RISC V Assignments -17/11/2024

NARAHARI SUMANTH

RSAGWPVLSI14

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.**

   C CODE:

```c
#include <stdio.h>

int main() {
    // Simulating RISC-V registers
    int x10 = 5;  // A, input in x10
    int x11 = 3;  // B, input in x11
    int x12 = 2;  // D, input in x12
    int x13;      // Output, result stored in x13

    // Perform the operation C = A * B + D
    x13 = (x10 * x11) + x12;

    // Output the result, which is stored in x13
    printf("The result (stored in x13) is: %d\n", x13);

    return 0;
}
```

HERE IS THE ASSEMBLY CODE FOR RISCV

USING GODBOLT COMPILER I CONVERTED C CODE INTO ASSEMBLY CODE

```
.LC0:
        .string "The result (stored in x13) is: %d\n"
main:
        addi    sp,sp,-32
        sw      ra,28(sp)
        sw      s0,24(sp)
        addi    s0,sp,32
        li      a5,5
        sw      a5,-20(s0)
        li      a5,3
        sw      a5,-24(s0)
        li      a5,2
        sw      a5,-28(s0)
```

```
        lw      a4,-20(s0)
        lw      a5,-24(s0)
        mul     a5,a4,a5
        lw      a4,-28(s0)
        add     a5,a4,a5
        sw      a5,-32(s0)
        lw      a1,-32(s0)
        lui     a5,%hi(.LC0)
        addi    a0,a5,%lo(.LC0)
        call    printf
        li      a5,0
        mv      a0,a5
        lw      ra,28(sp)
        lw      s0,24(sp)
        addi    sp,sp,32
        jr      ra
```

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.**

HERE IS THE C CODE

```c
#include <stdio.h>

void reverse_array(int* x10, int x11) {
    int start = 0;
    int end = x11 - 1;

    // Reverse the array using two-pointer technique
    while (start < end) {
        // Swap the elements at start and end
        int temp = x10[start];
        x10[start] = x10[end];
        x10[end] = temp;

        // Move the pointers
        start++;
        end--;
    }
}

int main() {
    // Example array
    int x10[] = {1, 2, 3, 4, 5};
    int x11 = sizeof(x10) / sizeof(x10[0]);

    // Reverse the array
    reverse_array(x10, x11);
```

```
    // Print the reversed array
    printf("Reversed array: ");
    for (int i = 0; i < x11; i++) {
        printf("%d ", x10[i]);
    }
    printf("\n");

    return 0;
}
```

HERE IS THE ASSEMBLY CODE FOR RISCV

USING GODBOLT COMPILER I CONVERTED C CODE INTO ASSEMBLY CODE

```
reverse_array:
        addi    sp,sp,-48
        sw      ra,44(sp)
        sw      s0,40(sp)
        addi    s0,sp,48
        sw      a0,-36(s0)
        sw      a1,-40(s0)
        sw      zero,-20(s0)
        lw      a5,-40(s0)
        addi    a5,a5,-1
        sw      a5,-24(s0)
        j       .L2
.L3:
        lw      a5,-20(s0)
        slli    a5,a5,2
        lw      a4,-36(s0)
        add     a5,a4,a5
        lw      a5,0(a5)
        sw      a5,-28(s0)
        lw      a5,-24(s0)
        slli    a5,a5,2
        lw      a4,-36(s0)
        add     a4,a4,a5
        lw      a5,-20(s0)
        slli    a5,a5,2
        lw      a3,-36(s0)
        add     a5,a3,a5
        lw      a4,0(a4)
        sw      a4,0(a5)
        lw      a5,-24(s0)
        slli    a5,a5,2
        lw      a4,-36(s0)
        add     a5,a4,a5
        lw      a4,-28(s0)
```

```
        sw      a4,0(a5)
        lw      a5,-20(s0)
        addi    a5,a5,1
        sw      a5,-20(s0)
        lw      a5,-24(s0)
        addi    a5,a5,-1
        sw      a5,-24(s0)
.L2:
        lw      a4,-20(s0)
        lw      a5,-24(s0)
        blt     a4,a5,.L3
        nop
        nop
        lw      ra,44(sp)
        lw      s0,40(sp)
        addi    sp,sp,48
        jr      ra
.LC1:
        .string "Reversed array: "
.LC2:
        .string "%d "
.LC0:
        .word   1
        .word   2
        .word   3
        .word   4
        .word   5
main:
        addi    sp,sp,-48
        sw      ra,44(sp)
        sw      s0,40(sp)
        addi    s0,sp,48
        lui     a5,%hi(.LC0)
        addi    a5,a5,%lo(.LC0)
        lw      a1,0(a5)
        lw      a2,4(a5)
        lw      a3,8(a5)
        lw      a4,12(a5)
        lw      a5,16(a5)
        sw      a1,-44(s0)
        sw      a2,-40(s0)
        sw      a3,-36(s0)
        sw      a4,-32(s0)
        sw      a5,-28(s0)
        li      a5,5
        sw      a5,-24(s0)
        addi    a5,s0,-44
        lw      a1,-24(s0)
```

```
        mv      a0,a5
        call    reverse_array
        lui     a5,%hi(.LC1)
        addi    a0,a5,%lo(.LC1)
        call    printf
        sw      zero,-20(s0)
        j       .L5
.L6:
        lw      a4,-20(s0)
        addi    a5,s0,-44
        slli    a4,a4,2
        add     a5,a4,a5
        lw      a5,0(a5)
        mv      a1,a5
        lui     a5,%hi(.LC2)
        addi    a0,a5,%lo(.LC2)
        call    printf
        lw      a5,-20(s0)
        addi    a5,a5,1
        sw      a5,-20(s0)
.L5:
        lw      a4,-20(s0)
        lw      a5,-24(s0)
        blt     a4,a5,.L6
        li      a0,10
        call    putchar
        li      a5,0
        mv      a0,a5
        lw      ra,44(sp)
        lw      s0,40(sp)
        addi    sp,sp,48
        jr      ra
```

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.**

C CODE:

```c
#include <stdio.h>

int main() {
    // Simulating RISC-V registers
    int x10 = 29; // Input number N (stored in x10)
    int x11 = 1;  // Output register (1 for prime, 0 for not prime)

    // Check if x10 (N) is a prime number
```

```
    if (x10 <= 1) {
        x11 = 0; // Numbers <= 1 are not prime
    } else {
        for (int i = 2; i * i <= x10; i++) {
            if (x10 % i == 0) {
                x11 = 0; // Not a prime number
                break;
            }
        }
    }

    // Output the result in x11
    printf("x10 (N) = %d, Prime status (x11): %d\n", x10, x11);

    return 0;
}
```

HERE IS THE ASSEMBLY CODE FOR RISCV

USING GODBOLT COMPILER I CONVERTED C CODE INTO ASSEMBLY CODE

```
.LC0:
        .string "x10 (N) = %d, Prime status (x11): %d\n"
main:
        addi    sp,sp,-32
        sw      ra,28(sp)
        sw      s0,24(sp)
        addi    s0,sp,32
        li      a5,29
        sw      a5,-28(s0)
        li      a5,1
        sw      a5,-20(s0)
        lw      a4,-28(s0)
        li      a5,1
        bgt     a4,a5,.L2
        sw      zero,-20(s0)
        j       .L3
.L2:
        li      a5,2
        sw      a5,-24(s0)
        j       .L4
.L6:
        lw      a4,-28(s0)
        lw      a5,-24(s0)
        rem     a5,a4,a5
        bne     a5,zero,.L5
        sw      zero,-20(s0)
        j       .L3
.L5:
```

```
        lw      a5,-24(s0)
        addi    a5,a5,1
        sw      a5,-24(s0)
.L4:
        lw      a5,-24(s0)
        mul     a5,a5,a5
        lw      a4,-28(s0)
        bge     a4,a5,.L6
.L3:
        lw      a2,-20(s0)
        lw      a1,-28(s0)
        lui     a5,%hi(.LC0)
        addi    a0,a5,%lo(.LC0)
        call    printf
        li      a5,0
        mv      a0,a5
        lw      ra,28(sp)
        lw      s0,24(sp)
        addi    sp,sp,32
        jr      ra
```

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.**

```c
#include <stdio.h>
#include <stdlib.h>  // Include this header for malloc and free

int main() {
   // Simulating RISC-V registers
   int x10 = 10;   // Input number n (stored in x10) for Fibonacci sequence
   int *x11 = (int*)malloc(x10 * sizeof(int));  // Base address in x11 for storing Fibonacci sequence

   // Check if x11 (memory) is valid
   if (x11 == NULL) {
      printf("Memory allocation failed!\n");
      return 1;
   }

   // Base cases
   if (x10 > 0) {
      x11[0] = 0;  // F(0)
   }
   if (x10 > 1) {
      x11[1] = 1;  // F(1)
   }
```

```
    // Calculate Fibonacci sequence up to the n-th term
    for (int i = 2; i < x10; i++) {
        x11[i] = x11[i - 1] + x11[i - 2];
    }

    // Output the Fibonacci sequence
    printf("Fibonacci sequence up to term %d:\n", x10);
    for (int i = 0; i < x10; i++) {
        printf("F(%d) = %d\n", i, x11[i]);
    }

    // Free allocated memory
    free(x11);

    return 0;
}
```

RISCV 32

HERE IS THE ASSEMBLY CODE FOR RISCV

USING GODBOLT COMPILER I CONVERTED C CODE INTO ASSEMBLY CODE

```
.LC0:
        .string "Memory allocation failed!"
.LC1:
        .string "Fibonacci sequence up to term %d:\n"
.LC2:
        .string "F(%d) = %d\n"
main:
        addi    sp,sp,-32
        sw      ra,28(sp)
        sw      s0,24(sp)
        addi    s0,sp,32
        li      a5,10
        sw      a5,-28(s0)
        lw      a5,-28(s0)
        slli    a5,a5,2
        mv      a0,a5
        call    malloc
        mv      a5,a0
        sw      a5,-32(s0)
        lw      a5,-32(s0)
        bne     a5,zero,.L2
        lui     a5,%hi(.LC0)
```

```
    addi    a0,a5,%lo(.LC0)
    call    puts
    li      a5,1
    j       .L3
.L2:
    lw      a5,-28(s0)
    ble     a5,zero,.L4
    lw      a5,-32(s0)
    sw      zero,0(a5)
.L4:
    lw      a4,-28(s0)
    li      a5,1
    ble     a4,a5,.L5
    lw      a5,-32(s0)
    addi    a5,a5,4
    li      a4,1
    sw      a4,0(a5)
.L5:
    li      a5,2
    sw      a5,-20(s0)
    j       .L6
.L7:
    lw      a4,-20(s0)
    li      a5,1073741824
    addi    a5,a5,-1
    add     a5,a4,a5
    slli    a5,a5,2
    lw      a4,-32(s0)
    add     a5,a4,a5
    lw      a3,0(a5)
    lw      a4,-20(s0)
    li      a5,1073741824
    addi    a5,a5,-2
    add     a5,a4,a5
    slli    a5,a5,2
    lw      a4,-32(s0)
    add     a5,a4,a5
    lw      a4,0(a5)
    lw      a5,-20(s0)
    slli    a5,a5,2
    lw      a2,-32(s0)
    add     a5,a2,a5
    add     a4,a3,a4
    sw      a4,0(a5)
    lw      a5,-20(s0)
    addi    a5,a5,1
    sw      a5,-20(s0)
.L6:
```

```
    lw      a4,-20(s0)
    lw      a5,-28(s0)
    blt     a4,a5,.L7
    lw      a1,-28(s0)
    lui     a5,%hi(.LC1)
    addi    a0,a5,%lo(.LC1)
    call    printf
    sw      zero,-24(s0)
    j       .L8
.L9:
    lw      a5,-24(s0)
    slli    a5,a5,2
    lw      a4,-32(s0)
    add     a5,a4,a5
    lw      a5,0(a5)
    mv      a2,a5
    lw      a1,-24(s0)
    lui     a5,%hi(.LC2)
    addi    a0,a5,%lo(.LC2)
    call    printf
    lw      a5,-24(s0)
    addi    a5,a5,1
    sw      a5,-24(s0)
.L8:
    lw      a4,-24(s0)
    lw      a5,-28(s0)
    blt     a4,a5,.L9
    lw      a0,-32(s0)
    call    free
    li      a5,0
.L3:
    mv      a0,a5
    lw      ra,28(sp)
    lw      s0,24(sp)
    addi    sp,sp,32
    jr      ra
```

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)?**

In the RISC-V instruction set architecture, the **R-type** and **I-type** formats are used for different types of operations, specifically in how they handle operands and perform various tasks like arithmetic or immediate value operations. Let's explore each type, their structure, and the differences in operand usage.

1. R-type Instructions (Register-to-Register Operations):

R-type instructions are used for operations where all operands are registers. These instructions generally perform **arithmetic, logical, and comparison** operations between two registers. The result is stored in a destination register.

R-type Instruction Format:

The R-type instruction format is used for operations that involve two source registers and a destination register. It is structured as follows:

- **funct7**: 7 bits (used for some instructions to determine the exact operation, like signed or unsigned operations, shift operations, etc.)
- **rs2**: 5 bits (source register 2, used as the second operand)
- **rs1**: 5 bits (source register 1, used as the first operand)
- **funct3**: 3 bits (defines the operation to perform between rs1 and rs2)
- **rd**: 5 bits (destination register, where the result is stored)
- **opcode**: 7 bits (indicates the instruction type, e.g., ADD, SUB, AND, etc.)

