Estimation of Communication in SystemC Transaction Level Models

Lochi Yu, Samar Abdi, Daniel Gajski

Technical Report CECS-09-05
May, 2009

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

lochi.yu@uci.edu, sabdi@uci.edu, gajski@uci.edu

Abstract

This report describes the modeling style for estimation of communication in busses in Transaction Level Models. We present the general structure of our functional bus model, followed by a survey of the most important features present in current bus protocols. We next show how the three basic components of a transaction: arbitration, synchronization, and data transfer phases are modeled and estimated, and how our tool estimates the timing for each one of them. Our experimental results show that a mp3 decoder platform estimation can be improved if the communication estimation tool is used alongside the computation estimation.

Contents

List of Figures

Listings

Estimation of Communication in SystemC Transaction Level Models

L. Yu, S. Abdi, D.Gajski
Center for Embedded Computer Systems
University of California, Irvine

May, 2009

1 Introduction

This technical report presents the bus modeling style used for the Embedded Systems Environment (ESE) for communication estimation for Transaction Level Models (TLM). TLMs in SystemC has emerged as a new paradigm for system modeling, due to the rise of complexity, size and heterogeneity of current embedded systems.

Since the level of abstraction has risen above RTL, simulation speeds have decreased, but at the expense of decreasing its accuracy. For maximum accuracy, Bus Functional Models are commonly used, but its detailed signal information may not be needed at the Transaction Level and its simulation speed may be too slow.

Communication behavior is commonly unpredictable due to the dynamic bus requests of the processes running inside Processing Elements. Other factors also increase this unpredictability, such as bus contention, preemption by other masters, split transactions by slow slaves, and so on. All these reasons make the simulation-based approach for estimation inevitable.

In order to have an estimation which runs rapidly and produces accurate estimation, an optimal bus model should be defined. Starting from the Universal Bus Channel [1] used in our TLM, we propose a modeling style to accurately produce timing information depending on a chosen protocol, such as ARM's Advanced High Performance Bus, IBM's On-Chip Peripheral Bus or any other bus protocol.

Related work Estimation of communication in Embedded Systems has been tackled with several different approaches. In [10] the authors statically calculate the maximum length of time it takes for a given transaction to take place. This approach suffers the drawback that they only consider the timing after bus arbitration has been done, and if a system has few busses and several Processing Elements running a communication-intesive task (as multimedia applications), the estimated communication time would not be accurate enough to be valuable for the system architect.

In [7], communication estimation is made considering arbitration time, but memory trace data must be obtained first, through either instruction set simulators or hardware design language simulators. Afterwards, trace driven simulation is performed with a candidate communication architecture.

Other approaches are based on libraries, such as [2], in which a selection or communication protocols is made. These libraries must be constructed beforehand so that the simulation can take place.

Finally, [11] calculates timing delays using the worst-case execution time for the whole system. This may prove useful for making decisions based on worst-case scenarios, and obviously do not contribute to do design exploration and achieve the optimal case.

Our approach does not suffer from these drawbacks, since no separate simulation is needed and arbitration and other bus-related activities are taken into account while quickly producing accurate communication information. We do need however, to construct a bus delay database beforehand.

Our communication estimation does depend on the estimation done in the computation components, since the bus congestion at any given time depends on the assumption that some processes have finished computation and entered in a communication phase. We will not discuss estimation at the computation level in this report.

This report will describe in Section 2 the Transaction Level Models of our bus. In Section 3 we will describe how we model our TLM bus in order to simulate the most common bus protocol features. Finally, Section 4 presents Experimental Results and Section 5 presents the Conclusions and future work.

2 Transaction Level Modeling Style

Our model abstracts the system bus as a single unit of communication. It provides the basic communication services of synchronization, arbitration and data transfer that are part of a transaction. At the transaction level, we are do not distinguish between different bus protocols. The bus is modeled as a *sc_channel*, implementing a *sc_interface* which provides 5 public bus communication functions:

1. *Send/Recv* for synchronized communication between 2 different processes.
2. *Read/Write* for processes accessing an external memory.
3. *MemoryAccess* for exposing memory space to the bus.

There are also 2 private functions, used by the above functions:

1. *ArbiterRequest/ArbiterRelease* for mutual exclusion.
2. *Synchronize* for synchronization of processes.

In the present model, UBCs can only be connected to Processing Elements and transducers.

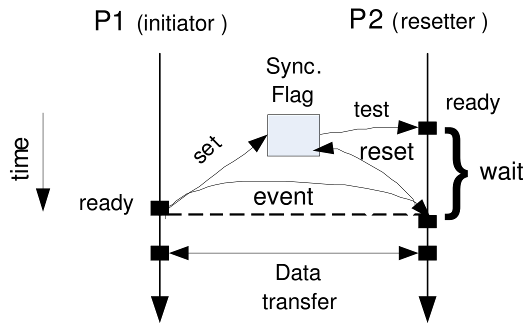

Figure 1: Flag-based synchronization between processes

2.1 Synchronization

Synchronization is required for two processes to exchange data reliably. A sender process must wait until the receiver process is ready, and vice versa. A Synchronization Table in the UBC keeps the flags and events (indexed by process ids) that are used by a process to notify its transaction partner process that it is ready. Synchronization between two processes takes place by one process setting the flag and the other process checking and resetting the flag. Once the flag has been reset, the transacting processes are said to be synchronized. We will refer to the process setting the flag as the *initiator* and the process resetting the flag as *resetter*. The initiator and resetter processes for a given transaction are determined at compile time. In Figure 1 , assume P1 is the initiator process and P2 as the resetter process. Hence, P1 sets the synchronization flag. If P2 is ready before P1, it must keep reading the flag until P1 sets it. P1 notifies this event when it sets the synchronization flag. Once P2 reads the flag as set, it recognizes that P1 is ready and resets the flag.

