PHASE I (March 21)

The Operating Systems project is divided into two (possibly three) phases: a simple batch operating system and a multiprogrammed batch operating system (and possibly a more sophisticated processor manager).

The description of Phase I is given below in three parts: 1. characteristics of the hardware, 2. machine language - assembly language, 3. characteristics of the software.

CHARACTERISTICS OF THE HARDWARE

The computer to be simulated has the following characteristics.

MEMORY: The main memory consists of 4096 words (locations 0 to 4095). A word is the basic addressing unit. Each word is 12 bits long.

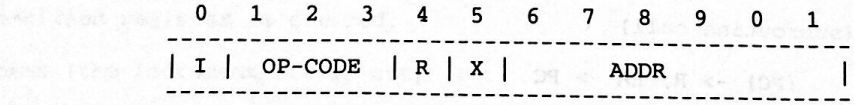
CPU: The CPU contains ten registers (registers R0 to R9). Registers R0 and R1 are used for storing constant numbers 0 and 1, respectively. Register R2 is the program counter, PC, and register R3 is the instruction register, IR. Registers R4 through R9 are scratch-pad or general-purpose registers that may be used for arithmetic operations, but only register R4 can be used as an index register. Register R5 is the accumulator. The R bit (as specified below) can be either 0 or 1, which refers to the accumulator R5 or the index register R4, respectively. All arithmetic is done in 2's complement. A system-wide CLOCK (maintained in the CPU) will be used to time the execution of user programs. In Phase I, the CLOCK is to be reset before each user program is loaded and executed (since each batch is of size one in the absence of a memory manager, a scheduler, etc.). The CLOCK will be incremented by one virtual time unit each time an instruction is executed. If the instruction is an I/O instruction, the CLOCK will be incremented an additional 10 virtual time units.

INPUT/OUTPUT DEVICES: In the first phase of the project, all I/O for user jobs will be done interactively. In other words, the keyboard/screen will simulate or virtualize the input/output devices for user programs. For outputting the information collected and/or generated by your operating system, files will serve as virtualized output devices.

MACHINE LANGUAGE - ASSEMBLY LANGUAGE

The instruction set of our computer can be divided into four types, each having a different format, as explained below.

TYPE I INSTRUCTIONS



where
- I - indirect bit
- OP_CODE - operation code
- R - arithmetic register
- X - index bit
- ADDR - address field

Instructions of this type specify an operand whose effective address has to be calculated. The simulated machine supports PC relative addressing with two addressing modes: indirect addressing (deferred addressing) and indexing.

The addressing mode is determined by the status of the indirect bit and the index bit as follows:

I	X	
1	1	indirection + indexing
1	0	indirection
0	1	indexing
0	0	direct addressing

The addressing mode determines how the EFFECTIVE ADDRESS is to be calculated.

EFFECTIVE ADDRESS CALCULATION

For every effective address calculation, the first step is to add the ADDR field in the instruction to the current content of the PC.

A = PC(0-11) + INSTR(6-11SE) SE = Sign Extended

This address is interpreted based on the addressing mode of the instruction.

```
EA = A           direct
EA = A + (R4)    indexing
EA = (A)         indirection
EA = (A) + (R4)  indexing + indirection
```

Six instructions have type I format. They as described below.

MNEMONIC OP-CODE	NUMERIC OP-CODE	SEMANTICS

1. Logical AND

   AND 000 (R) ^ (EA) -> R

   The contents of register R are logically ANDed with the contents of memory location EA, and the result is placed in register R.

2. Addition

   ADD 001 (R) + (EA) -> R

   Add the contents of memory location EA to the contents of register R, and place the result in register R.

3. Store

   STR 010 (R) -> EA

   Store the contents of register R in memory location EA.

4. Load

   LD 011 (EA) -> R

   Load the contents of memory location EA into register R.

