

CSC3050 Project1 Report

121090017

1 Big Pictures Thoughts and Ideas

Implementing ALU functions in RISC-V assembly offers profound insights into computational basics and hardware operations. This task involves creating missing instructions like 'not', 'subi', and 'bg', emphasizing understanding of data flow and instruction execution. For instance, the 'not' operation can be achieved by flipping register bits through XOR with a mask. Selecting from different instruction sets (e.g., 'add/sub', 'beq/bne/slt') highlights the nuances between similar operations and the importance of handling overflows separately for deeper error management. Task 2 focuses on optimizing a given assembly program to reduce execution time and expand its range tenfold, suggesting improvements from $O(n^2)$ to $O(n \cdot \sqrt{n})$. This requires algorithmic efficiency and leveraging mathematical properties to minimize computations. By completing these tasks, we gain practical skills in low-level programming, problem-solving, and optimization techniques, bridging theoretical knowledge with real-world application. This project underscores the balance between simplicity and functionality in instruction set design and the critical role of efficient coding practices in enhancing performance. Overall, it provides a comprehensive understanding of processor architecture and system programming essentials.

2 Implementation of Task1

2.1 statement

I have chosen add,beq,or,srl. i did not do the extra problem.

2.2 implementation details

2.2.1 NOT Function

```
NOT:
    # please start your code
    lui t0, %hi(registers)
    addi t0, t0, %lo(registers)
    slli t1, a0, 2
    add t1, t0, t1
    lw t2, 0(t1)
    not t2, t2
    slli t3, a1, 2
    add t3, t0, t3
    sw t2, 0(t3)
    li a0, 0
    ret
    .size NOT, --NOT
    .align 2
    .globl SUBI
    .type SUBI, @function
```

The NOT function implements a bitwise NOT operation on the value stored in the register indexed by a0 and stores the result in the register indexed by a1. First, the base address of the registers is loaded into t0 using lui and addi. The index a0 is scaled by 4 to account for 32-bit elements, and the address of the source register is calculated. The value from this address is loaded into t2, where the not instruction flips all bits. The destination register's address is similarly computed using a1, and the result is stored back. Finally, a0 is set to 0 (indicating no branch) and the function returns. This approach ensures that the bitwise NOT operation is efficiently applied to the specified registers.

2.2.2 SUBI Function

```
39 SUBI:
40     # please start your code
41     lui t0, %hi(registers)
42     addi t0, t0, %lo(registers)
43     slli t1, a0, 2
44     add t1, t0, t1
45     lw t2, 0(t1)
46     sub t2, t2, a1
47     slli t3, a2, 2
48     add t3, t0, t3
49     sw t2, 0(t3)
50     li a0, 0
51     ret
52     .size SUBI, --SUBI
53     .align 2
54     .globl BG
55     .type BG, @function
56
```

The SUBI function performs subtraction between the value in the register indexed by a0 and an immediate value a1, storing the result in the register indexed by a2. Similar to NOT, it begins by loading the base address of the registers into t0. The source register's address is calculated by scaling a0 by 4, and its value is loaded into t2. The subtraction is then performed with sub t2, t2, a1. The destination register's address is computed using a2, and the result is stored back. Setting a0 to 0 indicates no branching, and the function returns. This method efficiently handles immediate subtraction while maintaining clarity and simplicity in the assembly code.

2.2.3 BG Function

The BG function checks if the value in the register indexed by a0 is greater than the value in the register indexed by a1, returning 1 if true and 0 otherwise. It starts by loading the base address of the registers into t0. Both source registers' addresses are calculated by scaling a0 and a1 by 4, and their values are loaded into t2 and t3, respectively. The slt instruction sets a0 to 1 if t3 (from a1) is less than t2 (from a0), effectively performing a "greater than" comparison. The function then returns immediately, using a0 as the return value to indicate whether the condition was met. This

```
58 BG:
59     # please start your code
60     lui t0, %hi(registers)
61     addi t0, t0, %lo(registers)
62     slli t1, a0, 2
63     add t1, t0, t1
64     lw t2, 0(t1)
65     slli t1, a1, 2
66     add t1, t0, t1
67     lw t3, 0(t1)
68     slt a0, t3, t2
69     ret
70     .size BG, --BG
71     .section .rodata
72     .align 2
73
```

implementation leverages RISC-V's slt instruction to perform efficient conditional comparisons without additional branches or logic operations.

