University of Nevada Las Vegas. Department of Electrical and Computer Engineering Laboratories.

Class:	**CPE300L Digital Systems Architecture and Design 1001**	Semester:	**Fall 2024**
Points		Document author:	**Narek Kalikian**
		Author's email:	**kalikn1@unlv.nevada.edu**
		Document topic:	**Postlab 7**
Instructor's comments:			

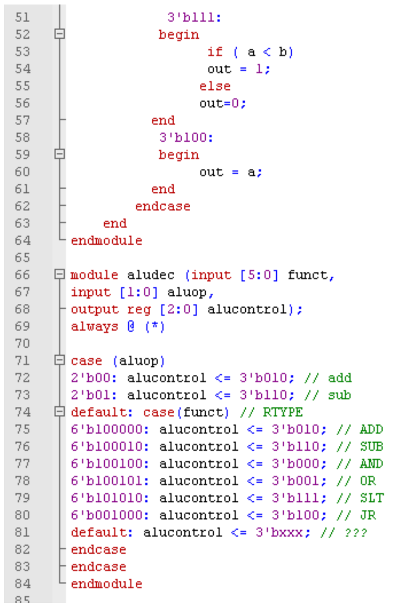
1. Introduction / Theory of Operation

In this lab, we gained important experience designing and implementing a complete single-cycle processor with a subset of MIPS instructions. We added jump register and jump-and-link instructions to the single-cycle instruction set and learned how different components of the processor such as the datapath, controller, and decoders need to be changed to allow for these additional instructions.

2. Description of Experiments

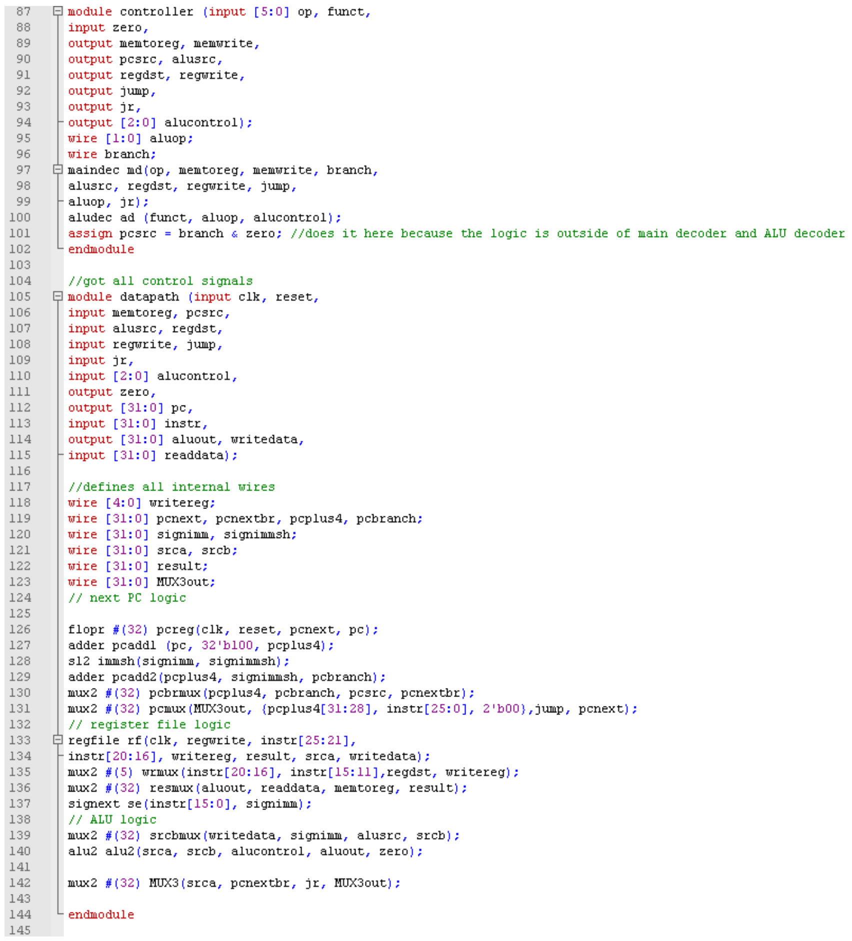
Experiment 1 - Changes Made to Verilog Code:

- *Added JR case based on necessary alucontrol value*



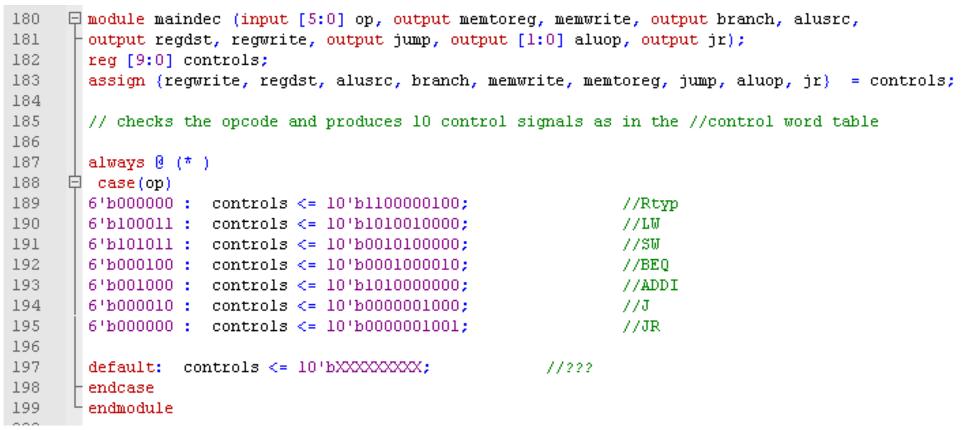
```verilog
              3'b111:
             begin
                   if ( a < b)
                   out = 1;
                  else
                  out=0;
           end
             3'b100:
             begin
                  out = a;
           end
         endcase
     end
endmodule

module aludec (input [5:0] funct,
input [1:0] aluop,
output reg [2:0] alucontrol);
always @ (*)

case (aluop)
2'b00: alucontrol <= 3'b010; // add
2'b01: alucontrol <= 3'b110; // sub
default: case(funct) // RTYPE
6'b100000: alucontrol <= 3'b010; // ADD
6'b100010: alucontrol <= 3'b110; // SUB
6'b100100: alucontrol <= 3'b000; // AND
6'b100101: alucontrol <= 3'b001; // OR
6'b101010: alucontrol <= 3'b111; // SLT
6'b001000: alucontrol <= 3'b100; // JR
default: alucontrol <= 3'bxxx; // ???
endcase
endcase
endmodule
```

- *Added JR signal to controller and datapath port lists. Added one new MUX, rewired existing connections, and added one new wire for the output of the new MUX.*



