/\* ==================================================================================

ECE 486 / Winter 2015 / PDP-8 simulator project

Team:

Deborah Denhart

Jeremiah Franke

==================================================================================

File:     OpTable.cpp

Date:     03/02/2015

Description:  This file contains the classes Opcode7inst, Opcode7List, OpRow and

                optable

================================================================================== \*/

#ifndef \_\_OPTABLE\_H\_INCLUDED\_\_

#define \_\_OPTABLE\_H\_INCLUDED\_\_

//Dependencies

//==================================================================================

class alu;

//Defines

//==================================================================================

//#define DEBUG\_OP

//Class OpRow

//==================================================================================

class OpRow

{

private:

    unsigned short m\_iTotalCycles;

    char\* m\_sMnemonic;

    unsigned short m\_iCycles;

    unsigned short m\_iTotalUsed;

    unsigned short m\_rOpcode;

    void fillOpRow();

    void clearOpRow();

public:

    OpRow();

    OpRow(const char\* mnem, unsigned short cycles, unsigned short index);

    ~OpRow();

    char\* getName();

    unsigned short getOpcode();

    unsigned short getCycles();

    unsigned short getTotalCycles();

    unsigned short getTotalUsage();

    unsigned short incOpsCycle();

    void incOneCycle();

    void setRow(const char\* mnem, unsigned short cycles, unsigned short index);

    void printRowStats();

};

//Class optable

//==================================================================================

class optable

{

private:

    OpRow m\_opTable[OP\_TABLE\_LENG];

    //Opcode7List m\_op7;

    alu\* m\_alu;

    unsigned short m\_totalCycles;

    unsigned short m\_totalInstructions;

public:

    optable();

    optable(alu\* mem);

    ~optable();

    char\* getMnemonic(unsigned short index);

    unsigned short getIndex(char\* *name*);

    unsigned short getTotalOpsCycles(unsigned short index);

    unsigned short getTotalCycles();

    unsigned short getTotalUsed(char\* *name*);

    void incrementOpsCycle(unsigned short index);

    void incrementOneCycle(unsigned short index);

    void printUsageAndCycles();

};

#endif //\_\_OPTABLE\_H\_INCLUDED\_\_

/\* ==================================================================================

ECE 486 / Winter 2015 / PDP-8 simulator project

Team:

Deborah Denhart

Jeremiah Franke

==================================================================================

File:     OpTable.cpp

Date:     03/02/2015

Description:  This file contains the classes Opcode7inst, OpRow and

                optable

================================================================================== \*/

#include <cstdio>

#include <cstring>

#include "Common.h"

#include "alu.h"

#include "OpTable.h"

//External objects

//==================================================================================

extern ErrorTable Error;

//==================================================================================

//Description: constructor

//==================================================================================

OpRow::OpRow()

{

    m\_iTotalCycles = 0;

    m\_iTotalUsed = 0;

    m\_sMnemonic = *NULL*;

    m\_iCycles = 0;

}

//==================================================================================

//Description: initializes an opcode row with it's name, number of cycles and

//  opcode index

//==================================================================================

OpRow::OpRow(const char\* mnem, unsigned short cycles, unsigned short index)

{

    m\_iTotalCycles = 0;

    m\_iTotalUsed = 0;

    if(mnem)

    {

        int strsize = *strlen*(mnem)+1;

        m\_sMnemonic = new char[strsize];

*strcpy*(m\_sMnemonic, mnem);

    }

    else

    {

        m\_sMnemonic = *NULL*;

    }

    m\_iCycles = cycles;

    m\_rOpcode = index;

}

//==================================================================================

//Description: deconstructor

//==================================================================================

OpRow::~OpRow()

{

    clearOpRow();

}

//==================================================================================

//Description: clears all fields and deletes the string name

//==================================================================================

void OpRow::clearOpRow()

{

    m\_iTotalCycles = 0;

    m\_iTotalUsed = 0;

    if(m\_sMnemonic)

    {

        delete[] m\_sMnemonic;

        m\_sMnemonic = *NULL*;

    }

    m\_iCycles = 0;

    m\_rOpcode = 0;

}

//==================================================================================

//Description: returns the name of the opcode

//==================================================================================

char\* OpRow::getName()

{

    char\* mnem = *NULL*;

    int strsize = *strlen*(m\_sMnemonic)+1;

    mnem = new char[strsize];

*strcpy*(mnem, m\_sMnemonic);

    return mnem;

}

//==================================================================================

//Description: returns the opcode id

//==================================================================================

unsigned short OpRow::getOpcode()

{

    return m\_rOpcode;

}

//==================================================================================

//Description: returns the number of cycles necessary to run this instruction

//==================================================================================

unsigned short OpRow::getCycles()

{

    return m\_iCycles;

}

//==================================================================================

//Description: returns the total number of cycles ran for this instruction

//==================================================================================

unsigned short OpRow::getTotalCycles()

{

    return m\_iTotalCycles;

}

//==================================================================================

//Description: returns the total number of times this instruction has run

//==================================================================================

unsigned short OpRow::getTotalUsage()

{

    return m\_iTotalUsed;

}

//==================================================================================

//Description: increments the cycle count by one

//==================================================================================

void OpRow::incOneCycle()

{

    ++m\_iTotalCycles;

}

//==================================================================================

//Description: increments the cycles by number of cycles necessary to run

//  the instruction

//==================================================================================

unsigned short OpRow::incOpsCycle()

{

    m\_iTotalCycles += m\_iCycles;

    ++m\_iTotalUsed;

    return m\_iCycles;

}

//==================================================================================

//Description: resets a row's name, cycles to run and opcode id

//==================================================================================

void OpRow::setRow(const char\* mnem, unsigned short cycles, unsigned short index)

{

    m\_iTotalCycles = 0;

    if(m\_sMnemonic)

    {

        delete[] m\_sMnemonic;

        m\_sMnemonic = *NULL*;

    }

    if(mnem)

    {

        int strsize = *strlen*(mnem)+1;

        m\_sMnemonic = new char[strsize];

*strcpy*(m\_sMnemonic, mnem);

    }

    else

    {

        m\_sMnemonic = *NULL*;

    }

    m\_iCycles = cycles;

    m\_rOpcode = index;

}

//==================================================================================

//Description: prints the name, total times run and total cycles run

//==================================================================================

void OpRow::printRowStats()

{

*fprintf*(*stdout*, "%s:  executed: %d times  consumed: %d cycles\n",

            m\_sMnemonic, m\_iTotalUsed, m\_iTotalCycles);

}

//==================================================================================

//Description: constructor

//==================================================================================

optable::optable()

{

    const char\* temp1 = OPSTRING\_AND;

    const char\* temp2 = OPSTRING\_TAD;

    const char\* temp3 = OPSTRING\_ISZ;

    const char\* temp4 = OPSTRING\_DCA;

    const char\* temp5 = OPSTRING\_JMS;

    const char\* temp6 = OPSTRING\_JMP;

    const char\* temp7 = OPSTRING\_IO;

    const char\* temp8 = OPSTRING\_MICRO;

    m\_opTable[0].setRow(temp1, 2, 0);

    m\_opTable[1].setRow(temp2, 2, 1);

    m\_opTable[2].setRow(temp3, 2, 2);

    m\_opTable[3].setRow(temp4, 2, 3);

    m\_opTable[4].setRow(temp5, 2, 4);

    m\_opTable[5].setRow(temp6, 1, 5);

    m\_opTable[6].setRow(temp7, 0, 6);

    m\_opTable[7].setRow(temp8, 1, 7);

    m\_alu = *NULL*;

}

//==================================================================================

//Description: opcode constructor with accumulator pointer

//==================================================================================

optable::optable(alu\* mem)

{

    const char\* temp1 = OPSTRING\_AND;

    const char\* temp2 = OPSTRING\_TAD;

    const char\* temp3 = OPSTRING\_ISZ;

    const char\* temp4 = OPSTRING\_DCA;

    const char\* temp5 = OPSTRING\_JMS;

    const char\* temp6 = OPSTRING\_JMP;

    const char\* temp7 = OPSTRING\_IO;

    const char\* temp8 = OPSTRING\_MICRO;

    m\_opTable[0].setRow(temp1, 2, 0);

    m\_opTable[1].setRow(temp2, 2, 1);

    m\_opTable[2].setRow(temp3, 2, 2);

    m\_opTable[3].setRow(temp4, 2, 3);

    m\_opTable[4].setRow(temp5, 2, 4);

    m\_opTable[5].setRow(temp6, 1, 5);

    m\_opTable[6].setRow(temp7, 0, 6);

    m\_opTable[7].setRow(temp8, 1, 7);

    if(mem)

    {

        m\_alu = mem;

    }

    else

    {

        m\_alu = *NULL*;

    }

}

//==================================================================================

//Description: deconstructor

//==================================================================================

optable::~optable()

{

    if(m\_alu)

    {

        delete m\_alu;

        m\_alu = *NULL*;

    }

}

//==================================================================================

//Description: finds the id of the opcode from it's name

//==================================================================================

unsigned short optable::getIndex(char\* name)

{

    unsigned short iFoundIndex = OP\_TABLE\_LENG;

    char\* tablename = *NULL*;

    if(name)

    {

        for(int i = 0; i < OP\_TABLE\_LENG; ++i)

        {

            tablename = m\_opTable[i].getName();

            if(tablename)

            {

                if(!*strcmp*(tablename, name))

                {

                    iFoundIndex = i;

                    break;

                }

                delete[] tablename;

                tablename = *NULL*;

            }

            else

            {

                Error.printError(ERROR\_NULL, FILE\_OPTABLE);

            }

        }

    }

    else

    {

        Error.printError(ERROR\_NULL, FILE\_OPTABLE);

