Matthew Hagerty
Sat, Apr 26, 2014
FPGA, VHDL, SDRAM, DDR, memory, electronics

FPGA VHDL SDRAM Controller

Code on GitHub: https://github.com/dnotq/sdram

Introduction

Modern SDRAM, DDR, DDR2, DDR3, etc. are designed for modern computer systems and require a memory controller. The memory controller will accept memory requests from the CPU, analyze the requests, rearrange them, queue them up, and dispatch them to the SDRAM in the most efficient manner. While fine for a modern computer, a memory controller like that is very complicated for someone who just needs a basic controller to allow using an SDRAM with their simpler FPGA projects.

SDRAM is cheap (around $3.50 for 32MiB at the time of writing) and most FPGA development boards come with some sort of SDRAM on-baord, so it makes sense to utilize this memory when the FPGA's Block-RAM capacity is not sufficient. Faster and easier to use SRAM can also be used if your development board has it, but in my experience most FPGA development boards do not come with SRAM, probably due to the higher cost.

I developed this controller while working on a project that used an 8-bit CPU. What I needed was a simple controller that would allow a constant access time, like SRAM, for any random memory read or write to a 64KiB address space. Not every system design needs 32-bit access or involves streaming massive amounts of data, which are the memory patterns that make SDRAM acceptable when coupled with a modern 23-bit or 64-bit CPU. If you have large data workload requirements then this controller is probably not what you are looking for; there are already plenty of other SDRAM/DDR controllers out there for that kind of memory access.

My projects are more 8-bit or 16-bit oriented systems with completely random access patterns. This kind of memory access is a nightmare for SDRAM, which achieves its high bandwidth by transferring multi-byte data from consecutive addresses. That is all fine and well if you need 64-bit (or wider) data, or are processing chunks of consecutive memory. However, when all you need is a single 8-bit byte, and the next memory access is another 8-bit byte somewhere else in memory, you can never take advantage of SDRAM's features. You will always have the worst case access time (about 70ns) for every memory access.

Design Goals

Target SDRAM: `Winbond W9825G6JH 4M x 4 Banks x 16-bit SDRAM`

The controller needed to satisfy these goals:

1. Have a constant access time for any read or write to any memory location.
2. A refresh cycle cannot hold up a read or write request.
3. Provide an 8-bit data size.

To satisfy the first requirement I determined from datasheet specifications that any read, write, or refresh cycle could be performed in 70ns. That is about 14.2MiB/sec and can easily keep up with an 8-bit or 16-bit CPU running at 10MHz or so. Since most of my designs use CPUs typically running around 1MHz to 4MHz, 70ns gives a lot of time to multiplex RAM access between other parts of the system like video generation, ROM, etc.

A refresh cycle needs to be issued at least once every 7.2us to keep the SDRAM from losing its contents. To keep the refresh cycles from interfering with normal reads or writes, the host system is responsible for issuing refresh cycles. For example, a 1MHz CPU will only make one memory request every 1us, so after each CPU cycle where a memory access is performed a refresh cycle can be issued. One refresh every 1us is well under the minimum requirement of one refresh every 7.2us.

The 8-bit access is facilitated by the SDRAM having upper and lower byte enables. The LS-bit of the address is used to control the UB and LB enables for writes, and controls a byte-select mux for reads. Since the SDRAM has 16-bit words, the memory can support 8-bit and 16-bit host CPUs directly.

SDRAM Terminology and Basic Operation

Datasheets are always written with the assumption that you know what they are talking about, and just about all the information I could find on SDRAM operation also assumed some previous knowledge about how SDRAM works. When you are trying to learn, this can be frustrating to say the least. Below is some information that I could never find explained very well.

SDRAM stores data in banks, and most modern SDRAMs have multiple banks (the SDRAM I'm using has four banks). Each bank is configured as x-rows of y-bits (columns). The number of banks, rows, and bits-per-row (columns) determines the SDRAM's density.

Unlike SRAM and other memory ICs, SDRAM is a "command based" memory similar to SPI-flash, SD-cards, etc.. To read or write data you have to issue commands to the memory, along with the address and data (if writing). Due to the way SDRAM stores data (memory cells use a capacitor to store a bit of data as a charge), reading data from a memory cell destroys the value. Thus, after reading any data, the value has to be written back to the memory cell to restore the data.

To facilitate the restoration of data during normal memory operations, the SDRAM has what are called "sense amplifiers". When memory is addressed, the requested data is destructively read from the memory bank into the sense amplifiers. When the read operation is done, the sense amplifiers restore the data by writing back to the original memory cells. If the memory operation was a write, then the data in the sense amplifiers is replaced with the new values, which are then stored in the memory cells.

To move data from a memory bank to the sense amplifiers for reading or writing is called "activation". Once a bank is active you can issue read and write commands on the retrieved data.

When you are done reading and writing, you must close the bank with a precharge command. This will write the data back to memory and prepare the sense amps to receive data for the next activate command. This can be confusing because a precharge generally happens *after* the activation and read / write operations.

It is this concept of precharge and activate that was the most confusing for me, since most documentation assumes you already understand this process.

