1.) [30 Points]

Architecture Type	Execution Sequence	Variables Destroyed	Overhead Instruction	Total Code Size	Data Moved to/from Memory	Overhead Data Bytes
I.) Accumulator	**Load** A **Add** B *#to acc* **Store** C **Load** A **Subtract** E *#from acc* **Store** D **Add** C **Store** F	A and C	2 (Load A, Add C)	Instruction - 8 bits; Register - 6 bits; Memory - 32 bits (40*8)/8 = **40 bytes**	(32-bit * 8 instructions)/8 = **32 bytes**	(2*32)/8 = **8 bytes**
II.) Memory-Register	**Load** r1, A **Add** r2, r1, B **Store** r2, C **Subtract** r3, r1, E **Store** r3, D **Add** r4, r3, C **Store** r4, F	None	1 (Add C)	([8*7]+[6*10]+[7*32])/8 = **42.5 bytes**	(32-bit * 7 instructions)/8 = **28 bytes**	**4 bytes**
III.) Register-Register	**Load** r1, A **Load** r2, B **Add** r3, r1, r2 **Store** r3, C **Load** r4, E **Subtract** r5, r1, r4 **Store** r5, D **Add** r6, r3, r5 **Store** r6, F	None	None	([8*9]+[6*15]+[6*32])/8 = **44.25 bytes**	(32-bit * 6 instructions)/8 = **24 bytes**	None

2.) [15 Points]

```
Struct foo {
        char a;
        bool b;
        int c;
        double d;
        short e;
        float f;
        double g;
        char *cptr
        float *fptr;
        int x;
};
```

Data Type	**Data size on 64-bit machine (bytes)**
Char	1
Bool	1
Int	4
Long	8
Double	8
Short	2
Float	4
pointer	8

1.) 1 byte (char a) + 1 byte (bool b) + 4 bytes (int c) + 8 bytes (double d) + 2 bytes (short e) + 4 bytes (float f) + 8 bytes (double g) + 8 bytes (char *cptr) + 8 bytes (float *fptr) + 4 bytes (int x) = 48 bytes

2.) The memory size required for an instance of the foo struct would be larger than the predetermined size of the struct, as the 64-bit machine will pad additional bytes for data types that are less than 8 bytes. **4 bytes** for int **+ 2 bytes** for char*2 and int **+ 2 bytes** for short and float **= 8 bytes** of padding. Thus, the required memory would be 48+8 = **56 bytes**.

3.) If we were to rearrange the struct sequence such that the larger data types were ordered from top to bottom (largest size to lowest size) it would eliminate the required padding, as the smaller data types at the bottom would add up to 8 bytes. Thus, the smallest memory size that is required is the same size as the struct, which is 48 bytes.

3.) [10 Points]

Given the instruction mix provided, the average number of cycles per instruction type can be calculated:

Load: 0.25 * 4 = 1 cycle
Store: 0.15 * 4 = 0.6 cycles
Branch: 0.15 * 3 = 0.45 cycles
Integer arithmetic/logical: 0.35 * 1 = 0.35 cycles
Floating point: 0.05 * 12 = 0.6 cycles
Other: 0.05 * 1 = 0.05 cycles

Each instruction type can be weighed by its frequency in the mix and sum the results to get the overall CPI:

CPI = 1 + 0.6 + 0.45 + 0.35 + 0.6 + 0.05 = *3.05 cycles per instruction*

4.) [20 Points]

```
# Load A and B from memory
fld      f0, 0(a0)        # Let f0 = A
fld      f1, 0(a1)        # Let f1 = B
# Subtract B from A to get C
fsub.d  f2, f0, f1   # f2 = C
# Compute 2 - A + B to get D
li       t0, 0x40000000  # Load hex of 2 as an integer
fcvt.d.w f3, t0           # Convert integer to double-precision float (f3 = 2.0)
fsub.d  f4, f3, f0        # f4 = 2.0 - A
fadd.d  f5, f4, f1        # f5 = 2.0 - A + B
fsd      f2, 0(a2)        # Store C in memory
fsd      f5, 0(a3)        # Store D in memory
# Check if I equals J
beq      t1, t2, add_a_b  # If I equals J, jump to add_a_b
sub      t3, x0, x0       # If I doesn't equal J, set t3 to 0
add_a_b:
# Add A and B and store result in A if I equals J; otherwise subtract A from B and store result in B
beq      t1, t2, add_a   # If I equals J, jump to add_a
sub      t3, x0, x0       # If I doesn't equal J, set t3 to 0
add_a:
fadd.d  f0, f0, f1        # A = A + B
fsd      f0, 0(a0)        # Store updated A in memory
j        end              # Jump to end of code
sub_b:
fsub.d  f1, f1, f0        # B = B - A
fsd      f1, 0(a1)        # Store updated B in memory
end:
```

5.) [25 Points]

1.)

```
ld      t0, 0(a1)       # Load d into t0 (temp register) //4 cycles
li      t1, 0           # Initialize i to 0 //4 cycles
li      x10, 100        # Load int 100 into register x10 for comparison //4 cycles
ld      t2, 0(x0)       # Load the address of X into t2 //4 cycles
loop:                   # Create loop (for statement)
ld      t3, 0(t2)       # Load X[i] into t3 //4 cycles
add     t3, t3, t0      # Add d to X[i] //1 cycle
sd      t3, 0(t2)       # Store the updated value of X[i] back into memory //4 cycles
addi    t1, t1, 1       # Increment i by 1 //1 cycle
ble     t1, x10, loop   # If i <= 100, branch to loop //3 cycles
```

2.)
ld t0, 0(a1) is a 32-bit instruction that loads a 64-bit value from memory, so it requires 6 bytes.
li t1, 0 is a 32-bit instruction that loads a 32-bit immediate value, so it requires 4 bytes.
li x10, 100 is a 32-bit instruction that loads a 32-bit immediate value, so it requires 4 bytes
ld t2, 0(a0) is a 32-bit instruction that loads a 64-bit address from memory, so it requires 6 bytes.
ld t3, 0(t2) is a 32-bit instruction that loads a 64-bit value from memory, so it requires 6 bytes.
add t3, t3, t0 is a 32-bit instruction that performs a 64-bit arithmetic operation, so it requires 4 bytes.
sd t3, 0(t2) is a 32-bit instruction that stores a 64-bit value to memory, so it requires 6 bytes
addi t1, t1, 1 is a 32-bit instruction that performs a 32-bit arithmetic operation, so it requires 4 bytes.
ble t1, x10, loop is a 32-bit instruction that performs a 32-bit comparison, so it requires 4 bytes.

[(6*4 bytes) + (4 bytes + 4 bytes + 6 bytes + 6 bytes + 4 bytes + 6 bytes + 4 bytes + 4 bytes = 40 bytes

But this does not take into account the 101 loops, so instead, add 20 bytes prior to the loop to the loop size * 101.

[20 + (24*101)] = **2444 bytes = 2.444 kilobytes**

3.) Assuming that the loop runs for all 101 iterations, the CPI can be calculated as follows:

16 cycles before the loop (4 loads) + 13 cycles every time the loop is run, and the loop is executed 101 times: (16+[13*101]) = 1329 cycles

4 initial instructions + 5 in the loop which are executed 101 times (the extra 1 is for the last loop through where t1 is greater than x10): (4+[5*101]) = 509 instructions

CPI = 1329/509 = **2.611 cycles per instruction**