```
#ifndef PROCESSOR_H
#define PROCESSOR_H 1

#include <iostream.h>
#include <string.h>
#include <stdlib.h>

//Set the op codes as integer constants
#define LD 0x0
#define LDI 0x1
#define ST 0x2
#define ADD 0x3
#define SUB 0x4
#define MUL 0x5
#define DIV 0x6
#define OR 0x7
#define AND 0x8
#define SHL 0x9
#define SHR 0xA
#define IO 0xB
#define BR 0xC
#define BRZ 0xD
#define BRN 0xE
#define BRS 0xF

class Processor {
private:
    int P_Counter;
    int reg_array[4];
    int Leading_Ones(int);
    int Leading_Zeroes(int);
    bool Is_Negative_Value(int);
    int S_Of_X(int, int);
    bool Is_Overflow(int);
    bool Is_Printable_Ascii(char);
public:
     Processor();
    ~Processor();
    void Increment_PC();
    void Set_PC(int);
    int Get_PC();
    void Load_Register(int, int);
    int Get_Register(int);
    void Dump_Regs(fstream&);
    void No_Op();
    bool Do_This(Memory&, int, int, int, int, fstream&, fstream&, fstream&);
};

//Private Functions
int Processor::Leading_Ones(int twentybit)
{
    int sigf = 0xfff00000;
    int thirtytwobit = sigf|twentybit;
    return thirtytwobit;
}
int Processor::Leading_Zeroes(int thirtytwobit)
{
    int sigf = 0x000fffff;
    int twentybit = sigf&thirtytwobit;
    return twentybit;
}
bool Processor::Is_Negative_Value(int value)
{
    int shifted = value/(int)pow(2, 19);

    if (shifted != 0)
    {
    return true;
```

```
    }
    return false;
}
int Processor::S_Of_X(int S, int X)
{
    if (X==0)
    {
    return S;
    }
    else
    {
    int reg_contents = Get_Register(X);
    if (Is_Negative_Value(reg_contents))
    {
        reg_contents = Leading_Ones(reg_contents);
    }
    return (S + reg_contents);
    }
}
bool Processor::Is_Overflow(int val)
{
    int shifted = (val/(int)pow(2, 20));
    if (shifted != 0)
    {
    return true;
    }
    else
    {
    return false;
    }
}
bool Processor::Is_Printable_Ascii(char c)
{
    if (c == 9 || (c >= 32 && c <= 126))
    {
    return true;
    }
    else
    {
    return false;
    }
}
//Public Functions

Processor::Processor()
{
    // Default Constructor definition
    P_Counter = 0;

}
Processor::~Processor()
{
    // Default Destructor definition
}

void Processor::Increment_PC()
{
    P_Counter++;
}

void Processor::Set_PC(int n)
{
    P_Counter = n;
}

int Processor::Get_PC()
{
    return P_Counter;
```

```
}

void Processor::Load_Register(int reg_num, int reg_value)
{
    reg_array[reg_num] = reg_value;
}

int Processor::Get_Register(int reg_num)
{
    return reg_array[reg_num];
}

void Processor::Dump_Regs(fstream& outs)
{
    int n = 0;
    while (n < 4)
    {
    outs << "\nR" << n << ':' << hex <<  Get_Register(n);
    n++;
    }
    outs << "\nPC: " << dec << Get_PC();
}
void Processor::No_Op()
{
    Increment_PC();
}


bool Processor::Do_This(Memory& mem, int OP, int R, int X, int S, fstream& aux_in, fstream↙
    & outs, fstream& t_outs)
{
    t_outs << "\nOld PC: " << dec << Get_PC();
    Increment_PC();
    t_outs << "\nNew PC: " << dec << Get_PC();
    switch(OP)
    {
        case LD:
    {
        int addr = S_Of_X(S, X);
        if (addr < 0 || addr > 255)
        {
        t_outs << "\nError Code #3\n" << addr;
        }
        else
        {
        t_outs << "\nLD\n" << "Address: " << dec << addr << '\n'
               << "Address Content: " << hex << mem.Get_Memory(addr) << '\n'
               << "Register: " << R << '\n' << "Old Contents: "
               << Get_Register(R) << '\n';
        Load_Register(R, mem.Get_Memory(addr));
        t_outs << "New Contents: " << Get_Register(R) << '\n';
        }
    }return false;
        case LDI:
    {
        int imm = S_Of_X(S, X);

        if (Is_Overflow(imm))
        {
        t_outs << "\nError Code #4\n";
        }
        else
        {
        imm = Leading_Zeroes(imm);
        t_outs << "\nLDI\n" << "S(X): " << hex << imm << '\n'
               << "Register: " << R << '\n' << "Old Contents: "
               << Get_Register(R) << '\n';
        Load_Register(R, imm);
```

```
    t_outs << "New Contents: " << Get_Register(R) << '\n';
    }

}return false;
    case ST:
{
