North South University

Department of Electrical & Computer Engineering

CSE332

Computer Organization and Architecture

Project Report

Submitted by:

Name: Md. Mashruf Ehsan

ID: 1831253642

Submitted to:

Tanjila Farah (TnF)

Introduction: In this project, I created a 16-bit CPU that can perform simple arithmetic, logical, branching, and data transfer operations. There are currently ten operations. These operations are carried out using 16 registers.

Components:

- ROM
- 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.

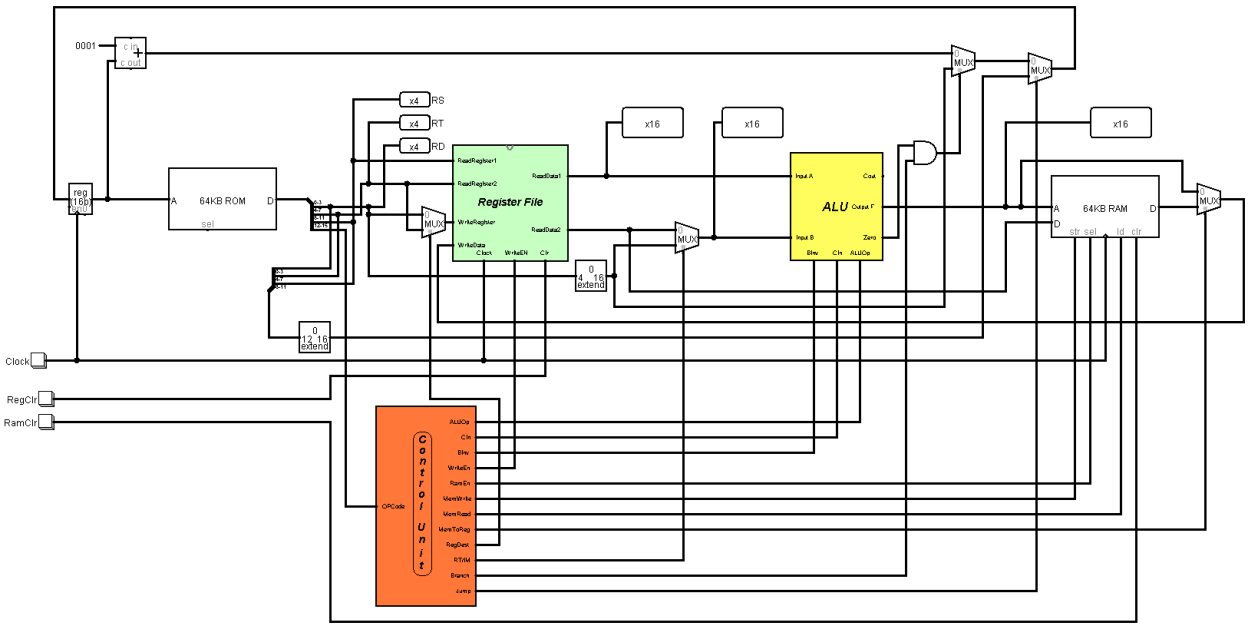

Fig: Complete CPU Datapath

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. 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.

