Modeling Process Synchronization in Multiprocessor Systems on Chip (MPSoC)

Ines Viskic, Daniel Gajski

Technical Report CECS-08-07
May 10th, 2008

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8919

iviskic@uci.edu , gajski@ics.uci.edu

Abstract

Increasing application complexity and short time to market have led to the wide use of transaction level modeling and platform based design of multiprocessor systems on chip (MPSoCs). However, their models use one-case-fits-all communication scheme or leave the communication and process synchronization to be designed manually. This report provides the designer with a taxonomic set of process synchronization schemes for MPSoCs. All schemes are classified and implemented in our modeling tool. Using the tool, the designer can insert each scheme in the transaction level model (TLM) generated by our tool. After simulation, the designer can compare the executions of different models with the range of performance criteria and decide on the best fit before the system's implementation. This is both faster and more efficient than exploring different synchronization schemes in the implemented system. The goal of this report is enable fast exploration of communication and to guide to the solution best suited for the particular application with regards to the restrictions imposed by the application and the MPSoC platform.

Contents

1. Introduction **5**

2. Related Work **7**

3. Taxonomy of Synchronization in Process Communication **8**

4. Blocking Communication Schemes **10**

4.1. Direct Communication 11
4.1.1. Local Processes Communicating Directly 13
4.1.2. Remote Processes Communicating Directly. 13
4.2. Indirect Communication. 15

5. Process Synchronization in TL Modeling **17**

6. Experiment Setup **21**

6.1. TLM1: CPU + 2 dct modules 21
6.2. TLM2: CPU + 2 imdct modules 22
6.3. TLM3: CPU + 4 HW modules 23
6.4. TLM4: 3 CPU in pipeline 23

7. Results of Experiments **24**

7.1. TLM1 Performance Measurements 25
7.2. TLM2 Performance Measurements 27
7.3. TLM3 Performance Measurements 29
7.4. TLM4 Performance Measurements 31

8. Conclusion **32**

9. Acknowledgements **33**

References **34**

A Appendix **35**

A.1 Source Code **35**
A.1.1 Definition of module classes (output.cpp) **35**
A.1.2 Definition of UBC class (ubc.sc) **43**

List of Figures:

1. An example of a general MPSoC Platform **6**
2. A simple send/recv transaction **8**
3. Taxonomy of process synchronization in MPSoC Platforms **9**
4. Block diagram of processors in direct communication **11**
5. Block diagram of processors in indirect communication **11**
6. Block diagram of process synchronization in direct communication **12**
7. Time diagram of process synchronization in direct communication, when (a) Sender sets F_R and (b) Receiver sets F_S **13**
8. Time diagram of polling based process synchronization: Sender (bus slave) contains the flag; Receiver is the bus master **14**
9. Time diagram of interrupt based process synchronization: Sender (bus master) contains the flag; Receiver is the bus slave **15**
10. Block diagram of process synchronization in indirect communication **16**
11. Time diagram of process synchronization in indirect communication **16**
12. Types of TLM components **17**
13. Internal structure of UBC component **18**
14. A simple TL transaction **19**
15. Implementation of synchronization with 2 flags **20**
16. Time diagram of BUS READ data transfer **21**
17. TLM 1: CPU + 2 dct Modules **21**
18. TLM 2: CPU + 2 imdct Modules **22**
19. TLM 3: CPU + 4 HW Modules **23**
20. TLM 4: 3 CPU Modules in pipeline **24**
21. Performance Analysis of CPU Processes in TLM1 thought TLM4 . . . . **25**
22. Communication delay estimates for CPU process in TLM1 . . . . **26**
23. Communication delay estimates for LFIL and RFIL processes in TLM1 . . . . **26**
24. Distribution of communication tasks in transactions of CPU process in TLM1 . . . **27**
25. Communication delay estimates for CPU process in TLM2 . . . . **28**
26. Communication delay estimates for LPCM and RPCM processes in TLM2 . . . . . . **28**
27. Distribution of communication tasks in transactions of CPU process in TLM2 . . . **28**
28. Communication delay estimates for CPU process in TLM3 . . . . **29**
29. Communication delay estimates for HW processes in TLM3 . . . . **30**
30. Distribution of communication tasks in transactions of CPU process in TLM3 . . . **30**
31. Communication delay estimates for CPU processes in TLM4 . . . . **31**
32. Distribution of communication tasks in transactions of *DecodeFrame*() in TLM4 . **31**
33. Distribution of communication tasks in transactions of *SynthSample*() in TLM4 . . **31**
34. Distribution of communication tasks in transactions of *OutputSample*() in TLM4 . **32**

List of Tables:

1. Performance Estimates of CPU processes in models TLM1 though TLM4 . . . . . . **24**
2. Communication delay estimates with different synchronization schemes for processes in TLM1 . . . . **25**
3. Communication delay estimates with different synchronization schemes for processes in TLM2 . . . . **27**
4. Communication delay estimates with different synchronization schemes for processes in TLM3 . . . . **29**
5. Communication delay estimates with different synchronization schemes for processes in TLM4 . . . . **30**

Process Synchronization Modeling in Multiprocessor Systems on Chip (MPSoC)

Ines Viskic, Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine

1. Introduction:

The constant challenge in system design is to efficiently bring together the strict performance requirements with growing application complexity and short time-to-market projections. Recent advances in system implementation technology have made multiprocessor system on chips (MPSoC) a reality. MPSoC systems are faster than traditional single core SoCs because their multiple computing components execute in parallel. MPSoC components usually include application-specific, heterogeneous processors (CPUs), units of digital signal processing hardware (DSP units), memory units and controllers, high-speed on-chip communication interfaces and sophisticated communication protocols.

However, designing such MPSoC is complex and time consuming. Applying traditional design methodologies was proved to be slow and inefficient for MPSoCs. Modern approaches raise the level of design abstraction to transaction level, hiding pin and cycle accurate implementation details from the user. Transaction level models (TLM) often reuse pre-defined components to form new platform configurations, relieving the user from manually modeling each component anew. Further, TLM simulation provides fast validation and performance evaluation of user's design choices. Due to its efficiency and simplicity of use, TL modeling is emerging as a new standard for design space exploration and early development. However, such approach either uses one-case-fits-all schemes for communication or leaves the communication to be designed manually by the user. Pressed with stringent time-to-market deadlines, the designers usually conform to standard solutions that may be suboptimal for their specific application.

This report provides a complete taxonomy of available process synchronization schemes in MPSoC design. The presented schemes are modeled at the transaction level (TL), which provides for fast and efficient system validation of selected schemes through TLM simulations. The goal of this report is to speed up design process of complex communication schemes for MPSoCs by guiding the designer to select a scheme that is best suited for a particular MPSoC platform.

At the TL, the MPSoC platform is presented as an interconnected set of modules that include processing elements (PEs), memory elements, buses and bridges. The PEs include one or more computing components (processes) that communicate with each other with send, receive routines of universal bus channels (UBCs) and access the memory with UBC’s read and write operations. The bridge elements (transducers) translate messages between communicating processes that support different bus channel protocols.

Figure 1. shows an abstraction of the MPSoC platform. It consists of four PEs (*PE 1* through *PE 4*), two IP units (*IP 1* and *IP 2*) and an accelerator unit *HW 1*. The computing components are connected to each other and a single global shared memory unit (*Memory*) with four buses (*Bus 1* through *Bus 4*) and two bridge units (*Bridge 1* and *Bridge 2).* Components connected to the same bus support its bus channel protocol for communication. For example, *PE 1* and *PE 2* can communicate among themselves and/or access the shared memory using the protocol of *Bus 1*, but need the translating services of bridges to access other component in the system. More specifically, *Bridge 1* translates between three protocols of *Bus 1, Bus 2* and *Bus 3*, while *Bridge 2* translates between protocols of *Bus 2* and *Bus 4*. Therefore, PE 1 must use Bridge 1 unit to communicate to either PE 3 (*Bus1-to-Bus2* translation) or PE 4 (*Bus1-to-Bus3* translation). To access *IP 2* that is connected to *Bus 4*, the same component must utilize both *Bridge 1* and *Bridge 2*.

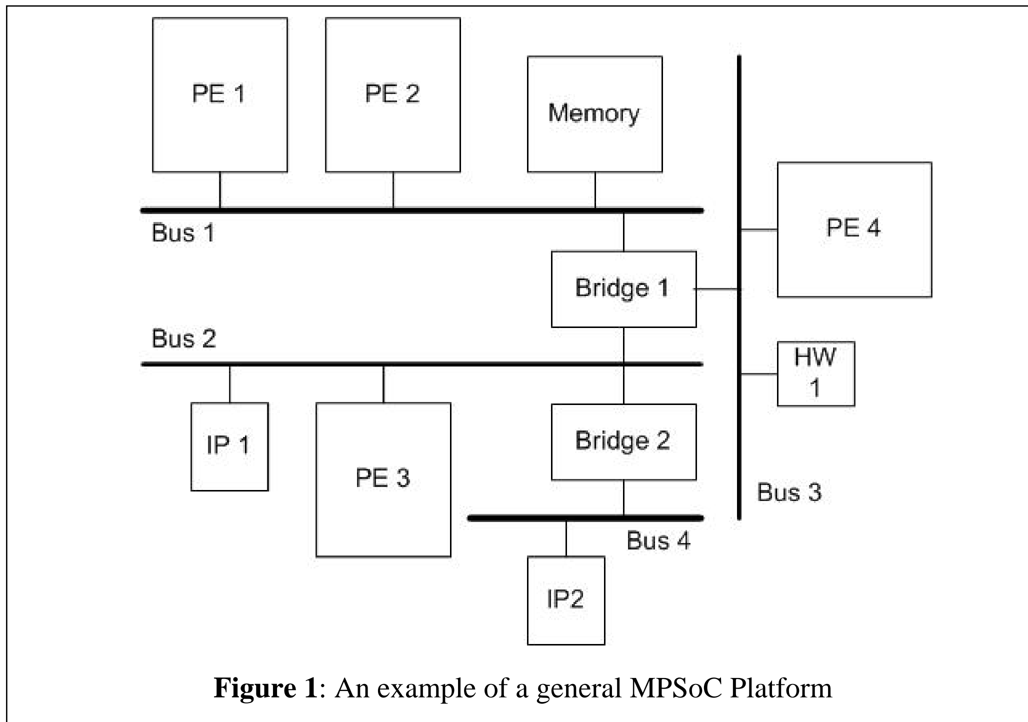

Figure 1: An example of a general MPSoC Platform

In such complex MPSoC system, the communication significantly influences the overall performance of the system. We propose a set of synchronization schemes which can be inserted in the TLM by a simple selection of channel properties. After simulation and performance comparison of each simulated option, the designer can decide on the best scheme based on the particular demands of his platform and application.

The rest of the report is organized in section as follows. Section 2 outlines related work. Section 3 provides a taxonomy tree of available synchronization schemes in TL models (TLM) of MPSoC, while Section 4 and 5 focus on synchronization of processes in direct and indirect communication, respectively. Section 6 presents the models of described schemes in different TLMs simulated in our experiments. Section 7 reports on the results of conducted experiments. We conclude the report with general observations derived from our research in Section 8.

2. Related Work

Communication and synchronization schemes have been a focus of great amount of research. Steinhammer, Obermaisser at al. [1], [2] propose a time-triggered network on chip (NoC) managed by a *Trusted Network Authority* (*TNA*) with the global clock and global communication time-slot schedule table. This approach is a form of non-blocking wait-free synchronization scheme because all communication is determined with global TNA controlled timetable. However, relying on a TNA's global synchronization presents a single point of failure for the whole system. Similarly, non-blocking scheduling is applied in the work of Vincentelli et al [3], who describe the scheduling of task/message activation in the form of ILP (integer linear programming) equations to optimize the latency and meet the proscribed deadlines. The resulting schedule tables are distributed among system components and not confined to a single unit (as is TNA).

Our work offers choices beyond the centralized or statically scheduled schemes that can be difficult to implement in very large MPSoC. Also, most of today's real-time application demand dynamic message scheduling.

The authors in [4] have modeled 5 high-speed busses with different properties and evaluated their influence on the overall system performance. The buses differ on support for pipelining, reduced propagation delay, the type of interconnect (shared bus, bidirectional FIFO, crossbar) and the supported protocol. The focus of this work, however, is on individual analysis and not on classification and comparison.

The work of Huang, Pillai and Shin [5] discusses using dynamic process scheduling to achieve efficient and fast process synchronization. They propose an extension of the EDF (earliest deadline first) to cover systems running in over-load conditions. Their proposed scheduler takes preference to importance (indicates utility) over urgency (as ranked by the deadline) when scheduling the processes during overload. The authors propose an improvement to algorithms for non-blocking inter-process communication (IPC) based on shared variables. The algorithms focus on intra-processor communication where the algorithm and the OS scheduler ensure there are no conflicts among reader and writer processes for the shared variable. Our work covers both intra- and inter-processor communication schemes.

Blocking communication schemes have also driven a lot of research. The Montechiero et al. [7] address the problem of long busy-wait latencies in blocking lock-based communication architectures. The approach is to have a hardware unit to locally manage the polling on shared locations, holds the list of the contending threads, and updates lock/barrier ownership when the contended lock is released. Our taxonomy covers this scheme as a HW implemented process scheduler.

The authors in [8] investigate the lock-based synchronization of local processes in the context of priority-driven preemptive scheduling. They propose protocols that solve the priority inversion as well as prevent deadlocks and chain blockings, raising the schedulability of the system. The work of Gajski, Peir [9] defines essential problems in implementing synchronization in multi-processor systems. Our work is based on those research efforts but is broadened to include today's embedded systems and support the real-time performance demands of today's applications.

3. Taxonomy of Synchronization in Process Communication

Process communication at the transaction level includes defining a route between sending and receiving process, synchronizing them and finally, transferring the data. Figure 2 shows a simple process communication. Two PEs, *PE1* and *PE2*, are connected with a channel *BusChannel* that models a bus. Each PE contains a process: *PE1* has process *Sender*() that sends the data to the receiving process *Receiver()*, residing in *PE2*. The send()/recv() routines contain communication primitives implementing routing (*FindRoute()*), process synchronization (*Synchronize*()) and data transfer (*SendData(), RecvData()*).

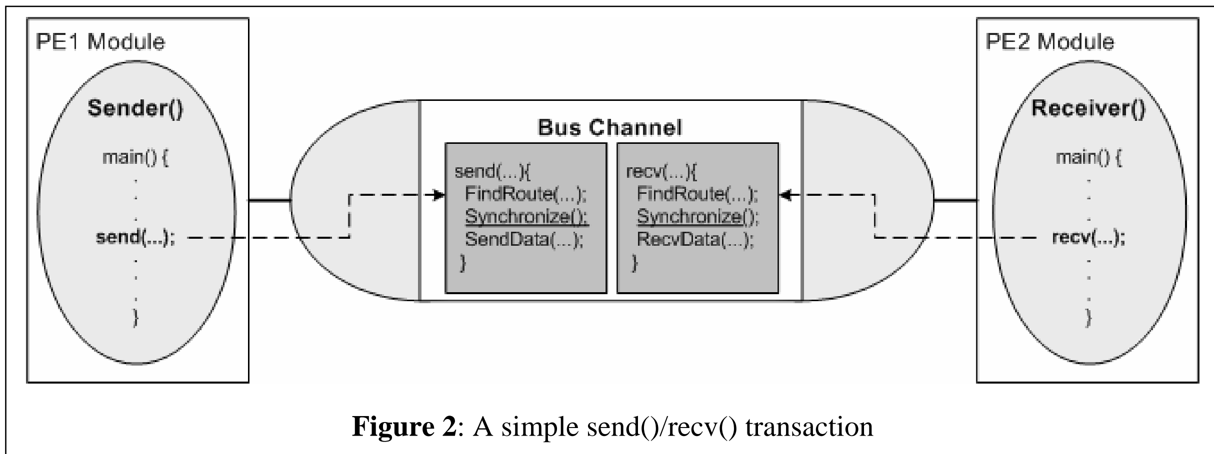

Figure 2: A simple send()/recv() transaction

Routing determines a bus or a set of bus and bridge components that connect PEs containing the *Sender()* and the *Receiver()*. Process synchronization ensures reliable data transfer by making sure the *Sender()* is ready to send and the *Receiver()* is ready to receive the data. Data transfer includes either transferring the entire message between processes or packaging the message into fixed size packets at the sending side and assembling the message from packets at the receiving side.

This report defines and classifies different process synchronization schemes. The schemes are differentiated by:

1. type of communication (i.e. blocking/non-blocking communication),
2. type of communication route (i.e. direct/in-direct communication),
3. number of synchronization flags (1 or 2),
4. position of synchronization flags (in shared memory, or Sender/Receiver PE),
5. type of access to the sync. flag (polling/interrupt by master/slave).

The taxonomy is outlined in Figure 3, with the general case shown at the root of the taxonomy tree, and implementation specific cases represented with gray highlighted squares. The dash lined squares represent cases for which the implementation is not possible. For example, the case of direct blocking communication (left branch in Figure 3) where the Sender is the bus master and contains the flag, the Receiver is the bus slave and therefore does not have the ability to use the bus to access the remote flag via polling (leftmost dashed-lined square).

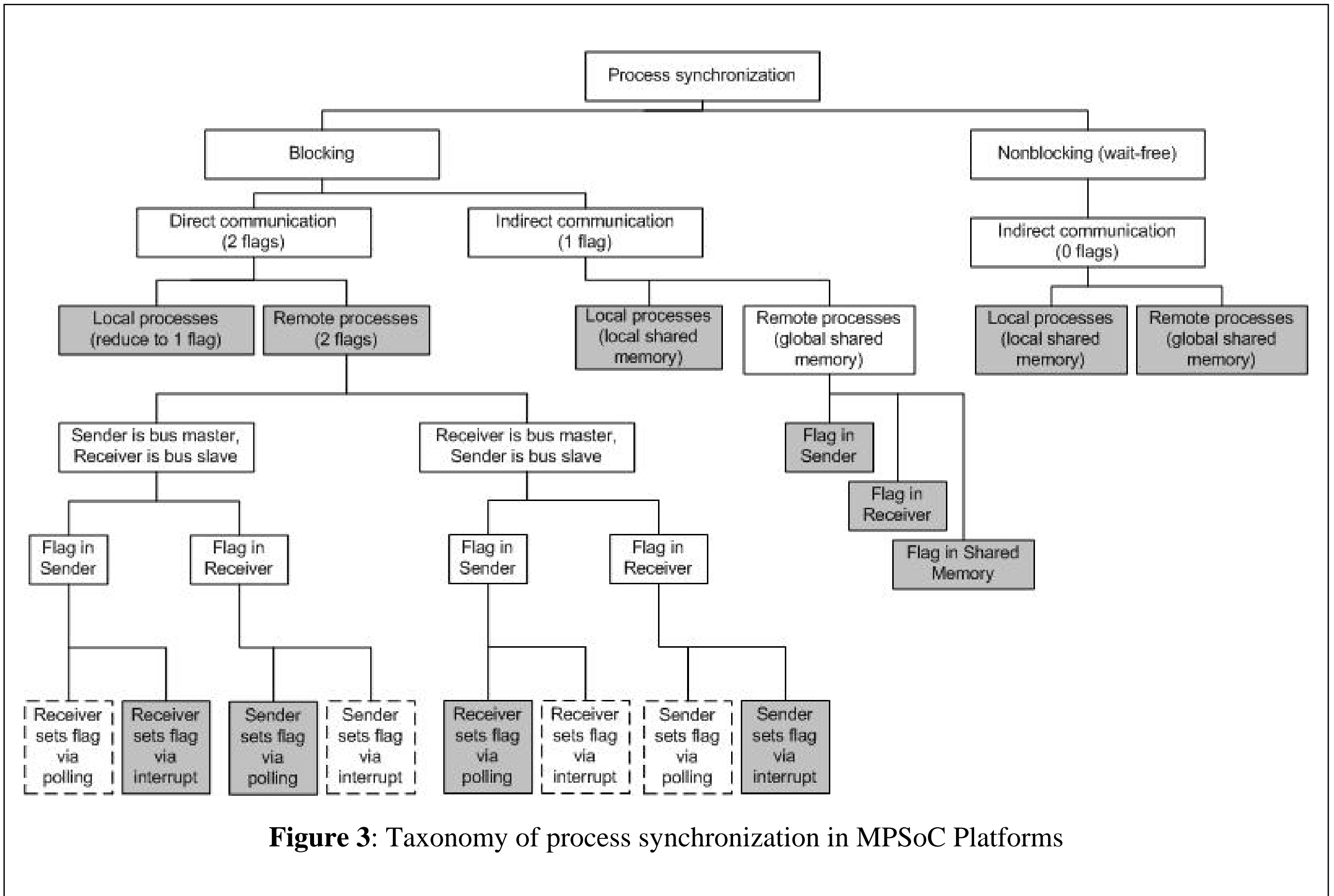

Figure 3: Taxonomy of process synchronization in MPSoC Platforms

The type of communication determines the first branching. If the *Sender()* waits (blocks) until the *Receiver()* finishes receiving the data, the communication is said to be **blocking** (top left branch in Figure 3). On the other hand, if the *Sender()*continues with the computation after sending without knowing whether the Receiver received the data, the communication **is non-blocking** or **wait-free** (top right branch). **Non-blocking schemes** rely on a process scheduler and data buffers to ensure correct communication without explicit synchronization routine. In non-blocking schemes, all invocations of communication routines are statically scheduled with regard to the capacity intermediate buffers, so the sending data can always fit in the corresponding buffer. Therefore, in such schemes there is no need for the *Sender()*to block until the *Receiver()* is ready to accept the buffered data. In other words, the process scheduler ensures that the data will reach the *Receiver()* on time and uncorrupted.

