EC 362 Problem Set #2

1) MIPS has one of the simpler instruction sets in existence. However, it is possible to imagine even simpler instruction sets. Consider a hypothetical machine call SIC, for Single Instruction Computer. As its name implies, SIC has only one instruction: subtract and branch if negative, or `sbn` for short. The `sbn` instruction has three operands, each consisting of the address of a word in memory:

```
sbn   a, b, c   # Mem[a] = Mem[a] - Mem[b]; if (Mem[a]<0) go to c
```

The instruction will subtract the number in memory location b from the number in location memory location a and place the result back in memory location a, overwriting the previous value. If the result is greater than or equal to 0, the computer will take its next instruction from the memory location just after the current instruction. If the result is less than 0, the next instruction is taken from memory location c. SIC has no registers and no instructions other than `sbn`.

Although it has only one instruction, SIC can imitate many of the operations of more complex instruction sets by using clever sequences of `sbn` instructions. For example, here is a program to copy a number from location a to location b:

```
start:   sbn  temp, temp, .+1    #  Sets  temp  to  zero
         sbn  temp, a, .+1       #  Sets   temp  to  -a
         sbn  b, b, .+1          #  Sets  b  to  zero
         sbn  b, temp, .+1       #  Sets  b  to  -temp, which is a
```

In the program above, the notation .+1 means "the address after this one," so that each instruction in this program goes on to the next in sequence whether or not the result is negative. We assume temp to be the address of a spare memory word that can be used for temporary results.

Part a) Write a SIC program to add a and b, leaving the result in a and leaving b unmodified. Your SIC program should only require 3 instructions.

Part b) Write a SIC program to multiply a by b, putting the result in c. Assume that memory location "one" contains the number 1. Assume that a and b are greater than 0 and that it's OK to modify a or b. (Hint: What does this program compute?)

```
c = 0;  while (b > 0) {b = b – 1; c = c + a;}
```

Your SIC program should only require 5 instructions.

2) This problem compares the memory efficiency of four different styles of instruction sets for two code sequences. The architecture styles are the following:

- *Accumulator*: All operations occur in a single register called the accumulator (acc).
- *Memory-memory*: All three operands of each instruction are in memory.
- *Stack*: All operations occur on top of the stack. Only push and pop access memory, and all other instructions remove their operands from the stack and replace them with the result. The implementation uses a stack for the top two entries; accesses that use other stack positions are memory references.
- *Load-store*: All operations occur in registers, and register-to-register instructions have three operands per instruction. There are 16 general-purpose registers, and register specifiers are 4 bits long. This is the architecture style for MIPS (but slightly modified).

Consider the following C code:

```
a = b + c;     # a, b, and c are variables in memory
```

This code would be translated as follows using an accumulator style instruction set. Notice that all operations implicitly reference the `acc` register, and everything else is a memory reference:

```
load  B     # acc ← Memory[B]
add   C     # acc ← acc + Memory[C]
store A     # Memory[A] ← acc
```

The same code would be translated as follows using a memory-memory instruction set:

```
add   A, B, C     # Memory[A] ← Memory[B] + Memory[C]
```

In this case, there are no registers, just memory locations. So, all references are to memory. The same code would be translated as follows using a stack instruction set:

```
push  C     # top+=4; stack[top] ← Memory[C]
push  B     # top+=4; stack[top] ← Memory[B]
add         # stack[top-4] ← stack[top] + stack[top-4]; top-=4
pop   A     # Memory[A] ← stack[top]; top-=4
```

Here all operations act on the elements on the top of a stack. Again, there are no registers, just a stack. Everything else is a memory reference.

Finally, the load-store translation would be:

```
load  reg1, C           # reg1 ← Memory[C]
load  reg2, B           # reg2 ← Memory[B]
add   reg3, reg1, reg2
store reg3, A           # Memory[A] ← reg3
```

These are each equivalent assembly language code segments for the different styles of instruction sets. For a given code sequence, we can calculate the instruction bytes fetched and the memory data bytes transferred using the following assumptions about all four instruction sets:

- The opcode is always 1 byte (8 bits)
- All memory addresses are 2 bytes (16 bits)
- All memory data transfers are 4 bytes (32 bits)
- All instructions are an integral number of bytes in length
- There are no optimizations to reduce memory traffic

For example, a register load in the load-store ISA will require four instruction bytes (one for the opcode, one for the register destination, and two for a memory address) to be fetched from memory along with four data bytes. A memory-memory add instruction will require seven instruction bytes (one for the opcode and two for each of the three memory addresses) to be fetched from memory and will result in 12 data bytes being transferred (eight from memory to the processor and four from the processor back to memory). The following table displays a summary of this information for each of the architectural styles for the code appearing above:

Style	Instructions for a=b+c	Code bytes	Data bytes
Accumulator	3	3+3+3	4+4+4
Memory-memory	1	7	12
Stack	4	3+3+1+3	4+4+0+4
Load-store	4	4+4+3+4	4+4+0+4

For the following C code, write an equivalent assembly language program in each architectural style (assume all variables are initially in memory):

```
a = b + c;
b = a + c;
d = a – b;
```

Assume that the stack instruction set has instructions such as push, pop, add, sub, negate (computes the unary "minus" of the element on the top of the stack), and duplicate (makes a copy of the top of the stack), and assume that the accumulator instruction set has instructions such as load, store, add, sub, and negate.

For each code sequence, calculate the instruction bytes fetched and the memory data bytes transferred (read or written). Be efficient in writing your code, i.e. write the code as efficiently as you can so as to minimize the memory traffic. Which architecture is most efficient as measured by code size? Which architecture is most efficient as measured by total memory bandwidth required (code + data)? If the answers are not the same, why are they different?