5. Unconditional branching

   JMP 100 EA -> PC

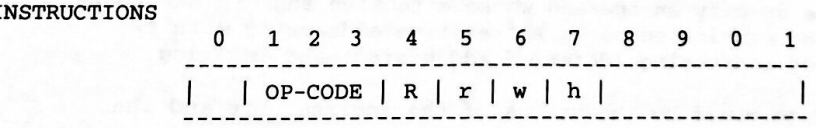
   Transfer control to the instruction at memory location EA. To do so, the address of the desired memory location has to be placed in the PC.

6. Jump and Link (subroutine call)

   JPL 101 (PC) -> R; EA -> PC

   Save the current content of the PC (return address) and transfer control to the subroutine at location EA.

TYPE II INSTRUCTIONS



where
- bit 0 - not used
- OP-CODE - operation code (110)
- R - arithmetic register
- r - read bit
- w - write bit
- h - halt bit

All I/O operations are instructions of type II. All of these instructions have the same op-code. The status of the read, write, and halt bits determines the operation to be performed.

```
read bit  = 1   ---> read operation
write bit = 1   ---> write operation
halt bit  = 1   ---> halt operation
```

The op-code will cause only one of the operations specified above. If more than one of the 3 bits is set, the CPU will trap to the ERROR_HANDLER and an

appropriate error message will be generated and output followed by a memory dump.

Operations:

1. Read (the read bit is set)

   RD 110 keyboard -> R

   Get a 3 digit HEX number (one word of internal data) from the keyboard (the user input device for this phase), and place it in register R.

2. Write (the write bit is set)

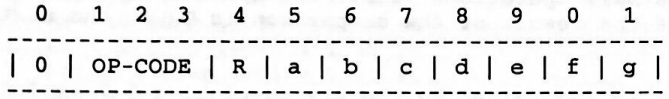
   WR 110 (R) -> screen

   The contents of register R (3 HEX digits) will be displayed on the screen (user output device for this phase).

3. Halt (the halt bit is set)

   HLT 110 stop execution

TYPE III INSTRUCTIONS



where
- bit 0 - instruction type specifier (0)
- OP-CODE - operation code (111)
- R - the accumulator or the index register
- a - clear bit
- b - increment bit
- c - complement bit
- d - byte swap bit
- e - rotate left bit
- f - rotate right bit
- g - shift magnitude

Instructions of type III and type IV have OP-CODE 111. Bit 0 is used to specify the type of instruction, 0 for type III and 1 for type IV.

Bits 5-11 determine what actions are to be performed on the accumulator. Not all six actions may be specified in one instruction. If a wrong combination of actions is specified, the CPU will trap to the ERROR_HANDLER and an appropriate error message will be generated and output followed by a memory dump.

1. Clear (the clear bit is set)

   CLR 111 0 -> R

   The specified register is cleared.

2. Increment (the increment bit is set)

   INC 111 (R) + 1 -> R

   Increment the specified register by 1.

3. Complement (the complement bit is set)

   COM 111 $\overline{(R)}$ -> R

   One's complement of the data in the register is placed in the register.

4. Byte swap (the byte swap bit is set)

   BSW 111 (R) shift (6) -> R

   The contents of the register are shifted circularly six bits (either way).

5. The magnitude bit specifies how many bits are to be shifted circularly right or left by the rotate operations.

   if g = 0, rotate 1 bit, otherwise rotate 2 bits

5a. Rotate left (the rotate left bit is set)

RTL 111 (R) shift(1 or 2) left -> R

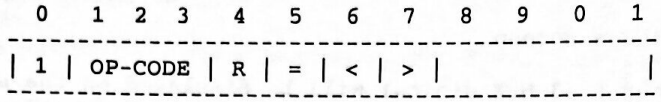
Rotate the contents of the register one or two bits left.

5b. Rotate right (the rotate right bit is set)

RTR 111 (R) shift(1 or 2) right -> R

Rotate the contents of the register one or two bits right.

