МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение

Национальный исследовательский ядерный университет «МИФИ»

Институт Интеллектуальных Кибернетических Систем

Отчёт

по курсовой работе на тему:

Разработка конечного автомата для обмена с внешней памятью SDRAM и выполнения присвоений выходных сигналов в зависимости от текущего состояния конечного автомата

Дисциплина: Схемотехника цифровых устройств

Подготовили студенты группы С23-501

Горбатенко И.А.

Серебрякова Д.Р.

Москва, 2026

Содержание

Введение

В рамках курсового проекта была разработана модель контроллера SDRAM, упрощённая структурная схема которого представлена на Рисунке 1.

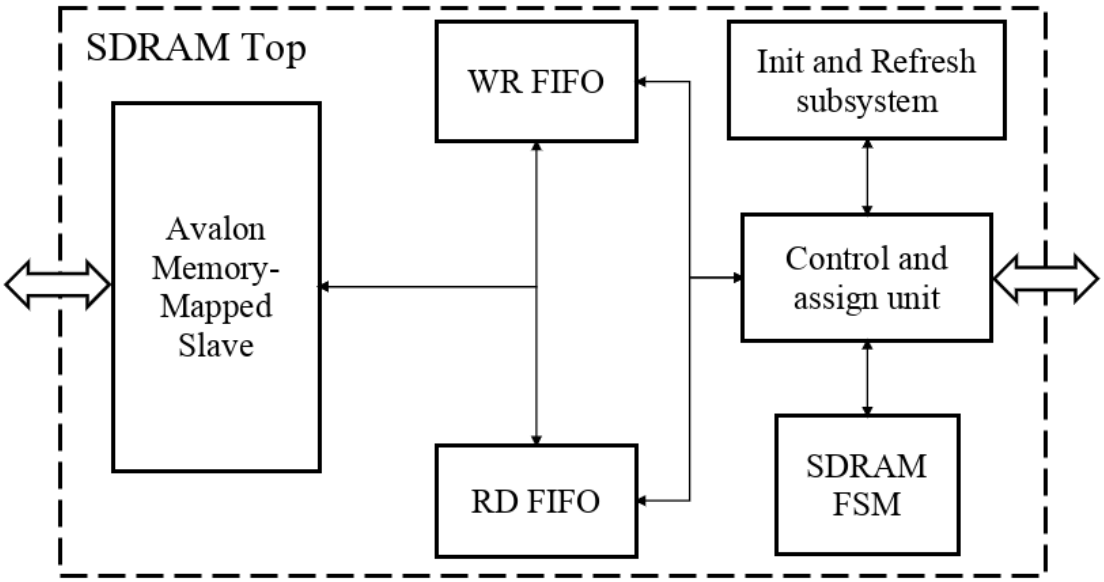

Рисунок 1 — Упрощённая структурная схема контроллера SDRAM

Требования к проекту:

1) Ширина шины данных для интерфейса Avalon Memory-Mapped фиксирована и составляет 64 бита;
2) Отдельная буферизация для чтения и записи;
3) Параметры SDRAM должны задаваться через generics: Burst length, ширина данных для внешней памяти, режим записи, время задержки CAS (CAS latency).
4) Режим адресации для SDRAM – последовательный (Sequential Mode);
5) Автоматический refresh и precharge в фоновом режиме.

Тестирование проводилось на ПЛИС 10CL025YU256C8G и SDRAM W9864G6JT-6.

В данной части работы был разработан конечный автомат для обмена с внешней памятью SDRAM и выполнения присвоений выходных сигналов в зависимости от текущего состояния конечного автомата.

Описание работы устройства

Модуль SDRAM FSM должен корректно взаимодействовать с модулем Init and Refresh subsystem и с Avalon Memory-Mapped Slave посредством FIFO.

Для реализации блоков Init and Refresh subsystem и Control and assign unit в структуре контроллера SDRAM было принято решение использовать мультиплексор (Arbiter). Необходимость его применения обусловлена тем, что как конечный автомат FSM (Control and assign unit), так и подсистема инициализации, обновления и предварительной зарядки (Init and Refresh subsystem) формируют управляющие сигналы для микросхемы SDRAM, при этом в каждый момент времени управление памятью может осуществляться только из одного источника. Схема соединения конечного автомата, подсистемы инициализации, обновления и предварительной зарядки, арбитра контроллера SDRAM представлена на Рисунке 2.

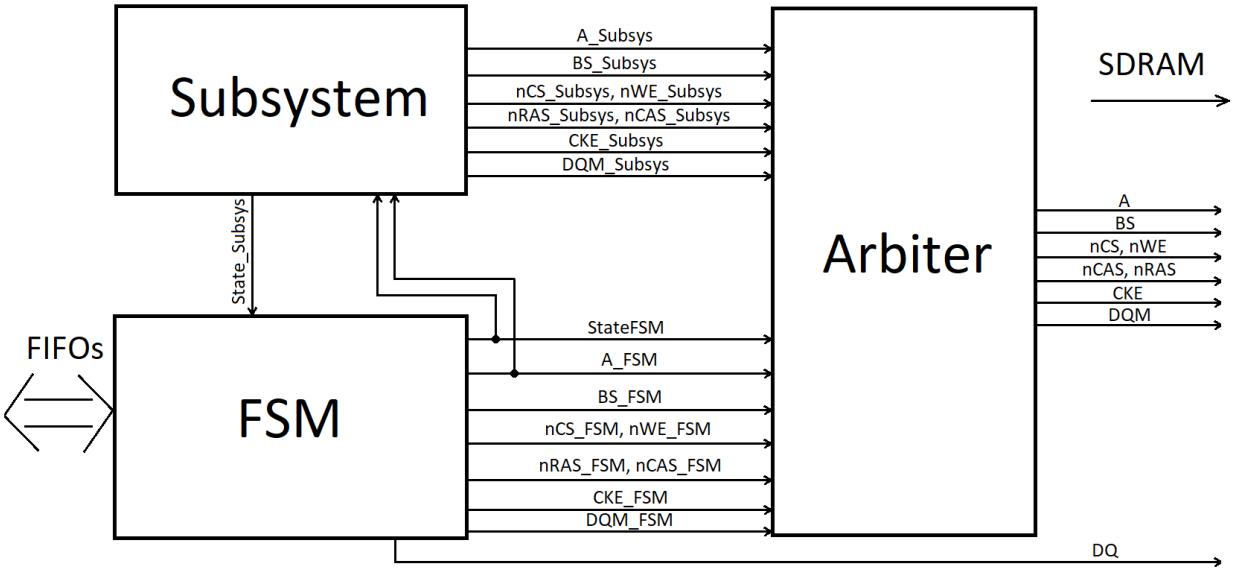

Рисунок 2 — Схема соединения арбитра, подсистемы и автомата контроллера SDRAM

Подсистема управления инициализацией, обновления данных и предварительной зарядкой является частью контроллера SDRAM и выполняет следующие задачи:

- Поддержка обычного и пакетного режима передачи данных;
- Соблюдение временных параметров микросхемы: задержку между активацией строки и операцией чтения/записи (tRCD), задержку выдачи данных при чтении (CAS latency), задержку между предзарядкой и следующей активацией (tRP);

- Корректная передача управление шиной SDRAM через арбитр и завершение текущей транзакции перед передачей управления подсистеме инициализации и refresh.

Условное графическое изображение разработанного устройства приведено на Рисунке 3.

Картинка

Рисунок 3 — УГО FSM

Состояния конечного автомата

Диаграмма состояний разработанного конечного автомата представлена на Рисунке 4 в виде графа состояний с условиями их перехода.

Картинка

Рисунок 4. Диаграмма состояний конечного автомата

Имена состояний и их значение

Назначение и характеристики сигналов

В Таблице 1 приведено описание входных и выходных сигналов разработанного конечного автомата.

Таблица 1. Входные и выходные сигналы конечного автомата

Название сигнала	Вход/ выход	Разрядность, бит	Описание
Clk	in	1	Тактовый сигнал
nRst	in	1	Сигнал асинхронного сброса

дописать

Дополнительные модули устройства

Для реализации контроллера SDRAM потребовалось разработать регистры для состояния записи в память из FIFO и для состояния чтения из памяти в FIFO.

1. Регистр для записи

Для операции записи из FIFO был реализован сдвиговый регистр, УГО которого приведено на Рисунке 5.

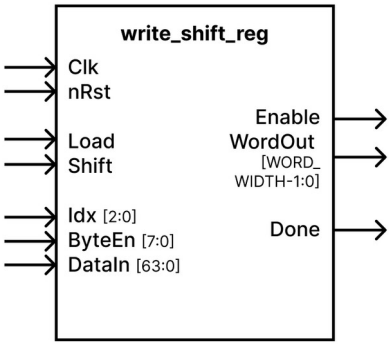

Рисунок 5. УГО сдвигового регистра для записи

В Таблице 2 приведено описание входных и выходных сигналов разработанного регистра.

Таблица 2. Входные и выходные сигналы регистра для записи

Название сигнала	Вход/ выход	Разрядность, бит	Описание
Clk	in	1	Тактовый сигнал
nRst	in	1	Сигнал асинхронного сброса
Load	in	1	Сигнал разрешения загрузки. «1» - если регистр пуст и его можно заполнить, «0» - в противном случае
Shift	in	1	Сигнал сдвига данных. «1» - если на выходе нужно получить сдвиг, «0» - в противном случае
Idx	in	3	Номер, по которому производится расчёт стартового индекса данных для занесения в регистр
ByteEn	in	8	Байт разрешения записи 8-битных данных, соответствующих биту из него

DataIn	in	64	Шина входных данных для сдвига
Enable	out	1	Сигнал разрешения записи в память выходной порции данных
WordOut	out	world_width	Выходная порция данных
Done	out	1	Сигнал завершения выдачи данных. «1» - если регистр пуст, «0» - в противном случае

Принцип работы регистра заключается в параллельном занесении burst*world_width бит из входной 64-битной шины данных в него и последовательном сдвиге в количестве world_width бит за один такт. При этом с помощью дженерика задаются значения burst и world_width. Все возможные их комбинации приведены в Таблице 3 и рассмотрены для работы регистра.

Таблица 3. Комбинации значений burst и world_width и соответствующие им параметры

burst	world_width	Максимальное количество битов в регистре (FRAG_BITS)	Количество записей в регистр из 64-битной шины
1	8	8	8
1	16	16	4
1	32	32	2
1	64	64	1
2	8	16	4
2	16	32	2
2	32	64	1
2	64	128	1
4	8	32	2
4	16	64	1
4	32	128	1
4	64	256	1
8	8	64	1
8	16	128	1
8	32	256	1
8	64	512	1

На Рисунке 6 приведена схема работы сдвигового регистра и его взаимодействие с входными и выходными данными в случае burst = 4 и world_width = 16.

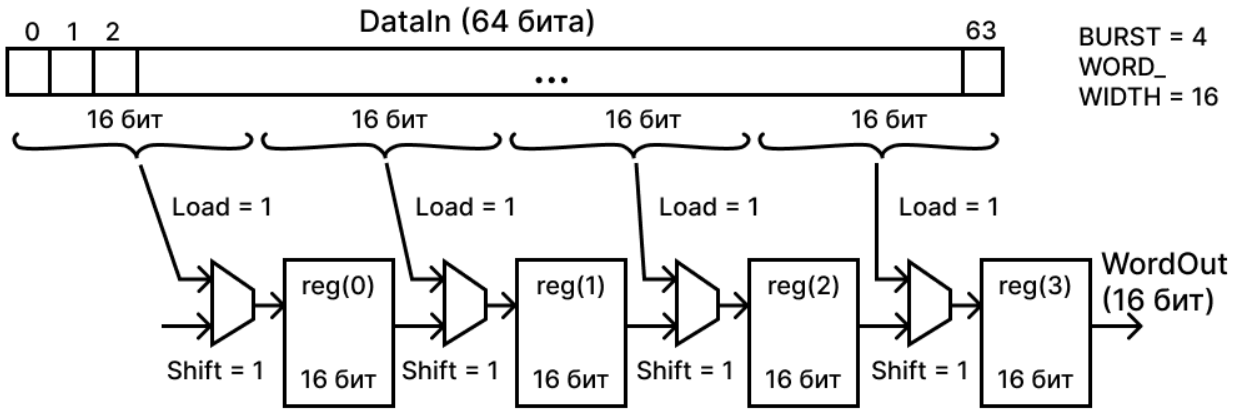

Рисунок 6. Схема работы сдвигового регистра для записи при burst = 4 и world_width = 16

