MIPS ISA Processor Design

Course Name: EG 212, Computer Architecture - Processor Design

Aishwarya Sharma IMT2023515

Satyam Ambi IMT2023623

Hithanshu Seth IMT2023100

Introduction :

The MIPS (Microprocessor without Interlocked Pipeline Stages) architecture is a Reduced Instruction Set Computer (RISC) architecture that was developed by MIPS Computer Systems, which was later acquired by Imagination Technologies. It was first introduced in 1981 and has since been used in a wide range of applications, from embedded systems to supercomputers. The MIPS architecture is known for its simplicity, which allows for high performance and low power consumption.

0.1 Instruction Set Architecture (ISA):

The **MIPS ISA** is based on a fixed-length, 32-bit instruction format.

- Instructions are divided into three types: **R-type, I-type, and J-type.**
  - R-type instructions are used for arithmetic and logical operations.
  - I-type instructions are used for data transfer and immediate operations.
  - J-type instructions are used for jump and branch operations.
- The ISA includes a set of 32 general-purpose registers, labeled $0 to 31$, where \$0 is hardwired to zero and \$31 is used as the return addresses for function calls.

- The ISA also includes a set of floating-point registers, labeled $f0 to $f31, which are used for floating-point operations.
- The MIPS ISA supports both little-endian and big-endian byte orderings.

0.2 Memory Organization:

- The MIPS architecture uses a flat, linear memory model.
- Memory is byte-addressable, and the address space is typically 32 bits or 64 bits.
- The MIPS architecture supports virtual memory, which allows programs to use more memory than is physically available.

0.3 Pipeline Stages:

- The MIPS architecture uses a five-stage pipeline, which consists of the following stages:
  - Instruction Fetch (IF)
  - Instruction Decode (ID)
  - Execute (EX)
  - Memory Access (MEM)
  - Write Back (WB)
- The pipeline allows multiple instructions to be executed simultaneously, which improves performance.

0.4 Privileged Instructions:

- The MIPS ISA includes a set of privileged instructions for operating system and kernel-level operations.
- These instructions are used to control hardware resources, such as the interrupt controller, timer, and memory management unit (MMU).

0.5 Multi-threading and Multi-processing:

- The MIPS ISA includes support for multi-threading and multi-processing.
- Multi-threading allows multiple threads to execute concurrently on a single processor core.
- Multi-processing allows multiple processor cores to execute concurrently, which improves performance.

0.6 Exception Handling:

- The MIPS architecture includes support for handling exceptions, such as interrupts, system calls, and program errors.
- When an exception occurs, the processor switches to kernel mode and executes a predefined exception handler.

0.7 Performance and Power Consumption:

- The MIPS architecture is designed to be simple, efficient, and easy to implement, which allows for high performance and low power consumption.
- The architecture is optimized for pipelining, which improves performance by allowing multiple instructions to be executed simultaneously.
- The architecture is also optimized for low power consumption, which makes it suitable for battery-powered devices.
- The architecture is also optimized for low power consumption, which makes it suitable for battery-powered devices.

Overall, the MIPS architecture is a versatile and efficient architecture that has been widely used in a variety of applications. It is known for its simplicity, high performance, and low power consumption, which makes it a popular choice for a wide range of applications, from embedded systems to supercomputers.

This report presents our work on the EG 212, Computer Architecture Assignment 2 on MIPS ISA processor design.

The assignment consists three major tasks:

- Writing Assembly programs for three C++ Codes.

- Assemble the code and implement in MARS MIPS Simulator.
- Design of Single Non-Pipelined Processor.

Following will be the C++ codes alog with thier Assembly Codes , Single Lined Non-Pipelined Processor and the corresponding results.

