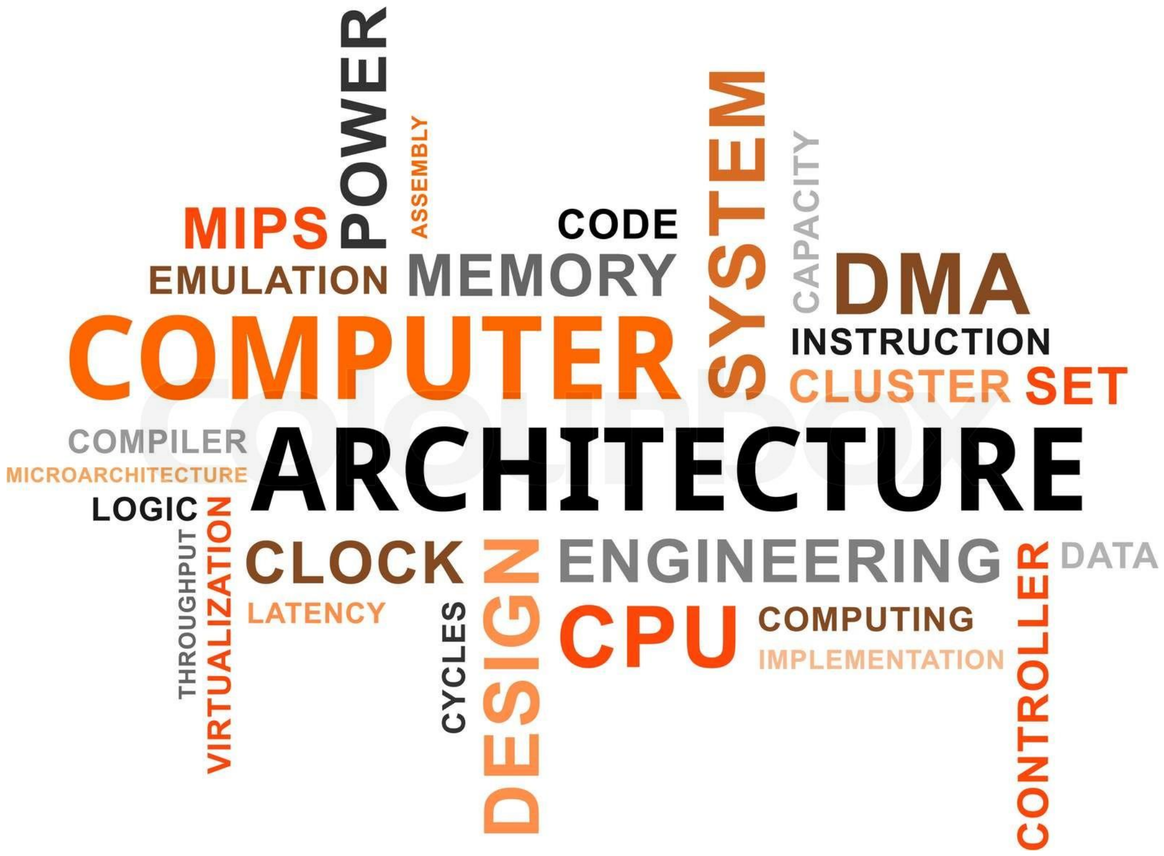


CS322 MINI PROJECT

EXTENDING MARS FUNCTIONALITIES BY WRITING TOOLS

Adarsh Kumar | 2001CS02 | 22-11-2022

Introduction:

MARS can be customized and extended in four different ways:

i. writing new MARS Tools,
ii. writing new MIPS system calls,
iii. writing new MIPS pseudo-instructions, and
iv. writing new MIPS basic instructions.

For my project, I have written four new tools:

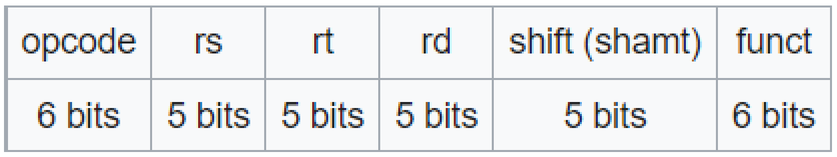
1. R-Type Instruction Counter
2. I-Type Instruction Counter
3. J-Type Instruction Counter
4. Change in Register’s Value

This report thoroughly discusses the functionalities of all of the tools in the upcoming section.

