Course Name: Computer Architecture	**Course Code:** CMPE-421L
Assignment Type: Lab	**Dated:** 9th October 2023
Semester: 7th	**Session:** 2020
Lab/Project/Assignment #: 6	**CLOs to be covered: CLO 2**
Lab Title: Load/Store Operation and Data Memory Interface	**Teacher Name:** Engr. Afeef Obaid

Lab Evaluation

CLO 2	Understand the basics of RISC-V architecture, its assembly & design of basic Datapath components of a single cycle RISC-V processor.					
Levels (Marks)	**Level1**	**Level2**	**Level3**	**Level4**	**Level5**	**Level6**
(10)						
					Total	**/10**

Rubrics for Current Lab Evaluation

Scale	Marks	Level	Rubric
Excellent	**9-10**	L1	Submitted all lab tasks, BONUS task, have good understanding.
Very Good	**7-8**	L2	Submitted the lab tasks but have good understanding
Good	**5-6**	L3	Submitted the lab tasks but have weak understanding.
Basic	**3-4**	L4	Submitted the lab tasks but have no understanding.
Barely Acceptable	**1-2**	L5	Submitted only one lab task.
Not Acceptable	**0**	L6	Did not attempt

Lab # 6

Lab Goals

By reading this manual, students will be able to:

- Understand RISC-V (RV) Load Operations.
- Understand RISC-V (RV) Store Operations
- Understand Data Memory Interface in RISC-V (RV).

Equipment Required

- Computer system with ModelSim or Xcelium, installed on it.

Load/Store Operation and Data Memory Interface

The load/store operations are performed for data transfer from/to data memory. To implement these operations, we first need to connect data memory with the processor data path at the data memory access phase. A simple tightly coupled data memory interface is shown in figure below.

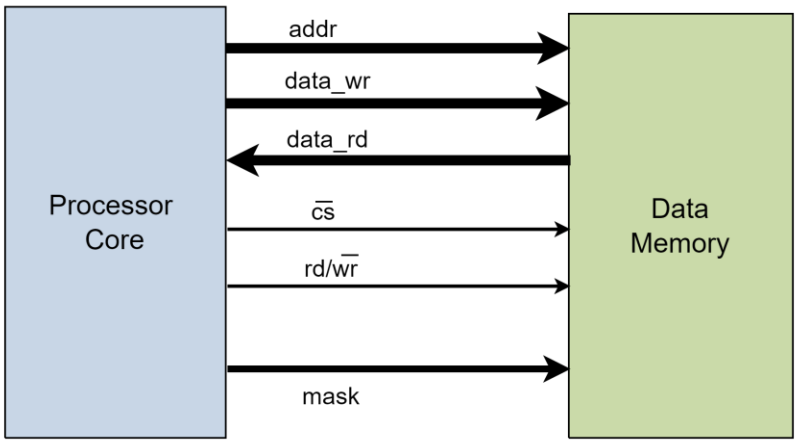

Figure 5.1. Data memory interface.

The data write operation is performed by setting up the address (addr), data to be written (data_wr) and mask followed by the cs and wr signals. The memory write operation is synchronous. The mask signal is used to inform the memory 1) the size of the data to be written, which can be of size byte, half-word or word and, 2) the location of the byte/half-word on a bus width equal to word size.

Load Operation – Memory Access Phase (M)

The code segment below illustrates setting up the signals during the data memory access phase when performing a load or store operation.

```
// Prepare the signals to perform load/store operations
assign st_ops        = |exe2mem_ctrl.mem_st_ops;
assign mem2dmem.addr = exe2mem_data.alu_result;
assign mem2dmem.cs   = ~(st_ops | (|exe2mem_ctrl.mem_ld_ops)); // cs is active low
assign mem2dmem.wr   = ~st_ops;  // Memory write/store is active low
```

Listing 5.1. Setting up the signals in the data memory access phase for load/store operation.

The data load operation corresponds to memory read and here for single cycle implementation we have used asynchronous memory read. The mask signal is of no significance because the load operation always receives 32-bit data from data memory as can be observed from the following code segment.

```
// Asynchronous read operation
assign dmem2mem.data_rd = ((~mem2dmem.cs) & (~mem2dmem.wr))
                        ? data_memory[addr_ff]
                        : '0;
```

Listing 5.2. Asynchronous data memory read for load operation.

The received 32-bit data is processed during the data memory access phase for the requested size and the corresponding address location.

```
// Extract the right size from the read data
always_comb begin
   dmem_rdata_byte  = '0;
   dmem_rdata_hword = '0;
   dmem_rdata_word  = '0;

   case (exe2mem_ctrl.mem_ld_ops)
      MEM_LD_OPS_LB,
      MEM_LD_OPS_LBU : begin
         case (mem2dmem.addr[1:0])
            2'b00 : begin
               dmem_rdata_byte = dmem2mem.data_rd[7:0];
            end
            2'b01 : begin
               dmem_rdata_byte = dmem2mem.data_rd[15:8];
            end
                        ...
            default : begin
            end
         endcase
      end // MEM_LD_OPS_LB, MEM_LD_OPS_LBU
      MEM_LD_OPS_LH,
      MEM_LD_OPS_LHU : begin
         case (mem2dmem.addr[1])
            1'b0 : begin
               dmem_rdata_hword = dmem2mem.data_rd[15:0];
            end
                     ...
         endcase
      end // MEM_LD_OPS_LH, MEM_LD_OPS_LHU
      MEM_LD_OPS_LW : begin
         dmem_rdata_word = dmem2mem.data_rd;
      end
      default : begin
      end
   endcase // mem_ld_ops
end
```

Listing 5.3. Asynchronous data memory read for load operation.

Next we need to perform either sign- or zero-extension of the received data, before it is put into the destination register, as illustrated below.

```
// Extend the load data for sign/zero
always_comb begin
    case (exe2mem_ctrl.mem_ld_ops)
        MEM_LD_OPS_LB  : mem2wb_data.dmem_rdata = {{24{dmem_rdata_byte[7]}},  dmem_rdata_byte};
        MEM_LD_OPS_LBU : mem2wb_data.dmem_rdata = { 24'b0,                    dmem_rdata_byte};
        MEM_LD_OPS_LH  : mem2wb_data.dmem_rdata = {{16{dmem_rdata_byte[15]}}, dmem_rdata_hword};
        MEM_LD_OPS_LHU : mem2wb_data.dmem_rdata = { 16'b0,                    dmem_rdata_hword};
        MEM_LD_OPS_LW  : mem2wb_data.dmem_rdata = {                           dmem_rdata_word};
        default        : mem2wb_data.dmem_rdata = '0;
    endcase // mem_ld_ops
end
```

Listing 5.4. Sign- or zero-extension of the loaded data.

Store Operation – Memory Access Phase (M)

Listing 5.1 illustrates setting up the signals for data memory access, which are equally applicable for store operation. In addition, the data that is to be stored and the corresponding mask are also prepared for the store operation, which resolves the data size as well as its address location. The code segment below illustrates this aspect of the data memory access.

```
// Prepare the write data and mask for store
always_comb begin
   mem2dmem.data_wr = '0;
   mem2dmem.mask    = '0;

   case (exe2mem_ctrl.mem_st_ops)
      MEM_ST_OPS_SB : begin
         case (mem2dmem.addr[1:0])
            2'b00 : begin
               mem2dmem.data_wr[7:0]   = exe2mem_data.rs2_data[7:0];
               mem2dmem.mask = 4'b0001;
            end
                    ...
            2'b11 : begin
               mem2dmem.data_wr[31:24] = exe2mem_data.rs2_data[31:24];
               mem2dmem.mask = 4'b1000;
            end
            default : begin
            end
         endcase
      end // MEM_ST_OPS_SB
      MEM_ST_OPS_SH : begin
         case (mem2dmem.addr[1])
            1'b0 : begin
               mem2dmem.data_wr[15:0]  = exe2mem_data.rs2_data[15:0];
```

```
                mem2dmem.mask = 4'b0011;
             end
                              ...
          endcase
       end // MEM_ST_OPS_SH
       MEM_ST_OPS_SW : begin
          mem2dmem.data_wr = exe2mem_data.rs2_data;
          mem2dmem.mask = 4'b1111;
       end
       default : begin
          mem2dmem.data_wr = '0;
       end
    endcase // mem_st_ops
end
```

Listing 5.5. Data and mask preparation for store operation.

