

CS2651 Computer Architecture (Fall 2022)

Lab 2: Assembly Calculator

Xiangyuan Xue (521030910387)

1 Basic Structure

Main procedure of the calculator is preferentially implemented. In this procedure, global variables such as stack pointer are initialized, and an iterative structure is started, which keeps reading user commands and decoding operations. Ten commands are successively examined and jump to corresponding subroutine accordingly.

1. “X”: exit program
2. “C”: clear value stack
3. “D”: display top value
4. “+”: operation plus
5. “*”: operation multiply
6. “-”: operation negate
7. “/”: operation DIV
8. “%”: operation MOD
9. “@”: operation XOR
10. value: insert integer value

2 Native Command

For each command, a subroutine is written to accomplish the procedure, which makes the program clean and easy to read. Note that exit, clear, display, plus, multiply, negate and insert are supported by LC-3 calculator in the textbook. We don't make modification in their algorithm. We only make a little optimization to ensure that they serve well in our calculator and add a few prompts to make our calculator more friendly when interacting with users.

3 Calculation Stack

All the operations fetch operands on the calculation stack. So a global calculation stack is required. The address of stack base and maximum capacity are also global variables. However, an annoying problem occurs. When we use LEA instruction to visit stack space, assembler yields an error, because our program is too long and 9-bit offset cannot express the target address. To solve this problem, we restate a label which points to the address of calculation stack every 300 lines. When we need the address, just use LD to load the closest label.

```

; program stack and buffer
StackLimit .BLKW      #15
StackBase  .FILL      x0000          ; reserve stack space
Buffer     .BLKW      #4
           .FILL      X0000
           .END

;
; relaying label
BP1        .FILL      Buffer
SH1        .FILL      StackLimit
SL1        .FILL      StackBase
;
; effective visit
           LD         R0,BP1
           LD         R1,SH3
           LD         R6,SL1

```

4 Extended Operation

4.1 MOD Operation

MOD operation is dual with DIV operation, we might as well consider the implementation of DIV operation first. Both dividend and divisor can be either positive or negative. To simplify the calculation, use following equality:

$$x \text{ div } y = (-x) \text{ div } (-y)$$

We can describe DIV algorithm as follow:

1. Fetch dividend x and divisor y as native binary operations do.
2. If divisor $y = 0$, yield a divide-by-zero error.
3. If divisor $y < 0$, negate both dividend x and divisor y .
4. If dividend $x = 0$, the result is 0.

5. If dividend $x > 0$, do iterative subtraction y from x until y is going to be negative.
6. If dividend $x < 0$, do iterative addition y to x until y is non-negative.
7. The result is the time that subtraction or addition happens.

Note that MOD operation is essentially same with DIV operation. To implement that, the result is just the remaining dividend x . Kernel codes are shown below.

```

; operands are in R1,R2 and suppose divisor is positive
MODStart    AND        R0,R0,#0
            ADD        R1,R1,#0
            BRp        MODPlus        ; dividend is positive
            BRn        MODMinus       ; dividend is negative
            BRnzp      MODPush        ; dividend is zero
MODPlus     NOT        R2,R2
            ADD        R2,R2,#1        ; flip divisor
MODLoop     ADD        R0,R1,R2
            BRn        MODPush
            ADD        R1,R0,#0
            BRnzp      MODLoop        ; test by loop
MODMinus    ADD        R1,R1,R2
            BRzp       MODPush
            BRnzp      MODMinus       ; test by loop
MODPush     ADD        R0,R1,#0
            JSR        PUSH
            BRnzp      MODFinish      ; push result to stack

```

4.2 XOR Operation

Note that XOR is a logical operation with a binary base. We know AND and NOT consists a complete logical system, so XOR can be implemented by following deduction:

$$\begin{aligned}
 x \text{ xor } y &= (x \wedge \neg y) \vee (\neg x \wedge y) \\
 &= \neg(\neg(x \wedge \neg y) \wedge \neg(\neg x \wedge y)) \\
 &= \text{not } \{[\text{not}(x \text{ and } (\text{not } y))] \text{ and } [\text{not}((\text{not } x) \text{ and } y)]\}
 \end{aligned}$$

Therefore, kernel codes that implement XOR operation are extremely simple.

```

; operands are already loaded in R0,R1
            NOT        R2,R0
            AND        R2,R2,R1
            NOT        R2,R2
            NOT        R5,R1
            AND        R5,R5,R0

```

	NOT	R5,R5	
	AND	R0,R2,R5	
	NOT	R0,R0	; exclude operation
XORPush	JSR	PUSH	
	BRnzp	XORFinish	

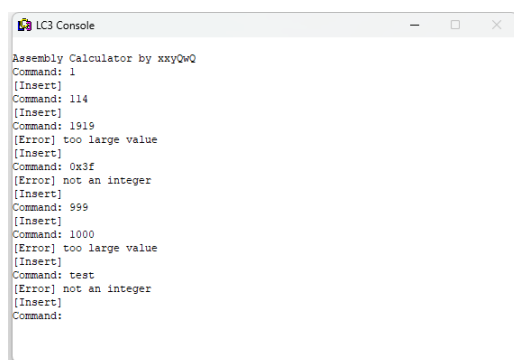
5 Runtime Test

5.1 Insert Value and Display Value

Three types of test input:

- Legal value between 0 and 999.
- Illegal value larger than 999.
- Illegal input containing other characters.

Display top value by command “D”.

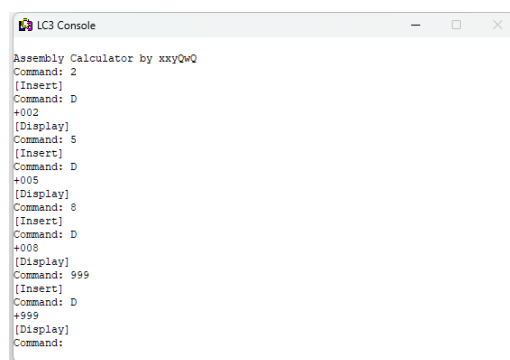


```

Assembly Calculator by xxyQwQ
Command: 1
[Insert]
Command: 114
[Insert]
Command: 1919
[Error] too large value
[Insert]
Command: 0x3f
[Error] not an integer
[Insert]
Command: 999
[Insert]
Command: 1000
[Error] too large value
[Insert]
Command: test
[Error] not an integer
[Insert]
Command:

```

(a) Insert Value



```

Assembly Calculator by xxyQwQ
Command: 2
[Insert]
Command: D
+002
[Display]
Command: 5
[Insert]
Command: D
+005
[Display]
Command: 8
[Insert]
Command: D
+008
[Display]
Command: 999
[Insert]
Command: D
+999
[Display]
Command:

```

(b) Display Value

Figure 1: Insert Value and Display Value

5.2 Clear Stack and Exit Program

Clear stack by command “C”.

Exit program by command “X”.

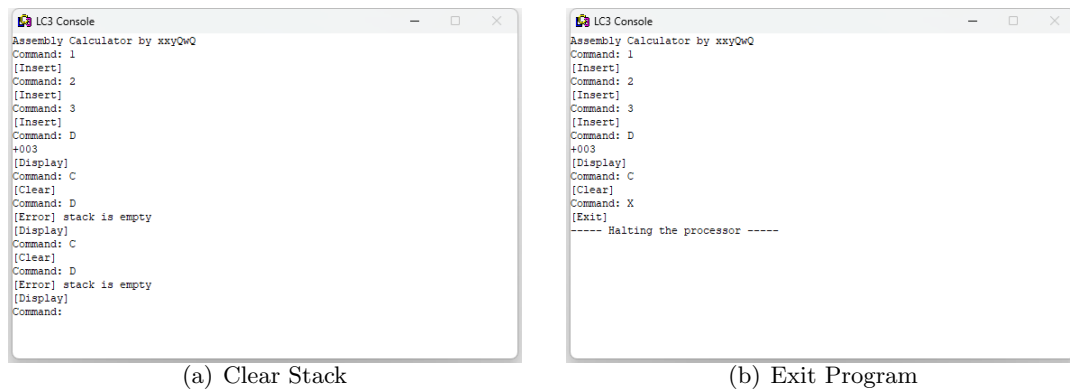


Figure 2: Clear Stack and Exit Program

5.3 Negate Operation

Two types of test case:

- Legal case for existed operand.
- Illegal case with no operand.