R-type Examples:

- **ADD** (Addition):

  arduino
  Copy code
  ADD rd, rs1, rs2 // rd = rs1 + rs2

  This performs an addition between the values in registers rs1 and rs2, storing the result in rd.

- **SUB** (Subtraction):

  arduino
  Copy code
  SUB rd, rs1, rs2 // rd = rs1 - rs2

  This subtracts the value in rs2 from rs1 and stores the result in rd.

- **AND** (Logical AND):

  arduino
  Copy code

```
AND rd, rs1, rs2  // rd = rs1 & rs2
```

This performs a bitwise AND operation between rs1 and rs2, with the result stored in rd.

R-type Operand Usage:

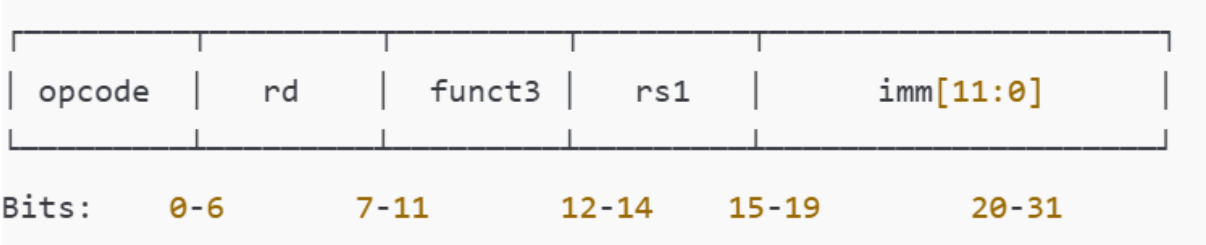
- Operands are taken from registers: **rs1** and **rs2**.
- The result is placed into the **rd** register.
- **Arithmetic** operations like addition, subtraction, multiplication, etc., are typical in R-type instructions.

2. I-type Instructions (Immediate Operand Operations):

I-type instructions involve **immediate values** (constants directly encoded within the instruction) in addition to registers. These are used for operations that involve one register and one immediate value (a constant). The immediate value is encoded within the instruction and is sign-extended to the appropriate size during execution.

I-type Instruction Format:

The I-type instruction format is used for operations where one operand is a register and the other is an immediate value. The format is as follows:



- **imm[11:0]**: 12 bits (immediate value, which is directly provided in the instruction)
- **rs1**: 5 bits (the source register, which is involved in the operation)
- **funct3**: 3 bits (defines the operation to perform, such as addition, comparison, etc.)
- **rd**: 5 bits (destination register, where the result is stored)
- **opcode**: 7 bits (indicates the instruction type)

I-type Examples:

- **ADDI** (Add Immediate):

```
arduino
Copy code
ADDI rd, rs1, imm  // rd = rs1 + imm
```

This adds an immediate value imm to the value in register rs1 and stores the result in rd.

- **LW** (Load Word):

  scss
  Copy code
  LW rd, imm(rs1) // rd = memory[rs1 + imm]

  This loads a word from memory, where the address is calculated as rs1 + imm.

- **BEQ** (Branch if Equal):

  arduino
  Copy code
  BEQ rs1, rs2, imm // If rs1 == rs2, branch to address (PC + imm)

  This compares the values in registers rs1 and rs2. If they are equal, the program branches to the address PC + imm.

I-type Operand Usage:

- Operands include a **register** (rs1) and an **immediate value** (imm).
- The result is stored in the **rd** register, except for instructions like **LW**, where the result is loaded into memory.

Key Differences Between R-type and I-type Instructions:

Aspect	R-type Instructions	I-type Instructions
Operands	Two registers and one destination register (rs1, rs2, rd)	One register and one immediate value (rs1, imm, rd)
Use of Immediate	No immediate value (operates purely on registers)	Uses an immediate value (constant)
Operations	Arithmetic, logical, and comparison operations between registers	Operations with a register and an immediate (e.g., ADDI, LW)
Instruction Format	`funct7	rs2
Common Use Cases	Register-to-register arithmetic, logical operations	Operations involving constants (immediates), memory loads, branches
Example Instructions	ADD, SUB, AND, OR, XOR	ADDI, LW, SW, BEQ, BNE

Summary:

- **R-type instructions** are used for operations that involve two registers. These instructions perform arithmetic, logical, and comparison operations between the values in the registers.
- **I-type instructions** are used for operations that involve one register and an immediate value. These include arithmetic operations with constants, memory load and store operations, and branch operations.

Both formats are fundamental in the RISC-V architecture, enabling a wide variety of operations based on whether immediate values or register-to-register interactions are required.

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.**

In RISC-V architecture, the **S-type** and **B-type** instructions serve different purposes, primarily related to store operations and branch control. Let's break down their key differences and provide examples of their usage.

1. S-type Instructions:

S-type instructions are used primarily for **store operations**. They perform a store to memory, where data from a register is written to a memory address. The S-type format contains both an immediate value (for the offset) and a base register that holds the address.

- **Format**:

  ```
  css
  Copy code
  | opcode | rs2 | rs1 | funct3 | imm[11:5] | imm[4:0] |
  ```

  - **opcode**: 7 bits (specifies the type of instruction, e.g., SW, SH, SB for store operations)
  - **rs1**: 5 bits (the base register containing the memory address)
  - **rs2**: 5 bits (the register holding the data to be stored)
  - **funct3**: 3 bits (determines the exact store type, like SW for storing a word)
  - **imm**: 12 bits (immediate value used as an offset from the base address)
- **Usage**:
  S-type instructions are used for storing data from a register into memory. The value to be stored is found in rs2, and the memory address is calculated by adding the immediate value (split into two parts) to the value in rs1.
- **Examples**:
  - **SW** (Store Word):
    Stores a 32-bit word from register rs2 into the memory address calculated as rs1 + imm.

```
scss
Copy code
SW x2, 0(x3)  // Store the value in register x2 into memory address 0 + contents of x3
```

- **SH** (Store Half-word):
  Stores a 16-bit half-word from register rs2 into memory.

```
scss
Copy code
SH x5, 4(x6)  // Store the value in x5 into the memory address 4 + contents of x6
```

- **SB** (Store Byte):
  Stores an 8-bit byte from register rs2 into memory.

```
scss
Copy code
SB x7, 8(x8)  // Store the value in x7 into the memory address 8 + contents of x8
```

2. B-type Instructions:

B-type instructions are used for **branching operations**, i.e., controlling program flow by making decisions based on a comparison between registers. They are used to implement conditional branches (e.g., jump if equal, jump if not equal).

- **Format**:

```
css
Copy code
| opcode | imm[12] | rs2 | rs1 | funct3 | imm[10:5] | imm[4:1] | imm[11] |
```

  - **opcode**: 7 bits (e.g., BEQ, BNE for branch instructions)
  - **rs1** and **rs2**: 5 bits (the two registers to be compared)
  - **funct3**: 3 bits (determines the comparison type, e.g., BEQ for equal, BNE for not equal)
  - **imm**: 12 bits (used for the offset that determines the branch target, split into several parts)

- **Usage**:
  B-type instructions compare the values in registers rs1 and rs2, and if the comparison is true, the program branches to the target address. The target address is calculated as the immediate value (imm) added to the current program counter (PC).
- **Examples**:
  - **BEQ** (Branch if Equal):
    If the contents of registers rs1 and rs2 are equal, branch to the target address.

```
css
Copy code
BEQ x1, x2, label  // If x1 == x2, jump to the label
```

- **BNE** (Branch if Not Equal):
  If the contents of registers rs1 and rs2 are not equal, branch to the target address.

  ```
  css
  Copy code
  BNE x3, x4, label  // If x3 != x4, jump to the label
  ```

- **BLT** (Branch if Less Than):
  If the contents of rs1 are less than rs2, branch to the target.

  ```
  css
  Copy code
  BLT x5, x6, label  // If x5 < x6, jump to the label
  ```

Key Differences:

Aspect	S-type Instructions	B-type Instructions
Purpose	Store data to memory (e.g., store word, byte)	Conditional branch (e.g., branch if equal)
Instruction Format	Stores value from a register into memory	Compares two registers and branches if the condition is met
Use of Immediate	Immediate used as an offset to memory address	Immediate used as a branch offset
Examples	SW, SH, SB (store instructions)	BEQ, BNE, BLT (branch instructions)
Registers Involved	rs1 (base address), rs2 (data to store)	rs1 and rs2 (registers to compare)
Operation	Memory store operation (writing data)	Program flow control (conditional branching)

Summary:

- **S-type** instructions are primarily used for store operations, writing data from a register into memory.
- **B-type** instructions are used for branching operations, where the program flow is altered based on the result of a comparison between two registers.

Both types of instructions are critical in implementing low-level control mechanisms in a RISC-V program.