Original C++ Codes along with their assembly codes :

---

1) Number of Permutations :

The Code takes two numbers (n and r) and gives the total number of permutations.

$$\begin{aligned} nPr &= \frac{n!}{(n-r)!} \\ &= \frac{n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-r+1)}{1 \cdot 2 \cdot 3 \cdot \ldots \cdot r} \end{aligned}$$

Original C++ Code :

```cpp
#include <bits/stdc++.h>
using namespace std;

int factorial(int n){
    int f=1;
    while(n>0){
        f = f * n;
        n--;
    }
    return f;
}
//"Finding the Permutation."
int main(){
    int n,r;
    cout<<"Enter the value of n : ";
    cin>>n;
    cout<<"Enter the value of r : ";
    cin>>r;
    if(n<r){
        cout<<"Permutation cannot be found."<<endl;
    }
    else{
        int permutation = (factorial(n))/(factorial(n-r));
        cout<<"The permutation for entered n & r is: "<<permutation<<endl;
    }
    return 0;
}
```

Assembly Code :

Equivalent to the above C++ code we will write a assembly code in MIPS ISA format.The assembly code contains two blocks ”.data” and ”.text” .

```
.data
.text
main:
    #"Enter the value of n: "
    lw $t3,0($zero)
    #"Enter the value of r: "
    lw $t4,1($zero)
    addi $t1,$zero,1
    # Checking condition : (n < 0)
    slt $s0,$t3,$zero
    beq $s0,$t1,not_applicable
    # Checking condition : (n = r)
    beq $t3, $t4, calculate_permutation
    # Checking condition : (n < r)
    slt $t0, $t3, $t4
    beq $t0, $zero, calculate_permutation
    j not_applicable
calculate_permutation:
    # Calculating factorial(n)
    factorial1:  #For n!
        addi $t8,$zero,1
        addi $t6,$zero,1
        add $t7,$t3,$zero   # Load Input in $t3 into $t7 , substitute of
   move.
        loop1:
            beq $t7, $zero, loop_end1
            mult $t6, $t7
            mflo $t6
            sub $t7,$t7,$t8
            j loop1
        loop_end1:
            add $s2,$t6,$zero   # Return the result , substitute for move
    # Calculating factorial(n-r)
    sub $s5,$t3,$t4
    factorial2: #For (n-r)!
        addi $t5,$zero,1
        addi $t9,$zero,1
        add $s7,$s5,$zero   # Load Input in $t4 into $s7 , substitute of
   move.
    ...CONTINUE...
not_applicable:
    addi $s6,$zero,-1   #FINAL RESULT of nPr is in $s6
    sw $s6,2($zero)
    j end
end:
    addi $v0,$zero,10
    syscall
```

Find the full code in the file - mips1_Permutation_final.asm

2) Palindrome Checker :

The Code takes an integer and tells that the entered number is **PALINDROME** or **NOT A PALINDROME**.

Original C++ Code :

```cpp
//Only works for POSITIVE NUMBERS.

#include <bits/stdc++.h>
using namespace std;

int isPalindrome(int num) {
    int originalNum = num;
    int reversedNum = 0;
    while (num > 0) {
        int digit = num % 10;
        reversedNum = reversedNum * 10 + digit;
        num /= 10;
    }
    if(originalNum==reversedNum){
        return 1;
    }
    else{
        return 0;
    }
}
int main() {
    int n;
    cout << "Enter the number: ";
    cin >> n;
    if(n<0){
        cout<<"PALINDROME CAN'T BE CHECKRD FOR NEGATIVE NUMBERS."<<endl;
    }
    if (isPalindrome(n)==1) {
        cout << "PALINDROME NUMBER." << endl;
    }
    else {
        cout << "NOT A PALINDROME NUMBER." << endl;
    }
    return 0;
}
```

Assembly Code :

