Spring 2022

California State University, Northridge

Department of Electrical & Computer Engineering

Final Project: 16bit RISC Processor ECE 526

Written By: Ben Cooper

INTRODUCTION – 16b-RISC Circuit Design:

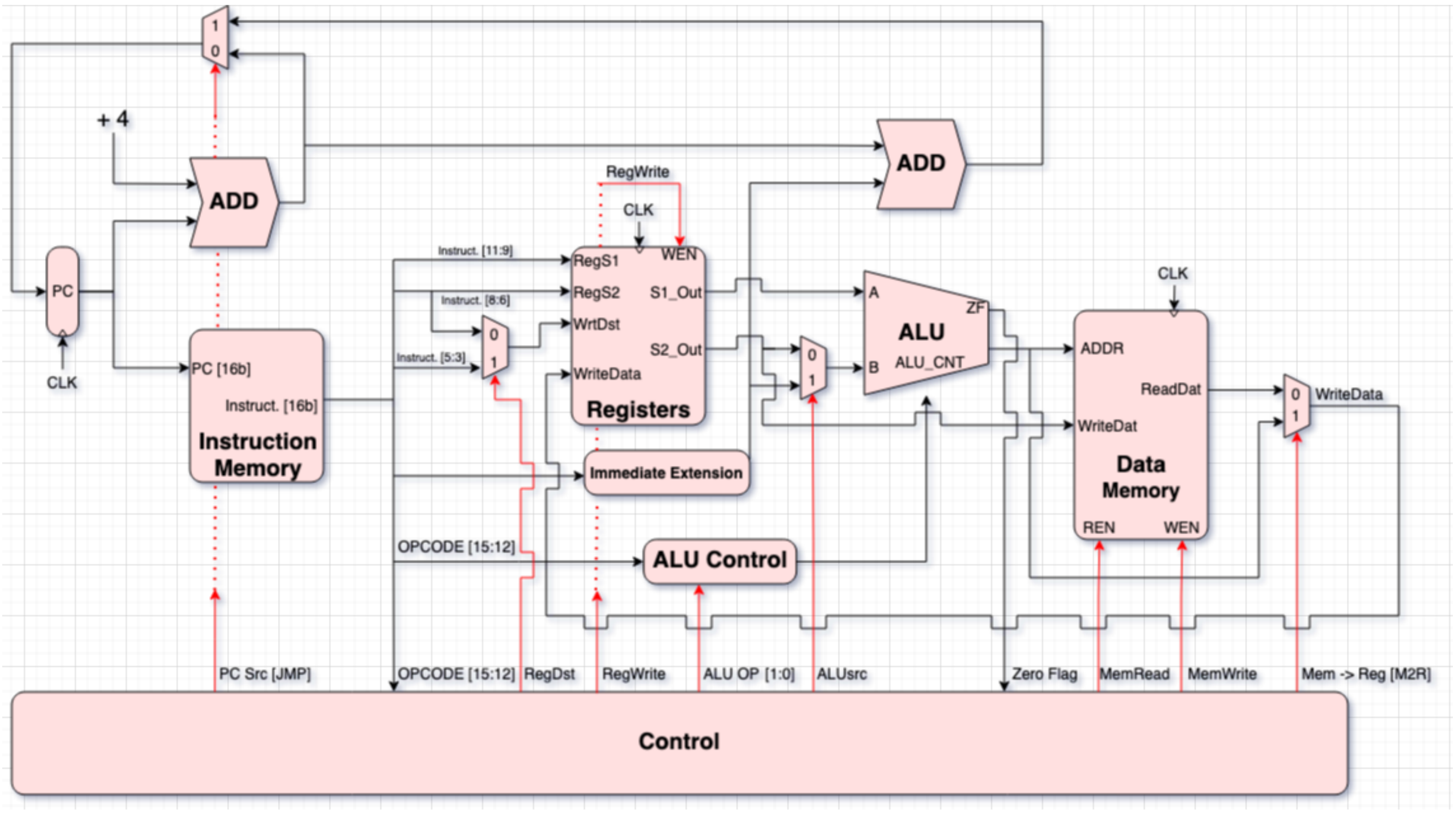
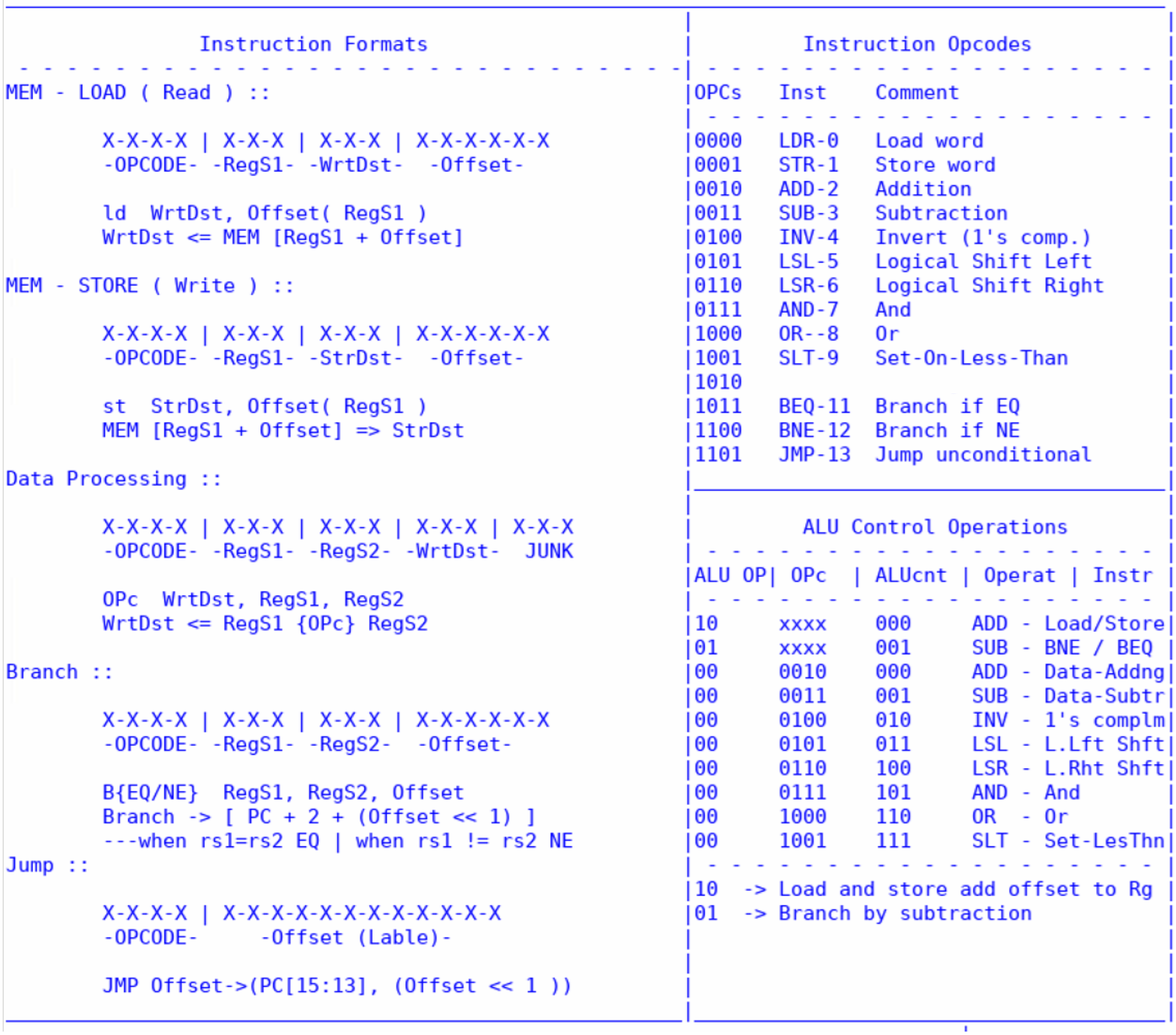


Figure 1: 16-b RISC Processor

Overview

Shown above is Figure 1, which demonstrates the 16 - bit **R**educed **I**nstruction **S**et **C**omputer [RISC] that was constructed in this project. This design uses 9 different components; Instruction Memory, Registers, ALU Control, ALU, ADD, Control, Datapath, Data Memory, and PC. Hardware is enabled or disabled using control signals based on the operation being performed from the instruction, and the wires/multiplexors connecting the units are instantiated in the Datapath module based on the control signals. There are 3 types of instructions that are stored in the Instruction Memory: Memory Access, Data Processing, and Flow Control. Figure 2 shows the instructions in RISC where they have an Opcode [4-b], at least two operand registers addresses [3b], and an offset with a constant or address [3 to 11b]. The exception is the JMP operation which has 12 bits in the offset field, and its format has the opcode of the previous instruction with the offset shifted by 1 bit to the left.

