The Operating Systems course project is divided into three steps:

1. a simple batch system
2. a simple batch system with memory management
3. a multiprogramming batch system with process management

The description of Step1 (due February 25) will be given in 3 parts:

1. characteristics of the hardware
2. instructions
3. characteristics of the software

1. CHARACTERISTICS OF THE HARDWARE

The computer to be simulated has the following characteristics:

MEMORY

The main memory consists of 256 words (locations 0 to 255). A word is the basic addressing unit. Each word is 16 bits long.

CPU

The computer being simulated is a stack machine; that is, an architecture which supports one or more stacks.

a. The architecture for this machine supports one stack (S), which consists of 7 registers, each register being 16 bits wide.

S [7 : 1] <15 : 0>

The top element of the stack may be used as an accumulator.
Initially the stack is empty (the Top Of Stack pointer is 0).

b. The Top Of Stack (TOS) points to the current top of the stack.
Since there are at the most 7 elements on the stack, 3 bits are necessary to hold the value of TOS.
A value of zero represents an empty stack.
Attempting to decrement the TOS below zero results in an illegal instruction trap.

c. The CPU also needs to keep a register for the program counter (PC).
The PC is 7 bits wide. Both the PC and the address calculation will allow an individual user segment to address only half of memory.

d. The CPU maintains a register to hold the instruction being executed, called the INSTRUCTION REGISTER (IR). The IR has to be 16 bits wide.

e. A BASE REGISTER (BR) is necessary to hold the base address of the program being executed. It has to be 8 bits wide to address all of memory.

INPUT/OUTPUT DEVICES

Input and output devices are generally simulated by files (for user jobs in loader format, trace files, etc.), however in Step1 the individual user job I/O is via keyboard/screen.

CLOCK

A system-wide CLOCK will be used to time the execution of programs in terms of virtual time units (vtu). Later in the specification, it will be made clear when the CLOCK is to be incremented.

2. INSTRUCTIONS

The system supports data of types integer and bit.
The range of integer values may span from -2^13 to (2^13)-1. The values will be represented in 2's complement.

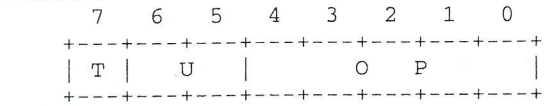
Instruction format will be of 2 types, zero-address and one-address instructions (short or long). Due to the nature of the machine and its high reliance on stacks and stack operations, many instructions are zero address instructions that utilize the top 2 elements of the stack. Furthermore, because of the flexibility introduced through the stack architecture, many operations may be used with both zero and one

address.

1. ZERO-ADDRESS INSTRUCTIONS (7 bits)

A zero address instruction is an instruction of the simplest form. This instruction format will utilize the top or top two elements of the stack depending upon the operation to be performed or will generate a program halt.

FORMAT:



where

T = instruction type (short = 0)
U = unused
OP = op code

FUNCTION:

```
S[TOS-1] <- (S[TOS]) op (S[TOS-1])
TOS      <- TOS-1
```

The operator "op" is applied to the contents of the top two elements of the stack and the result is placed on the stack. Or

S[TOS] <- op (S[TOS])

The operator "op" is applied to the contents of the top of the stack and the result is placed on the stack. TOS does not change with the execution of this instruction.

Whenever possible, there will be two instructions of type 1 per memory word. If it is not possible to have two instructions of type 1 in a word, the unused bits will be padded with zeros (i.e., a NOP will be inserted).

2. ONE-ADDRESS INSTRUCTIONS (16 bits)

A one-address instruction will contain an instruction type indicator of 1 bit, a 5-bit op code, an indexing bit, and a memory displacement address. This instruction type will also utilize the stack directly for operations requiring more than one operand.

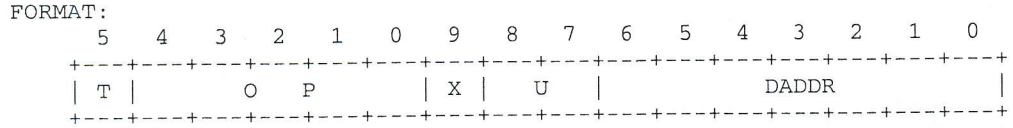
EFFECTIVE ADDRESS CALCULATION

The effective address calculation computes the virtual address of the operand. There are two modes of addressing:

```
EA = DADDR              without indexing
EA = DADDR + (S[TOS])   with indexing
```

The conversion of the virtual address into a real address is done in the Memory Routine when the contents of the BR are added.

