Student Name: Ashna Mittal

Student Number: 251206758

CS 3350B: Assignment 3

Solution 1: Evaluating the MIPS instruction set, lets evaluate the MIPS function 'exer1':

- **lw $t0, 0($a0)** : loads the word from the address in $a0 into $t0.
- **lw $t1, 4($a0)** : loads the word from the address in $a0 + 4 into $t1.
- **lw $t2, 0($a1)** : loads the word from the address in $a1 into $t2.
- **lw $t3, 4($a1)** : loads the word from the address in $a1 + 4 into $t3.
- **slt $t4, $t0, $t2** : sets $t4 to 1 if $t0 is less than $t2, else to 0.
- **slt $t5, $t1, $t3** : sets $t5 to 1 if $t1 is less than $t3, else to 0.
- **and $t4, $t4, $t5** : performs a bitwise AND operation between $t4 and $t5.
- **beq $t4, $0, foo** : branches to foo if $t4 is equal to 0.
- **sub $t3, $t3, $t2** : subtracts $t2 from $t3 and stores the result in $t3.
- **j bar** : jumps to the label bar.
- **foo: sub $t3, $t2, $t3** : label for the branch destination; subtracts $t3 from $t2 and stores the result in $t3.
- **bar: sw $t3, 0($a0)** : label for the jump destination; stores the word in $t3 into the address contained in $a0.
- **jr $ra** : jumps back to the address contained in the return address register $ra.

The C code for the above MIPS functions exer1:

```
void exer1(int* a0, int* a1) {
   int t0 = a0[0];
   int t1 = a0[1];
   int t2 = a1[0];
   int t3 = a1[1];

   if (t0 < t2 && t1 < t3) {
```

```
        // Branch not taken, fall through to the next instruction
    } else {
        t3 = t2 - t3;  // This is the instruction at 'foo' label
    }

    // Continue to the instruction at 'bar' label
    a0[0] = t3;

    // 'jr $ra' equivalent in C is to just return from the function.
}
```

The above C code assumes that **$a0** and **$a1** are pointers to integers since they are being used to access memory with **lw** and **sw**. The **lw** instructions are translated to assignments using array indexing, assuming that **a0** and **a1** are arrays of int. The **slt** instructions are translated to the less-than comparison. The **and** instruction is translated to a logical AND operation because it's used in a branch condition. The **beq** instruction leads to a conditional that checks if both comparisons are true. If not, it jumps to **foo**. The **sub** instruction within the **foo** label is translated directly into a subtraction operation. The **sw** instruction is translated to an assignment to the first element of array **a0**. The **jr $ra** instruction at the end of the function is implied in C by simply reaching the end of the function.

The jump to **bar** and the operation at **foo** essentially create an if statement in C, where **foo** contains the instructions executed when the condition is false. Since **foo** only has one instruction before jumping to bar, and **bar** only has one instruction before returning, these can be merged into the else branch of the if statement. The **jr $ra** at the end of bar is implicit in C as the function's return.

Solution 2: Mapping the operations and control structures (loops and function calls) from C to their MIPS equivalents: use the **$a0** register to pass the argument **v** to the functions, **$v0** to store the return value, and use the stack for saving and restoring any registers that are used which are not temporary (such as **$ra** for the return address).

The C function **exer2a** is a loop that repeatedly calls **exer2b** until **v** becomes non-positive and counts how many times this occurs. The function exer2b increments **v**, then shifts right by 2 bits, effectively dividing by 4 and returning this value.

The MIPS assembly translation is

```
exer2a:
    addi $sp, $sp, -8         # Allocate stack space for saved registers
    sw $ra, 4($sp)            # Save return address
    sw $a1, 0($sp)            # Save any other registers, if used
    move $a1, $a0             # Move argument v into another register for manipulation
    li $v0, 0                 # Initialize i to 0

exer2a_loop:
    blez $a1, exer2a_end   # If v <= 0, exit loop
    jal exer2b                # Call exer2b(v)
    move $a1, $v0             # Update v with return value from exer2b
    addi $v0, $v0, 1          # Increment i
    j exer2a_loop             # Repeat the loop

exer2a_end:
    lw $ra, 4($sp)            # Restore return address
    lw $a1, 0($sp)            # Restore any other registers
    addi $sp, $sp, 8          # Deallocate stack space
    jr $ra                    # Return from exer2a

exer2b:
    addi $sp, $sp, -4         # Allocate stack space for saved registers
```

```
sw $ra, 0($sp)          # Save return address
addi $v0, $a0, 1        # t = v + 1
sra $v0, $v0, 2         # t = t >> 2 (arithmetic shift right by 2)
lw $ra, 0($sp)          # Restore return address
addi $sp, $sp, 4        # Deallocate stack space
jr $ra                  # Return from exer2b
```

For the above MIPS code, in **exer2a**, we start by allocating stack space and saving the return address (and any other necessary registers). We use **$v0** to keep track of **i** and **$a1** for **v** because **$a0** will be used to pass parameters to **exer2b**.We use the **blez** instruction to check if **v** is less than or equal to 0 and if so, exit the loop. We call **exer2b** using **jal** and update **v** with the return value stored in **$v0** after the call. Increment **i** with **addi** and jump back to the beginning of the loop. After the loop, we restore the registers and deallocate the stack space before returning. In **exer2b**, we perform the operations as described in the function and return.

Solution 3:

The control signals for the MIPS Datapath to execute the given instructions are:

a) **slt $s1 $t3 $t5** instruction control signals:

This is an R-type instruction where the ALU performs a "set less than" operation, setting the destination register to 1 if `$t3` is less than `$t5`, and 0 otherwise.

Summary:

Control Signal	Value
nPC_sel	0
RegDst	1
RegWrite	1
ExtOp	x
ALUSrc	0
MemWrite	0
MemtoReg	0
Branch/Jump	0
ALUCtr	slt

b) **bne $s2 $s3 128** instruction control signals:

This is an I-type instruction which branches to the address offset in `$t1` if `$s2` is not equal to `$s3`.

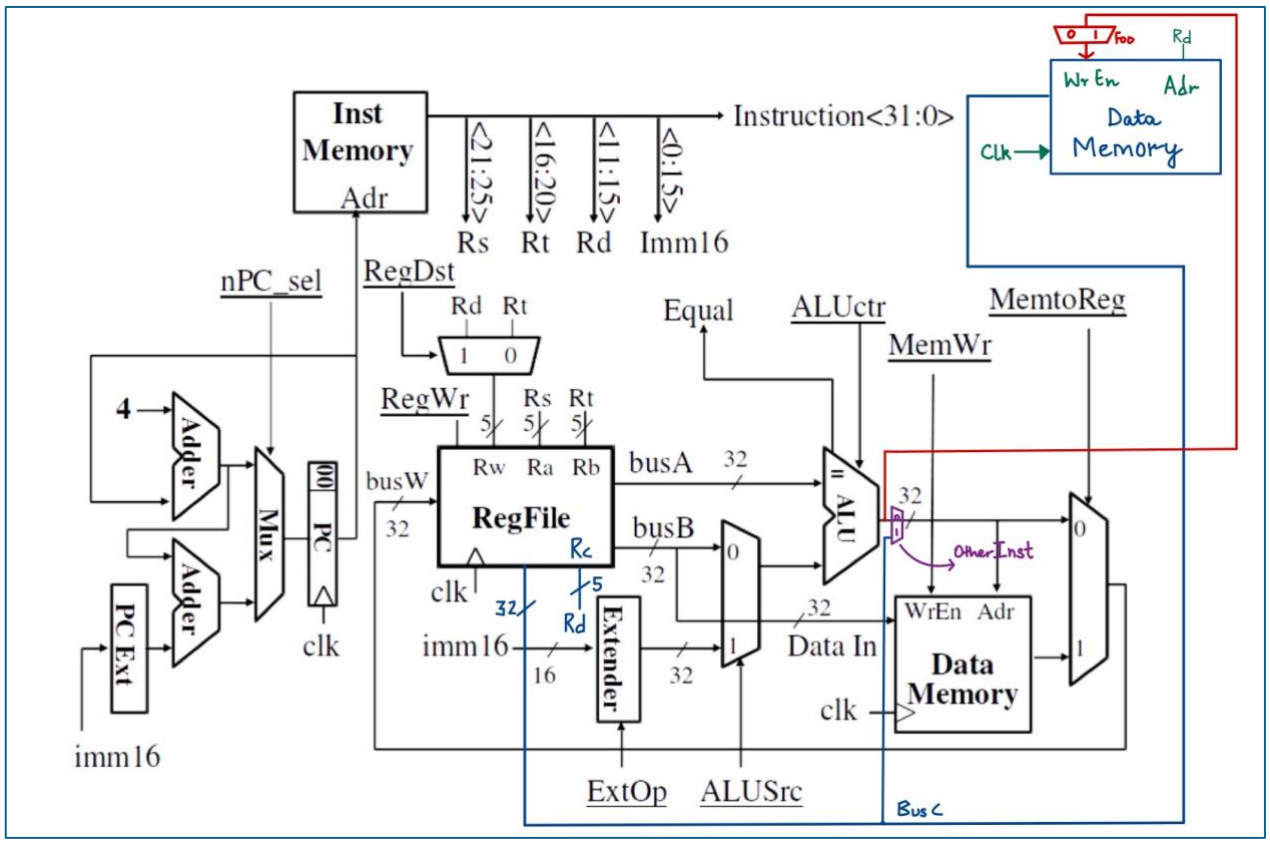
Summary:

Control Signal	Value
nPC_sel	1
RegDst	x

RegWrite	0
ExtOp	x
ALUSrc	0
MemWrite	0
MemtoReg	x
Branch/Jump	0
ALUCtr	bne

Solution 4:

a) Modified MIPS datapath:



b) In adapting the MIPS Datapath to accommodate the new `foo` instruction, I introduced a multiplexer (MUX) right after the existing ALU to control the data flow for this specific operation. I've added several control signals and made necessary wiring changes to integrate this function seamlessly into the existing circuit.

Originally, the Datapath directs data from the program counter (PC) to execute instructions and increments the PC by 4 using an adder. However, the `foo` instruction requires a novel operation: storing the value from a destination register (Rd) into memory, a capability not provided by the original Datapath. To achieve this, I drew a new wire from the Rd output in the Register File, which passes through BusC. This bus carries the Rd value to Data Memory for writing.

For any given operation, the ALU combines inputs from BusA and BusB, sending the result to Data Memory, which is configured to handle only write operations. During the execution of `foo`, the ALU's result, which is a combination of inputs from BusA and BusB, is written to the memory at the address specified by Rd. The write enable for this operation is controlled by a new signal that I introduced, which explicitly indicates whether the `foo` instruction is being executed.

I added a MUX to select inputs based on whether the current operation is `foo` or another instruction. If `foo` is the operation, the MUX is configured to read data from

memory. For other instructions, this MUX will choose whether to read or write data based on the standard control flow.

This careful inclusion of a MUX, new control signals, and the additional wiring is crucial as it distinguishes the execution path of the `foo` instruction from other MIPS instructions, ensuring that the modified Datapath can execute `foo` by writing the ALU's result into the specified memory address while also maintaining the correct execution of all other MIPS instructions.

c) The values of the control signals required to execute the instruction are:

Control Signal	**Value**
nPC_sel	0
RegDst	0
RegWrite	1
ExtOp	x
ALUSrc	0
MemWrite	1
MemtoReg	1
Branch/Jump	0
ALUCtr	add
OtherInst	0
foo	1

d) When any other instruction besides foo is being executed, the new control signals specific to foo would be set to not affect the normal operation of the Datapath, i.e., otherInst would be 1, and foo would be 0.
