



CPU Simulator in Java - Final Year Project

Daniel Alexandre
NO. 2727225

2011/2012

BEng (HONS) Computer Systems and Networks 2388

Table of Contents

1. Abstract	- page 3;
2. Introduction	- page 3;
3. Project planning	- page 10;
4. Experimental procedure	- page 11;
5. Results	- page 12;
6. Discussion and conclusion	- page 54;
7. List of figures and tables	- page 55;
8. References	- page 57;
9. Appendices	- page 59;

1. ABSTRACT

The purpose of this report is to explain the background theory of a CPU and to show its implementation through a Java project (named “CPU Simulator”). Its main goal is to emulate the work of a CPU while interacting with the user. It contains user interface which communicates with the CPU model (named “CPU16Model”). The emulated CPU receives instructions from the user, until the main thread is interrupted. Significance of the report is that it can be used by students and people, interested in this computer science area and making their first steps in understanding and implementing a CPU.

2. INTRODUCTION

The CPU itself is divided into three sections: data registers (doing memory references and faster access to the addresses and calculations), arithmetic logic unit (ALU), and the control unit (CU). It is interacting with different devices – like display, keyboard and memory card. The main “work unit” of the CPU is called “instruction”, consisting of an operation code and operands. CU extracts instructions from memory and decodes and executes them, calling on the ALU when necessary. The topic of this report is to show the way the system was implemented. First some background theory is needed.

Basic scheme of computer model:

Figure 1

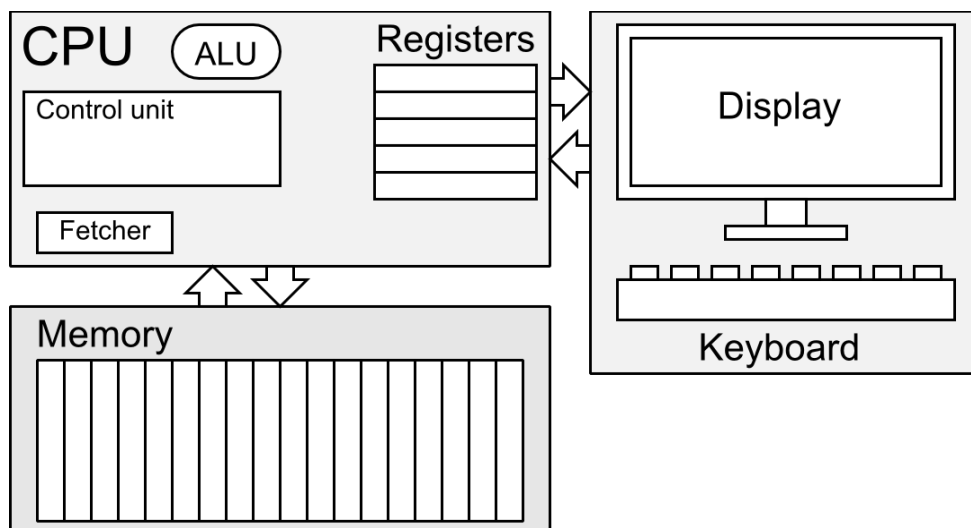


Table of instructions:

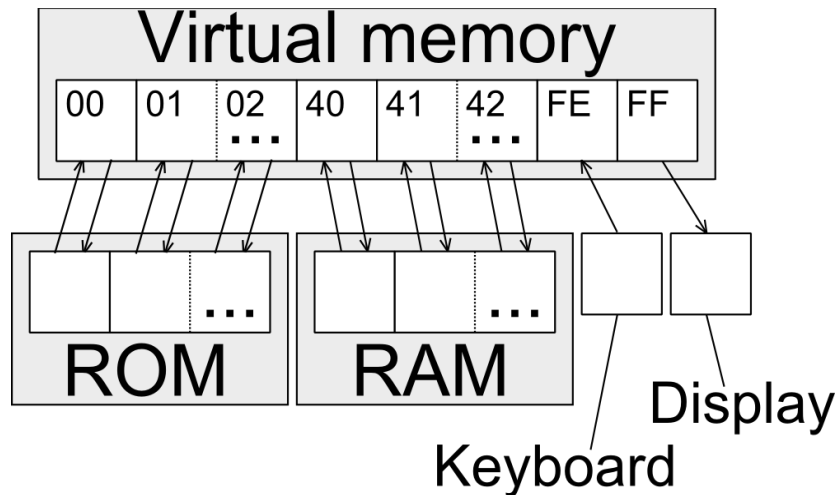
Table 1

0 th	1 st	2 nd	3 rd	Name	Instruction description
0				HALT	Stops program execution.
1	Ra	ADDR		LOAD	Reads data from virtual memory by ADDR address and puts it into Ra
2	ADDR		Ra	STORE	Writes data from Ra register to virtual memory cell with ADDR index.
3	Ra	Rb	Rc	ADDI	Adds integer numbers stored in Rb and Rc registers and writes a result to
4	Ra	Rb	Rc	ADDF	Not yet implemented.
5	Ra	Rb		MOVE	Copies data from Rb register to Ra.
6	Ra	Rb		NOT	Writes to Ra register negative value of Rb.
7	Ra	Rb	Rc	AND	Performs logic AND to Rb and Rc, then writes result to Ra register.
8	Ra	Rb	Rc	OR	Performs logic OR to Rb and Rc, then writes result to Ra register.
9	Ra	Rb	Rc	XOR	Performs logic XOR to Rb and Rc, then writes result to Ra register.
A	Ra			INC	Adds 1 to integer value in register Ra and stores result into Ra.
B	Ra			DEC	Subtracts 1 from integer value in register Ra and stores result into Ra.
C	Ra	N	D	ROTATE	Performs a rotate to the Ra register N times in the D direction.
D	Ra	AD	JU	If Ra isn't	
E					Unused.
F					Unused.

Memory management.

Our virtual memory will have 256 cells, each of which is 16 bit sized. Each cell of VM can have attached or unattached device; same cell can't be attached to device twice or be attached to more than one device at the same time.

Figure 2



Our computer model will have these attachments:

Table 2

Range	Siz	Device
0x00-0x3F	64	Read only memory, used to hold program code.
0x40-	19	Random access memory, used to hold program variables.
0xFE	1	Keyboard device, changes memory cell when user enters data.
0xFF	1	Display device that displays number written to this memory cell.

Devices description.

Memory. Memory device can have any specified amount of cells, each of which attaches to virtual memory. As described above, our application will have one ROM device with 64 cells and one RAM device with 190 memory cells. Each memory cell attaches to corresponding virtual memory cell from given attachment offset.

There is a modified type of memory device - ROM. Read only memory used to store code and program constants (or unchangeable variables). Writing to the ROM will cause an error: execution will halt and error message will be shown. ROM is used to protect application code from corruption when program execution becomes uncontrollable because of program mistakes. Our ROM isn't programmable (write allowed) by program and is programmable by simulator application's data manager and code compiler, our ROM behaves as EEPROM.

Keyboard. It's a read-only device used to let simulation application user input data via keyboard. Device has one attachment read-only cell. Attempt to write data to keyboard cell will be ignored.

We assume that user enters only signed integer numbers from -32767 to 32767 (from $-2^{15}-1$ to $2^{15}-1$ inclusive); entered number will be taken by program. Other user data will be considered as illegal and when typed will cause error and print message to user.

There is two considered modes of keyboard:

With delay – when program reads number from attached keyboard, CPU stops running and waits for entered data. This model can be useful for one-by-one computing schema and simpler to implement in assembler language.

Without delay – when program retrieves info from attached keyboard, program gets -2^{15} value, if there is no number was entered or value in $[-32767; 32767]$ range that is entered integer value. W/o delay model can be used in multi- task computing schema, unlike W/ delay schema, which cann't be used in multitask computing.

User can switch keyboard mode according to program algorithm, using GUI.

Display. It's used to make programs print information to user. Display have 1 write only cell, writing to which will make written integer number visible on the display. Reading data from connected cell will retrieve 0 to program.

Assembler language.

To make developing programs easier, we have to introduce our own assembler language.

First of all, we should make decisions:

Our assembler language will have instructions to identify and parameterize each machine operations and labels to make flow control easier.

To make computer model simpler, we will have no code segmentation in assembler language. All code will be stored in the ROM after compilation.

Assembler commands will have this syntax:

LabelName: InstructionName OPERAND1, OPERAND2, OPERAND3

There is any spaces and tabs count can be placed before and/or after instruction name, operands, label.

Label is optional in any case, operands is optional depend on instruction name. New line symbol finishes the command, so when compiler will see new line symbol, it will decide to start compiling of another command.

Instruction can be named by one of elements from Name column of Table 1: HALT, READ, WRITE, MOVE, ADDI, AND, OR, XOR, NOT, INC, DEC, ROTATE or JUMP.

Operands can be one of these types:

Register operand: R0..R19;

Integer value operand:

- Hexadecimal representation: CAFEh;
- Binary representation: 10100101b;
- Decimal representation: 255, 37;

Code address operand:

- Representation by label name: LABEL_1, START, END, CYCLE_1;
- Representation by absolute code address: #<integer>, for example #0, #0x15, #35;

Memory address operand: [<integer>], for example [FEh].

Examples of assembler instructions:

- HALT (have no parameters, causes CPU to stop executing)
- READ R2, [FEh] (reads one number from keyboard and puts it to the R2 register)
- WRITE [FFh], R2 (prints number in the R2 to the display)
- MOVE R3, R2 (copies data from R2 register to R3)
- ADDI R1, R2, R3 (performs $R1 := R2 + R3$)
- AND R1, R2, R3 (performs $R1 := R2 \text{ and } R3$)
- OR R1, R2, R3 (performs $R1 := R2 \text{ or } R3$)
- XOR R1, R2, R3 (performs $R1 := R2 \text{ xor } R3$)

- NOT R1, R1 (performs $R1 := \text{not } R1$)
- INC R4 (performs $R4 := R4 + 1$)
- DEC R5 (performs $R5 := R5 - 1$)
- ROTATE R2, 4, 0 (performs rotate to the left operation 4 times to the R2 register, in other words, this operation will multiply R2 by $2^4=16$)
- ROTATE R3, 2, 1 (performs rotate to the right operation 2 times to the R3 register, in other words, this operation will divide R3 by $2^2=4$)
- JUMP R3, ERROR (jumps to ERROR label offset if R3 register is not equals to R0)
- JUMP R1, #0 (jumps to the start of the code if R1 register isn't equal to R0)

As we have no special NOP instruction (“no operation” instruction, used to delay program execution), there is a some variants to make a NOP:

- AND R1, R1, R1 (and operation on the same register will not cause any changes)
- OR R2, R2, R2 (or operation on the same register willn't cause any changes like an AND)
- ROTATE R3, 0, 0 (rotation in any direction also leaves machine state unchanged)
- JUMP R0, #0 (R0 always equals to itself, so jump will never be performed)
- INC R1; DEC R1; (two-instruction delay, will return machine to it's previous state)
- NOT R1, R1; NOT R1, R1; (another two-instruction delay)

Instruction set. The implemented computer model is having a set of sixteen instructions, although mainly will be used just fourteen of them. Each computer instruction consists of two parts: the operation code and the operand(s). The operation code specifies the type of operation to be performed on the operand(s). Each instruction consists of sixteen bits divided into four 4-bit fields. The leftmost field contains the opcode and the other three fields contains the operand or address of operand(s). [8]

Processing the instructions. It is one of the most important parts of the current Java project. In fact that is the real work of the CPU. This simulated system (as most of the real systems) uses the so called “machine cycles”. Each cycle is one full processing of an instruction. It is made of three phases: *fetch*, *decode*, and *execute*. During the *fetch* phase, the instruction whose address is determined by the PC is obtained from the memory and loaded into the IR. The PC is then incremented to point to the next instruction (it will be processed after the current instruction is executed). During the *decode* phase, the instruction in IR is decoded and the required operands are fetched from the register or from memory. During the *execute* phase, the instruction is executed and the results are placed in the appropriate memory location or the register. Once the third phase is completed, the control unit starts the cycle again, but now the PC is pointing to the next instruction. The process continues until the CPU reaches a HALT instruction. For more information (including a practical example) – see [8].

3. Project Planning

This project has to demonstrate to the user how an exemplary CPU is working. So generally it has to consist of a user interface part and “technical” part – where the full implementation is made.

The user interface includes visualization of all the implemented CPU's processes. The user can choose the concrete details of the simulation and start it – using the “Run” command. This is creating a thread where the CPU's work is wrapped in. The simulation ends with interrupting the process and showing the final results.

The “simulation” part consists of the full implementation of the CPU's process, the way it is described in the theoretical background here. It includes the CPU model (named “CPU16Model”) which works as a cooperation (using concrete rules) between the elements of the PC.

The coding plan of the project consists of several steps:

- first, you have to determine the key points of the whole system – i.e. to describe the devices which are interacting with each other; The very basic level is to create the needed interfaces. After that they have to be implemented by the system elements (i.e. the devices).
- Second, you have to create the real model of work of the whole system. That means to implement the interaction rules between the elements which we already created;
- the last part is to create a user interface and to make an integration between it and the model part. The simulation will use a created thread.

4. EXPERIMENTAL PROCEDURE

Coding the current CPU simulator passed through different phases:

First, it have to be coded the very basic interfaces. The core one is “Memory16” - cause all the operations are released there. The next one is “Device” - cause the instructions of the CPU are about the different devices. Here it is needed to define the abstract class “Breakpoint16” also.

Second, it have to be implemented the interfaces – the different kind of memories (BreakpointMemory16, DataMemory16, InstructionMemory16, VirtualMemory16) and devices (Display, Keyboard, MemoryCard).

Third, it have to be coded the “Operand” and “Instruction” classes. Each instruction use some operands, so the logical order is this.

Fourth, it have to be defined the “AssemblyException”.

Now all the devices and the possible operations between them are defined. We are ready to continue will implementing the whole model.

Fifth, it have to be implemented the model - “CPU16Model” and “CodeManager”.

Sixth, it is time to release the user interface and to connect it with the model. The main one is “MainFrame”, but also have to code the “AsmSourceDialog” and the “InfoMessageDialog”.

For more information about the behavior and the meaning of each single class – please check Appendix 1 at page 59.

5. RESULTS

This section contains all the checked test cases with an explanation of their meaning, expected results and outcome.

BlackBox Test Case:

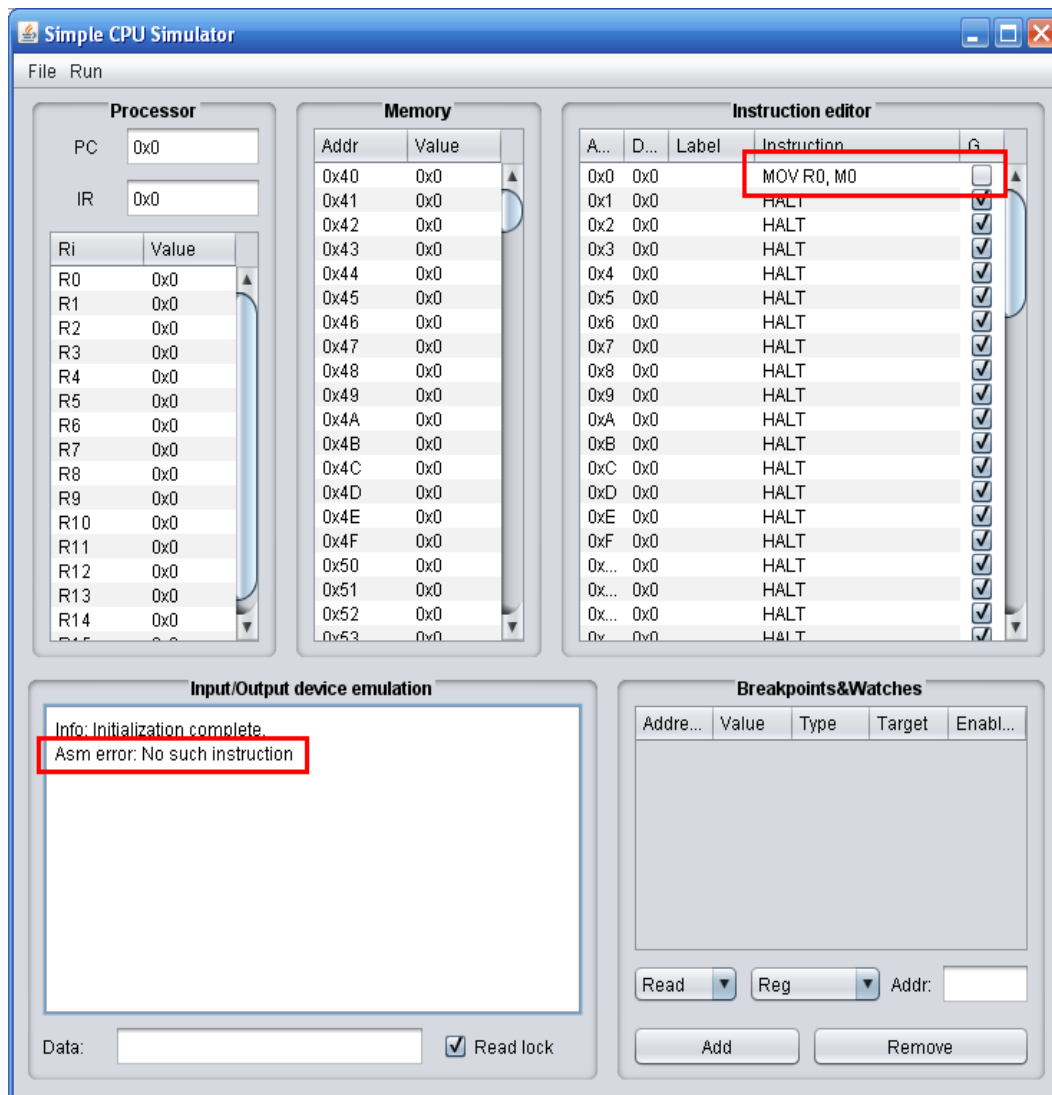
Invalid instruction set inputted:

Test Case 1:

MOV R0, M0 //Invalid Input

HALT

Figure 3:



Test Case 2:

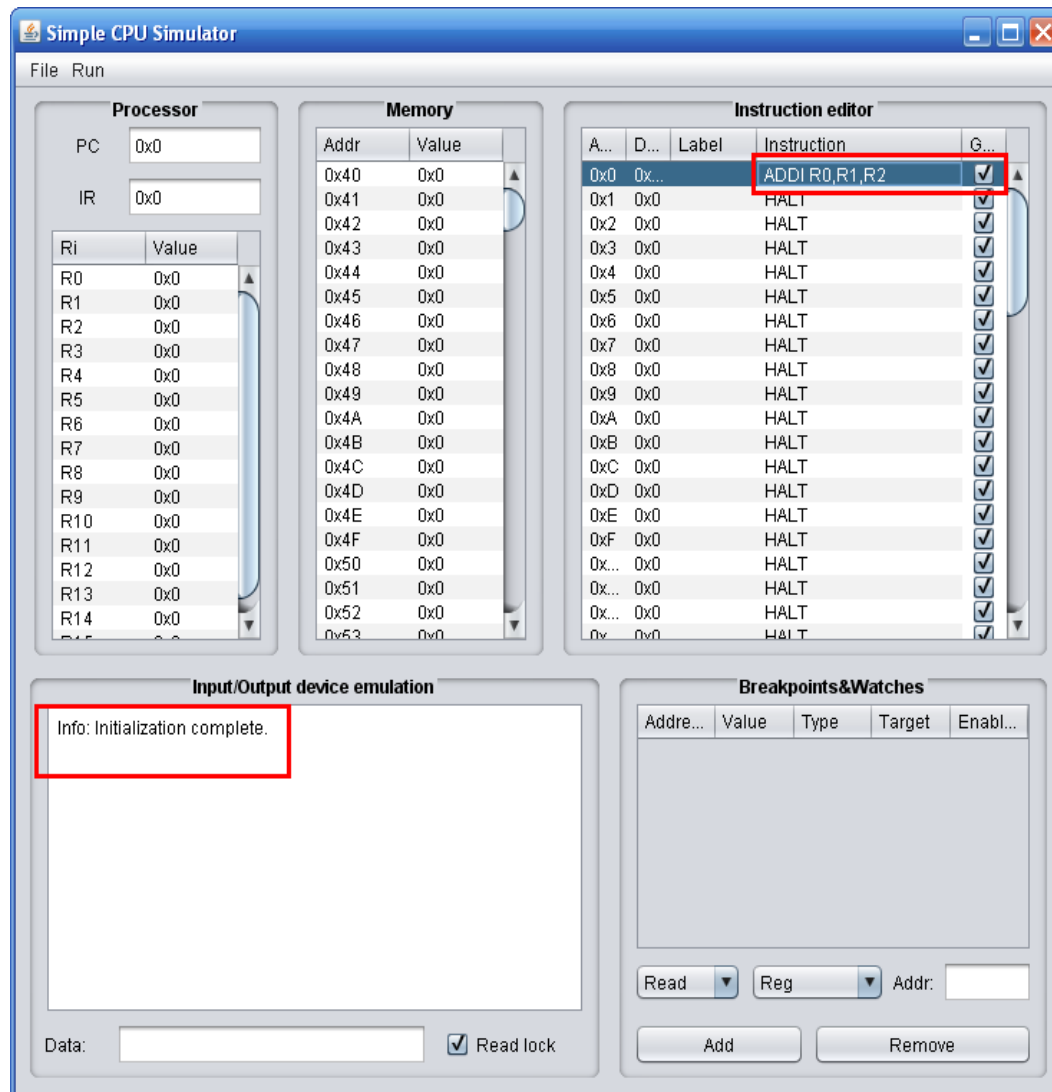
Input Valid Instruction Set:

HALT

ADDI R0,R1,R2

HALT

Figure 4:

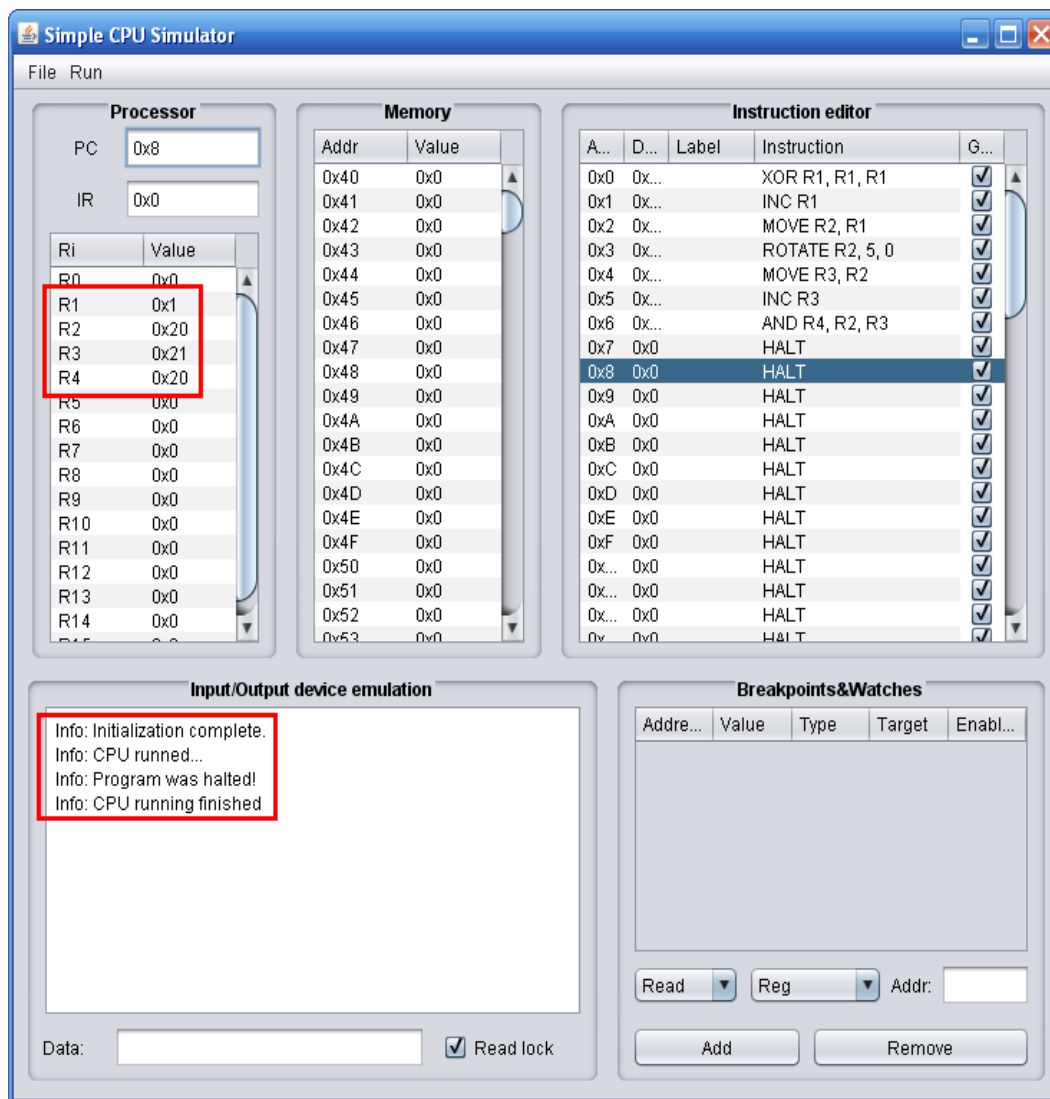


Test Case 3:

Test register input/output.

XOR R1, R1, R1	// R1 = R1 ^ R1, guaranteed to be 0
INC R1	// R1 = R1 + 1, guaranteed to be 1
MOVE R2, R1	// R2 = R1, guaranteed to be 1
ROTATE R2, 5, 0	// R2 = R2 shl 5, guaranteed to be 32
(0x20) MOV R3, R2	// R3 = R2, guaranteed to be 32 (0x20)
INC R3	// R3 = R3 + 1, guaranteed to be 33
(0x21) AND R4, R2, R3	// R4 = R2 and R3, guaranteed to be 32
(0x20)	

Figure 5:



Test Case 4:

Register addressation.

MOVE R0, R0 // Success

expected here MOVE R1, R1 // Success

expected here MOVE R2, R2 // Success

expected here

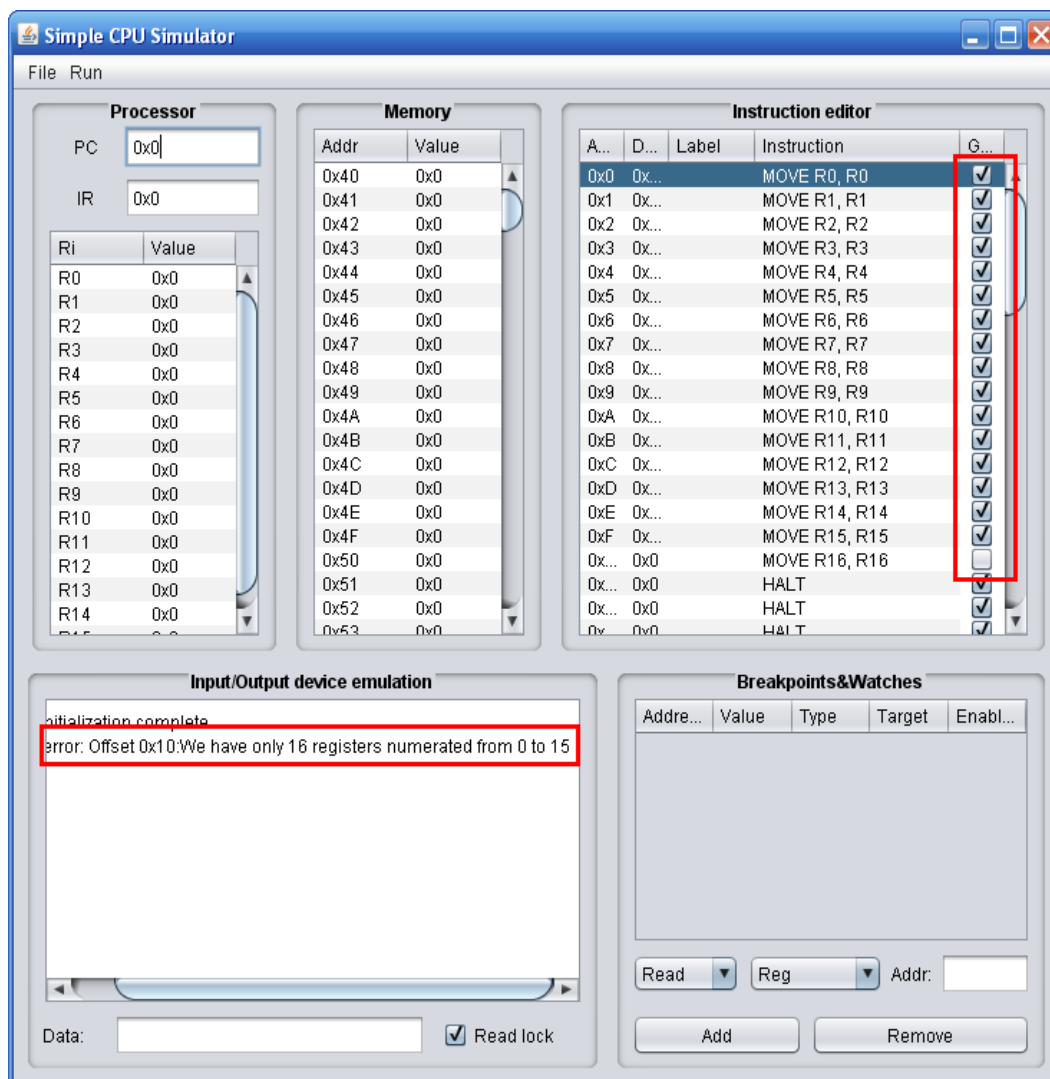
...

MOVE R14, R14 // Success

expected here MOVE R15, R15 // Success

expected here MOVE R16, R16 // Error

expected here *Figure 6:*



Test Case 5:

Memory I/O.