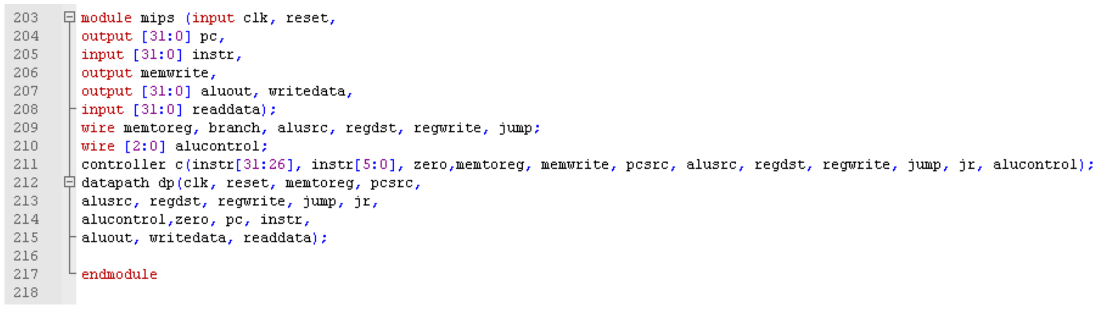
```verilog
module controller (input [5:0] op, funct,
input zero,
output memtoreg, memwrite,
output pcsrc, alusrc,
output regdst, regwrite,
output jump,
output jr,
output [2:0] alucontrol);
wire [1:0] aluop;
wire branch;
maindec md(op, memtoreg, memwrite, branch,
alusrc, regdst, regwrite, jump,
aluop, jr);
aludec ad (funct, aluop, alucontrol);
assign pcsrc = branch & zero; //does it here because the logic is outside of main decoder and ALU decoder
endmodule

//got all control signals
module datapath (input clk, reset,
input memtoreg, pcsrc,
input alusrc, regdst,
input regwrite, jump,
input jr,
input [2:0] alucontrol,
output zero,
output [31:0] pc,
input [31:0] instr,
output [31:0] aluout, writedata,
input [31:0] readdata);

//defines all internal wires
wire [4:0] writereg;
wire [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
wire [31:0] signimm, signimmsh;
wire [31:0] srca, srcb;
wire [31:0] result;
wire [31:0] MUX3out;
// next PC logic

flopr #(32) pcreg(clk, reset, pcnext, pc);
adder pcadd1 (pc, 32'b100, pcplus4);
sl2 immsh(signimm, signimmsh);
adder pcadd2(pcplus4, signimmsh, pcbranch);
mux2 #(32) pcbrmux(pcplus4, pcbranch, pcsrc, pcnextbr);
mux2 #(32) pcmux(MUX3out, {pcplus4[31:28], instr[25:0], 2'b00},jump, pcnext);
// register file logic
regfile rf(clk, regwrite, instr[25:21],
instr[20:16], writereg, result, srca, writedata);
mux2 #(5) wrmux(instr[20:16], instr[15:11],regdst, writereg);
mux2 #(32) resmux(aluout, readdata, memtoreg, result);
signext se(instr[15:0], signimm);
// ALU logic
mux2 #(32) srcbmux(writedata, signimm, alusrc, srcb);
alu2 alu2(srca, srcb, alucontrol, aluout, zero);

mux2 #(32) MUX3(srca, pcnextbr, jr, MUX3out);

endmodule
```

- *Added JR to port list of main decoder and control assignment statement (JR assigned as last control bit). Changed the width of the "controls" register from [8:0] to [9:0] and also reflected that in the op case statement (updated from 9'b… to 10'b…). Added the necessary case for JR with opcode 000000*



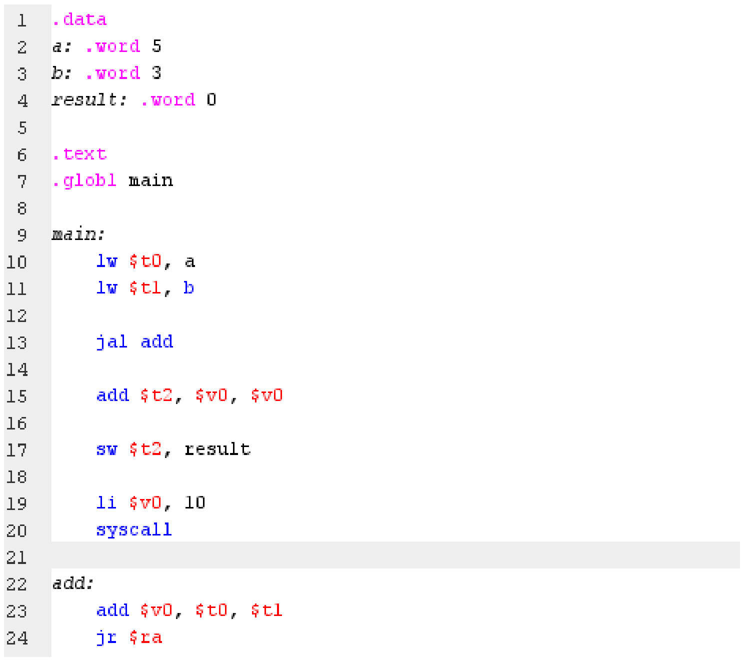
```
module maindec (input [5:0] op, output memtoreg, memwrite, output branch, alusrc,
output regdst, regwrite, output jump, output [1:0] aluop, output jr);
reg [9:0] controls;
assign {regwrite, regdst, alusrc, branch, memwrite, memtoreg, jump, aluop, jr}  = controls;

// checks the opcode and produces 10 control signals as in the //control word table

always @ (* )
 case(op)
6'b000000 :  controls <= 10'b1100000100;              //Rtyp
6'b100011 :  controls <= 10'b1010010000;              //LW
6'b101011 :  controls <= 10'b0010100000;              //SW
6'b000100 :  controls <= 10'b0001000010;              //BEQ
6'b001000 :  controls <= 10'b1010000000;              //ADDI
6'b000010 :  controls <= 10'b0000001000;              //J
6'b000000 :  controls <= 10'b0000001001;              //JR

default:  controls <= 10'bXXXXXXXXX;           //???
endcase
endmodule
```

