Hardware Realization of a Computer System

3002

MD SAKIBUL ALAM 7650714

Module teacher
Sam Amiri

Table of Contents

Introduction

MIPS architecture is one of the most common basic level processor designs. The architecture was designed by Standford University in 1984. This processor is RISC (Reduced Instruction Set Computer) based processor meaning it has fewer simpler instructions than processors like the Intel Haswell, Skylake or the KabyLake architecture. This report will discuss a MIPS processor design in depth and build upon a previous design provided by the module leader, Sam Amiri.

Design

Name	Type
ADD	R-type
SUB	R-type
AND	R-type
OR	R-type
SLT	R-type
SLL	R-type
LW	
SW	
BEQ	
ADDI	
J	
SLTI	
LUI	
BLEZ	

Table 1

Figure 1 displays the type and names of the instructions supported by this processor. This design will demonstrate the implementation of SLL, SLTI, LUI, and BELZ. This processor is a single cycle processor that supports 32 bits instruction. Single cycle meaning in each processor clock cycle, this CPU will only execute one instruction.

A MIPS processor can be split into 4 main sections. The Instruction Fetching, the instruction decoding, arithmetic logical unit and memory section. There is another section typically named write back but it is implemented into the instruction decoding section. This overall diagram represents the data path of the MIPS processor.

This design also includes a Control Unit from where all the devices of the circuit is being controlled. Signals from control unit will manipulate what signals to carry through the circuit and what to leave behind.

SLL (R-type)

Shift Left Logical is an R-type function. The MIPS instruction format will orchestrate the SLL design. Since SLL is an R-type instruction, the R-type instruction format is being utilized here.

Opcode	Source reg	Source reg	Destination reg	Shamt	function
31:26	25:21	20:16	15:11	10:6	5:0

Table 2

From table 2, the bits from 26:31 is always the opcode of an instruction. But an R-type instruction requires a function field to differentiate between different arithmetic logical operations. The system's ALU will use these signals to carry out instructions.

Opcode	Funct	Name	Description	Operation
000000	000000	SLL	shift left logical	[rd] = [rt] << shamt

Table 3

Table 3 explains the operation of a SLL instruction. Essentially this command will shift the incoming data by a certain number of bits and output it. But this simple operation creates a problem. This operation will require ALU involvement. Figure 1 shows that the ALU control is only 3 bits.

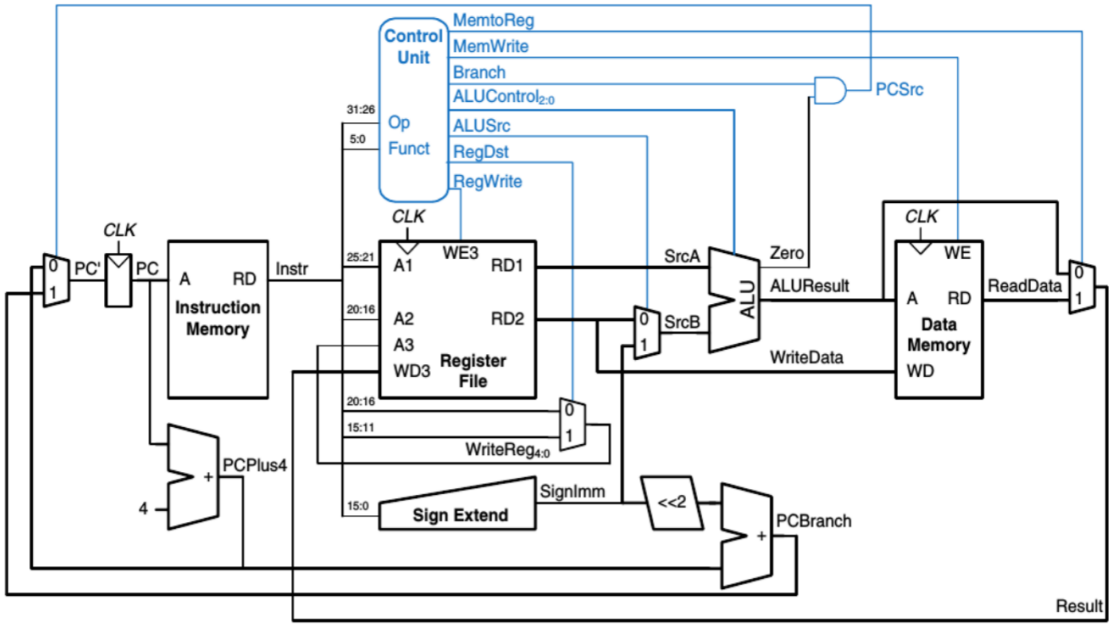

Figure 1: Single cycle MIPS Datapath

Control unit is split into two sections like figure 2. The opcode coming from the instruction set gets recivied into the main decoder and the 2 bits of the main decoder, AluOp in this case is sent to the Alu decoder. Depending on the Aluop signals, the Alu decoder will output Alu control to the ALUnit.

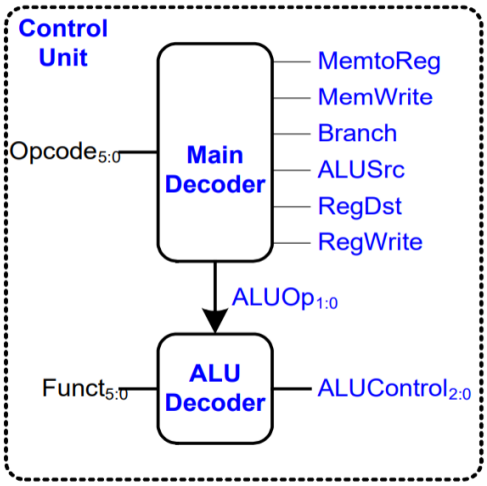

Figure 2: ALU decoder

When the ALUop is 10, the system will look at the function. And the problem occurs when the system goes to the function.

AluOp 1:0	Meaning
00	ADD
01	SUBSTRACT
10	GO TO FUNCTION
11	UNUSED

Table 4

The MIPS processor already has some predefined functions as shown in table 5.

Aluop 1:0	Function	AluControl 2:0
00	X	010 ADD
01	X	110 SUBSTRACT
10	100000	010 ADD
10	100010	110 SUBSTRACT
10	100100	000 AND
10	100101	001 OR
10	101010	111 SLT

Table 5

This implementation leaves no space to include another instruction bitwise. Thus, no ways of selecting the shifted value after the result.

One way of tackling this problem is to add one more bit into the Alucontrol to select the new SLL result. The datapath will look like figure 3.

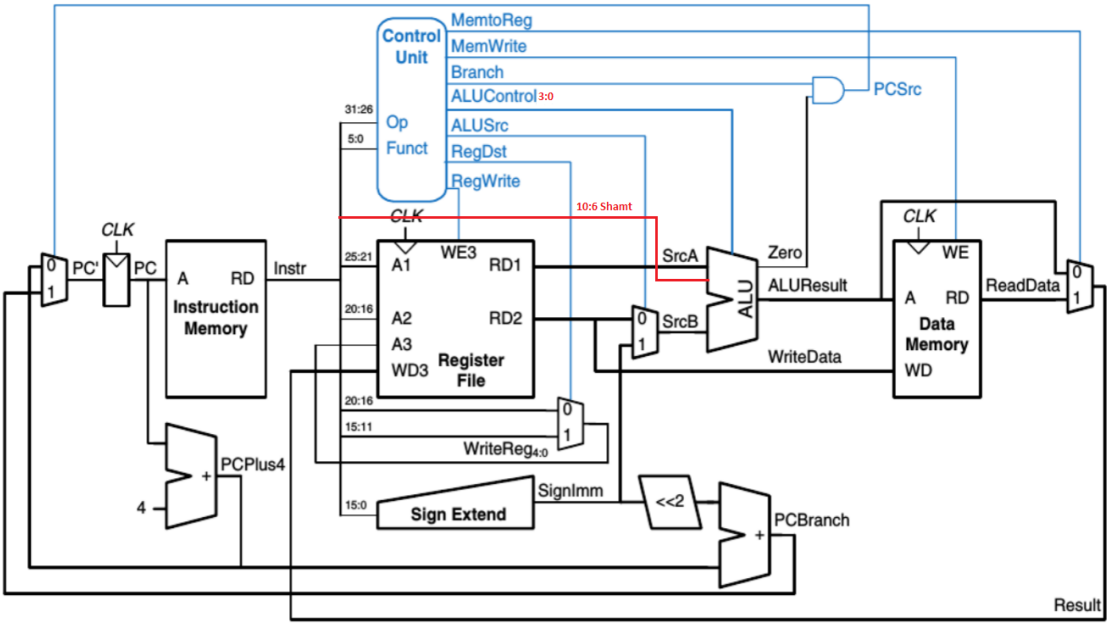

Figure 3: SLL Datapath

The new selection table now becomes table 6. By adding one extra bit, the system now has space to fit in the SLL command.

AluControl 3:0	Operation
0000	A and B
0001	A or B
0010	A+B
0011	UNUSED
1000	A and not B
1001	A or not B
1010	A-B
1011	SLT
0100	SLL

Table 6

The ALU is getting extra information now. This means, support for the SLL instruction must also be included in the ALUnit. The ALUnit receives 4 bits, it uses the MSB to select a B or an inverted B. The other three bits are left to select which output to pass through the result. The new SLL instruction should have a separate output system to carry through. The shamt line, as shown in figure 3, will also come into the ALUnit and a shifting system is added to accommodate this like figure 4.

When the corresponding opcode is selected and the ALUnit has recived 0100, this MIPS processor will perform a logical left shift operation of x amount, x being the value received from shamt control.

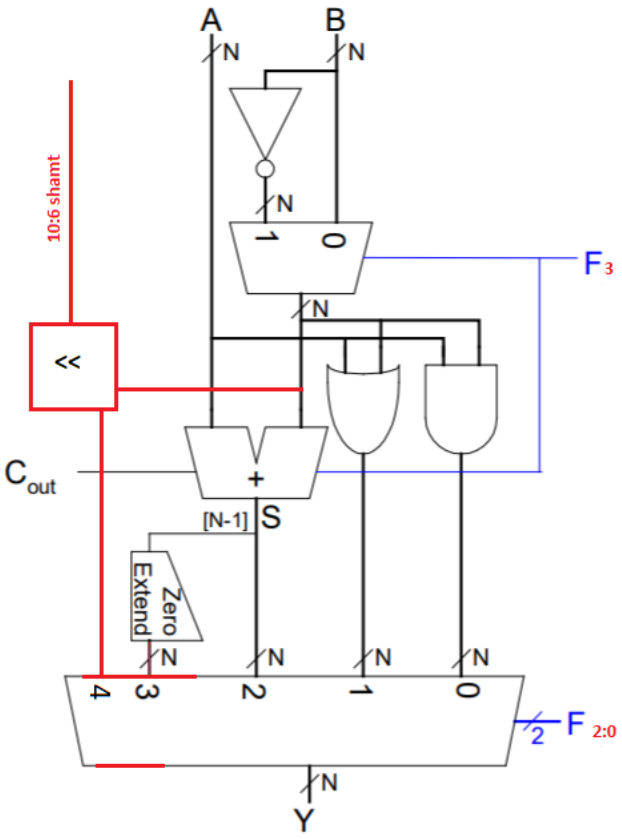

Figure 4: ALU of SLL

Vhdl Code:

To implement the SLL into the Vhdl code, some changes in the data path must be made as well as the design for the ALU control should be changed to a four-bit configuration.

```
signal alucontrol: STD_LOGIC_VECTOR(3 downto 0);
```

Now the system will get an ALUcontrol signal of 4 bits. The code should also update its ALUdecoder to accommodate this change.

```
architecture behave of aludec is
begin
  process(aluop, funct) begin
    case aluop is
      when "00" => alucontrol <= "0010"; -- add (for lw/sw/addi)
      when "01" => alucontrol <= "1010"; -- sub (for beq)
      when "11" => alucontrol <= "1011"; -- the 2nd MSB is the added extra ---bit for SLTI
        when others => case funct is       -- R-type instructions
                  when "100000" => alucontrol <= "0010"; -- add ; the --second MSB bit is -----fitted to accomodate
the slti feature
                  when "100010" => alucontrol <= "1010"; -- sub
                  when "100100" => alucontrol <= "0000"; -- and
                  when "100101" => alucontrol <= "0001"; -- or
                  when "101010" => alucontrol <= "1011"; -- slt
                  when "000000" => alucontrol <= "0100"; -- sll  ------new sll command
```

```
                when others       => alucontrol <=  "----"; -- ???
            end case;
    end case;
  end process;
end;
```

The second MSB is now representing the SLL command. The design must have a way to do the shifting of bits in the ALU decoder.

```
entity alu is
  port(a, b:      in  STD_LOGIC_VECTOR(31 downto 0);
       alucontrol: in  STD_LOGIC_VECTOR(3 downto 0);
       shamt:     in STD_LOGIC_VECTOR(4 downto 0); --shamt control
       result:    buffer STD_LOGIC_VECTOR(31 downto 0);
       zero:      out STD_LOGIC;
       ltez:      out STD_LOGIC); -- new out for less than imm
end;
```

Since shamt is 10:6 which 5 bits, the code will represent this as 4 down to 0 meaning 5 bits. To shift a value by x amount, x being the shamt value, the code will have a process where it receives the shamt amount and the signal to be converted.

Data path of the system must also include shamt control as an input since shamt comes from the instruction line.

```
component datapath
   port(clk, reset:       in  STD_LOGIC;
        memtoreg, pcsrc:  in  STD_LOGIC;
        alusrc:           in  STD_LOGIC_VECTOR(1 downto 0); --accomodating --for LUI
        regdst:           in  STD_LOGIC;
        regwrite, jump:   in  STD_LOGIC;
        alucontrol:       in  STD_LOGIC_VECTOR(3 downto 0); -- ----------accomodating for SLL
        zero:             out STD_LOGIC; --added output for blez
        ltez:             out  STD_LOGIC; --aaded output into controller
        shamt:            in STD_LOGIC_VECTOR(4 downto 0); -- since the ----instruction is an input into the
datapath
        pc:               buffer STD_LOGIC_VECTOR(31 downto 0);
        instr:            in STD_LOGIC_VECTOR(31 downto 0);
        aluout, writedata: buffer STD_LOGIC_VECTOR(31 downto 0);
        readdata:         in  STD_LOGIC_VECTOR(31 downto 0));
  end component;
```

This means the shamt port should be mapped into the data path. It is done by passing shamt into the port mapping function. Do note, that only the 10:6 bits are supposed to be passed through the Datapath.

```
dp: datapath port map(clk, reset, memtoreg, pcsrc, alusrc, regdst,
                      regwrite, jump, alucontrol, zero, ltez, instr(10 downto 6), pc, instr, --following the chronological
order of datapath
                      aluout, writedata, readdata);
```

Since the main ALU receives the shamt function, the shamt should be passed in as well.

```
mainalu: alu port map(srca, srcb,alucontrol,shamt, aluout, zero);
```

Now the system has all the references to shamt as a proper instruction line to data path and the actual task of SLL can be implemented.

```
architecture behave of alu is
  signal condinvb,shifted, sum: STD_LOGIC_VECTOR(31 downto 0);
begin
  condinvb <= not b when alucontrol(2) = '1' else b;
  shifted <= std_logic_vector(unsigned(condinvb) sll to_integer(unsigned(shamt))); -- -----shifting by shamt amount
--must convert to ----unsigned and integer to use sll
  sum <= a + condinvb + alucontrol(2);
```

Here, the default SLL conversion function of vhdl is being used. The SLL conversion function requires an integer that represents the amount of conversion. Thus, the shamt is being converted into an integer. The condinvb is the B value which the system will shift logically to the left. But the SLL requires an unsigned integer to do conversion.

The control system will include a selection method to select the shifted signal as the output.

```
  process(alucontrol, a, b, sum) begin
    case alucontrol(2 downto 0) is
      when "000"   => result <= a and b;
      when "001"   => result <= a or b;
      when "010"   => result <= sum;
      when "100"   => result <= shifted; -- when the input is 100 the output --is selected as shifted
      -- slt should be 1 if most significant bit of sum is 1
      when "011"   => result <= (0 => sum(31), others => '0');
      when others => result <= (others => 'X');
    end case;
  end process;
```

Here, the system will set the result of the ALU to be the shifted value when the selection is 100.

There are other ways to implement the SLL such as running a loop x amount of times, x being the shamt value, and shifting the B value by '0' each time. A quick example is shown below.

```
  process(alucontrol, a, b, sum)
   variable X : --declare variable for loop
        integer range 0 to 32;
   begin

     X := 0; --set counter to 0
    while (I <= to_integer(unsigned(shamt))) loop --start the loop
           shifted <= condinvb(30 downto 0) & "0"; --shift left by 1 ------bit(run it a shamt ----times);
          X := X + 1; --increment counter
        end loop; --end of loop
```

This code is not implemented. It is rather a thought experiment.

Now the system is tested and designed in vhdl and it looks like figure 5. Where the blue line coming into the main ALU gets shift by the x amount where x is the shamt value and being sent to the output as the ALUresult.

The Select line can select this shifted data as the output and the system can carry on with its instruction.

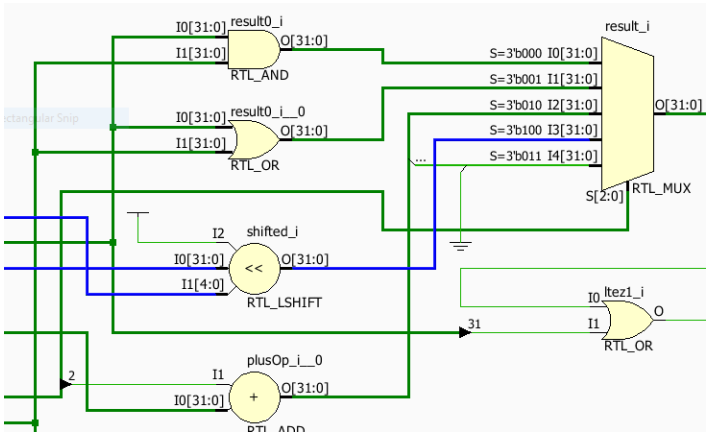

Figure 5: SLL digital Datapath

The SLL can also be implemented by having a control signal and setting the value to '1' only when an SLL operation is required. A shift component can carry B and shamt line as input and shift the value while passing it to the output of ALU. This is also not implemented.

SLTI

SLTI is set less than immediate. If a given value is less than the immediate value, the rt is set to '1', otherwise it is set to '0'. For SLTI, the previous design from SLL is resourced. The extra bits left from ALUcontrol can be set to accommodate SLTI. The instruction format and logic for SLTI is shown in table 7.

Opcode	Funct	Name	Description	Operation
001010	--	slti	set less than immediate	[rs] < SignImm ? [rt] = 1: [rt] = 0

Table 7

If the given condition is satisfied, the controls will be set accordingly meaning the system will perform the SLTI instruction.

The control format should be set like Table 8.

Instructions	OP [5:0]	Reg Write	Reg Dst	ALU Src [0:1]	Branch	Mem Write	Memto Reg	Jump	ALUOp [1:0]
SLTI	001010	1	0	10	0	0	0	0	11

Table 8

Now, the ALU control logic should be updated to match the SLTI instruction. The previous increment of ALUsrc leaves the design with enough bits to assign another instruction. Table 9 shows the new instruction.

