ELC010 CPU Cache Simulation

Matt Blickem – B022220

March 2014

Contents

[Instructions 1](#_Toc383786768)

[Source Code 1](#_Toc383786769)

[Cache.h 1](#_Toc383786770)

[CPUTrace.h 4](#_Toc383786771)

[Util.h 4](#_Toc383786772)

[Cache.cpp 5](#_Toc383786773)

[CacheBlock.cpp 10](#_Toc383786774)

[CPUInstruction.cpp 13](#_Toc383786775)

[CPUTrace.cpp 14](#_Toc383786776)

[Main.cpp 15](#_Toc383786777)

[MainMemory.cpp 19](#_Toc383786778)

# Instructions

The trace.txt file should be in the same directory as the executable. Follow onscreen instructions after running the executable. View output.log file (created in the same directory as the executable) for details on the simulation execution.

Source code is available at: https://github.com/explosivose/ELC010\_CW2

# Source Code

### Cache.h

#pragma once

#include <vector>

using namespace std;

/\* Cache Simulation

Implemented as Cache container class and CacheBlock class

\*/

class MainMemory

{

public:

// constructor / destructor

MainMemory();

MainMemory(const unsigned int bytes);

~MainMemory();

unsigned int getLength();

vector<unsigned int> ReadBlock(const unsigned int address); // return data at memory address

void WriteBlock(const unsigned int address, const vector<unsigned int>& data); // write data to memory address

private:

void init();

unsigned int length; // size of memory in bytes (default is 4096)

vector<vector<unsigned int>> Data;

bool ValidAddress(const unsigned int address);

};

class CacheBlock

{

public:

// class statics

static void setLineLength(const unsigned int);

static unsigned int getLineLengthBytes();

static unsigned int getLineLengthWords();

// constructor / destructor

CacheBlock();

~CacheBlock();

// getters

bool isValid();

bool isDirty();

unsigned int Tag(); // returns the tag of this block

unsigned int ReadWord(const unsigned int); // returns a data word from the cache line

vector<unsigned int> ReadLine();

// setters

void isValid(const bool);

void isDirty(const bool);

void WriteWord(const unsigned int offset, const unsigned int data);

void LineFillFromCPU(const unsigned int Tag, const vector<unsigned int>& data); // copy CPU data into cache line, set valid and dirty

void LineFillFromMemory(const unsigned int Tag, const vector<unsigned int>& data); // copy memory data into cache line, set valid and not dirty

private:

static unsigned int lineLength;

bool valid;

bool dirty;

unsigned int tag;

vector<unsigned int> line;

};

class Cache

{

public:

// constructor / destructor

Cache();

Cache(MainMemory\*);

Cache(MainMemory\*, unsigned int size);

Cache(MainMemory\*, unsigned int size, unsigned int Ways);

~Cache();

unsigned int getLength();

unsigned int Read(const unsigned int address); // return data at memory address

void Write(const unsigned int address, const unsigned int data); // write data to memory address

float MissRate();

private:

void init(); // allocate memory for cache, calculate bit lengths for sel, tag and index

void ProcessAddress(unsigned int address); // calculate sel, tag and index from CPU address

bool Hit(unsigned int w); // true or false for cache hit

bool ValidIndex(); // check index range

unsigned int Evict(unsigned int address); // figure out where in main memory the indexed cache block belongs and write it back

unsigned int length; // size of the cache in bytes (default is 1024)

bool cacheReady; // false if no reference to MainMemory;

unsigned int selectBitsLength; // number of bits required to select a data word

unsigned int indexLength; // number of bits required to index the cache

unsigned int tagLength; // number of bits required for tag

unsigned int sel; // select bits choose a data word in a cache line or memory block

unsigned int tag; // tag indicates which memory block is currently in cache line

unsigned int index; // index selects a cache block

unsigned int ways; // number of cache lines a memory block can occupy

unsigned int rr; // round robin index for writebacks

unsigned int reads; // count the number of reads

unsigned int writes; // count the nmuber of writes

unsigned int hits; // count the number of cache hits

unsigned int misses; // count the number of cache misses

MainMemory\* memory; // pointer to single instance of main memory

vector<vector<CacheBlock>> block; // dynamic array of cache blocks

};

