***Name: Kanak Dhiman***

***Roll no: 2022-BCS-035***

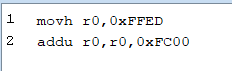
**COA THEORY ASSIGNMENT 1**

1. **Write a program to load the value 0xFFEDFC00 into r0. Try to minimize the number of instructions.**

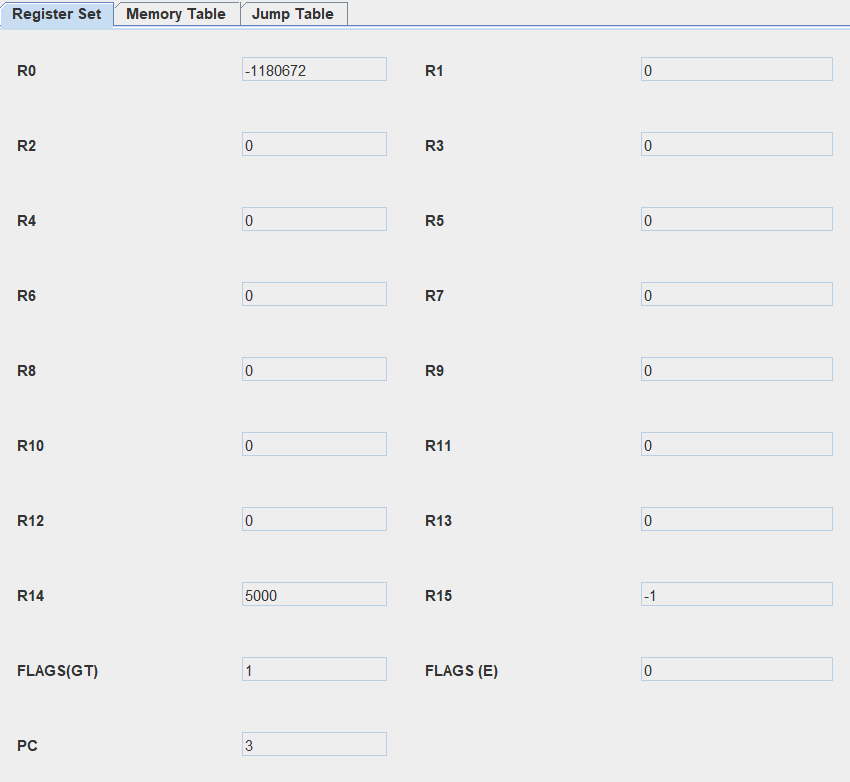
**Methodology:**

* Move the MSB 16-bits using ‘h’ modifier to register r0.
* Add the LSB 16-bits using modifier ‘u’ to register r0.

**Code:**



**Output :**

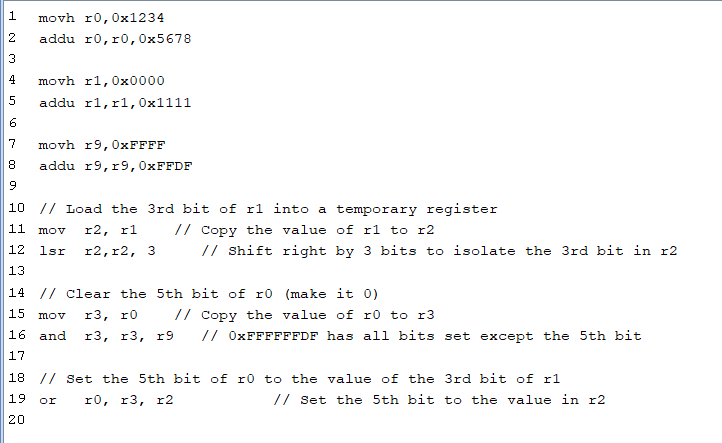


1. **Write an assembly program to set the 5th bit of register r0 to the value of 3rd bit of r1. Keep the rest of the contents of r0 the same. The convention is that the LSB is the first bit, and the MSB is the 32nd bit. Try to minimize the number of instructions.**

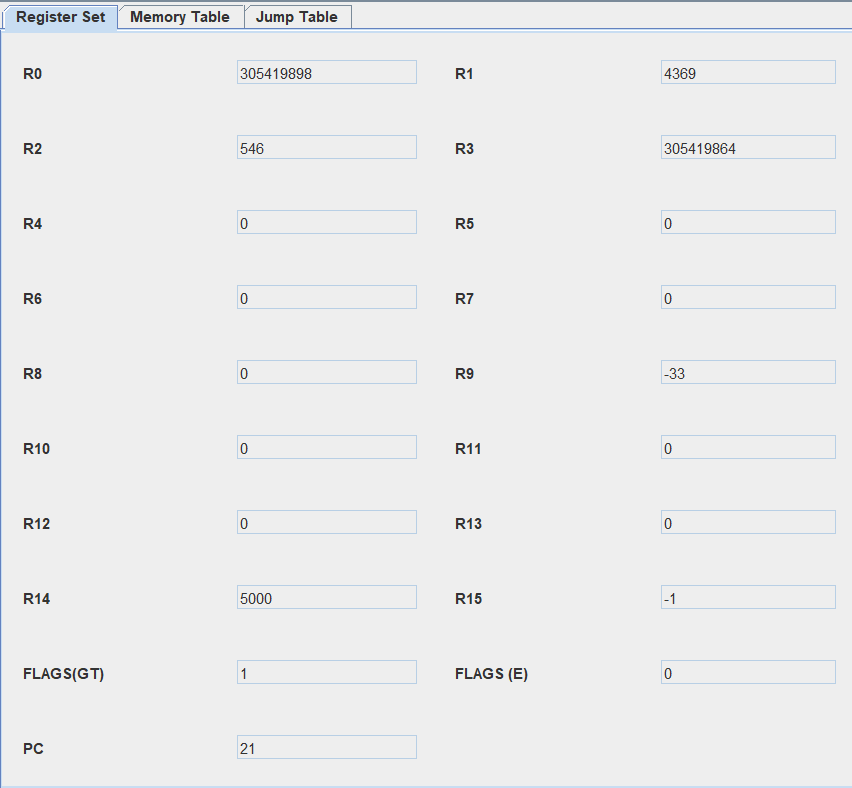
**Methodology**:

* Load two 16-bit immediate values into registers `r0` and `r1`.
* Add two 16-bit immediate values to registers `r0` and `r1`.
* Load a 16-bit immediate value into register `r9` and add another 16-bit immediate value to it.
* Copy the value in register `r1` to register `r2`.
* Shift the value in register `r2` right by 3 bits to isolate the 3rd bit of the original value in `r1`.
* Copy the value in register `r0` to register `r3`.
* Perform a bitwise AND operation between the value in `r3` and the value in `r9` to clear the 5th bit of the value in `r3`.
* Perform a bitwise OR operation between the value in `r3` (with the 5th bit cleared) and the value in `r2` (containing the isolated 3rd bit). This sets the 5th bit of the value in `r3` to the value in `r2`.

**Code**:



**Output**:



1. **Write a simpleRISC program to convert an integer stored in memory from the little endian to the big endian format.**

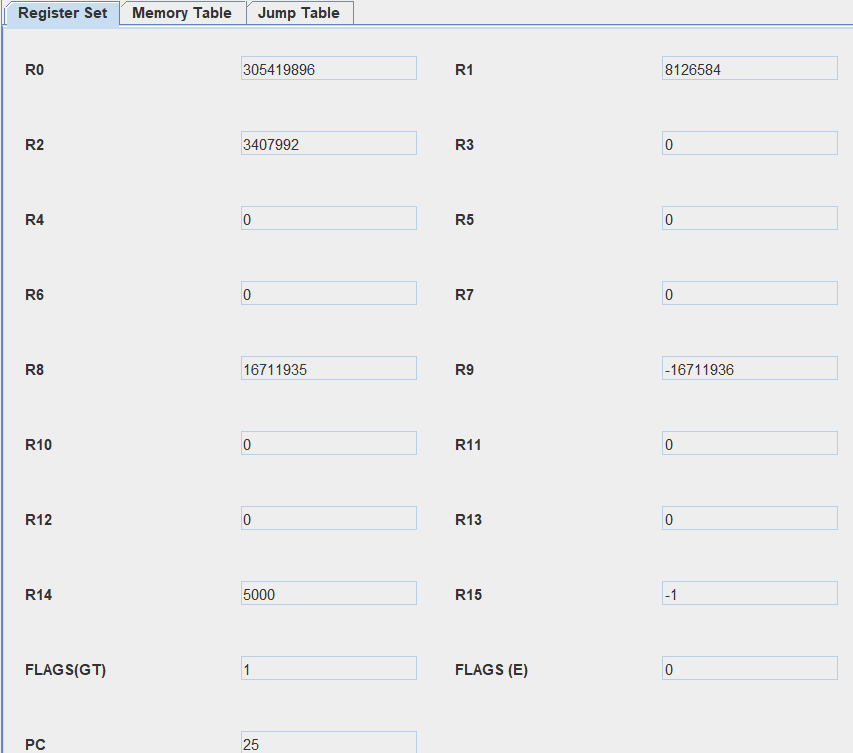
**Methodology:**

* Load 16-bit immediate values into registers `r0`, `r8`, and `r9`, and add other immediate values to them.
* Copy the value from `r0` to `r1`.
* Swap the bytes in `r1` to convert from little-endian to big-endian format:
* - Extract the most significant byte (MSB) into `r2` by shifting right by 24 bits and clear it from `r1`.
* - Shift `r1` left by 8 bits to make room for the MSB and place the MSB in the correct position.
* - Extract the second most significant byte into `r2` by shifting right by 8 bits and clear it from `r1`.
* - Shift `r1` left by 8 bits to make room for the second most significant byte and place it in the correct position.
* After these operations, the value in `r1` is in big-endian format (0x87654321).

**Code:**



**Output:**

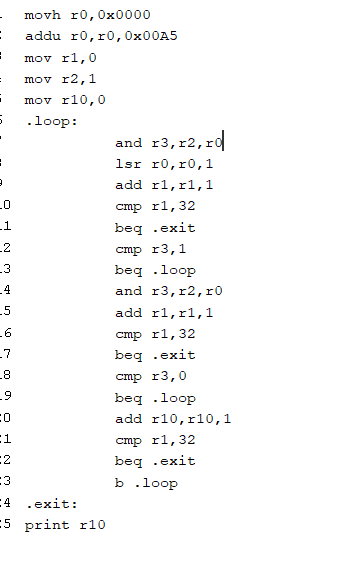


1. **Given a 32-bit integer in r3, write a simpleRISC assembly program to count the number of 1 to 0 transitions in it.**

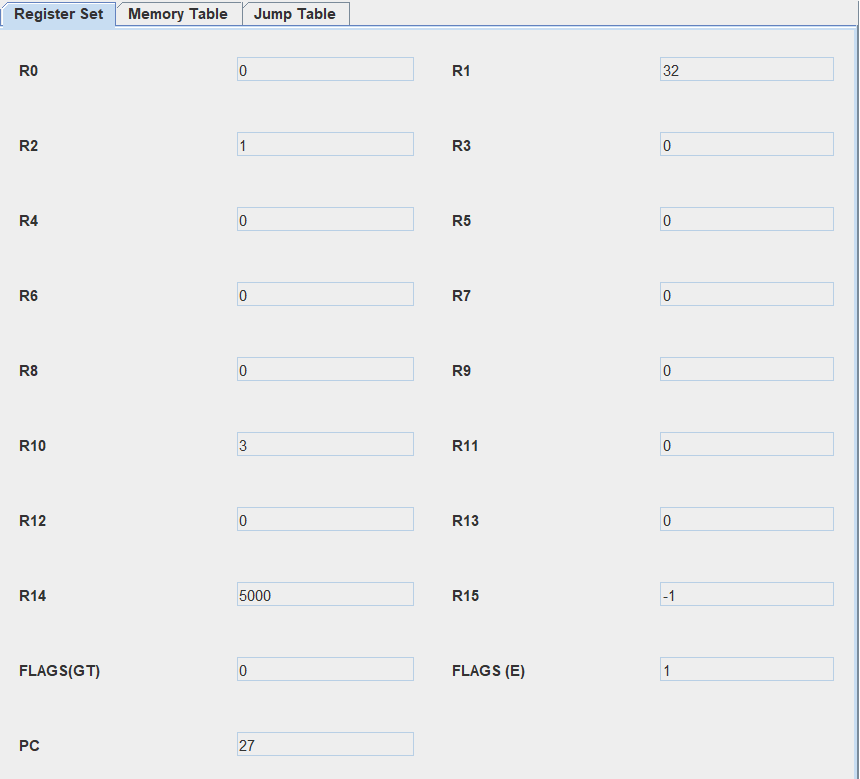
**Methodology:**

* It loads the 32-bit integer 0x00A5 into register `r0`, setting the upper 16 bits to 0x0000 and adding the lower 16 bits.
* It initializes several registers, including `r1`, `r2`, and `r10`, which are used for various purposes in the loop.
* The main loop processes each bit of the 32-bit integer in `r0`. It checks if the current bit is 1 and increments `r10` when transitioning from 1 to 0.
* The loop continues until all 32 bits have been processed.
* Finally, it prints the count of 1 to 0 transitions stored in `r10`.

**Code:**



**Output:**



1. **Design a simpleRISC program that examines a 32 bit value stored in r1 and counts the number of contiguous sequences of 1s. For example, the value: 01110001000111101100011100011111 contains six sequences of 1s.**

**Methodology:**

* The program starts by initializing `r2` and `r3` to 0.
* It then iterates through each bit from the least significant bit to the most significant bit.
* For the given input, here are the sequences of 1s and their lengths:

- Sequence 1: 111 (3 bits)

- Sequence 2: 1 (1 bit)

- Sequence 3: 1111 (4 bits)

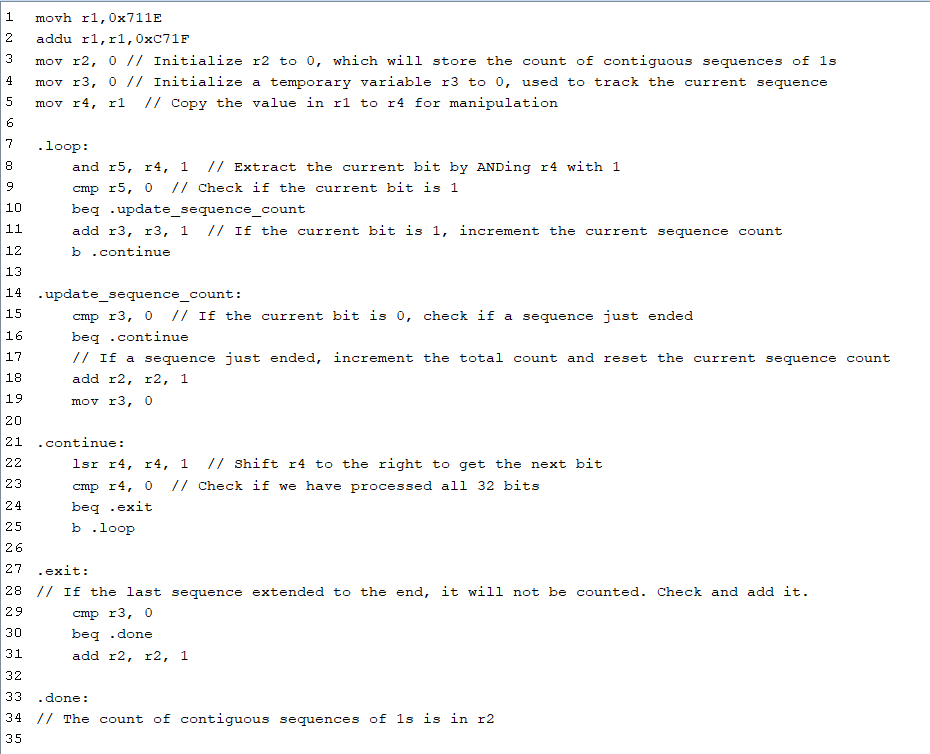
- Sequence 4: 11 (2 bits)

- Sequence 5: 111 (3 bits)

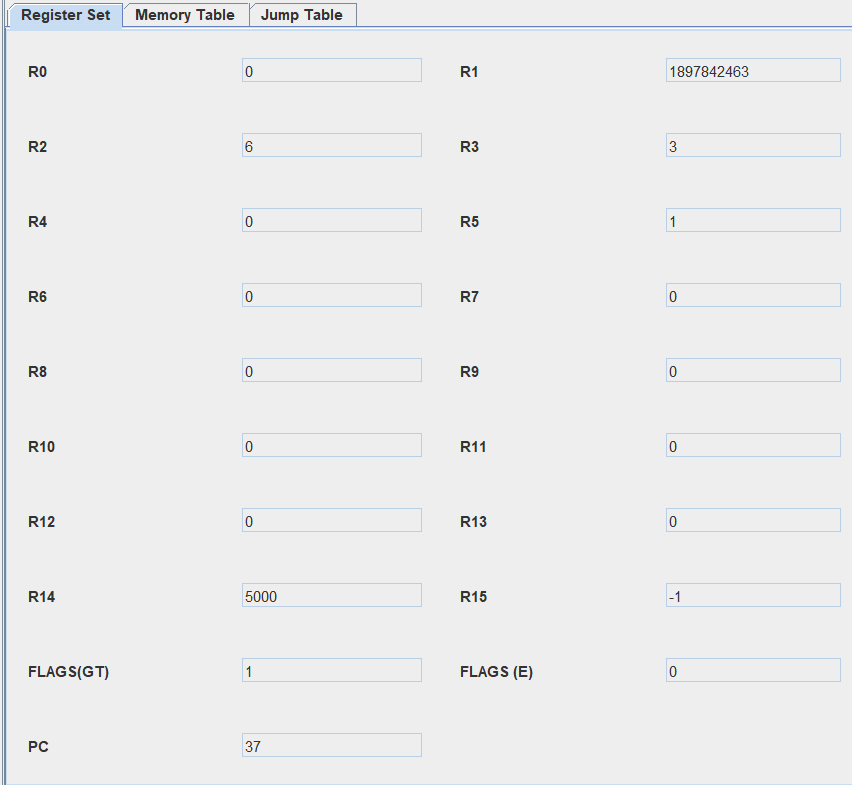
- Sequence 6: 1111 (4 bits)

* The program counts the number of contiguous sequences of 1s and stores the result in `r2`.
* At the end of the program, `r2` will contain the count of contiguous sequences of 1s, which is 6 for the input value 0x711EC71F.

**Code:**



**Output:**



1. **Write a program in simpleRISC assembly to subtract two 64 bit numbers, where each number is stored in two registers.**

**Methodology:**

* Load 32-bit values into registers `r1`, `r2`, `r3`, and `r4` with both high and low 16-bit portions.

- `r1` gets a 32-bit value where the high 16 bits are set to 0xFFFF, and the low 16 bits are set to 0x0000.

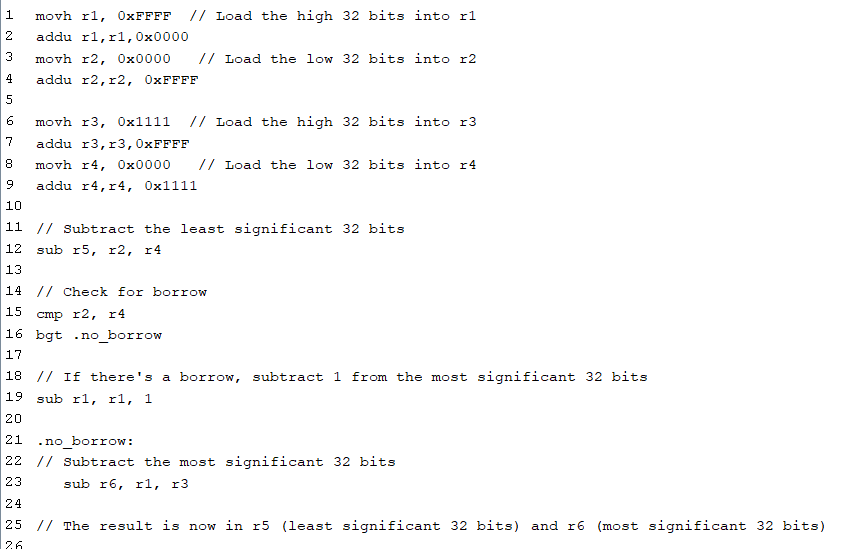
- `r2` gets a 32-bit value where the high 16 bits are set to 0x0000, and the low 16 bits are set to 0xFFFF.

- `r3` gets a 32-bit value where the high 16 bits are set to 0x1111, and the low 16 bits are set to 0xFFFF.

- `r4` gets a 32-bit value where the high 16 bits are set to 0x0000, and the low 16 bits are set to 0x1111.

* Subtract the least significant 32 bits by performing `sub r5, r2, r4`. This computes the difference between the values in `r2` and `r4`, storing it in `r5`.
* Check for a borrow during the subtraction by comparing the values in `r2` and `r4` using `cmp r2, r4`.
* If there's a borrow (indicated by `bgt .no\_borrow`), subtract 1 from the most significant 32 bits in `r1` to correct for the borrow.
* After addressing the borrow (or if there was no borrow), subtract the most significant 32 bits by performing `sub r6, r1, r3`. This computes the difference between the values in `r1` and `r3`, storing it in `r6`.
* The result is now in `r5` (representing the least significant 32 bits) and `r6` (representing the most significant 32 bits). This operation effectively subtracts a 64-bit number where each 32-bit half is stored in two registers.

**Code:**



**Output:**

