISHED ]
```

If you don't see these messages, be sure to check that you had correctly followed all of the previous steps.

Note that **Simulation has ended** indicates the execution time for processing on image of 128 x 128 pixels, taking 0.00229917 sec. In other words, it would be possible to continuously 435 images per second.

As well, **simulation wall clock time** indicates the real world duration of the simulation, to the second.

From Elix to Simtek

Simtek is used to create system architectures and implements a model of type *Approximately Timed* (AT), versus *Loosely Timed* (LT) as was the case for functional specification created with Elix. SpaceStudio can map your functional specification in Elix to an initial system architecture in Simtek. First we will set up the mapping and then refine the communication channel:

1. Click on **Architecture/Configuration**
2. In the dropdown menu, click on **New Configuration...**
3. We will simply reuse the previous configuration created in Elix.
 - 3.1. In the tab *Technology*, select Simtek
 - 3.2. In the tab *Configuration*, type Partition1
 - 3.3. Select the option *Based on previous configuration*
 - 3.4. Choose Elix for the technology and **validation** for the configuration
4. Click on **OK**

A window will appear (Figure 3) in order to choose a more refined bus model for *TFChannel* (e.g., AT) – i.e., a functional bus (LT) is not relevant for architectural level.

5. In the **Import configuration** window click on **AMBA_AHBBus** under “New Subtype” and replace it with **AMBA_AXIBus_LT**
6. Then click on **Finish**

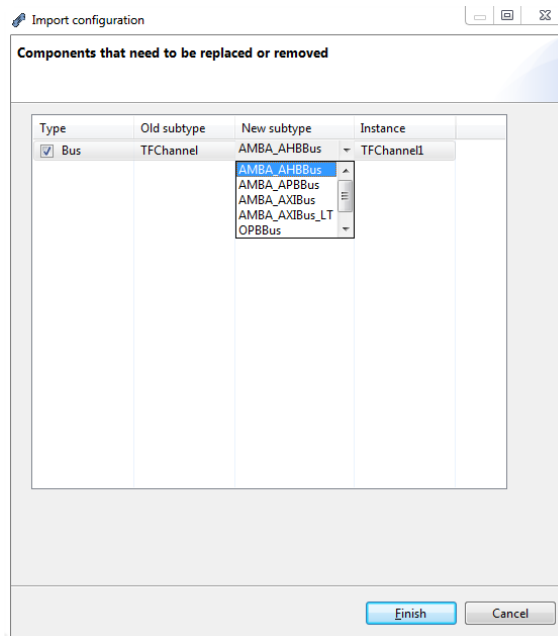


Figure 3 Refinement of the bus model

In the System Map window, the graphical view of Partition1 should resemble Figure 4. If that is not the case, redo the steps and be sure to connect all of the peripherals. The arrangement of the elements on the figure may differ from yours.

Partition1 in Simtek is a second configuration at the design architecture level that is completely hardware, since all modules are connected to TFChannel1 (now an AMBA_AXI_LT bus model) as coprocessors. Note also that the modules have a HW label (in the top left corner of each), which was not the case in Elix (since it was at the functional level, where no HW/SW distinction exists).