Basically, after the SDRAM is initialized the normal command flow would be:

```
Activate -> Read / Write -> Precharge
```

Since all banks are precharged during initialization, they are ready for activation. The precharge at the end of the sequence commits the data back to the memory cells and prepares the sense amps for another activate.

The amount of time a bank can remain active is limited, and also depends on the memory address. You can also have multiple banks active at once which allows greater flexibility and data throughput, however this kind of bank manipulation is better left to an intelligent memory controller and is beyond what I did with my controller.

There is a minimum amount of time from the activate command to when the data is ready to be read or written. There is also a minimum amount of time between activate to precharge, and then from precharge to another activate.

The cumulative time for the SDRAM I am using is 70ns from activate to activate, which becomes the minimum and fixed access time for my controller. A refresh cycle is just like a read or write cycle, except the refresh command is issued instead of a read or write command, and takes the same 70ns:

```
Activate -> Refresh -> Precharge
```

The SDRAM takes care of determining what row to refresh during a refresh cycles, so all you have to worry about is issuing a refresh command at least once every 7.2us.

The amount of data transferred from a bank to the sense amps is limited. To access data outside of the current activated bank requires the bank to be precharged (written back to memory), and a new address specified followed by an activate command.

Theory of Operation

After power-up the SDRAM needs to be initialized before you can start reading or writing data. All SDRAMs will have specific requirements, but they are probably all very similar:

1. Wait 200us with DQM signals high and NOP command issued.
2. Precharge all banks.
3. Issue eight refresh cycles.
4. Set mode register.
5. Issue eight refresh cycles.

The mode register specifies how many words of data will be accessed during a read or write operation, among other options. This is also known as the “burst size” and in the SDRAM I am using it can be 1, 2, 4, or 8 16-bit words. Because I’m keeping things simple my burst size is 1.

The controller’s host interface is pretty simple:

```
-- Host side
clk_100m0_i : in  std_logic;                    -- Master clock
reset_i     : in  std_logic;                    -- Reset, active high
refresh_i   : in  std_logic;                    -- Initiate a refresh cycle, active high
rw_i        : in  std_logic;                    -- Initiate a read or write operation, active high
we_i        : in  std_logic;                    -- Write enable, active low
```

```
addr_i      : in  std_logic_vector(23 downto 0); -- Address from host to SDRAM
data_i      : in  std_logic_vector(15 downto 0); -- Data from host to SDRAM
ub_i        : in  std_logic;                     -- Data upper byte enable, active low
lb_i        : in  std_logic;                     -- Data lower byte enable, active low
ready_o     : out std_logic;                     -- Set to '1' when the memory is ready
done_o      : out std_logic;                     -- Read, write, or refresh, operation is done
data_o      : out std_logic_vector(15 downto 0); -- Data from SDRAM to host
```

The clock input must be 100MHz. After reset the host should wait for ready_o to go high, at which point the SDRAM is ready for use.

~~NOTE: Since SDRAM samples on the rising edge of the clock and most HDL is written to do register-transfer on the rising edge of the clock, this controller does its register transfer on the falling edge of the clock. Therefore the interaction between the rw_i or refresh_i input and the done_o output is 5ns. If you need more time, i.e. a full 10ns to change state, then the external SDRAM should be driven by a clock that is 180-degrees from the clk_100m0_i input and the design changes to use the rising edge.~~

I changed my design to use the rising edge of the clock. There were some later issues with using the falling edge, and all the strange issues were resolved by using the rising edge and buffering the output data. Your use may vary depending on how your SDRAM clock is wired to your FPGA.

Example FSM for using the Controller

```
state_x <= state_r;
rw_i <= '0';
we_i <= '1';
refresh_i <= '0';

case state_r is

when ST_IDLE =>
  if cpu_access_tick = '1' then
    if cpu_we_i = '1' then
      state_x <= ST_READ;
    else
      state_x <= ST_WRITE;
    end if;
  end if;

when ST_READ =>
  if done_o = '0' then
    rw_i <= '1';  -- Hold rw_i high until done_o is raised.
  else
    state_x <= ST_REFRESH;
    data <= data_o;
  end if;

when ST_WRITE =>
  if done_o = '0' then
    rw_i <= '1';
    we_i <= '0';  -- Hold we_i low until done_o is raised.
  else
    state_x <= ST_REFRESH;
  end if;

when ST_REFRESH =>
  if done_o = '0' then
    refresh_i <= '1';
  else
    state_x <= ST_IDLE;
  end if;
```