### CPUTrace.h

#pragma once

#include<fstream>

class CPUInstruction

{

public:

CPUInstruction(void);

CPUInstruction(const unsigned int address);

CPUInstruction(const unsigned int address, const unsigned int data);

~CPUInstruction(void);

bool isWrite();

unsigned int getAddress();

unsigned int getData();

private:

bool write;

unsigned int address;

unsigned int data;

};

class CPUTrace

{

public:

CPUTrace(void);

~CPUTrace(void);

void GetNextInstruction(CPUInstruction&, std::ifstream&);

private:

};

### Util.h

#pragma once

static int powbase2(int);

static int powbase2(unsigned int power)

{

int result = 1;

for (unsigned int i = 0; i < power; i++)

result \*= 2;

return result;

}

### Cache.cpp

#include <iostream>

#include <iomanip>

#include <cmath>

#include <vector>

#include <bitset>

#include "Util.h"

#include "Cache.h"

using namespace std;

// Constructor / Destructor

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//

//

Cache::Cache()

{

length = 1024;

cacheReady = false;

ways = 1;

}

Cache::Cache(MainMemory\* mainMemory)

{

length = 1024;

memory = mainMemory;

ways = 1;

init();

}

Cache::Cache(MainMemory\* mainMemory, unsigned int size)

{

length = size;

memory = mainMemory;

ways = 1;

init();

}

Cache::Cache(MainMemory\* mainMemory, unsigned int size, unsigned int Ways)

{

length = size;

memory = mainMemory;

ways = Ways;

init();

}

Cache::~Cache()

{

}

// Public function

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//

//

unsigned int Cache::Read(const unsigned int address)

{

cout << hex << "Addr:\t0x" << address << endl;

ProcessAddress(address); // extract sel, index and tag

cout << "Cache Read..." << endl;

unsigned int w = -1;

for (unsigned int i = 0; i < ways; i++)

{

if (Hit(i)) w = i;

}

if( w == -1 )

{

cout << "Cache Miss!" << endl;

misses++;

w = Evict(address);

}

else

{

cout << "Cache Hit!" << endl;

cout << "Using way:\t" << w << endl;

hits++;

}

unsigned int data = block[w][index].ReadWord(sel);

cout << hex << "Cache Read Data: 0x" << data << endl;

/\*

bitset<32> b\_data(block[index].ReadWord(sel));

cout << "Data: \t\t" + b\_data.to\_string() + " from cache read." << endl;

cout << endl;

\*/

return data;

}

void Cache::Write(unsigned int address, unsigned int data)

{

cout << hex << "Addr:\t0x" << address << "\tData:\t0x" << data << endl;

ProcessAddress(address); // extract sel, index and tag

cout << "Cache Write..." << endl;

unsigned int w = -1;

for (unsigned int i = 0; i < ways; i++)

{

if (Hit(i)) w = i;

}

if (w == -1)

{

cout << "Cache Miss!" << endl;

misses++;

w = Evict(address);

}

else

{

cout << "Cache Hit!" << endl;

cout << "Using way:\t" << w << endl;

hits++;

}

cout << "Write data word to cache!" << endl;

block[w][index].WriteWord(sel, data);

block[w][index].isDirty(true);

}

float Cache::MissRate()

{

cout << dec << "Hits:\t" << hits << "\tMisses:\t" << misses << endl;

float m = misses;

float hm = hits+misses;

return m / hm;

}

// Private functions

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//

//

void Cache::init()

{

// calculate length of bits to extract from cpu addresses

// for log2() i'm using maths: log2(x) = log(x)/log(2)

// number of select bits depends on the number of words in a cache line

// = ceil(log2(CacheBlock::lineLength/4))

// the select bits are the first bits in the address (from the left)

selectBitsLength = ceil(log(CacheBlock::getLineLengthWords())/log(2));

// number of index bits depends on number of cache blocks

// = ceil(log2((Cache::length/ways)/CacheBlock::lineLength))

indexLength = ceil(log((length/ways)/CacheBlock::getLineLengthBytes())/log(2));

//indexLength = ceil(log(length)/log(2)); // this is incorrect!

