



**Names and IDs:** Youssef Mentawy 900203210, Mohamed ElShemy

**Course:** CSCE 2303/231 – Computer Organization and Assembly Programming

**Assignment:** Project #1 Report

**Instructor:** Dr. Nourhan Zayed

**Submission Date:** 11/05/2023

# Project #1 Report

## I. Project Description

This Project is a functional RISC-V simulator program capable of tracing the execution of 40 RV32I instructions from page 130 in “The RISC-V Instruction Set Manual Volume I: Unprivileged ISA.” The program is implemented in C++ that reads the instructions from a text file called “testCase,” and starts tracing and executing the RV32I instructions found in the text file. The program can also print the registers' contents in three different formats: Decimal, Binary, and Hexadecimal.

## 2. Design

The program mainly contains the 37 C++ functions equivalent to the 37 required instructions, excluding the three halting instructions (ecall, ebreak, and fence). The program also contains other utility functions that facilitate the usage of the program, like `get_offset`. This section also contains a description of each function.

### A. Shifting Functions

- **Bitwise AND is useful for extracting only the desired bits (i.e., If we want the lower five bits only from a 32-bit number, we bitwise AND with 0b11111)**

- The static casting of the integers is significant to ensure that it is zero-extended and not sign-extended for the logical shift or any unsigned operation. It is used in the right shift and not the left simply because the sign bits are on the left, so we need to consider them when shifting right.

- **sra (shift right arithmetic)**

```
void sra(int* x, int* y, int* z)
{
    int temp = *z & 0b11111;
    *x = *y >> temp;
}
```

- The sra function performs an arithmetic right shift on the value in the y register, with the number of shift positions specified by the least significant five bits of the z register. The result is stored in the x register. This operation maintains the sign bit, which is crucial for handling negative numbers.

- **srai (shift right arithmetic immediate)**

```
void srai(int* x, int* y, int z)
{
    int temp = z & 0b11111;
    *x = *y >> temp;
}
```

- The srai function is similar to sra, but the shift amount is specified as an immediate value (integer) z rather than a register value. This offers more flexibility and faster execution in certain situations.

- **Srl (shift right logical)**

```
void srl(int* x, int* y, int* z)
{
    int temp = *z & 0b11111;
    *x = static_cast<uint32_t>(*y) >> temp;
    cout << "in srl x = " << *x << endl;
}
```

- The srl function performs a logical right shift on the value in the y register by the number of positions specified by the least significant five bits of the z register. The result is stored in the x register. This operation fills the leftmost bits with zeros, which is useful for unsigned numbers.

- **Srli (shift right logical immediate)**

```
void srli(int* x, int* y, int z)
{
    int temp = z & 0b11111;
    *x = static_cast<uint32_t>(*y) >> temp;
}
```

- The srli function is similar to srl, but the shift amount is specified as an immediate value (integer) z rather than a register value, offering more flexibility and faster execution in certain situations.

- **Sll (shift left logical)**

```
void sll(int* x, int* y, int* z)
{
    int temp = *z & 0b11111;
    *x = *y << temp;
}
```

- The sll function performs a logical left shift on the value in the y register by the number of positions specified by the least significant five bits of the z register. The result is stored in the x register. This operation is useful for multiplication and division by powers of two.

- **Slli (shift left logical immediate)**

```

void slli(int* x, int* y, int z)
{
    int temp = z & 0b11111;
    *x = *y << temp;
}

```

- The slli function is similar to sll, but the shift amount is specified as an immediate value (integer) z rather than a register value, offering more flexibility and faster execution in certain situations.

## B. Set less than Functions

### • Slt (set less than)

```

void slt(int* x, int* y, int* z)
{
    if (*y < *z)
        *x = 1;
    else
        *x = 0;
}

```

- The slt function compares the values in the y and z registers and sets the x register to 1 if y is less than z, and 0 otherwise. This operation helps in conditional branching and decision-making.