    int addr = S_Of_X(S, X);
    if (addr < 0 || addr > 255)
    {
    t_outs << "\nError Code #3\n";
    }
    else
    {
    t_outs << "\nST\nRegister: " << R << "\nRegister Contents: "
           << hex << Get_Register(R) << "\nAddress: " << dec << addr
           << "\nOld Contents: " << hex << mem.Get_Memory(addr);
    mem.Load_Memory(addr, Get_Register(R));
    t_outs << "\nNew Contents: " << mem.Get_Memory(addr) << '\n';
    }

}return false;
    case ADD:
{
    int addr, val, sum;
    addr = S_Of_X(S, X);
    if (addr < 0 || addr > 255)
    {
    t_outs << "\nError Code #3\n";
    }
    else
    {
    sum = Get_Register(R);
    val = mem.Get_Memory(addr);
    if (Is_Negative_Value(sum))
    {
        sum = Leading_Ones(sum);
    }
    if (Is_Negative_Value(val))
    {
        val = Leading_Ones(val);
    }
    if ((val * sum) >= 0)
    {
        if (Is_Overflow((val + sum)))
        {
        t_outs << "\nError Code #4\n";
        }
    }
    else
    {
        t_outs << "\nADD\nRegister: " << R << "\nRegister Contents: "
           << hex << Get_Register(R) << "\nValue to add: "<< dec << val;
        sum += val;
        Load_Register(R,Leading_Zeroes(sum));
        t_outs << "\nNew Contents: " << Get_Register(R) << '\n';

    }
    }

}return false;
    case SUB:
{
    int addr, val, sum;
    addr = S_Of_X(S, X);
    if (addr < 0 || addr > 255)
    {
    t_outs << "\nError Code #3\n";
    }
```

```
    else
    {
    sum = Get_Register(R);
    val = mem.Get_Memory(addr);
    if (Is_Negative_Value(sum))
    {
        sum = Leading_Ones(sum);
    }
    if (Is_Negative_Value(val))
    {
        val = Leading_Ones(val);
    }
    if ((val * sum) < 0)
    {
        if (Is_Overflow((sum - val)))
        {
        t_outs << "\nError Code #4\n";
        }
    }
    else
    {
        t_outs << "\nSUB\nRegister: " << R << "\nRegister Contents: "
           << hex << Get_Register(R) << "Value to subtract: " << dec << val;
        sum -= val;
        Load_Register(R,Leading_Zeroes(sum));
        t_outs << "\nNew Contents: " << Get_Register(R) << '\n';

    }
    }
}return false;
    case MUL:
{
    int addr, val, sum;
    addr = S_Of_X(S, X);
    if (addr < 0 || addr > 255)
    {
    t_outs << "\nError Code #3\n";
    }
    else
    {
    sum = Get_Register(R);
    val = mem.Get_Memory(addr);
    if (Is_Negative_Value(sum))
    {
        sum = Leading_Ones(sum);
    }
    if (Is_Negative_Value(val))
    {
        val = Leading_Ones(val);
    }
    if (Is_Overflow((sum * val)))
    {
        t_outs << "\nError Code #4\n";
    }
    else
    {
        t_outs << "\nMUL\nRegister: " << R << "\nRegister Contents: "
           << hex << Get_Register(R) << "Value to multiply: " << dec <<  val;
        sum *= val;
        Load_Register(R,Leading_Zeroes(sum));
        t_outs << "\nNew Contents: " << Get_Register(R) << '\n';

    }
    }
}return false;
    case DIV:
{
    int addr, val, sum;
```

```
    addr = S_Of_X(S, X);
    if (addr < 0 || addr > 255)
    {
    t_outs << "\nError Code #3\n";
    }
    else
    {
    sum = Get_Register(R);
    val = mem.Get_Memory(addr);
    if (Is_Negative_Value(sum))
    {
        sum = Leading_Ones(sum);
    }
    if (Is_Negative_Value(val))
    {
        val = Leading_Ones(val);
    }
    if (Is_Overflow((sum / val)))
    {
        t_outs << "\nError Code #4\n";
    }
    else
    {
        t_outs << "\nDIV\nRegister: " << R << "\nRegister Contents: "
           << hex << Get_Register(R) << "Value to divide: " << dec << val;
        sum /= val;
        Load_Register(R,Leading_Zeroes(sum));
        t_outs << "\nNew Contents: " << Get_Register(R) << '\n';

    }
    }
}return false;
    case OR:
{
    int addr = S_Of_X(S, X);
    int val, val2;
    if(addr < 0 || addr > 255)
    {
    t_outs << "\nError Code #3\n";
    }
    t_outs << "\nOR\nRegister: " << R << "\nRegister Contents: " << Get_Register(R)
       << "\nMemory Address: " << dec << addr << "\nMemory Contents: " << hex
       << mem.Get_Memory(addr);
    val=Get_Register(R);
    val2=mem.Get_Memory(addr);
    val=val||val2;
    Load_Register(R,val);
