Morpheus DDR SDRAM Controller Documentation

Sean Whitty
whitty@ida.ing.tu-bs.de

November 5, 2007

This is the official documentation of the DDR SDRAM controller (CMC, Central Memory Controller) used in the Morpheus project.

1 Architecture

1.1 General Architecture

The DDR-SDRAM controller architecture is shown in Figure 1.

Requests to the SDRAM controller are made by applications via an ST Bus Protocol Type 3 interface, which provides the connection from the Morpheus ST Bus Protocol Type 3 based STNoC Network On Chip to a configurable number of *application read* and *write ports*. In the context of the Morpheus project, the number of ports has been set to 2 application read ports and 2 application write ports. See Section 2.2.2 for a description of the exact number of words read/written per memory access request and Section 3.2 for a detailed description of the application port protocol.

Many applications can perform concurrent memory accesses, however it is *not guaranteed* that requests *to the same memory address from different ports* are executed in order. See also Section 1.5.

Memory access requests first enter the `NoC-CMC Interface`, where read and write requests from Morpheus applications are buffered and converted from NoC packets to regular CMC read and write requests buffered and sent to the CMC in burst request format (see Section 1.2).

After entering the CMC (central memory controller), memory access requests first reach the *address translator*, where the logical address is translated into the physical rank/bank/row/column quadruple required by the SDRAM, where a "rank" designates a single or group of SDRAM modules controlled by a unique chip select signal. See Section 2.2.3 for more details.

Concurrently, at the *data buffers*, the write request data is stored until the request has been scheduled; for read requests a buffer slot for the data read from the SDRAM is reserved.

The requests then enter the core part of the SDRAM controller, the two-stage buffered memory access scheduler (see Section 1.3). After one request is selected, it is executed by the *access controller* and the data transfer to/from the corresponding data buffer is initiated by the *data I/O* module. Finally, external data transport and signal synchronization for DDR transfers is managed by the `DDR Interface` (see Section 1.4).

1.2 NoC-CMC Interface

The `NoC-CMC Interface` connects the CMC to the Morpheus Network On Chip using the ST Bus Type 3 Protocol [5]. It is responsible for ensuring adherence of NoC requests and CMC responses to their respective transfer protocols, as well as the internal buffering of memory requests and data. In addition, it serves as an error detection unit by identifying unsupported CMC commands and issuing the proper error response to the transfer initiator. As shown in Figure 2, the interface consists of four components: *Command Dispatch*, *Command Buffer Read*, *Command Buffer Write* and *Response Dispatch*.

1.2.1 Command Dispatch

The Command Dispatch module is responsible for ensuring the validity of incoming requests, the forwarding of valid requests, and the generation of error response packets. If an request is deemed valid, it is forwarded to the proper Command Buffer. If an error was detected, the Command Dispatch module generates an error response packet, which is transmitted to the Response Dispatch module. See Table 1 for a list of supported commands and parameters.

Although the `NoC-CMC Interface` has an address bus width of 29 bits, the user must ensure that only the valid address space of the CMC is addressed. The `NoC-CMC Interface` uses *word addressing*, while the CMC uses

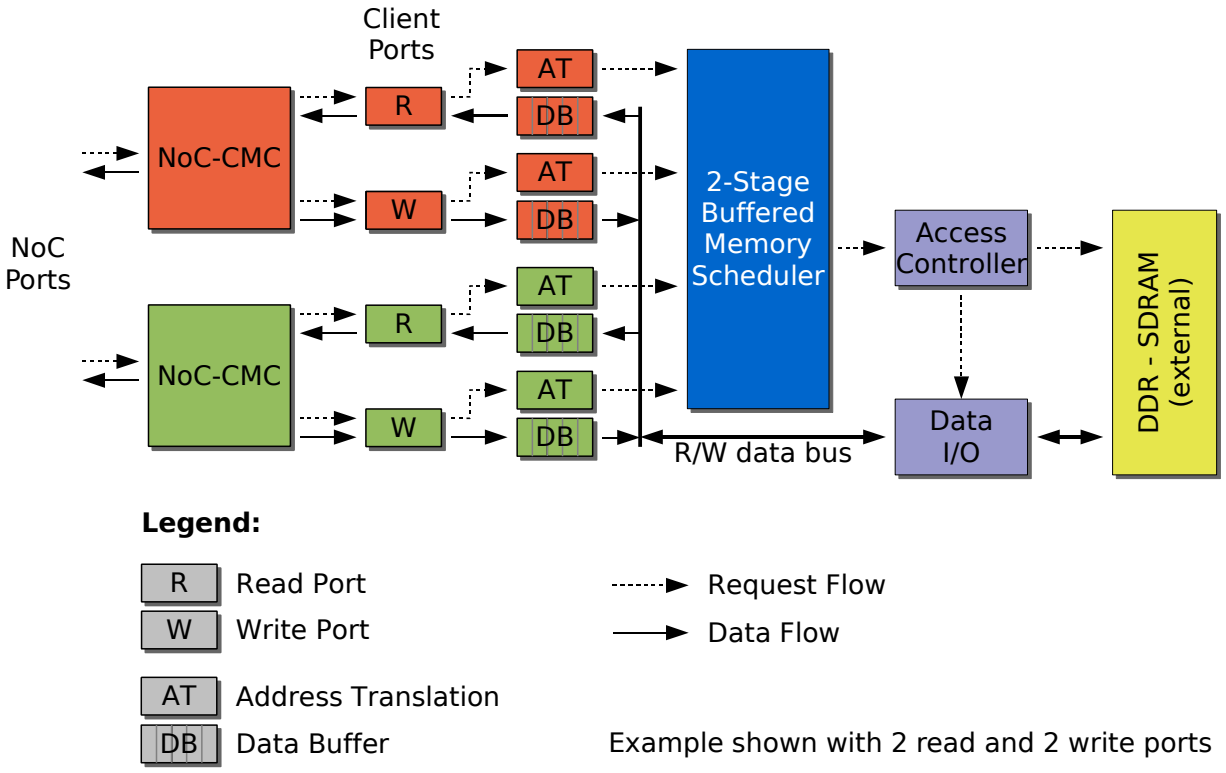

Figure 1: SDRAM controller architecture

byte addressing. Therefore, the usable NoC address space is smaller. The exact size can be calculated as

$$\begin{aligned} &\textit{valid NoC addr. bits} = \\ &\quad \textit{CMC address bits} - \log_2(\textit{wordsize in bytes}) \end{aligned} \tag{1}$$

If a non-burst aligned address is accessed (the address does not correspond to the first word in a burst), the user must ensure that the last word of the request is located within the same burst. For example, it is possible to issue an `ST32` request to address 9, since the request contains 4 data words. Therefore the final address of the command would be 13, and this is a valid request. An invalid example would be an `ST64` request to address 9, which contains 8 words and ends at address 17. This overlaps the burst border (a burst ends at address 1 and the next burst ends at address 16) and is therefore not supported.

1.2.2 Command Buffer Write

The Command Buffer Write module accepts incoming write requests from Command Dispatch and transfers these requests in *burst format* to the CMC.

The internal buffer structure is divided into a data and address buffer. Write requests, which can have different word lengths, require only a single address, which is independent of request size. Therefore, the sizes of the two buffers at synthesis time are completely independent of one another. The only requirement is that the data

Supported commands and parameters

Signal name	*Available parameters*
Opcode (`opc`)	ST08, ST16, ST32, ST64 LD08, LD16, LD32, LD64
Lock (`lck`)	Lock is not supported, `lck = '0'`
Byte enable (`be`)	Byte is not supported, `be = '11111111'`
Address (`address`)	Only the lowest address bits can be utilized. The exact size of the address space is determined using Equation 1. The data corresponding to the command must be contained within a single burst
Transaction ID (`tid`)	No restrictions
Source (`src`)	No restrictions
Priority (`pri`)	Priorities are not supported; signal ignored
Attribute (`attr`)	Attributes are not supported; signal ignored

Table 1: Supported commands and parameters

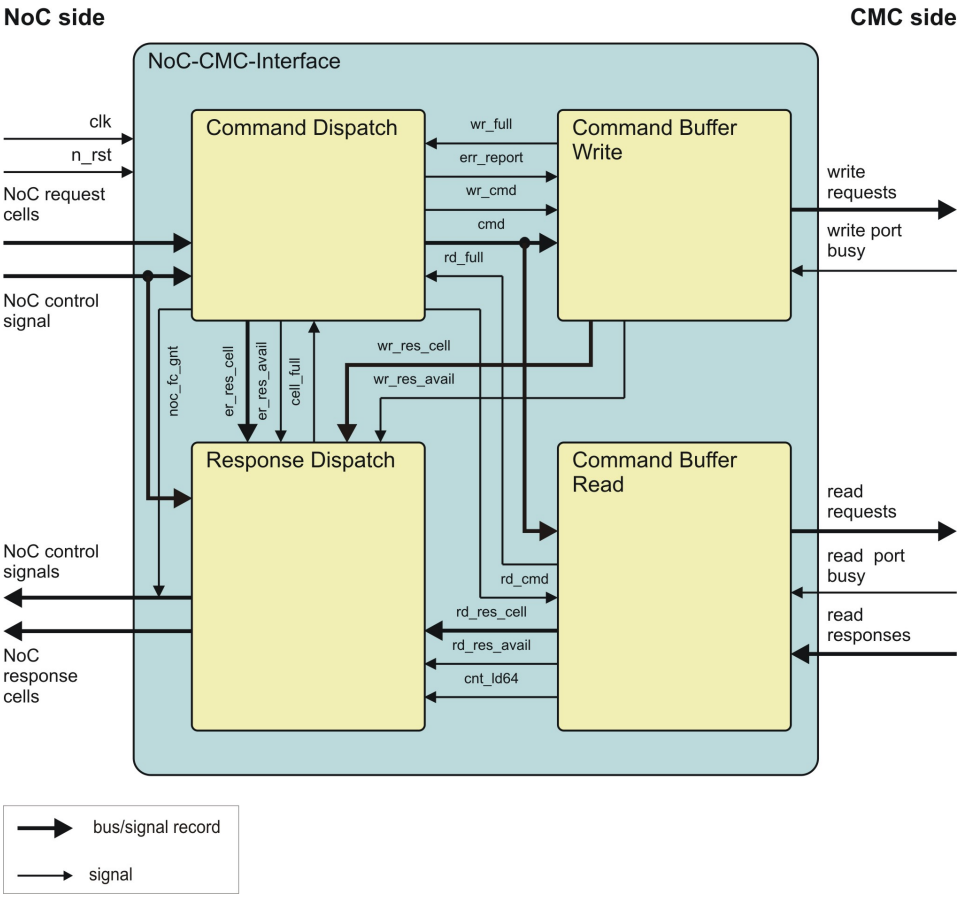

Figure 2: NoC-CMC Interface

buffer s hould always be able to store more than one complete burst. Also, because applications can best utilize the CMC's scheduling optimizations when working with `ST64` commands (8 data words), an address buffer should be $\frac{1}{8}$ the size of the data buffer.

When data is written from the buffer to the CMC, the conversion of the request into a burst-oriented CMC request takes place. If a request is smaller than a burst, the additional words remaining in the burst, which can be technically deemed "empty," are masked out when the data is written to SDRAM. A counter initialized to the lowest three address bits is used to define the position of the first valid data word inside a burst.

1.2.3 Command Buffer Read

The Command Buffer Read module receives read requests from Command Dispatch and forwards them to the CMC. In addition, it is responsible for receiving data words read from the CMC and converting this burst-oriented data into valid NoC response packets.

The main component of the Command Buffer Read module is a temporary storage FIFO. Using multiple pointers, this buffer keeps track of the position for the next incoming request, the position for the next request that will be sent to the CMC, and the position of the read request that next expects to receive read data from the CMC. A relatively simple pointer management is possible due to the internal CMC re-ordering of read requests into their original request order.

Read requests that are smaller than a complete burst are treated similarly as in Command Buffer Write. Additionally, only the burst address of the request is passed to the CMC. A counter initialized to the lowest three address bits is used to define the position of the first valid data word inside a burst. A second counter, which is set to the number of words to be read, defines the final data word to be read.

1.2.4 Response Dispatch

Response Dispatch controls the distribution of response packets to the NoC, as well as the management of the response packet control signals. Since each of the other three `NoC-CMC Interface` modules can transmit a response packet, the Response Dispatch module has been implemented with two separate buffers and a simple arbitration algorithm.

The first buffer handles write responses and error responses, which always consist of a single cell since no real data is transmitted. The second buffer is responsible for

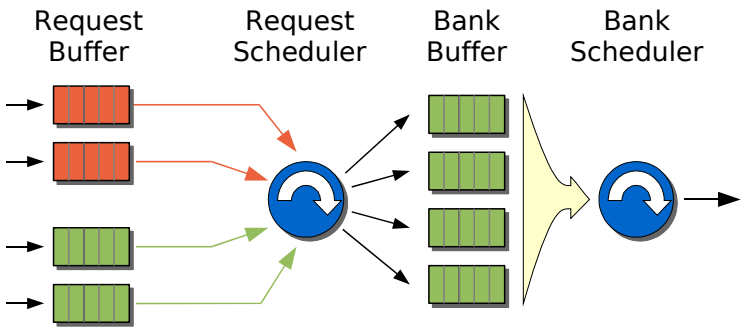

Figure 3: Two-stage buffered scheduler

read responses. A read response packet can consist of 1-8 cells.

The transfer of the outgoing packet to the NoC uses priority arbitration. After successful transfer of a full packet, the current active buffer assigns the transfer priority to the other buffer, except in the case that the second buffer is empty.

1.3 Two-Stage Buffered Scheduler

Figure 3 shows the scheduling stages. For now, a second high priority branch, including flow control, is omitted. This feature is not currently supported in the CMC designed for the Morpheus architecture. However, this feature can be easily incorporated into a later version of the CMC.

1.3.1 Request Buffer, Request Scheduler

The single-slot request buffers are used to decouple the clients from the following scheduling stages. The first scheduler stage, the request scheduler, selects requests from these buffers, one request per two clock cycles, and forwards them to the bank buffer FIFOs. By applying a round-robin arbitration policy, a minimum access service level is guaranteed.

1.3.2 Bank Buffer, Bank Scheduler

The bank buffer FIFOs, one for each bank, store the requests according to the addressed bank. The second scheduler stage, the bank scheduler, selects requests from these bank buffer FIFOs and forwards them to the access controller for execution. In order to increase bandwidth utilization, the bank scheduler performs *bank interleaving* to hide bank access latencies and *request bundling* to minimize stalls caused by read-write and rank switches.

For more details on the CMC architecture, see [1, 2].

1.4 DDR Interface

A significant challenge for the CMC design was the timing synchronization along the DDR-SDRAM data path. This `DDR Interface` was especially difficult to implement due to the fact that the controller itself is clocked at half the speed of the memory, as well as the lack of information such as exact signal delays on the board itself through tristate drivers and I/O pads. A functional interface must therefore be able to adapt itself to many timing requirements. For this purpose, three features were included in the interface:

- DLL elements used to create delay of Data Strobe signal (`dqs`)
- Flexible Capture Unit for transferring data with system clock
- Moving Data Valid Window for acquisition of data on the internal data bus

Figure 4 shows a schematic diagram of the interface. It is important to note that 8 Data signals (`dq`) are always associated with a single Data Strobe `dqs` and Data Masking `dqm` bit. The interface drives a total of 8 `dqs` signals in order to achieve a word size of 64 bits. Therefore, the structure shown is often repeated in the actual implementation. To simplify the schematic, however, redundant elements are eliminated. The following sections describe the `DDR Interface` in terms of *write* and *read requests*. All signal names are based on Figure 4. For more information, consult [4].

1.4.1 SDRAM Write Access

During *write requests*, the CMC assumes bus control and drives all interface signals via the signals `dq_drive` and `dqs_drive`. To double the data rate on the `dq` lines, a simple double data rate Flip Flop was developed with two data inputs and a single data output. On a rising clock edge, input `D0` is registered and on a falling clock edge, input `D1` is registered. The Data Mask signal is similarly generated, which exists to mask out specific data words inside a burst in order to prevent overwriting data in memory. Within the controller itself, all signals are completely synchronous to one another, and the synthesis process is responsible for the observation of setup and hold times of individual gates.

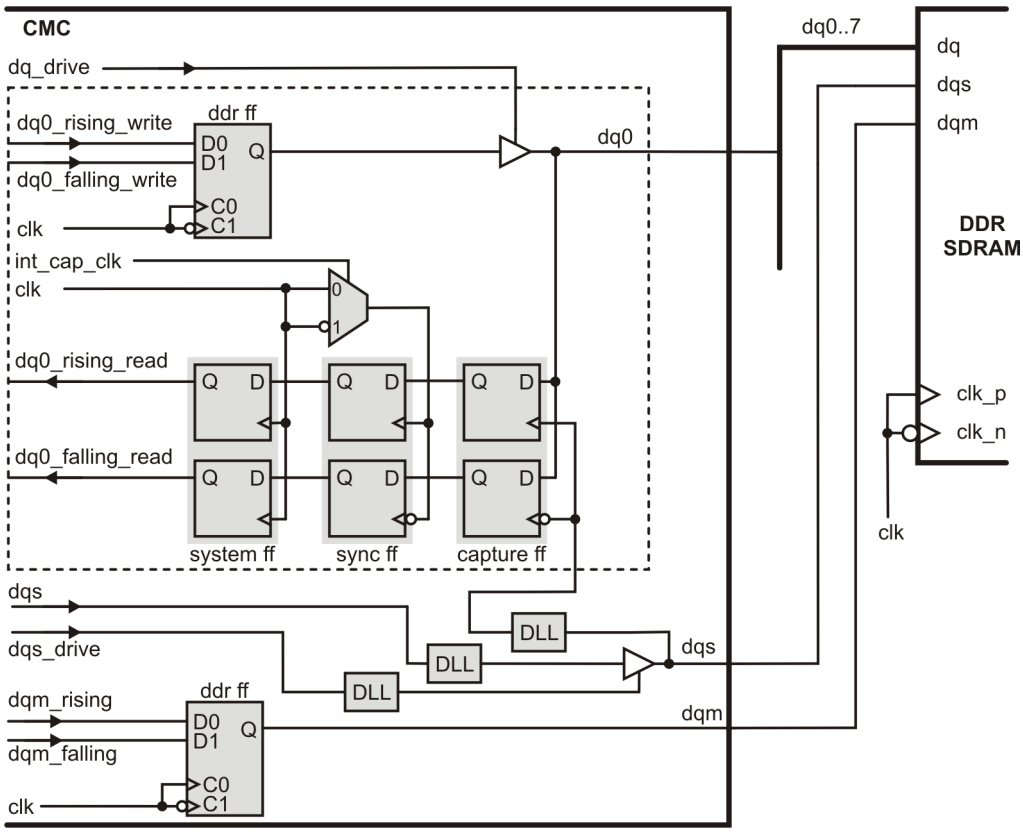

Figure 4: Interface to DDR-SDRAM

However, due to unknown timing information such as signal layout delays, an exact timing cannot be determined at synthesis time.

The delay introduced to the `dqs'` signal can only guarantee that the Data Strobe arrives at the SDRAM after the data itself and therefore that the correct data is registered. It must be noted, however, that different memory modules on the chip can be placed different distances away from the controller, amongst other possible tolerance issues related to manufacturing. Therefore, the interface was designed with maximum flexibility in mind. Each of the 8 `dqs` signals has its own DLL, and all DLLs can be configured independently of each other via the `configspace` module. See Section 2.5 for detailed DLL information.

Figure 5 represents a simplified example of a *write request*. A burst of 8 data words is written into DDR-SDRAM. The first data word is masked out (`dqm` = 1), meaning this position is not overwritten in memory. The Data `dq` and the corresponding Data Strobe `dqs'`, which is the reference signal for data acquisition by the memory, are generated by the CMC and therefore synchronous to the system clock. If we assume the simplification that the PCB trace for the `dq` and `dqs'` signals are the same length, meaning that the data arrives at the same time as the system clock, it becomes clear that the Setup Time is violated. In this case, the DLLs shift the `dqs'` signal to the middle (timing-wise) of the data word in order to generate the most tolerant timing situation possible (the DLLs are also subject to jitter, so the delay is not constant). This creates a new delayed Data Strobe signal called `dqs`.

1.4.2 SDRAM Read Access

The unknown delay times can also create communication problems between the CMC and DDR-SDRAM modules during *read requests*. After the CMC sends the `CAS` signal, the memory module does not transfer the data onto the `dq` bus and set the Data Strobe until the amount of time specified by the CAS latency parameter (2, 2.5, or 3 clock cycles) has passed.

The DDR-SDRAM specifications require that all 8 Data signals that belong to a single Data Strobe do not become valid in the same instant. As shown in Figure 6, all Data signals can only be acquired via a DQS edge during a given time window. This time window is known as the *Data Valid Window* and its duration can be calculated as follows, using the SDRAM Timings shown in Table 2:

$$t_{HP} = min(t_{CH} \cdot t_{CK}, t_{CL} \cdot t_{CK}) \quad (2)$$

$$t_{QH} = t_{HP} - t_{QHS} \quad (3)$$

$$t_{DV} = t_{QH} - t_{DQSQ} \quad (4)$$

To provide timing synchronization, the Data Strobe signals can be supplied in the middle of the *Data Valid Win-*

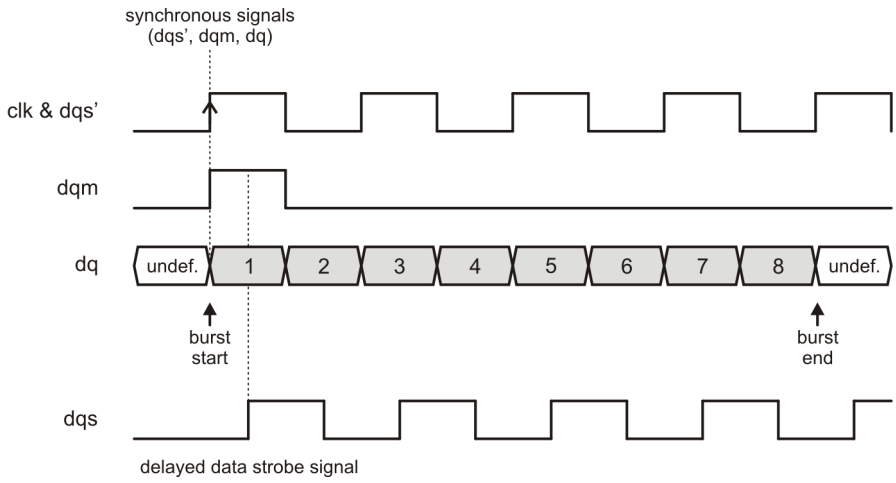

Figure 5: Burst write with data masking

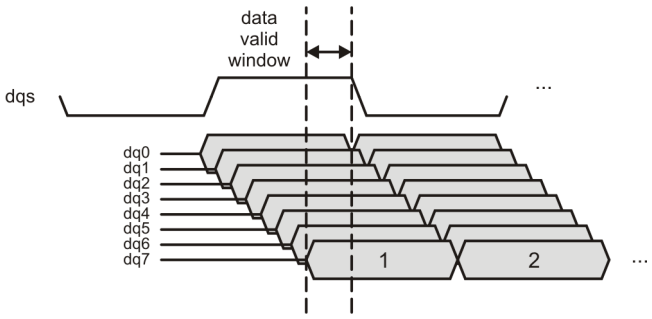

Figure 6: DDR Data Valid Window

dow, with the help of the DLL elements provided for the DQS signals generated by the DDR-SDRAM during *read requests*. This, in combination with the three-stage Capture Unit, allows for proper acquisition of data as shown in Figure 4.

