Using Documentation

The instructions that have been implemented

The first 42 basic instructions are implemented, including:

```
lui add addi sub slt slti sltu
sltiu and andi or ori xor xori
nor sll sllv srl srlv sra srav
lw lwx lh lhx lhu lhux sw
swx sh shx beq bne bgezal j
jal jr jalr mfc0 mtc0 eret syscall
```

Or, list by function,

data transfer instructions:

```
lui, lw, lwx, lh, lhx, lhu, lhux, sw, swx, sh, shx
```

arithmetic operation instructions:

```
add, addi, sub              ;addition and subtraction
slt, slti, sltu, sltiu      ;less than comparsion
```

logic operation instructions:

```
and, andi, or, ori, xori, nor
```

shift instructions:

```
sll, sllv, srl, srlv, sra, srav
```

branch instructions:

```
beq, bne               ;conditional branch
j, jr                  ;unconditional branch
bgezal, jal, jalr      ;subroutine call
```

coprocessor instructions:

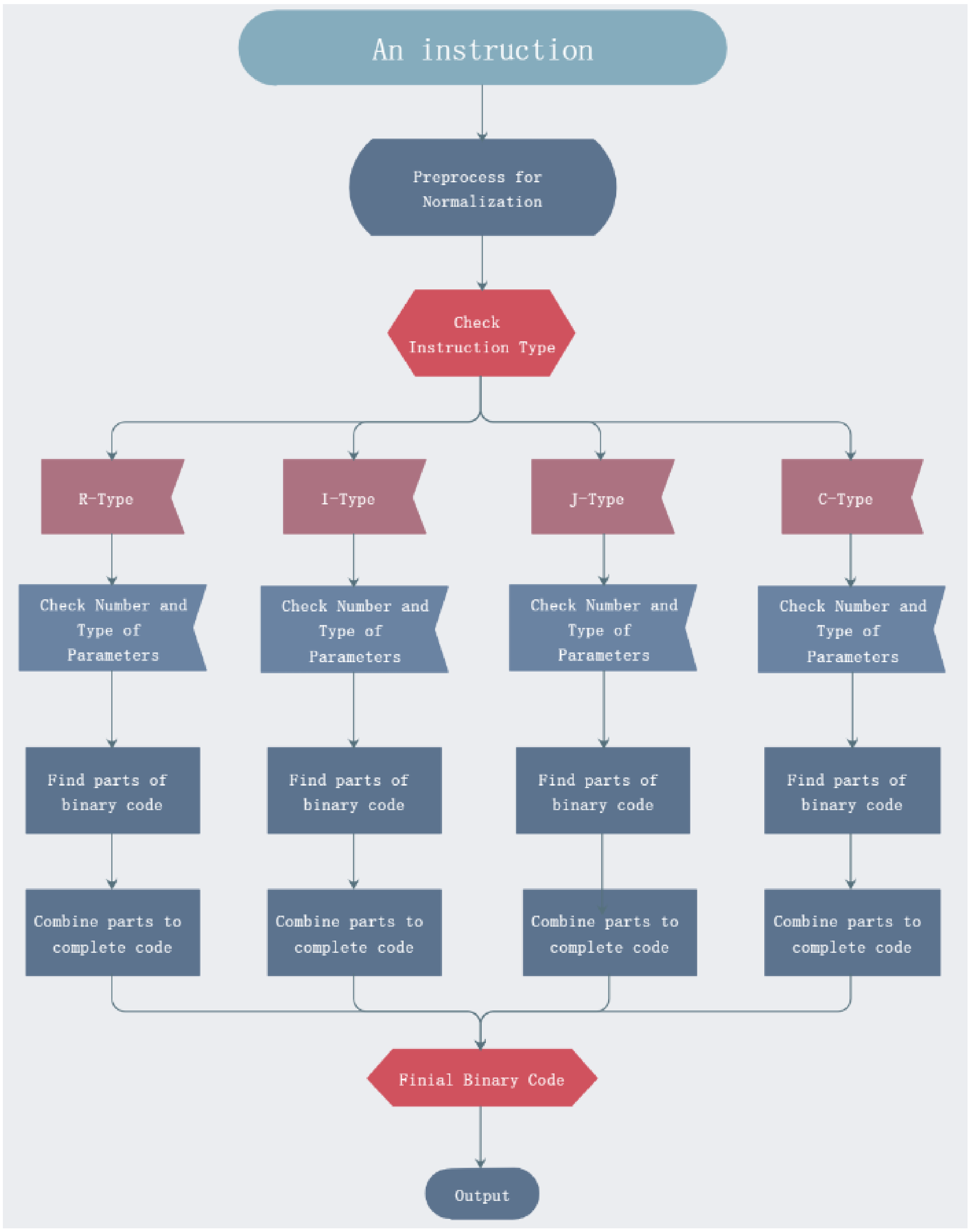
```
mfc0, mtc0
```

system function call instructions:

```
eret, syscall
```

Block diagram

The simple block diagram of instruction processing is following:


An instruction
Preprocess for Normalization
Check Instruction Type
R-Type
I-Type
J-Type
C-Type
Check Number and Type of Parameters
Check Number and Type of Parameters
Check Number and Type of Parameters
Check Number and Type of Parameters
Find parts of binary code
Find parts of binary code
Find parts of binary code
Find parts of binary code
Combine parts to complete code
Combine parts to complete code
Combine parts to complete code
Combine parts to complete code
Finial Binary Code
Output

Explanation:

When an instruction is inputted, the program executed following steps:

- preprocessing, to obtain a normal form of the instruction;
- split parameters from the instruction after preprocessing;
- determine the type(RIJC) of the instruction;

- find the according code to the operand, register numbers, immediate, shamt, or function;
- combine code parts into a complete binary code;

Some relational functions in source code:

```
string prepare(string s);   //preprocessing function
int isRIJC(string op);      //check the instruction type(RIJC)
int translateR(int op, vector<string> instr); //process R-type instr
int translateI(int op, vector<string> instr); //process I-type instr
int translateJ(int op, vector<string> instr); //process J-type instr
int translateC(int op, vector<string> instr); //process C-type instr
int regNum2(string regName);    //convert the regsiter name to number
int opcode(string op);  //obtain the opcode from the operand
int funcNum(string op); //find the function code
```

How to use the program

This program only process **single** instruction once.

The program can used by following steps:

- run the executable file in console or just double click it;
- input an instruction in the console and end with an enter;
- the program will output the process result under the input instruction;
- keep inputting and the program will keep outputting the process results;

- the inputting can be ended with inputting just an enter;

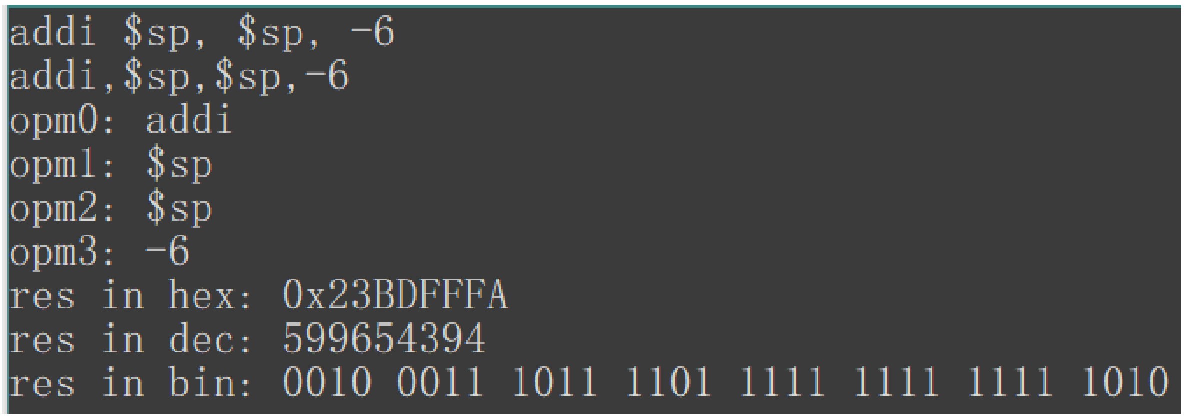
Case Analysis

Test Case 1:

Input instruction:

```
addi $sp, $sp, -6
```

Running result:



```
addi $sp, $sp, -6
addi,$sp,$sp,-6
opm0: addi
opm1: $sp
opm2: $sp
opm3: -6
res in hex: 0x23BDFFFA
res in dec: 599654394
res in bin: 0010 0011 1011 1101 1111 1111 1111 1010
```

Analysis:

after preprocessing, all extra spaces are removed, and spaces will be replaced by comma;

there are 4 parameters in total;

this instruction is I-type;

each part's corresponding code:

```
opcode: 001000
rs: 11101
rt: 11101
immed: 1111_1111_1111_1010
```

finial result:

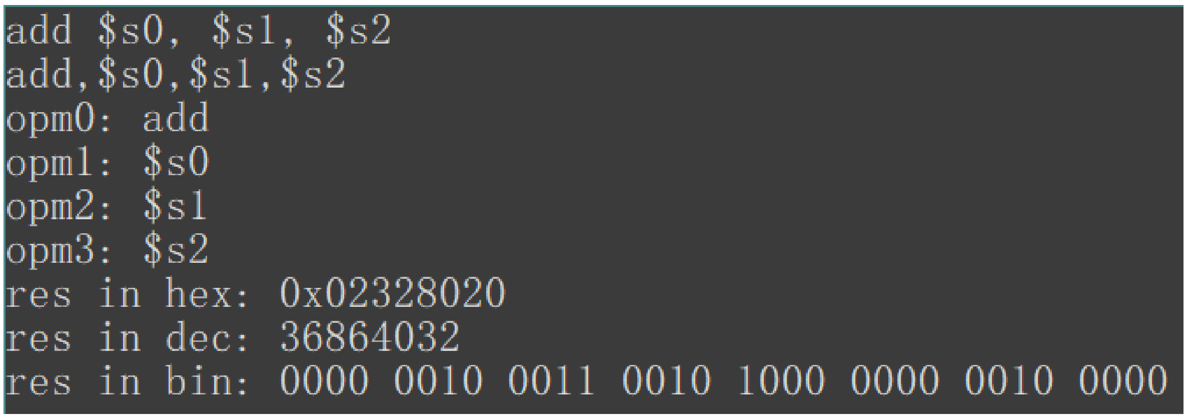
```
hex: 23BDFFFAh
dec: 599654394
bin: 0010_0011_1011_1101_1111_1111_1111_1010
```

Test Case 2:

Input instruction:

```
add $s0, $s1, $s2
```

Running result:



```
add $s0, $s1, $s2
add,$s0,$s1,$s2
opm0: add
opm1: $s0
opm2: $s1
opm3: $s2
res in hex: 0x02328020
res in dec: 36864032
res in bin: 0000 0010 0011 0010 1000 0000 0010 0000
```

Analysis:

after preprocessing, all extra spaces are removed, and spaces will be replaced by comma;

there are 4 parameters in total;

this instruction is R-type;

each part's corresponding code:

```
opcode: 000000
rs: 10001
rt: 10010
rd: 10000
shamt: 00000
function: 100000
```

finial result:

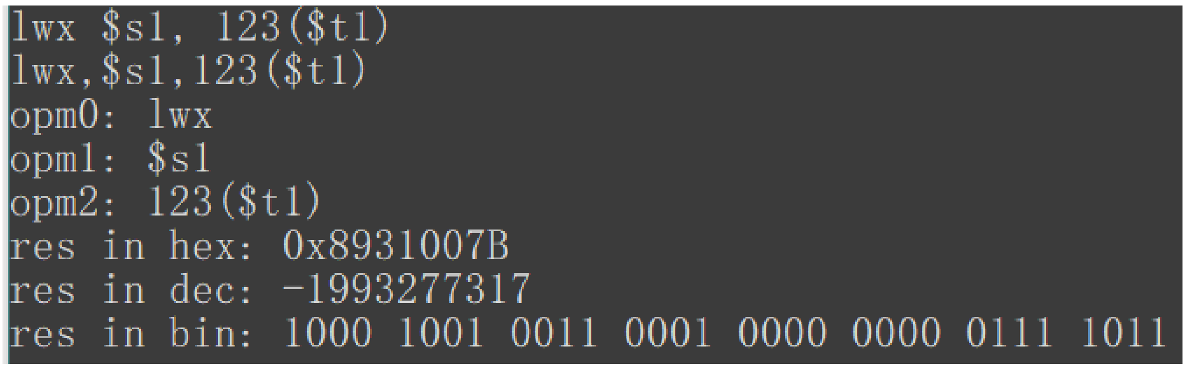
```
hex: 02328020h
dec: 36864032
bin: 0000_0010_0011_0010_1000_0000_0010_0000
```

Test case 3:

Input instruction:

```
lwx $s1, 123($t1)
```

Running result:


```
lwx $s1, 123($t1)
lwx,$s1,123($t1)
opm0: lwx
opm1: $s1
opm2: 123($t1)
res in hex: 0x8931007B
res in dec: -1993277317
res in bin: 1000 1001 0011 0001 0000 0000 0111 1011
```

Analysis:

after preprocessing, all extra spaces are removed, and spaces will be replaced by comma;

there are 3 parameters in total;

this instruction is I-type;

each part's corresponding code:

```
opcode: 100010
rs: 01001
rt: 10001
immed: 0000_0000_0111_1011
```

finial result:

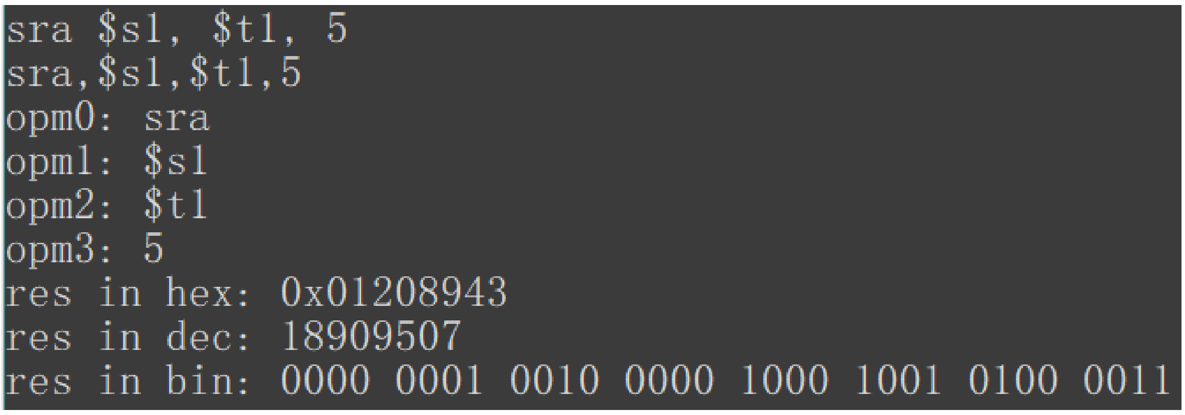
```
hex: 8931007Bh
dec: -1993277317
bin: 1000_1001_0011_0001_0000_0000_0111_1011
```

Test Case 4:

Input instruction:

```
sra $s1, $t1, 5
```

Running result:



```
sra $s1, $t1, 5
sra,$s1,$t1,5
opm0: sra
opm1: $s1
opm2: $t1
opm3: 5
res in hex: 0x01208943
res in dec: 18909507
res in bin: 0000 0001 0010 0000 1000 1001 0100 0011
```

Analysis:

after preprocessing, all extra spaces are removed, and spaces will be replaced by comma;

there are 4 parameters in total;

this instruction is R-type;

each part's corresponding code:

```
opcode: 000000
rs: 01001
rt: 00000
rd: 10001
shamt: 00101
function: 000011
```

finial result:

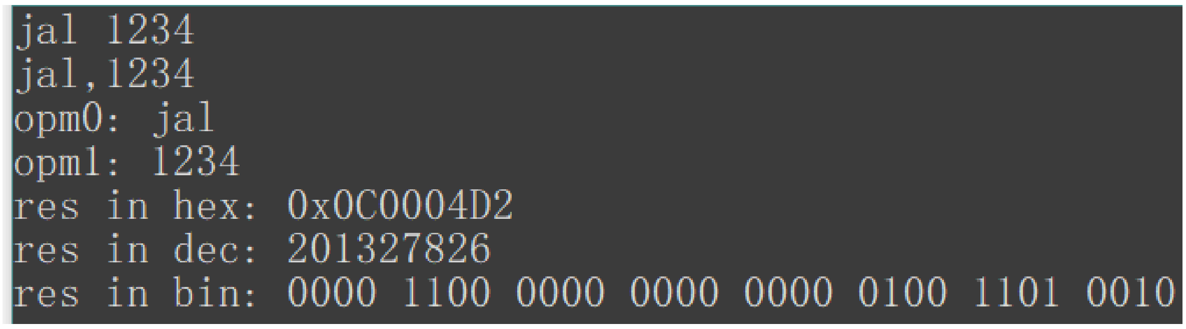
```
hex: 01208943h
dec: 18909507
bin: 0000_0001_0010_0000_1000_1001_0100_0011
```

Test Case 5:

Input instruction:

```
jal 1234
```

Running result:



```
jal 1234
jal,1234
opm0:  jal
opm1:  1234
res in hex:  0x0C0004D2
res in dec:  201327826
res in bin:  0000 1100 0000 0000 0000 0100 1101 0010
```

Analysis:

after preprocessing, all extra spaces are removed, and spaces will be replaced by comma;

there are 2 parameters in total;

this instruction is J-type;

each part's corresponding code:

```
opcode: 000011
addr: 00_0000_0000_0000_0100_1101_0010
```

finial result:

```
hex: 0C0004D2h
dec: 201327826
bin: 0000_1100_0000_0000_0000_0100_1101_0010
```