XOR R1, R1, R1 // R1

is 0 now INC R1 // R1

is 1 now ROTATE R1, 3, 0 // R1

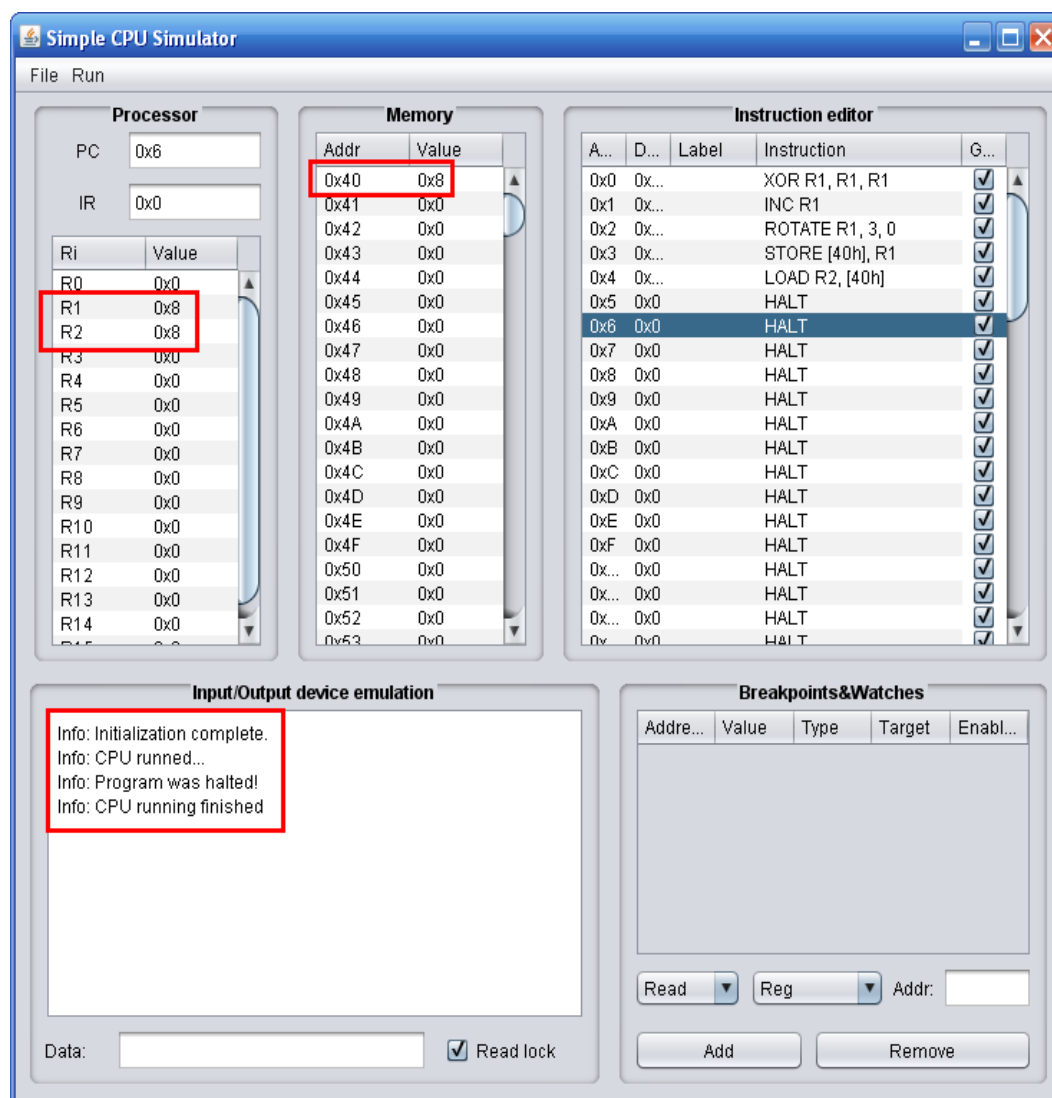
is 8 now

STORE [40h], R1 // 0x40th memory cell equals
to 8 now

LOAD R2, [40h] // R2
is 8 now

HALT

Figure 7:



Test Case 6.

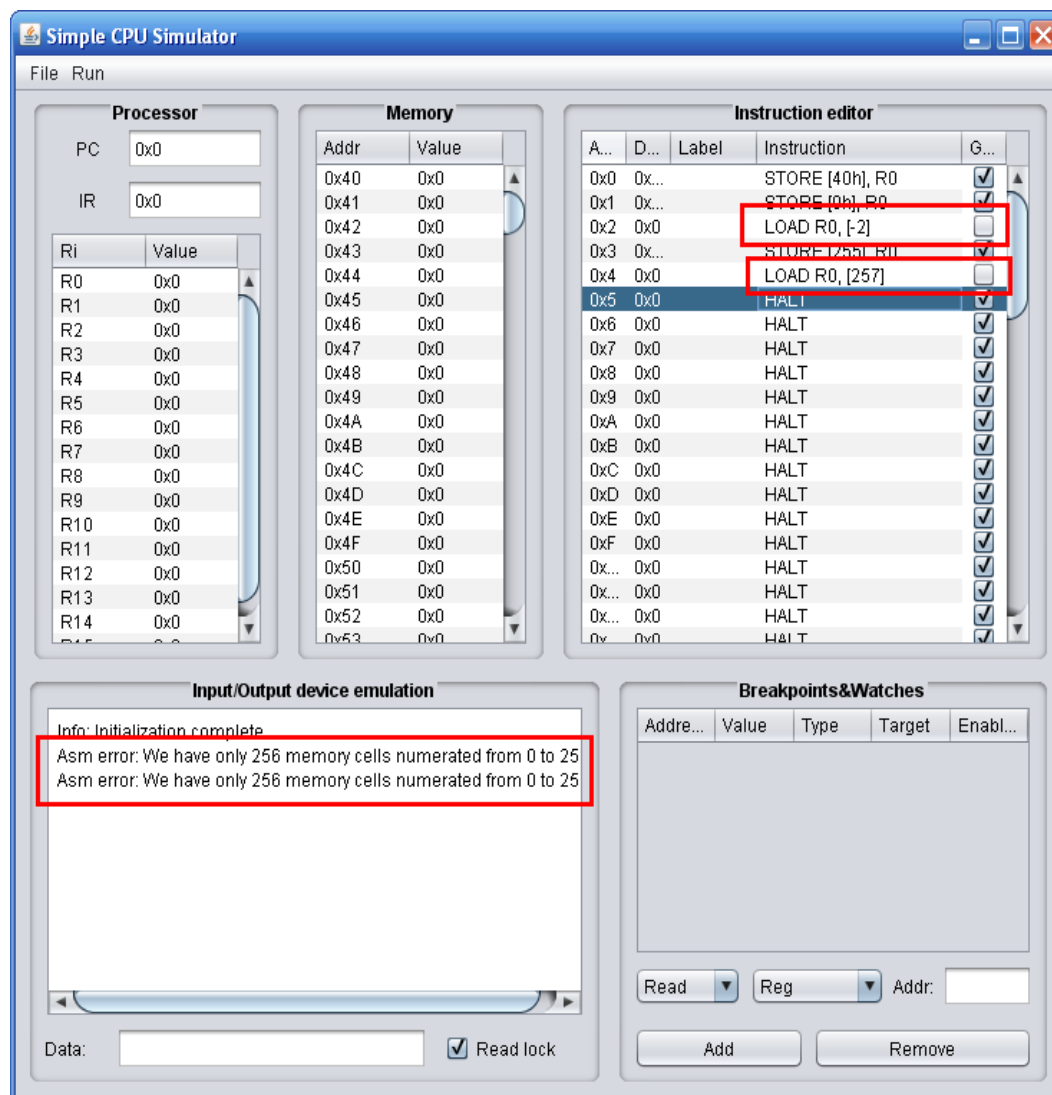
Checking memory range.

```

STORE [40h], R0          // Expected to be ok
STORE [0h], R0           // Assembling is expected
to be ok LOAD R0, [-2]    // Assembling should
print error STORE [255], R0 // Assembling should be
ok
LOAD R0, [257]           // Error expected here

```

Figure 8:



Test Case 7.

Memory read-only testing.

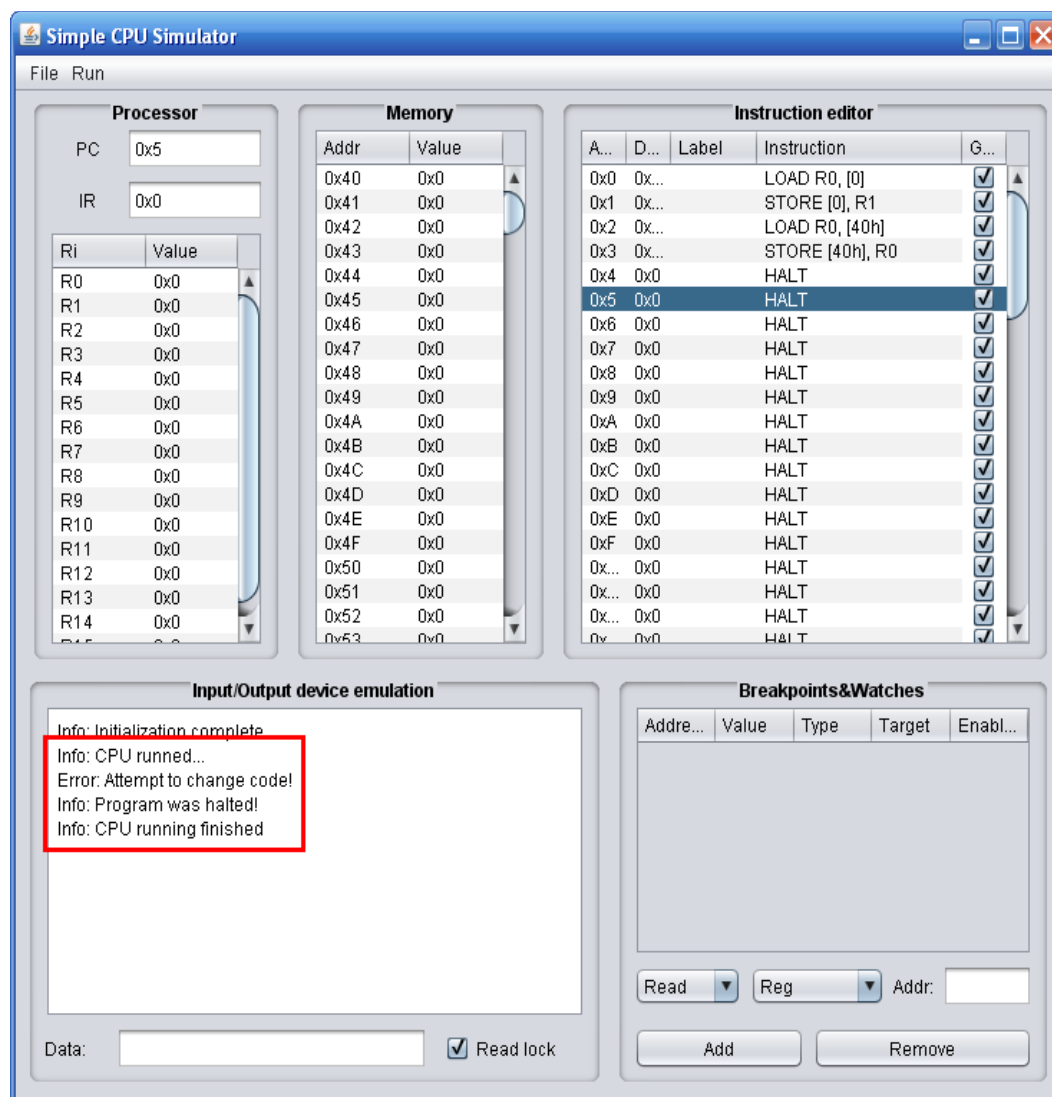
```
LOAD R0, [0]           // R0 will be 0x1000, the binary format of 'LOAD
R0, [0]' STORE [0], R1  // Runtime error expected here, but assembling
must be OK
```

// It's because 0-0x1F address range is write-protected

```
LOAD R0, [40h]         // Should be ok
```

```
STORE R0, [40h]        // Should be ok
```

Figure 9:



Test Case 8.

Test display output.

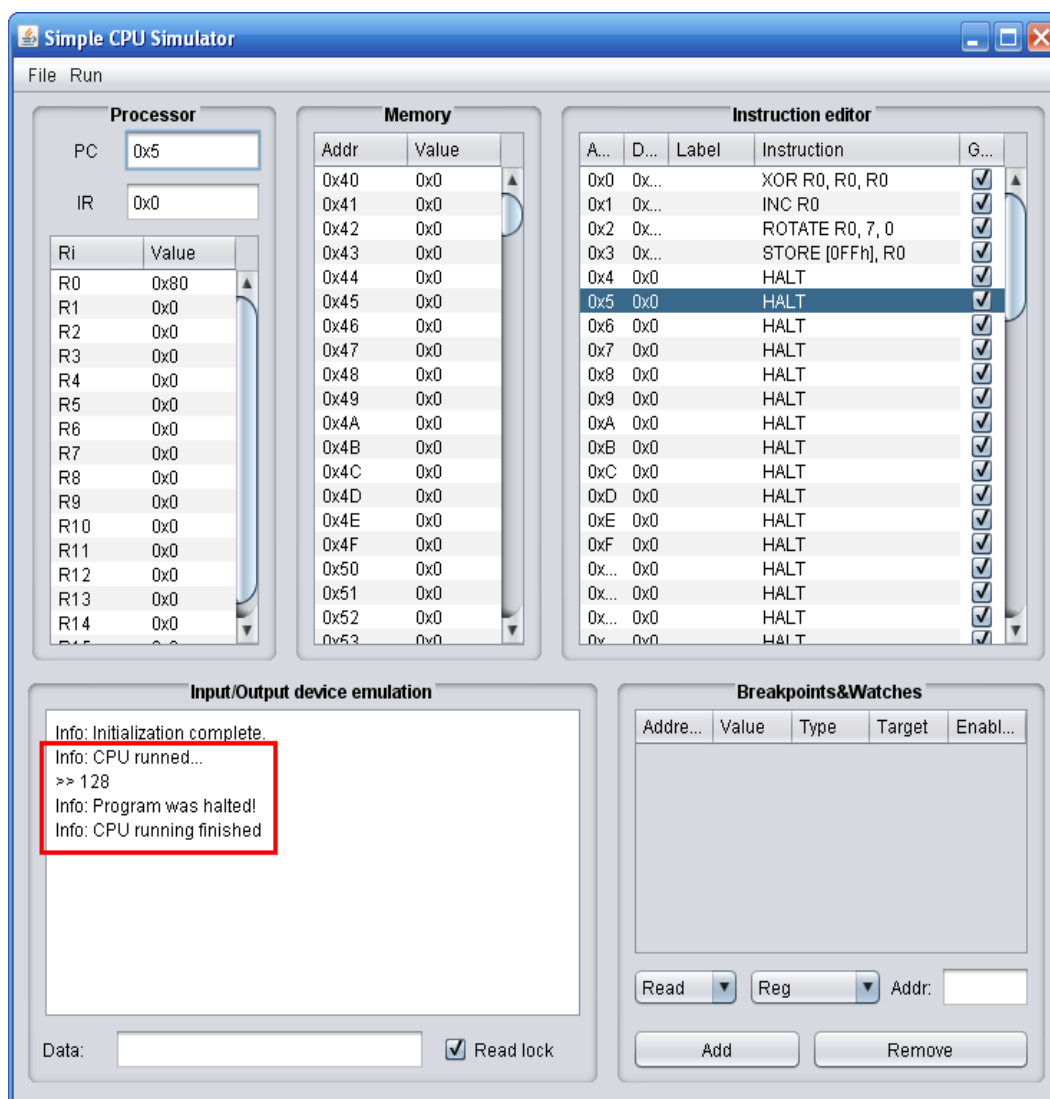
XOR R0, R0, R0

INC R0

ROTATE R0, 7, 0 // R0 expected to be 128

STORE [FFh], R0

Figure 10:



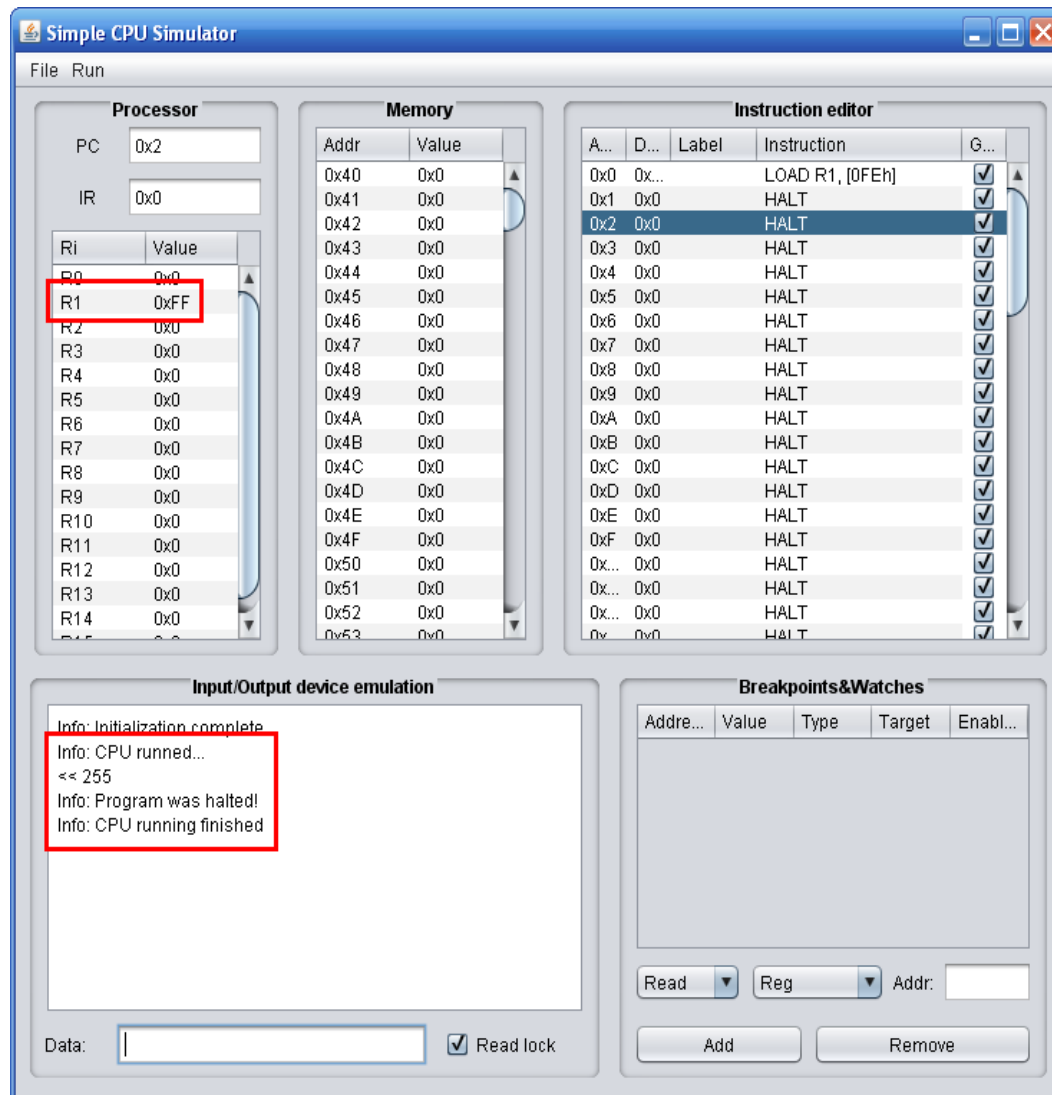
Test Case 9.

Keyboard input.

LOAD R1, [0FEh]

HALT

Figure 11:



Test Case 10.

Breakpoints test.

INC R2

STORE [40h], R2

STORE [41h], R2

STORE [42h], R2

LOAD R2, [40h]

LOAD R2, [41h]
instruction

// Program should be stopped after executing of this

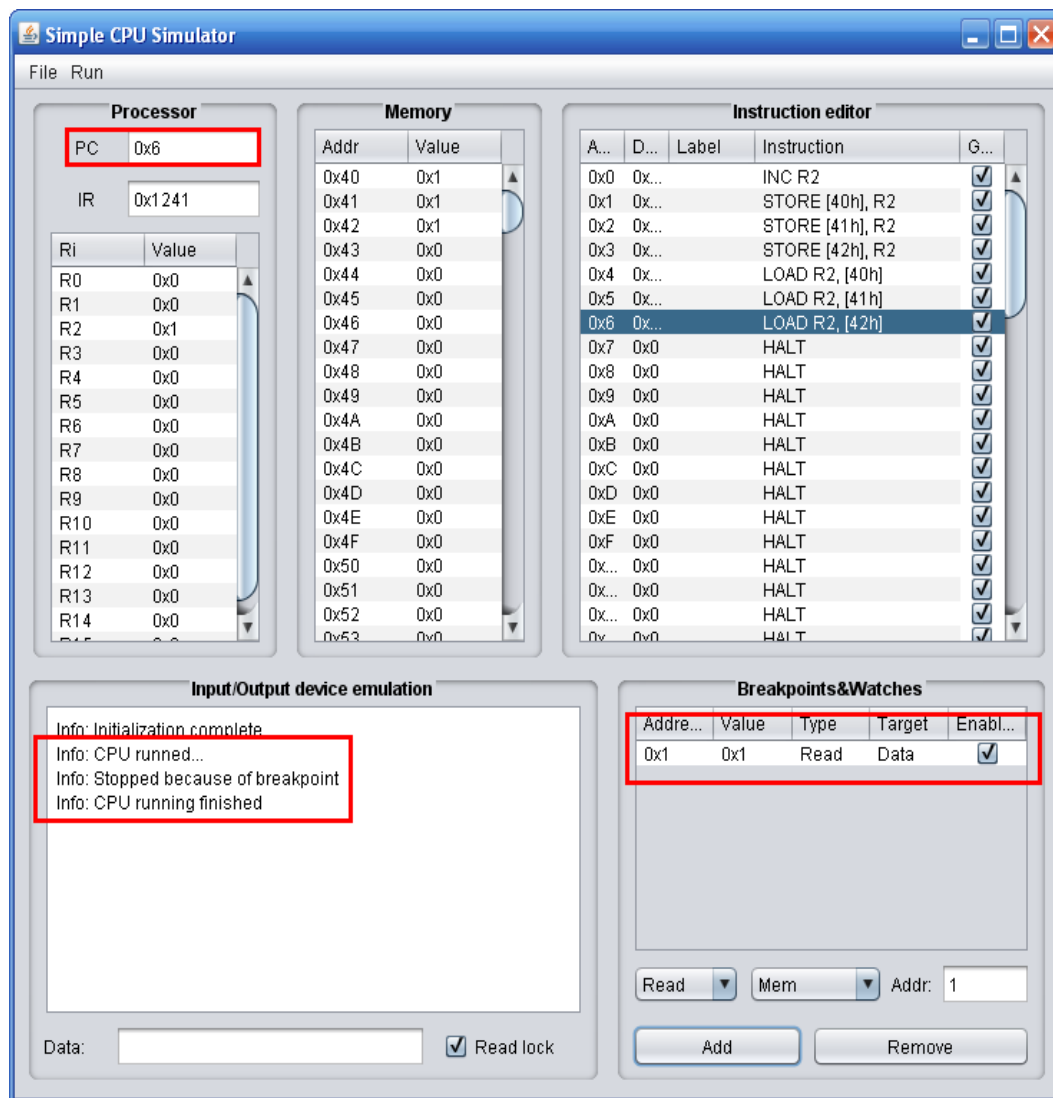
LOAD R2, [42h]

// So PC expected to be here and should be equal to 6

HALT
run the program

// Set up 'read' breakpoint to memory cell #0x41 and

Figure 12:



A bit of more complicated blackbox tests. These tests will be as an example how to use made computer model.

Test 1. Simple “add two numbers program”.

Test objectives:

Check keyboard and display virtual devices work correctly;

Check LOAD and STORE instructions work correctly;

Check ADDI instruction works correctly.

What program does:

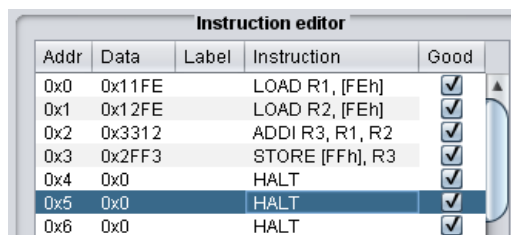
1. Gets two numbers from the keyboard.
2. Calculates a sum of readed two numbers.
3. Writes a result to two numbers.

Program code:

LOAD R1, [FEh]	<i>(reads first number from the keyboard)</i>
LOAD R2, [FEh]	<i>(reads second number from the keyboard)</i>
ADDI R3, R1, R2	<i>(adds first and second numbers and writes result to R3 register)</i>
STORE [FFh], R3	<i>(prints result to the display)</i>
HALT	<i>(finishes program)</i>

Program compilation:

Figure 13:

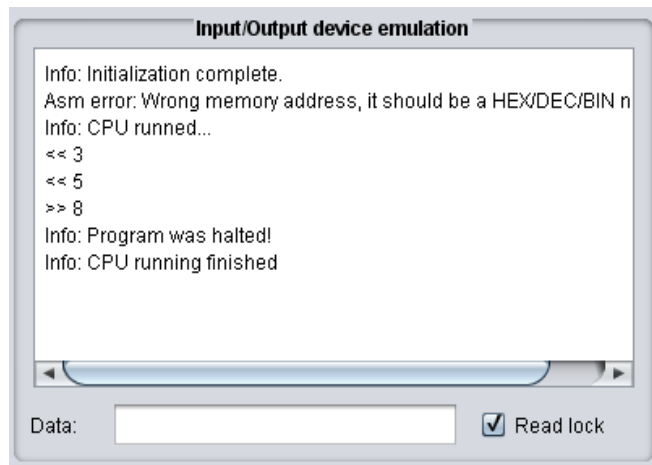


Addr	Data	Label	Instruction	Good
0x0	0x11FE		LOAD R1, [FEh]	<input checked="" type="checkbox"/>
0x1	0x12FE		LOAD R2, [FEh]	<input checked="" type="checkbox"/>
0x2	0x3312		ADDI R3, R1, R2	<input checked="" type="checkbox"/>
0x3	0x2FF3		STORE [FFh], R3	<input checked="" type="checkbox"/>
0x4	0x0		HALT	<input checked="" type="checkbox"/>
0x5	0x0		HALT	<input checked="" type="checkbox"/>
0x6	0x0		HALT	<input checked="" type="checkbox"/>

All good flags is marked which means each line of code has been compiled successful.

Running result (screenshot of console):

Figure 14:



I have entered 3 and 5 (“>>” signs mean output from display, “<<” signs mean input from keyboard). Program have gave 8 as result, because $3+5=8$.

Test results:

Instructions	5
Ticks	5
Program type	One-by-one
Instructions had tested:	
LOAD	OK (Keyboard)
STORE	OK (Display)
ADDI	OK
HALT	OK

Test 2. Multiplication of two numbers.

Test objectives:

Prove that some operations that are missed as instructions can be manually implemented;

Test AND and XOR

instructions; Test INC and DEC

instructions;

Test MOVE and ROTATE instructions.

Test labels operating and JUMP instruction.

What program does:

1. Reads two numbers from the keyboard.
2. Does multiplication of two numbers.
3. Prints a result.

Algorithm:

It's a classic multiplication algorithm "in column", but it is formed by binary numbers. We have to form second multiplier's shifts; then calculate a sum for shifts, relative bits of first number of which is 1s.

Here is a principle:

Figure 15:

$$\begin{array}{r} 12 \times 41 \\ \hline \begin{array}{l} 000101001 \\ 00101001 \\ 00101001 \\ 00101001 \\ 00101001 \\ 00101001 \\ 00101001 \\ 00101001 \end{array} \\ \hline 00000110011010 = 410 \end{array}$$

Registers explanation:

Table 3:

R0	Contains 0, used to compare numbers for JUMP instruction executing.
R1	Here is stored the first number, gotten from keyboard.
R2	Here is stored the second number, typed from the keyboard.
R9	Contains shift of the second number.
R10	Multiplication result (the sum of the shifts).
R11	Contains rightest bit of the shift of the first number.
R12	Contains the second number.

R13	Contains "1".
R14	Contains the cycle counter.
R15	Contains "0x10" constant.

Program code:

```

LOAD R1, [FEh]                (reads first number from the
keyboard) LOAD R2, [FEh]      (reads second number from the
keyboard) XOR R0, R0, R0      (sets R0 to zero)
XOR R15, R15, R15             (sets R15
to zero) XOR R10, R10, R10    (sets R10
to zero) INC R15              (sets R15
to 1) MOVE R13, R15           (sets R13
to 1)
ROTATE R15, 4, 0              (sets R15 to R15 << 4 which is equals to 0x10)
MOVE R14, R15                 (sets R14 to 0x10)
MOVE R12, R1                  (now R12 is first
number) MOVE R9, R2           (now R9 is second
number) C1_S:
AND R11, R12, R13             (R11 = R12 & R13 = R12 & 1; R11 will equal
to rightest bit of R12)
JUMP R11, C2_E                (if R11 != R0 (if first bit isn't equal to zero), do
ADDI
after C2_E)
JUMP R14, C2_END              (else skip ADDI before C2_END, because R14 is
never equals to 0)
C2_E:
ADDI R10, R10, R9             (adds second number to sum)
C2_END:
ROTATE R9, 1, 0               (shifts second number one bit
to left) ROTATE R12, 1, 1     (shifts first number one bit to
right) DEC R14                (decrements cycle counter)
JUMP R14, C1_S                (if cycle counter isn't equals to zero, continues
the cycle)
C1_E:
STORE [FFh], R10              (prints program result to the display)
HALT                          (stops the CPU)

```

Running result (a copy from the console):

Info: Initialization complete.

Info: CPU runned...

<< 9

```

<< 8
>> 72
Info: Program was halted!
Info: CPU running finished
Info: CPU runned...
<< 5
<< 7
>> 35
Info: Program was halted!
Info: CPU running finished
Info: CPU runned...
<< 105
<< 105
>> 11025
Info: Program was halted!
Info: CPU running finished

```

Explanation:

First time I've entered 9 and 8, and program gives me a result 72, which is correct, because $9 * 8 = 72$.

Second time I've entered 5 and 7 and got $5 * 7 = 35$.

Last time: $105^2 = 11025$, so program works correctly.

Test results:

Table 4:

Instructions	21
Ticks	125-141
Program type	One-by-one
Instructions had tested:	
<i>Instructions from the previous test and...</i>	
XOR	OK
AND	OK
INC	OK
DEC	OK
ROTATE	OK (Left and right)
MOVE	OK
JUMP	OK (W/ labels)

Test 3. Fibonacci number calculation.

Test objectives:

Implement well-known algorithm in our assembler language for our computer model;

What program does:

1. Prints Fibonacci numbers that can be placed in 16-bit register.
2. Then halts the

CPU. Algorithm.

There is a well-known algorithm for Fibonacci number generation:

- Assume $A1 = 0$, $A2 = 0$, $A3 = 1$;
- Print $A3$ and do $A1 := A2$, $A2 := A3$, $A3 := A1 + A2$;
- Repeat previous step until $A3$ will become larger than

8000h.

Registers explanation:

Table 5:

R0	Contains 0, used to compare numbers for JUMP instruction executing.
R11	Contains the A1 number (see “Algorithm” section above).
R12	Contains the A2 number (see “Algorithm” section above).
R13	Contains the A3 number (see “Algorithm” section above).
R14	Used to check is R13’s left bit is.
R15	Contains 1.

Program code:

```
XOR R0, R0, R0           (makes R0 equals to 0)
XOR R15, R15, R15        (makes R15 equals to 0)
INC R15                  (makes R15 equals to 1)
MOVE R11, R0             (initializes A1 := 0)
MOVE R12, R0             (initializes A2 := 0)
MOVE R13, R15            (initializes A3 := 1)
CYCLE:
STORE [FFh], R13          (prints the A3 number)
MOVE R11, R12            (does A1 := A2)
MOVE R12, R13            (does A2 := A3)
ADDI R13, R11, R12        (does A3 := A1 + A2)
MOVE R14, R13            (now start checking of number overflow, make a draft of A3)
```

ROTATE R14, 15, 1	<i>(rotates A3 to the right 15 times, so it's leftist bit will become rightest)</i>
NOT R14, R14	<i>(does bit inversion for R14)</i>
AND R14, R14, R15	<i>(cuts off all bits except of rightest one)</i>
JUMP R14, CYCLE	<i>(continue cycle if A3's inverted bit isn't equal to 0, means if it's equals to 0)</i>
HALT	<i>(finishes the application)</i>

Running result (a copy from the

console): Info: CPU runned...

```
>> 1
>> 1
>> 2
>> 3
>> 5
>> 8
>> 13
>> 21
>> 34
>> 55
>> 89
>> 144
>> 233
>> 377
>> 610
>> 987
>> 1597
>> 2584
>> 4181
>> 6765
>> 10946
>> 17711
>> 28657
```

```
Info: Program was
halted! Info: CPU
running finished
```

Test results:

Table 6:

Instructions	16
Ticks	214
Program type	Generator
Instructions had tested:	

XOR	OK
AND	OK
INC	OK
STORE	OK (Display)
ROTATE	OK (Right)
MOVE	OK
JUMP	OK (W/ labels)
ADDI	OK
NOT	OK

Test 4. “Guess a number” game.

