**CO503 : Practical 3**

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

**E/17/154**

**Part 1: Producer-Consumer Applications on a Shared Memory Multi-Processor**

The first implementation of the Producer-Consumer Application on a shared memory was done using a software implementation of the FIFO method.

The producer and the consumer is created on the SoC by using two independent CPU cores along with separate systems.

CLK RESET INS INS INS DATA DATA

**Clock**

**System ID**

**Shared**

**Ins Mem**

**Data Mem**

**JTAG**

**Timer**

**Timer**

**JTAG**

**Data Mem**

**Ins Mem**

**CPU0**

**CPU1**

FIGXX: System Design

The system design is as shown above. Two systems on the same system on chip run on the same clock input. The system includes a single SystemID to provide a unique identification. Other than that the two systems are deigned with identical components of Timer. JTAG and memory.

**System Memory**

The total available on chip memory of the FPGA is 256KB. This was distributed so that we can create the required memory regions for the implementation. The memory part of the design is as follows,

Ins mem 🡪 individual memory spaces to store instructions

Data mem 🡪 individual memory space to store data

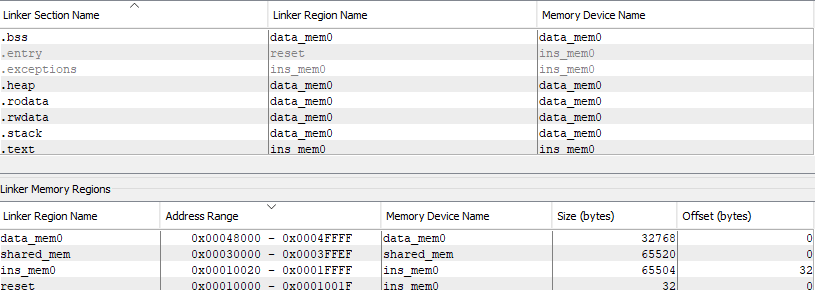
Shared mem 🡪 Single memory space to implement the FIFO

For this the memory was divided using the Qsys tool. However we can also define separate instruction memories and use a single memory element to store data of both private memory spaces of the CPUs as well as the shared memory space. Then the memory division can be defined in the linker script and be added as new linker region in the same memory device.

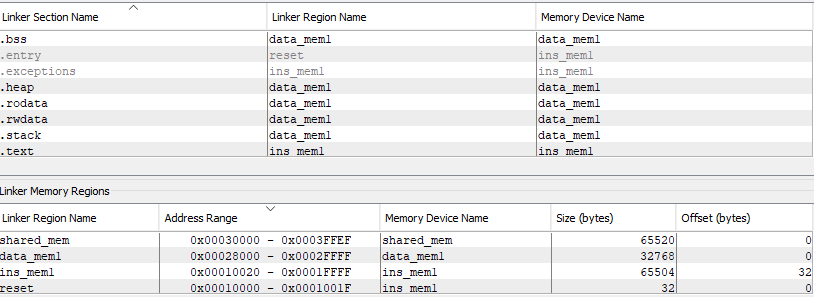
|  |  |  |  |
| --- | --- | --- | --- |
| Memory element | Address Begin | Address End | Size (bytes) |
| Ins\_mem0 | 0x0001\_0000 | 0x0001\_ffff | 65536 |
| Data\_mem0 | 0x0004\_8000 | 0x0004\_ffff | 32768 |
| Ins\_mem1 | 0x0001\_0000 | 0x0001\_ffff | 65536 |
| Data\_mem1 | 0x0002\_8000 | 0x0002\_ffff | 32768 |
| Shared | 0x0003\_0000 | 0x0003\_ffef | 65520 |

TABLEXX: Defined memory regions

But either way the memory regions will be the same as defined above and is assigned to relevant linker sections in the linker script of the project BSPs.



FIGXX: Linker script on cpu0



FIGXX: Linker script on cpu1

The regions divided between the instruction memory and the private data memory of the relevant CPUs.

|  |  |
| --- | --- |
| Linker section name | Memory Device name |
| .bss | Data mem |
| .entry | Ins mem |
| .exceptions | Ins mem |
| .heap | Data mem |
| .rodata | Data mem |
| .rwdata | Data mem |
| .stack | Data mem |
| .text | Ins mem |

TableXX : Linker Regions

**Software**

The software implementation was done using the provided NIOS software. As there are two CPUs for on the system we have to create two separate projects in two separate workspaces.

**Initialization**

Before operation the FIFO must be initialized with correct addresses to the relevant pointers and correct flag values. The FIFO\_1.h holds basic information as the base memory address (a begin address inside the shared memory space), capacity(depth of the FIFO), unit size(size of single data entry), startp (offset to the start of the fifo relative to the memory base address)

|  |  |
| --- | --- |
| Definitions | Values |
| MEM\_BASE | 0x00020048 |
| CAPACITY | 20 |
| UNIT\_SIZE | 0x4 |
| STARTP | 0xC |

The startp actually leaves out a space of 3 words in the shared memory space. The 3 words are for the full, empty and count variables.

full 🡪 flag to keep track of whether FIFO is full or not  
empty 🡪 flag to keep track of whether FIFO empty or not  
count 🡪 variable to keep track of the current available data units

The pointers defined are,  
 writep 🡪 offset pointer to next space to write data element.  
 readp 🡪 offset pointer to the next element to be read(popped out)  
 fullp 🡪 offset pointer to access full flag  
 emptyp 🡪 offset pointer to access empty flag

On initialization the writep and readp are assigned the start of the FIFO (STARTP). FULLP is assigned 0x0 as it is at the beginning of the memory space. Emptyp is a UNIT size from fullp and countp unit size from emptyp.

