UNIVERSITY OF YORK

Department of Computer Science

BlueIO: Real-time I/O Virtualisation for NoC Many-core Systems (v1.0)

Author:
Zhe Jiang

Supervisor:
Neil Audsley

November 18, 2016

Contents

Chapter 1

BlueIO

The BlueIO enables:

- *Off-load I/O operations to hardware* - BlueIO moves the VMM and low level I/O drivers from kernel mode to the BlueIO. In a system with BlueIO, while an I/O request being submitted to the BlueIO, the software parts do not need to wait for the completion of IO operations. Therefore, in an I/O-bounded system, the execution of I/O parts can not be a constraint of the software.

- *Logic isolation between guest VMs* - BlueIO shares/virtualizes one physic I/O device to multiple virtual I/O devices, that a guest VM can only request and operate its own virtual I/O device, which improves the security of guest VMs.

- *Abstracted high layer access* - An user application in a guest virtual machine is able to request and operate an I/O device via invoking the high layer drivers, which can be simple. For example, an user application can request to read a series of data from a SPI-Flash via sending a very simple request to the BlueIO: "*Reading SPI-Flash* (instruction), between *the start address* and *the end address* (parameters)".

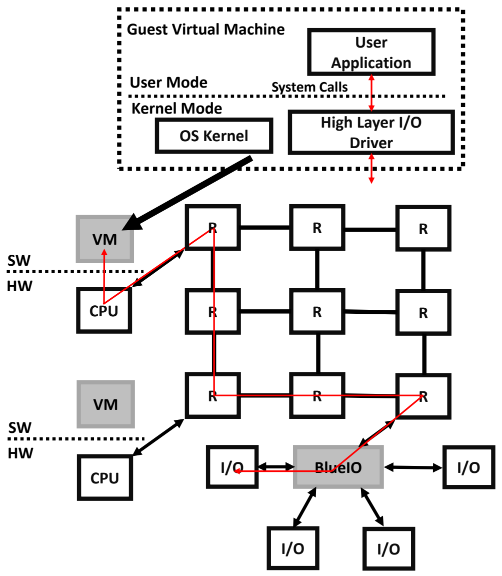


Figure 1.1: System Model

The virtualization technology in whole system has following features:

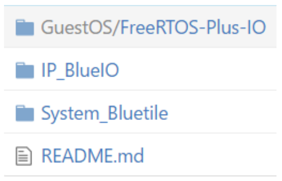
- *Bare-metal virtualization[7]* - Does not require the supports of a host OS.
- *Para-virtualization[5]* - Guest OS running in each guest VM should be modified. Specifically, the I/O management module of each original OS should be replaced by our high layer I/O drivers, which can reduce the software overhead significantly.

Typical use of the BlueIO within a NoC architecture is shown in Figure 1.1 – all the consumptions of I/O parts are performed by the BlueIO rather than remotely by the software. At run-time an application in a guest VM can invoke a high layer I/O driver on the BlueIO to achieve required I/O. The communications packets are transferred between the CPU and the BlueIO via routers in the NoC. As an example, the path of such a I/O request message is shown in Figure 1.1 as a red line.

Note that in the Figure 1.1, routers are labelled as "R" and guest virtual machine is labelled as "VM". Note that use of a NoC is not required by our system and hence a shared bus can also be used alternatively. However, in our experiments we use a NoC and this will not overestimate our performance as the use of a shared bus can further reduce the I/O predictability.

1.1 Source Project - Repository

The whole the source project can be accessed via link: *https://github.com/RTSYork/BlueIO*. The structure of the repository is shown in Figure 1.2.



GuestOS/FreeRTOS-Plus-IO
IP_BlueIO
System_Bluetile
README.md

Figure 1.2: Repository

Specifically, folder *System_Bluetile* contains the implementation of the many-core NoC mesh - Bluetile; floder *IP_BlueIO* contains the implementation the IP cores inside BlueIO, which includes the *BlueGrass* and the *IO-VMMs* for multiple I/Os; folder *GuestOS* contains the Guest OS can be ran in the guest VMs. Currently, we only upload the FreeRTOS with modified I/O manager, and more guest OSs will be uploaded in the future.

1.2 Design Idea

BlueIO provides the I/O virtualization for guest VMs, that a physical I/O device can be virtualized to multiple virtual I/O for virtual machines.

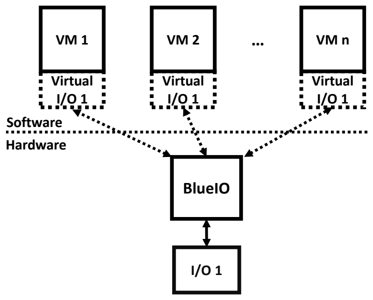


Figure 1.3: IO Virtualization

The applications in the guest VMs can only request the virtual I/O, rather than the physical I/O. The I/O virtualization is achieved via *Instruction Translation*. Specially, the BlueIO translates an I/O request to single or multiple I/O instruction(s) for the physical I/O.

1.3 Architecture of BlueIO

The architecture of the BlueIO consists of the following main parts (see Figure 1.4):

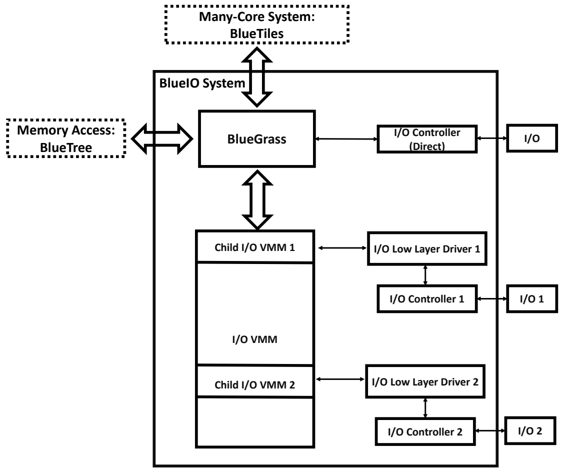


Figure 1.4: Architecture of BlueIO

- *BlueGrass* - provides the interface to/from application CPUs via the NoC mesh and the interface to/from external memory. In our approach, we use BlueTile[6] as the NoC mesh and BlueTree[**?**] for the memory accesses;
- *I/O Virtual Machine Monitor (I/O VMM)* - provides the functionality of virtualization for I/O deivces, which separates a physical I/O devices to multiple virtual I/O devices;
- *I/O Low Layer Drivers* - encapsulates the specific drivers of I/O controllers;
- *I/O Controllers* - controls the I/O devices directly; The I/O controllers can be connected to the VMM to achieve the functionality of virtualization, or be connected to the BlueGrass as a simple I/O controller.

These architectural elements are detailed in the following subsections.

1.3.1 BlueGrass

BlueGrass communicates with application CPUs, allocating incoming messages (I/O requests) to either the VMM or I/O controllers directly connected. The architecture of BlueGrass is shown in Figure 1.5, with the right part allocating incoming requests from the NoC; the middle part takes ending back data from BlueIO to CPUs; and the left part provides an interface for the memory accesses.

1.3.2 I/O Virtual Machine Monitor (I/O VMM)

I/O VMM maintains the virtualization of I/O devices. Considering the functionalities and features of I/O devices are different, it is very difficult to build a general-purpose module to achieve the virtualization for all kinds of I/O devices. Therefore, we create some specific-purpose virtualization modules inside I/O VMM

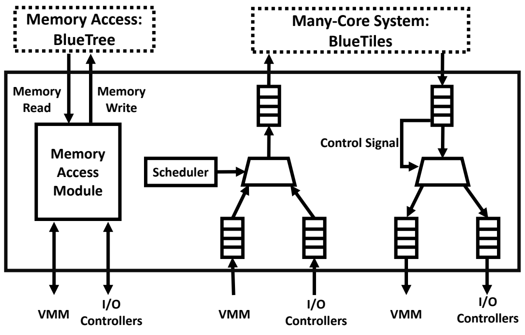
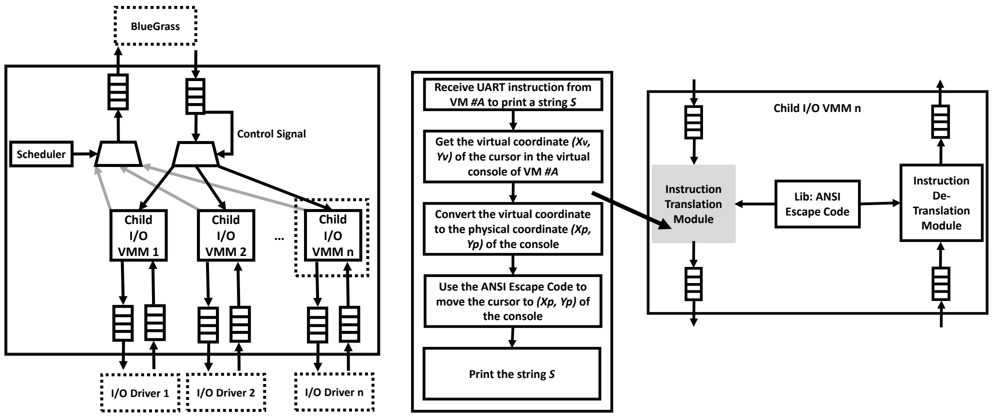


