Justin Arias

Homework 2

2.1 For the following C statement, what is the corresponding MIPS assembly code? Assume that the variables f, g, h, and i are given and could be considered 32-bit integers as declared in a C program. Use a minimal number of MIPS assembly instructions.
f = g + (h − 5);

Solution
addi $t0, $s2, -5
add $s0, $s1, $t0

2.2 For the following MIPS assembly instructions above, what is a corresponding C statement?
add f, g, h
add f, i, f

Solution
f = g + h;
f = i + f;

2.3 For the following C statement, what is the corresponding MIPS assembly code? Assume that the variables f, g, h, i, and j are assigned to registers $s0, $s1, $s2, $s3, and $s4, respectively. Assume that the base address of the arrays A and B are in registers $s6 and $s7, respectively.
B[8] = A[i−j];

Solution
```
sub  $t0, $s3, $s4      # $t0 = i - j
sll  $t1, $t0, 2        # $t1 = (i - j) * 4 (to account for word size, 4 bytes)
add  $t2, $s6, $t1      # $t2 = Address of A[i - j]
lw   $t3, 0($t2)        # Load the value from A[i - j] into $t3
sw   $t3, 32($s7)       # Store the value into B[8] (8 * 4 = 32)
```

2.4 For the MIPS assembly instructions below, what is the corresponding C statement? Assume that the variables f, g, h, i, and j are assigned to registers $s0, $s1, $s2, $s3, and $s4, respectively. Assume that the base address of the arrays A and B are in registers $s6 and $s7, respectively.
```
sll $t0, $s0, 2          # $t0 = f * 4
add $t0, $s6, $t0        # $t0 = &A[f]
sll $t1, $s1, 2          # $t1 = g * 4
add $t1, $s7, $t1        # $t1 = &B[g]
lw $s0, 0($t0)           # f = A[f]
addi $t2, $t0, 4
lw $t0, 0($t2)
add $t0, $t0, $s0
sw $t0, 0($t1)
```

Solution
f = A[f];
B[g] = f + A[f + 1];

2.5 For the MIPS assembly instructions in Exercise 2.4, rewrite the assembly code to minimize the number if MIPS instructions (if possible) needed to carry out the same function.

Solution
```
sll  $t0, $s0, 2     # $t0 = f * 4
add  $t0, $s6, $t0   # $t0 = &A[f]
sll  $t1, $s1, 2     # $t1 = g * 4
add  $t1, $s7, $t1   # $t1 = &B[g]
lw   $t2, 0($t0)     # $t2 = A[f]
lw   $t3, 4($t0)     # $t3 = A[f + 1]
add  $t2, $t2, $t3   # $t2 = A[f] + A[f + 1]
sw   $t2, 0($t1)     # B[g] = A[f] + A[f + 1]
```

2.6 The table below shows 32-bit values of an array stored in memory.

Address	Data
24	2
38	4
32	3
36	6
40	1

2.6.1 For the memory locations in the table above, write C code to sort the data from lowest to highest, placing the lowest value in the smallest memory location shown in the figure. Assume that the data shown represents the C variable called Array, which is an array of type int, and that the first number in the array shown is the first element in the array. Assume that this particular machine is a byte-addressable machine and a word consists of four bytes.

Solution
```
#include <stdio.h>

// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
```

```
    *b = temp;
}

// Function to sort the array using bubble sort
void bubbleSort(int array[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - 1 - i; j++) {
            if (array[j] > array[j + 1]) {
                swap(&array[j], &array[j + 1]);
            }
        }
    }
}

int main() {
    // The values are not stored in sequential memory locations
    // We'll simulate this by creating a custom array
    // with the elements matching their respective addresses.
    int Array[5] = {2, 4, 3, 6, 1};  // Corresponding to addresses: 24, 38, 32, 36, 40

    // Size of the array
    int size = sizeof(Array) / sizeof(Array[0]);

    // Sort the array
    bubbleSort(Array, size);

    // Print the sorted array
    printf("Sorted Array:\n");
    for (int i = 0; i < size; i++) {
        printf("%d ", Array[i]);
    }

    return 0;
}
```

2.6.2 For the memory locations in the table above, write MIPS code to sort the data from lowest to highest, placing the lowest value in the smallest memory location. Use a minimum number of MIPS instructions. Assume the base address of Array is stored in register $s6.

