Question 4.1–1: (Solution, p 5) Define the fetch-execute cycle as it relates to a computer processing a program. Your definition should describe the primary purpose of each phase.

Question 4.1–2: (Solution, p 5) Explain in detail what the HYMN CPU does during the fetch phase of the fetch-execute cycle. (Your explanation should describe how the computer accesses values in registers and memory.)

Question 4.2–1: (Solution, p 5) Suppose that the HYMN CPU begins with the following in memory.

addr	data	(translation)
00000	100 11110	LOAD 11110
00001	101 11111	STORE 11111
00010	110 11110	ADD 11110
00011	101 11111	STORE 11111
00100	110 11110	ADD 11110
00101	101 11111	STORE 11111
00111	000 00000	HALT

If the user typed multiples of 25 starting at 25 (25, then 50, then 75,. . .) when prompted, what would the computer display?

Question 4.2–2: (Solution, p 5) Suppose that the HYMN CPU begins with the following in memory.

addr	data	(translation)
00000	100 11110	LOAD 11110
00001	110 11110	ADD 11110
00010	011 00001	JPOS 00001
00011	000 00000	HALT

If we repeatedly type the number $32_{(10)}$ when prompted, how many times would we type it before the computer halts?

Question 4.2–3: (Solution, p 5)

Suppose that the HYMN CPU begins with memory contents at right.

a. List all new values stored in memory as the program executes. Express your answers in binary or hexadecimal.

b. What values does the AC hold in the course of executing this program? Express your answers in binary or hexadecimal.

addr	data	(translation)
00000	100 01001	LOAD 01001
00001	010 01000	JZER 01000
00010	110 01010	ADD 01010
00011	101 01010	STORE 01010
00100	100 01001	LOAD 01001
00101	110 00000	ADD 00000
00110	101 00000	STORE 00000
00111	001 00000	JUMP 00000
01000	000 00000	HALT
01001	000 00001	1
01010	000 00010	2
01011	000 00100	4
01100	000 00000	0

Question 4.3–1: (Solution, p 5) Translate the following HYMN assembly language program into machine language. Express your answer in bits.

```
       READ
top:   WRITE
       ADD one
       JPOS top
       HALT
one:   1
```

addr	data
00000	
00001	
00010	
00011	
00100	
00101	
00110	
00111	

Question 4.3–2: (Solution, p 5) Write a HYMN assembly language program that reads a number n from the user and then displays n's absolute value. (The **absolute value** of a number is that number with any negative sign removed. The absolute value of -5 is 5, while the absolute value of 3 is 3 itself.)

Question 4.3–3: (Solution, p 6) Write a HYMN assembly language program that reads a number n and displays the powers of two that are less than n. Your program may assume that n is more than 1.

Question 5–1: (Solution, p 6) Consider the following Intel assembly code.

```
        movl $7, %eax
        movl $4, %ebx
        movl $4, %ecx
again:  pushl %eax
        addl %ebx, %eax
        popl %ebx
        decl %ecx
        jnz again
```

Show all values taken on by the registers as this program executes.

```
eax
ebx
ecx
```

Question 5–2: (Solution, p 6) Translate each of the following Intel assembly programs, generated by *gcc*, back to their nearest C equivalents.

a.

```
.section .rodata
.LC0:       .string "%d"
.LC1:       .string "%d %d\n"
.section .text
.globl main
main:       pushl %ebp
            movl %esp, %ebp
            subl $4, %esp
            leal -4(%ebp), %eax
            pushl %eax
            pushl $.LC0
            call scanf
            movl $1, %ecx
            xorl %eax, %eax
            addl $8, %esp
            movl -4(%ebp), %edx
            cmpl %edx, %eax
            jge .L18
.L20:       addl %eax, %ecx
            incl %eax
            cmpl %edx, %eax
            jl .L20
.L18:       pushl %ecx
            pushl %edx
            pushl $.LC1
            call printf
            xorl %eax, %eax
            leave
ret
```

b.

```
.section .rodata
.LC0:       .string "%d"
.LC1:       .string "%d\n"
.section .text
            .align 4
main:       pushl %ebp
            movl %esp, %ebp
            subl $4, %esp
            pushl %ebx
            leal -4(%ebp), %eax
            pushl %eax
            pushl $.LC0
            call scanf
            xorl %ebx, %ebx
            addl $8, %esp
            cmpl -4(%ebp), %ebx
            jge .L18
.L20:       pushl %ebx
            pushl $.LC1
            call printf
            addl $8, %esp
            addl %ebx, %ebx
            cmpl -4(%ebp), %ebx
            jl .L20
.L18:       xorl %eax, %eax
            movl -8(%ebp), %ebx
            leave
            ret
```

Question 6.1–1: (Solution, p 6) Suppose that eax held $104_{(16)}$ and esp held $20C_{(16)}$ when an x86 processor begins to execute the instruction "pushl %eax." Explain how the CPU alters the values in registers and memory.

Question 6.1–2: (Solution, p 6) Suppose we have a C function `myst` that takes two integers as an argument.

```
int myst(int x, int y);
```

Write an x86 assembly language code fragment that places the value of `myst(6, 10)` into the edx register. The fragment should include code to restore the program stack to its original state.

Question 6.2–1: (Solution, p 6) Explain what the Intel processor does when it executes the instruction "call fact." That is, explain how the CPU alters the values in registers and memory.

Question 6.2–2: (Solution, p 7) What operations does an Intel processor perform in executing a ret instruction? That is, how do the values in registers change? How does the computer determine which instruction to execute next?

Question 6.2–3: (Solution, p 7) How are parameter values passed to a subroutine, according to the Intel processor conventions? How does the subroutine communicate its return value?

