UNIVERSITY OF YORK

Department of Computer Science

GPIOCP: GPIO Command Processor (v1.0)

Author:
Zhe Jiang

Supervisor:
Neil Audsley

September 8, 2016

Contents

Chapter 1

GPIO Command Processor(GPIOCP)

The GPIO Command Processor (GPIOCP) is built as a general purpose I/O controller, which connects a many-core system and I/O devices. The user application that is running on a processor can access and operate I/O devices via the GPIOCP. In our approach, the GPIOCP is connected to a NoC system called Bluetiles [2] configured in a 2D mesh topology. The structure of the system is shown in Figure 1.1.

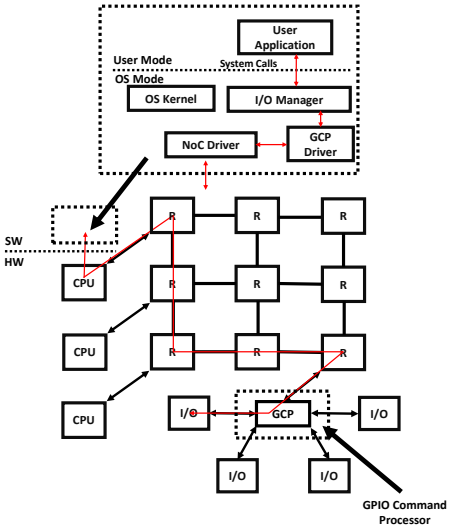

Figure 1.1: System model

Note that in the Figure 1.1, routers are labelled as "R" and GPIOCP is labelled as "GCP". 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 as the use of a shared bus can further reduce the I/O predictability.

In our system, a user application running on a CPU communicates with an I/O device by operating GPIOCP. The communications packets are transferred between the CPU and the GPIOCP 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.

1.1 Design Idea

The key idea of our approach is moving the I/O operations from user application to the GPIOCP, and bounding with timing constraints. In our approach, we provide a set of composable I/O control instructions for user applications, which are named as *GPIO sub-commands*. These GPIO sub-commands are divided into I/O control sub-commands and timing control sub-commands. User applications are allowed to encapsulate

these GPIO sub-commands as a new I/O control instruction (named as *GPIO command*) and store them into the Block RAM(BRAM) inside GPIOCP. For example, user application can encapsulate two GPIO sub-commands: "Read an I/O PIN with index x" and "Running the next sub-command at time T_y", as a new GPIO command - "At time 200, read I/O pin 3 ".These customized encapsulated GPIO commands can be a simple I/O reading or writing operation, as well as a complicated bus protocol. After the GPIO command is created, user applications can invoke the stored GPIO commands to control I/O devices via very brief requests, e.g. "Run GPIO command 5 on GPIO CPU 3". Inside GPIOCP, GPIO sub-commands are executed by an independent component - GPIO CPU. The GPIO CPU can be dynamically allocated to different processors with user bounded I/O pins. The number of GPIO CPU is generic, therefore GPIOCP is able to use multiple GPIO CPUs to run GPIO commands and operate different I/O devices in parallel.

1.1.1 GPIO Sub-commands

There are 9 GPIO sub-commands provided in our approach, which will be categorized as writing sub-commands, reading sub-commands and a control sub-command.

The functionality of GPIO writing sub-commands supported are shown below:

- 1. Executing the next writing sub-command at a specific time;
- 2. Setting a specific I/O pin to high/low;
- 3. Setting a group of I/O pins to specific values;
- 4. Do nothing in following specific clock cycles;

The functionality of the types of GPIO reading sub-commands supported are shown below:

- 5. Executing the next reading sub-command at a specific time;
- 6. Reading the value(s) of an specified I/O pin(s);
- 7. Reading the value(s) of an specified I/O pin(s) while a predefined I/O pin triggered high/low;
- 8. Do nothing in following specific clock cycles.

The functionality of the types of GPIO control subcommand supported is shown below:

- 9. Going back and rerunning from a previous GPIO sub-command;

The formats of GPIO sub-commands, and examples will be described in the Section 2.

1.1.2 GPIO Commands

Requesting GPIOCP via a GPIO command is very simple. Each GPIO command, which is stored in BRAM, has a unique index. User applications are only required to provide the index of the GPIO command and the index of GPIO CPU, to run this GPIO command in the GPIOCP. The formats of GPIO commands, and examples will be described in the Section 2.

1.2 Implementation a GPIOCP

In this section, we will describe the implementation details of the GPIOCP, including the inner structure and the external connection with a many-core system - Bluetile system[2]. All the source code can be accessed via link: *https://github.com/RTSYork/GPIOCP*.

GPIOCP is comprised by four modules: hardware manager, command memory controller, command queue and synchronization processor, which are implemented via *Bluespec System Verilog* [1]. The interconnected system is illustrated in Figure 1.2.

