1. Write a RISC-V assembly program to calculate C=A×B+DC = A \times B + DC=A×B+D, where A,B,A, B,A,B, and DDD are given as inputs in registers x10, x11, and x12. Store the result in x13.

   RISC-V assembly code for C = A * B + D

   ```
   mul x13, x10, x11    # x13 = A * B

   add x13, x13, x12    # x13 = x13 + D (C = A * B + D)
   ```

2. Implement a RISC-V assembly program that reverses the elements of an array stored in memory. The base address of the array is in x10, and the size of the array is in x11.

   RISC-V Assembly Code to Reverse an Array:

   ```
   mv t0, x10         # t0 = base address of the array (start pointer)

   add t1, x10, x11 # t1 = base address + size (end pointer, points to one past the end)

   addi t1, t1, -4    # t1 = last element address (end pointer adjusted for zero indexing)

   reverse_loop:

   bge t0, t1, done # If start pointer >= end pointer, exit loop

   lw t2, 0(t0)       # Load the element at the start pointer into

   t2 lw t3, 0(t1)   # Load the element at the end pointer into t3

   sw t3, 0(t0)    # Store the element from the end pointer into the start pointer

   sw t2, 0(t1) # Store the element from the start pointer into the end pointer

   addi t0, t0, 4  # Move start pointer to the next element

   addi t1, t1, -4   # Move end pointer to the previous element

   j reverse_loop # Repeat the loop

   done:

   nop                # No operation (end of program)
   ```

3. Write a RISC-V program to determine if a number NNN (stored in x10) is a prime number. If NNN is prime, store 1 in x11; otherwise, store 0.

   RISC-V Assembly Code to Determine if N is Prime

```
 mv      t0, x10          # t0 = N (the number to check for primality)
   li      t1, 2            # t1 = 2 (start checking divisibility from 2)

   # Check if N is less than 2 (not prime)
   blt     t0, t1, not_prime   # if N < 2, it's not prime (store 0)

   # Special case for N = 2 (prime)
   li      t2, 2            # t2 = 2
   beq     t0, t2, is_prime    # if N == 2, it's prime (store 1)

   # Set t3 = sqrt(N) using integer division approximation
   # We will check divisibility up to sqrt(N). For simplicity, we use an approximation method.
   # Instead of calculating sqrt directly, we use a loop to check up to N / 2.

   div     t3, t0, t1          # t3 = N / 2 (approximation of sqrt(N), but safe for prime check)
   addi    t3, t3, 1           # we check from 2 to N / 2

check_loop:
   # t1 = divisor candidate (2, 3, 4,...)
   div     t4, t0, t1          # t4 = N / t1 (quotient)
   mul     t5, t4, t1          # t5 = t4 * t1 (check if t4 * t1 == N)
   beq     t5, t0, not_prime   # if N % t1 == 0, N is not prime (store 0)

   addi    t1, t1, 1           # increment divisor by 1
   bge     t1, t3, is_prime    # if t1 >= sqrt(N), N is prime (store 1)

   j       check_loop          # repeat loop

not_prime:
   li      x11, 0           # N is not prime, store 0 in x11
   j       done

is_prime:
   li      x11, 1           # N is prime, store 1 in x11

done:
   nop                      # end of program
```

4. Develop a RISC-V program to calculate the Fibonacci sequence up to the n-th term, where n is given in x10. Store the result in memory starting at a base address provided in x11.

RISC-V Assembly Code for Fibonacci Sequence

```
mv      t0, x10             # t0 = n (the number of
  terms in Fibonacci sequence)
 mv      t1, x11             # t1 = base address to
  store the results

  li      t2, 0               # t2 = F(0) = 0 (first
  Fibonacci number)
 li      t3, 1               # t3 = F(1) = 1 (second
  Fibonacci number)

  # Store F(0) at base address
  sw      t2, 0(t1)         # store F(0) at address t1

 # If n > 0, store F(1) at base address + 4 (next
  memory location)
  addi    t1, t1, 4
  sw      t3, 0(t1)         # store F(1) at address t1

  # Check if n is greater than 1, otherwise we
  are done
  bge     t0, t2, fib_loop   # if n >= 2, enter loop

  j       done              # if n == 1, we are done
```

```
fib_loop:
    # We are in the loop to calculate Fibonacci
    from F(2) to F(n)
    li      t4, 2           # t4 = i (loop counter,
    starting at 2)

loop_start:
    # Calculate F(i) = F(i-1) + F(i-2)
    add     t5, t2, t3      # t5 = F(i-1) + F(i-2)

    # Store the result in memory
    addi    t1, t1, 4       # increment the address
    by 4 to store next value
    sw      t5, 0(t1)       # store F(i) at address t1

    # Update F(i-2) = F(i-1) and F(i-1) = F(i)
    mv      t2, t3          # t2 = F(i-1) (previous
    Fibonacci number)
    mv      t3, t5          # t3 = F(i) (current
    Fibonacci number)

    addi    t4, t4, 1       # increment the loop
    counter
    bge     t4, t0, done    # if i >= n, end the loop
    j       loop_start      # repeat the loop
```

```
done:
    nop                 # end of program
```

5. Explain the role of the R-type and I-type instruction formats in RISC-V. How are these formats structured, and how do they differ in terms of operand usage and operations (e.g., arithmetic vs. immediate values)?

R-type

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

R-type instruction machine code format

There are 10 R-type instructions in total. The opcode name is OP and the value is 7'b011_0011. Small instructions are determined according to funct3, and if it is not enough, funct7 is used to determine. As shown in Figure 5-1, it can be found that it is similar to the immediate instruction.

1. ADD

rd,rs1,rs2 addition

funct7==7'b0000000;

funct3==3'b000;

This instruction writes the result of rs1+rs2 into rd, ignoring overflow (overflow can be handled by software).

2.SLT rd, rs1,rs2

Funct7==7'b0000000;

funct3==3'b010 ;

rs1 and rs2 are compared with signed numbers, if rs1<rs2, rd is set to 1, otherwise it is set to 0

3.SLTU rd,rs1,rs2

Funct7==7'b0000000;

funct3==3'b011 ;

Use the unsigned number to compare rs1 and rs2, if rs1<rs2, rd is set to 1, otherwise it is set to 0

4. AND

rd,rs1,rs2

funct7==7'b0000000;

funct3==3'b111;

Write the result of rs1&rs2 into rd, where & means that rs1 and rs2 are AND together bit by bit

5. OR rd,

rs1,rs2

funct7==7'b0000000,

funct3==3'b110

Write the result of rs1|rs2 into rd, where | means or

6. XOR

rd,rs1,rs2

funct7==7’b0000000,

funct3==3’b100 ;

Write the result of rs1^rs2 to rd, where ^ means rs1 XOR rs2 bit by bit

7. SLL

rd,rs1,rs2 Logical

shift left

Funct7==7’b0000000,

Funct3==3’b001;

Shift rs1 logically to the left according to the number specified in the lower 5 bits of rs2, fill in the lower bits with zeros, and write the result to rd

8. SRL rd,

rs1,rs2 Logical shift

right

Funct7==7’b0000000,

funct3==3’b101 ;

rs1 is logically shifted to the right according to the specified number of low 5 bits in rs2, and the high bits of rs1 are filled with zeros, and the result is written into rd

9. SRA rd,rs1,rs2

Arithmetic shift right

Funct7==7’b010_0000,

funct3==3’b101 ;

Arithmetic shift rs1 to the right according to the specified number of low 5 bits in rs2, the high bit is determined by the original rs1[31], and the result is written into rd. Note: When shifting, only the value of RS1 is copied to the temporary variable for shifting, and the original value remains constant.

10. SUB rd,rs1,rs2

Subtraction

funct7==7’b010_0000,

funct3==3’b000;

Write the result of rs1-rs2 to rd, ignore overflow (overflow can be handled by software)

The only difference between addition and subtraction instructions is funct7

Addition funct7==7’b000_0000

Subtraction funct7==7’b010_0000

I-type

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

I-type format

There are 15 instructions in total for I-type. Now introduce the first 6 instructions. Please refer to Figure 3-1 for I-type command format. The opcode corresponding to the I-type instruction is named OP-IMM. It means immediate operation code (IMM: immediate).

The immediate opcode OP-IMM==7’b001_0011. When opcode==OP-IMM==7’b001_0011, it proves that the instruction is an I-type instruction, and the specific behavior of this instruction is determined by the value of funct3.

The following part describes the instructions in I-type one by one.

ADDI：ADDI rd,rs1,imm[11:0]

ADD stands for addition, I stands for immediate, and the ADDI instruction means addition with immediate. This instruction adds the value in the rs1 register to the immediate, and then stores the result of the addition in rd. ADDI corresponds to funct3 == 3’b000, which means that funct3==3’b0000 in the I-type instruction means that the instruction is an ADDI instruction.

A new concept pseudo-instruction is introduced here, taking the MV instruction as an example.

MV：MV rd,rs1

MV (move) instruction. This instruction is to move the value in rs1 to rd. There is a move instruction in x86 and also in mcs51. What are pseudo instructions? Pseudo instructions are instructions that do not exist in the assembly instruction set. These instructions are convenient for assembly programmers and are often used. For example, in the assembly program, there are often shifts between registers. So the MV instruction is often used. Take the MV instruction as an example, its actual meaning is ADDI rd, rs1,0. That is to say, the value in the rs1 register is added up with the immediate 0 , and finally stored in the rd register. Because the value of rs1 does not change after rs1 is added to zero, the MV instruction moves the value of rs1 to rd. The programmer can write such a pseudo-instruction MV when writing the assembler. The compiler software will translate this instruction into ADDI rd, rs1, 0 when the program is compiled, and then send the ADDI instruction to the CPU to run.

SLTI：SLTI rd,rs1,imm[11:0]

In the second instruction SLTI, S stands for Set, and its role is to set rd. Set the value in the rd register to 0 or 1. It should be noted that in the instruction, if the condition is satisfied, the bit is set to 1, and if the condition is not satisfied, the bit is set to 0. The setting condition of this instruction is LT: less than. I stands for immediate. So the judgment condition is whether the value of rs1 is less than the immediate. Because the rs1 register is 32-bit in all operations, and the immediate in this instruction is 12-bit, the immediate must be expanded first, and then the operation will be performed after the expansion. Note that the extensions here are all signed extensions.

Here is an example to explain the signed extension. For example, there is a 12-bit immediate. If the highest bit is 0, it means that the immediate is positive. If the highest bit is 1, it means that the immediate is negative. When the positive number is sign-extended, the upper 20 bits are all filled with 0, and when the negative number is sign-extended, the upper 20 bits are all filled with 1.

SLTIU：SLTIU rd,rs1,imm[11:0]

In the instruction SLTIU, U represents an unsigned number. It means to set after comparing unsigned numbers.

Example: Compare 8-bit binary numbers. -1: 8’b1111_1111, -2:8’b1111_1110. When compared as a signed number, -2<-1, if it is used as an unsigned number, the comparison is still valid, but if you use -2 and +1 8’b0000_0001 to compare an unsigned number. At this time, 1111_1110>0000_0001. (254>1)

Pseudo-instruction SEQZ：SEQZ rd,rs1 (SLTIU rd,rs1,1)

If rs1==0, then rd is set to 1.

This pseudo-instruction is a special case of SLTIU and will be used frequently.

ANDI: ANDI rd,rs1,imm[11:0]

The immediate is expanded to 32 bits as a signed number, then perform AND with rs1 after the extension, and store the result in rd.

ORI: ORI rd,rs1,imm[11:0]

The immediate is expanded to 32 bits as a signed number and do OR with rs1. The result is written into rd.

XORI: XORI rd,rs1,imm[11:0]

The immediate is expanded to 32 bits as a signed number and do XOR with rs1. The result is written to rd.

The format reflects the flexibility of instructions. The format does not mandate which register rs1 and rd are, and programmers need to select registers from 32 general-purpose registers when writing assembly programs.

Pseudo-instruction NOT：NOT rd,rs1 (XORI rd,rs1,12'hfff)

NOT is a negation instruction. The function is to reverse the value in rs1 and store it in rd. It will be translated into (XORI rd,rs1,12'hffff)

The assembly instruction format and corresponding funct3 are shown in Figure 3-2 below

Figure:some I-type assembly instructions and funct3 machine code

Shift instruction

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

Figure:Shift instruction

1.SLLI rd,rs1,shamt[4:0]

SLLI ：shift left logical by shamt(shift amount)， The number of shifts is determined by imm “4:0”. This instruction means to shift the value of rs1 to the left in shamt[4:0], use 0 to fill the low bits of rs1, and write the result to Rd.

2. SRLI rd,rs1,shamt[4:0]

SRLI：shift right logical .Shift the value in rs1 to the right by shamt[4:0] bits, add zero to the high bits of rs1, and write the result to rd.

3. SRAI rd,rs1,shamt[4:0]

SRAI：Arithmetic shift right, shift the value in rs1 to right by shamt[4:0] bits, the high bits of rs1 are filled with the original rs1[31], and the result is written into rd.

Comparison of R-Type and I-Type

Aspect	**R-Type**	**I-Type**
Operands	Two registers (rs1 and rs2)	One register (rs1) and one immediate (imm).
Result Storage	Register (rd).	Register (rd).
Usage	Arithmetic, logical, shift (register-register).	Arithmetic with constants, memory access, control flow.
Example Operations	add, sub, and, sll.	addi, lw, jalr, slti.

6. In RISC-V, the S-type and B-type instructions are used for different operations. Describe the key differences between these instruction types and provide examples of how they are used in store operations and branch instructions.

S-Type Instructions

Purpose:

S-type instructions are used for **store operations**, where data is written from a register to memory. These instructions involve:

- A memory address computed using a base address and an immediate offset.
- A value stored from a source register into the computed memory address.

Structure:

Field	Size (bits)	Purpose
opcode	7	Specifies the operation (store type).
imm[4:0]	5	Lower 5 bits of the immediate value.
rs1	5	Base address register.
rs2	5	Source register containing the data to store.
funct3	3	Specifies the store operation (e.g., byte, word).
imm[11:5]	7	Upper 7 bits of the immediate value.

Key Characteristics:

- Combines the immediate into a single 12-bit signed value: imm[11:5] concatenated with imm[4:0].
- The effective memory address is calculated as rs1 + imm.

Example:

assembly

Copy code

```
sw x5, 12(x1)  # Store the word in x5 to the memory address x1 + 12
```

- **opcode** specifies a store operation.
- **rs1** is x1, the base address register.
- **rs2** is x5, the register containing the data.
- **imm** is 12.

B-Type Instructions

Purpose:

B-type instructions are used for **branch operations**, enabling conditional jumps based on comparisons between two registers. These instructions involve:

- Comparing the values of two registers.
- Calculating a target address using an immediate offset for the branch.
- **Structure:**

Field	**Size (bits)**	**Purpose**
opcode	7	Specifies the operation (branch type).
imm[11]	1	Most significant bit of the immediate
imm[4:1]	1	Bits 4 to 1 of the immediate.
Funct3	3	Specifies the branch operation (e.g., equal, less than)
rs1	5	First source register for comparison
rs2	5	Second source register for comparison
funct3	3	Specifies the store operation (e.g., byte, word).
imm[10:5]	7	Middle bits of the immediate.

Key Differences Between S-Type and B-Type

Feature	S-Type	B-Type
Purpose	Store data from a register to memory.	Perform conditional branching.

Immediate Value	Split across imm[4:0] and imm[11:5].	Split across imm[11], imm[4:1], imm[10:5], and imm[12].
Usage Examples	sw, sh, sb (store operations).	beq, bne, blt, bge (branch instructions).
Target Address	Calculated as rs1 + imm.	Calculated as PC + imm.