Labs 9 & 10: Simple Cache and Cache Controller Implementation

Objectives

The objective of this lab is to implement a simple cache system and integrate it with the RISC-V processor you previously implemented.
This lab is organized as follows:

1. An introduction
2. Experiments to be conducted
3. Lab Report Requirements

Introduction

In this lab, we will work on implementing a simple caching system for the RISC-V processor. For simplicity, we will integrate the caching system with the **single-cycle implementation**. Additionally, we assume the following:

- Only data memory will be cached. The instruction memory will not be affected.
- We will have only one level of caching.
- The main memory module is assumed to have a capacity of 4 Kbytes (word addressable using 10 bits or byte addressable using 12 bits)
- Main memory access (for read or write) takes 4 clock cycles
- The data cache geometry is (512, 16, 1). This means that the total cache capacity is 512 bytes, that each cache block is 16 bytes (implying that the cache has 32 blocks in total), and that the cache uses direct mapping.
- The cache uses write-through and write-around policies for write hit and write miss handling and no write buffers exist. This implies that all SW instructions need to stall the processor.
- LW instructions will only stall the processor in case of a miss.

In order to build the caching system, we are going to replace the data memory in the single cycle implementation with a data memory system module that includes a cache memory module, a cache controller module, and the data memory module. The new memory system module is shown in Figure 1. The memory system has an extra control signal called `stall`. The `stall` signal is asserted when the memory system needs to temporarily stop the processor. The `stall` signal will remain asserted until the processor is allowed to resume normal execution.

The cache controller encapsulates the array of tags and valid bits and uses the index and tag parts of the requested memory address to decide whether there is a hit or a miss. It is also responsible for generating the `stall` control signal in addition to controlling both the cache module and the memory module as explained in the 4 scenarios below:

- The processor requests a **read operation** (executing a LW instruction) and the cache controller decides that it is a **hit**. In this case, there is no stall necessary and the data is read from the cache module.

- The processor requests a **read operation** (again executing a LW instruction) and the cache controller decides that it is a **miss**. In this case, the `stall` signal is asserted and the data is read from the data memory module which provides 1 block (16 bytes or 128 bits) of data to the data cache. When this data is available, the data memory module asserts a `ready` signal that the cache controller uses to ask the data cache to fill the corresponding block with the data coming from the memory and to de-assert the `stall` signal.
- The processor requests a **write operation** (executing a SW instruction) and the cache controller decides that it is a **hit**. In this case, the word to be stored has to be written both in the cache memory and in the data memory (due to the **write-through policy**). So the cache controller asserts the `stall` signal until the memory confirms that it finished writing via its `ready` signal. Simultaneously the cache controller asks the cache memory to update the value.
- The processor requests a **write operation** (again executing a SW instruction) and the cache controller decides that it is a **miss**. In this case, the word to be stored is written in the data memory only (due to the **write-around policy**); however, in this case too, the cache controller asserts the `stall` signal until the memory finishes the storing.