The control signal operations are performed in the control unit which operates based on the logic table in Fig 3. The PCsrc [JMP], RegDst, ALUsrc, and M2R [Writeback] control signals act as the select function for the multiplexed inputs shown in Fig 1. RegDst, when enabled, changes the destination register of the ALU operation (WrtDst) from writing to Operand 2 (RegS2) to writing to the address specified in Instruction[5:3]. RegDst is only enabled for Data Processing instructions as specified in the control signal table in Fig 3.



```
Instruction Formats

MEM - LOAD ( Read ) ::

        X-X-X-X | X-X-X | X-X-X | X-X-X-X-X-X
        -OPCODE- -RegS1- -WrtDst-  -Offset-

        ld  WrtDst, Offset( RegS1 )
        WrtDst <= MEM [RegS1 + Offset]

MEM - STORE ( Write ) ::

        X-X-X-X | X-X-X | X-X-X | X-X-X-X-X-X
        -OPCODE- -RegS1- -StrDst-  -Offset-

        st  StrDst, Offset( RegS1 )
        MEM [RegS1 + Offset] => StrDst

Data Processing ::

        X-X-X-X | X-X-X | X-X-X | X-X-X | X-X-X
        -OPCODE- -RegS1- -RegS2- -WrtDst-  JUNK

        OPc  WrtDst, RegS1, RegS2
        WrtDst <= RegS1 {OPc} RegS2

Branch ::

        X-X-X-X | X-X-X | X-X-X | X-X-X-X-X-X
        -OPCODE- -RegS1- -RegS2-  -Offset-

        B{EQ/NE}  RegS1, RegS2, Offset
        Branch -> [ PC + 2 + (Offset << 1) ]
        ---when rs1=rs2 EQ | when rs1 != rs2 NE
Jump ::

        X-X-X-X | X-X-X-X-X-X-X-X-X-X-X-X
        -OPCODE-     -Offset (Lable)-

        JMP Offset->(PC[15:13], (Offset << 1 ))
```

Instruction Opcodes

OPCs	Inst	Comment
0000	LDR-0	Load word
0001	STR-1	Store word
0010	ADD-2	Addition
0011	SUB-3	Subtraction
0100	INV-4	Invert (1's comp.)
0101	LSL-5	Logical Shift Left
0110	LSR-6	Logical Shift Right
0111	AND-7	And
1000	OR--8	Or
1001	SLT-9	Set-On-Less-Than
1010		
1011	BEQ-11	Branch if EQ
1100	BNE-12	Branch if NE
1101	JMP-13	Jump unconditional

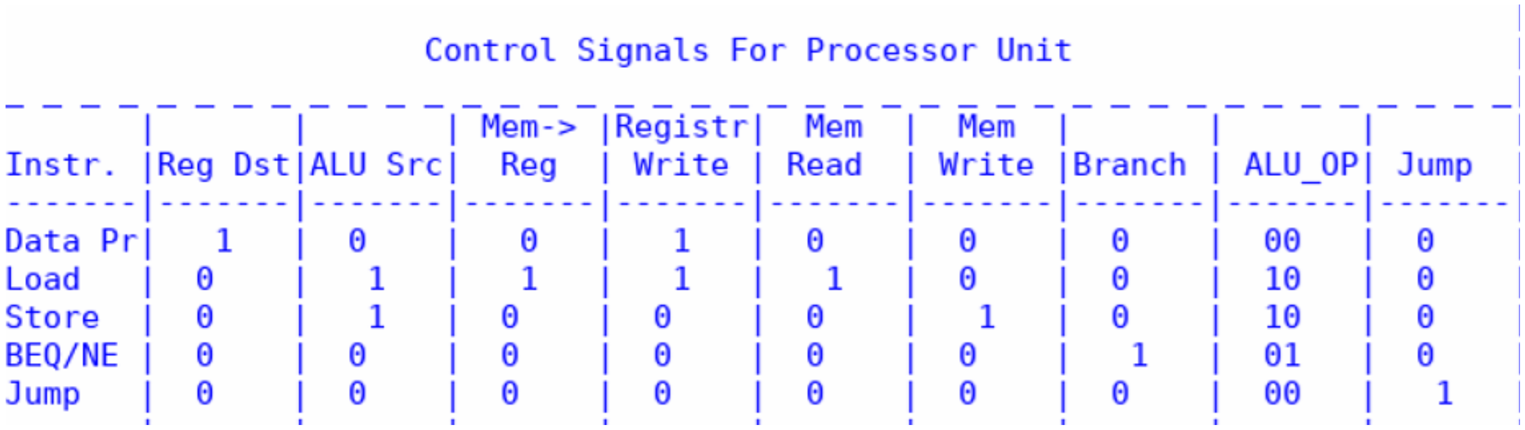
ALU Control Operations

ALU OP	OPc	ALUcnt	Operat	Instr
10	xxxx	000	ADD	- Load/Store
01	xxxx	001	SUB	- BNE / BEQ
00	0010	000	ADD	- Data-Addng
00	0011	001	SUB	- Data-Subtr
00	0100	010	INV	- 1's complm
00	0101	011	LSL	- L.Lft Shft
00	0110	100	LSR	- L.Rht Shft
00	0111	101	AND	- And
00	1000	110	OR	- Or
00	1001	111	SLT	- Set-LesThn

```
10  -> Load and store add offset to Rg
01  -> Branch by subtraction
```

Figure 2: Instruction Formats

Figure 3: Control Signals



Control Signals For Processor Unit

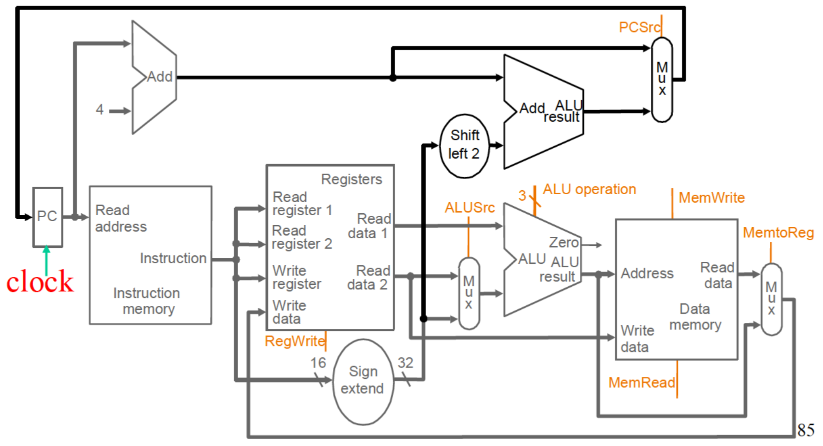
Instr.	Reg Dst	ALU Src	Mem-> Reg	Registr Write	Mem Read	Mem Write	Branch	ALU_OP	Jump
Data Pr	1	0	0	1	0	0	0	00	0
Load	0	1	1	1	1	0	0	10	0
Store	0	1	0	0	0	1	0	10	0
BEQ/NE	0	0	0	0	0	0	1	01	0
Jump	0	0	0	0	0	0	0	00	1

Control Signals for Instruction Operations:

The ALUsrc control signal will determine if operand "B" in the ALU operation is driven by an Immediate w/ Sign Extension or driven by the register output (S2_Out) from the General-Purpose Registers (GPRs). M2R or Memory to Register is the control signal that is activated during a load operation to change the write back input from ALU_Out to the contents of Data Memory at the instruction address. Furthermore, the RegisterWrite signal is used at writeback to store the contents of the instruction (WriteData) at the write destination (WrtDst),. Memory (Read or Write) are two control signals that determines if the Data Memory component is going to read from the passed address or write to the passed address, and the WriteData is an input from S2_out while the address comes from ALU_Out. In relation to RISC instructions, the previous control signals are used to perform load and store operations on the circuit.

