Term Project Report

Description:

- The following program is a project done to recreate and re-imagine the CISCO binary game through MIPS Assembly. It should include all the features that I've notice when playing the original or my best attempt to try and replicate a feature within the bounds of the MARS emulator and my limited experience coding in MIPS. The main focus was modularity while maintaining a reasonable software development scope and in addition applying game development knowledge that I know.

Challenges I Encountered during development:

MIPS Programming Language

1. During this project, I realized again that I still have a hard time understanding the MIPS programming language syntax. The level of abstraction would still leave me questioning whether I correctly called the right registers, stored something in memory correctly, and wrote the right instructions for the function I want to write. So while the PSEUDOCODE wasn't difficult to implement from a logical perspective, breaking it down from Human language to what I would usually do in C, C++, or Java would be easy, but not for MIPS. However, for the first week, my program was filled with compiler errors, and I was struggling to make the basic game loop work with jump statements.
2. To mitigate my weakness with the language I relied heavily on MIPS Assembly Programming by Robert Winkler, and any relevant instruction slides that I could help give me a better understanding of how make the correct instruction call. The biggest method that help speed up development was completing my project first in C++ and then learning how to covert each function that I wanted or most often case break down. However that allowed for development time to increase rapidly compared to before.

Jump Statement and Link

1. This was the most confusing part of the whole process in it self and the most time consuming. I had to use for loops, and even if I didn't need to use loop I would need it call function from other files that may call a function from another file. It was quite repetitive and I had to use the stack a lot to make sure I didn't lose a return address to get back to the main game loop.
2. In all I just power through and did a little bit of tracing and error checking to see where my code will fail and just had to stomach it throughout the development. Looking at example code snippets to see if I forgotten to add an instruction or not really help.

Mars Emulator

1. Even though I am retaking this class, I somehow forgotten during summer how to compile multiple files together for MIPS.
2. I checked setting and realized it was right there to assembled all files in directory.

What I've Learned

- The level of abstraction between high level languages and low level language, and how much complexity it adds to development. In addition the limitations I had compared to

using higher level language in which the compiler dose lot of the work for me.

Algorithms and Techniques

```
game_loop:
    lw $t2, game_over
    bne $t2, $zero, end_game

    lw $t1, lines_displayed
    li $t3, 7
    bge $t1, $t3, player_loses

    jal generate_random_number
    jal get_player_input
    jal check_answer

    j game_loop
```

- From the start I knew I needed to create a game loop for this project to work. Or at least for the way I wanted it to turn out, a for loop structure could have work, but it wasn't the best loop structure compared to an infinite while loop structure which mimics real game design. However I still need to added end conditions for the game for when the player loses through the *end_game* function and the *game_over* variable flag.
- The modularity is a factor I'm really proud of and allows for me to continually call the necessary functions and in different orders

```
function_example:
    addi $sp, $sp, -12
    sw $ra, 0($sp)
    sw $s0, 4($sp)
    sw $s1, 8($sp)

    # Function body
    jal another_function

    lw $ra, 0($sp)
    lw $s0, 4($sp)
    lw $s1, 8($sp)
    addi $sp, $sp, 12
    jr $ra
```

- This example demonstrates the stack management I had to do for every function call as I had nested function calls, and I didn't want to use array as I would have keep track of indexing. It would be easier to use the stack as it nature will lead me back to first $ra that was added.

```
convert_decimal_to_binary:
    move $s0, $a0
    li $s1, 0
    li $t0, 1

convert_loop:
    beq $s0, $zero, convert_done
```

```
    andi $t1, $s0, 1
    mul $t2, $t1, $t0
    add $s1, $s1, $t2
    srl $s0, $s0, 1
    li $t3, 10
    mul $t0, $t0, $t3
    j convert_loop

convert_done:
    move $v0, $s1
    jr $ra
```

- The conversion being used is bitwise manipulation instead of the traditional division method. Same underlying algorithm to covert.

```
get_binary_retry:
    li $v0, 5
    syscall
    move $t0, $v0

    move $a0, $t0
    jal validate_binary_input

    beq $v0, $zero, invalid_input
    move $v0, $t0
    jr $ra

invalid_input:
    li $v0, 4
    la $a0, invalid_choice
    syscall
    j get_binary_retry
```

- My input validation biggest part was checking if player input is valid, as one of the biggest things I have to consider is if the player input is wrong. Having it been check and then make sure I go back to check all future retries.

```
// Start timer when question begins
FUNCTION start_timer():
    current_time = GET_SYSTEM_TIME()
    start_time = current_time
END FUNCTION

// Check if player exceeded time limit
FUNCTION check_timeout():
    current_time = GET_SYSTEM_TIME()
    elapsed_time = current_time - start_time
    elapsed_seconds = elapsed_time / 1000

    IF elapsed_seconds >= time_limit THEN
        DISPLAY "TIME'S UP!"
        RETURN timeout_occurred = TRUE
    ELSE IF elapsed_seconds >= (time_limit - 10) THEN
```

```
        DISPLAY "Warning: 10 seconds left!"
        RETURN timeout_occurred = FALSE
    ELSE
        RETURN timeout_occurred = FALSE
    END IF
END FUNCTION

// Main game integration
FUNCTION main_game_loop():
    WHILE game_not_over DO
        SET time_limit based on current_level
        CALL start_timer()
        CALL display_question()
        CALL get_player_input()

        timeout_result = CALL check_timeout()
        IF timeout_result == TRUE THEN
            INCREMENT lines_displayed
            DISPLAY "Timeout counts as wrong answer"
        END IF

        // Continue with answer validation...
    END WHILE
END FUNCTION
```

- Biggest issue when working with MIPS was that I couldn't make it so that while do player input the line amount would increase. So I had to compromise in having another process being ran while input is happening to replicate it. I could then scale to increase the amount of problems, but I realized that would have increased the complexity and I was running out of the time so settle for this instance. In addition I had so that has the levels increase the time limit would change according to that level to make it more difficult.

Suggestion

- Notify future students to enable the feature to assemble all files in directory, it comes unselected in MARS from the start. It very easy to assume with all the advance code editors and IDE would do it automatically.