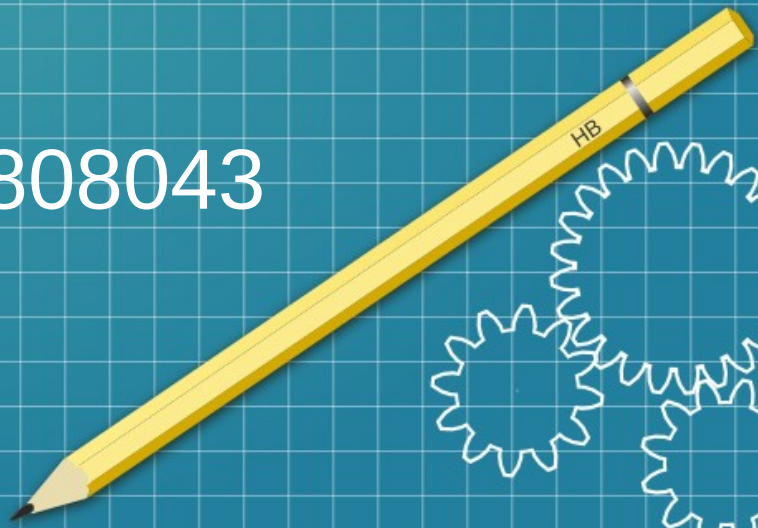
CSE206 CORG

CPU Emulator

Teacher: Taner Danışman

Student: Ali Fuat Akyemiş-20210808043

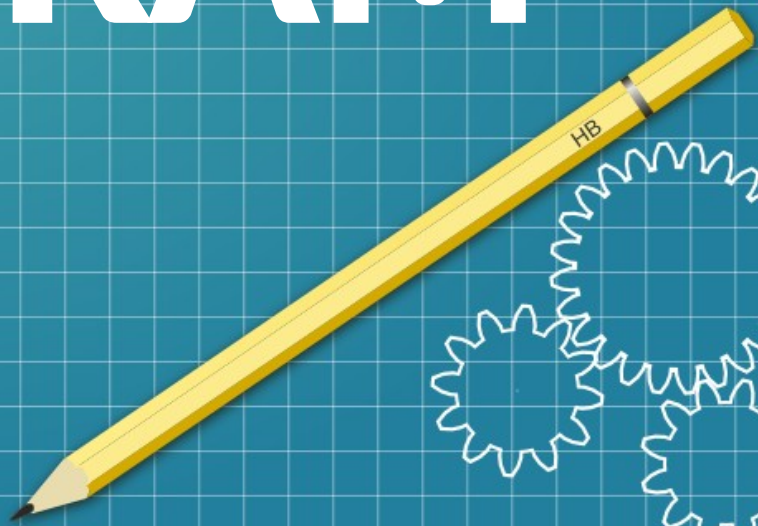


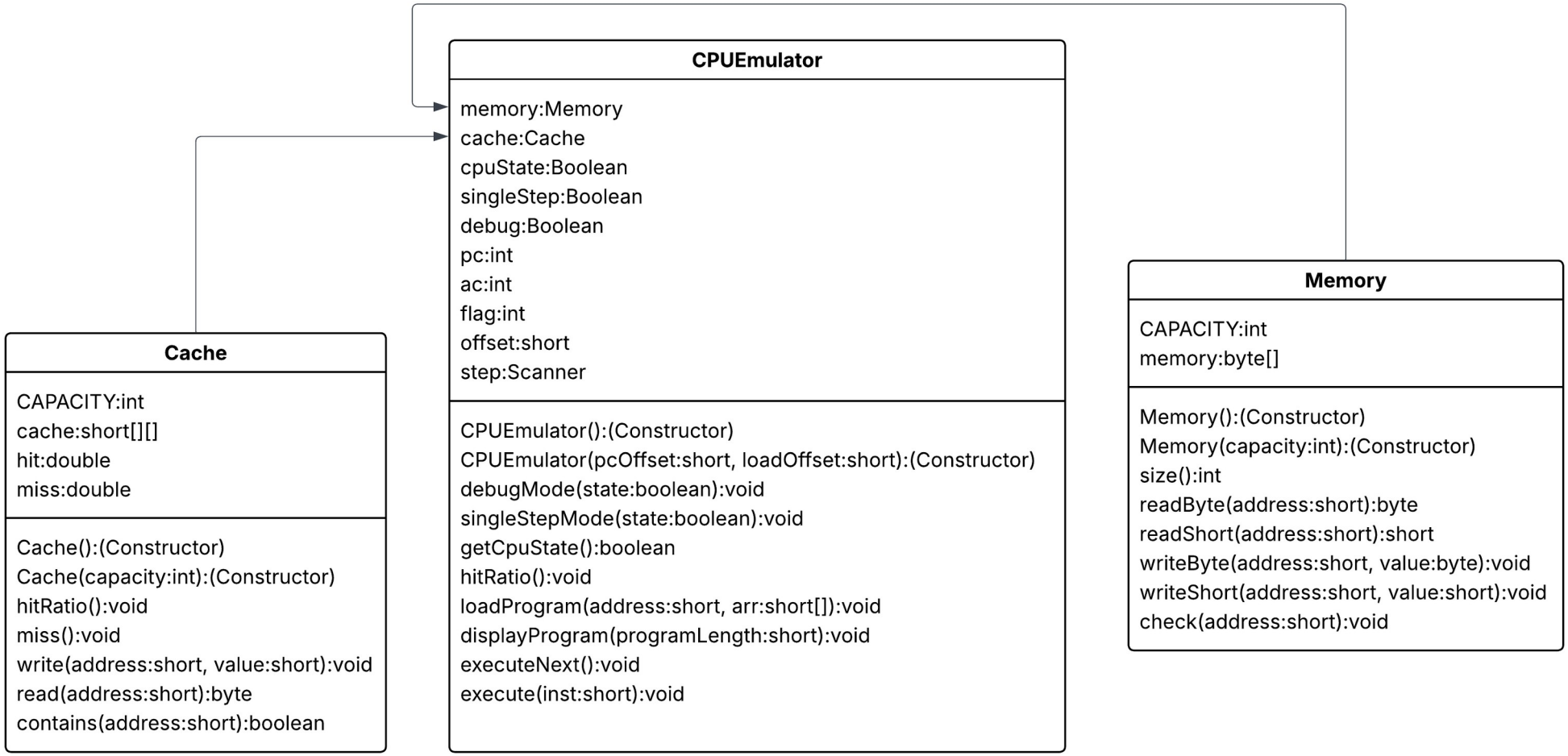
Summary

- Code structure
- Explanation of functionality
- Fixed bugs during designing

CODE STRUCTURE

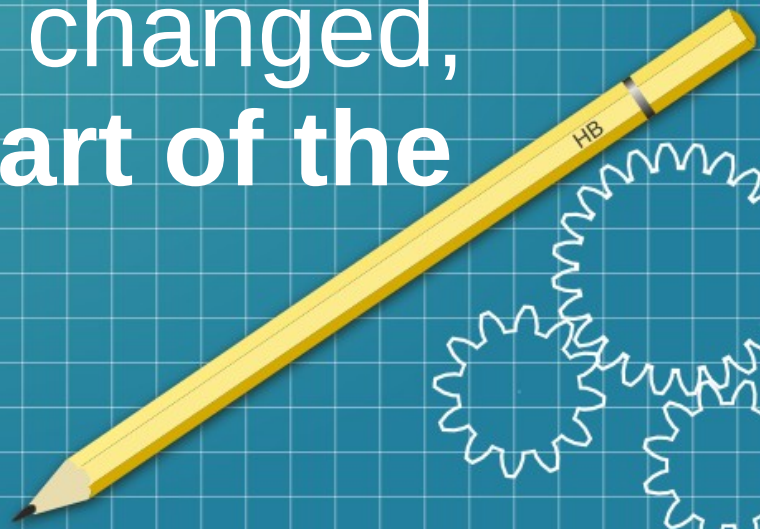
UML DIAGRAM




CPUEmulator
memory:Memory
cache:Cache
cpuState:Boolean
singleStep:Boolean
debug:Boolean
pc:int
ac:int
flag:int
offset:short
step:Scanner
CPUEmulator():(Constructor)
CPUEmulator(pcOffset:short, loadOffset:short):(Constructor)
debugMode(state:boolean):void
singleStepMode(state:boolean):void
getCpuState():boolean
hitRatio():void
loadProgram(address:short, arr:short[]):void
displayProgram(programLength:short):void
executeNext():void
execute(inst:short):void
Cache
CAPACITY:int
cache:short[][]
hit:double
miss:double
Cache():(Constructor)
Cache(capacity:int):(Constructor)
hitRatio():void
miss():void
write(address:short, value:short):void
read(address:short):byte
contains(address:short):boolean
Memory
CAPACITY:int
memory:byte[]
Memory():(Constructor)
Memory(capacity:int):(Constructor)
size():int
readByte(address:short):byte
readShort(address:short):short
writeByte(address:short, value:byte):void
writeShort(address:short, value:short):void
check(address:short):void