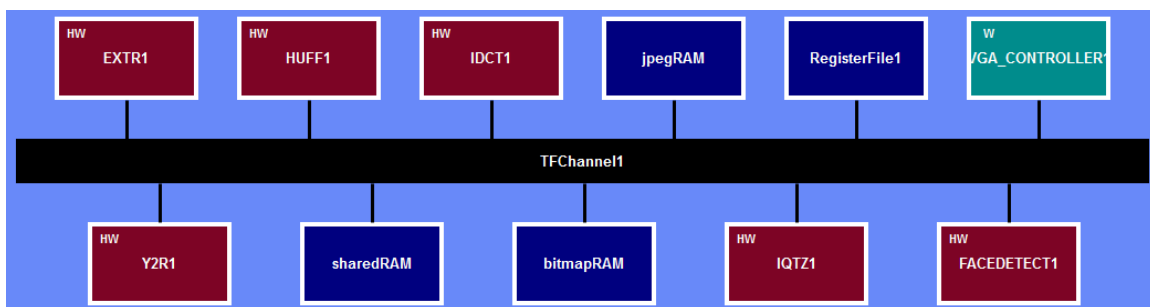


Figure 4 Simtek design architecture - all hardware (Partition1)

4.3. Addition of a μ Blaze and moving EXTR and HUFF from hardware to software

The role of *EXTR* is to read the header of the JPEG image. Since the format may evolve, *EXTR* may be updated in the future. Also, as *HUFF* is tightly connected to *EXTR* (Figure 1), we will assign *EXTR* and

HUFF to the processor (SW partition), while the rest will stay in hardware. Therefore, using Partition1, follow these instructions:

1. Click again on **Architecture/Configuration** (do not forget to select Partition1)
2. In the dropdown menu, click on **Configuration Manager...**
3. In the section *Components* of the configuration manager (lower table), add the μ Blaze processor (via Add >>)
4. A window will appear asking if you want to bind the μ Blaze on the existing bus (TFChannel1) or create a new bus. Select **Bind to an existing bus** and click on **Next**. You should see the additional support peripherals added automatically for the μ Blaze. Then click **Finish**
5. Click on **Next** (in the **Configuration Manager's** Instance Manager window)
6. In this section, the Binding Manager, you must connect each module to TFChannel1 except the modules *HUFF* and *EXTR*, which will be placed in the μ Blaze.
7. Click on **Finish**
8. In the *Architecture* tab of the design viewer, right-click on the μ Blaze component and select *uC/OS-II* (if necessary). Note that you have no other choice than *uC/OS-II* since *Unity* is used when there is only one task (bare-metal) whereas there are now two.
9. Click on **Tools**
10. In the dropdown menu, click on **Generate...**
11. Select the option, **Release**
12. Click **OK**.

In the System Map window, the graphical view of your system should resemble Figure 5. If that is not the case, redo the steps and be sure to connect all of the peripherals. The arrangement of the elements on the figure may differ from yours.

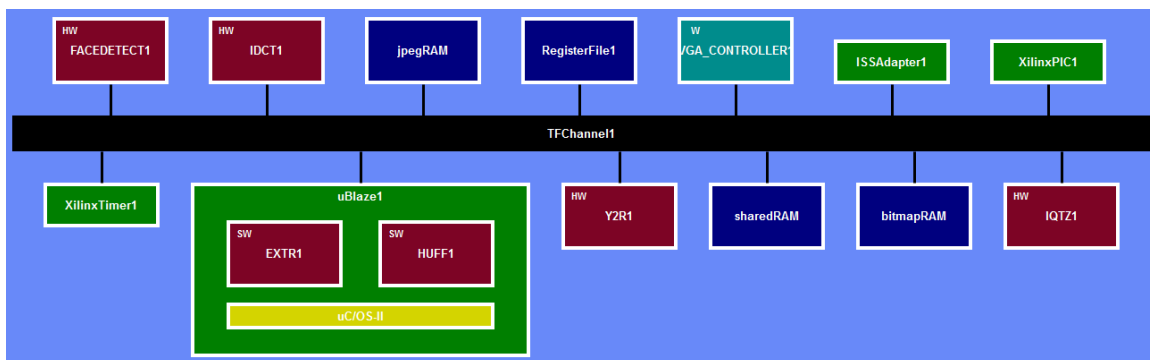




Figure 5 Updated System Architecture in Simtek

[N.B., hidden components associated with the μ Blaze processor (local memory bus, etc.) are not shown.]

As mentioned in the Elix case, if you modify your system architecture in Simtek, you must regenerate your configuration by following the previous steps 9 through 12.

To compile your project in SpaceStudio, just click on the  (build) icon, and then launch the simulation by pressing on the  (run) icon. If the simulation runs correctly, you will be able to see a progress message output in the console window, at the bottom of the GUI. The message output is similar to what you saw with the Elix module.



It should take around 0.0250155 seconds to decode one image (**Simulation has ended**), or about 40 images decoded per second continuously.

Suggestion : Repeat section 4.3, but this time place HUFF in hardware and replace uC/OS-II with Unity. You should see a significant acceleration in decoding. Why?

4.6 Monitoring of a system architecture with 1 μ Blaze using Simtek

Next, you must regenerate your configuration in *Monitoring* mode. To do that:

1. Click on **Tools**
2. In the dropdown menu, click on **Generate...**
3. Select the option, **Monitoring**
4. Finally, click on **OK**

To compile your project, you merely need to click on the  (build) icon. Launch the analysis by pressing on  (run). It is important to let the simulation finish on its own, normally, to ensure that it saves the data created for performance profiling.

Once the simulation finishes, open the *Monitoring Explorer dashboard* (MonitoringExplorer.xlsm) located in **%SPACE_CODESIGN_ENV%\util\MonitoringExplorer**

where **%SPACE_CODESIGN_ENV%** is SpaceStudio's installation directory.

And then :

1. From the dashboard, open the generated database located in **%PROJECT_ROOT%\Simtek\Partition1\Partition1\build\monitoring\monitoring.db3**
2. Click the **Task** tab and then, press on **Refresh**. You will then see the task execution chart for μ Blaze1.

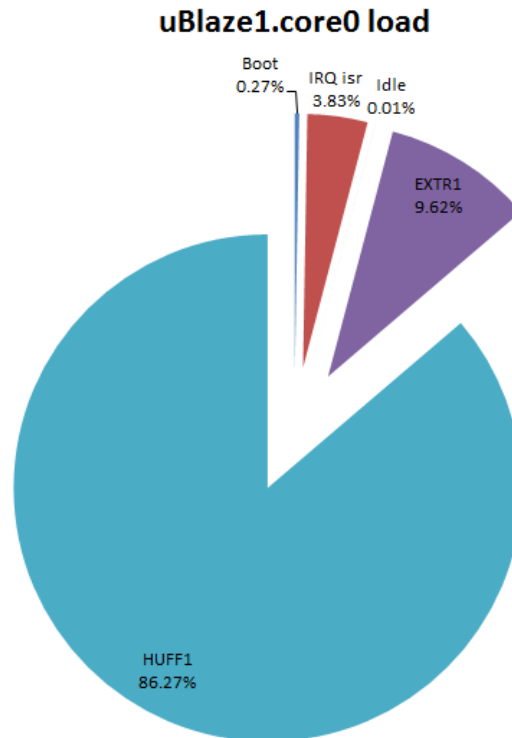


Figure 6 Task Profile for μ Blaze1

4.7 Hardware/Software Co-debugging

In this section, we will perform hardware/software co-debugging. During this exercise, we will examine hardware/software communications – in this case, communication between the *HUFF* and *IQTZ* modules. For this section, we will continue working with the configuration *Partition1* created in the previous section. (If that configuration does not exist, you will need to create it using the previous instructions.)


Before starting, add a debugger port to the processor. To do that, perform the following steps:

1. Click on **Tools**
2. In the dropdown menu, click on **Preferences...**
3. In the menu, on the left, expand (or double-click on) **Partition1** and select *ISS Debug Options*
4. In the settings window that appears, choose the tab μ Blaze1 to change the value *Debugger Socket Port*³ to **1234**.
5. Click on **OK**

Next, it is necessary to compile your project in *Debug* mode. To do that, perform the following steps:

³ If port 1234 is already used or restricted on your local network, you might need to select another port number.

1. Click on **Tools**
2. In the dropdown menu, click on **Generate...**
3. Choose the option **Debug All**
4. Check the option for *μBlaze1* below **Check ISS for Debugging** (if not already checked)
5. Next click on **OK**


Compile your project for hardware/software debugging by clicking on the  (build) icon. An up-to-date window may appear. Then to launch the co-simulation, you must perform the following steps:

1. Click on **Run**
2. In the dropdown menu, click on **Debug**
3. Then, select **HW/SW Co-debug...**

Once you have launched the co-simulation, a pop-up window titled “Confirm Perspective Switch” will appear, asking for approval to change the window layout on SpaceStudio. Select “Yes” to change to the debugging “perspective”.

We will focus on a blocking message-passing communication between modules HUFF and IQTZ, when the HUFF module writes an 8x8 block of data to IQTZ.

At the start, the debugger perspective only shows the hardware portion of your design architecture as shown in Figure 7, so select the tab *IQTZ.cpp* and then insert a *breakpoint* at line 166 of that file with a double left mouse-click. (You may need to activate line numbering in the editor window, do a right-click and select “Show Line Numbers” in Preferences (General → Editors → Text Editors), if that is not already enabled in your site’s SpaceStudio installation.)

Next, start the hardware simulation by clicking on the  button, i.e., Resume (F8). All the components will activate including the MicroBlaze processor model, which launches the processing of the software modules.

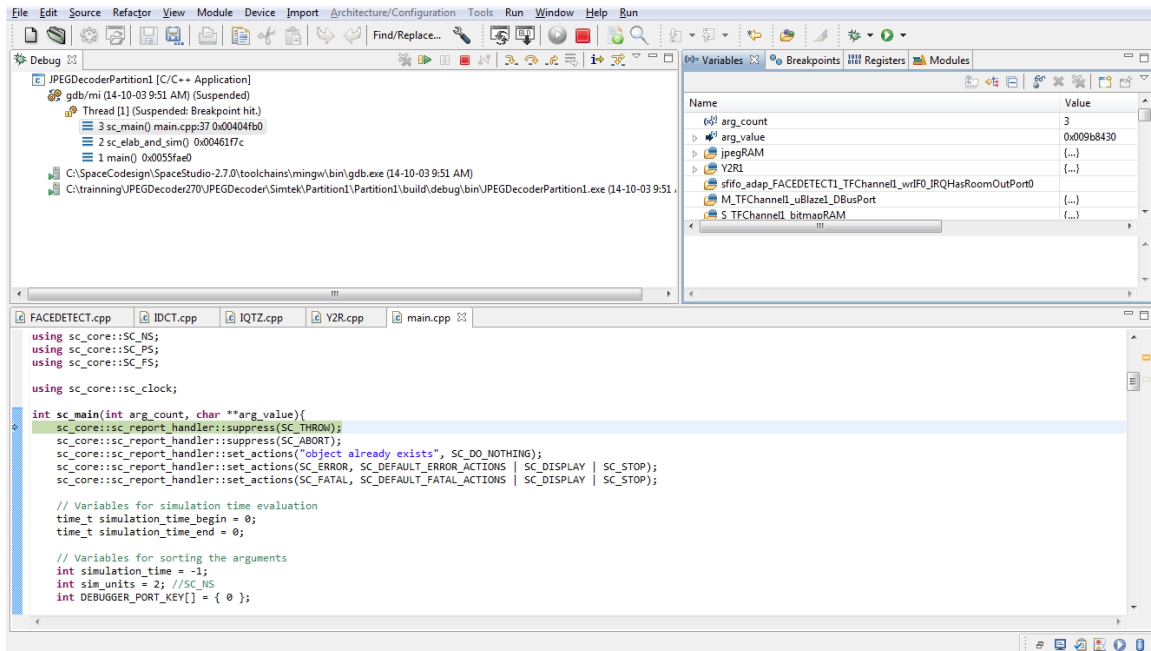


Figure 7 HW/SW Debugger - Before Activation

The debug console window will change, as seen in Figure 8, when the GDB Server appears for the software execution on the hardware. Additional tabs will appear for the software debugging windows. Select the tab for the file *HUFF.cpp* and then insert a *breakpoint* at line 398 with a double left mouse-click. This *breakpoint* will be activated when the software execution reaches that instruction. You have now 1 hardware breakpoint and 1 software breakpoint for the matching blocking communication. We thus should expect a kind of “ping pong” mechanism when stepping through the debugging session.

