ASSEMBLER

ABSTRACT

For this project I had to design an Instruction Set Architecture and implement an assembler for the same. I used the C++ programming language for the implementation of the assembler. The assembler inputs the symbolic program in the form of a .txt file and outputs the machine code representation as a string of 0's and 1's on the terminal. The instructions are fixed length instructions of length 32 bits each.

Machine Organization

The hypothetical machine for which the ISA is designed consists of:

1. A 32 bit data bus and a 16 bit address bus.
2. Fixed length instruction size of 32 bits.
3. There are 2 general purpose registers named R1 and R2 respectively. They are of 32 bits.
4. There are 6 flags namely zero flag, carry flag, overflow flag, input flag, output flag, and interrupt flag.
5. There exists mainly two more registers: Instrucon Pointer and Address Register. 6. Instrucon Pointer is the register that stores the address of the current executing instrucon. It is of 16 bits.
7. Memory is accessible only through the address register, hence all memory addresses must go through the address register. This register is connected to the memory address bus and stores the memory addresses that we access. It is of 16 bits.
8. No direct memory operations allowed. All operations on memory operands must go through the accumulator register named here as R1. All memory reference instructions transfer there data into the R1 register.

Instrucon Format

The instructions are majorly grouped into memory reference instructions, register instructions and input output instructions. There are 32 bits in each instrucon. Various groups of bits are assigned to addressing modes, opcodes and operands. The machine code format for each type of instrucon is described below.

Memory Reference Instructions:

0 3 15 32

Addressing Mode	Opcode	Address

Register Reference Instructions:

0 3 15 23 32

Addressing Mode	Opcode	Register 1	Register 2

/

Input – Output Instructions:

0 3 32

Addressing Mode	Opcode		

Addressing Modes:

There are 3 addressing modes in the ISA. These are

1. Immediate: The operand is part of the instrucon and is specified in the instrucon only. We append IM at the end of the instrucon to specify this addressing mode.
2. Direct: The operand is found at the address specified in the instrucon. We append D at the end of the instrucon to specify that the addressing mode is direct.
3. Indirect: The operand is found at the address found in the memory address specified in the instrucon. We append # at the end of the instrucon to specify that it is an indirect instrucon.
   In addition to these memory addressing modes we have 2 more addressing modes that refer to register mode and I/O mode that help the assembler in identifying the instrucon. In register reference instructions we have to append an R at the end of the instrucon to specify that instrucon is a register reference, similarly for IO instructions we append IO at the end.

Codes for addressing modes:

1. IM	0000	Immediate addressing mode
2. D	0011	Direct addressing mode
3. #	0010	Indirect addressing mode
4. IO	0111	I/O addressing mode
5. R	1111	Register mode

Instrucons

INSTRUCTION	BINARY CODE	FUNCTION
1. LD	000000000001	Loading instruction from memory or from other register
2. AND	000000010000	Performs AND operation
3. ADD	000000000010	Performs ADD operaon
4. XOR	000000000100	Performs XOR operation

/

5. ADC	000000001111	Performs AND operation including previous carry
6. XCHG	000000010001	Exchange the operands
7. STA	000000000101	Store content of R1 to memory
8. BUN	000000000111	Branch unconditionally
9. BSA	000000001010	Branch and save return address
10. NOT	000000100000	NOT the operand
11. IN	000000100001	Increment operand
12. DEC	000000100101	Decrement operand
13. SZ	111100000000	Skip if 0
14. SP	111100000001	Skip if > 0
15. SN	111100010000	Skip if < 0