    }

    if(tablename)

    {

        delete[] tablename;

        tablename = *NULL*;

    }

    return iFoundIndex;

}

//==================================================================================

//Description: finds the name of an instruction from its id

//==================================================================================

char\* optable::getMnemonic(unsigned short index)

{

    char\* temp = *NULL*;

    if(index < OP\_TABLE\_LENG)

    {

        temp = m\_opTable[index].getName();

    }

    else

    {

        Error.printError(ERROR\_OUT\_OF\_RANGE, FILE\_OPTABLE);

    }

    return temp;

}

//==================================================================================

//Description: returns the total number of cycles for the input id

//==================================================================================

unsigned short optable::getTotalOpsCycles(unsigned short index)

{

    unsigned short temp = 0;

    if(index < OP\_TABLE\_LENG)

    {

        temp = m\_opTable[index].getTotalCycles();

    }

    else

    {

        Error.printError(ERROR\_OUT\_OF\_RANGE, FILE\_OPTABLE);

    }

    return temp;

}

//==================================================================================

//Description: returns the total number of cycles run

//==================================================================================

unsigned short optable::getTotalCycles()

{

    return m\_totalCycles;

}

//==================================================================================

//Description: returns the total number of times the input instruction has been used

//==================================================================================

unsigned short optable::getTotalUsed(char\* name)

{

    unsigned short index = 0;

    unsigned short used = 0;

    if(name)

    {

        index = getIndex(name);

        if(index < OP\_TABLE\_LENG)

        {

            used = m\_opTable[index].getTotalUsage();

        }

        else

        {

            Error.printError(ERROR\_OUT\_OF\_RANGE, FILE\_OPTABLE);

        }

    }

    else

    {

        Error.printError(ERROR\_NULL, FILE\_OPTABLE);

    }

    return used;

}

//==================================================================================

//Description: increase the number of cycles for the inputs id by it's default

//==================================================================================

void optable::incrementOpsCycle(unsigned short index)

{

    unsigned short count = 0;

    if(index < OP\_TABLE\_LENG)

    {

        count = m\_opTable[index].incOpsCycle();

        m\_totalCycles += count;

        ++m\_totalInstructions;

    }

    else

    {

        Error.printError(ERROR\_OUT\_OF\_RANGE, FILE\_OPTABLE);

    }

}

//==================================================================================

//Description: increments the opcode id by one cycle

//==================================================================================

void optable::incrementOneCycle(unsigned short index)

{

    if(index < OP\_TABLE\_LENG)

    {

        m\_opTable[index].incOneCycle();

        ++m\_totalCycles;

    }

    else

    {

        Error.printError(ERROR\_OUT\_OF\_RANGE, FILE\_OPTABLE);

    }

}

//==================================================================================

//Description: prints the total usage of all instructions

//==================================================================================

void optable::printUsageAndCycles()

{

*fprintf*(*stdout*, "\n%s\nStatistic Summary\n%s\n", PRINT\_BREAK, PRINT\_BREAK);

*fprintf*(*stdout*, "Total instructions: %d   Total clock cycles: %d \n%s\n",

            m\_totalInstructions, m\_totalCycles, PRINT\_BREAK);

    for(int i = 0; i < OP\_TABLE\_LENG; ++i)

    {

        m\_opTable[i].printRowStats();

    }

}

/\* ==================================================================================

ECE 486 / Winter 2015 / PDP-8 simulator project

Team:

Deborah Denhart

Jeremiah Franke

==================================================================================

File:     memarray.cpp

Date:     03/02/2015

Description:  This file contains the classes memReg and memarray

================================================================================== \*/

#ifndef \_\_memarray\_H\_INCLUDED\_\_

#define \_\_memarray\_H\_INCLUDED\_\_

//Dependencies

//==================================================================================

class *fstream*;

//Defines

//==================================================================================

//#define DEBUG\_MEM

#define MREAD 0

#define MWRITE 1

#define MFETCH 2

#define NONVAL -1

//Consts

//==================================================================================

const short MSIZE = 4096;

const char tracename[] = "memdump.dmp\0";

//Class memarray

//==================================================================================

class memarray

{

private:

    struct memReg

    {

        unsigned short value;

        bool access;

    };

    memReg mem[MSIZE];

    //Reading and writing memarray

    //to read memarray, a 12 bit address is loaded into the MA and

    //a read signal is asserted. The contents appear in the MB

    unsigned short rMA; //12 bit central processor memarray address register

    unsigned short rMB; //12 bit memarray Buffer register

    char\* traceFileName;

*FILE*\* tracefile;

*FILE*\* memfile;

    memarray(const memarray &source);

    void writeline();

    void readline();

    void logtrace(unsigned short *address*, short *type*);

    char\* getMemoryOp(short *type*);

public:

    memarray();

    ~memarray();

    void store(unsigned short addy, unsigned short value);

    void load(unsigned short addy);

    void fetch(unsigned short addy);

    bool getAccess(unsigned short addy);

    void checkValidAddy(unsigned short addy);

    void setAccess(unsigned short addy);

    void writeMemoryAccesses();

    unsigned short readMB();

    bool pcMemoryValid(unsigned short rpc);

};

#endif //\_\_memarray\_H\_INCLUDED\_\_

/\* ==================================================================================

ECE 486 / Winter 2015 / PDP-8 simulator project

Team:

Deborah Denhart

Jeremiah Franke

==================================================================================

File:     memarray.cpp

Date:     03/02/2015

Description:  This file contains the classes memReg and memarray

================================================================================== \*/

#include <fstream>

#include "Common.h"

#include "memory.h"

//External objects

//==================================================================================

extern ErrorTable Error;

extern void pauseandexit();

//==================================================================================

//Description: constructor

//==================================================================================

memarray::memarray()

{

    const char\* defaultfile = "memarrayTrace.dmp";

    int strsize = *strlen*(defaultfile) + 1;

    traceFileName = new char[strsize];

*strcpy*(traceFileName, defaultfile);

    tracefile = *fopen*(traceFileName, "a");

    for(short i = 0; i < MSIZE; ++i)

    {

        mem[i].value = 0;

        mem[i].access = 0;

    }

}

//==================================================================================

//Description: copy constructor

//==================================================================================

memarray::memarray(const memarray &source)

{

    int strsize = *strlen*(source.traceFileName) + 1;

    traceFileName = new char[strsize];

*strcpy*(traceFileName, source.traceFileName);

    tracefile = *fopen*(traceFileName, "a");

    for(short i = 0; i < MSIZE; ++i)

    {

        mem[i].value = source.mem[i].value;

        mem[i].access = source.mem[i].access;

    }

}

//==================================================================================

//Description: deconstructor

//==================================================================================

memarray::~memarray()

{

    if(tracefile)

    {

*fclose*(tracefile);

        tracefile = *NULL*;

    }

    if(traceFileName)

    {

        delete[] traceFileName;

        traceFileName = *NULL*;

    }

}

//==================================================================================

//Description: writes the value in MB to the address in MA in the memarray array

//==================================================================================

void memarray::writeline()

{

    checkValidAddy(rMA);

    checkValidAddy(rMB);

    mem[rMA].value = rMB; //write the memarray line

    mem[rMA].access = true; //change the access flag to true

}

//==================================================================================

//Description: saves data from MB to the memarray array at the address from MA

//==================================================================================

void memarray::readline()

{

    checkValidAddy(rMA);

    if(mem[rMA].access)

    {

        rMB = mem[rMA].value; //read the memarray line

    }

    /\*else

    {

        fprintf(stderr, "Error: Invalid Memory Access...\n ");

        fprintf(stderr, "Shutting down...\n ");

        pauseandexit();

    }\*/

}

//==================================================================================

//Description: writes all memarray lines that have been accessed to an external file

//==================================================================================

void memarray::writeMemoryAccesses()

{

    char filename[] = "pdp8\_Memory\_Accesses.txt";

    memfile = *fopen*(filename, "a");

    if(memfile)

    {

*fprintf*(memfile, "%s\n", PRINT\_BREAK);

*fprintf*(memfile, "PDP-8 Memory\n"); //print a header

*fprintf*(memfile, "%s\n", PRINT\_BREAK);

*fprintf*(memfile, "Address  Data\n");

*fprintf*(memfile, "%s\n", PRINT\_BREAK);

        for(short i = 0; i < MSIZE; ++i)

        {

            if(mem[i].access) //if the memarray has been used

            {

*fprintf*(memfile, "%o      %o\n", i, mem[i].value);  //print the address and value

            }

        }

*fclose*(memfile);

    }

    else

    {

        Error.printError(ERROR\_FILE, FILE\_MEM);

    }

}

//==================================================================================

//Description: writes an address to the MA and value to the MB and log it

//==================================================================================

void memarray::store(unsigned short address, unsigned short value)

{

    short type = MWRITE;

#ifdef DEBUG\_MEM

*fprintf*(*stdout*, "DEBUG: writing at address %o value %o\n", address, value);

#endif

    checkValidAddy(address); //check for valid address

    checkValidAddy(value);

    rMA = address; //set the address in the MA

    rMB = value; //set the data in the MB

    writeline(); //initiate the write

    logtrace(address, type); //log the memarray

}

//==================================================================================

//Description:loads an address to the MA, initiates a read and logs it

//==================================================================================

void memarray::load(unsigned short address)

{

    short type = MREAD;

#ifdef DEBUG\_MEM

*fprintf*(*stdout*, "DEBUG: reading address %o\n", address);

#endif

    checkValidAddy(address);  //check for valid address

    rMA = address; //write the address to the MA

    readline(); //initiate the memarray read

    logtrace(address, type); //log the memarray access

}

//==================================================================================

//Description: log a fetched address and store it in memarray

//==================================================================================

void memarray::fetch(unsigned short address)

