Building a better memory controller: architectural performance exploration of an AXI memory controller

Laurent Isenegger, CoFluent Design - October 15, 2010

Communication between processors and memories is often a major bottleneck, making the design of the memory controller a critical task in determining overall system-level performance. The memory controller, in addition to collecting the requests from the masters and forwarding them to the memory, is tasked with re-ordering these requests in order to optimize specific system characteristics, such as latencies and power consumption. Numerous optimization techniques have been developed over the last few years. Many of these techniques require a specific internal architecture, which must be defined at the very beginning of the project. Early design-space and architecture exploration is critical for memory-controller design.

System description

This system is a memory controller that collects the requests from the masters connected to the AXI (advanced extensible interface) bus and forwards them to an SDRAM-DDR (synchronous dynamic random access memory - double data rate) memory, as shown in Figure 1. The physical architecture is based on hardware processing units for the AXI masters. The controller and communication nodes are configured as buses to represent the AXI channels and DDR interface signals.

The AXI protocol is the third generation of AMBA (advanced microcontroller bus architecture). ARM introduced AXI in 2003, targeting high-performance, high-frequency designs. The protocol includes several features appropriate for high-speed submicron interconnects. AXI protocol is burst-based and relies on five different channels. Read and write transactions have address and control information, respectively, on the ReadAddress and WriteAddress channels. Data is transferred on the WriteData and ReadData channels. An additional channel, WriteAck, is available, enabling the AXI slave to signal when the write transaction is completed. Up to 16 data transfers can be encapsulated into one single AXI transaction, and each data transfer can be sized up to 1024 bits.
The SDRAM-DDR protocol is also burst-based. Utilizing the rising and falling edges of the clock signals, 2, 4, and 8 data transfers can be completed in 1, 2, and 4 clock cycles, respectively. Internally, a memory module is composed of several banks that are themselves structured in rows and columns. To access data stored at a specific location \( \{\text{row, column}\} \), the controller first activates the row in the proper bank. This operation consists of transferring the full content of a row in a row-buffer. Once the row is activated, the controller provides the column address to the memory. For write operation, the data value is received on the DQ signals and written to the memory. For read operations, data read is available on DQ signals after a delay known as CAS (column access strobe) latency.

Only one row per bank can be active at a time. Therefore, when a request is sent to another row in the same bank, it is required to pre-charge (equivalent to deactivate) the current active row before activating the new one. These additional tasks have a negative impact on latencies as well as power consumption.

**Architecture exploration objectives**

The complexity of the memory-controller architecture and the AXI and DDR protocols necessitates making many architectural decisions early in the project. Within these choices, this example focuses especially on four distinct areas:

1. Which strategy to use for sending write acknowledge
2. The impact of multiple row activations based on various arbitration schemes
3. The impact of automatically pre-charging rows and how to monitor power consumption
4. Selecting the optimized architecture based on AXI data bus width, AXI burst length, memory width, and memory burst length

**Behavioral modeling and simulation**

Application model capture, shown in Figure 2, is the first step of the timed-behavioral modeling. This step builds a graphical representation showing the functional behavior of both the system and its environment, including data and control flows. The environment functions are later used as a testbench during the simulation.

The AXI masters functions *Sender*, *ReceiveWACK*, and *ReceiveRData* generate and send read or write requests, receive the write acknowledge, and receive the data read to complete a read request,
respectively. These three functions are encapsulated in a structure declared as a multiple instance (vector of the same structure). The number of instances is defined by a generic parameter, $C_{Nbr}$. Using a generic parameter enables modifying the number of masters communicating with the memory controller during the performance analysis very easily without any additional SystemC re-generation and re-compilation.

The requests are sent on the AXI channels, represented by message queues, and received by the AXI interface of the memory controller. This interface is also declared as a multiple instance in order to model the possibility to use a multiport memory controller. All requests are then forwarded to a single-instance function, CollectRequests. This function stores the pending requests using the shared variable ListsRequestPtr. Once the requests are internally stored, the function CollectRequests sends a trigger to the Arbitration function.

This function reads the shared variable, selects a request depending on its arbitration scheme, and forwards it to DDRCommandGeneration. This last function receives the request selected and, depending on the current status of the memory concerned, generates the proper command in order to activate/precharge/read/write specific locations in the memory following the DDR protocol.

Memories are also declared as multiple instances in order to model the possibility to use multiple memory chips connected to the controller. Inside the multiple instances, BankDmuxer decodes which bank is concerned by the incoming request. Once detected, the command is forwarded to the proper bank through a message queue (FIFO buffer), also declared as multiple instances. The corresponding instance of MemoryCommandExecution processes the command and sends the acknowledge to the controller: data read for a read operation and an acknowledge for a write operation. The acknowledge for write operations is not a part of the DDR specifications, since in real systems, the duration of write operations in memory is assumed by the memory controller. Modeling this assumption by a wait on a write acknowledge generated by the memory has no impact on the system's performance and improves the readability of the model created.

The memory controller receives the information from the memory and forwards it to the proper AXI channel. Additional elements have been introduced in this model to monitor the impact of different system characteristics. After the application capture, SystemC is automatically generated and compiled. For this design, CoFluent Studio was utilized.

**Platform and architectural model**

The first simulations are run after defining the functional behavior of the system with the application model. Behavioral simulation supposes infinite resources, as if one processing unit was available for every function and one bus for each data exchange. Obviously, this is not the case in real system architectures.

The purpose of the platform model is to describe the physical architecture of the system under study. The platform model defines the available resources, as well as additional constraints that must be considered.
The platform model for this design is very simple since the system under study, the AXI memory controller, is a single chip. As shown in Figure 3, the masters are represented with one single hardware processing unit. The behavior of each master is independent from the others, and there is no need for limiting the number of simultaneous active masters. Consequently, the processing unit is configured as hardware.

The memory controller and memory are also modeled as hardware processing units. Communication between the different processing units is carried out by communication nodes configured as buses: one modeling the AXI channels between the masters and the controller, one modeling the SDRAM-DDR signals between the controller and the memory.

Once both the available resources in the platform model and the behavioral model are defined, the next step associates the elements defined in the TBM (timed behavioral model) with the resources in the platform model. This operation is the first step of the architecture modeling.

The architectural modeling operation associates different functions with the processing units. The data flow elements are mapped onto the communication nodes included in the platform model. The five channels of the AXI protocol are mapped on the node AXIbus defined in the platform modeling, and the DQS and DDRCommand message queues are mapped on the node SDRAM interface. Once all the function and relations are mapped, additional attributes and algorithms are included in the model before the automatic SystemC generation. After the generation, the code is compiled and can be run in CoFluent Studio ESL toolset. The simulation environment is used to extract relevant information used to make architectural decisions.

Results analysis

1. Write acknowledge strategy

When a write request is sent to the memory controller, the AXI master expects in return an ACK (acknowledge) message on the corresponding AXI channel. Two strategies can be envisaged for generating these ACK messages. The straightforward approach consists of sending the acknowledgement only after the data has been written to the proper memory. Another solution that decreases the system's latency requires the write acknowledge generation occur immediately after the request has been taken into account by the memory controller. This means that return messages are sent to the AXI masters even before data has been effectively written to the memory. An additional hardware block—not represented in this model—is necessary to ensure that no read operations are performed before data is effectively written in the memory. The additional hardware block is the "price to pay" for the latency reduction.
This new strategy has been taken into account in the application capture. The *FastAcknowledge* is generated by the *Arbitration* function, whereas the normal acknowledge is generated only by the *ResponseForward* function. The selection of the acknowledge strategy is done through the generic parameter named *WriteAckStrategy*, for which values can be *NORMAL* or *FAST*. **Table 1** compares latencies for the two write acknowledge strategies depending on various simulation parameters.

<table>
<thead>
<tr>
<th>Varying parameters</th>
<th>Write latency with normal write acknowledge</th>
<th>Write latency with fast write acknowledge</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{Nbr} = 4$; AXI burst length = 2</td>
<td>67 cycles</td>
<td>35 cycles</td>
<td>48%</td>
</tr>
<tr>
<td>$C_{Nbr} = 8$; AXI burst length = 2</td>
<td>116 cycles</td>
<td>55 cycles</td>
<td>53%</td>
</tr>
<tr>
<td>$C_{Nbr} = 4$; AXI burst length = 4</td>
<td>132 cycles</td>
<td>61 cycles</td>
<td>54%</td>
</tr>
<tr>
<td>$C_{Nbr} = 4$; AXI burst length = 8</td>
<td>924 cycles</td>
<td>794 cycles</td>
<td>14%</td>
</tr>
</tbody>
</table>

Fixed parameters: master delay = 150 cycles, arbitration scheme = first request, 4 banks per memory, auto precharge enabled, AXI transfer size = 256 bits, AXI burst length = 2, effective memory width = 32 bits, burst length memory = 4, default value for other parameters

It is important to focus on relative values more than on absolute values. Absolute values are typically based on estimations of the future implementation and are more subject to change. Relative values provide accurate information for a comparison between different algorithms or strategies. The results shown in **Table 1** indicate that the latency between the emission of a write request and the corresponding returning acknowledgement can be reduced down to 50% by using the new fast acknowledge strategy. However, the new strategy requires an additional hardware block in order to maintain memory data integrity and thus increases the silicon area. In the last line of the table, the latency reduction is only of 14% because the memory controller cannot handle all the incoming requests fast enough. Therefore, the general latency drastically increases up to more than 700 cycles and the benefits of the new strategy are significantly reduced. This enables highlighting in which use cases the new write acknowledge strategy is the most appropriate.

2. Power consumption estimations

Power consumption is a key differentiator for embedded system components and IPs. For optimal results, power consumption must be considered early in the project during the system architecture exploration. Unfortunately, it is very difficult to get accurate and dynamic power consumption estimation with traditional tools, since most tools are based on worst-case strategies. The SystemC toolset used for this design includes power consumption estimation and analysis during the simulation. A profiling table provides accurate average, maximum, and minimum power consumption values.

The automatic precharge option is available in the generic parameters. When enabled, the memory automatically de-activates the current active row after any read or write command. If the next memory access concerns another row in the same bank, then this technique improves the general latency, since the controller does not need to explicitly de-activate the previous row accessed. However, when consecutive accesses concern the same row in the same bank, the controller needs to open that row for each read or write command. This results in higher latencies and power consumption.
The decision on whether or not to activate this capability can be monitored on the power consumption profiles shown in Figure 4. On the left side, the automatic precharge is enabled. Between every write command, a precharge command and a new active command are sent to the memory. On the right side, only one precharge command is issued before multiple read commands.

Figure 4 Power consumption evolution depending on automatic precharge enabling

3. Arbitration schemes and row openings

The memory controller is not only in charge of adapting the AXI protocol to the DDR protocol, it also includes an arbitration algorithm to reduce the number of row openings. New row openings in memory banks have a negative impact on the system's performances in terms of latency and power consumption. When a new row needs to be activated, the controller precharges the current row before activating the new one. When the same row is accessed, the controller directly sends the read or write command. Consequently, the controller's arbitration scheme reduces the number of row-openings by selecting the next request to be sent to the memory within the set of pending requests of the AXI masters.

For this design, the architectural exploration environment compared two different arbitration schemes. FirstRequest forwards the request that was first received to the memory. FirstBurst ensures all the requests of a same AXI burst are processed consecutively by the memory. Due to the AXI masters configuration, the multiple data transfers within an AXI burst usually access the same row in the same bank. Consequently, processing all these data transfers consecutively will reduce the number of row openings.

By selecting the arbitration algorithm in the generic parameters, it is possible to monitor the impact of each one on the general system's performance. With the model created, it has been shown that the FirstBurst algorithm reduces the number of row openings and increases the average bandwidth by 5%.

By building such a model, the system's architect can evaluate the benefits of different arbitration algorithms depending on the selected use cases.

4. AXI data bus width, burst lengths, and memory widths

The SystemC-based toolset also aids in determining the correlation between several system characteristics. For this design, the analysis highlights the interdependencies between the burst length in the DDR protocol, the AXI data bus width, and the effective memory width.

The architecture exploration environment ran several consecutive simulations with different values
of generic parameters. It determined average bandwidths with different DDR burst lengths (BurstLengthMemory varying from 1 to 4) and AXI data bus widths.

When the AXI data transfer size, which is equivalent to the AXI data bus width, is set to 32, the size of the data transfers matches with the effective memory width. In this case, only one data transfer in the DDR protocol is required to process the input request. Using multiple data transfers in the DDR protocol would require additional cycles, but the same amount of data would be exchanged. This causes the average bandwidth to decrease as several cycles are not used for data transfer.

When the AXI data transfer size is set to 64, two data transfers to the memory are required to process the incoming data. Therefore, a burst length of 2 for the DDR transactions is the most efficient. A value of 1 would require generating additional commands from the controller. A value of 4 would use clock cycles that are not necessary, and consequently decrease the bandwidth. Similarly, if the AXI data transfer width is set to 128, then four data transfers are necessary in the DDR protocol to process the incoming data. The most appropriate value for the memory burst length is 4.

Architectural analysis determined the correlation between several system characteristics. System designers use this data to make informed decisions on the system's environment and performance concerns.

**Design results**

The ESL toolset was used for model-based specification, architecture exploration, and performance analysis of electronic systems for an AXI memory-controller example case. First, the functional behavior of the memory controller is described through the CoFluent Studio graphical representation. Simulations are run at this step for validating functionally the designed model. Then, the available resources are defined and the previously specified functions are associated to these resources. After refining the behavior of the system by setting attributes for the different elements and inserting legacy C code, system performance estimations are obtained through simulations for the different explored configurations. The impact of architecture choices and power consumption algorithms are estimated, taking into account application performance parameters such as latencies and average bandwidths. Physical characteristics such as power consumption were also considered.

By providing timed-executable specifications, facilitating communications, and offering a base for early trade-off analysis, the ESL toolset and architectural analysis capabilities contribute to bridging the gap from specification to implementation. The reference system models and associated testbenches can be used and refined throughout the project's lifecycle to capture the electronic system design know-how, serving as true knowledge repository for accelerating new project inception and verification.