2.1.1 Implementation

The UBC model will have one flag and one *sc_event* for each pair of communicating processes. The synchronization by the two processes using Send/Recv functions is achieved by both calling the *Synchronize* function, which does one of two things, depending if the calling process is the initiator or the resetter (see Listing 1).

2.2 Arbitration

After synchronization, the resetter process will attempt to reserve the bus for data transfer. This is necessary since the bus is a shared resource and multiple transactions attempted at the same time must be ordered sequentially. The resetter process will *request* an arbitration to the bus, and if the UBC model is exclusive for functional verification, the arbiter is modeled as a mutex (which is a

Listing 1: Synchorize function

```
unsigned int Synchronize(unsigned int MyID, unsigned int PartnerID,
   unsigned int MyMode) {
      if (MyMode==UBC_INITIATOR && MyID==P_ID_Tx2
            && PartnerID==P_ID_Intra) {
         sync_Tx2_Intra =1;
         ev_sync_Tx2_Intra.notify();
         return UBC_INITIATOR;
      }
      if (MyMode==UBC_RESETTER && PartnerID==P_ID_Tx2
            && MyID==P_ID_Intra) {
         while(sync_Tx2_Intra != 1){
            wait(ev_sync_Tx2_Intra);
         }
         sync_Tx2_Intra=0;
         return UBC_RESETTER;
       }
       ...
```

sc_mutex in SystemC. An arbitration request corresponds to a mutex lock operation and once the transaction is complete, the process will *release* the arbitration with a mutex unlock operation.

2.3 Addressing and data transfer

In order to do addressing and data transfer, the UBC uses the following variables and events:

1. Variable *BusAddress* that stores the starting address of the active transaction;
2. Event *AddrSet* that is notified when *TxAddress* is set (it is implemented as a *sc_event*);
3. Variable *DataPtr* that keeps the pointer to the transacted data;
4. Variable *DataSize* that keeps the size in bytes of the transacted data;
5. Variable *RdWr* that identifies if a transaction is read or write (for Read/Write functions).

For synchronized communication, the *resetter* process sets *BusAddress* to the appropriate value from the bus address table. This is done by checking the process IDs and assigning the corresponding bus address (see Listing 2).

For memory transactions, the reader or write process sets *BusAddress*. This is followed by the notification of event *AddrSet* that wakes up the other process or memory controller that is snooping the address bus (see Listing 3). In case of memory transaction, the memory controller reads the address *BusAddress* to check if the address falls in its range and computes the offset. If it is a *read*

Listing 2: Bus Addressing

```
if(MyProcID==P_ID_Intra && SendProcID==P_ID_Tx2)
        BusAddress=ADDR_DH_Tx2_Intra;
else if(MyProcID==P_ID_Trans && SendProcID==P_ID_Tx2)
        BusAddress=ADDR_DH_Tx2_Trans;
```

Listing 3: Waiting for Bus Address

```
if(MyProcID==P_ID_Tx2 && SendProcID==P_ID_Intra){
        while(BusAddress!=ADDR_DH_Intra_Tx2){
                wait(AddrSet);
        }
}
```

it sets *DataPtr* to the right address in the local memory according to computed offset, and if it's a *write*, it will proceed with the memory copy (see Listing 4).

3 Estimation of Communication

3.1 Types of Communication

We can classify communication among components into two types: implicit and explicit communication. We will address both communication types, but mainly the explicit one.

Explicit communication We call *explicit* all the communication that is produced by the user-defined communication channels, such as the application reading data from a memory or sending data to a transducer. This kind of communication can be directly controlled by the user (data size, sequence, frequency) in his/her application code.

Implicit communication Consists of the embedded processor fetching instruction from the instruction memory (when there is a Instruction Cache miss) and of reading and writting program data to the data memory. The user can only indirectly control this traffic by changes on the platform: changing the sizes of the instruction and data caches, or connecting the instruction bus port of the processor to a separate bus.

3.2 Timing Analysis

Estimation can be defined as the calculated approximation of a result which is usable even if input data may be incomplete or uncertain. To do estimation, there are basically two approaches: static

Listing 4: Memory Controller

```
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) {
         wait (AddrSet);        // every time some address is set
      }
      if (RdWr == UBC_READ) {   // I am addressed for read operation
         DataPtr = local_mem + (BusAddress - MEM_LOW);  // base + offset
         wait (SETUP_DELAY, SC_NS); // only for simulation
         wait (HOLD_DELAY+1, SC_NS); // only for simulation
      }
      else if (RdWr == UBC_WRITE){      // I am addressed for write operation
         memcpy (local_mem + (BusAddress - MEM_LOW),DataPtr, DataSize);
         wait (HOLD_DELAY+1, SC_NS); // only for simulation
      }
   } // elihw (1)
} // end of MemoryAccess method
```

and dynamic analyses [3].

Static Timing Analysis (STA) This method computes the expected timing without requiring simulation. This analysis may use abstract models to take into account all possible behaviors that may arise over time. The obvious advantage is speed: not having to run a lengthy simulation speeds up the design process, but the disadvantage and challege is to develop a good abstraction function, since the model must be sound no matter what inputs or what environment the model is run. Due to the complexity of an embedded system, doing a STA on the entire model is not feasible.

Dynamic Timing Analysis (DTA) Operates by executing a program and observing the executions. Testing and profiling are standard dynamic analyses. The advantage is that it can be very precise, showing the actual execution time. The disadvantage is that its results may not generalize to future executions. There is no guarantee that the test suite over which the model was run is characteristic of all possible program executions. Therefore, the challenge is choosing a representative set of test cases. In our case, embedded systems by definition is a system tailored to a specific set of inputs and a specific set of applications. This eases on the disadvantages of using DTA.

3.3 Timing annotation and estimation

In order to have timing information in our model, we need to add it to the code explicitly. This is known as *Timing Annotation*. Formally, timing annotation updates a code with additional information regarding time. Meanwhile, back-annotation updates a more abstract design (high level descriptions) with information from later design stages (low level descriptions) to correct for inaccuracies.

Annotation is done to the code so that when the model runs, the added information will produce the correct delay for a specified portion of the code. This calculation of the delay can be done in two ways: on-line and off-line.

Off-line delay estimation : the delay is precalculated (in our case, generation phase) from values in the database. The code would look like this: `wait(20ns)`.

On-line delay estimation : the delay is calculated using simulation variables at runtime and generated values. The generated values can come from the database and the simulation variables can be packet size, for instance. The resulting code can be something like this: `wait(datasize*10ns)`.

3.4 Bus Features

The model described in the previous section achieves complete functionality, and the next step would be to use this bus model and add accurate timing information to reflect a bus with a specific protocol. Bus protocols differ in many different features. For simplicity, we have grouped them into four areas:

1. Physical properties
2. Arbitration
3. Control
4. Data transfer

As a reference, we present in the appendix a comparison of several bus protocols, with the features outlined below.

3.4.1 Physical properties

The physical properties that characterize a bus include *Address size* and *Bus width*. Another feature is the maximum *number of Master PEs* that it allows. In some protocols like OPB[6], it is determined by the implementation, either limited by the maximum capacitance of the bus (I2C[9]) or by what the power supports [8], while in others such as AMBA[4] the maximum number is fixed.

3.4.2 Arbitration

Arbitration is needed when one or more masters request control of the bus. Each master needs to wait for the arbiter to grant control before proceeding, since only one master can be on the bus at any time. After certain number of cycles, the arbiter grants the bus to a chosen master (depends on the arbitration policy) by asserting a control signal. Once the selected master detects this, it acknowledges by asserting a bus control signal and proceeds to take control of the bus. Common bus arbitration policies used in busses are:

1. Fixed priority
2. Dynamic priority
3. First-come-first-served
4. Round Robin
5. Least Recently used

Bus Parking This feature is also called *Default master*. In this case, the bus is granted by default to one master. In case no other master requests the bus, the bus will remain in control of this default one. This feature is useful for masters which continuously request the bus, since they save cycles by not having to signal the arbiter constantly.

Arbitration pipelining The arbitration occurs before the master starts its control phase, since no master is aware of who controls the bus until the arbiter asserts the corresponding lines. Arbitration pipelining can occur by starting the arbitration process *before* the last cycle of the current data transfer. That way, the next master can assert its control lines and take ownership of the bus immediately after the previous master finishes its data transfer.

3.4.3 Control

Once the master PE has the ownership of the bus, several events may occur before or during the data transfer phase:

Timeout If the slave does not answer the request of the master, after a specific amount of time, the master will do a *timeout*, release the bus and abort the transaction.

Wait The slave might need some additional cycles in order to be ready when the Master addresses it, so it may signal it to wait for a determined amount of time. In the case of a Wait signal, the Master will disable the timeout counter in response but will keep ownership of the bus until the data transfer is done.

Retry In case the slave is not ready for the data transfer, it may issue a Retry signal which will ask the master to give up the control of the bus and re-ask for it after a certain number of cycles. The data transfer with the slave will resume in an indetermined number of cycles, since the master must request arbitration to take ownership of the bus, and this may vary, depending on the number of masters also requesting the bus.

Split Similar to last case, the slave may be busy at the time the master is requesting communication. The slave can also issue a Split signal, which will ask the arbiter to re-take the ownership of the bus from the master and ignore its subsequent requests until further notice from the slave. This allows other masters to take the bus and continue their transactions, while the slave concludes its current task. Once it has finished it, the slave will signal the arbiter to re-allow requests from the original master. After that, normal transactions can continue.

Preemption In the case where the arbitration policy is based on priority, a master making a transaction can be preempted by a higher-priority master. Once this master has concluded its transaction, control of the bus is returned to the preempted master. In some protocols, a master has control of a bus signal which can allow or disallow preemption by another PE.

3.4.4 Data Transfer

Typically, data transfer phases include an address phase, where the master controls the address lines of the bus, followed by a data phase, where the slave either reads from or writes to the bus. This is repeated for each individual address the master wishes to access.

There are two features that optimize the data transfer speed:

Burst mode and Pipelining If the data transfer involves more than one address and the addresses are sequential, the *burst mode* will allow the data reads or writes to happen in every cycle, by *pipelining* the transfers, overlapping the control or address phase with the data phase.

3.5 Bus Modeling

The most of the features described in the previous section can be easily added to our model, while others cannot. The features that were added are arbitration, data transfer. These will add more timing information about which master is transfering data at any given time, and how much time it took to transfer that data.

Control features take charge mainly for error conditions on either the slave (needs more time or takes too long) or a higher priority master preempting a low priority one. These error conditions simulate in a lower abstraction level, hence we will not simulate them with our model.

3.5.1 Bus delay database

Our estimation setup consists of dynamic timing analysis (simulation based) with online and offline delay computation. We need a delay database because the delay for a single-word transaction is affected by the following factors:

1. Type of memory (SRAM, DRAM)
2. Location of memory (internal, external)
3. Type of memory operation (read,write)

Our overall setup for timed TLMs is shown in Figure 2. As seen in this figure, our TLM Generator uses bus properties values from a bus database in order to compute the delays.

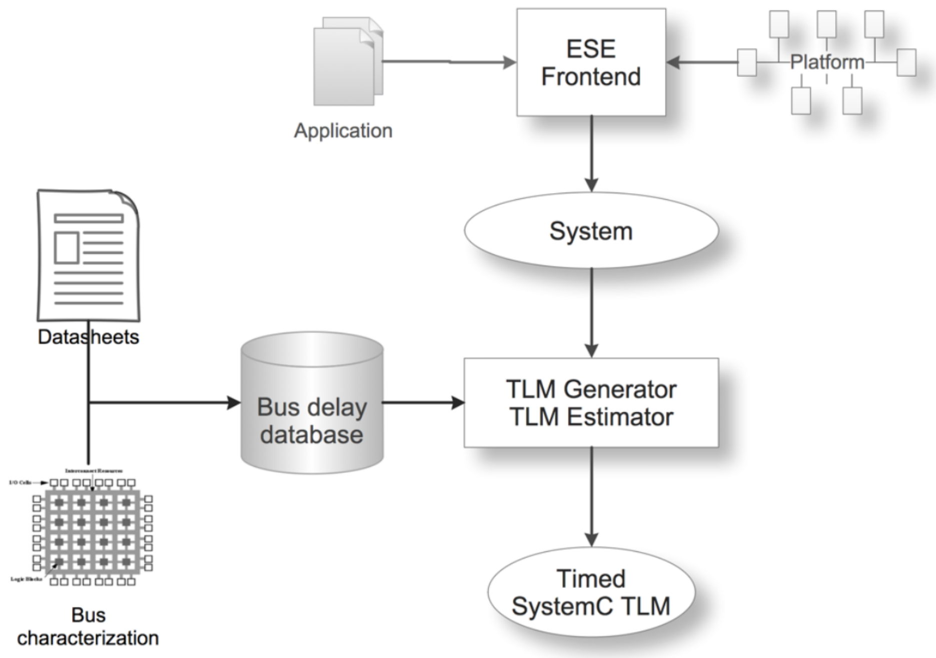

Figure 2: Bus delay database setup

This database consists of two parts:

1. **Bus properties** For each available protocol, all implementable features are listed, including bus physical properties (bus data width, address space), arbitration options, data transfer options. These features are obtained from the datasheets.

2. **Bus delay values** This part of the database lists delay values in multiple test setups. The setups differ in: bus protocol, memory written or read, type of memory accessed, and location of memory (local or external). All setups followed these conditions: read/write one single word to a memory, with no instruction traffic on the bus and only one master connected to the bus. The goal was to record the absolute delay for a transaction, without the interference of instruction or program data fetches, and arbitration delays. These two effects will be discussed below.

3.5.2 Transaction delay modeling

Our aim is to obtain an accurate *transaction delay* for any communication between two processes or between a process and a memory. Each individual transaction is composed of 3 parts:

1. Arbitration delay: t_{arb}
2. Synchronization delay: t_{sync}
3. Data transfer delay: t_{dt}

The total transaction delay is:

$$T_{total} = t_{arb} + t_{sync} + t_{dt} \tag{1}$$

Although we can model the three components of the transaction, we cannot estimate arbitration and synchronization; we can only measure it during simulation. In the data transfer phase, estimation can be done depending on the bus features present.

There are some delay components in arbitration and synchronization which we can estimate, such as time to set/reset a flag, or to raise the signal granting the ownership of the bus. These delays will be produced from the database, offline, during the generation phase. The actual time of the synchronization or arbitration phase will be measured during simulation. The data transfer phase's delay estimation will be calculated online.

3.5.3 Arbitration modeling

Our model supports the following features: Bus parking, arbitration pipelining and these arbitration policies: Round Robin (RR), Priority and First-Come-First-Served (FCFS). As a reminder, our functional bus model used a mutual exclusion (*sc_mutex*) policy.

Implementation There are two ways of implementing an arbiter, either by using a separate module or thread to exclusively arbitrate each bus, or use each process requesting the bus to do its part of the arbitration task. We chose the latter to improve simulation speed.