На вход регистра подаётся 64-битная шина данных, которая загружается в регистр за один такт путём подачи единицы на вход Load. После этого происходит сдвиг порции по world_width бит при выставлении сигнала Shift единицей и выдача этих выходных данных за один такт. Одновременная подача единицы на Load и Shift невозможна, потому что Shift будет в приоритете из условия, приведённого в коде разработанного регистра. Возможны несколько ситуаций для подачи загрузки и сдвига, которые рассматривает регистр:

- Если общее количество битов в регистре равно 64, то есть FRAG_BITS = 64, то индекс Idx подаётся один раз в значении «000», происходит одна загрузка, после чего burst сдвигов;
- Если общее количество битов в регистре меньше 64, то есть FRAG_BITS < 64, то происходит загрузка и burst сдвигов в количестве 64/FRAG_BITS раз, при этом каждую загрузку инкрементируется индекс Idx, начиная со значения «000» (в случае burst = 1 и world_width = 8 последняя загрузка регистра будет происходить при Idx = «111»);
- Если общее количество битов в регистре больше 64, то есть FRAG_BITS > 64, то индекс Idx подаётся один раз в значении «000», происходит одна загрузка, после чего burst сдвигов, а оставшиеся свободные биты в регистре заполняются нулями.

На вход регистра подаётся байт разрешения ByteEn для записи в память, каждый бит которого соответствует одному байту из входной 64-битной шины. В ходе процесса происходит пересчёт для каждой порции данных в регистре разрешённого для неё бита и

на выходе вместе с данными подаётся сигнал Enable, соответствующий этой порции. ByteEn может принимать значения «11111111» (все входные биты разрешены для записи в память), «01111111» (разрешены для записи в память последние 7 входных байтов), «00111111» (разрешены для записи в память последние 6 входных байтов), …, «00000001» (разрешены для записи в память последний 1 входной байт), «00000000» (ни один входной бит не разрешён для записи в память), «10000000» (разрешены для записи в память первый 1 входной байт), ..., «11111110» (разрешены для записи в память первые 7 входных байтов), а также комбинации, которые внутри содержат единицы, а по краям нули, например, ByteEn = «01111100». Для случая world_width = 8 каждый выходной бит разрешения соответствует биту из входного байта, а при других значениях world_width происходит пересчёт, описанный в коде регистра.

Сигнал Done выставляется в значение «1», когда регистр завершил работу с данными, то есть произошёл последний сдвиг.

RTL-схема сдвигового регистра со значениями burst = 4 и world_width = 16 для записи представлена на Рисунке 7.

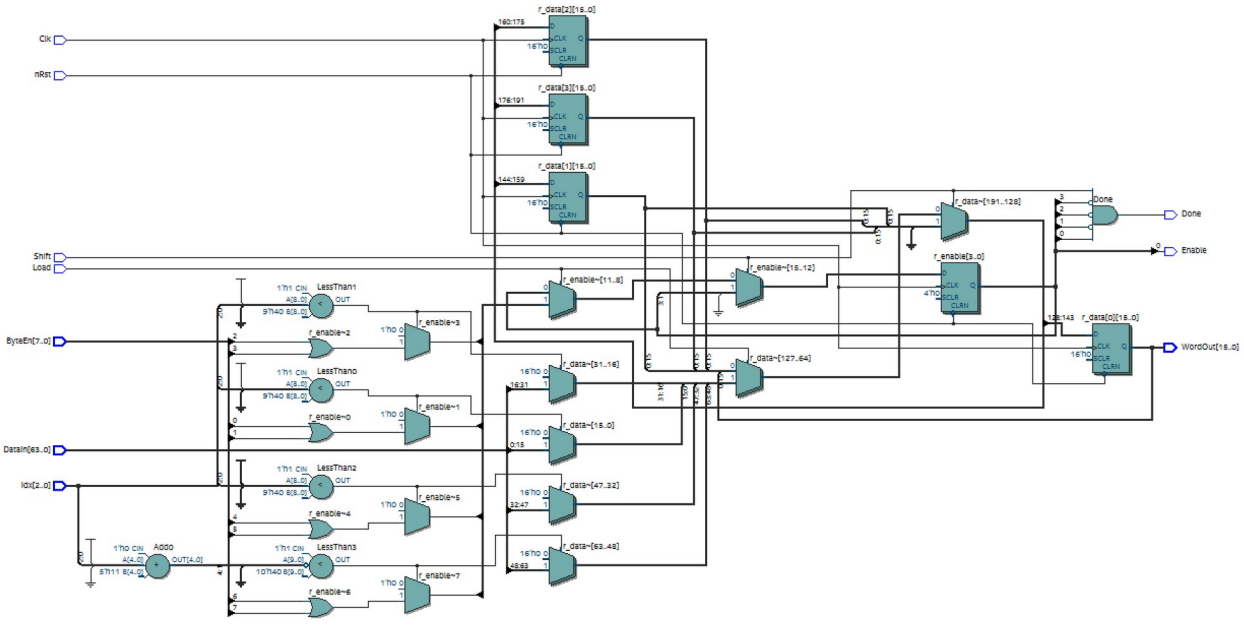

Рисунок 7. RTL-схема сдвигового регистра для записи со значениями burst = 4 и world_width = 16

Исходный код разработанного сдвигового регистра для записи из FIFO приведён в Приложении 1.

Для регистра были написаны тесты, демонстрирующие его работу при всех возможных комбинациях значений burst и world_width, а также при различных значениях ByteEn. Результаты тестирования для значений burst = 4 и world_width = 8 показаны на Рисунке 8. Исходный код тестбэнча и тестера приведены в Приложениях 2-3.

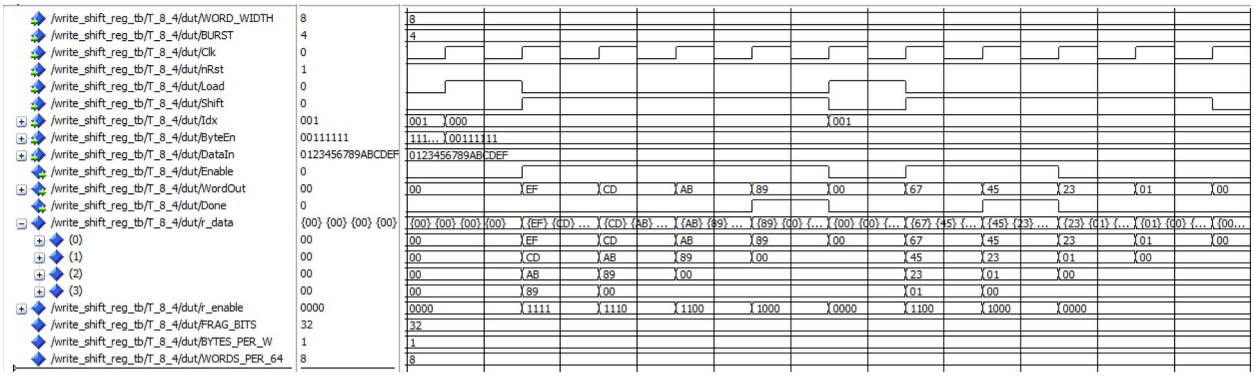

Рисунок 8. Фрагмент временной диаграммы выполнения операции сдвига в регистре для записи для значений burst = 4 и world_width = 8

2. Регистр для чтения

Для операции чтения из SDRAM был реализован сдвиговый регистр, УГО которого приведено на Рисунке 9.