### • Slti (set less than immediate)

```

void slti(int* x, int* y, int z)
{
    if (*y < z)
        *x = 1;
    else
        *x = 0;
}

```

- The slti function is similar to slt, but the comparison is made between the value in the y register and an immediate integer value z. This offers more flexibility and faster execution in certain situations.

- **Sltu (set less than unsigned)**

```
void sltu(int* x, int* y, int* z)
{
    uint32_t unsigned_y = static_cast<uint32_t>(*y);
    uint32_t unsigned_z = static_cast<uint32_t>(*z);

    if (unsigned_y < unsigned_z)
    {
        *x = 1;
    }
    else
    {
        *x = 0;
    }
}
```

- The sltu function is an unsigned version of slt. It compares the unsigned values in the y and z registers. It sets the x register to 1 if the unsigned y is less than the unsigned z, and 0 otherwise.

- **Sltiu (set less than immediate unsigned)**

```
void sltiu(int* x, int* y, int z)
{
    uint32_t unsigned_y = static_cast<uint32_t>(*y);
    uint32_t unsigned_z = static_cast<uint32_t>(z);

    if (unsigned_y < unsigned_z)
    {
        *x = 1;
    }
    else
    {
        *x = 0;
    }
}
```

- The sltiu function is an unsigned version of slti. It compares the unsigned value in the y register with an immediate unsigned

integer value z. It sets the x register to 1 if the unsigned y is less than the unsigned z and 0 otherwise.

### C. Load functions

- **Load functions assume a memory simulation in an array called `memory[]`. Whenever we want to access the array, call the address stored in pointer `m`.**

- **Lw (load word)**

```
void lw(int* rd, int* rs1, int offset) {  
    if (offset % 4 != 0) {  
        cout << "Error: offset not valid\n";  
        return;  
    }  
    int value = *(rs1 + (offset / 4));  
    *rd = value;  
}
```

- The lw function loads a word (four bytes) from the memory location specified by the sum of the rs1 register and the offset. It then stores this word in the rd register. This operation is critical for accessing data stored in memory.

- **Lb (load byte)**

```
void lb(int* rd, int* rs1, int offset) {  
    int byteIndex = offset % 4;  
    int wordIndex = offset / 4;  
    int8_t byte = *(rs1 + wordIndex) & 0xFF;  
    *rd = static_cast<int32_t>(byte);  
}
```

- The lb function loads a byte from the memory location specified by the sum of the rs1 register and the offset. It then sign-extends this byte to a 32-bit word and stores it in the rd register.

- **Lh (load halfword)**

```
void lh(int* rd, int* rs1, int offset) {  
    if (offset % 2 != 0) {  
        cout << "Error: offset not valid\n";  
        return;  
    }  
    int byteIndex = offset % 4;  
    int wordIndex = offset / 4;  
    int16_t half = *(rs1 + wordIndex) & 0xFFFF;  
    *rd = static_cast<int32_t>(half);  
}
```

- The lh function loads a halfword (two bytes) from the memory location specified by the sum of the rs1 register and the offset. It then sign-extends this halfword to a 32-bit word and stores it in the rd register.

- **Lbu (load byte unsigned)**

```
void lbu(int* rd, int* rs1, int offset) {  
    int byteIndex = offset % 4;  
    int wordIndex = offset / 4;  
    uint8_t byte = *(rs1 + wordIndex) & 0xFF;  
    *rd = static_cast<uint32_t>(byte);  
}
```

- The lbu function is similar to lb, but it zero-extends the loaded byte instead of sign-extending it.

- **Lhu (load halfword unsigned)**

```
void lhu(int* rd, int* rs1, int offset) {  
    if (offset % 2 != 0) {  
        cout << "Error: offset not valid\n";  
        return;  
    }  
    int byteIndex = offset % 4;  
    int wordIndex = offset / 4;
```



```

uint16_t half = *(rs1 + wordIndex) & 0xFFFF;
*rd = static_cast<uint32_t>(half);
}

```

- The lhu function is similar to lh, but it zero-extends the loaded halfword instead of sign-extending it.

- **Li (load immediate)**

```

void li(int* x, int y)
{
    *x = y;
}

```

- The li function loads an immediate integer value y into the x register. This operation is a quick way to store a constant value in a register.

- **Lui (load upper immediate)**

```

void LUI(int* x, int y) {
    *x = y << 12;
}

```

- The LUI function loads an immediate value y into the upper 20 bits of the x register, while the lower 12 bits are filled with zeros. This operation is useful when dealing with large constants or addresses.

- **Auipc (add upper immediate to pc)**

```

void AUIPC(int* x, int y, int z) {
    *x = z + (y << 12);
}

```

- The AUIPC function adds an immediate value y, left-shifted by 12 bits, to the program counter z. The result is stored in the x register.

This operation is useful for building large constants or addresses at runtime.

## D. Store Functions

- **Sw (store word)**

```
void sw(int* rs1, int* rd, int offset) {  
    if (offset % 4 != 0) {  
        cout << "Error: offset not valid\n";  
        return;  
    }  
    *(rs1 + (offset / 4)) = *rd;  
}
```

- The sw function stores a word from the rd register to the memory location specified by the sum of the rs1 register and the offset. This operation is used to store data back to memory after processing.

- **Sb (store byte)**

```
void sb(int* rs1, int* rd, int offset) {  
    int byteIndex = offset % 4;  
    int wordIndex = offset / 4;  
    int8_t byte = *rd & 0xFF;  
    char* bytePtr = (char*)(rs1 + wordIndex);  
    bytePtr[byteIndex] = byte;  
}
```

- The sb function stores a byte from the rd register to the memory location specified by the sum of the rs1 register and the offset. This operation is useful for storing byte-sized results back to memory.

- **Sh (store halfword)**

```
void sh(int* rs1, int* rd, int offset) {  
    if (offset % 2 != 0) {
```

```

        cout << "Error: offset not valid\n";
        return;
    }
    int byteIndex = offset % 4;
    int wordIndex = offset / 4;
    int16_t half = *rd & 0xFFFF;
    char* bytePtr = (char*)(rs1 + wordIndex);
    bytePtr[byteIndex] = half & 0xFF;
    bytePtr[byteIndex + 1] = (half >> 8) & 0xFF;
}

```

- The sh function stores a halfword from the rd register to the memory location specified by the sum of the rs1 register and the offset. This operation is useful for storing two-byte results back to memory.

## E. Utility Functions

### · Get\_offset

```

void get_offset(string& rs1Temp, int& offset) {
    int imm_start = rs1Temp.find("(");
    int imm_end = rs1Temp.find(")");
    offset = stoi(rs1Temp.substr(0, imm_start));
    rs1Temp = rs1Temp.substr(imm_start + 1, imm_end - imm_start -
1);
}

```

- The get\_offset function extracts the offset value from a given string rs1Temp. The offset is identified as the substring between

the characters "(" and ")." It then converts this substring into an integer and stores it in the variable offset. The string rs1Temp is updated to contain only the remaining part after extraction. This utility function aids in parsing complex instructions where offsets are embedded within strings.

F. R-Format:

- Add

```
void add(int* x, int* y, int* z) {  
  
    *x = *y + *z;  
  
}
```

The add function stores the value of the pointers passed by references rs1 plus rs2 in rd.

- Sub

```
void sub(int* x, int* y, int* z) {  
  
    *x = *y - *z;  
  
}
```

The sub function stores the value of the pointers passed by references rs1 minus rs2 in rd.

- Mul

```
void mul(int* x, int* y, int* z) {  
  
    *x = *y * *z;  
  
}
```

The mul function stores the value of the pointers passed by references rs1 times rs2 in rd.

- And

```
void AND(int* x, int* y, int* z) {  
  
    *x = *y & *z;  
  
}
```

The and function does “and” bit wise operation for the pointers passed by references rs1 and rs2 and stores the result in rd.

- Or

```
void OR(int* x, int* y, int* z) {  
  
    *x = *y | *z;  
  
}
```

The or function does “or” bit wise operation for the pointers passed by references rs1 and rs2 and stores the result in rd.