Next distinction is based on the type of route the *Sender()*and *Receiver()* PEs share. If the route between the *Sender()*and the *Receiver()* contains only a single bus, they are said to **communicate directly**. In **indirect communication** the processes of remote PEs communicate by accessing intermediate shared memory and/or bridge units which serve as data storage. The *Sender()* writes to the storage and the receiver reads from it (the *Receiver()* process).

The next level of taxonomy differentiates between communication of processes residing in the same PE (**local processes**) and processes located in different PEs (**remote processes**). In direct communication, local processes are connected with a channel managed by the operating system (i.e. OS pipe), while the PEs of remote processes are connected via shared bus. For indirect communication, local processes use local shared memory, while remote processes use global memory accessible via a shared bus (or two buses, in case of a dual port memory).

The next level limits the **number of flags**: instead of using two separate flags to indicate the readiness of *Sender()* / *Receiver()*, the processes use a single flag to synchronize. A set flag indicates one of the processes is ready, and the reset flag indicates the readiness of both (or none) processes. A single flag can be located in the *Sender()* or *Receiver()*, or the shared memory element (for indirect communication only).

The decision to limit the **access to the address bus** either to the *Sender()* or *Receiver()* makes the next branch in taxonomy. Note that this branching is applicable only in inter-processor communication since intra-processor communication does not access the bus. The communicating processes can either both be able to access the address bus, or only one of them can do so. Denying a process access to the address bus makes the corresponding PE a **bus slave**. The bus slave can only responds to the bus muster requests by reading from the address bus and reading/writing the data bus. The **bus master** is the PE with a process that initiates the data transfer by asserting the address lines of the bus. Indirect communication requires both processors to the *Sender()* and the *Receiver()*can access the address bus to read/write from the common storage unit. In direct communication, the slave PE cannot access any register via bus. It must either have that register locally (flag in slave, polling based synchronization), or it must use a dedicated interrupt line to write into that location (flag in master, interrupt based synchronization).

4. Blocking Communication Schemes

Blocking communication schemes require the processes ready first to block until the data transfer takes place. It can be realized by:

a. The **Sender initiates the writing** of the data in the Receiver's local memory/registers. The data's destination must be accessible to the Sender via bus.
b. The **Receiver initiates the reading** of the data in the Sender's local memory/registers via bus. The data's source must be accessible to the Receiver.
c. Both Sender and Receiver initiate writing and reading of the data (respectively) from the **shared memory**. The memory must be bus accessible to both processes.

Approaches (a) and (b) cover the direct communication schemes, and approach (c) is the indirect communication. Figures 4 and 5 show an abstract view of two PEs configured for direct and indirect communication, respectively.

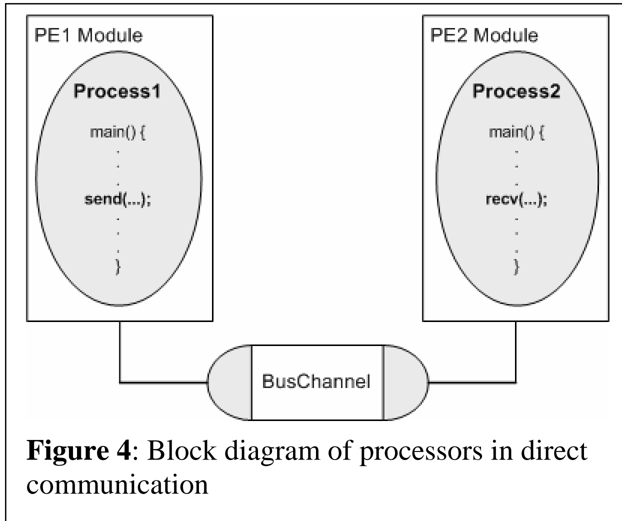

Figure 4: Block diagram of processors in direct communication

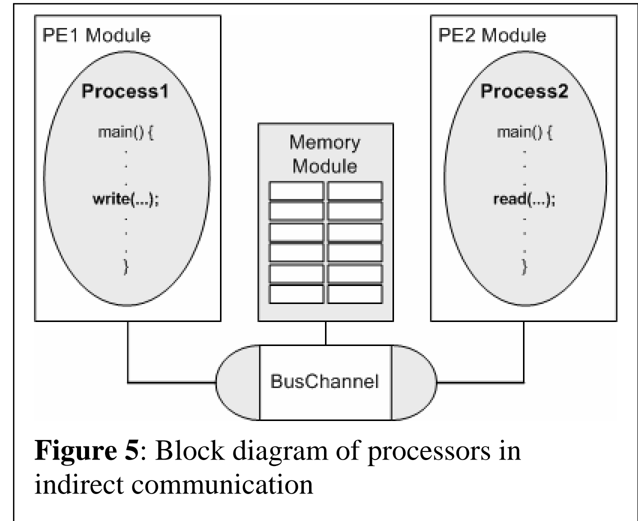

Figure 5: Block diagram of processors in indirect communication

In Figure 4, the two PE modules are connected via a shared bus channel and their processes are exchanging data by reading from and writing to their respective data registers or local memory. The sending process (*Process1*) must load the data register with the data before or during synchronization. After synchronization, the bus write or read operation (depending on whether the transfer is initiated by the *Process1* or *Process2*, respectively) copies the data from one data register to another.

Figure 5, shows two PEs communicating through a shared global *Memory Module*. Both processes *Process1* and *Process2* access the *Memory* with read() and write() routines encapsulated in either a shared bus channel (shown on figure 5), or two separate bus channels. *Process2* must wait for the data to be written in the shared memory by *Process1* before it can read it.

Direct communication implements process synchronization differently than in indirect communication because they require both the Sender and the Receiver to engage in the data transfer at the same time. The following sections provide more details of process synchronization in direct and indirect communication schemes.

4.1. Direct Communication:

In the most general case, process synchronization in direct communication requires two synchronization flags, each indicating the status of one process (0= process not ready, 1= process ready). The flag local to one process signals the readiness of the other process in the communication. Therefore, each process can test the status of the remote process by performing a local read without accessing the bus.

Figure 6. shows the block diagram of synchronization in both Sender and Receiver. A process first checks its local flag (F_L). If the flag is not set, the other process is not yet ready, so the first-ready process sets its remote flag (F_R) and blocks. On the other hand, if the local flag is set, both processes are now ready and process initiates the message transfer.

In order to avoid process deadlock, we must ensure two processes don't set their respective remote flags in two consecutive buss accesses. In order words, each process

must check its local flag (F_L) and set the remote flag (F_R) with the atomic operation. This is achieved by requiring the process to seize the bus before checking its local flag.

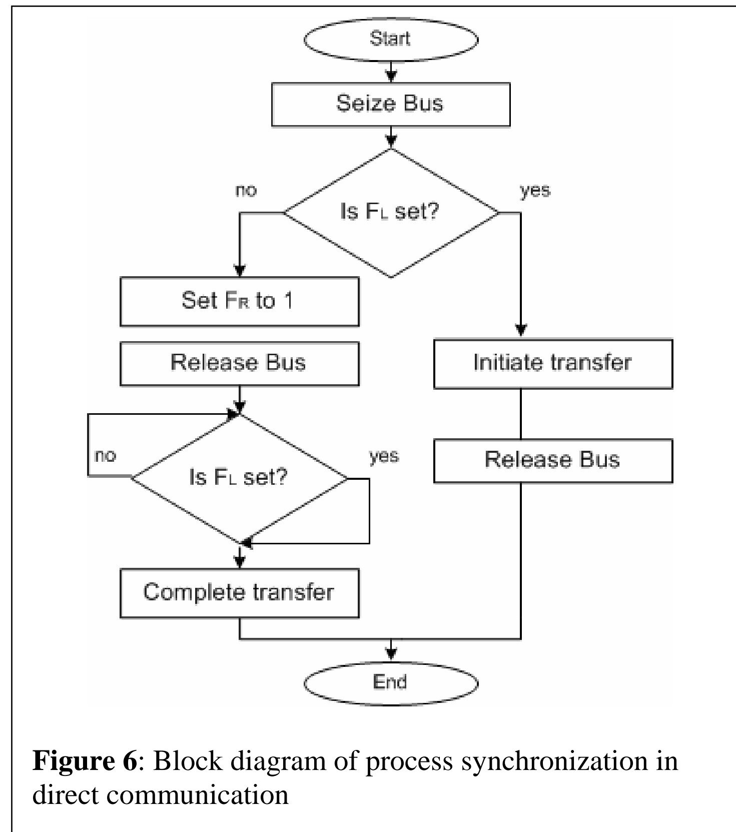

Figure 6: Block diagram of process synchronization in direct communication

Figure 7. shows the described process synchronization in time, as viewed from the shared link between the Sender and the Receiver.

The upper time diagram in Figure 6 shows the case when the **Sender is ready first.** At time t1, the Sender sets the remote flag F_R (the Receiver's local flag) and blocks (time interval t2 to t3). At time t3, the Receiver checks its local flag, resets it and initiates the transfer with **bus read** at time t4. The transfer finishes at time t5.
On the other hand, lower time diagram of the same figure shows the Receiver being ready first and setting its remote flag at time t1. The Sender gets ready at t3, resets the flag and initiates a **bus write** at t4. Similarly, the transfer ends at t5.

This synchronization routine can be implemented using only one status flag residing in either processor, without the loss of generality. The only difference being that the process without the flag must inquire about the status of the other processor via bus. To ensure correct execution, the access to the flag must be sequential (atomic check and set/reset operations). This means that local read/write to the flag is disabled when the flag is being accessed via bus.

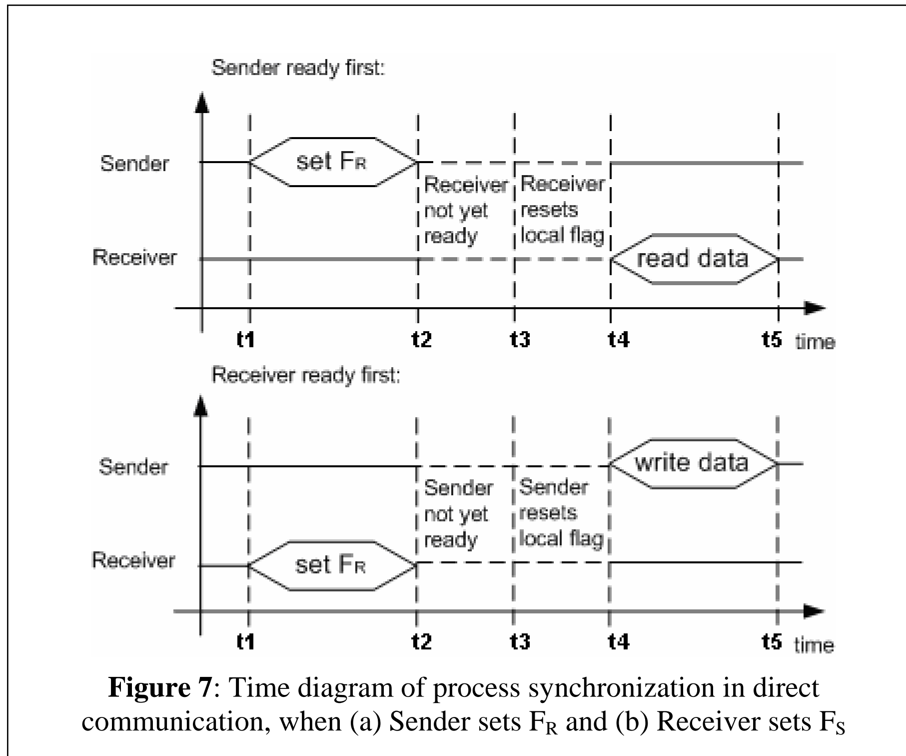

Figure 7: Time diagram of process synchronization in direct communication, when (a) Sender sets F_R and (b) Receiver sets F_S

4.1.1. Local Processes Communicating Directly

If the two processes involved in direct communication reside in the same processor, the **communication is local** to that processor. Since the processor cannot run more than one process at a time, the local communication between the Sender and Receiver needs to be scheduled by the operating system. Therefore, the operating system will control the processes setting and resetting the flag, manage the flag status and block/unblock the processes depending on the flag values.

4.1.2. Remote Processes Communicating Directly

In **remote communication** the processes synchronize with bus accesses. The previously described synchronization scheme assumes that both Sender and Receiver PEs have the same access rights to the shared bus. However, traditional bus-centric architectures divide system components into bus masters and bus slaves, allowing only the masters to initiate data transfer. Therefore, after flag reset if master PE is the Sender, it will initiate the **WRITE DATA operation,** or **READ operation** if the master is the Receiver. This scheme is modeled with limiting the access to the address bus to one PE. (see Figure 3. for synchronization taxonomy).

Polling based scheme:

Figure 8. shows two timing diagrams case where the bus slave is sending and bus master is receiving the data.

Upper diagram (a) represents a case when the slave reaches the synchronization point first and sets its local flag sometime before time t1. At t1, the master tests-and-resets the flag with a single polling operation and initiates the transfer.

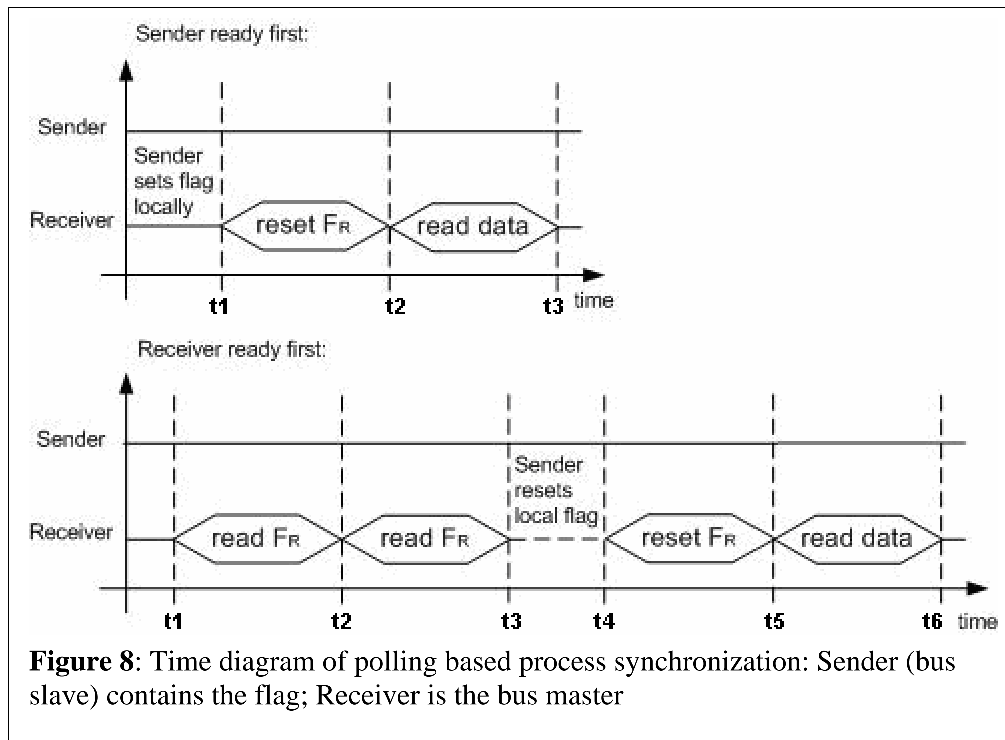

Figure 8: Time diagram of polling based process synchronization: Sender (bus slave) contains the flag; Receiver is the bus master

Lower diagram (b) represents the case when the master is ready first and has to **repeatedly poll** the remote flag (times t1 and t2) until it is set. This consumes both bus cycles (since the bus is utilized for every flag access) and computing cycles of the master (since it could be executing another, independent process). At time t3, the slave becomes ready and sets the flag. On the first consecutive poll at t4, the master will reset the flag and at time t5 start the transfer.

Interrupt based scheme:

If the flag is in the master PE, the slave PE cannot access the flag via bus, but does so asserting a dedicated (interrupt) line with the **interrupt** signal.

The interrupt scheme is shown on Figure 9 with two timing diagrams, the upper showing the Sender/Master reaching the synchronization point first, and lower when the Receiver/Slave is ready first. In both diagrams show the slave interrupts the master (whose interrupt handler sets the flag) at time t1. The master will initiate the transfer after it resets the flag locally: for upper diagram that is at time t3, and for lower at time t4.

The interrupt based scheme is very efficient in terms of time needed to synchronize, simply because the slave does not need to arbiter and utilize the bus to access the flag. However, each slave needs to have its own dedicated interrupt line for this scheme, which makes it expensive and impractical to implement in large systems.

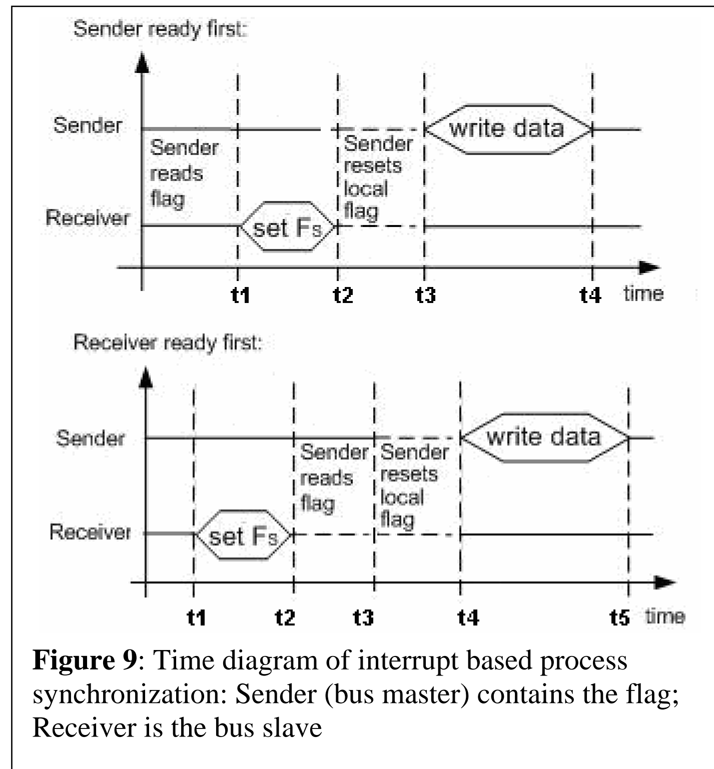

Figure 9: Time diagram of interrupt based process synchronization: Sender (bus master) contains the flag; Receiver is the bus slave

4.2. Indirect Communication:

In indirect communication, a shared memory disassociates the Sender from the Receiver. It is no longer important that the processes are ready at the same time, but rather that the data is ready to be written (or read) when the Sender (or Receiver) becomes ready. Therefore, the scheme requires a single synchronization flag (F_D) indicating the status of the data (0 = ready to be written, 1 = ready to be read). Further, since the write always precedes the read, the Sender will always set flag, and the Receiver will always reset it back to 0.

As in direct communication, we make a distinction between communication in **local** and **remote processes**. In local indirect communication, process synchronization is managed **by the operating system** with mutex locks as synchronization flags. The OS will **grant** access to the Receiver only if the Sender has already written the data and set the synchronization flag. Otherwise, it will **block** the Receiver and schedule the Sender process to run. The Receiver will get unblocked after the flag is set. Similarly, the Sender will get blocked on the attempt to write the data when the synchronization flag is not reset (i.e. the Receiver has not yet read the previous data).

In the **remote communication**, the synchronization flag can be located at either the Sender or Receiver processor, or in global shared memory, with the Sender always setting the flag and the Receiver always resetting it. The block diagram shown in Figure 10 outlines this approach. The left diagram shows the functionality of the Sender (Writer), who checks the flag (F_D), writes the data if the flag is clear and then sets the F_D. If the flag is not yet cleared, the bus is released without writing. The diagram on the right

shows the Receiver (Reader) in the indirect communication. It reads the data from the shared memory if the flag is set and resets it after the read. If the flag is not yet set, the Receiver releases the bus without reading the memory.