TYPE IV INSTRUCTIONS



where bit 0 - instruction type specifier (1)
OP-CODE - operation code (111)
R - arithmetic register
= - equal bit
< - less bit
> - greater bit

All operations of type IV have op-code 111.

Operations of type IV are SKIP operations, i.e., the content of register R is compared to zero, if the result of the comparison is TRUE, the next instruction is skipped.

The comparison to be made is determined by the status of bits 5-7 as follows.

equal	less	greater	
0	0	0	---> no skip
0	0	1	---> greater
0	1	0	---> less
0	1	1	---> not equal
1	0	0	---> equal
1	0	1	---> greater or equal
1	1	0	---> less or equal
1	1	1	---> unconditional skip

1. No skip

NSK 111 no operation (no-op)

If bits 5-7 are 0, the next instruction will be executed normally.

2. Greater (the greater bit is set)

GTR 111 if (R) > 0 then (PC) + 1 -> PC

If the contents of register R is greater than zero, increment the PC (skip the next instruction).

3. Less (the less bit is set)

LSS 111 if (R) < 0 then (PC) + 1 -> PC

4. Not equal (the less and the greater bits are set)

NEQ 111 if (R) <> 0 then (PC) + 1 -> PC

5. Equal (the equal bit is set)

EQL 111 if (R) = 0 then (PC) + 1 -> PC

6. Greater or equal (the equal and the greater bits are set)

GRE 111 if (R) >= 0 then (PC) + 1 -> PC

7. Less or equal (the equal and the less bits are set)

LSE 111 if (R) <= 0 then (PC) + 1 -> PC

8. Unconditional skip (all three bits are set)

USK	**111**	(PC) + 1 -> PC

The next instruction is skipped regardless of the contents of register R.

SUMMARY OF INSTRUCTIONS (op-codes)

AND	**000**	(R) ^ (EA) -> R
ADD	**001**	(R) + (EA) -> R
STR	**010**	(R) -> EA
LD	**011**	(EA) -> R
JMP	**100**	EA -> PC
JPL	**101**	(PC) -> R; EA -> PC
RD	**0110R1000000**	keyboard -> R
WR	**0110R0100000**	(R) -> screen
HLT	**0110R0010000**	stop execution
CLR	**0111R1000000**	0 -> R
INC	**0111R0100000**	(R) + 1 -> R
COM	**0111R0010000**	(R) -> R
BSW	**0111R0001000**	(R) shift(6) -> R
RTL	**0111R000010x**	(R) shift(1 or 2) left -> R
RTR	**0111R000001x**	(R) shift(1 or 2) right -> R
NSK	**1111R0000000**	no operation
EQL	**1111R1000000**	if (R) = 0 then (PC) + 1 -> PC
GTR	**1111R0010000**	if (R) > 0 then (PC) + 1 -> PC
LSS	**1111R0100000**	if (R) < 0 then (PC) + 1 -> PC
NEQ	**1111R0110000**	if (R) <> 0 then (PC) + 1 -> PC
GRE	**1111R1010000**	if (R) >= 0 then (PC) + 1 -> PC
LSE	**1111R1100000**	if (R) <= 0 then (PC) + 1 -> PC
USK	**1111R1110000**	(PC) + 1 -> PC

CHARACTERISTICS OF THE SOFTWARE

The SYSTEM, AKA Phase I, (i.e., the main function) will be the driver for this simulation. The SYSTEM is to contain or control four subsystems: LOADER, CPU, MEMORY, and ERROR_HANDLER.

The LOADER will be responsible for loading each user program into main memory. In Phase I, each user program is loaded into the main memory from location 0. A user program has to be converted from HEX to BINARY (either explicitly or implicitly) before it may be loaded into memory. This conversion is external to the SYSTEM. The LOADER will need to access the main memory, but this access can only be made via the MEMORY subsystem by using a "buffer" of appropriate size (the DMA block transfer analogy).

After a user program has been loaded, it has to be executed. The CPU routine will be responsible for the execution. Whenever a user program terminates, control is transferred back to the SYSTEM.

ERROR_HANDLER PROCEDURE

ERROR_HANDLER (N) with N -> error number

This routine handles warnings, errors, and special conditions. In case of an warning or error (e.g., overflow, suspected infinite loop, invalid loader format character, invalid op-code, address out of range, bad trace flag, illegal input, program size too large, etc.), the CPU or LOADER will trap to the ERROR_HANDLER by a number (error number). This routine will output an appropriate warning or error message (using the warning/error number and based on the nature of the warning/error) and simply output a warning message or, in case of an error, dump the memory (only the first 256 words) by calling the MEMORY procedure with the DUMP option.

MEMORY PROCEDURE

MEMORY (X, Y, Z) with
- X -> READ, WRIT, or DUMP (the control signal)
- Y -> memory address (EA) (the memory address register or MAR)
- Z -> variable (the memory buffer register or MBR)

READ operation: The contents of memory location EA will be read into

variable Z. (EA) -> Z

WRIT operation: The contents of variable Z (i.e., the contents of where Z points to) will be written into memory location Y.

(Z) -> EA

Variable Z may be a register used by the CPU, or it may be a buffer used by the LOADER to help load each user program into main memory.

DUMP operation: The contents of physical memory will be output as follows.

0000	word0	word1	word2	word3	word4	word5	word6	word7
0008	word8	word9	wordA	wordB	wordC	wordD	wordE	wordF
0010	word10	word11	word12	word13	word14	word15	word16	word17
:	:	:	:	:	:	:	:	:
:	:	:	:	:	:	:	:	:
0FF8	. . .	. . .	. . .	. . .	. . .	. . .	. . .	WordFFF

For Phase I of the project, only the first 256 words of memory (i.e., 1/16 of the physical memory) will have to be output with the above format.

LOADER PROCEDURE

LOADER (X, Y) with X as the starting address and Y as the trace switch

The LOADER loads the information, i.e., each user job, from the input device (i.e., a file). All the information in the input records is in HEX.

Format of a set of records:

first: length (the length of a user program as a three-digit HEX number)
user program records (each record consists of twelve HEX digits specifying four words of information).
last: start address trace switch
where: "start address" is the address of the first instruction to be executed; it will be given as a three-digit HEX number; it will be the initial value of the PC in the CPU routine; the "start address" is separated from the "trace switch" by a blank space

"trace switch" consists of one bit; if the value of the "trace switch" is 1, a trace has been requested and it must be generated

If the trace switch parameter of the procedure LOADER is set by a user job, the CPU will generate and output to a file (trace_file) the following information in HEX for each instruction.

1. PC
2. instruction
3. R register and EA (if applicable)
4. contents of R and EA (if applicable) before execution
5. contents of R and EA (if applicable) after execution

The loader has a buffer (one or more registers of size 12 bits each) to be used for block transfer between the loader and the Memory. A function called HEXBIN may be written to convert a string of 3 Hex digits to one 12-bit binary value. A function called BINHEX may be written to do the opposite.

CPU PROCEDURE

CPU (X, Y) with X as the value of the PC and Y as the trace switch

The CPU will loop indefinitely executing instructions until a HLT instruction or an I/O instruction is interpreted.

At the time when an HLT instruction is encountered, control is returned to the SYSTEM, and the next program in the batch (if there is one) will be run. When there are no more user programs, the simulation terminates. Note that the next program to execute will be selected by the scheduler from among memory-resident user programs, however, in Phase I each batch is of size one job (multiprogramming and scheduling will be added in Phase II).

When an I/O instruction is encountered, control is returned to the SYSTEM and the CLOCK is incremented by 10. The SYSTEM reads data from the input device (i.e., the keyboard in Phase I) and stores it in register R. After this is done, the SYSTEM calls the CPU again. Notice that when an I/O operation is executed, the CLOCK will first be incremented by one virtual time unit in the CPU (execution time of one instruction) and then by another ten virtual time units in the SYSTEM (wait time for I/O). The value of the CLOCK in HEX is output at the termination of each user program.

A test user job is provided below. You may want to write your own trivial test jobs to exercise/test individual instructions and groups of instructions before attempting to execute this test job on your simulation.

FACTORIAL: The following program finds the factorial of a given number. If the given number is 0 or 1, a 1 is output. Otherwise, the factorial of the input is output. The number and its factorial are output.

```
MULTIPLY    STR, R5, RETURN       ;save return address
            LD,  R5, I            ;load the loop control
            STR, R4, STEP         ;save the number to be multiplied
            ADD, R5, DCR          ;i = i - 1;
            NEQ, R5,              ;if i = 0;
            JMP,     *RETURN      ;then return('*' is for indirect)

LOOP        ADD, R4, STEP         ;else add step to R1
            ADD, R5, DCR          ;i = i - 1;
            NEQ, R5,              ;if i = 0;
            JMP,     *RETURN      ;then return
            JMP,     LOOP         ;else multiplication not done

START       RD,  R5               ;get number
            STR, R5, NUM          ;save it in memory
            LD,  R4, ONE          ;in case of num = 0
            NEQ, R5               ;if num = 0
            JMP,     PRINT        ;then fact(0) = 1

FACT        STR, R5, I            ;else calculate factorial
            JPL, R5, MULTIPLY     ;call subroutine to multiply
            LD,  R5, I            ;restore i
            ADD, R5, DCR          ;i = i - 1
            EQL, R5               ;if i = 0
            JMP,     FACT         ;then fact not over

PRINT       LD,  R5, NUM          ;get number
            WR,  R5               ;output number
            WR,  R4               ;output its factorial
            HLT

NUM         DATA     000
I           DATA     000
STEP        DATA     000
DCR         DATA     FFF
ONE         DATA     001
RETURN      DATA     000
```

Loader Format ("input" to the SYSTEM, i.e., "input" to your simulator as a file)

```
020
21E319299119
F30C19195115
F30C1543B640
20D390F30406
20A52E308109
F4043A303620
6A0610000000
000FFF001000
00B 0
```

This is a user job consisting of the first record, the user program, and the last record. This is the "input" to your simulation. The "input" that this program needs will be provided at the keyboard.

NOTES:

SPECIFICATION Note: The LOADER subsystem will read/load as specified above. There will be no preprocessing/parsing/checking of the loader format. Any deviation from the specification must be approved by the instructor. You must keep up with class discussions concerning the project specification. As a result of the discussions in class, there may be some changes in the project specification as outlined in this document.

NAMING STANDARDS: The driver of the first step of the project is to be named SYSTEM if possible (this depends on the language of implementation). Other descriptive names such as MEMORY, CPU, LOADER, ERROR_HANDLER, etc. must be used, as specified in this document, to name the related functions.

IMPORTANT Design Note: Conceptually, your simulation is a virtual machine. Conventional architecture and operating system component functionalities and restrictions should be adhered to, i.e., your code should be properly modularized not by size but by function. There is ample opportunity for tailoring/customizing or personalizing/individualizing your simulation during implementation. This refers to the relative internal implementation latitude vs. strict external functional behavior. Nonetheless, any significant departure from the specification must be approved by the instructor.

DOCUMENTATION GUIDELINES: Your simulation program (i.e., the instruction set simulation of the underlying architecture and the batch operating system) must include external and internal documentation. External documentation (for conceptualization) appears in the form of comments as header blocks and is an explanation of the functionality of the respective program/subprogram/function/method/module, a description of the global variables, and a discussion of the implementation approach. Internal documentation (for readability) is the documentation that is mixed with the code and is used to clarify potentially obscure segments of code such as case statements, loops, or conditions. Use meaningful names and blank lines to enhance the readability and understandability of your code. DO NOT pollute the user's environment with unnecessary echos and prompts. Note that the assembly version of the test jobs are programs too, and thus must be documented and commented. You are to follow the last section of this document titled "DOCUMENTATION GUIDELINES FOR ALL PROGRAMMING ASSIGNMENTS".

