Tutorial 1
Architecture Exploration and hardware/software partitioning

1. Objective

The objective of this tutorial is to familiarize with the SpaceStudio development environment to design, simulate and profile a system across two levels of abstraction. You will be working with the dataflow sequence shown in Figure 1. The application is a JPEG Decoder combined with contour, facial and eye detection in post-processing. The decompressed images are faces.

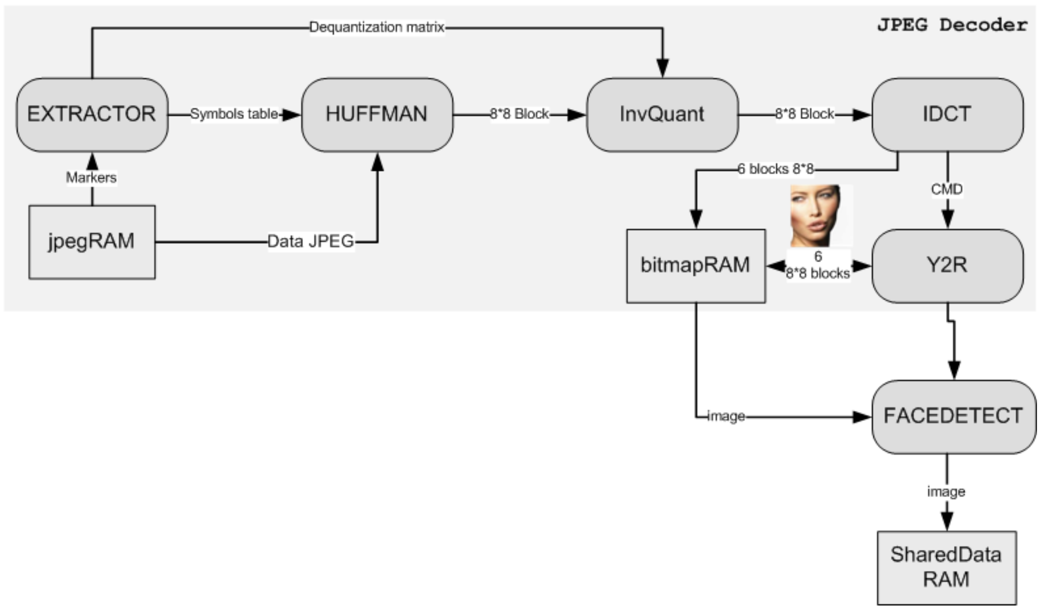


Figure 1 Data flow for JPEG Decoding

2. Target Processors

You will be working with a µBlaze (MicroBlaze) which is a 32-bit RISC processor that is proprietary to Xilinx. The µBlaze connects to an AMBA AXI (data port) and LMB (data and instruction port) buses. The AMBA AXI 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 (µC/OS II) replaced by Unity (a bare-metal pseudo OS), to see how the application can be accelerated.

3. A few remarks before starting

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

All of the modules for the JPEG application have been pre-characterized for timing (wait[1]) using a high level synthesis (HLS) tool named Vivado HLS from Xilinx. Thus when one of these modules is mapped to hardware, SpaceStudio activates the timing models derived from that characterization. When the module is mapped to software, SpaceStudio automatically deactivates the timing annotations, since the execution time is then determined by the processor on which the module runs.

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. JPEG Decoder

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%.**

To open the project, double-click the file *JPEGDecoder.spacestudio* from **%PROJECT_ROOT%.**

4.1.Functional specification

In SpaceStudio, you can create two types of design specifications: 1) non-functional/functional specification (algorithm) and 2) system specification (architecture details) depending on the abstraction of the models that compose the design.

The main objective of non-functional and functional specification is architecture sizing (do I need to increase the frequency? do I need to parallelize the algorithm? etc.). For functional specification, there is another objective which is to validate the functionality of the algorithm (does the algorithm fulfil the intended work?).

[1] In SpaceStudio the *wait()* statement has been replaced by the *ComputeFor()* statement

An architecture is an assembly of modules (i.e., pieces of C/C++ code) where lies an algorithm previously partitioned and library components (busses, memory, etc.) which connect these blocks together.

A functional specification is an architecture where all modules are mapped in hardware and connected by an abstract interconnect. SpaceStudio provides all the interconnects so the designers focus on the algorithm and architecture sizing.