The link providing access to the shared memory (Bus) can either be shared between the processes or separate (in case of a 2 port memory element)

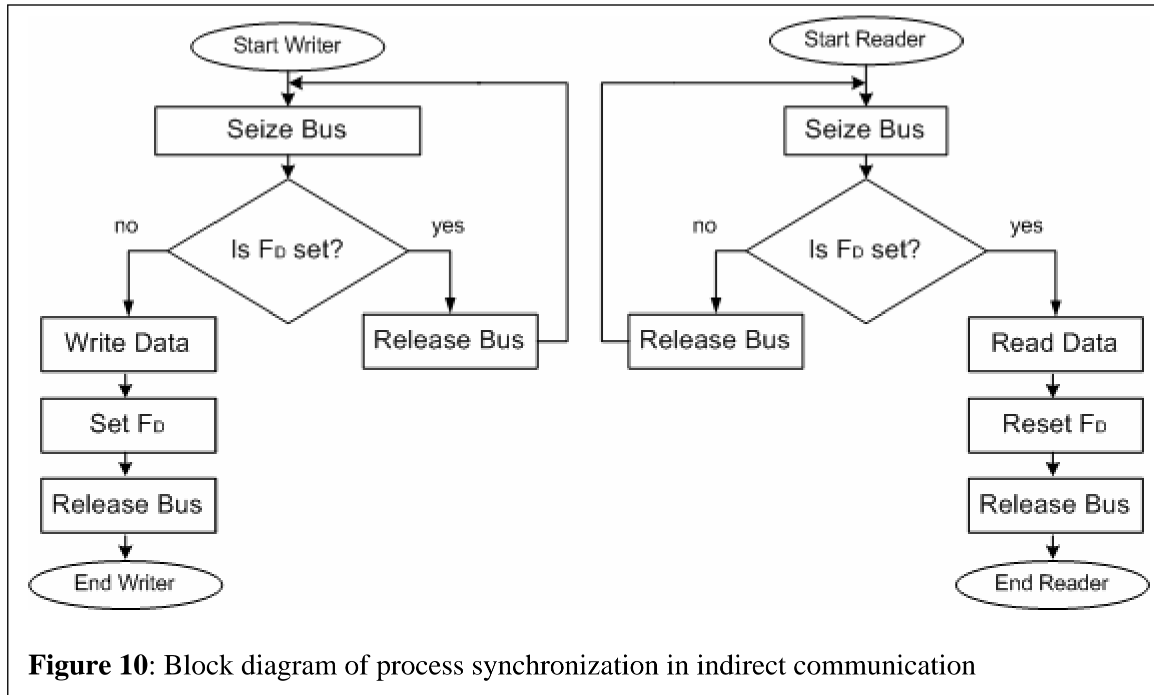

Figure 10: Block diagram of process synchronization in indirect communication

Figure 11. shows the timing diagram of the described scheme. At time t1, the Sender gets ready and acquires the bus to read the flag. If the flag is 0, the Sender writes the data into the appropriate address (time t2) and sets the flag (at t3). When the Receiver gets ready (t5), it reads the flag in the memory and, if the flag is set, it reads the data from the appropriate address. After the read, the Receiver resets the data at time t7.

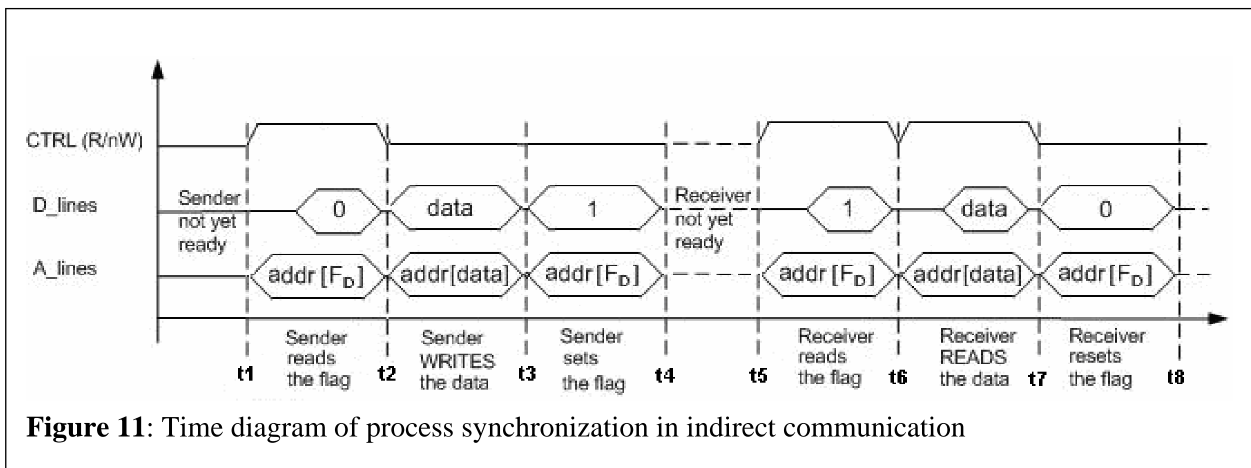

Figure 11: Time diagram of process synchronization in indirect communication

Note that setting/resetting operations can be implemented as **polling routines** (read and write operations that use the bus) or as **interrupt events** (both Sender and Receiver notify the other process that the data is ready to be read/written by asserting their respective interrupt lines).

5. Process Synchronization in TL Modeling

At the transaction level, the MPSoC components are implemented as SystemC classes and are strictly separated into **computing, communicating** and **storage** elements.

Computing elements are modeled as modules containing concurrently executing processes. Processes perform computation over local variables and communicate with other processes only by explicitly invoking communication routines (S*end/Recv/Read/Write*). **Storage modules** contain a set of global variables modeling memory locations and a process modeling the memory controller. The system's **communication** is encapsulated within the universal bus channel (UBC) that models a system bus. It provides basic communication services of routing, synchronization, arbitration and data transfer. The Bridge is another communication component, whose process accepts a message, temporarily stores it and forwards it through its output port.

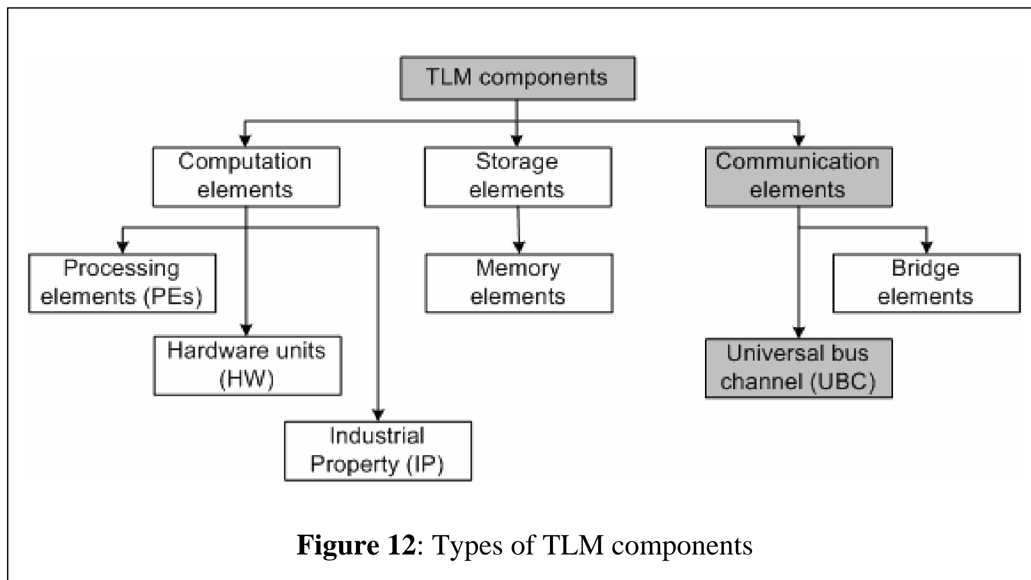

Figure 12: Types of TLM components

Figure 12. outlines the components of TL modeling. Computing elements include PEs, HW units and IPs (shown on Figure 12, on the left), while Memory units are used as storage components (middle of the Figure 12). Finally, communication components, shown on the right, are UBC and Bridge elements. The path from general TLM component down to UBC class is highlighted gray.

All components of the TL modeling except the UBC are implemented as SystemC modules and belong to class *sc_module*. UBC, on the other hand, belongs to a SystemC class *sc_channel.* Therefore, UBC does not contain any active processes, but in turn provides a public communication interface. Figure 13. shows the structure of the UBC component. It contains three classes of communication routines:

1. Routines for synchronized communication*: Send()* and *Recv()*.
2. Routines for unsynchronized communication: *Read()* and *Write()*.
3. Routines for memory control: *MemoryAccess for reading/writing shared memory locations*

Routines for synchronized communication contain primitives for route determination (*FindRoute()*), process synchronization (*Synchronize_[schemeID]()*) and data transfer (*SendData(), RecvData()*). If the process accesses a remote location, these functions will first acquire the bus using the arbitration routine *BusRequest()*, and after the set/reset releases the bus with a call to *BusRelease()*. *Read()* and *Write()* routines do not include process synchronization primitives because the target memory unit is presumed always to be ready for read or write operations.

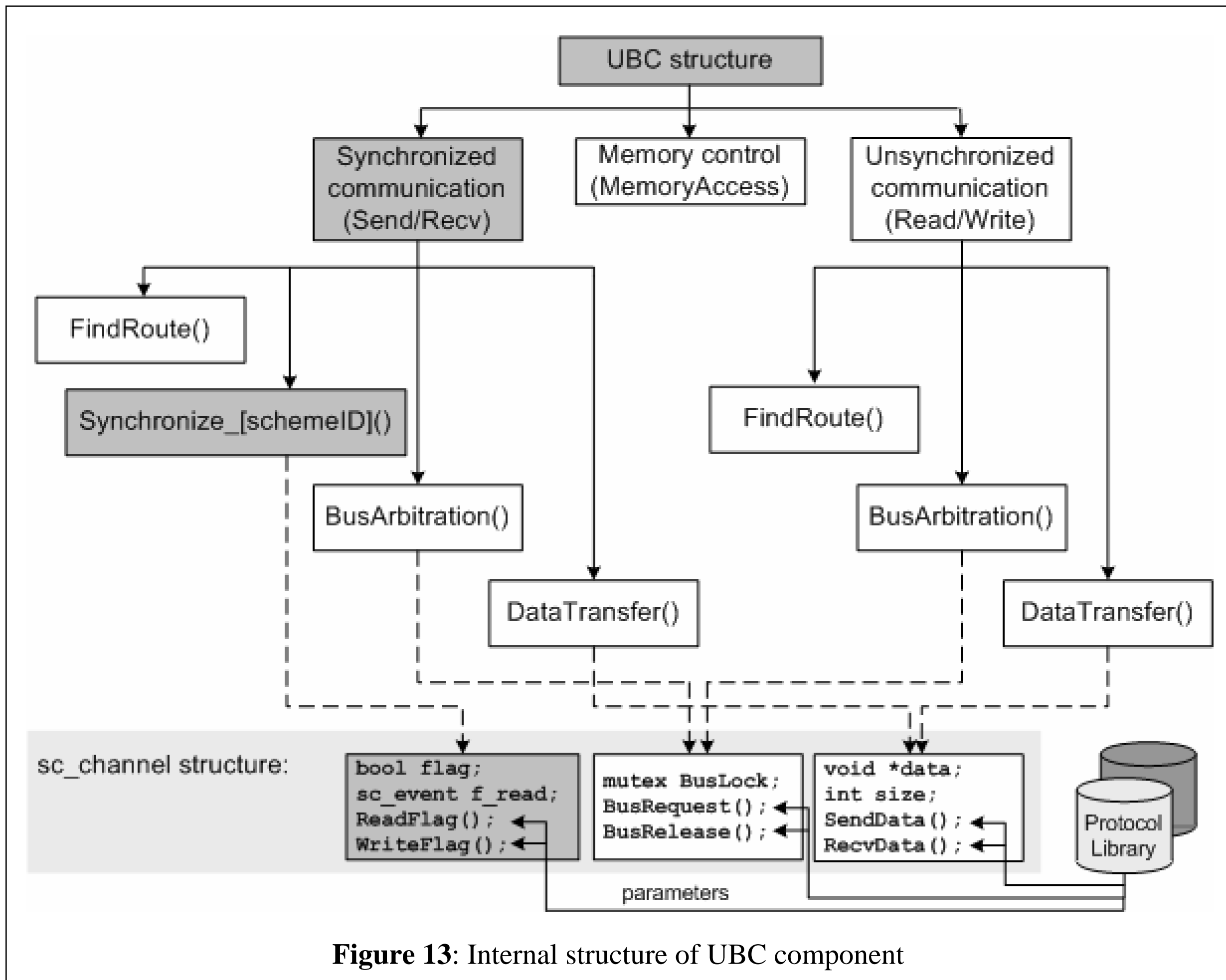

Figure 13: Internal structure of UBC component

Further, bus arbitration primitives (*BusRequest()* and *BusRelease()*) use a single mutex variable *BusLock* for ensuring sequential access to the bus. Process synchronization primitives contain one or two synchronization flags (*bool flag*), depending on the selected synchronization scheme, for each pair of communicating processes. Each flag is tightly coupled with an event (*sc_event f_read*) that notifies the corresponding processes that the flag has been accessed. Data transfer primitives use a size variable and a data pointer to implement the copying contents of one memory

location to the next. The number of clock cycles each primitive takes to execute is determined with the parameters of the bus channel imported from the *Protocol Library* stored in the tool's data base.

Figure 14. outlines a simple TLM transaction, with two remote processes *P1* and *P2*, each belonging to its module (*Module1* and *Module2*, respectively), transferring data via a UBC *Send/Recv* functions. Each function will then call on UBC's internal primitives (*FindRoute(), Synchronize_[schemeID](), Send(), Recv()*) to execute the transfer.

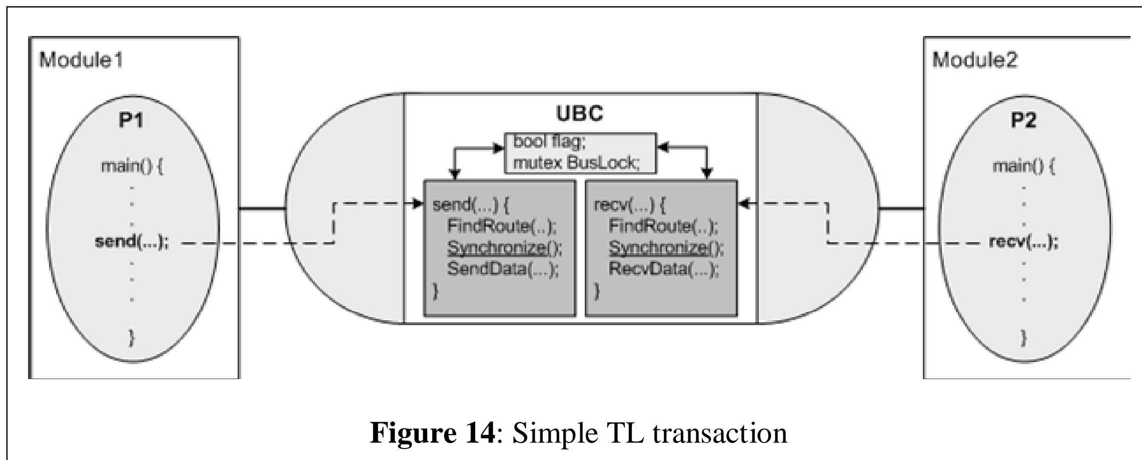

Figure 14: Simple TL transaction

The *Synchronize_[schemeID]* function implements process synchronization, where *[schemeID]* uniquely identifies one of the following synchronization schemes:

1. Synch with 2 flags, both processes can assert address bus (no master/slave PEs)
2. Synch with 1 flag, both processes can assert address bus (no master/slave PEs)
3. Synch with 1 flag, polling based scheme (flag is in slave PE)
4. Synch with 1 flag, interrupt based scheme (flag is in master PE)
5. Synch with 1 flag, shared memory scheme (flag is in shared memory, no master/slave PEs)

Figure 15 contains an excerpt of the implementation of the 2 flag synch. scheme Lines 2-4 assign a flag (*flag1* and *flag2*) and an event (*ev_f1_read* and *ev_f2_read*) to each of the processes (*MyID, PartnerID*) in communication. Lines 7-19 and 20-32 model the behaviors of these processes, respectively. Each process first arbiters the bus and reads its local flag (lines 8-9 and 21-22). If the flag is not set they set their remote flag via the acquired bus and wait (lines 10-15 and 23-28). After synchronization, they enter data transfer routine with a tag *UBC_INITIATOR*. On the other hand, if their local flag is set, they reset it and return to start the data transfer with the tag *UBC_RESETTER* (lines 15-18 and 28-32).

This and other process synchronization schemes are implemented in SystemC and included in the UBC implementation. The example of TLM model with all process synchronization schemes is attached in the Appendix of this report. The TLM models communication of one CPU module with four HW modules (Platform configuration TLM3 as described in Section 6.3)

```
 1  unsigned int Synchronize_2flags(int MyID, int PartnerID) {
 2    bool *flag1 = flag1_of(MyID, PartnerID);
 3    bool *flag2 = flag2_of(MyID, PartnerID);
 4    sc_event *ev_flag1_read = event_of(*flag1);
 5    sc_event *ev_flag2_read = event_of(*flag2);
 6
 7  if (has_flag1(MyID)) {
 8    ArbiterRequest(MyID);
 9    LocalRead();
10    if (not_set(*flag1)){
11          BusWrite(set_flag2);
12          ArbiterRelease(MyID);
13          wait(*ev_flag2_read);
14          return UBC_INITIATOR;   // go to transfer
15    }else{
16          LocalWrite(reset_flag1);
17          ev_flag1_read->notify();
18          return UBC_RESETTER;    // go to transfer
19    }
20  } else { // has flag2
21    ArbiterRequest(MyID);
22    LocalRead();  // f2, wait(1,SC_NS);
23    if (not_set(flag2)){
24          BusWrite(set_flag1);
25          ArbiterRelease(MyID);
26          wait(*ev_flag1_read);
27          return UBC_INITIATOR;   // go to transfer
28    }else{
29          LocalWrite(reset_flag2);
30          ev_flag2_read->notify();
31          return UBC_RESETTER;   // go to transfer
32          }
33  }
37 }
```

Figure 15: Implementation of synchronization with 2 flags

After synchronization:

The process that has reached its synchronization point first is named the initiator of the transaction and is attributed with the tag *UBC_INITIATOR*. The process that comes in second is attributed with *UBC_RESETTER*, since it resets the synchronization flag. After synchronization, the process *UBC_RESETTER* starts the data transfer by acquiring the bus (call to *BusRequest()*) and asserting the address/control lines of the bus. *UBC_INITIATOR* will respond by either asserting the data lines (in case of bus read operation) or by storing the data from the data lines (bus write operation). The transfer is complete when *UBC_RESETTER* releases the bus with *BusRelease()* function call.

Note that in master/slave architectures the roles of *UBC_INITIATOR* and *UBC_RESETTER* are predetermined: the slave is always *UBC_INITIATOR*, since it is always idle until the master is ready, and the master is *UBC_RESETTER*, since it controls the immediately following data transfer.

Figure 16. shows a diagram of bus read operation, when process *Sender* acts as a *UBC_INITIATOR* and process *Receiver* is the *UBC_RESETTER*. At time t1, Sender will

set the flag, Receiver will read/reset it at time t2 and start the BUS READ operation (times t3 to t4). The transaction ends at time t4, with Receiver releases the bus.

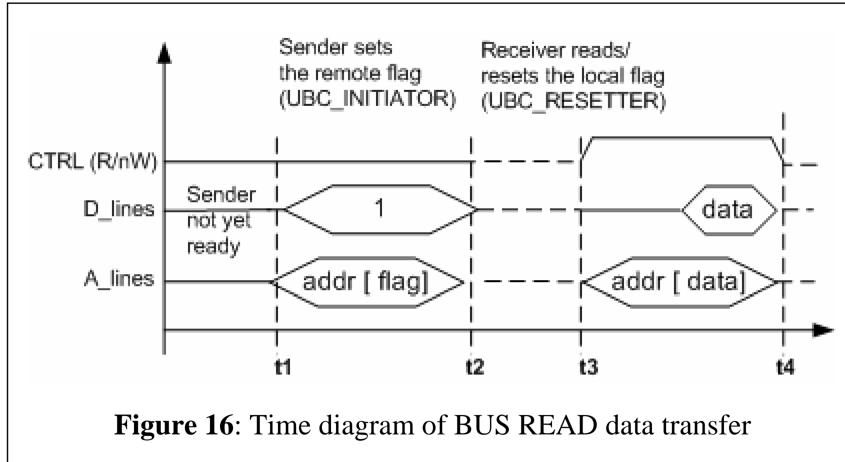

Figure 16: Time diagram of BUS READ data transfer

6. Experiment setup

We have modeled five different process synchronization schemes in four the TLMs of an industrial strength application: the MP3 decoding algorithm for decompression of a MP3 input stream organized in frames that outputs audio (PCM) data. All TLM models were implemented in SystemC V 2.0, compiled with the gcc and executed on a server machine with dual Opteron 246 CPU, 2GHz processing speed and 2 GB of memory

6.1. TLM1: CPU + 2 dct modules

The outline of the MPSoC platform is shown on Figure 17 The main part of the algorithm's code is mapped to the process of a CPU Module called *MP3_main()*.