16. CLA	111100000010	Clear R1
17. CLE	111100000100	Clear E
18. CLC	111100110000	Clear C
19. CLZ	111100110000	Clear Z
20. CLR	111100100001	Clear the register
21. HLT	111111111111	Halt computer
22. CIR	111100110001	Circulate right R1 and E
23. CIL	111101000000	Circulate le R1 and E
24. CMA	111101000011	Complement R1
25. CME	111101001000	Complement E
26. BLCFILL	111101001010	Fill from lowest bit of R1
27. BLCI	111101001100	Isolate lowest clear bit
28. BLCIC	111101010011	Isolate lowest clear bit and complement
29. BLCMSLK	111101100000	Mask from lowest clear bit
30. BLCS	111110000111	Set lowest clear bit
31. INP	11010000000100000000000000000	Input operand to R1
32. OUT	11010001001100000000000000000	Output operand from R1
33. SKI	11010101001000000000000000000	Skip on input flag

/

34. SKO	11010110001100000000000000000	Skip on output flag
35. ION	11010110001100000000000000000	Interrupt On
36. IOF	11010111010000000000000000000	Interrupt Off

CODE:-

```
#include <iostream>
#include <bits/stdc++.h>
#include <fstream>

using namespace std;

// convert decimals to binary strings
string decimal_to_binary(int num){

	string dec;
	string temp;

	while(num != 0){
			int digit = num % 2;
			if(digit == 1){
				dec += '1';
			}else{
				dec += '0';
			}
		num /= 2;
	}

	reverse(dec.begin(), dec.end());
	int n = dec.length();



	for(int i = 0; i < (16 - n); i++){
		temp += '0';
	}

	temp += dec;
	return temp;
```

```
}


int main()
{
   //hashmaps
   unordered_map<string,string> opcode;
   unordered_map<string,bool> pseudo;
   unordered_map<string,string> Register;
   unordered_map<string,string> addressing_mode;

// IM -> IMMEDIATE
// D -> DIRECT
// # -> INDIRECT
// IO -> INPUT/OUTPUT
// R -> REGISTER

   pseudo["ORG"]=true;
   pseudo["HLT"]=true;
   pseudo["END"]=true;
   pseudo["HEX"]=true;
   pseudo["DEC"]=true;

   addressing_mode["IM"]="0000";



                                                                                                    /
   addressing_mode["D"]="0011";
   addressing_mode["#"]="0010";
   addressing_mode["IO"]="0111";
   addressing_mode["R"]="1111";
```

```
Register["R1"]="00100001";
Register["R2"]="01011000";


/*
        pseudo["END"]="0000000000000000";
pseudo["ORG"]="0000000000000000";
pseudo["HALT"]="0000000000000000";
pseudo["DEC"]="0000000000000000";
pseudo["HEX"]="0000000000000000"; */

opcode["LD"]="000000000001";
opcode["AND"]="000000010000";
opcode["ADD"]="000000000010";
opcode["XOR"]="000000000100";
opcode["ADC"]="000000001111";
opcode["XCHG"]="000000010001";
opcode["STA"]="000000000101";
opcode["BUN"]="000000000111";
opcode["BSA"]="000000001010";
opcode["NOT"]="000000100000";
opcode["IN"]="000000100001";
opcode["DEC"]="000000100101";
opcode["SZ"]="111100000000";


                                                                                /
opcode["SP"]="111100000001";
opcode["SN"]="111100010000";
opcode["CLA"]="111100000010";
opcode["CLE"]="111100000100";
```

```
opcode["CLC"]="111100100000";
opcode["CLZ"]="111100110000";
opcode["CLR"]="111100100001";
opcode["HLT"]="111111111111";
opcode["CIR"]="111100110001";
opcode["CIL"]="111101000000";
opcode["CMA"]="111101000011";
opcode["CME"]="111101001000";
opcode["BLCFILL"]="111101001010";
opcode["BLCI"]="111101001100";
opcode["BLCIC"]="111101010011";
opcode["BLCMSK"]="111101100000";
opcode["BLCS"]="111110000111";
opcode["INP"]="11010000000100000000000000000";
opcode["OUT"]="11010001001100000000000000000";
opcode["SKI"]="11010100000100000000000000000";
opcode["SKO"]="11010101001000000000000000000";
opcode["ION"]="11010110001100000000000000000";
opcode["IOF"]="11010111010000000000000000000";

//locally store symbolic program in vector of strings
vector<string*> v;
string s;
        // file inout
ifstream input("cao_input.txt");
if(input.is_open())
{



   while(getline(input,s))
   {
      string* temp = new string;
```

```
        *temp=s;
        v.push_back(temp);
    }
    input.close();
}

        /*ofstream output("cao_output.txt");
output.open("cao_output.txt");
for(int m=0;m<v.size();m++)
{
    string temp;
    temp = *(v[m]);
    cout<<temp<<endl;
    output<<temp<<'/n';
}*/
unordered_map<string,string> Variables;
//call for first pass

        // locaon counter
int LC = 0;
        // counter for number of lines
        int i = 0;
        // number of lines in input
        int n = v.size();

        while ((n--) != 0){
                string temp = *(v[i++]);
```

