

**North South University**

# Department of Electrical & Computer Engineering

**CSE332**

**Computer Organization and Architecture**

**Project Report**

**Submitted by:**

Name: Arifur Rahaman

ID: 2014198642

**Submitted to:**

Tanjila Farah (TnF)

**Submitted date:**

05 january 2022

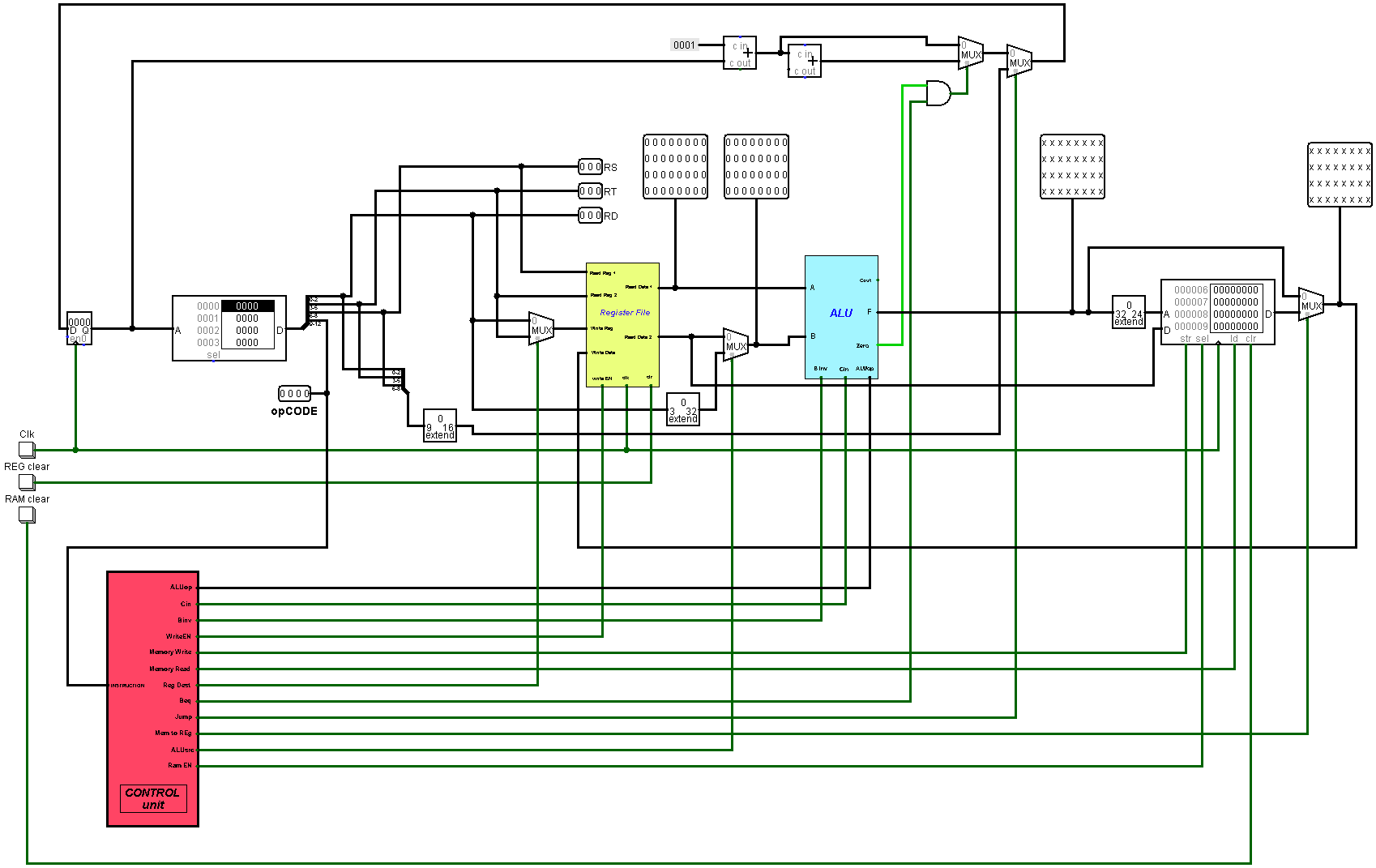
**Introduction:** In this project, I have to create a 16-bit CPU that can perform simple arithmetic, logical, conditional branch, unconditional jump, and data transfer operations. There are currently ten operations. These operations are carried out using 8 registers.