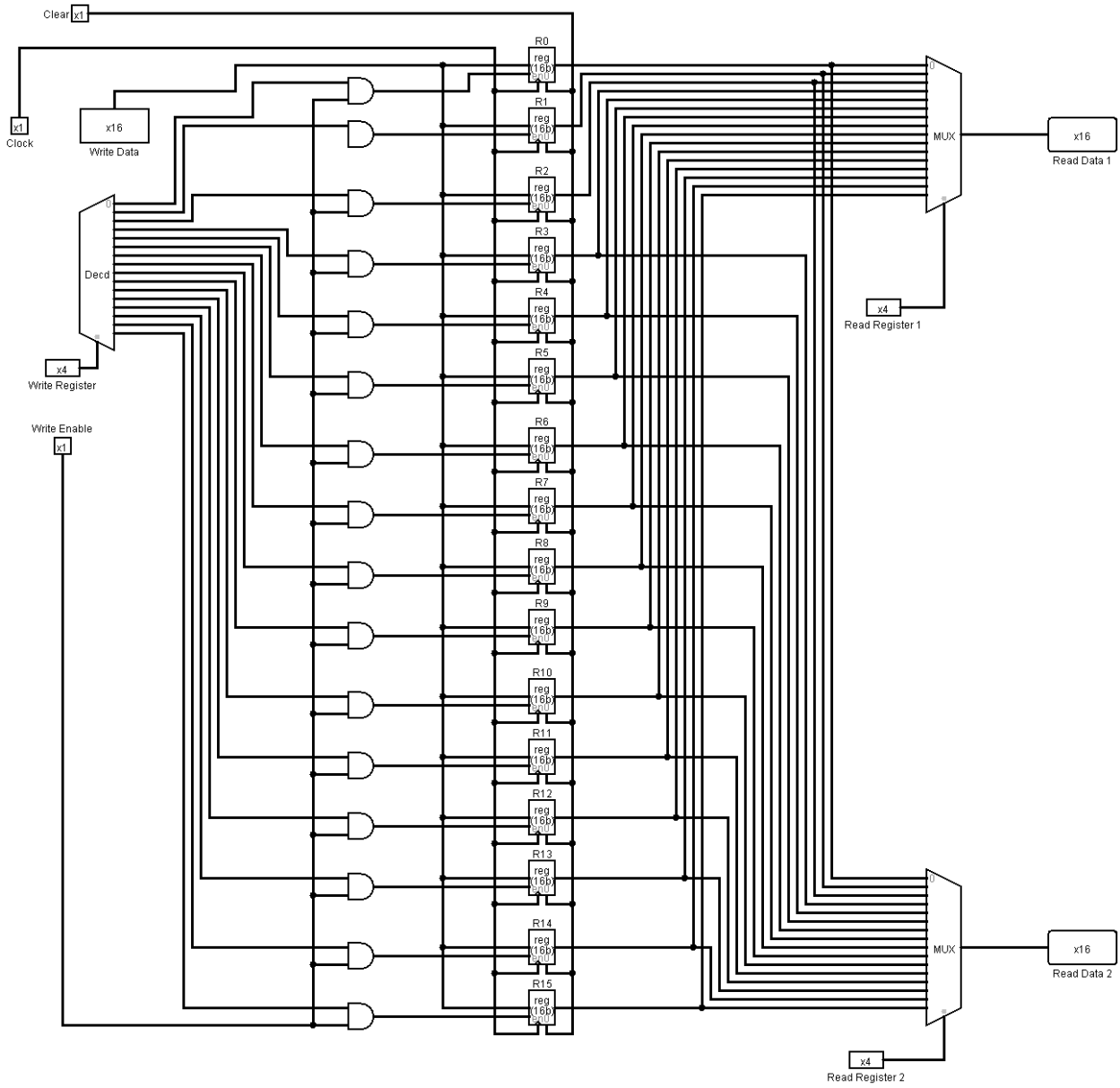

Fig: Register

ALU: 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 for addition, subtraction, lw,sw etc. operations.

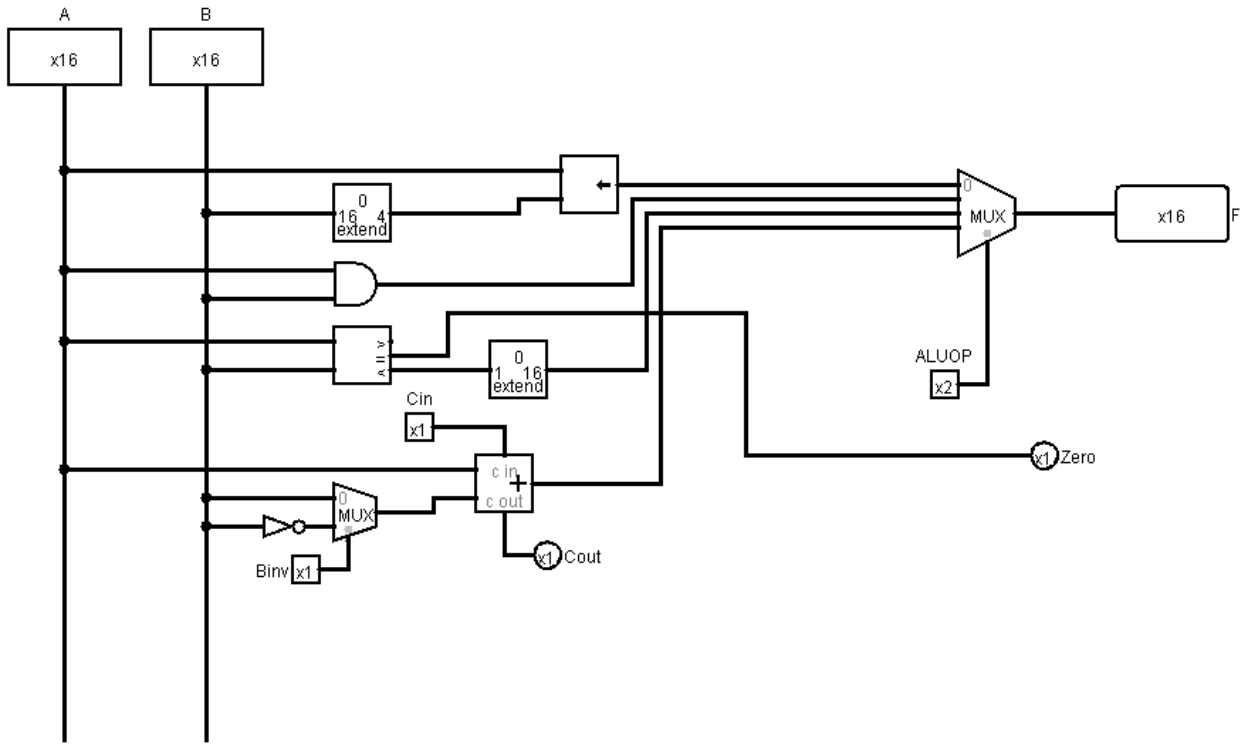

Fig: ALU

Control Unit: I need a control unit to send some specific signal for a specific operation.

- **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.
- **RT/IM:** 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. As per given instructions I needed to use a 4:16 decoder. According to the OpCodes given it will generate specific signal.

Hex	OP	Instruction	ALU OP		Cin	Binv	WriteEn	RamEn	MemWrite	MemRead	MemToReg	RegDest	RT/IM	Branch	Jump
0	0000	nop	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0001	sll	0	0	0	0	1	0	0	0	0	0	0	0	0
2	0010	and	0	1	0	0	1	0	0	0	0	0	0	0	0
3	0011	lw	1	1	0	0	1	1	0	1	1	1	1	0	0
4	0100	slt	1	0	0	0	1	0	0	0	0	0	0	0	0
5	0101	add	1	1	0	0	1	0	0	0	0	0	0	0	0
6	0110	addi	1	1	0	0	1	0	0	0	0	1	1	0	0
7	0111	sub	1	1	1	1	1	0	0	0	0	0	0	0	0
8	1000	sw	1	1	0	0	0	1	1	0	0	0	1	0	0
9	1001	jmp	0	0	0	0	0	0	0	0	0	0	0	0	1
A	1010	beq	0	0	0	0	0	0	0	0	0	0	0	1	0

