Reverse Engineering

Assembly Review

Instructor:	G Leaden	**Due:**	See Syllabus (Friday)
Email:	g.leaden1@marist.edu	**Place:**	Hancock 2023

Goals:

Understanding assembly instructions is essential to reversing a compiled program. This worksheet should be filled out, then used as a reference in the future.

Instructions:

Explain what each instruction does, and give an example of C/C++ code that would compile into that instruction.

1. MOV
2. POP
3. PUSH
4. LEA
5. ADD
6. SUB
7. JNE
8. JLE
9. JMP
10. CMP
11. CALL, RET
12. IMUL, MUL
13. NOP
14. AND
15. OR

Submitting:

Print out the answers, and hand it in on the due date.

Grading Rubric:

Each Problem 6.66666666667%

ANSWERS

1. MOV
   Move.
   Copies data from one location to another.

```
void f(){
    int a;
    a=1; // Here
}
```

2. POP
   Pop. Pop data from stack.

```
void f(){
    int a;
    a=1;
    int b = a + 1; // Here
}
```

3. PUSH
   Push. Push data onto the stack.

```
void f(){ // Here
    int a;
    a=1;
    int b = a + 1;
}
```

4. LEA
   Load Effective Address. Calculate the address of an array element by adding the array address, element index, with multiplication of element size.

```
int f(int a, int b){
    return a*8+b; \\ Here
};
```

5. ADD
   Add two values.

```
void f(){
    int a;
    a=1;
    int b = a + 1; // Here
}
```

6. SUB
   Subtract two values.

```
void f(){
    int a;
    a=1;
    int b = a - 1; // Here
}
```

7. JNE
   Jump Not Equal.
   Jump to address when the result of a CMP is not equal to 0

```
int f(int a){
    if (a == 8){ // Here
        return 1;
    }
}
```

8. JLE
   Jump Less Equal.
   Jump to address when the result of a CMP is lesser or equal

```
int f(int a){
    if (a > 8){ // Here
        return 1;
    }
}
```

9. JMP
   Jump.
   Jump to an address.

```
int f(int a){
    if (a > 8){
        goto asdf; // Here
    }
    int b = a;
    asdf: if (a == 3){
            return 3;
        }
    return 0;
}
```

10. CMP
    Compare. Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results.

```
int f(int a){
    if (a > 8){ // Here
        return 1;
    }
}
```

11. CALL, RET
    Call and Return.
    The call instruction first pushes the current code location onto the hardware supported stack in memory and then performs an unconditional jump to the code location indicated by the label operand. Unlike the simple jump instructions, the call instruction saves the location to return to when the subroutine completes.
    The ret instruction implements a subroutine return mechanism. This instruction first pops a code location off the hardware supported in-memory stack. It then performs an unconditional jump to the retrieved code location.

```
int f(int a){
    if (a > 8){
        return 1; // And Here
    }
}

void main(){
    int value = f(9); // Here
}
```

12. IMUL – Know all forms. Only need to provide one example.
    Signed multiply. Performs a signed multiplication of two operands. This instruction has three forms, depending on the number of operands.

    One-operand form. This form is identical to that used by the MUL instruction. Here, the source operand (in a general-purpose register or memory location) is multiplied by the value in the AL, AX, or EAX register (depending on the operand size) and the product is stored in the AX, DX:AX, or EDX:EAX registers, respectively.

    Two-operand form. With this form the destination operand (the first operand) is multiplied by the source operand (second operand). The product is then stored in the destination operand location (a register)

    Three-operand form. This form requires a destination operand (the first operand) and two source operands (the second and the third operands). Multiply second and third operands togehter, storing in the first operand register.

```
int f(int a){
    if (a > 8){
        return a*90; // Here
    }
}
```

13. NOP
    No Operation.
    Performs no operation. This instruction is a one-byte instruction that takes up space in the instruction stream but does not affect the machine context, except the EIP register.

```
void f(){
}
```

14. AND
    Bitwise AND.
    These instructions perform a logical bitwise and on its operands, placing the result in the first operand location.

```
int f(int a){
    if (a > 8){
        return a&90; // Here
    }
}
```

15. OR
    Bitwise OR.
    These instructions perform a logical bitwise or on its operands, placing the result in the first operand location.

```
int f(int a){
    if (a > 8){
        return a|90; // Here
    }
}
```