

# Report

## 1. ALU

### 1.1. Ideas:

Implementing these functions involves loading values into registers, performing the operation, and storing the result back into a register.

### 1.2. Implementation Details:

For and, dub, nor, or, xor, slt, sll, srl, sra, just simply load, perform the operation, and store.

For beq and bne, use **sub** instruction. By subtracting two registers and checking if the result is zero, it was possible to determine equality or inequality and perform the appropriate branch.

### 1.3. Screenshots of Results:

```
misaka@zsh:~/桌面/Project1_part1$ ./alu
test1 about add
success
test2 about add
success
test3 about add
success
test4 about sub
success
test5 about sub
success
test6 about and
success
test7 about nor
success
test8 about or
success
test9 about xor
success
test10 about beq
success
test11 about beq
```

```
success
test12 about bne
success
test13 about bne
success
test14 about slt
success
test15 about slt
success
test16 about slt
success
test17 about sll
success
test18 about sll
success
test19 about srl
success
test20 about srl
success
test21 about sra
success
test22 about sra
success
congratulation! Pass all tests.
```

### 1.4. Problems and Solutions

I first implemented the nor function, I used nor t4, t1, t2, however, there was no RISC-V instruction for “nor”, I solved this by using “or” and “xor”.

```
alu.s: Assembler messages:
alu.s:91: 错误: unrecognized opcode `nor t4,t1,t2'
```

## 2. Branch

### 2.1 Ideas:

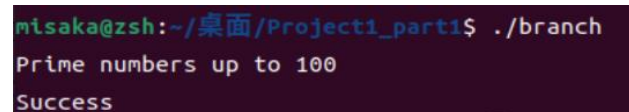
The implementation involved setting up an outer loop to iterate through numbers

from 2 to n and an inner loop to check for factors of each number. Use branch instructions to exit the inner loop early if a factor was found.

## 2.2. Implementation Details:

- Array Management: the 'primes' array is managed using the pointer 't5', which is incremented each time a new prime number is found and stored.
- Control Flow: branch instructions (bgt, beqz, j) are used to control the flow of the loops and the prime checking logic.
- Stack Management: save and restore caller-saved registers (ra, t1, t2, and t5) on the stack around the call instruction to ensure that values of registers do not change across function calls.
- Use of Saved Registers: instead of directly using *bgt t1, a0, end\_i\_loop* in the *i\_loop*, I move the value of a0 (size) into s0 and then set *bgt t1, s0, end\_i\_loop*, since a0, as a function argument register, may be changed by function calls.

## 2.3. Result:.



```
misaka@zsh:~/桌面/Project1_part1$ ./branch
Prime numbers up to 100
Success
```

## 2.4. Problems and Solutions:

At first, I frequently encountered segmented fault or the problem of no output. The reason is the wrong implementation of "save and restore register" process: forgot to save and load **ra**, allocated the space of stack wrongly, did not match the save and store sequence... By solving this problem, I have a better understand of stack space and use of callee-saved, caller-saved registers.

# 3. Array

## 3.1. Ideas:

Use three loops to iterate over the bitmap array. The outer loops (*i\_loop* and *j\_loop*) iterate over the 2D array, while the inner loop (*k\_loop*) iterates over each bit within a byte. This approach breaks down the problem of printing a bitmap into smaller, manageable tasks:

- Byte Iteration: Traverse each byte in the bitmap array.
  - Bit Extraction: Extract each bit from the current byte.
  - Bit Printing: If the extracted bit is '1', print '1', otherwise print a space (' ').
- Between every two bytes, print a space (' '). After each iteration of *i\_loop*, change a line.

## 3.2. Implementation Details:

- Bitwise Operations: use **sll** instruction to get a mask code, use **and** instruction along with the mask code to extract individual bits from a byte.
- Control Flow: use branch instructions (bge, blt, bnez, and j) to control the flow of

the loops and conditionally execute code.

- Stack Management: Save and restore caller-saved registers (ra, t1, t2, t3, and t5) on the stack around the call instruction to ensure that values of registers do not change across function calls.

### 3.3. Result:

```
misaka@zsh:~/桌面/Project1_part1$ ./array
 1      1      1      1      1      1
 1      1      1      1      1      1111
1111111 1111 11111 111111 1      1      11111
      1      1      1      1 11 11111 1
1 1 1      1      1      11 1 1 1 1
 1 1 1      1      1      1 1 1 1 1 1 1 1
11111111 111111 11111 11111 1 1 1 1 1 1
 1      1 1      1      1      1 1 1 1 111111 111111
 1      1 1      1      1      1 1 111 1111111 1
11111111 1 1      1      1      1      1      1 1 1 1
 1      1 1      11111111 1111111 1      1 1      1 1 1
 1 1 1 1 1      1      1      1 1      1 1 1
1 1 1 1 1      1      1 1 1 1 1 1 1 1
1 1 1 1 1      1      1 1 1 1 1 1 1 1
 1 1 1 1      1      1 1 1 1 1 1 1 1
 1 1 1      1      1      1 1      11 1
```

### 3.4. Problems and Solutions:

a.

```
array.s: Assembler messages:
array.s:18: 错误: illegal operands `bge t2,8,j_end'
array.s:21: 错误: illegal operands `blt t3,0,k_end'
```

I overlooked the fact that both **bge** and **blt** instructions take registers instead of intermediate. I solved this by using registers to store the value 0, 8. Enhanced my understanding of these instructions.

b.

```
misaka@zsh:~/桌面/Project1_part1$ ./array
 1      1      11111 1      1 1      1 1 1 1 1
1111111 11111 1 1 1 1 1 1 1 1 1 1 1 1
 1      1
```

I wrongly set the increment of t5 at the inside of k\_loop. Solved by set it at the inside of j\_loop but outside k\_loop.