R-Type Instruction Counter:

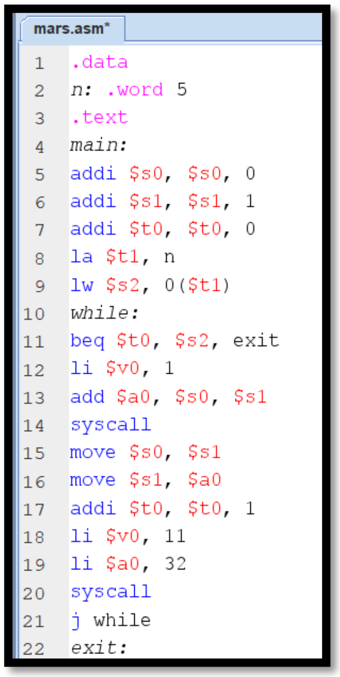
This tool counts the number of R-Type instructions that has been executed.

R instructions are used when all the data values used by the instruction are located in registers.



opcode	rs	rt	rd	shift (shamt)	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

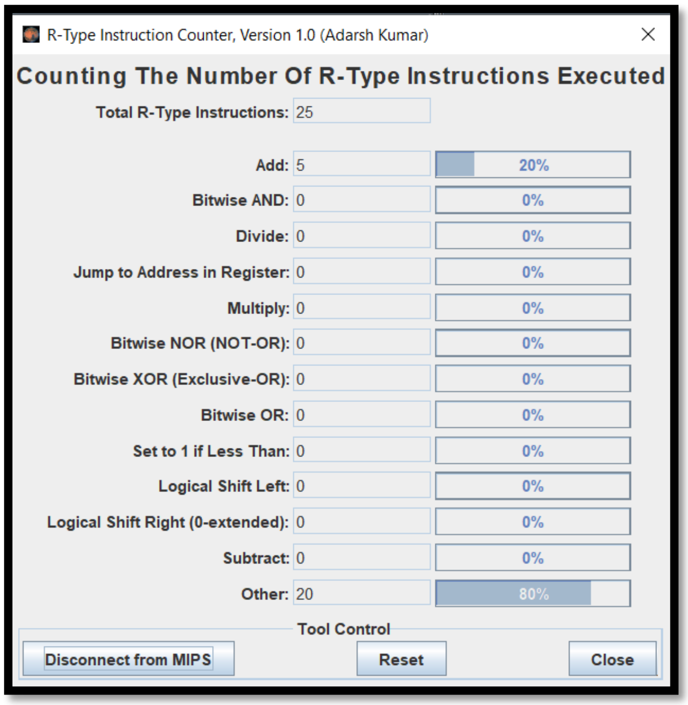
For the following (Fibonacci) MIPS code,



mars.asm*

```
1  .data
2  n: .word 5
3  .text
4  main:
5  addi $s0, $s0, 0
6  addi $s1, $s1, 1
7  addi $t0, $t0, 0
8  la $t1, n
9  lw $s2, 0($t1)
10 while:
11 beq $t0, $s2, exit
12 li $v0, 1
13 add $a0, $s0, $s1
14 syscall
15 move $s0, $s1
16 move $s1, $a0
17 addi $t0, $t0, 1
18 li $v0, 11
19 li $a0, 32
20 syscall
21 j while
22 exit:
```

the tool will display the output as below.



R-Type Instruction Counter, Version 1.0 (Adarsh Kumar)

Counting The Number Of R-Type Instructions Executed

Total R-Type Instructions:	25	
Add:	5	20%
Bitwise AND:	0	0%
Divide:	0	0%
Jump to Address in Register:	0	0%
Multiply:	0	0%
Bitwise NOR (NOT-OR):	0	0%
Bitwise XOR (Exclusive-OR):	0	0%
Bitwise OR:	0	0%
Set to 1 if Less Than:	0	0%
Logical Shift Left:	0	0%
Logical Shift Right (0-extended):	0	0%
Subtract:	0	0%
Other:	20	80%

Since line number 13 gets executed 5 times (because of the loop), our tool shows that ‘add’ instruction has been executed 5 times. It also shows that this instruction is contributing to 20% of the total R-Type instruction that has been executed.

RTypeInstructionCounter.java extends AbstractMarsToolAndApplication class. By doing so, we get the following elements,

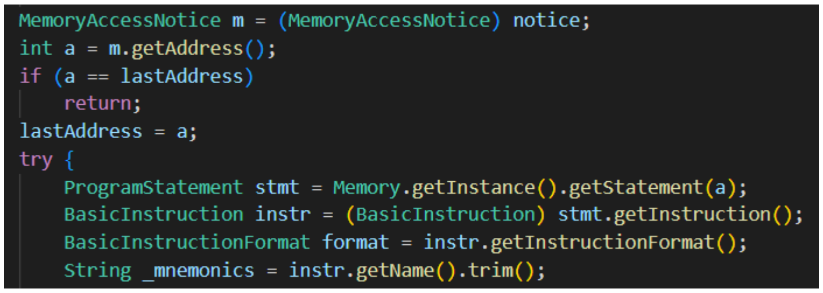
- ability to run either from the Tools menu or as a free-standing application,
- basic user interface JDialog with Tool Control section (for tools),
- basic user interface JFrame with Application Control section (for applications),
- basic user interface layout (BorderLayout) and rendering algorithm,
- basic MIPS memory and register observer capabilities.