// number of tag bits depends on memory length and cache length

// = ceil(log2(Memory::length / Cache::length))

// ceil(log2(4096/1024)) = 2

tagLength = ceil(log(memory->getLength()/length)/log(2));

// intialise the cache blocks

block.resize(ways);

for (unsigned int w = 0; w < ways; w++)

{

block[w].resize(length/ways);

}

rr = 0;

hits = 0;

misses = 0;

reads = 0;

writes = 0;

cacheReady = true;

cout << "---Cache Parameters---" << endl;

cout << "Number of ways:\t" << ways << endl;

cout << "Cache Size:\t" << length << endl;

cout << "Index Length:\t" << indexLength << endl;

cout << "Tag Length:\t" << tagLength << endl;

cout << endl;

}

unsigned int Cache::getLength()

{

return length;

}

void Cache::ProcessAddress(unsigned int address)

{

// from address take the tag, index, select bits

// bits to extract = pow(2, length) - 1 << offset

sel = (powbase2(selectBitsLength) - 1) << 0;

sel &= address;

sel = sel >> 0;

index = (powbase2(indexLength) - 1) << selectBitsLength;

index &= address;

index = index >> selectBitsLength;

tag = (powbase2(tagLength) - 1) << (selectBitsLength + indexLength);

tag &= address;

tag = tag >> (selectBitsLength + indexLength);

cout << hex << "Tag:\t0x" << tag << endl;

cout << hex << "Index:\t0x" << index << endl;

/\* using bitset to print binary

bitset<32> b\_address(address);

//bitset<32> b\_sel(sel);

//bitset<32> b\_index(index);

//bitset<32> b\_tag(tag);

cout << "Address: \t" + b\_address.to\_string() << endl;

//cout << "select: \t" + b\_sel.to\_string() << endl;

//cout << "index: \t\t" + b\_index.to\_string() << endl;

//cout << "tag: \t\t" + b\_tag.to\_string() << endl;

//\*/

}

bool Cache::Hit(unsigned int w)

{

if (!ValidIndex()) return false;

bool hit = false;

if (block[w][index].isValid())

{

if (block[w][index].Tag() == tag)

{

hit = true;

}

}

return hit;

}

bool Cache::ValidIndex()

{

// check range of address

if (index >= length/ways)

{

cout << "Cache index out of range!!" << endl;

return false;

}

else

{

return true;

}

}

// figure out where in main memory the indexed cache block belongs and write it back

// eviction shall select which way index to use in cache

// Evict() is called on cache miss

// find a cache block to populate

// writeback to main memory if we're using a valid dirty block

// return the ways index that was used

unsigned int Cache::Evict(unsigned int address)

{

// first find invalid cache blocks to use

for (unsigned int w = 0; w < ways; w++)

{

if ( !block[w][index].isValid() )

{

cout << "Using way:\t" << w << endl;

cout << "Cache Fill from memory!" << endl;

block[w][index].LineFillFromMemory(tag, memory->ReadBlock(address));

return w;

}

}

cout << "All mapped cache blocks are in use." << endl;

// next favour clean cache blocks

// this shouldn't really happen though!

// just use round robin after failing to find invalid cache block

for (unsigned int w = 0; w < ways; w++)

{

if ( !block[w][index].isDirty() )

{

cout << "Using way:\t" << w << endl;

cout << "Cache fill from memory!" << endl;

block[w][index].LineFillFromMemory(tag, memory->ReadBlock(address));

return w;

}

}

cout << "All mapped cache blocks are dirty." << endl;

// round robin method for writebacks

unsigned int w = rr;

rr++;

if (rr >= ways) rr = 0;

cout << "Using way:\t" << w << endl;

// reconstruct address

unsigned int t = block[w][index].Tag() << (selectBitsLength + indexLength);

unsigned int i = index << selectBitsLength;

unsigned int addr = t | i;

cout << "Writeback!" << endl;

cout << hex << "\tAddr:\t0x" << addr << endl;

memory->WriteBlock(addr, block[w][index].ReadLine());

cout << "Cache line fill from memory!" << endl;

block[w][index].LineFillFromMemory(tag, memory->ReadBlock(address));

return w;

}

### CacheBlock.cpp

#include <iostream>

#include "Cache.h"

using namespace std;

// Cache Block Statics

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//

//

// Cache Line Length measured in bytes

// Default length is 16 bytes (4 data words)

unsigned int CacheBlock::lineLength = 16;

// sets a new cache line length

// in future this should also resize the line\*

// or do nothing if there are any instances of CacheBlock

void CacheBlock::setLineLength(const unsigned int length)

{

CacheBlock::lineLength = length;

}

// return the length of a cache line

unsigned int CacheBlock::getLineLengthBytes()

{

return CacheBlock::lineLength;

}

unsigned int CacheBlock::getLineLengthWords()

{

return CacheBlock::lineLength/4;

}

// Constructor / Destructor

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//

//

// constructor alocates memory for cache line

CacheBlock::CacheBlock()

{

line.resize(CacheBlock::lineLength/4, 0);

valid = false;

}

// destructor releases memory used by cache line

CacheBlock::~CacheBlock()

{

}

// Cache Block getters

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//

//

// return the valid flag for this cache block

bool CacheBlock::isValid()

{

return valid;

}

// return the dirty flag for this cache block (can only be dirty if valid)

bool CacheBlock::isDirty()

{

return dirty && valid;

}

// return the tag for this cache block

unsigned int CacheBlock::Tag()

{

return tag;

}

vector<unsigned int> CacheBlock::ReadLine()

{

return line;

}

unsigned int CacheBlock::ReadWord(const unsigned int offset)

{

if (offset > CacheBlock::lineLength)

{

cout << "Error! Data offset greater than cache line length" << endl;

return 0;

}

return line[offset];

}

// Cache Block setters

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//

//

// set the valid flag for this cache block

void CacheBlock::isValid(const bool v)

{

valid = v;

}

// set the dirty flag for this cache block

void CacheBlock::isDirty(const bool d)

{

dirty = d;

}

void CacheBlock::WriteWord(const unsigned int offset, const unsigned int data)

{

if (offset > line.size())

{

cout << "Error! Data offset greater than cache line length" << endl;

return;

}

line[offset] = data;

}

// problem with the two LineFill functions: size of data[] cannot be determined here.

// if size of data[] is less than CacheBlock::lineLength, we're writing garbage to line[]!

// use C++ vectors instead? then change function to return bool indicating success

// (another advantage of using vectors instead: much easier to debug! data in vectors is exposed to debugger)

// or ensure that only memory blocks are passed somehow (since their size is always CacheBlock::lineLength)

// fill the cache line with data

void CacheBlock::LineFillFromCPU(const unsigned int Tag, const vector<unsigned int>& data)

{

if (line.size() == data.size())

{

tag = Tag;

valid = true;

dirty = true;

line = data;

}

else

{

cout << "Line Fill Error: CPU data size differs to cache line size!" << endl;

}

}

void CacheBlock::LineFillFromMemory(const unsigned int Tag, const vector<unsigned int>& data)

{

if (line.size() == data.size())

{

tag = Tag;

valid = true;

dirty = false;

line = data;

}

else

{

cout << "Line Fill Error: Memory block size differs to cache line size!" << endl;

}

}

### CPUInstruction.cpp

#include "CPUTrace.h"

using namespace std;

// Constructor / Destructor

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//

//

CPUInstruction::CPUInstruction(void)

{

write = false;

address = 0;

data = 0;

}

CPUInstruction::CPUInstruction(const unsigned int Address)

{

write = false;

address = Address;

data = 0;

}

CPUInstruction::CPUInstruction(const unsigned int Address, const unsigned int Data)

{

write = true;

address = Address;

data = Data;

}

CPUInstruction::~CPUInstruction(void)