TRACE Information Output Note: When the trace bit is on, the trace information generated should be put in a separate file (i.e., the trace file) in columns with appropriate column headers. Note that the trace file you will generate and the trace file that you will turn in as part of your deliverables are not necessarily the same. To save paper, you will turn in only some representative parts of the actual trace file that your system generates by following the steps outlined in the PROCEDURE appendix below. Voluminous printouts will not be accepted. Printout font point size must not be less than 10.

DUE DATE Note: You are to sign up for a demonstration of your project, which is going to be on the day the project is due. Demonstrations will be on CSX (the Computer Science Department's main instructional computer), thus you must develop your program on CSX, or upload your program for demonstration. In the latter case, you should be aware of and handle potential language and compiler incompatibility problems between your development platform and the demonstration platform. In short, your project (simulation program) must compile and run on CSX. Also, the deliverables for this step are to be submitted at the time of the demonstration. All deliverables are to be hard copy (i.e., paper copy) submissions. No soft copy submissions such as email, CDs, or flash drives will be accepted. No handin submissions are required either.

LATE PENALTY: 10 points per calendar day late, out of a total of 100 points. The precise due time is the demo time. For late projects, no later than one week after the due date, penalty calculations will be based on the system time stamp on CSX for ALL of the deliverables including the software engineering report.

INPUT: The input to your STEP I is a file of hex digits which is a "user job" or "user program" in loader format. The "input" to the user job, if indeed it requires any input, will be provided at the keyboard.

OUTPUT: It is only the user job's output (more specifically, the result of execution of WR instructions) that is to be displayed on the screen. The following information is to be output to a file (a virtualized output device) upon completion of a user job. Note that this "output" consists of some information generated by your operating system on behalf of the user job.

1. Cumulative job identification number (this number is reset each time that you start your operating system simulation). This should be 1 for this step since your simulation will handle exactly one "user job" each time that you run it. The reason is the absence of a multiprogramming memory manager and a scheduler, and the lack of a JCL in STEP I. Consequently, your operating system will run a single job and then stop, there can be no transition to another job, and, therefore, the output file and the trace_file are not cumulative.

2. Any warning messages resulting from handling a user job.

3. A message indicating the nature of termination: normal or abnormal; and, if abnormal, a descriptive error message.

4. Output of the current user job (if job terminated normally). As mentioned earlier, this is in addition to the output being displayed on the screen.

5. CLOCK value in HEX at termination.

6. Run time for the job in decimal, subdivided into execution time, input/output time, etc. Note that all "times" refer to virtual time periods as measured by differences between values of the simulated system's CLOCK at different instances.

In addition to the above-mentioned output file, there may be another output file called a trace_file that is generated as a result of the execution of a user job provided that the trace bit of the user job is set.

Remember to tag all relevant output in the output file by adding the term "HEX" or "DECIMAL", as appropriate.

YOU ARE TO TURN IN THE FOLLOWING ITEMS (by following the steps outlined in the PROCEDURE appendix below):

- Your Phase I simulation source code listing (modular, readable, and well-documented).

- Your own non-trivial test job in the given assembly language format consisting of English explanation and assembly code, with in-line comments as necessary, followed by its loader format. (Use the test job provided above in this specification as a template.)

- Sample compilation and executions of the test job provided with the trace switch on and with the trace switch off, along with a printout of some of the trace_file (see TRACE Information Output Note above).

- Sample compilation and executions of your own test job with the trace switch on and with the trace switch off, along with a printout of some of the trace_file (see TRACE Information Output Note above).

- Sample executions (plus their assembly and load module) of a number of your own small test jobs (or modified versions of the above-mentioned sample test jobs) containing injected errors (see ERROR_HANDLER above) in order to exercise your ERROR_HANDLER module; the trace switch should be off for the test jobs used in these runs.