2.2.4 ADD Function

```
21 ADD:
22 # please start your code
23 # r1:a0 r2:a1 r0:a3
24
25 lui a6,ahi(registers) #lui 只能够加载一个 20 位的立即数 (高 20 位)
26 addi a6,a6,%lo(registers) #addi 用来加载低 12 位的地址部分
27
28 slli a0,a0,2 #将数组索引转为字节偏移量 (乘以 4), 注意在二进制中左移2位相当于乘以4
29 slli a1,a1,2
30 slli a2,a2,2
31
32 #a3 存储的是 registers 数组的起始地址
33 add a0, a6, a0 #a0 现在保存的是 registers[a0] 的具体地址
34 add a1, a6, a1 #registers[a1] 的地址
35 add a2, a6, a2 #registers[a2] 的地址
36
37 lw a0,0(a0) #从 地址 a0 指向的内存位置加载一个 4 字节 (偏移量是0) 的数据到寄存器 a0
38 lw a1,0(a1)
39
40 add a6, a0, a1
41
42 sw a6, 0(a2) #将 a3 中的值存储到地址 registers[a2] 中
43
44 li a0, 0 #optional
45 jr ra #jr 是跳转指令, ra 是返回地址寄存器 (Return Address Register)
46
47 .size ADD, .-ADD
48 .section .rodata
49 .align 2
50
```

The ADD function performs addition between the values stored in the registers indexed by a0 and a1, and stores the result in the register indexed by a2. The implementation begins by loading the base address of the registers into a6 using lui and addi, which handle the high and low parts of the address, respectively. To correctly access the register values, the indices a0, a1, and a2 are scaled by 4 (using slli), converting them from array indices to byte offsets since each register value is 32 bits (4 bytes).

The addresses for the source registers (registers[a0] and registers[a1]) and the destination register (registers[a2]) are then calculated by adding the scaled indices to the base address. The lw instruction loads the values from these addresses into a0 and a1. The addition operation is performed with add a6, a0, a1, storing the result in a6. Finally, the result is stored back into the destination register using sw a6, 0(a2). The function concludes by optionally setting a0 to 0 and returning control to the caller using jr ra.

Special Tricks Used:

Address Calculation: The use of lui and addi efficiently loads the full address of the registers array, ensuring that both high and low parts of the address are correctly handled.

Index Scaling: By scaling the indices with slli, the code converts register indices to byte offsets, facilitating correct memory addressing for 32-bit values.

Register Usage: The temporary register a6 is used for intermediate calculations, keeping the original input registers (a0, a1, a2) intact until the final store operation.

Efficient Memory Access: The combination of lw and sw instructions ensures minimal overhead in loading and storing values, making the function efficient and straightforward.

2.2.5 BEQ Function

```
19
20 BEQ:
21 # r1:a0 r2:a1
22
23 lui a3,ahi(registers)
24 addi a3,a3,%lo(registers)
25
26 slli a0,a0,2
27 slli a1,a1,2
28
29 add a0, a3, a0
30 add a1, a3, a1
31
32 lw a0, 0(a0)
33 lw a1, 0(a1)
34
35 beq a0, a1, jump_for_beq
36
37 li a0, 0
38 jr ra
39
40 jump_for_beq:
41 li a0, 1
42 jr ra
43
44 .size BEQ, .-BEQ
45 .section .rodata
46 .align 2
47
48
```

The BEQ function checks if the values stored in the registers indexed by a0 and a1 are equal. If they are, it returns 1; otherwise, it returns 0. The implementation starts by loading the base address of the registers into a3 using lui and addi, which handle the high and low parts of the address, respectively. The indices a0 and a1 are scaled by 4 (using slli) to convert them from array indices to byte offsets, ensuring correct memory addressing for 32-bit values.

