Load Store Unit

Лабораторная работа #5

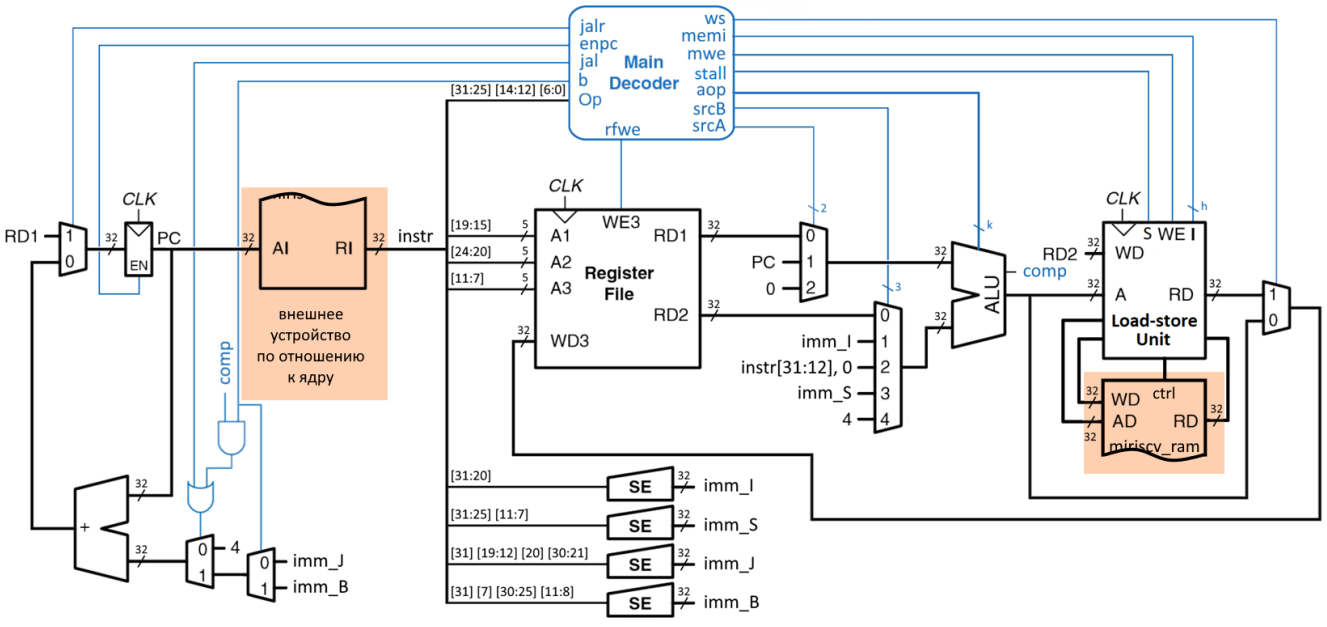
Модуль Load-store Unit (LSU, **miriscv_lsu**) служит для исполнения инструкций типа LOAD и STORE: он соответственно считывает содержимое из памяти данных или записывает в нее требуемые значения. В процессорах с RISC архитектурой с помощью LSU осуществляется обмен данными между регистрами общего назначения и памятью данных.

Прототип устройства

```
module miriscv_lsu
(
  input           clk_i,           // синхронизация
  input           arstn_i,         // сброс внутренних регистров

  // memory protocol
  input           data_gnt_i,      // 1 - память восприняла запрос
  input           data_rvalid_i,   // 1 - на выходе памяти запрошенные данные
  input   [31:0]  data_rdata_i,    // запрошенные данные
  output          data_req_o,      // 1 - обратиться к памяти
  output          data_we_o,       // 1 - это запрос на запись
  output  [3:0]   data_be_o,       // к каким байтам слова идет обращение
  output  [31:0]  data_addr_o,     // адрес, по которому идет обращение
  output  [31:0]  data_wdata_o,    // данные, которые требуется записать

  // core protocol
  input   [31:0]  lsu_addr_i,      // адрес, по которому хотим обратиться
  input           lsu_we_i,        // 1 – если нужно записать в память
  input   [2:0]   lsu_size_i,      // размер обрабатываемых данных
  input   [31:0]  lsu_data_i,      // данные для записи в память
  input           lsu_req_i,       // 1 - обратиться к памяти
  output          lsu_stall_req_o, // используется как !enable pc
  output  [31:0]  lsu_data_o       // данные считанные из памяти
);
```



Задание и условия выполнения

Необходимо реализовать блок LSU, служащий «прослойкой» между ядром и памятью (внешним, по отношению к ядру, устройством). Рисунок выше демонстрирует упрощенное подключение LSU к общей памяти команд-данных.

Ранее разработанные модули памяти Data Memory и Instruction Memory удаляются из проекта. Вместо них подключается двухпортовая память **miriscv_ram.sv** (написана на SystemVerilog), которая инициализируется одним файлом, содержащим и инструкции для процессора RISC-V, начинающиеся с нулевых адресов, и статические данные, расположенные далее. Используя специальный синтаксис можно указывать в фале адреса областей памяти, которые требуется инициализировать.

```
reg [3:0][7:0] mem [0:NUM_WORDS];
$readmemh("my_program.dat", mem);
```

my_program.dat

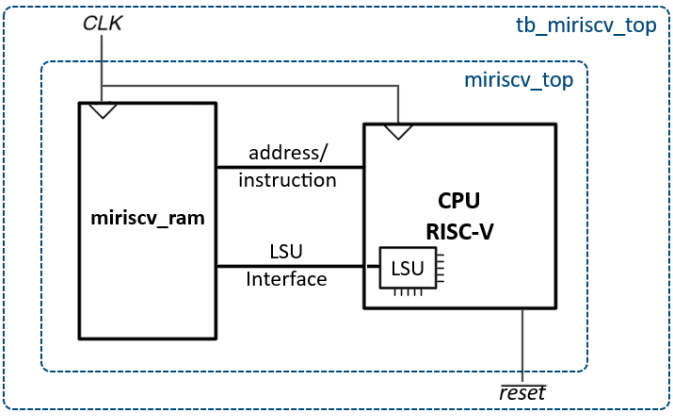
```
// some comment
// data from address 0
0x00000317
0x00830067
0x00000493
// data from address 0x100
@100
0x00000001
0x00000002
0x00000003
```

Содержимое памяти

```
mem
  0: 00 00 03 17
  1: 00 83 00 67
  2: 00 00 04 93
  3: XX XX XX XX
---------
256: 00 00 00 01
257: 00 00 00 02
258: 00 00 00 03
```

Процессор соединяется с памятью через LSU, однако, из-за того, что один из портов памяти (для считывания инструкций) используется только для операций чтения, его входы и выходы присоединяются к **miriscv_ram** напрямую, а не через LSU.

К дополнительным материалам лабораторной работы так же относится файл верхнего уровня иерархии **miriscv_top.sv**, который описывает (и демонстрирует) подключение процессора к памяти. После реализации и добавления блока LSU в процессор его входы и выходы должны совпадать со входами и выходами, описанными в модуле **miriscv_top**.



Внимательно изучи как все организовано в модуле **miriscv_top**, и как в нем подключаются модули процессора и памяти. На рисунке упрощенное подключение модулей внутри **miriscv_top**.

Для того, чтобы верифицировать работу системы память-процессор, дополнительные материалы к лабораторной включают в себя файл тестового окружения **tb_miriscv_top.v**. Модуль содержит два параметра: **RAM_SIZE** (указывает размер **miriscv_ram** в байтах) и **RAM_INIT_FILE** (указывает имя txt-файла для инициализации памяти). Для проверки написать осмысленную программу, считывающую *из*, и записывающую *в* память байты и полуслова.

Функциональное описание сигналов со стороны процессора

На входной порт **lsu_addr_i** от процессора поступает адрес ячейки памяти, к которой будет произведено обращение. Намеренье процессора обратиться к памяти (и для чтения, и для записи) отражается выставлением сигнала **lsu_req_i** в единицу. Если процессор собирается записывать в память, то сигнал **lsu_we_i** выставляется в единицу, а сами данные, которые следует записать, поступают от него на вход **lsu_data_i**. Если процессор читает из памяти, то сигнал **lsu_we_i** находится в нуле, а считанные данные подаются для процессора на выход **lsu_data_o**.

Инструкции LOAD и STORE в RV32I поддерживают обмен 8-битными, 16-битными или 32-битными значениями, однако в самом процессоре происходит работа только с 32-битными числами, поэтому загружая байты или полуслова из памяти их необходимо предварительно расширить до 32-битного значения. Для выбора разрядности на вход LSU подается сигнал **lsu_size_i**, принимающий следующие значения:

Название	Значение	Пояснение
LDST_B	3'd0	Знаковое 8-битное значение
LDST_H	3'd1	Знаковое 16-битное значение
LDST_W	3'd2	32-битное значение
LDST_BU	3'd4	Беззнаковое 8-битное значение
LDST_HU	3'd5	Беззнаковое 16-битное значение

Формат представления числа (является оно знаковым или беззнаковым) имеет значение только для операций типа LOAD: если число знаковое, то производится **расширение знака** (sign extension – операция добавления знакового бита числа со стороны его старшего разряда) до 32 бит, если беззнаковое – **расширение нуля** (zero extension – операция добавления нулевого бита со стороны старшего разряда числа).

Эти операции позволяют увеличить разрядность числа без изменения его значения. Разберем следующий пример: требуется увеличить разрядность 8-битного слова S до 32 бит так, чтобы полученное слово L сохранило первоначальное значение. Если S – знаковое, производим расширение знака следующим образом:

```
wire signed [7:0]  S;
wire signed [31:0] L;

assign L = {{24{S[7]}}, S};
```

Если S – беззнаковое, производим расширение нуля следующим образом:

```
wire [7:0]  S;
wire [31:0] L;

assign L = {24'b0, S};
```

Для операций типа STORE формат представления чисел не важен, для них **lsu_size_i** сможет принимать значение только от 0 до 2.

Выходной сигнал **lsu_stall_req_o** нужен для управления входом enable pc, и должен выдавать единицу начиная с момента обращения к памяти, до тех пор, пока память не выдала на своем выходе **data_valid_i** единицу, что значит, что на вход **data_rdata_i** поданы считанные из памяти данные.

lsu_kill_i должен выводить FSM взаимодействия с памятью в состояние сброса.

Функциональное описание сигналов со стороны памяти

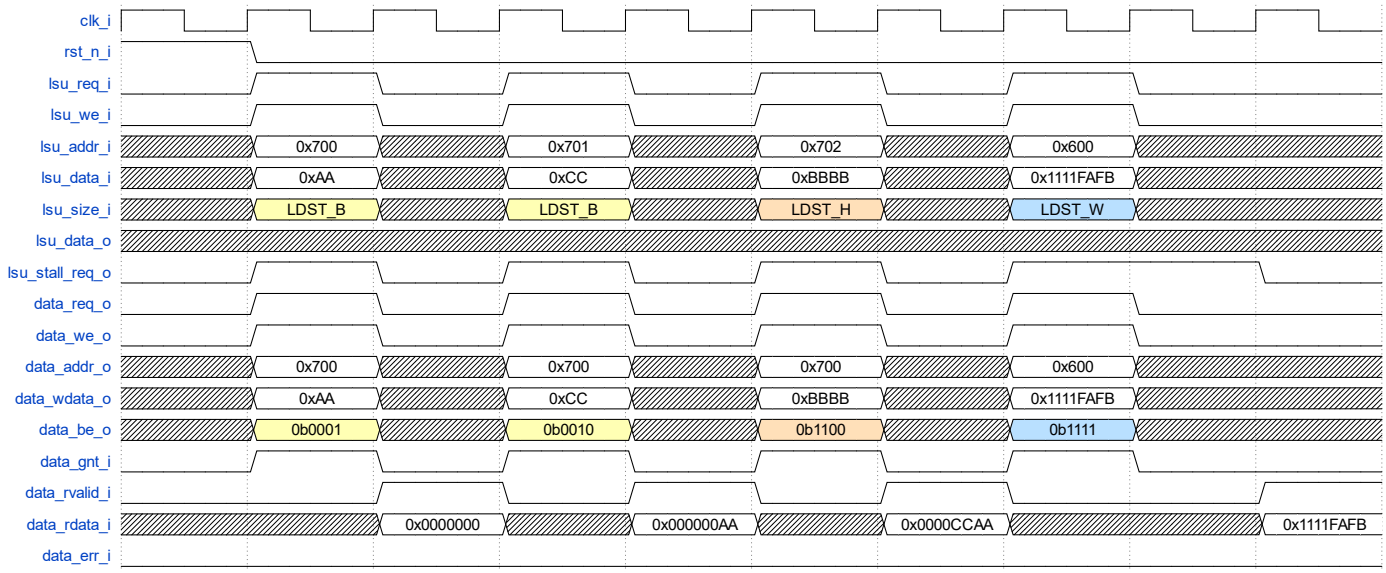
Память данных имеет 32-битную разрядность ячейки памяти (слово, word) и поддерживает побайтовую адресацию. Это значит, что существует возможность записи значения по одному байту в пределах одного слова (4-байтовой ячейки памяти). Для указания на необходимые байты интерфейс к памяти предусматривает использование 4-битного сигнала **data_be_o**, подаваемого вместе с адресом слова **dara_addr_o**. Позиции битов 4-битного сигнала соответствуют позициям байтов в слове. Если конкретный бит **data_be_o** равен 1, то соответствующий ему байт будет записан в память. Данные для записи подаются на выход **data_wdata_o**. На результат чтения из памяти состояние **data_be_o** не влияет, так как чтение производится всегда по 32-бита.

