KALYAN BOPPANA

011608741

VERILOG REPORT

CS-525 ADVANCED COMPUTER ARCHITECTURE

**TABLE OF CONTENTS:**

**INTRODUCTION-------------------------------------------------------------2-3**

**ADD INSTRUCTION---------------------------------------------------------3-5**

**ADDI INSTRUCTION -------------------------------------------------------6-8**

**LW INSTRUCTION ----------------------------------------------------------8-10**

**SW INSTRUCTION -----------------------------------------------------------11-14**

**NAND INSTRUCTION -------------------------------------------------------14-17**

**LUI INSTRUCTION ----------------------------------------------------------18-20**

**BEQ INSTRUCTION----------------------------------------------------------20-23**

**JALR INSTRUCTION --------------------------------------------------------23-25**

**EXTRA OBSERVATIONS---------------------------------------------------26**

**DESIGN FILE-------------------------------------------------------------------27-41**

**TESTBENCH -------------------------------------------------------------------42**

**WAVEFORM--------------------------------------------------------------------43**

**MIPS PROCESSOR:** The processor without any interlocked pipeline stages is called as MIPS processor which has reduced set of instructions. Which can perform limited set of operations but with more speed and accuracy. It is developed MIPS computer systems.

**RISC 16**: RiSC stands for Ridiculously Simple Computer

Here we design the microprocessor for 16-bit instructions and then we are going to check the output for specific operations we are going to perform on microprocessor.

**DIVISION OF 16-BITS:**

Here we are having 16 bits to pass the instruction for microprocessor and we should specify the required operation, target registers and source registers.

So, when we have 16 bits of data we are going to declare specific bits for particular operations below

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Op-code | RegA | RegB | reserved | RegC |

Op-code = [15:13]

RegA = [12:10]

RegB = [9:7]

Reserved = [6:2]

RegC = [0:2]

So, here the most significant bits from [15:13] are considered as **OPCODE.**

**OPCODE**: The code which calls the required function to perform operation in ALU. The next sequence of bits **[12:10**] are assigned for **Target Register**.

[9:7] is going to be the **SOURCE REGISTER-1** and the next sequence is **RESERVED BITS** which can be used or allocated depending upon the operation we perform. [2:0] is going to be sequence for **SOURCE REGISTER-2.**

**LOAD INSTRCUTIONS**: So for any microprocessor instructions can be given by two types:

1. From the High level languages
2. From the Text file which has instructions.

Here we are going to give the instructions to processor by text file called “**P2INST.MIPS**”.

// **Load Instructions**

$readmemb("p2inst.mips",instruction\_memory);

**INSTRUTCION FETCHING:** After loading instructions the they should be fetched to perform operations. The instructions are fetched into INSTRUCTION MEMORY

// Instruction Fetch

$display("\nPC = %b", pc);

instruction = instruction\_memory[pc];

Instructions are being fetched from Instruction memory.

**Execute**: Next stage is to execute the instructions and storing the result in destination registers. Before doing that we should DECODE the Instructions.

This report consists of implementation of instruction using VERILOG along with results and data path diagram of each Instructions.

First let’s take ADD instruction:

**ADD**:

When we are dealing with microprocessors we should pass the instructions through machine code which takes the following format for particular ADD operation

**add regA, regB, regC R[regA] <- R[regB] + R[regC]**

So we can see the ADD operation at the very beginning of the instruction and then **target register regA** and **source** **regB** and **regC** from which the input values are generated.

**//add instruction**

if(opcode == 3'b110) begin

$display("Instruction = ADD : %b = %b + %b --> (R%d = R%d + R%d) " ,

instruction[12:10], instruction[9:7], instruction[2:0], instruction[12:10], instruction[9:7], instruction[2:0]);

reg\_address\_A = instruction[12:10];

cs\_write\_reg = 1;

cs\_alu = 4'b0110;

registers[1] = 5;

registers [2] = 3;

cs\_alu\_select = 0; //selecting source 2 register

end

From the above code we are going to perform ADD instruction

So, **reg\_address\_A** is assigned for output value and **cs\_write\_reg** bit is set to **1** which allows to **write back** the output in destination register.

So, as mentioned in design file we are selecting the **cs\_alu\_select** =**0** to use the source register2 for this **ADD** operation.

Now we should go through the ALU operations

**// ALU START**

case (cs\_alu)

4'b0110: // add

begin

alu\_result = alu\_operand0 + alu\_operand1;

alu\_overflow = 0;

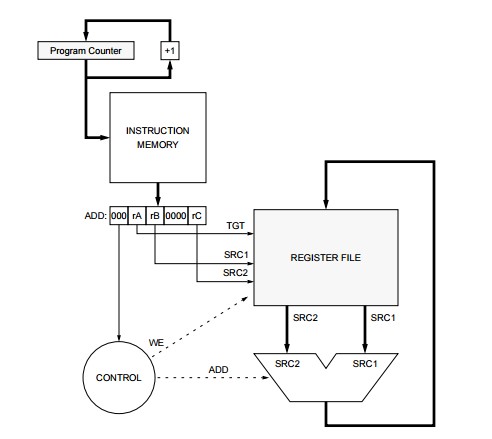
alu\_zero = (alu\_result == 0) ? 1 : 0;

$display ("Added %d + %d = %d", alu\_operand0, alu\_operand1, alu\_result);

end

So, here we are going to add **operand0** and **operand1** and store the result in **alu\_result**. After this execution the program counter is incremented by one and goes for next instruction.

The flow of control of instructions are going to be shown in figure below:



R3= 011

Target register

R1=001

SR-1

R2=010

SR-2

**OUTPUT**: After simulating the code for ADD we got the following output

# KERNEL: PC = 0000000000000000  
# KERNEL: IF = 1100110010000010  
**# KERNEL: Opcode = 110  
# KERNEL: Instruction = ADD : 011 = 001 + 010 --> (R3 = R1 + R2)**   
# KERNEL: Added 5 + 3 = 8  
**# KERNEL: Write R3 = 8**# KERNEL:   
# KERNEL: Print Registers:  
# KERNEL: R 0 = 0  
# KERNEL: R 1 = 5  
# KERNEL: R 2 = 3  
**# KERNEL: R 3 = 8**  
# KERNEL: R 4 = 0  
# KERNEL: R 5 = 0  
# KERNEL: R 6 = 0  
# KERNEL: R 7 = 0

**PC: Program counter**: Which is required to increment the value and should go for another instruction.

**IF: Instruction Fetch** and we have given OPCODE for ADD=110

Now the operation is being performed and stored in R3.

The whole DESING.V FILE and OUTPUT are given at the end of report.

Above we are just mentioning the output for ADD instruction.

**ADDI: ADD Immediate value**

Sometimes we can give a direct value to microprocessor to perform operation. So **ADDI** deals with this kind of operations in which we can give direct value in binary form to the ALU.

The machine format is mentioned below:

**ADDI regA regB signed immediate (-64 to 63)**

So ADDI is the operand and regA is destination register and regB and immediate value.

Now we should decode the instruction from P2INST.MIPS file.

**//addi instruction**

if(opcode == 3'b001) begin

$display ("Instruction = ADDI : %b = %d", instruction[6:0], instruction[6:0]);

reg\_address\_A = instruction[12:10];

cs\_write\_reg = 1;

cs\_alu = 4'b0001;

cs\_alu\_select = 1; //selecting immediate value

end

So, here **cs\_alu\_select=1;**

Because we are going to take an immediate value [6:0] and cs\_write\_reg=1; to write the result in destination registers. Now we can have a look towards ALU side

4'b0001: // Signed add

begin

alu\_result = alu\_operand0 + alu\_operand1;

if ((alu\_operand0 >= 0 && alu\_operand1 >= 0 && alu\_result < 0) ||

(alu\_operand0 < 0 && alu\_operand1 < 0 && alu\_result >= 0)) begin

alu\_overflow = 1;

end else begin

alu\_overflow = 0;

end

alu\_zero = (alu\_result == 0) ? 1 : 0;

$display("Added %d + %d = %d",alu\_operand0, alu\_operand1, alu\_result);

end

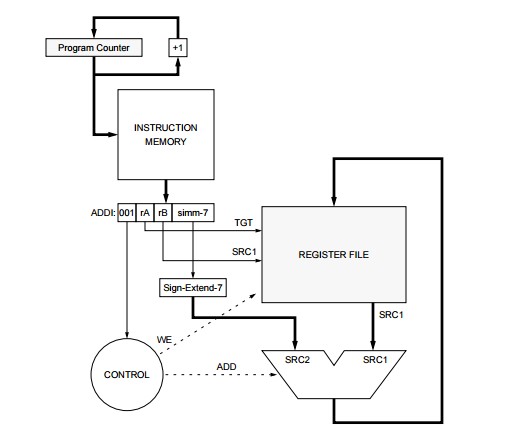
So, from ADD to ADDI the program counter is incremented to 1 and then Fetching the Instruction from the File to perform operation and then the result is going to be stored in R4.

The flow of controls is explained in the below figure:

Directly fetched to ALU

OPCODE

Immediate value



R1

(001)

for next

R0

(000)

7

Now the ALU performs the operation and in this SORUCE REGISTER-2 is not going to be used, we are giving the whole bits [0:6] as **immediate value** and calling that directly to the ALU to perform ADDI operations. The output is given below for this ADDI instruction.

**OUTPUT**:

# KERNEL: PC = 0000000000000001  
# KERNEL: IF = 0011000100101010  
**# KERNEL: Opcode = 001  
# KERNEL: Instruction = ADDI : 0101010 = 42  
# KERNEL: Added 3 + 42 = 45  
# KERNEL: Write R4 = 45**# KERNEL:   
# KERNEL: Print Registers:  
# KERNEL: R 0 = 0  
# KERNEL: R 1 = 5  
# KERNEL: R 2 = 3  
# KERNEL: R 3 = 8  
**# KERNEL: R 4 = 45**  
# KERNEL: R 5 = 0  
# KERNEL: R 6 = 0  
# KERNEL: R 7 = 0

**LW: Load Word**

In this the **memory** is loaded in to the regA. The **memory** is fetched from the address formed by adding immediate value to the regB.

So while performing this instruction we have additional data memory, which takes care of data\_addresses.

The machine format of LW is formed by following pattern:

**lw regA, regB, immediate**

So, the design for this instruction is given below:

//lw instruction

if(opcode == 3'b010) begin

$display("Instruction = LW : %b = %d", instruction[6:0], instruction[6:0]);

reg\_address\_A = instruction[12:10];

cs\_read\_data\_memory = 1;

data\_memory[2] = 21;

data\_memory[4] = 37;

registers[1] = 5;

cs\_alu = 4'b0010;

cs\_alu\_select = 1; //selecting immediate value

end

Here cs\_alu\_select=1; because we are going to use immediate value in this instruction. We use **cs\_read\_data\_memory** instead of **cs\_write\_data** because load operation is done to read the data from memory to registers.

Now ALU operation is illustrated below:

4'b0010:

begin

data\_address = immediate; //Data Access

alu\_result = alu\_operand0 + data\_memory[data\_address];

alu\_overflow = 0;

alu\_zero = (alu\_result == 0) ? 1 : 0;

$display("LW Added: R[%d] =(R[%d] = %d) + (D[%d] = %d) = %d",reg\_address\_A,reg\_address\_B, alu\_operand0, data\_address, data\_memory[data\_address], alu\_result);

end

Now we should load the values into the registers.

