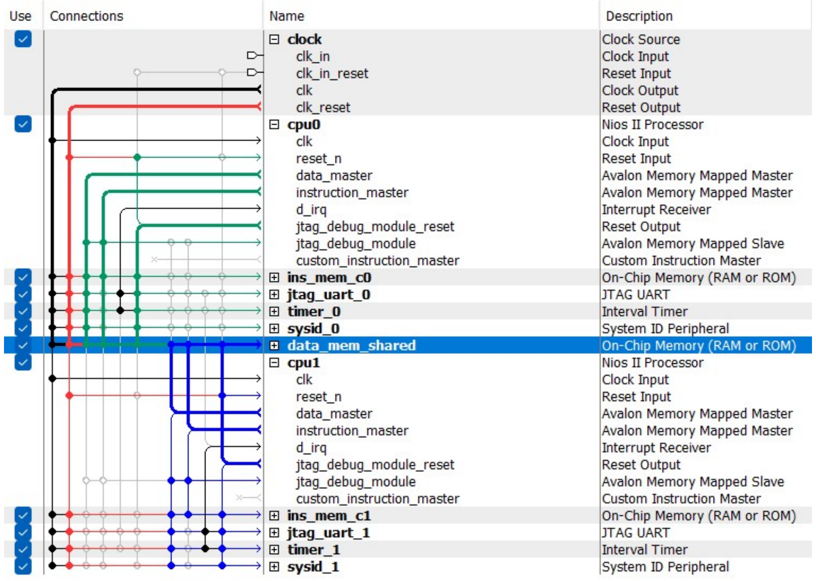
CO503 - ADVANCED EMBEDDED SYSTEMS
Practical 3 - Multi-Processor System-on-Chip (MPSoC) Design
Group 12 - E/19/057, E/19/063

Introduction

MPSoC designs are essential concept in modern embedded systems for optimizing performance and efficiency. And as such when we have multiple processors in a single design one important aspect, we need consider is how we implement the inter-processor communication. In this lab, we investigate how we can design a Multi-Processor System-on-Chip (MPSoC) using FPGA design tools and how we can implement inter-process communication using both a software and hardware FIFO system. The practical is divided into two parts. In Part 1, we design a producer-consumer application using shared memory for inter-processor communication. In Part 2, we replace the shared memory with a dedicated hardware FIFO queue for improved performance.

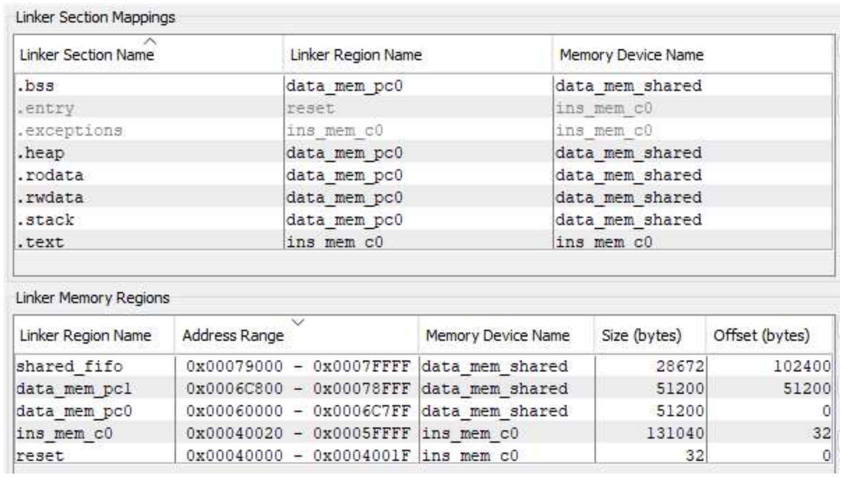
Part I: Shared Memory FIFO Design

Here we design our MPSoC in Qsys such that each CPU has its own on-chip memory core (128kB x 2) to store program instructions while another single on-chip memory core (128kB) implements the data memory that is shared between the two CPUs. One important design necessity was that we had to connect both instruction_master and data_master from each CPU to their corresponding memory devices irrespective of whether they stored instruction or data. Otherwise, we couldn't correctly run the program using the SoC due to some errors. Other than separate instruction memories, we also implemented separate JTAG UART cores (to communicate with each CPU independently using Serial) and System ID cores (To correctly identify each CPU when uploading the different programs).



