**Superscalar out of order architectural simulator (With Memory Hierarchy)**

A project for CSEN702 Microprocessors

Supervised by:

* Dr. Mohamed Taher
* Eng. Jolly Salah

Team Members:

* Ahmed Tarek 31-4177
* Mostafa Ibrahim
* Omar
* Nada Bahaa

How the work was divided:

* Memory Hierarchy: Ahmed Tarek
* Functional Units: Nadeen
* Tomasulo: Mostafa Ibrahim, Nada Bahaa, Omar Yousry
* Utilities: Ahmed Tarek
* Assembler: Ahmed Tarek

**Implementation**

*Memory Hierarchy Stage:*

Block: The basic block unit.

* Attributes:
  + String [] data; // to hold data in bytes
  + private String tag; // tag bits
  + private int validBit; // determines if the content is valid
  + private int dirtyBit; // used in writeBack only

Set: is the building unit of a cache, it holds the blocks

* Attributes:
  + Block[] blocks; // Blocks in a Set

Cache: represents a single cache

Attributes:

Set [] sets; // array of sets containing blocks, 1 set if fully Associative, same as number of blocks if direct mapped

private int size; // size of cache

private int lineSize; // line size of cache

private int m; // associativity

private String writePolicyHit; // writeThrough or writeBack

private String writePolicyMiss; // writeThrough or WB

private int accessCycles; // access time (in cycles)

private int totalHits; // cache hits

private int totalMisses; // cache misses

// tracking cache state

private boolean isBeingAccessed; // Data is currently being accessed from a D-Cache

private int accessCyclesRemaining; // tracks the number of cycles remaining complete instruction fetch.

private boolean isBeingFetched; // An instruction is currently being fetched from an I-Cache

private int fetchCyclesRemaining; // tracks the number of cycles remaining complete instruction fetch

Functions:

// This one takes a string address and data, it writes the data to that address (byte addressable memory)

public void writeByte(String address, String data)

// This one takes a string address , it reads the data in that address location and returns it

public String read(String address)

// takes a string address and returns true if it is a cache hit and false otherwise

public boolean hit(String address)

// takes an address and returns the tag bits of that address in string form

public String getTagBits(String addr)

// takes an address and returns the tag bits of that address in string form

public String getIndexBits(String addr)

// takes an address and returns the offset bits of that address in string form

public String getOffsetBits(String addr)

// Calculated Attrs Getters

//This one returns the hit rate of the cache

public double getHitRate()

public double getMissRate()

// The miss rate is the number of totalMisses / TotalCacheAccesses

public double getTotalCacheAccesses() {

// The total number of cache accesses is the number of hits + the number of misses

Memory: represents the main memory

* Attributes:

String [] memory; // Main memory array, length should be 2^16

private int accessTime; // Input: Number of cycles to access

private int totalCycles; // Output: Total number of cycles in the current stimulation.

private boolean isBeingAccessed; // tracking memory state

private int fetchCyclesRemaining; //tracks cycles remaining to fetch instruction

private int dataAccessCyclesRemaining; //tracks cycles remaining to access data

* Functions:

// Takes an int address as an input and returns the data in the memory location associated with it.

public String read(int address)

// Takes an int address as an input and a string data, writes the data to the memory location of index address.

public void write(int address, String data)

Memory Hierarchy: binds the cache and memory together and construct the memory hierarchy class to be dealt with.

* Attributes:

Public Cache [] caches; holds the caches of the microprocessor

Public Memory memory ; holds the main memory

* Functions:

/\* This one takes 2 String address and data as inputs

\* It first checks if the data is cached in the top level cache if so, it writes and keeps on writing to the cache levels below until a cache with WB policy is encountered

\* If the data is not cached in the first level, the data is first read and cached in the lower level caches

\*/

public void write(String address, String data)

/\*

\* This one takes an address in string form, it returns the index of the cache level where the data resides

\* If the data resides in the memory 1+ number of cache levels is returned to indicate main memory

\* This function does not do caching in anyway, it is a simple check for Tomasulo calculations

\* If the address is invalid i.e: it is not in the memory -1 1 is returned.

\*/

public int getCorrespondingCacheLevel(String address)

/\* This one takes an address in string form

\* Returns A string represents the block where the byte resides in within cache level 1

\* The Data is returned only when the cycles required to access it are finished

\* other wise the method returns null since the data was not yet accessed

\*/

public String loadData(String address)

/\*

\* This one takes a string address as an input and a cache level.

