Addressing Shared IP Instances in a Multi-CPU System Using Fabric Switch

Priyanka Gharat (priyanka@siliconinterfaces.com), VLSI Engineer
Avnita Pal (avnita@siliconinterfaces.com), VLSI Engineer
Sastry Puranapanda (sastry@siliconinterfaces.com), VLSI Engineer

Silicon Interfaces®
a software and vlsi design center™

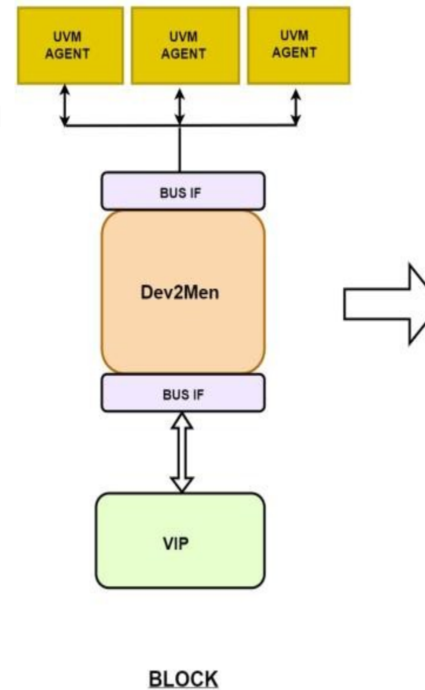
accellera
SYSTEMS INITIATIVE

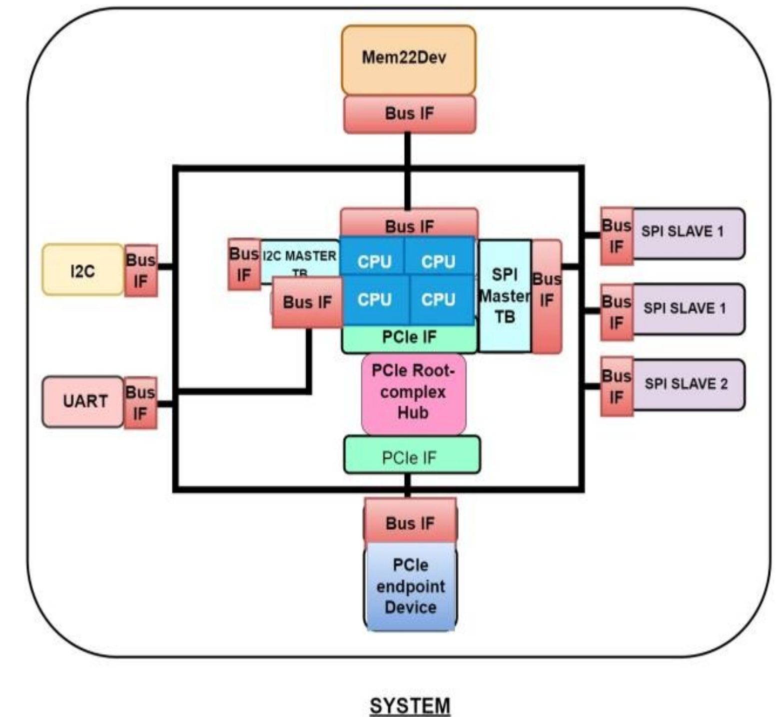
Abstract

- The objective of this presentation is to utilize the capabilities of PSS-based DSL language features, such as byte addressability, resource sharing-locking, multiple component instances, and true parallel scenarios.
- The purpose is to address problems related to data integrity and bottlenecks in multi-core processors that communicate with multiple devices.
- This is achieved by implementing a learning heterogeneous switch fabric with address storage and translation. With this feature, cores can communicate with any endpoint device on one of the switch ports while leaving other ports free for communication, thus enabling parallel traversal operations.
- The result is a method that maintains parallelism while ensuring data integrity through resource sharing/locking.

Challenges

- In modern System-on-Chip (SoC) design, data flow between various devices within the SoC occurs through the bus interface.
- The bi-directional channels enable data to flow between the memory and multiple instances of the same device or different devices.
- The ability to concurrently exchange data between multiple devices while maintaining data integrity is a crucial challenge in SoC design.





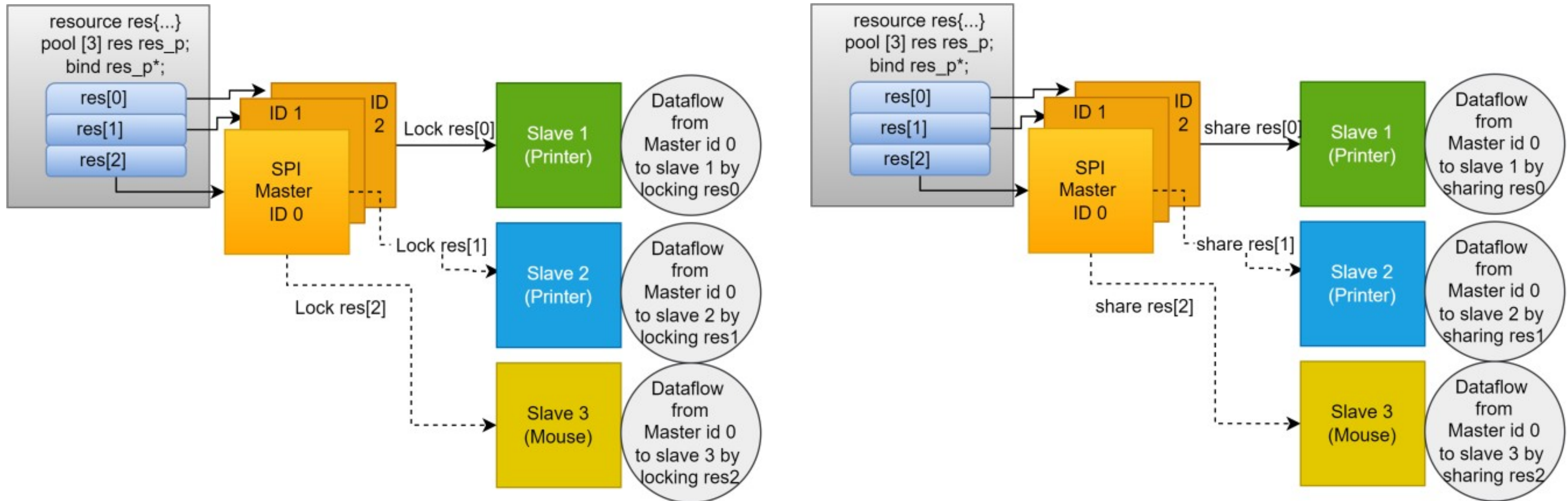
- To address this challenge, the Portable Stimulus Standard (PSS) introduces the powerful feature of component instances, which enables the creation of copies of IPs that can be reconfigured based on the selected component.
- These component instances can have exclusive or shared access to the bus interface, allowing for concurrent data exchange. The resources are mapped to agents in the UVM target test-bench, enabling effective testing of the SoC design.
- In this paper, the solution is implemented using **component instance array**, address region/space, byte addressability and discuss how they facilitate efficient data exchange and address challenges in SoC design.

What Is Fabric Switch?

- A multi-core CPU SOC requires access to various IPs (Communications, Networking, Audio/Video, and Memory) through a common Bus Interface. When the Bus and Channels intersect, they form a matrix-like structure known as a fabric or mesh.
- The Fabric Switch is like a crossbar exchange wherein data may flow from any one upstream input port to any one downstream output port.
- The idea is that we have a matrix connectivity from upstream to downstream. The ports are bi-directional so upstream/downstream and input/output are only notional.
- The upstream may be connected to multi-Core and downstream ports may be connected to several different or even same sets of devices.

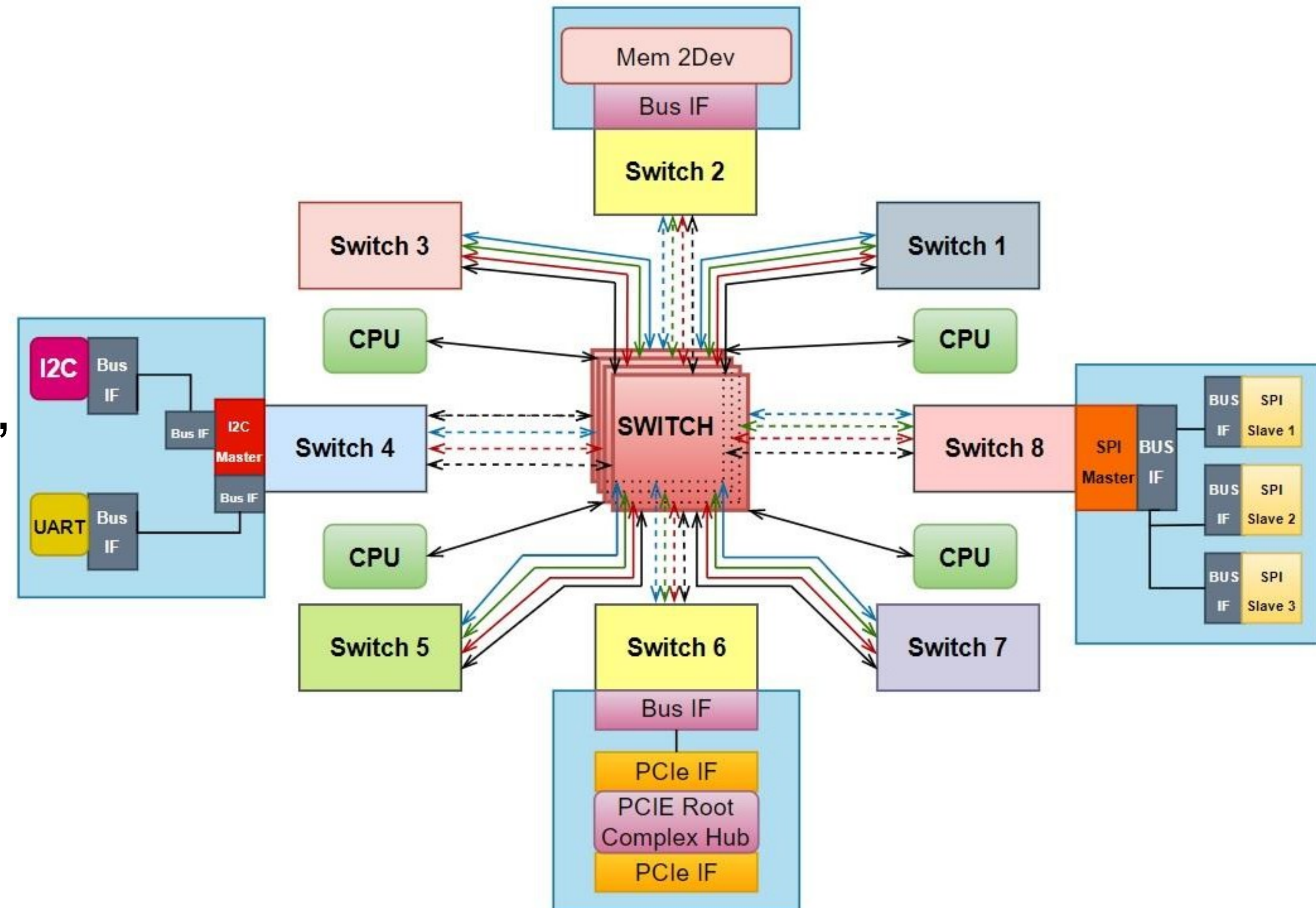
Data flow is managed through parallel channels to maintain data integrity. CPU signals to share data between **resource pools** when transferring data from memory to a device. Figure 2 illustrates the sharing/locking process.

Channel resource may be locked or Device ID matched for data intended for different devices. Endpoint device claims address space location for secure data storage. Common BUS IF used for multiple IPs via data flow object.