Equivalent to the above C++ code we will write a assembly code in MIPS ISA format. The assembly code contains two blocks ".data" and ".text" .

```
.data
.text
main:
    #Enter the value of n:
    lw $t3,0($zero)
    # Checking condition : (n < 0)
    addi $t6,$zero,1
    addi $t8,$zero,0
    slt $t0, $t3,$zero
    beq $t0,$t6,negative
    add $a0,$t3,$zero
    j Checking_Palindrome
Checking_Palindrome:
    add $s3,$a0,$zero #originalnum
    add $t9,$a0,$zero #copy_originalnum
    addi $s4,$zero,0 #reversenum
    addi $s5,$zero,10
    loop:
        beq $t9 ,$zero ,loop_end
        div $t9 , $s5
        mfhi $t7
        mult $s4,$s5
        mflo $s6
        add $s4,$s6,$t7
        div $t9,$s5
        mflo $t9
        j loop
    loop_end:
        beq $s3,$s4,true
        addi $s6,$zero,0
        j end
negative:
    addi $s6,$zero,-1
    sw $s6,1($zero)
    j end
true:
    addi $s6,$zero,1
    sw $s6,1($zero)
    j end
end:
    addi $v0,$zero,10
    syscall
```

3) Checking Twin Primes :

The Code takes two positive number and tells that the entered numbers are **TWINPRIME** or **NOT TWINPRIME**.

Original C++ Code :

```
//Only works for POSITIVE NUMBERS.
#include <bits/stdc++.h>
using namespace std;

int prime(int x){
    int c=0;
    int i=2;
    while(i<x){
        if(x%i==0){
            c++;
        }
        i++;
    }
    return c;
}

int main(){
    int n1,n2,k;
    cout<<"Enter the first number: ";
    cin>>n1;
    cout<<"Enter the second number: ";
    cin>>n2;
    if(n1-n2>0){
        k = n1-n2;
    }
    else if(n1-n2<0){
        k = n2-n1;
    }
    if(prime(n1)==0 and prime(n2)==0 and k==2){
        cout<<"The two numbers are prime."<<endl;
    }
    else{
        cout<<"The two numbers are not prime."<<endl;
    }

}
```

Assembly Code :

Equivalent to the above C++ code we will write a assembly code in MIPS ISA format. The assembly code contains two blocks ".data" and ".text" .

```
.data
.text
    #Input the first number n1 and store it in $t0.
    lw $t0,0($zero)
    #Input the second number n2 and store it in $t1.
    lw $t1,1($zero)
    check:
      beq $t0,1,end1
      beq $t1,1,end1
    check2:
      slt $t4,$t0,$t1
      beq $t4,1,swap
      j continue
    swap:
      add $t3,$zero,$t0
      add $t0,$zero,$t1
      add $t1,$zero,$t3
    continue:
      sub $t6,$t0,$t1
    prime1:
        addi $t7,$zero,0
        addi $t9,$zero,2 #value of 'i'
        add $s6,$t0,$zero
        loop1:
            beq $t9,$s6,loop_end1
            div $s6,$t9
            mfhi $t2 #Storing remainder in $t2
            beq $t2,$zero,increment_c1
            addi $t9,$t9,1
            j loop1
    ...CONTINUE...
    beq $s2,$zero,true1
    j false
...CONTINUE...
end1:
    addi $s7,$zero,1
    j end
end:
    addi $v0,$zero,10
    syscall
```

Find the full code in the file - mips3_Twinprime_final.asm

Memory Data :

The below is the data stored in memory on which the processor will operate.This gets genrateed from MARS when we will dump it to **Binary Text**.

```
10001100000010110000000000000000
00100000000011100000000000000001
00100000000011000000000000000000
00000001011000000100000000101010
00010001000011100000000000010010
00000001011000000010000000100000
00001000000000000000000000000111
00000000100000001001100000100000
00000000100000001100100000100000
00100000000101000000000000000000
00100000000101010000000000001010
00010011001000000000000000001000
00000011001101010000000000011010
00000000000000000011110000010000
00000010100101010000000000011000
00000000000000000101100000010010
00000010110011111010000000100000
00000011001101010000000000011010
00000000000000000110010000010010
00001000000000000000000000001011
00010010011101000000000000000101
00100000000101100000000000000000
00001000000000000000000000011100
00100000000101101111111111111111
10101100000101100000000000000001
00001000000000000000000000011100
00100000000101100000000000000001
00001000000000000000000000011100
00100000000000100000000000001010
00000000000000000000000000001100
```

Processor :

Processor reads the `Memory_<codename>.txt` file and accordingly identify the formats to R,I or J and then decode the **OPCODE** and according to the decoded **OPCODE** follow the 5-Stages.