Figure 1.5: Architecture of BlueGrass


BlueGrass
Scheduler
Control Signal
Child I/O VMM 1
Child I/O VMM 2
...
Child I/O VMM n
I/O Driver 1
I/O Driver 2
I/O Driver n
Receive UART instruction from VM #A to print a string S
Get the virtual coordinate (Xv, Yv) of the cursor in the virtual console of VM #A
Convert the virtual coordinate to the physical coordinate (Xp, Yp) of the console
Use the ANSI Escape Code to move the cursor to (Xp, Yp) of the console
Print the string S
Child I/O VMM n
Instruction Translation Module
Lib: ANSI Escape Code
Instruction De-Translation Module

(a) Architecture of I/O VMM (b) Architecture of Child I/O VMM for UART

Figure 1.6: I/O VMM

for some commonly used I/O devices including UART, VGA, DMA, etc., these modules are named as *child I/O VMM*s. The architecture of I/O VMM is shown in Figure1.6.

As shown in Figure 1.6(a), once an I/O request is received by I/O VMM, it will be allocated to a corresponding child I/O VMM via the multiplexer. The child I/O VMM mainly takes charge of the instruction translation of income requests. When an I/O request is sent from the guest VM, and received by child I/O VMM, this request will be translated into actual I/O instructions of the corresponding underlying I/O drivers. The Figure 1.6(b) shows the architecture of child I/O VMM for UART.

1.3.3 I/O Low Layer Driver

I/O low layer driver Encapsulate the specific I/O drivers for I/O controllers. Figure 1.7 shows the architecture of it. We encapsulate the functions of an I/O drivers into separate hardware modules, e.g. Func 1. I/O low layer driver is the running place of I/O instructions.

In addition, we also provide an optional independent interrupt controller inside, i.e. Intc. While an interrupt sent from an I/O device is received by I/O low layer driver, the Intc will send the interrupt request to the I/O VMM, and then send the request to the guest VM directly.

Interrupt Handling

Another important feature of an I/O system is the processing capacity of interrupt - a shorter response time of the interrupt handling indicates a more powerful I/O system. We do not provide a special interrupt handling module in the BlueIO. In our approach, a clock level accurate interrupt handling can be achieved

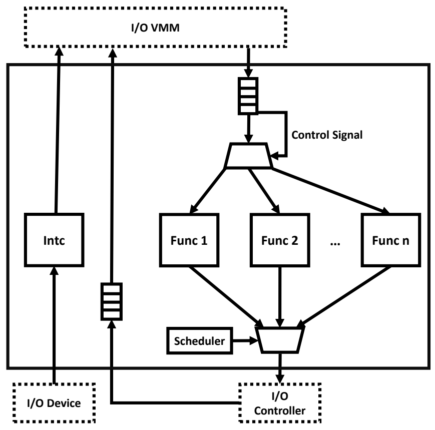


Figure 1.7: I/O Low Layer Driver

via the I/O controller illustrated in our previous paper[1].

[1]The paper will can be accessed soon

Chapter 2

Building an NoC Mesh with BlueIO

In this chapter, we will describe the implementation details of the BlueIO, including the inner structure and the external connection with a many-core system - Bluetile system[6]. All the source code can be accessed via link: *https://github.com/RTSYork/BlueIO*.

BlueIO is comprised by three partitions: BlueGrass, I/O VMM and I/O controllers, which are implemented via *Bluespec System Verilog*[1]. The interconnected system is illustrated in Figure 1.4. Actually, we integrate I/O VMM and I/O controllers in the actual implementation. All the source code of IP cores can be found in the the folder *IP_BlueIO*, respectively: *BG_Grass*, *BG_UART*, *BG_VGA* and *BG_Flash*. Users are able to execute the script *build.sh* in each folder to generate *verilog* files of these components.

2.1 Implementing the BlueGrass

The top level of the BlueGrass can be found as *BlueGrass.v*. This top level of BlueGrass in VIVADO is shown in Figure 2.1.

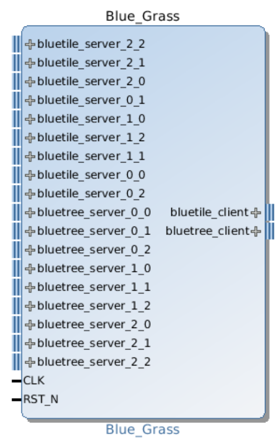


Figure 2.1: The Top Level of the IP Core - BlueGrass

As shown, the top level has 4 types of input ports, 2 types of output ports. Among these ports, the port *CLK* and port *RST_N* should be respectively connected to the clock source and the reset of the whole system. The ports named as *bluetree_server* and *bluetile_server* should be connected to the I/O VMM or I/O controllers directly. Additionally, the port *bluetile_client* should be connected to a router on the Bluetile system, which provides a communication interface between BlueIO and the processors mounted on the NoC. Finally, the port *bluetree_client* should be connected to the memory access module, providing a communication interface for the memory access.

2.2 Implementing the I/O VMM

As described in Section 1.3.2, I/O VMM contains multiple child I/O VMMs. The implementation of I/O VMM is actually the implementation of child I/O VMMs. To make the design clear, we encapsulate the child I/O VMMs, low layer I/O drivers and I/O controllers into the same components, which are *BG_UART*, *BG_VGA* and *BG_Flash*.

2.2.1 Child I/O VMM - BG_UART

BG_UART contains the child I/O VMM of UART, the low layer driver of the UART, and the UART controller. The top level of the BG_UART can be found as *BG_UART.v*. This top level of BG_UART in VIVADO is shown in Figure 2.2.

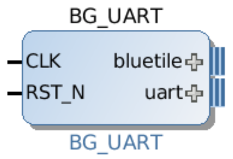


Figure 2.2: The Top Level of the IP Core - BG_UART

As shown, the top level has 4 types of ports. Specifically, the port *CLK* and port *RST_N* should be respectively connected to the clock source and the reset of the whole system. The port *bluetile_client* can be connected to a router on the Bluetile system or the BlueGrass, which provides a communication interface between *BG_UART* and the processors mounted on the NoC. Finally, the port *uart* is the physical pins to be connected to the external of the system, which is also known as *"tx"* and *"rx"*.

2.2.2 Child I/O VMM - BG_VGA

BG_VGA contains the child I/O VMM of VGA, the low layer driver of the VGA, and the VGA controller. The top level of the BG_VGA can be found as *BG_VGA.v*. This top level of BG_VGA in VIVADO is shown in Figure 2.3.

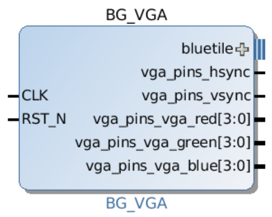


Figure 2.3: The Top Level of the IP Core - BG_VGA

As shown, the top level has 4 types of ports. Specifically, the port *CLK* and port *RST_N* should be respectively connected to the clock source and the reset of the whole system. The port *bluetile_client* can be connected to a router on the Bluetile system or the BlueGrass, which provides a communication interface between *BG_VGA* and the processors mounted on the NoC. Finally, the port *vga_pins_hsync*, *vga_pins_vsync*, *vga_pins_vga_red[3:0]*, *vga_pins_vga_green[3:0]* and *vga_pins_vga_blue[3:0]* are the physical pins to be connected to the external of the monitor.

2.2.3 Child I/O VMM - BG_Nor-Flash

BG_Nor-Flash contains the child I/O VMM, the low layer driver, and the controller of Nor-Flash (model: *S25FL128S*). The top level of the BG_Nor-Flash can be found as *BG_Nor-Flash.v*. This top level of BG_Nor-

Flash in VIVADO is shown in Figure 2.4.

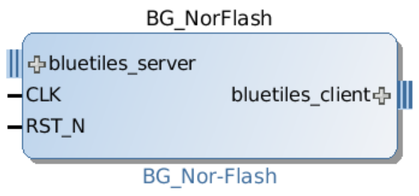


Figure 2.4: The Top Level of the IP Core - BG_Nor-Flash

As shown, the top level has 4 types of ports. Specifically, the port *CLK* and port *RST_N* should be respectively connected to the clock source and the reset of the whole system. The port *bluetile_client* can be connected to a router on the Bluetile system or the BlueGrass, which provides a communication interface between *BG_Nor-Flash* and the processors mounted on the NoC. Finally, the port *bluetile_server* is used to connected with the controller of the Nor-Flash, which is a SPI controller.