After the data has been acquired by the Capture Flip Flops using the corresponding Data Strobe, they must be properly transfered into the Sync Flip Flops. There is no relationship between the `DQS` signal and the CMC system clock, and therefore the Sync Flip Flops can be clocked with either the CMC system clock or an inverted system clock. The choice depends on which of the two clocks provide more tolerance in regards to meeting timing requirements and properly registering the incoming data. Figure 7 illustrates both possible clocking methods for a data word stored into the Capture Unit Flip Flops using a rising DQS edge. The sum of t_{setup} and t_{hold} is the same in both cases. However, in this example, acquisition with the falling edge of the system clock creates a more balanced result, with timing tolerance in both directions. Next, in the last stage of the Capture Unit, the System Flip Flops ensure that both data paths are now forwarded to the internal data bus with the positive edge of the system clock. Two data paths are required due to the CMCs internal reduction from double data rate to single data rate transfers.

Finally, due to remaining unknown delay times, the *Data Valid Window* may also be required to be shifted back so that all words in a burst can be properly captured by the internal bus. From this moment on, the data words are completely in the system clock domain and can be further processed by the CMC.

1.4.3 Alternatives to DLL Usage

As a possible alternative to the use of DLL elements, delays built into the PCB trace itself were considered. Here, the advantage is the removal of the complex logic used to implement the interface using DLLs as well as a significant savings in chip area. However, the disadvantages of this solution prevented it from being considered for serious use. The delay for individual signal lines must be determined via its length on the board. With the memory blocks used during testing [4], a delay of at least 0.85 ns is required. With an approximate propagation speed of $v_{signal} = 20\frac{cm}{ns}$, this would require a signal length of $l = 17cm$. This length is unrealistic for a modern architecture. In addition, for an exact length calculation all delay times must be known, which would force the controller to be restricted to a specific memory module or in the best case, those modules with almost identical timing behavior.

Parameter	Description	Unit
t_{CK}	clock period	ns
t_{CH}	clock high-level width minimum	t_{CK}
t_{CL}	clock low-level width minimum	t_{CK}
t_{DQSQ}	DQS to last DQ valid	ns
t_{QHS}	clock half period	ns
t_{QH}	DQ-DQS hold	ns
t_{DV}	data valid window	ns

Table 2: SDRAM Timing Parameters

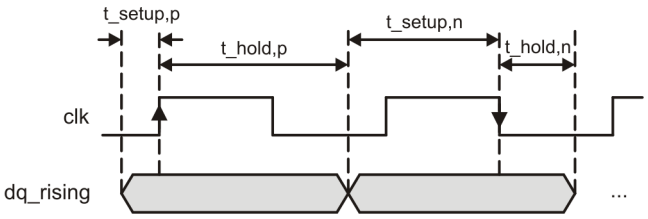

Figure 7: Data synchronization

1.5 Memory Coherency

- Reads and writes from *different ports* to the *same addresses* are potentially executed out-of-order. Provided that the bank buffers do not fill up, a distance of $2n$ clock cycles, with n being the number of ports, is enough to exclude any hazards.

- Reads from *one port* to *different addresses* might be executed out-of-order, however they finish in-order. This implies that the application always receives the requested data in-order. The reordering takes place inside the data buffers.

- Writes from *one port* to *different addresses* might be executed out-of-order. This is a non-issue, however, since they occur at different addresses.

1.6 Modules (Entities)

The complete controller consists of three modules (VHDL entities), `cmc_core`, `configspace` and `noc-cmc`. The entity `cmc_core` is the memory controller itself. This controller core contains the modules described above, implemented as separate entities. It also contains the `DDR Interface`, which consists of the `sdram_datapath` and `sdram_addresspath` entities that implement the single data rate (address, control signals, clock) and double data rate (data, data strobe, data mask) logic for communicating with the SDRAM. This design allows for optimal hierarchy control.

Finally, a single top-level module `cmc_top` exists to instantiate each of the above entities.

2 Configuration

2.1 Overview

The SDRAM controller is largely configured by setting generics (Table 3) for the top module (`cmc_top` entity). However, a few generics need to be specified for the controller core and the `NoC-CMC` interface (`cmc_core` and `noc-cmc` entities). See Tables 4 and 5.

In addition, a few constants specifying maximum values must to be defined. These constants exist in the `cmc_package.vhd` file (Table 6). There is no effect on the generated code if these maximum values are larger than actually needed, but the synthesis tool might generate more warnings. If the values are too small, synthesis and simulation will fail and appropriate messages will be printed. If these values are larger than necessary for the actual configuration, information messages will report which constants could be reduced. However, simulation and synthesis does not know if more than one controller is used, which can produce constant values that are too low for a specific controller instance. Therefore, it is best to leave the constants at larger values, since it has no negative impact other than additional warnings generated by simulation and synthesis tools.

2.2 Configuration Options

2.2.1 SDRAM Support

The CMC was designed to support a wide variety of DDR-SDRAM modules. Its flexibility allows it to function properly with all DDR-SDRAM modules that support the generic values defined in `G_SD_`$\star$ in Table 3. These generics, along with the `configspace` module, configure the current DDR-SDRAM and appropriate CMC settings. In addition, the controller was designed to operate at the

Generic	Description	Current value
SDRAM		
G_SD_DBUS_BITS	databus width	64
G_SD_ABUS_BITS	addressbus width	14
G_SD_BANK_BITS	bank addressbus width	2
G_SD_NCS	number of chip selects	2
G_SD_ROW_BITS	number of row address bits	14
G_SD_COL_BITS	number of column address bits	10
G_SD_PRE	precharge bit	10
G_SD_BURSTLENGTH	burstlength	8
NoC-CMC Interface		
G_BUFF_CMD_LENGTH_RD	read address buffer length	4
G_BUFF_DATA_LENGTH_RD	read data buffer length	16
G_BUFF_ADDR_LENGTH_WR	write address buffer length	4
G_BUFF_DATA_LENGTH_WR	read data buffer length	16
G_BUFF_CELL_LENGTH	response buffer length for error and write	4
G_BUFF_CELL_LENGTH_RD	response buffer length for read	16
G_NOC_ADDR_BITS	number of address bits of NoC/buffer	29
G_NOC_DATA_BITS	number of data bits of NoC/buffer	64
G_APP_BURST_LENGTH	CMC burst length	8
G_CMC_ADD2ADD_CYC	min number of cycles between two read requests to cmc	3
Application		
G_APP_NPORTS_RD_STD	number of standard priority read ports	2
G_APP_NPORTS_RD_HIGH	number of high priority read ports	0
G_APP_DATA_BITS_RD	read port data width in bits (array)	64
G_APP_ADDR_BITS_RD	read port address width in bits (array)	27
G_APP_NPORTS_WR_STD	number of standard priority write ports	2
G_APP_NPORTS_WR_HIGH	number of high priority write ports	0
G_APP_DATA_BITS_WR	write port data width in bits (array)	65
G_APP_ADDR_BITS_WR	write port address width in bits (array)	27
G_APP_SINGLE_BB_STD	single bank buffer for standard priority	false
G_APP_SINGLE_BB_HIGH	single bank buffer for high priority	false
Address Translator		
G_AT_USE_TABLES	disable / enable usage of address translation tables	false
G_AT_TABLES_RD	read port address translation tables (2 dimensional array)	(see text)
G_AT_TABLES_WR	write port address translation tables (2 dimensional array)	(see text)
G_AT_MAPPER	address mapper	MAP_LOW
Flow Control (high priority ports only)		
G_FC_T	window size T in clock cycles ($T = 0$: flow control disabled)	0
G_FC_N	requests: N requests within window of T clock cycles ($N = 0$: flow control disabled)	0
Misc		
G_SIMULATE	compile for simulation, valid for behavioral simulation only	false
G_SIMULATE_TCLK	simulation clock cycle time	8 ns
G_MEM_EN	enable SDRAM simulation module instantiation	false
G_INT_DATA_BITS_WR	internal data buffer $->$ data handler bus width	130
G_INT_DATA_BITS_RD	internal data handler $->$ data buffer bus width	128

Table 3: Core CMC generics. Fixed values for Morpheus CMC shown

Generic	Description	Current value
G_SD_DBUS_BITS	data bus width, MUST match core G_SD_DBUS_BITS generic	64
G_SIMULATE	Compile for behavioral simulation (adjust some timing parameters)	false
G_SIMULATE_TCLK	System clock period, required if G_SIMULATE is set to true	false

Table 4: DDR datapath generics

Generic	description	possible values
G_SD_ABUS_BITS	address bus width, MUST match core G_SD_ABUS_BITS generic	14
G_SD_BANK_BITS	bank address bus width, MUST match core G_SD_BANK_BITS generic	2
G_SD_NCS	number of chip selects, MUST match core G_SD_NCS_BITS generic	2

Table 5: SDR datapath generics

Constant	Description
C_APP_DATA_BITS_MAX	maximum width of application port data bus
C_APP_ADDR_BITS_MAX	maximum width of application port address bits
C_DATATAG_BITS_MAX	maximum number of data tag bits, used internally
C_NCS_BITS_MAX	maximum number of chip selects
C_BANK_BITS_MAX	maximum width of bank address
C_ROW_BITS_MAX	maximum width of row address
C_COL_BITS_MAX	maximum width of column address
C_INT_DATA_BITS_WR	maximum width of data buffer -> data handler bus (must be multiple of (G_SD_DBUS_BITS + word mask))
C_INT_DATA_BITS_RD	maximum width of data handler -> data buffer bus (must be multiple of G_SD_DBUS_BITS)
C_DB_NBURSTS	maximum data buffer size in bursts
C_LOGBANK_BITS_MAX	maximum width of logical bank address

Table 6: Maximum constants in cmc_package.vhd

fixed speed of 200 MHz, which implies the use of DDR-SDRAM operating at 400 MHz.

2.2.2 Application Ports

The generics G_APP_* in Table 3 configure the application ports. There are only read and write ports, no read-write ports. A maximum of 8 read and 8 write ports are currently supported. These maximum values are not verified, however exceeding them will certainly lead to severe clock frequency degradation. As stated above, the this version of the Morpheus CMC has 2 independent read and 2 independent write ports.

The SDRAM controller has different generics and interfaces for read and write ports. Ports are numbered from $0 \ldots n_{rd}-1$ for read ports and from $0 \ldots n_{wr}-1$ for write ports. If used, high priority ports come before standard priority ports (they have the lower port numbers).

Number of Ports. The number of ports is specified by the generics G_APP_NPORTS_RD_HIGH and G_APP_NPORTS_RD_STD for read ports (high priority and standard priority), G_APP_NPORTS_WR_HIGH and G_APP_NPORTS_WR_STD specify the number of write ports (high and standard priority). The Morpheus CMC includes standard priority ports only; therefore the generics defining the number of high priority ports is set to 0.

Port data width. Each application port can have a different data width, however it must be a power of 2 and the maximum size is twice the SDRAM data bus width [1]. To specify the port widths, the generics G_APP_DATA_BITS_RD and G_APP_DATA_BITS_WR are used. Both generics take arrays of positive values [2] that specify the port widths. For read ports, the array range is $0 \ldots n_{rd}-1$; for write ports, $0 \ldots n_{wr}-1$ (write ports). The Morpheus CMC fixes the data width of each port to 64 bits.

Port address width. Each application port's address bus can have a different size for instances when a given application does not need to access the full memory range. The maximum address width depends on the current SDRAM configuration and can be calculated using the following formula:

$$\begin{aligned} addrwidth \;=\; & \texttt{G\_SD\_ROW\_BITS} \\ & + \texttt{G\_SD\_COL\_BITS} \\ & + \texttt{G\_SD\_BANK\_BITS} \\ & + \lceil \log_2(\texttt{G\_SD\_NCS}) \rceil \\ & + \log_2(\texttt{G\_SD\_DBUS\_BITS}/8) \end{aligned}$$

The port address widths are set using the generics G_APP_ADDR_BITS_RD and G_APP_ADDR_BITS_RD in the same way as for the application port data width. For the current Morpheus configuration, this equation provides an address width of 29 bits.

[1] Provided that the C_APP_DATA_BITS_MAX constant is set to at least this value.

[2] type t_natural_array, defined in the utilities package

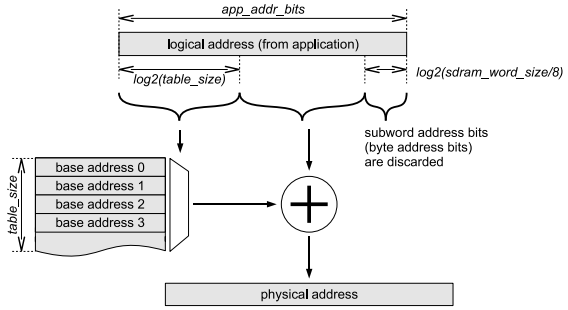

Figure 8: Address translation, 1st step

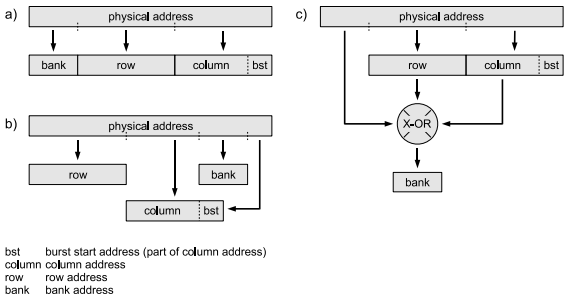

Figure 9: Address translation, 2nd step

Application burst length. SDRAM accesses are always performed at full SDRAM burst length (currently, 8 SDRAM words). Therefore, the CMC expects to receive a full burst from the NoC−CMC interface on a memory write access, or it receives a full burst from the SDRAM controller on a memory read access, respectively. The number of words the NoC−CMC interface has to send or receive per memory access request depends on the SDRAM burst length (currently, always 8) and the relationship of the SDRAM data bus width to the application data bus width and can be calculated using the following formula:

$$burstlength = \texttt{G\_SD\_BURSTLENGTH} \cdot \frac{\texttt{G\_SD\_DBUS\_BITS}}{\texttt{G\_APP\_DATA\_BITS\_*(i)}}$$

Example: with a SDRAM burst length of 8, a SDRAM data bus width of 64 bits and a port width of 64 bits, which corresponds to the current Morpheus CMC settings, the NoC−CMC interface always writes or reads 8 words per memory access request.

2.2.3 Address Translation

The address translation consists of two steps, described in Figures 8 and 9.

Logical to physical address translation. Since the logical address is *byte-aligned*, but the SDRAM controller can only address *SDRAM words*, the logical address is SDRAM-word-aligned by discarding the

$$\log_2(\texttt{G\_SD\_DBUS\_BITS}/8)$$

LSBs.

Example: if the SDRAM controller has a databus size of 64 bits (8 bytes), the 3 lowest bits of the logical address are discarded. Addresses are therefore byte-aligned. It is the user's responsibility to not use these lower bits and set them to "0" (the superfluous address lines are discarded during synthesis). During simulation, a warning message is printed if these bits are not "0", however this cannot be checked during synthesis.

The following table-based address translation is optional. To use it, the generic G_AT_USE_TABLES has to be set to true. If this generic is set to false – the default value – the tables are disabled and the SDRAM-word aligned logical address is used as physical address. In the Morpheus CMC, table-based address translation is disabled due to lack of specifications necessary to create the tables, as well as to reduce logic complexity. However, as it is a standard CMC feature, it is described below.

The MSBs of the SDRAM-word aligned logical address are used to select a base address from the address translation table, which is then added to the remaining logical address bits to form the final physical address.

The size and content of the address translation table can be configured on a per-port basis via the generics G_AT_TABLES_RD and G_AT_TABLES_WR (read and write ports). These generics each take a 2-dimensional array, where the 1st dimension is the port number and the 2nd dimension the table entries for each port. To be able to specify different table sizes for each port, the last entry of each table has to be set to "-1". If desired, just one single base address can be used by specifying a single entry address translation table.

Table address entries are specified using bytes, but since the SDRAM can only address words, the lowest

$$\log_2(\texttt{G\_SD\_DBUS\_BITS}/8)$$

bits are discarded.

Physical to (rank)/bank/row/column translation. Internally, different SDRAM ranks are treated as memory bank extensions, since different memory modules may operate independently similar to (but not exactly like) banks within a single module.

The number of *logical banks* is therefore

$$\texttt{G\_SD\_NCS} \cdot 2^{\texttt{G\_SD\_BANK\_BITS}}.$$

The following address translation step converts the physical address into the logical_bank/row/column triple.

To perform this step, three address translation *mappers* exist. Regardless of the mapper used, the

$$\log_2(\texttt{G\_SD\_BURSTLENGTH})$$

MSBs of the physical address (always the lowest 3 bits with a burst length of 8) are used as the "burst start address"

and must be sent to the SDRAM as the lowest column address bits. These specify which word of an SDRAM burst is read/written first, implementing a critical word first approach.

a) *MAP_HIGH:* The MSBs of the physical address bits are used as the logical bank address, followed by the row address, followed by the column address. This is a very simple address mapping and does not distribute memory requests across different banks, since the MSBs generally do not change often. Used mostly for debug purposes.

b) *MAP_LOW:* Like "MAP_HIGH", but the physical address bits just above the "burst start bits" are used as the logical bank address. This avoids the disadvantage of "MAP_HIGH" because the lower address bits change frequently. The Morpheus CMC utilizes the "MAP_LOW" mapper.

c) *MAP_EXOR:* Like "MAP_HIGH", but the final logical bank address is created by XOR-ing the bank address with the rest of the physical address (except the "burst start address" bits). This distributes memory request across banks even better than "MAP_LOW". However, the generated address sequences appear relatively random, making debugging more difficult.

The mapper is set globally for all ports by setting the `G_AT_MAPPER` generic to `MAP_HIGH`, `MAP_LOW` (default) or `MAP_EXOR` as desired.

If `MAP_EXOR` is selected, the `G_AT_MAPXOR_TABLE` generic specifies which bits of the physical address are XORed with the bank address. The generic provides an array of 32-bit masks, one mask for each logical bank address bit:

$$logicalbankbits = \texttt{G\_SD\_BANK\_BITS} \cdot \log_2(\texttt{G\_SD\_NCS})$$

Each mask specifies the bits of the physical address that are to be XORed with the corresponding bit of the logical bank address. For example, with a two bank system, if `G_AT_MAPXOR_TABLE[0] = 16#0100#` and `G_AT_MAPXOR_TABLE[1] = 16#0200#`, bit 8 of the physical address will be XORed with bit 0 of the bank address, and bit 9 will be XORed with bit 1 of the bank address.

Like the table addresses, the bits in the masks assume byte-oriented physical addresses. As explained above, single bytes cannot be addressed, and since different burst start addresses specify the same memory area that is transferred by a burst access, the burst start address bits must never change the bank address. For these reasons, the lower

$$\log_2(\texttt{G\_SD\_DBUS\_BITS}/8) + \log_2(\texttt{G\_SD\_BURSTLENGTH})$$

bits of the masks are treated as zero. For example, take a system with a data bus width of 32 bit (2 byte address bits) and a burstlength of 8 (3 burst start address bits). In this case, the lower 5 bits of the tables are treated as zero.

If the table provides only one entry and if the entry is 0 (the default value), a default scheme will be used in which the physical address bits are XORed in a cyclic way with the bank bits. For example, take the example above and assume 4 banks (2 bank address bits): bit 5 of the physical address is XORed with bank bit 0, bit 6 with bank bit 1, bit 7 with bank bit 0, and so forth. With an 8 bank system (for example, 2 ranks of 4-bank SDRAMs), physical address bits 5, 8, 11,... are XORed with bank bit 0, physical address bits 6, 9, 12,... are XORed with bank bit 1, and physical address bits 7, 10, 13,... are XORed with bank bit 2.

2.3 Morpheus Specific Run-Time Configuration

The exact values of the configuration generics described above clearly depend on the type of DDR-SDRAM used, the clock frequency, and overall board layout. For the Morpheus CMC, many values for the generics have been determined based on design requirements and cannot be changed. This is in contrast to the flexible nature of the original CMC design, which was created for use in an flexible FPGA environment.

Despite the requirement that many parameters, such as address bus width, data bus width, and the number of application ports, must be determined before logic synthesis, a certain degree of flexibility must remain in the Morpheus CMC so that it may support different DDR-SDRAM modules. To achieve this goal, a programmable `configspace` module was created.

The `configspace` module is a programmable 16-bit register set used to configure the CMC for a specific SDRAM module. It allows run-time configuration of SDRAM timing, SDRAM layout, and of the DDR path DLL elements used to generate necessary signal delays. Additionally, the module contains two diagnostic registers, which provide flags to assist in testing the controller functionality.

Its interface to the Morpheus architecture is implemented as an ST Bus Target Interface Type 1 [5]. The interface only supports the Type 1 opcode “store 2 bytes” (0010), without byte enable. Furthermore, only a single cell operation is accepted. In the event of an unsupported configuration attempt, the interface detects violations and transmits a response opcode that indicates a configuration failure, in which case the configuration registers do not change state.

See Tables 7- 18 for a description of the function of each register, as well as valid configuration values.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved					trfc					trp	trw	tcl		trcd	

Table 7: SDRAM Timing Register 1 (Address 0x00000000)

Register value	Physical parameter (clock cycles)
RAS to CAS delay (trcd)	
00	2
01	3
10	4
11	reserved
CAS latency (tcl)	
00	2
01	2.5
10	3
11	reserved
Write recovery (twr)	
0	2
1	3
Row precharge (trp)	
0	2
1	3
Refresh cycles (trfc)	
XXXXX	Value between 00000 and 111111 represents the number clock periods during a refresh cycle

Table 8: SDRAM Timing Register 1 configuration

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
tref															

Table 9: SDRAM Timing Register 2 (Address 0x00000002)

Register value	Physical parameter (clock cycles)
Refresh interval (tref)	
XXXXXXXXXXXXXXXX	Value directly represents the number of clock cycles between SDRAM refresh cycles

Table 10: SDRAM Timing Register 2 configuration

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved											ncs	ncols		nrows	

Table 11: SDRAM Layout Register (Address 0x00000004)

Register value	Physical parameter (clock cycles)
Number of row address bits (nrows)	
00	12
01	13
10	14
11	reserved
Number of column address bits (ncols)	
00	9
01	10
10	11
11	12
Number of chip selects (ncs)	
0	1
1	2

Table 12: SDRAM Layout Register configuration

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved												idd			icc

Table 13: CMC Setup Register (Address 0x00000006)

Register value	Physical parameter
Polarity of internal SDRAM capture clock (icc)	
0	Normal system clock used
1	Inverted system clock used
Additional delay to data available signal (idd)	
XXX	Value directly represents the number of clock cycles for additional delay to wait before accepting incoming SDRAM data as valid

Table 14: CMC Setup Register configuration

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved							user_cmd								

Table 15: DLL Setup Register X (Address 0x00000008-0x00000028)

Register value	Physical parameter
Additional delay to data available signal (user_cmd)	
XXXXXXXXX	Value between 000000000 (min) and 110001111 (max) directly represents delay command applied to DLL element. There are 17 total DLLs; 8 input, 8 output, 1 data strobe.