String getName() returns the tool's display name.

JComponent buildMainDisplayArea() constructs the central area of the tool's graphical user interface. In this method, we create text fields and progress bars for each mnemonic.

addAsObserver() method is available for registering as a memory and/or register observer.

In processMIPSUpdate(), we extract the format of the current instruction and the current mnemonic.



```java
MemoryAccessNotice m = (MemoryAccessNotice) notice;
int a = m.getAddress();
if (a == lastAddress)
    return;
lastAddress = a;
try {
    ProgramStatement stmt = Memory.getInstance().getStatement(a);
    BasicInstruction instr = (BasicInstruction) stmt.getInstruction();
    BasicInstructionFormat format = instr.getInstructionFormat();
    String _mnemonics = instr.getName().trim();
```

If the current mnemonic belongs to the set {add, and, div, jr, mult, nor, xor, or, slt, sll, srl, sub}, we increase its corresponding counter.

All the remaining R-Type instruction goes to the 'other' section.

At the end of the execution of all the instructions, we update the display using updateDisplay() method. We set the value for each text field and progress bar.

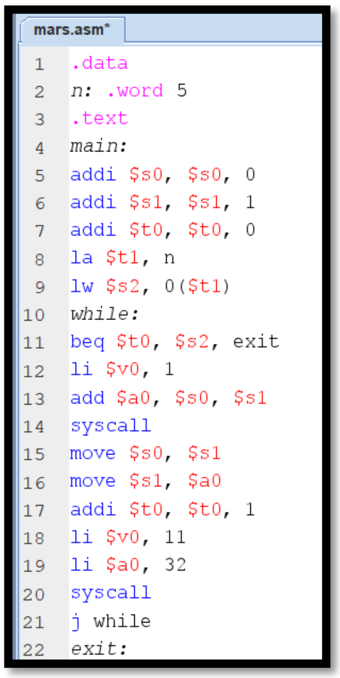
I-Type Instruction Counter:

This tool counts the number of I-Type instructions that has been executed.

I instructions are used when the instruction must operate on an immediate value and a register value. Immediate values may be a maximum of 16 bits long. Larger numbers may not be manipulated by immediate instructions.

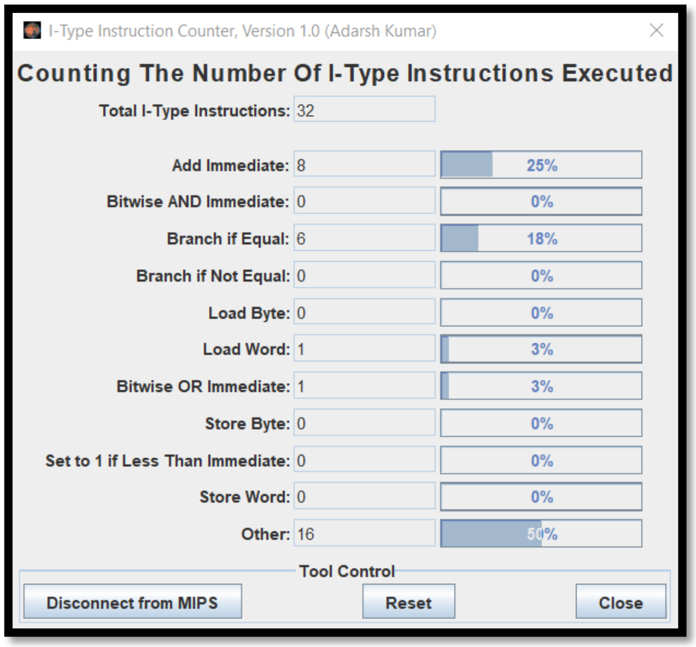
opcode	rs	rt	IMM
6 bits	5 bits	5 bits	16 bits

For the following (Fibonacci) MIPS code,



```
.data
n: .word 5
.text
main:
addi $s0, $s0, 0
addi $s1, $s1, 1
addi $t0, $t0, 0
la $t1, n
lw $s2, 0($t1)
while:
beq $t0, $s2, exit
li $v0, 1
add $a0, $s0, $s1
syscall
move $s0, $s1
move $s1, $a0
addi $t0, $t0, 1
li $v0, 11
li $a0, 32
syscall
j while
exit:
```

the tool will display the output as below.