Implementation Using Fabric Switch

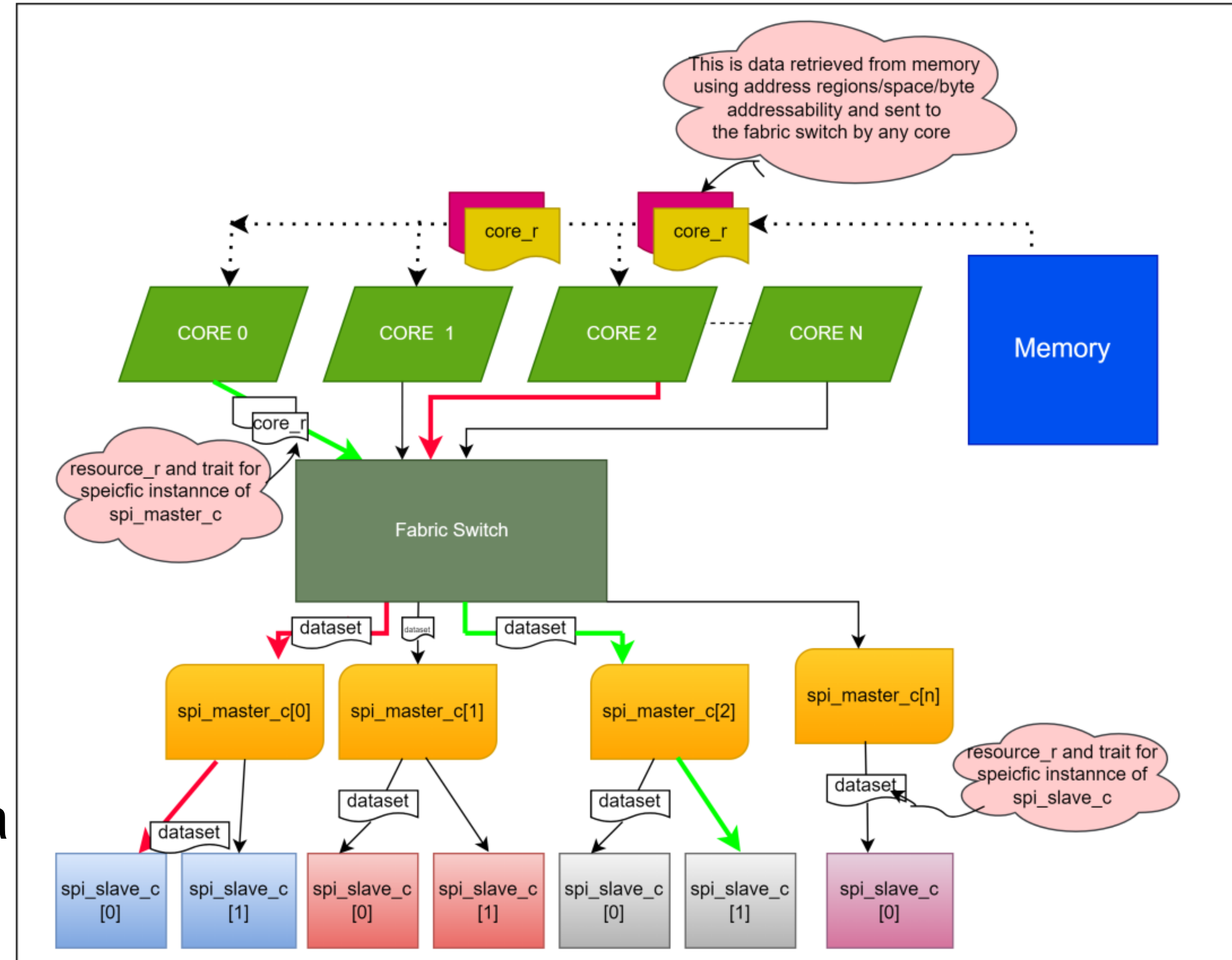
- Concurrent data exchange between multiple devices while ensuring data integrity is a crucial challenge in SoC design, as it can lead to bottlenecks, data corruption, and loss.
- When multiple CPUs attempt to write or read through the same shared bus, there is a risk of address and data corruption.
- This paper aims to utilize PSS/DSL language features and specifically enhance SPI with Master Slave communication for a multi-Core environment addressing multi-Devices.



Pictorial Representation of Created Scenario

- Implementation has been written for multi-Cores with multiple SPI slaves. Each SPI master has been configured as regions and an address space, byte addressability, and memory has been allocated using PSS version 2.0 DSL language construct have unique traits and claimed using traits.

- The component tree has been used to resolve the address space and create scenarios such as write data/read data performed parallel and considering fabric switch the pipelining delay.



Evidence for Implementation

- The following source code shows the transfer of data to different slave from master using address space, region, and byte addressability which are the construct of PSS version 2.0.

- This data is part of resources are being addressed from the pool created at the top-level component and the different multiple instances of master and slave get bind from those pool where they are being shared and locked during a particular transfer.

```
package pkg{
        resource core_r {rand bit [3:0] data;}
        resource spi_dataset_r{
                rand bit [31:0] data;
        }
}
//Struct definition for Trait which is the device ID.
struct TRAIT{rand int device_ID;};

enum cache_attr_e {PCI, PCIe, AXI, WB};
enum security_level_e {level0, level1, level2, level3};

struct mem_traits_s:TRAIT {
      rand cache_attr_e ctype;
      rand security_level_e sec_level;
}

struct addr_region_base_s {
        bit[64] size;
};

//Since Tools still are not supporting PSS 2.0 we
 are using global address region.
//struct addr_region_s <struct TRAIT = null_trait_s>

struct addr_region_s:addr_region_base_s
{
        mem_traits_s trait;
};

//struct transparent_addr_region_s <struct TRAIT = null_trait_s>

struct transparent_addr_region_s:addr_region_s
{
        bit[64] addr;
};
struct fb_trait_s {}
```

- By leveraging the powerful features provided by PSS/DSL, a complex task has been successfully implemented with ease. One of the major challenges encountered when dealing with multiple CPUs in an SOC or sub-system is the utilization of multiple channels to access the same IP.