2.2.4 Customize a Child I/O VMM

Users are allowed to customize a child I/O VMM, and connect it to the BlueGrass. The customized child I/O VMM should provide a interface - *bluetiles_client*, which is used to connect to a router on the Bluetile system or the BlueGrass. *Bluetiles_client* also provides a communication interface between the customized I/O VMM and the processors mounted on the NoC. The packets transmitted via this interface should conform a unified format, which is defined in[1], and also shown in Figure 2.5.

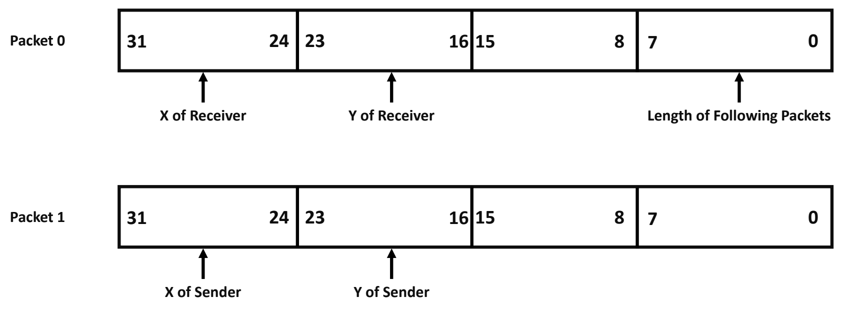


Figure 2.5: Formats of Transmission Packets

Users are able to use the bluetiles_client via invoking the library *Bluetiles*.

Listing 2.1: Step 2: Changing the macro inside Command Queue

```
// Invoking the library
import Bluetiles::*;

// Define the Interfaces
interface BlueClient bluetile_client;
   interface response = toPut(i_client);
   interface request = toGet(o_client);
endinterface
```

2.3 Connecting BlueIO to Bluetile System

Blueile system is s a Manhattan grid (mesh) interconnect for a network on chip (NoC) built using Bluespec System Verilog [6]. The interconnect enables a large number of CPUs and other processing elements to exchange messages in the form of network packets, the more ditals of Bluetile can be found in the website *https://rtslab.wikispaces.com/Bluetiles.*

Bluetile system implements a Manhattan grid interconnect. Two sorts of component are important:

- A router. Each router has five connections - each a bidirectional 32-bit channel of type "BlueBits" (defined in Bluetiles.bsv). Four of these are named North, East, South and West and are connected to other routers (or, at the edge of the grid, nothing at all). The fifth is named Home and connects to a local component. Each router has an address expressed in the form (x, y): these are Cartesian co-ordinates representing a grid location. The address is used when packets are routed. The router compares its own address against the destination address in a network packet, then directs the packet to one of the five interfaces accordingly.
- A local component. This could be a CPU, an I/O device, or a co-processor. It implements the other side of the Home connection, which allows it to send and receive messages over the network. BlueIO is one of the local component.

The source code related to Bluetile systems can be found in folder *System_Bluetile*, which is accesed via the link: *https://github.com/RTSYork/*.

2.3.1 Building Bluetile system

The source code of the router and local components, e.g., mutex, are written via *Bluespec System Verilog*. There are four steps are compulsory while building a Bluetile system: 1) Compiling the source code of each components to *verilog* files; 2) Encapsulating the *verilog* files as the Vivado IP cores; 3) Building the NoC via connecting routers; 4) Adding local components and connecting them on the NoC, including CPUs, UART ,etc.. The flow chart of these steps are shown in Figure 2.6.

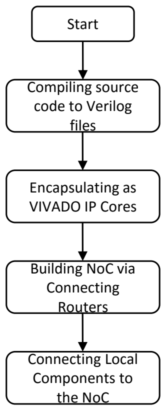

Start
Compiling source code to Verilog files
Encapsulating as VIVADO IP Cores
Building NoC via Connecting Routers
Connecting Local Components to the NoC

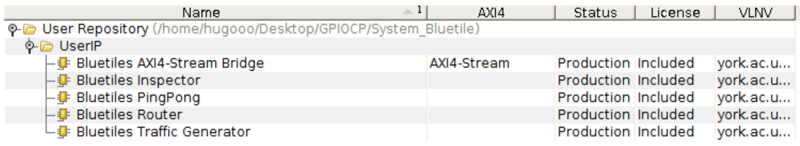
Figure 2.6: Flow of Building Bluetile System

Compile Bluespec System Verilog Files

To compile the source code of all the components, users can run the script *build_all.sh* located in the *System_Bluetile* folder.