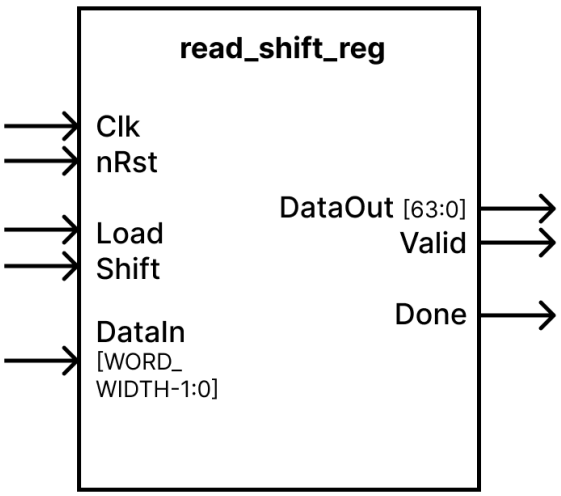

Рисунок 9. УГО сдвигового регистра для чтения

В Таблице 3 приведено описание входных и выходных сигналов разработанного регистра.

Таблица 3. Входные и выходные сигналы регистра для чтения

Название	Вход/	Разрядность,	Описание

сигнала	выход	бит	
Clk	in	1	Тактовый сигнал
nRst	in	1	Сигнал асинхронного сброса
Load	in	1	Сигнал разрешения загрузки. «1» - если регистр пуст и его можно заполнить, «0» - в противном случае
Shift	in	1	Сигнал сдвига данных. «1» - если на выходе нужно получить сдвиг, «0» - в противном случае
DataIn	in	world_width	Шина входных данных из памяти для записи в регистр
DataOut	out	64	Выходная порция данных, записываемая в FIFO
Valid	out	1	Сигнал валидности выходных данных. «1» - если данные валидны, «0» - в противном случае
Done	out	1	Сигнал завершения выдачи данных. «1» - если регистр пуст, «0» - в противном случае

На Рисунке 10 приведена схема работы сдвигового регистра и его взаимодействие с входными и выходными данными в случае burst = 4 и world_width = 16.

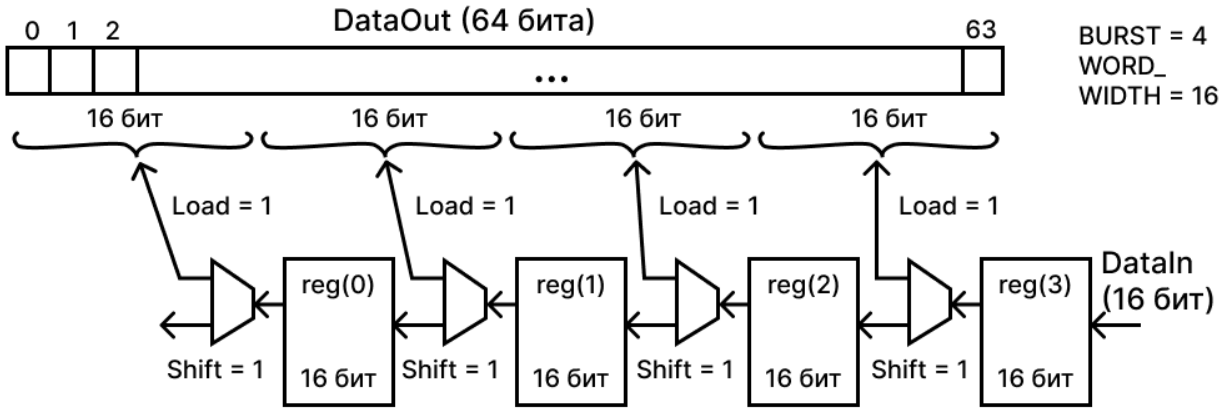

Рисунок 10. Схема работы сдвигового регистра для чтения при burst = 4 и world_width = 16

Принцип работы регистра заключается в параллельном занесении burst*world_width бит из памяти в него и последовательном сдвиге данных в количестве world_width бит за один такт. При этом с помощью дженерика задаются значения burst и

world_width. Все возможные их комбинации приведены в Таблице 3 и рассмотрены для работы регистра.

На вход регистра подаётся world_width бит данных, которые загружаются в регистр за один такт путём подачи единицы на вход Shift. После этого происходит сдвиг порции по world_width бит и занесение в регистр новой порции данных из памяти при выставлении сигнала Shift единицей. После заполнения регистра входными данными подаётся единица на вход Load, за один такт данные из регистра выгружаются в выходную 64-битную шину и сигнал Valid становится единицей. Одновременная подача единицы на Load и Shift невозможна, потому что Shift будет в приоритете из условия, приведённого в коде разработанного регистра. Возможны несколько ситуаций для подачи загрузки и сдвига, которые рассматривает регистр:

- Если общее количество битов в регистре равно 64, то есть FRAG_BITS = 64, то происходит burst сдвигов, после чего одна загрузка;
- Если общее количество битов в регистре меньше 64, то есть FRAG_BITS < 64, то происходит burst сдвигов и загрузка выходных данных, которые будут дозаполнены нулями в старших битах;
- Если общее количество битов в регистре больше 64, то есть FRAG_BITS > 64, то происходит burst сдвигов, после чего загрузка выходной полностью заполненной 64-битной шины, такие итерации повторяются FRAG_BITS/64 раз.

Сигнал Done выставляется в значение «1», когда регистр завершил работу с данными, то есть произошла последняя загрузка.

RTL-схема сдвигового регистра со значениями burst = 4 и world_width = 16 для записи представлена на Рисунке 11.