- To address this, we have implementation, we have created multiple instances of a component and allocated them to several cores. For instance, core_c[0] communicates with slave1 (printer), core_c[1] communicates with slave2 (printer), and core_c[2] communicates with slave3 (mouse).

```
//===========================
//Fabric Switch
//===========================
component fabric_switch_c<struct TRAIT =null_trait_sss>{
   import pkg::spi_dataset_r;
   pool[5] spi_dataset_r dataset_p;
   bind dataset_p*;
   rand TRAIT trait_s;

   //Component Instantiated under fabric component
   spi_master_c spi_master[3];
   core_c core[3];
   action root_a{
      activity {
         schedule{
            core[0].write_a with {dataset_p.resource_id=0;
                                  dataset_p.data =4'b1011};
            spi_master[0].read_a with {dataset_p.resource_id=0;
                            dataset_p.data= core[0].write_a.dataset_p.data};
         }
         parallel{
            core[1].write_a with {dataset_p.resource_id=1;};
            spi_master[1].read_a with {dataset_p.resource_id=1;};

        }
    }
}

//===========================
//SPI_MASTER
//===========================
component spi_master_c <struct TRAIT = null_trait_s>
:addr_space_base_c{
   import pkg::*;
   /*Action to perform the write operation
   on the SPI slave where resources are locked */
   action write_a {
      output m_data_buff_b wr_prod_o;
      rand bit [7:0]write_data;
```

- Concurrently, different packets of data are sent to the printers, while the Switch Ports lock resources from a resource pool to gain exclusive access to the bus interface and initiate the data transfer. At the same time, core_c[2] also locks a resource and facilitates the exchange of data flow.

```
          assert(req.randomize() with{
              wr_data = {{write_data}};
              ctrl_reg[7] = {{wr_prod_o.ctrl_reg.spie}};
          });
          finish_item(req);
        end
        ......;
    }

  action write_res_share_a:write_a{
      share core_r core_s;
    }

  action write_res_lock_a:write_a{
      //lock core_r core_l,core_l0,core_l1;
    }
  //Action to perform the read operation on the SPI SLAVE
  action read_a {
      lock core_r core_l;
      share core_r core_s;
      input data_buff_b  rd_cons_i;
    }

}
//==========================
//SPI_SLAVE
//==========================
component spi_slave_c<struct TRAIT = null_trait_s>
:addr_space_base_c {
import pkg::*;

    //Action To receive the data during write opertaion
    action rx_wr_data_a {
        lock core_r core_l;
        share core_r core_s;
        input m_data_buff_b wr_cons_i;
    }
```

```
//Component definitions of address spaces
component addr_space_base_c{}

//component contiguous_addr_space_c
<struct TRAIT = null_trait_s>:addr_space_base_c
component contiguous_addr_space_c:addr_space_base_c{

   //function void add_region(addr_region_s<TRAIT> r);
   function void add_region();

   //functionvoid add_nonallocatable_region(addr_region_s<> r);
   function void add_nonallocatable_region()

   bool byte_addressable = true;
};

//component transparent_addr_space_c
<struct TRAIT = null_trait_s>:contiguous_space_c<TRAIT>
transparent_addr_space_c:contiguous_addr_space_c{

/*It is illegal to pass a non-transparent region to the
add_region() function.*/
}

component pss_top{}

component my_ip{}

extend component pss_top {
   my_ip ip;

   //transparent_addr_space_c<> sys_mem;
   transparent_addr_space_c sys_mem;

   //transparent_addr_space_c<> local_mem;
   transparent_addr_space_c local_mem;
   transparent_addr_region_s r0;
```

Conclusion

- In conclusion, this research paper explains the feasibility of using specification PSS v2.0 for simultaneous upstream/downstream of byte-addressable data to multiple shared/locked devices in multi-CPU SOC environments.
- Additionally, the paper demonstrates that significant speed-ups can be achieved if the implementation is done in a concurrent and/or distributed environment.
- This research presents a promising approach to improving the performance of multi-CPU SOC systems and can be useful for future developments in this field.

THANK YOU

Questions Please ?