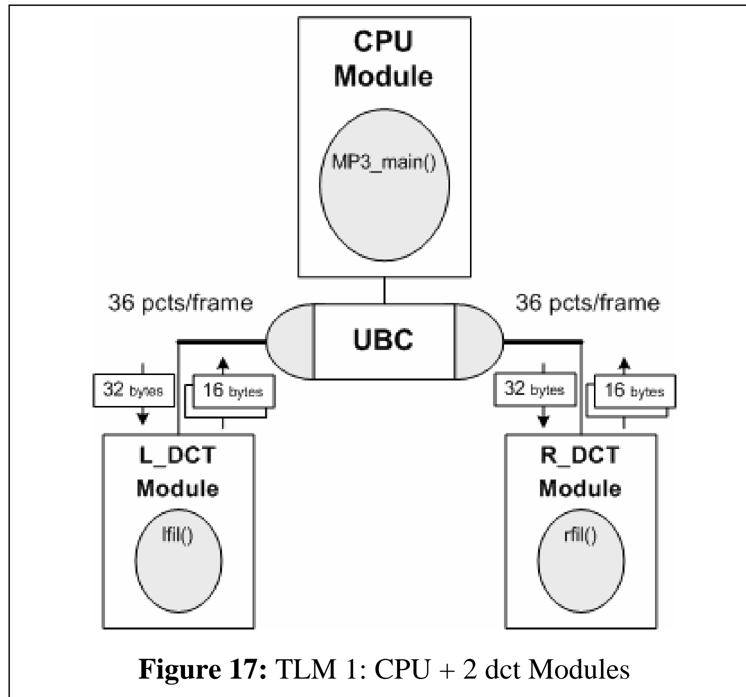

Figure 17: TLM 1: CPU + 2 dct Modules

The *MP3_main()* process handles reading the input from the main memory, creating the MP3 frames from input data, decoding the frames, synthesizing the PCM samples from the MP3 frames and outputting them to the speaker application. The other two modules, L_DCT and R_DCT Modules, are mapped with functions performing polyphase filtering (DCT). Since the DCT filtering takes place within the PCM sample synthesis, the *MP3_main()* process sends out parts of the frames to the left and right DCT components. The data exchanged between the SW and DCTs is transferred in 72 packets per frame, ranging in size from 32 bytes (input sent to HW) to twice as many (144) packets per frame of 16 bytes as output received from HW filters.

After receiving the output, the process completes the synthesis and outputs the PCM samples to the speaker. The alternative validation is a cross-check comparison of the synthesized samples to the reference PCM samples in the golden file.

6.2. TLM2: CPU + 2 imdct modules

This configuration represents the alternative mapping to the one previously described. Similarly to TLM1, this model maps most of the application to the CPU Module. Also, as before, *MP3_main()* process creates the MP3 frames from the input file, decoding the frames and synthesizing the PCM samples. However, unlike the previous model, the PCM synthesis is done entirely by the MP3_main process in the CPU Module. Instead, L_IMDCT and R_IMDCT Modules are mapped with the IMDCT sampling functions (lpcm() and rpcm()), for left and right stereo sound frames.

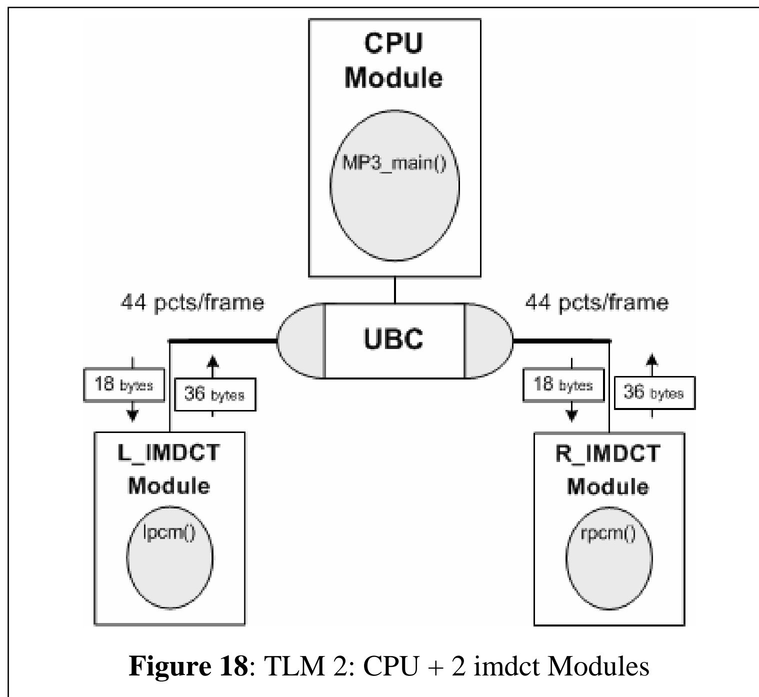

Figure 18: TLM 2: CPU + 2 imdct Modules

IMDCT sampling takes place within the decoding of the frames, so the MP3_main process sends out part of the frames to the left and right IMDCT components during the decoding of each frame and waits for their outputs before continuing decoding algorithm. The data exchanged between the CPU and IMDCT units is transferred packets

ranging in size from 18 bytes (input to IMDCT Modules) to 36 bytes (output from IMDCT Modules). The communication includes 88 transfers with 18 byte packets and 88 transfers with 36 byte packets per MP3 frame. The outline of the described TLM 2 is shown on Figure 18.

6.3. TLM3: CPU + 4 HW modules

This platform contains both the IMDCT and DCT components of the previous models. The process MP3_main in CPU Module will perform MP3 frame decoding and PCM sample synthesis with all four modules acting as HW accelerators for both polyphase filtering (L_DCT, R_DCT Modules) and sample filtering (L_IMDCT, R_IMDCT Modules). The platform is shown on Figure 19.

As in previous models, the data transfer includes transferring 88 packets per frame for the MP3 frame decoding phase (88 for IMDCT input and 88 for output) and 72/144 packets per frame for DCT synthesis (72 input to DCTs, 144 as output)

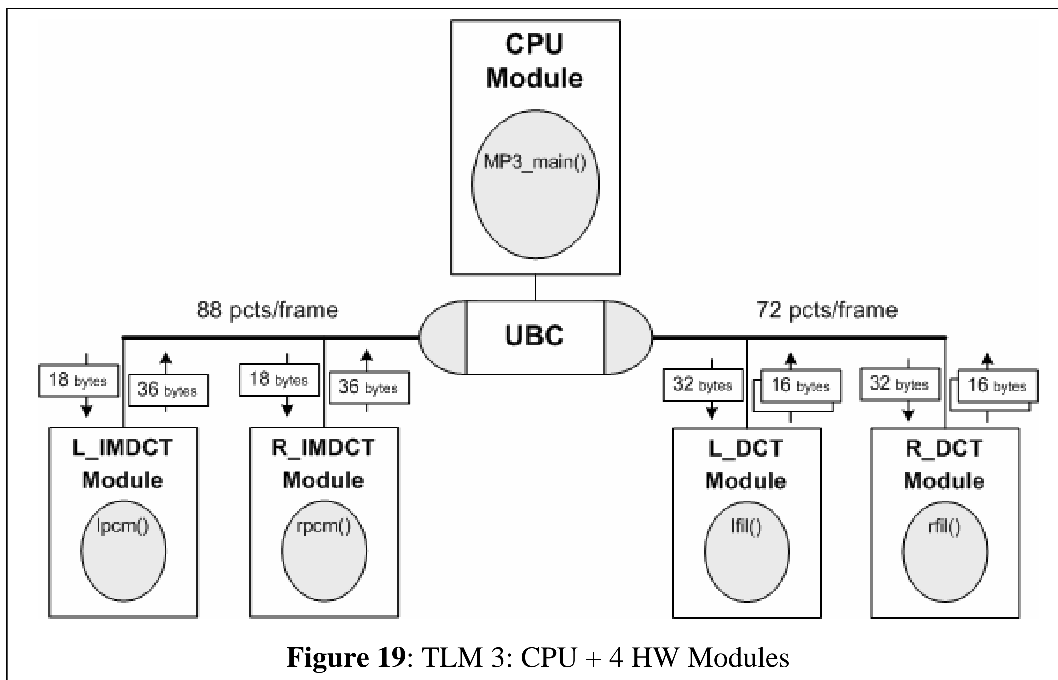

Figure 19: TLM 3: CPU + 4 HW Modules

6.4. TLM4: 3 CPU Modules in pipeline

This TLM represents the pipelined architecture that is often in use for media streaming systems. Here the architecture directly reflects the properties of the application: each component is mapped with the corresponding phase in the application algorithm. In MP3 decoder, we can identify 3 distinct phases of data processing:

(b) decoding phase, in which the MP3 frames are decoded
(c) synthesis phase, where the PCM samples are created from the decoded frames
(d) output phase, where the PCM samples are outputted to the speaker application

The pipelined model is shown in figure20. The process of CPU1 Module (*DecodeFrame*()) reads the input and decodes the MP3 frames before passing them on to the process of CPU2 Module. *SynthSample*() accepts the decoded frames, synthesizes the corresponding PCM samples and sends them to the final process (*OutputSample*()) which outputs them to the speaker application. The transaction between pipeline the first and second pipeline stage transfers the entire MP3 frames, so each packet has 3470 bytes. The other transaction (second to third stage) transfers the entire PCM sample (2307 bytes).

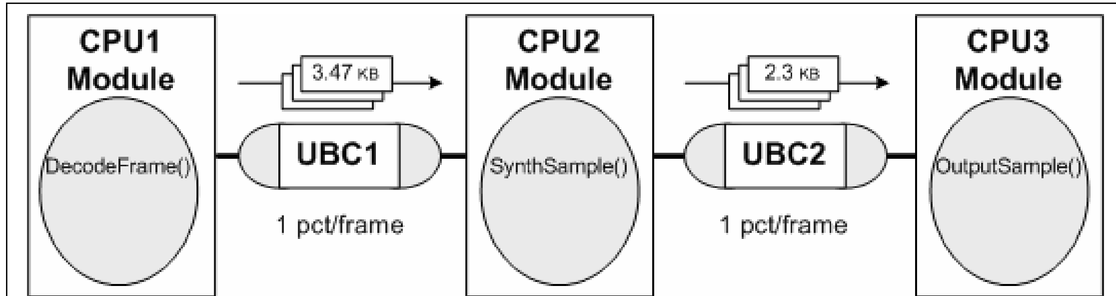

Figure 20: TLM 4: 3 CPU Modules in a pipeline

7. Results of Experiments

Table 1 compares the estimated computation, communication delays and the total execution times (in millions of clock cycles) of the described TLMs. The first column is a reference TLM containing only a single module that executes the entire MP3 Decoder in 1889.16 million clock cycles. The next three columns show the timing results (in millions of clock cycles) for *MP3_main* processes in platform configurations TLM1, TLM2 and TLM3, respectively (all described in previous sections). The final three columns represent timings of three pipelined components in TLM4: processes *DecodeFrame*, *SynthSample* and *OutputSample*.

	CPU	CPU+2dct	CPU+2imdct	CPU+4HW	3CPUs in pipeline TLM4		
	Single process	MP3_main in TLM1	MP3_main in TLM2	MP3_main in TLM3	Decode Frame	Synth Sample	Output Sample
Computation Estimate	1889.16	1765.03	1556.25	1432.12	1359.35	856.18	103.18
Communication Estimate	0	7.36	41.75	55.81	26.44	51.44	936.00
Total Execution Estimate	1889.16	1781.03	1598.00	1487.93	1378.13	1378.13	1378.13

Table 1: Performance Estimates of CPU processes in models TLM1 though TLM4

However, all configurations are computation, not communication intensive: the communication in CPU processes takes between 0.5% and 4% of total simulation time. The exception is the communication of *OutputSample*() of TLM4, but there the process spends most of its execution time <u>waiting</u> for the input data (i.e. the reset of the

synchronization flag). We assume that the choice of synchronization scheme would more significantly influence the performance of communication intensive systems.

Figure 21. shows the same results with a graph. The initial TLM with no communication executes the slowest (1889 million cycles), while the TLM that exploits the parallelism of the application the most (*pipelined TLM*) executes the fastest (1378 million cycles).

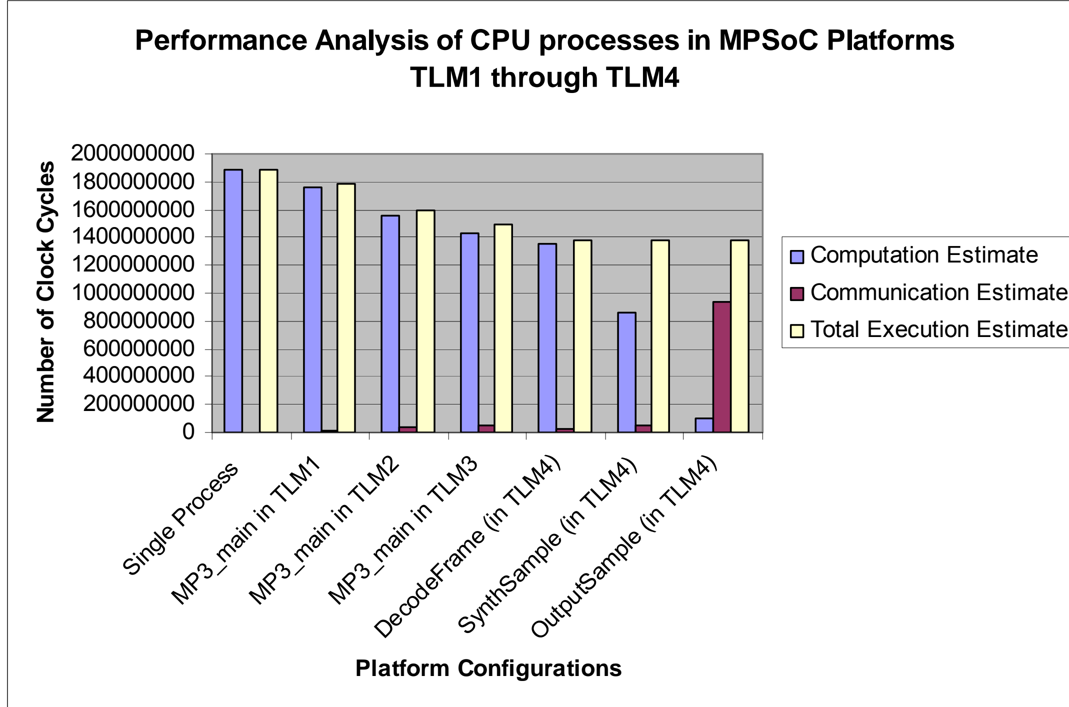

Figure 21. Performance Analysis of CPU Processes in TLM1 thought TLM4

7.1. TLM1 Performance Measurements

Table 2 compares the estimated communication delays simulated with TLM1 platform configuration. The communication of CPU process MP3_main takes less than 0.5% of the total execution of the TLM. The fastest communication is achieved with *2-flag synchronization scheme* (1st row in Table 2). Processes in filter modules spend most of the simulation (99%) waiting idly for resetting of the synchronization flag(s).

Synch. Scheme	CPU Module: MP3_main()	L_DCT Module: lfil()	R_DCT Module: rfil()
2 flags	7.36	1772.20	1772.20
1 flag in filters	7.80	1772.64	1772.64
1 flag in mp3_main	7.54	1772.38	1772.38
1 flag, polling	7.96	1772.80	1772.80
1 flag, interrupt	8.13	1772.97	1772.97
1 flag, shared memory	9.88	1774.72	1774.72
Total Execution of TLM1:	**1781.03**		

Table 2: Communication delay estimates with different synchronization schemes for processes in TLM1

Graphs on Figures 22 and 23 show timings of those synchronization schemes in CPU Module process and filter module processes, respectively. In both, the fastest synchronization is achieved with 2 flag scheme. The difference between the fastest (*2 flag synchronization*) and the slowest (*1 flag, shared memory*) synchronization scheme in CPU process is 25% (in Figure 22), and the difference between the same two schemes in filters is 0.15% (Figure 23).

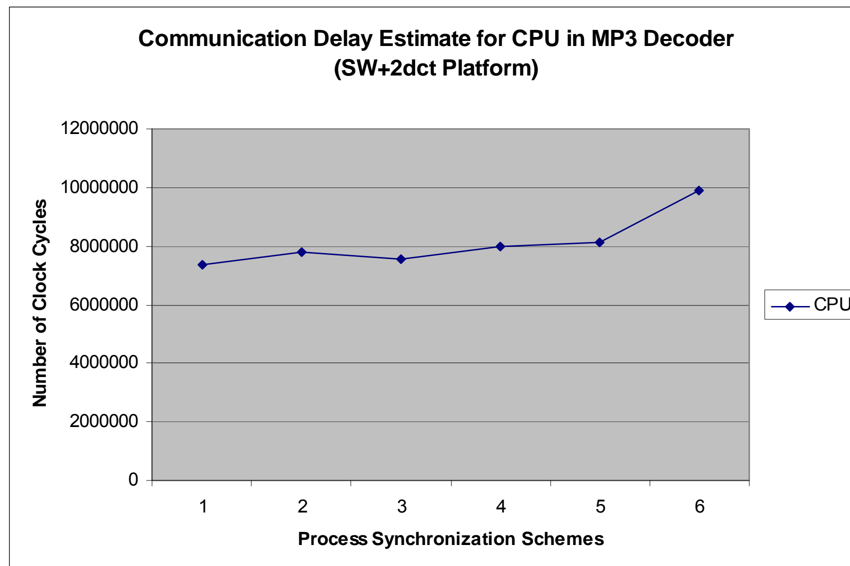

Figure 22: Communication delay estimates for CPU process in TLM1

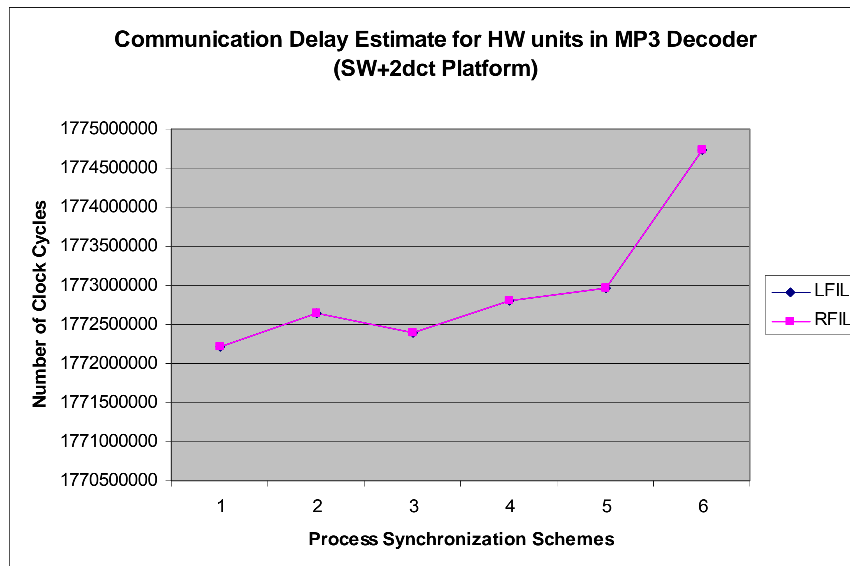

Figure 23: Communication delay estimates for LFIL and RFIL process in TLM1

Finally, Figure 24. presents the distribution of communication tasks in transactions of CPU process in TLM1. Data Transfer takes 95% of the total comm. time (69.8 out of 7.36 mil of clock cycles), and synchronization 3% (7.05 out of 7.36 mill of

cycles). Arbitration delay contributes to the total communication delay with 2%. This distribution indicates that the process is very rarely idle (waiting for the DCT filters).

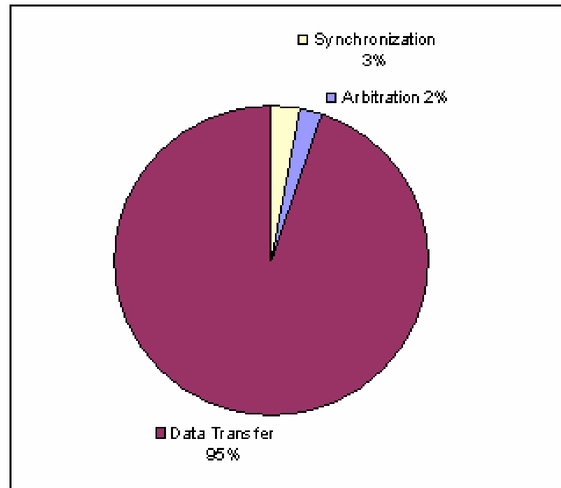

Figure 24. Distribution of communication tasks in transactions of CPU process in TLM1

7.2. TLM2 Performance Measurements

For TLM2, the estimated synchronization delays are shown in Table 3. The communication in this model represents close to 3% of total execution time. Again, the fastest synchronization scheme is achieved with 2 flag algorithm. As before, LPCM and RPCM processes are idly waiting for the flag reset in synchronization most of the time.

Synch. Scheme	**CPU Module: MP3_main()**	**L_IMDCT Module: lpcm()**	**R_IMDCT Module: rpcm()**
2 flags	41.75	1560.71	1560.82
1 flag in filters	42.01	1560.96	1561.08
1 flag in mp3_main	41.89	1560.85	1560.96
1 flag, polling	42.56	1561.52	1561.64
1 flag, interrupt	42.36	1561.32	1561.43
1 flag, shared memory	43.52	1562.48	1562.60
Total Execution of TLM2:	**1598.00**		

Table 3: Comm. delay estimates with different synch. schemes for processes in TLM2

Figures 25 and 26 graphically show the communication delay for CPU and IMDCT (LPCM and RPCM) processes with regard to used process synchronization schemes. The fastest synchronization scheme is the scheme using *2 flags* (1st row in Table 3), the slowest the scheme using *1 flag stored in shared memory* (6th row in Table 3). Figure 27 shows the distribution of communication tasks in communication of CPU process. As apposed to transfer in TLM1, the data transfer here takes a much smaller slice of total the communication delay, 20%, with the dominant influence of synchronization delay (80%), since the CPU process must wait for IMDCT filters to compute and output the data. Arbitration's influence in the total delay is negligible, with less than 1%.

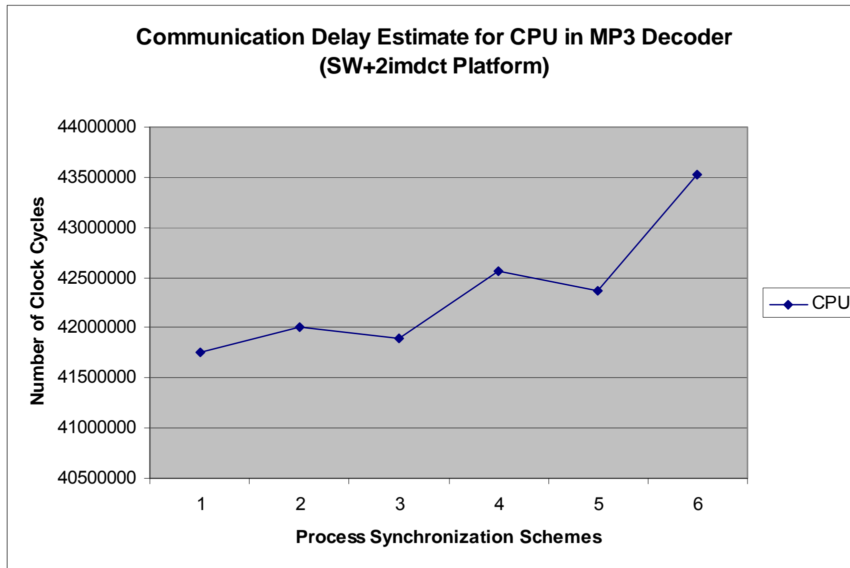

Figure 25: Communication delay estimates for CPU process in TLM2

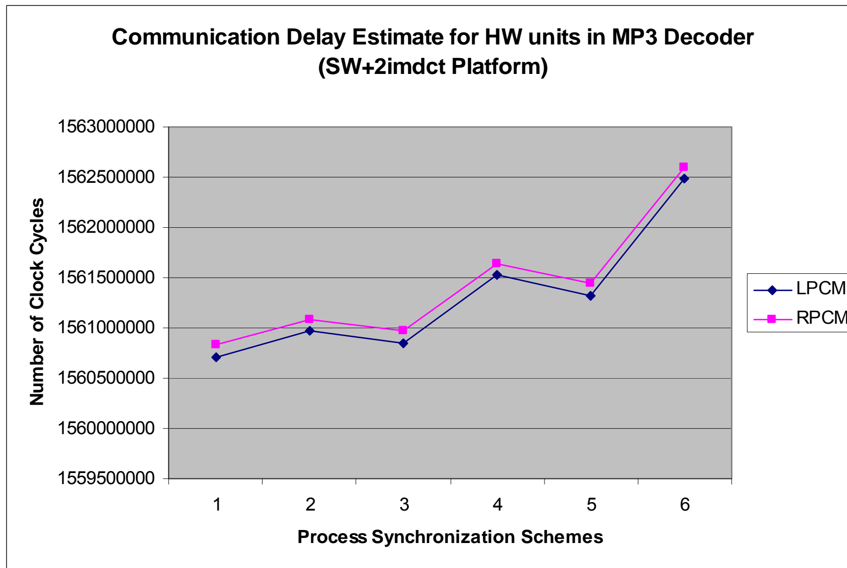

Figure 26: Communication delay estimates for LPCM and RPCM processes in TLM2

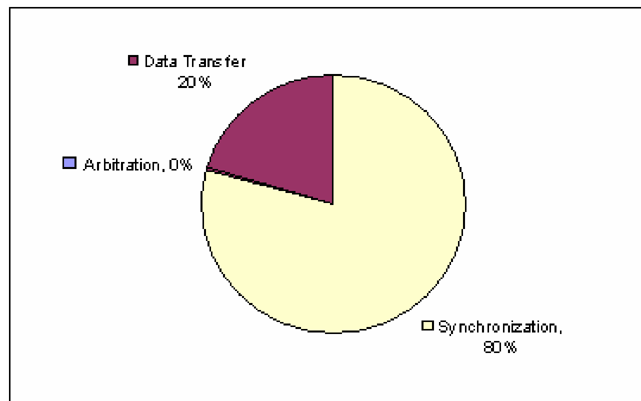

Figure 27: Distribution of communication tasks in transactions of CPU process in TLM2

7.3. TLM3 Performance Measurements

Table 4 compares the estimated communication delays simulated with TLM3 with different synchronization schemes. The communication of CPU process MP3_main takes 3.75% of the total execution of the TLM.

Synch. Scheme	CPU Module: MP3_main()	L_DCT Module: lfil()	R_DCT Module: rfil()	L_IMDCT Module: lpcm()	R_IMDCT Module: rpcm()
2 flags	55.81	1450.90	1450.89	1478.91	1478.91
1 flag in filters	56.24	1451.25	1451.25	1479.27	1479.27
1 flag in mp3_main	56.16	1451.33	1451.33	1479.34	1479.34
1 flag, polling	57.35	1452.44	1452.43	1480.45	1480.45
1 flag, interrupt	56.91	1452.00	1451.99	1480.01	1480.01
1 flag, shared memory	61.58	1456.66	1456.66	1484.68	1484.68
Total Execution of TLM3:	**1487.93**				

Table 4: Communication delay estimates with different synchronization schemes for processes in TLM3

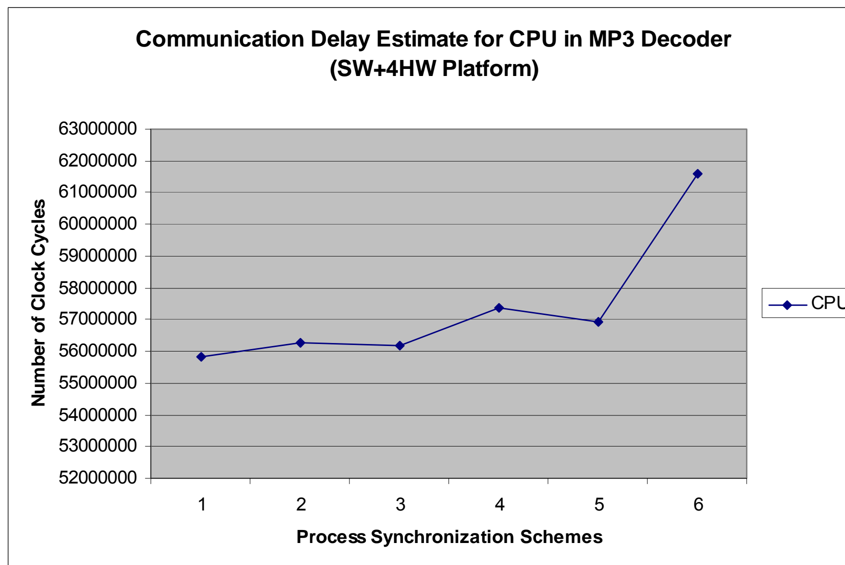

Figure 28. Communication delay estimates for CPU process in TLM3

Again, the most efficient synchronization scheme is the scheme using 2 flags (55.81 million cycles), vs the slowest scheme with 1 flag in shared memory (61.58 million cycles). Figures 28 and 29 show graphs with the said delays for both CPU and HW processes. Distribution of synchronization (71%), arbitration (1%) and data transfer (28%) in communication of CPU process in TLM3 is shown on Figure 30. The synchronization is again a dominant task in communication, because the CPU process is waiting idly for the output of the IMDCT and DCT filters.

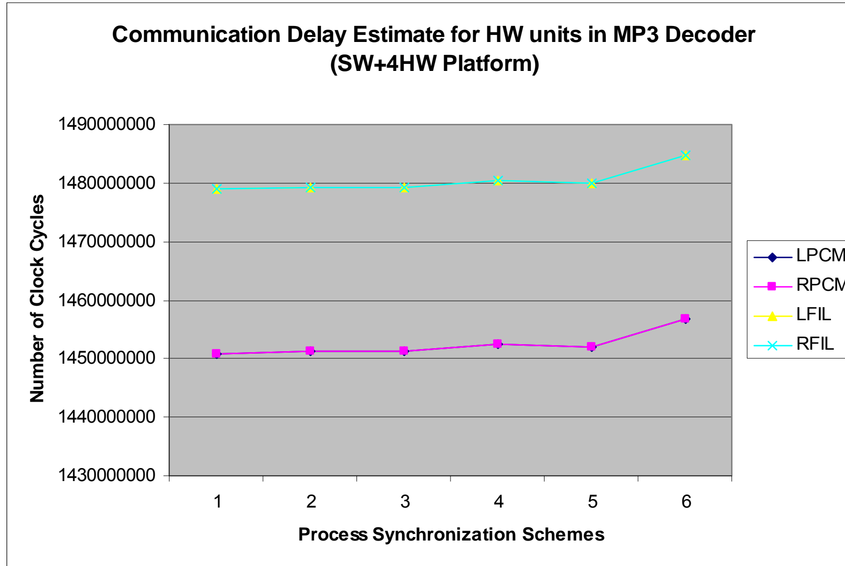

Figure 29: Communication delay estimates for HW processes in TLM3

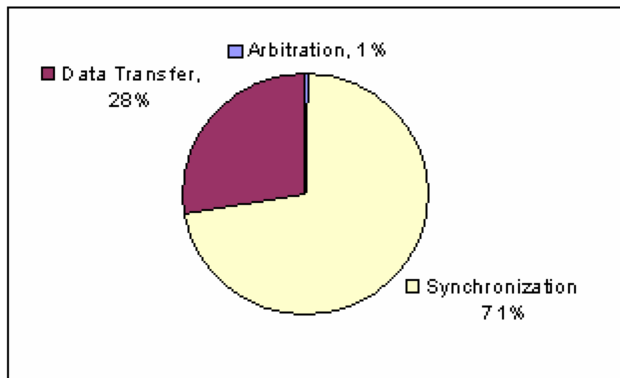

Figure 30: Distribution of communication tasks in transactions of CPU process in TLM3

Synch. Scheme	CPU Module: DecodeFrame()	CPU Module: SynthSample()	CPU Module: OutputSample()
2 flags	26.44	51.44	936.00
1 flag in filters	26.44	51.44	936.00
1 flag in mp3_main	26.44	51.45	936.00
1 flag, polling	26.45	51.46	936.01
1 flag, interrupt	26.45	51.45	936.01
1 flag, shared memory	26.47	51.47	936.03
Total Execution of TLM4:	**1378.13**		

Table 5: Communication delay estimates with different synchronization schemes for processes in TLM4

7.4. TLM4 Performance Measurements

Finally, for the pipelined platform in TLM4, Table 5 compares the estimated communication delays simulated with different synchronization schemes. Figure 31 shows the in graphical form. The fastest schemes are the 2-flag and single flag synchronization schemes, with 26.44. million cycles each.

The communication of CPU processes takes between 1.9% (for *DecodeFrame*() process) and 3.7% for *SynthSample*() process. The third process (*OutputSynth*()) is idle most of the time, so he waits for the synch. flag to be reset cca 60% of the total execution of the TLM.

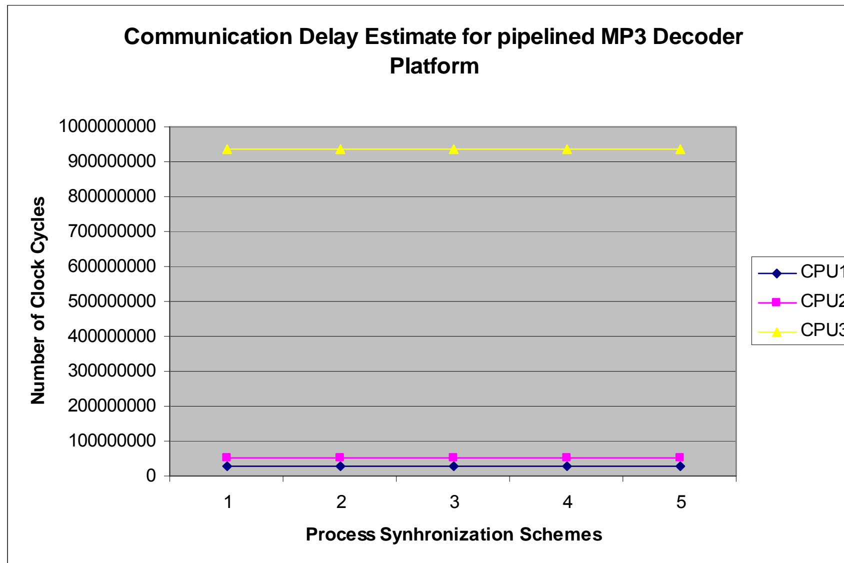

Figure 31 Communication delay estimates for CPU processes in TLM4

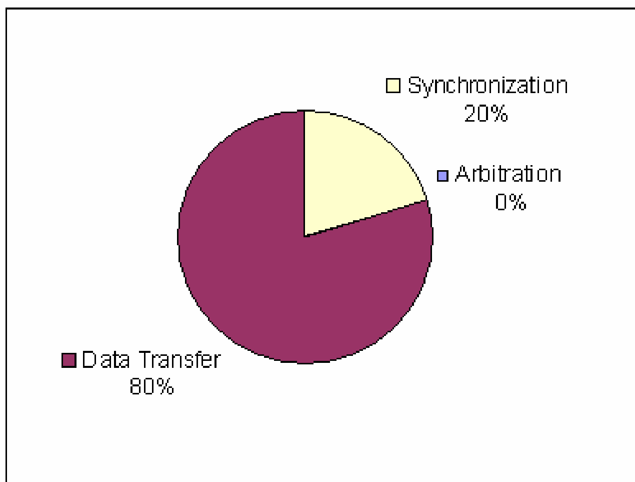

Figure 32: Distribution of communication tasks in transactions of *DecodeFrame*()in TLM4

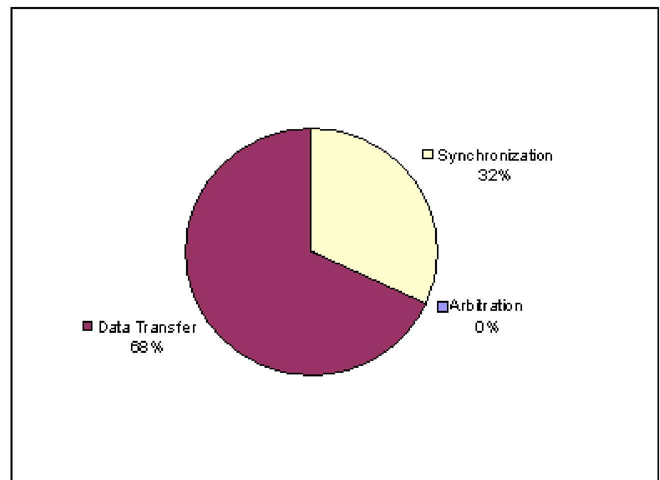

Figure 33: Distribution of communication tasks in transactions of *SynthSample*()in TLM4

Figures 32, 33 and 34 show the distribution of communication tasks in communication of *DecodeFrame*(),*SynthSample*() and *OutputSample*() process in TLM4, respectively. Process *DecodeFrame*() uses 20% of communication time to synchronize and 80% to transfer data. Process *SynthSample* () uses 32% of communication time to synchronize and 68% to transfer data, and process *OutputSample*() uses most of its communication delay for synchronization.

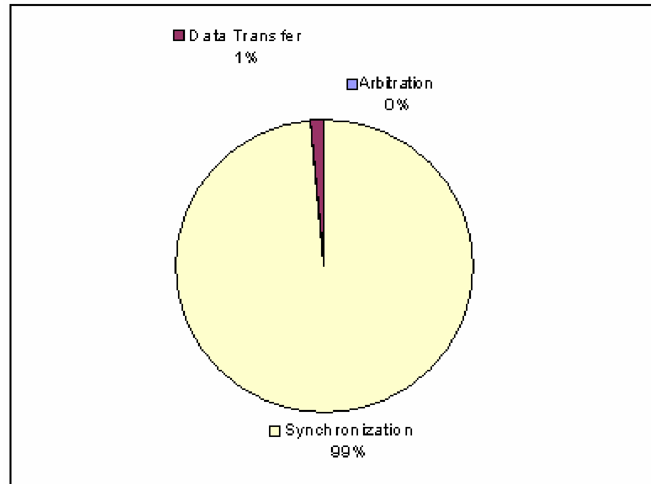

Figure 34: Distribution of communication tasks in transactions of *OutputSample*()in TLM4

8. Conclusion

In complex and communication intensive MPSoC systems, an efficient communication scheme influences the overall performance. Process synchronization is an important part of communication because it ensures a reliable data transfer. The traditional solutions for system communication with standard bus architectures implement synchronization by restricting the components to either bus masters or bus slaves. Since only bus masters have control over the bus communication, such solutions can limit the performance of system where all components have equal properties.

The report offers a broader set of possible solutions for process synchronization for MPSoC communication. Depending on the type of communication protocol and the architecture restrictions imposed by the MPSoC platform, the processes can synchronize with each other either directly or indirectly, locally or remotely, by always setting or always resetting the synchronization flag(s). All schemes are defined, classified and implemented in our modeling tool. Using the tool, the designer can insert each scheme in the transaction level model (TLM) generated by our tool. This report provides the designers with a comprehensive set of parameters and decisions that can guide them into more efficient communication scheme that is optimized for their particular application/architecture.

9 Acknowledgements

This work is part of the Embedded System Environment (ESE) project covering modeling and synthesis of MPSoCs.We would like to thank all members of ESE project for contributing to this report with their discussions and comments. Special thanks go to Lochi Yu for providing the general SystemC UBC model implementation and Pramod Chandraiah for providing the MP3 Decoder reference code.

References

[1] K. Steinhammer, A. Ademaj: "**HW implementation of the Time-Triggered Ethernet Controller**". Proceedings of the International Embedded Systems Symposium (IESS), Irvine, USA, 2007

[2] R. Obermaisser, H. Kopetz, C. El Salloum, B. Huber: "**Error containment in the time-triggered SoC architecture**". Proceedings of the International Embedded Systems Symposium (IESS), Irvine, USA, 2007

[3] W. Zheng, A. Sangiovanni-Vincentelli, M. Di Natale, C. Pinello, P. Giusto: "**Synthesis of task and message activation models in real-time distributed automotive systems**". Proceedings of Design Automation and Test in Europe Conference and Exposition (DATE), Nice, France, 2007.

[4] K.K. Ryu, E. Shin, V. J. Mooney: "**A Comparison of Five Different Multiprocessor SoC Bus Architectures**" Proceedings of EUROMICRO Symposium on Digital Systems Design, Warsaw, Poland 2001

[5] H. Cho, B. Ravindran, E.D. Jensen: "**Lock-Free Synchronization for Dynamic Embedded Real-Time Systems**". Proceedings of Design Automation and Test in Europe Conference and Exposition (DATE), Munich, Germany, 2006

[6] H. Huang, P. Pillai, K.G. Shin: "**Improving Wait-Free Algorithms for Interprocess Communication in Embedded Real-Time Systems**", Proceedings of USENIX Annual Technical Conference, Monterey, USA, 2002

[7.] M. Monchiero, G. Palermo, C. Silvano, O. Villa: "**An Efficient Synchronization Technique For Multiprocessor SoC**". Workshop on Memory Performance: Dealing with Applications , Systems and Architecture (MEDEA), Saint Louis, USA, 2005

[8] L. Sha, R. Rajkumar, J.P. Lehoczky: "**Priority Inheritance Protocols: An Approach to Real-Time Synchronization**", IEEE Transactions on Computers, Volume 39 , Issue 9, 1990

[9] D. Gajski, J.K. Peir: "**Essential Issues in Multiprocessor Systems**", IEEE Transactions on Computers, Volume 18, Issue 6, 1985

[10] L. Yu, S. Abdi, D. Gajski: "**Transaction Level Platform Modeling in SystemC for Multi-Processor Designs"**, Technical report TR07-01, UC Irvine, 2007

A Appendix

A.1 Source Code

A.1.1 Definition of module classes (output.cpp)

```
/* -----------------------------------------------------------------------------
 *   CPU, LPCM, RPCM, LFIL and RFIL modules f o r MPSoC Platform CPU+4 Modules (MP3 Decoder App)
 * -----------------------------------------------------------------------------
 */

#include <systemc.h>
#include "ubc.sc"

// Types of process synchronization
#define SYNC_2FLAGS
//#define SYNC_1FLAG
//#define SYNC_1FLAG_POLL
//#define SYNC_1FLAG_INTRPT
//#define SYNC_1FLAG_SM

/* -----------------------------------------------------------------------------
 * Definition of CPU module with MP3_main() process
 *-----------------------------------------------------------------------------
 */
extern "C" int mp3_main(void);
void *ptr_mp3_main;

class P_mp3_main: public sc_module{
  public:
		SC_HAS_PROCESS(P_mp3_main);
		P_mp3_main(sc_module_name name):sc_module(name){
				SC_THREAD(main);
		}
		sc_port<i_ubc> MyOPBbusport;
		void main(){
				ptr_mp3_main=this;
				mp3_main();
		}

};

extern "C" void send_P_ID_mp3_main_P_ID_lpcm_imdct36(void *ptr, int size){
		P_mp3_main *p = (P_mp3_main *) ptr_mp3_main;

#ifdef SYNC_2FLAGS
		p->MyOPBbusport->send_2flags(P_ID_mp3_main,P_ID_lpcm_imdct36,ptr,size, P_ID_mp3_main, P_ID_lpcm_imdct36);
#endif
#ifdef SYNC_1FLAG
		p->MyOPBbusport->send_1flag(P_ID_mp3_main,P_ID_lpcm_imdct36,ptr,size, P_ID_mp3_main, P_ID_lpcm_imdct36);
#endif
#ifdef SYNC_1FLAG_POLL
		p->MyOPBbusport->send_1flag_polling(P_ID_mp3_main,P_ID_lpcm_imdct36,ptr,size,P_ID_mp3_main,P_ID_lpcm_imdct36);
#endif
#ifdef SYNC_1FLAG_INTRPT
		p->MyOPBbusport->send_1flag_interrupt(P_ID_mp3_main,P_ID_lpcm_imdct36,ptr,size, P_ID_mp3_main,P_ID_lpcm_imdct36);
#endif
#ifdef SYNC_1FLAG_SM
		p->MyOPBbusport->send_1flag_sm(P_ID_mp3_main,P_ID_lpcm_imdct36,ptr,size, P_ID_mp3_main,P_ID_lpcm_imdct36);
#endif
}

extern "C" void recv_P_ID_mp3_main_P_ID_lpcm_imdct36(void *ptr, int size){
		P_mp3_main *p = (P_mp3_main*) ptr_mp3_main;
		unsigned int src= P_ID_lpcm_imdct36;
		unsigned int dest= P_ID_mp3_main;
```

```
#ifdef SYNC_2FLAGS
		p->MyOPBbusport->recv_2flags(P_ID_mp3_main,P_ID_lpcm_imdct36,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG
		p->MyOPBbusport->recv_1flag(P_ID_mp3_main,P_ID_lpcm_imdct36,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_POLL
		p->MyOPBbusport->recv_1flag_polling(P_ID_mp3_main,P_ID_lpcm_imdct36,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_INTRPT
		p->MyOPBbusport->recv_1flag_interrupt(P_ID_mp3_main,P_ID_lpcm_imdct36,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_SM
		p->MyOPBbusport->recv_1flag_sm(P_ID_mp3_main,P_ID_lpcm_imdct36,ptr,size, &src,&dest);
#endif
}

extern "C" void send_P_ID_mp3_main_P_ID_rpcm_imdct36(void *ptr, int size){
		P_mp3_main *p = (P_mp3_main *) ptr_mp3_main;

#ifdef SYNC_2FLAGS
		p->MyOPBbusport->send_2flags(P_ID_mp3_main,P_ID_rpcm_imdct36,ptr,size, P_ID_mp3_main, P_ID_rpcm_imdct36);
#endif
#ifdef SYNC_1FLAG
		p->MyOPBbusport->send_1flag(P_ID_mp3_main,P_ID_rpcm_imdct36,ptr,size, P_ID_mp3_main, P_ID_rpcm_imdct36);
#endif
#ifdef SYNC_1FLAG_POLL
		p->MyOPBbusport->send_1flag_polling(P_ID_mp3_main,P_ID_rpcm_imdct36,ptr,size, P_ID_mp3_main,P_ID_rpcm_imdct36);
#endif
#ifdef SYNC_1FLAG_INTRPT
		p->MyOPBbusport->send_1flag_interrupt(P_ID_mp3_main,P_ID_rpcm_imdct36,ptr,size, P_ID_mp3_main,P_ID_rpcm_imdct36);
#endif
#ifdef SYNC_1FLAG_SM
		p->MyOPBbusport->send_1flag_sm(P_ID_mp3_main,P_ID_rpcm_imdct36,ptr,size, P_ID_mp3_main,P_ID_rpcm_imdct36);
#endif
}

extern "C" void recv_P_ID_mp3_main_P_ID_rpcm_imdct36(void *ptr, int size){
		P_mp3_main *p = (P_mp3_main*) ptr_mp3_main;
		unsigned int src= P_ID_rpcm_imdct36;
		unsigned int dest= P_ID_mp3_main;
#ifdef SYNC_2FLAGS
		p->MyOPBbusport->recv_2flags(P_ID_mp3_main,P_ID_rpcm_imdct36,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG
		p->MyOPBbusport->recv_1flag(P_ID_mp3_main,P_ID_rpcm_imdct36,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_POLL
		p->MyOPBbusport->recv_1flag_polling(P_ID_mp3_main,P_ID_rpcm_imdct36,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_INTRPT
		p->MyOPBbusport->recv_1flag_interrupt(P_ID_mp3_main,P_ID_rpcm_imdct36,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_SM
		p->MyOPBbusport->recv_1flag_sm(P_ID_mp3_main,P_ID_rpcm_imdct36,ptr,size, &src,&dest);
#endif
}

extern "C" void send_P_ID_mp3_main_P_ID_lfil_dct32(void *ptr, int size){
		P_mp3_main *p = (P_mp3_main *) ptr_mp3_main;
#ifdef SYNC_2FLAGS
		p->MyOPBbusport->send_2flags(P_ID_mp3_main,P_ID_lfil_dct32,ptr,size, P_ID_mp3_main, P_ID_lfil_dct32);
#endif
#ifdef SYNC_1FLAG
		p->MyOPBbusport->send_1flag(P_ID_mp3_main,P_ID_lfil_dct32,ptr,size, P_ID_mp3_main, P_ID_lfil_dct32);
#endif
#ifdef SYNC_1FLAG_POLL
		p->MyOPBbusport->send_1flag_polling(P_ID_mp3_main,P_ID_lfil_dct32,ptr,size, P_ID_mp3_main, P_ID_lfil_dct32);
#endif
#ifdef SYNC_1FLAG_INTRPT
```

```
	p->MyOPBbusport->send_1flag_interrupt(P_ID_mp3_main,P_ID_lfil_dct32,ptr,size, P_ID_mp3_main, P_ID_lfil_dct32);
#endif
#ifdef SYNC_1FLAG_SM
	p->MyOPBbusport->send_1flag_sm(P_ID_mp3_main,P_ID_lfil_dct32,ptr,size, P_ID_mp3_main, P_ID_lfil_dct32);
#endif
}

extern "C" void recv_P_ID_mp3_main_P_ID_lfil_dct32(void *ptr, int size){
	P_mp3_main *p = (P_mp3_main*) ptr_mp3_main;
	unsigned int src= P_ID_lfil_dct32;
	unsigned int dest= P_ID_mp3_main;
#ifdef SYNC_2FLAGS
	p->MyOPBbusport->recv_2flags(P_ID_mp3_main,P_ID_lfil_dct32,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG
	p->MyOPBbusport->recv_1flag(P_ID_mp3_main,P_ID_lfil_dct32,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_POLL
	p->MyOPBbusport->recv_1flag_polling(P_ID_mp3_main,P_ID_lfil_dct32,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_INTRPT
	p->MyOPBbusport->recv_1flag_interrupt(P_ID_mp3_main,P_ID_lfil_dct32,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_SM
	p->MyOPBbusport->recv_1flag_sm(P_ID_mp3_main,P_ID_lfil_dct32,ptr,size, &src,&dest);
#endif
}


extern "C" void send_P_ID_mp3_main_P_ID_rfil_dct32(void *ptr, int size){
	P_mp3_main *p = (P_mp3_main *) ptr_mp3_main;
#ifdef SYNC_2FLAGS
	p->MyOPBbusport->send_2flags(P_ID_mp3_main,P_ID_rfil_dct32,ptr,size, P_ID_mp3_main, P_ID_rfil_dct32);
#endif
#ifdef SYNC_1FLAG
	p->MyOPBbusport->send_1flag(P_ID_mp3_main,P_ID_rfil_dct32,ptr,size, P_ID_mp3_main, P_ID_rfil_dct32);
#endif
#ifdef SYNC_1FLAG_POLL
	p->MyOPBbusport->send_1flag_polling(P_ID_mp3_main,P_ID_rfil_dct32,ptr,size, P_ID_mp3_main, P_ID_rfil_dct32);
#endif
#ifdef SYNC_1FLAG_INTRPT
	p->MyOPBbusport->send_1flag_interrupt(P_ID_mp3_main,P_ID_rfil_dct32,ptr,size, P_ID_mp3_main, P_ID_rfil_dct32);
#endif
#ifdef SYNC_1FLAG_SM
	p->MyOPBbusport->send_1flag_sm(P_ID_mp3_main,P_ID_rfil_dct32,ptr,size, P_ID_mp3_main, P_ID_rfil_dct32);
#endif
}

extern "C" void recv_P_ID_mp3_main_P_ID_rfil_dct32(void *ptr, int size){
	P_mp3_main *p = (P_mp3_main*) ptr_mp3_main;
	unsigned int src= P_ID_rfil_dct32;
	unsigned int dest= P_ID_mp3_main;
#ifdef SYNC_2FLAGS
	p->MyOPBbusport->recv_2flags(P_ID_mp3_main,P_ID_rfil_dct32,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG
	p->MyOPBbusport->recv_1flag(P_ID_mp3_main,P_ID_rfil_dct32,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_POLL
	p->MyOPBbusport->recv_1flag_polling(P_ID_mp3_main,P_ID_rfil_dct32,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_INTRPT
	p->MyOPBbusport->recv_1flag_interrupt(P_ID_mp3_main,P_ID_rfil_dct32,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_SM
	p->MyOPBbusport->recv_1flag_sm(P_ID_mp3_main,P_ID_rfil_dct32,ptr,size, &src,&dest);
#endif
}

/**************************** end of CPU module **********************************/
```

```
/* -----------------------------------------------------------------------------
 * Definition of LCPM module with lpcm_imdct36() process
 * -----------------------------------------------------------------------------
 */
extern "C" int lpcm_imdct36(void);
void *ptr_lpcm_imdct36;

class P_lpcm_imdct36: public sc_module{
  public:
		SC_HAS_PROCESS(P_lpcm_imdct36);
		P_lpcm_imdct36(sc_module_name name):sc_module(name){
			SC_THREAD(main);
		}
		sc_port<i_ubc> MyOPBbusport;
		void main(){
			ptr_lpcm_imdct36=this;
			lpcm_imdct36();
		}
};

extern "C" void recv_P_ID_lpcm_imdct36_P_ID_mp3_main(void *ptr, int size){
		P_lpcm_imdct36 *p = (P_lpcm_imdct36*) ptr_lpcm_imdct36;
		unsigned int src= P_ID_mp3_main;
		unsigned int dest= P_ID_lpcm_imdct36;

#ifdef SYNC_2FLAGS
		p->MyOPBbusport->recv_2flags(P_ID_lpcm_imdct36,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG
		p->MyOPBbusport->recv_1flag(P_ID_lpcm_imdct36,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_POLL
		p->MyOPBbusport->recv_1flag_polling(P_ID_lpcm_imdct36,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_INTRPT
		p->MyOPBbusport->recv_1flag_interrupt(P_ID_lpcm_imdct36,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_SM
		p->MyOPBbusport->recv_1flag_sm(P_ID_lpcm_imdct36,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
}

extern "C" void send_P_ID_lpcm_imdct36_P_ID_mp3_main(void *ptr, int size){
		P_lpcm_imdct36 *p = (P_lpcm_imdct36 *) ptr_lpcm_imdct36;

#ifdef SYNC_2FLAGS
	p->MyOPBbusport->send_2flags(P_ID_lpcm_imdct36,P_ID_mp3_main,ptr,size, P_ID_lpcm_imdct36, P_ID_mp3_main);
#endif
#ifdef SYNC_1FLAG
	p->MyOPBbusport->send_1flag(P_ID_lpcm_imdct36,P_ID_mp3_main,ptr,size, P_ID_lpcm_imdct36, P_ID_mp3_main);
#endif
#ifdef SYNC_1FLAG_POLL
	p->MyOPBbusport->send_1flag_polling(P_ID_lpcm_imdct36,P_ID_mp3_main,ptr,size,P_ID_lpcm_imdct36,P_ID_mp3_main);
#endif
#ifdef SYNC_1FLAG_INTRPT
	p->MyOPBbusport->send_1flag_interrupt(P_ID_lpcm_imdct36,P_ID_mp3_main,ptr,size,P_ID_lpcm_imdct36,P_ID_mp3_main);
#endif
#ifdef SYNC_1FLAG_SM
	p->MyOPBbusport->send_1flag_sm(P_ID_lpcm_imdct36,P_ID_mp3_main,ptr,size, P_ID_lpcm_imdct36, P_ID_mp3_main);
#endif
}

/****************************** end of LCPM module ********************************/
```

```
/* --------------------------------------------------------------------------------
 * Definition of RCPM module with rpcm_imdct36() process
 * --------------------------------------------------------------------------------
 */
extern "C" int rpcm_imdct36(void);
void *ptr_rpcm_imdct36;

class P_rpcm_imdct36: public sc_module{
  public:
        SC_HAS_PROCESS(P_rpcm_imdct36);
        P_rpcm_imdct36(sc_module_name name):sc_module(name){
                SC_THREAD(main);
        }
        sc_port<i_ubc> MyOPBbusport;
        void main(){
                ptr_rpcm_imdct36=this;
                rpcm_imdct36();
        }
};

extern "C" void recv_P_ID_rpcm_imdct36_P_ID_mp3_main(void *ptr, int size){
        P_rpcm_imdct36 *p = (P_rpcm_imdct36*) ptr_rpcm_imdct36;
        unsigned int src= P_ID_mp3_main;
        unsigned int dest= P_ID_rpcm_imdct36;

#ifdef SYNC_2FLAGS
        p->MyOPBbusport->recv_2flags(P_ID_rpcm_imdct36,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG
        p->MyOPBbusport->recv_1flag(P_ID_rpcm_imdct36,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_POLL
        p->MyOPBbusport->recv_1flag_polling(P_ID_rpcm_imdct36,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_INTRPT
        p->MyOPBbusport->recv_1flag_interrupt(P_ID_rpcm_imdct36,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_SM
        p->MyOPBbusport->recv_1flag_sm(P_ID_rpcm_imdct36,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
}

extern "C" void send_P_ID_rpcm_imdct36_P_ID_mp3_main(void *ptr, int size){
        P_rpcm_imdct36 *p = (P_rpcm_imdct36 *) ptr_rpcm_imdct36;

#ifdef SYNC_2FLAGS
    p->MyOPBbusport->send_2flags(P_ID_rpcm_imdct36,P_ID_mp3_main,ptr,size, P_ID_rpcm_imdct36, P_ID_mp3_main);
#endif
#ifdef SYNC_1FLAG
    p->MyOPBbusport->send_1flag(P_ID_rpcm_imdct36,P_ID_mp3_main,ptr,size, P_ID_rpcm_imdct36, P_ID_mp3_main);
#endif
#ifdef SYNC_1FLAG_POLL
    p->MyOPBbusport->send_1flag_polling(P_ID_rpcm_imdct36,P_ID_mp3_main,ptr,size,P_ID_rpcm_imdct36,P_ID_mp3_main);
#endif
#ifdef SYNC_1FLAG_INTRPT
    p->MyOPBbusport->send_1flag_interrupt(P_ID_rpcm_imdct36,P_ID_mp3_main,ptr,size,P_ID_rpcm_imdct36, P_ID_mp3_main);
#endif
#ifdef SYNC_1FLAG_SM
    p->MyOPBbusport->send_1flag_sm(P_ID_rpcm_imdct36,P_ID_mp3_main,ptr,size, P_ID_rpcm_imdct36, P_ID_mp3_main);
#endif
}

/*************************** end of RCPM module ***********************************/
```

```
/* ------------------------------------------------------------------------------
 * Definition of LFIL module with lfil_dct32() process
 * ------------------------------------------------------------------------------
 */
extern "C" int lfil_dct32(void);
void *ptr_lfil_dct32;

class P_lfil_dct32: public sc_module{
  public:
        SC_HAS_PROCESS(P_lfil_dct32);
        P_lfil_dct32(sc_module_name name):sc_module(name){
                SC_THREAD(main);
        }
        sc_port<i_ubc> MyOPBbusport;
        void main(){
                ptr_lfil_dct32=this;
                lfil_dct32();
        }
};

extern "C" void recv_P_ID_lfil_dct32_P_ID_mp3_main(void *ptr, int size){
        P_lfil_dct32 *p = (P_lfil_dct32*) ptr_lfil_dct32;
        unsigned int src= P_ID_mp3_main;
        unsigned int dest= P_ID_lfil_dct32;

#ifdef SYNC_2FLAGS
        p->MyOPBbusport->recv_2flags(P_ID_lfil_dct32,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG
        p->MyOPBbusport->recv_1flag(P_ID_lfil_dct32,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_POLL
        p->MyOPBbusport->recv_1flag_polling(P_ID_lfil_dct32,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_INTRPT
        p->MyOPBbusport->recv_1flag_interrupt(P_ID_lfil_dct32,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_SM
        p->MyOPBbusport->recv_1flag_sm(P_ID_lfil_dct32,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
}

extern "C" void send_P_ID_lfil_dct32_P_ID_mp3_main(void *ptr, int size){
        P_lfil_dct32 *p = (P_lfil_dct32 *) ptr_lfil_dct32;

#ifdef SYNC_2FLAGS
    p->MyOPBbusport->send_2flags(P_ID_lfil_dct32,P_ID_mp3_main,ptr,size, P_ID_lfil_dct32, P_ID_mp3_main);
#endif
#ifdef SYNC_1FLAG
    p->MyOPBbusport->send_1flag(P_ID_lfil_dct32,P_ID_mp3_main,ptr,size, P_ID_lfil_dct32, P_ID_mp3_main);
#endif
#ifdef SYNC_1FLAG_POLL
    p->MyOPBbusport->send_1flag_polling(P_ID_lfil_dct32,P_ID_mp3_main,ptr,size,P_ID_lfil_dct32,P_ID_mp3_main);
#endif
#ifdef SYNC_1FLAG_INTRPT
    p->MyOPBbusport->send_1flag_interrupt(P_ID_lfil_dct32,P_ID_mp3_main,ptr,size,P_ID_lfil_dct32,P_ID_mp3_main);
#endif
#ifdef SYNC_1FLAG_SM
    p->MyOPBbusport->send_1flag_sm(P_ID_lfil_dct32,P_ID_mp3_main,ptr,size, P_ID_lfil_dct32, P_ID_mp3_main);
#endif
}

/****************************** end of LFIL module ********************************/
```

```
/* -----------------------------------------------------------------------------
 * Definition of RFIL module with rfil_dct32() process
 * -----------------------------------------------------------------------------
 */
extern "C" int rfil_dct32(void);
void *ptr_rfil_dct32;

class P_rfil_dct32: public sc_module{
  public:
	SC_HAS_PROCESS(P_rfil_dct32);
	P_rfil_dct32(sc_module_name name):sc_module(name){
		SC_THREAD(main);
	}
	sc_port<i_ubc> MyOPBbusport;
	void main(){
		ptr_rfil_dct32=this;
		rfil_dct32();
	}
};

extern "C" void recv_P_ID_rfil_dct32_P_ID_mp3_main(void *ptr, int size){
	P_rfil_dct32 *p = (P_rfil_dct32*) ptr_rfil_dct32;
	unsigned int src= P_ID_mp3_main;
	unsigned int dest= P_ID_rfil_dct32;

#ifdef SYNC_2FLAGS
	p->MyOPBbusport->recv_2flags(P_ID_rfil_dct32,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG
	p->MyOPBbusport->recv_1flag(P_ID_rfil_dct32,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_POLL
	p->MyOPBbusport->recv_1flag_polling(P_ID_rfil_dct32,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_INTRPT
	p->MyOPBbusport->recv_1flag_interrupt(P_ID_rfil_dct32,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
#ifdef SYNC_1FLAG_SM
	p->MyOPBbusport->recv_1flag_sm(P_ID_rfil_dct32,P_ID_mp3_main,ptr,size, &src,&dest);
#endif
}

extern "C" void send_P_ID_rfil_dct32_P_ID_mp3_main(void *ptr, int size){
	P_rfil_dct32 *p = (P_rfil_dct32 *) ptr_rfil_dct32;

#ifdef SYNC_2FLAGS
    p->MyOPBbusport->send_2flags(P_ID_rfil_dct32,P_ID_mp3_main,ptr,size, P_ID_rfil_dct32, P_ID_mp3_main);
#endif
#ifdef SYNC_1FLAG
    p->MyOPBbusport->send_1flag(P_ID_rfil_dct32,P_ID_mp3_main,ptr,size, P_ID_rfil_dct32, P_ID_mp3_main);
#endif
#ifdef SYNC_1FLAG_POLL
    p->MyOPBbusport->send_1flag_polling(P_ID_rfil_dct32,P_ID_mp3_main,ptr,size,P_ID_rfil_dct32,P_ID_mp3_main);
#endif
#ifdef SYNC_1FLAG_INTRPT
    p->MyOPBbusport->send_1flag_interrupt(P_ID_rfil_dct32,P_ID_mp3_main,ptr,size,P_ID_rfil_dct32, P_ID_mp3_main);
#endif
#ifdef SYNC_1FLAG_SM
    p->MyOPBbusport->send_1flag_sm(P_ID_rfil_dct32,P_ID_mp3_main,ptr,size, P_ID_rfil_dct32, P_ID_mp3_main);
#endif
}

/*************************** end of RCPM module ***********************************/
```

```
/* -------------------------------------------------------------------------
 * Definition of Top module (testbench)
 * -------------------------------------------------------------------------
 */
class Top : public sc_module{
  public:
		//all PEs
		P_mp3_main *mp3_main_instance;
		P_lpcm_imdct36 *lpcm_imdct36_instance;
		P_rfil_dct32 *rpcm_imdct36_instance;
		P_lfil_dct32 *lfil_dct32_instance;
		P_rfil_dct32 *rfil_dct32_instance;
		//busses
		MyOPB *MyOPB_instance;

		Top(sc_module_name name, int argc, char **argv) : sc_module(name) {
			mp3_main_instance = new P_mp3_main("mp3_main");
			lpcm_imdct36_instance = new P_lpcm_imdct36("lpcm_imdct36");
			rpcm_imdct36_instance = new P_rpcm_imdct36("rpcm_imdct36");
			lfil_dct32_instance = new P_lfil_dct32("lfil_dct32");
			rfil_dct32_instance = new P_rfil_dct32("rfil_dct32");
			MyOPB_instance = new MyOPB("MyOPB");

			//PE connections to the buses
			mp3_main_instance->MyOPBbusport(*MyOPB_instance);
			lpcm_imdct36_instance->MyOPBbusport(*MyOPB_instance);
			rpcm_imdct36_instance->MyOPBbusport(*MyOPB_instance);
			lfil_dct32_instance->MyOPBbusport(*MyOPB_instance);
			rfil_dct32_instance->MyOPBbusport(*MyOPB_instance);
		}
};

int sc_main (int argc, char** argv) {
		Top top1("top",argc,argv);
		sc_start();
		return 0;
}

/*************************** end of Top module ***********************************/
```

A.1.2 Definitioon of UBC class (ubc.sc)

```
/*********UBC macros*********/
// Process IDs
#define P_ID_mp3_main           (unsigned int) 0
#define P_ID_lpcm_imdct36       (unsigned int) 1
#define P_ID_rpcm_imdct36       (unsigned int) 2
#define P_ID_lfil_dct32         (unsigned int) 3
#define P_ID_rfil_dct32         (unsigned int) 4
#define UBC_ERROR               (unsigned int) 0
#define UBC_SEND                (unsigned int) 1
#define UBC_RECV                (unsigned int) 2
#define UBC_INITIATOR           (unsigned int) 3
#define UBC_RESETTER            (unsigned int) 4
#define UBC_EITHER              (unsigned int) 5
#define UBC_READ                (unsigned int) 6
#define UBC_WRITE               (unsigned int) 7

//Address Table for UBC
#define ADDR_NONE                               (unsigned int) 0xffffffff
#define ADDR_MyOPB_mp3_main_lfil_dct32          (unsigned int) 10
#define ADDR_MyOPB_lfil_dct32_mp3_main          (unsigned int) 10
#define ADDR_MyOPB_mp3_main_rfil_dct32          (unsigned int) 11
#define ADDR_MyOPB_rfil_dct32_mp3_main          (unsigned int) 11
#define ADDR_MyOPB_mp3_main_lpcm_imdct36        (unsigned int) 12
#define ADDR_MyOPB_lpcm_imdct36_mp3_main        (unsigned int) 12
#define ADDR_MyOPB_mp3_main_rpcm_imdct36        (unsigned int) 13
#define ADDR_MyOPB_rpcm_imdct36_mp3_main        (unsigned int) 13

// SYNCH TYPES
// #define F_IN_P1
#define SYNCH_2FLAGS                    (unsigned int) 0
#define SYNCH_1FLAG                     (unsigned int) 1
#define SYNCH_1FLAG_POLL                (unsigned int) 2
#define SYNCH_1FLAG_INTRPT              (unsigned int) 3
#define SYNCH_1FLAG_SM                  (unsigned int) 4
#define INTERRUPT_HANDLER_DELAY (unsigned int)  10 // ns
#define POLL_FREQ_DELAY                 (unsigned int)  50 // ns
#define LOCAL_ACCESS_DELAY              (unsigned int)   1 // ns
#define BUS_ACCESS_DELAY                (unsigned int)   4 // ns
#define TRANSACTION1_DELAY_OPB   (unsigned int) 1 // ns
#define TRANSACTION2_DELAY_OPB   (unsigned int) 1 // ns
#define ARB_DELAY_OPB                   (unsigned int) 2 // ns
#define DATA_TRANSFER_DELAY_OPB(unsigned int) 1 // ns

/********** Interface Definitions ************/

class i_ubc : virtual public sc_interface {
  public:
        virtual void send_2flags (unsigned int MyProcID, unsigned int RecvProcID, void * data_ptr, unsigned int size,
                unsigned int SourceID, unsigned int DestID) = 0;
        virtual void recv_2flags (unsigned int MyProcID, unsigned int SendProcID, void * data_ptr, unsigned int size,
                unsigned int *SourceID, unsigned int *DestID) = 0;
        virtual void send_1flag (unsigned int MyProcID, unsigned int RecvProcID, void * data_ptr, unsigned int size,
                unsigned int SourceID, unsigned int DestID) = 0;
        virtual void recv_1flag (unsigned int MyProcID, unsigned int SendProcID, void * data_ptr, unsigned int size,
                unsigned int *SourceID, unsigned int *DestID) = 0;
        virtual void send_1flag_polling (unsigned int MyProcID, unsigned int RecvProcID, void * data_ptr, unsigned int size,
                unsigned int SourceID, unsigned int DestID) = 0;
        virtual void recv_1flag_polling (unsigned int MyProcID, unsigned int SendProcID, void * data_ptr, unsigned int size,
                unsigned int *SourceID, unsigned int *DestID) = 0;
        virtual void send_1flag_interrupt (unsigned int MyProcID, unsigned int RecvProcID, void * data_ptr, unsigned int size,
                unsigned int SourceID, unsigned int DestID) = 0;
        virtual void recv_1flag_interrupt (unsigned int MyProcID, unsigned int SendProcID, void * data_ptr, unsigned int size,
                unsigned int *SourceID, unsigned int *DestID) = 0;
         virtual void send_1flag_sm (unsigned int MyProcID, unsigned int RecvProcID, void * data_ptr, unsigned int size,
                unsigned int SourceID, unsigned int DestID) = 0;
        virtual void recv_1flag_sm (unsigned int MyProcID, unsigned int SendProcID, void * data_ptr, unsigned int size,
                unsigned int *SourceID, unsigned int *DestID) = 0;
```

```
        virtual void write (unsigned int MyProcID, unsigned int addr, void * data_ptr, unsigned int size) = 0;
        virtual void read  (unsigned int MyProcID, unsigned int addr, void * data_ptr, unsigned int size) = 0;
        virtual void MemoryAccess (unsigned int LowAddr, unsigned int HighAddr, unsigned char *local_mem) = 0;
};

/*****************Bus Definitions***************/
class MyOPB : public sc_channel, public i_ubc{
  private:
        // variables and events for data transfer
        void *DataPtr;
        unsigned int BusAddress;
        unsigned int RdWr;              // UBC_READ or UBC_WRITE
        unsigned int DataSize;          // in bytes
        sc_event AddrSet;               // notifies setting of address bus

        // routing info variables
        unsigned int Source;
        unsigned int Destination;

        // synchronization schemes variables, 2 flags and events for each of process pairs
        bool f1_mp3_main_lpcm_imdct36, f2_mp3_main_lpcm_imdct36;
        sc_event ev_f1_read_mp3_main_lpcm_imdct36, ev_f2_read_mp3_main_lpcm_imdct36;
        bool f1_mp3_main_rpcm_imdct36, f2_mp3_main_rpcm_imdct36;
        sc_event ev_f1_read_mp3_main_rpcm_imdct36, ev_f2_read_mp3_main_rpcm_imdct36;
        bool f1_mp3_main_lfil_dct32, f2_mp3_main_lfil_dct32;
        sc_event ev_f1_read_mp3_main_lfil_dct32, ev_f2_read_mp3_main_lfil_dct32;
        bool f1_mp3_main_rfil_dct32, f2_mp3_main_rfil_dct32;
        sc_event ev_f1_read_mp3_main_rfil_dct32, ev_f2_read_mp3_main_rfil_dct32;

        // abribtration variables
        sc_mutex f_mtx;

        void LocalRead (void) {
                f_mtx.lock();
                wait(LOCAL_ACCESS_DELAY, SC_NS);
                f_mtx.unlock();
                return;
        }

        void LocalWrite (bool* flag, bool value) {
                f_mtx.lock();
                wait(LOCAL_ACCESS_DELAY, SC_NS);
                *flag = value;
                f_mtx.unlock();
                return;
        }

        void BusRead (void) {
                f_mtx.lock();
                wait(BUS_ACCESS_DELAY, SC_NS);
                f_mtx.unlock();
                return;
        }

        void BusWrite (bool* flag, bool value) {
                f_mtx.lock();
                wait(BUS_ACCESS_DELAY, SC_NS);
                *flag = value;
                f_mtx.unlock();
                return;
        }

        sc_mutex ubc_arbiter; // a very primitive arbiter indeed

        // Simple methods for arbitration
        void ArbiterRequest (unsigned int ProcID) {
                wait(ARB_DELAY_OPB, SC_NS);
                ubc_arbiter.lock();
                return;
        }
```

```
void ArbiterRelease (unsigned int ProcID) {
	wait(ARB_DELAY_OPB, SC_NS);
	ubc_arbiter.unlock();
	return;
}

unsigned int Synchronize_2flags(unsigned int MyID, unsigned int PartnerID) {
	bool *flag1, *flag2;
	sc_event *ev_flag1_read;
	sc_event *ev_flag2_read;

	if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_lpcm_imdct36)) ||
		((PartnerID == P_ID_mp3_main) && (MyID == P_ID_lpcm_imdct36)) ){
			flag1 = &f1_mp3_main_lpcm_imdct36;
			flag2 = &f2_mp3_main_lpcm_imdct36;
			ev_flag1_read = &ev_f1_read_mp3_main_lpcm_imdct36;
			ev_flag2_read = &ev_f2_read_mp3_main_lpcm_imdct36;
	} else if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_rpcm_imdct36)) ||
		 ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_rpcm_imdct36)) ){
			flag1 = &f1_mp3_main_rpcm_imdct36;
			flag2 = &f2_mp3_main_rpcm_imdct36;
			ev_flag1_read = &ev_f1_read_mp3_main_rpcm_imdct36;
			ev_flag2_read = &ev_f2_read_mp3_main_rpcm_imdct36;
	} else if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_lfil_dct32)) ||
			((PartnerID == P_ID_mp3_main) && (MyID == P_ID_lfil_dct32)) ){
			flag1 = &f1_mp3_main_lfil_dct32;
			flag2 = &f2_mp3_main_lfil_dct32;
			ev_flag1_read = &ev_f1_read_mp3_main_lfil_dct32;
			ev_flag2_read = &ev_f2_read_mp3_main_lfil_dct32;
	} else if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_rfil_dct32)) ||
			((PartnerID == P_ID_mp3_main) && (MyID == P_ID_rfil_dct32)) ){
			flag1 = &f1_mp3_main_rfil_dct32;
			flag2 = &f2_mp3_main_rfil_dct32;
			ev_flag1_read = &ev_f1_read_mp3_main_rfil_dct32;
			ev_flag2_read = &ev_f2_read_mp3_main_rfil_dct32;
	} else {
		printf("Error in Sync (2 flags)\n");
		exit(1);
	}
	// mp3_main has f1
	if (MyID == P_ID_mp3_main) {
		ArbiterRequest(MyID);
		LocalRead();   // f1, wait(1,SC_NS);
		if (*flag1==false){
			BusWrite(flag2, true); // wait(4,SC_NS); f2=true;
			ArbiterRelease(MyID);
			wait(*ev_flag2_read);
			return UBC_INITIATOR;
		}else{
			LocalWrite(flag1, false); // wait(1,SC_NS); f1=false;
			ev_flag1_read->notify();
			return UBC_RESETTER;     // i.e. start data transfer
		}
	} else if (PartnerID == P_ID_mp3_main) {
		ArbiterRequest(MyID);
		LocalRead();  // f2, wait(1,SC_NS);
		if (*flag2==false){
			BusWrite(flag1, true); // wait(4,SC_NS); f1=true;
			ArbiterRelease(MyID);
			wait(*ev_flag1_read);
			return UBC_INITIATOR;
		}else{
			LocalWrite(flag2, false); // f2=false; wait(1,SC_NS);
			ev_flag2_read->notify();
			return UBC_RESETTER;   // i.e. start data transfer
		}
	}else{
		printf("Synchronization error: MyID=%i,PartnerID=%i\n",MyID,PartnerID);
		exit(1);
```

```
		}
}

// PE2 without the flag
unsigned int Synchronize_1flag_in_pe1 (unsigned int MyID, unsigned int PartnerID) {
		bool *flag1;
		sc_event *ev_flag1_read;

	if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_lpcm_imdct36)) ||
	     ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_lpcm_imdct36)) ){
		flag1 = &f1_mp3_main_lpcm_imdct36;
		ev_flag1_read = &ev_f1_read_mp3_main_lpcm_imdct36;
	} else if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_rpcm_imdct36)) ||
	     ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_rpcm_imdct36)) ){
		flag1 = &f1_mp3_main_rpcm_imdct36;
		ev_flag1_read = &ev_f1_read_mp3_main_rpcm_imdct36;
	} else if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_lfil_dct32)) ||
	     ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_lfil_dct32)) ){
		flag1 = &f1_mp3_main_lfil_dct32;
		ev_flag1_read = &ev_f1_read_mp3_main_lfil_dct32;
	} else if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_rfil_dct32)) ||
	     ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_rfil_dct32)) ){
		flag1 = &f1_mp3_main_rfil_dct32;
		ev_flag1_read = &ev_f1_read_mp3_main_rfil_dct32;
	} else {
	     printf("Error in Sync (1 flag in PE1)\n");
	     exit(1);
	}
		// PE2 without the flag
		if (PartnerID == P_ID_mp3_main) {
			ArbiterRequest(MyID);
			BusRead(); // f1, wait(4,SC_NS);
			if (*flag1 == false){
				BusWrite(flag1, true); // wait(4,SC_NS); f1=true;
				ArbiterRelease(MyID);
				wait(*ev_flag1_read);
				return UBC_INITIATOR;
			}else{
				BusWrite(flag1, false); // f1=false; wait(4,SC_NS); //ines
				ev_flag1_read->notify();
				return UBC_RESETTER; // i.e. start data transfer
			}
		}
		// PE1 with the flag
		else if (MyID==P_ID_mp3_main) {
			LocalRead(); // f1, wait(1,SC_NS);
			if (*flag1==false){
				LocalWrite(flag1, true); // wait(1,SC_NS); f1=true;
				wait(*ev_flag1_read);
				return UBC_INITIATOR;
			}else{
				LocalWrite(flag1, false); // f1=false; wait(1,SC_NS);
				ev_flag1_read->notify();
				ArbiterRequest(MyID);
				return UBC_RESETTER;
			}
		}else{
			printf("Synchronization error: MyID=%i,PartnerID=%i\n",MyID,PartnerID);
			exit(1);
		}
}

// PE1 without the flag
unsigned int Synchronize_1flag_in_pe2 (unsigned int MyID, unsigned int PartnerID) {
		bool *flag2;
		sc_event *ev_flag2_read;

		if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_lpcm_imdct36)) ||
			((PartnerID == P_ID_mp3_main) && (MyID == P_ID_lpcm_imdct36)) ){
			flag2 = &f2_mp3_main_lpcm_imdct36;
```

```
                ev_flag2_read = &ev_f2_read_mp3_main_lpcm_imdct36;
        } else if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_rpcm_imdct36)) ||
            ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_rpcm_imdct36)) ){
                flag2 = &f2_mp3_main_rpcm_imdct36;
                ev_flag2_read = &ev_f2_read_mp3_main_rpcm_imdct36;
        } else if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_lfil_dct32)) ||
            ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_lfil_dct32)) ){
                flag2 = &f2_mp3_main_lfil_dct32;
                ev_flag2_read = &ev_f2_read_mp3_main_lfil_dct32;
        } else if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_rfil_dct32)) ||
            ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_rfil_dct32)) ){
                flag2 = &f2_mp3_main_rfil_dct32;
                ev_flag2_read = &ev_f2_read_mp3_main_rfil_dct32;
        } else {
            printf("Error in Sync (1 flag in PE2)\n");
            exit(1);
        }

        // PE1 without the flag
        if (MyID==P_ID_mp3_main) {
                ArbiterRequest(MyID);
                BusRead(); // f2, wait(4,SC_NS);
                if (*flag2==false){
                        BusWrite(flag2, true); // wait(4,SC_NS); f2=true;
                        ArbiterRelease(MyID);
                        wait(*ev_flag2_read);
                        return UBC_INITIATOR;
                }else{
                        BusWrite(flag2, false); // f2=false; wait(4,SC_NS); //ines
                        ev_flag2_read->notify();
                        return UBC_RESETTER; // i.e. start data transfer
                }
        }
        // PE2 with the flag
        else if (PartnerID==P_ID_mp3_main) {
                LocalRead(); // f2, wait(1,SC_NS);
                if (*flag2==false){
                        LocalWrite(flag2, true); // wait(1,SC_NS); f2=true;
                        wait(*ev_flag2_read);
                        return UBC_INITIATOR;
                }else{
                        LocalWrite(flag2, false); // f2=false; wait(1,SC_NS);
                        ev_flag2_read->notify();
                        ArbiterRequest(MyID);
                        return UBC_RESETTER;
                }
        }else{
                printf("Synchronization error: MyID=%i,PartnerID=%i\n",MyID,PartnerID);
                exit(1);
        }
}

// mp3_main is the master without the flag
unsigned int Synchronize_1flag_polling(unsigned int MyID, unsigned int PartnerID) {
        bool *flag2;
        sc_event *ev_flag2_read;

        if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_lpcm_imdct36)) ||
            ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_lpcm_imdct36)) ){
                flag2 = &f2_mp3_main_lpcm_imdct36;
                ev_flag2_read = &ev_f2_read_mp3_main_lpcm_imdct36;
        } else if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_rpcm_imdct36)) ||
            ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_rpcm_imdct36)) ){
                flag2 = &f2_mp3_main_rpcm_imdct36;
                ev_flag2_read = &ev_f2_read_mp3_main_rpcm_imdct36;
        } else if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_lfil_dct32)) ||
            ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_lfil_dct32)) ){
                flag2 = &f2_mp3_main_lfil_dct32;
                ev_flag2_read = &ev_f2_read_mp3_main_lfil_dct32;
        } else if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_rfil_dct32)) ||
```

```
        ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_rfil_dct32)) ){
            flag2 = &f2_mp3_main_rfil_dct32;
            ev_flag2_read = &ev_f2_read_mp3_main_rfil_dct32;
    } else {
        printf("Error in Sync (polling, flag in PE2)\n");
        exit(1);
    }
     // mp3_main is master without the flag
     if (MyID == P_ID_mp3_main) {
            while(1) {
                    ArbiterRequest(MyID);
                    BusRead(); // f1, wait(4,SC_NS);
                    if (*flag2==false){
                            ArbiterRelease(MyID);
                            wait(POLL_FREQ_DELAY, SC_NS);   // poll frequency is 50ns
                    } else {
                            break;
                    }
            }
            BusWrite(flag2, false); // f2=false; wait(4,SC_NS);
            ev_flag2_read->notify();
            ArbiterRelease(MyID);
            return UBC_RESETTER;  // i.e. start data transfer
     }
     // dct32 is slave with the flag
     else if (PartnerID == P_ID_mp3_main) {
            LocalWrite(flag2, true); // wait(1,SC_NS); f1=true;
            wait(*ev_flag2_read);
            return UBC_INITIATOR;
     }else{
            printf("Synchronization error: MyID=%i,PartnerID=%i\n",MyID,PartnerID);
            exit(1);
     }
}

// mp3_main is the master with the flag
unsigned int Synchronize_1flag_interrupt(unsigned int MyID, unsigned int PartnerID) {
     bool *flag1;
    sc_event *ev_flag1_read;

    if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_lpcm_imdct36)) ||
        ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_lpcm_imdct36)) ){
            flag1 = &f1_mp3_main_lpcm_imdct36;
            ev_flag1_read = &ev_f1_read_mp3_main_lpcm_imdct36;
    } else if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_rpcm_imdct36)) ||
        ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_rpcm_imdct36)) ){
            flag1 = &f1_mp3_main_rpcm_imdct36;
            ev_flag1_read = &ev_f1_read_mp3_main_rpcm_imdct36;
    } else if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_lfil_dct32)) ||
        ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_lfil_dct32)) ){
            flag1 = &f1_mp3_main_lfil_dct32;
            ev_flag1_read = &ev_f1_read_mp3_main_lfil_dct32;
    } else if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_rfil_dct32)) ||
        ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_rfil_dct32)) ){
            flag1 = &f1_mp3_main_rfil_dct32;
            ev_flag1_read = &ev_f1_read_mp3_main_rfil_dct32;
    } else {
        printf("Error in Sync (interrupt, flag in PE1)\n");
        exit(1);
    }
     // mp3_main is master with the flag
     if (MyID == P_ID_mp3_main) {
            LocalRead(); // f1, wait(1,SC_NS);
            while (*flag1==false){
                    wait(*ev_flag1_read);
            }
            wait(INTERRUPT_HANDLER_DELAY, SC_NS);
            LocalWrite(flag1, false);  // f1=false; wait(1,SC_NS);
            return UBC_RESETTER;     // i.e. start data transfer
     }
```

```
        // partner component is slave without the flag
        else if (PartnerID == P_ID_mp3_main) {
                *flag1 = true;
                ev_flag1_read->notify();
                return UBC_INITIATOR;
        }else{