Finally the data prepared during the memory phase for store operation is sent to the data memory for store operation and is illustrated in the below code segment.

```
// Memory store operation
always_ff @(negedge clk)
begin
   if ( !cs_ff && !wr_ff ) begin
        if (mask_ff[0])
                data_memory[addr_ff][7:0] = data_wr_ff[7:0];
        if (mask_ff[1])
                data_memory[addr_ff][15:8] = data_wr_ff[15:8];
        if (mask_ff[2])
                data_memory[addr_ff][23:16] = data_wr_ff[23:16];
        if (mask_ff[3])
                data_memory[addr_ff][31:24] = data_wr_ff[31:24];
   end
end
```

Listing 5.6. Storing the data synchronously to the data memory.

Load/Store Operation – Decode Phase

The decode operation for load and store instructions use func3 bit-field of the instruction machine code to define the size and type of the data variable that is to be exchanged with the data memory. Recall that load instructions are I-type while store operations follow S-type encoding format. The memory address for load/store operations is constructed using register-offset memory addressing mode. The code listings implementing the load and store operations at instruction decode phase are given in Listing 5.7 and 5.8, respectively.

```
// Load operations
OPCODE_LOAD_INST : begin
    id2exe_ctrl.rd_wb_sel    = RD_WB_MEM;
    id2exe_ctrl.alu_opr1_sel = ALU_OPR1_REG;
    id2exe_ctrl.alu_opr2_sel = ALU_OPR2_IMM;
    id2exe_ctrl.alu_ops      = ALU_OPS_ADD;
    id2exe_ctrl.rd_wr_req    = 1'b1;
     case (funct3_opcode)
               3'b000  : id2exe_ctrl.mem_ld_ops = MEM_LD_OPS_LB;    // Load byte signed
               3'b001  : id2exe_ctrl.mem_ld_ops = MEM_LD_OPS_LH;    // Load halfword signed
               3'b010  : id2exe_ctrl.mem_ld_ops = MEM_LD_OPS_LW;    // Load word
               3'b100  : id2exe_ctrl.mem_ld_ops = MEM_LD_OPS_LBU;   // Load byte unsigned
               3'b101  : id2exe_ctrl.mem_ld_ops = MEM_LD_OPS_LHU;   // Load halfword unsigned
               default : id2exe_ctrl.mem_ld_ops = MEM_LD_OPS_NONE;  // Load type unknown
    endcase // funct3_opcode

end // OPCODE_LOAD_INST
```

Listing 5.7. Instruction decoding for load operation.

```
// Store operations
OPCODE_STORE_INST : begin
    id2exe_ctrl.alu_opr1_sel = ALU_OPR1_REG;
    id2exe_ctrl.alu_opr2_sel = ALU_OPR2_IMM;
    id2exe_ctrl.alu_ops      = ALU_OPS_ADD;
    id2exe_data.imm          = {{21{instr_codeword[31]}}, instr_codeword[30:25],
                                instr_codeword[11:7]};
     case (funct3_opcode)
               3'b000  : id2exe_ctrl.mem_st_ops = MEM_ST_OPS_SB;    // Store byte signed
               3'b001  : id2exe_ctrl.mem_st_ops = MEM_ST_OPS_SH;    // Store halfword signed
               3'b010  : id2exe_ctrl.mem_st_ops = MEM_ST_OPS_SW;    // Store word
               default : id2exe_ctrl.mem_st_ops = MEM_ST_OPS_NONE;  // Store word
    endcase // funct3_opcode

end // OPCODE_STORE_INST
```

Listing 5.8. Instruction decoding for store operation.

Load/Store Operation – Execute Phase

The address generation for load/store operations during the instruction execution phase simply uses the ADD operation for register-immediate offset addressing.

As discussed, a RISC V instruction normally goes through different phases starting with the instruction fetch phase. We will implement the necessary building blocks to perform the actions required at each phase.