The values at beginning   
  
 full 🡪 0x0  
 empty 🡪 0x1  
 count 🡪 0x0

FIFO

COUNT

EMPTY

FULL

FIGXX : Shared Memory Arrangement

**Producer**

Main operation of the producer is to write to the FIFO structure in the shared memory region when space is available. The function defined is WRITE\_FIFO\_1 and a reference to the data to be written is passed as argument. At the beginning of the function we check for the full flag and wait if the FIFO is full. If not we can write data to the current address of the pointer writep. Once written writep is incremented with unit size. Here we check for the modulus of the writep with regards to CPACITY \* UNIT\_SIZE. This creates a operation of writing to the FIFO even after exceeding the defined space for the implementation by going back to the beginging. We also increment the count variable after writing in to the FIFO. Finally a check is run for count == CAPACITY && count == 0x0 to update full and empty flags.

**Consumer**

Main operation of the consumer is to read from the FIFO structure in the shared memory region when data is available. The function defined is READ\_FIFO\_1 and a reference to the data pointer to return is passed as argument. At the beginning of the function we check for the empty flag and wait if FIFO is empty. If not we can read data from the current address of the pointer readp. Once read readp is incremented with unit size. Here we check for the modulus of readp with regards to CAPACITY \* UNIT\_SIZE. This creates a operation of reading from FIFO even after exceeding defined space for the implementation by going back to the beginging. We also decrement the count variable after reading from the FIFO. Finally a check is run as before to update the full and empty flags.

**Part 2: Hardware FIFO**

The second part of the lab is focused on replacing the previously implemented software FIFO using a hardware FIFO core in the Qsys design. Keeping the CPU specific memory (Data and Ins) the shared data memory is replaced. In this implementation since its hardware implemented we do not have to handle pointers to keep track of the FIFO as before. Instead all operation on the FIFO are handled by the API functions provided. For our purpose of producer consumer operation we use the status register to check for FIFO status and the read and write functions.

The implementation was done by removing the shared memory portion from the Qsys design and replacing with FIFO core. The bus connection for the new component is as follows,

Clk in -> clock  
 rest\_in -> reset  
 in -> write memory   
 in\_csr -> status memory  
 out -> read memory

|  |  |
| --- | --- |
| Name | Connection |
| clk\_in | Clock input to the system |
| reset\_in | clock reset of the system |
| in | Write memory (producer) |
| in\_csr | Status memory |
| out | Read Memory (consumer) |

TableXX : FIFO core connections

The definitions in the FIFO\_1.h file are as follows,

|  |  |  |
| --- | --- | --- |
| Definitions | Values | Description |
| OUTPUT\_FIFO\_OUT\_FIFO\_DEPTH | 64 | Depth of the FIFO |
| ALMOST\_EMPTY | 2 | Almost empty threshold |
| ALMOST\_FULL | OUTPUT\_FIFO\_OUT\_FIFO\_DEPTH-5 | Almost full threshold |
| CTRL | 0x00000000 | Control/status memory |
| IN\_BASE | 0x00031030 | input memory address |
| OUT\_BASE | 0x00031030 | Output memory address |
| CAPACITY | 64 | Capacity of FIFO |
| UNIT\_SIZE | 32 | Unit size of FIFO element |

TableXX : Definitions

Once the connections are in place it is a matter of calling the correct function of the API for out required operation.

**altera\_avalon\_fifo\_init(CTRL,0x000000,ALMOST\_EMPTY,ALMOST\_FULL);**

Used to initialize the FIFO. As agruments we pass in the CTRL(control memory), interrupts(none for this practicle), almost empty and almost full values for the FIFO.

**altera\_avalon\_fifo\_read\_status(CTRL,mask)**

The function is used to read the status register. As parameters along with the control memory we pass in a mask specifying the flag we want to read. This is used to check for the full and empty flags.

Full = 0x01  
 Empty = 0x02

**altera\_avalon\_fifo\_write\_fifo(IN\_BASE,CTRL,\*buffer)**

Function to write to the FIFO core called after checking for full flag.

**altera\_avalon\_fifo\_read\_fifo(OUT\_BASE,\*buffer)**

Function to read from the FIFO core called after checking for empty flag.

During operation when producer is run first it writes data into to the FIFO until the FULL flag is set. Then the operation halts and waits for the FIFO to be free. Running the consumer would clear the FULL flag and keep reading written data thus completing the memory transaction.

If the consumer is run first it waits for a readable data element in the FIFO. At this instance running the producer would show how the data is read as it comes in and does not halt again until all data is finished reading.

**Performance Comparison**

Timer calculation for software implementation -> 40ms