Solution

```
    # Assume base address of the array is in $s6
    # Array elements correspond to addresses 24, 38, 32, 36, 40
    # Use temporary registers for storing and comparing values

    # Load the values into registers
    lw   $t0, 0($s6)      # Load value at address 24 (Array[0])
```

```
    lw   $t1, 14($s6)     # Load value at address 38 (Array[1])
    lw   $t2, 8($s6)      # Load value at address 32 (Array[2])
    lw   $t3, 12($s6)     # Load value at address 36 (Array[3])
    lw   $t4, 16($s6)     # Load value at address 40 (Array[4])

    # Outer loop to ensure all elements are sorted
outer_loop:
    li   $t5, 0           # $t5 is the swap flag, initialize to 0

    # Compare Array[0] and Array[2] (address 24 and 32)
    bge  $t0, $t2, swap_0_2
    nop

    # Compare Array[2] and Array[3] (address 32 and 36)
    bge  $t3, $t4,  swap
```

2.7 Show how the value 0xabcdef12 would be arranged in memory of a little-endian and a big-endian machine. Assume the data is stored starting at address 0.

Solution

Little Endian: 0x12 0xEF 0xCD 0xAB

Big Endian: 0xAB 0xCD 0xEF 0x12

2.9 Translate the following C code to MIPS. Assume that the variables f, g, h, i, and j are assigned to registers $s0, $s1, $s2, $s3, and $s4, respectively. Assume that the base address of the arrays A and B are in registers $s6 and $s7, respectively. Assume that the elements of the arrays A and B are 4-byte words:
B[8] = A[i] + A[j];

Solution

```
    # Calculate the address of A[i]
    sll  $t0, $s3, 2       # $t0 = i * 4 (shift left logical by 2)
    add  $t0, $s6, $t0     # $t0 = &A[i] (base address of A + i * 4)

    # Load A[i]
    lw   $t1, 0($t0)       # $t1 = A[i]

    # Calculate the address of A[j]
    sll  $t2, $s4, 2       # $t2 = j * 4 (shift left logical by 2)
    add  $t2, $s6, $t2     # $t2 = &A[j] (base address of A + j * 4)

    # Load A[j]
```

```
lw   $t3, 0($t2)       # $t3 = A[j]

# Add A[i] and A[j]
add  $t4, $t1, $t3     # $t4 = A[i] + A[j]

# Calculate the address of B[8]
li   $t5, 32           # $t5 = 8 * 4 (since B[8] is 32 bytes from base)
add  $t5, $s7, $t5     # $t5 = &B[8] (base address of B + 32)

# Store the result in B[8]
sw   $t4, 0($t5)       # B[8] = A[i] + A[j]
```

2.10 Translate the following MIPS code to C. Assume that the variables f, g, h, i, and j are assigned to registers $s0, $s1, $s2, $s3, and $s4, respectively. Assume that the base address of the arrays A and B are in registers $s6 and $s7, respectively.

```
addi $t0, $s6, 4
add $t1, $s6, $0
sw $t1, 0($t0)
lw $t0, 0($t0)
add $s0, $t1, $t0
```

Solution

```
#include <stdio.h>

int main() {
   int A[2]; // Array A with at least 2 elements
   int f;    // Variable to hold the result

   // Initialize the first element for demonstration
   A[0] = 10; // Assume some value for A[0]

   // MIPS translation to C
   A[1] = A[0]; // A[1] = A[0]
   f = A[0] + A[1]; // f = A[0] + A[1]

   printf("f: %d\n", f); // Output the result
   return 0;
}
```

2.12 Assume that registers $s0 and $s1 hold the values 0x80000000 and 0xD0000000, respectively.

2.12.1 What is the value of $t0 for the following assembly code?

add $t0, $s0, $s1

Solution

$s0 = 0x80000000 = 231=2147483648
$s1 = 0xD0000000 = 13 * 228=3489660928

$t0 = 5638284288 = 0x150000000 = 0x50000000

2.12.2 Is the result in $t0 the desired result, or has there been overflow?

Solution
Since the decimal equivalent of $t0 exceeds the max value for a 32-bit integer, there is an overflow. The actual value stored in $t0, which is 0x50000000, does not represent the intended sum of the two original values. It represents the wrapped-around value due to overflow.