Listing 5: Arbiter Request delay

```
1 void ArbiterRequest (unsigned int ProcID) {
2   if (!(bus_parking && parked_pe.contain(ProcID))){
3     wait(ARB_REQ_DELAY_OPB,SC_NS);
4   }
5   arb_lock(ProcID);
6   return;
7 }
```

When a master asks for control of the bus, a call to *ArbiterRequest* is made, and the time it takes to do the arbitration is logged at once, unless the *bus parking* feature is active (see Listing 5). Before the delay is taken into account, two things must be checked: first if the bus parking feature is activated and second if the process asking for the bus resides in the PE in which the bus is parked (line 2). The arbiter request delay (line 3) is obtained from the database entry corresponding to that bus protocol.

Arbitration policies The implementation for each policy differs but they follow a common procedure: the process lists itself into a queue, and waits for its signal to proceed. The process checks for the top process in the queue to be its process ID, if different, will wait until this top process changes, via an event notification. The selection which is the next process to run is the actual arbitration policy: a simple queue for FCFS, a priority queue or by taking turns in a RR.

The steps for each policy are shown in figures 3, 4 and 5.

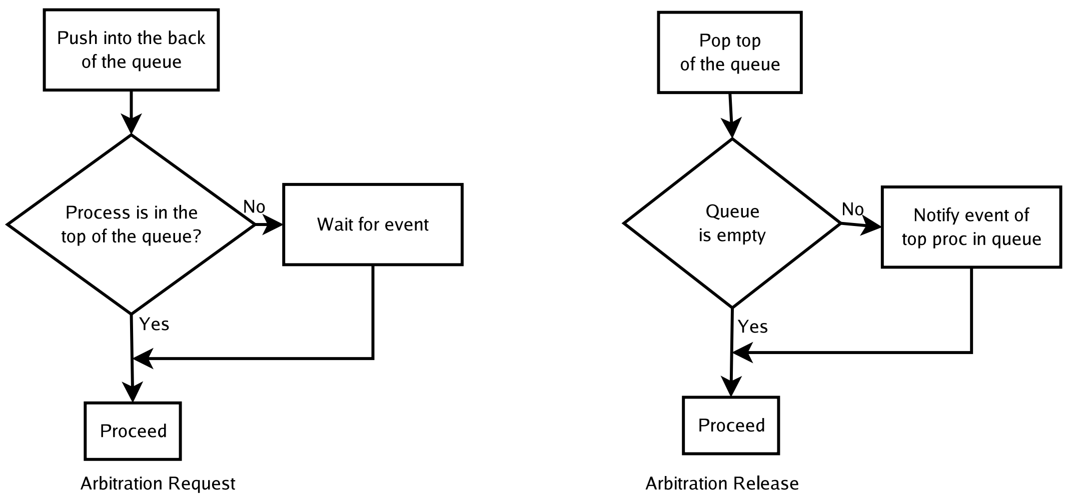

Figure 3: FCFS Arbitration

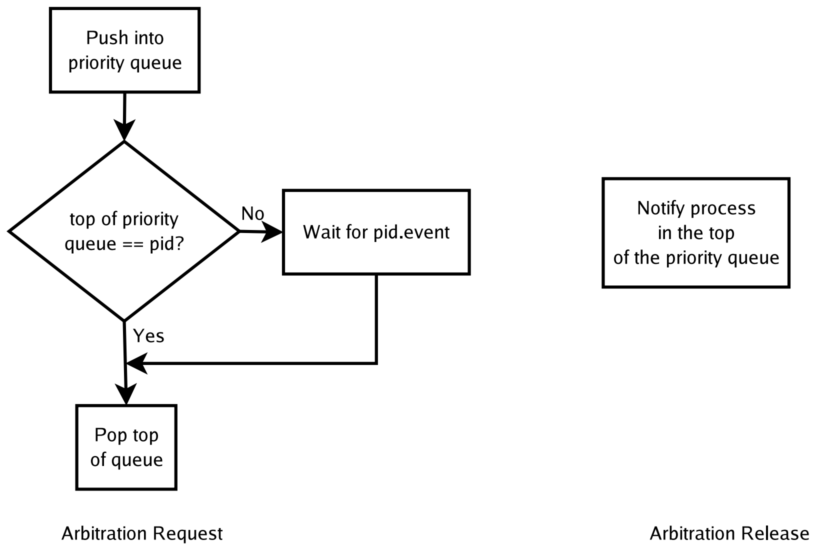

Figure 4: Fixed Priority Arbitration

The implementation of the steps is shown in Listing 6.

In the code, the process lists itself into the queue in lines 4, 10 and 19, and checks if it can continue at once (lines 5, 11 and 21). If not, it will wait for the corresponding event before it starts its transaction (lines 6, 12 and 22).

Once the process has finished its transaction, a call to *ArbiterRelease* is made, which will call *arb_unlock* (see Listing 7).
The process will remove itself from the queue if it has not done so (line 4) and notify the remaining processes in the queue (lines 6, 10 and 29). That way, there is no need for an arbiter thread to be present, since the processes are managing the queues and doing the notification themselves.

3.5.4 Data transfer modeling

For the explicit communication case, for each transaction, the model has the size information of the data to be written or read. Based on the burst or pipelining features present and fetching the appropiate data from the bus delay database, the overall timing of the transaction can be determined.