So we should set the **cs\_read\_data\_memory = 1**; and the code is given below:

**//For LW Instruction: Load From Memory to Reg**

if(cs\_read\_data\_memory == 1) begin

registers[reg\_address\_A] = alu\_result;

$display("Register Changed: R%d = %d", reg\_address\_A, registers[reg\_address\_A]);

$display("\nPrint Data Memory:");

for(i = 0; i < 8; i = i+1) begin

$display("D[%d] = %d",i, data\_memory[i]);

end

$display("\nPrint Registers:");

for(i = 0; i < 8; i = i+1) begin

$display("R%d = %d",i, registers[i]);

end

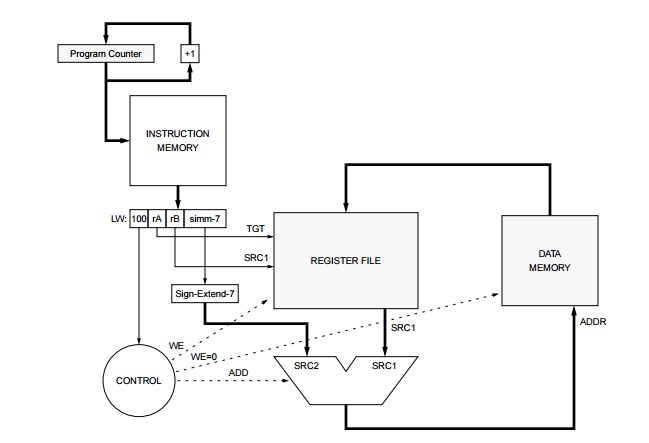
end

So, now we can see **The output** of this instruction:

**# KERNEL: Instruction = LW : 0000010 = 2  
# KERNEL: LW Added: R[5] =(R[1] = 5) + (D[ 2] = 21) = 26  
# KERNEL: Register Changed: R5 = 26**# KERNEL:   
# KERNEL: Print Data Memory:  
# KERNEL: D[ 0] = x  
# KERNEL: D[ 1] = x  
**# KERNEL: D[ 2] = 21**  
# KERNEL: D[ 3] = x  
**# KERNEL: D[ 4] = 37**  
# KERNEL: D[ 5] = x  
# KERNEL: D[ 6] = x  
# KERNEL: D[ 7] = x  
# KERNEL:   
# KERNEL: Print Registers:  
# KERNEL: R 0 = 0  
# KERNEL: R 1 = 5  
# KERNEL: R 2 = 3  
# KERNEL: R 3 = 8  
# KERNEL: R 4 = 53  
**# KERNEL: R 5 = 26**  
# KERNEL: R 6 = 0  
# KERNEL: R 7 = 0

So, from the above output you can see that the data from reg[1] and data\_memory[2] are getting added and going to be stored in the output register R5.

The flow of controls in this LW instruction is given below:



R[1]=5

R[5]=26

Opcode=010

D [ 2]=21

D [4 ]=37

From the above Flow of Controls R[5] = output of the instruction //Target register

D[2]=21; D[4]=37; are the values assigned for the data memory which are to be accessed and R[1]=5 is the value to which the data\_memory value is to be added.

**SW: STORE WORD**

The store word instruction is used to store the data from a regA into the regB value which is formed by adding an immediate value to the contents in regB.

The machine format for SW instruction is described below:

**sw regA, regB, immediate**

**//sw instruction**

if(opcode == 3'b011)

begin

$display("Instruction = SW : %b = %d", instruction[6:0], instruction[6:0]);

reg\_address\_A = instruction[12:10];

cs\_write\_data\_memory = 1;

cs\_alu = 4'b0011;

registers[1] = 22;

registers[7] = 45;

cs\_alu\_select = 1;

end

**The cs\_alu\_select=1**; // Here the **immediate value** is going to be utilized.

The ALU operation is given by the following code:

4'b0011: begin

data\_address = immediate; //Data Access

alu\_result = registers[reg\_address\_A] + alu\_operand0;

alu\_overflow = 0;

alu\_zero = (alu\_result == 0) ? 1 : 0;

$display("SW Added: (R[%d] = %d) + (R[%d] = %d) = %d", reg\_address\_A, registers[reg\_address\_A], reg\_address\_B, alu\_operand0, alu\_result);

end

Now we should write the values into the registers.

So we should set the **cs\_write\_reg=1**; and the code is given below:

//**For SW Instruction: Store from Reg to Memory**

if (cs\_write\_data\_memory == 1) begin

data\_memory[data\_address] = alu\_result;

$display("Data\_Memory Changed: D[%d] = %d", data\_address, data\_memory[data\_address]);

$display("\nPrint Registers:");

for(i = 0; i < 8; i = i+1) begin

$display("R%d = %d",i, registers[i]);

end

$display("\nPrint Data Memory:");

for(i = 0; i < 8; i = i+1) begin

$display("D[%d] = %d",i, data\_memory[i]);

end

end

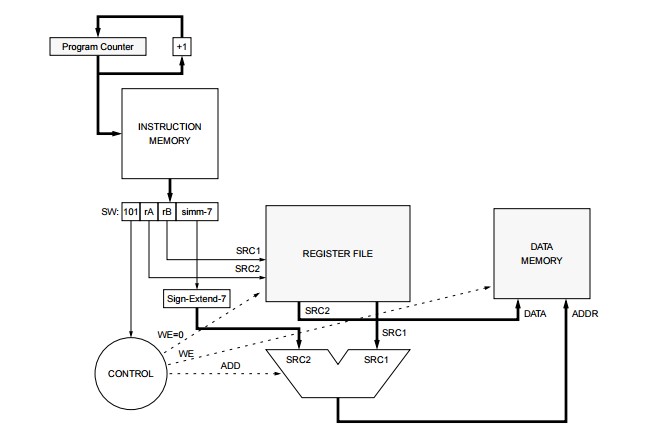
Here we are going to begin the ALU operation by giving its opcode. Now the output is given below:

# KERNEL: PC = 0000000000000011  
# KERNEL: IF = 0111110010000100  
**# KERNEL: Opcode = 011  
# KERNEL: Instruction = SW : 0000100 = 4  
# KERNEL: SW Added: (R[7] = 45) + (R[1] = 22) = 67  
# KERNEL: Data\_Memory Changed: D[ 4] = 67**# KERNEL:   
# KERNEL: Print Registers:  
# KERNEL: R 0 = 0  
**# KERNEL: R 1 = 22**  
# KERNEL: R 2 = 3  
# KERNEL: R 3 = 8  
# KERNEL: R 4 = 53  
# KERNEL: R 5 = 26  
# KERNEL: R 6 = 0  
**# KERNEL: R 7 = 45**  
# KERNEL:   
# KERNEL: Print Data Memory:  
# KERNEL: D[ 0] = x  
# KERNEL: D[ 1] = x  
# KERNEL: D[ 2] = 21  
# KERNEL: D[ 3] = x  
**# KERNEL: D[ 4] = 67**  
# KERNEL: D[ 5] = x  
# KERNEL: D[ 6] = x  
# KERNEL: D[ 7] = x

From the above output we can see that program counter is implemented first and then Instruction is going to be fetched to perform operations.

Then we are displaying the opcode given to the SW operation and then the memory in the R[7] and R[1] are going to be stored in the Data memory of Data address[4].

The flow of control of data paths is going to be illustrated in figure below:



Immediatevalue

Write bit enabled

R[1] =22

R[7]=45

So, first the values of R[1]=22 and R[7]=45 are added and stored in the data memory[4] which is D[4]=67 which can be seen in **PRINT DATA MEMORY.**

**NAND INSTRUCTION:** The negation for the AND function can be called as NAND function. The truth table for NAND is:

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **NAND** |
| **0** | **0** | **1** |
| **0** | **1** | **1** |
| **1** | **0** | **1** |
| **1** | **1** | **0** |

**The machine format for NAND gate is:**

**nand regA, regB, regC**

The function of NAND can be shown like**:**

**regA= ~ (regB & regC)**

**The design instruction of NAND is given below:**

**//nand instruction**

if(opcode == 3'b111)

begin

reg\_address\_A = instruction[12:10];

registers[1] = 5;

registers[2] = 4;

$display("Instruction = NAND : (R[%d] = %d = %b) ~& (R[%d] = %d = %b)" ,

instruction[9:7], registers[instruction[9:7]], registers[instruction[9:7]], instruction[2:0],

registers[instruction[2:0]], registers[instruction[2:0]]);

cs\_alu = 4'b0111;

cs\_write\_reg = 1;

cs\_alu\_select = 0; //taking source register 2

end

Now after the design file we should give instructions to **ALU**:

**4'b0111:**

begin

alu\_result = ~(alu\_operand0 & alu\_operand1);

alu\_overflow = 0;

alu\_zero = (alu\_result == 0) ? 1 : 0;

$display("NAND %b ~& %b = %b", alu\_operand0, alu\_operand1, alu\_result);

trigger\_A = 1;

end

end

So, **trigger** is used for this NAND instruction. First this trigger is a control signal which can be given for specific operation and we are declaring **trigger=1**; This is for NAND operation only and we should end this trigger after NAND operation, because this can make an impact on the further instructions.

Now we should activate this trigger when we are writing back to registers by following code:

if(trigger\_A == 1) begin

$display("NAND R[%d] = %b", reg\_address\_A, registers[reg\_address\_A]);

end

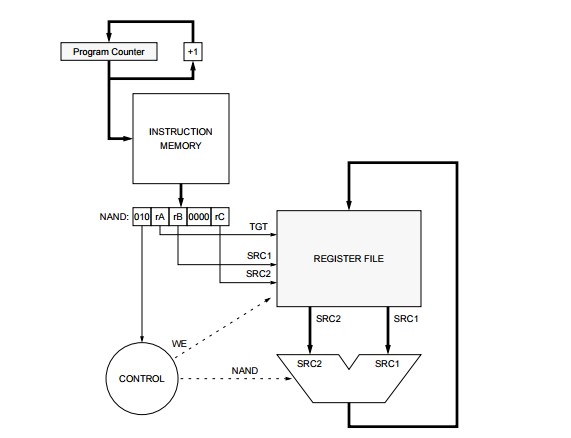
After this the output will look like this:

# KERNEL: PC = 0000000000000100  
# KERNEL: IF = 1110110010000010  
**# KERNEL: Opcode = 111  
# KERNEL: Instruction = NAND : (R[1] = 5 = 0000000000000101) ~& (R[2] = 4 = 0000000000000100)**# KERNEL: NAND 0000000000000101 ~& 0000000000000100 = 1111111111111011  
**# KERNEL: Write R3 = 65531**  
**# KERNEL: NAND R[3] = 1111111111111011**  
# KERNEL:

# KERNEL: Print Registers:  
# KERNEL: R 0 = 0  
# KERNEL: R 1 = 5  
# KERNEL: R 2 = 4  
**# KERNEL: R 3 = 65531**  
# KERNEL: R 4 = 53  
# KERNEL: R 5 = 26  
# KERNEL: R 6 = 0  
# KERNEL: R 7 = 45

After increasing the program counter the instruction is fetched and we can see the NAND opcode which is 111. So, now the value in register 1 and register 2 are meant to perform NAND operation and result is stored in register 3

The flow of control data path is illustrated by the figure below:



R[2]=4

R[3]=65531

R[1]=5

Write back o/p to reg

Control signal to perform NAND

So the output is stored in R[3] after performing the NAND operation on R[1] & R[2] values.

The NAND signal is issued from the control to ALU and then the operation is performed and stored in Register Fi

**LUI – Shift-6 bits for left**

In this instruction the data is shifted to 6-bits left. We use << shift left operator to do that.

The immediate value is shifted left 6 bits and stored in the register.

The machine format for LUI shift right is:

**alu\_result = alu\_operand1 << 6**

The design code for shifting 6 bits is given below:

//lui shift6

if(opcode == 3'b101)

begin

$display("Instruction = LUI (Shift 6) : R[%d] = %d ", instruction[12:10], instruction[9:0]);

reg\_address\_A = instruction[12:10];

cs\_alu = 4'b0101;

trigger\_B = 1;

cs\_write\_reg = 1;

cs\_alu\_select = 1;

end

end

The ALU operation for this left shift-6 bits is:

**4'b0101**:

begin

alu\_result = alu\_operand1 << 6;

alu\_overflow = 0;

alu\_zero = (alu\_result == 0) ? 1 : 0;

$display("LUI (Shifted 6 Left) (%b = %d) = (%b = %d)", alu\_operand1, alu\_operand1, alu\_result, alu\_result);

end

From the above code the data is shifted left for 6 bits and the **OUTPUT:**

# KERNEL: PC = 0000000000000110  
**# KERNEL: IF = 1011010000000010  
# KERNEL: Opcode = 101  
# KERNEL: Instruction = LUI (Shift 6) : R[5] = 2**   
# KERNEL: LUI (Shifted 6 Left) (0000000000000010 = 2) = (0000000010000000 = 128)  
**# KERNEL: Write R5 = 128**  
# KERNEL:   
# KERNEL: Print Registers:  
# KERNEL: R 0 = 0  
# KERNEL: R 1 = 5  
# KERNEL: R 2 = 4  
# KERNEL: R 3 = 65531  
# KERNEL: R 4 = 53  
**# KERNEL: R 5 = 128**  
# KERNEL: R 6 = 0  
# KERNEL: R 7 = 45

Here we can see that in 16 bits of data the second bit is 1, whose value is 2 and then after performing shift-6bits operation, the 1 got shifted left for 6 bits and stored in 7th value which is 128.

