COE 501: Computer Architecture

Problem Set 2: Instruction Set Principles

1) (4 pts) The MIPS dynamic instruction mix for the gcc program is given below:

load	store	add	sub	compare	cond branch	jump	call	return	shift	and	or	xor	other ALU
25.1%	13.2%	19.0%	2.2%	6.1%	12.1%	0.7%	0.9%	0.9%	1.1%	4.6%	8.5%	2.1%	3.5%

Given the following measurements of average CPI for instruction types:

Instruction Type	All ALU Instructions	Load & Store	Taken Branch	Not Taken Banch	Jump, call, and return
Average CPI	1.0	1.5	1.8	1.2	1.3

Assume that 70% of the conditional branches are taken, compute the effective CPI for the above gcc program.

2) (12 pts) Consider the following for-loop written in C code:

```
for (i = 0; i < 100; i++) { A[i] = B[i] + C; }
```

Where **A** and **B** are arrays of 64-bit integers, and **C** and **i** are 64-bit integers. Array **A** starts at address 1000 in memory, array **B** starts at address 3000, and variable **C** is at address 5000. The value of variable **C** should be loaded once into a register (**R8**) before entering the loop, and used in all loop iterations. Allocate a register for variable **i** (**R9**), and do not load or store its value in memory.

a) (6 pts) Write the code for MIPS64.
b) (2 pts) How many instructions are executed dynamically?
c) (2 pts) How many load/store instructions are executed dynamically?
d) (2 pts) What is the code size in bytes?

3) (12 pts) Assume that the values of A thru F reside in memory. Instruction opcodes are 8 bits, memory addresses are 32 bits, and register addresses are 4 bits. Instructions can vary in size.

a) (8 pts) Consider four different classes of instruction set architectures: Accumulator, Stack, Register-Memory (2-address instructions), and Register-Register (3-address instructions). For each instruction set class, write the code sequence to compute:

```
C = A + B
D = A – E
F = C + D
```

b) (4 pts) What is the code size (bits) for each instruction set class? All memory references are 32-bit absolute addresses encoded in the instruction format.

4) (8 pts) Consider a C structure (Object) that includes the following members. Assume a 64-bit architecture with 64-bit memory addresses:

```
struct Object {
  bool b;
  double d;
  short s1;
  char * cptr;   // pointer to char
  short s2;
  short s3;
  float * fptr;  // pointer to float
  float f;
  int i;
};
```

a) (2 pts) If the compiler does not align data in memory, what is the minimum size of the struct Object in bytes?

b) (3 pts) If the compiler aligns data in memory according to their size, but does not reorder the members of the structure, what will be the size of the struct Object? How many bytes are wasted internally inside the structure due to memory alignment?

c) (3 pts) If the compiler aligns data in memory and can also reorder the members of the structure, how the members of the struct Object should be reordered? What will be the size of the struct Object?

5) (14 pts) The MIPS architecture divides registers into two groups: Caller-Saved and Callee-Saved. Those registers that are **caller-saved** should be preserved by the caller (if needed) before making a procedure call. Those registers that are **callee-saved** should be preserved inside the called procedure (if needed) before modifying them. Registers are saved by storing their values on the runtime stack and restored by loading their values.

Register Number	Usage	Preserved by
R0	Always zero	Not applicable
R1	Reserved for assembler use	Not applicable
R2 – R3	For returning function results	Caller
R4 – R7	For passing arguments	Caller
R8 – R15	For temporary results	Caller
R16 – R23	For saved variables	Inside procedure (Callee)
R24 – R25	More temporary results	Caller
R26 – R27	OS kernel	Not applicable
R28	Global data pointer	Inside procedure (Callee)
R29	Stack pointer	Inside procedure (Callee)
R30	Frame pointer	Inside procedure (Callee)
R31	Return address	Inside procedure (Callee)

Suppose we want to translate function **f** using the above MIPS register conventions. Function **f** receives four arguments in registers **R4** thru **R7** and returns its result in register **R2**. We know that function **f** calls **g**, but we do not have the code for **g** and we don't know which registers are used in **g**. We only know that **g** receives two integer arguments and returns one integer result according to the MIPS register conventions shown above.

a) (5 pts) Translate function f into MIPS64 code:

```
int f(int a, int b, int c, int d) {
 return g(g(a,b), c+d);
}
```

b) (5 pts) Translate function f into MIPS64 code:

```
int f(int a, int b, int c, int d) {
  if (a+b<c+d) return g(a+b, c+d);
  else return g(c+d, a+b);
}
```

c) (4 pts) Can we use tail-call optimization that uses the **J** (jump) instruction rather than **JAL** (jump-and-link) to call the function **g** in part a and b? If no, explain why not. If yes, show how the code should be optimized in part a and b?