- Xor

```
void XOR(int* x, int* y, int* z) {  
  
    *x = *y ^ *z;  
  
}
```

The xor function does “xor” bit wise operation for the pointers passed by references rs1 and rs2 and stores the result in rd.

G. I-Format:

- Addi

```
void addi(int* x, int* y, int z) {  
  
    *x = *y + z;  
  
}
```

The addi function stores the value of the pointer passed by references rs1 plus the immediate value inputted by the user in rd.

- Muli

```
void muli(int* x, int* y, int z) {  
  
    *x = *y * z;  
  
}
```

The muli function stores the value of the pointer passed by references rs1 times the immediate value inputted by the user in rd.

- Andi

```
void ANDI(int* x, int* y, int z) {
```

```
    *x = *y & z;
```

```
}
```

The andi function does “and” bit wise operation for the pointer passed by references rs1 and the immediate value inputted by the user and stores the result in rd.

- Ori

```
void ORI(int* x, int* y, int z) {
```

```
    *x = *y | z;
```

```
}
```

The ori function does “or” bit wise operation for the pointer passed by references rs1 and the immediate value inputted by the user and stores the result in rd.

- Xori

```
void XORI(int* x, int* y, int z) {
```

```
    *x = *y ^ z;
```

```
}
```

The xori function does “xor” bit wise operation for the pointer passed by references rs1 and the immediate value inputted by the user and stores the result in rd.

H. B-Format:

All the B-Format functions have an if statement in main to check whether they return true or not.

If they do return true, there will be a use of 2 functions: .clear() and .seekg(0) to return the indicator of the text file to its very start. A temporary string is being used to go over searching the text file till it reaches the label already stored in another string.

- Beq (Branch if equal)

```
bool beq(int* x, int* y) {  
  
    if (*x == *y)  
  
        return true;  
  
    else  
  
        return false;  
  
}
```

A boolean function returns true if the pointers passed by reference rd and rs1 are equal and false otherwise.

- Bne (Branch if not equal)



```

bool bne(int* x, int* y) {

    if (*x != *y)

        return true;

    else

        return false;

}

```

A boolean function returns true if the pointers passed by reference rd and rs1 are not equal and false otherwise.

- Blt (Branch if less than)

```

bool blt(int* x, int* y) {

    if (*x < *y)

        return true;

    else

        return false;

}

```

A boolean function returns true for the pointers passed by reference if rd is less than rs1.

- Bge (Branch if greater than or equal)

```

bool bge(int* x, int* y) {

    if ((*x > *y) || (*x == *y))

        return true;

    else

        return false;

}

```

A boolean function returns true for the pointers passed by reference if rd is greater than or equal rs1.