Test objectives:

Prove that our computer model can execute simple games;

Test instructions and CPU functioning one more time.

What program does:

1. Firstly, user have to enter two numbers A and B which mean a range [A; B] that will imply C number;
2. Then program generates some number C;
3. User will have to guess C number by entering some numbers; if entered by user ordinary number is equals to C, program will print “0” and game will over, else program will print “-1” if typed number is lesser that C or “1” if typed number is greater than C.

Algorithm:

We will need a comparison operation to compare two numbers, but we haven’t it as particular CMP instruction. So we have to implement comparison using instructions we have.

Generally, we have to subtract one number from other, then check first bit from the left side of the result: if it equals to 0 then difference is number above 0, in other case number is below 0. It’s because first bit from the left side is a sign bit.

There are 2 numbers to compare: A and B. We have to do these operations:

- Calculate $C = A - B$;
- Check if C equals 0, if it is, then conclude: A is equals to B (because $A - B = 0$);
- Check if first bit from the left side of C is eq to 1, if it is, then conclude: $C < 0$, so A is less than B (because $A - B < 0$);

- Check if first bit from the left side of C is eq to 0, if it is, then conclude: $C > 0$, so A is greater than B (because $A - B > 0$);

Output form of algorithm will be -1, 0 or 1 (less, equals to or greater, resp.).

As for pseudo-number generation algorithm, we can use this pseudo-random algorithm:

1. Get A and B numbers from user, these numbers means [A; B] range;
2. Then calculate difference of A and B, call it C;
3. After that we should generate some number using XOR instruction of A, B and/or C;
4. Add A to the random number;
5. Check if $A > C$, if it's, subtract C from the random number; repeat this check until $A \leq C$.

Registers explanation:

R0	Contains 0, used to compare numbers for JUMP instruction executing.
R1	Least allowed number (A).
R2	Greatest allowed number (B).
R3	Range length ($B - A + 1$).
R4	Secret random number, it always $\geq A$ and $\leq B$.
R5	Used as temporary register in several calculations.
R6	User's guessing number typed from the keyboard.
R15	Contains 1.

Program code:

```

XOR R0, R0, R0           (makes R0
equals to 0) XOR R15, R15, R15 (makes R15
equals to 0) INC R15      (makes R15
equals to 1)
LOAD R1, [254]           (reads left limit from the
keyboard) LOAD R2, [254] (reads right limit from the
keyboard) NOT R3, R1
INC R3
ADDI R3, R3, R2
INC R3                   (R3 := R2 - R1 + 1, R3 will equal to range
length)
XOR R4, R1, R2           (here we're generating pseudorandom number of
any range...)
ROTATE R4, 5, 0

```

```

XOR R4, R4, R3
ADDI R4, R4, R3
ADDI R4, R4, R1
ADDI R4, R4, R2                                (...we got it, it can be any 16-bit number)
CHECK:                                          (now decrease it to make it fit in the [0; C] range)
NOT R5, R3
INC R5
ADDI R5, R5, R4
JUMP R5, CONT_CHECK    (generated number isn't equals to C, so jump
to comparing code)
JUMP R15, BREAK_CHECK (generated number is equals to C, so it is in the range
[0; C], go to guessing)
CONT_CHECK:
ROTATE R5, 15, 1
AND R5, R5, R15                                (gets last first bit from the left side of a
generated number)
JUMP R5, BREAK_CHECK (if it's 1, G is in the range [0; C], so we can jump to
guessing)
NOT R5, R3
INC R5
ADDI R4, R4, R5                                (subtract C from guessing number –  $G := G -$ 
C) JUMP R15, CHECK    (...and continue cycle until G become  $\leq C$ )
BREAK_CHECK:                                (here we have pseudorandom number in [0; C]
range) ADDI R4, R4, R1    (make it be in [A, B] range)
USER_TRY:
LOAD R6, [254]                                (reads user's guess)
NOT R5, R4
INC R5
ADDI R5, R5, R6                                (compares R6 to R4 – guessing number to the
generated number)
MOVE R6, R5
JUMP R6, FAIL    (if the guessing number isn't equals to the secret one,
jump to fail check)
JUMP R15, OK    (else jump to finish game)
FAIL:
ROTATE R6, 15, 1
AND R6, R6, R15                                (gets leftist bit of difference between generated
and guessing number)
JUMP R6, GREATER    (if bit  $\neq 0$  (means  $\neq 0$ ), then difference  $> 0$ ,
so jump
to code that prints "1")
XOR R6, R6, R6
DEC R6    (now R6 is equals to -1)
STORE [255], R6    (prints "-1")
JUMP R15, USER_TRY    (go to next user's try handling)

```

GREATER:
STORE [255], R15 *(prints "1")*
JUMP R15, USER_TRY *(go to next user's try handling)*
OK: *(here we will be when user will correctly guess*
a number)
STORE [255], R0 *(prints "0")*
HALT *(halts CPU)*

Running result (a copy from the console):

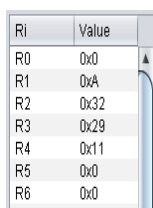
Info: CPU runned...

```
<< 10
<< 50
<< 40
>> -1
<< 30
>> -1
<< 20
>> -1
<< 15
>> 1
<< 17
>> 0
```

Info: Program was halted!

Info: CPU running finished

Also, here is a after-execution snapshot which proves that generated number do was same as guessed by me:



Ri	Value
R0	0x0
R1	0xA
R2	0x32
R3	0x29
R4	0x11
R5	0x0
R6	0x0

We can see here, that R4 register (with generated number) contains 0x11 that in decimal representation is 17.

Test results:

Table 7:

Instructions	48
Ticks	Too hard to calculate
Program type	One-by-one

Instructions had tested:	
XOR	OK
INC	OK
LOAD	OK (Keyboard)
NOT	OK
ADDI	OK
ROTATE	OK (Right)
MOVE	OK
JUMP	OK (W/ labels)
AND	OK
STORE	OK (Display)
HALT	OK

Summary blackbox testing results:

There is a conflict with label operands and register operands detected. Label names cannot start with “R” symbol, because application parses it as a register operand and prints an error about invalid register number. To workaround with this bug, we can just keep in mind to use labels that aren’t starts with the “R” symbol.

Whitebox testing.

Test 1. Take a look to the output to the console.

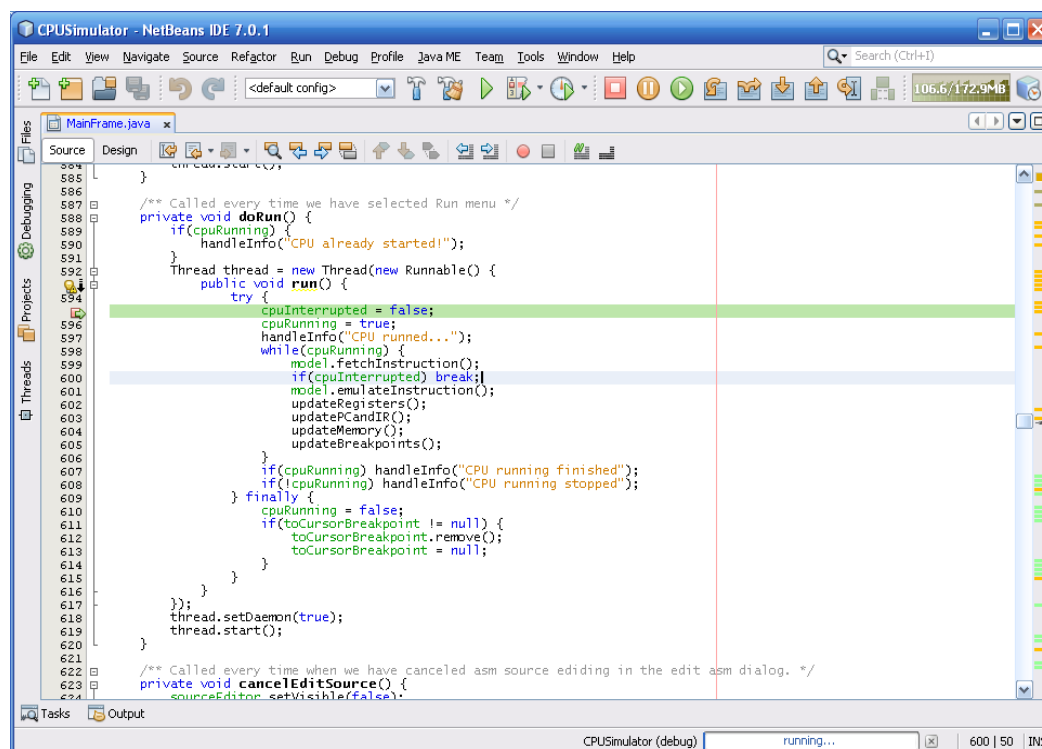
Let run this small code:

STORE [FFh], R0

HALT

After I've typed a command at CPU Sim window and selected "Run" in the main window menu, NetBeans debugging stopped here, at the method doRun() that is assigned to the "Run" menu action.

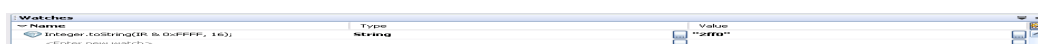
Figure 16:



Skip execution of CPU Sim to the model.emulateInstruction(); line, go inside the method.

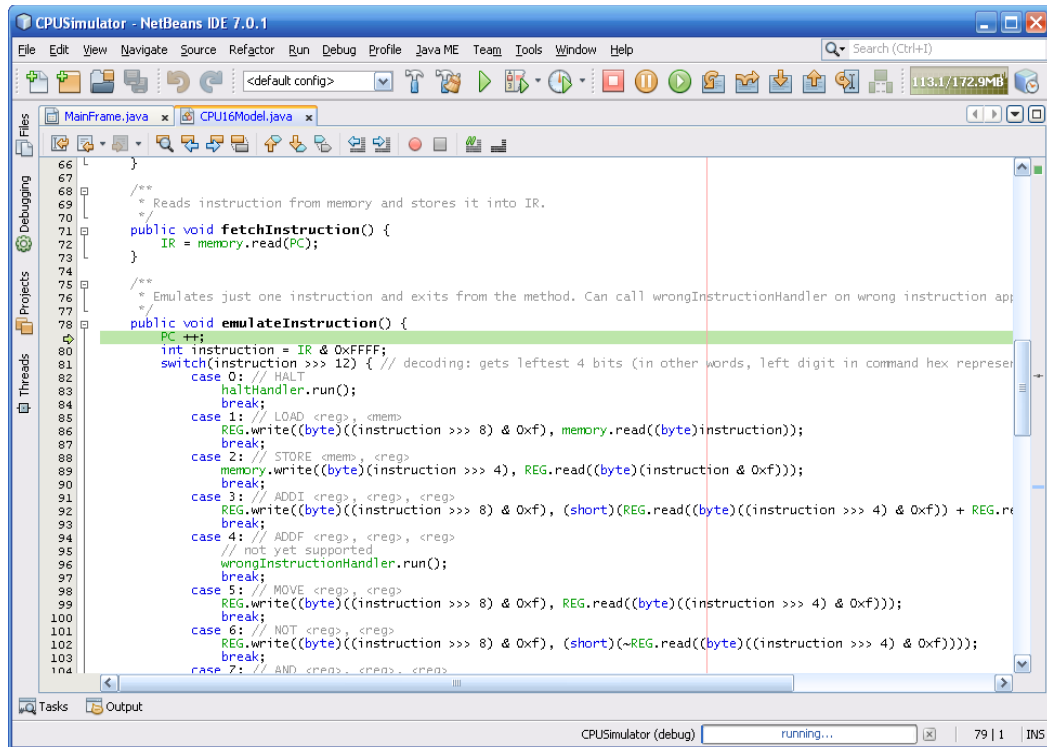
So now instruction has been fetched from the memory and processor have IR value 0x2FF0 ('STORE [0FFh], R0' command binary format). Take look at the watches:

Figure 17:



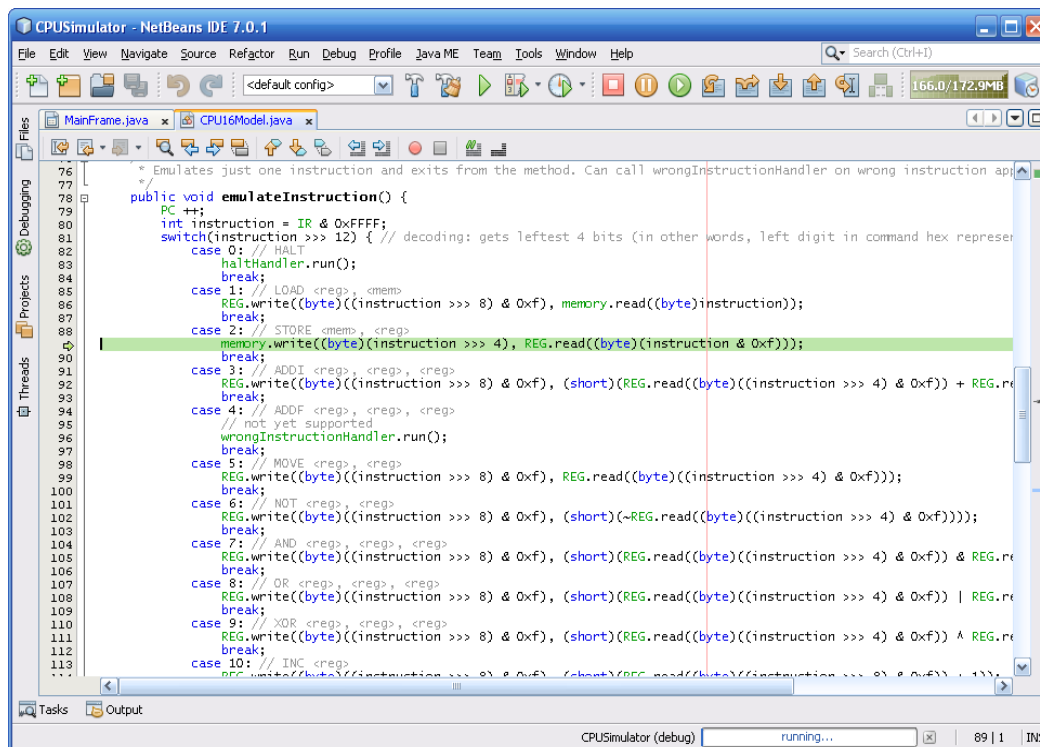
Then we will be here:

Figure 18:



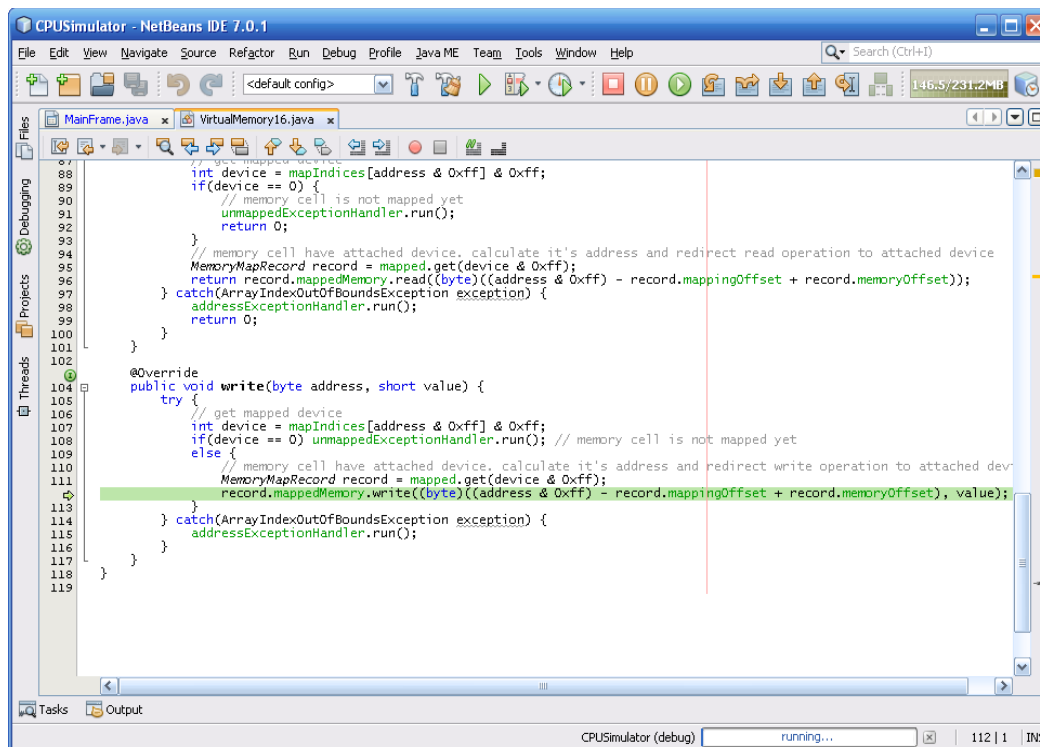
Our instruction have '2' code, so switch statement will set flow control to the 'case 2':

Figure 19:



When we will go inside memory.write(...) method, we should be in the VirtualMemory16 class, inside the overwritten write(...) method:

Figure 20:

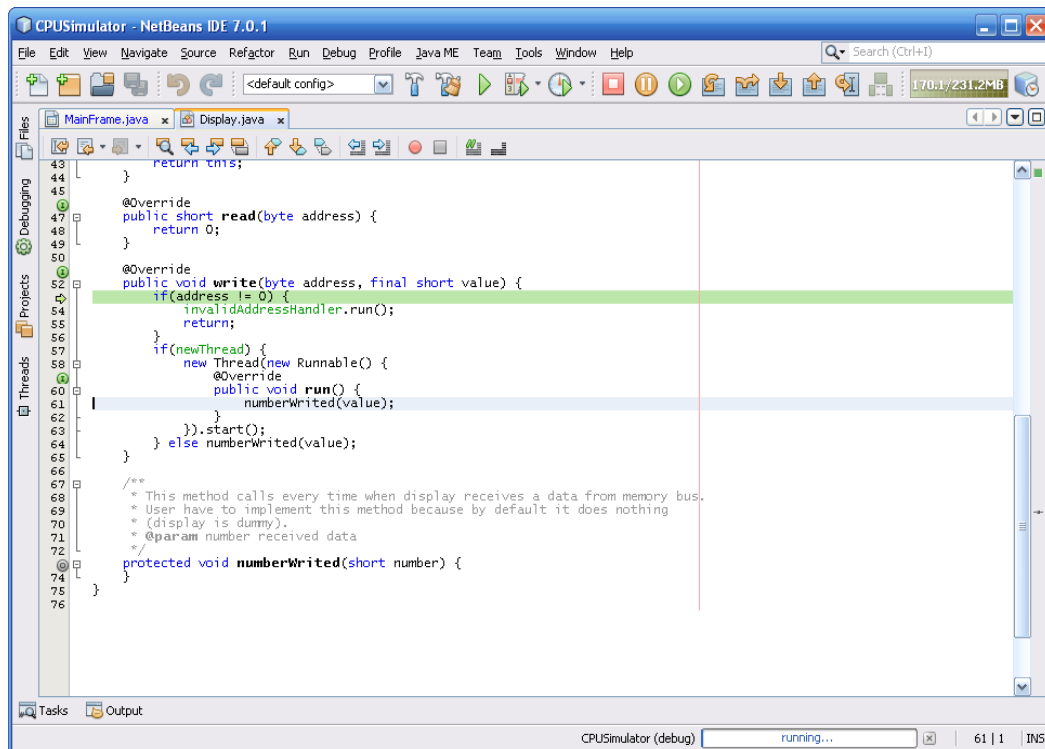


As we can see, flow control skipped `unmappedExceptionHandler.run()` statement, because it should be executed in case of using unmapped memory cell. But all of cells in our memory is mapped, so we got `MemoryMapRecord` that contains information about attached device.

After this, `VirtualMemory16` calculates absolute offset in the memory and calls `write(<absolute_offset>, value)`. It should be overwritten `write(...)` method of `Display` class, because we're performing attempt to watch display output.

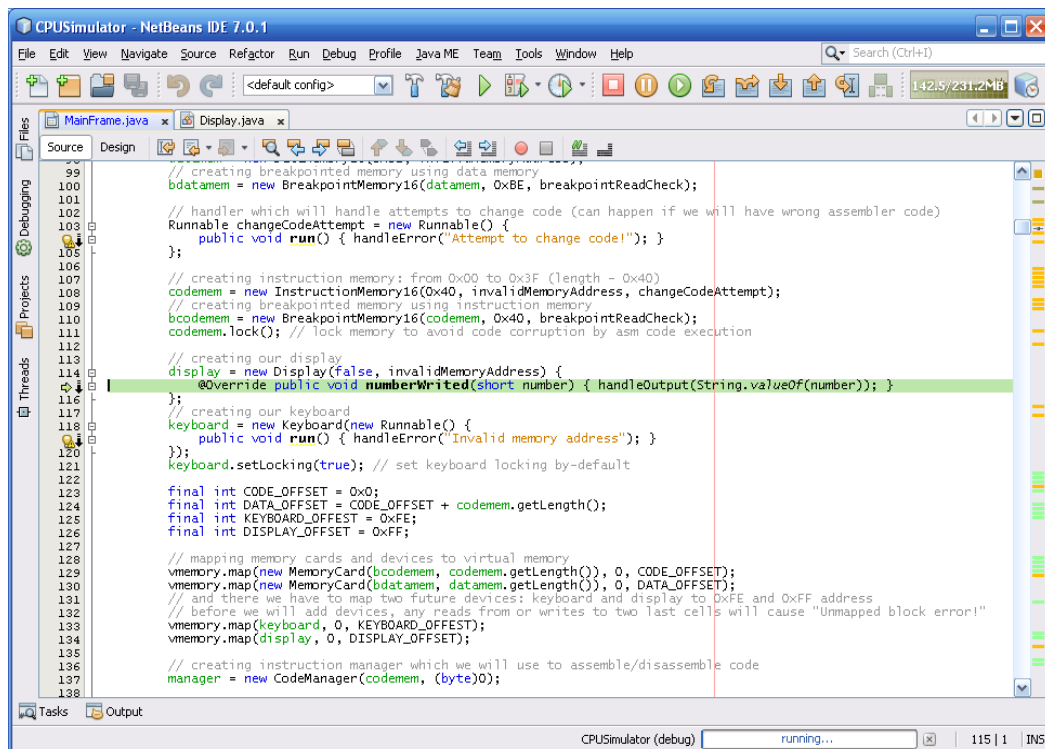
When we're going inside the `write` method (by pressing F7 – Step Into debug command), we can see this in the NetBeans debugger:

Figure 21:



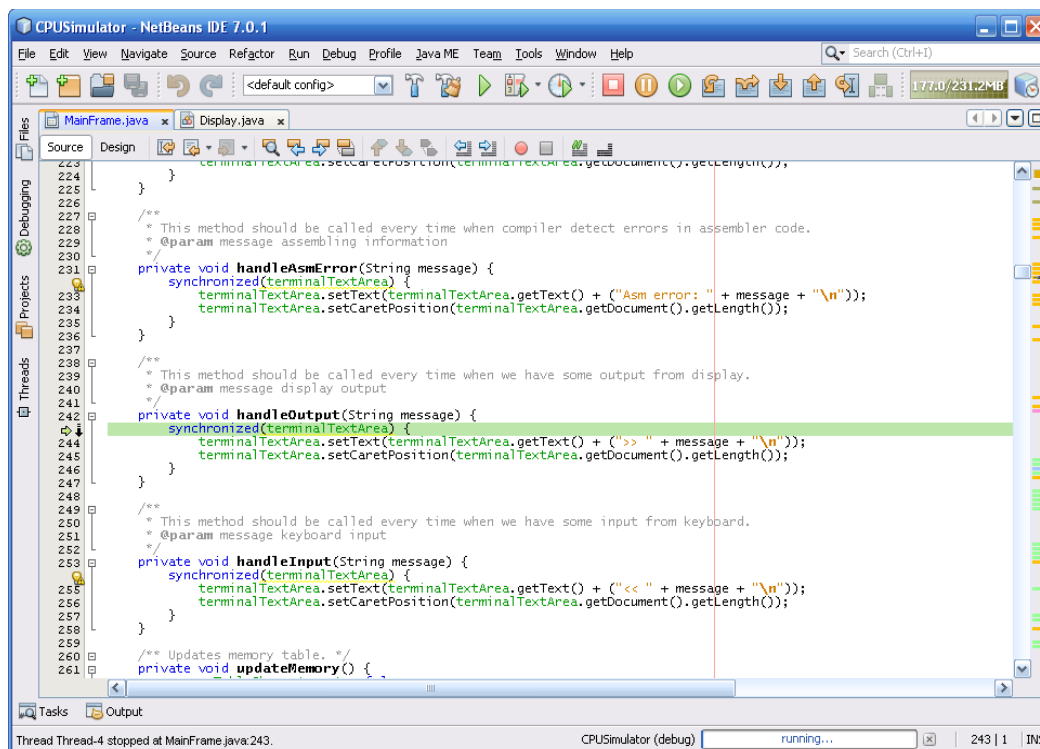
Then, let's set look inside numberWritten(value); line.

Figure 22:



We're got flow control to our link method that redirects display output to main window.

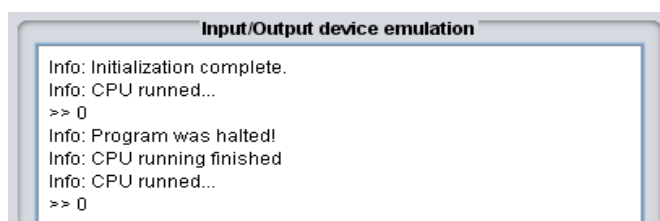
Figure 23:



Then we can see flow control reached the code when printed text will be added to the terminal by changing text in the terminalTextArea JTextArea variable.

We can see result of execution here:

Figure 24:

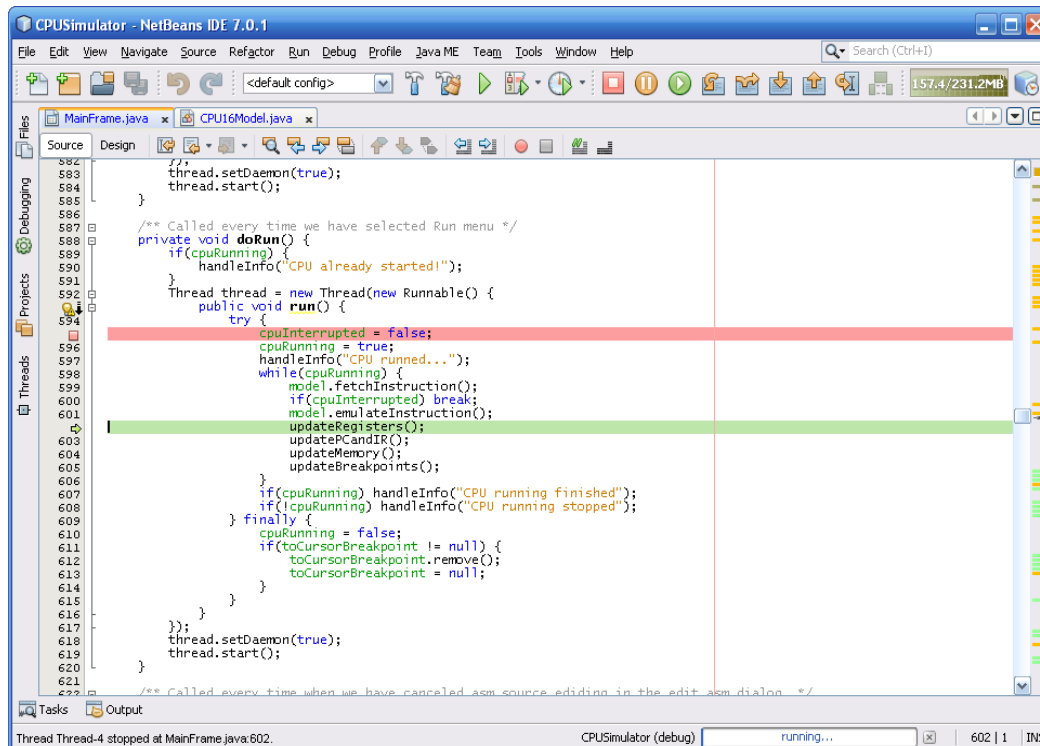


(We're have two zeroes because I've runned debug before).

But notice that ">> 0" prints after exiting synchronized block only, so there is no conflict between simulated Display modifying JTextArea variable and system renders JTextArea to the real display.