{

    short type = MFETCH;

#ifdef DEBUG\_MEM

*fprintf*(*stdout*, "DEBUG: fetching address %o\n", address);

#endif

    checkValidAddy(address); //check for valid address

    logtrace(address, type); //log the memarray access

}

//==================================================================================

//Description: retrieves the access flag

//==================================================================================

bool memarray::getAccess(unsigned short addy)

{

    bool bRes = 0;

    checkValidAddy(addy);  //check for valid address

    bRes = mem[addy].access; //copy the flag

    return bRes;

}

//==================================================================================

//Description: checks if an address is within bounds of the memarray array

//==================================================================================

void memarray::checkValidAddy(unsigned short addy)

{

    if(MAX\_OCT\_ADDRESS < addy)

    {

*fprintf*(*stderr*, "ERROR: Invalid Address: %o\n", addy);

*fprintf*(*stderr*, "Shutting down...\n ");

        pauseandexit();

    }

}

//==================================================================================

//Inputs: unsigned short pointer holding the relevant address

//==================================================================================

void memarray::setAccess(unsigned short addy)

{

    checkValidAddy(addy);

    mem[addy].access = true;

}

//==================================================================================

//Description: prints reads, writes or fetches to an external file

//==================================================================================

void memarray::logtrace(unsigned short address, short type)

{

    checkValidAddy(address);

    if(tracefile)

    {

        //write the type and address to the file

*fprintf*(tracefile, "%d   %o\n", type, address);

    }

    else

    {

        Error.printError(ERROR\_FILE, FILE\_MEM);

    }

}

//==================================================================================

//Description: reads the memarray buffer register

//==================================================================================

unsigned short memarray::readMB()

{

    return rMB;

}

//==================================================================================

//Description: gets the string name for the memarray operation type

//==================================================================================

char\* memarray::getMemoryOp(short type)

{

    const char\* temp1 = "READ";

    const char\* temp2 = "WRITE";

    const char\* temp3 = "FETCH";

    const char\* temp4 = "NO VALUE";

    char\* temp = *NULL*;

    int strsize = 0;

    if(MREAD == type)

    {

        strsize = *strlen*(temp1)+1;

        temp = new char[strsize];

*strcpy*(temp, temp1);

    }

    else if(MWRITE == type)

    {

        strsize = *strlen*(temp2)+1;

        temp = new char[strsize];

*strcpy*(temp, temp2);

    }

    else if(MFETCH == type)

    {

        strsize = *strlen*(temp3)+1;

        temp = new char[strsize];

*strcpy*(temp, temp3);

    }

    else

    {

        strsize = *strlen*(temp4)+1;

        temp = new char[strsize];

*strcpy*(temp, temp4);

    }

    return temp;

}

/\* ==================================================================================

ECE 486 / Winter 2015 / PDP-8 simulator project

Team:

Deborah Denhart

Jeremiah Franke

==================================================================================

File:     Main.cpp

Date:     03/02/2015

Description:  This file contains the main

================================================================================== \*/

#include <fstream>

#include <iostream>

#include <cstdio>

#include "common.h"

#include "controlunit.h"

class ErrorTable Error;

void pauseandexit();

int *main*(int argc, char \*\*argv)

{

    const char\* memdump = "pdp8log.txt\0";

    //bool silent = false; //pass to error class

    //bool bDebugOn = false;

    //bool bHelp = false;

    bool bHex = false;

    bool bOct = false;

    bool bBin = false;

    bool bAbort = false;

    ControlUnit controlUnit;

    char\* infile = *NULL*;

    char\* outfile = *NULL*;

    short mode = 0;

    int strsize = 0;

    /\* Read command line arguments

    flags:

    [-h <help>]

    [-f <trace input file name>]

    [-d <debug mode>]

    [-m <memarray output file name>]

    [-v | -o | -b <ASCII hexadecimal format> | <ASCII octal format> | <binary object file> ]

    \*/

    for(short i = 0; i < argc; ++i)

    {

        //if(!strcmp(argv[i], "-h"))

        //{

        //    fprintf(stdout, "Usage: ");

        //}

        //else

        if(!*strcmp*(argv[i], "-f"))

        {

            //bInputFile = true;

            //get the next argument for the file name

            ++i;

            if(argv[i][0] == '-')

            {

                //missing file name

                bAbort = true;

            }

            else

            {

                strsize = *strlen*(argv[i])+1;

                infile = new char[strsize];

*strcpy*(infile, argv[i]);

            }

        }

        else if(!*strcmp*(argv[i], "-v"))

        {

            bHex = true;

        }

        else if(!*strcmp*(argv[i], "-o"))

        {

            bOct = true;

        }

    }

    if(infile)

    {

        if((bHex && !bOct && !bBin) ||

           (!bHex && bOct && !bBin))

        {

            if(bHex)

            {

                mode = INPUT\_HEX;

            }

            else if(bOct)

            {

                mode = INPUT\_OCTAL;

            }

        }

        else if(!bHex && !bOct && !bBin)

        {

            bBin = true;

            mode = INPUT\_BINARY;

        }

        else

        {

            //error output: -o and -v can't both be set

            bAbort = true;

*fprintf*(*stdout*, "Error: input can't have both --o and --v flags set\n");

*fprintf*(*stderr*, "Shutting down...\n ");

            pauseandexit();

        }

        strsize = *strlen*(memdump)+1;

        outfile = new char[strsize];

*strcpy*(outfile, memdump);

    }

    else

    {

        bAbort = true;

    }

    if(!bAbort)

    {

        controlUnit.loadFile(infile, mode);

    }

    if(infile)

    {

        delete[] infile;

        infile = *NULL*;

    }

    if(outfile)

    {

        delete[] outfile;

        outfile = *NULL*;

    }

*fprintf*(*stderr*, "Halt...\n ");

    pauseandexit();

    return 0;

}

void pauseandexit()

{

    //fprintf(stderr, "Shutting down...\n ");

*std*::*cin*.*sync*(); // Flush The Input Buffer Just In Case

*std*::*cin*.*ignore*(); // There's No Need To Actually Store The Users Input

*exit*(*EXIT\_FAILURE*);

}

/\* ==================================================================================

ECE 486 / Winter 2015 / PDP-8 simulator project

Team:

Deborah Denhart

Jeremiah Franke

==================================================================================

File:     controlunit.cpp

Date:     03/02/2015

Description:  This file contains the classes InstFormat, EffectiveAddress,

ControlUnit and Statistics

================================================================================== \*/

#ifndef \_\_CONTROLUNIT\_H\_INCLUDED\_\_

#define \_\_CONTROLUNIT\_H\_INCLUDED\_\_

//Dependencies

//==================================================================================

class optable;

class memarray;

class alu;

//Defines

//==================================================================================

//#define DEBUG\_CONTROL

//Class InstFormat

//==================================================================================

class InstFormat

{

private:

    unsigned short rIR;  //3 bit instruction register, holds opcode of current instruction

    unsigned short m\_rInstruction;

    bool m\_bMRI; //memarray reference instruction format, opcodes 0-5

    bool m\_bOperate; //instructions that operate on alu and link, opcode 7

    bool m\_bTestIO; //input/output instructions, opcode 6

    optable\* m\_opTable;

    unsigned short m\_iMicroCode;

    void setInstType();

    void setIR(unsigned short inst);

public:

    InstFormat();

    ~InstFormat();

    void reset(); //resets the format boolean types

    void loadInstruction(unsigned short inst);

    char\* getInstType(); //returns the type of format in a string

    bool isInstMRI();

    bool isInstOperate();

    bool isInstTestIO();

    //unsigned short getIOdeviceNum(); //TestIO format, 6 bit device number

    //unsigned short getIOfcn(); //TestIO format, 3 bit fcn

    unsigned short getOpcode();

    unsigned short getInstruction();

    char\* getInstructionName(unsigned short opcode);

    void incrementCycles(unsigned short  ops);

    void incrementCyclesDefer(unsigned short  ops);

    void incrementInstUsage(unsigned short opcode);

    void printStats();

};

//Class EffectiveAddress

//==================================================================================

class EffectiveAddress

{

private:

    unsigned short m\_rpc; //copy of the current pc

    unsigned short m\_rOffset; //7 bit offset

    unsigned short m\_rZeroPage; //5 bit, the first memarray page

    unsigned short m\_rCurrPage;  //5 bit, the current page

    unsigned short m\_rResultAddress; //12 bit resulting address

    bool m\_indirect;

    memarray \*m\_pmem;

    void setPCOffset(unsigned short rpc);

    unsigned short effAdzeroPage();  //if the memarray page bit is 0, calculate 00000 + offset, range is 0000o-0177o

    unsigned short effAdcurrentPage(); //if the memarray page bit is 1, calculate instruction page + offset, range instruction page

    unsigned short effAdindirectAddressZero(); //if the indirect bit is 1, calculate  c(address), range is any address but costs 2 mem reads

    unsigned short effAdindirectAddressCurr(); //if the indirect bit is 1, calculate  c(address), range is any address but costs 2 mem reads

    void loadOffset(unsigned short reg); //extracts the offset from an MRI instruction

    unsigned short readAddress(unsigned short address); //reads the new address from memory

public:

    EffectiveAddress();

    ~EffectiveAddress();

    //returns 12 bit address of a memarray location from an instruction

    unsigned short geteffAddress(unsigned short reg, unsigned short rpc);

    void setmem(memarray\* mem);

    bool isIndirect();

};

//Class ControlUnit

//==================================================================================

class ControlUnit

{

private:

    unsigned short rPC; //12 bit program counter, holds address next instruction

    //unsigned short rSR;  //12 bit console switch register, used to load memarray from console

    bool silent;

    bool running;