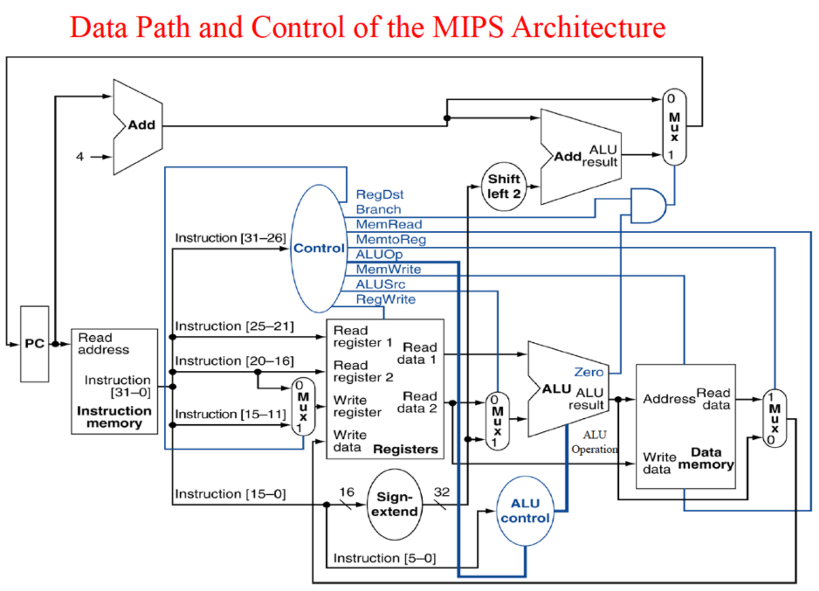
This processor has 2 different branching options, branch on equal / not equal, and they both activate the control signal "Branch" which determines if the ALU operation uses the signed immediate extension (extending bit 6 to the MSB and retaining first 6 bits as normal) or if the ALU operation uses S2_Out instead. The ALU operation is 01 when branching, but when performing a JMP the ALU operation is the same as that for Data processing and its control signal is "JUMP".

Instruction Format / Processor Design References:



[4] Figure 4: 32-bit MIPS Basic Layout

- Figure 4 demonstrates how components are connected in a typical RISC processor are connected.

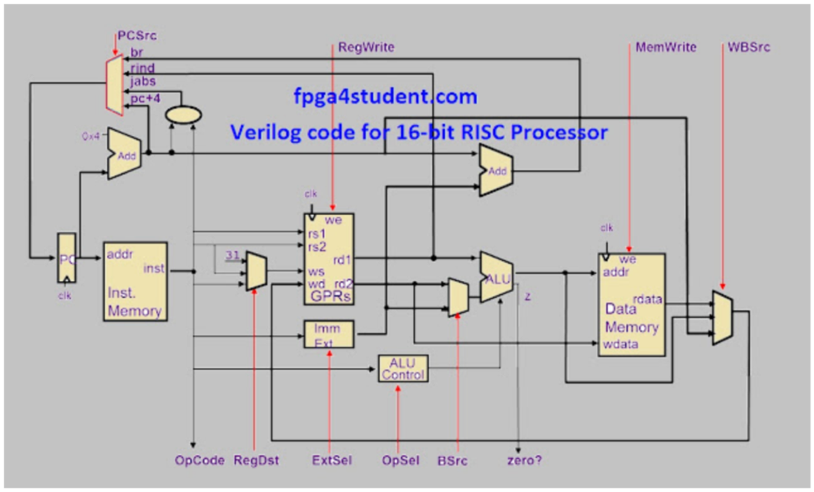


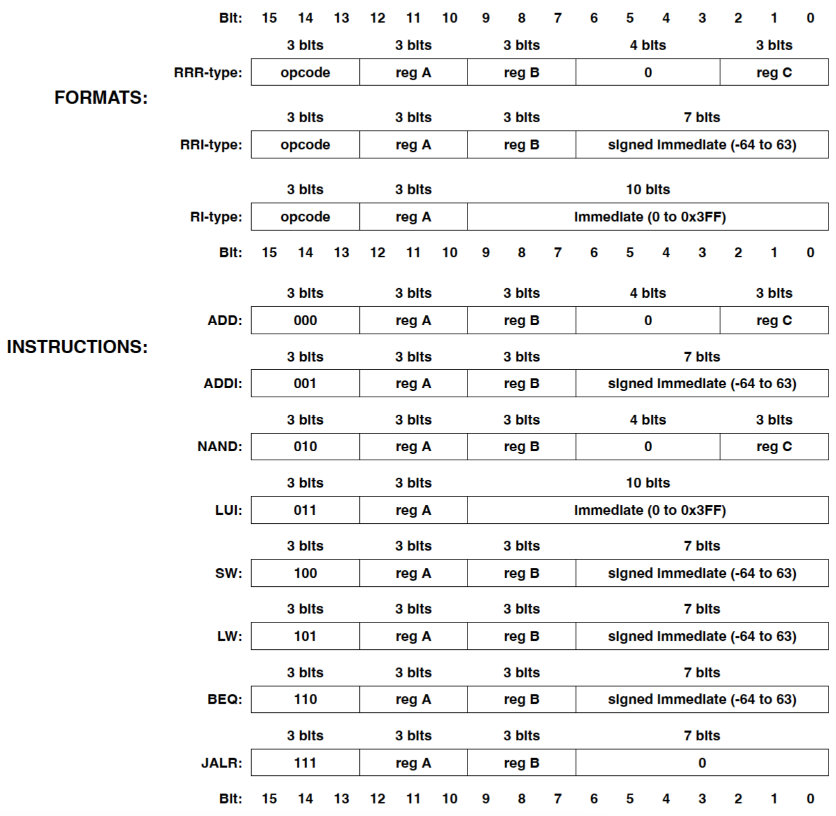
[4] Figure 5: 32-b MIPS Datapath / Control Signal

- This design shows the how the control signals address components of the MIPS processor. It also demonstrates a more complete Datapath that than Fig 5.

[1] Figure 6: 16-bit RISC Processor

- This design has a better relation to this project because the schematic is 16-bit based as well as being pure RISC instead of MIPS.

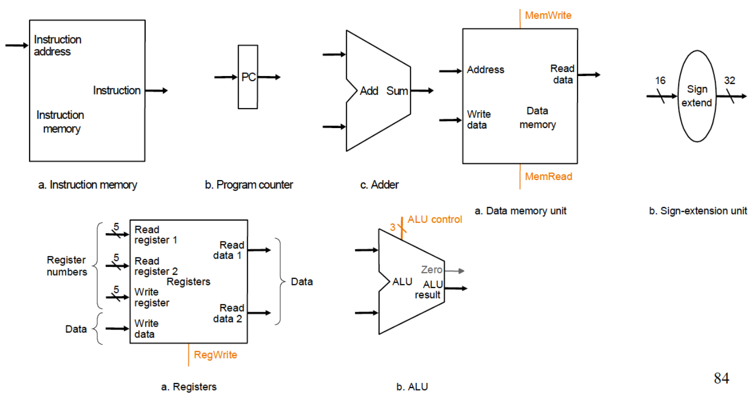




	Bit: 15 14 13	12 11 10	9 8 7	6 5 4 3	2 1 0
FORMATS:					
	3 bits	3 bits	3 bits	4 bits	3 bits
RRR-type:	opcode	reg A	reg B	0	reg C
	3 bits	3 bits	3 bits	7 bits	
RRI-type:	opcode	reg A	reg B	signed immediate (-64 to 63)	
	3 bits	3 bits	10 bits		
RI-type:	opcode	reg A	immediate (0 to 0x3FF)		
Bit:	15 14 13	12 11 10	9 8 7	6 5 4 3	2 1 0
INSTRUCTIONS:					
	3 bits	3 bits	3 bits	4 bits	3 bits
ADD:	000	reg A	reg B	0	reg C
	3 bits	3 bits	3 bits	7 bits	
ADDI:	001	reg A	reg B	signed immediate (-64 to 63)	
	3 bits	3 bits	3 bits	4 bits	3 bits
NAND:	010	reg A	reg B	0	reg C
	3 bits	3 bits	10 bits		
LUI:	011	reg A	immediate (0 to 0x3FF)		
	3 bits	3 bits	3 bits	7 bits	
SW:	100	reg A	reg B	signed immediate (-64 to 63)	
	3 bits	3 bits	3 bits	7 bits	
LW:	101	reg A	reg B	signed immediate (-64 to 63)	
	3 bits	3 bits	3 bits	7 bits	
BEQ:	110	reg A	reg B	signed immediate (-64 to 63)	
	3 bits	3 bits	3 bits	7 bits	
JALR:	111	reg A	reg B	0	
Bit:	15 14 13	12 11 10	9 8 7	6 5 4 3	2 1 0

[2] Figure 7: RISC Instructions Format

- The formats on the left are an example of how the RISC instructions can be built and used for the design.