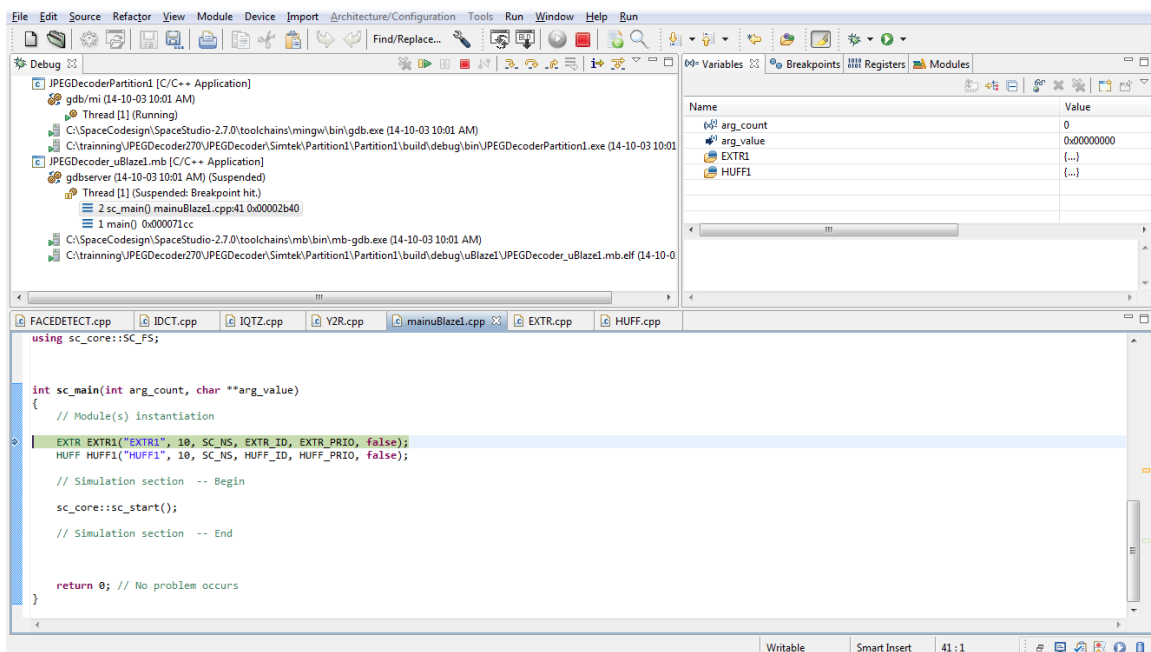








Figure 8 HW/SW Debugging - After Activation

At this point, the hardware debugger is running while the software debugger is suspended. To resume the software debugger, click on the  button to resume the whole simulation. The control of the simulation should then pass to the hardware debug window.

It is very important to understand that one window is active at a time (software or hardware). You can thus advance in the simulation (e.g., *step*, *continue* or *next*) one window at a time (e.g., hardware) while the other is blocked (e.g., software). Whenever you arrive at a communication by *ModuleRead* (or *ModuleWrite*), the simulation may change windows. For example, if *IQTZ* performs a blocking *ModuleRead* for data not yet received from *HUFF*, it will wait until a matching *ModuleWrite* by *HUFF* is completed before continuing. As a result, if *IQTZ* is executing in hardware and it becomes blocked by a blocking *ModuleRead*, the hardware window becomes blocked as the software window takes over to carry out the communication (i.e., the transfer of data).

At this moment, your simulation should be at line 166 of *IQTZ* (hardware window). Click on  and you should see control pass to *HUFF*(software window). Do it again, , and you should arrive at line 398 of *HUFF*. Then click again on  and you will return to line 166 of *IQTZ* (hardware window). And so on ...