*FILE*\* readfile;

    unsigned short m\_StartAddress;

    char m\_symbolTable[SYMBOLS\_MAX];

    InstFormat m\_format;

    EffectiveAddress m\_eAddy;

    memarray\* m\_mem;

    alu\* m\_alu;

    unsigned short getPC();

    void setPC(unsigned short addy);

    unsigned short hexAddressHandle(char\* addy);

    void modeHex(char\* filename);

    void modeBin(char\* filename);

    void modeOct(char\* filename);

    void instructionDefer(); //used to add extra cycles for indirection

    unsigned short readData(unsigned short address);

    void initSymbols();

    short convToNumber(char sIn);

public:

    ControlUnit();

    ~ControlUnit();

    void instructionFetch();

    void instructionDecode();

    void executeMicro(unsigned short inst);

    void executeMRI(unsigned short addy, unsigned short data);

    void loadFile(char\* in, short mode);

    void printMemoryHistory();

    void incrementPC();

};

#endif //\_\_CONTROLUNIT\_H\_INCLUDED\_\_

/\* ==================================================================================

ECE 486 / Winter 2015 / PDP-8 simulator project

Team:

Deborah Denhart

Jeremiah Franke

==================================================================================

File:     controlunit.cpp

Date:     03/02/2015

Description:  This file contains the classes InstFormat, EffectiveAddress and

ControlUnit

================================================================================== \*/

#include <fstream>

#include <cstdio>

#include <cstring>

#include <vector>

#include "Common.h"

#include "OpTable.h"

#include "memory.h"

#include "controlunit.h"

#include "alu.h"

//External objects

//==================================================================================

extern ErrorTable Error;

extern void pauseandexit();

//==================================================================================

//Description:constructor

//==================================================================================

ControlUnit::ControlUnit()

{

    silent = false;

    readfile = *NULL*;

    m\_alu = new alu;

    m\_mem = new memarray;

    m\_eAddy.setmem(m\_mem);

    m\_StartAddress = 0200;

    running = true;

    initSymbols();

}

//==================================================================================

//Description:deconstructor

//==================================================================================

ControlUnit::~ControlUnit()

{

    //silent = false;

    if(m\_mem)

    {

        delete m\_mem;

        m\_mem = *NULL*;

    }

    if(m\_alu)

    {

        delete m\_alu;

        m\_alu = *NULL*;

    }

}

//==================================================================================

//Description: Converts a 3 char hex value to a 4

//==================================================================================

unsigned short ControlUnit::hexAddressHandle(char\* addy)

{

    unsigned short reg = 0;

    for(int i = 0; i < ADDRESS\_LENGTH\_HEX; ++i)

    {

        reg = reg << 4;

        reg += convToNumber(addy[i]);

    }

#ifdef DEBUG\_CONTROL

    //fprintf(stdout, "DEBUG: convert hex input %s string to octal: %o\n", addy, reg);

#endif

    return reg;

}

//==================================================================================

//Description: parses a hex file

//==================================================================================

void ControlUnit::modeHex(char\* filename)

{

    char line[MAX\_BUFFER]; // String for getline

    int length = 0;

    bool bFirstLine = true;

    unsigned short reg = 0;

    unsigned short rpc = 0;

    if(*NULL* != filename)

    {

        readfile = *fopen*(filename, "r");

    }

    else

    {

*fprintf*(*stderr*, "Error: can't open %s\n", filename);

*fprintf*(*stderr*, "Shutting down...\n ");

        pauseandexit();

    }

    if(readfile)

    {

        while((!*feof*(readfile)) && (*fgets*(line, MAX\_BUFFER, readfile)))

        {

            length = *strlen*(line);

            if(length == (ADDRESS\_LENGTH\_HEX+1))

            {

                //it's data

                reg = hexAddressHandle(line);

                rpc = getPC();

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: read hex addy: %o  data: %o\n", rpc, reg);

#endif

                m\_mem->store(rpc, reg);

                incrementPC();

            }

            else if(length == (ADDRESS\_LENGTH\_OCT+1))

            {

                if('@' == line[0]) //address

                {

*memmove*(line, line+1, *strlen*(line)); //remove @

                    reg = hexAddressHandle(line);

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: read hex address: %o\n", reg);

#endif

                    if(bFirstLine)

                    {

                        m\_StartAddress = reg;

                    }

                    bFirstLine = false;

                    setPC(reg);

                }

                else

                {

                    Error.printError(ERROR\_UNEXPECTED\_VALUE, FILE\_CONTROL);

                }

            }

            else if(length == ADDRESS\_LENGTH\_HEX)

            {

                //it's data

                char tempstr[(ADDRESS\_LENGTH\_HEX+1)];

                tempstr[0] = '0';

                tempstr[1] = line[0];

                tempstr[2] = line[1];

                tempstr[3] = line[2];

                reg = hexAddressHandle(tempstr);

                rpc = getPC();

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: read hex addy: %o  data: %o\n", rpc, reg);

#endif

                m\_mem->store(rpc, reg);

                incrementPC();

            }

            else

            {

                Error.printError(ERROR\_OUT\_OF\_RANGE, FILE\_CONTROL);

            }

            reg = 0;

        }

*fclose*(readfile);

    }

    else

    {

*fprintf*(*stderr*, "Error: can't open %s\n", filename);

*fprintf*(*stderr*, "Shutting down...\n ");

        pauseandexit();

    }

}

//==================================================================================

//Description: parses a binary file

//==================================================================================

void ControlUnit::modeBin(char\* filename)

{

    bool bFirstLine = true;

    bool bufferfound = false;

    unsigned short addyfound = 0;

    unsigned short buff = 0;

    unsigned short rpc = 0;

    unsigned short conv = 0;

    unsigned char\* buffer = *NULL*;

*std*::*vector*<char> sBuff;

*std*::*vector*<char>::*iterator* it;

    if(*NULL* != filename)

    {

       readfile = *fopen*(filename, "r+b");

    }

    else

    {

*fprintf*(*stderr*, "Error: can't open %s\n", filename);

*fprintf*(*stderr*, "Shutting down...\n ");

        pauseandexit();

    }

    if(readfile)

    {

        //get the size of the file

*fseek*(readfile, 0, *SEEK\_END*);

        long fsize = *ftell*(readfile);

        buffer = new unsigned char[fsize+1];

*rewind*(readfile);

        //read the file

*fread*(buffer, 1, fsize, readfile);

*fclose*(readfile);

        buffer[fsize] = '\0';

        //get rid of headers

        for(short i = 0; i < fsize; ++i)

        {

            if(buffer[i] == 128)

            {

                //skip this character.. it's a buffer

            }

            else if((buffer[i] == 128) && (bufferfound))

            {

                break; //stop at the bell

            }

            else

            {

                sBuff.*push\_back*(buffer[i]);

                bufferfound = true;

            }

        }

        //parse the binary

        for(it = sBuff.*begin*(); it != sBuff.*end*(); ++it)

        {

            for(int i = 0; i < 2; ++i)

            {

                conv = \*it;

                if(0 == i)

                {

                    addyfound = conv & BIT5\_MASK;

                    addyfound = addyfound >> 6;

                    buff = conv & REG\_6BIT\_MASK;

                    buff = buff << REG\_UPPER;

                    ++it;

                    if(sBuff.*end*() == it)

                    {

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: invalid binary format: %o\n", buff);

#endif

                        --it;

                        break;

                    }

                }

                else

                {

                    conv = conv & REG\_6BIT\_MASK;

                    buff += conv;

                }

            }

            if(addyfound) //its an address

            {

                m\_mem->checkValidAddy(buff);

                if(bFirstLine)

                {

                    m\_StartAddress = buff;

                }

                bFirstLine = false;

                setPC(buff);

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: read binary addy: %o  data: %o\n", rpc, buff);

#endif

            }

            else //its data

            {

                rpc = getPC();

                m\_mem->store(rpc, buff);

                incrementPC();

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: read binary address: %o\n", buff);

#endif

            }

            buff = 0;

        }

    }

    else

    {

*fprintf*(*stderr*, "Error: can't open %s\n", filename);

*fprintf*(*stderr*, "Shutting down...\n ");

        pauseandexit();

    }

}

//==================================================================================

//Description:  parses an octal file

//==================================================================================

void ControlUnit::modeOct(char\* filename)

{

    char line[MAX\_BUFFER]; // String for getline

    bool bAddy = false;

    bool bFirstLine = true;

    int pair = 0;

    unsigned short conv = 0;

    unsigned short buff = 0;

    unsigned short rpc = 0;

    short length = 0;

*std*::*vector*<char> sBuff;

*std*::*vector*<char>::*iterator* it;

    readfile = *fopen*(filename, "r");

    if(readfile)

    {

        while(!*feof*(readfile) && *fgets*(line, MAX\_BUFFER, readfile))

        {

            length = *strlen*(line);

            if(length <= ADDRESS\_LENGTH\_OCT) // expected format including \0

            {

                if(0 == pair)

                {

                    if('1' == line[0])

                    {

                        bAddy = true;

                    }

                    else if('0' == line[0])

                    {

                        bAddy = false;

                    }

                    else

                    {

                        Error.printError(ERROR\_UNEXPECTED\_VALUE, FILE\_CONTROL);

                    }

                }

                ++pair;

                if(length == ADDRESS\_LENGTH\_OCT)

                {

                    sBuff.*push\_back*(line[1]);

                    sBuff.*push\_back*(line[2]);

                }

                else if(length == (ADDRESS\_LENGTH\_OCT-1))

                {

                    sBuff.*push\_back*('0');

                    sBuff.*push\_back*(line[2]);

                }

                else

                {

                    Error.printError(ERROR\_OUT\_OF\_RANGE, FILE\_CONTROL);

                }

            }

            else

            {

                Error.printError(ERROR\_OUT\_OF\_RANGE, FILE\_CONTROL);

            }

            if(2 == pair)

            {

                //parse the binary

                for(it = sBuff.*begin*(); it != sBuff.*end*(); ++it)

                {

                    buff = buff << 3;

                    conv = convToNumber(\*it);

                    buff += conv;

                }

                if(bAddy) //it's an address

                {

                    m\_mem->checkValidAddy(buff);  //check for valid address

                    if(bFirstLine)

                    {

                        m\_StartAddress = buff;

                    }

                    bFirstLine = false;

                    setPC(buff);

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: read octal address: %o\n", buff);

#endif

                }

                else //it's data

                {

                    rpc = getPC();

                    m\_mem->store(rpc, buff);

                    incrementPC();

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: read octal addy: %o  data: %o\n", rpc, buff);

#endif

                }

                    pair = 0;

                    buff = 0;

                    sBuff.*clear*();

            }

        }

*fclose*(readfile);

    }

    else

    {

*fprintf*(*stderr*, "Error: can't open %s\n", filename);

*fprintf*(*stderr*, "Shutting down...\n ");

        pauseandexit();

    }

}

//==================================================================================

//Description: reads the new address from memory

//==================================================================================

unsigned short ControlUnit::readData(unsigned short address)

{

    m\_mem->load(address);

    unsigned short temp = m\_mem->readMB();

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: loading address: %o data: %o\n", address, temp);

#endif

    return temp;

}

//==================================================================================

//Description: fetches an instruction from memory located at rPC and increments rPC

//==================================================================================

void ControlUnit::instructionFetch()

{

    //contents of the PC are loaded short to the MA

    unsigned short reg = 0;

    m\_mem->load(rPC);

    reg = m\_mem->readMB();

    m\_mem->fetch(rPC);

    m\_format.loadInstruction(reg);

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: fetched data: %o\n", reg);

#endif

}

//==================================================================================

//Description: muxes the opcode, retrieves indirect addresses if necessary and

// sends results to be executed

//==================================================================================

void ControlUnit::instructionDecode()

{

    unsigned short ops = 0;

    unsigned short addy = 0;

    unsigned short data = 0;

    unsigned short currInst = 0;

    ops = m\_format.getOpcode();

    //up cycle count

    m\_format.incrementCycles(ops);

    currInst = m\_format.getInstruction();

    if(m\_format.isInstMRI())

    {

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: MRI decode address: %o, data: %o\n", addy, data);

#endif

        addy = m\_eAddy.geteffAddress(currInst, rPC); //get the address

        instructionDefer();

        data = readData(addy);

        executeMRI(addy, data);

    }

    else if(m\_format.isInstOperate())

    {

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: micro decode opcode: %o\n", currInst);

#endif

        //micro setup

        executeMicro(currInst);

    }

    else if(m\_format.isInstTestIO())

    {

        //IO setup

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: IO decode: nops\n");

#endif

        incrementPC();

    }

    else

    {

        Error.printError(ERROR\_UNEXPECTED\_VALUE, FILE\_CONTROL);

    }

}

//==================================================================================

//Description: checks if the instruction is indirect and retrieves the new address,

// incrementing the number of necessary cycles

//==================================================================================

void ControlUnit::instructionDefer()

{

    unsigned short ops = 0;

    //if indirect, costs 2 cycles

    if(m\_eAddy.isIndirect())

    {

        ops = m\_format.getOpcode();

        //up cycle count

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: defer: opcode %s\n", m\_format.getInstructionName(ops));

#endif

        m\_format.incrementCyclesDefer(ops);

    }

}

//==================================================================================

//Description: returns the value of rPC

//==================================================================================

unsigned short ControlUnit::getPC()

{

    return rPC;

}

//==================================================================================

//Description: checks addy to make sure it is valid and sets rPC to addy

//==================================================================================

void ControlUnit::setPC(unsigned short addy)

{

    m\_mem->checkValidAddy(addy);

    rPC = addy;

}

//==================================================================================

//Description: executes an MRI instruction

//==================================================================================

void ControlUnit::executeMRI(unsigned short addy, unsigned short data)

{

    unsigned short temp = 0;

    unsigned short opcode = m\_format.getOpcode();

    bool skipIncrement = false;

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: executing %s\n",  m\_format.getInstructionName(opcode));

#endif

    if(OPCODE\_AND == opcode)

    {

        m\_alu->andALU(data);

    }

    else if(OPCODE\_TAD == opcode)

    {

        m\_alu->sumALU(data);

    }

    else if(OPCODE\_ISZ == opcode)

    {

        temp = data + 1; //increment the data

        m\_mem->store(addy, temp); //store back in memory

        if(temp == 0)

        {

            incrementPC();

        }

    }

    else if(OPCODE\_DCA == opcode)

    {

        //store result of alu

        m\_mem->store(addy, m\_alu->getAC());

        m\_alu->*clear*();

    }

    else if(OPCODE\_JMS == opcode)

    {

        m\_mem->store(addy, rPC); //store the pc in the current address

        addy += 1; //increment the address

        m\_mem->checkValidAddy(addy);

        setPC(addy); //set pc to the new addy

    }

    else if(OPCODE\_JMP == opcode)

    {

        m\_mem->checkValidAddy(addy);

        setPC(addy); // set pc to the new addy

        skipIncrement = true;

    }

    else

    {

        Error.printError(ERROR\_UNEXPECTED\_VALUE, FILE\_CONTROL);

    }

    if(!skipIncrement)

    {

        incrementPC();

    }

}

//==================================================================================

//Description: executes the micro instructions

//==================================================================================

void ControlUnit::executeMicro(unsigned short inst)

{

    unsigned short bit3 = inst & BIT3\_MASK;

    bit3 = bit3 >> 8;

    unsigned short bit4 = inst & BIT4\_MASK;

    bit4 = bit4 >> 7;

    unsigned short bit5 = inst & BIT5\_MASK;

    bit5 = bit5 >> 6;

    unsigned short bit6 = inst & BIT6\_MASK;

    bit6 = bit6 >> 5;

    unsigned short bit7 = inst & BIT7\_MASK;

    bit7 = bit7 >> 4;

    unsigned short bit8 = inst & BIT8\_MASK;

    bit8 = bit8 >> 3;

    unsigned short bit9 = inst & BIT9\_MASK;

    bit9 = bit9 >> 2;

    unsigned short bit10 = inst & BIT10\_MASK;

    bit10 = bit10 >> 1;

    unsigned short bit11 = inst & BIT11\_MASK;

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: executing micro instruction: %o\n", inst);

#endif

    if(bit3 == 0)  //Group 1

    {

        if(bit4 == 1)

        {

            m\_alu->clearAC();

        }

        if(bit5 == 1)

        {

            m\_alu->clearLC();

        }

        if(bit6 == 1)

        {

            m\_alu->complementALU();

        }

        if(bit7 == 1)

        {

            m\_alu->complementLC();

            if(bit11 == 1)

                m\_alu->incrementALU();

        }

        if(bit8 == 1)

        {

            m\_alu->rotateRight();

            if(bit10 == 1)

                m\_alu->rotateRight();

        }

        if(bit9 == 1)

        {

            m\_alu->rotateLeft();

            if(bit10 == 1)

                m\_alu->rotateLeft();

        }

    }

    else

    {

        if(bit8 == 0)

        {

            if((bit5 == 1) || (bit6 == 1) || (bit7 == 1))

            {

                if(bit5 == 1)

                {

                    if(m\_alu->isNegative())

                        incrementPC();

                }

                if(bit6 == 1)

                {

                    if(m\_alu->isZero())

                        incrementPC();

                }

                if(bit7 == 1)

                {

                    if(m\_alu->getLB() != 0)

                        incrementPC();

                }

            }

        }

        else

        {

            short skip = false;

            /\*if ( ( (bit5 == 1) && (bit6 == 1) )

                || ( (bit5 == 1) && (bit7 == 1) )

                || ( (bit6 == 1) && (bit7 == 1) ) )

            {

                incrementPC();

            }\*/

            if(bit5 == 1)

            {

                skip = false;

                if(!m\_alu->isNegative())

                {

                    skip = true;

                }

            }

            if(bit6 == 1)

            {

                skip = false;

                if(!m\_alu->isZero())

                {

                    skip = true;

                }

            }

            if(bit7 == 1)

            {

                skip = false;

                if(m\_alu->getLB() == 0)

                {

                    skip = true;

                }

            }

            if(skip)

            {

                incrementPC();

            }

            if((bit4 == 0) && (bit5 == 0)

               && (bit6 == 0) && (bit7 == 0)

               && (bit8 == 1) && (bit9 == 0)

               && (bit10 == 0) && (bit11 == 0))

            {

                incrementPC();

            }

            if(bit4 == 1)

            {

                m\_alu->clearAC();

                running = false;

            }

        }

    }

    incrementPC();

}

//==================================================================================

//Description: loads the input file and parses flags

//==================================================================================

void ControlUnit::loadFile(char\* filename, short mode)

{

    if(INPUT\_BINARY == mode)

    {

        modeBin(filename);

    }

    else if(INPUT\_HEX == mode)

    {

        modeHex(filename);

    }

    else if(INPUT\_OCTAL == mode)

    {

        modeOct(filename);

    }

    else

    {

*fprintf*(*stderr*, "Error: Unknown File Format...\n ");

*fprintf*(*stderr*, "Shutting down...\n ");

        pauseandexit();

    }

    m\_mem->writeMemoryAccesses();

    setPC(m\_StartAddress);

    while(HALT\_CODE != m\_mem->readMB())

    {

        instructionFetch();

        instructionDecode();

    }

    m\_format.printStats();

    m\_mem->writeMemoryAccesses();

}

//==================================================================================

//Description: writes all memarray accesses to file

//==================================================================================

void ControlUnit::printMemoryHistory()

{

    if(m\_mem)

    {

        m\_mem->writeMemoryAccesses();

    }

    else

    {

        Error.printError(ERROR\_NULL, FILE\_CONTROL);

    }

}

//==================================================================================

//Description: checks addy to make sure it is valid and sets rPC to addy

//==================================================================================

void ControlUnit::incrementPC()

{

    ++rPC;

    m\_mem->checkValidAddy(rPC);

}

//==================================================================================

//Description: initialize symbol table

//==================================================================================

void ControlUnit::initSymbols()

{

    m\_symbolTable[0] = '0';

    m\_symbolTable[1] = '1';

    m\_symbolTable[2] = '2';

    m\_symbolTable[3] = '3';

    m\_symbolTable[4] = '4';

    m\_symbolTable[5] = '5';

    m\_symbolTable[6] = '6';

    m\_symbolTable[7] = '7';

    m\_symbolTable[8] = '8';

    m\_symbolTable[9] = '9';

    m\_symbolTable[10] = 'A';

    m\_symbolTable[11] = 'B';

    m\_symbolTable[12] = 'C';

    m\_symbolTable[13] = 'D';

    m\_symbolTable[14] = 'E';

    m\_symbolTable[15] = 'F';

}

//==================================================================================

//Description: finds a hex equivalent for the input symbolic char

//==================================================================================

short ControlUnit::convToNumber(char sIn)

{

    short i = 0;

    short res = -1;

    char conv = *toupper*(sIn);

    for(i = 0; i < SYMBOLS\_MAX; ++i)

    {

        if(conv == m\_symbolTable[i])

        {

            res = i;

            break;

        }

    }

    return res;

}

//==================================================================================

//Description: constructor

//==================================================================================

InstFormat::InstFormat()

{

    m\_opTable = new optable;

    reset();

}

//==================================================================================

//Description:deconstructor

//==================================================================================

InstFormat::~InstFormat()

{

    if(m\_opTable)

    {

        delete m\_opTable;

        m\_opTable = *NULL*;

    }

}

//==================================================================================

//Description: resets all saved addresses and data

//==================================================================================

void InstFormat::reset()

{

    m\_bMRI = false;

    m\_bOperate = false;

    m\_bTestIO = false;

    m\_iMicroCode = 0;

}

//==================================================================================

//Description:extracts the opcode from the instruction and sets it to the rIR

//==================================================================================

void InstFormat::setIR(unsigned short inst)

{

    rIR = inst & IR\_MASK;

    rIR = rIR >> 9;

}

//==================================================================================

//Description: returns the contents of the IR register

//==================================================================================

unsigned short InstFormat::getOpcode()

{

    return rIR;

}

//==================================================================================

//Description: sets the type of instruction format

//==================================================================================

void InstFormat::setInstType()

{

    if(OPCODE\_JMP >= rIR)

    {

        m\_bMRI = true;

        m\_bTestIO = false;

        m\_bOperate = false;

    }

    else if(OPCODE\_IO == rIR)

    {

        m\_bMRI = false;

        m\_bTestIO = true;

        m\_bOperate = false;

    }

    else if(OPCODE\_MICRO == rIR)

    {

        m\_bMRI = false;

        m\_bTestIO = false;

        m\_bOperate = true;

    }

    else

    {

        Error.printError(ERROR\_UNEXPECTED\_VALUE, FILE\_CONTROL);

    }

}

//==================================================================================

//Description:loads the current instruction

//==================================================================================

void InstFormat::loadInstruction(unsigned short inst)

{

    reset();

    m\_rInstruction = inst;

    setIR(inst);

    setInstType();

    m\_iMicroCode = inst;

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: format loading instruction: %o\n", inst);

#endif

}

//==================================================================================

//Description:returns the type of instruction in a string

//==================================================================================

char\* InstFormat::getInstructionName(unsigned short opcode)

{

    return m\_opTable->getMnemonic(opcode);

}

//==================================================================================

//Description:returns the type of format in a string

//==================================================================================

char\* InstFormat::getInstType()

{

    char\* temp = *NULL*;

    char stemp1[] = "Memory Reference\0";

    char stemp2[] = "Micro Instruction\0";

    char stemp3[] = "Test I/O\0";

    int strsize = 0;

    if(m\_bMRI)

    {

        strsize = *strlen*(stemp1)+1;

        temp = new char[strsize];

*strcpy*(temp, stemp1);

    }

    else if(m\_bOperate)

    {

        strsize = *strlen*(stemp2)+1;

        temp = new char[strsize];

*strcpy*(temp, stemp2);

    }

    else if(m\_bTestIO)

    {

        strsize = *strlen*(stemp3)+1;

        temp = new char[strsize];

*strcpy*(temp, stemp3);

    }

    else

    {

        Error.printError(ERROR\_UNEXPECTED\_VALUE, FILE\_CONTROL);

    }

    return temp;

}

//==================================================================================

//Description:returns the micro instruction

//==================================================================================

unsigned short InstFormat::getInstruction()

{

    return m\_iMicroCode;

}

//==================================================================================

//Description:checks if instruction is a MRI format

//==================================================================================

bool InstFormat::isInstMRI()

{

    return m\_bMRI;

}

//==================================================================================

//Description:checks if instruction is a OPP format

//==================================================================================

bool InstFormat::isInstOperate()

{

    return m\_bOperate;

}

//==================================================================================

//Description:checks if instruction is an IO format

//==================================================================================

bool InstFormat::isInstTestIO()

{

    return m\_bTestIO;

}

//==================================================================================

//Description: increments cycles for MRI

//==================================================================================

void InstFormat::incrementCycles(unsigned short ops)

{

    if(ops < OP\_TABLE\_LENG)

    {

        m\_opTable->incrementOpsCycle(ops);

    }

    else

    {

        Error.printError(ERROR\_OUT\_OF\_RANGE, FILE\_CONTROL);

    }

}

//==================================================================================

//Description: increments a cycle if the instruction is deferred

//==================================================================================

void InstFormat::incrementCyclesDefer(unsigned short ops)

{

    if(ops < OP\_TABLE\_LENG)

    {

        m\_opTable->incrementOneCycle(ops);

    }

    else

    {

        Error.printError(ERROR\_OUT\_OF\_RANGE, FILE\_CONTROL);

    }

}

//==================================================================================

//Description:prints the stat summary for all instructions

//==================================================================================

void InstFormat::printStats()

{

    m\_opTable->printUsageAndCycles();

}

//==================================================================================

//Description:constructor

//==================================================================================

EffectiveAddress::EffectiveAddress()

{

    m\_rpc = 0; //copy of the current pc

    m\_rOffset = 0; //7 bit offset

    m\_rZeroPage = 0; //5 bit, the first memarray page

    m\_rCurrPage = 0;  //5 bit, the current page

    m\_rResultAddress = 0; //12 bit resulting address

    m\_indirect = false;

    m\_pmem = *NULL*;

}

//==================================================================================

//Description: deconstructor

//==================================================================================

EffectiveAddress::~EffectiveAddress()

{

}

//==================================================================================

//Description: sets the memory pointer

//==================================================================================

void EffectiveAddress::setmem(memarray\* mem)

{

    if(mem)

    {

        m\_pmem = mem;

    }

    else

    {

        Error.printError(ERROR\_NULL, FILE\_CONTROL);

    }

}

//==================================================================================

//Description: set the lower six bytes of the pc to the offset of the current page

//==================================================================================

void EffectiveAddress::setPCOffset(unsigned short rpc)

{

    m\_rCurrPage = rpc & OFFSET\_PC\_MASK;

}

//==================================================================================

//Description:if the memarray page bit is 0, calculate 00000 + offset,

//              range is 0000o-0177o

//==================================================================================

unsigned short EffectiveAddress::effAdzeroPage()

{

    unsigned short temp = 0;

    //add page 0 to the offset

    temp = temp | m\_rOffset;

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: zero page: %o\n", temp);

#endif

    return temp;

}

//==================================================================================

//Description: if the memarray page bit is 1, calculate instruction page + offset,

//                  range instruction page

//==================================================================================

unsigned short EffectiveAddress::effAdcurrentPage()

{

    unsigned short temp = 0;

    temp = m\_rCurrPage | m\_rOffset;

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: current page: %o\n", temp);

#endif

    return temp;

}

//==================================================================================

//Description:if the indirect bit is 1, calculate  c(address),

//                  range is any address but costs 2 mem reads

//==================================================================================

unsigned short EffectiveAddress::effAdindirectAddressZero()

{

    unsigned short temp = 0;

    temp = effAdzeroPage(); //get the effective address

    //auto index

    //calculate c(0010o - 0017o) + 1 effective address

    if((temp > OFFSET\_AUTOINC\_MIN) && (temp <= OFFSET\_AUTOINC\_MAX))

    {

        temp += 1;

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: auto incremented: %o\n", temp);

#endif

    }

    //normal zero page indirect

    temp = readAddress(temp); //read the memory contents of the address

    loadOffset(temp); //load the address to the offset

    temp = effAdzeroPage(); //recalcuate the new effective address

    return temp;

}

//==================================================================================

//Description:if the indirect bit is 1, calculate  c(address),

//              range is any address but costs 2 mem reads

//==================================================================================

unsigned short EffectiveAddress::effAdindirectAddressCurr()

{

    unsigned short temp = 0;

    temp = effAdcurrentPage(); //get the effective address

    temp = readAddress(temp); //read the memory contents of the address

    loadOffset(temp); //load the address to the offset

    temp = effAdcurrentPage(); //recalcuate the new effective address

    return temp;

}

//==================================================================================

