**Department of Electrical and Computer Engineering**

**North South University, Dhaka-1229, Bangladesh**



Project Report

**Submitted by - Group 2**

Khalid Bin Shafiq - 1911342642

Shahjalal Ekbal - 1712397642

Shad Abdullah - 1912215642

Submitted to:

Dr. Tanzilur Rahman

CSE332.2

Summer 2021

# Table of Contents

[**Table of Contents**](#_heading=h.3znysh7) **2**

[**Introduction**](#_heading=h.2et92p0) **3**

[**Objective**](#_heading=h.tyjcwt) **3**

[**Usage**](#_heading=h.3dy6vkm) **3**

[**List of Tables**](#_heading=h.1t3h5sf) **4**

[**Register List**](#_heading=h.4d34og8) **4**

[**Op-Code List**](#_heading=h.2s8eyo1) **5**

[**Design**](#_heading=h.17dp8vu) **6**

[**Operation Type**](#_heading=h.3rdcrjn) **6**

[**Example Code**](#_heading=h.26in1rg) **8**

# Introduction

Our task was to design an assembler which will convert the assembly code to machine language. We designed a new 14 bit single-cycle CPU for a vendor company. A requirement in our design was that it needed to have separate Data and Instruction Memory. The instructions in our ISA can be used to solve logical, arithmetic, branching, and loop operations.

# Objective

Writing machine code by hand is very tedious. So, a programmer of our machine may opt in to write code in assembly. Since our machine can understand only machine code, we will have to write an assembler that will generate a machine code from a file containing assembly language that fits our system. The assembler reads a program written in an assembly language, then translates it into hex code and generates an output file containing hex machine code.

Assembly

Hex Code

Binary

Hex

Assembler

ASM

*Figure 01: Assembler diagram*

# Usage

Our Assembler is a Python CLI program that takes as arguments the name for the input file containing the assembly code, and the name for the output where the machine code will be stored.

*python assembler.py <inputfile.txt> <outputfile.txt>*

For example:

*python assembler.py input.txt output.txt*

# List of Tables

## Register List

We have selected registers from $zero, $sp, $s0-$s3, $t0 and $t1.

|  |  |  |
| --- | --- | --- |
| **Name of the Registers** | **Register Number** | **Value Assigned** |
| $zero | 0 | 000 |
| $sp | 1 | 001 |
| $s0 | 2 | 010 |
| $s1 | 3 | 011 |
| $s2 | 4 | 100 |
| $s3 | 5 | 101 |
| $t0 | 6 | 110 |
| $t1 | 7 | 111 |

## Op-Code List

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Category** | **Instruction** | **For**  **mat** | **Syntax** | **Meaning** | **Op**  **code** | **Addressing Mode** |
| Logical | AND | R | AND rd, rs, rt | rd = rs&rt | 00000 | Register |
| Logical | OR | R | OR rd, rs, rt | rd = rs | rt | 00001 | Register |
| Arithmetic | ADD | R | ADD rd, rs, rt | rd = rs + rt | 00111 | Register |
| Arithmetic | SUB | R | SUB rd, rs, rt | rd = rs - rt | 01000 | Register |
| Logical | NAND | R | NAND rd, rs, rt | Rd=rs NAND rt | 00010 | Register |
| Logical | NOR | R | NOR rd, rs, rt | rd = rs NOR rt | 00011 | Register |
| logical | XOR | R | XOR rd, rs, rt | rd=rs XOR rt | 00100 |  |
| Arithmetic | ADDi | I | ADDi rs, 2 | rs = rs + 2 | 01001 | Immediate |
| Data Transfer | LW | I | LW rs, imm | rs=Mem[$s3+imm] | 01010 | Indirect |
| Data Transfer | SW | I | SW rs, imm | Mem[$s3+imm] = rs | 01011 | Indirect |
| Logical | Sll | I | Sll rs, imm | rs=rs<<2 | 00101 | Immediate |
| Logical | Srl | I | Srl rs, imm | rs=rs>>2 | 00110 | Immediate |
| Conditional | Beq | I | Beq rs,imm | If (rs==$sp)  Go to line imm | 01100 | Direct |
| Conditional | Bne | I | Bne rs,8 | If(rs!=$sp)  Go to line imm | 01101 | Direct |
| Conditional | Slt | R | Slt rd,rs,rt | If(rs<rt) rd=1,else  rd=0 | 01110 | Register |
| Conditional | Slti | I | Slti rs,imm | If(rs<imm)  rs=1,else  rs=0 | 01111 | Immediate |
| Unconditional | Jump | J | J address | Jump to line imm | 10000 | Direct |
| Data Transfer | Din | R | Din rs | User input is stored in rs | 10001 | Register |
| Data Transfer | Dout | R | Dout rs | From rs,content is displayed | 10010 | Register |

# Design

Our assembly code is case insensitive. The assembler converts all the text in uppercase before parsing the code.

I type and R type instructions require a 1-bit data size input after the opcode. This bit indicates the size of the operands to take into account. However, in our assembly code, this bit is optional. When it is not assigned any value in our codes, it gets assigned a default value (1) by our compiler.

## Operation Type

In our Design, we have 5 types of operations.

1. **Arithmetic**: Add, Sub, Addi(Number of Items from Arithmetic:3)3

Theoretically, these operations can be used to implement any other arithmetic operations.

1. **Logical**: AND, OR, NOR, Sll (Number of Items form Logical: 4)

As NOR operation is a universal operation, it is enough to implement any logical operation. The other operations are provided as syntactic sugar.

1. **Conditional**: beq, bne, Slt, Slti (Number of Items from Conditional:4)

If we have equality and lower value checking implementation, we can use them to check any kind of relational operations by combining them. Here, BEQ and BNE checks for equality a=b, while SLT and SLTI can check if a<b. Besides relational checking, they can be used for branching, which can make up for if-else statements.

1. **Data Transfer**: lw, sw, Din, Dout (Number of Items from Data Transfer:4)

As our CPU will be capable of reading or writing to memory, it will have enough space to solve any general program. The CPU will also have user interactions through the Din and Dout operations.

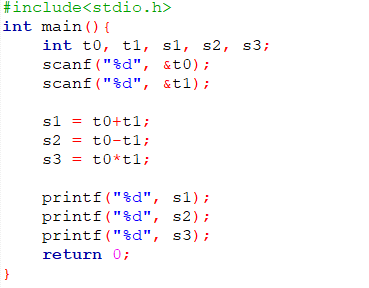
1. Unconditional: J (Number of Item from Unconditional:1)

Unconditional jump will be useful to implement any loop kind of structures in our programs.

Therefore, our design will be able to perform any general kind of logical, arithmetic, if-else branching, looping, and external reading/writing operations.

# Example Code

1. Output the sum of an array that is stored in memory 3 to 6. Perform addition, subtraction and multiplication operation between two integers.



L1: *DIN $t0* ; Store user input in t0

L2: *DIN $t1* ; Store user input in t1  
L3: *ADD $s1, $t0, $t1* ; Add t0 and t1 and store in s1  
L4: *SUB $s2, $t0, $t1* ; Subtract t1 from t0 and store in s2

L5: *MUL $s3, $t0, $t1* ;Multiply t0 by t1 and store in s3

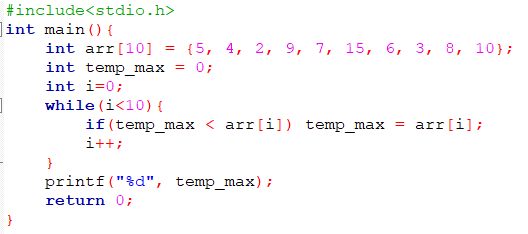
L6: *DOUT $s1* ;Display content of s1

L7: *DOUT $s2*  ; Display content of s2

L8: *DOUT $s3* ; Display content of s3

Execution time: 8 clock cycles

1. Find the maximum of an array that is stored from memory 3 to 12 using Loop Structure:



L0: *MOVI $s3, 3* ; $s3 is default address register

L1: *LW $s1, 0* ; **Loop Starts here**, load current array value  
L2: *SLT $s2, $t0, $s1* ; if (t0 < s1) s2 = 1, else s2 = 0

L3: *AND $sp, $sp, $zero* ; t0 is still maximum  
L3: *BEQ $s2, L6* ; t0 is still maximum

L4: *AND $t0, $zero, $t0* ; L5: *ADD $t0, $s1, $t0* ; t0 swapped with $s1  
  
L6: *ADDi $s0, 1* ; $s0 is loop counterL7: *MOVI $sp, 10* ; Loop needs to iterate 10 times

 L8: *BNE, $s0, L1* ; **Loop Ends** here

L9: *DOUT $t0* ; Display Maximum