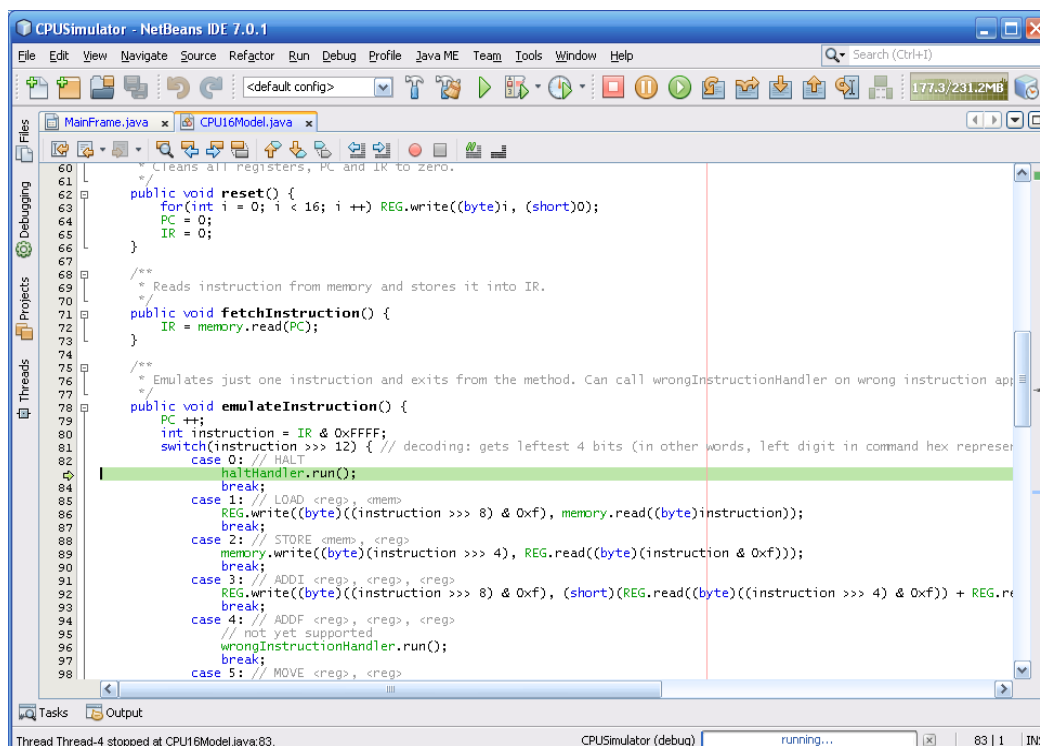
After this we can see rollback of stack to this point:

Figure 25:



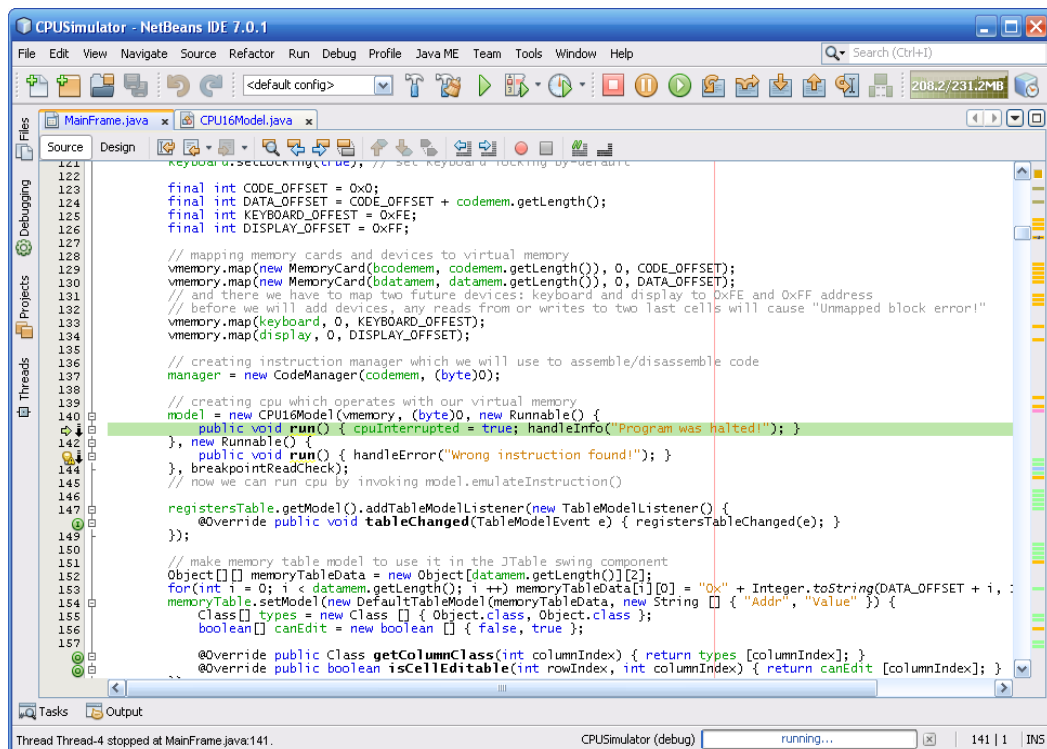
After updating JTextArea elements we're going into CPU simulation once more. Switch instruction should take program control to 'case 0' line because it's a halt now.

Figure 26:



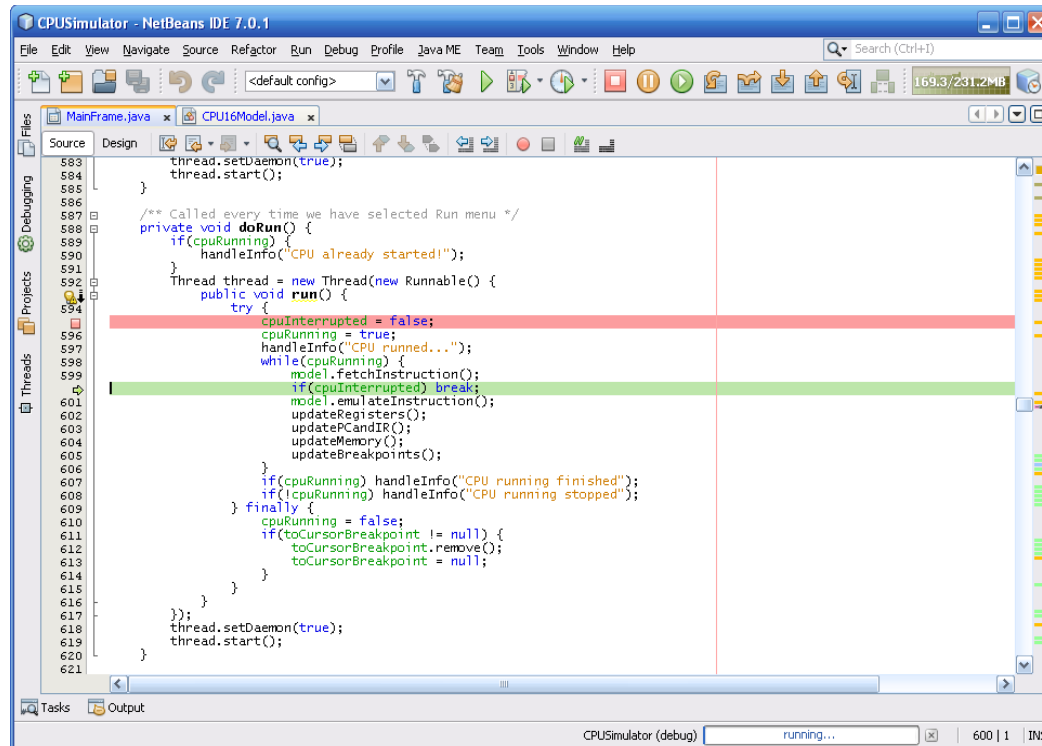
And it does! Now halt handler halts the CPU:

Figure 27:



After CPU interrupted variable is setted up, we're finishing the main CPU simulation loop:

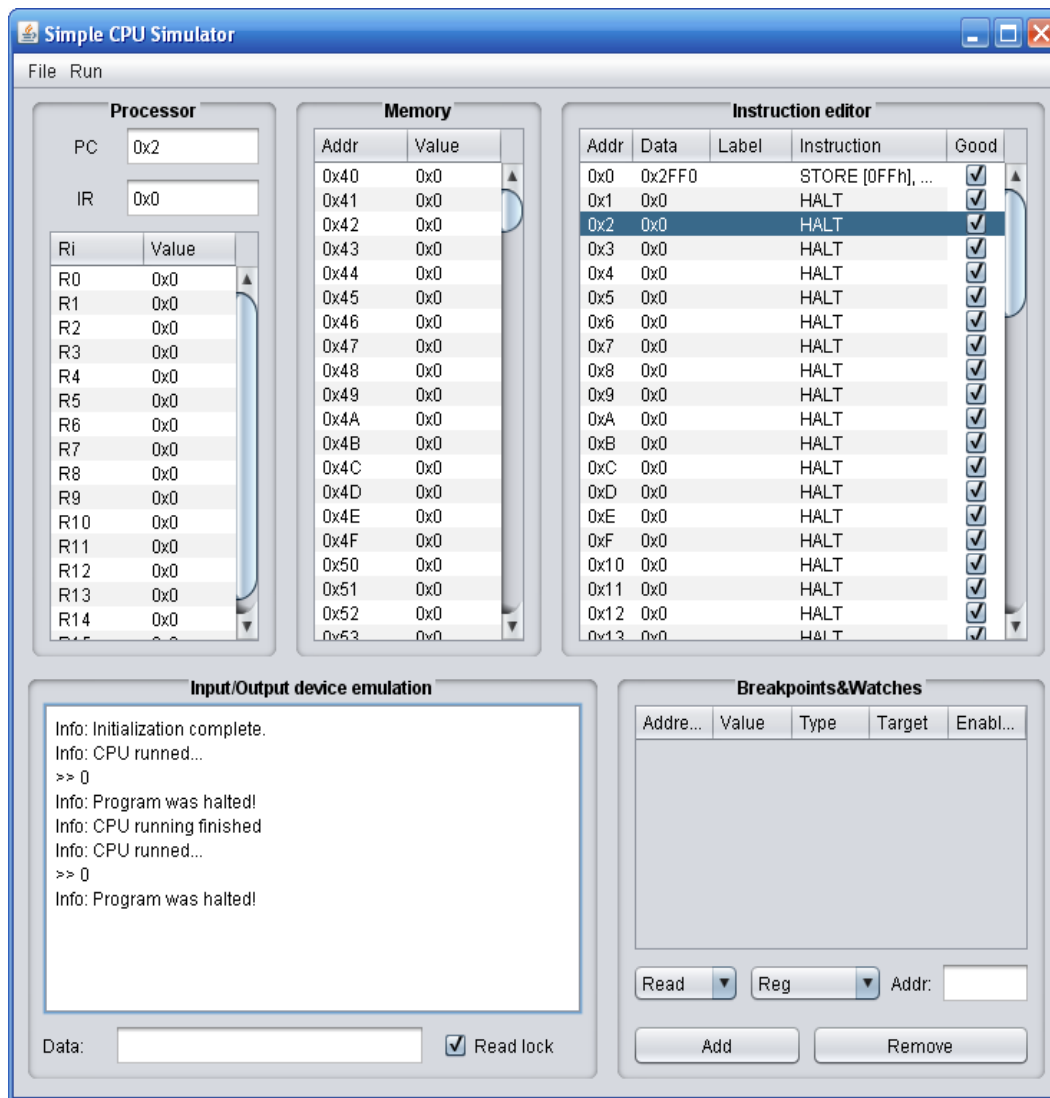
Figure 28:



Then thread finished execution and dies.

We've got this result:

Figure 29:



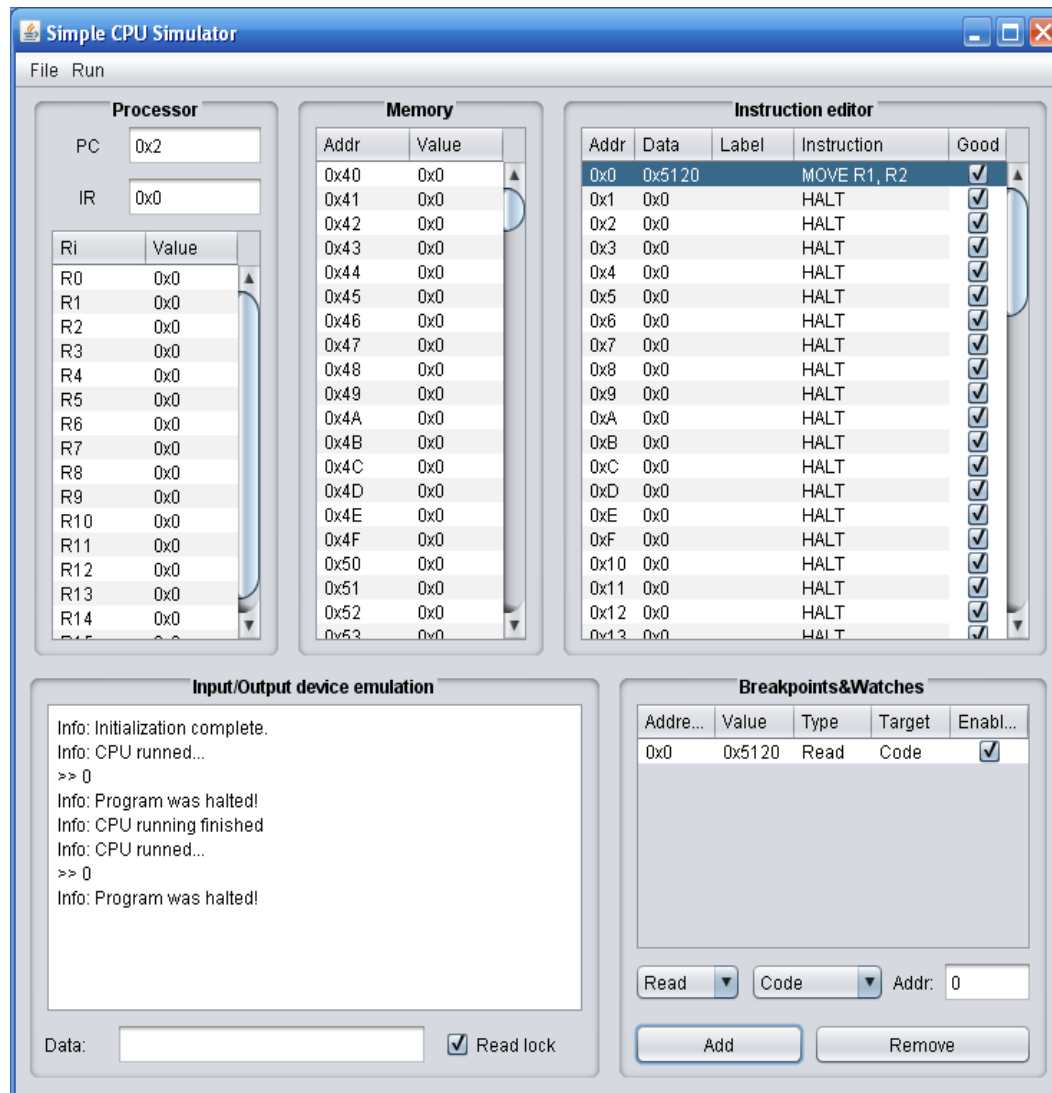
All is done as expected by program execution: it's 0 value printed to the console and program was halted as we've expected.

Glassbox test 2.

Breakpoint stop.

Let create small program and try to debug program execution.

Figure 30:



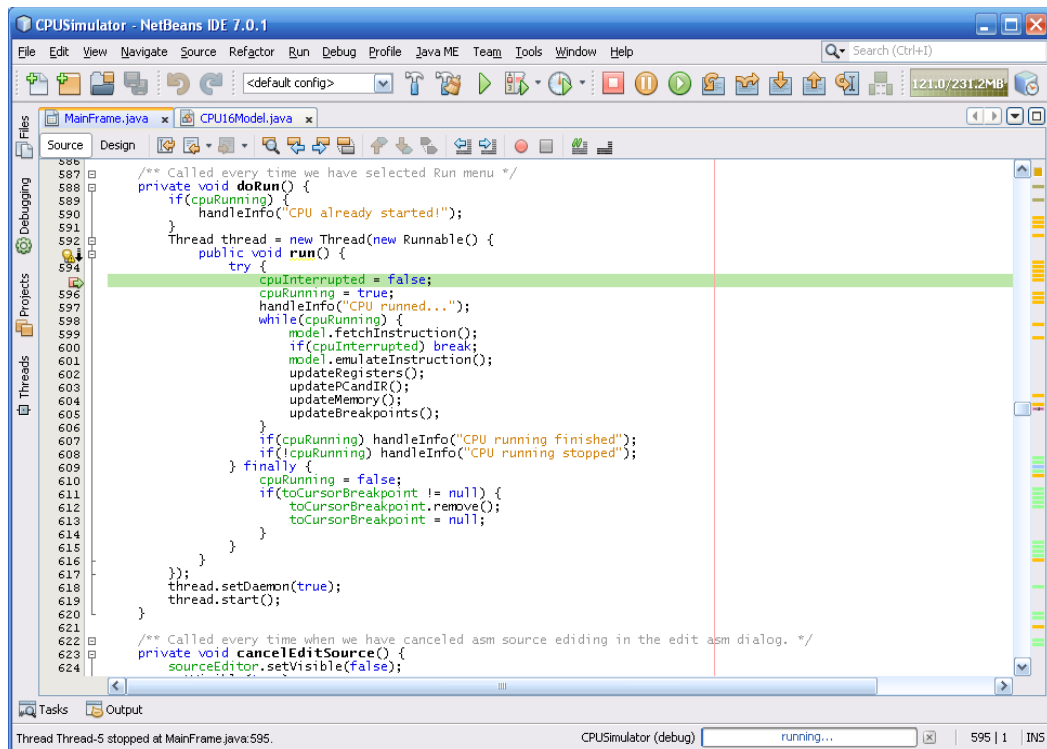
As we can see, I've added breakpoint to the 0th instruction in the code.

So we should stop BEFORE execution of 0th instruction.

I've started app using NetBeans debugger and used "Run" menu item of CPU Sim.

Program flow have stopped here:

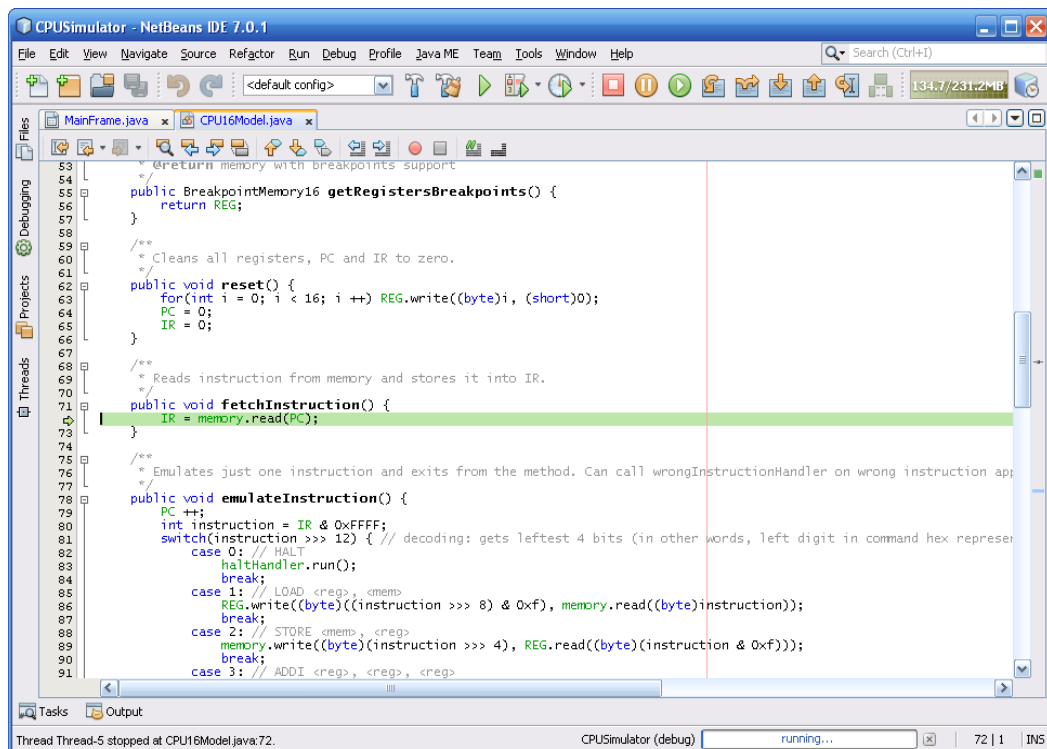
Figure 31:



So, CPU Sim is starting to run program.

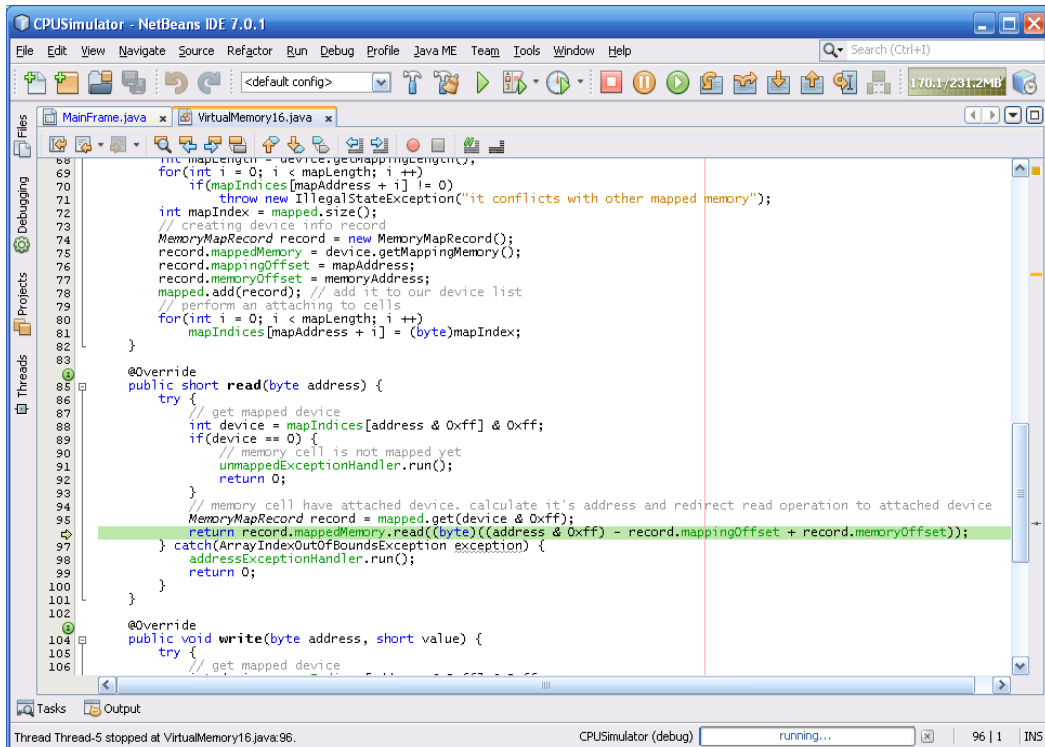
Let's go inside of `fetchInstruction()` method:

Figure 32:



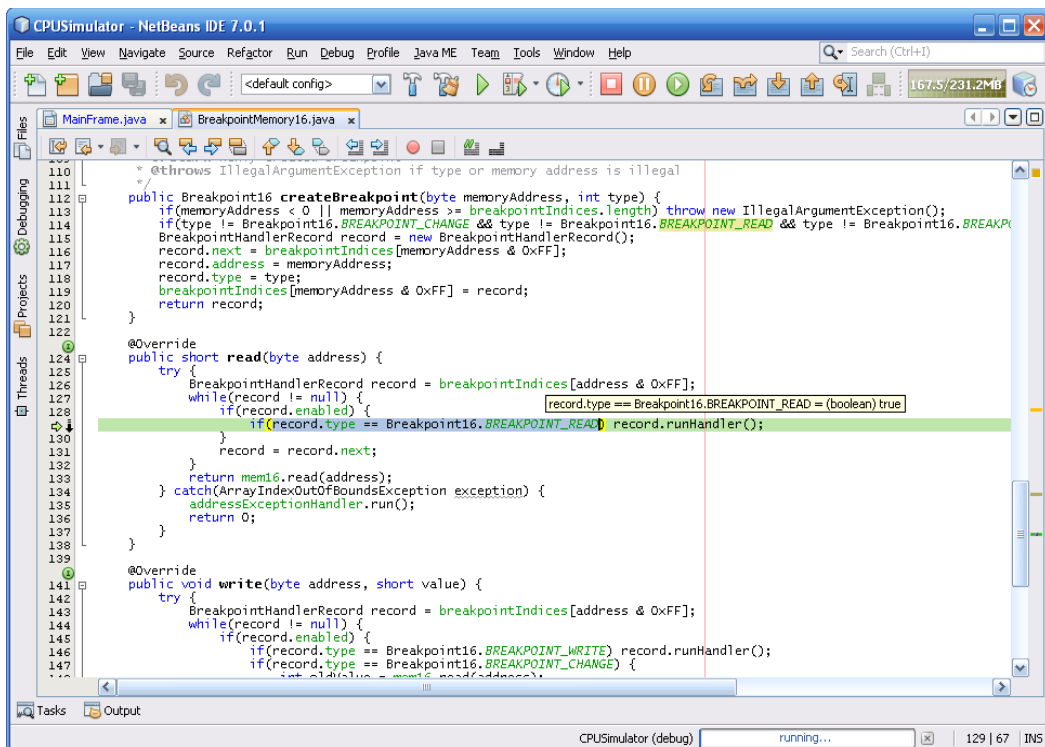
Before execution we must fetch instruction, so we're reading instruction from attached to CPU memory, and we're got here:

Figure 33:



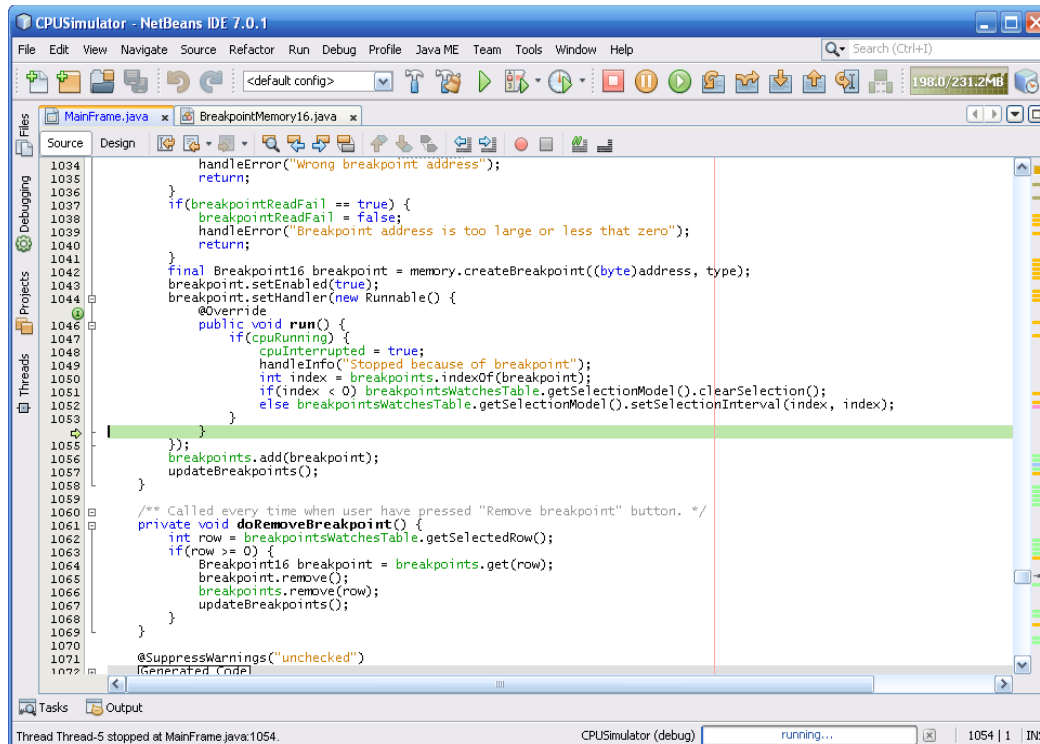
Yes, we got attached mem device (and it's instruction memory inside of breakpoint mem).

Figure 34:



Before reading from the instruction memory we're caught by breakpoint memory layer and have breakpoint handler signal. Let's go look inside it:

Figure 35:



I've ran all of handler code inside and we can see that breakpoint have highlighted on the main frame:

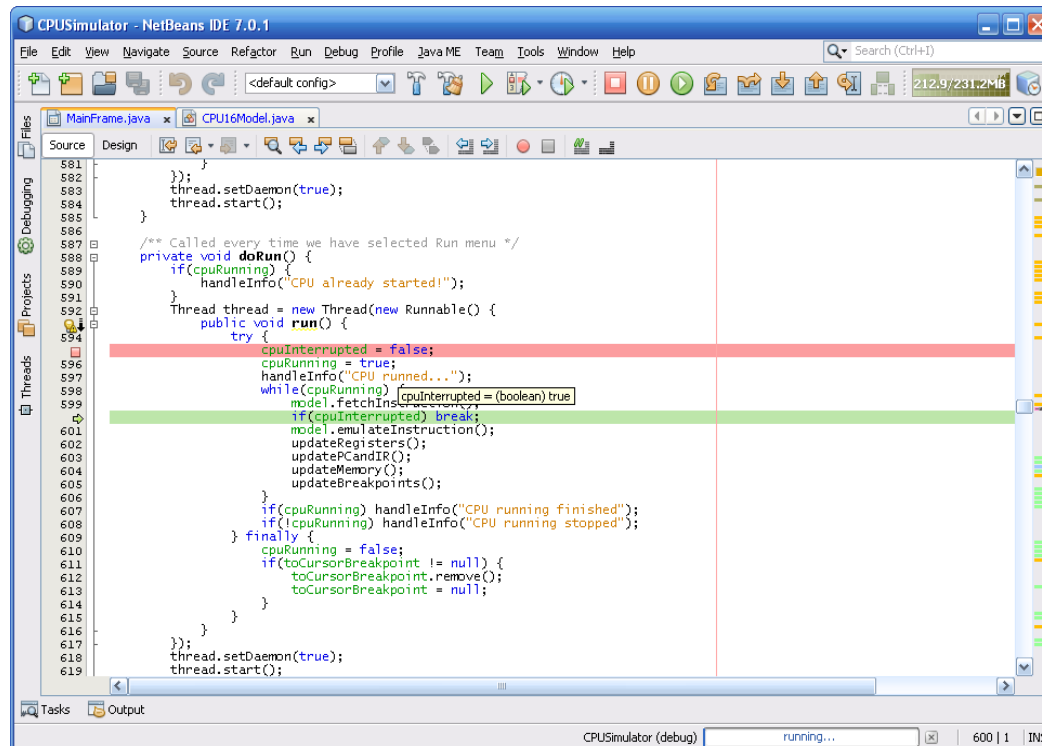
Figure 36:



Then breakpoint memory will take flow control to the instruction memory.

Finally we're got fetched instruction, but before executing we have interrupt checking here:

Figure 37:



And because of breakpoint handler have been ran, we have `cpuInterrupted` variable setted to true, which will cause executing cycle to break and execution be stopped.

Let look at the console and PC&IR after exit from the execution cycle:

Figures 38 & 39:



We can see that PC still have 0 value, so if we'll hit "Run" menu item once more, nothing will happen until we'll turn off breakpoint.

6. DISCUSSION AND CONCLUSIONS

The simulated CPU of 16bit performs the basic operations as per the requirement. The CPU code is well formed and classes use various functionality of the OOP which enables the entire code to function properly.

The CPU emulator performance is little slow as the language on which it is implemented is slower than the actual Hardware of the system. The entire code functionality is limited to lesser number of devices and scope of improvement and advancement is there in the code which can enable newer features and the functionality of the CPU emulator.

The display classes and memory classes are limited to the simple CPU functionality which can be enhanced further.

This project was realized with the next **Recommendations** (see the next page).

Recommendations

The major Recommendations are:

1. Data memory can be mapped and enhanced more than 256 as implemented in the system.
2. The CPU implemented contains only 16 basic instructions of the CPU which can be further enhanced in-order to develop a more complex working CPU emulator.
3. The system of CPU implemented can be modified in order to map memory address and instructions with the help of registers and virtual memory.
4. More devices should be added, for example: mouse, graphic display, web camera, file system abstraction layer.
5. Memory I/O statistics can be created to watch I/O loading of the some certain memory ranges.
6. Better text editor and syntax highlighting should be added, maybe on-fly assembling of the code (because of simplifity of assembler language).
7. Stack memory implementation can be added to make possible to implement procedural programming.
8. Interrupt subsystem can be added to have an ability parallel working of several devices and quick reaction of CPU input events.
9. Library management can be added to have a couple of implemented several function, like trigonometric math functions, sorting algorithms and other useful algorithms.

