Assignment 6

ECE 355

V00984826

Problem 1

Answer the following questions about IEEE-754 32-bit floating-point representation:

1. (10 points)

   (a) Convert the IEEE-754 number 1 11111111 00000000000000000000000 to its decimal format.

   (b) Convert the IEEE-754 number 0 00000000 11000000000000000000000 to its decimal format.

   (c) Represent the decimal number -0.625 in the 32-bit IEEE-754 format.

   (d) Given the two IEEE-754 numbers X and Y below:

   $$X = 0\ 01111011\ 10000000000000000000000, \quad Y = 1\ 01111110\ 11010000000000000000000,$$

   calculate $Z = X - Y$ in binary format. Then, convert the resulting Z from IEEE-754 binary format to its decimal equivalent.

Solution 1

Part a)

For 1 11111111 00000000000000000000000:

$$\begin{aligned} M_1 &= 0 \\ E &= 255 \\ \text{Result} &= \text{Not a number (NaN)} \end{aligned}$$

Part b)

For 0 00000000 11000000000000000000000:

$$\begin{aligned} S &= 0 \\ E &= 0 \\ M &= 1.1 \rightarrow 0 \\ E &= 0 \\ \text{Number} &= +1 \times 2^{-126} \times 0.11 \\ &= 0.75 \times 2^{-126} \\ &= 8.8162076 \times 10^{-39} \end{aligned}$$

Part c)

For -0.625:

$$\begin{aligned}
0.625 &= 0.5 + 0.125 \\
&= 2^{-1} + 2^{-3} \\
&= (0.101)_2 \times 2^0 \\
&= (1.01)_2 \times 2^{-1} \\
\text{Sign} &= 1 \\
\text{Exponent} &= -1 + 127 = 126 = (01111110)_2 \\
\text{Significand} &= 01000000000000000000000 \\
\text{IEEE-754} &= 1\ 01111110\ 01000000000000000000000
\end{aligned}$$

Part d)

Given two 32-bit IEEE-754 numbers:

$$X = 0\ 01111011\ 10000000000000000000000$$

$$Y = 1\ 01111110\ 11010000000000000000000$$

For X:

$$X = 1.5 \times 2^{-4} = 0.09375$$

For Y:

$$Y = -1.8125 \times 2^{-1} = -0.90625$$

Therefore:

$$Z = X - Y = 0.09375 - (-0.90625) = 1.0$$

Converting to IEEE-754:

$$Z = 0\ 01111111\ 00000000000000000000000$$

Problem 2

(5 points) Consider a pipelined datapath consisting of five stages:

- **F** – Fetch the instruction from the memory,
- **D** – Decode the instruction and read the source register(s),
- **C** – Execute the ALU operation specified by the instruction,
- **M** – Execute the memory operation specified by the instruction,
- **W** – Write the result in the destination register.

Identify data hazards in the code below and insert **NOP** instructions where necessary.

```
ADD #4, R0, R0          ; R0 = R0 + 4
ADD #4, R2, R2          ; R2 = R2 + 4
MOV (R0), R1            ; R1 = MEMORY[R0]
MOV (R2), R3            ; R3 = MEMORY[R2]
SUB R2, R0, R4          ; R4 = R2     R0
SUB R3, R1, R5          ; R5 = R3     R1
MOV R4, (R2)            ; MEMORY[R2] = R4
MOV R5, (R0)            ; MEMORY[R0] = R5
ADD #4, R0, R0          ; R0 = R0 + 4
ADD #4, R2, R2          ; R2 = R2 + 4
```

Solution 2

To resolve the data hazards in the given code, **NOP** (no-operation) instructions are inserted where necessary. The modified code with **NOP** instructions is as follows:

```
ADD #4, R0, R0         ; R0 = R0 + 4
ADD #4, R2, R2         ; R2 = R2 + 4
NOP
NOP
MOV (R0), R1           ; R1 = MEMORY[R0]
MOV (R2), R3           ; R3 = MEMORY[R2]
SUB R2, R0, R4         ; R4 = R2 - R0
NOP
NOP
SUB R3, R1, R5         ; R5 = R3 - R1
MOV R4, (R2)           ; MEMORY[R2] = R4
NOP
NOP
MOV R5, (R0)           ; MEMORY[R0] = R5
NOP
NOP
NOP
ADD #4, R0, R0         ; R0 = R0 + 4
ADD #4, R2, R2         ; R2 = R2 + 4
```

Explanation:

- After the first two `ADD` instructions, **NOP** instructions are inserted to prevent data hazards caused by dependent instructions that use `R0` and `R2`.
- For the `SUB` instructions, additional **NOP** instructions are added because the values of `R0` and `R2` (updated by `ADD` or `MOV`) are required before proceeding.
- Similarly, after the `MOV` instructions that store the results in memory, **NOP** instructions are added to allow time for memory updates.
- Additional **NOP** instructions are inserted before the final `ADD` instructions to ensure no unresolved hazards.

Problems 3 and 4

3. (2 points) Solve Problem 12.8 from the textbook.
4. (8 points) Solve Problem 12.7 from the textbook.

   Hint: Declare a shared counter variable, e.g., `volatile int thread_id_counter`, initialize it to 0 in `main()`, and poll it by each thread as follows:

   ```
   while (thread_id_counter != my_id);
   ```

   Each thread must increment `thread_id_counter` after updating `global dot_product`.

Solution for Problem 3

Given $P = 8$ and Speedup $= 5$, we need to solve the equation:

$$5 = \frac{1}{1 - f + \frac{f}{8}}$$

Rewriting the equation:

$$1 - f + \frac{f}{8} = \frac{1}{5}$$

$$1 - f + \frac{f}{8} = 0.2$$

$$1 - 0.2 = f - \frac{f}{8}$$

$$0.8 = f\left(1 - \frac{1}{8}\right)$$

$$0.8 = f \cdot \frac{7}{8}$$

$$f = \frac{0.8 \cdot 8}{7} = 0.914$$

Thus, $f = 0.914$, meaning the application program must be 91% parallelizable.

Solution for Problem 4

Below is the code for solving the problem using threads and a shared counter for synchronization. The shared variable ensures that each thread updates the global `dot_product` in a coordinated manner and avoiding race conditions.

```c
#include <stdio.h>
#include "threads.h"

#define N 100
#define P 4

double a[N], b[N];
double dot_product = 0.0;  // Global variable for storing the result
volatile int thread_id_counter = 0;  // Shared counter for synchronization
```

```c
Barrier bar;

void ParallelFunction (void)
{
    int my_id, i, start, end;
    double s;
    my_id = get_my_thread_id();
    start = (N/P) * my_id;
    end = (N/P) * (my_id + 1) - 1;

    // Step 1: Compute partial dot product for the assigned range
    s = 0.0;
    for (i = start; i <= end; i++)
        s = s + a[i] * b[i];

    // Step 2: Synchronize threads to update global sum one at a time
    while (thread_id_counter != my_id); // Wait for the current thread's turn
    dot_product = dot_product + s;      // Update the global dot product
    thread_id_counter = (thread_id_counter + 1) % P; // Allow the next thread access

    // Step 3: Synchronize all threads using a barrier
    barrier(&bar, P);
}

void main(void)
{
    int i;

    /* Initialize vectors a[] and b[] (details omitted for brevity). */
    init_barrier(&bar);                 // Initialize the barrier for thread synchronization
    thread_id_counter = 0;              // Initialize the shared counter
    dot_product = 0.0;                  // Initialize the global dot product

    // Step 4: Create threads for parallel computation
    for (i = 1; i < P; i++)
        create_thread(ParallelFunction);
    ParallelFunction();                 // Execute the function in the main thread

    // Print the final result
    printf("The dot product is %g\n", dot_product);
}
```

Explanation:

- **Global variables:** The `dot_product` variable is used to store the final result of the dot product calculation. The `thread_id_counter` ensures that threads update the global variable sequentially, avoiding race conditions.

- **Synchronization:** The `while` loop synchronizes threads by making them wait for their turn to update the global `dot_product`. The modulo operation ensures that access rotates among the threads.
- **Barrier:** After updating the global variable, threads wait at a barrier to ensure all threads complete their computations before proceeding.
- **Thread creation:** Threads are created in a loop, and each thread computes a portion of the dot product based on its assigned range.
- **Main thread:** The main thread also participates in the computation by executing `ParallelFunction()`.

This implementation ensures correct and efficient parallel computation of the dot product.