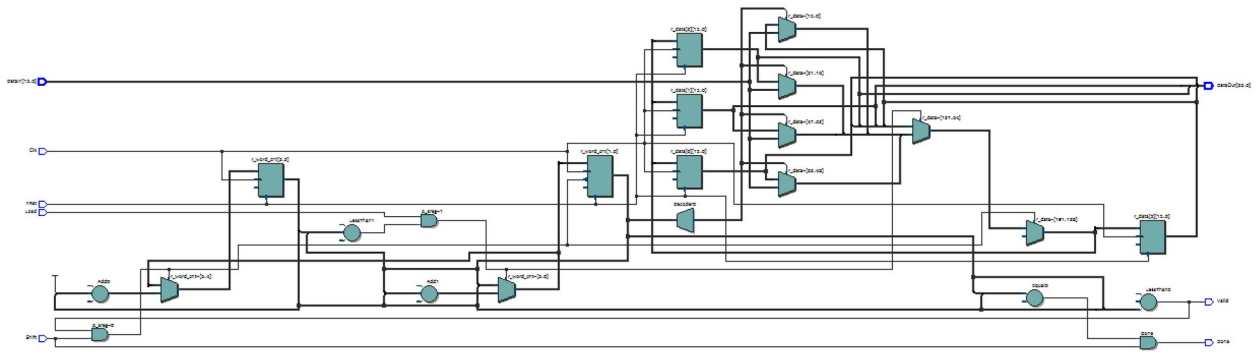

Рисунок 11. RTL-схема сдвигового регистра для чтения со значениями burst = 4 и world_width = 16

Исходный код разработанного сдвигового регистра для чтения из SDRAM приведён в Приложении 4.

Для регистра были написаны тесты, демонстрирующие его работу. Результаты тестирования для значений burst = 4 и world_width = 16 показаны на Рисунке 12. Исходный код тестбэнча и тестера приведены в Приложениях 5-6.

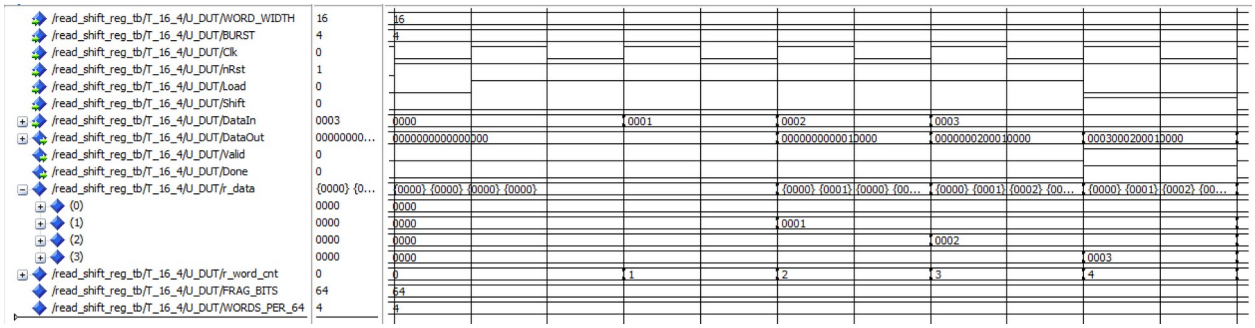

Рисунок 12. Фрагмент временной диаграммы выполнения операции сдвига в регистре для чтения для значений burst = 4 и world_width = 16

Тестирование

Для верификации подсистемы были разработаны тесты, перечисленные в Таблице X и показанные на Рисунках x-x.

Синтез и компиляция схемы

После успешного прохождения тестирования разрабатываемой подсистемы в программе Quartus были проведены компиляция, отчёт о которой представлен на Рисунке X, и синтез RTL-схемы, показанной на Рисунке X.

Картинка

Рисунок X. Отчёт о компиляции

Картинка

Рисунок X. RTL-схема, полученная в ходе синтеза

Синтез ещеее

Заключение

В ходе данной курсовой работы был разработан конечный автомат для обмена с внешней памятью SDRAM и выполнения присвоений выходных сигналов в зависимости от текущего состояния конечного автомата. Были описаны все его состояния, разработаны дополнительные модули и проведены тестирование и синтез.

Приложение

Приложение 1. Файл write_shift_reg.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity write_shift_reg is
  generic (
    WORD_WIDTH : integer := 16; -- 8, 16, 32, 64
    BURST      : integer := 4   -- 1, 2, 4, 8
  );
  port (
    Clk      : in  std_logic;
    nRst     : in  std_logic;

    Load     : in  std_logic;
    Shift    : in  std_logic;

      Idx        : in   std_logic_vector(2 downto 0); -- для длины регистра < 64, вычисляется
64/FRAG_BITS, 000 -> 001 -> 002 ...
    ByteEn   : in  std_logic_vector(7 downto 0);
    DataIn   : in  std_logic_vector(63 downto 0);

    Enable   : out std_logic;
    WordOut  : out std_logic_vector(WORD_WIDTH-1 downto 0);
    Done     : out std_logic -- когда Shift выдвигает последнее валидное слово
  );
end entity;

architecture rtl of write_shift_reg is

  type t_sreg_data   is array (0 to BURST-1) of std_logic_vector(WORD_WIDTH-1 downto 0);
  type t_sreg_enable is array (0 to BURST-1) of std_logic;

  signal r_data   : t_sreg_data;
  signal r_enable : t_sreg_enable;
```

```
  constant FRAG_BITS    : integer := WORD_WIDTH * BURST;
  constant BYTES_PER_W  : integer := WORD_WIDTH / 8;   -- 1,2,4,8
  constant WORDS_PER_64 : integer := 64 / WORD_WIDTH;  -- 1,2,4,8

begin

  WordOut <= r_data(0);
  Enable  <= r_enable(0);

  gen_done_1 : if BURST = 1 generate
      Done <= '1' when (Shift = '1' and r_enable(0) = '1') else '0';
  end generate;

  gen_done_2 : if BURST = 2 generate
      Done <= '1' when (Shift = '1' and r_enable(0) = '1' and r_enable(1) = '0') else '0';
  end generate;

  gen_done_4 : if BURST = 4 generate
      Done <= '1' when (Shift = '1' and r_enable(0) = '1' and
                        r_enable(1) = '0' and r_enable(2) = '0' and r_enable(3) = '0')
            else '0';
  end generate;

  gen_done_8 : if BURST = 8 generate
      Done <= '1' when (Shift = '1' and r_enable(0) = '1' and
                        r_enable(1) = '0' and r_enable(2) = '0' and r_enable(3) = '0' and
                               r_enable(4) = '0' and r_enable(5) = '0' and r_enable(6) = '0' and
r_enable(7) = '0')
            else '0';
  end generate;

  p_sreg : process(Clk, nRst)
  begin
    if nRst = '0' then
      for j in 0 to BURST-1 loop
        r_data(j)   <= (others => '0');
        r_enable(j) <= '0';
```

```
    end loop;

  elsif rising_edge(Clk) then

    if Shift = '1' then
      for j in 0 to BURST-2 loop
        r_data(j)   <= r_data(j+1);
        r_enable(j) <= r_enable(j+1);
      end loop;
      r_data(BURST-1)   <= (others => '0');
      r_enable(BURST-1) <= '0';

    elsif Load = '1' then

      if FRAG_BITS <= 64 then
        for j in 0 to BURST-1 loop
          if ((conv_integer(Idx)*BURST + j + 1) * WORD_WIDTH) <= 64 then
            r_data(j) <= DataIn(
              ((conv_integer(Idx)*BURST + j + 1) * WORD_WIDTH - 1) downto
              ((conv_integer(Idx)*BURST + j)     * WORD_WIDTH));

            if BYTES_PER_W = 1 then
              r_enable(j) <= ByteEn( ((conv_integer(Idx)*BURST + j) * WORD_WIDTH) / 8 );
            elsif BYTES_PER_W = 2 then
              r_enable(j) <=
                ByteEn( (((conv_integer(Idx)*BURST + j) * WORD_WIDTH) / 8) + 0 ) or
                ByteEn( (((conv_integer(Idx)*BURST + j) * WORD_WIDTH) / 8) + 1 );
            elsif BYTES_PER_W = 4 then
              r_enable(j) <=
                ByteEn( (((conv_integer(Idx)*BURST + j) * WORD_WIDTH) / 8) + 0 ) or
                ByteEn( (((conv_integer(Idx)*BURST + j) * WORD_WIDTH) / 8) + 1 ) or
                ByteEn( (((conv_integer(Idx)*BURST + j) * WORD_WIDTH) / 8) + 2 ) or
                ByteEn( (((conv_integer(Idx)*BURST + j) * WORD_WIDTH) / 8) + 3 );
            else
              r_enable(j) <=
                ByteEn(0) or ByteEn(1) or ByteEn(2) or ByteEn(3) or
                ByteEn(4) or ByteEn(5) or ByteEn(6) or ByteEn(7);
            end if;
```

```
          else
            r_data(j)   <= (others => '0');
            r_enable(j) <= '0';
          end if;
        end loop;


      else
        for j in 0 to BURST-1 loop
          if j < WORDS_PER_64 then
            r_data(j) <= DataIn(((j+1) * WORD_WIDTH - 1) downto (j * WORD_WIDTH));


            if BYTES_PER_W = 1 then
              r_enable(j) <= ByteEn( (j * WORD_WIDTH) / 8 );
            elsif BYTES_PER_W = 2 then
              r_enable(j) <=
                ByteEn( ((j * WORD_WIDTH) / 8) + 0 ) or
                ByteEn( ((j * WORD_WIDTH) / 8) + 1 );
            elsif BYTES_PER_W = 4 then
              r_enable(j) <=
                ByteEn( ((j * WORD_WIDTH) / 8) + 0 ) or
                ByteEn( ((j * WORD_WIDTH) / 8) + 1 ) or
                ByteEn( ((j * WORD_WIDTH) / 8) + 2 ) or
                ByteEn( ((j * WORD_WIDTH) / 8) + 3 );
            else
              r_enable(j) <=
                ByteEn(0) or ByteEn(1) or ByteEn(2) or ByteEn(3) or
                ByteEn(4) or ByteEn(5) or ByteEn(6) or ByteEn(7);
            end if;


          else
            r_data(j)   <= (others => '0');
            r_enable(j) <= '0';
          end if;
        end loop;
      end if;
    end if;
  end if;
end process;
```

```
end architecture;
```

Приложение 2. Файл write_shift_reg_tb.vhd

```
library ieee;
use ieee.std_logic_1164.all;

entity write_shift_reg_tb is
end entity;

architecture top of write_shift_reg_tb is
begin

  T_8_1 : entity work.write_shift_reg_tester
    generic map ( WORD_WIDTH => 8, BURST => 1 );

  T_8_2 : entity work.write_shift_reg_tester
    generic map ( WORD_WIDTH => 8, BURST => 2 );

  T_8_4 : entity work.write_shift_reg_tester
    generic map ( WORD_WIDTH => 8, BURST => 4 );

  T_8_8 : entity work.write_shift_reg_tester
    generic map ( WORD_WIDTH => 8, BURST => 8 );

  T_16_1 : entity work.write_shift_reg_tester
    generic map ( WORD_WIDTH => 16, BURST => 1 );

  T_16_2 : entity work.write_shift_reg_tester
    generic map ( WORD_WIDTH => 16, BURST => 2 );

  T_16_4 : entity work.write_shift_reg_tester
    generic map ( WORD_WIDTH => 16, BURST => 4 );

  T_16_8 : entity work.write_shift_reg_tester
    generic map ( WORD_WIDTH => 16, BURST => 8 );

  T_32_1 : entity work.write_shift_reg_tester
```

```
    generic map ( WORD_WIDTH => 32, BURST => 1 );

  T_32_2 : entity work.write_shift_reg_tester
    generic map ( WORD_WIDTH => 32, BURST => 2 );

  T_32_4 : entity work.write_shift_reg_tester
    generic map ( WORD_WIDTH => 32, BURST => 4 );

  T_32_8 : entity work.write_shift_reg_tester
    generic map ( WORD_WIDTH => 32, BURST => 8 );

  T_64_1 : entity work.write_shift_reg_tester
    generic map ( WORD_WIDTH => 64, BURST => 1 );

  T_64_2 : entity work.write_shift_reg_tester
    generic map ( WORD_WIDTH => 64, BURST => 2 );

  T_64_4 : entity work.write_shift_reg_tester
    generic map ( WORD_WIDTH => 64, BURST => 4 );