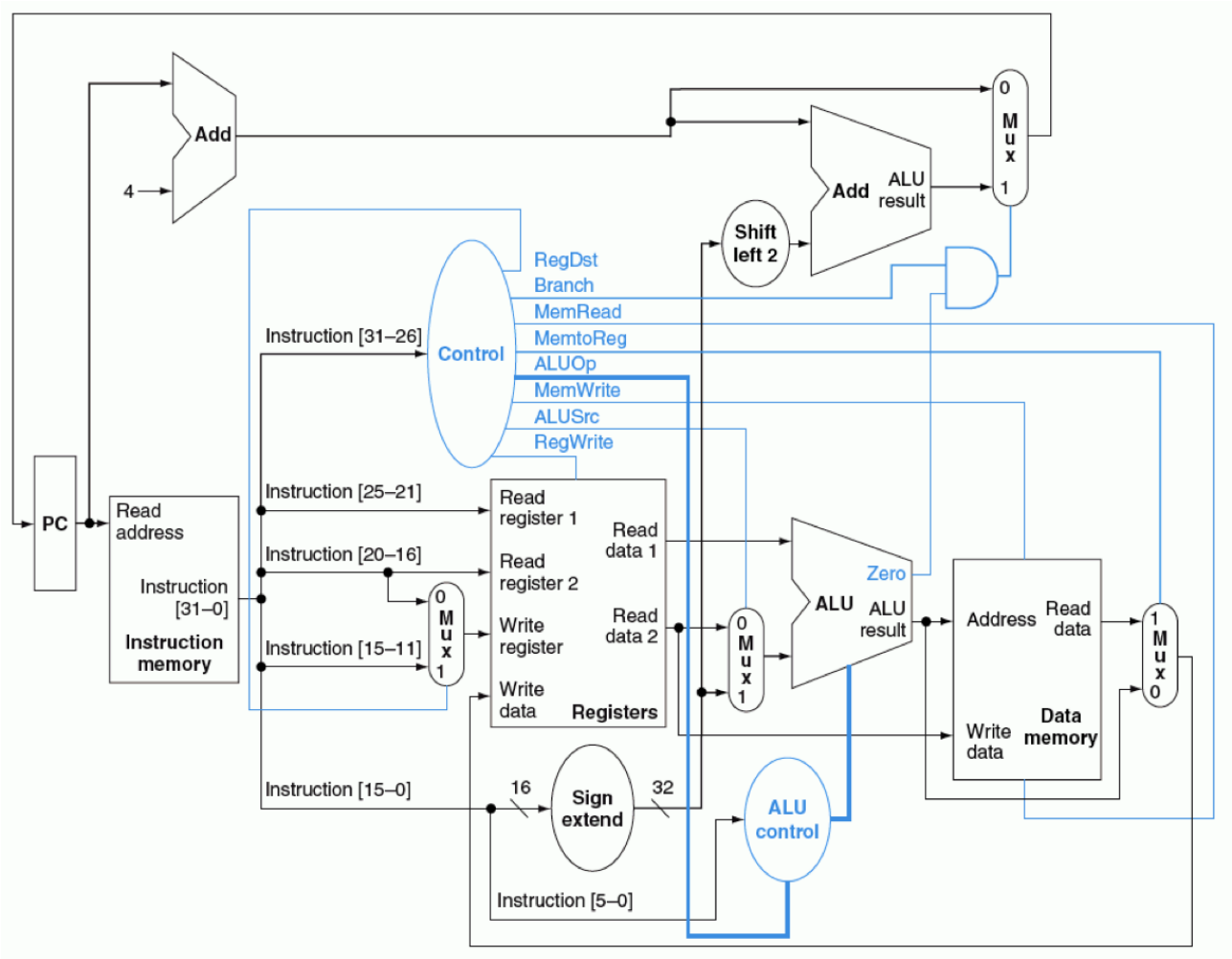

Figure 1: Control Signals and Data Path

Processor Code :

```cpp
#include <iostream>
#include <unordered_map>
using namespace std;
//defining functions
void checkRFormatFields(uint32_t instruction);
void checkIFormatFields(uint32_t instruction);
uint32_t extractJumpTarget(uint32_t instruction);
void executeRFormatInstruction(uint32_t instruction);
//...CONTINUE...
void printRegisters() { //for printing register values if needeD.
    cout << "Registers:" << endl;
    for (int i = 0; i < 32; ++i) {
        cout << "$" << i << ": " << registerFile[i] << endl;
    }
    cout << "HI: " << HI << endl;
    cout << "LO: " << LO << endl;
    cout << endl;
}
struct ControlSignals { // Define control signals struct
    bool RegDst;    // Selects destination register number
    bool RegWrite;  // Enable write to register
    bool MemRead;   // Read data from memory
    bool MemWrite;  // Write data to memory

//...CONTINUE...
int main() {
    uint32_t instructions1[29] = {0b00100000000001011111101001011111,
        0b00100000000011100000000000000001,
        0b00100000000110000000000000000000,
        0b00000001011000000100000000101010,
        0b00010001000011100000000000010010,
    //...CONTINUE...
    int choice; // User choice
    cout << "Select an option:" << endl; // Prompt user for choice
    cout << "1. Palindrome Checker" << endl;
    cout << "2. Number of permutations" << endl;
    cout << "3. Checking Twin Primes" << endl;
    cout << "Enter your choice (1, 2, or 3): ";
    cin >> choice;
    int numInstructions;