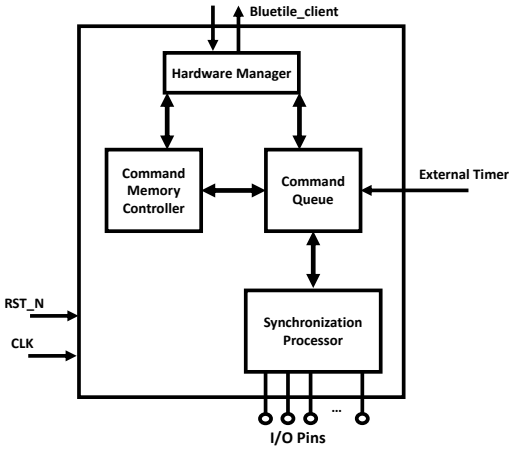

Figure 1.2: Top Level Architecture of GPIOCP

Corresponding to these four modules, the source code can be found in the the folder *IP_GPIOCP* respectively: *GPIOCMD_hw_manager.bsv*, *GPIOCMD_cmd_memory.bsv*, *GPIOCMD_cmd_q.bsv* and *GPIOCMD_cmd_processor.bsv*. Users can execute the script *build.sh* in *wrap* folder to compile the source files to *verilog* files of the GPIOCP. The top level of the GPIOCP can be found as *BS_GPIOProcessor.v*. This top level of GPIOCP in VIVADO is shown in Figure 1.3.

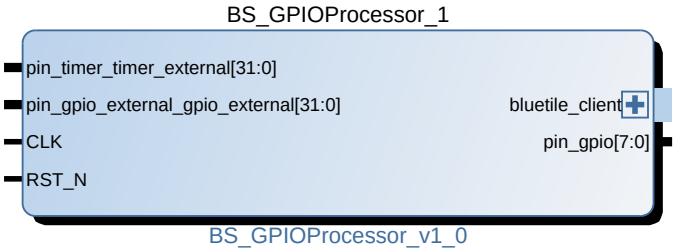

Figure 1.3: The Toplevel of the IP Core - GPIOCP

As shown, the top level has 4 input ports, 1 output port and 1 system interface. 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 port *pin_gpio_external_gpio_external[31:0]* and port *pin_gpio[7:0]* should be connected to the output and input GPIO pins of I/O devices respectively, which connects the GPIOCP and peripherals. The port *pin_timer_timer_external[31:0]* should be connected to the global timer of the whole system, whose resolution is 31-bit. This global timer provides a synchronization among GPIOCP and the whole system. Finally, the port *bluetile_client* should be connected to a router on the Bluetile system, which provides a communication interface between GPIOCP and the processors mounted on the NoC. In Bluetile system, all the communication are transmitted as packets. The format of the packets follows uniform rules illustrate in Section 2.

In the following subsections, we will describe the implementation details of each module, which is inside the GPIOCP.

1.2.1 Hardware Manager

Hardware manager is responsible for communicating with CPUs and allocating CPU requests to a right module: the command memory controller or the command queue. The source code of the hardware manager

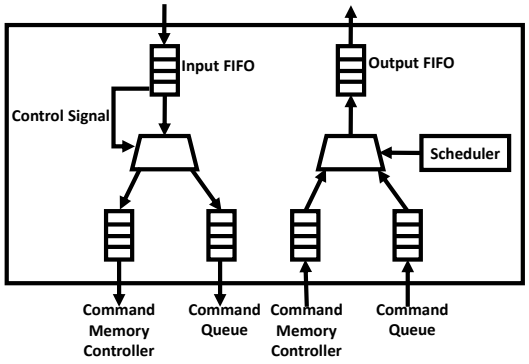

Figure 1.4: Architecture of Hardware Manager

is located in the folder *IP_GPIOCP*, which is named as *GPIOCMD_hw_manager.bsv*. The architecture of hardware manager is shown in Figure 1.4.

As shown, the hardware manager can be simply divided into two parts. The left part takes charge of allocating coming requests; and the right part takes charge of sending back messages from interior of GPIOCP to CPUs.

The left part of hardware manager is mainly comprised by an input FIFO, a multiplexer and two output FIFOs. There are two types of requests may be received by the GPIOCP: creating a new GPIO command (type 1) or invoking a ready-built GPIO command (type 2). The requests sent from CPUs will be firstly received by the input FIFO. After that, the multiplexer will allocate type 1 requests to the output FIFO which is connected to the command memory controller, and allocates type 2 requests to the output FIFO connected to the command queue.

Similarly, the right part of hardware manager is mainly comprised by two input FIFOs, a multiplexer, a output FIFO and a scheduler. The two input FIFOs are connected to the command memory controller and the command queue respectively, in order to receive the messages that need to be sent back to the CPUs. The scheduler controls the multiplexer to choose which input FIFO can transmit a message into the output FIFO. Specifically, we use round robin as the scheduling policy if both input FIFO is not empty.

