Name: SOLUTIONS
Closed book
You may use a calculator and one 8.5" x 11" double-sided notecard during the exam
Scratch paper will be provided for your answers

CS271 Midterm Examination

1. (1 pt) In the Intel x86 architecture, the register that always points to the next instruction to be executed is the:
   A. ESP
   B. EBP
   C. EAX
   D. **EIP**
   E. EDI

2. (2 pts) Please sort the following types of memory according to their relative access speeds. (Place the slowest type of memory at the left and the fastest type at the right)
   A. Main memory
   B. Cache
   C. Registers
   D. Optical Disk

   __D__ __A__ __B__ __C__

3. (4 pts) In a few sentences please explain the difference between the CALL and JMP x86 instructions (specifically their effect on the system stack).

   The CALL instruction pushes the address of the next instruction onto the stack and then loads the address of the called procedure into EIP. The JMP instruction has no effect on the system stack.

4. (11 pts) Given the binary number 101111011101101 please show the corresponding <u>odd-parity</u> Hamming code.
   Note that the parity bits are always at positions that are powers of 2. i.e. Bit positions 1, 2, 4, 8, 16, etc. Since we have 15 data bits, we need at least $log_2(15) + 1$ parity bits. Since $log_2(15)$ is not an integer, we need to round up to the next integer (4). Therefore we need 4+1=5 parity bits. Also note that we want odd parity.

```
P P D P D D D P D D D D D D D P D D D D
_ _ 1 _ 0 1 1 _ 1 1 0 1 1 1 0 _ 1 1 0 1

Bits that are covered by the first parity bit:
_ _ 1 _ 0 1 1 _ 1 1 0 1 1 1 0 _ 1 1 0 1
--> 101101010 --> Therefore parity bit #1 must be 0
```

```
Bits that are covered by the second parity bit:
_ _ 1 _ 0 1 1 _ 1 1 0 1 1 1 0 _ 1 1 0 1
--> 111101010 --> Therefore parity bit #2 must be 1

Bits that are covered by the third parity bit:
_ _ 1 _ 0 1 1 _ 1 1 0 1 1 1 0 _ 1 1 0 1
--> 01111101 --> Therefore parity bit #3 must be 1

Bits that are covered by the fourth parity bit:
_ _ 1 _ 0 1 1 _ 1 1 0 1 1 1 0 _ 1 1 0 1
--> 1101110 --> Therefore parity bit #4 must be 0

Bits that are covered by the fifth parity bit:
_ _ 1 _ 0 1 1 _ 1 1 0 1 1 1 0 _ 1 1 0 1
--> 1101 --> Therefore parity bit #5 must be 0

Therefore the final code is:
0 1 1 1 0 1 1 0 1 1 0 1 1 1 0 0 1 1 0 1
```

5. (2 pts) If you wish to use encode 128 bits of data, what is the minimum number of total bits that you can transmit and still ensure that you transmitted a valid Hamming code? (i.e. the total number of bits to be transmitted = number of data bits + number of parity bits)
   A. 130 bits
   B. 132 bits
   C. 136 bits
   D. 138 bits
   E. 156 bits

Since we have 128 data bits, we need at least $log_2(128) + 1$ parity bits. Therefore we need 7+1=8 parity bits. 8 parity bits + 128 data bits = 136 total bits.

6. (10 pts) Using the Intel IA32 instruction set please write a program that accepts an **unsigned** binary integer in the AX register and pushes the decimal ASCII equivalent value onto the system stack. For example, suppose that the AX register initially contains: 0b1001011001101011 (decimal value 38507). After your program has fully executed, the system stack should contain the following numbers:

55 (the ASCII representation of 7)
48 (the ASCII representation of 0)
53 (the ASCII representation of 5)
56 (the ASCII representation of 8)
51 (the ASCII representation of 3)

Note that the number at the bottom of the stack represents the 10,000's place, the number above it represents the 1,000's place, the number above it represents the 100's place, the number above it represents the 10's place and the top number on the stack represents the 1's place.

Hints: 1) As a reminder, the instruction **DIV BX** results in the following behavior: AX <- DX:AX / BX and DX <- remainder

2) Since the AX register can only hold 2 bytes, its maximum unsigned value is 65,535. Therefore, one way of approaching this assignment is to start by dividing AX by 10000, pushing the ASCII representation of the quotient onto the stack and then dividing the remainder by 1000. You can use this general idea to solve for the 10000's, 1000's, 100's, 10's and 1's places.