Negate value by command “-”.

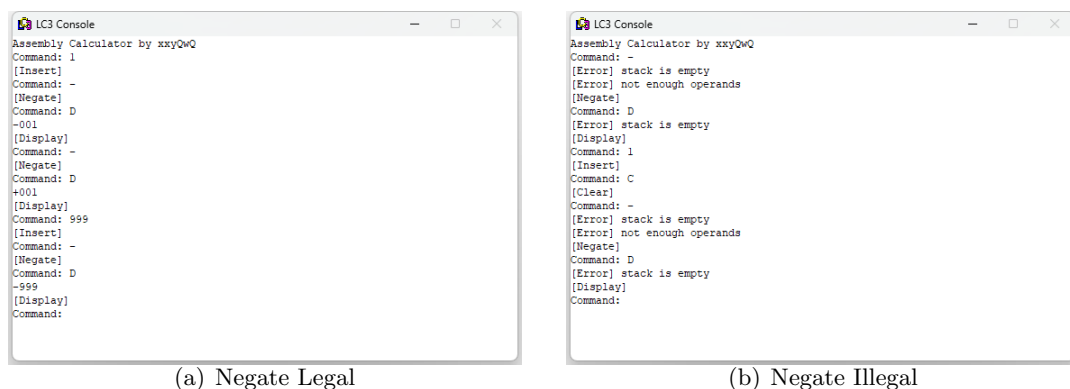


Figure 3: Negate Operation

5.4 Plus Operation

Three types of test case:

- Legal case for two operands.
- Illegal case for zero or one operand.
- Illegal case for range exceeding.

Plus values by command “+”.

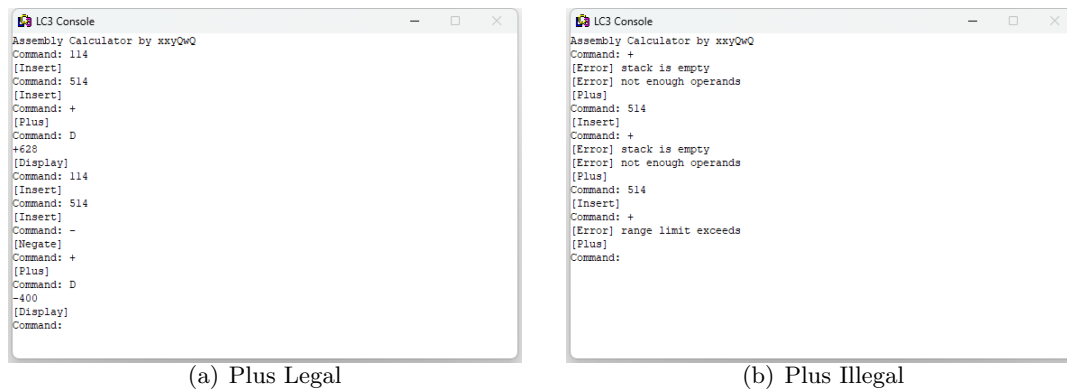


Figure 4: Plus Operation

5.5 Multiply Operation

Three types of test case:

- Legal case for two operands.
- Illegal case for zero or one operand.
- Illegal case for range exceeding.

Multiply values by command “*”.

