Assembly Programming

In RISC5

Hamming Code

Introduction:

In digital communications, data integrity during transmission is crucial. One common method to ensure data reliability is the use of error detection and correction codes. Hamming code, developed by Richard Hamming, is a widely used technique that allows not only the detection of single-bit errors but also their correction. This project aims to simulate the transmission of a message over a noisy communication channel using Hamming code for error detection and correction. The task involves converting several C functions into RISC5 assembly language to handle the generation, encoding, decoding, and mapping of messages. For more details , check out my GitHub repository: ouvh/Hamming-Code-in-Assembly: Enhance your understanding of RISC5 assembly language and master the implementation of the Hamming Code algorithm with this comprehensive guide. (github.com)

Project Features:

This project encompasses several key features to simulate a robust message transmission system using Hamming code:

- **Input functionality**: Implementing terminal-based input functionality to enhance user experience and interaction.
- **Message Generation**: Creation of a 24-bit message through bit manipulation and rotation.
- **Message Mapping:** Placement of the 24-bit message into a 32-bit format with specific bits reserved for parity checks.
- **Error Detection and Correction:** Encoding the message with Hamming code to allow the detection and correction of single-bit errors.
- **Message Decoding:** Decoding the received message, detecting errors, and correcting them if necessary.
- **Message Reconstruction:** Extracting the original 24-bit message from the corrected 32-bit code.

Project Documentation:

First, we will start by coding the necessary Helper function in RISC5 primarily the input function, and output functions.

Helper Function:

- input_from_terminal
- print_int
- print_hexa
- print_space
- print_new_line
- print_string

input_from_terminal:

We will obtain user input in two formats: decimal and hexadecimal. This input functionality is handled in RISC5 using the ecall codes 0x130 and 0x131. After receiving the input, each character is parsed and converted into a number. If the input starts with "0x", the characters are interpreted as a hexadecimal number:

```
input_from_terminal:
        store_stack:
            addi sp,sp,-24
            sw a1,0(sp)
            sw t1,4(sp)
            sw t2,8(sp)
            sw t3,12(sp)
            sw t4,16(sp)
            sw t0,20(sp)

        start:
            # Activate terminal input
            addi a0, x0, 0x130
            ecall

            # Initialize input variables
            la t0, input   # Load address of input into t0
            li t1, 0       # Initialize t1 to store the integer value
            addi t2,x0,10

        read_input:
            # Poll for console input
            addi a0, x0, 0x131
            ecall


            addi t4,x0,1

            # Check input status
            beq a0, x0, input_done  # If a0 == 0, all input has been read
            beq a0, t4, read_input  # If a0 == 1, still waiting for input

            # a0 == 2, valid input character read
            # a1 contains the character in UTF-16

            # Convert character to integer if it's a digit
            addi t3, a1, -48  # Convert ASCII to integer by subtracting '0' (48
in ASCII)


            addi t4,x0,72
            bne t3,t4,not_hex
                addi t2,x0,16
                addi t3,x0,0
            not_hex:


            blt t3, x0, read_input   # If character is less than '0', read next
input

            addi t4,x0,17      # A
            bne t3,t4,not_A
                addi t3,x0,10
                j ready
            not_A:
             addi t4,x0,18      # B
            bne t3,t4,not_B
                addi t3,x0,11
                j ready
            not_B:

             addi t4,x0,19      # C
            bne t3,t4,not_C
                addi t3,x0,12
                j ready
            not_C:

             addi t4,x0,20      # D
            bne t3,t4,not_D
                addi t3,x0,13
                j ready
            not_D:

             addi t4,x0,21      # E
            bne t3,t4,not_E
                addi t3,x0,14
                j ready
            not_E:

             addi t4,x0,22      # F
            bne t3,t4,not_F
                addi t3,x0,15
                j ready
            not_F:



            addi t4,x0,10
            bge t3, t4, read_input   # If character is greater than '9', read
next input




            ready:

            # Accumulate the integer value
            add t4,x0,t2
            mul t1, t1, t4   # Multiply current value by 10 or 16 in case of hex
            add t1, t1, t3   # Add the new digit
            j read_input     # Continue reading input

        input_done:


            # return the integer value
            addi a0,t1,0
            sw x0,0(t0)


            empty_stack:

                lw t0,20(sp)
                lw t4,16(sp)
                lw t3,12(sp)
                lw t1,4(sp)
                lw t2,8(sp)
                lw a1,0(sp)
                addi sp,sp,24


            jr ra

#
```

print_int:

The print_int function prints an integer to the terminal. It uses the ecall system call with code 1 to print the integer stored in register a1.

```
#
print_int:
    addi sp,sp,-4
    sw a1,0(sp)

    addi a1,a0,0
    addi a0,x0,1
    ecall

    lw a1,0(sp)
    addi sp,sp,4
    jr ra
```

print_hexa:

The print_hexa function prints an integer in hexadecimal format. It takes the integer value in register a0 and prints it as a hexadecimal number using a system call with code 34.

```
#
print_hexa:
    addi sp,sp,-4
    sw a1,0(sp)

    addi a1,a0,0
    addi a0,x0,34
    ecall

    lw a1,0(sp)
    addi sp,sp,4


    jr ra

#
```

print_space:

The print_space function prints a space character to the terminal. It uses the ecall system call with code 35 to print a space.

```
#
print_space:
    addi sp,sp,-4
    sw a1,0(sp)

    addi a1,x0,32
    addi a0,x0,11
    ecall

    lw a1,0(sp)
    addi sp,sp,4
    jr ra
```

print_new_line:

The print_new_line function prints a newline character to the terminal. It uses the ecall system call with code 36 to print a newline.

```
#
print_new_line:
    addi sp,sp,-4
    sw a1,0(sp)

    addi a1,x0,10
    addi a0,x0,11
    ecall

    lw a1,0(sp)
    addi sp,sp,4


    jr ra

#
```

print_string:

The print_string function prints a null-terminated string to the terminal. It uses the ecall system call with code 4 to print the string.

```
print_string:
    addi sp,sp,-4
    sw a1,0(sp)

    addi a1,a0,0
    addi a0,x0,4
    ecall
    lw a1,0(sp)
    addi sp,sp,4

    jr ra
```

Main functionality:

Sending a Message:

Before diving into the coding, let's clarify what we need to do in the context of Hamming Code. As mentioned in the introduction, Hamming Code is a bit manipulation technique used in data transmission to ensure accurate data transmission through the use of redundant bits called parity bits, which help detect errors.

The first step is to construct the redundant bits:

(Please refer to the Document Bit maps helper in the repository to find the masks)

Get_message_asm: This function will handle the message we want to send over the network. We will use a simple approach that involves inverting the message and applying a certain number of rotations, defined as 'r'.

To implement this in assembly, we will use bit manipulation instructions provided by RISC5, such as OR, AND, NOT, SLL, and SRL, to perform the necessary bit shifting.

```
#
get_message_asm:

    #store_stack:

        addi sp,sp,-8
        sw t0,0(sp)
        sw t1,4(sp)


    # Arguments: a0 = matricule, a1 = r
    # Result: a0 = message

    # Invert all bits of matricule
    not a0, a0

    # Perform left rotation by r positions
    sll t0, a0, a1          # t0 = a0 << a1
    sub t1, x0, a1           # t1 = 32 - a1
    srl t1, a0, t1          # t1 = a0 >> (32 - a1)
    or a0, t0, t1            # a0 = (a0 << a1) | (a0 >> (32 - a1))

    # Keep only the least significant 24 bits
    li t0, 0xFFFFFF          # t0 = 0xFFFFFF
    and a0, a0, t0           # a0 = a0 & 0xFFFFFF

    # Return the result in a0
    #empty_stack:
        lw t1,4(sp)
        lw t0,0(sp)
        addi sp,sp,8
    jr ra
```