/

```
                //TODO: change to check if there is no comma
                // check for pseudo instrucon
```

```
            if(temp[0] == ' ') {
                if(temp.substr(1,3) == "ORG"){
                    int len = temp.length();
                    string num = temp.substr(5,len-5);
                    LC = stoi(num);
                }
                else if(temp.substr(1,3) == "END"){
                    break;
                }
            // case if label found
            } else {
                string label;

                for(int j = 0; j < 4; j++){
                    if(temp[j] == ','){
                        break;
                    }
                    else{
                        label += temp[j];
                    }
                }
                Variables[label] = decimal_to_binary(LC);
            }

            // go to next line
            LC++;
        }
```

/

```
//second_pass
```

```
    // start scanning from the start again
    LC=0;

    for (int m = 0; m < v.size(); m++) {

string current_instrucon = *(v[m]);;
string machine_code = "";

            //points to character in current_instrucon
int i = 0;



//make i point to first character of opcode
            // if no label is found
            if(current_instrucon[0] == ' ')
{
   i = 1;
}
// if label found
            else
{
  while(current_instrucon[i] != ',')
  {
     i++;
  }
  i += 2;
            }

            // string that stores the opcode of the
```

```
string op_code = "";

// extract opcode and make i point to character aer opcode //
                note what if this is does not contain any operand?
                while(current_instrucon[i] != ' ')
{
   op_code = op_code + current_instrucon[i];
   i++;
}

                // check if it is a pseudoinstrucon
                if(pseudo.count(op_code) > 0)
{
                        // if ORG instrucon update LC
   if(op_code == "ORG")
   {
                            int len = current_instrucon.length();
                            string num = current_instrucon.substr(5, len - 5);
                            LC = stoi(num);
   }

                        // if END instrucon break out and generate output
   else if(op_code=="END")
   {
      break;
   }
}

                // otherwise check for non pseudo instrucon
                else if(opcode.count(op_code) > 0)
{
```

/

```
                    // add bits of opcode to binary code
machine_code = machine_code + opcode[op_code];


i++;
if(current_instrucon[i] != '/'&&i<current_instrucon.length()) {
                    // store the name of the label or register in string var_reg
  string var_reg = "";
  while(current_instrucon[i] != ' ')
  {
    var_reg = var_reg + current_instrucon[i];
    i++;
  }


                    // if it is a label
  if(Variables.count(var_reg) > 0)
  {
     machine_code = machine_code + Variables[var_reg];
  }
                    // if a register
  else if(Register.count(var_reg) > 0)
  {
   machine_code = machine_code + Register[var_reg];
  }
                    // if an addressing mode
  else if(addressing_mode.count(var_reg) > 0)
  {
     machine_code = addressing_mode[var_reg] + machine_code;
  }


  else
```