- *Added JR to port lists of controller c and datapath dp instantiations*

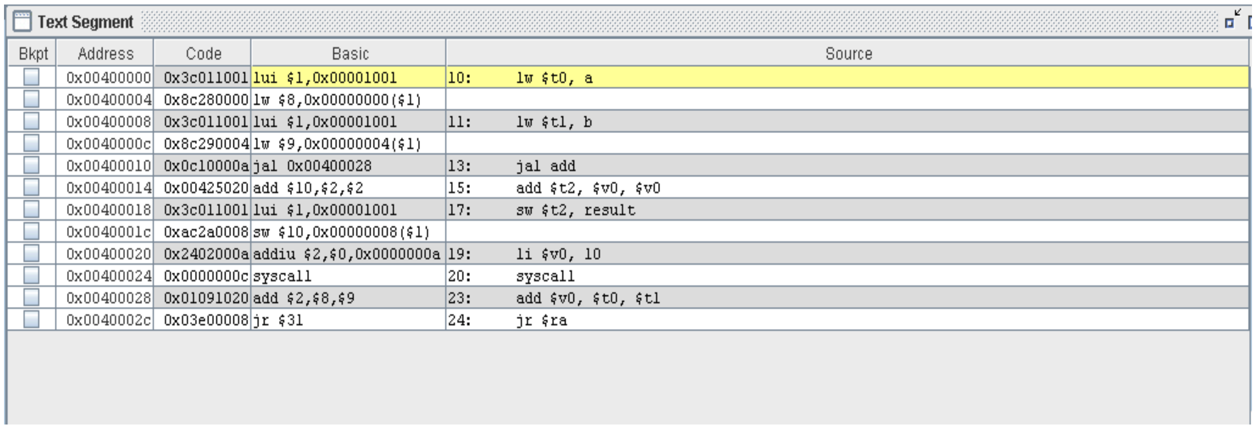


```
module mips (input clk, reset,
output [31:0] pc,
input [31:0] instr,
output memwrite,
output [31:0] aluout, writedata,
input [31:0] readdata);
wire memtoreg, branch, alusrc, regdst, regwrite, jump;
wire [2:0] alucontrol;
controller c(instr[31:26], instr[5:0], zero,memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, jump, jr, alucontrol);
datapath dp(clk, reset, memtoreg, pcsrc,
alusrc, regdst, regwrite, jump, jr,
alucontrol,zero, pc, instr,
aluout, writedata, readdata);

endmodule
```

Experiment 2 - Assembly Code & Run:



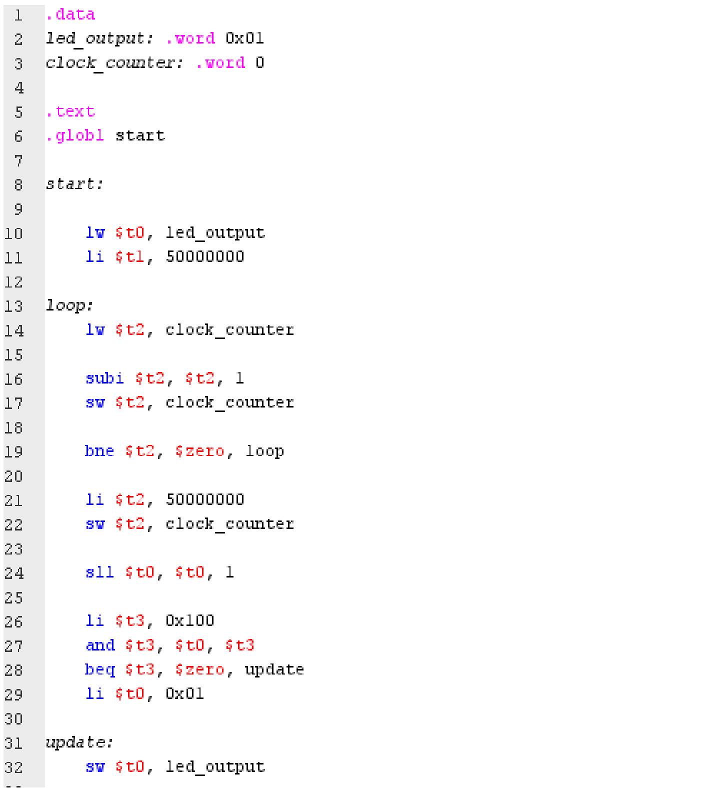
```
.data
a: .word 5
b: .word 3
result: .word 0

.text
.globl main

main:
    lw $t0, a
    lw $t1, b

    jal add

    add $t2, $v0, $v0

    sw $t2, result

    li $v0, 10
    syscall

add:
    add $v0, $t0, $t1
    jr $ra
```



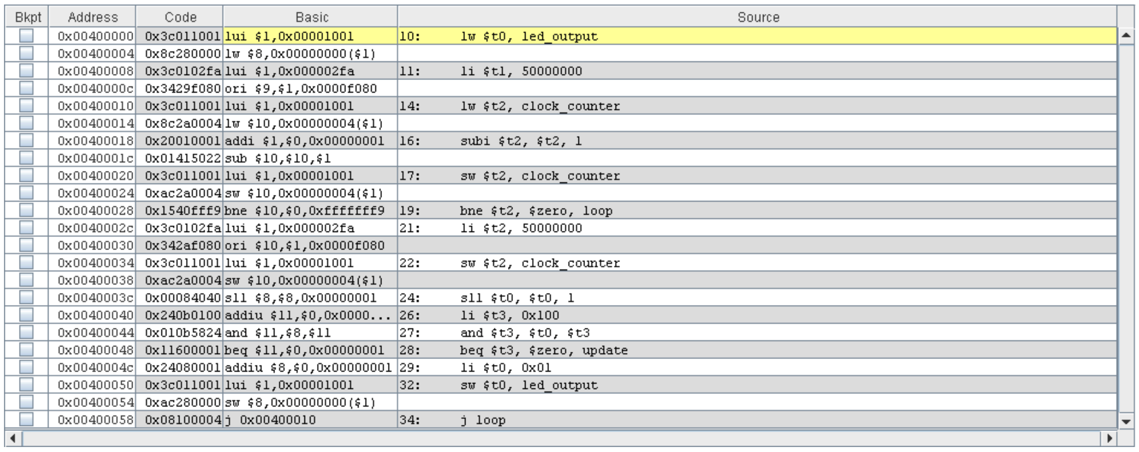
Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x3c011001	lui $1,0x00001001	10: lw $t0, a
☐	0x00400004	0x8c280000	lw $8,0x00000000($1)	
☐	0x00400008	0x3c011001	lui $1,0x00001001	11: lw $t1, b
☐	0x0040000c	0x8c290004	lw $9,0x00000004($1)	
☐	0x00400010	0x0c10000a	jal 0x00400028	13: jal add
☐	0x00400014	0x00425020	add $10,$2,$2	15: add $t2, $v0, $v0
☐	0x00400018	0x3c011001	lui $1,0x00001001	17: sw $t2, result
☐	0x0040001c	0xac2a0008	sw $10,0x00000008($1)	
☐	0x00400020	0x2402000a	addiu $2,$0,0x0000000a	19: li $v0, 10
☐	0x00400024	0x0000000c	syscall	20: syscall
☐	0x00400028	0x01091020	add $2,$8,$9	23: add $v0, $t0, $t1
☐	0x0040002c	0x03e00008	jr $31	24: jr $ra