- Bltu (Branch if less than unsigned)

```

bool bltu(int* x, int* y) {

    if (abs(*x) < abs(*y))

        return true;

    else

        return false;

}

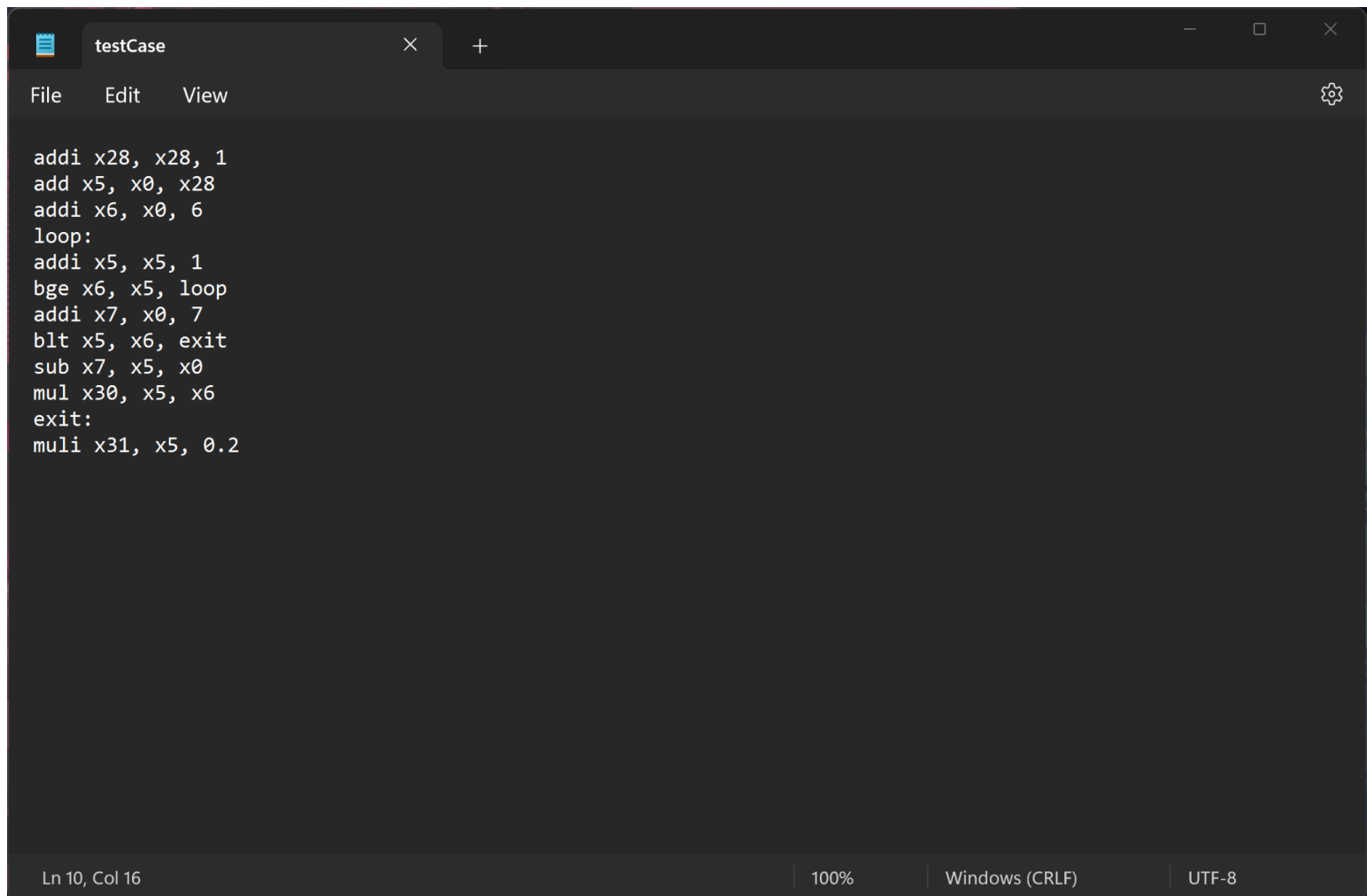
```

A boolean function returns true for the pointers passed by reference if the absolute value of rd is less than the absolute value of rs1.

- Bgeu (Branch if greater than or equal unsigned)

```
bool bgeu(int* x, int* y) {  
  
    if ((abs(*x) > abs(*y)) || (abs(*x) == abs(*y)))  
  
        return true;  
  
    else  
  
        return false;  
  
}
```

A boolean function returns true for the pointers passed by reference if the absolute value of rd is greater than or equal to the absolute value of rs1.



```
addi x28, x28, 1  
add x5, x0, x28  
addi x6, x0, 6  
loop:  
addi x5, x5, 1  
bge x6, x5, loop  
addi x7, x0, 7  
blt x5, x6, exit  
sub x7, x5, x0  
mul x30, x5, x6  
exit:  
mul x31, x5, 0.2
```

Ln 10, Col 16 | 100% | Windows (CRLF) | UTF-8

C:\Users\moham\source\repos\ConsoleApplication1\x64\Debug\ConsoleApplication1.exe

Registers before implementing the instructions:

```
x0 =  
Decimal: 0  
Binary: 00000000000000000000000000000000  
Hexadecimal: 0  
x5 is not yet initialized.  
x6 is not yet initialized.  
x7 is not yet initialized.  
x28 is not yet initialized.  
x29 is not yet initialized.  
x30 is not yet initialized.  
x31 is not yet initialized.
```