Use	Connections	Name	Description
✓		⊟ **clock**	Clock Source
		clk_in	Clock Input
		clk_in_reset	Reset Input
		clk	Clock Output
		clk_reset	Reset Output
✓		⊟ **cpu0**	Nios II Processor
		clk	Clock Input
		reset_n	Reset Input
		data_master	Avalon Memory Mapped Master
		instruction_master	Avalon Memory Mapped Master
		d_irq	Interrupt Receiver
		jtag_debug_module_reset	Reset Output
		jtag_debug_module	Avalon Memory Mapped Slave
		custom_instruction_master	Custom Instruction Master
✓		⊞ **ins_mem_c0**	On-Chip Memory (RAM or ROM)
✓		⊞ **jtag_uart_0**	JTAG UART
✓		⊞ **timer_0**	Interval Timer
✓		⊞ **sysid_0**	System ID Peripheral
✓		⊞ **data_mem_shared**	On-Chip Memory (RAM or ROM)
✓		⊟ **cpu1**	Nios II Processor
		clk	Clock Input
		reset_n	Reset Input
		data_master	Avalon Memory Mapped Master
		instruction_master	Avalon Memory Mapped Master
		d_irq	Interrupt Receiver
		jtag_debug_module_reset	Reset Output
		jtag_debug_module	Avalon Memory Mapped Slave
		custom_instruction_master	Custom Instruction Master
✓		⊞ **ins_mem_c1**	On-Chip Memory (RAM or ROM)
✓		⊞ **jtag_uart_1**	JTAG UART
✓		⊞ **timer_1**	Interval Timer
✓		⊞ **sysid_1**	System ID Peripheral

And inside Nios 2 BSP configuration we had to correctly partition the shared data memory such that each CPU had its own private partition (50kB x 2) to store stack, heap, read-only data, read-write data mappings while the shared FIFO had its own partition (28kB).



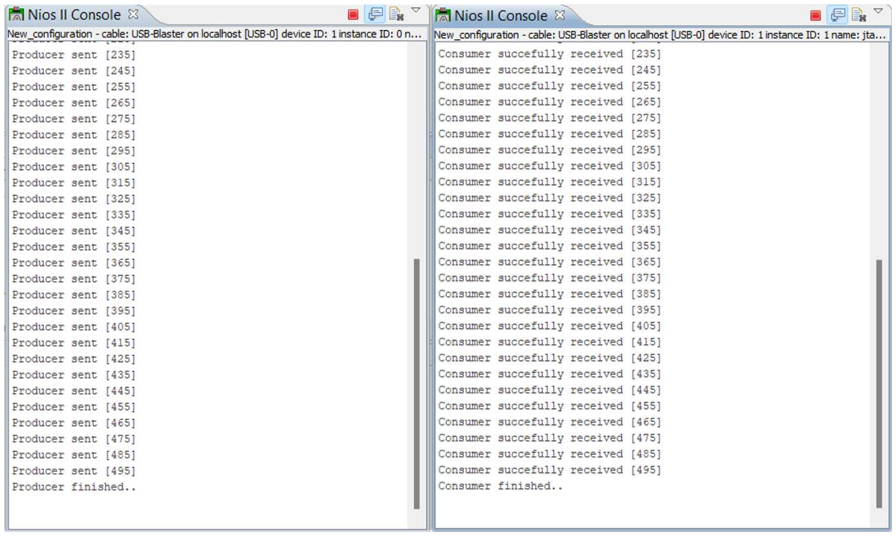
Linker Section Mappings

Linker Section Name	Linker Region Name	Memory Device Name
.bss	data_mem_pc0	data_mem_shared
.entry	reset	ins_mem_c0
.exceptions	ins_mem_c0	ins_mem_c0
.heap	data_mem_pc0	data_mem_shared
.rodata	data_mem_pc0	data_mem_shared
.rwdata	data_mem_pc0	data_mem_shared
.stack	data_mem_pc0	data_mem_shared
.text	ins_mem_c0	ins_mem_c0

Linker Memory Regions

Linker Region Name	Address Range	Memory Device Name	Size (bytes)	Offset (bytes)
shared_fifo	0x00079000 - 0x0007FFFF	data_mem_shared	28672	102400
data_mem_pc1	0x0006C800 - 0x00078FFF	data_mem_shared	51200	51200
data_mem_pc0	0x00060000 - 0x0006C7FF	data_mem_shared	51200	0
ins_mem_c0	0x00040020 - 0x0005FFFF	ins_mem_c0	131040	32
reset	0x00040000 - 0x0004001F	ins_mem_c0	32	0

And after we modify the given template code to implement the software FIFO using direct memory read and write operations, we can start both producer and consumer programs. Here we can see that, the data that is being enqueued to the FIFO by the producer program is dequeued by the consumer program.



