# 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*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

```
SUBI:
    # 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)
    sub t2, t2, a1
    slli t3, a2, 2
    add t3, t0, t3
    sw t2, 0(t3)
    li a0, 0
    ret
    .size SUBI, .-SUBI
    .align 2
    .globl BG
    .type BG, @function
```

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

```
BG:
    # 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)
    slli t1, a1, 2
    add t1, t0, t1
    lw t3, 0(t1)
    slt a0, t3, t2
    ret
    .size   BG, .-BG
    .section    .rodata
    .align  2
```

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

2.2.4 ADD Function

```
ADD:
    # please start your code
    # r1:a0 r2:a1 rd:a3

    lui a6,%hi(registers)     #lui 只能够加载一个 20 位的立即数 (高 20 位)
    addi a6,a6,%lo(registers) #addi 用来加载低 12 位的地址部分

    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
    slli a1,a1,2
    slli a2,a2,2

    #a3 存储的是 registers 数组的起始地址
    add a0, a6, a0 #a0 现在保存的是 registers[a0] 的具体地址
    add a1, a6, a1 #registers[a1] 的地址
    add a2, a6, a2 #registers[a2] 的地址

    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
    lw a1,0(a1)

    add a6, a0, a1

    sw a6, 0(a2) #将 a3 中的值存储到地址 registers[a2] 中

    li a0, 0   #optional
    jr ra      #jr 是跳转指令，ra 是返回地址寄存器 (Return Address Register)

    .size   ADD, .-ADD
    .section    .rodata
    .align  2

```

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

```

BEQ:
# r1:a0 r2:a1

    lui a3,%hi(registers)
    addi a3,a3,%lo(registers)

    slli a0,a0,2
    slli a1,a1,2

    add a0, a3, a0
    add a1, a3, a1

    lw a0, 0(a0)
    lw a1, 0(a1)

    beq a0, a1, jump_for_beq

    li a0, 0
    jr ra

jump_for_beq:
    li a0, 1
    jr ra

    .size   BEQ, .-BEQ
    .section    .rodata
    .align  2


```

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

```
OR:
    lui a5,%hi(registers)
    addi a5,a5,%lo(registers)
    slli a0,a0,2
    slli a1,a1,2
    slli a2,a2,2
    add a0, a5, a0
    add a1, a5, a1
    add a2, a5, a2
    lw a0,0(a0)
    lw a1,0(a1)
    or a5, a0, a1
    sw a5, 0(a2)
    li a0, 0
    jr ra

    .size   OR, .-OR
    .section    .rodata
    .align  2
```

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

```
SRL:
# rs1:a0 shamt:a1 rd:a2

    lui a3,%hi(registers)
    addi a3,a3,%lo(registers)

    slli a2,a2,2
    slli a0,a0,2

    add a2, a3, a2
    add a0, a3, a0

    lw a0,0(a0)
    srl a3, a0, a1
    sw a3, 0(a2)

    li a0, 0
    jr ra

    .size   SRL, .-SRL
    .section    .rodata
    .align  2
```

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
```

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
```

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)).

```
judge:
    addi    sp,sp,-48
    sw  s0,44(sp)
    addi    s0,sp,48
    sw  a0,-36(s0)
    lw  a4,-36(s0)
    li  a5,1
    bgt a4,a5,.L2
    li  a5,0
    j   .L3
.L2:
    li  a5,2
    sw  a5,-20(s0)
    j   .L4
.L6:
    lw  a4,-36(s0)
    lw  a5,-20(s0)
    rem a5,a4,a5
    bne a5,zero,.L5
    li  a5,0
    j   .L3
.L5:
    lw  a5,-20(s0)
    addi    a5,a5,1
    sw  a5,-20(s0)
.L4:
    lw  a3,-20(s0)
    mul a4,a3,a3
    lw  a5,-36(s0)
    blt a5,a4,.L2_exit
    j   .L6
.L2_exit:
    li  a5,1
.L3:
    mv  a0,a5
    lw  s0,44(sp)
    addi    sp,sp,48
    jr  ra
    .size   judge, .-judge
    .section    .rodata
    .align  2
.LC0:
    .string "%d"
    .text
    .align  2
    .globl  main
    .type   main, @function
main:
    addi    sp,sp,-32
    sw  ra,28(sp)
    sw  s0,24(sp)
    addi    s0,sp,32
    li  a5,1000000
    sw  a5,-24(s0)
    li  a5,2
    sw  a5,-20(s0)
    j   .L8
```

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.

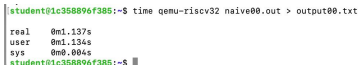
(2)Expanding the Range: The task also requires expanding the range of numbers the program operates on—from 1 to 1000 to 1 to 10000. This involves modifying the loop or range bounds to handle the larger number set. I made the necessary adjustments to the code to ensure that it could now handle numbers 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

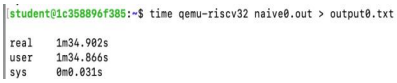
before optimization:




after optimization:


3.2.2 on expanded sequence range

before optimization:




after optimization: