Solution to Assignment 4

1. (a) Immediate: AC = 100 + 100 = 200
   (b) Direct: AC = 100 + M[100] = 100 + 600 = 700
   (c) Indirect: AC = 100 + M[M[100]] = 100 + M[600] = 100 + 700
   (d) Indexed: AC = 100 + M[100+400] = 100 + 400 = 500

   Immediate addressing allows the programmer to directly specify the value of the operand to be used in an instruction; in execution, there is no need to fetch the operand from memory (hence fast).

   Direct addressing is most often used to refer to program variables stored in memory, according to the knowledge of the programmer at programming time.

   Indirect addressing corresponds to pointers in programming; by adjusting their values dynamically at runtime, the same instruction may operate on different operands.

   Indexed addressing is useful for implementing data structures, such as arrays. The index (register) corresponds to the 'logical' index of a structure that can be used for accessing (traversing) it.

2

Three-address version:		#operand accesses	# bytes in instruction
(a) Subt	R1, X, Y	2	1+1+2*3
Add	R2, X, Y	2	8
Add	R2, R2, Z	1	6
Mult	Z, R1, R2	1	6

(b) Total # operand accesses (read or write) = 6

(c) Total # bytes in the code = 28

Two-address version:		#operand accesses	# bytes in instruction
(a) Load	R1, X	1	5
Subt	R1, Y	1	5
Load	R2, X	1	5
Add	R2, Y	1	5
Add	R2, Z	1	5
Mult	R1, R2	0	3
Store	Z, R1	1	5

(b) Total # operand accesses = 6

(c) Total # bytes in the code = 33

One-address version:		#operand accesses	# bytes in instruction
(a) Load	X	1	4
Subt	Y	1	4
Store	T	1	4
Load	X	1	4
Add	Y	1	4
Add	Z	1	4
Mult	T	1	4
Store	Z	1	4

(b) Total # operand accesses = 8
(c) Total # bytes in the code = 32

Stack version:		#operand accesses	# bytes in instruction
(a) Push	X	2	4
Push	Y	2	4
Subt		3	1
Push	X	2	4
Push	Y	2	4
Push	Z	2	4
Add		3	1
Add		3	1
Mult		3	1
Pop	Z	2	4

Note: Push X involves reading memory operand X and then writing it to the new top of the stack. Hence two operand accesses are involved. Similarly Subt involves popping the stack twice and pushing the result of the subtraction back onto the stack, and hence three memory operand accesses altogether.

(b) Total # operand accesses = 24
(c) Total # bytes in the code = 28

(d) In comparing the four versions, we can take note that # operand accesses affect the time performance aspect while the # bytes in the code affect both the space and time performance aspects. As a result, for the example arithmetic expression, the 3-address version appears to the best while the stack version the worst (in incurring excessive memory/stack accesses). The two-address version is close to the 3-address version.

3(a) The maximum speedup corresponds to pumping the 200 instructions into the pipe without any delay. The total number of cycles (until the last instruction exits from the pipe) is 4 + 200 – 1 = 203 cycles or 203*60 ns.
Hence the maximum speedup ration = 200*200 / (203*60)

(b) With an infinite number of instructions, the maximum speedup ratio is
200*n / [(n + 3)*60] as n approaches infinity, and it is equal to 200/60

(c) In instruction processing, progression of instructions from a sequential stream of instructions into a pipeline may have to be delayed. Such delays arise from (i) data dependency: the result of an earlier instruction is needed for a later instruction (which must be delayed until the former completes), (ii) control dependency: conditional branches/jumps can alter the control flow but until the condition is computed from the earlier instruction, the branch cannot be effectively performed, and (iii) pipeline segments that involve memory accesses may be delayed due to slowness/conflicts at the memory.

4(a) The program decrements ecx before executing jnz each time. Since initially ecx = 100, jnz S1 will be executed 100 times.

(b) In each iteration of the loop (between S1 and the jns instruction), the current array element (via [ebx]) is added to ax, and the array pointer ebx is moved up by 2. Hence upon exit from the loop, ebx contains X + 200.

(c) Continuing with the analysis in (b), the result in register ax is the sum of the 100 elements in the array starting from location X.

(d) One difficulty in rewriting the code in MARIE is the absence of index addressing in MARIE. To achieve the same purpose, we may need to use 'self-modifying' code so the operand address in an instruction can be directly modified at runtime. The following is a solution (assuming X is the starting address of the array), showing only the code portion:

```
S0      Load    Count           //assume Count = 100
        Skipcond  800           //skip if > 0
        Jump    Done
        Load    Sum             //assume Sum saves the current sum
S1      Add     X               //update current sum
        Store   Sum
        Load    Count
        Subt    One
        Store   Count
        Load    One
        Add     S1
        Store   S1              //Now X in S1 is incremented by 1
        Jump    S0
Done    Load    Sum
        Store   Y
```

Hints for problem 5:

A key issue is how one can delete the spaces in the message string before output it to the screen. You need to know how to do it manually (by hand) before automating it with a program. You can do so by scanning the message character by character; upon scanning a space, you can remove it from the data storage so that when the next character is scanned, the latter will move up to the storage occupied by the space character. Mechanizing this in programming involves two pointers in the message string: one for the current storage (to be filled) and the other for the current character being scanned. A partial piece of code to accomplish this is shown below:

```
        mov     ebx, msg            //ebx is the storage pointer
        mov     ecx, msg            //ecx is the message pointer
        mov     edx, 1              //edx is the length count of message
again   cmp     byte [ecx], 0x20    //scanned character a space?
        jz      skip
        cmp     byte[ecx], 0xA      //scanned character linefeed?
        jz      done
        mov     byte [ebx], [ecx]   //keep the character in the message
```

```
        inc     ebx
        inc     edx
skip    inc     ecx
        jump    again
done    mov     byte [ebx], [ecx]       //terminate the string with linefeed
```