FORMAT:



where

```
 0 -  6 : displacement addressing
 7 -  8 : unused
      9 : index bit (0:no indexing; 1:indexing)
10 - 14 : op code
     15 : instruction type (long = 1)
```

FUNCTION:

```
S[TOS] <- (S[TOS]) op (EA)

TOS    <- TOS + 1
S[TOS] <- (EA)        for the PUSH operation

EA     <- (S[TOS])
TOS    <- TOS - 1     for the POP operation
```

SUMMARY OF ZERO-ADDRESS INSTRUCTIONS

```
00000   NOP   no operation
00001   OR    S[TOS-1] <- (S[TOS]) V (S[TOS-1]); TOS <- TOS-1
00010   AND   S[TOS-1] <- (S[TOS]) /\ (S[TOS-1]); TOS <- TOS-1
00011   NOT   S[TOS]   <- ~(S[TOS])
00100   XOR   S[TOS-1] <- (S[TOS] XV (S[TOS-1]); TOS <- TOS-1

00101   ADD   S[TOS-1] <- (S[TOS] + (S[TOS-1]); TOS <- TOS-1
00110   SUB   S[TOS-1] <- (S[TOS-1] - (S[TOS]); TOS <- TOS-1
00111   MUL   S[TOS-1] <- (S[TOS] * (S[TOS-1]); TOS <- TOS-1
01000   DIV   S[TOS-1] <- (S[TOS-1] / (S[TOS]); TOS <- TOS-1
01001   MOD   S[TOS-1] <- (S[TOS-1] mod (S[TOS]); TOS <- TOS-1

01010   SL    S[TOS]   <- logical shift one bit left (S[TOS])
01011   SR    S[TOS]   <- logical shift one bit right (S[TOS])

01100   CPG   S[TOS+1] <- (S[TOS-1] > (S[TOS]); TOS <- TOS+1
01101   CPL   S[TOS+1] <- (S[TOS-1] < (S[TOS]); TOS <- TOS+1
01110   CPE   S[TOS+1] <- (S[TOS-1] = (S[TOS]); TOS <- TOS+1

01111   BR    -----
10000   BRT   -----
10001   BRF   -----
10010   CALL  -----

10011   RD    TOS <- TOS + 1; S[TOS] <- in
10100   WR    out <- (S[TOS]); TOS <- TOS-1

10101   RTN   PC <- (S[TOS]); TOS <- TOS-1

10110   PUSH  -----
10111   POP   -----

11000   HLT   program termination
```

NOTE:

The EA stored in the PC by all the branching type instructions is a virtual address, i.e., the contents of the BR are not added to it. If all the bits in a word are zero, we have a FALSE value, otherwise we have a TRUE value.

SUMMARY OF ONE-ADDRESS INSTRUCTIONS

```
00000   NOP   -----
00001   OR    S[TOS] <- (S[TOS]) V (EA)
00010   AND   S[TOS] <- (S[TOS]) /\ (EA)
00011   NOT   -----
00100   XOR   S[TOS] <- (S[TOS]) XV (EA)

00101   ADD   S[TOS] <- (S[TOS]) + (EA)
00110   SUB   S[TOS] <- (S[TOS]) - (EA)
00111   MUL   S[TOS] <- (S[TOS]) * (EA)
01000   DIV   S[TOS] <- (S[TOS]) / (EA)
01001   MOD   S[TOS] <- (S[TOS]) mod (EA)

01010   SL    -----
01011   SR    -----

01100   CPG   S[TOS+1] <- (S[TOS]) > (EA); TOS <- TOS + 1
01101   CPL   S[TOS+1] <- (S[TOS]) < (EA); TOS <- TOS + 1
01110   CPE   S[TOS+1] <- (S[TOS]) = (EA); TOS <- TOS + 1

01111   BR    PC <- EA
10000   BRT   if (S[TOS]) = TRUE  then PC <- EA; TOS <- TOS-1
10001   BRF   if (S[TOS]) = FALSE then PC <- EA; TOS <- TOS-1
10010   CALL  TOS <- TOS+1; S[TOS] <- (PC); PC <- EA

10011   RD    ----
10100   WR    ----
```

```
10101   RTN   ----

10110   PUSH  TOS <- TOS+1; S[TOS] <- (EA)
10111   POP   EA  <- (S[TOS]); TOS <- TOS-1

11000   HLT   -----
```

NOTE:

All Branch instructions except for RTN are one-address, full-word instructions. The RTN instruction will always be NOP padded by the assembler. Since branching can only occur to full-word addresses, additional NOP padding may be necessary for any instructions that are the target of a branch instruction. In this case the NOP would potentially precede the instruction in order to advance to a full-word boundary.

3. CHARACTERISTICS OF THE SOFTWARE

SYSTEM (Step1) will be the driver for this simulation. It will contain modules: LOADER, CPU, MEMORY, and ERROR_HANDLER. LOADER will be responsible for loading user jobs (in loader format) from the input device (file) into main memory via a LOADER buffer of size four words. A user job has to be converted from HEX to BINARY before it may be loaded. The LOADER will need to access main memory, but this access may be made only via the MEMORY routine.

After the program has been loaded, it has to be executed. The CPU routine will be responsible for execution. Whenever a program terminates, control is transferred to SYSTEM. SYSTEM will then verify if there is another incoming job (i.e., another user job in the input file) and the cycle will be repeated. For Step1, batches are of size one job each, so there should be no cycling.

MEMORY PROCEDURE

```
MEMORY (X, Y, Z)
    X   "READ" or "WRITE"
    Y   memory address (EA)
    Z   variable
```

The memory routine is responsible for translating all virtual addresses to real addresses and then performing the actual read or write operation on memory. In this first step of the project, the translation is to simply add the contents of the BR to the virtual address (later it will involve a look-up in the segment tables and page tables). Notice that this addition of (BR) is done not only to effective addresses specified in user instructions but also to the PC and to addresses passed by the LOADER routine.

READ operation:

Notice that the READ operation is a MEMORY MANAGER function and NOT the RD operation from the instruction set.

FUNCTION:
The contents of Y (memory location EA) will be read into the variable Z.

WRITE operation

Notice that the WRITE operation is a MEMORY MANAGEMENT function and NOT the WR operation from the instruction set.

FUNCTION:
The contents of the variable Z will be written into main memory location Y. The variable Z may be the top of the stack used by the CPU, a buffer used by the LOADER, or other temporary scratch pad registers needed by the CPU.

LOADER PROCEDURE

```
LOADER (X, Y)
    X -> starting address
```

Y -> trace switch

The LOADER loads the information from the input device. All information on the input lines is in HEX. The LOADER has a buffer of size four words (i.e., four local registers). A HEX_BINARY procedure may be written to convert strings of Hex digits to BINARY bits.

LOADER FORMAT:

first line: job id, load address, initial pc, size, trace flag
where job id is the two-digit HEX id of the job
load address is a two-digit HEX number indicating where this job should be loaded
initial pc is the address of the first instruction to be executed given as a 2-digit HEX number
size is the length of the job (# memory words) given as a two-digit HEX number
trace flag given as a 1-digit HEX number

job lines: (actual program instructions)
Each line consists of 16 HEX digits specifying 4 words of information. If fewer than 4 words appear on the last data line, there will be no zero padding.

If the trace flag parameter is set, the CPU will provide the following information for each instruction (to be output to a file called trace_file).

- (PC)
- (BR)
- (IR)
- (TOS) and (S[TOS]) before execution
- EA and (EA) before execution (if applicable)
- (TOS) and (S[TOS]) after execution
- EA and (EA) after execution (if applicable)

All information is to be given in HEX numbers and is to be labeled.

CPU PROCEDURE

CPU (X, Y)
X -> program counter (PC)
Y -> trace switch

The CPU will loop indefinitely, fetching, decoding, and executing instructions until a HLT instruction or an I/O instruction is interpreted. The SYSTEM CLOCK is incremented by 1 for every short (zero-address) instruction that is executed. The CLOCK is to be incremented by 4 for every long (one-address) instruction executed. The CLOCK is to be incremented by the CPU as the first step of execution of all instructions. If the IR contains 2 short instructions, both will be executed before the next (FETCH; PC <- PC+1) takes place. The only exception to this is RTN. IF RTN is the first instruction of a pair of short instructions, the NOP that follows the RTN will never be executed (and therefore will not cause the CLOCK to be incremented). When an I/O instruction is encountered, control is returned to the SYSTEM and the CLOCK is incremented by an additional 15 virtual time units.