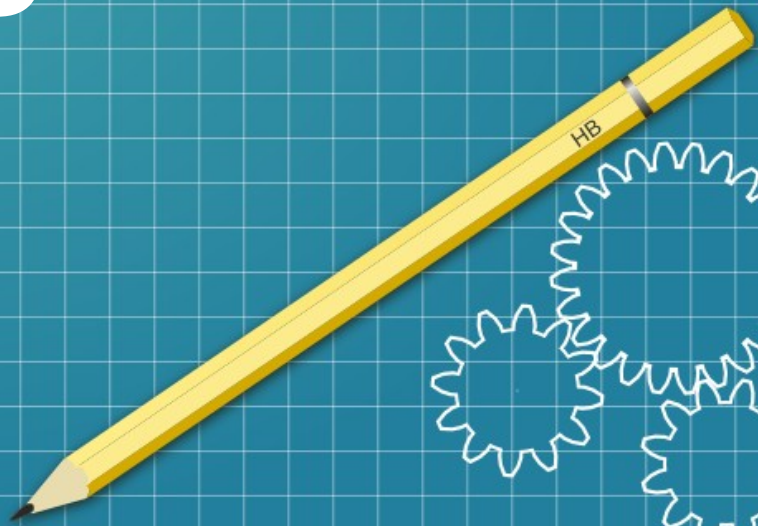
CODE STRUCTURE

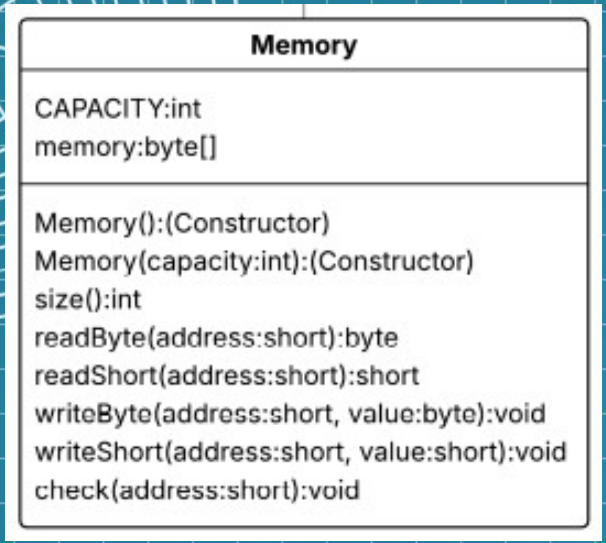
a120210808043.java hasn’t been added to UML Diagram because it is **just the place** that the **emulator started**. Since it can be changed, Main.java is **not the heart of the system.**



EXPLANATIN OF FUNCTIONALITY

Memory.java

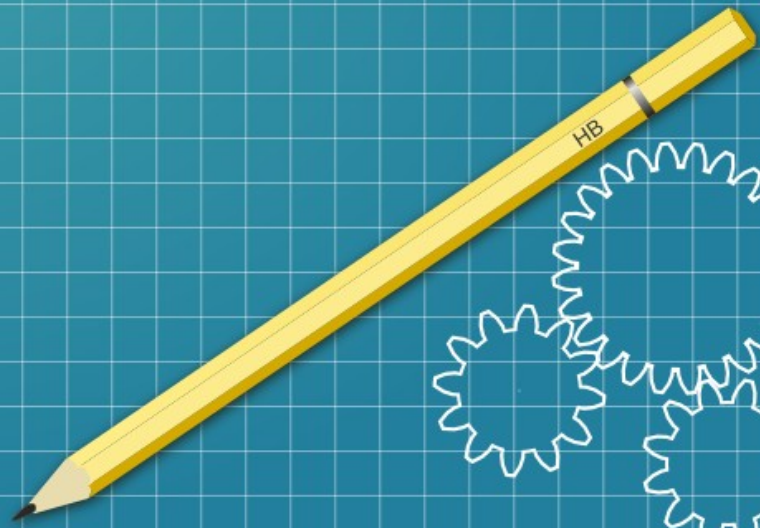


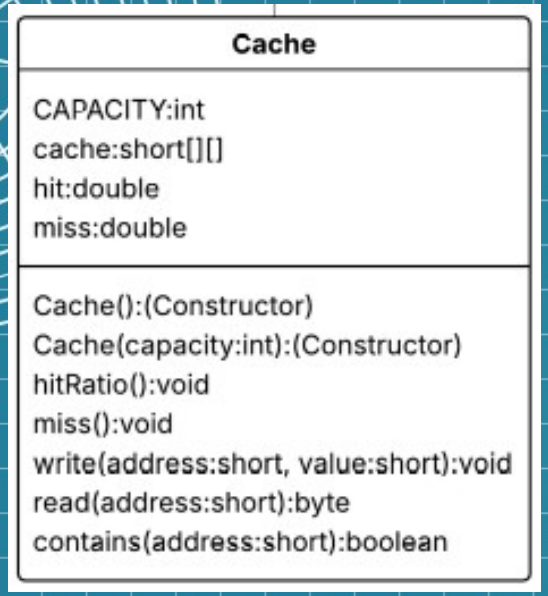


- Since the memory is **byte-addressable**, I prefered to hold data in a **byte array** because it is the most compatible abstraction for byte-addressable structure
- For the **read** and **write** operations, they have 2 options each. Because in most cases (like caching) memory is needed to be read as **block** (since the block size is 2 byte it means short) and several instructions need to access the **bytes** of the memory directly.
- There is an additional address check method to check physical address.