{

}

// public functions

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//

//

bool CPUInstruction::isWrite()

{

return write;

}

unsigned int CPUInstruction::getAddress()

{

return address;

}

unsigned int CPUInstruction::getData()

{

return data;

}

// Private Functions

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//

//

### CPUTrace.cpp

#include <iostream>

#include <fstream>

#include <string>

#include <cstdlib>

#include "CPUTrace.h"

using namespace std;

// Constructor / Destructor

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//

//

CPUTrace::CPUTrace(void)

{

}

CPUTrace::~CPUTrace(void)

{

}

// public functions

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//

//

// remember to document that the file must only use address and data hex values that are 4 words long

void CPUTrace::GetNextInstruction(CPUInstruction& i, ifstream& fs)

{

if ( fs.good() )

{

string readLine;

getline(fs, readLine);

if (fs.eof()) return;

unsigned int address = strtoul(readLine.substr(4, 8).c\_str(), NULL, 16);

unsigned int data = 0;

string rw = readLine.substr(0,1);

if ( rw == "W" || rw == "w" )

{

data = strtoul(readLine.substr(15, 8).c\_str(), NULL, 16);

i = CPUInstruction(address, data);

}

else if ( rw == "R" || rw == "r" )

{

i = CPUInstruction(address);

}

}

else

{

cout << "Could not read trace file!" << endl;

}

}

// Private Functions

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//

//

### Main.cpp

#include <iostream>

#include <fstream>

#include <string>

#include "CPUTrace.h"

#include "Cache.h"

using namespace std;

int main()

{

// ask the user for cache simulation parameters

cout << "CPU Cache Simulation" << endl;

cout << "Author:\tMatt Blickem" << endl;

cout << "March 2014" << endl;

cout << endl;

// open trace file for reading

ifstream tracefile;

tracefile.open("trace.txt");

if ( !tracefile.good() )

{

cout << "Could not open trace.txt" << endl;

cout << "Will now exit..." << endl;

cin.ignore();

return 0;

}

// open output.log for writing

ofstream outputfile;

outputfile.open("output.log");

if ( !outputfile.good() )

{

cout << "Could not open output.log" << endl;

cout << "Will now exit..." << endl;

cin.ignore();

return 0;

}

// user input

unsigned int cacheLineLength = 16; // default line length in bytes

unsigned int memorySize = 65535; // default memory size in bytes

unsigned int cacheSize = 1023; // default cache size in bytes

unsigned int numberOfWays = 1; // default number of cache ways

bool simulate = false;

int choice;

while (!simulate)

{

cout << endl << endl;

cout << "Memory Size:\t\t" << memorySize << " Bytes" << endl;

cout << "Cache Size:\t\t" << cacheSize << " Bytes" << endl;

cout << "Cache Line Length:\t" << cacheLineLength << " words" << endl;

cout << "Number of ways:\t\t" << numberOfWays << endl;

cout << endl;

cout << "Enter one of the following integer options:" << endl;

cout << "1. Change Memory Size" << endl;

cout << "2. Change Cache Size" << endl;

cout << "3. Change Cache Line Length" << endl;

cout << "4. Change number of ways" << endl;

cout << "5. Start Simulation" << endl;

for (choice = -1; !(1 <= choice && choice <= 5); )

{

cout << "Your choice: ";

if(!(cin >> choice))

{

cout << "Invalid input." << endl;

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n'); // ignore everything until user presses enter

}

}

switch(choice)

{

default:

cout << "Invalid choice." << endl;

break;

case 1:

unsigned int ms;

for (ms = 0; !(4095 <= ms && ms <= 4194303); )

{

cout << "New Memory Size (4095 to 4194303 bytes): ";

if(!(cin >> ms))

{

cout << "Invalid input." << endl;

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n'); // ignore everything until user presses enter

}

}

memorySize = ms;

break;

case 2:

unsigned int cs;

for (cs = 0; !(1023 <= cs && cs <= memorySize); )

{

cout << "New Cache Size (4095 to " << memorySize << " bytes): ";

if(!(cin >> cs))