**Components:**

* ROM
* PC
* 16-bit Register
* 16-bit ALU
* Control Unit
* RAM
* Bit Extender
* MUXs
* Adder
* Logic Gates

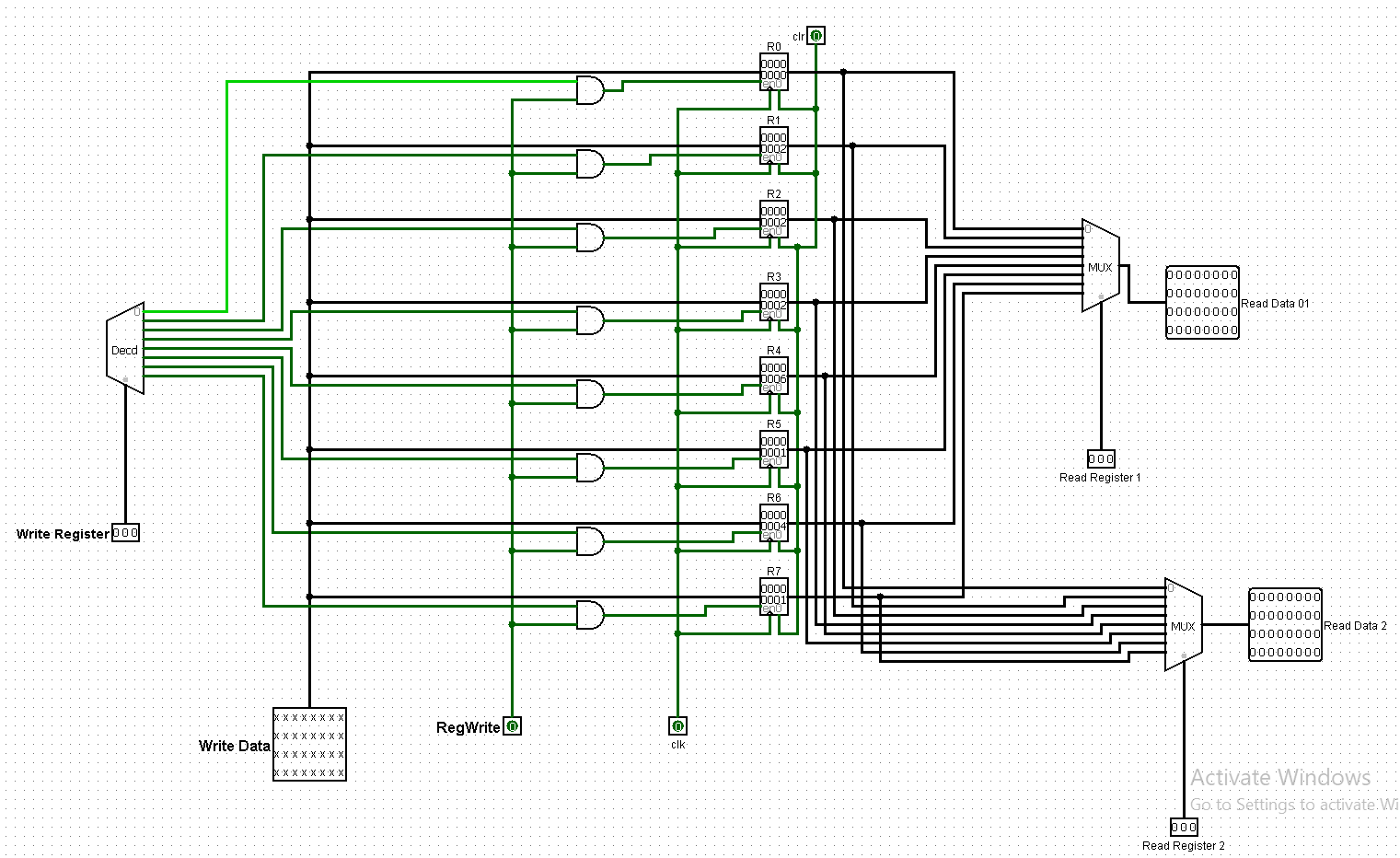
**Circuits:**

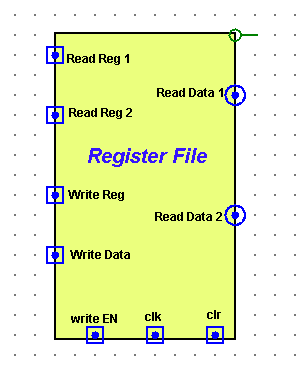
**Datapath:** This is the complete Datapath of the project. Here for a program counter, I used a rom and used the register as a buffer for program counter input. For storing data I used a ram and I used sub-circuits of the Register File, ALU and Control Unit to complete the operations. And also connected all the wires with the required pin to that it can perform the operation properly.



*Fig: Complete CPU Datapath*

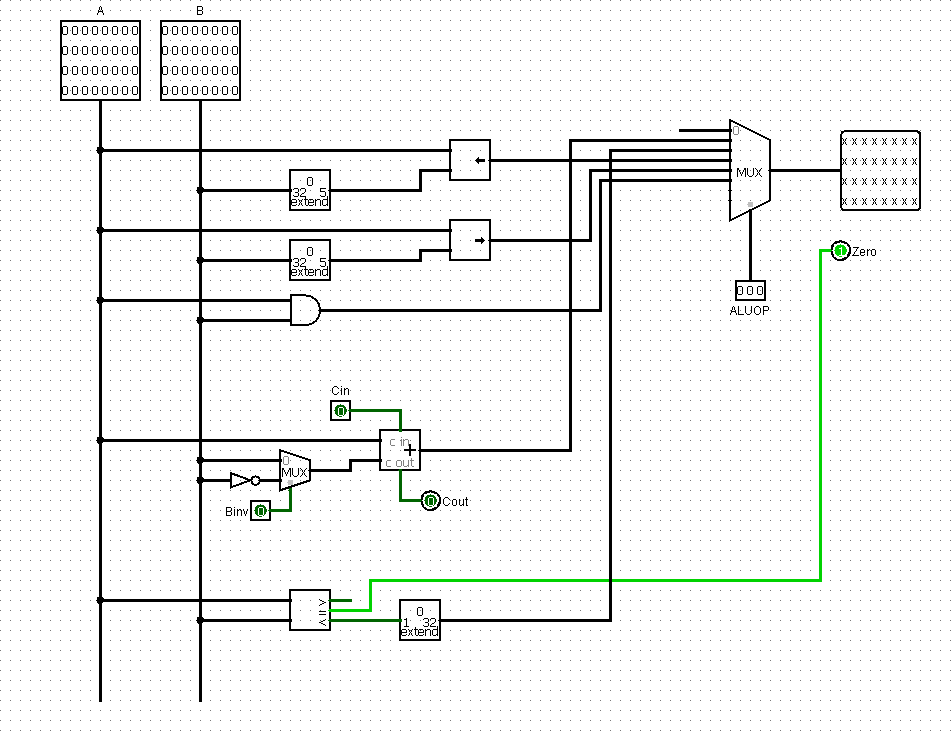
**Register file:** A register file is a small collection of high-speed storage cells located within the CPU. There are two read data pins, one write-data pin, and two register number pins on the register file. Here I used 16 registers. Here the “Write Data” input will be written on the register and the select pin named “Write Register” will determine the destination. The [instruction set architecture](https://en.wikipedia.org/wiki/Instruction_set_architecture) of a CPU will almost always define a set of registers which are used to stage data between memory and the functional units on the register. i also added a write enable button and connected with and gates so that I can control when data can be written in the register. Then there are “Read Register 1” and “Read Register 2” for reading data of the registers to pass to ALU. It receaves data and pass it to the ALU according to the instruction.