EXPLANATIN OF FUNCTIONALITY

Cache.java

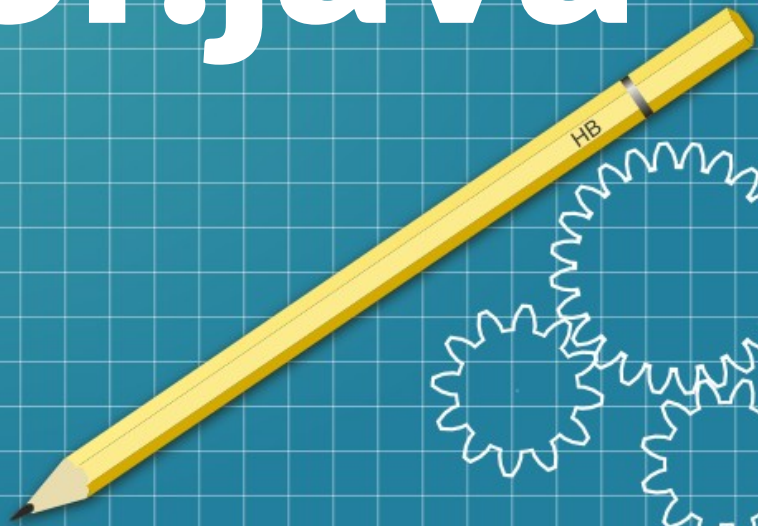


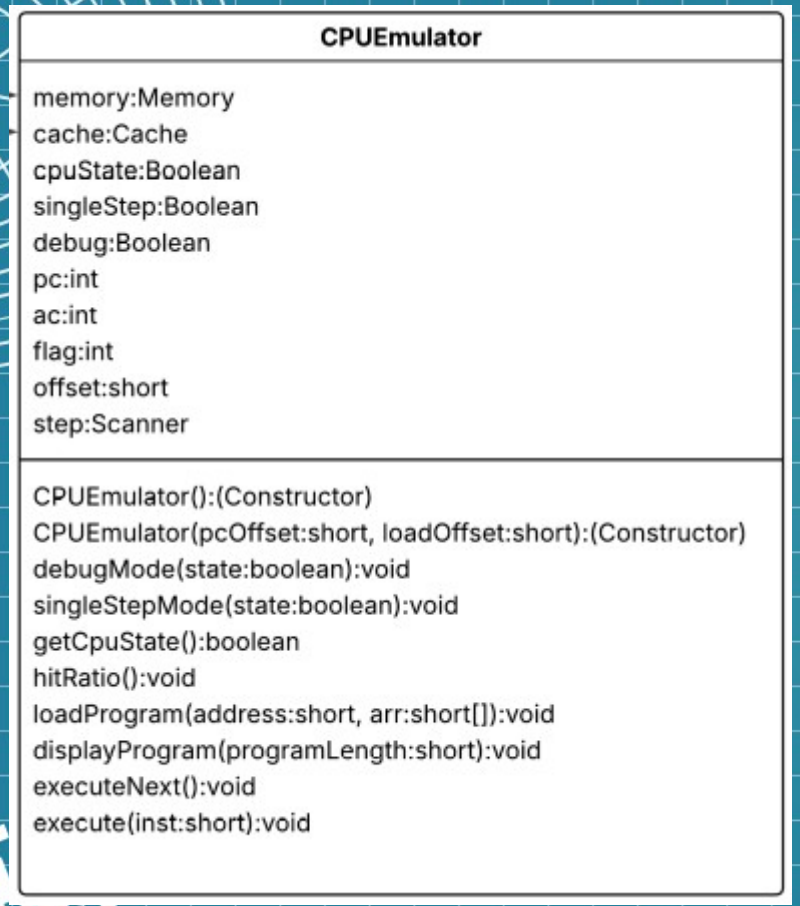


- In the cache memory I used 2D short array to hold memory blocks and their tag bits at the same time. Using 2D arrays made it easier.
- Default read-write methods that works in the same way with memory.(write-through feature is handled in the CPUEmuator.java)
- Additionally there is a **miss()** method that increments miss count for store instruction exceptionally.(this method is only used for store instruction)

EXPLANATIN OF FUNCTIONALITY

CPUEmulator.java

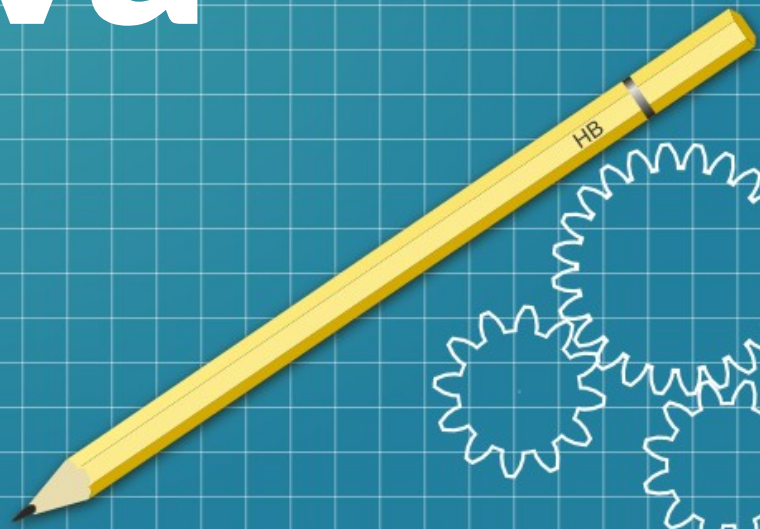




- **Memory** and **Cache** instances is being held in this part
- CPU has **single-step** mod that is provided with **java.util.Scanner**.(controlling by **singleStep:boolean**)
- Also a **debug** mod has been added to structure. (controlling by **debug:boolean**)
- **Program loading** operation is being executed during running the **Main.java**
- There is an additional **displayProgram()** method to debug memory program loading operation.