The addresses for the source registers (registers[a0] and registers[a1]) are then calculated by adding the scaled indices to the base address. The lw instruction loads the values from these addresses into a0 and a1. The beq a0, a1, jump_for_beq instruction checks if the loaded values are equal. If they are, control jumps to the jump_for_beq label, where a0 is set to 1 before returning. If the values are not equal, the function continues to the next instruction, setting a0 to 0 and returning.

Special Tricks Used:

Address Calculation: The use of lui and addi efficiently loads the full address of the registers array, ensuring that both high and low parts of the address are correctly handled.

Index Scaling: By scaling the indices with slli, the code converts register indices to byte offsets, facilitating correct memory addressing for 32-bit values.

Branching on Equality: The beq instruction directly compares the loaded values and branches based on equality, simplifying the logic and avoiding additional conditional checks or arithmetic operations.

Efficient Return Handling: Using labels like jump_for_beq allows for clear branching and return value setting, making the function easy to understand and

maintain. The use of `li a0, 1` and `li a0, 0` ensures that the function returns the correct result based on the comparison.

2.2.6 OR Function

```
19 OR:
20     lui a5,%hi(registers)
21     addi a5,a5,%lo(registers)
22     slli a0,a0,2
23     slli a1,a1,2
24     slli a2,a2,2
25     add a0, a5, a0
26     add a1, a5, a1
27     add a2, a5, a2
28     lw a0,0(a0)
29     lw a1,0(a1)
30     or a5, a0, a1
31     sw a5, 0(a2)
32     li a0, 0
33     jr ra
34
35     .size OR, .-OR
36     .section .rodata
37     .align 2
38
```

The OR function performs a bitwise OR operation on values from registers indexed by `a0` and `a1`, storing the result in the register indexed by `a2`. It starts by loading the base address of the registers into `a5` using `lui` and `addi`. The indices `a0`, `a1`, and `a2` are scaled by 4 (using `slli`) to convert them to byte offsets, ensuring correct memory addressing for 32-bit values.

The addresses for the source registers (`registers[a0]` and `registers[a1]`) and the destination register (`registers[a2]`) are calculated by adding the scaled indices to the base address. The `lw` instruction loads the values from these addresses into `a0` and `a1`. The bitwise OR operation is performed with `or a5, a0, a1`, and the result is stored back using `sw a5, 0(a2)`. Finally, the function sets `a0` to 0 and returns using `jr ra`.

Special Tricks Used:

Address Calculation: Efficiently loads the full address of the registers array using `lui` and `addi`.

Index Scaling: Converts register indices to byte offsets with `slli`, ensuring correct memory access.

Efficient Memory Access: Uses `lw` and `sw` for minimal overhead in loading and storing values.

Register Usage: Utilizes `a5` as a temporary register for the OR result, keeping inputs (`a0`, `a1`, `a2`) intact until the final store.

2.2.7 SRL Function

```
19 SRL:
20     # rsl:a0 shamt:a1 rd:a2
21
22     lui a3,%hi(registers)
23     addi a3,a3,%lo(registers)
24
25     slli a2,a2,2
26     slli a0,a0,2
27
28     add a2, a3, a2
29     add a0, a3, a0
30
31     lw a0,0(a0)
32     srl a3, a0, a1
33     sw a3, 0(a2)
34
35     li a0, 0
36     jr ra
37
38     .size SRL, .-SRL
39     .section .rodata
40     .align 2
41
```

The SRL function performs a logical right shift on the value from the register indexed by `a0` by the amount specified in `a1`, and stores the result in the register indexed by `a2`. It starts by loading the base address of the registers into `a3` using `lui` and `addi`. The indices `a0` and `a2` are scaled by 4 (using `slli`) to convert them to byte offsets for correct memory addressing.