[4] Figure 8: Basic Parts of RISC Processer ^

Verilog Module Header Format:

```
/*********************************************************************\
***                                                                 ***
*** ECE 526 | Final Proj - 16-b RISC Processor       Ben Cooper | Fall, 2022 ***
***                                                                 ***
*** Final Project - Design of 16-b RISC Processor                   ***
***                                                                 ***
***********************************************************************
*** Filename: 16b_RISC.v Created by Ben_Cooper, 5/31/22             ***
***                --- revision history, if any, goes here ---      ***
**********************************************************************/
```

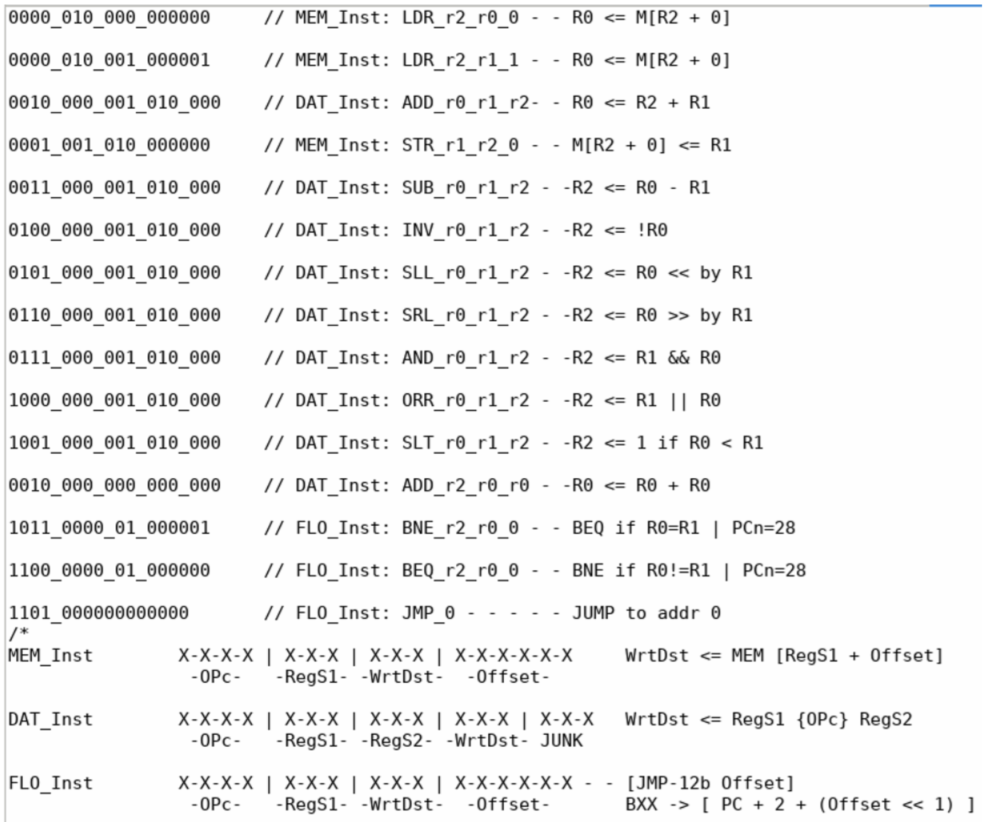
METHODOLOGY – Code Operation / Test Plan:

While designing a processor, the easiest way to verify that the components are functioning correctly is to test them before the entire processor is tested. This method of testing limits any confusion on compiler errors since the individual parts already function correctly on their own. The code that I am using for the Instruction Memory is similar to that of a ROM module, and therefore I only needed to modify an existing model that I had created in the 526 lab. Reusing my code from labs was what made the testing of individual components unnecessary for this design as I had already verified that the modules work. In this line of thinking, the Data Memory unit was reused from a RAM module, and the ALU was reused from a lab on designing an ALU. The modifications to these units were mostly to allow the transport of 16bit input/outputs, and for the ALU to receive the operation code from a secondary unit (ALU Control Unit).

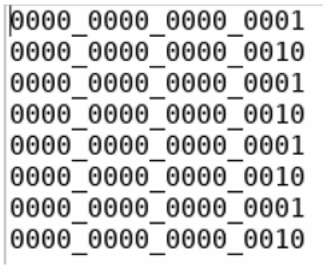
My methods for assembling the processor came from my experience in ECE 422 (Digital Design of Computers), I used a Datapath module to assign values to components based on Control Signals. By assigning values based on Control Signals, the behavior of the inputs/outputs can be modeled as a 2x1 multiplexor that is switched based on the control signal value. These logical multiplexors are shown in the reference designs as well as Figure 1 (self-made diagram), and they were instantiated as tristate assign operators in Datapath module. The control signals were based on the table shown in Figure 3, and they were assigned values based on a case statement using the OPCODE as input.

Testing methodology in a processor is done differently than single component tests, the only value that was created at the top-level hierarchical model was the clock. Since instructions are stored in the Instruction Memory module and data is stored in the Data Memory module, testing the processor involves writing machine code to signal what operations and instructions are run given the data in memory. The Instruction Memory I wrote for the processor is loaded into the IM module at initialization, it is displayed in Figure 9 and test every opcode operation at least once. The data memory initial contents are shown in Figure 10, their values after the instructions are run is shown in Figure 11. The 16b_RISC module instantiates the Datapath and Control modules and passes the clock, the PC starts at 0 and increments by 2 unless Branching or Jumping.

Figure 9: Instruction Memory Initial File (testIM.prog)

