BERKE ÖZDEMİR / 2016400246

İBRAHİM ÖZGÜRCAN ÖZTAŞ / 2016400198

CMPE 230 PROJECT 1

March 20, 2019

Project Description

In our project, we are requested to create a hexadecimal calculator without graphical user interface. My partner and I divide this problem into several separate sections to analyze and execute with efficiency.

Milestones

1. Reading a character.

   At the beginning, we create a loop of character reading that allows us to read and give the meaning of that character.

```
READCHARWISE:   ; read input char at a time.
mov ah, 01h     ; reading input section
int 21h         ; reading input section
mov bl, al      ; input is in al, we moved it to bl.
mov ax, 0h      ; we clear ax register to be used several times later.
cmp bl, 0Dh     ; compare the value of the current character.
je PRINT        ; if carriage return(enter key in keyboard), jump to print section.
cmp bl, 2Bh     ; compare the value of the current character.
je ADDITION     ; if plus sign, jump to addition section.
cmp bl, 2Ah     ; compare the value of the current character.
je MULTIPLY     ; if asterisk, jump to multiplication section.
cmp bl, 2Fh     ; compare the value of the current character.
je DIVISION     ; if slash, jump to division section.
cmp bl, 5Eh     ; compare the value of the current character.
je BITWISEXOR   ; if caret, jump to bitwise xor section.
cmp bl, 26h     ; compare the value of the current character.
je BITWISEAND   ; if ampersand, jump to bitwise and section.
cmp bl, 7Ch     ; compare the value of the current character.
je BITWISEOR    ; if pipe(|), jump to bitwise or section.
cmp bl, 20h     ; compare the value of the current character.
je HEXASPACE    ; if blank, jump to blank section.
cmp bl, 40h     ; compare the value of the current character.
jb HEXACONNUM   ; if digits(0-9), jump to number section.
ja HEXACONLETTER; if letters(A-F), jump to letter section.
```

2. Executing the operation.

We split our code into divisions that executes each operation and results a number.

⇒ Addition Operation:

```
ADDITION:       ; addition operation of our assembly program.
pop ax          ; we pop the trivial 0h from stack,
                ; which is a convention in our program structure.
pop ax          ; we pop the first operand of addition process.
pop bx          ; we pop the second operand of addition process.
add ax, bx      ; we add them up and store it on ax register.
push ax         ; we push the stored value to reuse later.
jmp CONTPOINT   ; end of addition operation.
```

⇒ Multiplication Operation:

```
MULTIPLY:       ; multiplication operation of our assembly program.
pop ax          ; we pop the trivial 0h from stack,
                ; which is a convention in our program structure.
pop ax          ; we pop the first operand of multiplication process.
pop bx          ; we pop the second operand of multiplication process.
mul bx          ; we multiply the two numbers in hexadecimal form
                ; and store it in ax register.
push ax         ; we push the stored value to reuse later.
jmp CONTPOINT   ; end of multiply operation.
```

⇒ Division Operation:

```
DIVISION:       ; division operation of our assembly program.
mov dx, 0h      ; to avoid fatal errors, we clear dx register via 0h.
pop bx          ; we pop the trivial 0h in out stack.
pop bx          ; we pop the first operand of division process.
pop ax          ; we pop the second operand of division process.
div bx          ; we divide the two numbers, ax is dividend and bx is divider,
                ; and store the result in ax register.
push ax         ; we push the stored value to reuse later.
jmp CONTPOINT   ; end of division operation.
```

⇒ **Bitwise AND Operation:**

```
BITWISEAND:     ; bitwiseand operation of our assembly program.
pop ax          ; we pop the trivial 0h from stack,
                ; which is a convention in our program structure.
pop ax          ; we pop the first operand of the bitwiseand process.
pop bx          ; we pop the second operand of the bitwiseand process.
and ax, bx      ; we take the bitwise and of the two numbers,
                ; the result is stored in ax register.
push ax         ; we push the stored value to stack to reuse later.
jmp CONTPOINT   ; end of bitwise and operation.
```

⇒ **Bitwise OR Operation:**

```
BITWISEOR:      ; bitwiseor operation of our assembly program
pop ax          ; we pop the trivial 0h from stack,
                ; which is a convention in our program structure.
pop ax          ; we pop the first operand of the bitwiseor process.
pop bx          ; we pop the second operand of the bitwiseor process.
or ax, bx       ; we take the bitwise or of the two numbers,
                ; the result is stored in ax register.
push ax         ; we push the stored value to stack to reuse later.
jmp CONTPOINT   ; end of bitwise or operation.
```

⇒ **Bitwise XOR Operation:**

```
BITWISEXOR:     ; bitwiseor operation of our assembly program
pop ax          ; we pop the trivial 0h from stack,
                ; which is a convention in our program structure.
pop ax          ; we pop the first operand of the bitwise xor process.
pop bx          ; we pop the second operand of the bitwise xor process.
xor ax, bx      ; we take the bitwise xor of the two numbers,
                ; the result is stored in ax register.
push ax         ; we push the stored value to stack to reuse later.
jmp CONTPOINT   ; end of bitwise xor operation.
```

3. Converting Characters into Hexadecimal Format.

We read characters that holds numerical value for hexadecimal operations, then calculate their real values by operating on their ASCII values.

⇒ **Digit Converting:**

```
HEXACONNUM:      ; to convert a number character to its hexadecimal value,
                 ; we use this section.
sub bl, 30h      ; 30h is the ascii value of 0, so we get the real
                 ; value of the digit character.
jmp HEXACON      ; end of number conversion.

HEXACON:         ; we use this section to calculate base
                 ; arithmetics of the current number.
pop ax           ; popping the previous part of number
                 ; (to avoid fatal errors, we push 0h initially to stack,
                 ; thus without changing number value, we make a
                 ; recursive way to create the number itself.)
mov cx, 10h      ; number needed to shift left.
mul cx           ; we multiply the number with 10h, to shift left 1 digit.
add ax, bx       ; we add the last significant digit and store it in ax register.
push ax          ; we push the stored value to stack to reuse later.
mov ax, 0h       ; we clear ax register.
jmp CONTPOINT    ; end of conversion.
```

⇒ **Letter Converting:**

```
HEXACONLETTER:  ; to convert a number greater than 9, we represent them
                ; via letters A-F. We use this section to convert it.
sub bl, 37h     ; we know that A represents 10d in hexadecimal,
                ; so we get the exact difference between ascii
                ; character of A and 10d, which is 37h.
jmp HEXACON     ; end of letter conversion.

HEXACON:        ; we use this section to calculate base
                ; arithmetics of the current number.
pop ax          ; popping the previous part of number
                ; (to avoid fatal errors, we push 0h initially to stack,
                ; thus without changing number value, we make a
                ; recursive way to create the number itself.)
mov cx, 10h     ; number needed to shift left.
mul cx          ; we multiply the number with 10h, to shift left 1 digit.
add ax, bx      ; we add the last significant digit and store it in ax register.
push ax         ; we push the stored value to stack to reuse later.
mov ax, 0h      ; we clear ax register.
jmp CONTPOINT   ; end of conversion.
```

4. Blank character:

We use blank character as a splitter of informational characters. And we create a dummy 0h number that comes when blank character is read. That allows us to convert characteristic number into real value of that number easily.

```
HEXASPACE:      ; we use this to add trivial 0h to stack and mark the blank character.
mov cx, 0h      ; create 0h.
push cx         ; pushing it in stack.
jmp CONTPOINT   ; end of blank section.
```

5. Splitting the result into characters.

We split the result into individual characters, which we need to print onto screen character by character.

```
PRINT:          ; we use this section to print each character in output.
pop ax          ; we pop the result of the operation.
mov bx, 7Eh     ; we use 7Eh as a sentinel value to exit our program.
push bx         ; we push it in stack.
push ax         ; we push the result in stack again, to see the
                ; sentinel value while processing print part.
mov ax, 0h      ; clearing register ax.
mov bx, 0h      ; clearing register bx.
mov cx, 0h      ; clearing register cx.
mov dx, 0h      ; clearing register dx.
jmp PRINTLOOP   ; end of print preprocessing.


PRINTLOOP:
pop ax          ; pop the result to ax.
cmp ax, 10h     ; check if the result is less than 10h.
jb CONTPOINT2   ; jump to contpoint2 if it is.
mov bx, 10h     ; move 10h to bx to divide by it.
div bx          ; divide the current number by 10h.
push dx         ; push the remainder to stack
                ; (least significant digit of the previous number).
push ax         ; push the result back to stack.
mov dx, 0h      ; clear the register dx.
mov ax, 0h      ; clear the register ax.
jmp PRINTLOOP   ; back to the start of the loop.
```

6. Printing onto Screen

Then, we pop characters from stack and print onto screen until we reach our sentinel value, character at each time. If the character is number, we we add 30h to get its ASCII value. If letter then, we add 37h to apply the same idea.

⇒ Digit Printing:

```
PRINTSCREEN:    ; process the digits of the result in the stack.
pop ax          ; pop the least significant digit to ax.
cmp ax, 7Eh     ; check if it is the terminating character, which should be in the bottom.
je ENDPROGRAM   ; jump to endprogram if it is.
cmp ax, 9h      ; check if the digit is lesss than 10d.
ja PRINTLETTER  ; go to printletter if it is above 10d.
jbe PRINTNUM    ; go to printnum if it is less than 10d.


PRINTNUM:       ; process to print numbers.
add ax, 30h     ; if it is a number add 30h to change it to its ASCII value.
mov dx, ax      ; move to dx for printing.
mov ah, 02h     ; printing preprocess.
int 21h         ; interrupt.
jmp PRINTSCREEN ; back to the start of printscreen.
```

⇒ Letter Printing:

```
PRINTSCREEN:    ; process the digits of the result in the stack.
pop ax          ; pop the least significant digit to ax.
cmp ax, 7Eh     ; check if it is the terminating character, which should be in the bottom.
je ENDPROGRAM   ; jump to endprogram if it is.
cmp ax, 9h      ; check if the digit is lesss than 10d.
ja PRINTLETTER  ; go to printletter if it is above 10d.
jbe PRINTNUM    ; go to printnum if it is less than 10d.


PRINTLETTER:    ; process to print letters.
add ax, 37h     ; if it is a letter add 37h to change it to its ASCII value.
mov dx, ax      ; move to dx for printing.
mov ah, 02h     ; printing preprocess.
int 21h         ; interrupt.
jmp PRINTSCREEN ; back to the start of printscreen.
```

7. Ending program

We send interrupt operation to finish our program.

```
ENDPROGRAM:     ; termination process.
int 20h         ; terminate the program.
```