{

cout << "Invalid input." << endl;

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n'); // ignore everything until user presses enter

}

}

cacheSize = cs;

break;

case 3:

unsigned int ll;

for (ll = 0; !(ll==4 || ll==8 || ll==16); )

{

cout << "New Cache Line Length (4, 8 or 16 words): ";

if(!(cin >> ll))

{

cout << "Invalid input." << endl;

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n'); // ignore everything until user presses enter

}

}

cacheLineLength = ll;

break;

case 4:

unsigned int cw;

for (cw = 0; !(cw==1 || cw==2 || cw==4); )

{

cout << "New Cache Ways (1, 2 or 4): ";

if(!(cin >> cw))

{

cout << "Invalid input." << endl;

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n'); // ignore everything until user presses enter

}

}

numberOfWays = cw;

break;

case 5:

simulate = true;

break;

}

}

// start simulation

// redirect cout stream to outputfile stream

streambuf \*outputbuf, \*backupbuf;

backupbuf = cout.rdbuf();

outputbuf = outputfile.rdbuf();

cout.rdbuf(outputbuf);

// start CPU Cache Simulation

CPUTrace trace;

CPUInstruction instruction;

CacheBlock::setLineLength(cacheLineLength\*4);

MainMemory memory(memorySize);

Cache cache(&memory, cacheSize, numberOfWays);

cout << endl;

cout << "SIMULATION START" << endl;

cout << endl;

while (!tracefile.eof())

{

trace.GetNextInstruction(instruction, tracefile);

if (instruction.isWrite())

{

cout << "WRITE" << endl;

cache.Write(instruction.getAddress(), instruction.getData());

cout << endl;

}

else

{

cout << "READ" << endl;

cache.Read(instruction.getAddress());

cout << endl;

}

}

tracefile.close();

cout << endl;

cout << "SIMULATION END" << endl;

cout << endl;

cout << fixed << "Cache Miss Rate:\t" << cache.MissRate() << endl;

// restore cout stream buffer

cout.rdbuf(backupbuf);

outputfile.close();

cout << "Simulation Complete." << endl;

cout << "See output.log file for details" << endl;

cin.clear();

cin.ignore();

return 0;

}

### MainMemory.cpp

#include <iostream>

#include "Cache.h"

using namespace std;

// Constructor / Destructor

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//

//

// constructor allocates memory for data\*

MainMemory::MainMemory()

{

length = 4096;

init();

}

MainMemory::MainMemory(const unsigned int size)

{

length = size;

init();

}

// destructor releases memory used by data\*

MainMemory::~MainMemory(void)

{

}

// public functions

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//

//

std::vector<unsigned int> MainMemory::ReadBlock(const unsigned int address)

{

if ( ValidAddress(address) )

{

return Data[address];

}

else

{

return Data[0];

}

}

void MainMemory::WriteBlock(const unsigned int address, const vector<unsigned int>& data)

{

if ( ValidAddress(address) )

{

for (unsigned int i = 0; i < CacheBlock::getLineLengthWords(); i++)

{

Data[address][i] = data[i];

}

}

}

// Private Functions

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// initialisation

void MainMemory::init()

{

Data.resize(length);

for (unsigned int i = 0; i < length; i++)

{

Data[i].resize(CacheBlock::getLineLengthWords());

}

cout << "---Main Memory Parameters---" << endl;

cout << "Memory Size:\t" << length << endl;

cout << endl;

}

// return the length of main memory

unsigned int MainMemory::getLength()

{

return MainMemory::length;

}

bool MainMemory::ValidAddress(const unsigned int address)

{

// check that the first few bits of the address are zero

// the number of bits to check is the number of bits required to express Cache Line Length

// extracting the first few bits:

// bits = address & (powbase2(cacheLineLength) -1)

// Data[address][bits]

// (powbase2(cacheLineLength) >> address) is what should be used to index Data[]

// then address & cache

// check range of address

if ( address >= MainMemory::length)

{

cout << "Memory address out of range!!" << endl;

return false;

}

else

{

return true;

}

}