**Mini Project Report  
Week 8 - 9**

|  |  |
| --- | --- |
| Student ID | 20235890 |
| Student Name | Nguyễn Đức Anh |

**Problem A-3:**

**Code:**

.data

    Input\_M:    .asciz "Input M: "

    Input\_N:    .asciz "Input N: "

    prime\_msg:   .asciz "Prime Num: "

    newline:     .asciz "\n"

.text

start:

    # Print Input messenger for M

    la a0, Input\_M

    li a7, 4

    ecall

    # Input M

    li a7, 5

    ecall

    addi s0, a0, 0

    # Print Input messenger for N

    la a0, Input\_N

    li a7, 4

    ecall

    # Input N

    li a7, 5

    ecall

    addi s1, a0, 0

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Prime Number Checking Algorithms:

# Base Case: if t0 (M) = 2 -> print M

# Even numbers > 2 basically is not the prime number -> ignore

# Numbers < 2 is not the prime number

# To check other numbers: Start from the divisor 3 and only check odd divisors up to sqrt(t0),

# Because any composite number greater than 2 must have at least one factor less than or equal to sqrt(t0).

# Increse the divisor by 2 to skip even numbers.

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    init\_index:

        addi t0, s0, 0  # t0 = M (starting from M)

    check\_prime:

        bgt t0, s1, end # Check if t0 > N -> jump to end

        # Case: Check if t0 == 2

        li t2, 2

        beq t0, t2, print\_prime

        # Check if t0 < 2 or t0 is even (t0 % 2 == 0)

        li t3, 2

        blt t0, t3, next\_num   # If t0 < 2, jump to next number

        rem t4, t0, t3         # t4 = t0 % t3 (M % 2)

        beq t4, zero, next\_num # If t0 is even, jump to next number

        # Set divisor to 3 and Bool value to check prime to 1 (true)

        li t1, 3           # Divisor starts at 3

        li t2, 1           # Prime Check value (1 (true) means prime, 0 (false) means not prime)

    is\_prime\_loop:

        # Break loop if divisor (t1) \* divisor (t1) > current number (t0)

        mul t3, t1, t1

        bgt t3, t0, print\_prime

        # Check if t0 % t1 == 0

        rem t4, t0, t1

        beq t4, zero, not\_prime

        # Increment divisor by 2 (only odd divisors)

        addi t1, t1, 2

        j is\_prime\_loop

    not\_prime:

        li t2, 0 # Set Prime check value to 0 (false) means not prime

    print\_prime:

        beq t2, zero, next\_num # If Prime check is still 1, print the number

        # Print "Prime Num: "

        la a0, prime\_msg

        li a7, 4

        ecall

        # Print the prime number (t0)

        mv a0, t0

        li a7, 1

        ecall

        # Print newline

        la a0, newline

        li a7, 4

        ecall

    next\_num:

        # Increse t0 and check the next number

        addi t0, t0, 1

        j check\_prime

end:

    # Exit the program

    li a7, 10

    ecall

**Explanation:**

My idea of this problem I implement into C code and from that I bring it into RISC-V.

int isPrime(int number)

{

    if (number == 2) return 1;

    if (number < 2 || number % 2 == 0) return 0;

    for (int i = 3; i \* i <= number; i += 2)

    {

        if (number % i == 0) return 0;

    }

    return 1;

}

Prime Checking Algorithm

* Setting the Starting Point (init\_index): t0 is initialized with M, the starting point for checking prime numbers.
* Loop (check\_prime): This loop iterates through each number from M to N, checking if each number is prime.
* If t0 > N, it jumps to end to exit the program.

Prime Number Check Logic

Special Cases:

* Case t0 == 2: If t0 is 2, it’s a prime number, so the program jumps to print\_prime to display it.
* Exclude Numbers < 2: Numbers less than 2 cannot be prime, so the program skips these by jumping to next\_num.
* Exclude Even Numbers > 2: If t0 is even, it is not a prime number (except 2), so it skips the rest of the checks and jumps to next\_num.

General Prime Check:

* The code then sets up a divisor starting at 3 and assumes the number is prime (t2 is set to 1, meaning "true" for prime).
* Divisor Check Loop (is\_prime\_loop):
* The divisor t1 is incremented by 2 in each iteration, checking only odd divisors up to the square root of t0.
* If any divisor divides t0 without remainder (t0 % t1 == 0), t2 is set to 0, indicating t0 is not prime, and it jumps to not\_prime.
* If no divisor divides t0 up to the square root, t0 is confirmed as prime, and it jumps to print\_prime.

Printing the Prime Number

* If t2 remains 1 (indicating t0 is prime), the code prints the prime number t0.

**Result:**

Case 1:

A screenshot of a computer program

Description automatically generated

Case 2:

A screenshot of a computer program

Description automatically generated

Case 3:

A screenshot of a computer

Description automatically generated

Case 4:

A screenshot of a computer

Description automatically generated

Case 5:

A screenshot of a computer

Description automatically generated

**Problem B-3:**

**Code:**

.data

    array:   .space 200      # Declare space for 50 integers (4 bytes each)

    msg\_and: .asciz " and "

.text

start:

    # Input n (number of elements)

    li a7, 5

    ecall

    init\_method:

    # Initialize loop to input array elements

        addi t1, a0, 0  # Load number of elements (t1 = n)

        la t0, array    # Load base address of array

        li t2, 0        # Initialize index i = 0

    input\_array\_loop:

        bge t2, t1, max\_adjacency\_sum # If i >= n, jump to max\_adjacency\_sum

        # Read the next array element

        li a7, 5

        ecall

        sw a0, 0(t0)        # Store a0 into array[i]

        addi t2, t2, 1      # i = i + 1

        addi t0, t0, 4      # Move to the next element (4 bytes)

        j input\_array\_loop  # Repeat the loop

    max\_adjacency\_sum:

        la t0, array           # Reset t0 to the base of the array

        lw t3, 0(t0)           # Load arr[0] into t3

        lw t4, 4(t0)           # Load arr[1] into t4

        add t5, t3, t4         # Initialize max\_Adjacentcy\_Sum (t5) = arr[0] + arr[1]

        addi t2, t1, -1        # Set t2 = n - 1 for loop limit

    # First loop to find the maximum adjacency sum

    find\_max\_sum:

        addi t2, t2, -1           # Decrese the index

        blt t2, zero, print\_pairs # If i < 0, go to second loop to print pairs

        lw t3, 0(t0)           # Load arr[i] (t3)

        lw t4, 4(t0)           # Load arr[i + 1] (t4)

        add t6, t3, t4         # Calculate arr[i] + arr[i + 1]

        # Check if arr[i] + arr[i + 1] > max\_Adjacentcy\_Sum

        ble t6, t5, next       # If current sum <= max, skip to next

        add t5, t6, zero       # Update max\_Adjacentcy\_Sum

    next:

        addi t0, t0, 4         # Move to next element

        j find\_max\_sum         # Repeat the loop

    print\_pairs:

        # Reset t0 to the base of the array for second loop

        la t0, array

        addi t2, t1, -1        # Set t2 = n - 1 for loop limit

    print\_loop:

        addi t2, t2, -1        # Decrese the index

        blt t2, zero, end      # If i < 0, end program

        lw t3, 0(t0)           # Load arr[i]

        lw t4, 4(t0)           # Load arr[i + 1]

        add t6, t3, t4         # Calculate arr[i] + arr[i + 1]

        # Check if arr[i] + arr[i + 1] == max\_Adjacentcy\_Sum

        bne t6, t5, skip\_print # If not equal, skip to next iteration

        # Print arr[i]

        addi a0, t3, 0

        li a7, 1

        ecall

        # Print " and "

        la a0, msg\_and

        li a7, 4

        ecall

        # Print arr[i + 1]

        addi a0, t4, 0

        li a7, 1

        ecall

        # Print newline

        li a7, 11

        li a0, 10

        ecall

    skip\_print:

        addi t0, t0, 4         # Move to next element

        j print\_loop           # Repeat the loop

end:

    # End the program

    li a7, 10

    ecall

**Explanation:**

My idea of this this problem I also implement it into C code and bring it into RISC-V.

max\_Adjacentcy\_Sum = arr[0] + arr[1];

    for (i = 0; i < n; i++)

    {

        If ((arr[i] + arr[i + 1]) > max\_Adjacentcy\_Sum) max\_Adjacentcy\_Sum = arr[i] + arr[i + 1];

    }

    for (i = 0; i < n; i++)

    {

        if ((arr[i] + arr[i + 1]) == max\_Adjacentcy\_Sum) printf("%d and %d", arr[i], arr[i + 1]);

    }

Calculate Maximum Adjacency Sum (Label: max\_adjacency\_sum and find\_max\_sum)

* Initializes max\_Adjacentcy\_Sum as the sum of the first two elements (arr[0] + arr[1]) using t5 as the max sum register.
* Starts the find\_max\_sum loop to go through the array from arr[0] to arr[n - 2], calculating the sum of each adjacent pair arr[i] + arr[i + 1].
* For each pair, it checks if the current sum (arr[i] + arr[i + 1], stored in t6) is greater than max\_Adjacentcy\_Sum. If it is, it updates max\_Adjacentcy\_Sum.
* The loop continues until all pairs are checked.

Print Pairs Matching Maximum Adjacency Sum (Label: print\_pairs and print\_loop)

* Resets t0 to the base of array to start a new loop through the elements.
* (print\_loop) calculates arr[i] + arr[i + 1] for each adjacent pair and checks if it matches max\_Adjacentcy\_Sum.
* If a pair matches max\_Adjacentcy\_Sum, print arr[i] and arr[i + 1]

**Result:**

Case 1:

A screenshot of a computer

Description automatically generated

Case 2:

A screenshot of a computer program

Description automatically generated

Case 3:

A screenshot of a computer program

Description automatically generated

Case 4:

A screenshot of a computer program

Description automatically generated

Case 5:

**A screenshot of a computer

Description automatically generated**

**Problem C-13:**

**Code:**

.data

    A: .space 32    # String A

    B: .space 32    # String B

.text

start:

    # Get input string A from keyboard

    li a7, 8

    la a0, A

    li a1, 32

    ecall

    # Get input string B from keyboard

    li a7, 8

    la a0, B

    li a1, 32

    ecall

    init\_pointer:

    # Initialize pointers for iterating through A and B

    la t0, A              # t0 points to start of string A

    la t1, B              # t1 points to start of string B

    loop\_A:

        lb t2, 0(t0)            # Load a byte from string A into t2

        beq t2, zero, end       # If t2 = null (end of string A), jump to end

        # Check if t2 is a lowercase letter (between 'a' and 'z')

        li t5, 'a'              # Load 'a' into t5

        slt t6, t2, t5          # if t2 < 'a', t6 = 1 or 0

        bne t6, zero, next\_A    # If t6 = 1, jump to next\_A

        li t5, 'z'              # Load 'z' into t5

        slt t6, t5, t2          # if 'z' < t2, t6 = 1 or 0

        bne t6, zero, next\_A    # If t6 = 1, jump to next\_A

    init\_checkVal:

        # Check if character in A appears in B

        la t1, B                # Reset t1 to the start of string B

        addi t3, zero, 0        # Set t3 = 0 to indicate not found in B

    loop\_B:

        lb t4, 0(t1)              # Load a byte from string B into t4

        beq t4, zero, check\_print # If t4 = null, jump to check\_print

        beq t2, t4, found\_B       # If t2 = t4 (char in A similar to char in B), jump to found\_B

        addi t1, t1, 1            # Move to next character in B

        j loop\_B                  # Repeat loop for B

    found\_B:

        li t3, 1              # Set t3 = 1 when found in B

    check\_print:

        beq t3, zero, print\_char # If t3 = 0 means not found in B, print char in A

        j next\_A                 # Else move to next char in A

    # Print the result

    print\_char:

        addi a0, t2, 0

        li a7, 11

        ecall

    next\_A:

        addi t0, t0, 1        # Move to next character in A

        j loop\_A              # Repeat loop for A

end:

    # Exit the program

    li a7, 10

    ecall

**Explanation:**

* The code reads two strings, A and B, from user input.

Pointer Initialization

* The pointers t0 and t1 are set to the start of A and B, respectively, to facilitate character-by-character iteration through these strings.

(loop\_A) - Iterating through A

* The loop reads one byte at a time from string A using lb t2, 0(t0).
* If t2 is a null byte (beq t2, zero, end), this indicates the end of string A, and the program jumps to end to exit.
* If not, it checks if t2 is a lowercase letter.

Character Check - Lowercase Letter in A

* This block checks if the current character t2 lies within the ASCII range for lowercase letters ('a' to 'z').
* The conditions slt t6, t2, t5 (for 'a') and slt t6, t5, t2 (for 'z') are used to skip non-lowercase characters.

(loop\_B) - Check if Character from A is in B

* For each valid character in A, loop\_B iterates through string B to see if the character exists there.
* t3 is initially set to 0, indicating "not found" in B.
* If a match between t2 and t4 (character in B) is found, t3 is set to 1 to indicate presence in B, and the program jumps to check\_print.
* If loop\_B completes without finding a match, t3 remains 0.

Print Character if Not in B

* After checking B, if t3 is still 0 (meaning the character is not in B), print the character.

**Result:**

Case 1:

A screenshot of a computer

Description automatically generated

Case 2:

A screenshot of a computer program

Description automatically generated

Case 3:

A screenshot of a computer program

Description automatically generated

Case 4:

A screenshot of a computer program

Description automatically generated

Case 5:

A screenshot of a computer

Description automatically generated

Case 6:

A screenshot of a computer program

Description automatically generated