To create an architecture:

1. Click on **Solution**
2. In the dropdown menu, click on **New Architecture…**
3. Enter **validation** for the name of the architecture and then click on **OK**

To configure the architecture:

1. Click on **Solution**
2. In the dropdown menu, click on **Architecture Manager…**
3. The first page of the Architecture Manager is called the <u>Instance Manager</u> where designers instantiate the application and components.
   3.1. add all *User Blocks* (i.e., application)
   3.2. RegisterFile
   3.3. GenericChannel
4. Also add 3 XilinxBRAM with the following properties:

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

5. Click on **Next**
6. The second page of the Architecture Manager is called the <u>Binding Manager</u>. Connect each peripheral to the channel *GenericChannel0*. To do that, in the column labeled *GenericChannel0*, click the entry for each case and select **Connected** for each **Instance.**
7. Press the **Finish** button.
8. Right-click on jpegRAM from the Project Explorer (left of the screen) and select **Edit properties…**
9. In the properties window, check **Add upload file** option.
10. Click on **Browse…** and enter navigate to the folder **%PROJECT_ROOT%\import\image\** and select the file **jpeg1.jpg**. Click **Ok**.

To generate the virtual platform for the architecture:

1. Click on **Tools**
2. In the dropdown menu, click on **Generate…**
3. Click on **OK**

Please note that if you make any changes to your architecture (e.g., add/modify a component, add/modify the application), you must regenerate your architecture according to previous steps 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 the architecture:

1. Click on **Tools**
2. In the dropdown menu, click on **Build.**

Alternatively, clicking on build icon from the toolbar will compile the architecture. When the compilation process is completed, it is possible to launch the simulation:

1. Click on **Run**
2. In the dropdown menu, click on **Execute**.

Alternatively, clicking on the run icon from the toolbar will launch the simulation. 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 VGAController component will open a window that displays the decoded JPEG. Use this window to validate the decoded image and close the window afterward (right-click in the opened windows to show the control menu). The progress message output should appear like the following:

```
[ STARTED... ]

             SystemC 2.3.0-ASI --- Dec 29 2013 18:24:23
        Copyright (c) 1996-2012 by all Contributors,
        ALL RIGHTS RESERVED

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Space Codesign Systems Inc.
Copyright 2005-2018. All rights reserved
http://www.spacecodesign.com
SpaceStudio 3.1.0
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

SpaceLib Verbose: [jpegRAM]
File C:/tutorial/JPEGDecoder/import/image/jpeg1.jpg successfully loaded
into memory (0x100000 - 0x100ce4) size:3301
********************************************************************

Starting simulation.

SpaceLib Verbose: [reset_manager]

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


EXTR:JPEG1

SpaceLib Warning: [EXTR0]
This module has requested the END OF THE SIMULATION
Waiting for SystemC to terminate... @[1061090000 PS]

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

```
Simulation has ended @ 0.00106109 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 0.00106109 sec. In other words, it would be possible to continuously 942 images per second.

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

4.2. System specification (architecture details)

In this section, we will create an architecture that implements a model of type *Approximately Timed* (AT), versus *Loosely Timed* (LT) as was the case for functional specification created previously. You can refine the solution for a more accurate timed design and test your application on various architectures. First, we will replace the abstract interconnect by the AMBA AXI model :

1. Click on **Solution**
2. In the dropdown menu, click on **New Architecture…**
3. We will reuse the previous architecture to jumpstart the new architecture.
   3.1. Type **partition1** as the architecture name
   3.2. Check the option ***Based on existing architecture***
   3.3. Choose **validation** for as the existing architecture
4. Click on **OK**

Now, in the project explorer, you have a new active architecture named **partition1** which is a copy of the previous one. To replace the bus, do the following steps:

1. Right-click on the node named **partition1** from the project explorer
2. From the popup menu, click on **Architecture Manager…**
3. In the Architecture instance, select the **GenericChannel** item and click on the **<< Remove** button.
4. Select **AMBA_AXIBus_LT** in the folder **Bus** in the available components and click **Add >>** button.
5. Click the **Next >** button.
6. Connect all components to the bus.

Suggestion: The Project Explorer can also be used to connect components *using the drag-and-drop feature. To do so, select one or more components and drag them on the bus.*

4.3. Addition of a µBlaze 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 the architecture partition1, follow these instructions:

1. Right-click on the node named **partition1** from the project explorer
2. From the popup menu, click on **Architecture Manager…**
3. Add the µBlaze processor (via **Add >>**)
4. A window will appear asking if you want to bind the *µBlaze* on the existing bus (AMBA_AXIBus_LT0) 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 µBlaze. Then click **Finish**
5. Click on **Next >**
6. Connect all modules to AMBA_AXIBus_LT0 except the modules *HUFF* and *EXTR,* which will be connected (i.e., mapped) to the µBlaze.
7. Click on **Finish**
8. In the *Project Explorer*, right-click on the *µBlaze* 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.