AluControl 3:0	Operation
1011	SLTI

Table 9

Vhdl Code:

The vhdl code will only require one additional when statement to make the system set the corresponding command.

```
architecture behave of maindec is
  signal controls: STD_LOGIC_VECTOR(10 downto 0);
begin
  process(op) begin
    case op is
       when "001010" => controls <= "10100000110"; -- SLTI opcode from wiki
```

The ALUcontrol should now be able to select the SLTI implementation. Here the remaining '11' from ALUop is being utilized.

```
architecture behave of aludec is
begin
  process(aluop, funct) begin
    case aluop is
      when "00" => alucontrol <= "0010"; -- add (for lw/sw/addi)
      when "01" => alucontrol <= "1010"; -- sub (for beq)
      when "11" => alucontrol <= "1011"; -- the 2nd MSB is the added extra ---bit for SLTI
```

As mentioned before, the control signals must pass the values set for SLTI. It is done by specifying the bits of controls and then assigning them to the control signals.

```
    regwrite <= controls(10);
    regdst   <= controls(9);
    alusrc   <= controls(8 downto 7);
    branch   <= controls(6);
    memwrite <= controls(5);
    memtoreg <= controls(4);
    jump     <= controls(3);
    aluop    <= controls(2 downto 1);
    blez     <= controls(0);
```

That is all the change needed for this instruction.

The SLTI instruction can be performed by sending the corresponding opcode and the system will analyze the opcode and set the control signal to accommodate this instruction and the ALUop will be set to '11' and the ALUcontrol to '1011' and the system will subtract the values and check if the result is 'not 0' as 0 meaning they are equal and being anything other than '0' is a less than, except when the sign bit (MSB) is one. If these conditions are met, set the control signals to '1'.

LUI

LUI stands for load upper immediate. This instruction will push a literal value into a register and shift it by 16 bits and store it back into the register. The instruction format is better represented in table 10.

Opcode	Funct	Name	Description	Operation
001111	--	lui	load upper immediate	[rt] = {imm, 16'b0}

Table 10

To implement this, a 16 bit shifter is required. But this introduces a new problem. As the selection for Source B (represented as Srcb) is done by the multiplexer which is connected to ALUsrc control. It is only a single bit meaning it can have either a '0' or a '1'. And the system requires a 16 bit shifted signal to go through the multiplexer. This can be resolved by having a bigger ALUsrc control. If it is incremented to 1:0, the system will have one more selection option and the 16 bit shifted value can then be passed as srcb. The design will look like figure 6.

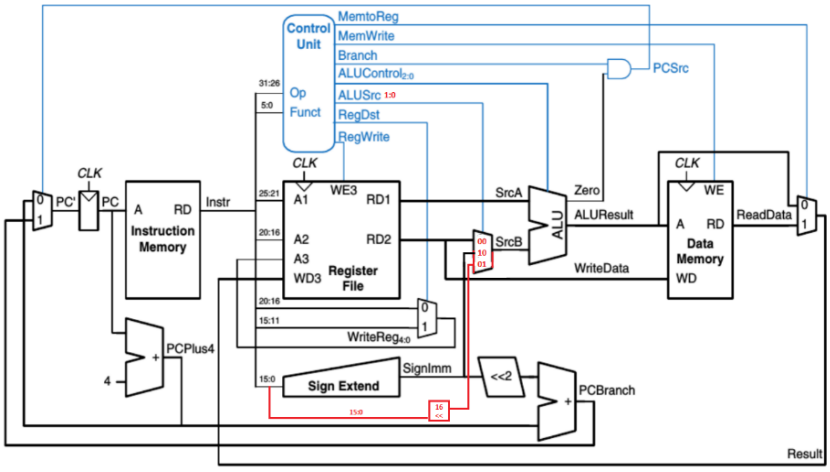

Figure 6: LUI datapath

Now the ALUsrc can select '01' to pass the 16-bit shifted value as SrcB. The updated control signal will look like table 11.

Instructions	OP [5:0]	Reg Write	Reg Dst	ALU Src [0:1]	Branch	Mem Write	Memto Reg	Jump	ALUOp [1:0]
LUI	001111	1	0	01	0	0	x	0	00

Table 11

Vhdl Code:

First the controller needs to have an incremented bit value.

```
alusrc:          out STD_LOGIC_VECTOR(1 downto 0); -- ----------accomodating for -----LUI output from controller is input to datapath
```

The opcode should be assigned to the controls signals.

```
process(op) begin
  case op is
    when "001111" => controls <= " 10010000000 "; -- LUI opcode from wiki
```

The component to shift must also be included.

```
component lui_shifter       -- this component will shift the 15:0 bits and make them 32 bits
  port(a: in STD_LOGIC_VECTOR(15 downto 0);
       y: out STD_LOGIC_VECTOR(31 downto 0));
end component;
```

The input is 15:0 and it will be shifted and set as a 32-bit output. To represent the 32-bit signal, the code must add a 32 bit vector line.

```
signal luiimmediate: STD_LOGIC_VECTOR(31 downto 0);
```

Now this shifted signal needs port mapping. The 15:0 bits of the instruction line and the LUI shift signal should be mapped into the new line.

```
luish: lui port map(instr(15 downto 0), luiimmediate); --shifting
```

The multiplexer at source B needs to take 3 inputs now. For this a new multiplexer design is required.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux3 is
  generic(width: integer); --generic lets the system to receive values -------outside of its -----declaration
  port(d0, d1, d2: in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:      in  STD_LOGIC_VECTOR(1 downto 0);
       y:      out STD_LOGIC_VECTOR(width-1 downto 0));
end;
architecture behave of mux3 is
begin
  y <= d0 when s = "00"
  else d1 when s = "01"
  else d2 when s = "10";
  end;
```

Next, this new multiplexer needs to be declared within the SrcB line for the system to use this multiplexer system.