We start by defining these values:

1. Total data transfer time in cycles: T_{dt}
2. Size of the transaction in bytes: S
3. Bus width of the bus in bits: B_{width}

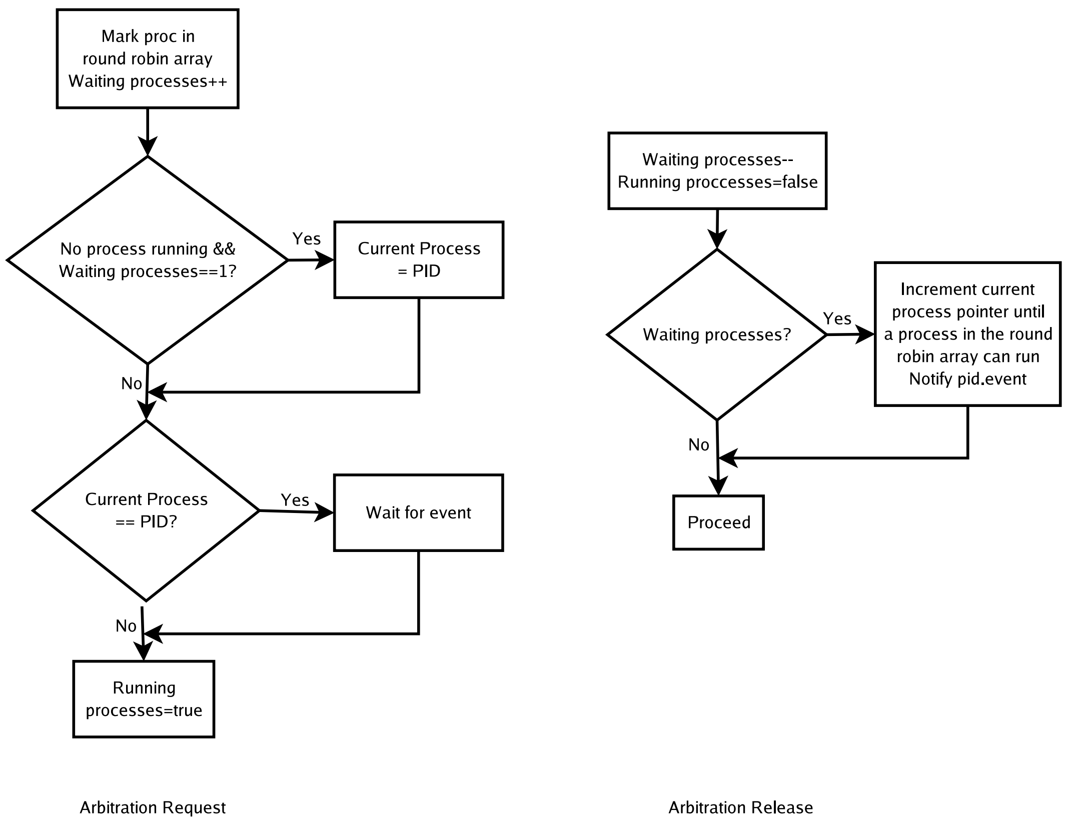
Mark proc in round robin array Waiting processes++
No process running && Waiting processes==1?
Yes
Current Process = PID
No
Current Process == PID?
Yes
Wait for event
No
Running processes=true
Arbitration Request
Waiting processes-- Running proccesses=false
Waiting processes?
Yes
Increment current process pointer until a process in the round robin array can run Notify pid.event
No
Proceed
Arbitration Release

Figure 5: Round Robin Arbitration

Listing 6: Arbitration policies implementation

```
void arb_lock(unsigned int ProcID){
  if(arb_policy==1){
    //FCFS
    queue.push_back(ProcID);
    if(queue.front()!=ProcID){
      wait(procev[ProcID]);
    }
  }elseif(arb_policy==2){
    //Priority
    prio_queue.push(priorities[ProcID]);
    if(running==true||prio_queue.top().getid()!=ProcID){
      wait(procev[ProcID]);
    }
    running=true;
    prio_queue.pop();
  }elseif(arb_policy==3){
    //RoundRobin
    waiting++;
    rr[ProcID]=true;
    if(!running&&waiting==1)current=ProcID;
    if(current!=ProcID){
      wait(procev[ProcID]);
    }
    running=true;
  }
}
```

4. Control/Address phase length in cycles: CA
5. Bus transaction delay in cycles: D_{bus}

The data transfer size changes depending on the transaction (on-line estimation), while the bus width and the control/address phase length are values that are taken from the bus protocol's datasheet. The delay value is fetched from the database during generation phase and inserted as a constant.

In case of a normal transaction, the total time would be:

$$T_{dt} = CA \cdot S + \lceil \frac{S \cdot 8}{B_{width}} \rceil \cdot D_{bus} \tag{2}$$

If the bus supports burst mode, the time would be:

$$T_{dt} = CA + \lceil \frac{S \cdot 8}{B_{width}} \rceil \cdot D_{bus} \tag{3}$$

Listing 7: Arbitration Release

```
void arb_unlock(unsigned int ProcID){
  if(arb_policy==1){
    //FCFS
    queue.pop_front();
    if (queue.size()!=0)
      procev[queue.front()].notify();
  }elseif(arb_policy==2){
    //Priority
    running=false;
    procev[prio_queue.top().getid()].notify();
  }elseif(arb_policy==3){
    //RoundRobin
    waiting--;
    running=false;
    //increment current until someone can run
    if(waiting!=0){
      current++;
      if(current>=MAX_P)
        current=0;
      int next=255;
      for(inti=0;i<MAX_P;i++){
        if (rr[i]==true && i<next && i>current-1)
          next=i;
      }
      current=next;
      if(current==255)
        current=0;
    }
    procev[current].notify();
  }
}
```

In both cases, equations 2 and 3 use the ceiling function because depending on the width of the bus, several bytes are written or read simultaneously, but even if the number of bytes to be transferred is less than the total capacity of the bus, one cycle would be needed.

Implementation The implementation of the timed bus functions involve the addition of the wait statements, with the calculated delay for the transaction. The rest of the function remains the same as the functional model.
Equations 2 and 3 are implemented as the *wait* statements in lines 13 and 11 of Listing 8, respectively. The read, send and receive functions are implemented in a similar way.

Listing 8: Write fuction

```
void write(unsigned int MyProcID, unsigned int addr,
    void * data_ptr , unsigned int size){
  ArbiterRequest(MyProcID);
  wait(CA,SC_NS);
  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 the bus is valid
  if (burst){
    wait(ceil((size*8)/BUS_WIDTH)*DELAY,SC_NS);
  }else{
    wait(CA*size+ceil((size*8)/BUS_WIDTH)*DELAY,SC_NS);
  }
  BusAddress = ADDR_NONE;
  ArbiterRelease(MyProcID);
  return;
}
```

3.5.5 Instruction and Program Data communication

Previously, we showed how we can model *explicit* communication on the bus. Nevertheless, for *implicit* communication, the case is more complicated if the instruction and data memories are indeed connected to the same bus as the processor's data bus port.

Instruction and data fetches occur during the program execution and during regular explicit bus communications. From a bus' point of view, the bus model has no way of determining the pattern of this instruction/data flow, since it acts as a slave for the processor. The only way to model this communication is from the processor side. Our timed TLM generator [5] estimates not only the probable delay for computation, but also has a cache model that dictates when there is a cache miss. It is on this cache miss that a new cache line (in the instruction cache or data cache) must be obtained by using the bus.

implementation We will group all implicit communication delay times, and add them to the next *explicit* transaction on that bus. Each time the cache model decides to have a cache line fetch, it will update a delay counter. At the next explicit transaction, during the data transfer phase, the UBC will "spend" this extra delay before executing its regular wait delay. This approach obviously sacrifices accuracy for speed, since we would not be producing the expected arbitration delays on the other lower priority processors while the cache line is being fetched. The arbitration effects are anyways only visible and produce an accurate result if the computation estimation is accurate, so in order for the entire model to have high accuracy, we rely on the computation estimation engine entirely. Two

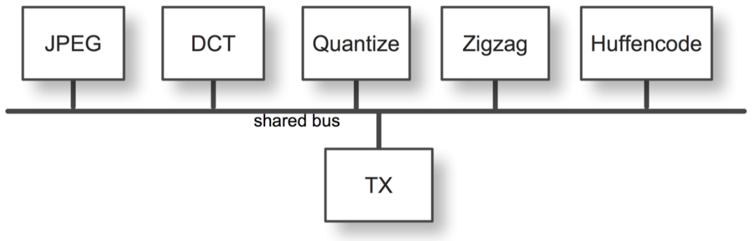

Figure 6: JPEG encoder platform

Table 1: Data Communication measurements in a JPEG encoder TLM

Data traffic	FPGA board (cycles)	TLM estimation (cycles)	% Error
jpeg to dct	171275	172980	1.00
dct to quantize	160242	156800	2.15
quantize to zigzag	321213	304640	5.16
zigzag to huffencode	324130	304640	6.01

aspects must be considered: computation estimation accuracy and cache model accuracy. These models will not be discussed in this report.

4 Experimental Results

To test our bus model, we chose a JPEG encoder application, and mapped it into a platform consisting of 5 PEs, 1 bus and 1 transducer. The platform is shown in Figure 6. All instruction and data memory of each PE is located in local busses, and only explicit inter-PE communication is happening on the shared bus. We used our TLM generator [12] to create a timed model and measured the explicit communication delays between all PEs. To see the accuracy of these estimations, we synthesized the platform to a Xilinx Virtex4 FPGA and did the same measurements between its MicroBlaze processors. The results are shown in Table 4. We can see that the error in the estimation is less than 6% and in the best case, of 1% (between the first and the second PE). A graphical representation of the results is shown in Figure 7.

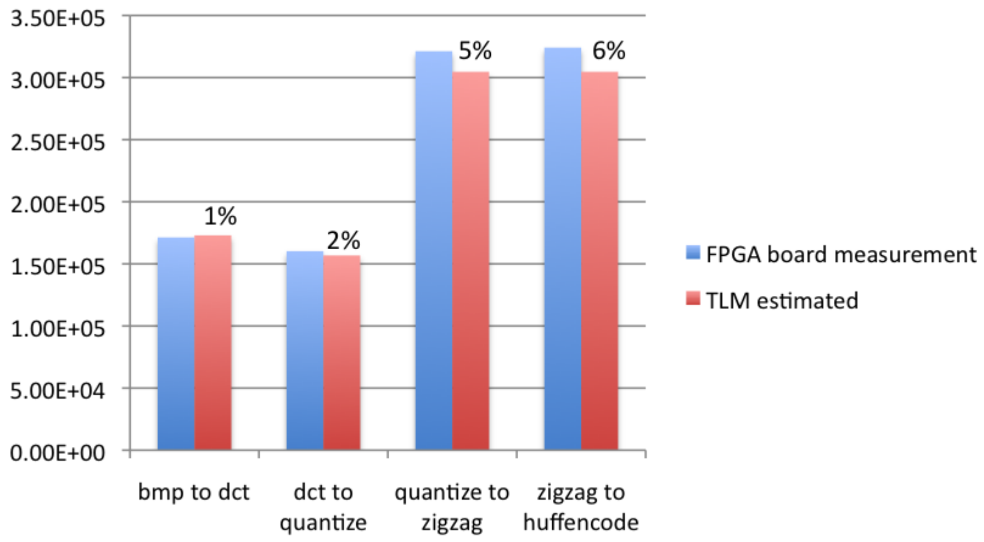

Figure 7: Communication delay between PEs

5 Conclusions

In this report, we presented the structure of the Universal Bus Channel model and its internal functions. Also, we reviewed the most important features that differentiate different bus protocols and selected a subset to be implemented into our UBC model in order to get accurate timing information that would represent an available protocol.

The features that were incorporated into our timed UBC model were: physical properties such as bus width and maximum number of masters, arbitration features such as different arbitration policies, bus parking and arbitration pipelining, and finally data transfer features such as pipelining and burst mode data transfers. These features along with actual data transfer delays in each protocol, are stored in a bus model database, which is accessed during the TLM generation phase. This information is used for the on-line estimation of the data transfer delay.

Our timing information is composed of the measurement of our arbitration model, synchronization functions and the estimation of the data transfer phase. Assuming that our system has an accurate computation estimation, the performance our Transaction Level Model can be easily measured and taken into account for design purposes.

Running our communication estimation tool in a jpeg encoder platform, we saw that the estimated data transfer time is within 94% of the FPGA board measurement. This reflects a good estimation accuracy for our tool.

References

[1] S. Abdi and D. Gajski. Ubc: A universal bus channel for transaction level modeling. Technical Report CECS-06-07, University of California, Irvine, April 2006.

[2] A. Baghdadi, N.-E. Zergainoh, W. O. Cesario, and A. A. Jerraya. Combining a performance estimation methodology with a hardware/software codesign flow supporting multiprocessor systems. *IEEE Transactions on Software Engineering*, 28:822–831, 2002.

[3] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, May 2003.

[4] A. Holdings. *AMBA Bus Specification (rev 2.0)*, 1999.

[5] Y. Hwang, S. Abdi, and D. Gajski. Cycle-approximate retargetable performance estimation at the transaction level. In *Proceedings of the Design, Automation and Test in Europe conference*, March 2008.

[6] IBM. *On chip Peripheral Bus (OPB)*.

[7] S. Kim, C. Im, and S. Ha. Schedule-aware performance estimation of communication architecture for efficient design space exploration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13, May 2005.

[8] PCI-Special Interest Group, www.pcisig.com/home. *PCI*.

[9] Phillips Semiconductor, www.semiconductors.phillips.com/i2c. *I2C*.

[10] P. Voigt Knudsen and J. Madsen. Communication estimation for hardware/software codesign. In *Proceedings of the Sixth International Workshop on Hardware/Software Codesign (CODES/CASHE '98)*, 1998.

[11] T.-Y. Yen and W. Wolf. Communication synthesis for distributed embedded systems. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design*, 1995.

[12] L. L. C. Yu Lo and S. Abdi. Automatic systemc tlm generation for custom communication platforms. In *Proceedings of 25th IEEE International Conference on Computer Design*, October 2007.

A Bus protocols comparison

Table 2: Protocols Comparison chart

Feature	OPB	AMBA-AHB	AMBA-APB	CAN	I2C	PCI
Arbitration policy	Fixed, dynamic, RR, LRU	Depends on application	No masters, all slaves	Priority, Fixed (wired), wired AND	wired AND, No priority	not part of spec
Maximum number of masters	implementation dependent	16 masters	NA	2^{11}	limited by maximum capacitance	electrically supports (5 average)
Burst mode	Bus lock + sequential address	4, 8, 16 beat wrapped and incremental	NA	No	No	Yes
Preemption	Yes	Yes, unless LOCK is asserted	No	No	No	No
Wait states	Yes	Yes, BUSY and HREADY	No	No	-	Yes
Pipelining	Overlapped bus arbitration and data transfer	Overlapped address phase and data phase	No	No	No	Arbitration is overlapped
Split	No	Yes	No	-	-	Yes, ”delayed transactions”
Address size	32 bits	32 bits	32 bits	-	7 bits + 16 reserved addresses = total of 112 devices	64 bits
Bus Parking (default master)	Yes	Yes	NA	NA	NA	Yes, ”arbitration parking”
Retry	Yes, back-off for 1 cycle	Yes, back-off for 2 cycles	No	No	No	Yes