CS224 - Spring 2023 -

MIPS Single-Cycle Datapath and Controller

Lab-04

Section 5

Tolga Han Arslan

22003061

Part 1. Setup and Test – Original10 (20 points)

a) **[3 Points]** Determine the assembly language equivalent of the machine codes given in the imem module in the "Complete MIPS model.txt" file posted on Moodle for this lab.

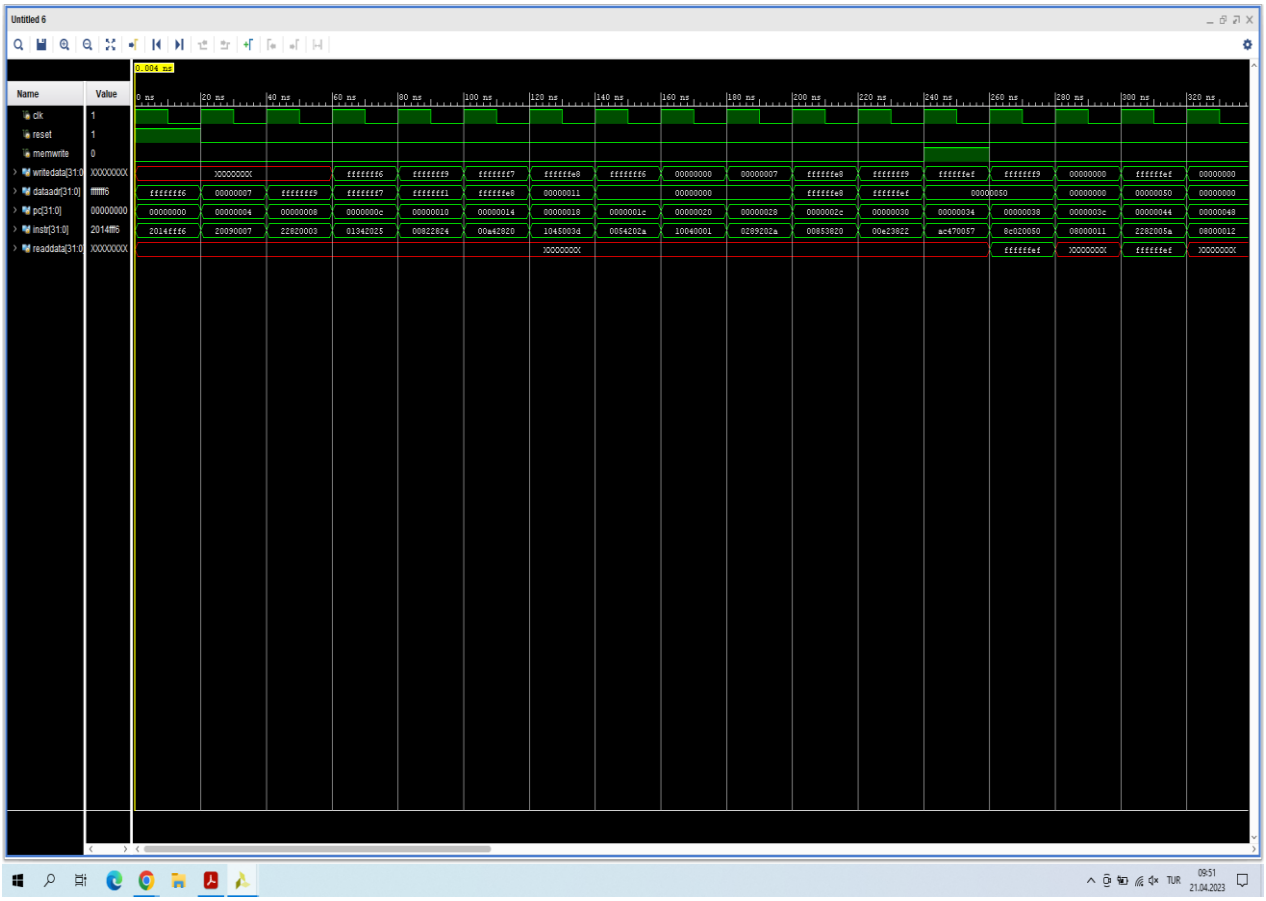
Location (hex)	Machine Instruction (hex)	Assembly Equivalent (hex)
00	2014fff6	addi $s4, $zero, -10
04	20090007	addi $t1, $zero, 7
08	22820003	addi $v0, $s4, 3
0C	01342025	or $a0, $t1, $s4
10	00822824	and $a1, $a0, $v0
14	00a42820	add $a1, $a1, $a0
18	1045003d	beq $v0, $a1, 61
1C	0054202a	slt $a0, $v0, $s4
20	10040001	beq $zero, $a0, 1
24	00002820	add $a1, $zero ,$zero
28	0289202a	slt $a0, $s4, $t1
2C	00853820	add $a3, $a0, $a1
30	00e23822	sub $a3, $a3, $v1
34	ac470057	sw $a3, 87($v0)
38	8c020050	lw $v0, 80($zero)
3C	08000011	j 0x00000044
40	20020001	addi $v0, $zero, 1
44	2282005a	addi $v0, $s4, 90
48	08000012	j 0x00000048