1.2.2 Command Memory Controller

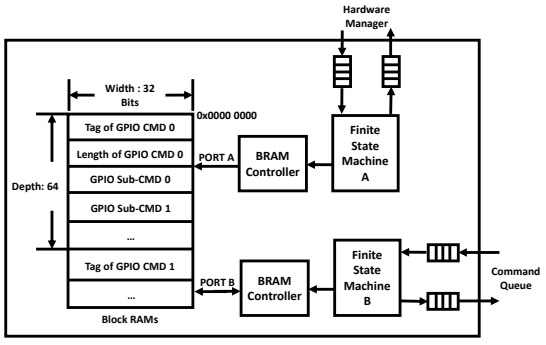

Figure 1.5: Architecture of Command Memory Controller

Command memory controller provides two functionalities: storing a new GPIO command into the BRAM; and translating a ready-built GPIO command to GPIO sub-command(s). The source code of the command memory controller is located in the folder *IP_GPIOCP*, which is named as *GPIOCMD_cmd_memory.bsv*.

The architecture of the command memory controller is shown in Figure 1.4. The architecture of command memory controller is shown in Figure 1.5.

Inside command memory controller, we adapted a 32 KB BRAM as the storage unit for GPIO commands, which is fixed to be divided into 128 pages. In our approach, we define the width of one page as 32 bits and the depth as 64. Each GPIO command is stored into an independent page, which owns the same index with it, e.g., GPIO command #59 should be stored into the page #59. All the stored GPIO commands are following uniform rules: 1) First 4 bytes stores the index of stored GPIO command; 2) Second 4 bytes stores the length of executing sub-commands; 3) All the stored GPIO sub-commands should follow uniform formats illustrated in Section 1.

Because a BRAM has dual ports, in our approach, we assigned different jobs to each port. Specifically, port A is connected to the BRAM controller belonged to finite state machine A (FSM A), which takes charge of receiving type 1 requests from hardware manager and writing a new GPIO command into the BRAM. Differently, port B is connected to the BRAM controller belonged to FSM B, which takes charge of receiving the index of a GPIO command from the command queue, then reading the particular contents of the GPIO command and send them back.

1.2.3 Command Queue

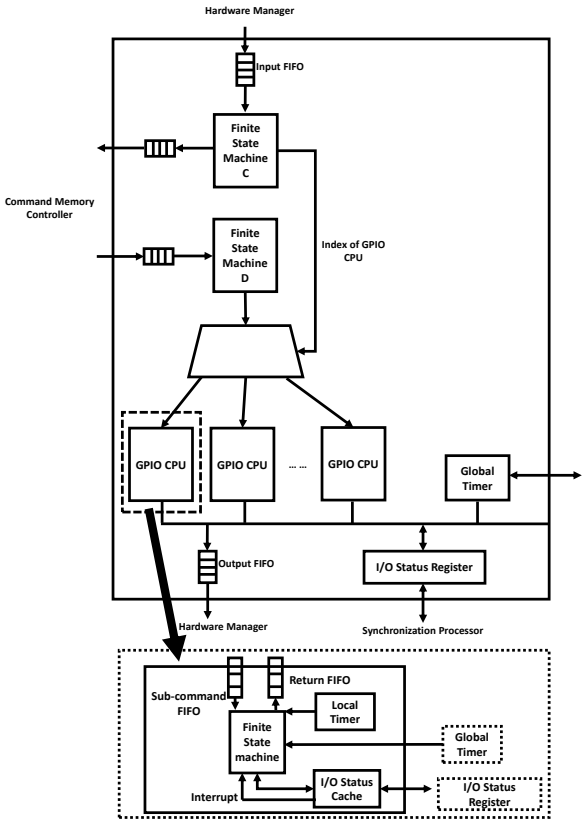

Figure 1.6: Architectur of GPIO Command Queue

Command queue is mainly responsible for the execution of GPIO sub-commands. The source code of the command queue is located in the folder *IP_GPIOCP*, which is named as *GPIOCMD_cmd_q.bsv*. The architecture of command queue is shown in Figure 1.6.

Inside the command queue, FSM C is designed to receive type 2 requests from hardware manager, and then translate the GPIO commands to GPIO sub-commands via sending the indexes of GPIO commands

to the command memory controller. The translated GPIO sub-commands sent back from the command memory controller will be allocated to different GPIO CPUs via a multiplexer.

GPIO CPU is designed as a simple finite state machine, which can guarantee the execution time of a GPIO command being predictable. Each GPIO CPU has a dedicated I/O status cache (4 bytes), which only stores the status of I/O pins belonged to this GPIO CPU. This dedicated cache synchronises its I/O status with a global shared register at a fixed frequency. The status of all I/O pins are stored in this shared register. Therefore, accessing to a shared register is avoided, which also avert the unpredictable synchronisation.

Inside command queue, a user is allowed to modify the number of GPIO CPUs via two steps.

The first step is adding GPIO CPUs into the command queue. Specifically, GPIO CPUs are encapsulated as finite state machine (FSM) - *fsm_cmd_q*. Users are allowed to add a GPIO CPU into the command queue via following functions.

Listing 1.1: Step 1: Adding GPIO CPUs into Command Queue

```
1  /* Declare GPIO CPU with an unique number */
2  Stmt fsm_cmd_q_2 =
3  seq
4   cmd_q(2);
5  endseq;
6  FSM fsm_cmd_q_2_FSM <- mkFSM(fsm_cmd_q_2);
7
8  /* Run the GPIO CPU */
9  rule cmd_q_2_FSM_rule;
10  fsm_cmd_q_2_FSM.start();
11 endrule
```

In Lisiting 1.1, the code between line 1 and line 6 shows the method of declaring a GPIO CPU; and the code from line 8 to line 11 demonstrates the process of starting this GPIO CPU. Note that, the number inside function *cmd_q()* is the index of the GPIOCPU, which should be unique.