Encapsulate Verilog Files as IP cores

Afterwards, users can run the script *launch_vivado.sh* to encapsulate the verilog fies as the Vivado IP cores. After the IP cores being built, users can invoke these compoents directly in Vivado. The IP cores are listed in Figure 2.7.



Name	AXI4	Status	License	VLNV
User Repository (/home/hugooo/Desktop/GPIOCP/System_Bluetile)				
UserIP				
Bluetiles AXI4-Stream Bridge	AXI4-Stream	Production	Included	york.ac.u...
Bluetiles Inspector		Production	Included	york.ac.u...
Bluetiles PingPong		Production	Included	york.ac.u...
Bluetiles Router		Production	Included	york.ac.u...
Bluetiles Traffic Generator		Production	Included	york.ac.u...

Figure 2.7: Encapsulated Bluetile System IP Cores

Building the NoC

We provide two methods for users to build a Bluetile NoC:

- Manual Building: Invoking the routers inside Vivado and connect corresponding communication ports.
- Automatic Building: Executing the provided tcl script to build a NoC with particular size. For example:

Listing 2.2: Building a 2*3 NoC via tcl script

```
bs::create_bluetiles_net_hier 2 3 BuleTile_NoC
```

After this script being executed, a size *2*3* NoC will be built, which named as *BlutTile_NoC*. Figure 2.8 illustrates this NoC.

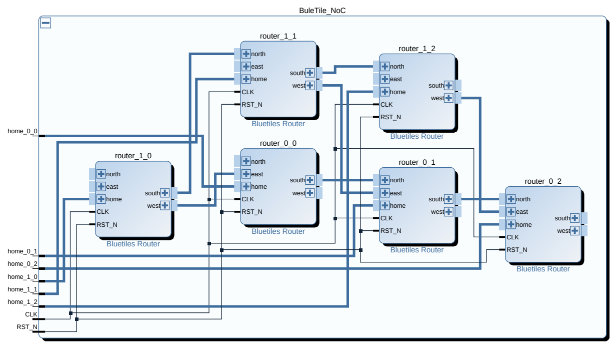


Figure 2.8: Size 2*3 BlutTile NoC

As it shown, the top level of a NoC has a clock signal port, a reset signal port and some home ports. Each home port belongs to a corresponding router, which can be used to connect local components.

Connecting Local Components

The communication method in Bluetile system are implemented via an communication interface provided by Bluespec System Verilog named ClientServer interface. The ClientServer interface provides two interfaces - Client interface and Server interface that can be used to define modules which have a request-response type of interface. In Bluetile system, we set the communication interfaces of all the routers are Server; and set the communication interfaces of all the local components are Client. Therefore, to connect local components, users are only required to connected the client interfaces of local components to the Server interfaces of the NoC. Figure 2.9 illustrates an example of connecting the IP Core *PINGPONG*[6] to the router whose coordinate is 0, 0) in the NoC.

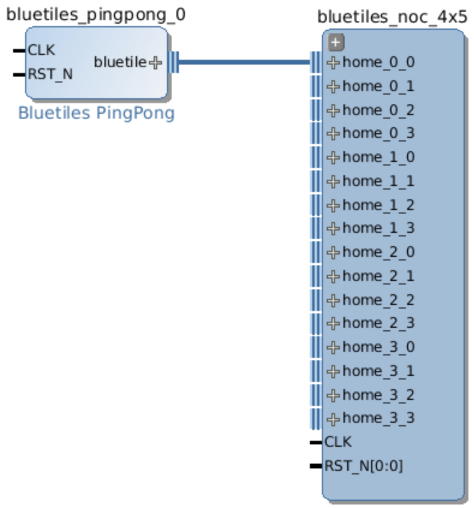


Figure 2.9: Connecting IP core PINGPONG to the NoC

Building a Bluetile System with Script

In the folder *zedboard_example*, we provide a script *create_project.tcl*, which can build an example Bluetile system with commonly used IP cores.

2.3.2 Connecting BlueIO to BlueTile system

Same as other local components, to connect a BlueIO, users are only required to connect the Client interface of the BlueIO to the Server interface on one of the router. Figure 2.10 shows an example that connect BlueIO on to the NoC.