  T_64_8 : entity work.write_shift_reg_tester
    generic map ( WORD_WIDTH => 64, BURST => 8 );



end architecture;
```

Приложение 3. Файл write_shift_reg_tester.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity write_shift_reg_tester is
  generic (
    WORD_WIDTH : integer := 16;
    BURST      : integer := 4
  );
end entity;
```

```
architecture tb of write_shift_reg_tester is

  constant CLK_PERIOD   : time    := 10 ns;
  constant FRAG_BITS  : integer := WORD_WIDTH * BURST;
  constant WORDS_PER_64 : integer := 64 / WORD_WIDTH;

  function calc_num_frags return integer is
  begin
    if FRAG_BITS = 0 then
      return 1;
    elsif FRAG_BITS >= 64 then
      return 1;
    else
      return 64 / FRAG_BITS;
    end if;
  end function;

  constant NUM_FRAGS : integer := calc_num_frags;

  type t_idx_array is array (0 to 7) of std_logic_vector(2 downto 0);
  constant IDX_TABLE : t_idx_array := (
    "000", "001", "010", "011",
    "100", "101", "110", "111"
  );

  constant DATA_TEMPLATE : std_logic_vector(63 downto 0) :=
    x"0123456789ABCDEF";

  type t_be_array is array (natural range <>) of std_logic_vector(7 downto 0);

  constant BE_PATTERNS : t_be_array := (
    "11111111",
    "00111111",
    "11111000",
    "01111100"
  );
```

```
  signal Clk    : std_logic := '0';
  signal nRst   : std_logic := '0';

  signal Load   : std_logic := '0';
  signal Shift  : std_logic := '0';

  signal Idx    : std_logic_vector(2 downto 0) := (others => '0');
  signal ByteEn : std_logic_vector(7 downto 0) := (others => '0');
  signal DataIn : std_logic_vector(63 downto 0) := (others => '0');

  signal Enable  : std_logic;
  signal WordOut : std_logic_vector(WORD_WIDTH-1 downto 0);
  signal Done    : std_logic;

begin

  p_clk : process
  begin
    Clk <= '0';
    wait for CLK_PERIOD/2;
    Clk <= '1';
    wait for CLK_PERIOD/2;
  end process;

  dut : entity work.write_shift_reg
    generic map (
      WORD_WIDTH => WORD_WIDTH,
      BURST      => BURST
    )
    port map (
      Clk     => Clk,
      nRst    => nRst,
      Load    => Load,
      Shift   => Shift,
      Idx     => Idx,
      ByteEn  => ByteEn,
      DataIn  => DataIn,
      Enable  => Enable,
```

```
    WordOut => WordOut,
    Done    => Done
  );

p_stim : process

  procedure set_pattern(
    constant be_idx : in integer
  ) is
  begin
    if be_idx < BE_PATTERNS'length then
      ByteEn <= BE_PATTERNS(be_idx);
    else
      ByteEn <= (others => '1');
    end if;
    DataIn <= DATA_TEMPLATE;
  end procedure;

begin
  nRst   <= '0';
  Load   <= '0';
  Shift  <= '0';
  Idx    <= (others => '0');
  ByteEn <= (others => '0');
  DataIn <= DATA_TEMPLATE;

  wait for 3*CLK_PERIOD;
  wait until rising_edge(Clk);
  nRst <= '1';

  wait for 2*CLK_PERIOD;

  for be_i in 0 to BE_PATTERNS'length-1 loop

    set_pattern(be_i);
    wait until rising_edge(Clk);

    if FRAG_BITS >= 64 then
```

```
  Idx  <= IDX_TABLE(0);
  Load <= '1';
  Shift <= '0';
  wait until rising_edge(Clk);
  Load <= '0';

  for w in 0 to WORDS_PER_64-1 loop
    Shift <= '1';
    wait until rising_edge(Clk);
  end loop;
  Shift <= '0';

  wait for 4*CLK_PERIOD;

else

  Idx  <= IDX_TABLE(0);
  Load <= '1';
  Shift <= '0';
  wait until rising_edge(Clk);
  Load <= '0';

  for frag in 0 to NUM_FRAGS-1 loop

    for k in 0 to BURST-1 loop
      Shift <= '1';
      wait until rising_edge(Clk);
    end loop;
    Shift <= '0';

    if frag < NUM_FRAGS-1 then
      Idx  <= IDX_TABLE(frag + 1);
      Load <= '1';
      wait until rising_edge(Clk);
      Load <= '0';
    end if;
```

```
        end loop;

        wait for 6*CLK_PERIOD;

      end if;

      Load  <= '0';
      Shift <= '0';
      wait for 5*CLK_PERIOD;

    end loop;

    wait;
  end process;

end architecture;
```

Приложение 4. Файл read_shift_reg.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity read_shift_reg is
  generic (
    WORD_WIDTH : integer := 16; -- 8, 16, 32, 64
    BURST      : integer := 4   -- 1, 2, 4, 8
  );
  port (
    Clk      : in  std_logic;
    nRst     : in  std_logic;

    Load     : in  std_logic;
    Shift    : in  std_logic;

    DataIn   : in  std_logic_vector(WORD_WIDTH-1 downto 0);

    DataOut  : out std_logic_vector(63 downto 0);
```

```
    Valid    : out std_logic;
    Done     : out std_logic
  );
end entity;

architecture rtl of read_shift_reg is

  type t_sreg_data is array (0 to BURST-1) of std_logic_vector(WORD_WIDTH-1 downto 0);
  signal r_data : t_sreg_data;
  signal r_word_cnt : std_logic_vector(3 downto 0);

  constant FRAG_BITS    : integer := WORD_WIDTH * BURST;
  constant WORDS_PER_64 : integer := 64 / WORD_WIDTH;

begin

  gen_data_out_small : if BURST <= WORDS_PER_64 generate

    gen_real : for i in 0 to BURST-1 generate
      DataOut((i+1)*WORD_WIDTH - 1 downto i*WORD_WIDTH) <= r_data(i);
    end generate;

    gen_zero : for i in BURST to WORDS_PER_64-1 generate
      DataOut((i+1)*WORD_WIDTH - 1 downto i*WORD_WIDTH) <= (others => '0');
    end generate;

  end generate;

  gen_data_out_large : if BURST > WORDS_PER_64 generate

    gen_real_only : for i in 0 to WORDS_PER_64-1 generate
      DataOut((i+1)*WORD_WIDTH - 1 downto i*WORD_WIDTH) <= r_data(i);
    end generate;

  end generate;

  pad_hi : if WORDS_PER_64*WORD_WIDTH < 64 generate
    DataOut(63 downto WORDS_PER_64*WORD_WIDTH) <= (others => '0');
```

```
end generate;


Valid <=
  '1' when (
    (FRAG_BITS >= 64  and conv_integer(r_word_cnt) >= WORDS_PER_64) or
    (FRAG_BITS <  64  and conv_integer(r_word_cnt) = BURST)
  ) else '0';