7. LIST OF FIGURES AND TABLES

1. Figure 1 – Basic schema of computer model	- page 3;
2. Figure 2 – Virtual memory	- page 5;
3. Figure 3 – Test case 1	- page 12;
4. Figure 4 – Test case 2	- page 13;
5. Figure 5 – Test case 3	- page 15;
6. Figure 6 – Test case 4	- page 16;
7. Figure 7 – Test case 5	- page 17;
8. Figure 8 – Test case 6	- page 18;
9. Figure 9 – Test case 7	- page 19;
10. Figure 10 – Test case 8	- page 20;
11. Figure 11 – Test case 9	- page 21;
12. Figure 12 – Test case 10	- page 23;
13. Figure 13 – Program compilation	- page 24;
14. Figure 14 – Running result	- page 25;
15. Figure 15 – Multiplication algorithm	- page 27;
16. Figure 16 – Test case	- page 38;
17. Figure 17 – Test watches	- page 38;
18. Figure 18 – Testing	- page 39;
19. Figure 19 – Testing	- page 40;
20. Figure 20 – Testing	- page 41;
21. Figure 21 – Testing	- page 42;
22. Figure 22 – Testing	- page 42;
23. Figure 23 – Testing	- page 43;
24. Figure 24 – Program execution	- page 43;
25. Figure 25 – Testing	- page 44;
26. Figure 26 – Testing	- page 44;
27. Figure 27 – Testing	- page 45;
28. Figure 28 – Testing	- page 46;
29. Figure 29 – Test results	- page 47;
30. Figure 30 – Testing	- page 48;
31. Figure 31 – Testing	- page 49;
32. Figure 32 – Testing	- page 50;
33. Figure 33 – Testing	- page 51;
34. Figure 34 – Testing	- page 51;
35. Figure 35 – Testing	- page 52;
36. Figure 36 – Testing	- page 52;
37. Figure 37 – Testing	- page 53;
38. Figure 38 & 39 – Test results	- page 53;
39. Figure 40 – Main frame	- page 60;
1. Table 1 – Table of instructions	- page 4;
2. Table 2 – Attachments of the computer model	- page 5;
3. Table 3 – Registers explanation	- page 27;
4. Table 4 – Test results 1	- page 29;
5. Table 5 – Registers explanation	- page 30;
6. Table 6 – Test results 2	- page 32;
7. Table 7 – Test results 3	- page 36;
8. Table 8 – Package cpusimulator	- page 58;
9. Table 9 – Package cpusimulator.code	- page 58;
10. Table 10 – Package cpusimulator.devices	- page 58;
11. Table 11 – Package cpusimulator.memory	- page 59;

8. REFERENCES

- [1] W. Stallings. Computer organization and architecture: designing for performance. 7th edition. Upper Saddle River: Pearson/Prentice Hall, 2006.

- [2] G. Bezanov. Computer architectures. London: MIG Consulting, 2010.

- [3] M. Balch. Complete digital design: a comprehensive guide to digital electronics and computer system architecture. New York: McGraw-Hill, 2003.

- [4] B. Parhami. Computer architecture: from microprocessors to supercomputers. New York: Oxford University Press, 2005.

- [5] S. Coope, J. Cowley, and N. Willis. Computer systems: architecture, networks and communications. London: McGraw-Hill, 2002.

- [6] P. Scherz. Practical electronics for inventors. 2nd edition. New York: McGraw-Hill, 2007.

- [7] N. Storey. Electronics: a systems approach. 4th edition. Harlow: Prentice Hall, 2009.

- [8] B. Forouzan, F. Mosharraf. Foundations of computer science

9. APPENDICES

APPENDIX 1. Package organization

There are four main packages in the CPU simulator. Here you can see their list with a brief explanation about the work of each class inside them. This section can add additional help about the understanding of the section “Experimental procedure”.

Table 8:

Package cpusimulator

AsmSourceDialog	This dialog used to edit asm source.
Breakpoint16	Breakpoint used to handle READ/WRITE operations during CPU simulation.
CPU16Model	This is 16-bit CPU model with ALU and attached memory
InfoMessageDialog	This form is used to show messages to user
MainFrame	This is a main form and a start point of the application.

Table 9:

Package cpusimulator.code

CodeManager	This class used to manage machine code in memory.
Instruction	Instruction enum.
Operand	This class contains operand type and data.
Operand.Type	Type of operand.

Table 10:

Package cpusimulator.devices

Display	This is a display device.
Keyboard	Keyboard device, used to handle input events.
MemoryCard	Memory card.

Table 11:

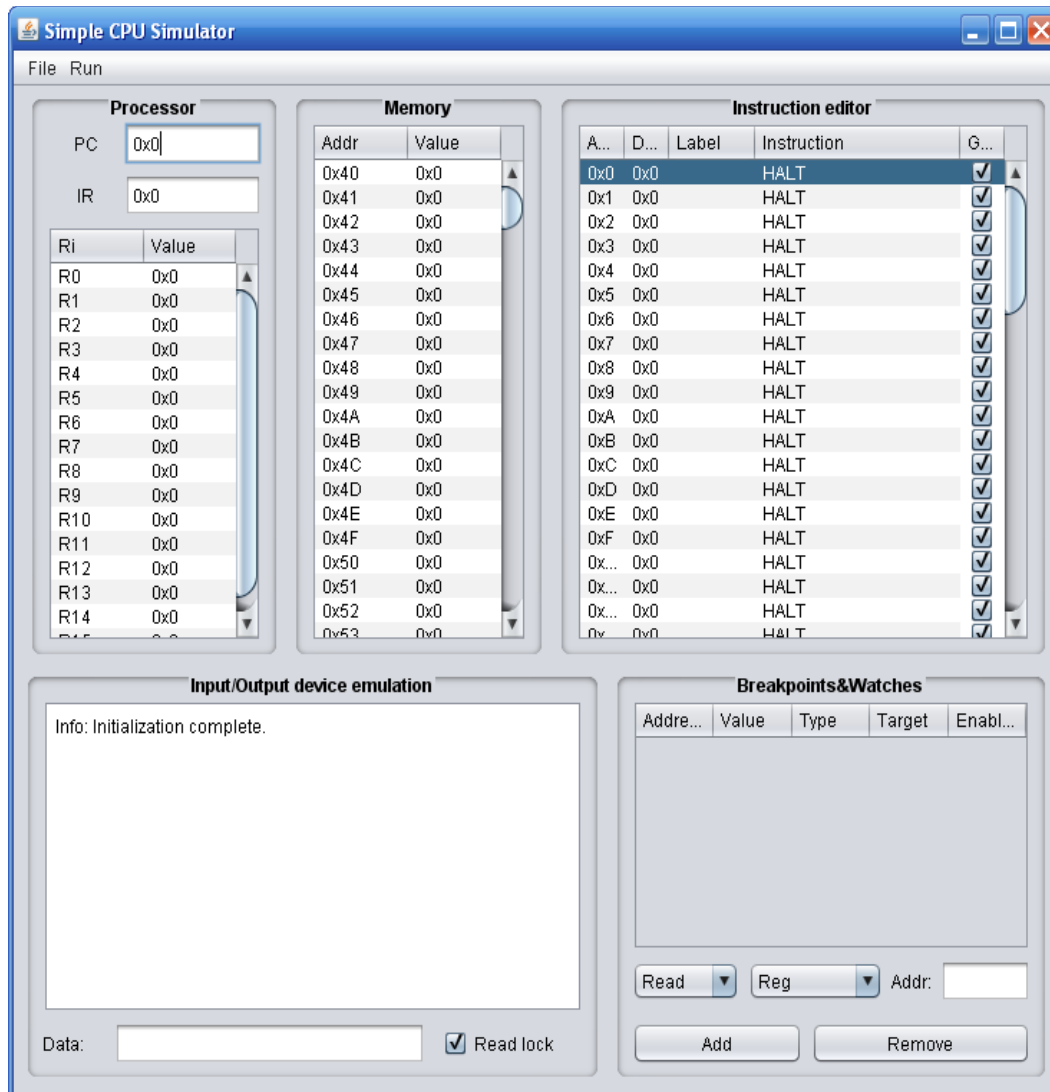
Package cpusimulator.memory

BreakpointMemory16	Memory used to handle read and writes at some addresses.
DataMemory16	Data memory model.
InstructionMemory16	Class represents instruction memory model.
VirtualMemory16	This is very useful thing designed to compose many devices into one bus system.

APPENDIX 2. Main view of the program

Mainframe.java

Figure 38:



APPENDIX 3. Source code

```
package cpusimulator;

import cpusimulator.code.AssembleException;
import cpusimulator.code.CodeManager;
import cpusimulator.devices.Display;
import cpusimulator.devices.Keyboard;
import cpusimulator.devices.MemoryCard;
import cpusimulator.memory.BreakpointMemory16;
import cpusimulator.memory.DataMemory16;
import cpusimulator.memory.InstructionMemory16;
import cpusimulator.memory.VirtualMemory16;
import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.util.ArrayList;
import javax.swing.JTable;
import javax.swing.event.TableModelEvent;
import javax.swing.event.TableModelListener;
import javax.swing.table.DefaultTableModel;

/**
 * This is a main form and a start point of the application.
 * This form have a CPU model and contains elements to rule that model.
 * All results (memory values, registers, code) will immediately shown for
 * user.
 *
 * @author Daniel Alexandre 2011
 */
public class MainFrame extends javax.swing.JFrame {

    /** Our CPU model. */
    private CPU16Model model;
    /** Memory to store data. */
    private DataMemory16 datamem;
    /** Memory to store code. */
    private InstructionMemory16 codemem;
    /** Breakpointed data memory. Used to add data memory breakpoints. */
    private BreakpointMemory16 bdatamem;
    /** Breakpointed code memory. Used to add code mmeory breakpoints. */
    private BreakpointMemory16 bcodemem;
    /** This is our display device, attached to virtual memory. */
    private Display display;
    /** This is our keyboard device, attached to virtual memory. */
    private Keyboard keyboard;
    /** Code manager is used to assemble/disassemble code. */
    private CodeManager manager;
    /** This flag used to avoid endless recursion caused by data changing
and data handling in the same class. */
    private boolean instructionTableChangeAccept = true;
    /** This flag used to avoid endless recursion caused by data changing
and data handling in the same class. */
    private boolean memoryTableChangeAccept = true;
    /** This flag used to avoid endless recursion caused by data changing
and data handling in the same class. */
```

```

private boolean registersTableChangeAccept = true;
/** This flag used to avoid endless recursion caused by data changing
and data handling in the same class. */
private boolean breakpointsTableChangeAccept = true;
/** This flag indicates CPU interruption state. */
private boolean cpuInterrupted = false;
/** This flag indicates CPU running state. */
private boolean cpuRunning = false;
/** This flag indicates that we have wrong breakpoint address in some
case. */
private boolean breakpointReadFail = false;
/** Our source editor dialog instance. */
private AsmSourceDialog sourceEditor = new AsmSourceDialog(this, true,
new Runnable() {
    @Override public void run() { cancelEditSource(); }

```

```

    }, new Runnable() {
        @Override public void run() { acceptEditSource(); }
    });
    /** Array of created breakpoints. */
    private ArrayList<Breakpoint16> breakpoints = new
ArrayList<Breakpoint16>();
    /** Breakpoint which is created and used only when we run code to
cursor. */
    private Breakpoint16 toCursorBreakpoint;

    /**
     * Creates new MainFrame.
     */
    public MainFrame() {
        initComponents(); // initializes components of main form

        // invalid memory address used to handle invalid memory errors
        Runnable invalidMemoryAddress = new Runnable() {
            public void run() { handleError("Invalid memory address!"); }
        };

        // creating virtual memory from 0x00 to 0xFF (it can't be used
until we haven't attached memory to it)
        VirtualMemory16 vmemory = new VirtualMemory16(256, new Runnable() {
            public void run() { handleError("Unmapped block error!"); }
        }, invalidMemoryAddress);
        // creating handler to handle breakpoint wrong operations
        Runnable breakpointReadCheck = new Runnable() {
            @Override public void run() { breakpointReadFail = true; }
        };

        // creating data memory: from 0x40 to 0xFD (length - 0xBE)
        datamem = new DataMemory16(0xBE, invalidMemoryAddress);
        // creating breakpointed memory using data memory
        bdatamem = new BreakpointMemory16(datamem, 0xBE,
breakpointReadCheck);

        // handler which will handle attempts to change code (can happen if
we will have wrong assembler code)
        Runnable changeCodeAttempt = new Runnable() {
            public void run() { handleError("Attempt to change code!"); }
        };

        // creating instruction memory: from 0x00 to 0x3F (length - 0x40)
        codemem = new InstructionMemory16(0x40, invalidMemoryAddress,
changeCodeAttempt);
        // creating breakpointed memory using instruction memory
        bcodemem = new BreakpointMemory16(codemem, 0x40,
breakpointReadCheck);
        codemem.lock(); // lock memory to avoid code corruption by asm code
execution

        // creating our display
        display = new Display(false, invalidMemoryAddress) {
            @Override public void numberWrited(short number)
{ handleOutput(String.valueOf(number)); }
        };
        // creating our keyboard
        keyboard = new Keyboard(new Runnable() {
            public void run() { handleError("Invalid memory address"); }
        });

```

```

keyboard.setLocking(true); // set keyboard locking by-default

final int CODE_OFFSET = 0x0;
final int DATA_OFFSET = CODE_OFFSET + codemem.getLength();
final int KEYBOARD_OFFSET = 0xFE;
final int DISPLAY_OFFSET = 0xFF;

// mapping memory cards and devices to virtual memory
vmemory.map(new MemoryCard(bcodemem, codemem.getLength()), 0,
CODE_OFFSET);
vmemory.map(new MemoryCard(bdatamem, datamem.getLength()), 0,
DATA_OFFSET);
// and there we have to map two future devices: keyboard and
display to 0xFE and 0xFF address
// before we will add devices, any reads from or writes to two last
cells will cause "Unmapped block error!"
vmemory.map(keyboard, 0, KEYBOARD_OFFSET);
vmemory.map(display, 0, DISPLAY_OFFSET);

// creating instruction manager which we will use to
assemble/disassemble code
manager = new CodeManager(codemem, (byte)0);

// creating cpu which operates with our virtual memory
model = new CPU16Model(vmemory, (byte)0, new Runnable() {
    public void run() { cpuInterrupted = true; handleInfo("Program
was halted!"); }
}, new Runnable() {
    public void run() { handleError("Wrong instruction found!"); }
}, breakpointReadCheck);
// now we can run cpu by invoking model.emulateInstruction()

registersTable.getModel().addTableModelListener(new
TableModelListener() {
    @Override public void tableChanged(TableModelEvent e)
{ registersTableChanged(e); }
});

// make memory table model to use it in the JTable swing component
Object[][] memoryTableData = new Object[datamem.getLength()][2];
for(int i = 0; i < datamem.getLength(); i++) memoryTableData[i][0]
= "0x" + Integer.toString(DATA_OFFSET + i, 16).toUpperCase();
memoryTable.setModel(new DefaultTableModel(memoryTableData, new
String [] { "Addr", "Value" }) {
    Class[] types = new Class [] { Object.class, Object.class };
    boolean[] canEdit = new boolean [] { false, true };

    @Override public Class getColumnClass(int columnIndex) { return
types [columnIndex]; }
    @Override public boolean isCellEditable(int rowIndex, int
columnIndex) { return canEdit [columnIndex]; }
});
// add table changing listener
memoryTable.getModel().addTableModelListener(new
TableModelListener() {
    @Override public void tableChanged(TableModelEvent e)
{ memoryTableChanged(e); }
});

// make instruction table model to use it in the JTable swing
component

```



```

        Object[][] instructionsTableData = new Object[codemem.getLength()]
[5];
        for(int i = 0; i < codemem.getLength(); i++)
instructionsTableData[i][0] = "0x" + Integer.toHexString(i).toUpperCase();
        instructionsTable.setModel(new
DefaultTableModel(instructionsTableData, new String[] { "Addr", "Data",
"Label", "Instruction", "Good" }) {
            Class[] types = new Class [] { Object.class, Object.class,
Object.class, Object.class, Boolean.class };
            boolean[] canEdit = new boolean [] { false, true, true, true,
false };

            @Override public Class getColumnClass(int columnIndex) { return
types [columnIndex]; }
            @Override public boolean isCellEditable(int rowIndex, int
columnIndex) { return canEdit [columnIndex]; }
        });
        // adding table changing listener
        instructionsTable.getModel().addTableModelListener(new
TableModelListener() {
            @Override public void tableChanged(TableModelEvent e)
{ instructionsTableChanged(e); }
        });
        // some cosmetical changes
        instructionsTable.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
        int totalWidth =
instructionsTable.getColumnModel().getTotalColumnWidth();
        instructionsTable.getColumnModel().getColumn(0).setPreferredWidth(t
otalWidth * 1 / 10);
        instructionsTable.getColumnModel().getColumn(1).setPreferredWidth(t
otalWidth * 1 / 10);
        instructionsTable.getColumnModel().getColumn(2).setPreferredWidth(t
otalWidth * 2 / 10);
        instructionsTable.getColumnModel().getColumn(3).setPreferredWidth(t
otalWidth * 5 / 10);
        instructionsTable.getColumnModel().getColumn(4).setPreferredWidth(t
otalWidth * 1 / 10);
        instructionsTable.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS)
;

        // adding listenern to breakpoints table
        breakpointsWatchesTable.getModel().addTableModelListener(new
TableModelListener() {
            @Override public void tableChanged(TableModelEvent e)
{ breakpointsTableChanged(e); }
        });

        // updating all data on the form to have zeros at start of app
        updateInputLock();
        updateMemory();
        updateRegisters();
        updatePCandIR();
        updateInstructions();
        updateBreakpoints();

        handleInfo("Initialization complete.");
    }

    /**
     * This method should be called every time we have some error - a
     serious problem.
     * @param message information about a problem

```



```

    */
    private void handleError(String message) {
        synchronized (terminalTextArea) {
            terminalTextArea.setText(terminalTextArea.getText() + ("Error: "
+ message + "\n"));
            terminalTextArea.setCaretPosition(terminalTextArea.getDocument(
).getLength());
        }
    }

    /**
     * This method should be called every time we have some information
     which can be useful by user.
     * @param message useful information
     */
    private void handleInfo(String message) {
        synchronized (terminalTextArea) {
            terminalTextArea.setText(terminalTextArea.getText() + ("Info: "
+ message + "\n"));
            terminalTextArea.setCaretPosition(terminalTextArea.getDocument(
).getLength());
        }
    }

    /**
     * This method should be called every time when compiler detect errors
     in assembler code.
     * @param message assembling information
     */
    private void handleAsmError(String message) {
        synchronized (terminalTextArea) {
            terminalTextArea.setText(terminalTextArea.getText() + ("Asm
error: " + message + "\n"));
            terminalTextArea.setCaretPosition(terminalTextArea.getDocument(
).getLength());
        }
    }

    /**
     * This method should be called every time when we have some output
     from display.
     * @param message display output
     */
    private void handleOutput(String message) {
        synchronized (terminalTextArea) {
            terminalTextArea.setText(terminalTextArea.getText() + (">> " +
message + "\n"));
            terminalTextArea.setCaretPosition(terminalTextArea.getDocument(
).getLength());
        }
    }

    /**
     * This method should be called every time when we have some input from
     keyboard.
     * @param message keyboard input
     */
    private void handleInput(String message) {
        synchronized (terminalTextArea) {
            terminalTextArea.setText(terminalTextArea.getText() + ("<< " +
message + "\n"));

```

```

        terminalTextArea.setCaretPosition(terminalTextArea.getDocument(
).getLength());
    }
}

/** Updates memory table. */
private void updateMemory() {
    memoryTableChangeAccept = false;
    for(int i = 0; i < datamem.getLength(); i++)
memoryTable.getModel().setValueAt("0x" +
Integer.toString(datamem.read((byte)i) & 0xFFFF, 16).toUpperCase(), i, 1);
    memoryTableChangeAccept = true;
}

/** Updates registers table. */
private void updateRegisters() {
    registersTableChangeAccept = false;
    for(int i = 0; i < 16; i++)
registersTable.getModel().setValueAt("0x" +
Integer.toString(model.getRegValue(i), 16).toUpperCase(), i, 1);
    registersTableChangeAccept = true;
}

/** Updates PC and IR textfields. */
private void updatePCandIR() {
    programCounterTextField.setText("0x" +
Integer.toString(model.getPCValue() & 0xFF, 16).toUpperCase());
    instructionRegisterTextField.setText("0x" +
Integer.toString(model.getIRValue() & 0xFFFF, 16).toUpperCase());
    instructionsTable.getSelectionModel().setSelectionInterval(model.ge
tPCValue() & 0xFF, model.getPCValue() & 0xFF);
}

/** Updates instructions table. */
private void updateInstructions() {
    instructionTableChangeAccept = false;
    manager.gotoOffset((byte)0);
    // fill static values at first
    for(int i = 0; i < codemem.getLength(); i++) {
        instructionsTable.getModel().setValueAt("0x" +
Integer.toHexString(codemem.read((byte)i) & 0xFFFF).toUpperCase(), i, 1);
        instructionsTable.getModel().setValueAt(null, i, 3);
        instructionsTable.getModel().setValueAt(new Boolean(false), i,
4);
    }
    int offset;
    // and secondary, disassemble instruction
    while((offset = manager.getOffset()) != codemem.getLength()) {
        String disassembled = manager.disassembleInstruction();
        instructionsTable.getModel().setValueAt(disassembled == null ?
"???" : disassembled, offset, 3);
        instructionsTable.getModel().setValueAt(new
Boolean(disassembled != null), offset, 4);
    }
    instructionTableChangeAccept = true;
}

/** Updates breakpoints table. */
private void updateBreakpoints() {
    breakpointsTableChangeAccept = false;

```



```

        DefaultTableModel model =
(DefaultTableModel)breakpointsWatchesTable.getModel();
        model.setNumRows(0); // remove all rows
        for(Breakpoint16 breakpoint : breakpoints) {
            // and add rows generated from breakpoints info in our list of
breakpoints
            model.addRow(new Object[] {
                "0x" +
Integer.toHexString(breakpoint.getAddress()).toUpperCase(),
                "0x" + Integer.toHexString(breakpoint.getValue() &
0xFFFF).toUpperCase(),
                breakpoint.getType() == Breakpoint16.BREAKPOINT_CHANGE ?
"Change" :
                breakpoint.getType() == Breakpoint16.BREAKPOINT_READ ?
"Read" :
                breakpoint.getType() == Breakpoint16.BREAKPOINT_WRITE ?
"Write" : "??",
                breakpoint.getMemory() == codemem ? "Code" :
                breakpoint.getMemory() == datamem ? "Data" : "Reg",
                new Boolean(breakpoint.isEnabled()),
            });
        }
        breakpointsTableChangeAccept = true;
    }

    /**
     * Method called when data is inputed from bottom textfield which used
as keyboard.
     * @param text inputed text
     */
    private void dataInputed(String text) {
        try {
            keyboard.inputNumber(Integer.parseInt(text)); // send typed
data to keyboard device (it's just emulation)
            handleInput(text); // print our inputed text to console
            inputTextField.setText(""); // and clear input textfield
        } catch (NumberFormatException exception) {
            handleError("Number expected!");
        }
    }

    /** Makes keyboard locked or unlocked depends on lock-input checkbox
state. */
    private void updateInputLock() {
        keyboard.setLocking(inputLockCheckBox.isSelected());
    }

    /**
     * Called every time when instructions table have been changed.
     * @param e info about event
     */
    private void instructionsTableChanged(TableModelEvent e) {
        if(instructionTableChangeAccept) {
            int column = e.getColumn();
            switch(column) {
                case 0: // address changed - strange, because address
changing is not allowed
                    throw new RuntimeException("row is not editable");
                case 1: // data changed: let disassemble instruction
                    {
                        int row = e.getFirstRow();

```

```

        String data = (String)instructionsTable.getValueAt(row,
column);
        if(data == null || (data = data.trim()).isEmpty()) data
= "";
        if(!data.isEmpty()) {
            try {
                int datai;
                if(data.startsWith("0x")) datai =
Integer.parseInt(data.substring(2), 16);
                else datai = Integer.parseInt(data);
                datai &= 0xFFFF; codemem.unlock();
                codemem.write((byte)row, (short)datai);
                codemem.lock();
                manager.gotoOffset((byte)row);
                String disassembled =
manager.disassembleInstruction();
                instructionTableChangeAccept = false;
                instructionsTable.setValueAt("0x" +
Integer.toHexString(datai).toUpperCase(), row, 1);
                instructionsTable.setValueAt(disassembled ==
null ? "???" : disassembled, row, 3);
                instructionsTable.setValueAt(new
Boolean(disassembled != null), row, 4);
                instructionTableChangeAccept = true;
            } catch (NumberFormatException exception) {
            }
        }
        updateBreakpoints();
        break;
    }
    case 2: // label changed: just make it correct and add
label to manager
    {
        int row = e.getFirstRow();
        String label =
(String)instructionsTable.getValueAt(row, column);
        if(label == null || (label = label.trim()).isEmpty() ||
label.equals(":")) {
            label = "";
        } else {
            if(label.endsWith(":")) label = label.substring(0,
label.length() - 1).trim();
        }
        instructionTableChangeAccept = false;
        instructionsTable.setValueAt(label, row, column);
        instructionTableChangeAccept = true;
        manager.removeLabel(row);
        if(!label.isEmpty()) manager.addLabel(label, row);
        break;
    }
    case 3: // instruction changed: let assemble instruction
    {
        int row = e.getFirstRow();
        String instruction =
(String)instructionsTable.getValueAt(row, column);
        if(instruction == null || (instruction =
instruction.trim()).isEmpty()) instruction = "";
        if(!instruction.isEmpty()) {
            boolean good;
            try {

```

```

        manager.gotoOffset((byte) row);
        manager.assembleInstruction(instruction);
        good = true;
    } catch (AssembleException exception) {
        handleAsmError(exception.getMessage());
        good = false;
    }
    instructionTableChangeAccept = false;
    instructionsTable.setValueAt("0x" +
Integer.toHexString(codemem.read((byte) row) & 0xFFFF).toUpperCase(), row,
1);

        instructionsTable.setValueAt(instruction, row, 3);
        instructionsTable.setValueAt(new Boolean(good),
row, 4);

        instructionTableChangeAccept = true;
    }
    updateBreakpoints();
    break;
}
case 4:
    throw new RuntimeException("row is not editable");
default:
    throw new RuntimeException("not implemented since table
have 2 editable columns");
    }
}

/**
 * Called each time when memory table have been changed.
 * @param e event information
 */
private void memoryTableChanged(TableModelEvent e) {
    if(memoryTableChangeAccept) {
        int column = e.getColumn();
        switch(column) {
            case 0: // address changed: strange, because address isn't
allowed to edit
                throw new RuntimeException("row is not editable");
            case 1: // data changed: let write changed data to memory
{
                int row = e.getFirstRow();
                String data = (String)memoryTable.getValueAt(row,
column);

                if(data == null || (data = data.trim()).isEmpty()) data
= "";

                if(!data.isEmpty()) {
                    try {
                        int datai;
                        if(data.startsWith("0x")) datai =
Integer.parseInt(data.substring(2), 16);
                        else datai = Integer.parseInt(data);
                        datai &= 0xFFFF;
                        datamem.write((byte) row, (short) datai);
                        memoryTableChangeAccept = false;
                        memoryTable.setValueAt("0x" +
Integer.toHexString(datai).toUpperCase(), row, 1);
                        memoryTableChangeAccept = true;
                    } catch (NumberFormatException exception) {
                    }
                }
            }
        }
    }
}

```



```

        updateBreakpoints();
        break;
    }
    default:
        throw new RuntimeException("not implemented since table
have 1 editable column");
    }
}

/**
 * Called each time when user changes registers table contents.
 * @param e event information
 */
private void registersTableChanged(TableModelEvent e) {
    if(registersTableChangeAccept) {
        int column = e.getColumn();
        switch(column) {
            case 0: // register name changed: strange, we haven't
allowed this
                throw new RuntimeException("row is not editable");
            case 1: // changed register data: let change it in the cpu
{
                int row = e.getFirstRow();
                String data = (String)registersTable.getValueAt(row,
column);
                if(data == null || (data = data.trim()).isEmpty()) data
= "";
                if(!data.isEmpty()) {
                    try {
                        int datai;
                        if(data.startsWith("0x")) datai =
Integer.parseInt(data.substring(2), 16);
                        else datai = Integer.parseInt(data);
                        datai &= 0xFFFF;
                        model.setRegValue(row, (short)datai);
                        registersTableChangeAccept = false;
                        registersTable.setValueAt("0x" +
Integer.toHexString(datai).toUpperCase(), row, 1);
                        registersTableChangeAccept = true;
                    } catch (NumberFormatException exception) {
                    }
                }
                break;
            }
            default:
                throw new RuntimeException("not implemented since table
have 1 editable column");
        }
    }
}

/**
 * Called every time when breakpoints table has been changed.
 * @param e event information
 */
private void breakpointsTableChanged(TableModelEvent e) {
    if(breakpointsTableChangeAccept) {
        int column = e.getColumn();
        switch(column) {
            case 0:

```



```

        throw new RuntimeException("row is not editable");
    case 1:
        throw new RuntimeException("row is not editable");
    case 2:
        throw new RuntimeException("row is not editable");
    case 3:
        throw new RuntimeException("row is not editable");
    case 4: // only breakpoint enabling/disabling is allowed
    {
        int row = e.getFirstRow();
        Boolean data =
(Boolean)breakpointsWatchesTable.getValueAt(row, column);
        Breakpoint16 breakpoint = breakpoints.get(row);
        breakpoint.setEnabled(data.booleanValue());
        break;
    }
    default:
        throw new RuntimeException("not implemented since table
have 1 editable column");
    }
}

/** Called every time we change PC field. */
private void doChangePC() {
    String data = programCounterTextField.getText();
    if(data == null || (data = data.trim()).isEmpty()) data = "";
    if(!data.isEmpty()) {
        try {
            int datai;
            if(data.startsWith("0x")) datai =
Integer.parseInt(data.substring(2), 16);
            else datai = Integer.parseInt(data);
            model.setPCValue((byte)datai);
            programCounterTextField.setText("0x" +
Integer.toHexString(datai).toUpperCase());
        } catch(NumberFormatException exception) {
        }
    }
}

/** Called every time we have selected Reset menu. */
private void doReset() {
    model.setPCValue((byte)0);
    updatePCandIR();
}

/** Called every time we have selected New menu. */
private void doNew() {
    codemem.unlock();
    for(int i = 0; i < codemem.getLength(); i++)
codemem.write((byte)i, (short)0);
    codemem.lock();
    for(int i = 0; i < datamem.getLength(); i++)
datamem.write((byte)i, (short)0);
    model.reset();
    updateRegisters();
    updatePCandIR();
    updateMemory();
    updateInstructions();
    updateBreakpoints();
}