Nios II Console
New_configuration - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 0 n...

```
Producer sent [235]
Producer sent [245]
Producer sent [255]
Producer sent [265]
Producer sent [275]
Producer sent [285]
Producer sent [295]
Producer sent [305]
Producer sent [315]
Producer sent [325]
Producer sent [335]
Producer sent [345]
Producer sent [355]
Producer sent [365]
Producer sent [375]
Producer sent [385]
Producer sent [395]
Producer sent [405]
Producer sent [415]
Producer sent [425]
Producer sent [435]
Producer sent [445]
Producer sent [455]
Producer sent [465]
Producer sent [475]
Producer sent [485]
Producer sent [495]
Producer finished..
```

Nios II Console
New_configuration - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 1 name: jta...

```
Consumer succefully received [235]
Consumer succefully received [245]
Consumer succefully received [255]
Consumer succefully received [265]
Consumer succefully received [275]
Consumer succefully received [285]
Consumer succefully received [295]
Consumer succefully received [305]
Consumer succefully received [315]
Consumer succefully received [325]
Consumer succefully received [335]
Consumer succefully received [345]
Consumer succefully received [355]
Consumer succefully received [365]
Consumer succefully received [375]
Consumer succefully received [385]
Consumer succefully received [395]
Consumer succefully received [405]
Consumer succefully received [415]
Consumer succefully received [425]
Consumer succefully received [435]
Consumer succefully received [445]
Consumer succefully received [455]
Consumer succefully received [465]
Consumer succefully received [475]
Consumer succefully received [485]
Consumer succefully received [495]
Consumer finished..
```

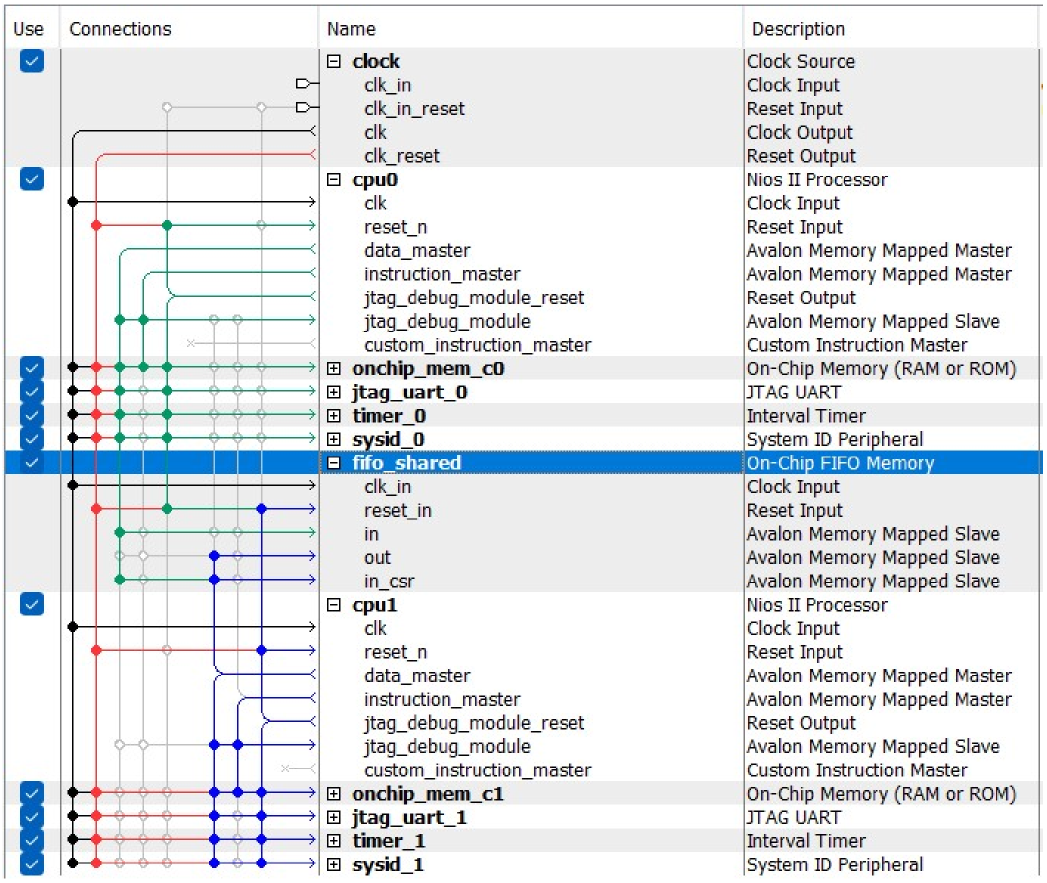
Several observations that we made here are,