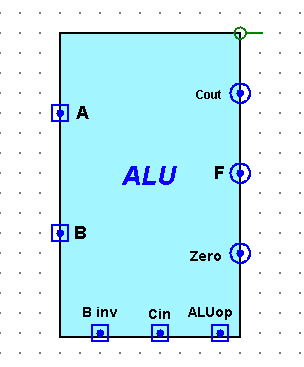




*Fig: Register file*

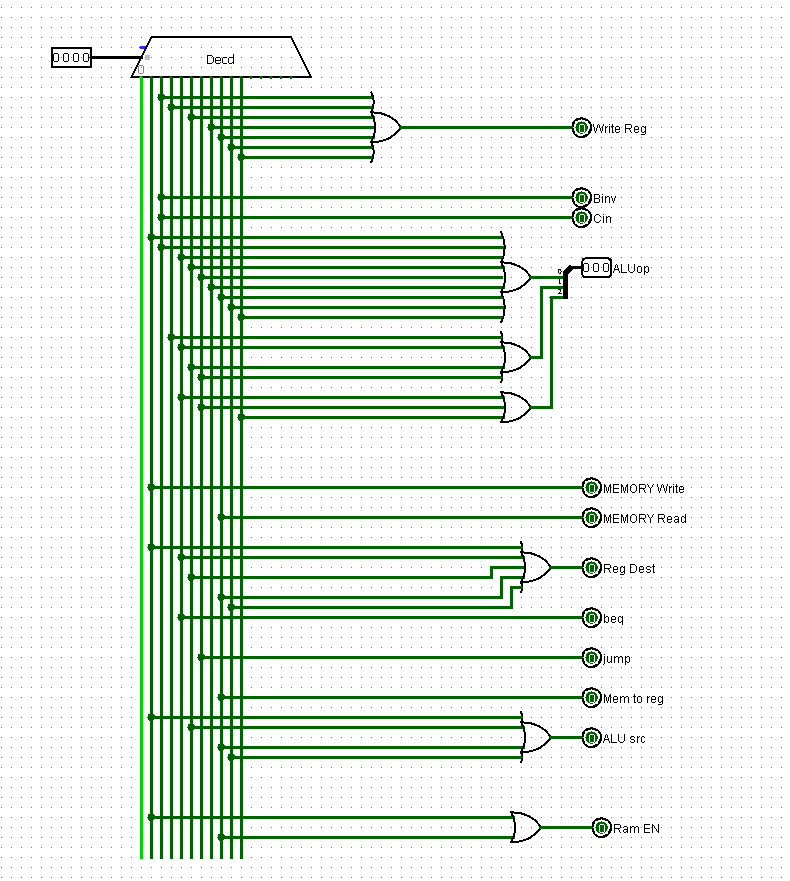
**ALU:** An arithmetic-logic unit is the part of a [central processing unit](https://whatis.techtarget.com/definition/processor) that carries out arithmetic and logic operations on the [operands](https://whatis.techtarget.com/definition/operand) in computer [instruction](https://whatis.techtarget.com/definition/instruction).Input A and B receives data from register and passes through the output F. As per the operations given the me I needed to use Shifter for shifting leftward, AND gate for AND operation, Comparator to check equality or smaller for slt and branch on equal operation then and lastly a modified adder which used to perform both addition and subtraction and also lw,sw ,addi, operations . the ALU has direct input and output access because it has direct connection with the main memory and with the input output line.

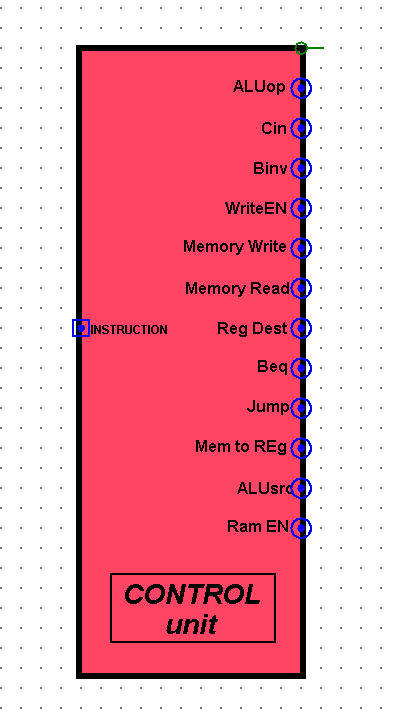




*Fig: ALU*

**Control Unit:** the control unit is the central processing unit of the architecture . The main function of the control unit is to fetch and execute instructions from the memory. It receives the input instruction from the user and converts it into [control signals](https://www.elprocus.com/dynamic-road-traffic-signal-control/), which goes to the main access memory. So, I need a control unit to send the specific signal for a specific operation that I have instructed to do :

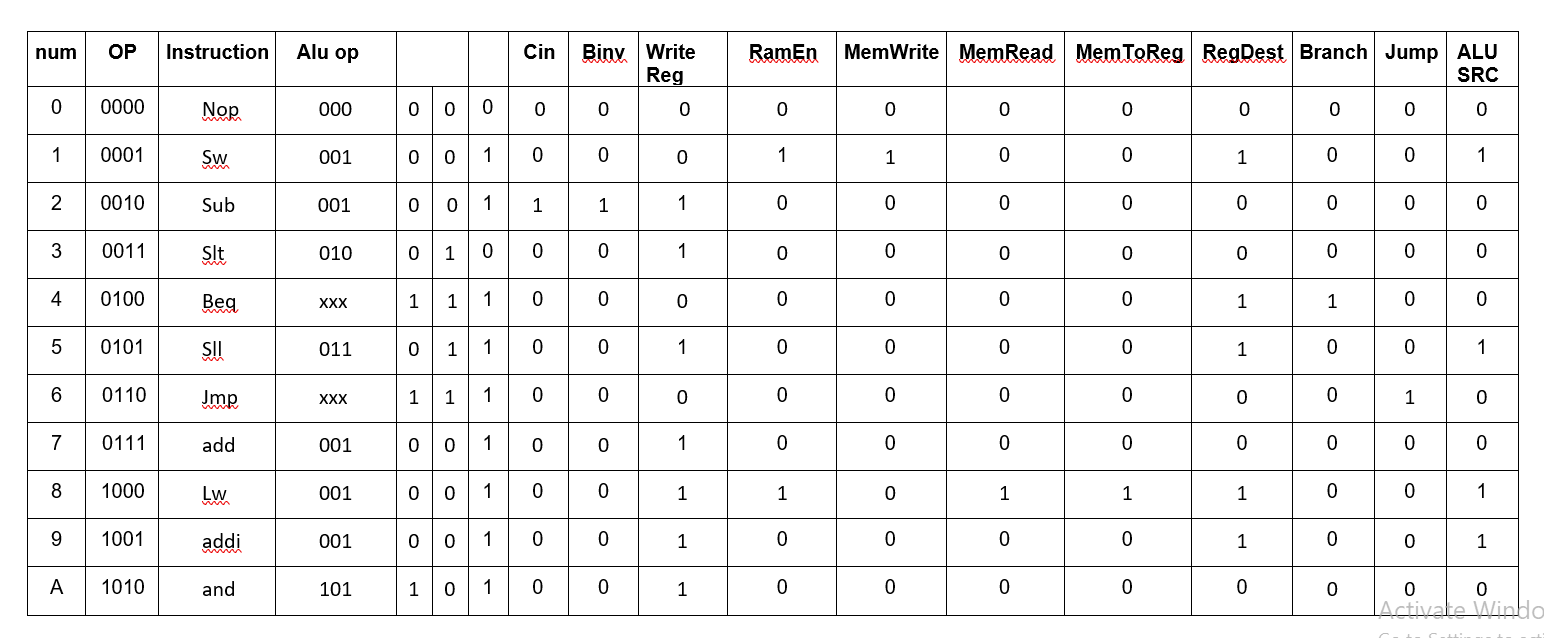




*FIG : Control Unit*

* **ALUop:** ALUop signal control the ALU operation. According to signal ALU execute the operation and give the output value.
* **Cin:** Control unit gives Cin signal only sub operation.
* **Binv:** Control unit gives this signal when the sub operation executed.
* **WriteEn:** This signal works for write into the register.
* **RAMEn:** This signal works when lw and sw instruction is executed.
* **MemWrite:** This signal use when store value into the memory.
* **MemRead:** Control unit provides this signal when need to read from memory.
* **MemToReg:** This signal provides when need to load a value.
* **RegDest:** RegDest select the register.
* **ALU SRC:** This works for ALU source register.
* **Branch:** This signal provides control unit only for beq.
* **Jump:** This signal provides control unit only for jmp.