```

```

}

/** Called every time we have selected Step menu. */
private void doStep() {
    Thread thread = new Thread(new Runnable() {
        public void run() {
            cpuInterrupted = false;
            handleInfo("CPU runned...");
            cpuRunning = true;
            model.fetchInstruction();
            if(!cpuInterrupted) model.emulateInstruction();
            cpuRunning = false;
            handleInfo("CPU running finished");
            updateRegisters();
            updatePCandIR();
            updateMemory();
            updateInstructions();
            updateBreakpoints();
        }
    });
    thread.setDaemon(true);
    thread.start();
}

/** Called every time we have selected Run menu */
private void doRun() {
    if(cpuRunning) {
        handleInfo("CPU already started!");
    }
    Thread thread = new Thread(new Runnable() {
        public void run() {
            try {
                cpuInterrupted = false;
                cpuRunning = true;
                handleInfo("CPU runned...");
                while(cpuRunning) {
                    model.fetchInstruction();
                    if(cpuInterrupted) break;
                    model.emulateInstruction();
                    updateRegisters();
                    updatePCandIR();
                    updateMemory();
                    updateBreakpoints();
                }
                if(cpuRunning) handleInfo("CPU running finished");
                if(!cpuRunning) handleInfo("CPU running stopped");
            } finally {
                cpuRunning = false;
                if(toCursorBreakpoint != null) {
                    toCursorBreakpoint.remove();
                    toCursorBreakpoint = null;
                }
            }
        }
    });
    thread.setDaemon(true);
    thread.start();
}

/** Called every time when we have cancelled asm source editing in the
edit asm dialog. */

```

```

private void cancelEditSource() {
    sourceEditor.setVisible(false);
    setVisible(true);
}

/** Called every time when we have accepted asm source editing in the
edit asm dialog. */
private void acceptEditSource() {
    String[] asmLines = sourceEditor.getAsmLines();
    if(asmLines.length > instructionsTable.getRowCount()) {
        new InfoMessageDialog(this, "Sorry, this code is too long to
store it into memory.").setVisible(true);
        return;
    } storeAsmLines(asmLines);
    sourceEditor.setVisible(false);
    setVisible(true);
}

/** Called every time when user selected "Run to cursor" menu item. */
private void doRunToCursor() {
    int selectedRow = instructionsTable.getSelectedRow();
    if(selectedRow >= 0) {
        toCursorBreakpoint =
bcodemem.createBreakpoint((byte)selectedRow, Breakpoint16.BREAKPOINT_READ);
        toCursorBreakpoint.setEnabled(true);
        toCursorBreakpoint.setHandler(new Runnable() {
            @Override
            public void run() {
                if(cpuRunning) {
                    cpuInterrupted = true;
                    handleInfo("Cursor is reached");
                }
            }
        });
        doRun();
    }
}

/** Called every time when user have selected Stop menu item. */
private void doStop() {
    cpuRunning = false;
}

/** Called every time when user have selected "Import data" menu item.
*/
private void doImportData() {
    jFileChooser_Import.showOpenDialog(this);
    File file = jFileChooser_Import.getSelectedFile();
    if(file != null) {
        if(!file.exists()) {
            new InfoMessageDialog(this, "File doesn't
exists").setVisible(true);
            return;
        }
        if(file.isDirectory()) {
            new InfoMessageDialog(this, "Please, select a file, not a
directory").setVisible(true);
            return;
        }
        long length = file.length();

```

```

        if(length != datamem.getLength() * 2) {
            new InfoMessageDialog(this, "File must have " +
datamem.getLength() * 2 + " bytes of length.").setVisible(true);
            return;
        }
        try {
            short[] data = new short[datamem.getLength()];
            DataInputStream input = new DataInputStream(new
FileInputStream(file));
            for(int i = 0; i < data.length; i++) data[i] =
input.readShort();
            input.close();
            for(int i = 0; i < data.length; i++)
datamem.write((byte)i, data[i]);
            handleInfo("Memory data has been loaded ok");
            updateMemory();
        } catch(IOException ioexception) {
            new InfoMessageDialog(this, "An I/O exception has been
ocurred.").setVisible(true);
        }
    }
}

/** Called every time when user have selected "Import asm code" menu
item. */
private void doImportAsmCode() {
    jFileChooser_Import.showOpenDialog(this);
    File file = jFileChooser_Import.getSelectedFile();
    if(file != null) {
        if(!file.exists()) {
            new InfoMessageDialog(this, "File doesn't
exists").setVisible(true);
            return;
        }
        if(file.isDirectory()) {
            new InfoMessageDialog(this, "Please, select a file, not a
directory").setVisible(true);
            return;
        }
        String[] asmLines;
        try {
            BufferedReader reader = new BufferedReader(new
InputStreamReader(new FileInputStream(file)));
            ArrayList<String> linesAL = new ArrayList<String>();
            String line;
            while((line = reader.readLine()) != null) {
                line = line.trim();
                if(!line.isEmpty()) linesAL.add(line);
            }
            reader.close();
            asmLines = new String[linesAL.size()];
            for(int i = 0; i < linesAL.size(); i++) asmLines[i] =
linesAL.get(i);
        } catch(IOException ioexception) {
            new InfoMessageDialog(this, "An I/O exception has been
ocurred.").setVisible(true);
            return;
        }
        if(asmLines.length > instructionsTable.getRowCount()) {
            new InfoMessageDialog(this, "Sorry, this code is too long
to store it into memory.").setVisible(true);

```



```

        return;
    }
    storeAsmLines(asmLines);
    updateInstructions();
    handleInfo("Assembler source file has been loaded");
}

/** Helper method which assembles and stores asm lines to instructions
table. */
private void storeAsmLines(String[] asmLines) {
    for(int i = 0; i < asmLines.length; i++) {
        String line = asmLines[i];
        String label = "";
        int colonIndex = line.indexOf(':');
        if(colonIndex >= 0) {
            label = line.substring(0, colonIndex).trim();
            line = line.substring(colonIndex + 1).trim();
        }
        instructionTableChangeAccept = false;
        instructionsTable.setValueAt(label, i, 2);
        instructionTableChangeAccept = true;
        manager.removeLabel(i);
        if(!label.isEmpty()) manager.addLabel(label, i);

        boolean good;
        try { manager.gotoOffset((byte)i);
            manager.assembleInstruction(line);
            good = true;
        } catch(AssembleException exception) {
            handleAsmError("Offset 0x" + Integer.toHexString(i &
0xFF).toUpperCase() + ":" + exception.getMessage());
            good = false;
        }
        instructionTableChangeAccept = false;
        instructionsTable.setValueAt("0x" +
Integer.toHexString(codemem.read((byte)i) & 0xFFFF).toUpperCase(), i, 1);
        instructionsTable.setValueAt(line, i, 3);
        instructionsTable.setValueAt(new Boolean(good), i, 4);
        instructionTableChangeAccept = true;
    }
}

/** Called every time when user have selected "Import binary code" menu
item. */
private void doImportBinaryCode() {
    jFileChooser_Import.showOpenDialog(this);
    File file = jFileChooser_Import.getSelectedFile();
    if(file != null) {
        if(!file.exists()) {
            new InfoMessageDialog(this, "File doesn't
exists").setVisible(true);
            return;
        }
        if(file.isDirectory()) {
            new InfoMessageDialog(this, "Please, select a file, not a
directory").setVisible(true);
            return;
        }
        long length = file.length();

```

```

        if(length != codemem.getLength() * 2) {
            new InfoMessageDialog(this, "File must have " +
codemem.getLength() * 2 + " bytes of length.").setVisible(true);
            return;
        }
        try {
            short[] data = new short[codemem.getLength()];
            DataInputStream input = new DataInputStream(new
FileInputStream(file));
            for(int i = 0; i < data.length; i++) data[i] =
input.readShort(); input.close();
            codemem.unlock();
            for(int i = 0; i < data.length; i++)
codemem.write((byte)i, data[i]);
            codemem.lock();
            handleInfo("Instruction data has been loaded ok");
            updateInstructions();
        } catch(IOException ioexception) {
            new InfoMessageDialog(this, "An I/O exception has been
ocurred.").setVisible(true);
        }
    }
}

/** Called every time when user have selected "Export data" menu item.
*/
private void doExportData() {
    jFileChooser_Export.showSaveDialog(this);
    File file = jFileChooser_Export.getSelectedFile();
    if(file != null) {
        if(file.exists()) {
            if(!file.delete()) {
                new InfoMessageDialog(this, "Cann't overwrite
file").setVisible(true);
                return;
            }
        }
        try {
            if(!file.createNewFile()) {
                new InfoMessageDialog(this, "Cann't create
file").setVisible(true);
                return;
            }
        } catch(IOException ioexception) {
            new InfoMessageDialog(this, "An I/O exception has been
ocurred.").setVisible(true);
        }
        try {
            short[] data = new short[datamem.getLength()];
            for(int i = 0; i < data.length; i++) data[i] =
datamem.read((byte)i);
            DataOutputStream output = new DataOutputStream(new
FileOutputStream(file));
            for(int i = 0; i < data.length; i++)
output.writeShort(data[i]);
            output.close();
            handleInfo("Memory data has been saved ok");
        } catch(IOException ioexception) {
            new InfoMessageDialog(this, "An I/O exception has been
ocurred.").setVisible(true);
        }
    }
}

```

```

    }
}

/** Called every time when user have selected "Export asm code" menu
item. */
private void doExportAsmCode() {
    jFileChooser_Export.showSaveDialog(this);
    File file = jFileChooser_Export.getSelectedFile();
    if(file != null) {
        if(file.exists()) {
            if(!file.delete()) {
                new InfoMessageDialog(this, "Cann't overwrite
file").setVisible(true);
                return;
            }
        }
        try {
            if(!file.createNewFile()) {
                new InfoMessageDialog(this, "Cann't create
file").setVisible(true);
                return;
            }
        } catch(IOException ioexception) {
            new InfoMessageDialog(this, "An I/O exception has been
ocurred.").setVisible(true);
        }
        String[] asmLines = loadAsmLines();
        try {
            String separator = System.getProperty("line.separator");
            OutputStreamWriter writer = new OutputStreamWriter(new
FileOutputStream(file));
            for(String line : asmLines) {
                writer.write(line);
                writer.write(separator);
            }
            writer.close();
            handleInfo("Instruction asm source has been saved ok");
        } catch(IOException ioexception) {
            new InfoMessageDialog(this, "An I/O exception has been
ocurred.").setVisible(true);
        }
    }
}

/** Called every time when user have selected "Export binary code" menu
item. */
private void doExportBinaryCode() {
    jFileChooser_Export.showSaveDialog(this);
    File file = jFileChooser_Export.getSelectedFile();
    if(file != null) {
        if(file.exists()) {
            if(!file.delete()) {
                new InfoMessageDialog(this, "Cann't overwrite
file").setVisible(true);
                return;
            }
        }
        try {
            if(!file.createNewFile()) {

```

```

        new InfoMessageDialog(this, "Cann't create
file").setVisible(true);
        return;
    }
} catch(IOException ioexception) {
    new InfoMessageDialog(this, "An I/O exception has been
ocurred.").setVisible(true);
}
try {
    short[] data = new short[codemem.getLength()];
    for(int i = 0; i < data.length; i++) data[i] =
codemem.read((byte)i);
    DataOutputStream output = new DataOutputStream(new
FileOutputStream(file));
    for(int i = 0; i < data.length; i++)
output.writeShort(data[i]);
    output.close();
    handleInfo("Instruction data has been saved ok");
} catch(IOException ioexception) {
    new InfoMessageDialog(this, "An I/O exception has been
ocurred.").setVisible(true);
}
}

}

/** Called every time when user have selected "Show asm source" menu
item. */
private void doEditAsmSource() {
    updateInstructions();
    String[] assemblerCode = loadAsmLines();
    sourceEditor.setAsmLines(assemblerCode);
    setVisible(false);
    sourceEditor.setVisible(true);
}

/** Helper method which loads and disassembles binary code from
instructions table. */
private String[] loadAsmLines() {
    String[] assemblerCode;
    synchronized(instructionsTable) {
        ArrayList<String> lines = new ArrayList<String>();
        for(int i = 0; i < instructionsTable.getRowCount(); i++) {
            String label = (String)instructionsTable.getValueAt(i, 2);
            String instruction =
(String)instructionsTable.getValueAt(i, 3);
            lines.add((label != null && !label.trim().isEmpty() ?
label.trim() + ": " : "") + instruction);
        }
        while(lines.size() > 1 && lines.get(lines.size() -
1).equalsIgnoreCase("halt") && lines.get(lines.size() -
2).equalsIgnoreCase("halt")) lines.remove(lines.size() - 1);
        assemblerCode = new String[lines.size()];
        for(int i = 0; i < lines.size(); i++) assemblerCode[i] =
lines.get(i);
    }
    return assemblerCode;
}

/** Called every time when user have pressed "Add breakpoint" button.
*/
private void doAddBreakpoint() {

```

```

        int type = breakpointTypeComboBox.getSelectedIndex();
        type = type == 0 ? Breakpoint16.BREAKPOINT_READ : type == 1 ?
Breakpoint16.BREAKPOINT_WRITE : type == 2 ?
Breakpoint16.BREAKPOINT_CHANGE : -1;
        if(type < 0) throw new RuntimeException("Unknown breakpoint type");
        int memtype = breakpointMemComboBox.getSelectedIndex();
        BreakpointMemory16 memory = memtype == 0 ?
model.getRegistersBreakpoints() : memtype == 1 ? bdatamem : memtype == 2 ?
bcodemem : null;
        if(memory == null) throw new RuntimeException("Unknown breakpoint
memory");
        String addressStr = breakpointAddressTextField.getText();
        if(addressStr == null || (addressStr =
addressStr.trim()).isEmpty()) return;
        int address;
        try {
            if(addressStr.startsWith("0x")) address =
Integer.parseInt(addressStr.substring(2), 16);
            else address = Integer.parseInt(addressStr);
            memory.read((byte)address);
        } catch(NumberFormatException exception) {
            handleError("Wrong breakpoint address");
            return;
        }
        if(breakpointReadFail == true) {
            breakpointReadFail = false;
            handleError("Breakpoint address is too large or less than
zero");
            return;
        }
        final Breakpoint16 breakpoint =
memory.createBreakpoint((byte)address, type);
        breakpoint.setEnabled(true);
        breakpoint.setHandler(new Runnable() {
            @Override
            public void run() {
                if(cpuRunning) {
                    cpuInterrupted = true;
                    handleInfo("Stopped because of breakpoint");
                    int index = breakpoints.indexOf(breakpoint);
                    if(index < 0)
breakpointsWatchesTable.getSelectionModel().clearSelection();
                    else
breakpointsWatchesTable.getSelectionModel().setSelectionInterval(index,
index);
                }
            }
        });
        breakpoints.add(breakpoint);
        updateBreakpoints();
    }

    /** Called every time when user have pressed "Remove breakpoint"
button. */
    private void doRemoveBreakpoint() {
        int row = breakpointsWatchesTable.getSelectedRow();
        if(row >= 0) {
            Breakpoint16 breakpoint = breakpoints.get(row);
            breakpoint.remove();
            breakpoints.remove(row);
            updateBreakpoints();
        }
    }

```



```

    }
}

@SuppressWarnings("unchecked")
// <editor-fold defaultstate="collapsed" desc="Generated Code">
private void initComponents() {

    jFileChooser_Import = new javax.swing.JFileChooser();
    jFileChooser_Export = new javax.swing.JFileChooser();
    processorPanel = new javax.swing.JPanel();
    jLabel1 = new javax.swing.JLabel();
    programCounterTextField = new javax.swing.JTextField();
    jLabel2 = new javax.swing.JLabel();
    instructionRegisterTextField = new javax.swing.JTextField();
    registersScrollPane = new javax.swing.JScrollPane();
    registersTable = new javax.swing.JTable();
    memoryPanel = new javax.swing.JPanel();
    memoryScrollPane = new javax.swing.JScrollPane();
    memoryTable = new javax.swing.JTable();
    instructionsPanel = new javax.swing.JPanel();
    instructionsScrollPane = new javax.swing.JScrollPane();
    instructionsTable = new javax.swing.JTable();
    ioPanel = new javax.swing.JPanel();
    terminalScrollPane = new javax.swing.JScrollPane();
    terminalTextArea = new javax.swing.JTextArea();
    jLabel3 = new javax.swing.JLabel();
    inputTextField = new javax.swing.JTextField();
    inputLockCheckBox = new javax.swing.JCheckBox();
    breakpointsWatchesPanel = new javax.swing.JPanel();
    breakpointsWatchesScrollPane = new javax.swing.JScrollPane();
    breakpointsWatchesTable = new javax.swing.JTable();
    addBreakpointButton = new javax.swing.JButton();
    removeBreakpointButton = new javax.swing.JButton();
    breakpointTypeComboBox = new javax.swing.JComboBox();
    breakpointMemComboBox = new javax.swing.JComboBox();
    jLabel4 = new javax.swing.JLabel();
    breakpointAddressTextField = new javax.swing.JTextField();
    jMenuBar1 = new javax.swing.JMenuBar();
    jMenu1 = new javax.swing.JMenu();
    jMenuItem10 = new javax.swing.JMenuItem();
    jSeparator1 = new javax.swing.JPopupMenu.Separator();
    jMenu3 = new javax.swing.JMenu();
    jMenuItem6 = new javax.swing.JMenuItem();
    jMenuItem7 = new javax.swing.JMenuItem();
    jMenuItem13 = new javax.swing.JMenuItem();
    jMenu4 = new javax.swing.JMenu();
    jMenuItem8 = new javax.swing.JMenuItem();
    jMenuItem9 = new javax.swing.JMenuItem();
    jMenuItem12 = new javax.swing.JMenuItem();
    jMenuItem14 = new javax.swing.JMenuItem();
    jSeparator2 = new javax.swing.JPopupMenu.Separator();
    jMenuItem5 = new javax.swing.JMenuItem();
    jMenu2 = new javax.swing.JMenu();
    jMenuItem2 = new javax.swing.JMenuItem();
    jMenuItem1 = new javax.swing.JMenuItem();
    jMenuItem3 = new javax.swing.JMenuItem();
    jMenuItem4 = new javax.swing.JMenuItem();
    jMenuItem11 = new javax.swing.JMenuItem();

    jFileChooser_Import.setCurrentDirectory(null);

```

```

jFileChooser_Export.setCurrentDirectory(null);
jFileChooser_Export.setDialogType(javax.swing.JFileChooser.SAVE_DIA
LOG);

setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE)
;

setTitle("Simple CPU Simulator");

processorPanel.setBorder(javax.swing.BorderFactory.createTitledBord
er(null, "Processor", javax.swing.border.TitledBorder.CENTER,
javax.swing.border.TitledBorder.TOP));

jLabel1.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
jLabel1.setText("PC");

programCounterTextField.addActionListener(new
java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        programCounterTextFieldActionPerformed(evt);
    }
});

jLabel2.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
jLabel2.setText("IR");

instructionRegisterTextField.setEditable(false);

registersTable.setModel(new javax.swing.table.DefaultTableModel(
    new Object [][] {
        {"R0", null},
        {"R1", null},
        {"R2", null},
        {"R3", null},
        {"R4", null},
        {"R5", null},
        {"R6", null},
        {"R7", null},
        {"R8", null},
        {"R9", null},
        {"R10", null},
        {"R11", null},
        {"R12", null},
        {"R13", null},
        {"R14", null},
        {"R15", null}
    },
    new String [] {
        "Ri", "Value"
    }
));
registersTable.setAutoResizeMode(javax.swing.JTable.AUTO_RESIZE_ALL
_COLUMNS);
registersTable.setSelectionMode(javax.swing.ListSelectionModel.SING
LE_SELECTION);
registersScrollPane.setViewportViewView(registersTable);

javax.swing.GroupLayout processorPanelLayout = new
javax.swing.GroupLayout(processorPanel);
processorPanel.setLayout(processorPanelLayout);
processorPanelLayout.setHorizontalGroup(

```



```

        processorPanelLayout.createParallelGroup(javax.swing.GroupLayout.
t.Alignment.LEADING)
            .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
processorPanelLayout.createSequentialGroup())
            .addGroup(processorPanelLayout.createParallelGroup(javax.sw
ing.GroupLayout.Alignment.LEADING, false)
                .addComponent(jLabel2,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE)
                .addComponent(jLabel1,
javax.swing.GroupLayout.DEFAULT_SIZE, 42, Short.MAX_VALUE))
            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement
.RELATED)
            .addGroup(processorPanelLayout.createParallelGroup(javax.sw
ing.GroupLayout.Alignment.TRAILING, false)
                .addComponent(programCounterTextField)
                .addComponent(instructionRegisterTextField,
javax.swing.GroupLayout.DEFAULT_SIZE, 96, Short.MAX_VALUE))
            .addComponent(registersScrollPane,
javax.swing.GroupLayout.DEFAULT_SIZE, 150, Short.MAX_VALUE)
        );
        processorPanelLayout.setVerticalGroup(
processorPanelLayout.createParallelGroup(javax.swing.GroupLayout.
t.Alignment.LEADING)
            .addGroup(processorPanelLayout.createSequentialGroup())
            .addGroup(processorPanelLayout.createParallelGroup(javax.sw
ing.GroupLayout.Alignment.BASELINE)
                .addComponent(programCounterTextField,
javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
                .addComponent(jLabel1))
            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement
.RELATED)
            .addGroup(processorPanelLayout.createParallelGroup(javax.sw
ing.GroupLayout.Alignment.BASELINE)
                .addComponent(instructionRegisterTextField,
javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
                .addComponent(jLabel2))
            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement
.RELATED)
            .addComponent(registersScrollPane,
javax.swing.GroupLayout.DEFAULT_SIZE, 274, Short.MAX_VALUE))
        );

        memoryPanel.setBorder(javax.swing.BorderFactory.createTitledBorder(
null, "Memory", javax.swing.border.TitledBorder.CENTER,
javax.swing.border.TitledBorder.TOP));

        memoryTable.setModel(new javax.swing.table.DefaultTableModel(
            new Object [][] {

                },
            new String [] {
                "Addr", "Value"
            }
        ));
        memoryTable.setAutoResizeMode(javax.swing.JTable.AUTO_RESIZE_ALL_CO
LUMNS);

```



```

        memoryTable.setSelectionMode(javax.swing.ListSelectionModel.SINGLE_
SELECTION);
        memoryScrollPane.setViewportViewView(memoryTable);

        javax.swing.GroupLayout memoryPanelLayout = new
javax.swing.GroupLayout(memoryPanel);
        memoryPanel.setLayout(memoryPanelLayout);
        memoryPanelLayout.setHorizontalGroup(
            memoryPanelLayout.createParallelGroup(javax.swing.GroupLayout.A
lignment.LEADING)
                .addComponent(memoryScrollPane,
javax.swing.GroupLayout.DEFAULT_SIZE, 151, Short.MAX_VALUE)
            );
        memoryPanelLayout.setVerticalGroup(
            memoryPanelLayout.createParallelGroup(javax.swing.GroupLayout.A
lignment.LEADING)
                .addComponent(memoryScrollPane,
javax.swing.GroupLayout.DEFAULT_SIZE, 324, Short.MAX_VALUE)
            );

        instructionsPanel.setBorder(javax.swing.BorderFactory.createTitledB
order(null, "Instruction editor", javax.swing.border.TitledBorder.CENTER,
javax.swing.border.TitledBorder.TOP));

        instructionsTable.setModel(new javax.swing.table.DefaultTableModel(
            new Object [][] {

                },
            new String [] {
                "Addr", "Data", "Label", "Instruction", "Good"
            }
        ) {
            Class[] types = new Class [] {
                java.lang.Object.class, java.lang.Object.class,
java.lang.Object.class, java.lang.Object.class, java.lang.Boolean.class
            };
            boolean[] canEdit = new boolean [] {
                false, true, true, true, false
            };

            public Class getColumnClass(int columnIndex) {
                return types [columnIndex];
            }

            public boolean isCellEditable(int rowIndex, int columnIndex) {
                return canEdit [columnIndex];
            }
        });
        instructionsTable.setAutoResizeMode(javax.swing.JTable.AUTO_RESIZE_
ALL_COLUMNS);
        instructionsTable.setSelectionMode(javax.swing.ListSelectionModel.S
INGLE_SELECTION);
        instructionsScrollPane.setViewportViewView(instructionsTable);

        javax.swing.GroupLayout instructionsPanelLayout = new
javax.swing.GroupLayout(instructionsPanel);
        instructionsPanel.setLayout(instructionsPanelLayout);
        instructionsPanelLayout.setHorizontalGroup(
            instructionsPanelLayout.createParallelGroup(javax.swing.GroupLa
yout.Alignment.LEADING)

```

```

        .addComponent(instructionsScrollPane,
javax.swing.GroupLayout.DEFAULT_SIZE, 316, Short.MAX_VALUE)
    );
    instructionsPanelLayout.setVerticalGroup(
        instructionsPanelLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
        .addComponent(instructionsScrollPane,
javax.swing.GroupLayout.DEFAULT_SIZE, 324, Short.MAX_VALUE)
    );

    ioPanel.setBorder(javax.swing.BorderFactory.createTitledBorder(null
, "Input/Output device emulation", javax.swing.border.TitledBorder.CENTER,
javax.swing.border.TitledBorder.TOP));

    terminalTextArea.setColumns(20);
    terminalTextArea.setEditable(false);
    terminalTextArea.setRows(5);
    terminalScrollPane.setViewportView(terminalTextArea);

    jLabel3.setText("Data:");

    inputTextField.addActionListener(new
java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            inputTextFieldActionPerformed(evt);
        }
    });

    inputLockCheckBox.setSelected(true);
    inputLockCheckBox.setText("Read lock");
    inputLockCheckBox.addActionListener(new
java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            inputLockCheckBoxActionPerformed(evt);
        }
    });

    javax.swing.GroupLayout ioPanelLayout = new
javax.swing.GroupLayout(ioPanel);
    ioPanel.setLayout(ioPanelLayout);
    ioPanelLayout.setHorizontalGroup(
        ioPanelLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(ioPanelLayout.createSequentialGroup()
                .addComponent(jLabel3,
javax.swing.GroupLayout.PREFERRED_SIZE, 44,
javax.swing.GroupLayout.PREFERRED_SIZE)
                .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                .addComponent(inputTextField,
javax.swing.GroupLayout.DEFAULT_SIZE, 202, Short.MAX_VALUE)
                .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
                .addComponent(inputLockCheckBox,
javax.swing.GroupLayout.PREFERRED_SIZE, 92,
javax.swing.GroupLayout.PREFERRED_SIZE)
                .addGap(0, 0, Short.MAX_VALUE)
                .addComponent(terminalScrollPane,
javax.swing.GroupLayout.Alignment.TRAILING,
javax.swing.GroupLayout.DEFAULT_SIZE, 366, Short.MAX_VALUE)
            );

```



```

        ioPanelLayout.setVerticalGroup(
            ioPanelLayout.createParallelGroup(javax.swing.GroupLayout.Align
ment.LEADING)
                .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
ioPanelLayout.createSequentialGroup()
                    .addComponent(terminalScrollPane,
javax.swing.GroupLayout.DEFAULT_SIZE, 199, Short.MAX_VALUE)
                    .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement
.RELATED)
                .addGroup(ioPanelLayout.createParallelGroup(javax.swing.Gro
upLayout.Alignment.BASELINE)
                    .addComponent(jLabel3)
                    .addComponent(inputTextField,
javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
                    .addComponent(inputLockCheckBox)))
        );

        breakpointsWatchesPanel.setBorder(javax.swing.BorderFactory.createT
itledBorder(null, "Breakpoints&Watches",
javax.swing.border.TitledBorder.CENTER,
javax.swing.border.TitledBorder.TOP));

        breakpointsWatchesTable.setModel(new
javax.swing.table.DefaultTableModel(
            new Object [][] {

                },
            new String [] {
                "Address", "Value", "Type", "Target", "Enabled"
            }
        ) {
            Class[] types = new Class [] {
                java.lang.Object.class, java.lang.Object.class,
java.lang.Object.class, java.lang.Object.class, java.lang.Boolean.class
            };
            boolean[] canEdit = new boolean [] {
                false, false, false, false, true
            };

            public Class getColumnClass(int columnIndex) {
                return types [columnIndex];
            }

            public boolean isCellEditable(int rowIndex, int columnIndex) {
                return canEdit [columnIndex];
            }
        });
        breakpointsWatchesTable.setSelectionMode(javax.swing.ListSelectionM
odel.SINGLE_SELECTION);
        breakpointsWatchesScrollPane.setViewportViewView(breakpointsWatchesTabl
e);

        addBreakpointButton.setText("Add");
        addBreakpointButton.addActionListener(new
java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                addBreakpointButtonActionPerformed(evt);
            }
        });
    });

```



```

        removeBreakpointButton.setText("Remove");
        removeBreakpointButton.addActionListener(new
java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                removeBreakpointButtonActionPerformed(evt);
            }
        });

        breakpointTypeComboBox.setModel(new
javafx.swing.DefaultComboBoxModel(new String[] { "Read", "Write",
"Change" }));

        breakpointMemComboBox.setModel(new
javafx.swing.DefaultComboBoxModel(new String[] { "Reg", "Mem", "Code" }));

        jLabel4.setText("Addr:");

        javafx.swing.GroupLayout breakpointsWatchesPanelLayout = new
javafx.swing.GroupLayout(breakpointsWatchesPanel);
        breakpointsWatchesPanel.setLayout(breakpointsWatchesPanelLayout);
        breakpointsWatchesPanelLayout.setHorizontalGroup(
            breakpointsWatchesPanelLayout.createParallelGroup(javafx.swing.G
roupLayout.Alignment.LEADING)
                .addGroup(breakpointsWatchesPanelLayout.createSequentialGroup())
                .addComponent(addBreakpointButton,
javafx.swing.GroupLayout.PREFERRED_SIZE, 119,
javafx.swing.GroupLayout.PREFERRED_SIZE)
                .addPreferredGap(javafx.swing.LayoutStyle.ComponentPlacement
.RELATED)
                .addComponent(removeBreakpointButton,
javafx.swing.GroupLayout.DEFAULT_SIZE, 145, Short.MAX_VALUE))
                .addGroup(breakpointsWatchesPanelLayout.createSequentialGroup())
                .addComponent(breakpointTypeComboBox,
javafx.swing.GroupLayout.PREFERRED_SIZE, 75,
javafx.swing.GroupLayout.PREFERRED_SIZE)
                .addPreferredGap(javafx.swing.LayoutStyle.ComponentPlacement
.RELATED)
                .addComponent(breakpointMemComboBox, 0, 69,
Short.MAX_VALUE)
                .addPreferredGap(javafx.swing.LayoutStyle.ComponentPlacement
.RELATED)
                .addComponent(jLabel4)
                .addPreferredGap(javafx.swing.LayoutStyle.ComponentPlacement
.RELATED)
                .addComponent(breakpointAddressTextField,
javafx.swing.GroupLayout.PREFERRED_SIZE, 63,
javafx.swing.GroupLayout.PREFERRED_SIZE))
                .addComponent(breakpointsWatchesScrollPane,
javafx.swing.GroupLayout.DEFAULT_SIZE, 270, Short.MAX_VALUE)
        );
        breakpointsWatchesPanelLayout.setVerticalGroup(
            breakpointsWatchesPanelLayout.createParallelGroup(javafx.swing.G
roupLayout.Alignment.LEADING)
                .addGroup(javafx.swing.GroupLayout.Alignment.TRAILING,
breakpointsWatchesPanelLayout.createSequentialGroup())
                .addComponent(breakpointsWatchesScrollPane,
javafx.swing.GroupLayout.PREFERRED_SIZE, 165,
javafx.swing.GroupLayout.PREFERRED_SIZE)
                .addPreferredGap(javafx.swing.LayoutStyle.ComponentPlacement
.RELATED)