In the second step, the macro defined the number of GPIO CPUs should be modified. Specifically The number of GPIO CPUs is defined as the macro *numb_cmdu* in the file *GPIOCP_CommandQueue.bsv*, which is default set as 2. For example, to build a command queue with *9* GPIO CPUs, the macro should be modified as the following Listing.

Listing 1.2: Step 2: Changing the macro inside Command Queue

```
1  Integer  numb_cmdu = 9;
```

After the GPIOCPU being added, users can follow the steps described in Section 2 to rebuild and encapsulate the GPIOCP.

1.2.4 Synchronization Processor

Synchronization processor is responsible for synchronising the values of I/O pins, which may be written by different GPIO CPUs and I/O devices. The source code of the synchronization processor is located in the folder *IP_GPIOCP*, which is named as *GPIOCMD_synchronization_processor.bsv*. The architecture of the synchronization processor is shown in Figure 1.7.

The synchronization processor is comprised by an I/O status buffer, a I/O status register, a fixed interval timer. a input FIFO and a output FIFO. Any modification the values of I/O pins received by input FIFO will be stored in the I/O status buffer, rather than being updated immediately. Fixed interval timer enables the synchronization between I/O status buffer and I/O status register every 5 clock cycle. Once the value of I/O status register is changed, the values of all the I/O pins will be sent back to the command queue via the output FIFO as well.

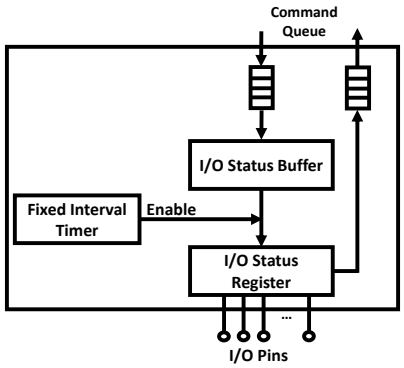

Figure 1.7: Architecture of Synchronization Processor

1.3 Connecting GPIOCP to Bluetile System

Blueile system is s a Manhattan grid (mesh) interconnect for a network on chip (NoC) built using Bluespec System Verilog [2]. 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. GPIOCP 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/*.

1.3.1 Building Bluetile system

The source code of the router and local components, e.g., UART and 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 1.8.

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 1.9.

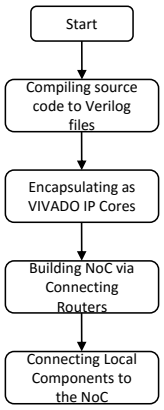

Figure 1.8: Flow of Building Bluetile System

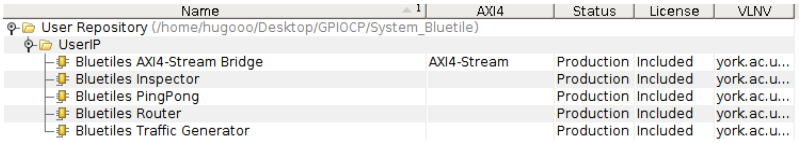

Figure 1.9: Encapsulated Bluetile System IP Cores

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 1.3: Building a 2*3 NoC via tcl script

```
1  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 1.10 illustrates this NoC.

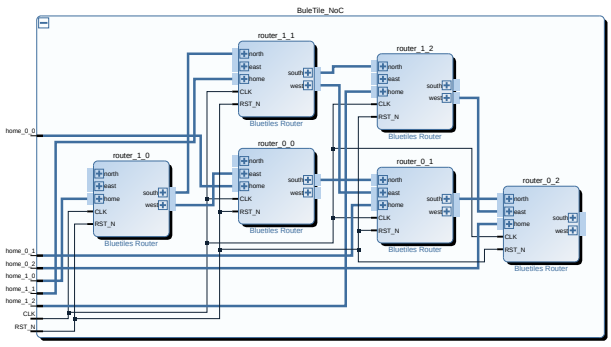

Figure 1.10: Size 2*3 BlutTile NoC

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 1.11 illustrates an example of connecting the UART to the router whose coordinate is 0, 0) in the NoC.

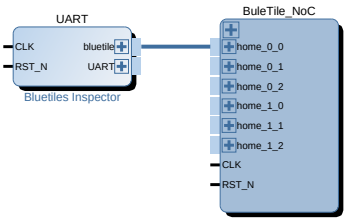

Figure 1.11: Connecting an UART to the NoC

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.

1.3.2 Connecting GPIOCP to BlueTile system

Following the steps in Chapter 2, users can build an GPIOCP shown as Figure 1.3. Same as other local components, to connect a GPIOCP, users are only required to connect the Client interface of the GPIOCP to the Server interface on one of the router, which is shown in Figure 1.12.

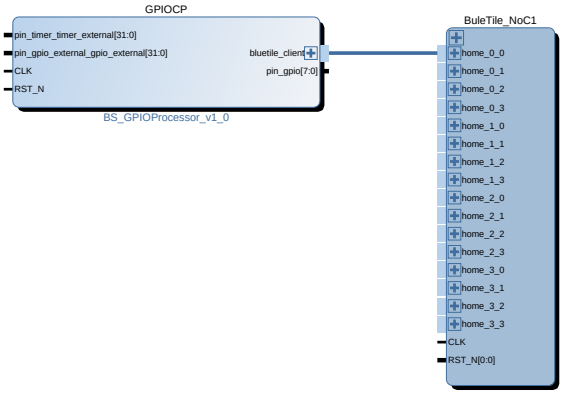

Figure 1.12: Conecting the GPIOCP on the NoC

Chapter 2

GPIOCP - Software Operations

In this chapter, we will describe how to control the GPIOCP in software, including the operation process, instructions formats and examples.

The software executed on the processors in Bluetile system can be written via C language. Processors are allowed to send 32-bit communication packets to the GPIOCP via our provided function: *Send.GPIOCP(u32 packet)*. The example of sending *0x00000001* to GPIOCP is shown in Listing 2.1.

Listing 2.1: Example of Sending 0x00000001 to GPIOCP