Done <=
  '1' when (
    Shift = '1' and (
      (FRAG_BITS >= 64 and conv_integer(r_word_cnt) = WORDS_PER_64) or
      (FRAG_BITS <  64 and conv_integer(r_word_cnt) = BURST)
    )
  ) else '0';


p_sreg : process(Clk, nRst)
begin
  if nRst = '0' then
    for j in 0 to BURST-1 loop
      r_data(j) <= (others => '0');
    end loop;
    r_word_cnt <= (others => '0');


  elsif rising_edge(Clk) then


    if (Shift = '1') and
       ((FRAG_BITS >= 64 and conv_integer(r_word_cnt) >= WORDS_PER_64) or
        (FRAG_BITS <  64 and conv_integer(r_word_cnt) = BURST)) then


      if FRAG_BITS >= 64 then
        for j in 0 to BURST-1 loop
          if j <= BURST-WORDS_PER_64-1 then
            r_data(j) <= r_data(j+WORDS_PER_64);
          else
            r_data(j) <= (others => '0');
          end if;
        end loop;
```

```
          r_word_cnt <= r_word_cnt - conv_std_logic_vector(WORDS_PER_64, 4);


        else
          for j in 0 to BURST-1 loop
            r_data(j) <= (others => '0');
          end loop;
          r_word_cnt <= r_word_cnt - conv_std_logic_vector(BURST, 4);
        end if;


      elsif (Load = '1') and (conv_integer(r_word_cnt) < BURST) then
        r_data(conv_integer(r_word_cnt)) <= DataIn;
        r_word_cnt <= r_word_cnt + conv_std_logic_vector(1, 4);
      end if;
    end if;
  end process;


end architecture;
```

Приложение 5. Файл read_shift_reg_tb.vhd

```
library ieee;
use ieee.std_logic_1164.all;


entity read_shift_reg_tb is
end entity;


architecture sim of read_shift_reg_tb is
begin


  T_8_1 : entity work.read_shift_reg_tester
    generic map ( WORD_WIDTH => 8, BURST => 1 );


  T_8_2 : entity work.read_shift_reg_tester
    generic map ( WORD_WIDTH => 8, BURST => 2 );


  T_8_4 : entity work.read_shift_reg_tester
    generic map ( WORD_WIDTH => 8, BURST => 4 );


  T_8_8 : entity work.read_shift_reg_tester
```

```
  generic map ( WORD_WIDTH => 8, BURST => 8 );

T_16_1 : entity work.read_shift_reg_tester
  generic map ( WORD_WIDTH => 16, BURST => 1 );

T_16_2 : entity work.read_shift_reg_tester
  generic map ( WORD_WIDTH => 16, BURST => 2 );

T_16_4 : entity work.read_shift_reg_tester
  generic map ( WORD_WIDTH => 16, BURST => 4 );

T_16_8 : entity work.read_shift_reg_tester
  generic map ( WORD_WIDTH => 16, BURST => 8 );

T_32_1 : entity work.read_shift_reg_tester
  generic map ( WORD_WIDTH => 32, BURST => 1 );

T_32_2 : entity work.read_shift_reg_tester
  generic map ( WORD_WIDTH => 32, BURST => 2 );

T_32_4 : entity work.read_shift_reg_tester
  generic map ( WORD_WIDTH => 32, BURST => 4 );

T_32_8 : entity work.read_shift_reg_tester
  generic map ( WORD_WIDTH => 32, BURST => 8 );

T_64_1 : entity work.read_shift_reg_tester
  generic map ( WORD_WIDTH => 64, BURST => 1 );

T_64_2 : entity work.read_shift_reg_tester
  generic map ( WORD_WIDTH => 64, BURST => 2 );

T_64_4 : entity work.read_shift_reg_tester
  generic map ( WORD_WIDTH => 64, BURST => 4 );

T_64_8 : entity work.read_shift_reg_tester
  generic map ( WORD_WIDTH => 64, BURST => 8 );
```

```
end architecture;
```

Приложение 6. Файл read_shift_reg_tester.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity read_shift_reg_tester is
  generic (
    WORD_WIDTH : integer := 16; -- 8, 16, 32, 64
    BURST      : integer := 4   -- 1, 2, 4, 8
  );
end entity;

architecture tb of read_shift_reg_tester is

  constant CLK_PERIOD  : time    := 10 ns;
  constant FRAG_BITS   : integer := WORD_WIDTH * BURST;
  constant NUM_SHIFTS  : integer := (FRAG_BITS + 63) / 64;
  constant WORDS_PER_64: integer := 64 / WORD_WIDTH;

  signal Clk      : std_logic := '0';
  signal nRst     : std_logic := '0';

  signal Load     : std_logic := '0';
  signal Shift    : std_logic := '0';

  signal DataIn   : std_logic_vector(WORD_WIDTH-1 downto 0) := (others => '0');
  signal DataOut  : std_logic_vector(63 downto 0);
  signal Valid    : std_logic;
  signal Done     : std_logic;

begin

  U_DUT : entity work.read_shift_reg
    generic map (
      WORD_WIDTH => WORD_WIDTH,
```

```
    BURST      => BURST
  )
  port map (
    Clk      => Clk,
    nRst     => nRst,
    Load     => Load,
    Shift    => Shift,
    DataIn   => DataIn,
    DataOut  => DataOut,
    Valid    => Valid,
    Done     => Done
  );

p_clk : process
begin
  Clk <= '0';
  wait for CLK_PERIOD/2;
  Clk <= '1';
  wait for CLK_PERIOD/2;
end process;

p_stim : process
begin
  nRst   <= '0';
  Load   <= '0';
  Shift  <= '0';
  DataIn <= (others => '0');

  wait for 3*CLK_PERIOD;
  nRst <= '1';

  wait until rising_edge(Clk);

  for i in 0 to BURST-1 loop
    Load   <= '1';
    Shift  <= '0';
    DataIn <= conv_std_logic_vector(i, WORD_WIDTH);
    wait until rising_edge(Clk);
```

```
    end loop;


    Load <= '0';


    for k in 0 to NUM_SHIFTS-1 loop
      Shift <= '1';
      Load  <= '0';
      wait until rising_edge(Clk);
    end loop;


    Shift <= '0';


    wait for 5*CLK_PERIOD;
    wait;
  end process;


end architecture;
```

Приложение 7. Файл xxx.vhd