Suggestion: *The drag-and-drop feature can also be used to connect a module to a processor as well.*

To generate the virtual platform for the architecture:

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

As previously mentioned, you must regenerate the architecture's virtual platform. To compile the architecture, click on the (build) icon. Launch the analysis by pressing on (run). 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.

It should take around 0.0249772 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 μBlaze

Next, you must regenerate your architecture in *Monitoring* mode:

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

To compile the architecture, click on the (build) icon. Launch the analysis by pressing on (run). It is important to let the simulation finish on its own 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. From the dashboard:

1. Open the generated database located in **%PROJECT_ROOT%\solutions\solution1\architectures\partition1\build\monitoring\monitoring.db3**
2. Click the **Task** tab and then, press on **Refresh.** You will then see the task execution chart for μBlaze0.

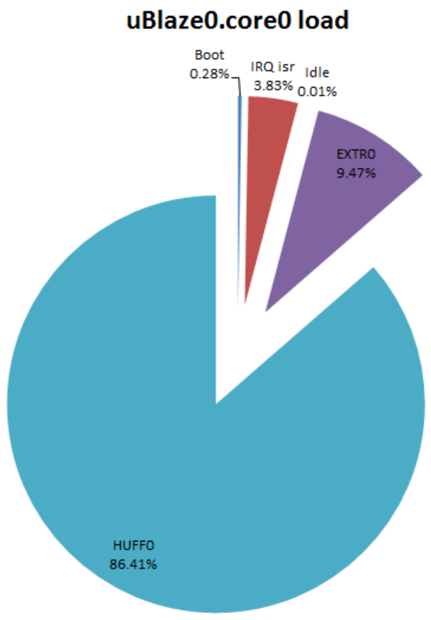


Figure 6 Task Profile for μBlaze0

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 architecture *partition1* created in the previous section. (If that architecture does not exist, you will need to create it using the previous instructions.)

Before starting, configure the processor's debugger port:

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 *µBlaze0* to change the value *Debugger Socket Port*[2] to **1234**.
5. Click on **OK**

Next, it is necessary to compile your project in *Debug* mode:

1. Click on **Tools**
2. In the dropdown menu, click on **Generate…**
3. Choose the option **Debug** and make sure all debuggable elements are checked
4. Next click on **OK**

Compile your project for hardware/software debugging by clicking on the (build) icon. 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".

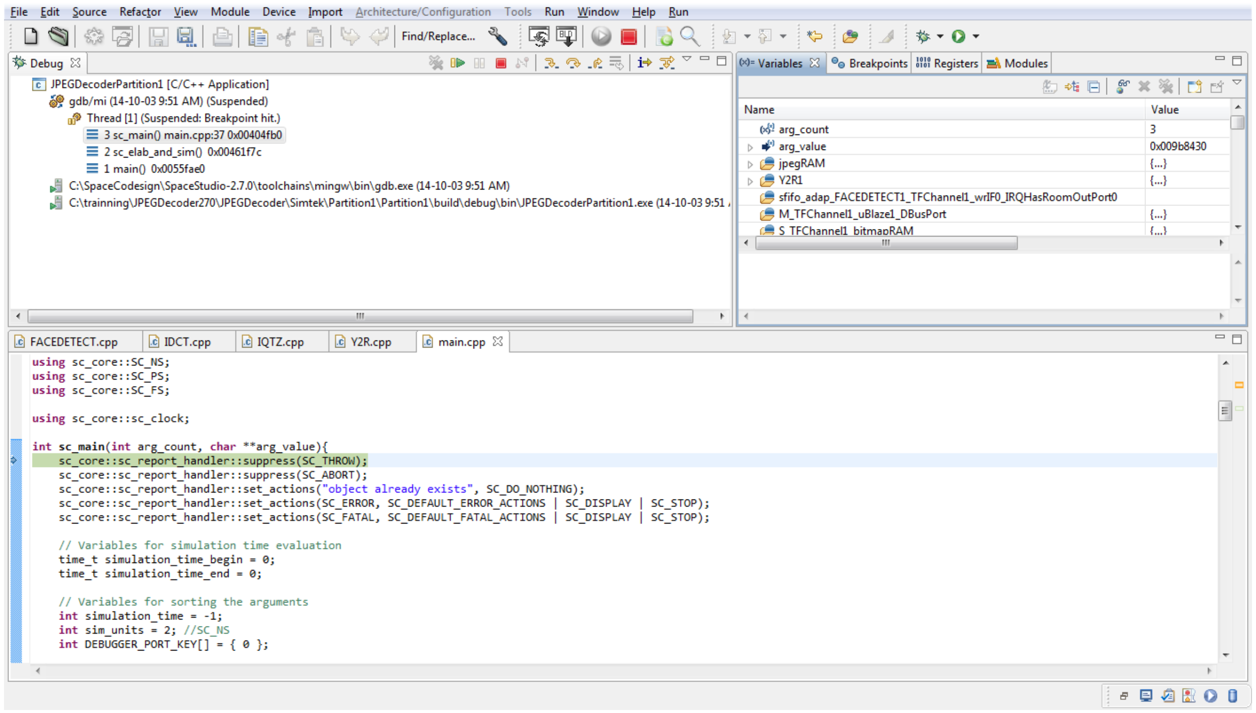
We will focus on a blocking message-passing communication between modules HUFF and IQTZ, when the HUFF module writes a block of data to IQTZ.