EXPLANATIN OF FUNCTIONALITY

Main.java



```
//-------------------------------------------------------
//initial pc and load address extraction
BufferedReader br = new BufferedReader(new FileReader(config));

int a = Integer.valueOf(br.readLine().substring(2).trim(), 16);
int b = Integer.valueOf(br.readLine().substring(2).trim(), 16);

short loadAddress = (short) a;
cpu = new CPUEmulator((short) b, (short) a);

br.close();
//-------------------------------------------------------
```

- First step is getting config.txt informations and initializing the CPUEmulator instance

```
//-------------------------------------------------------
//Determining the length of instruction file
br = new BufferedReader(new FileReader(filename));

String str = br.readLine();
int n = 0;

while (str != null) {
        n++;
        str = br.readLine();
}

br.close();
//-------------------------------------------------------
```

- Determining the length of the **program.txt** (count of the instructions. This is needed to load program to the memory)

```
//-------------------------------------------------------
//Putting all the instructions in a short array
br = new BufferedReader(new FileReader(filename));
short[] arr = new short[n];

for (int i = 0; i < n; i++) {
        int instruction = Integer.valueOf(br.readLine().trim(), 2);
        arr[i] = (short) instruction;
}

cpu.loadProgram(loadAddress, arr); //Program loading
br.close();
//-------------------------------------------------------
```

- Here I defined a short array to give as a parameter to **loadProgram() method(instruction count is used for this definition)**

```
//-------------------------------------------------------
//Checking the debug and single-step options
if (args.length > 2) cpu.debugMode(Boolean.parseBoolean(args[2]));
if (args.length > 3) cpu.singleStepMode(Boolean.parseBoolean(args[3]));
//-------------------------------------------------------
```

- This step is just about checking the starting options, if the user wants to **debug** or activate **single-step** mod

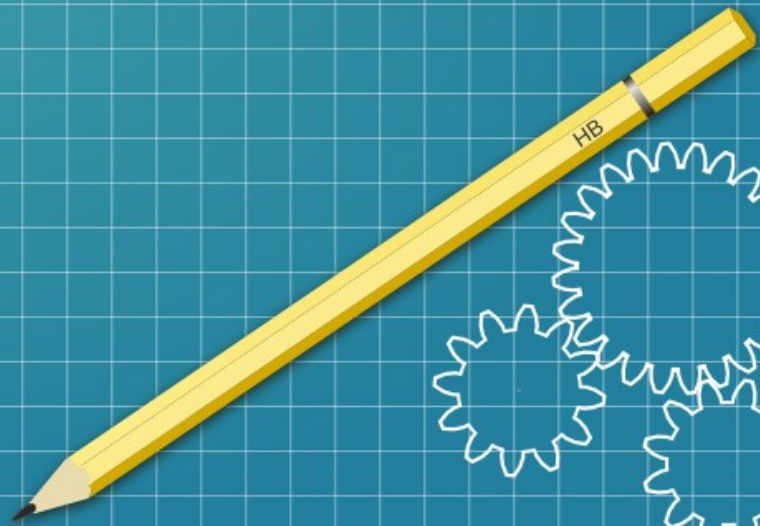
```
//-------------------------------------------------------
//Execution phase
do {
        cpu.executeNext();
} while (cpu.getCpuState());

cpu.hitRatio(); //Hit ratio
//-------------------------------------------------------
new Scanner(System.in).next();
```

- The last section is about execution according to **cpuState.**

- **cpuState** is also being checked in the CPUEmulator.java but **execution loop** placed in the Main.java so, this is just a **design preference.**

EXPLANATIN OF FUNCTIONALITY

run.sh

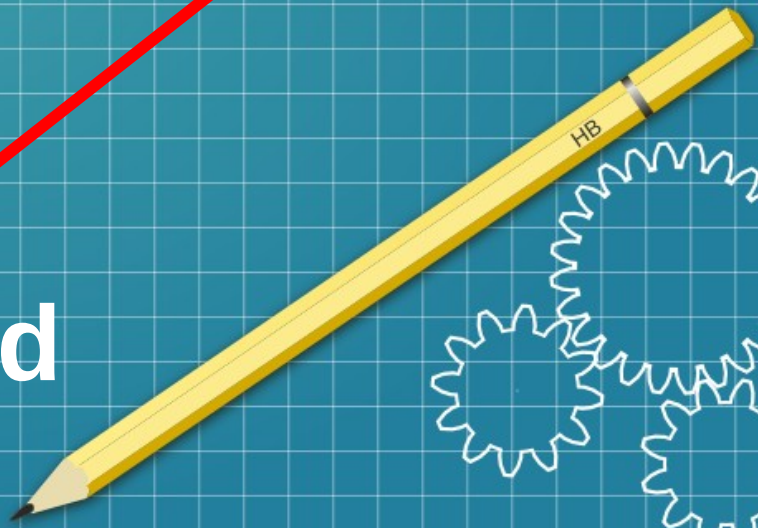


run.sh just compiles emulator Java files to bin directory
and runs the main program according to parameters

```
javac -d bin emulator/*.java
java -cp bin a120210808043 program.txt config.txt false false
```

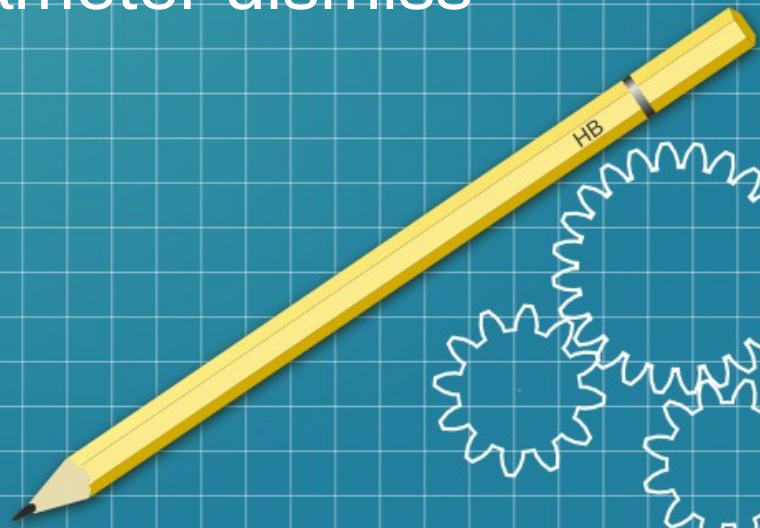
Debug mod

Single-step mod



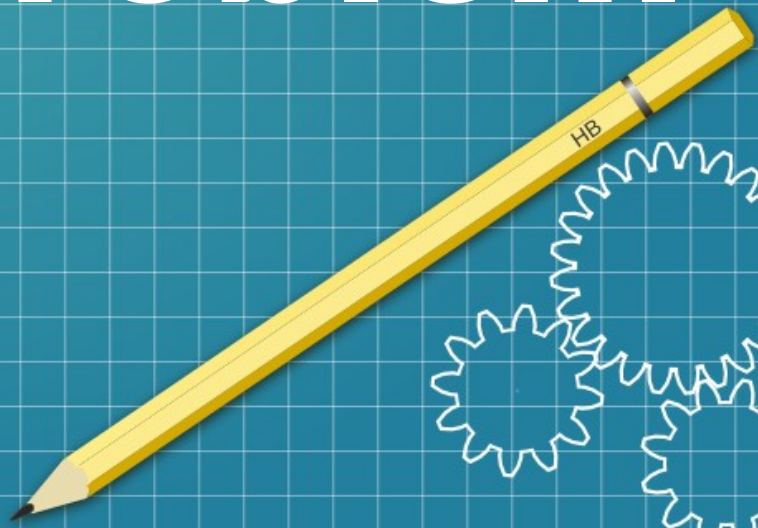
```
javac -d bin emulator/*.java
java -cp bin a120210808043 program.txt config.txt
```

They are not essential parameters.
CPU has default states due to parameter dismiss

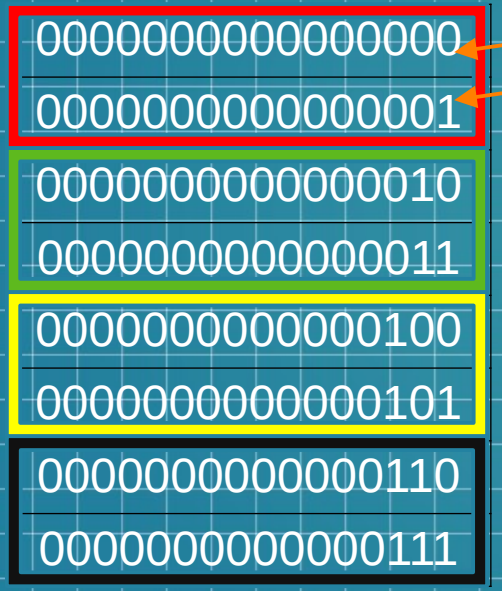


FIXED BUGS

Block Fetch Problem



Every memory access causes caching (fetching a block to cache) and memory block should be grouped like this:



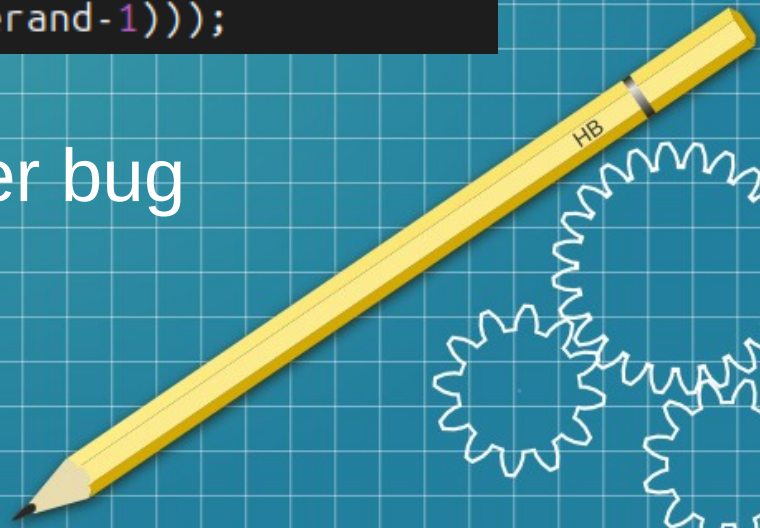
Group order should be always like this
1.Even Address
2.Odd Address