3) Consider two different implementations, M1 and M2, of the same instruction set. There are three classes of instructions (A, B, and C) in the instruction set. M1 has a clock rate of 4 GHz, and M2 has a clock rate of 2 GHz. The average number of cycles for each instruction class on M1 and M2 is given in the following table.

Class	CPI on M1	CPI on M2	C1 usage	C2 usage	Third-Party usage
A	4	2	30%	30%	50%
B	6	4	50%	20%	30%
C	8	3	20%	50%	20%

The table also contains a summary of how three different compilers use the instruction set. C1 is a compiler produced by the makers of M1, C2 is a compiler produced by the makers of M2, and the other compiler is a third-party product. Assume that each compiler uses the same number of instructions for a given program but that the instruction mix is as described in the table. Using C1 on both M1 and M2, how much faster can the makers of M1 claim that M1 is compared with M2? Using C2 on both M2 and M1, how much faster can the makers of M2 claim that M2 is compared with M1? If you purchase M1, which compiler would you use? Which machine would you purchase if we assume that all other criteria are identical, including costs?

Solutions to Problem Set #2

1a)
```
sbn    temp, temp, .+1  # temp = 0
sbn    temp, b, .+1     # temp = -b
sbn    a, temp, .+1     # a = a – temp = a + b
```

1b)
```
sbn    temp, temp, .+1  # temp = 0
sbn    b, one, .+2      # if b < 1 jump +2.  either way b=b-1
sbn    temp, a, .-1     # temp = temp – a, go back 1 instruction
sbn    c, c, .+1        # c = 0
sbn    c, temp, .+1     # c = c – temp = b*a
```

2)

Acc	*Code*	*Data*
Load b	3	4
Add c	3	4
Store a	3	4
Add c	3	4
Store b	3	4
Neg	1	0
Add a	3	4
Store d	3	4
Total	22	28

Total memory traffic = 50 bytes.

Mem/mem	*Code*	*Data*
Add a,b,c	7	12
Add b,a,c	7	12
Sub d,a,b	7	12
Total	21	36

Total memory traffic = 57 bytes.

For stack operations, I'll assume that the sub operation computes A-B where B is on top of A on the stack.

Stack	*Code*	*Data*
Push b	3	4
Push c	3	4
Add	1	0
Dup	1	0
Pop a	3	4
Push c	3	4
Add	1	0
Dup	1	0
Pop b	3	4
Neg	1	0
Push a	3	4
Add	1	0
Pop d	3	4
Total	27	28

Total memory traffic = 55 bytes.

Load/store	*Code*	*Data*
Load regB,b	4	4
Load regC,c	4	4
Add regA,regB,regC	3	0
Store regA,a	4	4
Add regB,regA,regC	3	0
Store regB,b	4	4
Sub regD,regA,regB	3	0
Store regD,d	4	4
Total	29	20

Total memory traffic = 49 bytes.

The load/store machine has the least memory traffic since everything can be held in registers, once fetched from memory. Of course, it has the largest code size since it cannot combine memory ops with arithmetic. On the other extreme is the mem/mem architecture which has the best code size, but terrible memory traffic since 3 operands are fetched for every statement!

3) Cpu = IC * CPI / Rate

$Rate_{M1}$ = 4 GHz = 4e9 Hz
$Rate_{M2}$ = 2 GHz = 2e9 Hz

Class	CPI on M1	CPI on M2	C1 usage	C2 usage	Third-Party usage
A	4	2	30%	30%	50%
B	6	4	50%	20%	30%
C	8	3	20%	50%	20%

We are told that the instruction counts (IC) are the same for each compiler on each machine.

$CPI_{C1,M1}$ = (4 * .3 + 6 * .5 + 8 * .2) = 5.8
$CPI_{C1,M2}$ = (2 * .3 + 4 * .5 + 3 * .2) = 3.2
$CPI_{C2,M1}$ = (4 * .3 + 6 * .2 + 8 * .5) = 6.4
$CPI_{C2,M2}$ = (2 * .3 + 4 * .2 + 3 * .5) = 2.9
$CPI_{C3,M1}$ = (4 * .5 + 6 * .3 + 8 * .2) = 5.4
$CPI_{C3,M2}$ = (2 * .5 + 4 * .3 + 3 * .2) = 2.8

$Perf_{C1,M1}$ / $Perf_{C1,M2}$ = $Cpu_{C1,M2}$ / $Cpu_{C1,M1}$ = (IC * 3.2/2e9) / (IC * 5.8/4e9) = 16ns / 14.5ns= 1.10, so that M1 is 1.1 times faster than M2 using compiler C1.

$Perf_{C2,M2}$ / $Perf_{C2,M1}$ = $Cpu_{C2,M1}$ / $Cpu_{C2,M2}$ = (IC * 6.4/4e9) / (IC * 2.9/2e9) = 16ns / 14.5ns = 1.10, so that M2 is 1.1 times faster than M1 using compiler C2.

$Perf_{C3,M1}$ / $Perf_{C3,M2}$ = $Cpu_{C3,M2}$ / $Cpu_{C3,M1}$ = (IC * 2.8/2e9) / (IC * 5.4/4e9) = 14ns / 13.5ns = 1.03, so that M1 is 1.03 times faster than M2 using C3.

If you own M1, you should use C3. If you own M2, you should use C3 also. This is poor news for the "native" compiler writers of these two machines! You should always use C3 as your compiler, and if you buy a machine, buy M1.