At the start, the debugger perspective only shows the hardware portion of your design architecture as shown in Figure 2, so select the tab *IQTZ0.cpp* and then insert a *breakpoint* at line 329 of that file, at the ModuleRead instruction, with a double left mouse-click. (You may need to activate line numbering in

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

the editor window, do a right-click on the grey space left of the editor and select “Show Line Numbers” from the popup menu)

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.



```
JPEGDecoderPartition1 [C/C++ Application]
  gdb/mi (14-10-03 9:51 AM) (Suspended)
    Thread [1] (Suspended: Breakpoint hit.)
      3 sc_main() main.cpp:37 0x00404fb0
      2 sc_elab_and_sim()  0x00461f7c
      1 main()  0x0055fae0
  C:\SpaceCodesign\SpaceStudio-2.7.0\toolchains\mingw\bin\gdb.exe (14-10-03 9:51 AM)
  C:\trainning\JPEGDecoder270\JPEGDecoder\Simtek\Partition1\Partition1\build\debug\bin\JPEGDecoderPartition1.exe (14-10-03 9:51
```

Name	Value
arg_count	3
arg_value	0x009b8430
jpegRAM	{...}
Y2R1	{...}
sfifo_adap_FACEDETECT1_TFChannel1_wrIF0_IRQHasRoomOutPort0	
M_TFChannel1_uBlaze1_DBusPort	{...}
S_TFChannel1_bitmapRAM	{...}

```
using sc_core::SC_NS;
using sc_core::SC_PS;
using sc_core::SC_FS;

using sc_core::sc_clock;

int sc_main(int arg_count, char **arg_value){
    sc_core::sc_report_handler::suppress(SC_THROW);
    sc_core::sc_report_handler::suppress(SC_ABORT);
    sc_core::sc_report_handler::set_actions("object already exists", SC_DO_NOTHING);
    sc_core::sc_report_handler::set_actions(SC_ERROR, SC_DEFAULT_ERROR_ACTIONS | SC_DISPLAY | SC_STOP);
    sc_core::sc_report_handler::set_actions(SC_FATAL, SC_DEFAULT_FATAL_ACTIONS | SC_DISPLAY | SC_STOP);

    // Variables for simulation time evaluation
    time_t simulation_time_begin = 0;
    time_t simulation_time_end = 0;

    // Variables for sorting the arguments
    int simulation_time = -1;
    int sim_units = 2; //SC_NS
    int DEBUGGER_PORT_KEY[] = { 0 };
```

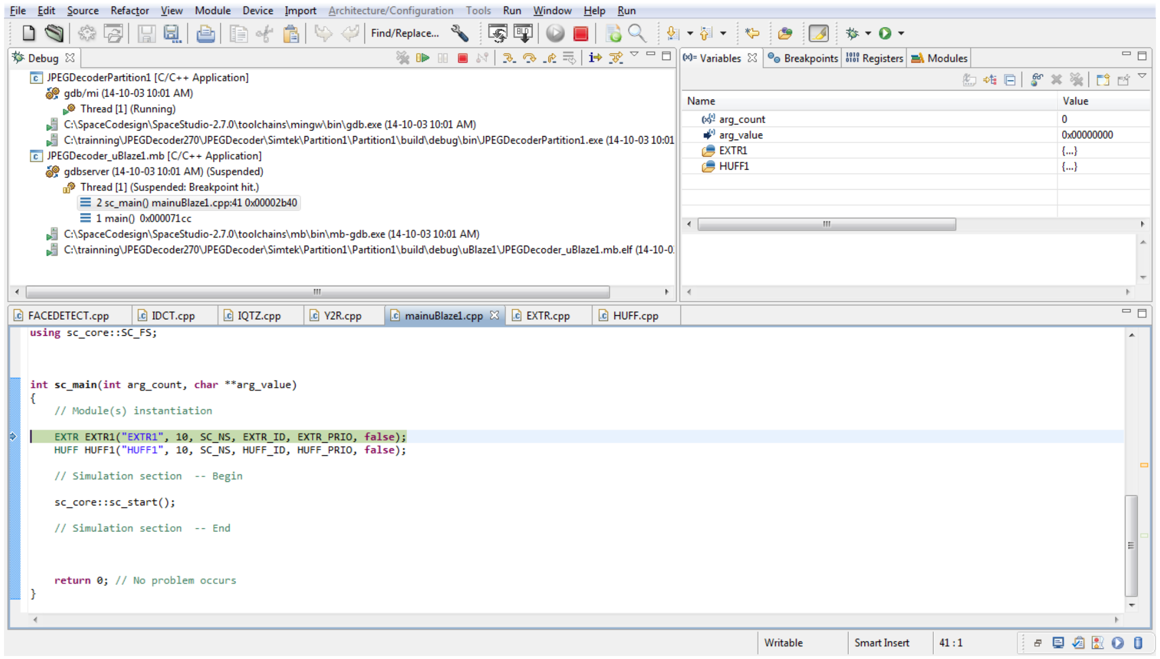
Figure 2: Hardware break

The debug console window will change, as seen in Figure 3, when the GDB server appears for the software execution on the hardware. Additional tabs will appear for software debuggable files.

Note: SpaceStudio generates an optimized version of software-mapped modules which include low-level drivers for communication with the system. It is the generated files that are compiled and suitable for debugging. When entering the debug perspective, SpaceStudio will automatically open the optimized version of the software-mapped module. Editing the generated files will not change the original file.

Select the tab for the file *HUFF0.cpp* and then insert a *breakpoint* at line 754 with a double left mouse-click. This *breakpoint* will be activated when the software execution reaches that instruction.

Right now, there are 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.



```
JPEGDecoderPartition1 [C/C++ Application]
    gdb/mi (14-10-03 10:01 AM)
        Thread [1] (Running)
    C:\SpaceCodesign\SpaceStudio-2.7.0\toolchains\mingw\bin\gdb.exe (14-10-03 10:01 AM)
    C:\trainning\JPEGDecoder270\JPEGDecoder\Simtek\Partition1\Partition1\build\debug\bin\JPEGDecoderPartition1.exe (14-10-03 10:01
JPEGDecoder_uBlaze1.mb [C/C++ Application]
    gdbserver (14-10-03 10:01 AM) (Suspended)
        Thread [1] (Suspended: Breakpoint hit.)
            2 sc_main() mainuBlaze1.cpp:41 0x00002b40
            1 main()  0x000071cc
    C:\SpaceCodesign\SpaceStudio-2.7.0\toolchains\mb\bin\mb-gdb.exe (14-10-03 10:01 AM)
    C:\trainning\JPEGDecoder270\JPEGDecoder\Simtek\Partition1\Partition1\build\debug\uBlaze1\JPEGDecoder_uBlaze1.mb.elf (14-10-0
```

Name	Value
arg_count	0
arg_value	0x00000000
EXTR1	{...}
HUFF1	{...}

```
using sc_core::SC_FS;


int sc_main(int arg_count, char **arg_value)
{
    // Module(s) instantiation

    EXTR EXTR1("EXTR1", 10, SC_NS, EXTR_ID, EXTR_PRIO, false);
    HUFF HUFF1("HUFF1", 10, SC_NS, HUFF_ID, HUFF_PRIO, false);

    // Simulation section  -- Begin

    sc_core::sc_start();

    // Simulation section  -- End



    return 0; // No problem occurs
}
```

Figure 3: Software break

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 *IQTZ* breakpoint (hardware window). Click on and you should see control pass to *HUFF*(software window). Do it again, , and you should arrive at *HUFF* breakpoint. Then click again on and you will return to *IQTZ* breakpoint (hardware window). And so on …

To summarize, each time that the thread *HUFF* executes (as a task on µC/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 right-clicking in the **Debug** window and by selecting **Terminate/Disconnect All**.