    switch (choice) { // Execute instructions based on user choice
        case 1:
            numInstructions = sizeof(instructions1) / sizeof(instructions1
    [0]); // Total number of instructions in the instruction memory.
            cout << "Checking if a number is palindrome or not:\n" << endl
    ;
     //...CONTINUE...
    return 0;
}
```

Find the full code in the file - Non-Pipelined_Processor.c++.

Execution Results :

The below screeshots shows how processor is executed.

NOTE : The Results are for 2nd code Palindrome Checker.

```
aish@aish-Lenovo-IdeaPad-S340-14IIL:~$ g++ Non-Pipelined_Processor.c++
aish@aish-Lenovo-IdeaPad-S340-14IIL:~$ ./a.out
Select an option:
1. Palindrome Checker
2. Number of permutations
3. Checking Twin Primes
Enter your choice (1, 2, or 3): 3
Calculating if two numbers are twin prime or not:

Give a number:
5
Give an another number:
3
Instruction 1:
Instruction type: I-format load (lw)
Source Register (RS): 0
Target Register (RT): 8
Immediate (signed): 0
lw: $8 = mem[$0 + 0]
mem[$0 + 0] =5
PC:0

Instruction 2:
Instruction type: I-format load (lw)
Source Register (RS): 0
Target Register (RT): 9
Immediate (signed): 1
lw: $9 = mem[$0 + 1]
mem[$0 + 1] =3
PC:4

Instruction 3:
Instruction type: I-format add immediate (addi)
Source Register (RS): 0
Target Register (RT): 1
Immediate (signed): 1
ADDI: $1 = $0 + 1
PC:8

Instruction 4:
Instruction type: I-format branch equal (beq)
Source Register (RS): 1
Target Register (RT): 8
Immediate (signed): 49
Branch not taken (registers $1 and $8 are not equal)
```