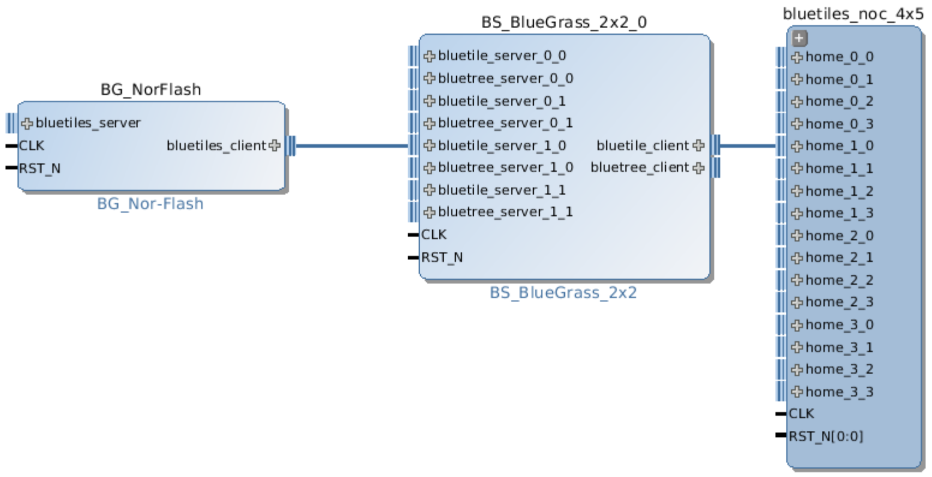


Figure 2.10: Conecting the BlueIO on the NoC

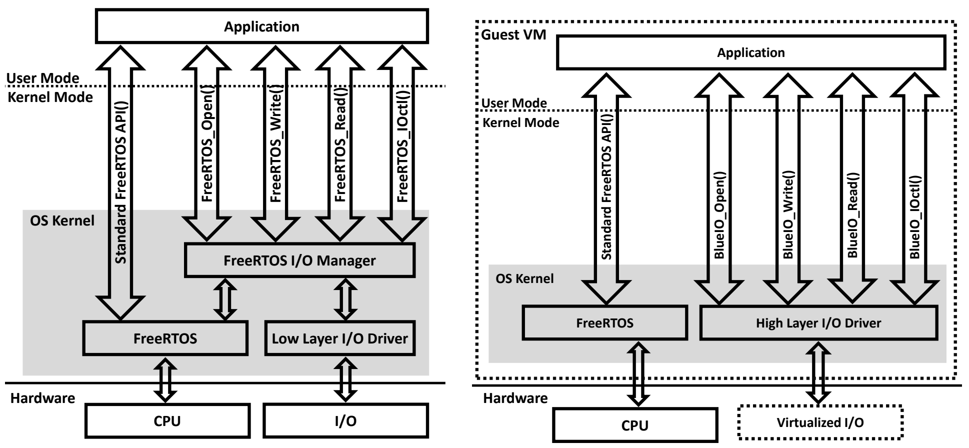
Chapter 3

Building the guest Virtual Machines (VMs)

3.1 Guest Virtual Machine (VM)

In our approach, one CPU has an individual guest virtual machine(VM). Due to the bare-metal virtualization is deployed (no host OS required), in each guest VM, a guest OS is able to execute in the kernel mode to achieve the full functionalities.

We also employ para-virtualization (modified OS kernel) to achieve a lighter software, that we build some high layer I/O drivers to replace the original I/O management. Currently, we have provided three modified OS to support the virtualization, which are the FreeRTOS[2], the ucosII[3] and the Xilkernel[4]. In Figure 3.1, we use FreeRTOS as an example to illustrate the modification of a guest OS kernel.



(a) FreeRTOS Kernel (b) Modified FreeRTOS Kernel in the Guest VM

Figure 3.1: FreeRTOS Kernels

As shown, compared with the original FreeROTS kernel (Figure 3.1(a)), the user application in the guest VM (Figure 3.1(b)) is able to access and operate the I/O via the high layer I/O drivers, which is independent of the central part of the FreeRTOS.

3.2 Running FreeRTOS

In our approach, we use the official version of the FreeRTOS kernel, which can be accessed via *http://www.freertos.org/*. Additionally, we also modify the official I/O module of FreeRTOS, the official I/O module can be accessed via *http://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_IO/FreeRTOS_Plus_IO.shtml.*

3.2.1 Building BSP of FreeRTOS

In the Github, the folder *OS_bsp* stores the the BSP of different OSs: *ucos ii(v1.41)* and *FreeRTOS(v9.0)*.

