

Name:  
Student ID:

10.01.2023

**CENG311 Final Exam - Fall 2022**  
**(120 minutes)**

**Q.1 (20 Points)** Assume we have 4-bit memory (word) addresses to load data from memory. Consider that we have two distinct systems: 1) direct-mapped cache with 4 one-word blocks, and 2) 2-way set-associative cache with the same size (initially empty). Answer the following questions:

**a)** Give one simple example scenario with four memory accesses supporting the following statement: Set associative caches reduce cache miss rates compared to direct-mapped caches (Direct-mapped Miss Rate > Set Ass. Miss Rate). Show the final content of the cache.

Access	Address	Hit/Miss for direct-mapped	Hit/Miss for set ass.
1	----	—	—
2	----	—	—
3	----	—	—
4	----	—	—

Direct-mapped Miss Rate = \_\_\_\_ Set Ass. Miss Rate = \_\_\_\_

**b)** Give one simple example scenario with four memory accesses supporting the following statement: Set associative caches have no effect on cache miss rates compared to direct-mapped caches (Direct-mapped Miss Rate = Set Ass. Miss Rate). Show the final content of the cache.

Access	Address	Hit/Miss for direct-mapped	Hit/Miss for set ass.
1	----	—	—
2	----	—	—
3	----	—	—
4	----	—	—

Direct-mapped Miss Rate = \_\_\_\_ Set Ass. Miss Rate = \_\_\_\_

Assume that you are required to extend the single-cycle MIPS implementation so that it handles the following R-type instruction:

The instruction reads the data from the memory address  $\text{Reg}[\text{Rt}] + \text{Reg}[\text{Rs}]$ , which is the summation of the values in the register Rt and Rs.

The diagram illustrates the MIPS architecture with the following components and connections:

- PC (Program Counter):** Provides the address to Instruction Memory.
- Instruction Memory:** Outputs instruction data based on the PC address. The instruction is split into fields: [31-0], [25-21], [20-16], [15-11], and [5-0].
- Control:** Receives the instruction fields [31-26] and [25-21] to generate control signals: RegDst, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, and RegWrite.
- Registers:** A set of 32 registers. Read register 1 and 2 output Read data 1 and 2. The Write register receives data from the ALU result and the Sign-extend block. The Write data is the 16-bit value from the instruction [15-0].
- Sign-extend:** Takes the 16-bit instruction [15-0] and extends it to 32 bits.
- ALU:** Performs operations on Read data 1, Read data 2, and the Sign-extended value. It uses ALUOp and ALUSrc control signals. The ALU result is sent to the Mux 1 and the ALU control block.
- Mux 1 (Multiplexer 1):** Selects between Read data 1, Read data 2, and the ALU result based on the RegWrite control signal. Its output goes to the PC and the Mux 0.
- Mux 0 (Multiplexer 0):** Selects between the ALU result and the Mux 1 output based on the MemtoReg control signal. Its output goes to the Data Memory.
- Data Memory:** Outputs Read data and Write data based on the Address and Write data inputs.
- ALU control:** Receives the ALUOp and ALUSrc control signals to determine the ALU operation.
- Shift left 2:** A barrel shifter that shifts the ALU result left by 2 bits, used for branch targets.
- Add ALU result:** An adder that adds the ALU result to the shifted branch target (from Shift left 2) to calculate the next PC value.
- Mux 1 (Multiplexer 1):** Selects between the ALU result and the shifted branch target based on the Branch control signal.

- b)** For the `lwr` instruction (with the opcode `0hex` and function code `6hex`), modify the Verilog code inside **control module** below (already given in your lab). Make sure that all the existing instructions still work correctly after your modifications.

```

module control(in, regdest, alusrc, memtoreg, regwrite, memread, memwrite, branch, aluop1,
aluop2);
    input [5:0] in;
    output regdest, alusrc, memtoreg, regwrite, memread, memwrite, branch, aluop1, aluop2;
    wire rformat,lw,sw,beq;
    assign rformat = ~| in;
    assign lw = in[5]& (~in[4])&(~in[3])&(~in[2])&in[1]&in[0];
    assign sw = in[5]& (~in[4])&in[3]&(~in[2])&in[1]&in[0];
    assign beq = ~in[5]& (~in[4])&(~in[3])&in[2]&(~in[1])&(~in[0]);
    assign regdest = rformat;
    assign alusrc = lw|sw;
    assign memtoreg = lw;
    assign regwrite = rformat|lw;
    assign memread = lw;
    assign memwrite = sw;
    assign branch = beq;
    assign aluop1 = rformat;
    assign aluop2 = beq;
endmodule

```

**Q.3 (20 Points)** The following MIPS code could be used to increment the value of two integers that are in the main memory.

```
1: li $t0,1 # load 1 into $t0
2: lw $t1,100($t5) # load first variable
3: add $t1,$t0,$t1 # increment
4: sw $t1,100($t5) # store
5: lw $t2,104($t5) # load 2nd variable
6: add $t2,$t0,$t2 # increment
7: sw $t2,104($t5) # store
```

**a)** Assume that the code is executed on a 5-stage pipelined MIPS processor (with the stages IF, ID, EX, MEM and WB). By considering the instructions between 2-7, draw the pipeline charts for the following cases. (Whenever a stage gets stalled, fill that stage with an “S”. It is allowed to write and read a register at the same cycle. If data forwarding is allowed, show the forwarding.)

**a.1)** Assume that data forwarding is not allowed.

**a.2)** Assume that data forwarding is allowed.

**b)** Without changing what the code does, reorder the instructions to reduce the number of stalls. Draw the pipeline chart for the new order by showing the stages and forwarding (if necessary) clearly.

#### Q.4 (30 Points)

```
1:    float dotprod(float x[8], float y[8])
2:    {
3:        float sum = 0.0;
4:        int i;
5:        for (i = 0; i < 8; i++)
6:            sum += x[i] * y[i];
7:        return sum;
8:    }
```

a) Does the function above have good spatial locality with respect to x and y arrays? Explain why.

b) With the following assumptions:

- Floats are 4 bytes, that x is loaded into the 32 bytes of contiguous memory starting at address 0 and that y starts immediately after x at address 32.
- A direct-mapped cache block is 16 bytes (big enough to hold four floats) and a total cache size of 32 bytes.
- The variables sum and i are stored in a CPU register and thus do not require a memory reference.

For each memory reference (first two are given below) in the code, identify it as a cache hit or miss. Show the intermediate steps and the final content of the cache.

Cache content after access to x[0]: **M**

x[0] x[1] x[2] x[3]

Cache content after access to y[0]: **M**

y[0] y[1] y[2] y[3]

c) If instead of defining  $x$  to be `float x[8]`, we define it to be `float x[12]` and assume  $y$  starts immediately after  $x$  in memory. For the same memory references in the code (still accessing only 8 elements of the arrays), does it help to improve the miss rates? (**Hint:** Consider the cache-memory mappings for `x[8]` and `x[12]` cases.)