```
1  // Sending 0x00000001 to GPIOCP
2  Send.GPIOCP(0x00000001);
```

Similarly, processors can receive the response packets sent from GPIOCP via the ready-built function: *u32 Receive.GPIOCP()*. e.g. Listing 2.2.

Listing 2.2: Example of Receiving a Packet from GPIOCP

```
1  u32 rev;
2  // Receiving a packet from GPIOCP
3  rev = Receive.GPIOCP();
```

2.1 Operation Process of GPIOCP

The operations of GPIOCP are classified into two types: creating a new GPIO command (type 1) and invoking ready-built GPIO commands (type 2).

2.1.1 Type 1 Operations

Creating a new GPIO command via a processor requires 3 main steps. The fist step is sending a packet to GPIOCP, which notifies the operation type of the following packets and their length. The second step is sending the specific GPIO subcommands, which are used to encapsulate the new GPIO command. The last step is waiting for the response packets sent by GPIOCP. If the new GPIO command has been successfully built, the index of this GPIO command will be sent in the response packets. Otherwise, an error message will be contained in this packet. The working flow is shown in Figure 2.1.

2.1.2 Type 2 Operations

Invoking a GPIO command has 4 main steps. In the first step, processor is required to send the operation type of following packets and their lenght. In the second step, processor is required to apply a free GPIO CPU from GPIOCP. If the GPIO CPU has been successfully applied, the index of the GPIO CPU will be sent back via the response packets. Otherwise, an error message will be sent in the packets. In the third

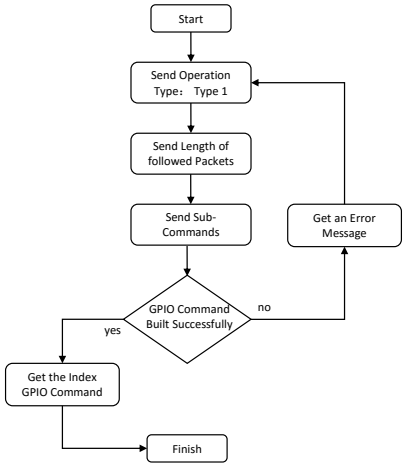

Figure 2.1: Working Flow of Creating New GPIOCMD

step, processor should bound the operated I/O pins to the applied GPIO CPU. A response packet will be sent back to the processor to tell if this operation has been operated successfully. In the final step, processor can send a packet to control GPIO CPU *"run GPIO Command #x on GPIO CPU #y"*. The working flow is shown in Figure 2.2.

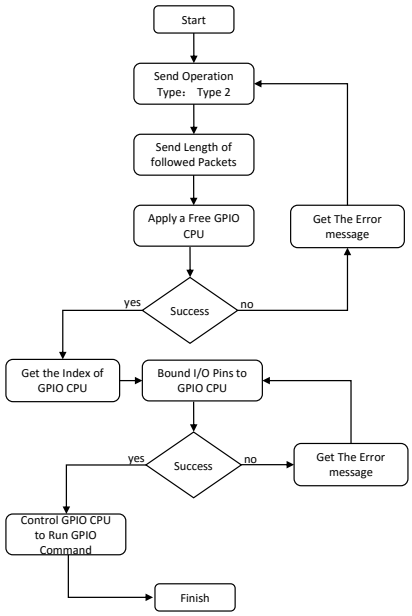

Figure 2.2: Working Flow of Invoking Ready-built GPIOCMD

2.2 Instruction Formats

As mentioned in Section **??** and Section 2, all the GPIOCP instructions are encapsulated as 32-bit packets. We classified these packets as transmission packets, operation type packet, GPIOC CPU control packets, GPIO sub-command packets and GPIO command packets. Following sections will describe the formats of

these packets.

2.2.1 Packet 0, Packet 1: Transmission Packets

While a series of packets being transmitted in Bluetile system, the first two packets are the transmission packets. The formats of transmission packets are defined by Bluetile system [2], rather than the GPIOCP. These transmission packets contain the information of sender's and receiver's coordinates on the NoC, as well as the length of the followed packets. Figure 2.3 illustrates these formats specifically.

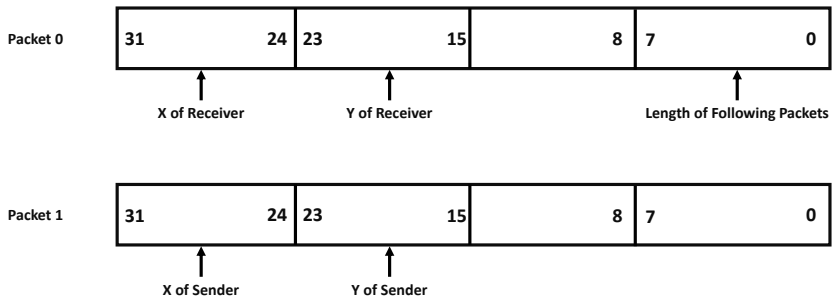

Figure 2.3: Formats of Transmission Packets

For example, if a processor with coordinate *(1, 2)* in the NoC is going to send 5 packets to the GPIOCP with coordinate *(3, 4)*, the C code can be written as Listing 2.3;

Listing 2.3: Example of Transmission Packets