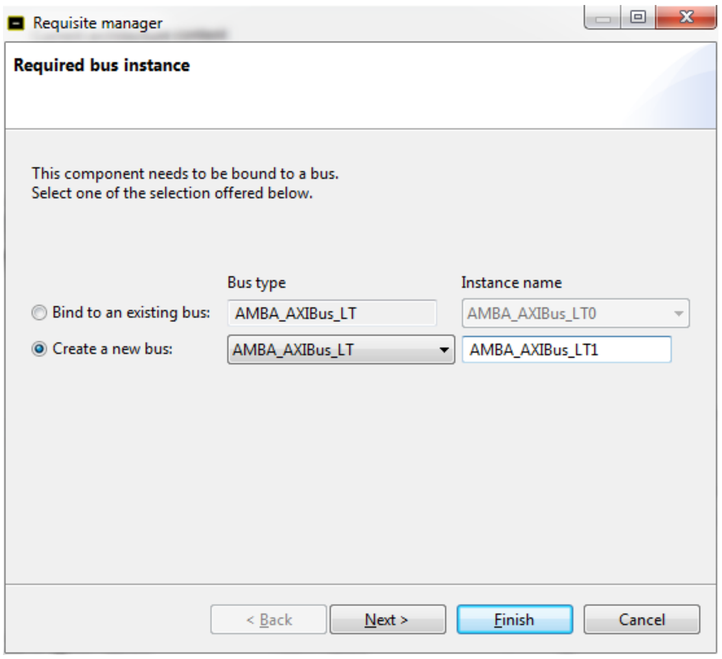
4.8 Creating a system architecture with 2 μBlaze

This section consists of creating a dual bus architecture, each one having a μBlaze. We will move away from **partition1** which used one μBlaze and create **partition2** which contains two busses, two μBlazes and a bridge to connect the two busses:

1. Click on **Solution**
2. In the dropdown menu, click on **New Architecture…**
3. We will reuse the previous architecture to jumpstart the new architecture.
    3.1. Type **partition2** as the architecture name
    3.2. Check the option ***Based on existing architecture***
    3.3. Choose **partition1** for as the existing architecture
4. Click on OK

To configure the architecture:

1. Right-click on the node named **partition2** from the project explorer
2. From the popup menu, click on **Architecture Manager…**
3. Add another μBlaze processor (via **Add >>**)
4. A window will appear asking if you want to bind the *μBlaze* on the existing bus (*AMBA_AXIBUS_LT0*) or create a new bus. This time, select **Create a new bus** (**AMBA_AXIBus_LT** as **Bus type** and *AMBA_AXIBus_LT1* as **Instance name**, see Figure 4) and click **Finish.**
5. Click on **Next** in the **Architecture Manager** Instance Manager window.
6. Disconnect the module Y2R from AMBA_AXIBus_LT0 and connect it to μBlaze1 (i.e., as SW).
7. Disconnect the module IDCT from AMBA_AXIBus_LT0 and connect it to AMBA_AXIBus_LT1.
8. Connect the dual-port memories bitmapRAM and jpegRAM to both busses (AMBA_AXIBus_LT0 and AMBA_AXIBus_LT1).
9. Click on **Finish**


Requisite manager

Required bus instance

This component needs to be bound to a bus.
Select one of the selection offered below.

	Bus type	Instance name
Bind to an existing bus:	AMBA_AXIBus_LT	AMBA_AXIBus_LT0
[✓ Create a new bus:]	AMBA_AXIBus_LT	AMBA_AXIBus_LT1

Figure 4: Selection of a second bus

To generate the virtual platform for the architecture:

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

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

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. From the dashboard:

1. Open the generated database located in **%PROJECT_ROOT%\solutions\solution1\architectures\partition2\build\monitoring\monitoring.db3**
2. Click the **Task** tab and then, press on **Refresh.** You will then see the task execution chart for µBlaze0. Repeat for **µBlaze1** in order to see the load on the 2 processors.

You will see that only one task runs on **µBlaze1**. You can thus experiment with using the Unity OS (by right-mouse click on **µBlaze1).** 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 …