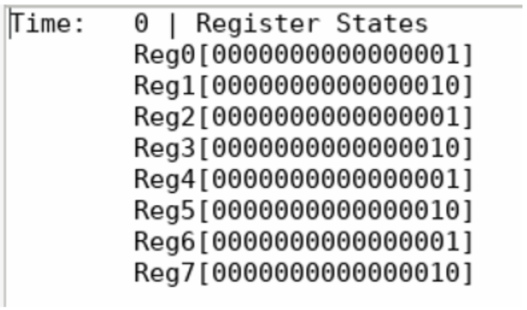


```
0000_010_000_000000     // MEM_Inst: LDR_r2_r0_0 - - R0 <= M[R2 + 0]

0000_010_001_000001     // MEM_Inst: LDR_r2_r1_1 - - R0 <= M[R2 + 0]

0010_000_001_010_000    // DAT_Inst: ADD_r0_r1_r2- - R0 <= R2 + R1

0001_001_010_000000     // MEM_Inst: STR_r1_r2_0 - - M[R2 + 0] <= R1

0011_000_001_010_000    // DAT_Inst: SUB_r0_r1_r2 - -R2 <= R0 - R1

0100_000_001_010_000    // DAT_Inst: INV_r0_r1_r2 - -R2 <= !R0

0101_000_001_010_000    // DAT_Inst: SLL_r0_r1_r2 - -R2 <= R0 << by R1

0110_000_001_010_000    // DAT_Inst: SRL_r0_r1_r2 - -R2 <= R0 >> by R1

0111_000_001_010_000    // DAT_Inst: AND_r0_r1_r2 - -R2 <= R1 && R0

1000_000_001_010_000    // DAT_Inst: ORR_r0_r1_r2 - -R2 <= R1 || R0

1001_000_001_010_000    // DAT_Inst: SLT_r0_r1_r2 - -R2 <= 1 if R0 < R1

0010_000_000_000_000    // DAT_Inst: ADD_r2_r0_r0 - -R0 <= R0 + R0

1011_0000_01_000001     // FLO_Inst: BNE_r2_r0_0 - - BEQ if R0=R1 | PCn=28

1100_0000_01_000000     // FLO_Inst: BEQ_r2_r0_0 - - BNE if R0!=R1 | PCn=28

1101_000000000000       // FLO_Inst: JMP_0 - - - - - JUMP to addr 0
/*
MEM_Inst        X-X-X-X | X-X-X | X-X-X | X-X-X-X-X-X     WrtDst <= MEM [RegS1 + Offset]
                 -OPc-   -RegS1- -WrtDst-  -Offset-

DAT_Inst        X-X-X-X | X-X-X | X-X-X | X-X-X | X-X-X   WrtDst <= RegS1 {OPc} RegS2
                 -OPc-   -RegS1- -RegS2- -WrtDst- JUNK

FLO_Inst        X-X-X-X | X-X-X | X-X-X | X-X-X-X-X-X - - [JMP-12b Offset]
                 -OPc-   -RegS1- -WrtDst-  -Offset-       BXX -> [ PC + 2 + (Offset << 1) ]
```



```
0000_0000_0000_0001
0000_0000_0000_0010
0000_0000_0000_0001
0000_0000_0000_0010
0000_0000_0000_0001
0000_0000_0000_0010
0000_0000_0000_0001
0000_0000_0000_0010
```

Figure 10: Data Memory File (test.data)

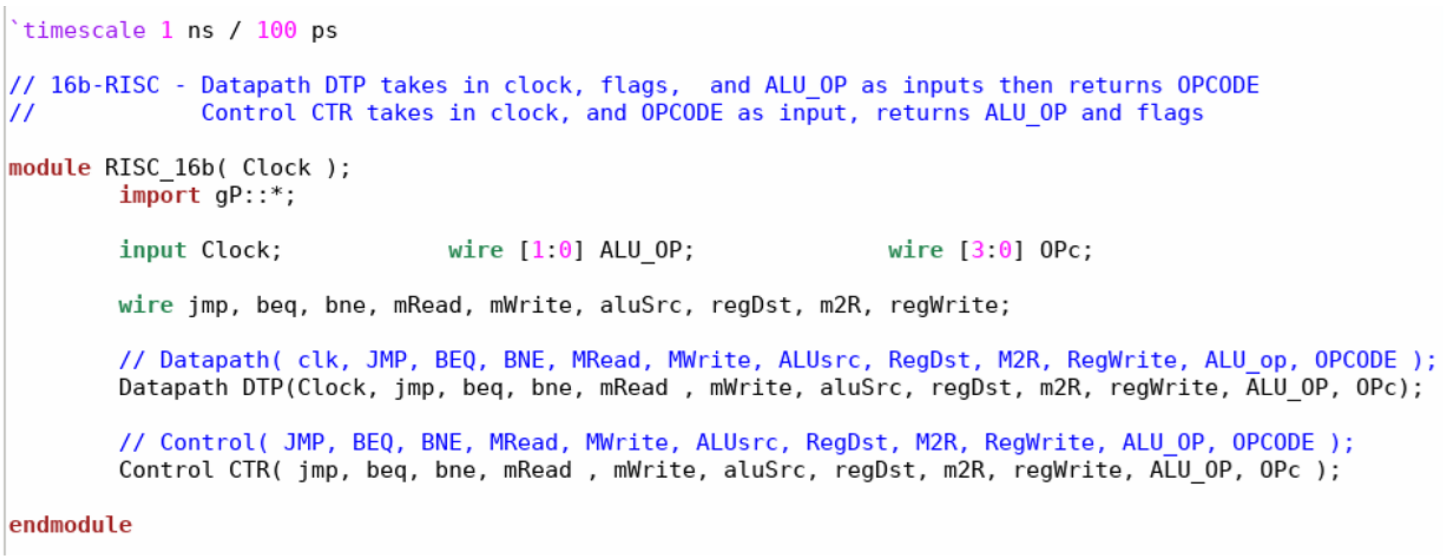


```
Time:   0 | Register States
        Reg0[0000000000000001]
        Reg1[0000000000000010]
        Reg2[0000000000000001]
        Reg3[0000000000000010]
        Reg4[0000000000000001]
        Reg5[0000000000000010]
        Reg6[0000000000000001]
        Reg7[0000000000000010]
```

Figure 11: Read Data Memory contents

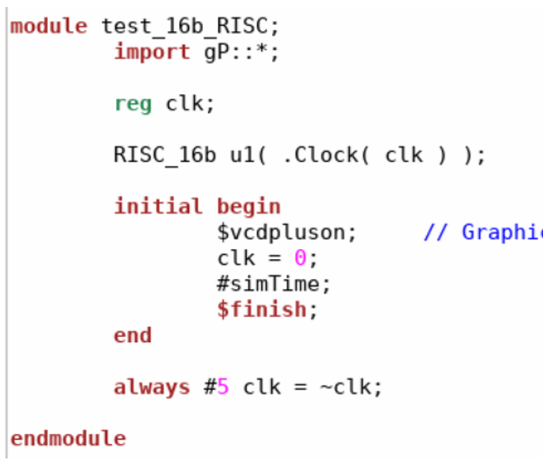
Upper-Level Modules & Analysis:

Lower-level modules in this circuit design include the ALU, Data Memory, Instruction Memory, Registers, and ALU control unit. The higher-level modules are where connections, signaling, and testbenches are performed and they are the following: test_16b_RISC, Datapath, Control, and RISC_16b. RISC_16b (*Fig.12*) is the top level module for the circuit, and it contains an instance of Datapath and another of Control which are wired together with the only necessary input being the CLK. test_16b_RISC (*Fig.13*) is a file that initiates RISC_16b using a clock with a 100 MHz frequency (10ns period) and initiates the graphical waveform.