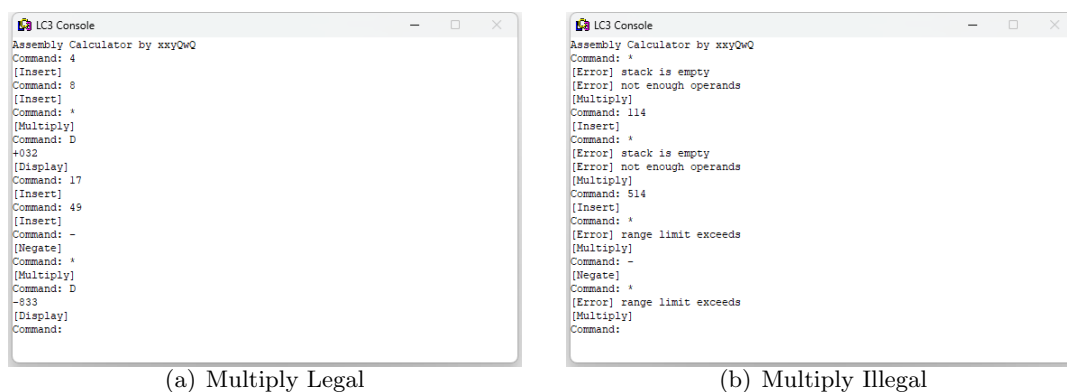


Figure 5: Multiply Operation

5.6 MOD Operation

Three types of test case:

- Legal case for two operands.
- Illegal case for zero or one operand.
- Illegal case for dividing zero.

MOD values by command “%”.

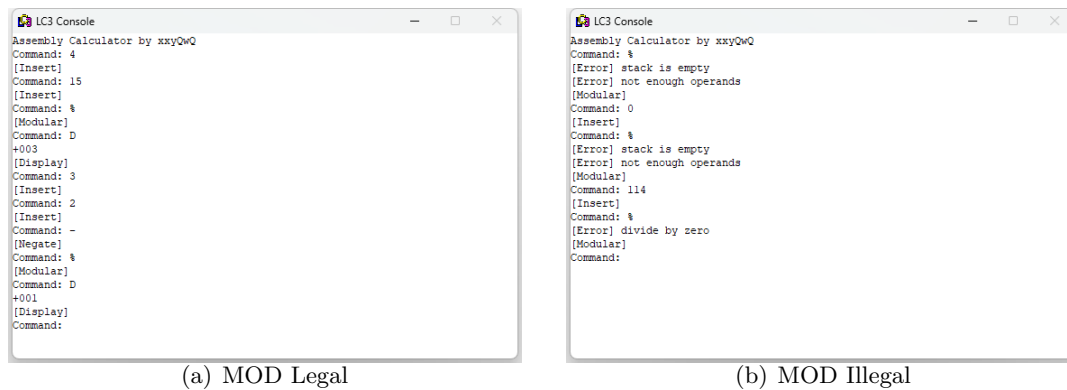


Figure 6: MOD Operation

5.7 XOR Operation

Two types of test case:

- Legal case for two operands.
- Illegal case for zero or one operand.

XOR values by command “@”.

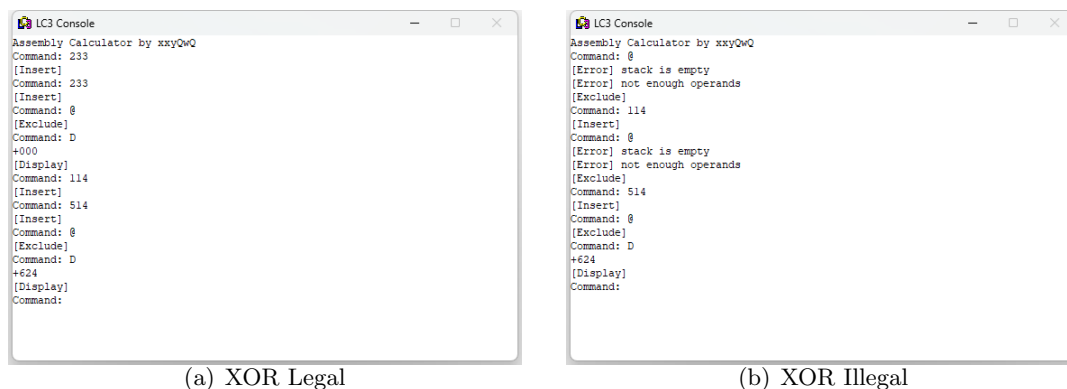


Figure 7: XOR Operation

References

- [1] Yale N. Patt and Sanjay J. Patel. Introduction to Computing Systems. New York: McGraw-Hill, 2020.