Once we have hidden our message, we will proceed with the Hamming code part. First, we need to construct a new message called the mask, which will contain our desired message along with some additional bits called parity bits. We will create a function called **hamming_map** for this purpose. This function will use the slli instruction and a loop to iterate over all the bits, placing them in the correct positions in the mask:

```
#
hamming_map_asm:

    addi sp,sp,-24
    sw t0,0(sp)
    sw t1,4(sp)
    sw t2,8(sp)
    sw t3,12(sp)
    sw t4,16(sp)
    sw t5,20(sp)

    # Arguments: a0 = 24-bit message
    # Result: a0 = 32-bit mapped integer with parity bits set to 0

    li t0, 0x0         # Initialize map to 0

    # Mapping each bit of msg to the appropriate position in map
    addi t1,x0,2
    addi t3 ,x0,3
    addi t4,x0,7
    addi t5,x0,15

    LOOP_MAP:
        addi t2 ,x0,29
        beq t1,t2,END_LOOP_MAP

        beq t1,t3,SKIP_BIT_MAP
        beq t1,t4,SKIP_BIT_MAP
        beq t1,t5,SKIP_BIT_MAP


        andi t2,a0,1
        sll t2,t2,t1
        or t0,t0,t2
        srli a0,a0,1




        SKIP_BIT_MAP:


        addi t1,t1,1
        j LOOP_MAP


    END_LOOP_MAP:


    addi a0, t0 ,0      # Return the result in a0


    lw t5,20(sp)
    lw t4,16(sp)
    lw t3,12(sp)
    lw t2,8(sp)
    lw t1,4(sp)
    lw t0,0(sp)
    addi sp,sp,24
    jr ra
```

You can notice that in this function we only place our message in the map, and we left the parity set to 0. Now we will implement the function that will calculate those bits. The parity bits hold the parity of the number of predefined high bits in the map.

we will use the **parity function** :

```
#
parity:
    # Calculate the parity of a0
    addi sp,sp,-8
    sw t0,0(sp)
    sw t1,4(sp)


    addi t0, x0,0           #initiate the counter with 0
    parity_loop:

        beq a0, x0, parity_end
        andi t1, a0, 1
        add t0, t0, t1
        srli a0, a0, 1
        j parity_loop

    parity_end:
        andi a0, t0, 1   #paity of the counter

        lw t1,4(sp)
        lw t0,0(sp)
        addi sp,sp,8


        jr ra
```

The next step is to build the hamming encode that will fill those bits and return the final map that we will send over the network.

```
#
hamming_encode:
    #stack storage

    addi sp,sp,-16
    sw t0,0(sp)
    sw t1,4(sp)
    sw t2,8(sp)
    sw ra,12(sp)



     # Arguments: a0 = mapped integer
    # Result: a0 = encoded integer with parity bits

    addi t0, a0,0      # Copy map to t0
    addi t1,a0,0       # Initialize code to map

    # Calculate parity and set the corresponding bits in code

    lw t2,parity_mask_1
    and t2, t0, t2
    addi a0,t2,0
    jal parity
    or t1, t1, a0

    lw t2,parity_mask_2
    and t2, t0,t2
    addi a0,t2,0
    jal parity
    slli a0, a0, 1
    or t1, t1, a0

    lw t2,parity_mask_3
    and t2, t0,t2
    addi a0,t2,0
    jal parity
    slli a0, a0, 3
    or t1, t1, a0

    lw t2,parity_mask_4
    and t2, t0,t2
    addi a0,t2,0
    jal parity
    slli a0, a0, 7
    or t1, t1, a0

    lw t2,parity_mask_5
    and t2, t0,t2
    addi a0,t2,0
    jal parity
    slli a0, a0, 15
    or t1, t1, a0

    addi a0, t1 ,0     # Return the result in a0

    lw ra,12(sp)
    lw t2,8(sp)
    lw t1,4(sp)
    lw t0,0(sp)
    addi sp,sp,16

    jr ra




#
```

Receiving a Message:

The next step of the project is to simulate the reception of the map. When we receive a map, the first task is to check for any errors that may have occurred during transmission. Keep in mind that Hamming Code aims to detect errors, but it's only an approximate method and can fail in cases with numerous changes. To detect an error, we need to verify the parity numbers. Let's build the **hamming_decode** function using the same bit manipulation instructions:

```
#
hamming_decode:
    #stack storage

    addi sp,sp,-16
    sw t0,0(sp)
    sw t1,4(sp)
    sw t2,8(sp)
    sw ra,12(sp)



    # Arguments: a0 = mapped integer
    # Result: a0 = encoded integer with parity bits

    addi t0, a0,0      # Copy map to t0
    addi t1,x0,0       # Initialize code to 0

    # Calculate parity and set the corresponding bits in code

    lw t2,parity_mask_1
    ori t2,t2,0b1
    and t2, t0, t2
    addi a0,t2,0
    jal parity
    or t1, t1, a0

    lw t2,parity_mask_2
    ori t2,t2,0b10
    and t2, t0,t2
    addi a0,t2,0
    jal parity
    slli a0, a0, 1
    or t1, t1, a0

    lw t2,parity_mask_3
    ori t2,t2,0b1000
    and t2, t0,t2
    addi a0,t2,0
    jal parity
    slli a0, a0, 2
    or t1, t1, a0

    lw t2,parity_mask_4
    ori t2,t2,0b10000000
    and t2, t0,t2
    addi a0,t2,0
    jal parity
    slli a0, a0, 3
    or t1, t1, a0

    lw t2,parity_mask_5
    addi a0,x0,1
    slli a0,a0,15
    or t2,t2,a0
    and t2, t0,t2
    addi a0,t2,0
    jal parity
    slli a0, a0, 4
    or t1, t1, a0

    addi a0, t0 ,0     # Return the result in a0

    beq t1 ,x0,no_error

    addi t0,x0,1
    addi t1,t1,-1
    sll t0,t0,t1
    xor a0,a0,t0



    no_error:


        lw ra,12(sp)
        lw t2,8(sp)
        lw t1,4(sp)
        lw t0,0(sp)
        addi sp,sp,16

        jr ra
```

This function corrects errors based on a 5-bit map created by recalculating the parity numbers of the predefined bits. If this map is 0, no error is detected. Otherwise, we flip the bit at the position corresponding to the integer value of the bit-map minus 1. The final step of the decode phase is to extract the message from the bit map by reversing all the steps performed during the encode phase:

```
#
hamming_unmap_asm:

    addi sp,sp,-24
    sw t0,0(sp)
    sw t1,4(sp)
    sw t2,8(sp)
    sw t3,12(sp)
    sw t4,16(sp)
    sw t5,20(sp)

    # Arguments: a0 = 24-bit message
    # Result: a0 = 32-bit mapped integer with parity bits set to 0

    li t0, 0x0         # Initialize map to 0

    # Mapping each bit of msg to the appropriate position in map
    addi t1,x0,28

    addi t3 ,x0,3
    addi t4,x0,7
    addi t5,x0,15

    LOOP_UNMAP:

        addi t2 ,x0,1
        beq t1,t2,END_LOOP_UNMAP

        beq t1,t3,SKIP_BIT_UNMAP
        beq t1,t4,SKIP_BIT_UNMAP
        beq t1,t5,SKIP_BIT_UNMAP


        addi t2,x0,1
        sll t2,t2,t1
        and t2,a0,t2
        srl t2,t2,t1


        slli t0,t0,1
        or t0,t0,t2



        SKIP_BIT_UNMAP:


        addi t1,t1,-1
        j LOOP_UNMAP


    END_LOOP_UNMAP:


    addi a0, t0 ,0      # Return the result in a0


    lw t5,20(sp)
    lw t4,16(sp)
    lw t3,12(sp)
    lw t2,8(sp)
    lw t1,4(sp)
    lw t0,0(sp)
    addi sp,sp,24


    jr ra
```

Conclusion

By completing this assignment, you will gain a deeper understanding of error detection and correction techniques using Hamming code, as well as practical experience in low-level programming with assembly language. This is a valuable exercise in both theoretical computer science concepts and practical system-level coding skills.