- A two to three page write-up of the software engineering issues involved in the design and development of your Phase I simulation that must include the following items.

  - Your general approach to the problem. This does not mean a description of your implementation or a description of the system. It refers to things like whether or not you used a design language, whether or not you used pseudo-code, whether or not you used a flow chart, etc. Note that this

does not imply that a flow chart, a pseudo-code, etc. of your design is required for STEP I. If you do need to include a flow chart, etc., it belongs in the external documentation part of your simulation code and not in the software engineering issues. This is a simple question with at a brief answer.

- A list of the utilities used (e.g., makefile, gmake, various debuggers such as gdb, gtester, SCCS, RCS, etc.).

- Bulk complexities of your simulation program including the following items:
  = the total number of lines of code as well as its subdivision into the number of declarations, comment lines, executable statements, blank lines, etc.
  = the number of decisions
  = the number of procedures/functions/methods
  = the number of classes, etc. as appropriate and applicable to your implementation language

- An approximate break-down of the time spent in the design, coding, and testing of your simulation.

- Your comment on the portability of your simulation to other operating systems, architectures, etc.

- A brief justification of your choice for the implementation language.

==========

GENERAL GUIDELINES FOR ALL PROGRAMMING ASSIGNMENTS

1. External Documentation - At the beginning of the main routine, there should be documentation containing items a through g below. At the beginning of all other routines (i.e., subprograms, functions, methods, modules, classes, etc.), there should be documentation containing items f and g only.

a. Your name.

b. Course number.

c. Assignment title or number.

d. Date

e. A brief description of the global variables, if any.

f. A brief description of what the routine does, i.e., a short explanation of the functionality of the program/subprogram/function/method/module/class.

g. If applicable, depending how thoroughly and accurately the implementation reflects the specification, a critique of the program/subprogram/function/method/module/class indicating the ways in which it does not meet the programming assignment's specification.

2. Internal Documentation - This is the documentation that is "mixed" or interspersed with the code to clarify the potentially obscure segments of the code, e.g., case/switch statements, loops, conditional statements, and procedure calls/returns.

a. All major variables should be commented in the declarations except those that are patently self-explanatory.

b. The code should also be commented, not too much though. As a general rule, when in doubt, use a comment.

c. Use meaningful names and blank lines to enhance the understandability of your code. Blank lines help isolate code segments as spatial localities.

3. Program Layout and Data Structures -

a. The code should be well-structured, with indentation showing the

general program logic and flow. GOTOs (for implementation languages that actually include GOTO among their instructions) and other unconditional transfers of control should be used only for a fairly good reason, otherwise they should be avoided.

b. The program should be modular. Program modules generally should not be larger than the size of a typical monitor screen in terms of the number of lines.

c. Choose your data structures carefully, e.g., linked lists are generally inefficient and should be avoided, use them only if their use is justifiable.

4. Input and Output -

a. Use prompts and echos only when they contribute to the overall user-friendliness of your program. Avoid unnecessary prompts and echos.

b. Your output should be well-formatted, commented, and understandable. To enhance comprehensibility, arrange your output in rows, columns, blocks, etc. with appropriate explanatory headers, within the framework delineated in the programming assignment's specification.

c. The submitted program and the output files should not include any debugging code or debugging comments.

Notes:

(1) Consider the probability that the system (i.e., CSX) could be down, or have system hardware/software problems, a few days before the due date.

(2) Going through the "design/desk-check/redesign cycle followed by coding" is more effective and efficient than going through the "quick design followed by code/output-check/re-code cycle".

(3) It should go without mentioning that a clean compilation with no warnings is called for (applicable to compiled languages and not interpreted ones).

(4) Keep a copy of the deliverables that you submit for your own reference.

==========

PROCEDURE Appendix

[A handout outlining the "procedure to obtain a final copy of your program/output" will be distributed in the near future.]

==========