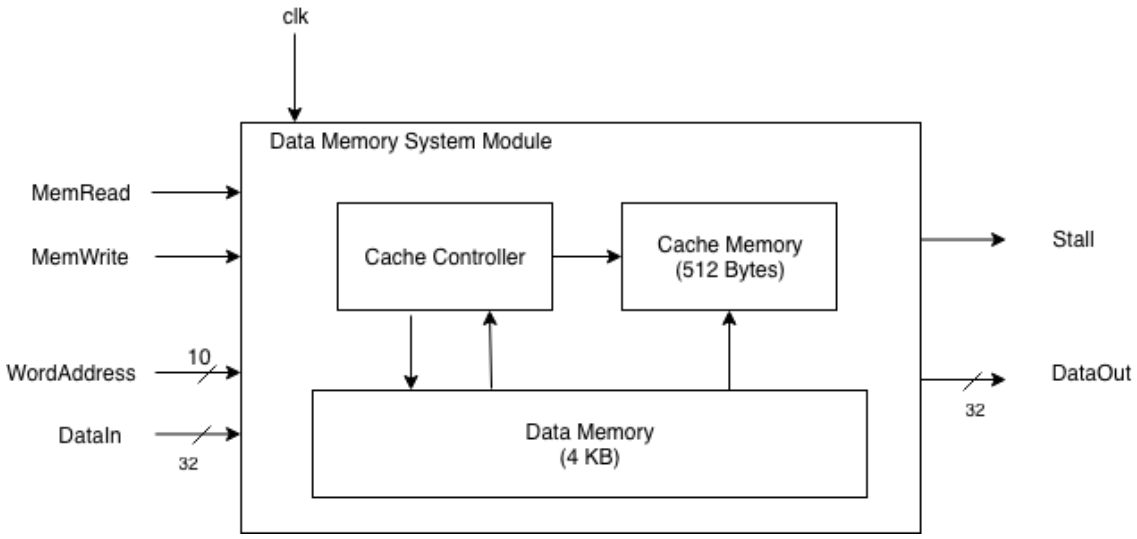

Figure 1 Data Memory System

In this lab we are going to conduct the following experiments:

1. Update and Simulate the Data Memory Module
2. Design and implement the Cache Memory Module
3. Design and implement the Cache Memory Controller Module
4. Integrate with RISC-V implementation

Experiments

Experiment 1: Update and Simulate the Data Memory Module

The data memory should be modified for the caching system we want to build. For convenience you can find here the memory module we used in labs 5 & 6 to implement the RISC-V processor:

```
module DataMem (input clk, input MemRead, input MemWrite,
                input [5:0] addr, input [31:0] data_in,
                output [31:0] data_out);

    reg [31:0] mem [0:63];
    always  @(posedge   clk)
    begin
        if (MemWrite)
            mem[addr] <= data_in;
    end
    assign  data_out = MemRead?mem[addr]:0;
    initial begin
        mem[0]=32'd17;
        mem[1]=32'd9;
        mem[2]=32'd25;
    end
endmodule
```

Here is the updated memory module to be used for the caching system:

```
module DataMem
    (input clk, input MemRead, input MemWrite,
     input [9:0] addr, input [31:0] data_in,
     output reg [127:0] MsDataOut, output reg Msready);

    reg [31:0] mem [0:1023];  // 4KB memory

    wire [9:0] bAddr;
    assign bAddr = {addr[9:2],2'b00};

    always  @(posedge clk)
    begin
        Msready <= 0;
        if (MemWrite) begin
            repeat (3) begin        // Wait for 3 clock cycles
                @ (posedge clk);
            end
            mem[addr] <= data_in;
            Msready <= 1;
        end

        else if (MemRead) begin
            repeat (3) begin
                @ (posedge clk);
            end
            MsDataOut <={mem[bAddr],mem[bAddr+1],mem[bAddr+2],mem[bAddr+3]};
            Msready <= 1;
        end
    end

    initial begin
        mem[0]=32'd17;
        mem[1]=32'd9;
        mem[2]=32'd25;
    end
endmodule
```

Carefully inspect the differences between both versions and make sure you understand the reason behind each of the changes. The justification for each will be included in your lab report.
Simulate the updated memory module by writing a testbench Verilog module.

Experiment 2: Design and implement the Cache Memory Module

The cache memory module is essentially a small memory. The cache memory module should have the following port declaration:

```
module DataCache (
    input clk,
    input update,
    input fill,
    input [1:0] wordoffset,
    input [4:0] index,
    input [31:0] data_in,
    output [31:0] data_out,
    input [127:0] MsDataIn);
endmodule
```

To facilitate the indexing of cache and the selection of one word within a block using the offset bits, the cache storage can be defined inside the DataCache module as a 2D array of words as follows:

```
    reg [31:0] cache [0:31][0:3];
```

Implement and simulate the Cache Memory module by writing a testbench Verilog module.