The addresses for the source (`registers[a0]`) and destination (`registers[a2]`) registers are calculated by adding the scaled indices to the base address. The `lw` instruction loads the value from the source register into `a0`. The logical right shift operation is performed with `srl a3, a0, a1`, storing the result in `a3`. Finally, the result is stored back using `sw a3, 0(a2)`. The function sets `a0` to 0 and returns using `jr ra`.

Special Tricks Used:

Address Calculation: Efficiently loads the full address of the registers array using `lui` and `addi`.

Index Scaling: Converts register indices to byte offsets with `slli` for correct memory access.

Logical Right Shift: Uses `srl` directly for the shift operation, simplifying the logic.

Register Usage: Uses `a3` as a temporary register for the shifted result, keeping inputs (`a0`, `a1`, `a2`) intact until the final store.

2.3 screenshots of running results

2.3.1 required.s

```

student@1c358896f385:~$ time qemu-riscv32 required.out
Test 1: NOT
Pass
Test 2: NOT
Pass
Test 3: NOT
Pass
End of NOT test
Test 1: SUBI
Pass
Test 2: SUBI
Pass
Test 3: SUBI
Pass
End of SUBI test
Test 1: BG
Pass
Test 2: BG
Pass
Test 3: BG
Pass
End of BG test

real    0m0.032s
user    0m0.006s
sys      0m0.012s
--

```

2.3.2 add.s

```

student@1c358896f385:~$ qemu-riscv32 add.out
Test 1: ADD
Pass
Test 2: ADD
Pass
Test 3: ADD
Pass
Do not include overflow detection: ADD
You haven't included it, good job!!!
End of test
--

```

2.3.3 beq.s

```

student@1c358896f385:~$ qemu-riscv32 beq.out
Test 1: BEQ
Pass
Test 2: BEQ
Pass
Test 3: BEQ
Pass
End of test
--

```

2.3.4 or.s

```

student@1c358896f385:~$ qemu-riscv32 or.out
Test 1: OR
Pass
Test 2: OR
Pass
Test 3: OR
Pass
End of test
student@1c358896f385:~$

```

2.3.5 srl.out

```

student@1c358896f385:~$ qemu-riscv32 srl.out
Test 1: SRL
Pass
Test 2: SRL
Pass
Test 3: SRL
Pass
End of test
--

```

```

--
11  judge:
12      addi    sp,sp,-40
13      sw      ra,44(sp)
14      addi    ra,ra,40
15      sw      ra,36(sp)
16      lw      a5,-36(sp)
17      li      a5,1
18      bgt     a4,a5,.L3
19      li      a5,0
20      j       .L3
21  .L2:
22      li      a5,2
23      sw      a5,-20(sp)
24      j       .L4
25  .L5:
26      lw      a4,-36(sp)
27      lw      a5,-20(sp)
28      rem     a5,a4,a5
29      bne     a5,zero,.L5
30      li      a5,0
31      j       .L3
32  .L5:
33      lw      a5,-20(sp)
34      addi    a5,a5,1
35      sw      a5,-20(sp)
36
37      .L4:
38      mul     a4,a3,a3
39      lw      a5,-36(sp)
40      blt     a5,a4,.L2_exit
41      j       .L6
42  .L2_exit:
43      li      a5,1
44      .L3:
45      sv      a8,a5
46      lw      ra,44(sp)
47      addi    sp,sp,40
48      jr      ra
49  .size judge,--judge
50  .section .rodata
51  .align 2
52  .LC0:
53      .string "q"
54      .text
55      .align 2
56      .global main
57      .type main,@function
58  main:
59      addi    sp,sp,-32
60      sw      ra,28(sp)
61      sw      ra,24(sp)
62      addi    ra,ra,32
63      li      a5,1000000000
64      sw      a5,-24(sp)
65      li      a5,2
66      sw      a5,-20(sp)
67      j       .L8
--

```

3 Implementation of Task2

3.1 statement

The initial RISC-V assembly code is designed to process a sequence of integers and perform some type of mathematical check or operation on each number in the sequence. Based on the provided code, the code is checking whether each number is a **prime number**.

