SDRAM Library

The XMOS SDRAM library is designed for read and write access of arbitrary length 32b long word buffers at up to 62.5MHz clock rates. It uses an optimized pinout with address and data lines overlaid along with other pinout optimizations to implement 16 bit read/writes to Single Data Rate (SDR) SDRAM devices of size up to 256Mb, while consuming a total of just 20 xCORE I/O pins.

Features

The SDRAM component has the following features:

- Configurability of:
  - SDRAM capacity
  - clock rate (62.5 to 25MHz steps are provided)
  - refresh properties
- Supports:
  - read of 32b long words
  - write of 32b long words
  - one or more clients
  - asynchronous command decoupling with a command queue of length 8 for each client
  - refresh handled by the SDRAM component itself
- Requires a single core for the server
- Requires 500MHz core clock operation

Components

- SDRAM server
- Memory address allocator

Resource Usage

This following table shows typical resource usage in some different configurations. Exact resource usage will depend on the particular use of the library by the application.

Configuration	Pins	Ports	Clocks	Ram	Logical cores
SDRAM server	20	4 (1-bit), 1 (16-bit)	1	~4.0K	1
Memory address allocator	0	0	0	~0.3K	0

Software version and dependencies

This document pertains to version 3.2.0 of this library. It is known to work on version 14.3.0 of the xTIMEcomposer tools suite, it may work on other versions.

The library does not have any dependencies (i.e. it does not rely on any other libraries).

Related application notes

- I/O Timings for xCORE200[1].

[1] https://www.xmos.com/download/private/I-O-timings-for-xCORE200%281.0%29.pdf

1 Hardware characteristics

The signals from the xCORE required to drive an SDRAM are:

Clock	Clock line, the master clock the SDRAM uses for sampling all the other signals.
DQ_AH	The 16-bit data bus and address bus multiplexed, see below.
WE	Write enable(Inverted).
RAS	The row address strobe(Inverted).
CAS	The coloumn address strobe(Inverted).

Table 1: SDRAM data and signal wires

Because of the multiplexing attention must paid to the physical wiring of the SDRAM to the xCORE.

A typical 256Mb SDRAM requires the following signals:

- CLK - Clock
- CKE - Clock Enable
- CS - Chip Select
- RAS - Row Address Strobe
- CAS - Col Address Strobe
- WE - Write Enable
- DQ[15:0] - Data
- DQM - Data Input/Output Mask
- A[12:0] - Address
- BA[1:0] - Bank Address

The exact count of Address lines and Bank Address line may vary. The examples in this document assume a 256Mb SDRAM device. This library is designed to work with a fixed 16 bit SDRAM data bus, although the API provides data in long words (32 bit).

The dq_ah bus is made up of 16 lines. The DQ bus is mapped directly to dq_ah. The address bus is mapped in order to the lower bits of dq_ah. Finally, the bank address bus is mapped to the higher bits of dq_ah.

Where the Address bus is 13 bits wide and the bank address is 2 bits wide the following setup is in place:

```
dq_ah[15:0] = DQ[15:0]
dq_ah[12:0] = A[12:0]
dq_ah[14:13] = BA[1:0]
```

The number of address bits plus the number of bank address bits must not exceed 16.

The DQM signal(s) is connected to the NOR of WE and CAS. An example of a suitable part is the TI SN74LVC1G02. In the case that the DQM is separated into high and low components then the output from the NOR is connected to both high and low DQM.

This library assumes that CS is pulled low, i.e. the SDRAM is always selected. If control of the CS is needed then it must be done from the client application level. This means that for the duration of the use of the SDRAM, CS must be asserted and when `sdram_server` is shutdown the CS can be deasserted.

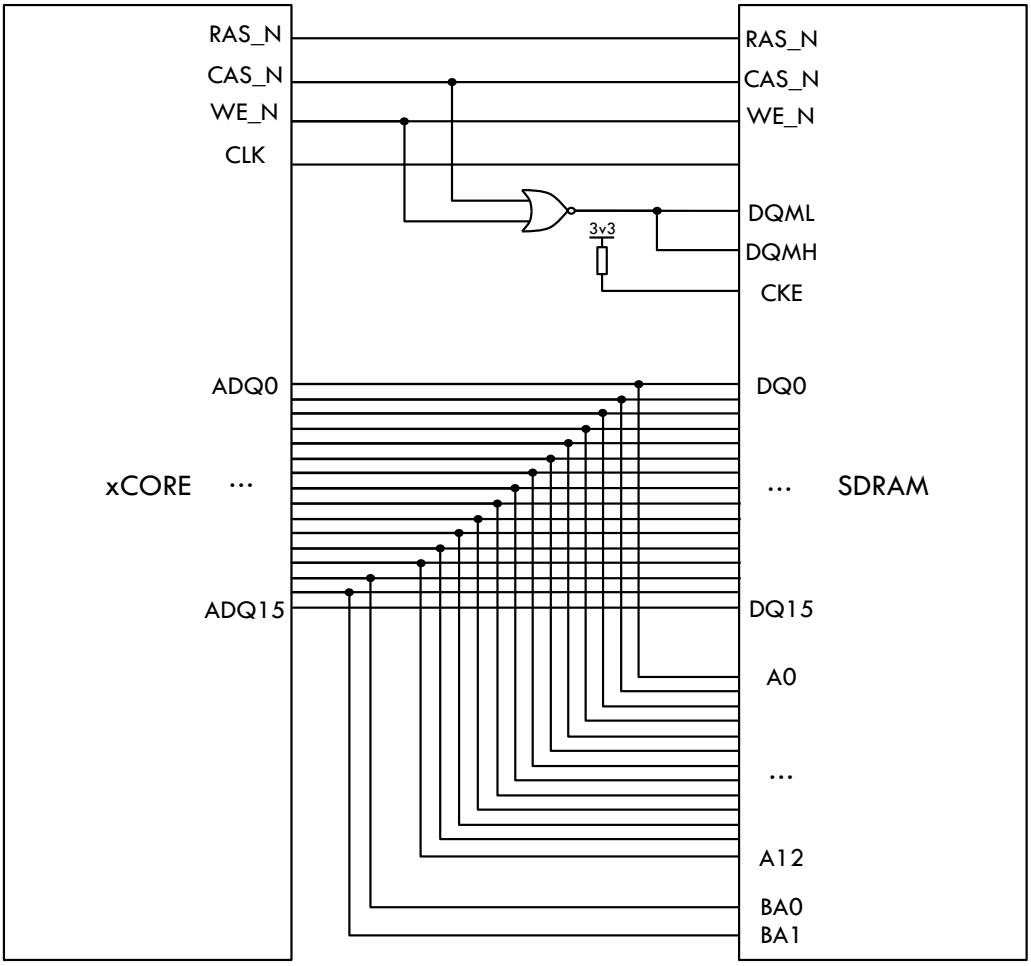

Figure 1: Example connection between xCORE to 256Mb SDRAM

2 SDRAM API

All SDRAM functions can be accessed via the sdram.h header:

```
#include <sdram.h>
```

You also have to add lib_sdram to the USED_MODULES field of your application Makefile.

SDRAM server and client are instantiated as concurrent tasks that run in a par statement. The client (your application) connects to the SDRAM server via a streaming channel.

The SDRAM server must be provided with enough MIPS to keep up with the specified SDRAM clock rate. For example, for 62.5MHz operation, the server core must always see a minimum of 62.5MHz. Typically this will be satisfied if the core clock is 500MHz and 8 cores are used.

The clock rate of the SDRAM server is controlled by the last parameter which is the clock divider. The resulting SDRAM clock, assuming a 500MHz core clock, will be defined as:

```
500 / (2 * n)
```

Typical supported clock rates are as listed in the below table. For each rate, the appropriate port delays are set to maximize the read window. See the code in server.xc for further details.

xCORE clock Divider setting	SDRAM server clock (MHz)
4	62.5
5	50
6	41.66
7	35.71
8	31.25
9	27.78
10	25