                printf("Synchronization error: MyID=%i,PartnerID=%i\n",MyID,PartnerID);
                exit(1);
        }
}

unsigned int Synchronize_1flag_sm(unsigned int MyID, unsigned int PartnerID, unsigned int Type) {
        bool *flag1;
       sc_event *ev_flag1_read;

       if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_lpcm_imdct36)) ||
            ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_lpcm_imdct36)) ){
                flag1 = &f1_mp3_main_lpcm_imdct36;
                ev_flag1_read = &ev_f1_read_mp3_main_lpcm_imdct36;
       } else if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_rpcm_imdct36)) ||
            ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_rpcm_imdct36)) ){
                flag1 = &f1_mp3_main_rpcm_imdct36;
                ev_flag1_read = &ev_f1_read_mp3_main_rpcm_imdct36;
       } else if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_lfil_dct32)) ||
            ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_lfil_dct32)) ){
                flag1 = &f1_mp3_main_lfil_dct32;
                ev_flag1_read = &ev_f1_read_mp3_main_lfil_dct32;
       } else if ( ((MyID == P_ID_mp3_main) && (PartnerID == P_ID_rfil_dct32)) ||
            ((PartnerID == P_ID_mp3_main) && (MyID == P_ID_rfil_dct32)) ){
                flag1 = &f1_mp3_main_rfil_dct32;
                ev_flag1_read = &ev_f1_read_mp3_main_rfil_dct32;
       } else {
            printf("Error in Sync (1 flag in PE1)\n");
            exit(1);
       }
        // PE is the reader/resetter
        if (Type == UBC_RECV) {
                while(1) {
                        ArbiterRequest(MyID);
                        BusRead(); // f1, wait(4,SC_NS);
                        if (*flag1 == false){
                                ArbiterRelease(MyID);
                                wait(POLL_FREQ_DELAY, SC_NS);   // poll frequency is 50ns
                        } else {
                                break;
                        }
                }
                BusWrite(flag1, false); // f1=false; wait(4,SC_NS);
                ArbiterRelease(MyID);
                ev_flag1_read->notify();
                return UBC_INITIATOR;
        }
        // PE is the writer/setter
        else if (Type == UBC_SEND) {
                while(1) {
                        ArbiterRequest(MyID);
                        BusRead(); // f1, wait(4,SC_NS);
                        if (*flag1 == true){
                                ArbiterRelease(MyID);
                                wait(POLL_FREQ_DELAY, SC_NS);   // poll frequency is 50ns
                        } else {
                                break;
                        }
                }
                BusWrite(flag1, true); // f1=true; wait(4,SC_NS);
                ArbiterRelease(MyID);
                wait(*ev_flag1_read);
                return UBC_RESETTER;  // i.e. start data transfer
        }else{
                printf("Synchronization error: MyID=%i,PartnerID=%i\n",MyID,PartnerID);
```

```
                        exit(1);
                }
    }

  public:
        // constructor
        char busid[100];
        MyOPB(sc_module_name name) : sc_channel(name) {strcpy(busid,"MyOPB");}

        void write (unsigned int MyProcID, unsigned int addr,void * data_ptr, unsigned int size) {
                ArbiterRequest (MyProcID);
                wait_ubc_data_transfer (TRANSACTION1_DELAY_OPB, "MyOPB",true);
                DataPtr = data_ptr;// setting the UBC data pointer
                DataSize = size;   // setting the size
                RdWr = UBC_WRITE;  // this is a write
                BusAddress = addr; // addressing
                AddrSet.notify();  // notification that data on bus is valid
                wait_ubc_data_transfer (TRANSACTION2_DELAY_OPB*size-1, "MyOPB",true);
                BusAddress = ADDR_NONE;
                ArbiterRelease (MyProcID);
                return;
        }
         // end of write method
        void read  (unsigned int MyProcID, unsigned int addr,void * data_ptr, unsigned int size) {
                ArbiterRequest (MyProcID);
                DataSize = size;   // setting the size
                RdWr = UBC_READ;   // this is a read
                BusAddress = addr; // addressing
                AddrSet.notify();          // notification that data on bus is valid
                wait_ubc_data_transfer (TRANSACTION1_DELAY_OPB, "MyOPB",true);
                memcpy (data_ptr, DataPtr, size);   // copy data into local memory
                wait_ubc_data_transfer (TRANSACTION2_DELAY_OPB*size-1, "MyOPB",true);
                BusAddress = ADDR_NONE;
                ArbiterRelease (MyProcID);
                return;
        } // end of read method

/***************************** FIND ROUTE ******************************************************/

        unsigned int FindRoute(unsigned int MyID, unsigned int PartnerID, unsigned int type,
                          unsigned int SourceID, unsigned int DestID) {
                if(type == UBC_SEND){
                        switch(MyID){
                                case P_ID_lpcm_imdct36:
                                        return ADDR_MyOPB_lpcm_imdct36_mp3_main;
                                case P_ID_rpcm_imdct36:
                                        return ADDR_MyOPB_rpcm_imdct36_mp3_main;
                                case P_ID_lfil_dct32:
                                        return ADDR_MyOPB_lfil_dct32_mp3_main;
                                case P_ID_rfil_dct32:
                                        return ADDR_MyOPB_rfil_dct32_mp3_main;
                                case P_ID_mp3_main:
                                        switch(PartnerID){
                                                case P_ID_lpcm_imdct36:
                                                        return ADDR_MyOPB_mp3_main_lpcm_imdct36;
                                                case P_ID_rpcm_imdct36:
                                                        return ADDR_MyOPB_mp3_main_rpcm_imdct36;
                                                case P_ID_lfil_dct32:
                                                        return ADDR_MyOPB_mp3_main_lfil_dct32;
                                                case P_ID_rfil_dct32:
                                                        return ADDR_MyOPB_mp3_main_rfil_dct32;
                                                default:
                                                        printf("Send routing error: undefined PartnerID\n");
                                                        exit(1);
                                        }
                                default:
                                        printf("Send routing error: undefined MyID\n");
                                        exit(1);
                                }
                } else if (type == UBC_RECV) {
```

```
            switch(PartnerID){
                case P_ID_lpcm_imdct36:
                    return ADDR_MyOPB_lpcm_imdct36_mp3_main;
                case P_ID_rpcm_imdct36:
                    return ADDR_MyOPB_rpcm_imdct36_mp3_main;
                case P_ID_lfil_dct32:
                    return ADDR_MyOPB_lfil_dct32_mp3_main;
                case P_ID_rfil_dct32:
                    return ADDR_MyOPB_rfil_dct32_mp3_main;
                case P_ID_mp3_main:
                    switch(MyID){
                        case P_ID_lpcm_imdct36:
                            return ADDR_MyOPB_mp3_main_lpcm_imdct36;
                        case P_ID_rpcm_imdct36:
                            return ADDR_MyOPB_mp3_main_rpcm_imdct36;
                        case P_ID_lfil_dct32:
                            return ADDR_MyOPB_mp3_main_lfil_dct32;
                        case P_ID_rfil_dct32:
                            return ADDR_MyOPB_mp3_main_rfil_dct32;
                        default:
                            printf("Recv routing error: undefined MyID\n");
                            exit(1);
                    }
                default:
                    printf("Recv routing error: undefined PartnerID\n");
                    exit(1);
            }
        } else {
            printf("Routing error: undefined transfer type\n");
            exit(1);
        }
        return 0;
    }

/***************************** SEND DATA *****************************************************/

  unsigned int SendData(unsigned int MyProcID, unsigned int SourceID, unsigned int DestID,
        void * data_ptr, unsigned int size, unsigned int Mode, unsigned int Address,
        unsigned int SynchType) {
        int i, n;

        if (Mode == UBC_INITIATOR){
            while(BusAddress != Address){
            wait(AddrSet);
        }
        DataPtr = data_ptr;
        Source = SourceID;
        Destination = DestID;
        wait(TRANSACTION1_DELAY_OPB, SC_NS);
        wait(TRANSACTION2_DELAY_OPB*size-1,SC_NS);
    }
    else if (Mode == UBC_RESETTER){
        if((SynchType != SYNCH_2FLAGS) && (SynchType != SYNCH_1FLAG))
            ArbiterRequest (MyProcID);
            BusAddress = Address;
            DataPtr = data_ptr;
            Source = SourceID;
            Destination = DestID;
            wait (TRANSACTION1_DELAY_OPB, SC_NS);
            AddrSet.notify();
            wait (TRANSACTION2_DELAY_OPB*size-1, SC_NS);
            BusAddress = ADDR_NONE;
            ArbiterRelease (MyProcID);
        }
        return 0;
    }

/***************************** RECV DATA *****************************************************/
```

```
unsigned int RecvData(unsigned int MyProcID, unsigned int * SourceID, unsigned int * DestID,
        void * data_ptr, unsigned int size, unsigned int Mode, unsigned int Address,
        unsigned int SynchType) {
                int i, n;
                unsigned int time1, time2;

                if (Mode == UBC_INITIATOR){
                        while(BusAddress!=Address){
                                wait(AddrSet);
                        }
                        memcpy (data_ptr, DataPtr, size);
                        *SourceID = Source;
                        *DestID = Destination;
                        wait (TRANSACTION1_DELAY_OPB, SC_NS);
                        wait (TRANSACTION2_DELAY_OPB*size-1, SC_NS);
                } else if (Mode == UBC_RESETTER) {
                        if((SynchType != SYNCH_2FLAGS) && (SynchType != SYNCH_1FLAG))
                                ArbiterRequest (MyProcID);
                        BusAddress = Address;
                        AddrSet.notify();
                        wait (TRANSACTION1_DELAY_OPB, SC_NS);
                        memcpy (data_ptr, DataPtr, size);
                        *SourceID = Source;
                        *DestID = Destination;
                        wait (TRANSACTION2_DELAY_OPB*size-1, SC_NS);
                        BusAddress = ADDR_NONE;
                        ArbiterRelease (MyProcID);
                }
                return 0;
        }

/****************************** 2 FLAGS send/recv ***********************************************/

        void send_2flags (unsigned int MyProcID, unsigned int RecvProcID,
          void * data_ptr, unsigned int size, unsigned int SourceID, unsigned int DestID) {
                unsigned int Mode;
                unsigned int Address;

                Address = FindRoute(MyProcID, RecvProcID, UBC_SEND, SourceID, DestID);
                Mode = Synchronize_2flags (MyProcID, RecvProcID);
                SendData (MyProcID, SourceID, DestID, data_ptr, size, Mode, Address, SYNCH_2FLAGS);
                return;
        } // end of send method

        void recv_2flags (unsigned int MyProcID, unsigned int SendProcID,
          void * data_ptr, unsigned int size, unsigned int *SourceID, unsigned int *DestID) {
                unsigned int Mode;
                unsigned int Address;

                Address = FindRoute(MyProcID, SendProcID, UBC_RECV, *SourceID, *DestID);
                Mode = Synchronize_2flags (MyProcID, SendProcID);
                RecvData(MyProcID, SourceID, DestID, data_ptr, size, Mode, Address, SYNCH_2FLAGS);
                return;
        } // end of recv method

/****************************** 1 FLAG  send/recv ***********************************************/

        void send_1flag (unsigned int MyProcID, unsigned int RecvProcID,
          void * data_ptr, unsigned int size, unsigned int SourceID, unsigned int DestID) {
                unsigned int Mode;
                unsigned int Address;

                Address = FindRoute(MyProcID, RecvProcID, UBC_SEND, SourceID, DestID);
#ifdef F_IN_PE1
                Mode = Synchronize_1flag_in_pe1 (MyProcID, RecvProcID);
#else // F_IN_PE1
                Mode = Synchronize_1flag_in_pe2 (MyProcID, RecvProcID);
#endif // F_IN_PE1
                SendData (MyProcID, SourceID, DestID, data_ptr, size, Mode, Address, SYNCH_1FLAG);
                return;
```

```
		} // end of send method

		void recv_1flag (unsigned int MyProcID, unsigned int SendProcID,
				void * data_ptr, unsigned int size, unsigned int *SourceID, unsigned int *DestID) {
				unsigned int Mode;
				unsigned int Address;

				Address = FindRoute(MyProcID, SendProcID, UBC_RECV, *SourceID, *DestID);
#ifdef F_IN_PE1
				Mode = Synchronize_1flag_in_pe1 (MyProcID, SendProcID);
#else // F_IN_PE1
				Mode = Synchronize_1flag_in_pe2 (MyProcID, SendProcID);
#endif // F_IN_PE1
				RecvData(MyProcID, SourceID, DestID, data_ptr, size, Mode, Address, SYNCH_1FLAG);
				return;
		} // end of recv method

/***************************** 1 FLAG POLL send/recv ***********************************************/

		void send_1flag_polling (unsigned int MyProcID, unsigned int RecvProcID,
		  void * data_ptr, unsigned int size, unsigned int SourceID, unsigned int DestID) {
				unsigned int Mode;
				unsigned int Address;

				Address = FindRoute(MyProcID, RecvProcID, UBC_SEND, SourceID, DestID);
				Mode = Synchronize_1flag_polling (MyProcID, RecvProcID);
				SendData(MyProcID, SourceID, DestID, data_ptr, size, Mode, Address, SYNCH_1FLAG_POLL);
				return;
		} // end of send method

		void recv_1flag_polling (unsigned int MyProcID, unsigned int SendProcID,
		  void * data_ptr, unsigned int size, unsigned int *SourceID, unsigned int *DestID) {
				unsigned int Mode;
				unsigned int Address;

				Address = FindRoute(MyProcID, SendProcID, UBC_RECV, *SourceID, *DestID);
				Mode = Synchronize_1flag_polling (MyProcID, SendProcID);
				RecvData(MyProcID, SourceID, DestID, data_ptr, size, Mode, Address, SYNCH_1FLAG_POLL);
				return;
		} // end of recv method

/***************************** 1 FLAG INTRPT send/recv ***********************************************/

void send_1flag_interrupt (unsigned int MyProcID, unsigned int RecvProcID,
		  void * data_ptr, unsigned int size, unsigned int SourceID, unsigned int DestID) {
				unsigned int Mode;
				unsigned int Address;

				Address = FindRoute(MyProcID, RecvProcID, UBC_SEND, SourceID, DestID);
				Mode = Synchronize_1flag_interrupt (MyProcID, RecvProcID);
				SendData (MyProcID, SourceID, DestID, data_ptr, size, Mode, Address, SYNCH_1FLAG_INTRPT);
				return;
		} // end of send method

		void recv_1flag_interrupt (unsigned int MyProcID, unsigned int SendProcID,
		  void * data_ptr, unsigned int size, unsigned int *SourceID, unsigned int *DestID) {
				unsigned int Mode;
				unsigned int Address;

				 Address = FindRoute(MyProcID, SendProcID, UBC_RECV, *SourceID, *DestID);
				Mode = Synchronize_1flag_interrupt (MyProcID, SendProcID);
				RecvData (MyProcID, SourceID, DestID, data_ptr, size, Mode, Address, SYNCH_1FLAG_INTRPT);
				return;
		} // end of recv method

/***************************** 1 FLAG SM send/recv ***********************************************/

void send_1flag_sm (unsigned int MyProcID, unsigned int RecvProcID,
	void * data_ptr, unsigned int size, unsigned int SourceID, unsigned int DestID) {
				unsigned int Mode;
```

```
                    unsigned int Address;

                    Address = FindRoute(MyProcID, RecvProcID, UBC_SEND, SourceID, DestID);
                    Mode = Synchronize_1flag_sm (MyProcID, RecvProcID, UBC_SEND);
                    SendData(MyProcID, SourceID, DestID, data_ptr, size, Mode, Address, SYNCH_1FLAG_SM);
                    return;
            } // end of send method

        void recv_1flag_sm (unsigned int MyProcID, unsigned int SendProcID,
            void * data_ptr, unsigned int size, unsigned int *SourceID, unsigned int *DestID) {
                    unsigned int Mode;
                    unsigned int Address;

                    Address = FindRoute(MyProcID, SendProcID, UBC_RECV, *SourceID, *DestID);
                    Mode = Synchronize_1flag_sm (MyProcID, SendProcID, UBC_RECV);
                    RecvData(MyProcID, SourceID, DestID, data_ptr, size, Mode, Address, SYNCH_1FLAG_SM);
                    return;
        } // end of recv method

/*****************************************************************************************************/

            void MemoryAccess (unsigned int MEM_LOW, unsigned int MEM_HIGH, unsigned char *local_mem) {
                    while (1) {  // memory is always servicing
                            while (BusAddress < MEM_LOW || BusAddress > MEM_HIGH) { // bus address is not in my range
                                    wait (AddrSet);                                  // every time some address is set
                            }
                            if (RdWr == UBC_READ) {                                  // I am addressed for a read
                                    DataPtr = local_mem + (BusAddress - MEM_LOW);    // base + offset
                                    wait (TRANSACTION1_DELAY_OPB, SC_NS);
                                    wait (TRANSACTION2_DELAY_OPB*DataSize-1, SC_NS);
                            } else if (RdWr == UBC_WRITE){                           // I am addressed for a write
                                    memcpy (local_mem + (BusAddress - MEM_LOW),DataPtr, DataSize);
                                    wait (TRANSACTION1_DELAY_OPB, SC_NS);
                                    wait (TRANSACTION2_DELAY_OPB*DataSize-1, SC_NS);
                            }
                    } // elihw (1)
            } // end of MemoryAccess method
};

// The channel interface
class i_channel : virtual public sc_interface{
 public:
  virtual void send(const void*, unsigned long) = 0;
  virtual void recv(void*, unsigned long) = 0;
  virtual void read(const void*, unsigned long) = 0;
  virtual void write(void*, unsigned long) = 0;
};
```