**Control Unit Table:** According to the operations given above I prepared this control unit table and created a circuit following the table. I have completed it as like my given sequence. As per given instructions I needed to use a 4:16 decoder because there is total 11 operation given. I have generated the opcode according to the sequence . According to the OpCodes given it will generate specific signal



*Fig: Control Unit table*

**Assembler Documentation:**

**Introduction:** Our task was to design an assembler which will convert the assembly code to machine language.

**Objective:** Our main goal was to generate a machine code from a file containing assembly language. The assembler reads a program written in an assembly language, then translate it into binary code and generates output file containing machine code.

**How to use:** In the input file the user has to give some instructions to convert into machine codes. The system will convert valid MIPS instructions into machine language and generate those codes into output file.

**Input File:**The input file named “inputs”. User will write down the MIPS code in this file.

**Output File:** Here the output filewill showm you the HEX value

## List of Tables

### Register List

We have selected registers from r0-r7,total 8 gegister for instruction purpose. We assigned 3 bits for each of the register as we know in the instruction field in our ISA containing the register rs, rt and rd contains 3 bits each.

|  |  |  |
| --- | --- | --- |
| **Conventional Name** | **Register Number** | **Binary Value** |
| r0 | 0 | 000 |
| r1 | 1 | 0001 |
| r2 | 2 | 010 |
| r3 | 3 | 011 |
| r4 | 4 | 100 |
| r5 | 5 | 101 |
| r6 | 6 | 110 |
| r7 | 7 | 111 |

***Op-Code List:*** We have selected following op codes and assigned op-code binary values (4 bits) for each of the op codes.

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **OpCode** |
| nop |  | 0000 |
| sw | I | 0001 |
| sub | R | 0010 |
| Slt | R | 0011 |
| Beq | I | 0100 |
| Sll | R | 0101 |
| Jmp | J | 0110 |
| Add | R | 0111 |
| Lw | I | 1000 |
| Addi | I | 1001 |
| and | R | 1010 |

**Instruction Description nop:** No operation.

**sll:** It shifts bits to the left and fill the empty bits with zeros. The shift amount is depended on the register value.

* **Operation:** r3 = r1 << r2
* **Syntax:** sll r1 r2 r3

**and:** It AND’s two register values and stores the result in destination register. Basically, it sets some bits to 0.

* **Operation:** r3 = r1 & r2
* **Syntax:** and r1 r2 r3

**lw:** It loads required value from the memory and write it back into the register.

* **Operation:** r1 = Mem[r0+immediate]
* **Syntax:** lw r0 immediate

**slt:** If r1 is less than r2, r3 is set to one. It gets zero otherwise.

* **Operation:** if (r1 < r2)

r3 = 1

else

r3 = 0

* **Syntax:** slt r1 r2 r3

**add:** It adds two registers and stores the result in destination register.

* **Operation:** r1 = r1 + r2
* **Syntax:** add r1 r2 r1

**addi:** It adds a value from register with an integer value and stores the result in destination register.

* **Operation:** r1 = r2 + immediate
* **Syntax:** addi r2 r1 immediate

**sub:** It subtracts two registers and stores the result in destination register.

* **Operation:** r1 = r1 – r2
* **Syntax:** sub r1 r2 r1

**sw:** It stores specific value from register to memory.

* **Operation:** Mem[r0+immediate] = r1
* **Syntax:** sw r0 r1 immediate

**jmp:** Jumps to the calculated address.

* **Operation:** jum to target address
* **Syntax:** jmp target

**beq:** It checks whether the values of two register s are same or not. If it’s same it performs the operation located in the address at offset value.

* **Operation:** if (r1==r2)

jump to immediate

else

goto next line

* **Syntax:** beq r1 r2 immediate

**Limitation:**

User can’t put anything without space in between the instruction line in the input file.If use put any thing in there , the input line will be shown as invalid input.