Full Source Code

```
-- Released under the 3-Clause BSD License:
--
```

```
-- Copyright 2010-2019 Matthew Hagerty (matthew <at> dnotq <dot> io)
--
-- Redistribution and use in source and binary forms, with or without
-- modification, are permitted provided that the following conditions are met:
--
-- 1. Redistributions of source code must retain the above copyright notice,
-- this list of conditions and the following disclaimer.
--
-- 2. Redistributions in binary form must reproduce the above copyright
-- notice, this list of conditions and the following disclaimer in the
-- documentation and/or other materials provided with the distribution.
--
-- 3. Neither the name of the copyright holder nor the names of its
-- contributors may be used to endorse or promote products derived from this
-- software without specific prior written permission.
--
-- THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
-- AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
-- IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
-- ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
-- LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
-- CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
-- SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
-- INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
-- CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
-- ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
-- POSSIBILITY OF SUCH DAMAGE.

-- Simple SDRAM Controller for Winbond W9812G6JH-75
--
-- Matthew Hagerty
--
-- Change Log:
--
-- Dec 14, 2019
--    Changed SDRAM input data setup to ST_RAS1 so it will be correctly
--    registered during ST_RAS2.
--    Comment cleanup.
--
-- Jan 28, 2016
--    Changed to use positive clock edge.
--    Buffered output (read) data, sampled during RAS2.
--    Removed unused signals for features that were not implemented.
--    Changed tabs to space.
--
-- March 19, 2014
--    Initial implementation.


library IEEE, UNISIM;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;
use IEEE.math_real.all;

entity sdram_simple is
   port(
      -- Host side
      clk_100m0_i    : in std_logic;            -- Master clock
      reset_i        : in std_logic := '0';     -- Reset, active high
      refresh_i      : in std_logic := '0';     -- Initiate a refresh cycle, active high
      rw_i           : in std_logic := '0';     -- Initiate a read or write operation, active high
      we_i           : in std_logic := '0';     -- Write enable, active low
      addr_i         : in std_logic_vector(23 downto 0);   -- Address from host to SDRAM
      data_i         : in std_logic_vector(15 downto 0);   -- Data from host to SDRAM
      ub_i           : in std_logic;            -- Data upper byte enable, active low
      lb_i           : in std_logic;            -- Data lower byte enable, active low
      ready_o        : out std_logic := '0';    -- Set to '1' when the memory is ready
      done_o         : out std_logic := '0';    -- Read, write, or refresh, operation is done
      data_o         : out std_logic_vector(15 downto 0);   -- Data from SDRAM to host

      -- SDRAM side
      sdCke_o        : out std_logic;           -- Clock-enable to SDRAM
      sdCe_bo        : out std_logic;           -- Chip-select to SDRAM
      sdRas_bo       : out std_logic;           -- SDRAM row address strobe
      sdCas_bo       : out std_logic;           -- SDRAM column address strobe
      sdWe_bo        : out std_logic;           -- SDRAM write enable
```

```
      sdBs_o         : out std_logic_vector( 1 downto 0);   -- SDRAM bank address
      sdAddr_o       : out std_logic_vector(12 downto 0);   -- SDRAM row/column address
      sdData_io      : inout std_logic_vector(15 downto 0); -- Data to/from SDRAM
      sdDqmh_o       : out std_logic;           -- Enable upper-byte of SDRAM databus if true
      sdDqml_o       : out std_logic            -- Enable lower-byte of SDRAM databus if true
   );
end entity;

architecture rtl of sdram_simple is


   -- SDRAM controller states.
   type fsm_state_type is (
   ST_INIT_WAIT, ST_INIT_PRECHARGE, ST_INIT_REFRESH1, ST_INIT_MODE, ST_INIT_REFRESH2,
   ST_IDLE, ST_REFRESH, ST_ACTIVATE, ST_RCD, ST_RW, ST_RAS1, ST_RAS2, ST_PRECHARGE);
   signal state_r, state_x : fsm_state_type := ST_INIT_WAIT;


   -- SDRAM mode register data sent on the address bus.
   --
   -- | A12-A10 |    A9    | A8  A7 | A6 A5 A4 |    A3   | A2 A1 A0 |
   -- | reserved| wr burst |reserved| CAS Ltncy|addr mode| burst len|
   --   0  0  0      0       0   0    0  1  0       0      0  0  0
   constant MODE_REG : std_logic_vector(12 downto 0) := "000" & "0" & "00" & "010" & "0" & "000";

   -- SDRAM commands combine SDRAM inputs: cs, ras, cas, we.
   subtype cmd_type is unsigned(3 downto 0);
   constant CMD_ACTIVATE         : cmd_type := "0011";
   constant CMD_PRECHARGE        : cmd_type := "0010";
   constant CMD_WRITE            : cmd_type := "0100";
   constant CMD_READ             : cmd_type := "0101";
   constant CMD_MODE             : cmd_type := "0000";
   constant CMD_NOP              : cmd_type := "0111";
   constant CMD_REFRESH          : cmd_type := "0001";

   signal cmd_r                  : cmd_type;
   signal cmd_x                  : cmd_type;

   signal bank_s                 : std_logic_vector( 1 downto 0);
   signal row_s                  : std_logic_vector(12 downto 0);
   signal col_s                  : std_logic_vector( 8 downto 0);
   signal addr_r                 : std_logic_vector(12 downto 0);
   signal addr_x                 : std_logic_vector(12 downto 0);    -- SDRAM row/column address.
   signal sd_dout_r              : std_logic_vector(15 downto 0);
   signal sd_dout_x              : std_logic_vector(15 downto 0);
   signal sd_busdir_r            : std_logic;
   signal sd_busdir_x            : std_logic;

   signal timer_r, timer_x       : natural range 0 to 20000 := 0;
   signal refcnt_r, refcnt_x     : natural range 0 to 8 := 0;

   signal bank_r, bank_x         : std_logic_vector(1 downto 0);
   signal cke_r, cke_x           : std_logic;
   signal sd_dqmu_r, sd_dqmu_x   : std_logic;
   signal sd_dqml_r, sd_dqml_x   : std_logic;
   signal ready_r, ready_x       : std_logic;

   -- Data buffer for SDRAM to Host.
   signal buf_dout_r, buf_dout_x : std_logic_vector(15 downto 0);

begin

   -- All signals to SDRAM buffered.

   (sdCe_bo, sdRas_bo, sdCas_bo, sdWe_bo) <= cmd_r;   -- SDRAM operation control bits
   sdCke_o     <= cke_r;      -- SDRAM clock enable
   sdBs_o      <= bank_r;     -- SDRAM bank address
   sdAddr_o    <= addr_r;     -- SDRAM address
   sdData_io   <= sd_dout_r when sd_busdir_r = '1' else (others => 'Z');   -- SDRAM data bus.
   sdDqmh_o    <= sd_dqmu_r;  -- SDRAM high data byte enable, active low
   sdDqml_o    <= sd_dqml_r;  -- SDRAM low date byte enable, active low

   -- Signals back to host.
   ready_o <= ready_r;
   data_o <= buf_dout_r;

   -- 23  22  | 21 20 19 18 17 16 15 14 13 12 11 10 09 | 08 07 06 05 04 03 02 01 00 |
```