Figure 2: Result

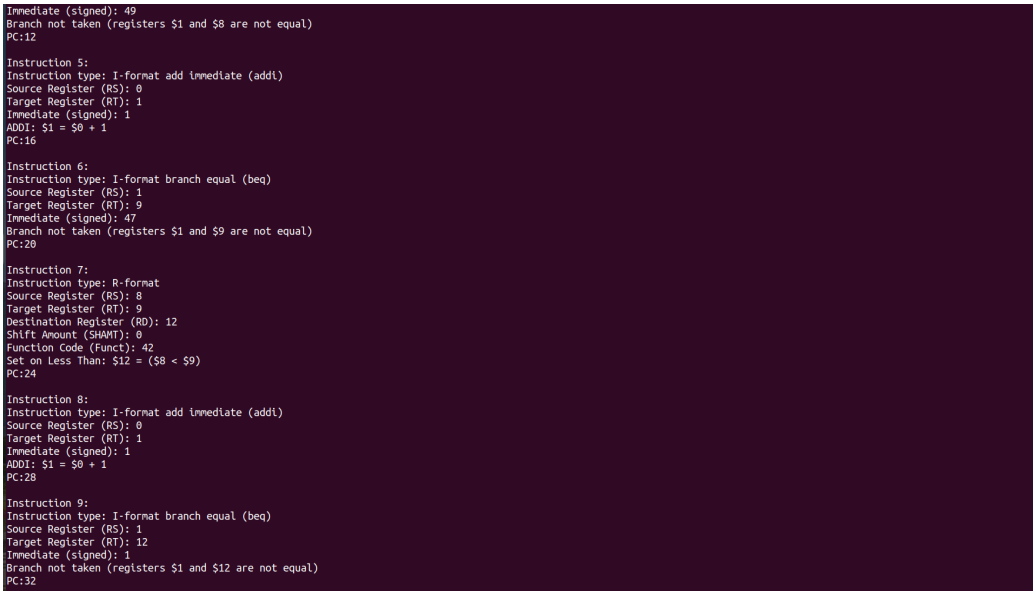

Figure 3: Result

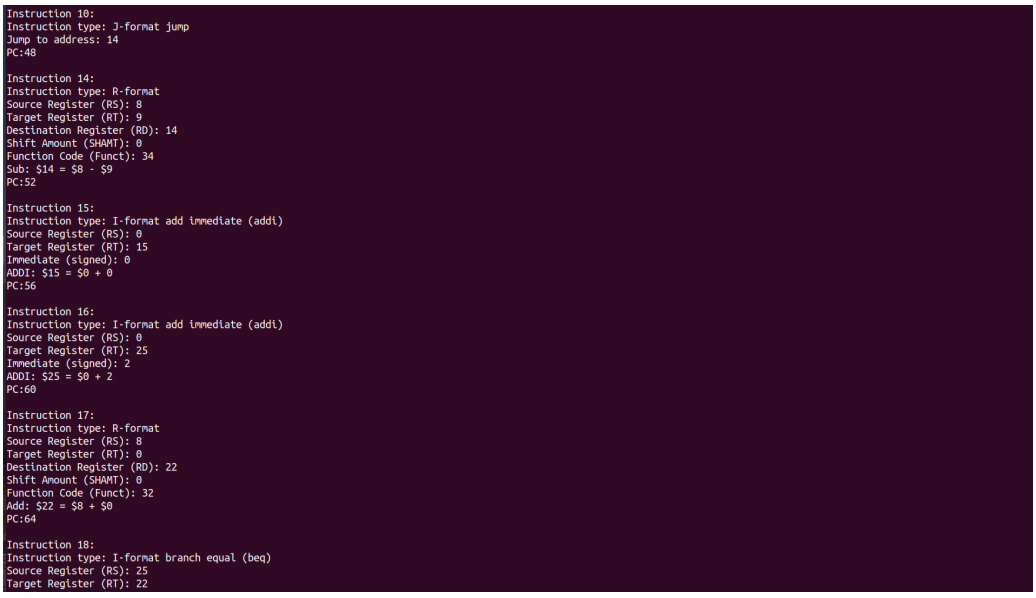

Figure 4: Result

Figure 5: Result

Figure 6: Result

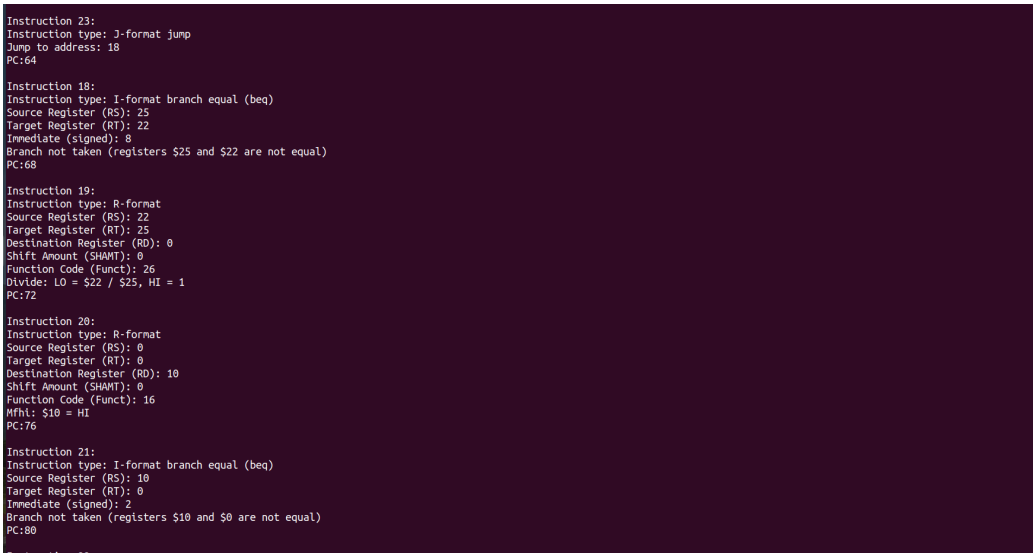

Figure 7: Result

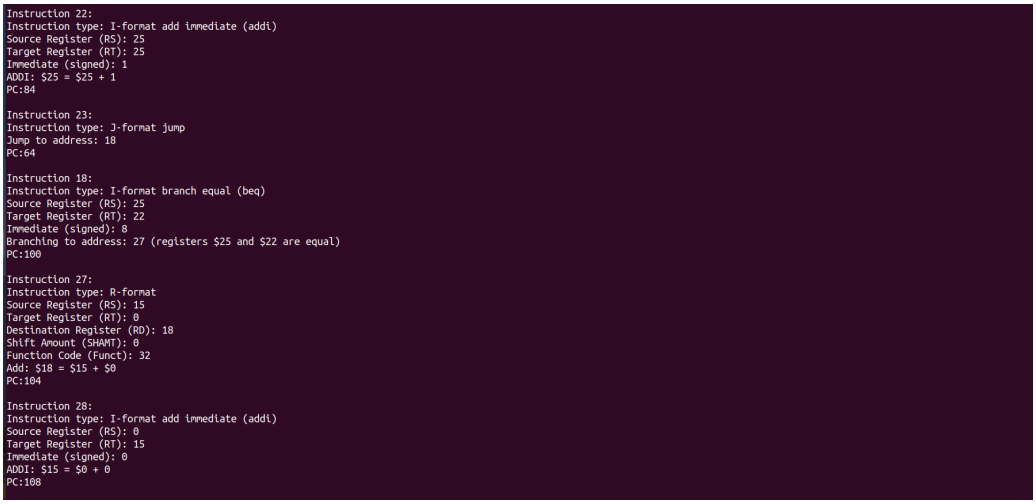

Figure 8: Result

```
Instruction 29:
Instruction type: I-format add immediate (addi)
Source Register (RS): 0
Target Register (RT): 25
Immediate (signed): 2
ADDI: $25 = $0 + 2
PC:112

Instruction 30:
Instruction type: R-format
Source Register (RS): 9
Target Register (RT): 0
Destination Register (RD): 22
Shift Amount (SHAMT): 0
Function Code (Funct): 32
Add: $22 = $9 + $0
PC:116

Instruction 31:
Instruction type: I-format branch equal (beq)
Source Register (RS): 25