Counting The Number Of I-Type Instructions Executed

Total I-Type Instructions:	32	
Add Immediate:	8	25%
Bitwise AND Immediate:	0	0%
Branch if Equal:	6	18%
Branch if Not Equal:	0	0%
Load Byte:	0	0%
Load Word:	1	3%
Bitwise OR Immediate:	1	3%
Store Byte:	0	0%
Set to 1 if Less Than Immediate:	0	0%
Store Word:	0	0%
Other:	16	50%

Since line number 11 gets executed 6 times (5 times because of the loop, 1 time to exit the loop), our tool shows that ‘beq’ instruction has been executed 6 times. It also shows that this instruction is contributing to 18% of the total I-Type instruction that has been executed.

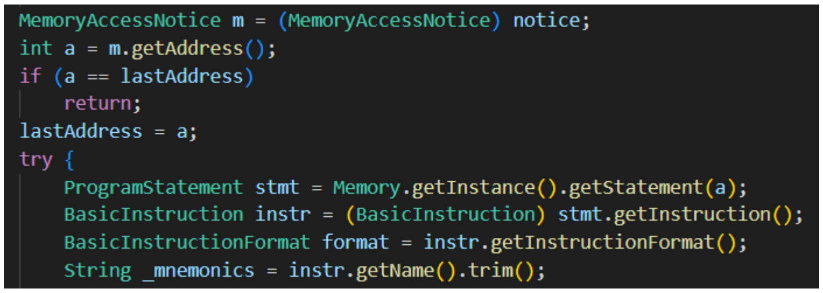
ITypeInstructionCounter.java extends AbstractMarsToolAndApplication class.

String getName() returns the tool’s display name.

JComponent buildMainDisplayArea() constructs the central area of the tool’s graphical user interface. In this method, we create text fields and progress bars for each mnemonic.

addAsObserver() method is available for registering as a memory and/or register observer.

In processMIPSUpdate(), we extract the format of the current instruction and the current mnemonic.



```
MemoryAccessNotice m = (MemoryAccessNotice) notice;
int a = m.getAddress();
if (a == lastAddress)
    return;
lastAddress = a;
try {
    ProgramStatement stmt = Memory.getInstance().getStatement(a);
    BasicInstruction instr = (BasicInstruction) stmt.getInstruction();
    BasicInstructionFormat format = instr.getInstructionFormat();
    String _mnemonics = instr.getName().trim();
```

If the current mnemonic belongs to the set {addi, andi, beq, bne, lb, lw, ori, sb, slti, sw}, we increase its corresponding counter.

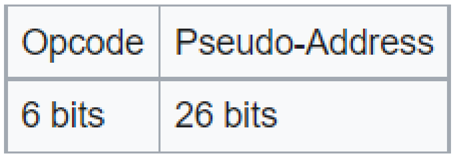
All the remaining I-Type instruction goes to the 'other' section.

At the end of the execution of all the instructions, we update the display using updateDisplay() method. We set the value for each text field and progress bar.

J-Type Instruction Counter:

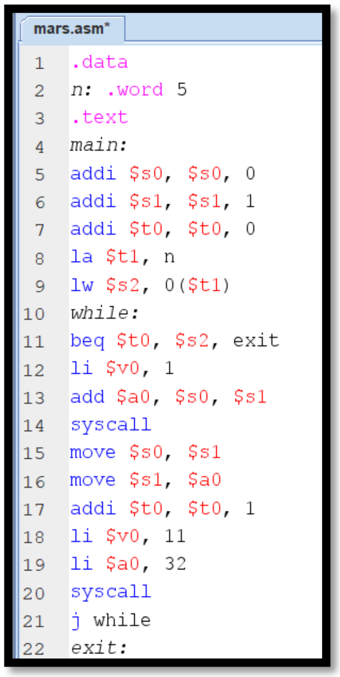
This tool counts the number of J-Type instructions that has been executed.

J instructions are used when a jump needs to be performed. The J instruction has the most space for an immediate value, because addresses are large numbers.