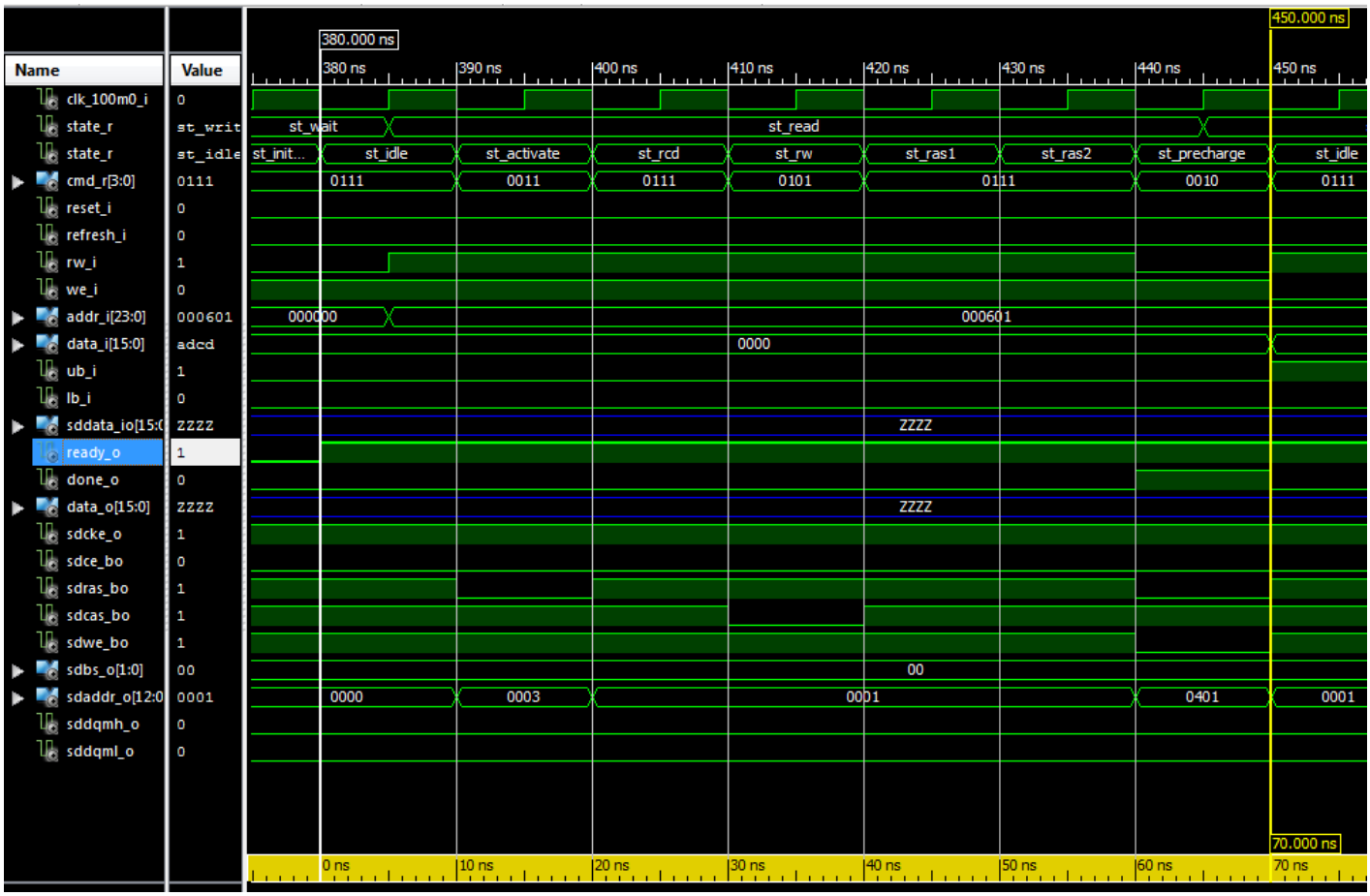
```
   -- BS0 BS1 |        ROW (A12-A0)  8192 rows         |   COL (A8-A0)  512 cols    |
   bank_s <= addr_i(23 downto 22);
   row_s <= addr_i(21 downto 9);
   col_s <= addr_i(8 downto 0);


   process (
   state_r, timer_r, refcnt_r, cke_r, addr_r, sd_dout_r, sd_busdir_r, sd_dqmu_r, sd_dqml_r, ready_r,
   bank_s, row_s, col_s,
   rw_i, refresh_i, addr_i, data_i, we_i, ub_i, lb_i,
   buf_dout_r, sdData_io)
   begin

      state_x     <= state_r;       -- Stay in the same state unless changed.
      timer_x     <= timer_r;       -- Hold the cycle timer by default.
      refcnt_x    <= refcnt_r;      -- Hold the refresh timer by default.
      cke_x       <= cke_r;         -- Stay in the same clock mode unless changed.
      cmd_x       <= CMD_NOP;       -- Default to NOP unless changed.
      bank_x      <= bank_r;        -- Register the SDRAM bank.
      addr_x      <= addr_r;        -- Register the SDRAM address.
      sd_dout_x   <= sd_dout_r;     -- Register the SDRAM write data.
      sd_busdir_x <= sd_busdir_r;   -- Register the SDRAM bus tristate control.
      sd_dqmu_x   <= sd_dqmu_r;
      sd_dqml_x   <= sd_dqml_r;
      buf_dout_x  <= buf_dout_r;    -- SDRAM to host data buffer.

      ready_x     <= ready_r;       -- Always ready unless performing initialization.
      done_o      <= '0';           -- Done tick, single cycle.

      if timer_r /= 0 then
         timer_x <= timer_r - 1;
      else

         cke_x       <= '1';
         bank_x      <= bank_s;
         -- A10 low for rd/wr commands to suppress auto-precharge.
         addr_x      <= "0000" & col_s;
         sd_dqmu_x   <= '0';
         sd_dqml_x   <= '0';

         case state_r is

         when ST_INIT_WAIT =>

            -- 1. Wait 200us with DQM signals high, cmd NOP.
            -- 2. Precharge all banks.
            -- 3. Eight refresh cycles.
            -- 4. Set mode register.
            -- 5. Eight refresh cycles.

            state_x <= ST_INIT_PRECHARGE;
            timer_x <= 20000;           -- Wait 200us (20,000 cycles).
--          timer_x <= 2;               -- for simulation
            sd_dqmu_x <= '1';
            sd_dqml_x <= '1';

         when ST_INIT_PRECHARGE =>

            state_x <= ST_INIT_REFRESH1;
            refcnt_x <= 8;              -- Do 8 refresh cycles in the next state.
--          refcnt_x <= 2;              -- for simulation
            cmd_x <= CMD_PRECHARGE;
            timer_x <= 2;               -- Wait 2 cycles plus state overhead for 20ns Trp.
            bank_x <= "00";
            addr_x(10) <= '1';          -- Precharge all banks.

         when ST_INIT_REFRESH1 =>

            if refcnt_r = 0 then
               state_x <= ST_INIT_MODE;
            else
               refcnt_x <= refcnt_r - 1;
               cmd_x <= CMD_REFRESH;
               timer_x <= 7;            -- Wait 7 cycles plus state overhead for 70ns refresh.
            end if;