    t_outs << "New Contents: " << Get_Register(R) << '\n';

}return false;
    case AND:
{
    int addr = S_Of_X(S, X);
    int val, val2;
    if(addr < 0 || addr > 255)
    {
    t_outs << "\nError Code #3\n";
    }
    t_outs << "\nAND\nRegister: " << R << "\nRegister Contents: " << Get_Register(R)
       << "\nMemory Address: " << dec << addr << "\nMemory Contents: " << hex
       << mem.Get_Memory(addr);
    val=Get_Register(R);
    val2=mem.Get_Memory(addr);
    val=val&val2;
    Load_Register(R,val);
    t_outs << "\nNew Contents: " << Get_Register(R) << '\n';

}return false;
```

```
    case SHL:
{
    int Sx, reg_value = Get_Register(R);
    Sx = S_Of_X(S, X);
    Sx = Leading_Zeroes(Sx);
    if (Sx > 19)
    {
    t_outs << "\nError Code 6\n";
    }
    else
    {
    t_outs << "\nSHL\nRegister: " << R << "\nShift by " << dec
           << Sx << " bits\nOriginal Content: "
           << hex << Get_Register(R);
    reg_value *= (int)pow(2, Sx);
    reg_value = Leading_Zeroes(reg_value);
    Load_Register(R, reg_value);
    t_outs << "\nNew Content: " << Get_Register(R) << '\n';
    }

}return false;
    case SHR:
{
    int Sx, reg_value = Get_Register(R);
    Sx = S_Of_X(S, X);
    Sx = Leading_Zeroes(Sx);
    if (Sx > 19)
    {
    t_outs << "\nError Code #6\n";
    }
    else
    {
    t_outs << "\nSHR\nRegister: " << R << "\nShift by " << dec << Sx
           << " bits\nOriginal Content: " << hex << Get_Register(R);
    int temp = reg_value;
    //must checkif Sign Bit is 1 or 0 and propagate accordingly
    temp /= (int)pow(2,19);
    if(temp == 0)
    {
        reg_value /= (int)pow(2,Sx);
    }else
    {
        int i = (int)pow(2,19);
        while(Sx > 0)
        {
        reg_value /= 2;
        reg_value += i;//add 1 back onto end of 20-bit int
        Sx--;
        }
    }
    Load_Register (R, reg_value);
    t_outs << "\nNew Content: " << Get_Register(R) << '\n';
    }

}return false;

    case IO:
{
    switch(R)
    {
        case 0:
    {
        int addr = S_Of_X(S, X);
        if(addr < 0 || addr > 255)
        {
        t_outs << "\nError Code #3\n";
        }
        else
```

```
        {
        int val;
        char ch[2];
        aux_in >> val;
        aux_in.getline(ch,1);
        t_outs << "\nIO, R=0\nInput Value:" << val
               << "\nAddress:" << dec << addr
               << "\nOld Value:" << hex << mem.Get_Memory(addr);
        mem.Load_Memory(addr, val);
        t_outs << "\nNew Value:" << dec << mem.Get_Memory(addr) << '\n';
        }
    }return false;
        case 1:
    {
        int addr = S_Of_X(S, X);
        if(addr < 0 || (addr+3) > 255)
        {
        t_outs << "\nError Code #3\n";
        }
        else
        {
        char input[8];
        char string[80];
        aux_in.getline(string, 80);
        int len = strlen(string), pad = 7 - len;
        if(pad >= 0 )
        {
            while(pad >= 0)
            {
            input[pad] = ' ';
            pad--;
            }
            input[8-len] = 0; //Must NULL terminate "string" so that strcat() will
work
        }
        if(len > 8)
        {len = 8;}
        strncat(input, string, len);
        int n = 0;
        t_outs << "\nIO, R=1\nInput Value:" << input;
        while (n < 4)
        {
            t_outs << "\nAddress:" << dec << (addr+n) << "\nOld Value:"
               << hex << mem.Get_Memory(addr+n);
            int store_it = input[2*n];
            store_it *= (int)pow(2, 8);
            store_it += input[(2*n+1)];
            store_it *= (int)pow(2, 4);
            mem.Load_Memory((addr+n), store_it);
            t_outs << "\nNew Value:" << mem.Get_Memory(addr+n)
               << '\n';
            n++;
        }
        }
    }return false;
        case 2:
    {
        int addr = S_Of_X(S, X);
        if (addr < 0 || addr > 255)
        {
        t_outs << "\nError Code #3\n";
        }
        else
        {
        t_outs << "\nIO, R=2\nOutputting Address:" << hex << addr << '\n';
        outs << '\n' << dec << mem.Get_Memory(addr) << '\n';
        }
    }return false;
```

```
        case 3:
    {
        int addr = S_Of_X(S, X);
