***CO503 - ADVANCED EMBEDDED SYSTEMS***

***Practical 3 - Multi-Processor System-on-Chip (MPSoC) Design***

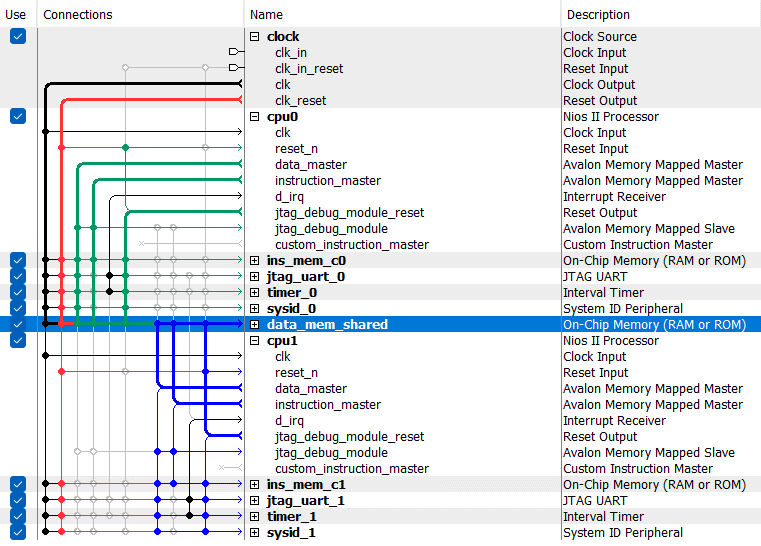
***Group 12 - E/19/057, E/19/063***

**Introduction**

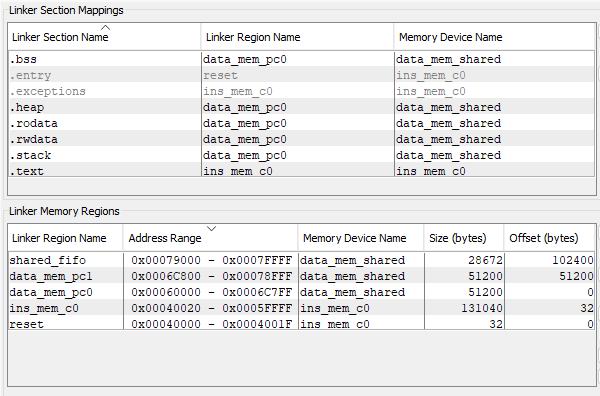
MPSoC designs are essential concept in modern embedded systems for optimizing performance and efficiency. And as such when we have multiple processors in a single design one important aspect, we need consider is how we implement the inter-processor communication. In this lab, we investigate how we can 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**

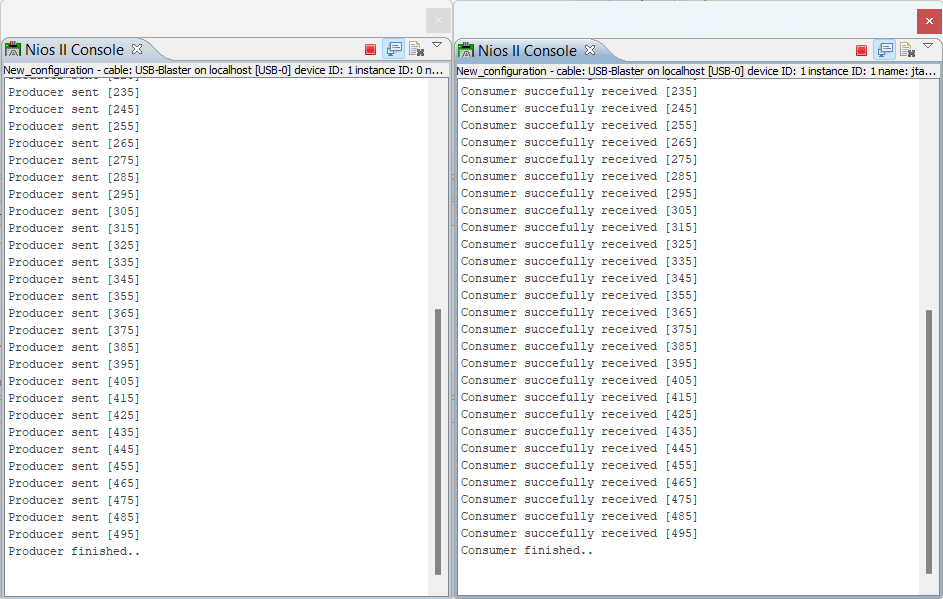
Here we design our MPSoC in Qsys such that each CPU has its own on-chip memory core (128kB x 2) to store program instructions while another single on-chip memory core (128kB) implements the data memory that is shared between the two CPUs. One important design necessity was that we had to connect both 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).



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).



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.



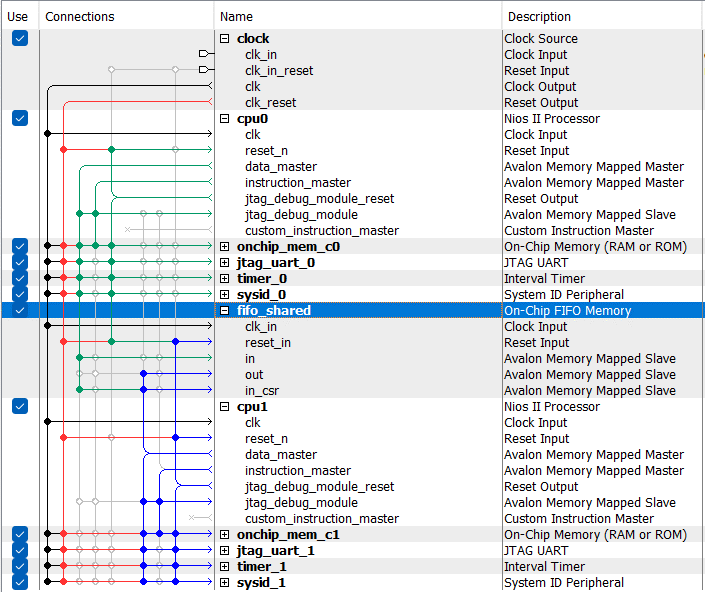
Several observations that we made here are,

* 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.



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.

**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.