         when ST_INIT_MODE =>
```

```
            state_x <= ST_INIT_REFRESH2;
            refcnt_x <= 8;             -- Do 8 refresh cycles in the next state.
--          refcnt_x <= 2;             -- for simulation
            bank_x <= "00";
            addr_x <= MODE_REG;
            cmd_x <= CMD_MODE;
            timer_x <= 2;              -- Trsc == 2 cycles after issuing MODE command.

         when ST_INIT_REFRESH2 =>

            if refcnt_r = 0 then
               state_x <= ST_IDLE;
               ready_x <= '1';
            else
               refcnt_x <= refcnt_r - 1;
               cmd_x <= CMD_REFRESH;
               timer_x <= 7;           -- Wait 7 cycles plus state overhead for 70ns refresh.
            end if;

      --
      -- Normal Operation
      --
         -- Trc  - 70ns - Activate to activate command.
         -- Trcd - 20ns - Activate to read/write command.
         -- Tras - 50ns - Activate to precharge command.
         -- Trp  - 20ns - Precharge to activate command.
         -- TCas - 2clk - Read/write to data out.
         --
         --         |<-----------       Trc      ------------>|
         --         |<----------- Tras ---------->|
         --         |<- Trcd  ->|<- TCas  ->|     |<-  Trp  ->|
         --  T0__   T1__   T2__   T3__   T4__   T5__   T6__   T0__   T1__
         -- __/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__
         -- IDLE  ACTVT  NOP  RD/WR  NOP   NOP  PRECG IDLE  ACTVT
         --     --<Row>---------------------------------------<Row>--
         --                ---<Col>---
         --                ---<A10>-------------<A10>---
         --                                  ---<Bank>---
         --                ---<DQM>---
         --                ---<Din>---
         --                                  ---<Dout>---
         --   ---<Refsh>-----------------------------------<Refsh>---
         --
         -- A10 during rd/wr : 0 = disable auto-precharge, 1 = enable auto-precharge.
         -- A10 during precharge: 0 = single bank, 1 = all banks.

         -- Next State vs Current State Guide
         --
         --  T0__   T1__   T2__   T3__   T4__   T5__   T6__   T0__   T1__   T2__
         -- __/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__
         -- IDLE  ACTVT  NOP  RD/WR  NOP   NOP  PRECG IDLE  ACTVT
         --       IDLE  ACTVT  NOP  RD/WR  NOP   NOP  PRECG IDLE  ACTVT


         when ST_IDLE =>
            -- 60ns since activate when coming from PRECHARGE state.
            -- 10ns since PRECHARGE.  Trp == 20ns min.
            if rw_i = '1' then
               state_x <= ST_ACTIVATE;
               cmd_x <= CMD_ACTIVATE;
               addr_x <= row_s;        -- Set bank select and row on activate command.
            elsif refresh_i = '1' then
               state_x <= ST_REFRESH;
               cmd_x <= CMD_REFRESH;
               timer_x <= 7;           -- Wait 7 cycles plus state overhead for 70ns refresh.
            end if;