Opcode	Pseudo-Address
6 bits	26 bits

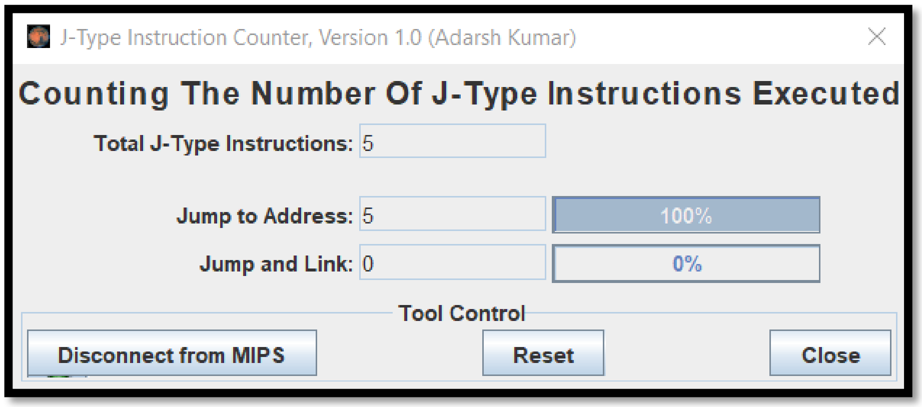
For the following (Fibonacci) MIPS code,



mars.asm*

```
1   .data
2   n: .word 5
3   .text
4   main:
5   addi $s0, $s0, 0
6   addi $s1, $s1, 1
7   addi $t0, $t0, 0
8   la $t1, n
9   lw $s2, 0($t1)
10  while:
11  beq $t0, $s2, exit
12  li $v0, 1
13  add $a0, $s0, $s1
14  syscall
15  move $s0, $s1
16  move $s1, $a0
17  addi $t0, $t0, 1
18  li $v0, 11
19  li $a0, 32
20  syscall
21  j while
22  exit:
```

the tool will display the output as below.



J-Type Instruction Counter, Version 1.0 (Adarsh Kumar)

Counting The Number Of J-Type Instructions Executed

Total J-Type Instructions:	5	
Jump to Address:	5	100%
Jump and Link:	0	0%

Since line number 21 gets executed 5 times (because of the loop), our tool shows that 'j' instruction has been executed 5 times. It also shows that this instruction is contributing to 100% of the total J-Type instruction that has been executed.

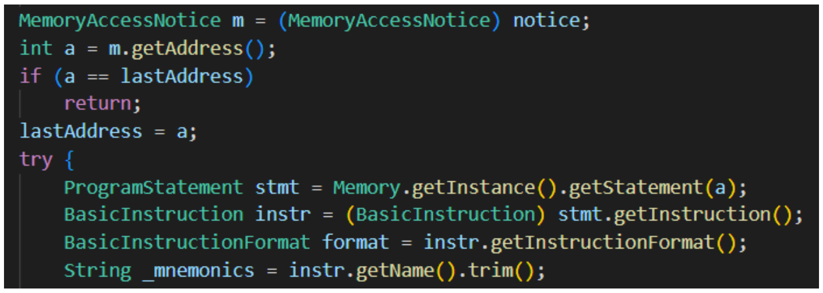
JTypeInstructionCounter.java extends AbstractMarsToolAndApplication class.

String getName() returns the tool's display name.

JComponent buildMainDisplayArea() constructs the central area of the tool's graphical user interface. In this method, we create text fields and progress bars for each mnemonic.

addAsObserver() method is available for registering as a memory and/or register observer.

In processMIPSUpdate(), we extract the format of the current instruction and the current mnemonic.



```
MemoryAccessNotice m = (MemoryAccessNotice) notice;
int a = m.getAddress();
if (a == lastAddress)
    return;
lastAddress = a;
try {
    ProgramStatement stmt = Memory.getInstance().getStatement(a);
    BasicInstruction instr = (BasicInstruction) stmt.getInstruction();
    BasicInstructionFormat format = instr.getInstructionFormat();
    String _mnemonics = instr.getName().trim();
```

If the current mnemonic belongs to the set {j, jal}, we increase its corresponding counter.

At the end of the execution of all the instructions, we update the display using updateDisplay() method. We set the value for each text field and progress bar.

Change in Register's Value:

This tool displays the instruction code (in binary) for each instruction. It also displays the initial value and the updated value of the registers after each instruction.

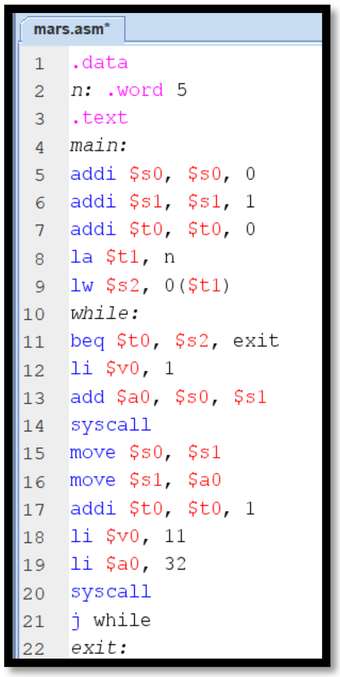
All MIPS instructions are encoded in binary and are 32 bits long. All instructions have an opcode (or op) that specifies the operation (first 6 bits).