d) **[7 Points]** Now make a SystemVerilog testbench file and using Xilinx Vivado, simulate your MIPS-lite processor executing the test program. In your testbench, do not forget to reset your processor at the beginning.

```
module top_tb();
    logic clk;
    logic reset;
    logic memwrite;
    logic [31:0] writedata, dataadr, pc, instr, readdata;

    top dut(clk, reset, writedata, dataadr, pc, instr, readdata, memwrite);

    initial
        begin
        reset <= 1; #20;
        reset <= 0;
        end
    always
        begin
        clk <= 1; #10;
        clk <= 0; #10;
        end
```



e) [5 Points] Make some observations on the waveform that you generated. These questions assume that you have used imem module as is without modifying any instruction there.

i) In an R-type instruction what does writedata correspond to?

In R-type instructions, writedata corresponds to the output of RD2 port of register file corresponds to RF[rt] which is determined by r-type instruction's rt field [20:16].

ii) Why is writedata undefined for some of the early instructions in the program?

It is undefined for first 3 instructions in the table since they are trying to read RF[rt] values which are not initialized yet. Specifically, RF[rt] registers for first three instructions are $s4, $t1, and $v0 whose values are undefined initially.

iii) In which instructions memwrite becomes 1?

For sw instructions memwrite becomes 1 since it is the control signal which determines whether or not to write to the data memory.

iv) Why is dataaddr 0xfffffe8 when PC is 0x14?

When PC is 0x14, corresponding instruction's assembly equivalent is add $a1, $a1, $a0 and dataaddr corresponds to address port of the data memory. $a0 and $a1's initial values can be determined by looking into previous instructions. According to them, $a0 is the result of the operation (-10 or 7) = (0xfffffff6 or 0x00000007) = 0xfffffff7 ,and $a1 is the result of the operation (0xfffffff7 and 0xfffffff9) = 0xfffffff1. Hence, ALUResult which corresponds to dataaddr is 0xfffffff7 + 0xfffffff1 = (-9) + (-15) = -24 = 0xffffffe8 in hexadecimal form.

v) In the example program, when is the output of readdata defined and why?

It is defined when PC is at 0x38 and 0x44 since the instruction at 0x38 lw $v0, 80($zero) initializes readdata.

f) **[5 Points]** Modify the ALU module in "Complete MIPS model.txt" so that when alucont = 011, ALU computes the result of a ^ b (a XOR b). Put the modified ALU module in your report.

```
module alu(input  logic [31:0] a, b,
           input  logic [2:0]  alucont,
           output logic [31:0] result,
           output logic zero);
   always_comb
      case(alucont)
         3'b010: result = a + b;
```

```
        3'b110: result = a - b;
        3'b000: result = a & b;
        3'b001: result = a | b;
        3'b111: result = (a < b) ? 1 : 0;
        3'b011: result = a ^ b;
        default: result = {32{1'bx}};
    endcase
  assign zero = (result == 0) ? 1'b1 : 1'b0;
endmodule
```

Part 2. Extend Original10 (30 points)

a) [3 Points] Register Transfer Level -Language- (RTL) expressions for the new instructions that you are adding (see Table 3), including the fetch and the updating of the PC.

lh:

IM[PC]