If you did not consider this ordering probably your code usually gets into infinite loop

To fix this problem caching had been done with this:

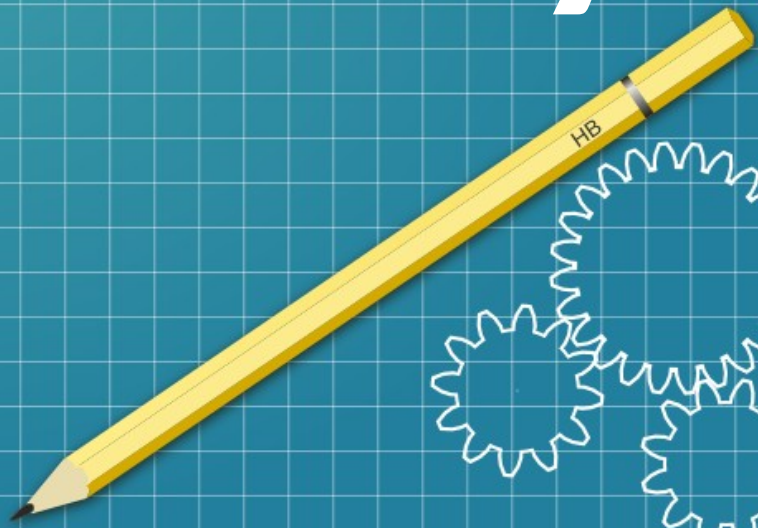
```
if (operand%2 == 0) cache.write((short) operand, memory.readShort(operand));
else cache.write(operand, memory.readShort((short) (operand-1)));
```

This if block solves the block order bug



FIXED BUGS

Hit Ratio Accuracy



Actually this is not completely a bug but causes **enormously high hit ratios**

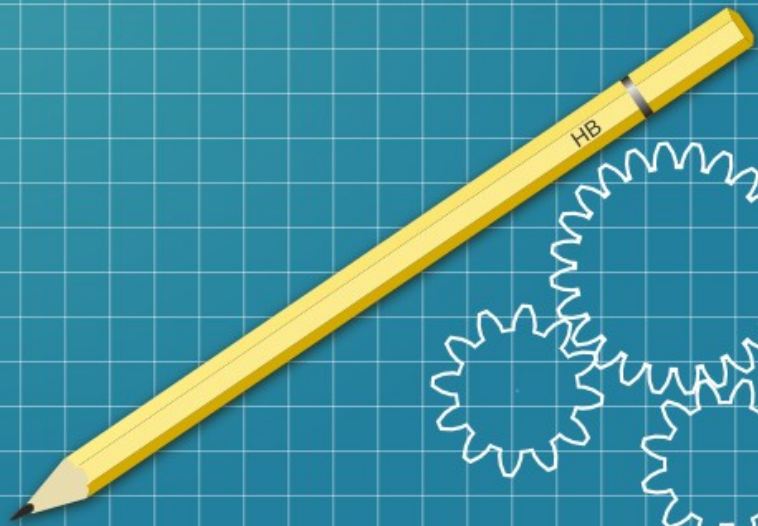
As I explained in the **functionality part** Cache.java has **miss()** function for **store** instruction exceptionally because store operation **guarantiees** the accessing to memory. And there should be a cache miss but if we use the same cache write function **it can count a hit**. So **exceptionally** miss() function increments miss count to solve this problem

```
Accumulator: 210
Hit ratio : 100.0
```

```
Accumulator: 210
Hit ratio : 65.87301587301587
```