         when ST_REFRESH =>

            state_x <= ST_IDLE;
            done_o <= '1';

         when ST_ACTIVATE =>
            -- Trc (Active to Active Command Period) is 65ns min.
            -- 70ns since activate when coming from PRECHARGE -> IDLE states.
            -- 20ns since PRECHARGE.
```

```
         -- ACTIVATE command is presented to the SDRAM.  The command out of this
         -- state will be NOP for one cycle.
         state_x <= ST_RCD;
         sd_dout_x <= data_i;       -- Register any write data, even if not used.

      when ST_RCD =>
         -- 10ns since activate.
         -- Trcd == 20ns min.  The clock is 10ns, so the requirement is satisfied by this state.
         -- READ or WRITE command will be active in the next cycle.
         state_x <= ST_RW;

         if we_i = '0' then
            cmd_x <= CMD_WRITE;
            sd_busdir_x <= '1';     -- The SDRAM latches the input data with the command.
            sd_dqmu_x <= ub_i;
            sd_dqml_x <= lb_i;
         else
            cmd_x <= CMD_READ;
         end if;

      when ST_RW =>
         -- 20ns since activate.
         -- READ or WRITE command presented to SDRAM.
         state_x <= ST_RAS1;
         sd_busdir_x <= '0';

      when ST_RAS1 =>
         -- 30ns since activate.
         -- Data from the SDRAM will be registered on the next clock.
         state_x <= ST_RAS2;
         buf_dout_x <= sdData_io;

      when ST_RAS2 =>
         -- 40ns since activate.
         -- Tras (Active to precharge Command Period) 45ns min.
         -- PRECHARGE command will be active in the next cycle.
         state_x <= ST_PRECHARGE;
         cmd_x <= CMD_PRECHARGE;
         addr_x(10) <= '1';         -- Precharge all banks.

      when ST_PRECHARGE =>
         -- 50ns since activate.
         -- PRECHARGE presented to SDRAM.
         state_x <= ST_IDLE;
         done_o <= '1';             -- Read data is ready and should be latched by the host.
         timer_x <= 1;              -- Buffer to make sure host takes down memory request before going IDLE.

      end case;
   end if;
end process;

process (clk_100m0_i)
begin
   if rising_edge(clk_100m0_i) then
   if reset_i = '1' then
      state_r  <= ST_INIT_WAIT;
      timer_r  <= 0;
      cmd_r    <= CMD_NOP;
      cke_r    <= '0';
      ready_r  <= '0';
   else
      state_r     <= state_x;
      timer_r     <= timer_x;
      refcnt_r    <= refcnt_x;
      cke_r       <= cke_x;         -- CKE to SDRAM.
      cmd_r       <= cmd_x;         -- Command to SDRAM.
      bank_r      <= bank_x;        -- Bank to SDRAM.
      addr_r      <= addr_x;        -- Address to SDRAM.
      sd_dout_r   <= sd_dout_x;     -- Data to SDRAM.
      sd_busdir_r <= sd_busdir_x;   -- SDRAM bus direction.
      sd_dqmu_r   <= sd_dqmu_x;     -- Upper byte enable to SDRAM.
      sd_dqml_r   <= sd_dqml_x;     -- Lower byte enable to SDRAM.
      ready_r     <= ready_x;
      buf_dout_r  <= buf_dout_x;

   end if;
   end if;
```

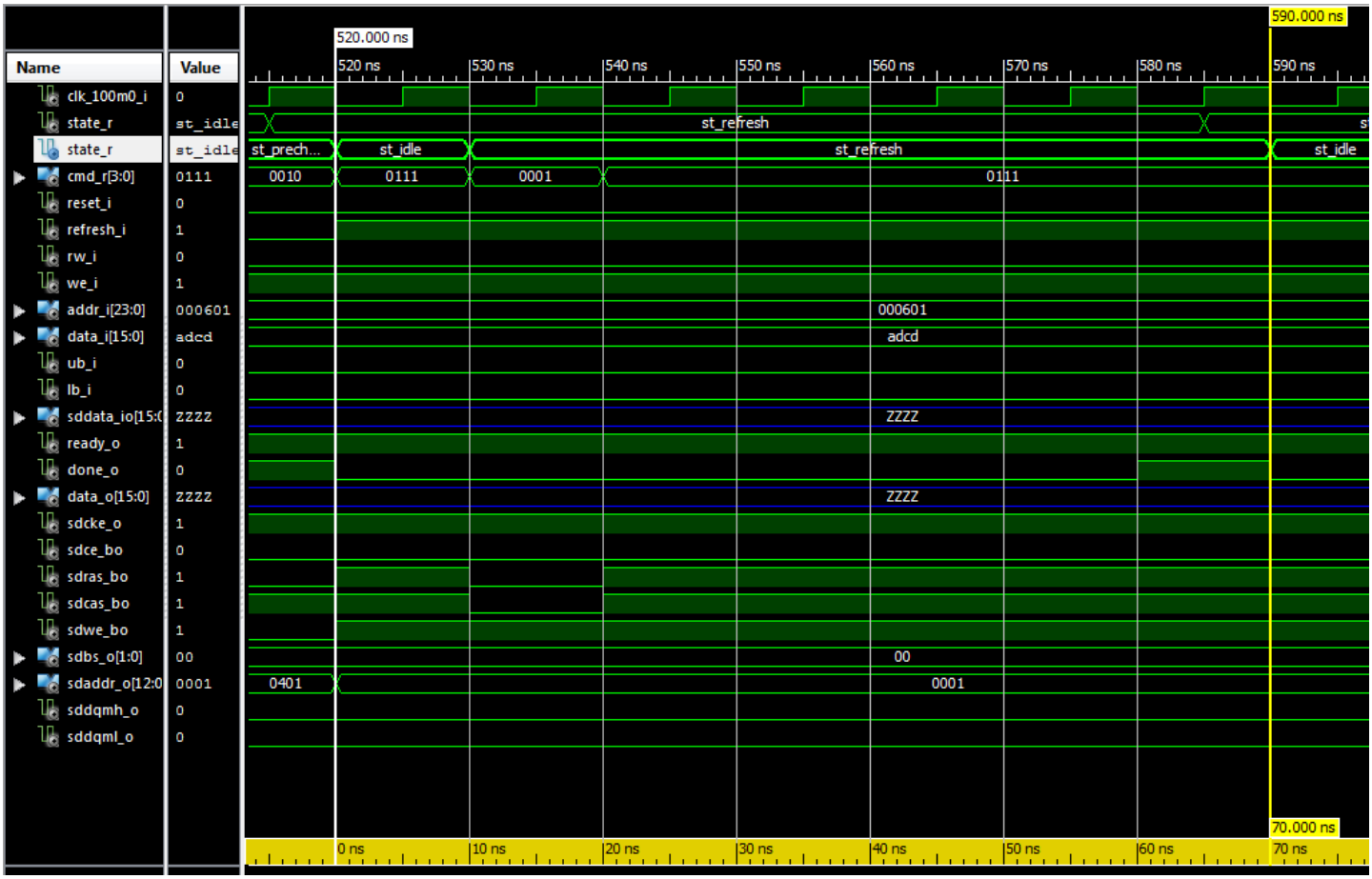
```
    end process;

end architecture;
```