```
  srcbmux: mux3 generic map(32) port map(writedata, luiimmediate, signimm, alusrc, ------new -----mux ---added
to source line
                                  srcb);
```

LUI command will shift the incoming bits into 32-bit output.

```
architecture behave of lui_shifter is
begin
  y <= X"0000" & a;   -- x for hex
end;
```

Here the 16 zeros are being concatenated with the ‘a’ where ‘a’ is the 16-bit input.

Now the system should have a new multiplexer design that takes in a two-bit vector and selects the new shifted LUI value to be pushed through Srcb. The new datapath is displayed in figure 7.

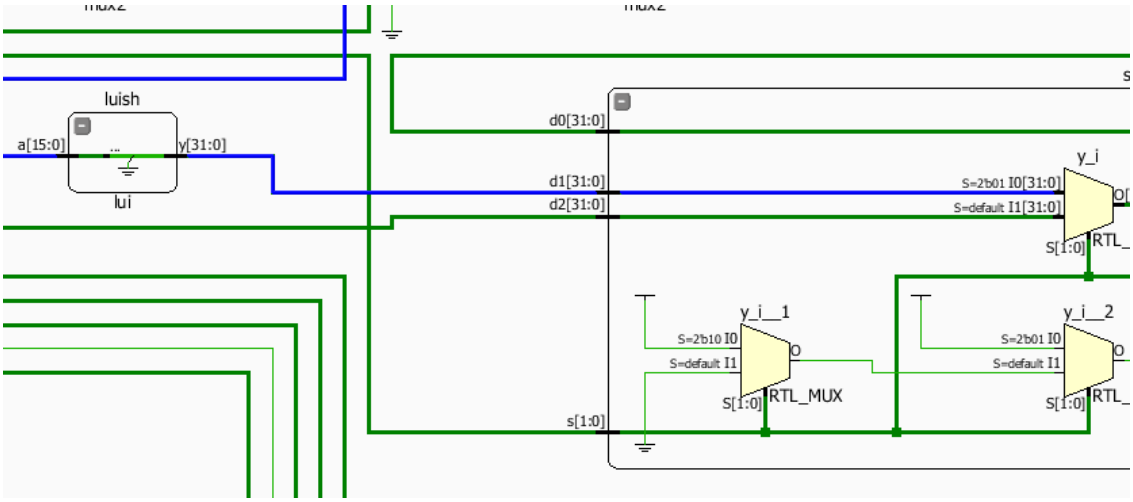

Figure 7: LUI digital Datapath

Other possible ways of implementing this would be to have a LUI control line that can select the shifted LUI value and pass it through a multiplexer. The system can then select this line if required and select the general ALUresult line if not required. But this method would require control changes for all the other instructions which is complicated and inefficient.

BLEZ

Blez stands for branch if less than or equal zero. In hardware logic. A value is less than zero when the MSB is '1' meaning a negative value as this system use sign bit to represent negatives and positives. To implement this, the system needs to do two checks, firstly checking if it is equal to zero, secondly check if its less than zero, and finally make sure at least one of these conditions match. If they do match, the system needs to do a branch operation which is already in place. The instruction logic for BLEZ is displayed in table 12.

Opcode	Funct	Name	Description	Operation
000110	--	blez	branch if less than or equal to zero	if ([rs] <= 0) PC = BTA

Table 12

For this, a new control signal is required. In the control Unit, a new control set is shown as Blez in Figure 8.

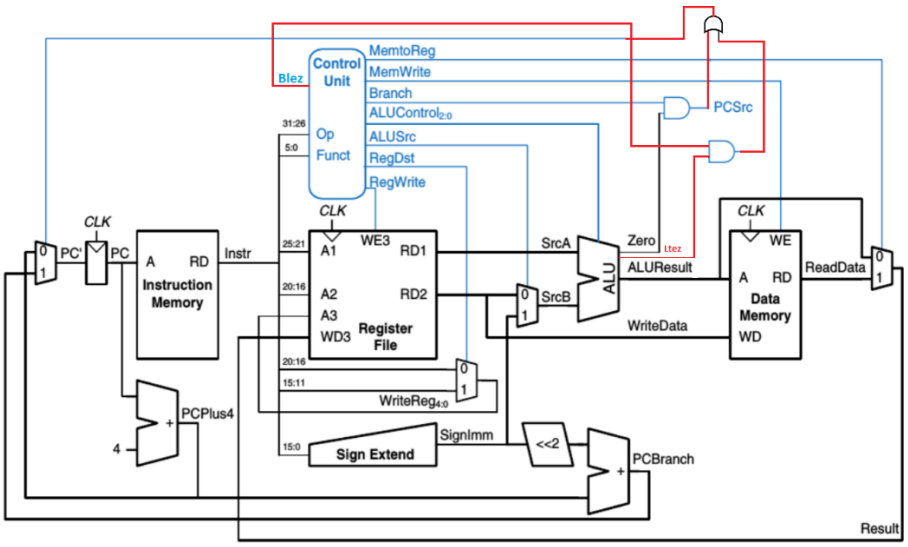

Figure 8: BLEZ data path

The system already performs a branch AND zero operation. AND gates produce a value of '1' only when both are '1'. But the sign bit needs to be '1' to represent that the value is less than '0'. To implement this, a logical operation is required in the ALUnit to check the sign bit.

This implementation requires the ALUnit to take the 31st bit of the output of the ALUresult and perform an OR operation against a literal zero. For instance, if the the last bit of ALUresult is '0' and it is ORed with a zero flag, the output of that OR gate is '0'.

$$x = 0; 0(31st\ Bit)\ OR\ (ZERO) = 0$$

But if the x value is '1', meaning if the ALUresult is negative, the output will be a '1'. In equation:

$$x = 1; 1(31st\ Bit)\ OR\ (ZERO) = 1$$

So, this implemented in the ALUnit will look like figure 9.

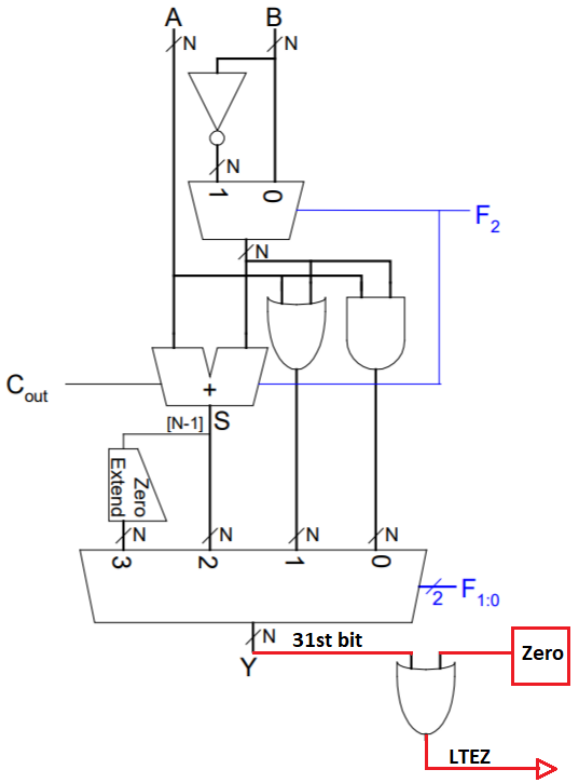

Figure 9: ALU of BLEZ

The ALUnit result can be equal if the subtraction is '0'. That is when the zero flag is set to '1'. But the other condition is if it less than zero. And figure 9 is checking if the result satisfied that condition. Now the system needs to check if one or both of the conditions match. The branch control is set to '1' as it is a branching operation, the zero flag may be set if the values are equal, the LTEZ may be a '0' or a '1' depending on the last bit of the Output. For example;

$$x = 1; 1(31st\ Bit)\ OR\ (ZERO) = 1; LTEZ = 1; Branch = 1; ZERO\ FLAG = 1; 1\ AND\ 1 = 1$$

This example satisfies both conditions. To make this work, the system needs to check if the LTEZ is required to be '0' or not. To do that, an AND operation with the control signal is needed.

$$LTEZ = 1; BLEZ = 1; 1\ AND\ 1 = 1$$

Since to perform this operation, only one or both conditions need to be met, an OR operation of 'less than 0' and 'equal to 0' can be performed.

$$(BLEZ\ AND\ LTEZ)\ OR\ (BRANCH\ AND\ ZERO\ FLAG)$$

If one of them is '1', the conditions are met since an OR gate produces '1' when one of the variables are '1'. If all the conditions are met, the system will select '1' for the PC' multiplexer, which is the PC BRANCH output.

Control signal for this operation is shown in table 13.

Instructions	OP [5:0]	Reg Write	Reg Dst	ALU Src [0:1]	Branch	Mem Write	Memto Reg	Jump	ALUOp [1:0]
BLEZ	000110	0	x	00	1	0	x	0	01

Table 13

Vhdl Code:

This design will require a blez and a less than or equal controller.