```
1  /*  Packet 0  */
2  Send.GPIOCP(0x03040006); // "06" equals to payload + Packet 1
3
4  /*  Packet 1  */
5  Send.GPIOCP(0x01020000);
```

2.2.2 Packet 2: Operation Type Packet

While communicating with the GPIOCP, the packet following the transmission packets is the operation type packet, which points out the type of this series of packets. Specifically, in the packet, *bit 31 - bit 24* describes the transmitted direction of the following packets - from a processor to the GPIOCP or from the GPIOCP to a processor; *bit 23 - bit 16* shows the type of the operations - type 1 or type 2; *bit 7 - bit 0* tells about the length of the packets followed. Figure 2.4 illustrates these formats specifically.

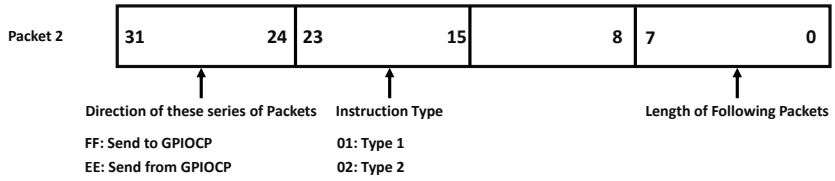

Figure 2.4: Format of Instruction Type Packet

For example, if a processor is designed to build a new GPIO command (type 1 operation) and the length of following packets is 6. The C code of operation type packet should be written as Listing 2.4;

Listing 2.4: Example of Instruction Type Packet

```
/*   Packet 2   */
Send.GPIOCP(0xFF010006);
```

2.2.3 Packet 3: GPIO CPU Control Packets

GPIO CPU control packets are only existed in type 2 operations. Before invoking a GPIO command, users are required to apply a GPIO CPU and bound I/O pins on it. These operations are provided by GPIO CPU control packets. Specifically, in the packet, *bit 31 - bit 24* describes the type of this GPIO CPU control - applying a GPIO CPU or bounding I/O pins to a GPIO CPU. If the operation is applying a GPIO CPU, *bit 23 - bit 0* are meaningless. If the operation is bound I/O pins to a GPIOCPU, *bit 23 - bit 16* should provide the index of the operated GPIO CPU; *bit 15 - bit 8* and *bit 7 - bit 0* should provide the index of the first bounded I/O pin and the index of the last bounded I/O pins, respectively. Figure 2.5 illustrates the formats specifically.

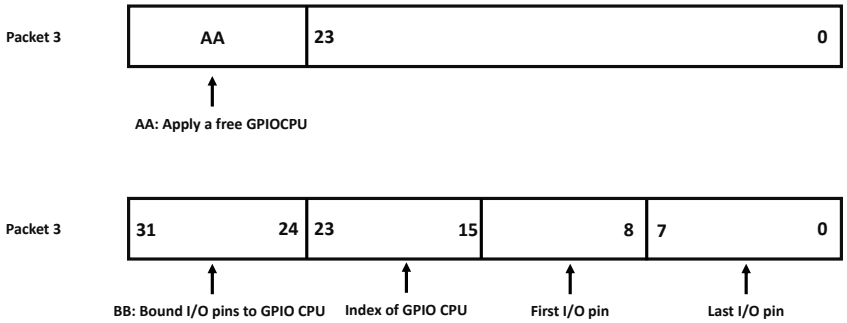

Figure 2.5: Format of Opeartion Packets

For example, if a processor is required to apply a GPIO CPU and bound 4 I/O pins on it, which starts from I/O pin *03*. The C code of GPIO CPU control packets should be written as Listing 2.5;

Listing 2.5: Example of GPIO CPU Control Packets

```
u32 index_GPIOCPU = 0x00000000;
/*   Packet 3   */
//  Applying a GPIO CPU
Send.GPIOCP(0xAA000000);

// Receive the Index of GPIO CPU
index_GPIOCPU = Receive.GPIOCP();

if (index_GPIOCPU != 0xFFFFFFFF) // Check if error
{
    /* Packet 3   */
    // Bound GPIO CPU to I/O pins
    Send.GPIOCP(0xBB000307 | (index_GPIOCPU << 16));
}
else
{
    return -1;
}
```

2.2.4 GPIO Sub-command Packets

The functionalities of nine GPIO sub-commands have been described in Chapter 2. GPIO sub-commands are encapsulated as 32 bit packets, the definition are described as follows:

- Bit24 - Bit31: The index of GPIO sub-command;
- Bit0 - Bit15: Operation parameters of GPIO sub-command, which provided different information for different sub-commands. Specifically, in sub-commands 1, 4, 5 and 8, they are timing information; in sub-commands 2, 3, 6, 7 and 9 they are the information of I/O operations.
- Bit16: The function type of GPIO sub-command - read/write; '1' stands for writing function and '0' represents reading function;

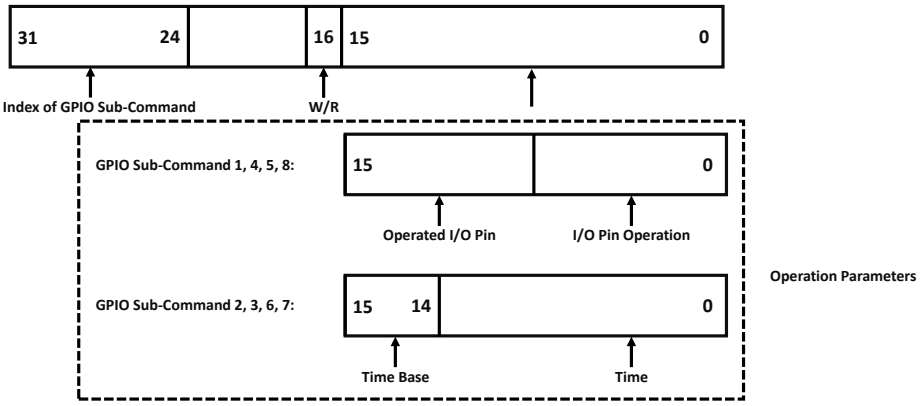

Figure 2.6: Format of GPIO Subcommand

In the following figures, we will describe the format of each GPIO sub-command.

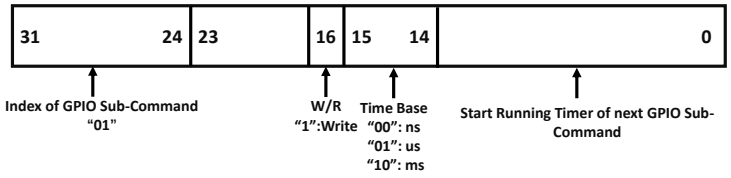

Figure 2.7: Format of GPIO Sub-command #1

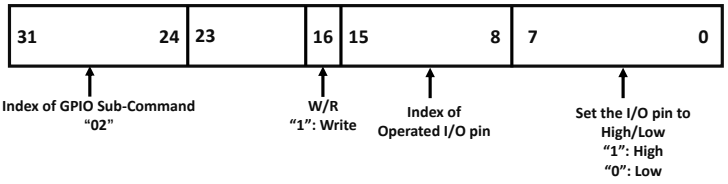

Figure 2.8: Format of GPIO Sub-command #2

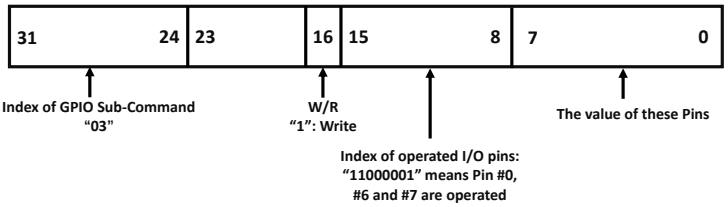

Figure 2.9: Format of GPIO Sub-command #3

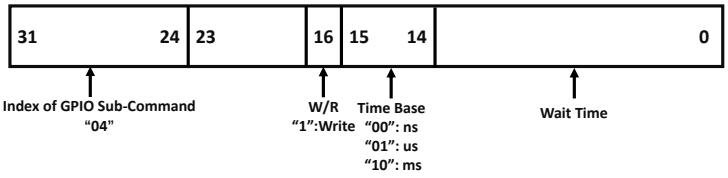

Figure 2.10: Format of GPIO Sub-command #4

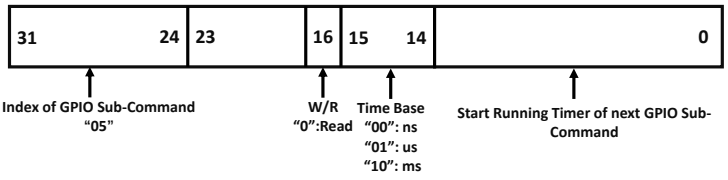

Figure 2.11: Format of GPIO Sub-command #5

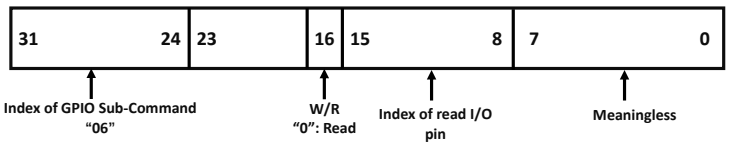

Figure 2.12: Format of GPIO Sub-command #6

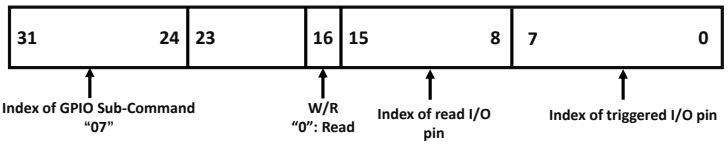

Figure 2.13: Format of GPIO Sub-command #7

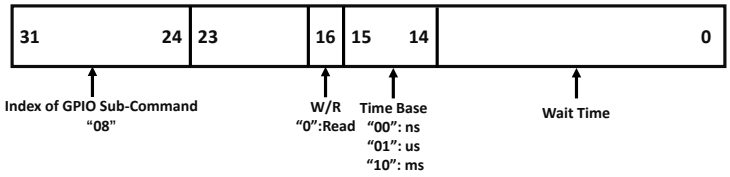

Figure 2.14: Format of GPIO Sub-command #8

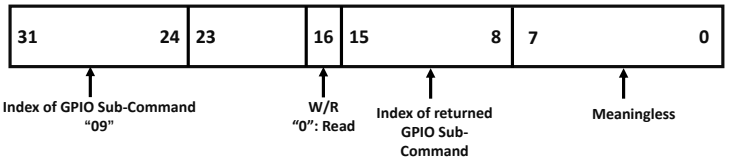

Figure 2.15: Format of GPIO Sub-command #9

Example

The following listing shows an example of how to group GPIO sub-commands as a GPIO Command that can generate a PWM signal on I/O pin #6 with 50% duty cycle.

Listing 2.6: Example of Grouping a GPIO Command via Sub-commandslabel