```



```

        .addGroup(breakpointsWatchesPanelLayout.createParallelGroup
(javax.swing.GroupLayout.Alignment.BASELINE)
        .addComponent(breakpointTypeComboBox,
javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
        .addComponent(breakpointMemComboBox,
javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
        .addComponent(breakpointAddressTextField,
javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
        .addComponent(jLabel4))
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement
.RELATED, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
        .addGroup(breakpointsWatchesPanelLayout.createParallelGroup
(javax.swing.GroupLayout.Alignment.BASELINE)
        .addComponent(addBreakpointButton)
        .addComponent(removeBreakpointButton)))
);

jMenu1.setText("File");

jMenuItem10.setAccelerator(javax.swing.KeyStroke.getKeyStroke(java.
awt.event.KeyEvent.VK_N, java.awt.event.InputEvent.CTRL_MASK));
jMenuItem10.setText("New");
jMenuItem10.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jMenuItem10ActionPerformed(evt);
    }
});
jMenu1.add(jMenuItem10);
jMenu1.add(jSeparator1);

jMenu3.setText("Import");

jMenuItem6.setText("Memory data");
jMenuItem6.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jMenuItem6ActionPerformed(evt);
    }
});
jMenu3.add(jMenuItem6);

jMenuItem7.setText("Asm file");
jMenuItem7.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jMenuItem7ActionPerformed(evt);
    }
});
jMenu3.add(jMenuItem7);

jMenuItem13.setText("Code binary");
jMenuItem13.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jMenuItem13ActionPerformed(evt);
    }
});
jMenu3.add(jMenuItem13);

```

```

jMenu1.add(jMenu3);

jMenu4.setText("Export");

jMenuItem8.setText("Memory data");
jMenuItem8.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jMenuItem8ActionPerformed(evt);
    }
});
jMenu4.add(jMenuItem8);

jMenuItem9.setText("Asm file");
jMenuItem9.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jMenuItem9ActionPerformed(evt);
    }
});
jMenu4.add(jMenuItem9);

jMenuItem12.setText("Code binary");
jMenuItem12.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jMenuItem12ActionPerformed(evt);
    }
});
jMenu4.add(jMenuItem12);

jMenu1.add(jMenu4);

jMenuItem14.setText("Edit asm source");
jMenuItem14.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jMenuItem14ActionPerformed(evt);
    }
});
jMenu1.add(jMenuItem14);
jMenu1.add(jSeparator2);

jMenuItem5.setAccelerator(javax.swing.KeyStroke.getKeyStroke(java.a
wt.event.KeyEvent.VK_F4, java.awt.event.InputEvent.ALT_MASK));
jMenuItem5.setText("Exit");
jMenuItem5.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jMenuItem5ActionPerformed(evt);
    }
});
jMenu1.add(jMenuItem5);

jMenuBar1.add(jMenu1);

jMenu2.setText("Run");

jMenuItem2.setAccelerator(javax.swing.KeyStroke.getKeyStroke(java.a
wt.event.KeyEvent.VK_R, java.awt.event.InputEvent.CTRL_MASK));
jMenuItem2.setText("Reset");
jMenuItem2.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jMenuItem2ActionPerformed(evt);
    }
}

```

```

    });
    jMenuItem2.add(jMenuItem2);

    jMenuItem1.setAccelerator(javax.swing.KeyStroke.getKeyStroke(java.a
wt.event.KeyEvent.VK_F7, 0));
    jMenuItem1.setText("Step");
    jMenuItem1.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            jMenuItem1ActionPerformed(evt);
        }
    });
    jMenuItem2.add(jMenuItem1);

    jMenuItem3.setAccelerator(javax.swing.KeyStroke.getKeyStroke(java.a
wt.event.KeyEvent.VK_F5, 0));
    jMenuItem3.setText("Run");
    jMenuItem3.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            jMenuItem3ActionPerformed(evt);
        }
    });
    jMenuItem2.add(jMenuItem3);

    jMenuItem4.setAccelerator(javax.swing.KeyStroke.getKeyStroke(java.a
wt.event.KeyEvent.VK_F4, 0));
    jMenuItem4.setText("Run to cursor");
    jMenuItem4.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            jMenuItem4ActionPerformed(evt);
        }
    });
    jMenuItem2.add(jMenuItem4);

    jMenuItem11.setAccelerator(javax.swing.KeyStroke.getKeyStroke(java.
awt.event.KeyEvent.VK_ESCAPE, 0));
    jMenuItem11.setText("Stop");
    jMenuItem11.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            jMenuItem11ActionPerformed(evt);
        }
    });
    jMenuItem2.add(jMenuItem11);

    jMenuItemBar1.add(jMenuItem2);

    setJMenuBar(jMenuBar1);

    javax.swing.GroupLayout layout = new
javax.swing.GroupLayout(getContentPane());
    getContentPane().setLayout(layout);
    layout.setHorizontalGroup(
        layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LE
ADING)
            .addGroup(layout.createSequentialGroup()
                .addGap(3, 3, 3)
                .addComponent(processorPanel,
javax.swing.GroupLayout.PREFERRED_SIZE,

```



```

javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
        .addPreferredGap(javax.swing.LayoutStyle.ComponentP
lacement.RELATED)
        .addComponent(memoryPanel,
javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
        .addPreferredGap(javax.swing.LayoutStyle.ComponentP
lacement.RELATED)
        .addComponent(instructionsPanel,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE))
        .addGroup(layout.createSequentialGroup())
        .addComponent(ioPanel,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE)
        .addPreferredGap(javax.swing.LayoutStyle.ComponentP
lacement.RELATED)
        .addComponent(breakpointsWatchesPanel,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE)))
        .addContainerGap())
    );
    layout.setVerticalGroup(
        layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LE
ADING)
        .addGroup(layout.createSequentialGroup())
        .addContainerGap()
        .addGroup(layout.createParallelGroup(javax.swing.GroupLayouto
ut.Alignment.LEADING, false)
        .addComponent(instructionsPanel,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE)
        .addComponent(memoryPanel,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE)
        .addComponent(processorPanel,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE))
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement
.RELATED)
        .addGroup(layout.createParallelGroup(javax.swing.GroupLayouto
ut.Alignment.LEADING)
        .addComponent(ioPanel,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE)
        .addComponent(breakpointsWatchesPanel,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE))
        .addContainerGap())
    );

    pack();
} // </editor-fold>

private void
inputLockCheckBoxActionPerformed(java.awt.event.ActionEvent evt) {
    updateInputLock();
}

```

```

    private void inputTextFieldActionPerformed(java.awt.event.ActionEvent
    evt) {
        dataInputed(inputTextField.getText());
    }

    private void jMenuItem2ActionPerformed(java.awt.event.ActionEvent evt)
    {
        doReset();
    }

    private void jMenuItem1ActionPerformed(java.awt.event.ActionEvent evt)
    {
        doStep();
    }

    private void
    programCounterTextFieldActionPerformed(java.awt.event.ActionEvent evt) {
        doChangePC();
    }

    private void jMenuItem10ActionPerformed(java.awt.event.ActionEvent evt)
    {
        doNew();
    }

    private void jMenuItem5ActionPerformed(java.awt.event.ActionEvent evt)
    {
        setVisible(false);
        dispose();
        System.exit(0);
    }

    private void jMenuItem3ActionPerformed(java.awt.event.ActionEvent evt)
    {
        doRun();
    }

    private void jMenuItem4ActionPerformed(java.awt.event.ActionEvent evt)
    {
        doRunToCursor();
    }

    private void jMenuItem11ActionPerformed(java.awt.event.ActionEvent evt)
    {
        doStop();
    }

    private void jMenuItem6ActionPerformed(java.awt.event.ActionEvent evt)
    {
        doImportData();
    }

    private void jMenuItem7ActionPerformed(java.awt.event.ActionEvent evt)
    {
        doImportAsmCode();
    }

    private void jMenuItem8ActionPerformed(java.awt.event.ActionEvent evt)
    {
        doExportData();
    }

```

```

    private void jMenuItem9ActionPerformed(java.awt.event.ActionEvent evt)
    {
        doExportAsmCode();
    }

    private void jMenuItem14ActionPerformed(java.awt.event.ActionEvent evt)
    {
        doEditAsmSource();
    }

    private void jMenuItem13ActionPerformed(java.awt.event.ActionEvent evt)
    {
        doImportBinaryCode();
    }

    private void jMenuItem12ActionPerformed(java.awt.event.ActionEvent evt)
    {
        doExportBinaryCode();
    }

    private void
removeBreakpointButtonActionPerformed(java.awt.event.ActionEvent evt) {
        doRemoveBreakpoint();
    }

    private void
addBreakpointButtonActionPerformed(java.awt.event.ActionEvent evt) {
        doAddBreakpoint();
    }

    /**
     * Main method which will create form and show it. Also it applies
     Nimbus skin.
     * @param args the command line arguments, it will be ignored
     */
    public static void main(String args[]) {
        try {
            for (javax.swing.UIManager.LookAndFeelInfo info :
javax.swing.UIManager.getInstalledLookAndFeels()) {
                if ("Nimbus".equals(info.getName())) {
                    javax.swing.UIManager.setLookAndFeel(info.getClassName(
));
                    break;
                }
            }
        } catch (ClassNotFoundException ex) {
            java.util.logging.Logger.getLogger(MainFrame.class.getName()).l
og(java.util.logging.Level.SEVERE, null, ex);
        } catch (InstantiationException ex) {
            java.util.logging.Logger.getLogger(MainFrame.class.getName()).l
og(java.util.logging.Level.SEVERE, null, ex);
        } catch (IllegalAccessException ex) {
            java.util.logging.Logger.getLogger(MainFrame.class.getName()).l
og(java.util.logging.Level.SEVERE, null, ex);
        } catch (javax.swing.UnsupportedLookAndFeelException ex) {
            java.util.logging.Logger.getLogger(MainFrame.class.getName()).l
og(java.util.logging.Level.SEVERE, null, ex);
        }

        java.awt.EventQueue.invokeLater(new Runnable() {

```



```

        public void run() {
            new MainFrame().setVisible(true);
        }
    });
}
// Variables declaration - do not modify
private javax.swing.JButton addBreakpointButton;
private javax.swing.JTextField breakpointAddressTextField;
private javax.swing.JComboBox breakpointMemComboBox;
private javax.swing.JComboBox breakpointTypeComboBox;
private javax.swing.JPanel breakpointsWatchesPanel;
private javax.swing.JScrollPane breakpointsWatchesScrollPane;
private javax.swing.JTable breakpointsWatchesTable;
private javax.swing.JCheckBox inputLockCheckBox;
private javax.swing.JTextField inputTextField;
private javax.swing.JTextField instructionRegisterTextField;
private javax.swing.JPanel instructionsPanel;
private javax.swing.JScrollPane instructionsScrollPane;
private javax.swing.JTable instructionsTable;
private javax.swing.JPanel ioPanel;
private javax.swing.JFileChooser jFileChooser_Export;
private javax.swing.JFileChooser jFileChooser_Import;
private javax.swing.JLabel jLabel1;
private javax.swing.JLabel jLabel2;
private javax.swing.JLabel jLabel3;
private javax.swing.JLabel jLabel4;
private javax.swing.JMenu jMenu1;
private javax.swing.JMenu jMenu2;
private javax.swing.JMenu jMenu3;
private javax.swing.JMenu jMenu4;
private javax.swing.JMenuBar jMenuBar1;
private javax.swing.JMenuItem jMenuItem1;
private javax.swing.JMenuItem jMenuItem10;
private javax.swing.JMenuItem jMenuItem11;
private javax.swing.JMenuItem jMenuItem12;
private javax.swing.JMenuItem jMenuItem13;
private javax.swing.JMenuItem jMenuItem14;
private javax.swing.JMenuItem jMenuItem2;
private javax.swing.JMenuItem jMenuItem3;
private javax.swing.JMenuItem jMenuItem4;
private javax.swing.JMenuItem jMenuItem5;
private javax.swing.JMenuItem jMenuItem6;
private javax.swing.JMenuItem jMenuItem7;
private javax.swing.JMenuItem jMenuItem8;
private javax.swing.JMenuItem jMenuItem9;
private javax.swing.JPopupMenu.Separator jSeparator1;
private javax.swing.JPopupMenu.Separator jSeparator2;
private javax.swing.JPanel memoryPanel;
private javax.swing.JScrollPane memoryScrollPane;
private javax.swing.JTable memoryTable;
private javax.swing.JPanel processorPanel;
private javax.swing.JTextField programCounterTextField;
private javax.swing.JScrollPane registersScrollPane;
private javax.swing.JTable registersTable;
private javax.swing.JButton removeBreakpointButton;
private javax.swing.JScrollPane terminalScrollPane;
private javax.swing.JTextArea terminalTextArea;
// End of variables declaration
}

```

Breakpoint16.java

```
package cpusimulator;

import cpusimulator.memory.Memory16;

/**
 * Breakpoint used to handle READ/WRITE operations during CPU simulation.
 * Main implementation for this class is BreakpointMemory16.
 *
 * @author Daniel Alexandre 2011
 */
public abstract class Breakpoint16 {

    /** Breakpoint with this type will be activated by reading */
    public static final int BREAKPOINT_READ = 1;
    /** Breakpoint with this type will be activated by writing */
    public static final int BREAKPOINT_WRITE = 2;
    /** Breakpoint with this type will be activated by changing value */
    public static final int BREAKPOINT_CHANGE = 3;

    /**
     * Returns true if this breakpoint is enabled.
     * If breakpoint isn't enabled, it must behave like there is
     no breakpoint.
     * @return true if this breakpoint is enabled
     */
    public abstract boolean isEnabled();

    /**
     * Enables/disables this breakpoint.
     * If breakpoint isn't enabled, it must behave like there is
     no breakpoint.
     * @param enabled true when we want to enable breakpoint,
     false otherwise
     */
    public abstract void setEnabled(boolean enabled);

    /**
     * Removes breakpoint. After this operation this breakpoint becomes
     an unusable.
     */
    public abstract void remove();

    /**
     * Sets some handler to breakpoint.
     * @param runnable callable object which will be called every
     time
     breakpoint catches some operation
     */
    public abstract void setHandler(Runnable runnable);

    /**
     * Returns breakpoint's attached address.
     * @return address of memory cell, to which breakpoint is attached
     */
    public abstract int getAddress();
```

```

/**
 * Returns type of this breakpoint.
 * @return one of BREAKPOINT_READ, BREAKPOINT_WRITE, BREAKPOINT_CHANGE
 */
public abstract int getType();

/**
 * Returns memory cell value from cell, to which this breakpoint is
attached.
 * @return memory cell value
 */
public abstract short getValue();

/**
 * Returns memory to which this breakpoint is attached.
 * @return memory to which this breakpoint is attached
 */
public abstract Memory16 getMemory();
}

```

CPU16Model.java

```

package cpusimulator;

import cpusimulator.memory.BreakpointMemory16;
import cpusimulator.memory.DataMemory16;
import cpusimulator.memory.Memory16;

/**
 * This is 16-bit CPU model with ALU and attached memory. Have 14
instructions
 * only. This class emulates CPU execution.
 *
 * @author Daniel Alexandre 2011
 */
public class CPU16Model {

    /** Memory with which CPU will work. */
    private Memory16 memory;
    /** Program counter, points to next instruction will executed. Maximum
count of commands - 256, because byte type is used. */
    private byte PC;
    /** Instruction register, contains current executing instruction when
executing or last executed instruction when not executing */
    private short IR;
    /** This object can be called and it's called when CPU executes HALT
command. */
    private Runnable haltHandler;
    /** This object can be called and it's called when CPU faced unknown
instruction. */
    private Runnable wrongInstructionHandler;
    /** Array with 16-bit register values, used to store values. */
    private BreakpointMemory16 REG;

    /**

```

```

    * Creates new 16-bit CPU model with specified memory, initial program
    counter, specified hand handler and wrong instruction handler.
    * @param connectedMemory memory with which CPU will work, shouldn't be
    null
    * @param startPC initial program counter, should be 0
    * @param haltHandler callable object to handle halt command execution,
    shouldn't be null
    * @param wrongInstructionFailedHandler callable object to handle wrong
    instructions, shouldn't be null
    * @param breakpointReadCheck callable object to handle breakpoint
    memory errors
    * @throws NullPointerException if connectedMemory, haltHandler or
    wrongInstructionFailedHandler is null
    */
    public CPU16Model(Memory16 connectedMemory, byte startPC, Runnable
    haltHandler, Runnable wrongInstructionFailedHandler, Runnable
    breakpointReadCheck) {
        if (connectedMemory == null) throw new NullPointerException();
        if (haltHandler == null) throw new NullPointerException();
        if (wrongInstructionFailedHandler == null) throw new
    NullPointerException();
        memory = connectedMemory;
        PC = startPC;
        this.haltHandler = haltHandler;
        wrongInstructionHandler = wrongInstructionFailedHandler;
        Runnable invalidAddress = new Runnable() {
            @Override public void run() { throw new
    RuntimeException("Invalid register address"); }
        };
        REG = new BreakpointMemory16(new DataMemory16(16, invalidAddress),
    16, breakpointReadCheck);
    }

    /**
    * Returns registers memory with breakpoints support.
    * @return memory with breakpoints support
    */
    public BreakpointMemory16 getRegistersBreakpoints() {
        return REG;
    }

    /**
    * Cleans all registers, PC and IR to zero.
    */
    public void reset() {
        for (int i = 0; i < 16; i++) REG.write((byte)i, (short)0);
        PC = 0;
        IR = 0;
    }

    /**
    * Reads instruction from memory and stores it into IR.
    */
    public void fetchInstruction() {
        IR = memory.read(PC);
    }

    /**
    * Emulates just one instruction and exits from the method. Can call
    wrongInstructionHandler on wrong instruction appearance.
    */

```



```

    public void emulateInstruction() {
        PC ++;
        int instruction = IR & 0xFFFF;
        switch(instruction >>> 12) { // decoding: gets leftest 4 bits (in
other words, left digit in command hex representation)
            case 0: // HALT
                haltHandler.run();
                break;
            case 1: // LOAD <reg>, <mem> REG.write((byte)
                ((instruction >>> 8) & 0xf),
memory.read((byte)instruction));
                break;
            case 2: // STORE <mem>, <reg>
                memory.write((byte)(instruction >>> 4), REG.read((byte)
(instruction & 0xf)));
                break;
            case 3: // ADDI <reg>, <reg>, <reg> REG.write((byte)
                ((instruction >>> 8) & 0xf), (short)
(REG.read((byte)((instruction >>> 4) & 0xf)) + REG.read((byte)(instruction
& 0xf))));
                break;
            case 4: // ADDF <reg>, <reg>, <reg>
                // not yet supported
                wrongInstructionHandler.run();
                break;
            case 5: // MOVE <reg>, <reg>
                REG.write((byte)((instruction >>> 8) & 0xf),
REG.read((byte)((instruction >>> 4) & 0xf)));
                break;
            case 6: // NOT <reg>, <reg>
                REG.write((byte)((instruction >>> 8) & 0xf), (short)
(~REG.read((byte)((instruction >>> 4) & 0xf))));
                break;
            case 7: // AND <reg>, <reg>, <reg>
                REG.write((byte)((instruction >>> 8) & 0xf), (short)
(REG.read((byte)((instruction >>> 4) & 0xf)) & REG.read((byte)(instruction
& 0xf))));
                break;
            case 8: // OR <reg>, <reg>, <reg> REG.write((byte)
                ((instruction >>> 8) & 0xf), (short)
(REG.read((byte)((instruction >>> 4) & 0xf)) | REG.read((byte)(instruction
& 0xf))));
                break;
            case 9: // XOR <reg>, <reg>, <reg>
                REG.write((byte)((instruction >>> 8) & 0xf), (short)
(REG.read((byte)((instruction >>> 4) & 0xf)) ^ REG.read((byte)(instruction
& 0xf))));
                break;
            case 10: // INC <reg>
                REG.write((byte)((instruction >>> 8) & 0xf), (short)
(REG.read((byte)((instruction >>> 8) & 0xf)) + 1));
                break;
            case 11: // DEC <reg>
                REG.write((byte)((instruction >>> 8) & 0xf), (short)
(REG.read((byte)((instruction >>> 8) & 0xf)) - 1));
                break;
            case 12: // ROTATE <reg>, <count>, <count>
                int value = REG.read((byte)((instruction >>> 8) & 0xf)) &
0xffff;

```

```

        if((instruction & 0xf) == 0) REG.write((byte)((instruction
>>> 8) & 0xf), (short)((value << ((instruction >>> 4) & 0xf)) | (value >>>
(16 - ((instruction >>> 4) & 0xf))));
        else if((instruction & 0xf) == 1) REG.write((byte)
((instruction >>> 8) & 0xf), (short)((value >>> ((instruction >>> 4) &
0xf)) | (value << (16 - ((instruction >>> 4) & 0xf))));
        else wrongInstructionHandler.run();
        break;
    case 13: // AND <reg>, <reg>, <reg>
        if(REG.read((byte)0) != REG.read((byte)((instruction >>> 8)
& 0xf))) PC = (byte)instruction;
        break;
    }
}

/**
 * Returns value of specified register.
 * @param reg register number, should be from 0 to 15 inclusive
 * @return containing data
 * @throws IllegalArgumentException if reg is out of 0..15 range
 */
public short getRegValue(int reg) {
    if(reg < 0 || reg >= 16) throw new IllegalArgumentException();
    return REG.read((byte)reg);
}

/**
 * Sets value of specified register.
 * @param reg register number, should be from 0 to 15 inclusive
 * @param newValue data to write to register
 * @throws IllegalArgumentException if reg is out of 0..15 range
 */
public void setRegValue(int reg, short newValue) {
    if(reg < 0 || reg >= 16) throw new IllegalArgumentException();
    REG.write((byte)reg, newValue);
}

/**
 * Returns program counter index: memory address which points to cell
with instruction which
 * will be executed on next emulateInstruction() method call.
 * @return memory address
 */
public byte getPCValue() {
    return PC;
}

/**
 * Sets program counter address to specified value
 * @param newPC new program counter address
 */
public void setPCValue(byte newPC) {
    PC = newPC;
}

/**
 * Returns currently loaded instruction in the instruction register
(IR).
 * @return currently executing/executed instruction
 */
public short getIRValue() {

```



```
        return IR;  
    }  
}
```

Device.java

```
package cpusimulator;

import cpusimulator.memory.Memory16;

/**
 * This class represents device. Device has it own memory through which I/O
 * operations between CPU and device is performed.
 *
 * * @author Daniel Alexandre 2011
 */
public interface Device {

    /**
     * Returns count of memory cells through which I/O operations should be
     * performed.
     * * @return count of memory cells
     */
    public int getMappingLength();

    /**
     * Returns device's memory. Can be readable and/or writable according
to
     * device specification.
     * * @return 16-bit memory
     */
    public Memory16 getMappingMemory();
}
```

AssemblerException.java

```
package cpusimulator.code;

/**
 * This exception should be thrown in some mistaken cases during assembling
 * ASM code to processor code.
 *
 * * @author Daniel Alexandre 2011
 */
public class AssembleException extends Exception {

    /**
     * Creates new assemble exception without details and enclosed
exception.
     */
    public AssembleException() {
        super();
    }

    /**
     * Creates new assemble exception with details but without enclosed
exception.
     * * @param message message about exception
     */
    public AssembleException(String message) {
```

```

        super(message);
    }

    /**
     * Creates new assemble exception wit details and enclosed exception.
     * @param message message about exception
     * @param cause exception by which this exception is caused
     */
    public AssembleException(String message, Throwable cause) {
        super(message, cause);
    }
}

```

CodeManager.java

```

package cpusimulator.code;

import cpusimulator.memory.InstructionMemory16;
import java.util.HashMap;
import java.util.StringTokenizer;

/**
 * This class used to manage machine code in memory.
 * There is assembling, disassembling methods and address-to-label and
 * label-to-
 * address conversion capabilities.
 *
 * @author Daniel Alexandre 2011
 */
public class CodeManager {

    /** Memory used to write and read instructions. */
    private InstructionMemory16 memory;
    /** Labels offsets by it's names. */
    private HashMap<String, Integer> labelNames = new HashMap<String,
Integer>();
    /** Labels names by it's offsets. */
    private HashMap<Integer, String> labelOffsets = new HashMap<Integer,
String>();
    /** Current instruction offset. Indicates cell with next instruction to
disassemble or
     * cell to which next assembled instruction will be written. */
    private byte offset;

    /**
     * Creates a new code manager which will do management on specified
memory
     * and with specified address.
     * @param memory instruction memory which will be modified by this
manager
     * @param offset current instruction offset, can be changed
     * @throws NullPointerException if memory is null
     */
    public CodeManager(InstructionMemory16 memory, byte offset) {
        if(memory == null) throw new NullPointerException();
        this.memory = memory;
        this.offset = offset;
    }

    /**

```

```

        * Adds label to label table if it isn't exists or causes exception
        otherwise.
        * @param label label name
        * @param offset label code address
        * @throws IllegalStateException if label with the same name OR offset
        is already available
        */
        public synchronized void addLabel(String label, int offset) {
            if(labelNames.containsKey(label)) throw new
IllegalArgumentExcepti
on("Already have that label");
            Integer off = new Integer(offset);
            if(labelOffsets.containsKey(off)) throw new
IllegalArgumentExcepti
on("Already have that label");
            labelNames.put(label, off);
            labelOffsets.put(off, label);
        }

        /**
        * Removes label by it's name or does nothing if table isn't contains
        label with specified name.
        * @param label name of label which have to be removed
        */
        public synchronized void removeLabel(String label) {
            Integer offset = labelNames.get(label);
            labelNames.remove(label);
            labelOffsets.remove(offset);
        }

        /**
        * Removes label by it's code offset or does nothing if table isn't
        contains label with specified code offset.
        * @param offset code offset of label which have to be removed
        */
        public synchronized void removeLabel(int offset) {
            Integer off = new Integer(offset);
            String name = labelOffsets.get(off);
            labelNames.remove(name);
            labelOffsets.remove(off);
        }

        /**
        * Does instruction assembling. Throws exception on fail or write
        instruction
        * code to memory on success. After writing offset will be moved to the
        next
        * memory cell (it will be incremented by 1).
        * @param instruction instruction's text representation
        * @throws AssembleException if there is errors in instruction's text
        representation
        */
        public synchronized void assembleInstruction(String instruction) throws
AssembleException {
            instruction = instruction.trim(); // skips leading and trailing
spaces
            // try to cut out instruction's name
            StringTokenizer tokenizer = new StringTokenizer(instruction, "
\t");
            if(!tokenizer.hasMoreTokens()) throw new
AssembleException("Instruction name expected");
            String insName = tokenizer.nextToken().trim().toLowerCase();

```



```

        // yes, we have it. now try to parse operands, separated by comma.
we need new tokenizer now
        tokenizer = new
StringTokenizer(instruction.substring(insName.length()).trim(), ",");
        Operand[] ops = new Operand[tokenizer.countTokens()];
        for(int i = 0; i < ops.length; i ++) {
            // try to parse operand
            ops[i] = Operand.parse(tokenizer.nextToken().trim());
            if(ops[i].getType() == Operand.Type.LABEL) {
                // if operand's type is address, try to replace label's
name by it's address
                Integer off = labelNames.get(ops[i].getString());
                if(off != null) ops[i].replaceLabel(off.intValue());
            }
        }
        // ok, we have correctly parsed operands. let write instruction's
code to memory
        synchronized(memory) {
            memory.unlock(); // this allows to write instructions to code
memory
            boolean thr = false; // indicates that we have instruction
            try {
                Instruction ins =
Instruction.valueOf(insName.toUpperCase()); // it throws exception if there
is no such instruction with specified name
                thr = true;
                memory.write(offset, ins.getBinary(ops)); // creating
instruction from operands and writing it to memory
            } catch(IllegalArgumentException exception) {
                if(thr) throw exception;
                throw new AssembleException("No such instruction");
            } finally {
                memory.lock(); // in any case (success or fail), we have to
lock instruction memory to avoid code segment corruption
            }
        }
        offset ++; // moves pointer to the next instruction
    }

/**
 * Does instruction disassembling. Returns null if disassembling failed
or
 * instruction's string representation if disassembling was completed
OK.
 * Always increases code address by 1.
 * @return instruction ASM representation
 */
public synchronized String disassembleInstruction() {
    // reading instruction's binary representation
    short opcode = memory.read(offset);
    offset ++;
    // determining instruction number
    int opindex = (opcode >>> 12) & 0xF;
    // checking for it's correctness
    if(opindex >= Instruction.values().length) return null;
    // creating instruction by it's number
    Instruction instr = Instruction.values()[opindex];
    // passing operands' binary data to it
    Operand[] ops = instr.getOperands(opcode);
    // now building instruction's ASM representation
    StringBuilder builder = new StringBuilder();

```



```

        builder.append(instr.getName()); // adding name to stringbuffer
        int index = 0;
        // ... and operands with commas and spaces in needed positions
        for(Operand op : ops) builder.append((index++) == 0 ? " " : ",
").append(op.getAsm(labelOffsets));
        // then convert stringbuffer to string and return disassembled
result
        return builder.toString().trim();
    }

    /**
     * Changes memory address to new one.
     * @param newOffset new memory address to read and write instructions
     */
    public synchronized void gotoOffset(byte newOffset) {
        offset = newOffset;
    }

    /**
     * Returns memory address (instruction reading source / writing
destination).
     * @return memory address
     */
    public synchronized int getOffset() {
        return offset;
    }
}

```

Instruction.java

```

package cpusimulator.code;

import java.util.ArrayList;

/**
 * Instruction enum. Represents possible instructions in this CPU model.
 * Used in assembling/disassembling, not in code execution.
 *
 * @author Daniel Alexandre 2011
 */
public class Instruction {

    public static ArrayList<Instruction> instances = new
ArrayList<Instruction>();

    // HERE IS: name of instruction and operand types needed for it
    // See Operand class and Operand.Type enum to get more description
about operands and their types
    public static final Instruction HALT = new Instruction(0, "HALT");
    public static final Instruction LOAD = new Instruction(1, "LOAD",
Operand.Type.REGISTER, Operand.Type.MEMADDR);
    public static final Instruction STORE = new Instruction(2, "STORE",
Operand.Type.MEMADDR, Operand.Type.REGISTER);
    public static final Instruction ADDI = new Instruction(3, "ADDI",
Operand.Type.REGISTER, Operand.Type.REGISTER, Operand.Type.REGISTER);
    public static final Instruction ADDF = new Instruction(4, "ADDF",
Operand.Type.REGISTER, Operand.Type.REGISTER, Operand.Type.REGISTER);
    public static final Instruction MOVE = new Instruction(5, "MOVE",
Operand.Type.REGISTER, Operand.Type.REGISTER);