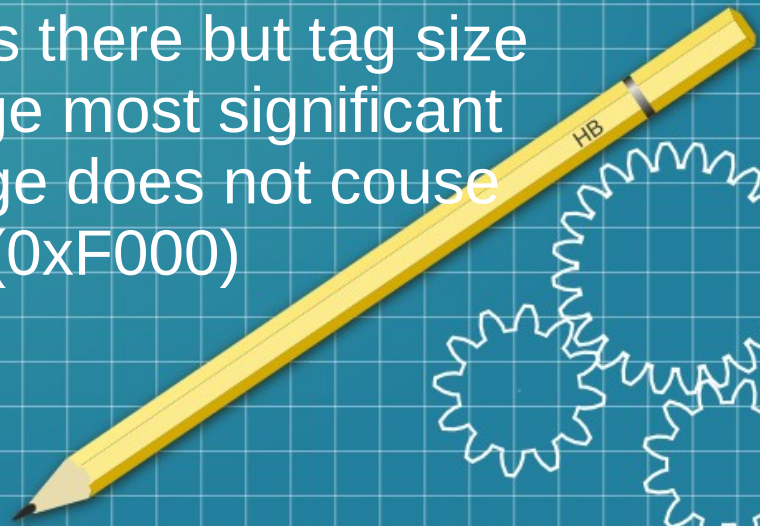
FIXED BUGS

Ghost Cache Hit



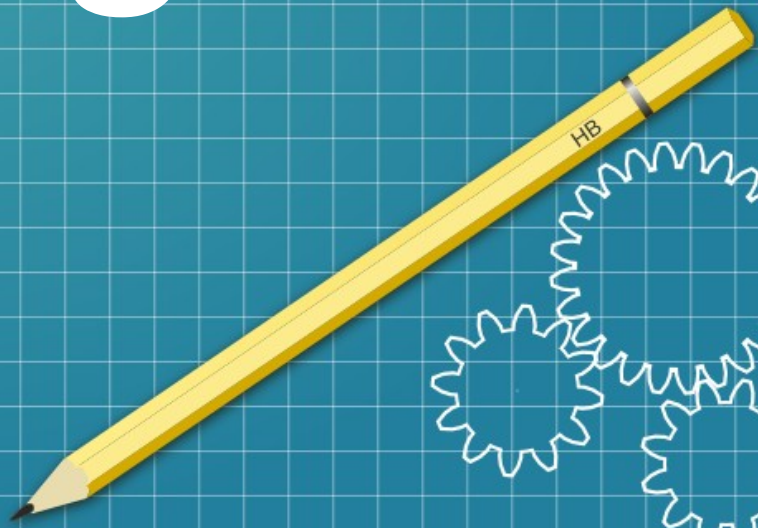
This is caused by cache array.
Since it is a 2D short array its initial value is 0 in every cell. So when the program went on (0000000000000xxxx) type of addresses, there can be many ghost cache hit unexpectedly.

This problem is solved by modifying tag cell values as 0xF000. Tag cell size is short size so 16 bit number is there but tag size is only using 12 bits.This allows us to change most significant 4 bit to prevent ghost cache hit. That change does not couse any conflict with physical tag size.(0xF000)



FIXED BUGS

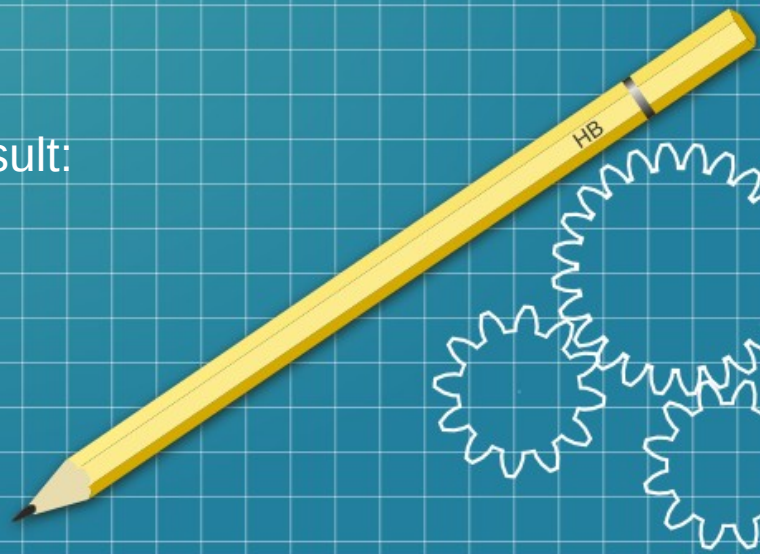
Signed-Unsigned



In Java the is no unsigned numbering for integers (It is general actually) and in some cases it caused problems.

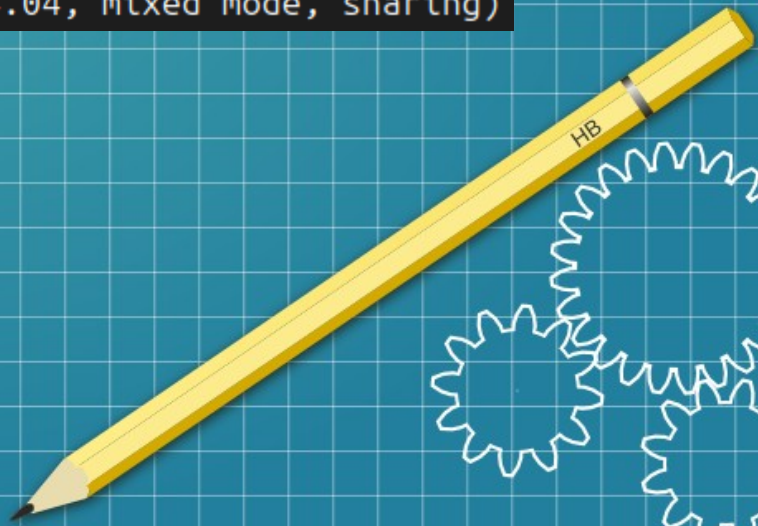
```
//Determining the opcode and operand
byte opcode = (byte) ((inst & 0xF000) >>> 12);
short operand = (short) (inst & 0xFFF);
```

As you can see “>>>” operater is used for bitwise right shift operation. This was used as “>>” operator. Since “>>” keeps the sign.
In some cases like halt operation,
(1110xxxxxxxxxxx)
it results this:
(1111111111111110)
so “>>>” operator(zero fill) solves this problem result:
(0000000000001110).



Finally here is the Java version:

```
openjdk 21.0.7 2025-04-15
OpenJDK Runtime Environment (build 21.0.7+6-Ubuntu-0ubuntu124.04)
OpenJDK 64-Bit Server VM (build 21.0.7+6-Ubuntu-0ubuntu124.04, mixed mode, sharing)
```



THANK YOU