Table 16: DLL Register configuration

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved							cmd	data	addr	eop	add burst	add space	opc	be	lck

Table 17: Diagnostics Register (Address 0x00000030-32)

Register value	Meaning
Lock (lck)	
0, 1	1 Signifies a protocol error
Byte enable (be)	
0, 1	1 Signifies byte enable usage caused a protocol error
Opcode (opc)	
0, 1	1 Signifies invalid opcode caused a protocol error
Address space (add_space)	
0, 1	1 Signifies access outside valid address space
Address offset (add_burst)	
0, 1	1 Signifies address offset creates an error
End of package (eop)	
0, 1	1 Signifies end of package caused a protocol error
Address buffer overflow (addr)	
0, 1	1 Signifies buffer overflow in `Command Buffer Write`
Data buffer overflow (data)	
0, 1	1 Signifies buffer overflow in `Command Buffer Write`
Command buffer overflow (cmd)	
0, 1	1 Signifies buffer overflow in `Command Buffer Read`

Table 18: Diagnostics Register

2.4 Initialization

Before the first read or write command can be issued, the CMC must be initialized. The initialization procedure includes the initialization of the SDRAM, during which the SDRAM is allowed to arrive at a stable state after system start-up, as well as the initialization of the controller itself via the `configspace` module (see Section 2.3), which must be loaded with valid initialization data. This configuration interface is the only module that does not automatically revert to a defined state via the Chip Reset `nreset` signal. When the Chip Reset signal is activated, the CMC begins its `configspace` module-dependent initialization procedure. This procedure first ensures that the Mode Registers of the DDR-SDRAM module are properly set, which contain the physical description of the DDR-SDRAM (Timing and Layout). The procedure also performs initial calibration of the DLL elements.

It should be noted that the DDR-SDRAM standard requires a delay of 200 μs after SDRAM clock stabilization before the initialization sequence can be started [3]. This delay ensures that the SDRAMs internal DLL elements have been initialized. These DLLs are not to be confused with the `DDR Interface` DLLs. This delay is not created by the CMC itself and must be verified by the user.

In addition to the above initialization steps, the controller must also be fine-tuned for error-free operation due to various design factors unknown at the time of implementation, such as gate-specific setup and hold times and signal delays created by the physical chip layout. Therefore, with the help of a *test phase*, it must be determined which settings ensure error-free controller operation. An example *test phase* could consist of a test burst of known data, which is written into memory via the CMC and then immediately accessed with a read request. With each "test access," consisting of one write and read operation, a different configuration could be used. In this way, a multi-dimensional test field can be generated, which demonstrates with which parameters the CMC is functional.

2.5 DLL Elements

DDR-SDRAM interfaces require precise timing constraints to guarantee proper data delivery to SDRAM and proper capturing of SDRAM data. Therefore, it is necessary to introduce certain delays to the signals involved in the DDR-SDRAM interface. To attain this goal, 8 Delay Locked Loops (DLL) exist for read signals, 8 DLLs exist for write signals, and a final DLL exists to delay the `DQS Tristate` drive signal. All 17 DLL elements are identical and contain a programmable register (`user_cmd`) that, via its 400 possible values, can generate a signal delay from 3.7 ns to 9.2 ns (see Table 15 and 16) [6]. The worst and base case behavior depends on input voltage and operating temperature. A more detailed explanation of the functionality of the DLL elements within the DDR-SDRAM interface can be found in Section 1.4.

2.6 Diagnostic Register

The ST Bus Protocol supports only very basic error reporting, with information regarding the source and the type of the failed request. However, information with a specific error code or status information is not provided. On the other hand, data traffic and the behavior of the Morpheus HREs are difficult to predict, which makes this type of information about the error's origin desirable.

The two diagnostic registers described in Tables 17 and 18 were created for this purpose. These registers contain status information from the `NoC-CMC Interface` and can be read over the existing ST Bus Type 1 Interface designed for CMC configuration. With the help of an ST Bus to AMBA bus converter, this information can be easily accessed, providing the user with information regarding buffer overflows and problematic read or write requests.

3 Usage

3.1 Application Interfaces

The Morpheus CMC's main interface is the `NoC-CMC Interface`. All clients compliant with the ST Bus Protocol Type 3 interface can send and receive requests from the CMC. However, this interface simply converts requests based on the Type 3 protocol into CMC formatted requests. For a full understanding of the CMC functionality, it is important to understand the design of the internal application interface as well, despite the fact that this interface is hidden to the client in the context of the Morpheus project.

The controller application interface (ports) use records (`t_app_reqwr`, `t_app_reqrd`, `t_app_reprd`). With a configurable CMC, the `data` and `addr` fields of these records must be declared using the `C_APP_DATA_BITS_MAX` and `C_APP_ADDR_BITS_MAX` constants from the `cmc_package` file, which will probably not match the real port data and address widths. Therefore, the exact range need to be specified when accessing these fields.

For the Morpheus CMC, because it is designed specifically for the Morpheus architecture, these constants do match the real port data and address widths. The secondary specification remains for flexibility in future CMC versions.

3.2 Application Protocols

For all read and write requests to the CMC, the following protocols must be followed to ensure proper operation. The `NoC-CMC Interface` ensures compliance with these requirements. However, clients connecting to the CMC via the `NoC-CMC Interface` should also support a similar read and write protocol to prevent stalling in the interface. Therefore, a detailed explanation of the CMC application protocol is provided.

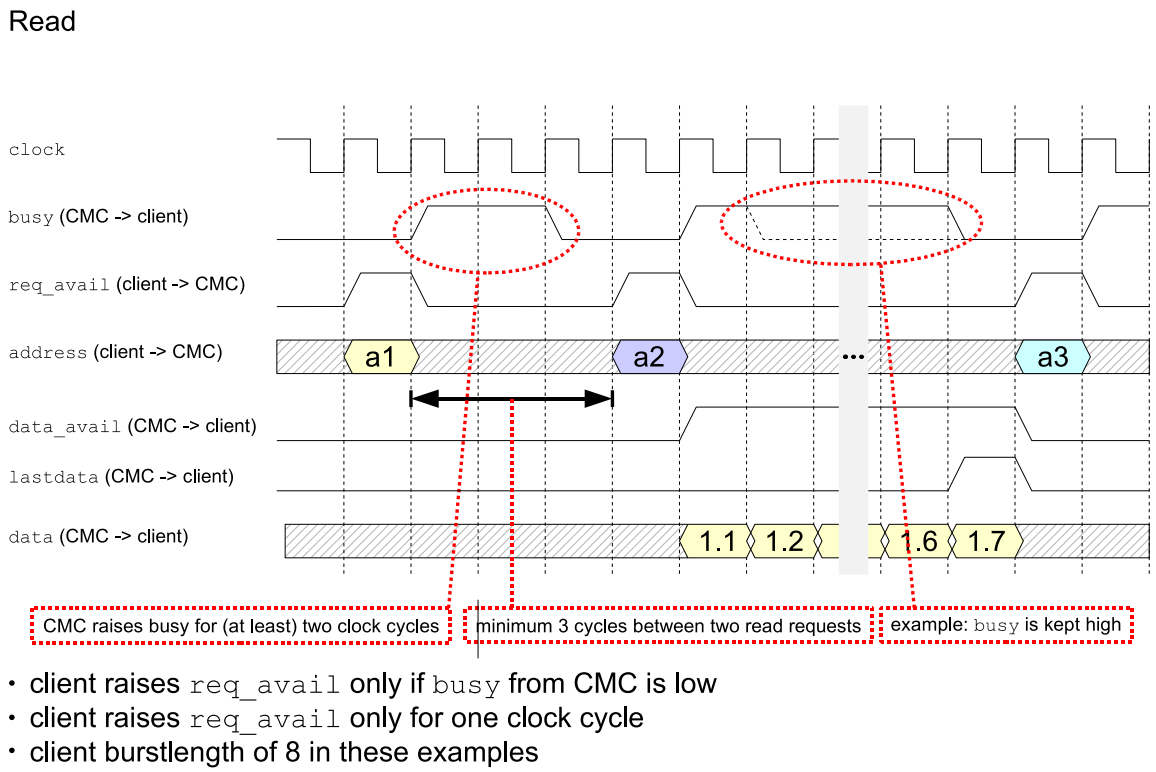

Figure 10: Application read protocol

3.2.1 Read Protocol

See Figure 10. Issuing requests:

t_0 *If* `busy` *is low*, the client places the logical address on `address` and raises `req_avail`.

t_1 Application releases `req_avail`, CMC acknowledges by raising `busy`.

t_2 CMC keeps `busy` high.

t_3 Controller releases `busy`.

t_4 Application is ready to issue next request.

Note t_3: Controller might delay releasing `busy` if internal buffers are full.
Read replies (finishing read requests):

t_0 Controller writes 1st data word on `data` and raises `data_avail`.

t_1 Controller writes 2nd data word on `data` and keeps `data_avail` high.

t_n Controller writes last (n-th) data word on `data`, keeps `data_avail` high and raises `last_data`.

t_{n+1} Controller releases `data_avail` and `last_data`.

3.2.2 Write Protocol

See Figure 11. Issuing requests:

t_0 *If* `busy` *is low*, the client places the logical address on `address`, puts 1st data word on `data` and raises `req_avail`.

t_1 Application puts 2nd data word on `data` and releases `req_avail`, controller acknowledges by rising `busy`.

t_n Application puts last data word on `data`, controller releases `busy`.

t_{n+1} Application is ready to issue next request.

Note t_n: Controller might delay releasing `busy` if internal buffers are full. If a write request to port i is finally executed, the bit i in the `o_app_donewr` vector will be set for one clock cycle.

3.3 Clocks

The Morpheus CMC is designed to operate at 200 MHz. It uses a single system clock input named `clk`. Internally, the SDRAM clocks are derived using this single clock input.

In the CMCs original FPGA incarnation, three phases of a 125 MHz clock were used to simplify the communication between the CMC and the DDR memory modules. The phases $0°$, $90°$ and $180°$ were therefore created and

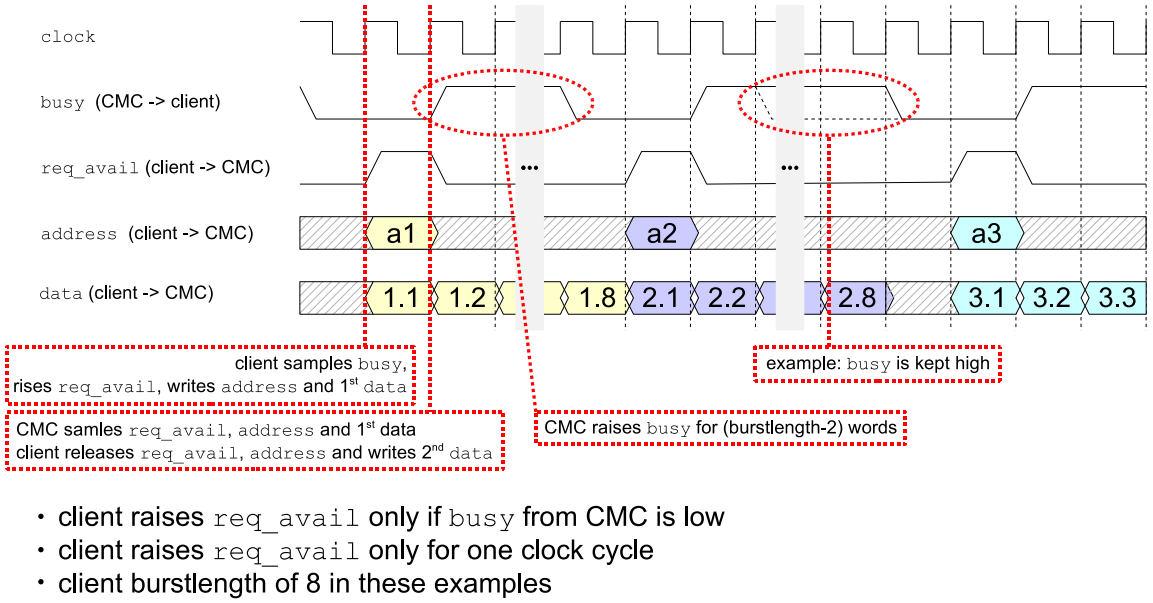

Figure 11: Application write protocol

included in the FPGA clock net, external to the CMC itself, using a Xilinx custom DCM block (digital clock management). Such blocks do not exist in the supplied ST Technology Libraries. Therefore, the design was redone to utilize two phases of the 200 MHz clock described above, $0\,^\circ$ and $180\,^\circ$. The $180\,^\circ$ phase shift is created using simple inversion but must be precise relative to the $0\,^\circ$ shifted clock.

3.4 Reset

The Morpheus CMC includes a reset signal input. This reset signal, which is active-low, launches the initialization procedure described in 2.4 when a digital high value is detected on its input for one clock cycle. When a reset is performed during normal operation, the CMC is re-initialized with the values currently stored in the `configspace` configuration registers, the SDRAM is re-initialized, and the contents of SDRAM are no longer considered valid.

3.5 Instantiation

The CMC modules use the so called "component-less" instantiation method, which means that no components are declared and that the `cmc` library has to be accessed directly by using the `"entity"` statement.

For compatibility with the Morpheus HDL database, two blocks are instantiated as components.

3.6 Resource usage

Resource usage heavily depends on the CMC configuration. For the Morpheus design, configuration decisions are fixed. Therefore, precise resource usage can be reported.

The Morpheus CMC relies on the ST Microelectronics 90 nm technology libraries, as well as Delay Locked Loop technology libraries and custom ST memory modules (memory cuts) for internal storage. In each of the above technology libraries, a single gate occupies a total area of 4.4 μm^2/gate.

Table 19 shows the resource usage for a SDRAM controller on the Morpheus platform with

- 2 NoC-CMC client ports
- 4 standard priority application ports, 2 read and 2 write
- 64-bit application data bus size
- MAP_LOW address mapping with default setup
- 64-bit SDRAM data bus
- 13-bit SDRAM address bus (13 bits row, 10 bits column)
- 4 SDRAM banks
- 2 chip selects
- QoS disabled (prioritization and flow control)
- Multiple bank buffer FIFOs for standard priority requests
- 17 total DLL elements used for internal DDR signal delays
- 2 custom 512 word x 65 bit ST memory cuts
- 2 custom 512 word x 64 bit ST memory cuts

Module	Size (μm^2)
cmc_core	1,353,859
ST memory cuts	1,145,855
configspace	5,725
dqs_delay	257,024
noc-cmc_top_0	118,222
noc-cmc_top_1	118,320
noc-cmc_top_2	118,433
Synthesizable area (μm^2)	568693
Synthesizable area (KiloGates)	126

Table 19: CMC example resource usage

3.7 VHDL Code

The Morpheus CMC source tree is maintained as a hierarchical HDL database. Consult Morpheus Deliverable 4.5.1 Preliminary Description of HDL Database for more information.

4 Conclusion

This concludes the documentation of the Morpheus DDR SDRAM controller. Application partners wishing to correctly utilize the controller to access external memory should consult [3, 4, 5, 6] for further relavant information.

References

[1] Sven Heithecker, Amilcar do Carmo Lucas, and Rolf Ernst. A Mixed QoS SDRAM Controller for FPGA-Based High-End Image Processing. In *Workshop on Signal Processing Systems Design and Implementation*, page TP.11. IEEE, 2003.

[2] Sven Heithecker and Rolf Ernst. An FPGA Based SDRAM Controller with Complex QoS Scheduling and Traffic Shaping. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, page 277. ACM, February 2005.

[3] JEDEC. *Double Data Rate (DDR) SDRAM Specification*. JEDEC Solid State Technology Association, JESD79C edition, March 2003.

[4] Micron Technology, Inc. 512Mb DDR SDRAM Component: MT46V64M8BN-5B. Data sheet, Micron Technology, Inc., April 2004.

[5] Alberto Scandurra. STBus Communication System: Concepts And Definitions. Technical Report V3.2, ST Microelectronics, April 2006.

[6] ST Microelectronics. *DLL User Manual*.