Question 6.2–4: (Solution, p 7) Define the purpose of the frame pointer (conventionally the ebp register on x86 processors).

Question 6.2–5: (Solution, p 7) Consider the following C function and its Intel assembly translation at right.

```
int add(int x, int y) {
    return x + y;
}
```

```
add:    pushl %ebp
        movl %esp, %ebp
        ???
        addl %ebx, %eax
        leave
        ret
```

What two instructions should go in place of "???" to load the x parameter into the eax register and the y parameter into the ebx register?

Question 6.3–1: (Solution, p 7) Distinguish between callee-save registers (ebx, esi, edi on Intel processors) and caller-save registers (ecx, eax, edx on Intel processors).

Solution 4.1–1: (Question, p 1) The fetch-execute cycle is the process by which a classical computer executes instructions. In the fetch phase, the computer determines the next instruction to be completed by fetching the instruction from memory. In the execute phase, the computer executes this instruction. The computer alternates between these two phases as long as it is on.

Solution 4.1–2: (Question, p 1) It looks into the PC for a memory address, requests the information at that address from RAM via the bus, and stores RAM's response in the IR.

Solution 4.2–1: (Question, p 1)

```
? 25
25
? 50
75
? 75
-106
```

(This last output is somewhat tricky: In the last ADD instruction, the CPU computes $75 + 75 = 150$, but this exceeds the maximum eight-bit two's-complement number. So the computer wraps around ends up at $150 - 256 = -106$.)

Solution 4.2–2: (Question, p 1) It would read from the user four times before halting (with the AC progressing from 32 to 64 to 96 to -128).

Solution 4.2–3: (Question, p 1)

a.	**address 00000:**	8A 8B 8C
	address 01010:	03 06 0A
b.	**AC:**	01 03 01 8A 03 06 01 8B 04 0A 01 8C 00

Solution 4.3–1: (Question, p 2)

addr	data	(translation)	
00000	100 11110	LOAD	11110
00001	101 11111	STORE	11111
00010	110 00101	ADD	00101
00011	011 00001	JPOS	00001
00100	000 00000	HALT	
00101	000 00001	1	

Solution 4.3–2: (Question, p 2)

```
    READ
    JPOS ok
    STORE n
    SUB n
    SUB n
ok: WRITE
    HALT
n:  0
```

Solution 4.3–3: (Question, p 2)

```
     READ
     STORE n
up:  LOAD i      # display i
     WRITE
     ADD i       # double i
     STORE i
     LOAD n      # repeat if n - i > 0
     SUB i
     JPOS up
     HALT
n:   0
i:   1
```

Solution 5–1: (Question, p 2)

```
eax  7 11 18 29 47
ebx  4  7 11 18 29
ecx  4  3  2  1  0
```

Solution 5–2: (Question, p 3) There will be considerable variation in the answers to these questions, but the following are the actual C programs used to generate the code.

a.

```
#include <stdio.h>

int main() {
    int i, a, n;

    scanf("%d", &n);
    a = 1;
    for(i = 0; i < n; i++) {
        a += i;
    }
    printf("%d %d\n", n, a);

    return 0;
}
```

b.

```
#include <stdio.h>

int main() {
    int i, n;

    scanf("%d", &n);
    for(i = 0; i < n; i *= 2) {
        printf("%d\n", i);
    }

    return 0;
}
```

Solution 6.1–1: (Question, p 3) The processor will first decrease the value in esp by 4, and then it will store the contents of eax in that memory address. In this case, esp would change to $208_{(16)}$, and the four bytes of memory beginning at address $208_{(16)}$ would change to hold $104_{(16)}$.

Solution 6.1–2: (Question, p 3)

```
pushl $10             # Push arguments onto stack
pushl $6
call myst             # Call subroutine
addl $8, %esp         # Pop arguments from stack
movl %eax, %edx       # Copy return value into edx
```

Solution 6.2–1: (Question, p 3) It pushes the current value of eip onto the stack (decreasing esp by 4 in the process) and then it places the address of fact (the first instruction of the subroutine) into eip. This way, when the computer fetches the next instruction to execute, it fetches the first instruction of the fact subroutine, and the return address is lying on the stack for a later ret instruction to pop.

Solution 6.2–2: (Question, p 3) The processor pops the top four bytes off the stack into the eip register. In doing this, it will add four to the esp register to represent the fact that the top four bytes are gone from the stack. The next instruction executed by the processor will be the instruction found at the address popped from the stack.

Solution 6.2–3: (Question, p 4) Before the subroutine is called, the calling code should push the parameters onto the stack, with the first parameter pushed last. The called subroutine, then, can access the parameter values by looking into the stack relative to the stack pointer it receives.
When a subroutine is to return a value, it should place this into the eax register, according to the Intel convention.

Solution 6.2–4: (Question, p 4) The frame pointer is meant to contain the value of the esp at the time the system enters the current subroutine. The purpose of maintaining the frame pointer is to provide a fixed reference point from which to access local variables located on the stack and parameters (accessing items on the stack relative to esp is inconvenient since it shifts with every push and pop instruction). (Secondarily, it is also useful to have this so that the program can restore esp to the proper value before returning from the subroutine without worrying about taking care to pop each thing off the stack that was pushed.)

Solution 6.2–5: (Question, p 4)

```
movl 8(%ebp), %eax
movl 12(%ebp), %ebx
```

Solution 6.3–1: (Question, p 4) A subroutine is allowed to change the caller-save registers without restoring them, but it must ensure that callee-save registers, if used, are restored to their values on entering the subroutine.