//Description: extracts the offset from an MRI instruction

//==================================================================================

void EffectiveAddress::loadOffset(unsigned short reg)

{

    m\_rOffset = 0;

    m\_rOffset = reg & OFFSET\_MASK;

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: offset: %o\n", m\_rOffset);

#endif

}

//==================================================================================

//Description: reads the new address from memory

//==================================================================================

unsigned short EffectiveAddress::readAddress(unsigned short address)

{

    m\_pmem->load(address);

    return m\_pmem->readMB();

}

//==================================================================================

//Description: returns true if the instruction was indirect

//==================================================================================

bool EffectiveAddress::isIndirect()

{

    return m\_indirect;

}

//==================================================================================

//Description:inputs an MRI instruction and returns the effective address

//==================================================================================

unsigned short EffectiveAddress::geteffAddress(unsigned short reg, unsigned short rpc)

{

    unsigned short temp = 0;

    unsigned short indirect = reg & INDIRECT\_MASK;

    indirect = indirect >> 8;

    unsigned short currpage = reg & MEMPAGE\_MASK;

    currpage = currpage >> 7;

    loadOffset(reg);

    setPCOffset(rpc);

    if(indirect == 1)

    {

        m\_indirect = true;

    }

    else

    {

        m\_indirect = false;

    }

#ifdef DEBUG\_CONTROL

*fprintf*(*stdout*, "DEBUG: offset: %o, PC offset: %o, Indirect: %o \n",

            m\_rOffset, m\_rCurrPage, m\_indirect);

#endif

    //zero page offset

    if(!currpage && !indirect)

    {

        temp = effAdzeroPage();

    }

    //zero page indirect

    else if(!currpage && indirect)

    {

        //auto indexing handled inside here

        temp = effAdindirectAddressZero();

    }

    //current page, no indirect

    else if(currpage && !indirect)

    {

        temp = effAdcurrentPage();

    }

    //current page and indirect

    else if(currpage && indirect)

    {

        temp = effAdindirectAddressCurr();

    }

    else

    {

        Error.printError(ERROR\_UNEXPECTED\_VALUE, FILE\_CONTROL);

    }

    return temp;

}

/\* ==================================================================================

ECE 486 / Winter 2015 / PDP-8 simulator project

Team:

Deborah Denhart

Jeremiah Franke

==================================================================================

File:     Common.cpp

Date:     03/02/2015

Description:  This file contains the classes ErrorTable and RegTable

================================================================================== \*/

#ifndef \_\_COMMON\_H\_INCLUDED\_\_

#define \_\_COMMON\_H\_INCLUDED\_\_

//Defines

//==================================================================================

#define INPUT\_BINARY 0

#define INPUT\_OCTAL 1

#define INPUT\_HEX 2

#define REG\_12BIT 12

#define REG\_8BIT 8

#define MAX\_OCT\_ADDRESS 07777

#define MAX\_DEC\_ADDRESS 4095

#define ADDRESS\_LENGTH\_OCT 4

#define ADDRESS\_LENGTH\_HEX 3

#define ADDRESS\_LENGTH\_BIN 12

#define INDIRECT\_MASK 256

#define MEMPAGE\_MASK 128

#define IR\_MASK 3584

#define OFFSET\_MASK 127

#define OFFSET\_PC\_MASK 3968

#define OFFSET\_AUTOINC\_MIN 8

#define OFFSET\_AUTOINC\_MAX 15

#define OP\_TABLE\_LENG 8

#define OPCODE\_AND 0

#define OPCODE\_TAD 1

#define OPCODE\_ISZ 2

#define OPCODE\_DCA 3

#define OPCODE\_JMS 4

#define OPCODE\_JMP 5

#define OPCODE\_IO 6

#define OPCODE\_MICRO 7

#define HALT\_CODE 3842

#define INST\_FORM\_MRI "Memory Reference\0"

#define INST\_FORM\_OP "Micro Instruction\0"

#define INST\_FORM\_IO "Test I/O\0"

#define BIT0\_MASK 2048

#define BIT1\_MASK 1024

#define BIT2\_MASK 512

#define BIT3\_MASK 256

#define BIT4\_MASK 128

#define BIT5\_MASK 64

#define BIT6\_MASK 32

#define BIT7\_MASK 16

#define BIT8\_MASK 8

#define BIT9\_MASK 4

#define BIT10\_MASK 2

#define BIT11\_MASK 1

#define REG\_12BIT\_MASK 4095

#define REG\_6BIT\_MASK 63

#define SYMBOLS\_MAX 16

#define REG\_UPPER 6

#define OPSTRING\_AND "AND\0"

#define OPSTRING\_TAD "TAD\0"

#define OPSTRING\_ISZ "ISZ\0"

#define OPSTRING\_DCA "DCA\0"

#define OPSTRING\_JMS "JMS\0"

#define OPSTRING\_JMP "JMP\0"

#define OPSTRING\_IO "IO \0"

#define OPSTRING\_MICRO "MIC\0"

//printing headers

#define PRINT\_BREAK "-----------------------------------------------"

//file codes

#define FILE\_MAIN 0

#define FILE\_COMMON 1

#define FILE\_MEM 2

#define FILE\_ALU 3

#define FILE\_CONTROL 4

#define FILE\_OPTABLE 5

#define FILE\_MAX 6

//errors

#define ERROR\_NONE 0

#define ERROR\_CONV\_FAILED -1

#define ERROR\_NULL -2

#define ERROR\_OUT\_OF\_RANGE -3

#define ERROR\_UNEXPECTED\_VALUE -4

#define ERROR\_FILE -5

#define ERROR\_MAX 6 //max is positive to allocate errorTable array

//Constants

//==================================================================================

const short ERRSIZE = ERROR\_MAX;

const short FILESIZE = FILE\_MAX;

const short MAX\_BUFFER = 100;

//Global Methods

//==================================================================================

//Class ErrorTable

//==================================================================================

class ErrorTable

{

private:

    bool errorsOn;

    char\* errorTable[ERRSIZE];

    char\* fileTable[FILESIZE];

public:

    ErrorTable();

    ~ErrorTable();

    void printError(short error, short fileCode);

    void debugOn(bool bSwitch);

};

#endif //\_\_COMMON\_H\_INCLUDED\_\_

/\* ==================================================================================

ECE 486 / Winter 2015 / PDP-8 simulator project

Team:

Deborah Denhart

Jeremiah Franke

==================================================================================

File:     Common.cpp

Date:     03/02/2015

Description:  This file contains the classes ErrorTable and RegTable

================================================================================== \*/

#include <fstream>

#include <cstdio>

#include "Common.h"

//External objects

//==================================================================================

//==================================================================================

//Description: constructor

//==================================================================================

ErrorTable::ErrorTable()

{

    errorsOn = true;

    const char\* temp1 = "No error";

    const char\* temp2 = "Conversion failed";

    const char\* temp3 = "Null Pointer";

    const char\* temp4 = "Out of Range";

    const char\* temp5 = "Unexpected Value";

    const char\* temp6 = "Can't access file";

    int strsize = 0;

    strsize = *strlen*(temp1)+1;

    errorTable[0] = new char[strsize];

*strcpy*(errorTable[0], temp1);

    strsize = *strlen*(temp2)+1;

    errorTable[1] = new char[strsize];

*strcpy*(errorTable[1], temp2);

    strsize = *strlen*(temp3)+1;

    errorTable[2] = new char[strsize];

*strcpy*(errorTable[2], temp3);

    strsize = *strlen*(temp4)+1;

    errorTable[3] = new char[strsize];

*strcpy*(errorTable[3], temp4);

    strsize = *strlen*(temp5)+1;

    errorTable[4] = new char[strsize];

*strcpy*(errorTable[4], temp5);

    strsize = *strlen*(temp6)+1;

    errorTable[5] = new char[strsize];

*strcpy*(errorTable[5], temp6);

    const char\* file0 = "main";

    const char\* file1 = "common";

    const char\* file2 = "memory";

    const char\* file3 = "alu";

    const char\* file4 = "control";

    const char\* file5 = "op table";

    strsize = *strlen*(file0)+1;

    fileTable[0] = new char[strsize];

*strcpy*(fileTable[0], file0);

    strsize = *strlen*(file1)+1;

    fileTable[1] = new char[strsize];

*strcpy*(fileTable[1], file1);

    strsize = *strlen*(file2)+1;

    fileTable[2] = new char[strsize];

*strcpy*(fileTable[2], file2);

    strsize = *strlen*(file3)+1;

    fileTable[3] = new char[strsize];

*strcpy*(fileTable[3], file3);

    strsize = *strlen*(file4)+1;

    fileTable[4] = new char[strsize];

*strcpy*(fileTable[4], file4);

    strsize = *strlen*(file5)+1;

    fileTable[5] = new char[strsize];

*strcpy*(fileTable[5], file5);

}

//==================================================================================

//Description: deconstructor

//==================================================================================

ErrorTable::~ErrorTable()

{

}

//==================================================================================

//Description: prints an error that is passed in and a file from where the error

// was found

//==================================================================================

void ErrorTable::printError(short error, short fileCode)

{

    short iErr = *abs*(error);

    if((iErr > 0) && (iErr < ERROR\_MAX) && (fileCode > 0)

                  && (fileCode < FILE\_MAX) && (errorsOn))

    {

        if(errorsOn) //if silent mode

        {

*fprintf*(*stderr*, "%s\n", PRINT\_BREAK);

*fprintf*(*stderr*, "\*\*\*Error found in %s: %s\*\*\*\n", fileTable[fileCode], errorTable[iErr]);

*fprintf*(*stderr*, "%s\n", PRINT\_BREAK);

        }

    }

}

//==================================================================================

//Description: turns on debug messages

//==================================================================================

void ErrorTable::debugOn(bool bSwitch)

{

    errorsOn = bSwitch;

}

/\* ==================================================================================

ECE 486 / Winter 2015 / PDP-8 simulator project

Team:

Deborah Denhart

Jeremiah Franke

==================================================================================

File:     alu.cpp

Date:     03/02/2015

Description:  This file contains the class alu

================================================================================== \*/

#ifndef \_\_alu\_H\_INCLUDED\_\_

#define \_\_alu\_H\_INCLUDED\_\_

//Dependencies

//==================================================================================

//Defines

//==================================================================================

//#define DEBUG\_ALU

//Class alu

//==================================================================================

class alu

{

private:

    bool m\_bZero;

    bool m\_bNegative;

    bool rLB; // 1 bit link bit (carry out)

    unsigned short rAC; //12 bit alu

    //unsigned short rMQ; //12 bit multiplier quotient

public:

    alu();

    ~alu();

    unsigned short getAC();

    bool getLB();

    void setAC(unsigned short rac);

    void setLB(bool rac);

    void sumALU(unsigned short reg);

    bool isNegative();

    bool isZero();

    void complementLC();

    void complementALU();

    void incrementALU();

    void negateALU();

    void clear();

    void clearAC();

    void clearLC();

    void rotateRight();

    void rotateLeft();

    void shiftRight(unsigned short  num);

    void shiftLeft(unsigned short  num);

    void andALU(unsigned short reg);

    void orALU(unsigned short reg);

    void printAll();

    void printALU();

};

#endif

// \_\_alu\_H\_INCLUDED\_\_

/\* ==================================================================================

ECE 486 / Winter 2015 / PDP-8 simulator project

Team:

Deborah Denhart

Jeremiah Franke

==================================================================================

File:     alu.cpp

Date:     03/02/2015

Description:  This file contains the class alu

================================================================================== \*/

#include <cstring>

#include <cstdio>

#include "Common.h"

#include "alu.h"

//External objects

//==================================================================================

extern ErrorTable Error;

//==================================================================================

//Description: constructor

//==================================================================================

alu::alu()

{

    m\_bZero = false;

    m\_bNegative = false;

    rLB = 0;

    rAC = 0;

}

//==================================================================================

//Description: deconstructor

//==================================================================================

alu::~alu()

{

    m\_bZero = false;

    m\_bNegative = false;

}

//==================================================================================

//Description: returns the value of rAC

//==================================================================================

unsigned short alu::getAC()

{

    return rAC;

}

//==================================================================================

//Description: sets AC to a register value

//==================================================================================

void alu::setAC(unsigned short rac)

{

    rAC = rac;

}

//==================================================================================

//Description: returns the value of rLB

//==================================================================================

bool alu::getLB()

{

    return rLB;

}

//==================================================================================

//Description: sets the rLB to a value

//==================================================================================

void alu::setLB(bool rlb)

{

    rLB = rlb;

}

//==================================================================================

//Description: adds the input register to the contents of rALU

//==================================================================================

void alu::sumALU(unsigned short reg)

{

    unsigned short temp = 0;

    temp = rAC + reg;

    if(temp > MAX\_OCT\_ADDRESS)

    {

        rLB = 1;

        temp = MAX\_OCT\_ADDRESS - rAC;

    }

    else

    {

        rLB = 0;

    }

#ifdef DEBUG\_ALU

*fprintf*(*stdout*, "DEBUG: sum alu rAC %o = %o + %o, rLB = %o\n", temp, reg, rAC, rLB);

#endif

    rAC = temp;

}

//==================================================================================

//Description: returns true if rAC's MSB is 1

//==================================================================================

bool alu::isNegative()

{

    unsigned short temp = 0;

    temp = rAC & BIT11\_MASK;

#ifdef DEBUG\_ALU

*fprintf*(*stdout*, "DEBUG: checking if rAC is negative: %o\n", temp);

#endif

    return temp;

}

//==================================================================================

//Description: returns true of rAC is 0

//==================================================================================

bool alu::isZero()

{

    bool temp = false;

    if(0 == rAC)

    {

        temp = true;

    }

#ifdef DEBUG\_ALU

*fprintf*(*stdout*, "DEBUG: checking if rAC is 0: %o\n", temp);

#endif

    return temp;

}

//==================================================================================

//Description: complement each bit in the alu

//==================================================================================

void alu::complementLC()

{

    bool temp = rLB;

    rLB = !rLB;

#ifdef DEBUG\_ALU

*fprintf*(*stdout*, "DEBUG: complemented rLB from %o to %o\n", temp, rLB);

#endif

}

//==================================================================================

//Description: complement each bit in the rAC

//==================================================================================

void alu::complementALU()

{

    unsigned short temp = rAC;

    rAC = MAX\_OCT\_ADDRESS - rAC;

#ifdef DEBUG\_ALU

*fprintf*(*stdout*, "DEBUG: complemented rAC from %o to %o\n", temp, rAC);

#endif

}

//==================================================================================

//Description: increments the value of rAC by 1

//==================================================================================

void alu::incrementALU()

{

    unsigned short temp = rAC;

    sumALU(1);

#ifdef DEBUG\_ALU

*fprintf*(*stdout*, "DEBUG: incrementing from %o to %o\n", temp, rAC);

#endif

}

//==================================================================================

//Description: 2's complement negate

//==================================================================================

void alu::negateALU()

{

    unsigned short temp = rAC;

    complementALU();

    incrementALU();

#ifdef DEBUG\_ALU

*fprintf*(*stdout*, "DEBUG: negated rAC from %o to %o\n", temp, rAC);

#endif

}

//==================================================================================

//Description: sets rLB to 0

//==================================================================================

void alu::clearLC()

{

    rLB = 0;

}

//==================================================================================

//Description: sets rAC to 0

//==================================================================================

void alu::clearAC()

{

    rAC = 0;

}

//==================================================================================

//Description: clears bot rLB and rAC

//==================================================================================

void alu::clear()

{

    clearAC();

    clearLC();

#ifdef DEBUG\_ALU

*fprintf*(*stdout*, "DEBUG: cleared rAC: %o and rLB: %o\n", rAC, rLB);

#endif

}

//==================================================================================

//Description:shifts rAC right by one and carries the LSB to the MSB

//==================================================================================

void alu::rotateRight()

{

    unsigned short temp = 0;

    unsigned short carry = 0;

    unsigned short rlb = 0;

    temp = rAC;

    rlb = rLB;

    rlb = rLB << REG\_12BIT;

    carry = rAC & BIT0\_MASK;

    rLB = carry >> REG\_12BIT;

    rAC = rAC >> 1;

    rAC += rlb;

    rAC &= REG\_12BIT\_MASK;

#ifdef DEBUG\_ALU

*fprintf*(*stdout*, "DEBUG: rotate right from %o to %o %o\n",

            temp, rLB, rAC);

#endif

}

//==================================================================================

//Description: shifts rAC left by one and carries the MSB to the LSB

//==================================================================================

void alu::rotateLeft()

{

    unsigned short temp = 0;

    unsigned short carry = 0;

    temp = rAC;

    carry = rLB;

    rLB = rAC & BIT0\_MASK;

    rLB = rLB >> REG\_12BIT;

    rAC = rAC << 1;

    rAC += carry;

    rAC &= REG\_12BIT\_MASK;

#ifdef DEBUG\_ALU

*fprintf*(*stdout*, "DEBUG: rotate left from %o to %o with a carry %o\n",

            temp, rAC, carry);

#endif

}

//==================================================================================

//Description: shifts rAC right by num amount

//==================================================================================

void alu::shiftRight(unsigned short num)

{

    unsigned short temp = rAC;

    rAC = rAC >> num;

    rAC &= REG\_12BIT\_MASK;

#ifdef DEBUG\_ALU

*fprintf*(*stdout*, "DEBUG: shift right from %o to %o\n", temp, rAC);

#endif

}

//==================================================================================

//Description: shifts rAC left by num amount

//==================================================================================

void alu::shiftLeft(unsigned short  num)

{

    unsigned short temp = rAC;

    rAC = rAC << num;

    rAC &= REG\_12BIT\_MASK;

#ifdef DEBUG\_ALU

*fprintf*(*stdout*, "DEBUG: shift left from %o to %o\n", temp, rAC);

#endif

}

//==================================================================================

//Description: ands the input reg with rAC

//==================================================================================

void alu::andALU(unsigned short reg)

{

    unsigned short temp = rAC;

    rAC &= reg;

#ifdef DEBUG\_ALU

*fprintf*(*stdout*, "DEBUG: and AC from %o to %o\n", temp, rAC);

#endif

}

//==================================================================================

//Description: ors the input reg with rAC

//==================================================================================

void alu::orALU(unsigned short reg)

{

    unsigned short temp = rAC;

    rAC |= reg;

#ifdef DEBUG\_ALU

*fprintf*(*stdout*, "DEBUG: or AC from %o to %o\n", temp, rAC);

#endif

}

//==================================================================================

//Description: prints a test of all functions of the alu

//==================================================================================

void alu::printAll()

{

    unsigned short temp = 07007;

    printALU();

    incrementALU();

    printALU();

    sumALU(temp);

    printALU();

    andALU(temp);

    printALU();

    orALU(temp);

    printALU();

    shiftLeft(2);

    printALU();

    shiftRight(2);

    printALU();

    rotateLeft();

    printALU();

    rotateRight();

    printALU();

    negateALU();

    printALU();

    clear();

    printALU();

}

//==================================================================================

//Description: prints the value of the alu in decimal, octal and hexidecimal

//==================================================================================

void alu::printALU()

{

*fprintf*(*stdout*, "ALU: dec: %d  oct: %o  hex: %x\n", rAC, rAC, rAC);

}