- If we start the consumer program first, it will start and wait until data becomes available from the producer program as expected.
- But if we start the producer program first it will enqueue data until it runs out of memory inside the FIFO and then only wait until the consumer begins to reestablish operation.

But in the second scenario, due to some errors in our FIFO implementation or due to other pitfalls in asynchronous access to shared memory our consumer program most of the time went out of sync with the producer program, leading to receiving errors! Such errors can be remedied by implementing a proper FIFO data structure using a Linked List or read flags for every data element inside the FIFO.

Part II: Hardware FIFO Design

Here we replaced our software FIFO mechanism with a hardware implementation using a dedicated On-Chip FIFO Memory core. This approach greatly simplifies our system design well as our software implementation since the device come with its own support libraries for the Nios 2 development platform allowing us to get rid of unsafe direct memory access operations.



Use	Connections	Name	Description
✓		⊟ **clock**	Clock Source
		clk_in	Clock Input
		clk_in_reset	Reset Input
		clk	Clock Output
		clk_reset	Reset Output
✓		⊟ **cpu0**	Nios II Processor
		clk	Clock Input
		reset_n	Reset Input
		data_master	Avalon Memory Mapped Master
		instruction_master	Avalon Memory Mapped Master
		jtag_debug_module_reset	Reset Output
		jtag_debug_module	Avalon Memory Mapped Slave
		custom_instruction_master	Custom Instruction Master
✓		⊞ **onchip_mem_c0**	On-Chip Memory (RAM or ROM)
✓		⊞ **jtag_uart_0**	JTAG UART
✓		⊞ **timer_0**	Interval Timer
✓		⊞ **sysid_0**	System ID Peripheral
✓		⊟ **fifo_shared**	On-Chip FIFO Memory
		clk_in	Clock Input
		reset_in	Reset Input
		in	Avalon Memory Mapped Slave
		out	Avalon Memory Mapped Slave
		in_csr	Avalon Memory Mapped Slave
✓		⊟ **cpu1**	Nios II Processor
		clk	Clock Input
		reset_n	Reset Input
		data_master	Avalon Memory Mapped Master
		instruction_master	Avalon Memory Mapped Master
		jtag_debug_module_reset	Reset Output
		jtag_debug_module	Avalon Memory Mapped Slave
		custom_instruction_master	Custom Instruction Master
✓		⊞ **onchip_mem_c1**	On-Chip Memory (RAM or ROM)
✓		⊞ **jtag_uart_1**	JTAG UART
✓		⊞ **timer_1**	Interval Timer
✓		⊞ **sysid_1**	System ID Peripheral

The On-Chip FIFO memory device has a dedicated control register which we can use as a status register to monitor the FIFO memory usage and implement a backpressure mechanism.

```c
void WRITE_FIFO_1(int *buffer)
{
	// Wait if the fifo is full
	while (altera_avalon_fifo_read_status(CTRL_BASE,1<<0) != 0)
	{}

	// Write the data to FIFO
	altera_avalon_fifo_write_fifo(MEM_BASE,CTRL_BASE,(int)*buffer);
}

void READ_FIFO_1(int *buffer)
{
	// Wait if the fifo is empty
	while (altera_avalon_fifo_read_status(CTRL_BASE,1<<1) != 0)
	{}

	// Read the data
	*buffer = altera_avalon_fifo_read_fifo(MEM_BASE,CTRL_BASE);
}
```

As an additional benefit with this approach we also remedy the problem with consumer and producer programs going out of sync no matter in which order we run them in.