At the time when a HLT instruction is encountered, control is returned to the SYSTEM, the job is terminated, and the next job in the batch, if any, is started (for Step1, batches are of size one job each).

An ERROR_HANDLER subsystem is required for warnings and errors. For example, CPU or the LOADER will trap to the ERROR_HANDLER by a number or an appropriate error code (as the error or warning number/code). This subsystem will generate the appropriate error/warning message, which may be classified into load-time, decoding-time, memory-reference, execution-time, etc. An error is "fatal" in that it stops execution, a warning is not fatal.

SAMPLE TEST JOB: Encryption
"assembly language format"

```
SWAP    PUSH,   HOLD    restore original value
        BR,     FIRST   shift left 4 bits
LOOP    ADD,    ONE
        POP,    INDEX
FIRST   SL
        PUSH,   INDEX
        CPL,    FOUR
        BRT,    LOOP
        POP,    INDEX
        POP,    TEMP
        RTN             return

START   RD              get value
        POP,    HOLD    save it
        PUSH,   HOLD    restore it
        CPG,    ZERO    if less than zero
        BRF,    M2      use method 2

M1      AND,    RIGHT   leave rightmost digit unmasked
        PUSH,   TABLE   get a value from table
        POP,    ANS     save the change
        CALL,   SWAP    change second digit
        PUSH,   TEMP    restore change value
        AND,    MID     leave middle digit unmasked
        ADD,    ANS     add it to the answer
        POP,    ANS     save it
        PUSH,   HOLD    restore original value
        AND,    LEFT    leave left digit unmasked
        NOT             complement
        AND,    LEFT    mask out again
        ADD,    ANS     include in answer
        POP,    ANS     save it
        PUSH,   HOLD    restore original value
        AND,    REST    leave 2 leftmost digits
        NOT             complement
        AND,    REST    mask out again
        ADD,    ANS     include in answer
        BR,     WRITE

M2      CALL,   SWAP
        PUSH,   TEMP
WRITE   PUSH,   HOLD
        WR              original value
        WR              encrypted value
        HLT

ZERO    0000
ONE     0001
FOUR    0004
ANS     0000
HOLD    0000
INDEX   0001
TEMP    0000
RIGHT   000F
MID     00F0
LEFT    0F00
REST    3000
TABLE   000A
        0003
        0006
        0008
        000B
        0002
        000D
        000F
        0001
        0004
        0005
        000E
        000C
        0007
        0009
        0000
```

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

```
01 00 0B 44 1
D82DBC04942ADC2E
000AD82EB42BC002
DC2EDC2F00150013
DC2DD82DB029C424
8830D834DC2CC800
D82F8831942CDC2C
D82D883200038832
942CDC2CD82D8833
00038833942CBC26
C800D82FD82D1414
0018000000010004
0000000000010000
000F00F00F003000
000A000300060008
000B0002000D000F
000100040005000E
000C000700090000
```

NOTES:

SPECIFICATION Note: 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 architectural and operating system component functionalities and restrictions should be adhered to. 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, e.g., case statements, loops, conditions, etc. 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, turn in only some representative parts of the actual trace file that your system generates (e.g., the first page, samplings of the middle of the file, and the last page). 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 for Step1 is a file of HEX digits which is a user job in loader format. The "input" for a typical 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 being the absence of a multiprogramming memory manager and a scheduler, and the lack of a JCL in Step1).

2. User-job-generated warning messages, if necessary/applicable.

3. A message indicating the nature of termination (normal or abnormal, plus a descriptive error message and memory dump if abnormal).

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

- Your Step1 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 including overflow, suspected infinite loop, invalid loader format character, invalid op code, attempt to divide by zero, address out of range, bad trace flag, etc., 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 Step1 simulation that must include the following items.

  - Your general approach to the problem, i.e., 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 mean that a flow chart, etc. of your design is required for Step1; if you 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 most a one-line answer.

  - A list of the utilities used (e.g., makefile, various debuggers, 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/object 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 and 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) Keep a copy of the deliverables that you submit for your own reference.