There are 32 registers. (Need 5 bits to uniquely identify all 32.)

`add $s0, $s1, $s2`   `(registers 16, 17, 18)`

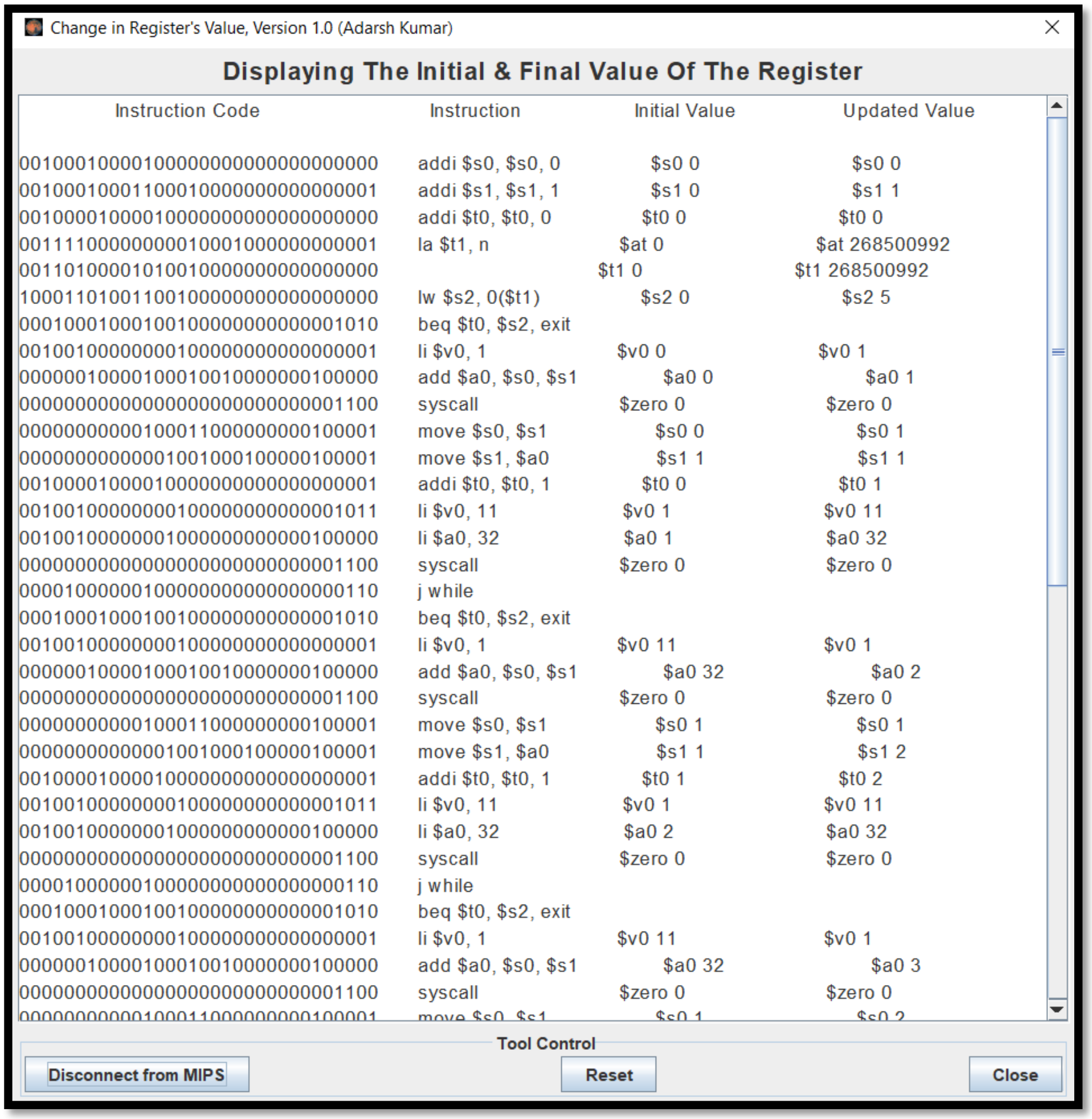
op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
000000	10001	10010	10000	00000	100000

For the following (Fibonacci) MIPS code,


mars.asm*

```
.data
n: .word 5
.text
main:
addi $s0, $s0, 0
addi $s1, $s1, 1
addi $t0, $t0, 0
la $t1, n
lw $s2, 0($t1)
while:
beq $t0, $s2, exit
li $v0, 1
add $a0, $s0, $s1
syscall
move $s0, $s1
move $s1, $a0
addi $t0, $t0, 1
li $v0, 11
li $a0, 32
syscall
j while
exit:
```

the tool will display the output as below.