После получения запроса на чтение/запись из ядра, LSU перенаправляет запрос в память данных, реагируя на приходящие от нее сигналы. Сигнал **data_req_o** сообщает памяти о наличии запроса. **data_gnt_i** сигнализирует, что память начала обрабатывать запрос. Через такт после получения **data_gnt_i** данные на *входе* памяти не обязаны сохранять свое значение.
Сигнал **data_we_o** сообщает памяти о типе запроса: **data_we_o** равен 1, если отправлен запрос на запись, 0 – если отправлен запрос на чтение.

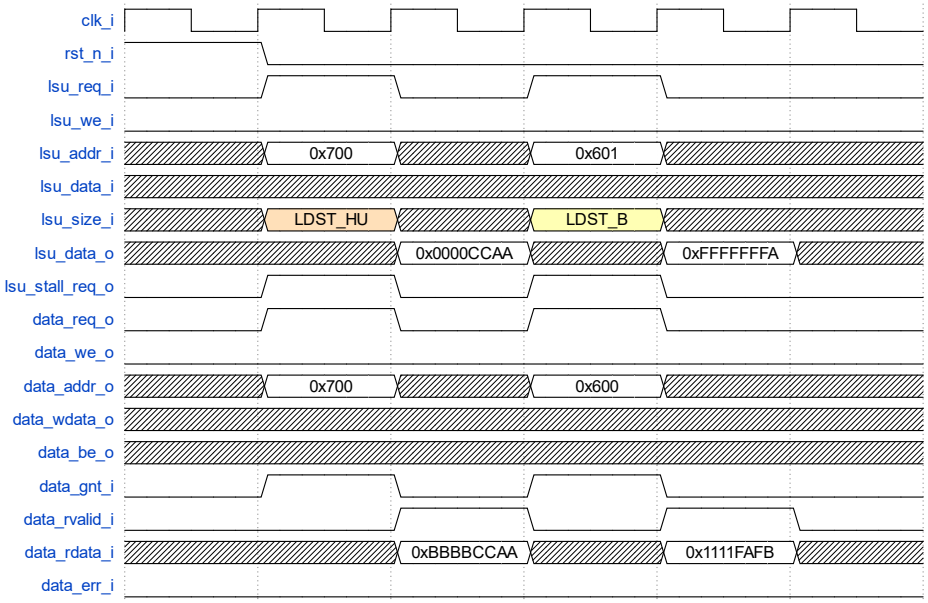
Сигнал **data_rvalid_i** сообщает о появлении ответа от памяти на линиях **data_rdata_i**.
Сигнал **data_rvalid_i** активен на протяжении только одного такта. При этом пока запрос в память обрабатывается (то есть **data_rvalid_i** находится в нуле), LSU должен поднять сигнал **lsu_stall_req_o** – это сообщит блоку управления, что работа ядра должна быть приостановлена, пока запрос не будет выполнен (об этом уже писалось ранее).
Сигнал **data_rdata_i** содержит данные из ячейки памяти на момент принятия запроса. Следовательно, после совершения записи в память **data_rdata_i** будет хранить *предыдущее* значение из ячейки, а не записанное.

Команды	**Address Byte Offset**	**data_wdata_o**	**data_be_o**
sb	00 01 10 11	{4{lsu_data_i[7:0]}}	0001 0010 0100 1000
sh	00 10	{2{lsu_data_i[15:0]}}	0011 1100
sw	00	lsu_data_i[31:0]	1111

Команды	**Address Byte Offset**	**lsu_data_o**
lb	00 01 10 11	{{24{data_rdata_i[7]}, data_rdata_i[7:0]} {{24{data_rdata_i[15]}, data_rdata_i[15:8]} {{24{data_rdata_i[23]}, data_rdata_i[23:16]} {{24{data_rdata_i[31]}, data_rdata_i[31:24]}
lh	00 10	{{16{data_rdata_i[15]}, data_rdata_i[15:0]} {{16{data_rdata_i[31]}, data_rdata_i[31:16]}
lw	00	data_rdata_i[31:0]
lbu	00 01 10 11	{24'b0, data_rdata_i[7:0]} {24'b0, data_rdata_i[15:8]} {24'b0, data_rdata_i[23:16]} {24'b0, data_rdata_i[31:24]}
lhu	00 10	{16'b0, data_rdata_i[15:0]} {16'b0, data_rdata_i[31:16]}



Пример временной диаграммы записи в память



Пример временной диаграммы чтения из памяти