```
/* Packet 2  */
Send.GPIOCP(0xFF010005); // Builiding a new GPIO Command
                         // The length of the GPIO Command is 5

/* Build A GPIO command via GPIO sub-commands */
// Execute the next sub-command at time 200 ns
Send.GPIOCP(0x010100C8);
// Pull I/O pin #6 to high
Send.GPIOCP(0x02010601);
// Do nothing and wait for 20 ms
Send.GPIOCP(0x04018014);
// Pull I/O pin #6 to low
Send.GPIOCP(0x02010600);
// Do nothing and wait for 20 ms
Send.GPIOCP(0x04018014);
```

2.2.5 GPIO Command Packets

Requesting GPIOCP via a GPIO command is very simple. Each GPIO command, which is stored in BRAM, has a unique index. User applications are only required to provide the index of the GPIO command and the index of GPIO CPU, to run this GPIO command in the GPIOCP.

We define a GPIO command as a 32 bit instruction, the definition are described as follows:

- Bit24 - Bit31: The index of GPIO command;
- Bit8 - Bit15: The index of GPIO CPU which will execute this GPIO command.
- Bit16: The function type of GPIO command - read/write; '1' stands for writing function and '0' represents reading function;

For example, if a user application requests the GPIO CPU #3 to execute the GPIO command #2, it will be only required to send one on-chip packet[2] and the example C code is shown in the following listing.

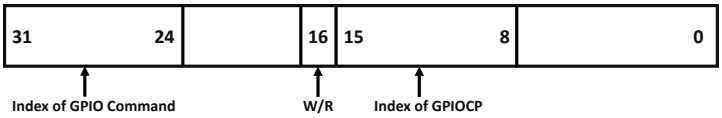

Figure 2.16: Format of GPIO Command

```
/* Packet 2   */
Send.GPIOCP(0xFF020001); //  Invoking a ready-built GPIO Command
                         //  The length of the GPIO Command is 1

/* Execute GPIO command #2 on GPIOCPU #3 */
Send.GPIOCP(0x02010300);
```

Chapter 3

Conclusion

In this report, we presented the concept of a programmable I/O controller (GPIOCP) with a clock cycle level granularity. It enables in many core systems with customized I/O control protocols, as well as operating multiple I/O devices in parallel with clock cycle level accuracy. In chapter **??**, we defined the timing-accuracy of I/O operations, and reviewed some related works: the I/O subsystems in Kalray MPPA-256, the PRU and the TPU. In chapter 1, we describes the key idea of our design is moving the I/O operations from user application to the GPIOCP with timing constraints, which eliminates the latency while transmitting an I/O request from a processor to the I/O devices. In chapter **??**·, we mainly shown the hardware configuration of the GPIOCP, including how to generate multiple GPIOCPU inside it and how to packaging the source code as an IP core. At last, we described the software operation GPIOCPU in Chapter 2, which includes the operation processes of two types operations and the instruction formats of the GPIOCP.

Bibliography

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

[2] 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.