Now we should declare **write back register**s in the design.v file.

if (cs\_write\_reg == 1) begin

registers[reg\_address\_A] = alu\_result;

$display("Write R%d = %d",reg\_address\_A, alu\_result);

if(trigger == 1) begin

$display("NAND R[%d] = %b", reg\_address\_A, registers[reg\_address\_A]);

end

$display("\nPrint Registers:");

for(i = 0; i < 8; i = i+1) begin

$display("R%d = %d",i, registers[i]);

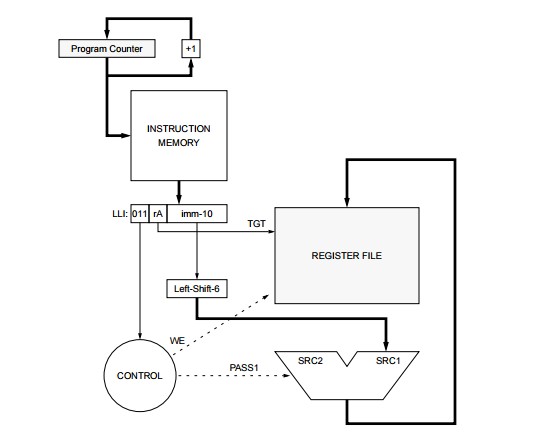
end

end

Immediatevalue 128

Left shift= 10-6=4

WE=1;



R[5]=128

**BEQ: BRANCH ON EQUAL**

Here the comparison is done on registers. If the values are matched then we should branch to the target register. The machine format is given below:

**beq regA, regB, immed**

So, what happens is when we got same registers then the program counter will be incremented by **pc+1+immediate** value

So, we can write the condition for BEQ like following:

**alu\_result = (alu\_operand2 == alu\_operand0) ? 1 : 0;**

So, here we are going to declare alu\_operand2 in the instruction and then we are going to use that here.

Now we should write a design code for this:

**//beq instruction**

if(opcode == 3'b100)

begin

registers[1] = 3;

registers[2] = 3;

$display("Instruction: BEQ : (R[%d] = %d) =/!= (R[%d] = %d)", instruction[12:10], registers[instruction[12:10]],instruction[9:7], registers[instruction[9:7]]);

reg\_address\_A = instruction[12:10];

alu\_operand2 = registers[reg\_address\_A];

cs\_alu = 4'b0100;

cs\_alu\_select = 1; //select the immediate value

end

Now we can see in the code, alu\_operand2 is going to read the registers[reg\_address\_A]. cs\_alu\_select=1; to consider an immediate value for this BEQ operation. ALU operation can be written as:

4'b0100:

//beq

begin

alu\_result = (alu\_operand2 == alu\_operand0) ? 1 : 0;

$display("Retruns: %d", alu\_result);

alu\_overflow = 0;

alu\_zero = (alu\_result == 0) ? 1: 0;

pc\_control = (alu\_result == 1) ? 3'b111 : 3'b000;

$display("pc\_result: %b", pc\_control);

end

So, here we are going to increment the program counter with an immediate value.

That can be represented as PC+1+immediate value

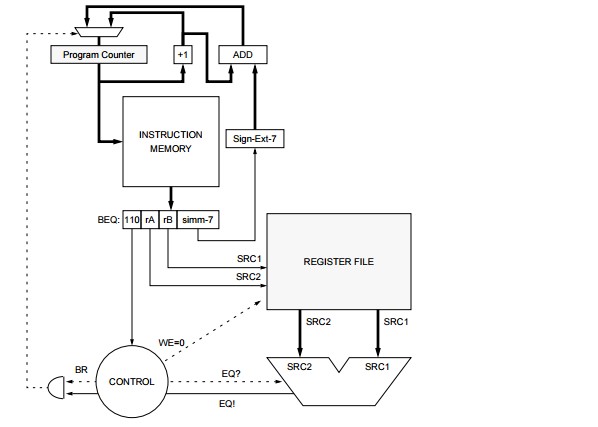
When we got the same register value then we get output=1; and then pc is incremented to the value given by above format.

**The output for this instruction:**

# KERNEL: PC = 0000000000000000  
# KERNEL: IF = 1000010100000100  
# KERNEL: Opcode = 100  
# KERNEL: Instruction: BEQ : (R[1] = 3) =/!= (R[2] = 3)  
# KERNEL: Retruns: 1  
# KERNEL: pc\_result: 0111  
# KERNEL: immediate: 0000000000000100  
# KERNEL: pc\_changed: 0000000000000101  
# KERNEL:   
# KERNEL: PC = 0000000000000101  
# KERNEL: IF = xxxxxxxxxxxxxxxx  
# KERNEL: Opcode = xxx

From the above output you can see that pc is incremented by the immediate value=4 +1 =5 cycles when we got same registers. So, whenever we get 1 in return value then PC is going to be incremented.

The flow control of data path in this instruction is given below:



OPCODE100

PC+1+5

From above figure we can see that first the opcode is send to the control and then the COMPARISON is done on the registers and 1/0 is assigned depending upon the output. Then the program counter is increased by the value of

PC+1+immediate value.

So, for this operation we are going to create a new property for ALU with alu\_operand2 and also new value b’111 which is a case for increasing the program counter clock cycles.

That is defined in the below line:

3'b111 :

begin

$display("immediate: %b", alu\_operand1);

pc = (pc + 1 + alu\_operand1); // add immediate value to the pc + 1;

$display("pc\_changed: %b", pc);

end

**JALR: JUMP-AND-LINK-THROUGH-REGISTER**

**jalr regA, regB PC <- R[regB], R[regA] <- PC + 1**

This function uses two of the register file’s three ports. A value is read from the register file and placed directly into the program counter, and the sum PC + 1 is written to a specified register in the reg.

The design code for JALR instruction can be implemented as shown below:

//jalr instruction

if(opcode == 3'b000)

begin

registers[3] = 16'b0000000000000010;

$display("Instruction = JALR = (R[%d] = PC + 1 ) and (R[%d] = %b)",

instruction[12:10], instruction[9:7], registers[instruction[9:7]]);

$display("Registers:");

for(i = 0; i < 8; i = i+1)

begin

$display("R%d = %d = %b",i, registers[i], registers[i]);

end

After this we should give instructions to ALU like

4'b0000: //jalr

begin

registers[reg\_address\_A] = pc + 1;

alu\_result = registers[reg\_address\_A];

pc\_control = 3'b101;

trigger\_C = 1;

$display("JALR R[%d] = %b = %d", reg\_address\_A, registers[reg\_address\_A], registers[reg\_address\_A]);

end

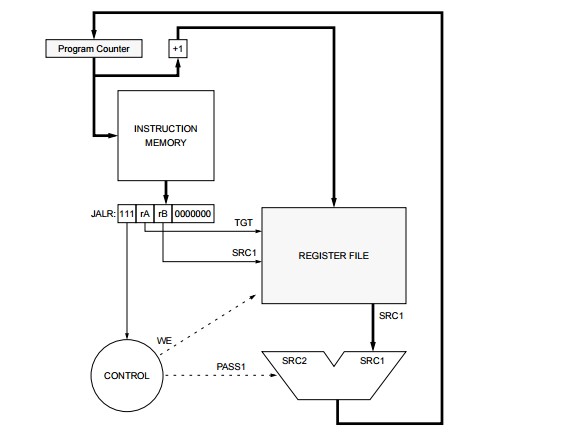
So, here the R[1] register is having pc+1 value and then when we add this to another register then the program counter value increases to next level.

The output can be seen like this

# KERNEL: PC = 0000000000000000  
**# KERNEL: IF = 0000011010000000  
# KERNEL: Opcode = 000  
# KERNEL: Instruction = JALR = (R[1] = PC + 1 ) and (R[5] = 0000000000000010)**  
# KERNEL: Registers:  
# KERNEL: R 0 = 0 = 0000000000000000  
# KERNEL: R 1 = 0 = 0000000000000000  
# KERNEL: R 2 = 0 = 0000000000000000  
# KERNEL: R 3 = 0 = 0000000000000000  
# KERNEL: R 4 = 0 = 0000000000000000  
**# KERNEL: R 5 = 2 = 0000000000000010**  
# KERNEL: R 6 = 0 = 0000000000000000  
# KERNEL: R 7 = 0 = 0000000000000000  
# KERNEL: **JALR R[1] = 0000000000000001 = 1**  
# KERNEL: **Write R1 = 1**  
# KERNEL:   
# KERNEL: Print Registers:  
# KERNEL: R 0 = 0 = 0000000000000000  
**# KERNEL: R 1 = 1 = 0000000000000001**  
# KERNEL: R 2 = 0 = 0000000000000000  
# KERNEL: R 3 = 0 = 0000000000000000  
# KERNEL: R 4 = 0 = 0000000000000000  
**# KERNEL: R 5 = 2 = 0000000000000010**  
# KERNEL: R 6 = 0 = 0000000000000000  
# KERNEL: R 7 = 0 = 0000000000000000  
# KERNEL: pc\_changed: 0000000000000010  
# KERNEL:   
# **KERNEL: PC = 0000000000000010**  
# KERNEL: IF = xxxxxxxxxxxxxxxx  
# KERNEL: Opcode = xxx  
**# KERNEL: JALR R[1] = 0000000000000011 = 3**

After this execution we can see that program counter is changed to next level.

The flow of control for the JALR instruction can be shown in below figure:



16BITS INPUT

OPCODE 000

R[5]=TRGT REGISTER

PC+1

**EXTRA OBSERVATIONS**:

In order to perform all the operations at the same time we increased the instruction memory given in the default design file from **[3:0] to [8:0].** Now the operations can be fetched and performed at the same time.

Reg [15:0] instruction\_memory [8:0]; //NUMBER OF INSTRUCTIONS

1.In the 16-bit registers the instruction memory is increased to [8:0]. Which represents the number of Instructions.

2.In testbench.v file we are increasing the clock cycles from 5 to 8 because when we are simulating all the instructions we are unable to see the whole output at single instance. So, we are increasing clock cycles by which we can get all of them in single simulation.

3.To get immediate value for LUI instruction we are going to increase them to 10 immediate bits. So, we added trigger\_B for them

immediate = { {6{1'b0}}, instruction[9:0]};

4.For NAND instruction we added trigger to display statement for getting binary values

**$display("NAND %b ~& %b = %b", alu\_operand0, alu\_operand1, alu\_result);**

**Trigger\_A = 1;**

**end**

5. For BEQ operation we need to add the alu\_operand2[15:0] in alu\_properties

6. For JALR operation we need another trigger\_C to increment its pc value and after execution the pc is incremented by pc+1 value as shown in the output of JALR.

**DESIGN FILE:**

module CPU( clk, rst );

input clk;

input rst;

reg [15:0] pc;

reg [15:0] instruction;

reg [15:0] instruction\_memory [8:0]; //NUMBER OF INSTRUCTIONS

**// Data Memory**

reg [15:0] data\_memory [10:0]; // Array[10] of 16bits in each slot

reg [15:0] data\_address;

**// Register Data**

reg [2:0] reg\_address\_A; // reg to write reg\_data\_write

reg [15:0] reg\_data\_A; // data to write to register[reg\_address\_write]

reg [2:0] reg\_address\_B; // address of register B

reg [15:0] reg\_data\_B; // data of register B

reg [2:0] reg\_address\_C; // address of register C

reg [15:0] reg\_data\_C; // data of register C

**// Control Signals**

reg [0:0] cs\_write\_reg; // Controls If reg\_data\_write is written to reg\_address\_write

reg [0:0] trigger\_A; //for nand operation use

reg [0:0] trigger\_B; // for lui operation

reg [0:0] trigger\_C; // for jalr instruction

reg [3:0] cs\_alu; // Controls Which ALU Operation Will Be Commited

reg cs\_alu\_select; // Controls If Operand1 = Instruction[9:0] OR reg\_data\_c

reg cs\_read\_data\_memory;

reg cs\_write\_data\_memory;

reg [3:0] pc\_control;

**// ALU Properties**

reg [15:0] alu\_operand0;

reg [15:0] alu\_operand1;

reg[15:0] alu\_operand2;

reg [15:0] alu\_result;

reg alu\_overflow;

reg alu\_zero;

**// Data**

reg [15:0] registers [7:0];

reg [2:0] opcode;

reg [15:0] immediate;

integer i;

initial begin

**// Load Instructions**

$readmemb("p2inst.mips",instruction\_memory);

**// Reset Registers**

$display("\nPrint Registers:");

for(i = 0; i < 8; i = i+1)

begin

registers[i] = 0;

$display("R%d = %d",i, registers[i]);

end

end

always @(posedge clk or posedge rst)

begin

if (rst) begin

pc = 16'd0; // Reset PC to address 0x00

end

end

always @(posedge clk) begin

**// Instruction Fetch**

$display("\nPC = %b", pc);

instruction = instruction\_memory[pc];

**// Print Instruction**

$display("IF = %b", instruction);

**// Reset Control Signals**

cs\_write\_reg = 0;

cs\_alu = 4'b0000;

cs\_alu\_select = 0;

cs\_write\_data\_memory = 0;

cs\_read\_data\_memory = 0;

**// Instruction Decode**

assign opcode = instruction[15:13]; // Set the opcode

$display("Opcode = %b" , opcode);

/\* ID START:

You Will Need to implement support for the instructions assigned in the assignment.

ADDI has been implemented as a sample for you.

Note: You may need to add more control signals to support some instructions

\*/

**//add instruction**

if(opcode == 3'b110) begin

$display("Instruction = ADD : %b = %b + %b --> (R%d = R%d + R%d) " ,

instruction[12:10], instruction[9:7], instruction[2:0], instruction[12:10], instruction[9:7], instruction[2:0]);

reg\_address\_A = instruction[12:10];

cs\_write\_reg = 1;

cs\_alu = 4'b0110;

registers[1] = 5;

registers[2] = 3;

cs\_alu\_select = 0; //selecting source 2 register

end

**//addi instruction**

if(opcode == 3'b001) begin

$display("Instruction = ADDI : %b = %d", instruction[6:0], instruction[6:0]);

reg\_address\_A = instruction[12:10];

cs\_write\_reg = 1;

cs\_alu = 4'b0001;

cs\_alu\_select = 1; //selecting immediate value

end

**//lw instruction**

if(opcode == 3'b010) begin

$display("Instruction = LW : %b = %d", instruction[6:0], instruction[6:0]);

reg\_address\_A = instruction[12:10];

cs\_read\_data\_memory = 1;

data\_memory[2] = 21;

data\_memory[4] = 37;

registers[1] = 5;

cs\_alu = 4'b0010;

cs\_alu\_select = 1; //selecting immediate value

end

**//sw instruction**

if(opcode == 3'b011)

begin

$display("Instruction = SW : %b = %d", instruction[6:0], instruction[6:0]);

reg\_address\_A = instruction[12:10];

cs\_write\_data\_memory = 1;

cs\_alu = 4'b0011;

registers[1] = 22;

registers[7] = 45;

cs\_alu\_select = 1; //selecting immediate value

end

**//nand instruction**

if(opcode == 3'b111)

begin

reg\_address\_A = instruction[12:10];

registers[1] = 5;

registers[2] = 4;

$display("Instruction = NAND : (R[%d] = %d = %b) ~& (R[%d] = %d = %b)" ,

instruction[9:7], registers[instruction[9:7]], registers[instruction[9:7]], instruction[2:0],

registers[instruction[2:0]], registers[instruction[2:0]]);

cs\_alu = 4'b0111;

cs\_write\_reg = 1;

cs\_alu\_select = 0; //taking source register 2

end

**//lui shift6**

if(opcode == 3'b101)

begin

$display("Instruction = LUI (Shift 6) : R[%d] = %d ", instruction[12:10], instruction[9:0]);

reg\_address\_A = instruction[12:10];

cs\_alu = 4'b0101;

trigger\_lui = 1;

cs\_write\_reg = 1;

cs\_alu\_select = 1;

end

end

**//BEQ INSTRUCTION**

if(opcode == 3'b100)

begin

registers[1] = 3;

registers[2] = 3;

$display("Instruction: BEQ : (R[%d] = %d) =/!= (R[%d] = %d)", instruction[12:10], registers[instruction[12:10]],instruction[9:7], registers[instruction[9:7]]);

reg\_address\_A = instruction[12:10];

alu\_operand2 = registers[reg\_address\_A];

cs\_alu = 4'b0100;

cs\_alu\_select = 1; //select the immediate value

end

**//jalr instruction**

if(opcode == 3'b000)

begin

registers[5] = 16'b0000000000000010;

$display("Instruction = JALR = (R[%d] = PC + 1 ) and (R[%d] = %b)",

instruction[12:10], instruction[9:7], registers[instruction[9:7]]);

$display("Registers:");

for(i = 0; i < 8; i = i+1)

begin

$display("R%d = %d = %b",i, registers[i], registers[i]);

end

**// Read Registers**

reg\_address\_B = instruction[9:7];

reg\_address\_C = instruction[2:0];

reg\_data\_B = registers[reg\_address\_B];

reg\_data\_C = registers[reg\_address\_C];

**// ALU Operation**

immediate = { {9{1'b0}}, instruction[6:0]}; // This just extends the immediate value to 16 bits

if(trigger\_lui == 1)

begin

immediate = { {6{1'b0}}, instruction[9:0]};

trigger\_lui = 0;

end

alu\_operand0 = reg\_data\_B;

alu\_operand1 = (cs\_alu\_select == 0) ? reg\_data\_C : immediate;

**// ALU START**

case (cs\_alu)