RF[rt] ← {16'b0, DM[RF[rs] + SignExtImm](15:0)}

PC ← PC + 4

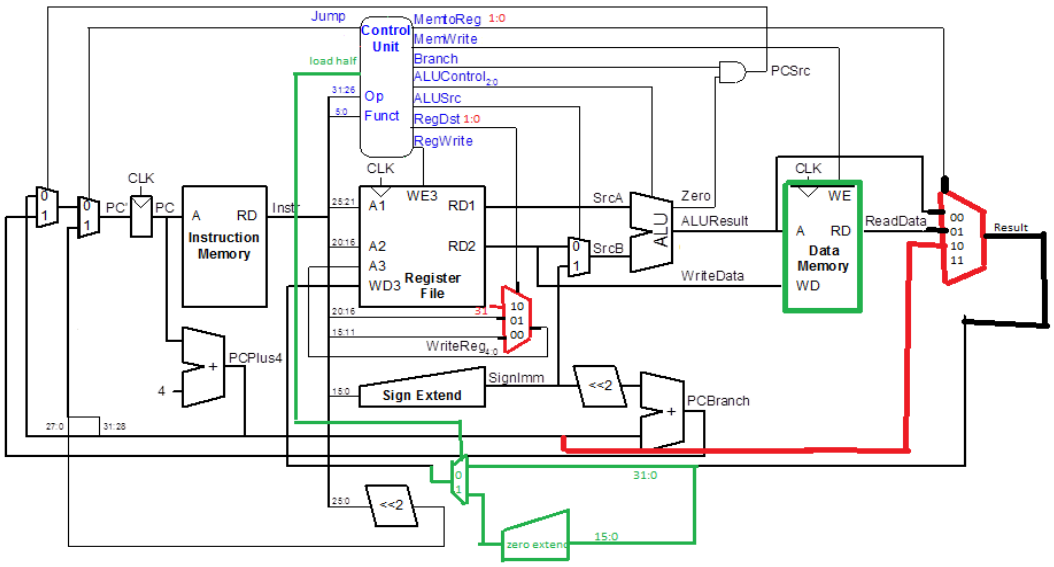
jal:

IM[PC]

RF[31] ← PC + 4

PC ← {PC+4[31:28], address, 2'b0}

b) **[15 Points]** Make any additions or changes to the datapath which are needed in order to make the RTLs for the instructions possible. The base datapath should be in black, with changes for marked in red and other colors (one color per new instruction). **Make your changes on "Final Datapath.png" file.**



Modified Single Cycle Processor, RED: jal instruction GREEN: lh instruction

For lh (load half) instruction, data memory is modified in order to fetch half words(16 bit) from it. If the immediate in the instruction is word aligned (multiple of 4) it fetches usual 32 bit word from data memory to readdata, and if the immediate in the instruction is half word aligned (multiple of 2 and not 4) it fetches the most significant 16 bits from the current word and least significant 16 bits from the upper word to readdata. Then, the zero-extender component converts the most significant 16 bits of result to 0 and write it to the RF[rt] by the help of a multiplexer with new select signal lh. If the instruction is not lh, this multiplexer produces usual 32 bit readdata result to write it to the destination register.

For jal (jump and link) instruction, 2 multiplexers in the datapath is modified to be 4-1 multiplexers in order to implement jal. The multiplexer after the datapath selects between the aluResult, readdata, and pcBranch values (used to save $ra register). And multiplexer controlled by regdst signal is modified to select between rt and rd fields of instruction along with the constant 31(indicates the position of $ra register in RF). Select signals of these two registers (memtoreg and regdst) are also modified to 1 bit to 2 bits in order to be used in 4x1 multiplexers. Finally controller module is modified to implement these changes.

c) **[12 Points]** Make a new row in the main decoder table for each new instruction being added, and if necessary, add new columns for any new control signals that are needed (input or out-put). You can use any opcode value you want unless it conflicts with the existing opcodes in the Ta-ble 1. Be sure to completely fill in the table—all values must be specified. **Make your changes on Table 1: Main Decoder for Original10.** If any changes are needed in the ALU decoder table (Table 2), give this table in its new form (with new rows, columns, etc). The base table should be in black, with changes marked in red and other colors. {Note: if you need new ALUOp bits to encode new values, you should also give a new version of Table 2, showing the new encoding}

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemToReg	ALUOp	Jump	Load half
R-type	000000	1	00	0	0	0	00	10	0	0
lw	100011	1	01	1	0	0	01	00	0	0
sw	101011	0	XX	1	0	1	XX	00	0	0
beq	000100	0	XX	0	1	0	XX	01	0	0
addi	001000	1	01	1	0	0	00	00	0	0
j	000010	0	XX	X	X	0	XX	XX	1	0
lh	100001	1	01	1	0	0	01	00	0	1
jal	000011	1	10	X	X	0	10	XX	1	0

ALUOp	Funct	ALUControl
00	X	010 (add)
01	X	110 (subtract)
1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)
1X	100100 (and)	000 (and)
1X	100101 (or)	001 (or)
1X	101010 (slt)	111 (set less than)
1X	100110 (xor)	011 (xor)