To summarize, each time that the thread *HUFF* executes (as a task on μ C/OS II) the software module sets itself up to send data to the hardware module. By clicking on the  (Resume) button, that has the effect of unblocking the hardware module once it receives its data. You can then observe that the communication is blocking and that the execution of *ModuleWrite* has the effect of unblocking the module that carries out reading the data using *ModuleRead*.

This example illustrates how hardware/software co-debugging can be used to investigate the interactions between hardware and software modules. In the embedded systems industry, this approach is often called *hardware/software co-simulation*. To exit the simulation, you can remove all breakpoints and resume the simulation or you can simply stop the debugging session by clicking the  (Stop current task).

4.8 Creating a system architecture with 2 μ Blaze using Simtek

This exercise consists of creating a dual bus architecture, each one having a μ Blaze. We will move away from **Partition1** which used one μ Blaze and create **Partition2** which contains two busses, two μ Blazes and a bridge to connect the two busses:

1. Click on **Architecture/Configuration**
2. In the dropdown menu, click on **New Configuration**.
3. In the tab **Technology**, select **Simtek**.
4. In the tab **Configuration** type **Partition2**
5. Next click on the option **Based on previous configuration**
6. Choose **Simtek** for **Technology** and **Partition1** from **Configurations**
7. Finally click on **OK**

Partition2 is now the current configuration, we can modify it:

1. Click again on **Architecture/Configuration**
2. In the dropdown menu, click on **Configuration Manager**
3. In the section *Components* of the configuration manager (lower table), add a second μ Blaze processor (via Add >>)
4. A window will appear asking if you want to bind the μ Blaze on the existing bus (*TFChannel1*) or create a new bus. This time, select **Create a new bus** (**AMBA_AXIBus_LT** as **Bus type** and *TFChannel2* as **Instance name**, see Figure 9) and click **Next**. You should see a small window listing the support peripherals added for the second μ Blaze. Click **Finish** on that small window.
5. Click on **Next** on the **Configuration Manager** Instance Manager window.
6. Disconnect the module Y2R from TFchannel1 and connect it to μ Blaze2 (i.e., as SW).
7. Disconnect the module IDCT from TFchannel1 and connect it to TFchannel2.
8. Connect the dual-port memories bitmapRAM and jpegRAM to both busses (TFchannel1 and TFchannel2).
9. Click on **Finish**
10. Click on **Tools**
11. In the dropdown menu, click on **Generate...**
12. Select the option, **Monitoring**
13. Then, click on **OK**

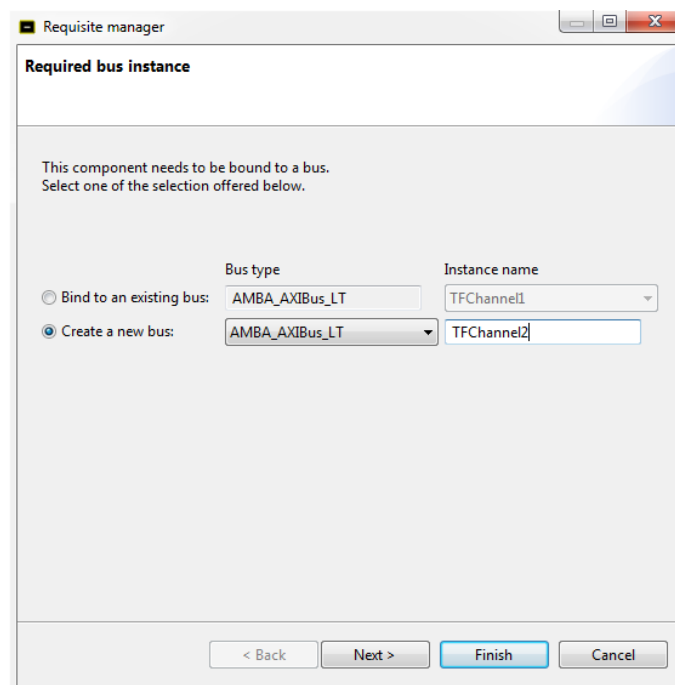


Figure 9 Selection of a second bus

In the System Map window, the graphical view of your system should resemble Figure 10 with the jpegRAM and bitmapRAM borders highlighted to indicate the dual-port connections.

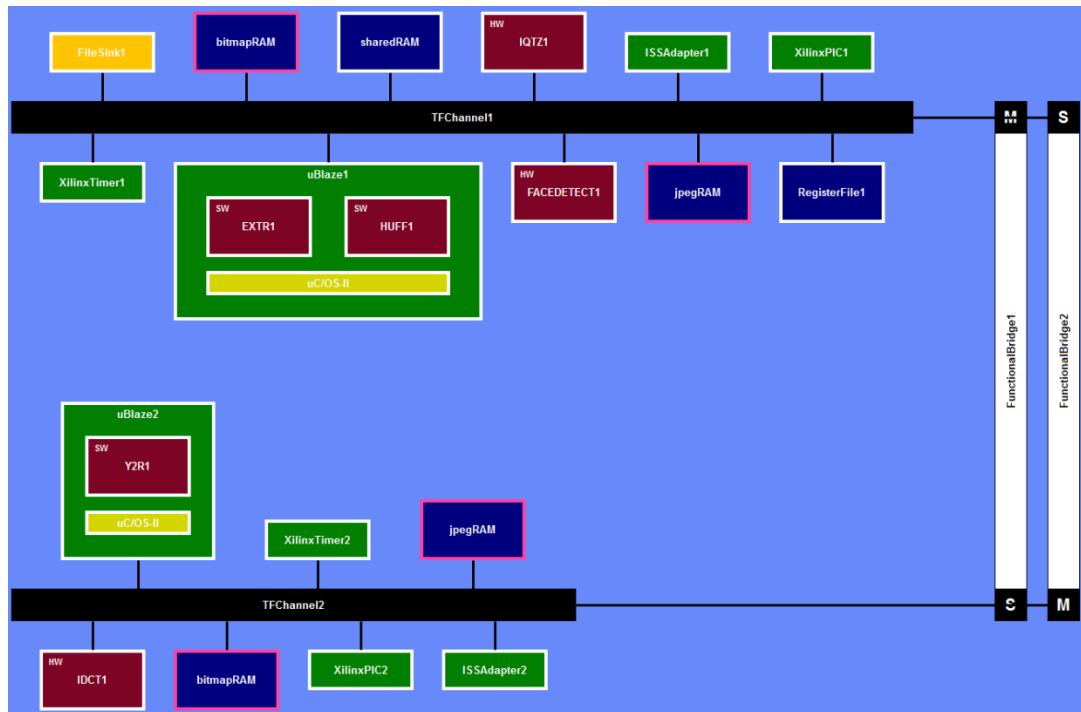


Figure 10 Multibus System Architecture in Simtek

Compile your project, with the  (build) icon. Launch the simulation by pressing on the  (run) icon.

In the same manner as for the other configurations, you can now analyze your simulation results by opening the *Monitoring Explorer dashboard* (MonitoringExplorer.xlsm) located in **%SPACE_CODESIGN_ENV%\util\MonitoringExplorer**

where **%SPACE_CODESIGN_ENV%** is SpaceStudio's installation directory.

And then :

1. From the dashboard, open the generated database located in **%PROJECT_ROOT%\Simtek\Partition2\Partition2\build\monitoring\monitoring.db3**
2. Click the **Task** tab and then, press on **Refresh**. You will then see the task execution chart for μ Blaze1. Repeat for **μ Blaze2** in order to see the load on the 2 processors.

You will see that only one task runs on **μ Blaze2**. You can thus experiment with using the Unity OS (by right-mouse click on **μ Blaze2**). Perform a new generation, compilation, execute the simulation and indicate the difference compared to a solution using *uC/OS-II*.

Finally, compare your simulation time (**Simulation has ended**) with that from section 4.3. Note that you now have 50% of the application in software. As well, examine the load on the processors, where there is a chance for future updates ...