Table 2: Supported SDRAM server clock rates

For example, the following code instantiates an SDRAM server running at 62.5MHz and connects an application to it:

```
out buffered port:32   sdram_dq_ah                 = XS1_PORT_16A;
out buffered port:32   sdram_cas                   = XS1_PORT_1B;
out buffered port:32   sdram_ras                   = XS1_PORT_1G;
out buffered port:8    sdram_we                    = XS1_PORT_1C;
out port               sdram_clk                   = XS1_PORT_1F;
clock                  sdram_cb                    = XS1_CLKBLK_1;

int main() {
  streaming chan c_sdram[1];
  par {
      //256Mb SDRAM
      sdram_server(c_sdram, 1,
            sdram_dq_ah,
            sdram_cas,
            sdram_ras,
            sdram_we,
            sdram_clk,
            sdram_cb,
            2,    //CAS latency
            256,  //Row long words
            16,   //Col bits (argument unused by server)
            9,    //Col addr bits
            13,   //Row bits
            2,    //Bank bits
            64,   //Milliseconds refresh
            8192, //Refresh cycles
            4);   //Clock divider
    application(c_sdram[0]);
  }
  return 0;
}
```

Note: The client and SDRAM server must be on the same tile as the line buffers are transferred by moving pointers from one task to another.

The SDRAM library uses movable pointers to pass buffers between the client and the server. This means that if the client passes a buffer in on-chip RAM to the SDRAM server, the client cannot access that buffer while the server is processing the command. To handle this the client sends commands using `sdram_read` and `sdram_write`, both of which take a movable pointer as an argument. To return the pointer to the client the client must call `sdram_complete` which will take back ownership of the pointer when the SDRAM server has finished processing the command.

`sdram_complete` can be selected to allow the client to event on data becoming ready or completing a write.

2.1 Client/Server model

The SDRAM server must be instantiated at the same level as its clients. For example:

```
streaming chan c_sdram[1];
par {
        sdram_server(c_sdram, 1,  ... );
        client_of_the_sdram_server(c_sdram[0]);
}
```

would be the minimum required to correctly setup the SDRAM server and connect it to a client. An example of a multi-client system would be:

```
streaming chan c_sdram[4];
par {
      sdram_server(c_sdram, 4,  ... );
      client_of_the_sdram_server_0(c_sdram[0]);
      client_of_the_sdram_server_1(c_sdram[1]);
      client_of_the_sdram_server_2(c_sdram[2]);
      client_of_the_sdram_server_3(c_sdram[3]);
}
```

2.2 Command buffering

The SDRAM server implements an 8 slot command buffer per client. This means that the client can queue up to 8 commands to the SDRAM server through calls to sdram_read or sdram_write. A successful call to sdram_read or sdram_write will return 0 and issue the command to the command buffer. When the command buffer is full, a call to sdram_read or sdram_write will return 1 and not issue the command. Commands are completed (i.e. a slot is freed) when sdram_complete returns. Commands are processed as in a first in first out ordering.

2.3 Initialization

Each client of the SDRAM server must declare the structure s_sdram_state only once and call sdram_init_state. This does all the required setup for the command buffering. From here on the client can call sdram_read and sdram_write to access the physical memory. For example:

```
s_sdram_state sdram_state;
sdram_init_state(c_server, sdram_state);
```

where c_server is the streaming channel to the sdram_server.

2.4 Safety through the use of movable pointers

The API makes use of movable pointers to aid correct multi-threaded memory handling. sdram_read and sdram_write pass ownership of the memory from the client to the server. The client is no longer able to access the memory. The memory ownership is returned to the client on a call return from sdram_complete. For example:

```
unsigned buffer[N];
unsigned * movable buffer_pointer = buffer;

//buffer_pointer is fully accessible

sdram_read (c_sdram_server, state, address, word_count, move(buffer_pointer));

//during this region the buffer_pointer is null and cannot be read from or written to

sdram_complete(c_sdram_server, state, buffer_pointer);

//now buffer_pointer is accessible again
```

Note that, despite supporting a 16 bit SDRAM data bus width, the native word length of the API is 32 bits. This means that the address provided to the read/wrote functions is the 32b address of the memory. For example, address 0x0001 returns a long word (32 bit) from the 4th to 7th byte location of the SDRAM. Only 32 bit operations are supported; the user should read-modify-write to support modification of smaller word sizes.

During the scope of the movable pointer variable the pointer can point at any memory location, however, at the end of the scope the pointer must point at its original instantiation.