**Instruction Set Architecture Design** (ISA)

**Objectives:** My objective was to design a 13 Bit ISA which can solve a particular problems like simple arithmetic & logic operations, branching and loops.

**Operands:** My goal is to use accumulator base ISA. For this reason, I am going to take two operands. I will address these two operands as **d**, **t** and **s**.

**Types of Operands:** I need register operands to implement arithmetic instructions, and memory operands to implement data transfer instructions from memory to register. As a result, I'll need two sorts of operands.

* Register based.
* Memory based.

**Operations:** Iwill allocate 4 bits opcode, so the executable instructions number will be 24 or 16.

**Types of operations:** In my design there will be five different types of operation. The operations are:

* Arithmetic
* Logical
* Data Transfer
* Conditional Branch
* Unconditional Jump

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Category** | **Operation** | **Name** | **Type** | **OpCode** | **Syntax** | **Comments** |
|  | No operation | nop |  | 0000 | nop |  |
| Data transfer | Store word | sw | I | 0001 | sw r0 r1 2 | Mem[r0+2] = r1 |
| Arithmetic | Subtraction | sub | R | 0010 | sub r1 r2 r3 | r3 = r1-r2 |
| Conditional | Compare less than | slt | R | 0011 | slt r1 r2 r3 | If(r1<r2) then r3 = 1 else r3 = 0 |
| Conditional | Check equality | beq | I | 0100 | beq r1 r2 4 | If(r1==r2) then go to line 4 |
| Logical | Shift left | sll | R | 0101 | sll r1 r2 r3 | r3 = r1<<r2 |
| Unconditional | Jump | jmp | J | 0110 | jmp 6 | Go to line 6 |
| Arithmetic | Add two numbers | add | R | 0111 | add r1 r2 r3 | r3 = r1 + r2 |
| Data transfer | Load word | lw | I | 1000 | lw r0 r1 2 | r1 = Mem[r0+2] |
| Arithmetic | Add number with an immediate | addi | I | 1001 | addi r1 r2 5 | r2 = r1+5 |
| Logical | Bit-by-bit and | and | R | 1010 | and r1 r2 r3 | R3 = r1 & r2 |

**Formats:**

I use three types of formats for my ISA. They are:

* Register Type – R type
* Immediate Type – I type
* Jump Type - J Type

**R Type ISA Format**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **OpCode** |  |  | **rs** | **rt** |  | **rd** |
| 4 bits |  |  | 3 bits | 3 bits |  | 3 bits |
|  |  | | **I Type IS** | **A Format** |  | |
| **OpCode** |  |  | **rs** | **rt** |  | **Immediate** |
| 4 bits |  |  | 3 bits | 3 bits |  | 3 bits |
|  |  | | **J Type IS** | **A Format** |  | |
|  | **OpCode** | |  |  | **Target Address** | |
|  | 4 bits | |  |  | 9 bits | |

**List of Register:**

As we have allocated four bits register so the number of registers will be 24 = 16.

|  |  |  |  |
| --- | --- | --- | --- |
| **Register Number** | **Conventional Name** | **Usage** | **Binary Value** |
| 0 | R0 | General Purpose | 0000 |
| 1 | R1 | General Purpose | 0001 |
| 2 | R2 | General Purpose | 0010 |
| 3 | R3 | General Purpose | 0011 |
| 4 | R4 | General Purpose | 0100 |
| 5 | R5 | General Purpose | 0101 |
| 6 | R6 | General Purpose | 0110 |
| 7 | R7 | General Purpose | 0111 |

**Translating some HLL codes using my designed 13-bit ISA**

|  |  |
| --- | --- |
| **1.** a = a + b | # r1 = a, r2 = q |
| add r1 r2 r1 | # r1 gets r1 + r2 |
| **2.** a = a – b | # r1 = a, r2 = b |
| sub r1 r2 r1 | # r1 gets r1 – r2 |
| **3.** a = a and b | # r1 = a, r2 = b |
| and r1 r2 r1 | # r1 gets r1 and r2 |
| **4**. if(a<b) | # r1 = a, r2 = b |
| c = 1 | # r3 = c |
| slt r1 r2 r3 | # r3 gets 1 if(r1<r2) else r3 gets 0 |

**5**.C = A[i] #r1 = I, r2=A , r3=C

sll r1, 2 # rp = r1<< 2

add r2 rp #r2 = r2 + rp

lsp r2 # rp = r2

lw r3 0 # getting the location value of A[i] from rp #and store them in s3.

**6**. if(i ==4)

i = i+2

else

i = i -2 # r1 = i

sub rp, rp # rp = 0

addi rp, 4 # rp = 4

beq r1 L #comparing r1 = i with rp = 4. If equal then go to L

addi r1, -2 # i = i – 2

jmp exit # Jump to exit

L: addi r1, 2 # i = i + 2

**7.** for(int j = 3; j < a; j++){

b = b\*8;

} # r1 = j, r2 = a, r3 = b

sub r1, r1 # r1 = 0

addi r1, 3 # initializing r1 = 3

sub rp, rp # rp = 0

add rp, $s2 # rp = r2

L2: beq r1 L1 #if r1 = sp then go to L1, otherwise continue loop.

sll r3 3 # rp = r3 << 3

add r3 $sp # r3 = r3 \* 8

addi r1 1 # increment r1 by 1 / i++

jmp L2 # go to the loop

**Limitations:** I have divided my 13 bits in 4 divisions. That’s why we couldn’t reserve space for shift amount. Thus, we used the immediate 3bit space in I type format for shifting purpose but it doesn’t allow me to take the immediate value which need more then three bit binary value to represent itself at a time.

**Discussion**

The primary goal of this project was to create a 13-bit CPU capable of simple arithmetic, logical, branching, and data transfer operations. First and foremost, I created an ISA that specifies the operations, op-codes, their syntaxes, instruction formats, and register list. I built the main Datapath in accordance with the ISA. There are two types of instructions. One is R-type, while the other is I-type. Because the CPU has 13 bits, I assigned 3 bits to each register. This CPU has

a total of 8 registers. Then, in accordance with the ISA, I built the ALU with the necessary arithmetic operations. To properly build the 32-bit ALU, I used 32-bit inputs and outputs, logic gates, adders, and MUXs. I separated the output's 32 bits and connected them to a NOR gate for 'zero detection.' Op-Codes are used in ALU. As a result, ALU will perform various operations based on the Op-Codes. Following that, I created a 32-bit register file. This has 8 registers, each of which is linked to a decoder. Different registers will be activated based on the decoder's 3-bit selection pin. Two registers can be passed to the output using two different MUXs. In addition, we can write data to a specific register. Then I began constructing the CPU's main Datapath. I needed ROM, a 32-bit register, a 32-bit ALU, RAM, a bit extender, MUXs, an adder, and logic gates to accomplish this. The ROM is used to serially organize instructions. To successfully complete all of the instructions, a register and an adder are connected to the ROM. The 13-bit instructions are then split from the ROM into four 4-bit binaries, which are then passed to the 32-bit register file. The register outputs are then connected to the ALU, and finally, a RAM is used to store values. The RAM is also linked to the register file, allowing it to store values in registers. We now have four sections in our instructions: Op-Code, RS, RT, and RD/IM. We've added more MUXs and bit extenders to ensure that all operations run smoothly. However, at the moment, we must manually turn each of them ON/OFF based on the Op-Code, which is inefficient and not 100% optimized. It can be optimized more and mor but for now it has some limitation for the instruction and also for other reasons. Finally, I built the control unit that will turn the necessary signals in the Datapath ON/OFF based on the Op-Code. In order to construct the control unit, I created a table that lists all of the operations and input signals required in the Datapath. Then, for each control signal, I used an OR gate to connect all of the operations that will use that signal. As a

result, whenever an Op-Code is entered, the Op-Code is sent to the control unit, and all of the necessary control signals for that operation are turned on. We eliminate the need to manually turn different input signals ON/OFF based on the Op-Code by using this control unit. Finally, I connected all of the circuit's input signals to the appropriate control unit signal. I tested the Datapath using multiple instructions based on the ISA. And the datapath is working properly and I can run the Instruction as required.

*…..…….END…………*