2.12.3 For the contents of registers $s0 and $s1 as specified above, what is the value of $t0 for the following assembly code?
sub $t0, $s0, $s1

Solution
$t0=$s0-$s1=0x80000000-0xD0000000=2147483648−3489660928
=−1342177280

$t0=−1342177280+4294967296=2952790016=0xB0000000

2.12.4 Is the result in $t0 the desired result, or has there been overflow?

Solution
Given the result of $t0, instead of an overflow occurring, an underflow happened instead. This is due to the fact that the subtraction of the two values yielded a negative value.

2.12.5 For the contents of registers $s0 and $s1 as specified above, what is the value of $t0 for the following assembly code?
add $t0, $s0, $s1
add $t0, $t0, $s0

Solution

$t0=$s0+$s1=0x50000000

$t0=$t0+$s0=0x50000000+0x80000000=1342177280+214748648
$t0=3489660928=0xD0000000

2.12.6 Is the result in $t0 the desired result, or has there been overflow?

Solution
For the first operation of $t0 addition, an overflow occurred, but for the second addition instruction, the result in $t0 is the desired result since it is a valid result in a 32-bit register and represents a positive integer.

2.18 Assume that we would like to expand the MIPS register file to 128 registers and expand the instruction set to contain four times as many instructions.

2.18.1 How this would this affect the size of each of the bit fields in the R-type instructions?

Solution
In order to represent 128 register, the number of bits required must be found. This is found through log2(128)=7 bits. Therefore, the rs, rt, and rd field would need 7 bits instead of 5. Furthermore, the instruction set must be expanded to 4x as many instructions, from 64 to 256 instructions. From this the new opcode size can be found from log2(256)=8 bits. With these changes found, for registers, the size of the bit fields of rs, rt, and rd will increase from 5 bits to 7 bits, and the size of the opcode will go from 6 to 8 bits

2.18.2 How this would this affect the size of each of the bit fields in the I-type instructions?

Solution
In the case of I-type instructions, expanding the MIPS register file to 128 registers and quadrupling the instruction set will see the rs and rt fields increase froom 5 to 7 bits, like the R-type instructions. This is to accommodate the additional registers. Furthermore, the opcode will also increase from 6 to 8 bits and lastly, the immediate field will remain unchanged at 16 bits.

2.18.3 How could each of the two proposed changes decrease the size of an MIPS assembly program? On the other hand, how could the proposed change increase the size of an MIPS assembly program?

Solution
Having a higher number of registers and a richer instruction set leads to greater efficiency and smaller MIPS assembly programs. This is due to reducing the need for memory access as well as simplifying operations. Additionally, there could also be an increased size due to wider instruction formats and possibly more complex code. As for the program size, that can depend on what specific programs are being written by the developer.

2.21 Provide a minimal set of MIPS instructions that may be used to implement the following pseudoinstruction:
not $t1, $t2 // bit-wise invert

Solution
```
nor $t1, $t2, $zero  # $t1 = ~$t2
```

2.22 For the following C statement, write a minimal sequence of MIPS assembly instructions that does the identical operation. Assume $t1 = A, $t2 = B, and $s1 is the base address of C.
A = C[0] << 4;

Solution
```
lw   $t0, 0($s1)     # Load C[0] into $t0
sll  $t1, $t0, 4     # Shift the value in $t0 left by 4 bits, store in $t1
```

2.26 Consider the following MIPS loop:
```
LOOP: slt $t2, $0, $t1
beq $t2, $0, DONE
subi $t1, $t1, 1
addi $s2, $s2, 2
j LOOP
DONE:
```

2.26.1 Assume that the register $t1 is initialized to the value 10. What is the value in register $s2 assuming $s2 is initially zero?

Solution
When $t1 is initialized to 10 and $s2 is initially zero, the value in register $s2 at the end of the loop will be 20.

2.26.2 For each of the loops above, write the equivalent C code routine. Assume that the registers $s1, $s2, $t1, and $t2 are integers A, B, i, and temp, respectively.

Solution
```
void loop_example(int *A, int *B) {
   int i = *A;  // Assuming $t1 is initialized to the value in *A (10)
   int temp = 0; // Assuming $s2 is initialized to 0 (initial value of B)

   while (i > 0) { // Equivalent to slt and beq
      i--;       // Decrement $t1 (i)
      temp += 2;  // Increment $s2 (temp) by 2
   }

   *B = temp;      // Store the result back into B
}
```

2.26.3 For the loops written in MIPS assembly above, assume that the register $t1 is initialized to the value N. How many MIPS instructions are executed?

Solution
The loop continues as long as $t1 > 0. Given that $t1 starts at N and is decremented by 1 each iteration, the loop runs N times. The total instructions = 5 * N=5N, which is the total number of MIPS instructions executed by the loop.

2.29 Translate the following loop into C. Assume that the C-level integer i is held in register $t1, $s2 holds the C-level integer called result, and $s0 holds the base address of the integer MemArray.

```
        addi $t1, $0, $0
LOOP: lw $s1, 0($s0)
        add $s2, $s2, $s1
        addi $s0, $s0, 4
        addi $t1, $t1, 1
        slti $t2, $t1, 100
        bne $t2, $s0, LOOP
```

Solution

```
#include <stdint.h> // For using integer types

void loop_example(int32_t *MemArray, int32_t *result) {
   int i = 0;          // Initialize i to 0
   *result = 0;        // Initialize result to 0

   while (i < 100) {  // Loop until i reaches 100
      *result += MemArray[i]; // Add MemArray[i] to result
      i++;           // Increment i
   }
}
```

2.38 Consider the following code:

```
lbu $t0, 0($t1)
sw $t0, 0($t2)
```

Assume that the register $t1 contains the address 0x1000 0000 and the register $t2 contains the address 0x1000 0010. Note the MIPS architecture utilizes big-endian addressing. Assume that the data (in hexadecimal) at address 0x1000 0000 is: 0x11223344. What value is stored at the address pointed to by register $t2?

Solution
From the MIPS code provided, the instruction lbu $t0, 0($t1) loads the byte at address 0x1000 0000, which is 0x11, into register $t0. The next instruction, sw $t0, 0($t2), stores the word

0x00000011 at the address 0x1000 0010. Therefore, the value stored at address 0x1000 0010 is 0x00000011.

2.47 Assume that for a given program 70% of the executed instructions are arithmetic, 10% are load/store, and 20% are branch.

2.47.1 Given this instruction mix and the assumption that an arithmetic instruction requires 2 cycles, a load/store instruction takes 6 cycles, and a branch instruction takes 3 cycles, find the average CPI.

Solution
Average CPI=(CPIarithmetic*fractionarithmetic)+(CPIload/store*fractionload/store)
+(CPIbranc*fractionbranch)
Average CPI=(2*0.70)+(6*0.10)+(3*0.20)=1.4+1.6+0.6=2.6

2.47.2 For a 25% improvement in performance, how many cycles, on average, may an arithmetic instruction take if load/store and branch instructions are not improved at all?

Solution
New CPI=Current CPI*(1-improvement percentage)

New CPI=2.6*(1-0.25)=2.6*0.75=1.95

New Average CPI=(x*0.70)+(6*0.10)+(3*0.20)
1.95=(x*0.70)+(6*0.10)+(3*0.20)
1.95=(x*0.70)+1.2
0.75=x*0.70
x1.0714

2.47.3 For a 50% improvement in performance, how many cycles, on average, may an arithmetic instruction take if load/store and branch instructions are not improved at all?

Solution
New CPI=Current CPI*(1-improvement percentage)

New CPI=2.6*(1-0.50)=2.6*0.50=1.3

New Average CPI=(x*0.70)+(6*0.10)+(3*0.20)
1.3=(x*0.70)+(6*0.10)+(3*0.20)
1.3=(x*0.70)+1.2
0.1=x*0.70
x0.1429