        if (addr < 0 || (addr+3) > 255)
        {
        t_outs << "\nError Code #3\n";
        }
        else
        {
        t_outs << "\nIO, R=3\nOutputting Addresses:" << dec << addr << " through "
               << dec << (addr+4) << '\n';
        outs << '\n';
        int n = 0;
        while (n < 4)
        {
            int temp = (mem.Get_Memory(addr+n)/(int)pow(2, 12));
            char op = temp;
            if (Is_Printable_Ascii(op))
            {
            outs << op;
            }
            temp = (mem.Get_Memory(addr+n)*(int)pow(2, 20));
            temp /= (int)pow(2, 24);
            op = temp;
            if (Is_Printable_Ascii(op))
            {
            outs << op;
            }
            n++;
        }
        outs << '\n';
        }
    }return false;
    }
}
    case BR:
{
    int Sx = S_Of_X(S, X);
    if (R == 0)
    {
    if(X == 0)
    {
        t_outs << "\nBR: R=0, X=0\n";;
        return true;
    }
    else if(X == 1)
    {
        //Dump all of Memory
        t_outs << "\nBR: R=0, X=1\n";
        outs << "\nDump Memory Values:\n";
        mem.Dump_Mem(outs);
        return true;
    }
    else if(X == 2)
    {
        //Dump all Registers (and PC)
        t_outs << "\nBR: R=0, X=2\n";
        outs << "\nDump Register and PC Values:\n";
        Dump_Regs(outs);
        outs << '\n';
        return true;
    }
    else // X ==3
    {
        //Dump all of Memory and Registers (and PC)
        t_outs << "\nBR: R=0, X=3\n";
        outs << "\nDump Register, PC, and Memory Values:\n";
        Dump_Regs(outs);
```

```
            outs << '\n';
            mem.Dump_Mem(outs);
            return true;
        }
        }
        else if(R == 1) //Dump all Mem, PC, and Regs, and Branch to address S(x)
        {
        //Branch to S(x)
        if (Sx < 0 || Sx > 255)
        {
            t_outs << "\nError Code #3\n";
        }
        else
        {
            //Dump Memory, Regs, and PC; the branch to S(x)
            t_outs << "\nBR: R=1\n";
            Dump_Regs(outs);
            outs << '\n';
            mem.Dump_Mem(outs);
            t_outs << "\nSetting PC to:" << dec << Sx << '\n';
            Set_PC(Sx);
        }
        }
        else if(R == 2) // Branch to address PC + S(x)
        {
        int addr = Sx + Get_PC();
        if (addr < 0 || addr > 255)
        {
            t_outs << "\nError Code #3\n";
        }
        else
        {
            t_outs << "\nBR: R=2\nSetting PC to:" << dec << addr << '\n';
            Set_PC(addr);
        }
        }
        else // R == 3  Branch to address S(x)
        {
        if (Sx < 0 || Sx > 255)
        {
            t_outs << "\nError Code #3\n";
        }
        else
        {
            t_outs << "\nBR: R=3\nSetting PC to:" << dec << Sx << '\n';
            Set_PC(Sx);
        }
        }
    }return false;
        case BRZ:
    {
        if (Get_Register(R) == 0)
        {
        int addr = S_Of_X(S, X);
        if(addr < 0 || addr > 255)
        {
            t_outs << "\nError Level #3\n";
        }
        else
        {
            t_outs << "\nBRZ\nSetting PC to:" << dec << addr << '\n';
            Set_PC(addr);
        }
        }
    }return false;
        case BRN:
    {
        if (Is_Negative_Value(Get_Register(R)))
```

```
        {
        int addr = S_Of_X(S, X);
        if(addr < 0 || addr > 255)
        {
            t_outs << "\nError Level #3\n";
        }
        else
        {
            t_outs << "\nBRN\nSetting PC to: " << dec << addr << '\n';
            Set_PC(addr);
        }
        }
    }return false;
        case BRS:
    {
        int addr = S_Of_X(S, X);
        if (addr < 0 || addr > 255)
        {
        t_outs << "\nError Code #3\n";
        }
        else
        {
        t_outs << "\nBRS\nRegister: " << R << "\nRegister Contents: " << hex <<
    Get_Register(R)
                << "\nNew Contents: ";
        Load_Register(R, Get_PC());
        t_outs << hex << Get_Register(R) << "\nSetting PC to: " << dec << addr << '\n';
        Set_PC(addr);
        }
    }return false;
    }
}

#endif
```