Change in Register's Value, Version 1.0 (Adarsh Kumar)

Displaying The Initial & Final Value Of The Register

Instruction Code	Instruction	Initial Value	Updated Value
00100010000100000000000000000000	addi $s0, $s0, 0	$s0 0	$s0 0
00100010001100010000000000000001	addi $s1, $s1, 1	$s1 0	$s1 1
00100001000010000000000000000000	addi $t0, $t0, 0	$t0 0	$t0 0
00111100000000010001000000000001	la $t1, n	$at 0	$at 268500992
00110100001010010000000000000000		$t1 0	$t1 268500992
10001101001100100000000000000000	lw $s2, 0($t1)	$s2 0	$s2 5
00010001000100100000000000001010	beq $t0, $s2, exit		
00100100000000100000000000000001	li $v0, 1	$v0 0	$v0 1
00000010000100010010000000100000	add $a0, $s0, $s1	$a0 0	$a0 1
00000000000000000000000000001100	syscall	$zero 0	$zero 0
00000000000100011000000000100001	move $s0, $s1	$s0 0	$s0 1
00000000000001001000100000100001	move $s1, $a0	$s1 1	$s1 1
00100001000010000000000000000001	addi $t0, $t0, 1	$t0 0	$t0 1
00100100000000100000000000001011	li $v0, 11	$v0 1	$v0 11
00100100000001000000000000100000	li $a0, 32	$a0 1	$a0 32
00000000000000000000000000001100	syscall	$zero 0	$zero 0
00001000000100000000000000000110	j while		
00010001000100100000000000001010	beq $t0, $s2, exit		
00100100000000100000000000000001	li $v0, 1	$v0 11	$v0 1
00000010000100010010000000100000	add $a0, $s0, $s1	$a0 32	$a0 2
00000000000000000000000000001100	syscall	$zero 0	$zero 0
00000000000100011000000000100001	move $s0, $s1	$s0 1	$s0 1
00000000000001001000100000100001	move $s1, $a0	$s1 1	$s1 2
00100001000010000000000000000001	addi $t0, $t0, 1	$t0 1	$t0 2
00100100000000100000000000001011	li $v0, 11	$v0 1	$v0 11
00100100000001000000000000100000	li $a0, 32	$a0 2	$a0 32
00000000000000000000000000001100	syscall	$zero 0	$zero 0
00001000000100000000000000000110	j while		
00010001000100100000000000001010	beq $t0, $s2, exit		
00100100000000100000000000000001	li $v0, 1	$v0 11	$v0 1
00000010000100010010000000100000	add $a0, $s0, $s1	$a0 32	$a0 3
00000000000000000000000000001100	syscall	$zero 0	$zero 0
00000000000100011000000000100001	move $s0, $s1	$s0 1	$s0 2

The initial value of the $s1 register was 0 which changed to 1 after executing line number 6 (addi $s1, $s1, 1). At line number 2 of our tool, the same can be observed.

Interestingly, line number 4 and 5 in our tool correspond to only one instruction at line number 8 (la $t1, n). This is because it is a pseudo-instruction made up of two different instructions – lui $1, 4097 and ori $9, $1, 0.

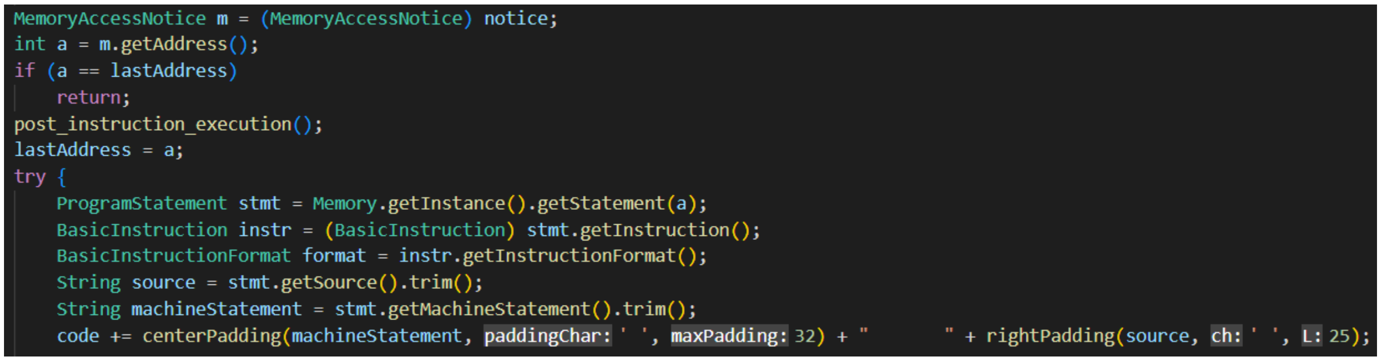
ChangeInRegister.java extends AbstractMarsToolAndApplication class.

String getName() returns the tool's display name.

JComponent buildMainDisplayArea() constructs the central area of the tool's graphical user interface. In this method, we create a text area.

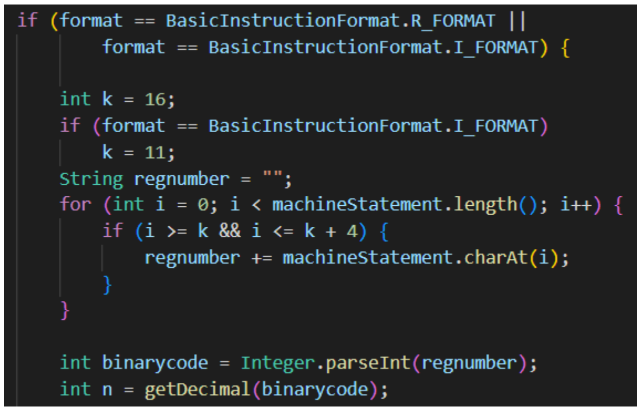
addAsObserver() method is available for registering as a memory and/or register observer.

In processMIPSUpdate(), we extract the source (using getSource()) and the machine code (using getMachineStatement()) of the current instruction.



```java
MemoryAccessNotice m = (MemoryAccessNotice) notice;
int a = m.getAddress();
if (a == lastAddress)
    return;
post_instruction_execution();
lastAddress = a;
try {
    ProgramStatement stmt = Memory.getInstance().getStatement(a);
    BasicInstruction instr = (BasicInstruction) stmt.getInstruction();
    BasicInstructionFormat format = instr.getInstructionFormat();
    String source = stmt.getSource().trim();
    String machineStatement = stmt.getMachineStatement().trim();
    code += centerPadding(machineStatement, paddingChar: ' ', maxPadding: 32) + "        " + rightPadding(source, ch: ' ', L: 25);
```

Then we extract the register number (rd in case of R, rt in case of I) using the machine code.

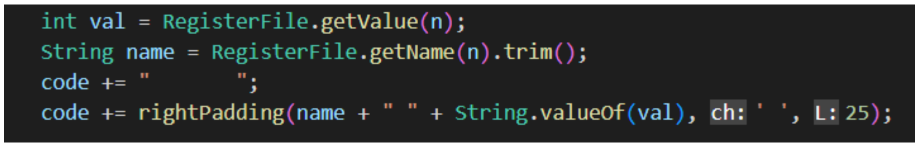


```java
if (format == BasicInstructionFormat.R_FORMAT ||
        format == BasicInstructionFormat.I_FORMAT) {

    int k = 16;
    if (format == BasicInstructionFormat.I_FORMAT)
        k = 11;
    String regnumber = "";
    for (int i = 0; i < machineStatement.length(); i++) {
        if (i >= k && i <= k + 4) {
            regnumber += machineStatement.charAt(i);
        }
    }

    int binarycode = Integer.parseInt(regnumber);
    int n = getDecimal(binarycode);
```

Now finally we extract the register's name using getName() method (need to be defined in '.\mars\mips\hardware\RegisterFile.java') and old value using getValue() method.



```java
int val = RegisterFile.getValue(n);
String name = RegisterFile.getName(n).trim();
code += "        ";
code += rightPadding(name + " " + String.valueOf(val), ch: ' ', L: 25);
```

To extract the new value of the register, we do the same process just before the execution of the next instruction.

At the end of the execution of all the instructions, we update the display using updateDisplay() method. We set the text using setText() method.

Conclusion:

All these java files need to be placed in `.\mars\tools\` and compiled using the command

```
javac '.\mars\tools\<filename>.java'
```

Now we need to create jar file using the command

```
.\CreateMarsJar.bat
```

And run the created jar file using the command

```
.\Mars.jar
```