```



```

    public static final Instruction NOT = new Instruction(6, "NOT",
Operand.Type.REGISTER, Operand.Type.REGISTER);
    public static final Instruction AND = new Instruction(7, "AND",
Operand.Type.REGISTER, Operand.Type.REGISTER, Operand.Type.REGISTER);
    public static final Instruction OR = new Instruction(8, "OR",
Operand.Type.REGISTER, Operand.Type.REGISTER, Operand.Type.REGISTER);
    public static final Instruction XOR = new Instruction(9, "XOR",
Operand.Type.REGISTER, Operand.Type.REGISTER, Operand.Type.REGISTER);
    public static final Instruction INC = new Instruction(10, "INC",
Operand.Type.REGISTER);
    public static final Instruction DEC = new Instruction(11, "DEC",
Operand.Type.REGISTER);
    public static final Instruction ROTATE = new Instruction(12, "ROTATE",
Operand.Type.REGISTER, Operand.Type.NUMBER4, Operand.Type.NUMBER4);
    public static final Instruction JUMP = new Instruction(13, "JUMP",
Operand.Type.REGISTER, Operand.Type.LABEL);

    public static Instruction valueOf(String name) {
        for(Instruction ins : instances) if(ins.getName().equals(name))
return ins;
        throw new IllegalArgumentException();
    }

    public static Instruction[] values() {
        return instances.toArray(new Instruction[instances.size()]);
    }

    private int ordinal;
    /** Instruction name, specified above as first argument */
    private String instructionName;
    /** Instruction types, specified above as second, third and more
argument */
    private Operand.Type[] types;

    /**
     * Creating new instruction with specified name and operand types.
     * Inside enum only because this is an enum, not class.
     * @param name instruction name, will be used in assembler/disassembler
     * @param types instruction's operands types
     */
    private Instruction(int ordinal, String name, Operand.Type... types) {
        this.ordinal = ordinal;
        this.instructionName = name;
        this.types = types;
        instances.add(this);
    }

    /**
     * Returns instruction's name.
     * @return instruction's name
     */
    public String getName() {
        return instructionName;
    }

    /**
     * Creates instruction's binary representation, using specified
operands.
     * Operands' types will be validated according to instruction
specification.
     * @param operands instruction's operands

```

```

    * @return instruction's binary representation
    * @throws AssembleException if operand type is wrong or there is too
much
    * or too few operands.
    * @throws NullPointerException if operands is null
    */
    public short getBinary(Operand[] operands) throws AssembleException {
        if(operands == null) throw new NullPointerException();
        // verify operands count
        if(operands.length != types.length) throw new
AssembleException(getName() + ": " + " " + types.length + " operands
expected, but " + operands.length + " got");
        int bits = 4; // each instruction have at least 4 bits - for it's
indentification.
        int data = ordinal; // instruction identification data, just a
number in 0 to 15 range, inclusive
        for(int i = 0; i < operands.length; i++) {
            // for each operand: check it type
            if(operands[i].getType() != types[i]) throw new
AssembleException(getName() + ": " + " " + getTh(i + 1) + " operand should
have " + types[i].getName() + " type, but it have " +
operands[i].getType().getName() + " type");
            // if operand's data is too large to compose it in 2-byte field
            if((operands[i].getData() & (1 <<
operands[i].getType().getSize()) - 1) != operands[i].getData()) throw new
IllegalArgumentException("wrong operand data size");
            // compose operand data into instruction binary representation
            bits += operands[i].getType().getSize();
            data = (data << operands[i].getType().getSize()) |
(operands[i].getData());
        }
        // if we got instruction which cann't lay in 2 bytes
        if(bits > 16) throw new RuntimeException("too much bits for
instruction");
        // align instruction to 2-byte align, according to 16-bit memory
model
        data <<= 16 - bits;
        // return instruction's representation
        return (short)data;
    }

    /**
     * Generates operands from it's binary representation.
     * @param opcode instruction's binary representation
     * @return disassembled array of operands
     */
    public Operand[] getOperands(short opcode) {
        // remove instruction number
        int iop = (opcode & 0xFFFF) << 4;
        // creating ops array according to instruction's specification
        Operand[] ops = new Operand[types.length];
        for(int i = 0; i < ops.length; i++) {
            // parse operand current
            ops[i] = types[i].createOperand(iop, 16);
            // and shift out instruction's bits because it was used
            iop <<= types[i].getSize();
        }
        // return resulting operand array
        return ops;
    }
}

```

```

/**
 * Just helper method.
 * @param num natural number
 * @return number's counting form
 */
private String getTh(int num) {
    return num == 1 ? "1st" : num == 2 ? "2nd" : num == 3 ? "3rd" : num
+ "th";
}
}

```

Operand.java

```

package cpusimulator.code;

import java.util.Map;

/**
 * This class contains operand type and data. Also there is operand
assembling
 * and checking capabilities.
 *
 * @author Daniel Alexandre 2011
 */
public class Operand {

    /** Specify registers count to make asm code correctly executable on
our CPU model */
    private static final int REGISTERS = 16;
    /** Specify memory cells count to make asm code correctly executable on
our CPU model */
    private static final int MEMCELLS = 256;
    /** Specify code cells count to make asm code correctly executable on
our CPU model */
    private static final int CODECELLS = 64;

    public static class Type {

        /** Operand type : register : R0, R1, R2 */
        public final static Type REGISTER = new Type(4, "register");
        /** Operand type : memory address : [FEh] */
        public final static Type MEMADDR = new Type(8, "address");
        /** Number : just a number: 25, 30h, 0, 1 */
        public final static Type NUMBER4 = new Type(4, "number");
        /** Label : my_l_123, end_code, #FAh */
        public final static Type LABEL = new Type(8, "label");

        /** Count of bits in operand's binary representation */
        private int size;
        /** Operand type's name. */
        private String name;

        /**
         * Creating a new operand type. Can be called only inside of enum.
         * @param size size of operand's binary representation, in bits
         * @param name operand type's name
         */
        private Type(int size, String name) {
            this.size = size;
            this.name = name;
        }
    }
}

```

```

    }

    /**
     * Returns size of operand's representation, int bits.
     * @return operand's bit count
     */
    public int getSize() {
        return size;
    }

    /**
     * Returns name of operand type
     * @return operand type's name
     */
    public String getName() {
        return name;
    }

    /**
     * Creates new operand from the operand's type.
     * @param value binary data from which operand's data will be
retrieved;
     * data will be cutted from the higher part of value's lowest
_bits_ bits
     * @param bits operand's data offset. for example, 8 means lowest
byte's
     * higher bits (or entire lowest byte if operand have 8 bits of
data)
     * @return newly created operand with cutted out operand data.
     */
    public Operand createOperand(int value, int bits) {
        if(bits < size) throw new IllegalArgumentException("too few
bits");
        return new Operand(this, (value >>> (bits - size)) & ((1 <<
size) - 1));
    }

    /** Operand's type */
    private Type type;
    /** Data of opperand: register number / memory cell / flag / code
address */
    private int data;
    /** Data of operand, normally contans label's name. */
    private String string;

    /**
     * Creates a new operand from specified type and bits.
     * @param type type of operand
     * @param data operand's data
     */
    private Operand(Type type, int data) {
        this.type = type;
        this.data = data;
    }

    /**
     * Creates a new operand from specified type and string (usually, type
is LABEL).
     * @param type type of operand
     * @param string string data

```



```

    */
    private Operand(Type type, String string) {
        this.type = type;
        this.string = string;
    }

    /**
     * Replaces label's name by offset. Useful in assembling.
     * @param data label's code address
     * @throws IllegalStateException if label already have no name or if
     operand isn't a label
     */
    public void replaceLabel(int data) {
        if(type != Type.LABEL || string == null) throw new
IllegalStateException("this operation can be done only for labels");
        this.string = null;
        this.data = data;
    }

    /**
     * Replaces label's code address by name. Useful in disassembling.
     * @param string label's name.
     * @throws IllegalStateException if label already have a name or if
     operand isn't a label
     */
    public void replaceLabel(String string) {
        if(type != Type.LABEL || string != null) throw new
IllegalStateException("this operation can be done only for labels");
        this.string = string;
        this.data = 0;
    }

    /**
     * Returns type of this operand.
     * @return operand's type
     */
    public Type getType() {
        return type;
    }

    /**
     * Returns data of this operand. If this operand is label with #ADDR
     address
     * format, method will parse that format and return label's address.
     * @return this operand's data
     */
    public int getData() {
        if(type == Type.LABEL && string != null && string.startsWith("#"))
{
            try {
                return parseInt(string.substring(1));
            } catch (NumberFormatException exception) {
                throw new IllegalStateException("operand with wrong
address");
            }
        }
        return data;
    }

    /**
     * Returns this operand's string.

```



```

    * @return string of ths operand or null if operand have null string
    */
    public String getString() {
        return string;
    }

    /**
     * Returns ASM operand's representation
     * @param labels map with labels which is used to replace label
addresses by names
     * @return disassembled operand
     */
    public String getAsm(Map<Integer, String> labels) {
        if(type == Type.REGISTER) return "R" + data;
        else if(type == Type.MEMADDR) return "[" + data + "]";
        else if(type == Type.NUMBER4) return String.valueOf(data);
        else if(type == Type.LABEL) {
            String name = labels.get(new Integer(data));
            return name == null ? "#" + String.valueOf(data) : name;
        } else throw new RuntimeException("unknown operand type");
    }

    /**
     * Assembles operand from its ASM representation.
     * @param operand operand's ASM representation
     * @return parsed Operand
     * @throws AssembleException if there was errors during operand
assembling
     */
    public static Operand parse(String operand) throws AssembleException {
        // first make all letters low case to make it case-insensitive
        operand = operand.toLowerCase();
        if(operand.startsWith("r") || operand.startsWith("R")) {
            // we have register-type operand
            try {
                int data = Integer.parseInt(operand.substring(1));
                if(data < 0 || data >= REGISTERS) throw new
AssembleException("We have only " + REGISTERS + " registers numerated from
0 to " + (REGISTERS - 1));
                return new Operand(Type.REGISTER, data);
            } catch(NumberFormatException exception) {
                throw new AssembleException("Wrong register number, it
should be a DEC number");
            }
        }
        if(operand.startsWith("[") && operand.endsWith("]")) {
            // we have address-type operand
            try {
                int data = parseInt(operand.substring(1, operand.length() -
1).trim());
                if(data < 0 || data >= MEMCELLS) throw new
AssembleException("We have only " + MEMCELLS + " memory cells numerated
from 0 to " + (MEMCELLS - 1));
                return new Operand(Type.MEMADDR, data);
            } catch(NumberFormatException exception) {
                throw new AssembleException("Wrong memory address, it
should be a HEX/DEC/BIN number");
            }
        }
        if(operand.startsWith("#")) {
            // we have label-type operand

```



```

        try {
            int data = parseInt(operand.substring(1,
operand.length()).trim());
            if(data < 0 || data >= CODECELLS) throw new
AssembleException("We have only " + CODECELLS + " code cells numerated from
0 to " + (CODECELLS - 1));
        } catch(NumberFormatException exception) {
            throw new AssembleException("Wrong label, it should be an
HEX/DEC/BIN number");
        }
    }
    try {
        // operand's type isn't ADDRESS or register. trying to parse.
        // successfull number parse means that operand has number type
        return new Operand(Type.NUMBER4, parseInt(operand));
    } catch(NumberFormatException exception) {
        // there was errors in integer number parsing, deciding to
        // create operand of label type
        return new Operand(Type.LABEL, operand);
    }
}

/**
 * Utility method to allow parse integers with H (hex) and B (bin)
 * suffixes.
 * This allows to use numbers of different radices in ASM code
 * @param string unparsed number
 * @return parsed number
 */
private static int parseInt(String string) {
    // firstly, we try to check and parse hexadecimal number
    if(string.endsWith("h") || string.endsWith("H")) return
Integer.parseInt(string.substring(0, string.length() - 1), 16);
    // then try to check and parse binary
    if(string.endsWith("b") || string.endsWith("B")) return
Integer.parseInt(string.substring(0, string.length() - 1), 2);
    // and decimal if two above lines of code haven't parsed the number
    return Integer.parseInt(string);
}
}

```

Display.java

```

package cpusimulator.devices;

import cpusimulator.Device;
import cpusimulator.memory.Memory16;

/**
 * This is a display device. It have 1 connecting memory cell and will call
 * numberWrited(short) method to deliver number to user. User have to
 * inherit this
 * class to implement numberWrited(short) method to be able to handle
 * number
 * and printout it to real display or something like that. Reading from
 * display
 * always will return 0.
 *
 * @author Daniel Alexandre 2011
 */

```

```

public class Display implements Device, Memory16 {

    /** If true, display will work in a new thread. */
    private boolean newThread;
    /** Callable object which will be called in the wrong device mapping
case. */
    private Runnable invalidAddressHandler;

    /**
     * Creates a new display with specified newThread flag and invalid
address
     * handler.
     * @param newThread if true, method numberWrited(short) will be invoked
     * into different thread than CPU emulation have.
     * @param invalidAddressHandler callable object which will be called on
     * wrong display device mapping and usage.
     */
    public Display(boolean newThread, Runnable invalidAddressHandler) {
        if(invalidAddressHandler == null) throw new NullPointerException();
        this.newThread = newThread;
        this.invalidAddressHandler = invalidAddressHandler;
    }

    @Override
    public int getMappingLength() {
        return 1;
    }

    @Override
    public Memory16 getMappingMemory() {
        return this;
    }

    @Override
    public short read(byte address) {
        return 0;
    }

    @Override
    public void write(byte address, final short value) {
        if(address != 0) {
            invalidAddressHandler.run();
            return;
        }
        if(newThread) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    numberWrited(value);
                }
            }).start();
        } else numberWrited(value);
    }

    /**
     * This method calls every time when display receives a data from
memory bus.
     * User have to implement this method because by default it does
nothing
     * (display is dummy).
     * @param number received data

```

```

        */
        protected void numberWrited(short number) {
    }
}

```

Keyboard.java

```

package cpusimulator.devices;

import cpusimulator.Device;
import cpusimulator.memory.Memory16;

/**
 * Keyboard device, used to handle input events. Have 1 cell in memory bus,
 * writing to which will cause nothing, and reading from it may blocks
 * execution
 * process. Blocking depends on locking flag. Anyway, last inputed number
 * will
 * be sent through memory bus to reader (in our case - CPU).
 *
 * @author Daniel Alexandre 2011
 */
public class Keyboard implements Device, Memory16 {

    /** Object used to notify CPU to continue execution and CPU wait for
    inputed number. */
    private final Object notifyAboutInput = new Object();
    /** Locking flag, see setLocking(boolean) description. */
    private boolean locking;
    /** Last inputed number. It's lesser that zero value means that we have
    no number inputed. */
    private int lastNumber;
    /** Callable object which will be called in some incorrect memory
    mapping cases. */
    private Runnable invalidAddressHandler;

    /**
     * Creates new keyboard with specified invalid address handler.
     * @param invalidAddressHandler callable object which will be called on
     * every incorrect access to keyboard device; it may be called only in
    some
     * bad memory mapping cases
     */
    public Keyboard(Runnable invalidAddressHandler) {
        if(invalidAddressHandler == null) throw new NullPointerException();
        this.invalidAddressHandler = invalidAddressHandler;
    }

    /**
     * Sets the locking flag.
     * @param locking if it's true, keyboard will lock CPU execution
    process
     * until we should have inputed data; if it's false - keyboard will
    just
     * send last inputed number to the bus
     */
    public synchronized void setLocking(boolean locking) {
        this.locking = locking;
        if(locking) lastNumber = -1;
    }
}

```



```

@Override
public int getMappingLength() {
    return 1;
}

@Override
public Memory16 getMappingMemory() {
    return this;
}

@Override
public short read(byte address) {
    if(address != 0) {
        // attempt to read from invalid address, because keyboard have
just one binding cell
        invalidAddressHandler.run();
        return 0;
    }
    boolean locking;
    int lastNumber;
    synchronized(this) {
        // to avoid thread conflict, do reading this class' fields in
synchronized state
        locking = this.locking;
        lastNumber = this.lastNumber;
    }
    if(locking) {
        // if we have locking, we have to wait if we have no number yet
short toReturn;
        if(lastNumber < 0) {
            // wait for inputed number
            try {
                synchronized(notifyAboutInput) {
                    notifyAboutInput.wait();
                }
            } catch(InterruptedException exception) {
                System.out.println("Input interrupted");
            }
        }
        // ok we have inputed number (or input was interrupted)
synchronized(this) {
            // do reading operations in synchronized state to avoid
thread conflicts
            toReturn = (short)this.lastNumber;
            this.lastNumber = -1;
        }
        // return inputed number
        return toReturn;
    } else {
        // we have no locking with all these damned synchronization, so
just return a last iputed number
        synchronized(this) {
            return (short)(lastNumber < 0 ? -32768 : lastNumber);
        }
    }
}

@Override
public void write(byte address, short value) {
    System.out.println("Attempt to write to keyboard");
}

```

```

    }

    /**
     * Stores number value to input, so CPU will be able to read it. If CPU is
in    * the locked state at the method invoking moment, it will stop waiting
and    * continue executing of next instructions.
     * @param number inputted data
     */
    public synchronized void inputNumber(int number) {
        lastNumber = number; // saving the data
        if(locking) {
            // resume CPU if there is some locking and CPU waiting
            synchronized(notifyAboutInput) {
                notifyAboutInput.notify();
            }
        }
    }
}

```

MemoryCard.java

```

package cpusimulator.devices;

import cpusimulator.Device;
import cpusimulator.memory.Memory16;

/**
 * Memory card. Used to attach (to map) memory to our CPU model.
 *
 * @author Daniel Alexandre 2011
 */
public class MemoryCard implements Device {

    /** Attached memory bus. */
    private Memory16 memory;
    /** Memory size. */
    private int length;

    /**
     * Creates new memory with given memory bus interface and memory cells
count    *
     * @param memory memory bus interface
     * @param length memory cells count, should be greater than or equal to
zero    *
     * @throws NullPointerException if memory equals to null
     * @throws IllegalArgumentException if length is less than zero
     */
    public MemoryCard(Memory16 memory, int length) {
        if(memory == null) throw new NullPointerException();
        if(length < 0) throw new IllegalArgumentException();
        this.memory = memory;
        this.length = length;
    }

    @Override
    public int getMappingLength() {
        return length;
    }
}

```



```

        @Override
        public Memory16 getMappingMemory() {
            return memory;
        }
    }
}

```

Breakpoint16Memory.java

```

package cpusimulator.memory;

import cpusimulator.Breakpoint16;

/**
 * Memory used to handle read and writes at some addresses.
 * Normally used for breakpoints.
 *
 * @author Daniel Alexandre 2011
 */
public class BreakpointMemory16 implements Memory16 {

    /**
     * A record with linked-list capabilities which contains all
     * breakpoint-related info.
     */
    private class BreakpointHandlerRecord extends Breakpoint16 {

        /** Each breakpoint can be enabled or disabled, here is stored
        information about breakpoint state */
        public boolean enabled;
        /** Breakpoint handler, this object will be called when breakpoint
        catches operation */
        public Runnable handler;
        /** Breakpoint address */
        public byte address;
        /** Breakpoint type: it can be one of
        BREAKPOINT_READ,
        BREAKPOINT_WRITE, BREAKPOINT_CHANGE */
        public int type;

        /** Link to the next breakpoint. Used to hold many breakpoints for
        the same address */
        public BreakpointHandlerRecord next;

        @Override
        public boolean isEnabled() {
            return enabled;
        }

        @Override
        public void setEnabled(boolean enabled) {
            this.enabled = enabled;
        }

        @Override
        public void remove() {
            if (breakpointIndices[address & 0xFF] == this) {
                breakpointIndices[address & 0xFF] = next;
            } else {
                BreakpointHandlerRecord record = breakpointIndices[address
& 0xFF];
                while (record.next != this && record.next != null) record =
record.next;
            }
        }
    }
}

```



```

        if(record.next == this) record.next = next;
    }
}

@Override
public void setHandler(Runnable runnable) {
    handler = runnable;
}

@Override
public int getAddress() {
    return address & 0xFF;
}

@Override
public int getType() {
    return type;
}

public void runHandler() {
    if(handler != null) handler.run();
}

@Override
public short getValue() {
    return mem16.read(address);
}

@Override
public Memory16 getMemory() {
    return mem16;
}
}

/** Original memory. All reads and writes will be forwarded to this
memory. */
public Memory16 mem16;
/** Breakpoint records for each cell of memory. */
public BreakpointHandlerRecord[] breakpointIndices;
/** This callable object will be called when illegal address operation
will occurred. */
public Runnable addressExceptionHandler;

/**
 * Creates new breakpoint memory with specified destination memory,
length and invalid address handler.
 * @param mem16 memory, to which read/write operations will be
forwarded
 * @param length bytes count, must be in range 0..255 inclusive
 * @param invalidAddressHandler handler to handler operations with
wrong addresses
 * @throws IllegalArgumentException if length is wrong
 * @throws NullPointerException if mem16 or invalidAddressHandler are
equals to null
 */
public BreakpointMemory16(Memory16 mem16, int length,
    Runnable invalidAddressHandler) {
    if(length > 256) throw new IllegalArgumentException();
    if(invalidAddressHandler == null) throw new NullPointerException();
    if(mem16 == null) throw new NullPointerException();
    this.mem16 = mem16;
}

```

```

        breakpointIndices = new BreakpointHandlerRecord[length];
        addressExceptionHandler = invalidAddressHandler;
    }

    /**
     * Creates a breakpoint with the specified type and the specified
     * memory address
     * @param memoryAddress memory cell to which new breakpoint will be
     * attached
     * @param type type of breakpoint, should be one of BREAKPOINT_READ,
     * BREAKPOINT_WRITE or BREAKPOINT_CHANGE
     * @return newly created breakpoint
     * @throws IllegalArgumentException if type or memory address is
     * illegal
     */
    public Breakpoint16 createBreakpoint(byte memoryAddress, int type) {
        if(memoryAddress < 0 || memoryAddress >= breakpointIndices.length)
            throw new IllegalArgumentException();
        if(type != Breakpoint16.BREAKPOINT_CHANGE && type !=
            Breakpoint16.BREAKPOINT_READ && type !=
            Breakpoint16.BREAKPOINT_WRITE) throw new
            IllegalArgumentException();
        BreakpointHandlerRecord record = new BreakpointHandlerRecord();
        record.next = breakpointIndices[memoryAddress & 0xFF];
        record.address = memoryAddress;
        record.type = type;
        breakpointIndices[memoryAddress & 0xFF] = record;
        return record;
    }

    @Override
    public short read(byte address) {
        try {
            BreakpointHandlerRecord record = breakpointIndices[address &
            0xFF];
            while(record != null) {
                if(record.enabled) {
                    if(record.type == Breakpoint16.BREAKPOINT_READ)
                        record.runHandler();
                }
                record = record.next;
            }
            return mem16.read(address);
        } catch(ArrayIndexOutOfBoundsException exception) {
            addressExceptionHandler.run();
            return 0;
        }
    }

    @Override
    public void write(byte address, short value) {
        try {
            BreakpointHandlerRecord record = breakpointIndices[address &
            0xFF];
            while(record != null) {
                if(record.enabled) {
                    if(record.type == Breakpoint16.BREAKPOINT_WRITE)
                        record.runHandler();
                    if(record.type == Breakpoint16.BREAKPOINT_CHANGE) {
                        int oldValue = mem16.read(address);
                        if(value != oldValue) record.runHandler();
                    }
                }
            }
        }
    }

```



```

        }
        record = record.next;
    }
    mem16.write(address, value);
} catch (ArrayIndexOutOfBoundsException exception) {
    addressExceptionHandler.run();
}
}
}

```

DataMemory16.java

```

package cpusimulator.memory;

/**
 * Data memory model. Each cell has 16 bits of data and can be readed or
 * written.
 * This memory bus have no capabilities to protect memory data from
 * corruption.
 *
 * @author Daniel Alexandre 2011
 */
public class DataMemory16 implements Memory16 {

    /** Each memory cell is emulated as java array cell of short (16bit)
    type. */
    private short[] memoryCells;
    /** This handler called every time when CPU attempts to read or write
    something out of memory range. */
    private Runnable exceptionHandler;

    /**
     * Creates new 16-bit data memory model with invalid address handler.
     * @param length count of memory cells
     * @param invalidAddressHandler callable object for handling incorrect
    uses
     * of this memory model
     */
    public DataMemory16(int length, Runnable invalidAddressHandler) {
        if (length > 256) throw new IllegalArgumentException();
        if (invalidAddressHandler == null) throw new NullPointerException();
        memoryCells = new short[length];
        exceptionHandler = invalidAddressHandler;
    }

    /**
     * Returns memory cells count.
     * @return count of memory cells in this data memory model
     */
    public int getLength() {
        return memoryCells.length;
    }

    @Override
    public short read(byte address) {
        try {
            return memoryCells[address & 0xff];
        } catch (ArrayIndexOutOfBoundsException exception) {
            exceptionHandler.run();
            return 0;
        }
    }
}

```



```

    }
}

@Override
public void write(byte address, short value) {
    try {
        memoryCells[address & 0xff] = value;
    } catch (ArrayIndexOutOfBoundsException exception) {
        exceptionHandler.run();
    }
}
}
}

```

InstructionMemory16.java

```

package cpusimulator.memory;

/**
 * Class represents instruction memory model. It has a capabilities to
 * protect
 * executing code from corruption. Use lock() and unlock() methods to deny
 * or
 * allow to write data to this memory. This memory should be always locked
 * and
 * unlocks for a short time only when we change program code right after
 * assembling process.
 *
 * @author Daniel Alexandre 2011
 */
public class InstructionMemory16 extends DataMemory16 {

    /** Memory write lock */
    private boolean locked;
    /** Callable object which will be called when CPU will attempt to write
     * some data in locked memory region. */
    private Runnable exceptionHandler;

    /**
     * Creates new instruction memory model with specified length and
     * invalid operation handlers
     * @param length count of memory cells
     * @param invalidAddressHandler callable object used for handling CPU's
     * program failure
     * @param operationOnLockHandler callable object used for handling
     * CPU's program failure
     * @throws NullPointerException if operationOnLockHandler or
     * invalidAddressHandler is null
     */
    public InstructionMemory16(int length, Runnable invalidAddressHandler,
        Runnable operationOnLockHandler) { super(length,
        invalidAddressHandler);
        if (operationOnLockHandler == null) throw new
        NullPointerException();
        exceptionHandler = operationOnLockHandler;
    }

    /**
     * Disables data writing to this memory model. It should be called
     * right
     * after code storing process will be complete.
     */
}

```

```

public void lock() {
    locked = true;
}

/**
 * Enables data writing to this memory model. It should be called only
 * before code storing process will be started.
 */
public void unlock() {
    locked = false;
}

@Override
public void write(byte address, short value) {
    if(locked) {
        // discard attempt to write there if this memory is locked
        exceptionHandler.run();
    } else {
        super.write(address, value);
    }
}
}

```

Memory16.java

```

package cpusimulator.memory;

/**
 * Abstract 16-bit memory model, designed to read and write data. Max size
 * is
 * limited by pointer size (8 bit), so we can have memory with max of 256
 * bits.
 *
 * @author Daniel Alexandre 2011
 */
public interface Memory16 {

    /**
     * Reads data from memory model by given address.
     * @param address memory pointer
     * @return readed data
     */
    public short read(byte address);

    /**
     * Writes data to memory model by given address.
     * @param address memory pointer
     * @param value data which we have to write
     */
    public void write(byte address, short value);
}

```

VirtualMemory16.java

```
package cpusimulator.memory;  
  
import cpusimulator.Device;  
import java.util.ArrayList;
```

```

/**
 * This is very useful thing designed to compose many devices into one bus
 * system.
 *
 * @author Daniel Alexandre 2011
 */
public class VirtualMemory16 implements Memory16 {

    /** Just a data class used to keep information about memory bus. */
    private static class MemoryMapRecord {

        /** Mapped memory. */
        public Memory16 mappedMemory;
        /** Offset in virtual memory space. */
        public int mappingOffset;
        /** Offset in device memory space. */
        public int memoryOffset;
    }

    /** List of mapped devices; max count of mapped devices is 255. */
    public ArrayList<MemoryMapRecord> mapped =
        new ArrayList<MemoryMapRecord>();

    /** Information about device map ranges. */
    public byte[] mapIndices;
    /** Callable object which will be called each time when CPU will try to
    read from unmapped memory cell */
    public Runnable unmappedExceptionHandler;
    /** Callable object to handle events with incorrect address usage in
    mapped memory devices. */
    public Runnable addressExceptionHandler;

    /**
     * Creates new virtual memory card with specified length and bad event
     handlers.
     *
     * @param length size of memory, max size is 256
     * @param operateOnUnmappedHandler callable object to handle attempts
     to read from and write to unmapped memory cells
     * @param invalidAddressHandler callable object to handle operations
     with some invalid address specified
     * @throws NullPointerException if one of handlers is null
     * @throws IllegalArgumentException during attempt to create virtual
     memory with more than 256 memory cells
     */
    public VirtualMemory16(int length, Runnable operateOnUnmappedHandler,
    Runnable invalidAddressHandler) {
        if(length > 256) throw new IllegalArgumentException();
        if(operateOnUnmappedHandler ==
            null)
            throw
                new
    NullPointerException();
        if(invalidAddressHandler == null) throw new NullPointerException();
        mapIndices = new byte[length];
        unmappedExceptionHandler = operateOnUnmappedHandler;
        addressExceptionHandler = invalidAddressHandler;
    }

```

```
mapped.add(null); // device with #0 mapping index means "no device
mapped"
}

/**
 * Attaches some device to specified virtual memory cells range.
 * @param device device which will be attached
 * @param memoryAddress device memory offset
```

```

    * @param mapAddress mapping offset
    * @throws IllegalArgumentException if there is no more devices slots
    available
    * or if memory addresses points out of the virtual memory range or
    during
    * attempt to attach device to memory range with already attached cells
    */
    public void map(Device device, int memoryAddress, int mapAddress) {
        // check for free device space availability
        if(mapped.size() > 256) throw new IllegalStateException("too much
mapped memories");
        // check for memory ranges
        if(memoryAddress < 0 || device.getMappingLength() < 0 || mapAddress
+ device.getMappingLength() > mapIndices.length)
            throw new IllegalArgumentException();
        // check for memory conflicts
        int mapLength = device.getMappingLength();
        for(int i = 0; i < mapLength; i++)
            if(mapIndices[mapAddress + i] != 0)
                throw new IllegalStateException("it conflicts with other
mapped memory");
        int mapIndex = mapped.size();
        // creating device info record
        MemoryMapRecord record = new MemoryMapRecord();
        record.mappedMemory = device.getMappingMemory();
        record.mappingOffset = mapAddress;
        record.memoryOffset = memoryAddress;
        mapped.add(record); // add it to our device list
        // perform an attaching to cells
        for(int i = 0; i < mapLength; i++)
            mapIndices[mapAddress + i] = (byte)mapIndex;
    }

    @Override
    public short read(byte address) {
        try {
            // get mapped device
            int device = mapIndices[address & 0xff] & 0xff;
            if(device == 0) {
                // memory cell is not mapped yet
                unmappedExceptionHandler.run();
                return 0;
            }
            // memory cell have attached device. Calculate its address and
            redirect read operation to attached device
            MemoryMapRecord record = mapped.get(device & 0xff);
            return record.mappedMemory.read((byte)((address & 0xff) -
record.mappingOffset + record.memoryOffset));
        } catch(ArrayIndexOutOfBoundsException exception) {
            addressExceptionHandler.run();
            return 0;
        }
    }

    @Override
    public void write(byte address, short value) {
        try {
            // get mapped device
            int device = mapIndices[address & 0xff] & 0xff;
            if(device == 0) unmappedExceptionHandler.run(); // memory cell
is not mapped yet

```

```

        else {
            // memory cell have attached device. Calculate its address
            and redirect write operation to attached device
            MemoryMapRecord record = mapped.get(device & 0xff);
            record.mappedMemory.write((byte)((address & 0xff) -
record.mappingOffset + record.memoryOffset), value);
        }
    } catch (ArrayIndexOutOfBoundsException exception) {
        addressExceptionHandler.run();
    }
}

```