How I Improved It:

(1)Optimizing Time Complexity: The original code likely used a brute-force approach to check whether a number is prime, which leads to $O(n^2)$ time complexity. This is because it may have iterated through all numbers from 1 to n for each check, making the process inefficient. To improve this, I optimized the algorithm to reduce the number of checks by using a more efficient method, reducing the time complexity to $O(n*\sqrt{n})$.

Specifically, when checking if a number is prime, instead of checking divisibility by all numbers from 2 up to n, we only need to check divisibility from 2 up to the square root of the number. This is because if a number n has a factor greater than \sqrt{n} , its corresponding factor must be smaller than \sqrt{n} . Therefore, checking up to the square root significantly reduces the number of checks.

(2)Expanding the Range: The task also requires expanding the range of numbers the program operates on—from 1 to 1000 to 1 to 10000. This involves modifying the loop or range bounds to handle the larger number set. I made the necessary adjustments to the code to ensure that it could now handle numbers up to 10000 without needing significant changes elsewhere.

The required changes for expanding the range were minimal and involved adjusting the bounds or limit conditions in the loop to accommodate the new range.

3.2 screenshots of code's execution time before and after optimization

3.2.1 on original sequence range

before optimization:

```
student@ic358896f385:~$ time qemu-riscv32 naive00.out > output00.txt

real    0m1.137s
user    0m1.134s
sys      0m0.004s

student@ic358896f385:~$ █
```

after optimization:

```
1 92861 92867 92870 92873 92883 92886 92891 92893 92899 92901 92911 92913 92923 92927 92941 92943 92987 92989
92881 92887 92891 92897 92899 92903 92907 92909 92919 92929 92931 92949 92953 92959 92967 92969 92979 92983 92989
92991 92993 92999 93001 93003 93007 93009 93019 93029 93031 93049 93053 93059 93067 93069 93079 93083 93089
93091 93093 93099 93101 93103 93107 93109 93119 93129 93131 93149 93153 93159 93167 93169 93179 93183 93189
93191 93193 93199 93201 93203 93207 93209 93219 93229 93231 93249 93253 93259 93267 93269 93279 93283 93289
93291 93293 93299 93301 93303 93307 93309 93319 93329 93331 93349 93353 93359 93367 93369 93379 93383 93389
93391 93393 93399 93401 93403 93407 93409 93419 93429 93431 93449 93453 93459 93467 93469 93479 93483 93489
93491 93493 93499 93501 93503 93507 93509 93519 93529 93531 93549 93553 93559 93567 93569 93579 93583 93589
93591 93593 93599 93601 93603 93607 93609 93619 93629 93631 93649 93653 93659 93667 93669 93679 93683 93689
93691 93693 93699 93701 93703 93707 93709 93719 93729 93731 93749 93753 93759 93767 93769 93779 93783 93789
93791 93793 93799 93801 93803 93807 93809 93819 93829 93831 93849 93853 93859 93867 93869 93879 93883 93889
93891 93893 93899 93901 93903 93907 93909 93919 93929 93931 93949 93953 93959 93967 93969 93979 93983 93989
93991 93993 93999 94001 94003 94007 94009 94019 94029 94031 94049 94053 94059 94067 94069 94079 94083 94089
94091 94093 94099 94101 94103 94107 94109 94119 94129 94131 94149 94153 94159 94167 94169 94179 94183 94189
94191 94193 94199 94201 94203 94207 94209 94219 94229 94231 94249 94253 94259 94267 94269 94279 94283 94289
94291 94293 94299 94301 94303 94307 94309 94319 94329 94331 94349 94353 94359 94367 94369 94379 94383 94389
94391 94393 94399 94401 94403 94407 94409 94419 94429 94431 94449 94453 94459 94467 94469 94479 94483 94489
94491 94493 94499 94501 94503 94507 94509 94519 94529 94531 94549 94553 94559 94567 94569 94579 94583 94589
94591 94593 94599 94601 94603 94607 94609 94619 94629 94631 94649 94653 94659 94667 94669 94679 94683 94689
94691 94693 94699 94701 94703 94707 94709 94719 94729 94731 94749 94753 94759 94767 94769 94779 94783 94789
94791 94793 94799 94801 94803 94807 94809 94819 94829 94831 94849 94853 94859 94867 94869 94879 94883 94889
94891 94893 94899 94901 94903 94907 94909 94919 94929 94931 94949 94953 94959 94967 94969 94979 94983 94989
94991 94993 94999 95001 95003 95007 95009 95019 95029 95031 95049 95053 95059 95067 95069 95079 95083 95089
95091 95093 95099 95101 95103 95107 95109 95119 95129 95131 95149 95153 95159 95167 95169 95179 95183 95189
95191 95193 95199 95201 95203 95207 95209 95219 95229 95231 95249 95253 95259 95267 95269 95279 95283 95289
95291 95293 95299 95301 95303 95307 95309 95319 95329 95331 95349 95353 95359 95367 95369 95379 95383 95389
95391 95393 95399 95401 95403 95407 95409 95419 95429 95431 95449 95453 95459 95467 95469 95479 95483 95489
95491 95493 95499 95501 95503 95507 95509 95519 95529 95531 95549 95553 95559 95567 95569 95579 95583 95589
95591 95593 95599 95601 95603 95607 95609 95619 95629 95631 95649 95653 95659 95667 95669 95679 95683 95689
95691 95693 95699 95701 95703 95707 95709 95719 95729 95731 95749 95753 95759 95767 95769 95779 95783 95789
95791 95793 95799 95801 95803 95807 95809 95819 95829 95831 95849 95853 95859 95867 95869 95879 95883 95889
95891 95893 95899 95901 95903 95907 95909 95919 95929 95931 95949 95953 95959 95967 95969 95979 95983 95989
95991 95993 95999 96001 96003 96007 96009 96019 96029 96031 96049 96053 96059 96067 96069 96079 96083 96089
96091 96093 96099 96101 96103 96107 96109 96119 96129 96131 96149 96153 96159 96167 96169 96179 96183 96189
96191 96193 96199 96201 96203 96207 96209 96219 96229 96231 96249 96253 96259 96267 96269 96279 96283 96289
96291 96293 96299 96301 96303 96307 96309 96319 96329 96331 96349 96353 96359 96367 96369 96379 96383 96389
96391 96393 96399 96401 96403 96407 96409 96419 96429 96431 96449 96453 96459 96467 96469 96479 96483 96489
96491 96493 96499 96501 96503 96507 96509 96519 96529 96531 96549 96553 96559 96567 96569 96579 96583 96589
96591 96593 96599 96601 96603 96607 96609 96619 96629 96631 96649 96653 96659 96667 96669 96679 96683 96689
96691 96693 96699 96701 96703 96707 96709 96719 96729 96731 96749 96753 96759 96767 96769 96779 96783 96789
96791 96793 96799 96801 96803 96807 96809 96819 96829 96831 96849 96853 96859 96867 96869 96879 96883 96889
96891 96893 96899 96901 96903 96907 96909 96919 96929 96931 96949 96953 96959 96967 96969 96979 96983 96989
96991 96993 96999 97001 97003 97007 97009 97019 97029 97031 97049 97053 97059 97067 97069 97079 97083 97089
97091 97093 97099 97101 97103 97107 97109 97119 97129 97131 97149 97153 97159 97167 97169 97179 97183 97189
97191 97193 97199 97201 97203 97207 97209 97219 97229 97231 97249 97253 97259 97267 97269 97279 97283 97289
97291 97293 97299 97301 97303 97307 97309 97319 97329 97331 97349 97353 97359 97367 97369 97379 97383 97389
97391 97393 97399 97401 97403 97407 97409 97419 97429 97431 97449 97453 97459 97467 97469 97479 97483 97489
97491 97493 97499 97501 97503 97507 97509 97519 97529 97531 97549 97553 97559 97567 97569 97579 97583 97589
97591 97593 97599 97601 97603 97607 97609 97619 97629 97631 97649 97653 97659 97667 97669 97679 97683 97689
97691 97693 97699 97701 97703 97707 97709 97719 97729 97731 97749 97753 97759 97767 97769 97779 97783 97789
97791 97793 97799 97801 97803 97807 97809 97819 97829 97831 97849 97853 97859 97867 97869 97879 97883 97889
97891 97893 97899 97901 97903 97907 97909 97919 97929 97931 97949 97953 97959 97967 97969 97979 97983 97989
97991 97993 97999 98001 98003 98007 98009 98019 98029 98031 98049 98053 98059 98067 98069 98079 98083 98089
98091 98093 98099 98101 98103 98107 98109 98119 98129 98131 98149 98153 98159 98167 98169 98179 98183 98189
98191 98193 98199 98201 98203 98207 98209 98219 98229 98231 98249 98253 98259 98267 98269 98279 98283 98289
98291 98293 98299 98301 98303 98307 98309 98319 98329 98331 98349 98353 98359 98367 98369 98379 98383 98389
98391 98393 98399 98401 98403 98407 98409 98419 98429 98431 98449 98453 98459 98467 98469 98479 98483 98489
98491 98493 98499 98501 98503 98507 98509 98519 98529 98531 98549 98553 98559 98567 98569 98579 98583 98589
98591 98593 98599 98601 98603 98607 98609 98619 98629 98631 98649 98653 98659 98667 98669 98679 98683 98689
98691 98693 98699 98701 98703 98707 98709 98719 98729 98731 98749 98753 98759 98767 98769 98779 98783 98789
98791 98793 98799 98801 98803 98807 98809 98819 98829 98831 98849 98853 98859 98867 98869 98879 98883 98889
98891 98893 98899 98901 98903 98907 98909 98919 98929 98931 98949 98953 98959 98967 98969 98979 98983 98989
98991 98993 98999 99001 99003 99007 99009 99019 99029 99031 99049 99053 99059 99067 99069 99079 99083 99089
99091 99093 99099 99101 99103 99107 99109 99119 99129 99131 99149 99153 99159 99167 99169 99179 99183 99189
99191 99193 99199 99201 99203 99207 99209 99219 99229 99231 99249 99253 99259 99267 99269 99279 99283 99289
99291 99293 99299 99301 99303 99307 99309 99319 99329 99331 99349 99353 99359 99367 99369 99379 99383 99389
99391 99393 99399 99401 99403 99407 99409 99419 99429 99431 99449 99453 99459 99467 99469 99479 99483 99489
99491 99493 99499 99501 99503 99507 99509 99519 99529 99531 99549 99553 99559 99567 99569 99579 99583 99589
99591 99593 99599 99601 99603 99607 99609 99619 99629 99631 99649 99653 99659 99667 99669 99679 99683 99689
99691 99693 99699 99701 99703 99707 99709 99719 99729 99731 99749 99753 99759 99767 99769 99779 99783 99789
99791 99793 99799 99801 99803 99807 99809 99819 99829 99831 99849 99853 99859 99867 99869 99879 99883 99889
99891 99893 99899 99901 99903 99907 99909 99919 99929 99931 99949 99953 99959 99967 99969 99979 99983 99989