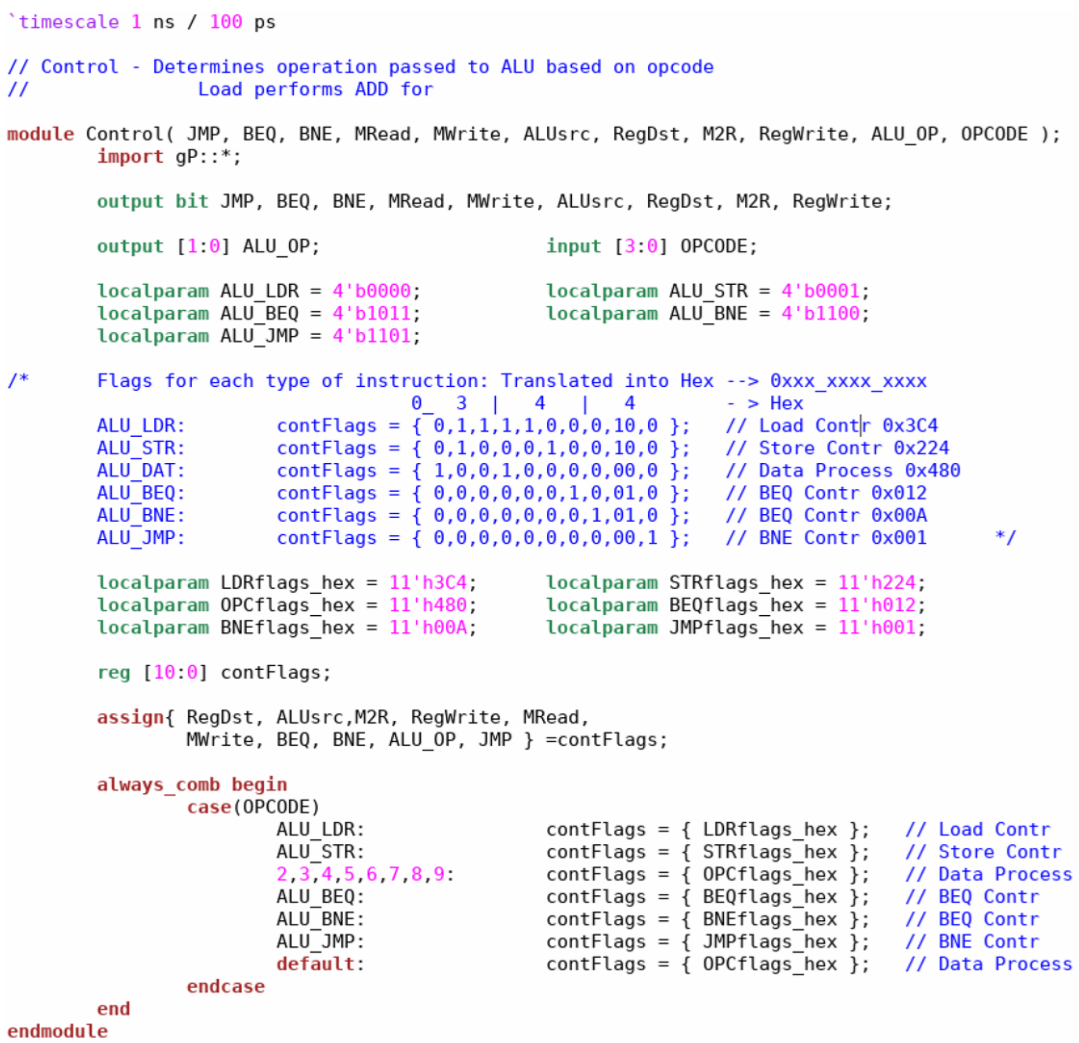
```
`timescale 1 ns / 100 ps

// 16b-RISC - Datapath DTP takes in clock, flags,  and ALU_OP as inputs then returns OPCODE
//            Control CTR takes in clock, and OPCODE as input, returns ALU_OP and flags

module RISC_16b( Clock );
        import gP::*;

        input Clock;            wire [1:0] ALU_OP;              wire [3:0] OPc;

        wire jmp, beq, bne, mRead, mWrite, aluSrc, regDst, m2R, regWrite;

        // Datapath( clk, JMP, BEQ, BNE, MRead, MWrite, ALUsrc, RegDst, M2R, RegWrite, ALU_op, OPCODE );
        Datapath DTP(Clock, jmp, beq, bne, mRead , mWrite, aluSrc, regDst, m2R, regWrite, ALU_OP, OPc);

        // Control( JMP, BEQ, BNE, MRead, MWrite, ALUsrc, RegDst, M2R, RegWrite, ALU_OP, OPCODE );
        Control CTR( jmp, beq, bne, mRead , mWrite, aluSrc, regDst, m2R, regWrite, ALU_OP, OPc );

endmodule
```

Figure 12: RISC_16b – Top Level Circuit Instance ^



```
module test_16b_RISC;
        import gP::*;

        reg clk;

        RISC_16b u1( .Clock( clk ) );

        initial begin
                $vcdpluson;     // Graphi
                clk = 0;
                #simTime;
                $finish;
        end

        always #5 clk = ~clk;

endmodule
```

Fig 13: Testbench & $Waveform ^



```
`timescale 1 ns / 100 ps

// Control - Determines operation passed to ALU based on opcode
//              Load performs ADD for

module Control( JMP, BEQ, BNE, MRead, MWrite, ALUsrc, RegDst, M2R, RegWrite, ALU_OP, OPCODE );
        import gP::*;

        output bit JMP, BEQ, BNE, MRead, MWrite, ALUsrc, RegDst, M2R, RegWrite;

        output [1:0] ALU_OP;                    input [3:0] OPCODE;

        localparam ALU_LDR = 4'b0000;           localparam ALU_STR = 4'b0001;
        localparam ALU_BEQ = 4'b1011;           localparam ALU_BNE = 4'b1100;
        localparam ALU_JMP = 4'b1101;

/*      Flags for each type of instruction: Translated into Hex --> 0xxx_xxxx_xxxx
                                  0_  3  |   4   |   4        - > Hex
        ALU_LDR:        contFlags = { 0,1,1,1,1,0,0,0,10,0 };   // Load Contr 0x3C4
        ALU_STR:        contFlags = { 0,1,0,0,0,1,0,0,10,0 };   // Store Contr 0x224
        ALU_DAT:        contFlags = { 1,0,0,1,0,0,0,0,00,0 };   // Data Process 0x480
        ALU_BEQ:        contFlags = { 0,0,0,0,0,0,1,0,01,0 };   // BEQ Contr 0x012
        ALU_BNE:        contFlags = { 0,0,0,0,0,0,0,1,01,0 };   // BEQ Contr 0x00A
        ALU_JMP:        contFlags = { 0,0,0,0,0,0,0,0,00,1 };   // BNE Contr 0x001      */

        localparam LDRflags_hex = 11'h3C4;      localparam STRflags_hex = 11'h224;
        localparam OPCflags_hex = 11'h480;      localparam BEQflags_hex = 11'h012;
        localparam BNEflags_hex = 11'h00A;      localparam JMPflags_hex = 11'h001;

        reg [10:0] contFlags;

        assign{ RegDst, ALUsrc,M2R, RegWrite, MRead,
                MWrite, BEQ, BNE, ALU_OP, JMP } =contFlags;

        always_comb begin
                case(OPCODE)
                        ALU_LDR:                contFlags = { LDRflags_hex };   // Load Contr
                        ALU_STR:                contFlags = { STRflags_hex };   // Store Contr
                        2,3,4,5,6,7,8,9:        contFlags = { OPCflags_hex };   // Data Process
                        ALU_BEQ:                contFlags = { BEQflags_hex };   // BEQ Contr
                        ALU_BNE:                contFlags = { BNEflags_hex };   // BEQ Contr
                        ALU_JMP:                contFlags = { JMPflags_hex };   // BNE Contr
                        default:                contFlags = { OPCflags_hex };   // Data Process
                endcase
        end
endmodule
```