```
architecture struct of mips is
  component controller
    port(op, funct:         in  STD_LOGIC_VECTOR(5 downto 0);
         zero,less_equal_zero:        in  STD_LOGIC;
```

The data path will have the same control lines but as an output.

```
component datapath
    port(zero,less_equal_zero:        out STD_LOGIC;
```

To map these, a signal line is needed to pass through the controller and data path mapping.

```
  signal zero,less_equal_zero: STD_LOGIC;
```

Then map these signals into the ports of data path and controller paths.

```
  cont: controller port map(instr(31 downto 26), instr(5 downto 0),
                            zero, less_equal_zero, memtoreg, memwrite, pcsrc, alusrc,
                            regdst, regwrite, jump, alucontrol);
  dp: datapath port map(clk, reset, memtoreg, pcsrc, alusrc, regdst,
                        regwrite, jump, alucontrol, zero, less_equal_zero,instr(10 downto 6), pc, instr,
                        aluout, writedata, readdata);
```

Blez is now also a control line from the main decoder.

```
blez:       out STD_LOGIC;
```

The control signals themselves should now have blez as a control line but it will only be '1' when blez is required.

```
begin
  process(op) begin
    case op is
      when "000110" => controls <= "00000100001"; -- BLEZ
```

Now the core calculation of Blez can be added into the ALU. Here, the code should set less_equal_zero to '1' if the output of the ALUresult is all '0' or the 31st-bit of the SrcA is '1' since in both cases, it's either less than or equal to zero. Otherwise it should be set to '0'.

```
less_equal_zero <= '1' when (result = X"00000000" or a(31)= '1') else '0';
```

But the PCsrc should branch if Blez or branch is required. To do that, the code needs an OR operation.

```
  pcsrc <= (branch and zero) or (blez and less_equal_zero);
```

That is all the changes required for BLEZ support.

Test Bench

The default test bench checks the value of address 84. The expected value is 7. This design will keep the expected value same for simplicity. Instead the opcodes along with some instructions are added to test the design. The list of opcodes along with the instructions are;

Instruction Hex	Instruction Names
20020005	ADDI
2003000c	ADDI
2067fff7	ADDI
00e22025	OR
00642824	AND
00a42820	ADD
00421000	SLL
3C0F6B6A	LUI
10a7000a	BEQ
28640005	SLTI
18800001	BLEZ
20050000	ADDI
00e2202a	SLT
00853820	ADD
00e23822	SUB
ac670044	SW
8c020050	LW
08000011	J
20020001	ADDI
ac020054	SW

The actual test bench code does not require any changes. A file with '.dat' extension should be created with these codes (only hex codes) is needed as the test bench code only checks for hex values.

After simulating the test bench, a wave graph like figure 10 is expected where the result is 7 and 84.

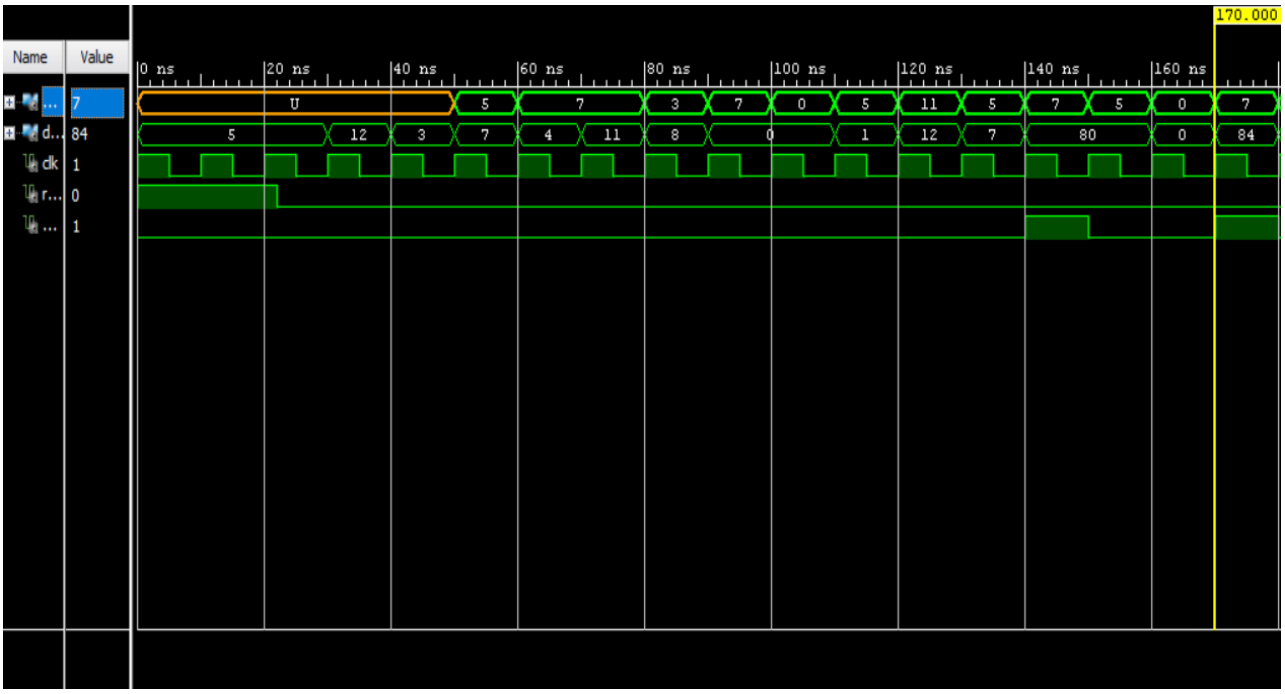

Figure 10: Wavelengths of write data and dataddr

The stream of data and address are that of the instructions, for instance the last address of the instruction is 84 and the data is 7. The memwrite control signal is also a '1' here. Do note that not all instructions can be represented through this graph. But since the result and address is being manipulated, it can be said that this test bench is working, and the design of this MIPS is responding properly to the instructions.

Links to the complete design file: Design

Links to the schematic of the whole design: Schemetic