Target Register (RT): 22
Immediate (signed): 8
Branch not taken (registers $25 and $22 are not equal)
PC:120

Instruction 32:
Instruction type: R-format
Source Register (RS): 22
Target Register (RT): 25
Destination Register (RD): 0
Shift Amount (SHAMT): 0
Function Code (Funct): 26
Divide: LO = $22 / $25, HI = 1
PC:124

Instruction 33:
Instruction type: R-format
Source Register (RS): 0
Target Register (RT): 0
Destination Register (RD): 10
Shift Amount (SHAMT): 0
Function Code (Funct): 16
Mfhi: $10 = HI
PC:128
```

Figure 9: Result

```
Instruction 34:
Instruction type: I-format branch equal (beq)
Source Register (RS): 10
Target Register (RT): 0
Immediate (signed): 2
Branch not taken (registers $10 and $0 are not equal)
PC:132

Instruction 35:
Instruction type: I-format add immediate (addi)
Source Register (RS): 25
Target Register (RT): 25
Immediate (signed): 1
ADDI: $25 = $25 + 1
PC:136

Instruction 36:
Instruction type: J-format jump
Jump to address: 31
PC:116

Instruction 31:
Instruction type: I-format branch equal (beq)
Source Register (RS): 25
Target Register (RT): 22
Immediate (signed): 8
Branching to address: 40 (registers $25 and $22 are equal)
PC:152

Instruction 40:
Instruction type: R-format
Source Register (RS): 15
Target Register (RT): 0
Destination Register (RD): 19
Shift Amount (SHAMT): 0
Function Code (Funct): 32
Add: $19 = $15 + $0
PC:156
```

Figure 10: Result

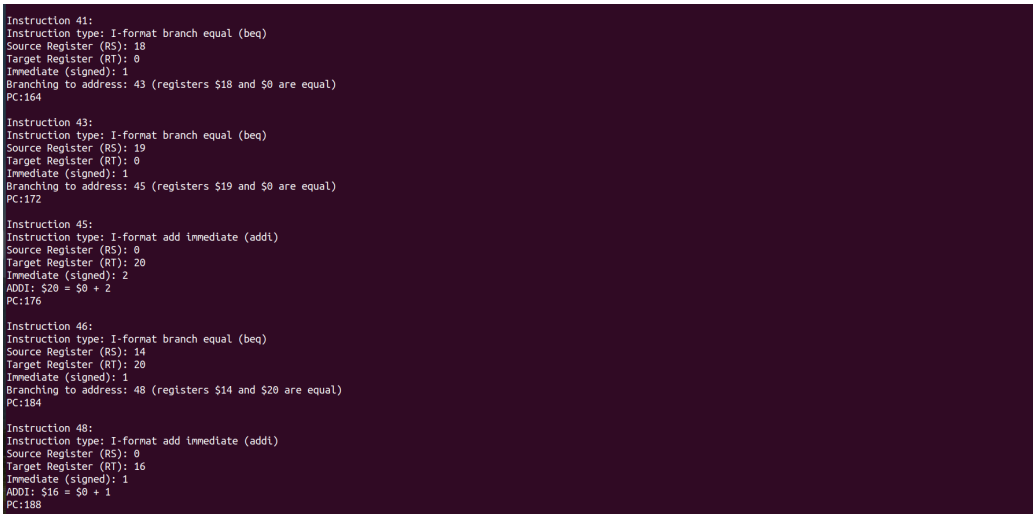

Figure 11: Result

Figure 12: Result