For example the following is acceptable:

```
{
   unsigned buffer_0[N];
   unsigned buffer_1[N];
   unsigned * movable buffer_pointer_0 = buffer_0;
   unsigned * movable buffer_pointer_1 = buffer_1;

   sdram_read (c_sdram_server, state, address, word_count, move(buffer_pointer_0));
   sdram_write (c_sdram_server, state, address, word_count, move(buffer_pointer_1));

   //both buffer_pointer_0 and buffer_pointer_1 are null here

   sdram_complete(c_server, sdram_state, buffer_pointer_0);
   sdram_complete(c_server, sdram_state, buffer_pointer_1);
}
```

but the following is not as the movable pointers are no longer point at the same memory when leaving scope as they were when they were instantiated:

```
{
   unsigned buffer_0[N];
   unsigned buffer_1[N];
   unsigned * movable buffer_pointer_0 = buffer_0;
   unsigned * movable buffer_pointer_1 = buffer_1;

   sdram_read (c_sdram_server, state, address, word_count, move(buffer_pointer_0));
   sdram_write (c_sdram_server, state, address, word_count, move(buffer_pointer_1));

   //both buffer_pointer_0 and buffer_pointer_1 are null here

   sdram_complete(c_server, sdram_state, buffer_pointer_1);   //return to opposite pointer
   sdram_complete(c_server, sdram_state, buffer_pointer_0);
}
```

2.5 Shutdown

The sdram_server may be shutdown, i.e. the thread and all its resources may be freed, with a call to sdram_shutdown.

3 Memory allocator API

The purpose of this library is to allow multiple tasks to share a common memory address space. All of the clients may request a number of bytes from the memory space and will either be allocated a base address to use the requested amount of memory from or will receive an error. All clients of the memory address allocator must be on the same tile.

4 API

Function	sdram_server
Description	The actual SDRAM server providing a software interface plus services to access the SDRAM. • Automatic SDRAM refresh, • Multi-client interface, • Client prioritisation, • Client command buffering, • Automatic multi-line SDRAM access. This provides the software interface to the physical SDRAM. It provides services including:

Type	`void`
	`sdram_server(streaming chanend c_client[client_count],`
	`  const static unsigned client_count,`
	`  out buffered port:32 dq_ah,`
	`  out buffered port:32 cas,`
	`  out buffered port:32 ras,`
	`  out buffered port:8 we,`
	`  out port clk,`
	`  clock cb,`
	`  const static unsigned cas_latency,`
	`  const static unsigned row_words,`
	`  const static unsigned col_bits,`
	`  const static unsigned col_address_bits,`
	`  const static unsigned row_address_bits,`
	`  const static unsigned bank_address_bits,`
	`  const static unsigned refresh_ms,`
	`  const static unsigned refresh_cycles,`
	`  const static unsigned clock_divider)`

Continued on next page

Parameters	`c_client` This is an ordered array of the streaming channels to the clients. It is in client priority order(element 0 being the highest priority). `client_count` The number of clients. `dq_ah` The data and address bus port. `cas` The CAS signal port. `ras` The RAS signal port. `we` The WE signal port. `clk` The SDRAM clock. `cb` Clock block to control the ports. `cas_latency` The CAS latency. `row_words` The number of 32b words in a SDRAM row (half the number of columns). `col_bits` The count of bits for a memory location, normally 16. `col_address_bits` The number of bits in the column address bus. `row_address_bits` The number of bits in the row address bus. `bank_address_bits` The number of bits in the bank address bus. `refresh_ms` The count of milliseconds for a full refresh cycle. `refresh_cycles` The count of refresh instructions per full refresh cycle. `clock_divider` The divider of the system clock to the SDRAM clock.

Function	**sdram_init_state**
Description	This is used to initialise the sdram_state that follows the channel to the SDRAM server. It must only be called once on the s_sdram_state that it is initialising. A client must have only one s_sdram_state that exists for the life time of the use of the SDRAM.

Continued on next page

Type	void sdram_init_state(streaming chanend c_sdram_server, s_sdram_state &sdram_state)
Parameters	c_sdram_server Channel to the SDRAM server. sdram_state State structure.
Returns	None.

Function	**sdram_complete**
Description	This is a blocking call that may be used as a select handler. It returns an array to a movable pointer. It will complete when a command has been completed by the server.
Type	void sdram_complete(streaming chanend c_sdram_server, s_sdram_state &state, unsigned *movable &buffer)

Function	**sdram_write**
Description	Request the SDRAM server to perform a write operation of a number of long (32b) words. This function will place a write command into the SDRAM command buffer if the command buffer is not full. This is a non-blocking call with a return value to indicate the successful issuing of the write to the SDRAM server. 1 for SDRAM command queue is full, write command has not been added.
Type	int sdram_write(streaming chanend c_sdram_server, s_sdram_state &state, unsigned address, unsigned word_count, unsigned *movable buffer)

Continued on next page

Parameters	`c_sdram_server`
	Channel to the SDRAM server.
	
	`state` State structure.
	
	`address` This is a long word address of the location in SDRAM to write from.
	
	`word_count`
	The number of long words to be written to the SDRAM.
	
	`buffer` A movable pointer from which the data to be written to the SDRAM will be read. Note, that the ownership of the pointer will pass to the SDRAM server.

Returns	0 for write command has successfully be added to SDRAM command queue.

Function	**sdram_read**
Description	Request the SDRAM server to perform a read operation of a number of long (32b) words. This function will place a read command into the SDRAM command buffer if the command buffer is not full. This is a non-blocking call with a return value to indicate the successful issuing of the read to the SDRAM server. 1 for SDRAM command queue is full, read command has not been added.

Type	`int`
	`sdram_read(streaming chanend c_sdram_server,`
	`s_sdram_state &state,`
	`unsigned address,`
	`unsigned word_count,`
	`unsigned *movable buffer)`

Parameters	`c_sdram_server`
	Channel to the SDRAM server.
	
	`state` State structure.
	
	`address` This is a long word address of the location in SDRAM to read from.
	
	`word_count`
	The number of long words to be read from the SDRAM.
	
	`buffer` A movable pointer from which the data to be read from the SDRAM will be written. Note, that the ownership of the pointer will pass to the SDRAM server.

Returns	0 for read command has successfully be added to SDRAM command queue.

Function	sdram_shutdown
Description	Terminates the SDRAM server.
Type	void sdram_shutdown(streaming chanend c_sdram_server)
Parameters	c_sdram_server Channel to the SDRAM server.
Returns	None.

Type	memory_address_allocator_i
Description	This interface is used to communication with a memory address allocator. It provides facilities for requesting an address for a region of memory from within a shared memory.
Functions	

Function	request
Description	Request an amount of memory from the common memory space.
Type	e_memory_address_allocator_return_code request(unsigned bytes, unsigned &address)
Parameters	bytes — The address of the slave device to write to. address — A return value for the base address of the memory requested.
Returns	Whether the allocation succeeded.

Function	memory_address_allocator
Description	The distributable server for providing memory address to multiple clients.
Type	[[distributable]] void memory_address_allocator(unsigned client_count, server interface memory_address_allocator_i rx[client_count], unsigned base_address, unsigned memory_size)

Continued on next page

Parameters	`client_count`
	The number of clients.
	
	`rx` Array of the clients wanting to request memory address space.
	
	`base_address`
	Value to be used as the base of memory address.
	
	`memory_size`
	The size of the memory.

APPENDIX A - Known Issues

- XS1 devices can support a maximum of 64 Mb SDRAM (8 MBytes) using a 8b column address. This is a technical limitation due to addressing modes in the XS1 device and cannot be worked around using the current library architecture.
- XS2 (xCORE-200) devices can support a maximum of 256 Mb SDRAM (32 MBytes) using a 9b column address. 512 Mb devices are supportable with some modifications. Please see the following github issue https://github.com/xmos/lib_sdram/issues/20 for details.
- No Application note is provided currently. Please see https://github.com/xmos/lib_sdram/examples for a simple usage example
- The IP assumes a 500MHz core clock. It may be possible to support other core clock frequencies. However, the I/O timing must be re-calculated to populate the read delay constants for the apprioriate clock divider. These may be found in `server.xc`.

APPENDIX B - SDRAM library change log

B.1 3.2.0

- Automatic setting of port delays for a given clock divider
- Fixes to initialization
- Improved read and write latency
- Documentaion and API fixes
- Support for new xCORE-200 Slice Kit in examples

B.2 3.1.0

- Support for 9b row address (128Mb and 256Mb SDRAMs) for xCORE-200 targets
- Fixes incorrect use of READ/WRITE with auto precharge
- Updated example and test to support xCORE-200 by default

B.3 3.0.2

- Update to source code license and copyright

B.4 3.0.1

- Added support for xCORE-200 series

B.5 3.0.0

- Consolidated version, major rework from previous SDRAM components