/

```
  {
     bool flag=true;
     // to check whether the operand ia an integer or not
     for(int c=0;c<var_reg.length();c++)
     {
        if(isdigit(var_reg[c])==false)
        {
           flag=false;
           break;
        }
     }
     //if operand is an integer
     if(flag==true)
     {
        int decimal_operand=stoi(var_reg);
        string binary_operand = decimal_to_binary(decimal_operand);
        machine_code=machine_code+binary_operand;
     }
     // nothing found output error
     else
     {
        cout<<var_reg<<" : label not declared"<<endl;
        exit(1);
     }

  }
  i++;
}
```

/

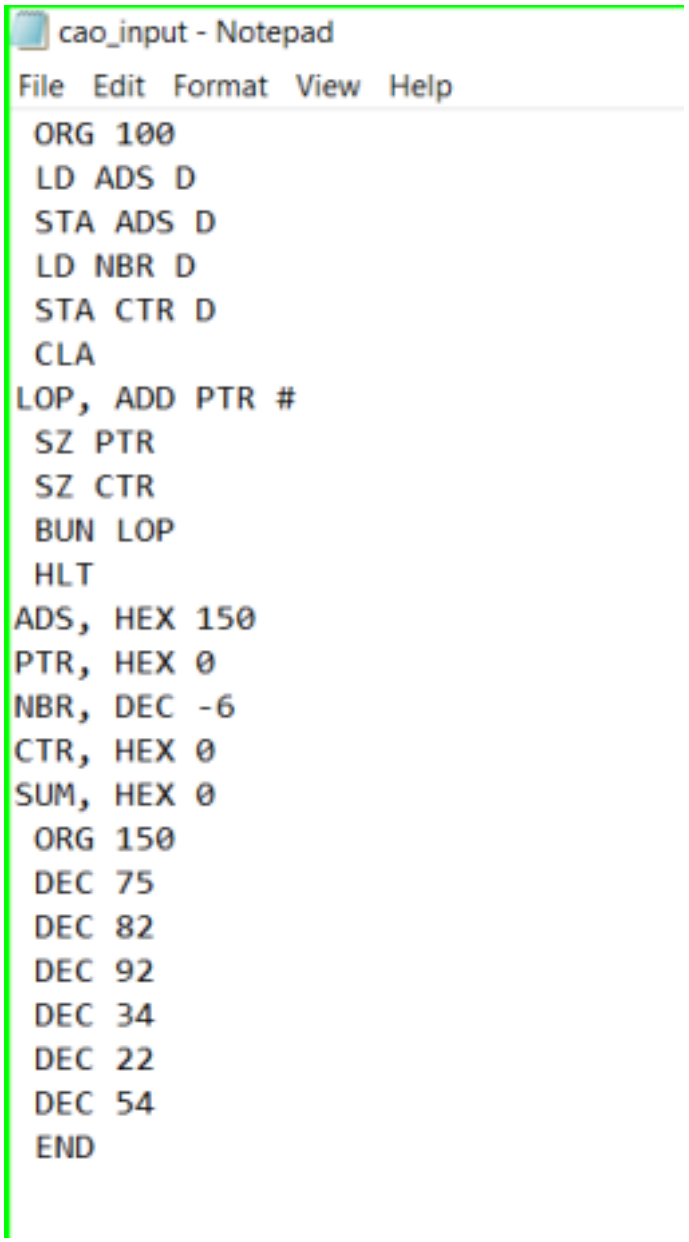
```
                    if(current_instrucon[i] != '/'&&i<current_instrucon.length()) {
string var_reg = "";
while (current_instrucon[i] != ' ')
{
  var_reg = var_reg + current_instrucon[i];
  i++;
}

if (Variables.count(var_reg) > 0)
{
   machine_code=machine_code + Variables[var_reg];
}
else if(Register.count(var_reg) > 0)
{
 machine_code = machine_code + Register[var_reg];
}
else if(addressing_mode.count(var_reg) > 0)
{
   machine_code = addressing_mode[var_reg] + machine_code;
}
else
{
   bool flag=true;
   // to check whether the operand ia an integer or not
   for(int c=0;c<var_reg.length();c++)
   {
      if(isdigit(var_reg[c])==false)
```

```
        {
            flag=false;
            break;



        }
    }
    //if operand is an integer
    if(flag==true)
    {
        int decimal_operand=stoi(var_reg);
        string binary_operand = decimal_to_binary(decimal_operand);
        machine_code=machine_code+binary_operand;
    }
    // nothing found output error
    else
    {
        cout<<var_reg<<" : label not declared"<<endl;
        exit(1);
    }
  }
  i++;
}

// repeat for next part of instrucon
if(current_instrucon[i] != '/' && i < current_instrucon.length()) {
  string var_reg = "";
  while(current_instrucon[i] != ' ')
  {
    var_reg=var_reg+current_instrucon[i];
    i++;
  }
```

```
        if(Variables.count(var_reg)>0)
        {
           machine_code = machine_code + Variables[var_reg];




        }
        else if(Register.count(var_reg) > 0)
        {
          machine_code = machine_code + Register[var_reg];
        }
        else if(addressing_mode.count(var_reg) > 0)
        {
           machine_code = addressing_mode[var_reg] + machine_code;
        }
        else
        {
           cout<<var_reg<<" : label not declared"<<endl;
           exit(3);
        }
      }
      // output code on the terminal
      while(machine_code.length()<32)
         machine_code += "0";
      cout<<machine_code<<"\n";
    }

                  else
    {
      cout<<op_code<<" : not an instrucon"<<endl;
    }
    LC++;
} }
```

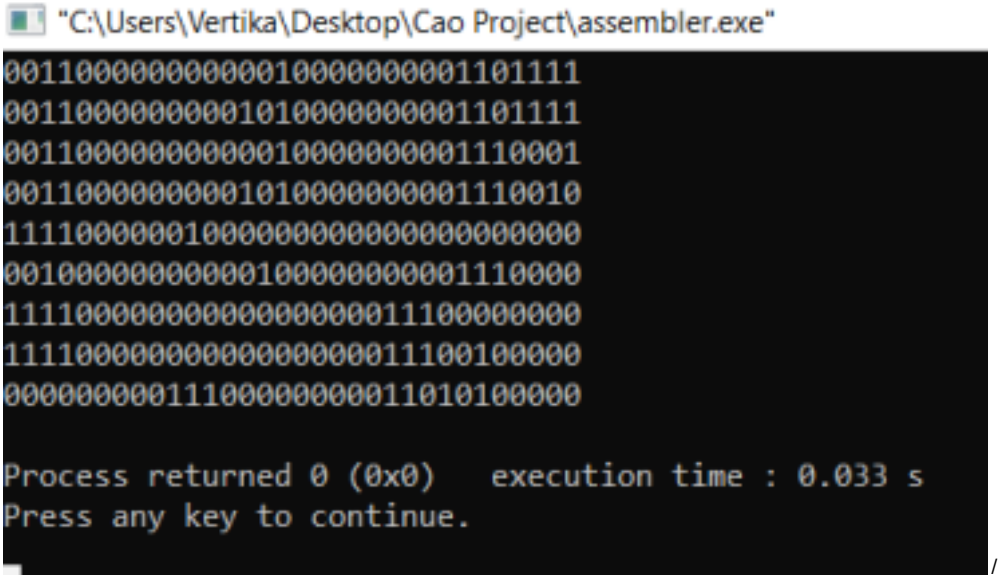
EXAMPLE

INPUT FILE:-



OUTPUT :-

/


"C:\Users\Vertika\Desktop\Cao Project\assembler.exe"
0011000000000001000000001101111
0011000000000101000000001101111
0011000000000001000000001110001
0011000000000101000000001110010
1111000000100000000000000000000
0010000000000010000000001110000
1111000000000000000001110000000
1111000000000000000001110010000
0000000001110000000011010100000
Process returned 0 (0x0)   execution time : 0.033 s
Press any key to continue.
/