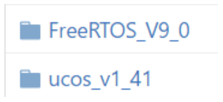
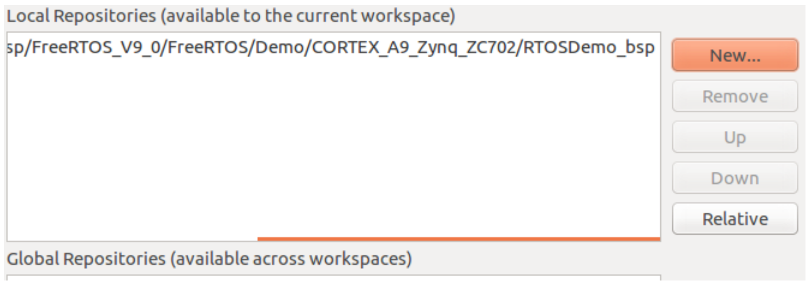


Figure 3.2: BSP for different OSs

To invoke a BSP into a project, users only need to click *Xilinx Tool*, *repositories* and add the BSP into it.


Local Repositories (available to the current workspace)

sp/FreeRTOS_V9_0/FreeRTOS/Demo/CORTEX_A9_Zynq_ZC702/RTOSDemo_bsp

Global Repositories (available across workspaces)

Figure 3.3: Add the BSP

After that, users can build a project with FreeRTOS via using the FreeRTOS BSP.

3.2.2 Adding the I/O Manager

Actually, as mentioned in Section3.1, no I/O manager is not required in our approach. However, we still provide a modified I/O manager for FreeRTOS, which can be access in the folder *FreeRTOS-Plus-IO*, which is shown in Figure3.4.

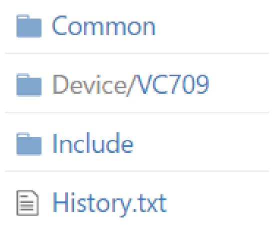
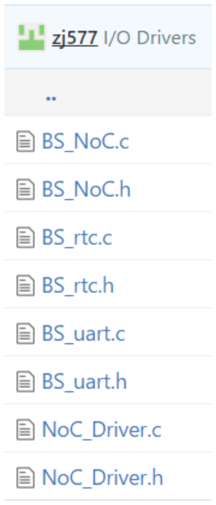


Figure 3.4: I/O manager in FreeRTOS

The folder *VC709* stores the drivers for the board VC709 (BG_UART, BG_VGA and BG_Flash) and etc..

3.2.3 Using High Layer I/O Drivers

We stores the High layer I/O drivers in the folder *I/O* drivers, which is shown in Figure 3.5.



zj577 I/O Drivers

..

BS_NoC.c

BS_NoC.h

BS_rtc.c

BS_rtc.h

BS_uart.c

BS_uart.h

NoC_Driver.c

NoC_Driver.h

Figure 3.5: I/O Drivers in FreeRTOS

Users can just invoke this high layer I/O drivers in the project directly. Specifically, *BS_NoC.c* is the driver for the BlueTile system; *BS_rtc.c* is the driver for the real-time clock; *BS_uart.c* is the driver for the BG_UART[2].

[2]More drivers will be added soon

Bibliography

[1] Bluespec inc. bluespec system verilog (bsv). `http://www.bluespec.com/products/`. Accessed September 27, 2015.

[2] Freertos official website. `http://www.freertos.org/`. Accessed September 27, 2015.

[3] ucos official website. `https://www.micrium.com/rtos/kernels/`. Accessed September 27, 2015.

[4] Xilinx official website. `https://www.Xilinx.com`. Accessed July 5, 2015.

[5] J. A. Landis, T. V. Powderly, R. Subrahmanian, A. Puthiyaparambil, and J. R. Hunter Jr. Computer system para-virtualization using a hypervisor that is implemented in a partition of the host system, July 19 2011. US Patent 7,984,108.

[6] G. Plumbridge, J. Whitham, and N. Audsley. Blueshell: a platform for rapid prototyping of multiprocessor nocs and accelerators. *ACM SIGARCH Computer Architecture News*, 41(5):107–117, 2014.

[7] J. Sahoo, S. Mohapatra, and R. Lath. Virtualization: A survey on concepts, taxonomy and associated security issues. In *Computer and Network Technology (ICCNT), 2010 Second International Conference on*, pages 222–226. IEEE, 2010.