Registers after implementing the instructions:

```
x0 =  
Decimal: 0  
Binary: 00000000000000000000000000000000  
Hexadecimal: 0  
x5 =  
Decimal: 7  
Binary: 00000000000000000000000000000011  
Hexadecimal: 7  
x6 =  
Decimal: 6  
Binary: 00000000000000000000000000000010  
Hexadecimal: 6  
x7 =  
Decimal: 1  
Binary: 00000000000000000000000000000001  
Hexadecimal: 1  
x28 =  
Decimal: 1  
Binary: 00000000000000000000000000000001  
Hexadecimal: 1  
x29 is not yet initialized.  
x30 =  
Decimal: 42  
Binary: 0000000000000000000000000000101010  
Hexadecimal: 2a  
x31 =  
Decimal: 14  
Binary: 000000000000000000000000000001110  
Hexadecimal: e
```

Press any key to continue . . .

```
lui x5, 1775856045  
auipc x6, 8  
li x7, 15  
slt x28, x6, x7  
slti x29, x7, 20  
sra x30, x5, x29  
srai x31, x5, 5
```

-

Registers before implementing the instructions:

x0 =

Decimal: 0

Binary: 00000000000000000000000000000000

Hexadecimal: 0

x5 is not yet initialized.

x6 is not yet initialized.

x7 is not yet initialized.

x28 is not yet initialized.

x29 is not yet initialized.

x30 is not yet initialized.

x31 is not yet initialized.

Registers after implementing the instructions:

x0 =

Decimal: 0

Binary: 00000000000000000000000000000000

Hexadecimal: 0

x5 =

Decimal: -1768239104

Binary: 10010110100110101101000000000000

Hexadecimal: 969ad000

x6 =

Decimal: 32769

Binary: 00000000000000000100000000000001

Hexadecimal: 8001

x7 =

Decimal: 15

Binary: 00000000000000000000000000001111

Hexadecimal: f

x28 =

Decimal: 0

Binary: 00000000000000000000000000000000

Hexadecimal: 0

x29 =

Decimal: 1

Binary: 00000000000000000000000000000001

Hexadecimal: 1

x30 =

Decimal: -884119552

Binary: 11001011010011010110100000000000

Hexadecimal: cb4d6800

x31 =

Decimal: -55257472

Binary: 11111100101101001101011010000000

Hexadecimal: fcb4d680

```
li x5, 1775856045
li x6, 5
srl x7, x5, x6
srli x28, x5, 5
li x29, -20
sltu x30, x29,x6|
```

-

Registers before implementing the instructions:

x0 =

Decimal: 0

Binary: 00000000000000000000000000000000

Hexadecimal: 0

x5 is not yet initialized.

x6 is not yet initialized.

x7 is not yet initialized.

x28 is not yet initialized.

x29 is not yet initialized.

x30 is not yet initialized.

x31 is not yet initialized.

in srl x = 55495501

Registers after implementing the instructions:

x0 =

Decimal: 0

Binary: 00000000000000000000000000000000

Hexadecimal: 0

x5 =

Decimal: 1775856045

Binary: 01101001110110010110100110101101

Hexadecimal: 69d969ad

x6 =

Decimal: 5

Binary: 00000000000000000000000000000101

Hexadecimal: 5

x7 =

Decimal: 55495501

Binary: 00000011010011101100101101001101

Hexadecimal: 34ecb4d

x28 =

Decimal: 55495501

Binary: 00000011010011101100101101001101

Hexadecimal: 34ecb4d

x29 =

Decimal: -20

Binary: 111111111111111111111111111101100

Hexadecimal: fffffffec

x30 =

Decimal: 0

Binary: 00000000000000000000000000000000

Hexadecimal: 0

x31 is not yet initialized.

```
li x5, 1775856045
li x6, 5
add x7, x0, x6
sw x6, 0(x7)
lw x28, 0(x7)
```

x29 is not yet initialized.  
x30 is not yet initialized.  
x31 is not yet initialized.

Registers after implementing the instructions:

x0 =

Decimal: 0

Binary: 00000000000000000000000000000000

Hexadecimal: 0

x5 =

Decimal: 1775856045

Binary: 01101001110110010110100110101101

Hexadecimal: 69d969ad

x6 =

Decimal: 5

Binary: 00000000000000000000000000000101

Hexadecimal: 5

x7 =

Decimal: 5

Binary: 00000000000000000000000000000101

Hexadecimal: 5

x28 =

Decimal: 5

Binary: 00000000000000000000000000000101

Hexadecimal: 5

x29 is not yet initialized.

x30 is not yet initialized.

x31 is not yet initialized.



```
li x5, 1775856045
li x6, 5
add x7, x0, x5
sw x5, 0(x7)
lh x28, 0(x7)
lb x30, 0(x7)
```

Registers after implementing the instructions:

x0 =

Decimal: 0

Binary: 00000000000000000000000000000000

Hexadecimal: 0

x5 =

Decimal: 1775856045

Binary: 01101001110110010110100110101101

Hexadecimal: 69d969ad

x6 =

Decimal: 5

Binary: 00000000000000000000000000000101

Hexadecimal: 5

x7 =

Decimal: 1775856045

Binary: 01101001110110010110100110101101

Hexadecimal: 69d969ad

x28 =

Decimal: 27053

Binary: 00000000000000000110100110101101

Hexadecimal: 69ad

x29 is not yet initialized.

x30 =

Decimal: -83

Binary: 1111111111111111111111110101101

Hexadecimal: fffffffad

x31 is not yet initialized.

---

```
li x5, 1775856045
li x6, 5
add x7, x0, m
sw x5, 0(x7)
lhu x28, 0(x7)
lbu x30, 0(x7)
```

-

Registers before implementing the instructions:

x0 =

Decimal: 0

Binary: 00000000000000000000000000000000

Hexadecimal: 0

x5 is not yet initialized.

x6 is not yet initialized.

x7 is not yet initialized.

x28 is not yet initialized.

x29 is not yet initialized.

x30 is not yet initialized.

x31 is not yet initialized.

Registers after implementing the instructions:

x0 =

Decimal: 0

Binary: 00000000000000000000000000000000

Hexadecimal: 0

x5 =

Decimal: 1775856045

Binary: 01101001110110010110100110101101

Hexadecimal: 69d969ad

x6 =

Decimal: 5

Binary: 00000000000000000000000000000101

Hexadecimal: 5

x7 =

Decimal: 1775856045

Binary: 01101001110110010110100110101101

Hexadecimal: 69d969ad

x28 =

Decimal: 27053

Binary: 00000000000000000110100110101101

Hexadecimal: 69ad

x29 is not yet initialized.

x30 =

Decimal: 173

Binary: 0000000000000000000000010101101

Hexadecimal: ad

x31 is not yet initialized.

```
li x5, 1775856045
add x7, x0, m
sh x5, 0(x7)
lw x28, 0(x7)|
```

```
x6 is not yet initialized.
x7 is not yet initialized.
x28 is not yet initialized.
x29 is not yet initialized.
x30 is not yet initialized.
x31 is not yet initialized.
```

Registers after implementing the instructions:

```
x0 =
Decimal: 0
Binary: 00000000000000000000000000000000
Hexadecimal: 0
x5 =
Decimal: 1775856045
Binary: 01101001110110010110100110101101
Hexadecimal: 69d969ad
x6 is not yet initialized.
x7 =
Decimal: 27053
Binary: 000000000000000000110100110101101
Hexadecimal: 69ad
x28 =
Decimal: 27053
Binary: 000000000000000000110100110101101
Hexadecimal: 69ad
x29 is not yet initialized.
x30 is not yet initialized.
x31 is not yet initialized.
```

### III. User Guide

- **You should run the source code with a compatible IDE for C++ and make sure that there is a text document file named “testCase” in the project’s directory.**
- **“testCase” is the place for writing the RISC-V instructions so enter the instructions separated by one line between each instruction**