Read Cycle



Write Cycle

Refresh Cycle



Testbench

Here is the simulation testbench HDL I used. This design has also been synthesized and proven in hardware as the main RAM and ROM in an 8-bit CPU SoC.

```
-- Released under the 3-Clause BSD License:
--
-- Copyright 2010-2019 Matthew Hagerty (matthew <at> dnotq <dot> io)
--
-- Redistribution and use in source and binary forms, with or without
-- modification, are permitted provided that the following conditions are met:
--
-- 1. Redistributions of source code must retain the above copyright notice,
-- this list of conditions and the following disclaimer.
--
-- 2. Redistributions in binary form must reproduce the above copyright
-- notice, this list of conditions and the following disclaimer in the
-- documentation and/or other materials provided with the distribution.
--
-- 3. Neither the name of the copyright holder nor the names of its
-- contributors may be used to endorse or promote products derived from this
-- software without specific prior written permission.
--
-- THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
-- AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
-- IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
-- ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
-- LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
-- CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
-- SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
-- INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
-- CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
-- ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
-- POSSIBILITY OF SUCH DAMAGE.

-- Matthew Hagerty
```

```
-- March 18, 2014
--
-- Testbench for Simple SDRAM Controller for Winbond W9812G6JH-75

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY tb_sdram_simple IS
END tb_sdram_simple;

ARCHITECTURE behavior OF tb_sdram_simple IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT sdram_simple
    PORT(
         clk_100m0_i : IN  std_logic;
         reset_i : IN  std_logic;
         refresh_i : IN  std_logic;
         rw_i : IN  std_logic;
         we_i : IN  std_logic;
         addr_i : IN  std_logic_vector(23 downto 0);
         data_i : IN  std_logic_vector(15 downto 0);
         ub_i : IN  std_logic;
         lb_i : IN  std_logic;
         ready_o : OUT  std_logic;
         done_o : OUT  std_logic;
         data_o : OUT  std_logic_vector(15 downto 0);
         sdCke_o : OUT  std_logic;
         sdCe_bo : OUT  std_logic;
         sdRas_bo : OUT  std_logic;
         sdCas_bo : OUT  std_logic;
         sdWe_bo : OUT  std_logic;
         sdBs_o : OUT  std_logic_vector(1 downto 0);
         sdAddr_o : OUT  std_logic_vector(12 downto 0);
         sdData_io : INOUT  std_logic_vector(15 downto 0);
         sdDqmh_o : OUT  std_logic;
         sdDqml_o : OUT  std_logic
        );
    END COMPONENT;


   --Inputs
   signal clk_100m0_i : std_logic := '0';
   signal reset_i : std_logic := '0';
   signal refresh_i : std_logic := '0';
   signal rw_i : std_logic := '0';
   signal we_i : std_logic := '0';
   signal addr_i : std_logic_vector(23 downto 0) := (others => '0');
   signal data_i : std_logic_vector(15 downto 0) := (others => '0');
   signal ub_i : std_logic := '0';
   signal lb_i : std_logic := '0';

        --BiDirs
   signal sdData_io : std_logic_vector(15 downto 0);

        --Outputs
   signal ready_o : std_logic;
   signal done_o : std_logic;
   signal data_o : std_logic_vector(15 downto 0);
   signal sdCke_o : std_logic;
   signal sdCe_bo : std_logic;
   signal sdRas_bo : std_logic;
   signal sdCas_bo : std_logic;
   signal sdWe_bo : std_logic;
   signal sdBs_o : std_logic_vector(1 downto 0);
   signal sdAddr_o : std_logic_vector(12 downto 0);
   signal sdDqmh_o : std_logic;
   signal sdDqml_o : std_logic;

   -- Clock period definitions
   constant clk_100m0_i_period : time := 10 ns;

        type state_type is (ST_WAIT, ST_IDLE, ST_READ, ST_WRITE, ST_REFRESH);
        signal state_r, state_x : state_type := ST_WAIT;
BEGIN
```

```
	-- Instantiate the Unit Under Test (UUT)
uut: sdram_simple PORT MAP (
       clk_100m0_i => clk_100m0_i,
       reset_i => reset_i,
       refresh_i => refresh_i,
       rw_i => rw_i,
       we_i => we_i,
       addr_i => addr_i,
       data_i => data_i,
       ub_i => ub_i,
       lb_i => lb_i,
       ready_o => ready_o,
       done_o => done_o,
       data_o => data_o,
       sdCke_o => sdCke_o,
       sdCe_bo => sdCe_bo,
       sdRas_bo => sdRas_bo,
       sdCas_bo => sdCas_bo,
       sdWe_bo => sdWe_bo,
       sdBs_o => sdBs_o,
       sdAddr_o => sdAddr_o,
       sdData_io => sdData_io,
       sdDqmh_o => sdDqmh_o,
       sdDqml_o => sdDqml_o
     );

-- Clock process definitions
clk_100m0_i_process :process
begin
		clk_100m0_i <= '0';
		wait for clk_100m0_i_period/2;
		clk_100m0_i <= '1';
		wait for clk_100m0_i_period/2;
end process;

     process (clk_100m0_i)
     begin
		if rising_edge(clk_100m0_i) then
			state_r <= state_x;
		end if;
     end process;

     process ( state_r, ready_o, done_o )
     begin

		state_x <= state_r;
		rw_i <= '0';
		we_i <= '1';
		ub_i <= '0';
		lb_i <= '0';


		case ( state_r ) is

		when ST_WAIT =>
			if  ready_o = '1' then
				state_x <= ST_READ;
			end if;

		when ST_IDLE =>
			state_x <= ST_IDLE;

		when ST_READ =>
			if done_o = '0' then
				rw_i <= '1';
				addr_i <= "00000000000001100000001";
			else
				state_x <= ST_WRITE;
			end if;

		when ST_WRITE =>
			if done_o = '0' then
				rw_i <= '1';
				we_i <= '0';
				addr_i <= "00000000000001100000001";
				data_i <= X"ADCD";
				ub_i <= '1';
```

```
                        lb_i <= '0';
                    else
                        state_x <= ST_REFRESH;
                    end if;

                when ST_REFRESH =>
                    if done_o = '0' then
                        refresh_i <= '1';
                    else
                        state_x <= ST_IDLE;
                    end if;
                end case;

    end process;

    -- Stimulus process
    stim_proc: process
    begin
       -- hold reset state for 100 ns.
                reset_i <= '1';
       wait for 20 ns;
                reset_i <= '0';
                wait;
         end process;


END;
```