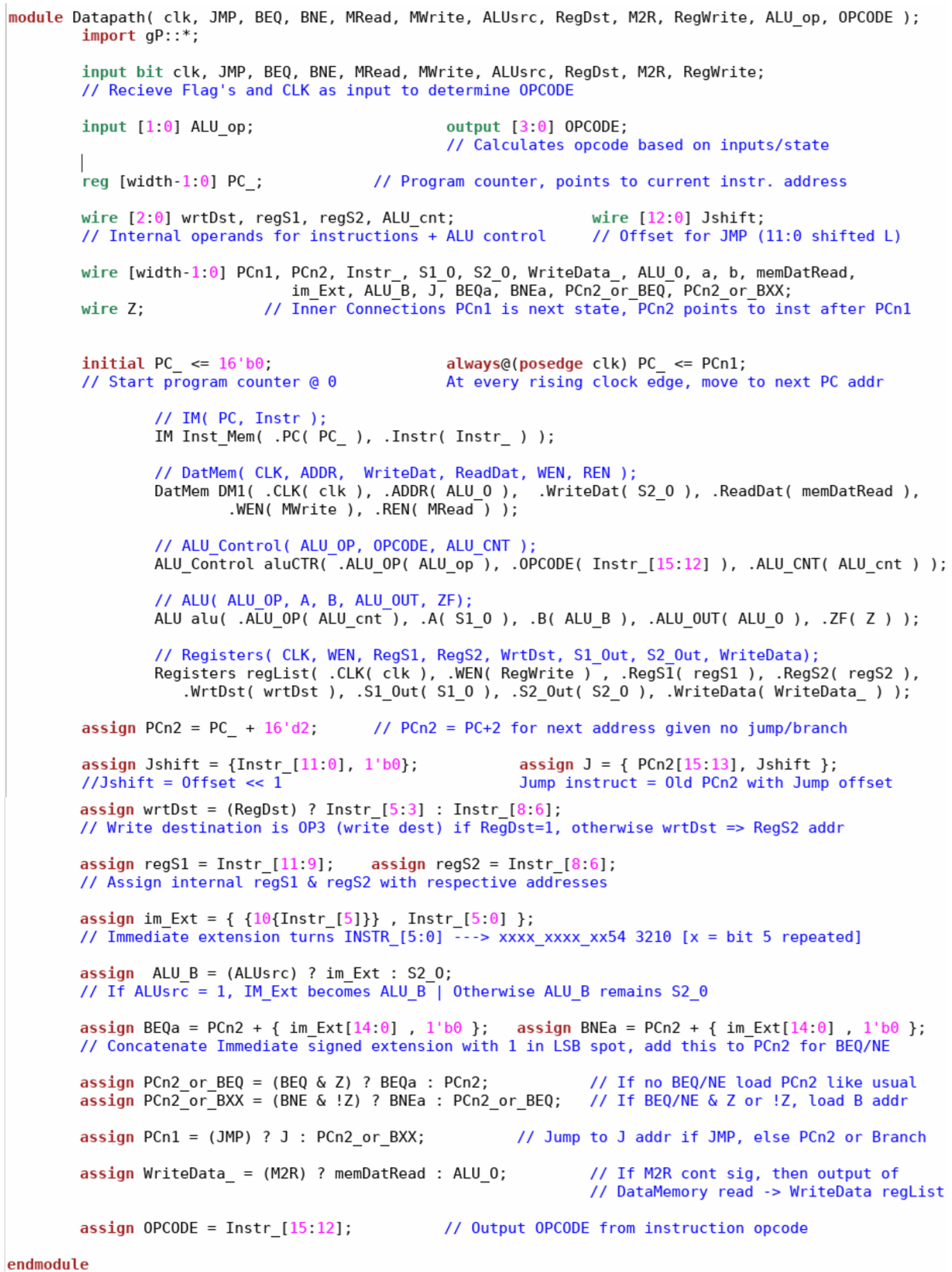
Fig 14: Control Module

- Control signals are addressed in hex format based on opcode values (localparam addressing)
- Each case specified by opcode matches the instructions that were used in Figure 3.

Datapath Module Analysis:

Datapaths represent the wiring of the circuit and it models the activation of components based on the control signals used as input. Some control signals work as a multiplexor 'select' input, and they switch the input/output of a component to different sources/destinations dependent on the Control modules logical output. All components are instantiated and passed internal wiring (reg or wire), and these wires are driven values by the assign operators in the second half of Figure 15.

Figure 15: Datapath Module



```
module Datapath( clk, JMP, BEQ, BNE, MRead, MWrite, ALUsrc, RegDst, M2R, RegWrite, ALU_op, OPCODE );
        import gP::*;

        input bit clk, JMP, BEQ, BNE, MRead, MWrite, ALUsrc, RegDst, M2R, RegWrite;
        // Recieve Flag's and CLK as input to determine OPCODE

        input [1:0] ALU_op;                         output [3:0] OPCODE;
                                                    // Calculates opcode based on inputs/state
        |
        reg [width-1:0] PC_;              // Program counter, points to current instr. address

        wire [2:0] wrtDst, regS1, regS2, ALU_cnt;                 wire [12:0] Jshift;
        // Internal operands for instructions + ALU control       // Offset for JMP (11:0 shifted L)

        wire [width-1:0] PCn1, PCn2, Instr_, S1_O, S2_O, WriteData_, ALU_O, a, b, memDatRead,
                                im_Ext, ALU_B, J, BEQa, BNEa, PCn2_or_BEQ, PCn2_or_BXX;
        wire Z;             // Inner Connections PCn1 is next state, PCn2 points to inst after PCn1


        initial PC_ <= 16'b0;                       always@(posedge clk) PC_ <= PCn1;
        // Start program counter @ 0                At every rising clock edge, move to next PC addr

                // IM( PC, Instr );
                IM Inst_Mem( .PC( PC_ ), .Instr( Instr_ ) );

                // DatMem( CLK, ADDR,  WriteDat, ReadDat, WEN, REN );
                DatMem DM1( .CLK( clk ), .ADDR( ALU_O ),  .WriteDat( S2_O ), .ReadDat( memDatRead ),
                        .WEN( MWrite ), .REN( MRead ) );

                // ALU_Control( ALU_OP, OPCODE, ALU_CNT );
                ALU_Control aluCTR( .ALU_OP( ALU_op ), .OPCODE( Instr_[15:12] ), .ALU_CNT( ALU_cnt ) );

                // ALU( ALU_OP, A, B, ALU_OUT, ZF);
                ALU alu( .ALU_OP( ALU_cnt ), .A( S1_O ), .B( ALU_B ), .ALU_OUT( ALU_O ), .ZF( Z ) );

                // Registers( CLK, WEN, RegS1, RegS2, WrtDst, S1_Out, S2_Out, WriteData);
                Registers regList( .CLK( clk ), .WEN( RegWrite ) , .RegS1( regS1 ), .RegS2( regS2 ),
                   .WrtDst( wrtDst ), .S1_Out( S1_O ), .S2_Out( S2_O ), .WriteData( WriteData_ ) );

        assign PCn2 = PC_ + 16'd2;      // PCn2 = PC+2 for next address given no jump/branch

        assign Jshift = {Instr_[11:0], 1'b0};           assign J = { PCn2[15:13], Jshift };
        //Jshift = Offset << 1                          Jump instruct = Old PCn2 with Jump offset

        assign wrtDst = (RegDst) ? Instr_[5:3] : Instr_[8:6];
        // Write destination is OP3 (write dest) if RegDst=1, otherwise wrtDst => RegS2 addr

        assign regS1 = Instr_[11:9];    assign regS2 = Instr_[8:6];
        // Assign internal regS1 & regS2 with respective addresses

        assign im_Ext = { {10{Instr_[5]}} , Instr_[5:0] };
        // Immediate extension turns INSTR_[5:0] ---> xxxx_xxxx_xx54 3210 [x = bit 5 repeated]

        assign  ALU_B = (ALUsrc) ? im_Ext : S2_O;
        // If ALUsrc = 1, IM_Ext becomes ALU_B | Otherwise ALU_B remains S2_0

        assign BEQa = PCn2 + { im_Ext[14:0] , 1'b0 };   assign BNEa = PCn2 + { im_Ext[14:0] , 1'b0 };
        // Concatenate Immediate signed extension with 1 in LSB spot, add this to PCn2 for BEQ/NE

        assign PCn2_or_BEQ = (BEQ & Z) ? BEQa : PCn2;             // If no BEQ/NE load PCn2 like usual
        assign PCn2_or_BXX = (BNE & !Z) ? BNEa : PCn2_or_BEQ;     // If BEQ/NE & Z or !Z, load B addr

        assign PCn1 = (JMP) ? J : PCn2_or_BXX;          // Jump to J addr if JMP, else PCn2 or Branch

        assign WriteData_ = (M2R) ? memDatRead : ALU_O;           // If M2R cont sig, then output of
                                                                  // DataMemory read -> WriteData regList

        assign OPCODE = Instr_[15:12];          // Output OPCODE from instruction opcode

endmodule
```

RESULTS – Signal Waveform:

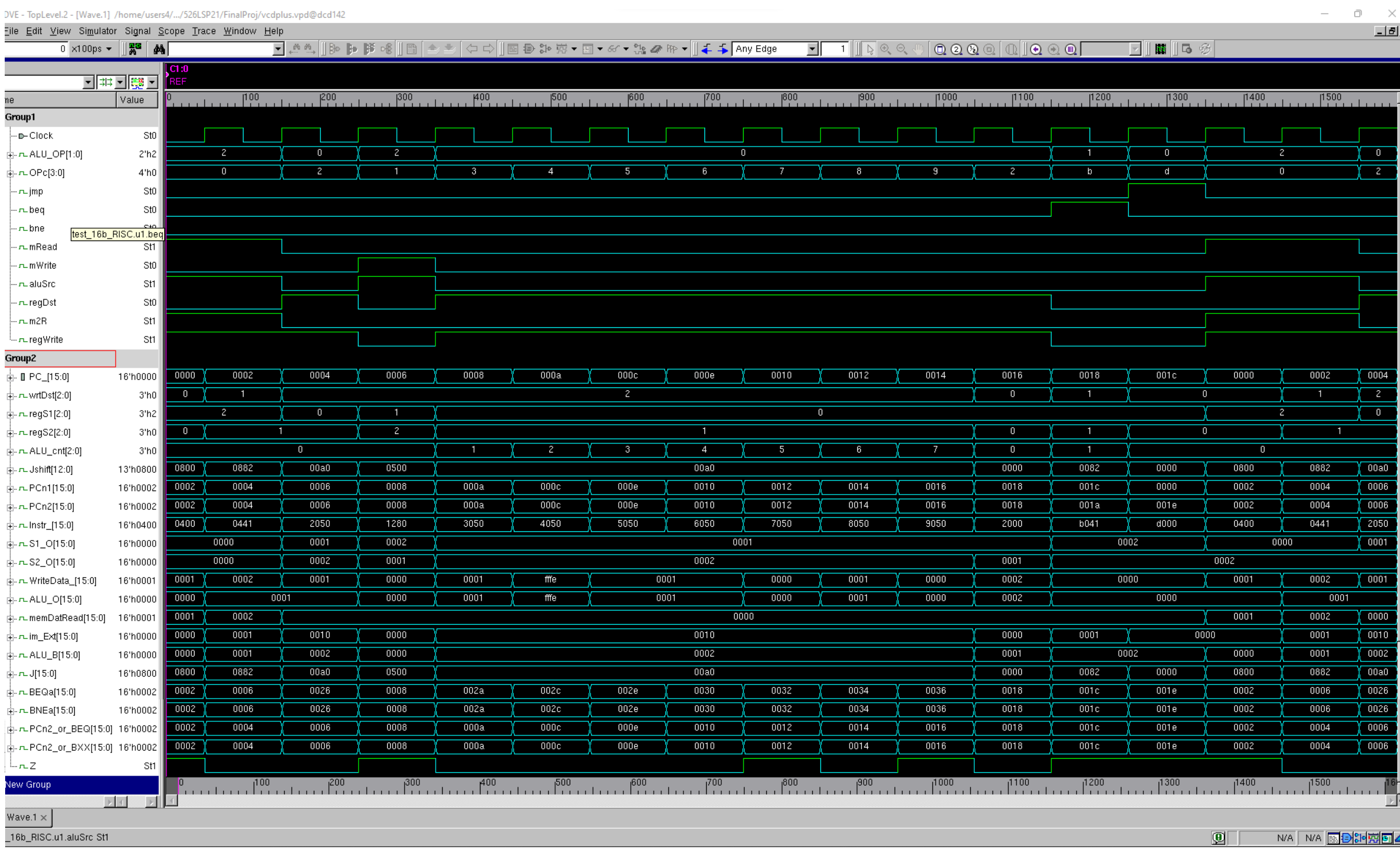
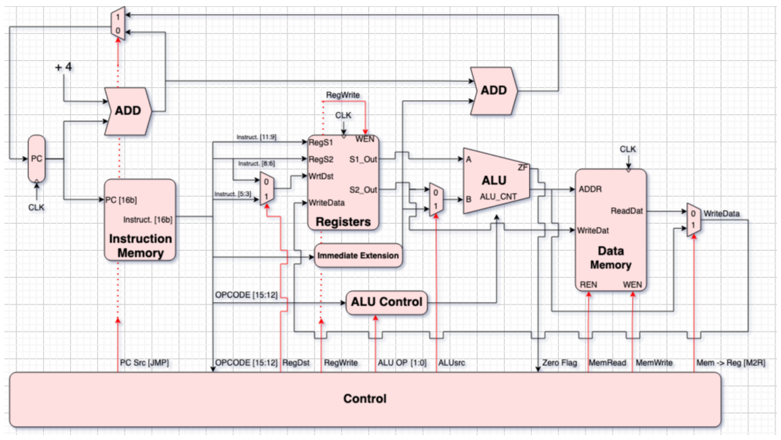


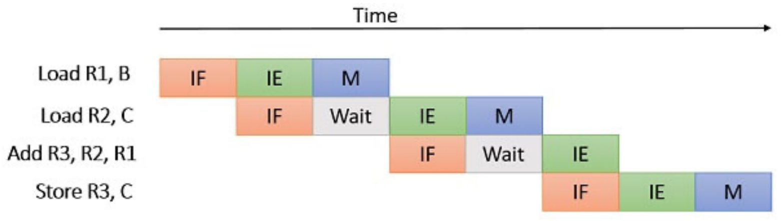
Figure 16: [**Results**] Group 1 := internal wirings for 'RISC_16b' | Group 2 := internal wirings for 'Datapath'

Results Analysis:

The circuit is positive edge triggered by the clock signal in Group 1. Initially the instructions have PC set to 0, and the first line of assembly codes of instructions are to load R0 <= MEM[R2 + 0]. Since all registers are initialized to 0 the load instruction moves the contents of Data Memory[R2 (0) + 0] to R0 (0->1), and this is shown in the WriteData wire before the first rising edge. The test bench is non exhaustive, but it is heuristic (written to demonstrate sufficient capabilities by hand) such that the machine code written in Fig. 9 controls the operations performed during that clock cycle. The results from this test show how the processor reacts for every opcode in the system, including BEQ, BNE, and JUMP (shown in the column of values where respective control signal is raised Fig. 3). The arithmetic operations are performed for the 8 data processing instructions, the result is displayed in the wire ALU_O. The wires wrtDst, regS1, regS2, contain the addresses that hold the values for the respective operands; their assignment is dependent on control signals / instruction being performed. Immediate Extension (im_Ext) functions as a signed extension tool for the circuit, where bit 6 is extended through to the MSB of the instruction and fed into the ALU. PCn1 and PCn2 are the respective registers to hold the value of PC's next and 2nd to follow instruction addresses. BEQa and BNEa hold the branch address that will be assigned to PCn2 if the control signal conditions are met. Jshift is a register that holds offset for the wire J which is the value of the jump location (only assigned with respective control signal JUMP).

CONCLUSION – Summary & References:





RISC Pipelining

Summary:

This processor serves as a proof of concept that reflects the stages of designing and implementing a basic RISC processor. In the process of building the circuit, it became evident on where future processors could improve to take advantage of hardware and clock speeds. Future designs of a RISC processor should implement some of the following changes which can be a great improvement to the current operation:

1. Pipelining : Multistage processing (stages build over cycles like a staircase).
2. Multicycle : Format with additional cycles.
3. Larger data pathways to provide a further reach in memory, and longer instructions.

References :

[1] "16-bit RISC processor," *FPGA Projects, Verilog Projects, VHDL Projects - FPGA4student.com*. [Online]. Available: https://www.fpga4student.com/2017/04/verilog-code-for-16-bit-risc-processor.html. [Accessed: 10-May-2022].

[2] B. Jacob, "The RiSC-16 Instruction-Set Architecture," *ENEE 446: Digital Computer Design — The RiSC-16 Instruction-Set Architecture*, 2000. .

[3] Divya and Deepty, *A Verilog-Based Design of 16–Bit RISC Processor*, Jun-2015. [Online]. Available: www.erpublication.org. [Accessed: 10-May-2022].

[4] N. E. Naga, "RISC-Architecture." Dr. N. El Naga, Simi Valley, Jan-2022.

[5] S. Gaonkar and A. M., "Design of 16-bit RISC Processor," *International Journal of Scientific Research in Physics and Applied Sciences*, 2014. [Online]. Available: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwiWo7C42t73AhUFK0QIHa1fDt0QFnoECCcQAQ&url=https%3A%2F%2Fwww.isroset.org%2Fpub_paper%2FIJSRPAS%2FISROSET-IJSRPAP-00068.pdf&usg=AOvVaw3DuvepdfrvIXE4TdG4C2dq. [Accessed: 10-May-2022].

[6] V. Rao, A. Anita, and K. Bhaaskaran, *Design of a 16 Bit RISC Processor*, Aug-2015.