There are many ways to write this code but the example on the next page illustrates one approach.

```
; AX already contains the unsigned integer
MOV BX, 10000 ; I'll use BX to store my divisor
MOV CL, 10 ; We are going to keep dividing the divisor by 10
           ; First the divisor will be 10000, then it will be 1000,
           ; then 100, then 10

loopAgain:
   MOV DX, 0
   ; The form of the DIV instruction that I'm uses divides DX:AX by
   ; 16 bit number and places the quotient in AX and the
   ; remainder in DX.
   DIV BX ; The quotient is in AX and the remainder is in DX
   ADD AX, 48 ; Convert the quotient into ASCII (i.e. if the quotient
              ; was 0, the number is now 48, which is ASCII for 0)
   PUSH AX ; Push the ASCII value onto the stack

   ; If the remainder was 0 then we can quit
   CMP DX, 0
   JE finished

   ; Now we need to calculate the next divisor by dividing the existing
   ; divisor by 10
   MOV AX, BX ; Copy the current divisor into AX
   DIV CL  ; Divide AX (the current divisor) by 10
           ; AL now contains the next divisor
   MOVZX BX, AL ; Move AL into BX and fill the top byte with zeros

   ; Now move the dividend into AX so we can divide it
   MOV AX, DX ; (DX has the remainder from the first division)

JMP loopAgain ; Repeat the loop
```

(20 pts) The assembly code on the following page (a slightly modified version of the first programming assignment) was written to accept an unsigned integer from the user and print the corresponding prime factorization. For example, inputting the number "99" will print: "3 * 3 * 11"

Notes: 1) Assume that the primeNums array is located at memory address 0x100.

2) You may assume that the WriteString and WriteDec procedures do not change the contents of any registers.

A. Suppose that the user enters the integer **12.**
   1. After the code has completed execution, what are the contents of the eax, ebx, ecx, and edx registers? (please provide the **<u>decimal values</u>** contained in each register)

      EAX = 3 (the last prime number that EBX pointed to)

      EBX = 257 (the memory location of the last prime number that was used while factoring)

      ECX = 3 (a copy of EAX)

      EDX = 281 (the memory location of the newLine array)
   2. What text is printed to the screen?

      "2 * 2 * 3 "

B. Suppose that the user enters the integer **1.**
   1. After the code has completed execution, what are the contents of the eax, ebx, ecx, and edx registers? (please provide the **<u>decimal values</u>** contained in each register)

      EAX = 1 (the number than the user initially provided)

      EBX = 256 (the memory location of the first prime number in the array)

      ECX = 0 (the code never changes ECX after it is initialized to 0)

      EDX = 281 (the memory location of the newLine array)
   2. What text is printed to the screen?

      Nothing (just a line break and carriage return)

```
TITLE Midterm Examination Code          (main.asm)

INCLUDE Irvine32.inc

.data
greeting BYTE "Please provide an integer between 2 and 100:",0dh,0ah,0
warning BYTE "The input you provided was not a valid integer between 2 and 100.",0dh,0ah,0
headerPart1 BYTE "Prime factorization of ",0
headerPart2 BYTE ":",0dh,0ah,0
primeNums BYTE 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97
newLine BYTE 0dh,0ah,0
case1 BYTE " ",0
case2 BYTE " * ",0
num BYTE ?

.code
main PROC
       MOV eax, 0
       MOV ebx, 0
       MOV ecx, 0
       MOV edx, 0
       CALL Clrscr
promptUser:
       CALL ReadDec ; Get an unsigned decimal integer from the user and place it into EAX

       MOV num, al
       MOV ebx, OFFSET primeNums ; Load the address of the prime number array

topOfLoop:
       CMP num, 1
       JE finished ; Jump out of the loop if num == 1
       MOV ah, 0
       MOV al, num
       DIV BYTE PTR[ebx] ; Divide AX by the number that ebx is pointing to
       ; The quotient is now stored in AL and the remainder is in AH
       MOV cl, BYTE PTR[ebx]
       CMP AH, 0
       JNE remNotZero
              MOV num, AL
              CMP num, 1
              JNE qNot1
              MOV eax, 0
              MOV al, cl
              CALL WriteDec
              MOV edx, OFFSET case1
              CALL WriteString
              JMP primePrinted
qNot1:
              MOV eax, 0
              MOV al, cl
              CALL writeDec
              MOV edx, OFFSET case2
              CALL WriteString
primePrinted:
       JMP topOfLoop
remNotZero:
              ; Point to the next prime number and repeat the loop
              ADD ebx, TYPE BYTE ; Increment the pointer
       JMP topOfLoop
finished:
       MOV edx, OFFSET newLine
       CALL WriteString

       EXIT

main ENDP

END main
```