**Lab 4 - Adding Data Memory and performing Load and Store Instructions**

**Introduction:** In Part 4A, we will first develop a data memory module and test it for reading and writing. Then in Part 4B, we will combine it with RF + ALU and see whether we can *load* memory data to registers (memory read operation) and in Part 4C, we will *store* register data to memory (memory write operation). For all these, we will be supplying the control signals (as if they are falling from the sky) but in the next lab (Lab 5), we will build the control unit consisting of Program Counter (PC), Instruction Memory (IM) and Instruction Decoder (ID) which can output the control signals correctly. ID is shown as a blue long ellipse and is called as ‘Control’ in figure such as Fig. 4.2, p.259 or with more detail in Fig. 4.17, p. 277. We will not consider CBZ Instruction now (it will be considered in a later Lab).

**4A – Simulating the Data memory**

The size of the Data Memory should be 64 EB (Byte thin, with 64 bits of address). If we look at the format of D type instructions, we find that the bits are arranged as given in Figure 4.14 (p.274), middle format – From left, 11 bits of Opcode, 9 bits of signed displacement (this means plus or minus 256 bytes offset from the base register’s pointed value), 2 bits of 00, 5 bits for pointing the base register (Rn) and 5 bits to point the destination register for Load or to point the source register for Store (Rt).

We will make some simplifying assumptions. We will assume only a Data Memory size of 256 Bytes (a far cry from the possible address space of 16 ExaBytes!). Also let us assume that the base register Rn is initialized to all 0s. (Actually we can use R31 for the base as it is wired permanently as zero) This means that we cannot give any negative address as there is nothing below 0. The 9 bit signed displacement in the instruction will have a zero as the sign bit (first bit) always. As we extend the address to 64 bits, we will get in hex 0000 0000 0000 00 xx. Also the maximum number of double words (64 bits or 8 Bytes) we can store is only 32 (256 B/8 B). Refer to the diagram below. Let us test it for reading and writing.

The memory map for the Data Memory will be as shown:

FFFF FFFF FFFF FF FF

Byte Thin

Not Used

0000 0000 0000 01 00

**RF’**

**(Write)**

(rd)

WriteReg

RegDst

1

0

1

MemtoReg

WriteData

clock

RegWrite

5

32

0000 0000 0000 00 FF

32 Double Words or

256 Bytes

18

10

0000 0000 0000 00 08

First (0th) Double Word

0000 0000 0000 00 00

Data Memory is RWM (Read Write Memory). The inputs are Address (in this case, 8 bits), WriteData (64 bits), MemWrite and MemRead (and the clock\*) and the output is ReadData (64 bits). Typically memory chips are not clocked devices. There will be **no clock pin in a memory chip**. However, in our case, we will write into memory in synchronism with the system clock. This is because when we wrote to Register File, data was written into a register at the positiinve edge of the clock if RegWrite == 1. So we may include clock as an input also to the memory module! (Incidentally, this type of memory is referred to as synchronous write, asynchronous read memory). Writing is timed with the clock. Reading is not dependent on the clock.

Note a difference though. In RF, we had Read Address pins and Write Address pins. For Memory, there will be only *one* set of Address pins. We can either read out a data or write a data with the same address pins. However we need to delineate the function and so we have another input signal called MemRead in addition to MemWrite. (There was no RegRead signal for the RF).

Address

Write data

MemWrite

MemRead

clock

ReadData

A single line describes the 256 B memory:

reg [7:0] DM [0:255]; //DM – Data Memory denoting say, Hard Drive.

Notice the difference in the order for the indices for the width and depth of the memory. This statement says there are 256 registers, each 8 bits wide.

As ReadData will have an assign statement later on, declare it as output ‘reg’. Remember it is a double word (8 bytes)

When there is an address given to the memory, data gets in or out of that location after certain delay called “memory access time”. For the time being, we will ignore this important delay (which is stupid. Access time is an important parameter that defines the ‘goodness’ of any memory). In a later lab, if we need to be rigorous, we may add delay for this and for all the other modules we developed. Students have done such labs too.

We have some complication. Even though memory is addressed byte thin, our WriteData and ReadData buses are 64 bits wide. Memory is assumed to be perfectly aligned in 8 byte boundaries. So when the address is 0, we actually read or write eight consecutive byte locations 0, 1, 2 …and 7).

(Declaring Address as 8 bit input as our memory is only 256 Byte deep)

For read, we may say

always @ (MemRead)

begin

if (MemRead ==1)

assign ReadData = {DM[Address + 0], DM[Address + 1], DM[Address + 2], DM[Address + 3], DM[Address + 4], DM[Address + 5], DM[Address + 6], DM[Address + 7]} ;

end // We are reading consecutive 8 bytes from the address and catenating them to form a double word.

For writing, when MemWrite = 1, we want WriteData to get in to eight memory locations pointed to by the Address[7:0] + 0, Address[7:0] + 1 etc. Since MemWrite will be one only during certain times, let us say that if ‘posedge’ of clock occurs **and** if MemWrite is 1, then memory will be written at that time.

So we may say, for write

always @ (posedge clock)

begin

if (MemWrite ==1)

begin

DM[Address + 0] = WriteData [63:56];

DM[Address +1] = WriteData [55:48];

DM[Address +2] = WriteData [47:40];

DM[Address +3] = WriteData [39:32];

DM[Address + 4] = WriteData [31:24];

DM[Address +5] = WriteData [23:16];

DM[Address +6] = WriteData [15:8];

DM[Address +7] = WriteData [7:0];

end

end // This is “Big Endian”, Higher significance byte is in lower address end.

It is possible to **initialize the memory contents** with some data. Let us initialize a few locations to simulate the nonvolatile hard drive. So we may say

initial

begin

DM[0] = 8’h00; // beginning of 0th double word

DM[1] = 8’h00;

DM[2] = 8’h00;

DM[3] = 8’h00;

DM[4] = 8’h00;

DM[5] = 8’h00;

DM[6] = 8’h00;

DM[7] = 8’h00; // end of 0th double word

//… // no need as we will not use these locations

DM[40] = 8’h55; // This is the beginning of the fifth double word. **Remember Byte addresses are in decimal here** (without 0x prefix).

DM[41] = 8’h55;

DM[42] = 8’h55;

DM[43] = 8’h55;

DM[44] = 8’h55;

DM[45] = 8’h55;

DM[46] = 8’h55;

DM[47] = 8’h55; //This is the end of the fifth double word

//……. // no need as we will not use these locations

DM[80] = 8’haa; //This is the beginning of the tenth (ath) double word

DM[81] = 8’haa;

DM[82] = 8’haa;

DM[83] = 8’haa;

DM[84] = 8’haa;

DM[85] = 8’haa;

DM[86] = 8’haa;

DM[87] = 8’haa; //This is the end of the tenth (ath) double word

//… // no need as we will not use these locations

end

In the test fixture module, let us have clock as usual to time everything. Remember all signal changes should occur **prior** to positive edge of clock and remain steady for a while (“set up” and “hold” time requirement). First let us test the read operation. Keep MemRead = 1, MemWrite = 0, give addresses as 40 and then as 80 and see whether the ReadData is right.

Then keep MemWrite =1 and MemRead = 0, address as 40 and WriteData as 64’haaaaaaaaaaaaaaaa and see whether 5555555555555555 is overwritten by aaaaaaaaaaaaaaaa (after clock edge) by displaying ReadData at that address.

mc, s17