1. Your first job as a new employee with ACME Computing is to upgrade their printer. Right now it has no memory and their computer has to spend, on the average, half the time of a job session just to feed the printer. You plan to add some memory as to increase printing speed. 4MB of memory will make printing by 90% faster and will increse the cost of the whole system by 15%. Your boss wants to know:

a) by how much faster job sessions will be;

$F = 50\%$
$S_0 = 1.9$

Amdahl's Law yields

$S_{overall} = 1/(1 - F + F/S_0) = 1.31$

Therefore the new system will be by 31% faster.

b) how the new job throughput will compare with the old one;

throughput = jobs_completed / time_unit

$throughput_{new}/throughput_{old} = \#jobs_{new}/\#jobs_{old} = (T/T_{new})/(T/T_{old}) = T_{old}/T_{new}$

$throughput_{new}/throughput_{old} = 1.31$

The new throughput will be by 31% greater than the present one.

c) whether the new system will be more cost effective than the current one.

CE = performance / cost

$CE_{new}/CE_{old} = (perf_{new}/cost_{new})/(perf_{old}/cost_{old}) = 1.31 / 1.15 = 1.139$

The new system will be by 13.9% more cost effective than the old one.

2. Consider the following C statement:

```
a = a + b*a;
```

The right hand side can be put in reverse Polish notation as:

```
aba*+
```

You have an 8-bit stack machine with 16 bit addresses; the only addressing mode is absolute. `a` and `b` are of type `very short int` (i.e. 8-bit) and are stored in memory starting with the address `0xDDD` (where DDD stands for the first three digits of your SSN). Their initial values are `0x2a` for `a` and `0x02` for `b`. The code starts in memory at address `0x5ffc`. Below is a list of opcodes (each opcode is one byte wide):

Instruction	Opcode
`add`	`0x20`
`sub`	`0x21`
`mul`	`0x22`
`div`	`0x23`
`push`	`0x24`
`pop`	`0x25`

a) Compile the C code for this stack machine; for each instruction in the program show the instruction format on the right hand sid; clearly mark the boundaries of each instruction.

Assume DDD = 123

push a	0x24	0x01	0x23				
push b	0x24	0x01	0x24				
push a	0x24	0x01	0x23				
mul	0x22						
add	0x20						
pop a	0x25	0x01	0x23				

b) Show the sequence of addresses issued by the CPU to execute this code; for each address indicate whether it's a memory read or write and what's on the data bus for that particular memory access.

Address	R/W	Data		
0x5ffc	R	0x24		
0x5ffd	R	0x01		
0x5ffe	R	0x23		
0x0123	R	0x2a		
0x5fff	R	0x24		
0x6000	R	0x01		
0x6001	R	0x24		
0x0124	R	0x02		
0x6002	R	0x24		
0x6003	R	0x01		
0x6004	R	0x23		
0x0123	R	0x24		
0x6005	R	0x22		
0x6006	R	0x20		
0x6007	R	0x25		
0x6008	R	0x01		
0x6009	R	0x23		
0x0123	W	0xfe		
18 memory accesses				

c) Assuming that each memory access takes on the average 2.5 clock cycles and that the execution phase of an arithmetic or logic instruction takes 2 clock cycles, find the average CPI of an instruction using the assembly program you wrote.

CPI_{avg} = clock_cycles / IC = (18 memory_accesses*2.5 + 2 alu_instructions*2) / 6

CPI_{avg} = 49/6 = 8.17

d) What is the average number of memory accesses per instruction?

N = memory_accesses / IC = 18/6 = 3

3. In some simple cases a simple instruction set like MIPS can synthesize powerful instructions found in richer instruction sets like the VAX. The following instruction uses a combination of base+displacement and indirect memory addressing

```
ADDB (R0), @5(R1)        # @5(R1) stands for M[M[R1]+5]; B in ADDB stands
                         # for Byte
```

Show the sequence of *native* MIPS instructions that would do the same job; use registers `$t0` and `$t1` to represent `R0` and `R1` respectively.

```
lw $t2, 0($t1) # do M[R1]; this is an address (size=word)

lb $t2, 5($t2) # completed @5(R1), data is of size byte

lb $t3, 0(t0) # completed M[R0]

add $t3, $t3, $t2 # add the two bytes

sb $t3, 0(t0) # result goes in memory
```

Please note that registers $t0 (R0) and $t1 (R1) must not be changed during the execution.