99991 99993 99999 100001 100003 100007 100009 100019 100029 100031 100049 100053 100059 100067 100069 100079 100083 100089
100091 100093 100099 100101 100103 100107 100109 100119 100129 100131 100149 100153 100159 100167 100169 100179 100183 100189
100191 100193 100199 100201 100203 100207 100209 100219 100229 100231 100249 100253 100259 100267 100269 100279 100283 100289
100291 100293 100299 100301 100303 100307 100309 100319 100329 100331 100349 100353 100359 100367 100369 100379 100383 100389
100391 100393 100399 100401 100403 100407 100409 100419 100429 100431 100449 100453 100459 100467 100469 100479 100483 100489
100491 100493 100499 100501 100503 100507 100509 100519 100529 100531 100549 100553 100559 100567 100569 100579 100583 100589
100591 100593 100599 100601 100603 100607 100609 100619 100629 100631 100649 100653 100659 100667 100669 100679 100683 100689
100691 100693 100699 100701 100703 100707 100709 100719 100729 100731 100749 100753 100759 100767 100769 100779 100783 100789
100791 100793 100799 100801 100803 100807 100809 100819 100829 100831 100849 100853 100859 100867 100869 100879 100883 100889
100891 100893 100899 100901 100903 100907 100909 100919 100929 100931 100949 100953 100959 100967 100969 100979 100983 100989
100991 100993 100999 101001 101003 101007 101009 101019 101029 101031 101049 101053 101059 101067 101069 101079 101083 101089
101091 101093 101099 101101 101103 101107 101109 101119 101129 101131 101149 101153 101159 101167 101169 101179 101183 101189
101191 101193 101199 101201 101203 101207 101209 101219 101229 101231 101249 101253 101259 101267 101269 101279 101283 101289
101291 101293 101299 101301 101303 101307 101309 101319 101329 101331 101349 101353 101359 101367 101369 101379 101383 101389
101391 101393 101399 101401 101403 101407 101409 101419 101429 101431 101449 101453 101459 101467 101469 101479 101483 101489
101491 101493 101499 101501 101503 101507 101509 101519 101529 101531 101549 101553 101559 101567 101569 101579 101583 101589
101591 101593 101599 101601 101603 101607 101609 101619 101629 101631 101649 101653 101659 101667 101669 101679 101683 101689
101691 101693 101699 101701 101703 101707 101709 101719 101729 101731 101749 101753 101759 101767 101769 101779 101783 101789
101791 101793 101799 101801 101803 101807 101809 101819 101829 101831 101849 101853 101859 101867 101869 101879 101883 101889
101891 101893 101899 101901 101903 101907 101909 101919 101929 101931 101949 101953 101959 101967 101969 101979 101983 101989
101991 101993 101999 102001 102003 102007 102009 102019 102029 102031 102049 102053 102059 102067 102069 102079 102083 102089
102091 102093 102099 102101 102103 102107 102109 102119 102129 102131 102149 102153 102159 102167 102169 102179 102183 102189
102191 102193 102199 102201 102203 102207 102209 102219 102229 102231 102249 102253 102259 102267 102269 102279 102283 102289
102291 102293 102299 102301 102303 102307 102309 102319 102329 102331 102349 102353 102359 102367 102369 102379 102383 102389
102391 102393 102399 102401 102403 102407 102409 102419 102429 102431 102449 102453 102459 102467 102469 102479 102483 102489
102491 102493 102499 102501 102503 102507 102509 102519 102529 102531 102549 102553 102559 102567 102569 102579 102583 102589
102591 102593 102599 102601 102603 102607 102609 102619 102629 102631 102649 102653 102659 102667 102669 102679 102683 102689
102691 102693 102699 102701 102703 102707 102709 102719 102729 102731 102749 102753 102759 102767 102769 102779 102783 102789
102791 102793 102799 102801 102803 102807 102809 102819 102829 102831 102849 102853 102859 102867 102869 102879 102883 102889
102891 102893 102899 102901 102903 102907 102909 102919 102929 102931 102949 102953 102959 102967 102969 102979 102983 102989
102991 102993 102999 103001 103003 103007 103009 103019 103029 103031 103049 103053 103059 103067 103069 103079 103083 103089
103091 103093 103099 103101 103103 103107 103109 103119 103129 103131 103149 10315
```