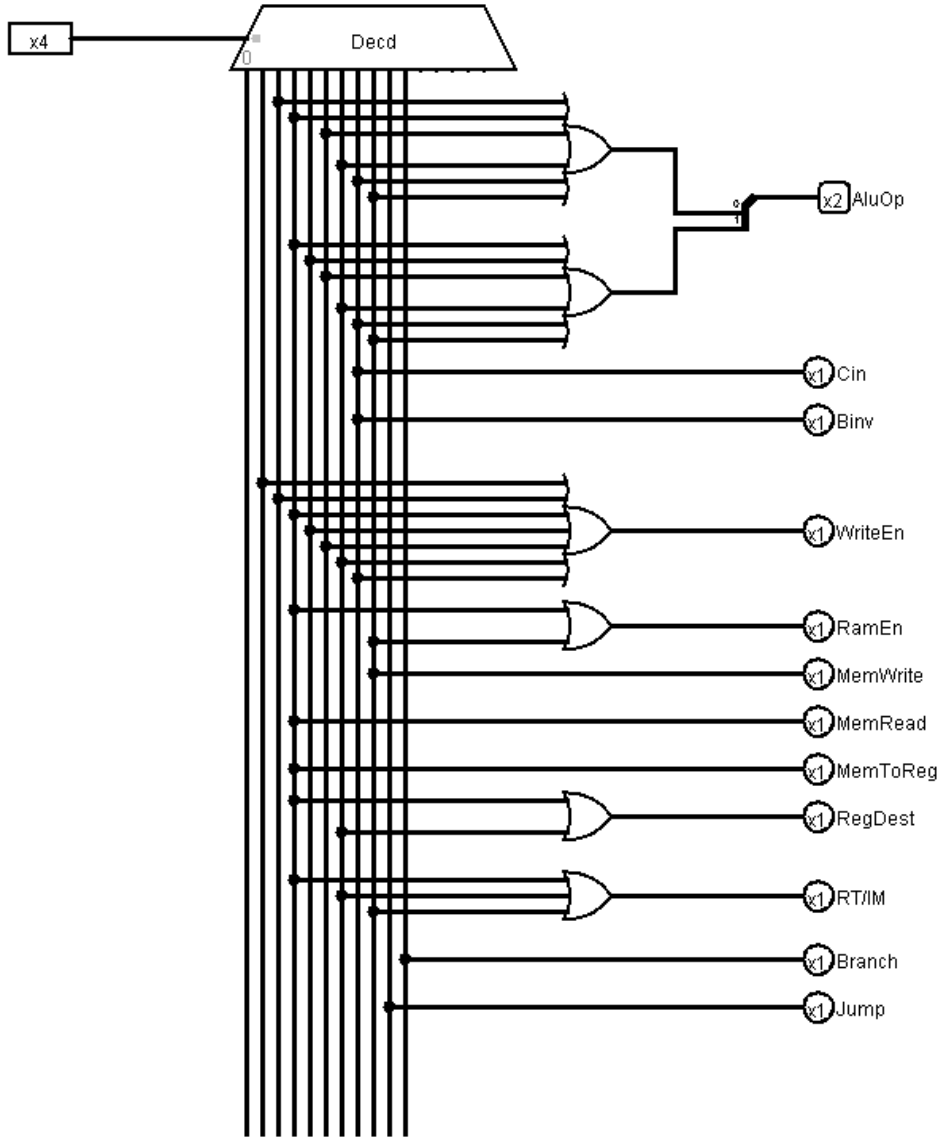

Fig: Control Unit

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.

List of Tables

Register List

We have selected registers from r0-r15 for general purpose. We assigned 4 bits for each of the register as we know in the instruction field in our ISA containing the register rs, rt and rd contains 4 bits each.

Conventional Name	Register Number	Binary Value
r0	0	0000
r1	1	0001
r2	2	0010
r3	3	0011
r4	4	0100
r5	5	0101
r6	6	0110
r7	7	0111
r8	8	1000
r9	9	1001
r10	10	1010
r11	11	1011

r12	12	1100
r13	13	1101
r15	14	1110
r15	15	1111

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
sll	R	0001
and	R	0010
lw	I	0011
slt	R	0100
add	R	0101
addi	I	0110
sub	R	0111
sw	I	1000
jmp	J	1001
beq	I	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:

The user has to give spaces between instruction words and nothing else like "," or "-" in between them in the "inputs" file. If user don't follow this format the system will show a valid code as invalid.

User Manual:

To run the program, one needs to run the python file called "**assembler.py**" which is provided in the folder. If one wants to see the code then open the "**assembler.py**" file*, it is absolutely necessary that the folder which is containing the program, has a file named "**inputs**" with no extension.* This is the file from where the assembler reads the assembly codes. The program reads the code from **"inputs"** file and writes the corresponding binary code in a file called **"outputs".** We have already provided an input file with corresponding output file in the project folder. If one wants to try his/her own assembly code then, he/she needs to write the codes to the application through the input file. Then he/she has to load the **"outpus"** in the rom. One important thing to notice is that, each line of the input file can only contain one instruction and words must be separated by spaces.

Discussion

The primary goal of this project was to create a 16-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 16 bits, I assigned 4 bits to each register. This CPU has

a total of 16 registers. Then, in accordance with the ISA, I built the ALU with the necessary arithmetic operations. To properly build the 16-bit ALU, I used 16-bit inputs and outputs, logic gates, adders, and MUXs. I separated the output's 16 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 16-bit register file. This has 16 registers, each of which is linked to a decoder. Different registers will be activated based on the decoder's 4-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 16-bit register, a 16-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 16-bit instructions are then split from the ROM into four 4-bit binaries, which are then passed to the 16-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. 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.

_____ *END* _____