Experiment 3: Design and implement the Cache Memory Controller Module

The cache memory controller should have the following port declaration:

```
module DataCacheController (
    input clk,
    input MemRead,
    input MemWrite,
    input [4:0] index,
    input [2:0] tag,
    input MsReady,
    output reg stall,
    output fill,
    output update,
    output MsRead);
endmodule
```

where:

- **MsReady**: is the control signal coming from the memory to indicate its readiness (either after finishing a read or a write).
- **MsRead**: is the control signal sent to the memory to request the reading of an entire block. This happens only on a read miss.
- **fill**: is the control signal sent to the cache memory to request that it fills one of its blocks (the one corresponding to the index bits which are also provided as input to the cache memory module) with the data coming from the main memory. This happens only on a read miss as soon as the **MsReady** is asserted.
- **update**: is the control signal sent to the cache memory to request that it updates a single word based on data coming from the processor due to a store instruction. This happens only on a write hit.

Note that in order to decide whether the access is a hit or a miss, the cache controller has to be provided with the index and the tag of the address being accessed and it also needs to have access to the array of tags and the array of valid bits corresponding to the cache blocks in the cache module. Since the cache has 32 blocks, these can be defined inside the cache controller as follows:

```
    reg [2:0] tags [0:31];
    reg valid_bits [0:31];
```

The **valid_bits** array needs to be initialized to zeros to indicate a cold cache start. This can be done using an initial block. Both arrays need to be updated when a new block is cached (at the rising edge when the **fill** signal is asserted).

To function properly, the cache controller needs to implement a finite state machine. The finite state machine can have 3 states: idle, reading, and writing. Initially the controller is in Idle state and it moves to either the reading or the writing state if there is a read miss or any kind of write respectively. Once the corresponding operation is finished, the controller reverts back to its original idle state. These states will help you assert the **stall** signal at the right timing.

Experiment 4: Integrate with single-cycle RISC-V Implementation

To integrate with the single-cycle RISC-V implementation, you need to define the Data Memory System Module first. It should have the following port declaration:

```
module DataMemSys (
    input clk,
    input MemRead,
    input MemWrite,
    input [9:0] addr,
    input [31:0] data_in,
    output [31:0] data_out,
    output stall);
endmodule
```

The **stall** signal should be used to prevent the PC from being updated (via the register load signal or via a mux to update or retain old value) whenever asserted.

Bonus Experiment: 2-Way Set Associative Cache

Replace the implemented cache with a cache having the following geometry (512, 16, 2). Everything else should remain the same.

Deliverables

General Report Submission Notes:

- Each student is required to submit an individual report.
- The submission should be composed of **two files only**:
  1. YourID_Report.pdf/doc:
     A single pdf/word file having the solution of the lab report questions in correct sequence.
  2. YourID_labwork.zip/rar/…:
     A zipped folder containing your codes for all the lab experiments organized in sequence.
- **Submission deadline**: half an hour before the start of the following week's session.

Lab 9 Report [10 pts]

1. [0 pts] Your name and student ID.
2. [2 pts] A technical summary of experiments 1 and 2 as conducted in the lab (steps, results, components, code functionality, etc.)
3. [1 pts] A listing of all the differences between the original data memory module and the updated one with a clear justification of each change.
4. [1 pts] Testbench module for the updated memory and snapshots of the simulation results.
5. [2 pts] Verilog code for the Cache module and its testbench along with snapshots of the simulation results.
6. [2 pts] An FSM diagram for experiment 3 clearly showing the state transitions and the input and output values (can be drawn by hand).
7. [2 pts] A full diagram of the data memory system that includes the data memory, cache memory, and cache memory controller (similar to Figure 1 here) clearly showing the exact inputs/outputs of each module and the flow of signals in between (can be drawn by hand).

Lab 10 Report [10 pts]

1. [0 pts] Your name and student ID.
2. [2 pts] A technical summary of experiments 3 and 4 as conducted in the lab (steps, results, components, code functionality, etc.)
3. [2 pts] Verilog code for the Cache Controller module
4. [2 pts] Verilog code for the Data Memory System module
5. [2 pts] Verilog code for the top-level module of the RISC-V module after integration with the data memory system module
6. [2 pts] Snapshots of the simulation output (waveforms) clearly showing the caching system in action. You must include a description of what each snapshot is illustrating.
7. [**Bonus**: 5 pts] Verilog code for the 2-way set associative data memory system, a technical summary of the experiment and the code functionality, and snapshots of the simulation output using it.