Experiment 3 - Assembly Code & Run:



```
1   .data
2   led_output: .word 0x01
3   clock_counter: .word 0
4
5   .text
6   .globl start
7
8   start:
9
10      lw $t0, led_output
11      li $t1, 50000000
12
13  loop:
14      lw $t2, clock_counter
15
16      subi $t2, $t2, 1
17      sw $t2, clock_counter
18
19      bne $t2, $zero, loop
20
21      li $t2, 50000000
22      sw $t2, clock_counter
23
24      sll $t0, $t0, 1
25
26      li $t3, 0x100
27      and $t3, $t0, $t3
28      beq $t3, $zero, update
29      li $t0, 0x01
30
31  update:
32      sw $t0, led_output
```



Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x3c011001	lui $1,0x00001001	10: lw $t0, led_output
☐	0x00400004	0x8c280000	lw $8,0x00000000($1)	
☐	0x00400008	0x3c0102fa	lui $1,0x000002fa	11: li $t1, 50000000
☐	0x0040000c	0x3429f080	ori $9,$1,0x0000f080	
☐	0x00400010	0x3c011001	lui $1,0x00001001	14: lw $t2, clock_counter
☐	0x00400014	0x8c2a0004	lw $10,0x00000004($1)	
☐	0x00400018	0x20010001	addi $1,$0,0x00000001	16: subi $t2, $t2, 1
☐	0x0040001c	0x01415022	sub $10,$10,$1	
☐	0x00400020	0x3c011001	lui $1,0x00001001	17: sw $t2, clock_counter
☐	0x00400024	0xac2a0004	sw $10,0x00000004($1)	
☐	0x00400028	0x1540fff9	bne $10,$0,0xfffffff9	19: bne $t2, $zero, loop
☐	0x0040002c	0x3c0102fa	lui $1,0x000002fa	21: li $t2, 50000000
☐	0x00400030	0x342af080	ori $10,$1,0x0000f080	
☐	0x00400034	0x3c011001	lui $1,0x00001001	22: sw $t2, clock_counter
☐	0x00400038	0xac2a0004	sw $10,0x00000004($1)	
☐	0x0040003c	0x00084040	sll $8,$8,0x00000001	24: sll $t0, $t0, 1
☐	0x00400040	0x240b0100	addiu $11,$0,0x0000...	26: li $t3, 0x100
☐	0x00400044	0x010b5824	and $11,$8,$11	27: and $t3, $t0, $t3
☐	0x00400048	0x11600001	beq $11,$0,0x00000001	28: beq $t3, $zero, update
☐	0x0040004c	0x24080001	addiu $8,$0,0x00000001	29: li $t0, 0x01
☐	0x00400050	0x3c011001	lui $1,0x00001001	32: sw $t0, led_output
☐	0x00400054	0xac280000	sw $8,0x00000000($1)	
☐	0x00400058	0x08100004	j 0x00400010	34: j loop

3. Questions and Answers

1. Delay Implementation: A delay could be added to this experiment by creating a loop (using jump instructions) in the program using the onboard clock, per the experiment's instructions. After the rotation of each LED, the program will enter this looped part of the code.

2. Jump vs Jump and Link: The jump and link instruction in MIPS architecture is used primarily for calling subroutines (similar to function calls in high-level programming languages) whereas the jump instruction is primarily used for more permanent movements within the code. For example, jump would be used if you need to go to a different part of the program without returning to the starting location in the code. In contrast, jump and link can return to the starting location where the instruction is called after executing the subroutine.

4. Conclusions

Understanding MIPS single-cycle processor architecture as it pertains to assembly and machine code is an integral part of logic design and a fundamental part of more complex computing (along with multicycle processors). MIPS processors use registers to perform arithmetic logic operations along the datapath and driven by the processor's control unit. Understanding how all the internal components on this lower level of abstraction work together to execute operations is very important and will help with processors that have more than a single cycle.