4'b0110: // add

begin

alu\_result = alu\_operand0 + alu\_operand1;

alu\_overflow = 0;

alu\_zero = (alu\_result == 0) ? 1 : 0;

$display("Added %d + %d = %d", alu\_operand0, alu\_operand1, alu\_result);

end

4'b0001: // Signed add

begin

alu\_result = alu\_operand0 + alu\_operand1;

if ((alu\_operand0 >= 0 && alu\_operand1 >= 0 && alu\_result < 0) ||

(alu\_operand0 < 0 && alu\_operand1 < 0 && alu\_result >= 0)) begin

alu\_overflow = 1;

end else begin

alu\_overflow = 0;

end

alu\_zero = (alu\_result == 0) ? 1 : 0;

$display("Added %d + %d = %d",alu\_operand0, alu\_operand1, alu\_result);

End

**4'b0010**:

begin

data\_address = immediate; //Data Access

alu\_result = alu\_operand0 + data\_memory[data\_address];

alu\_overflow = 0;

alu\_zero = (alu\_result == 0) ? 1 : 0;

$display("LW Added: R[%d] =(R[%d] = %d) + (D[%d] = %d) = %d",reg\_address\_A,reg\_address\_B, alu\_operand0, data\_address, data\_memory[data\_address], alu\_result);

end

**4'b0011**:

begin

data\_address = immediate; //Data Access

alu\_result = registers[reg\_address\_A] + alu\_operand0;

alu\_overflow = 0;

alu\_zero = (alu\_result == 0) ? 1 : 0;

$display("SW Added: (R[%d] = %d) + (R[%d] = %d) = %d", reg\_address\_A, registers[reg\_address\_A], reg\_address\_B, alu\_operand0, alu\_result);

end

**4'b0111**:

begin

alu\_result = ~(alu\_operand0 & alu\_operand1);

alu\_overflow = 0;

alu\_zero = (alu\_result == 0) ? 1 : 0;

$display("NAND %b ~& %b = %b", alu\_operand0, alu\_operand1, alu\_result);

trigger = 1;

end

end

**4'b0101**:

begin

alu\_result = alu\_operand1 << 6;

alu\_overflow = 0;

alu\_zero = (alu\_result == 0) ? 1 : 0;

$display("LUI (Shifted 6 Left) (%b = %d) = (%b = %d)", alu\_operand1, alu\_operand1, alu\_result, alu\_result);

end

default:

begin

alu\_zero = 0;

alu\_overflow = 0;

end

endcase

**4'b0100: //beq**

begin

alu\_result = (alu\_operand2 == alu\_operand0) ? 1 : 0;

$display("Retruns: %d", alu\_result);

alu\_overflow = 0;

alu\_zero = (alu\_result == 0) ? 1: 0;

pc\_control = (alu\_result == 1) ? 3'b111 : 3'b000;

$display("pc\_result: %b", pc\_control);

end

4'b0000: //jalr

begin

registers[reg\_address\_A] = pc + 1;

alu\_result = registers[reg\_address\_A];

pc\_control = 3'b101;

trigger\_C = 1;

$display("JALR R[%d] = %b = %d", reg\_address\_A, registers[reg\_address\_A], registers[reg\_address\_A]);

end

**// ALU END**

**// Write Back To Reg**

registers[reg\_address\_A] = alu\_result;

$display("Write R%d = %d",reg\_address\_A, alu\_result);

if(trigger\_A == 1) begin

$display("NAND R[%d] = %b", reg\_address\_A, registers[reg\_address\_A]);

trigger\_A = 0;

end

if(trigger\_C == 1)

begin

$display("\nPrint Registers:");

for(i = 0; i < 8; i = i+1) begin

$display("R%d = %d = %b",i, registers[i], registers[i]);

trigger\_C = 0;

end

end

else

begin

$display("\nPrint Registers:");

for(i = 0; i < 8; i = i+1) begin

$display("R%d = %d",i, registers[i]);

end

end

end

**//For SW Instruction: Store from Reg to Memory**

if (cs\_write\_data\_memory == 1) begin

data\_memory[data\_address] = alu\_result;

$display("Data\_Memory Changed: D[%d] = %d", data\_address, data\_memory[data\_address])

$display("\nPrint Registers:");

for(i = 0; i < 8; i = i+1) begin

$display("R%d = %d",i, registers[i]);

end

$display("\nPrint Data Memory:");

for(i = 0; i < 8; i = i+1) begin

$display("D[%d] = %d",i, data\_memory[i]);

end

end

**//For LW Instruction: Load From Memory to Reg**

if(cs\_read\_data\_memory == 1) begin

registers[reg\_address\_A] = alu\_result;

$display("Register Changed: R%d = %d", reg\_address\_A, registers[reg\_address\_A]);

$display("\nPrint Data Memory:");

for(i = 0; i < 8; i = i+1) begin

$display("D[%d] = %d",i, data\_memory[i]);

end

$display("\nPrint Registers:");

for(i = 0; i < 8; i = i+1) begin

$display("R%d = %d",i, registers[i]);

end

end

**// Increment PC According to pc\_control**

case (pc\_control)

3'b000:

begin

pc = pc + 1;

$display("pc\_changed: %b", pc);

end

3'b111:

begin

$display("immediate: %b", alu\_operand1);

pc = (pc + 1 + alu\_operand1); // add immediate value to the pc + 1;

$display("pc\_changed: %b", pc);

end

3'b101:

begin

pc = registers[reg\_address\_B];

$display("pc\_changed: %b", pc);

pc\_control = 3'b001; //reset

end

default:

begin

pc = pc + 1;

end

endcase

end

endmodule

**TESTBENCH**:

module tb\_CPU;

reg clk;

reg rst;

CPU U0

(

.clk(clk),

.rst(rst)

);

always

#5 clk = ~clk;

initial

begin

// time = 0

clk = 1'b0;

// Reset CPU

rst = 1'b1;

// run 1st iteration to reset cpu, and load first instruction

@(posedge clk);

// set Rest to 0

rst = 1'b0;

// Run through 5 CPU cycles

repeat(8)

@(posedge clk);

$finish();

end

endmodule

**WAVEFORM:**