\* it returns the number of cycles remaining to retrieve the data from the cache

\* or ZERO if the caching is done, Its also responsible for caching the item

\* If you want to fetch data from memory to cache level 3, you'd just wait for getCacheCyclesRemaining(4, address)

\* to return 0, then the data would successfuly be in level 3 cache

\*

\*/

public int getCacheCyclesRemaining(int cacheLevel, String address)

/\* This one takes an address in string form

\* It returns the Data associated with that address

\* The method works as follows, It loops through the caches in a non decreasing order till it gets a hit, or it accesses the memory

\* Once The above condition is satisfied, the data is cached to the lower cache in the caches list

\* Then the loop goes back to that cache, and cashes the data in the cache preceiding it. This goes on till the level one cache has the data cached in it

\* The data is then returned from the level one cache as planned

\*/

private String readAndCacheData(String address)

/\*

\* This one takes inputs an integer representing the cache index in the array of THIS instance, A string representing the address to read

\* and a boolean representing weather the block returned is an instruction

\* It reads the data corresponding to this address from the current cache and returns it in a block compatible with the cache below it

\* It returns a block compatible with the cache below this cache

\*/

private Block readFromCacheBelow(int cacheIndex, String address, boolean isInstruction)

/\* This one takes an address in string form and returns the block corresponding to that address from the Main memory

\* The block returned has to be the same size as the last cache level's line size to be able to insert it there

\* So the memory is to be divided into memorySize/lineSize Data Blocks to match the last level.

\*/

private Block readFromMemory(String address)

/\* This one takes the cache level and 2 strings one representing the address and the other representing the data

\* It writes the data in the block corresponding to the address in the current cache

\* If the hit write policy is write through a recursive call takes place to continue writing to the lower level cache,

\* until it either writes in the memory or encounters a cache with write back policy

\*/

private void writeToCacheLevel(int cacheLevel, String address, String data)

/\* This one takes as input a block to write and a string representation of the index bits and an integer specifying the cache level

\* It writes the given block in the cache, The replacement policy used is random replacement

\* If a block is dirty, it acts according to the write police write through or write back

\*/

private void writeBlock(Block blockToWrite, String indexBits, int cacheLevel)

/\* This one takes as input a block toReplace and a string representing its address, also an integer representing the cache level to interactr with

\* The Replacement policy used is random replacement, If the block toReplace is dirty, It is responsible for writing the data using

\* the specified writing policy for each cache

\*/

private void replaceBlock(Block blockToReplace, String blockToReplaceAddress, int cacheLevel)

/\* This one takes an address in string form

\* It returns the Data associated with that address

\* This one works the same ways as readAndCacheData, it loops over all the caches till it finds a hit,

\* and loops back to refill the caches where it missed

\*/

private  String readAndCacheInstruction(String address)

/\* This one takes an address in string form

\* It returns the Data associated with that address when it can be returned, or null otherwise

\* Once it is fetched it is returned and the function loops back to refill the caches where it missed

\*/

public String fetchInstruction(String address)

*Assembler:*

*The assembler class.*

*Assembles an assembly program file into machine instructions stored in the memory, also interacts with the user for pre-configurations.*

* Features:
  + Case insensitive.
  + Supports comments.
  + Supports labels.
  + Gives syntax error approximate location in case of err
  + Follows the default assembly syntax
  + Space insensitive
  + Uses JSON to interact with the user regarding configurations
* Functions:

/\*

\* This one takes an assembly instruction in string form, case insensitive and space insensitive

\* Returns A string containing the corresponding binary machine code in 16 bits

\* Exceptions thrown: illegal argument exception if the register entered is not between reg0 and reg7 inclusive

\* or if the instruction is not the microprocessor's instruction set

\*

\* Instructions Go as follows

\* lw: 000 , sw 001, jmp 010, beq 011, jalr 100,  ret 101, addi 110, Arithmetic 111

\* For arithmetic instrction the op codes are:  add: 0000 , sub: 0001, mul: 0010, nand 0011

\*/

public String assemble(String instruction)

*Stimulation Runner:*

This class is responsible for starting the stimulation, it parses the user input to initialize the hardware and then calls the assembler to assemble the program code into the memory then it starts the stimulation.

* Functions:

public void run()

The run function calls other helper functions to initialize the stimulation then runs it.

*Tomasulo:*

* Attributes: