Name: ______________________________

1. (5 points) For the following, Check T if the statement is true, or F if the statement is false.
   (a) ☐ T ☐ F : The Gnu Debugger (gdb) works at the C level. If you did not compile with the -g option, the Gnu Debugger is virtually useless.

   (b) ☐ T ☐ F : If the zero flag (ZF) is true, then the instruction "jne LINE3" will branch to LINE3.

   (c) ☐ T ☐ F : In X86-64 assembler, the memory referenced by -0x1C(%rbp) is the same as the memory referenced by -0x1C(%rbp,%rax,16) if the %rax register contains a zero.

   (d) ☐ T ☐ F : The x86/64 "push" instruction always decreases the value in the %rsp register. The "pop" isntruction always increases the value in the %rsp register.

   (e) ☐ T ☐ F : After executing a conditional jump instruction, the %rip register always points at the instruction after the jump instruction.

   (f) ☐ T ☐ F : In the X86-64 ISA instruction processing cycle, there are four phases of the cycle which potentially read or write from memory: the "fetch instruction" phase, the "evaluate address" phase, the "fetch operands" phase, and the "store results" phase.

   (g) ☐ T ☐ F : It is important that an older version of an ISA support everyting in a newer version of the same ISA so that new software can be run on both old and new hardware.

   (h) ☐ T ☐ F : The concept of dividing registers into caller-saved registers and callee-saved registers makes the resulting code much more efficient because we never have to save and restore registers that don't need to be saved and restored, and most of the time, no registers need to be saved or restored.

   (i) ☐ T ☐ F : The convention of using both %rbp and %rsp to delineate the current stack frame is not absolutely required. We only need to know where the stack frame starts or stops since we know the stack frame's length. In fact, there is a gcc compiler option to use only the %rsp register for stack frames instead of both %rsp and %rbp.

   (j) ☐ T ☐ F : The gcc compiler may decide to translate a switch statement using a jump table, or it may decide that a jump table is not very efficient because there is no easy way to translate cases into table indexes, or if there is, the resulting table will be very sparsely populated.

Answer the following by checking all correct answers.

2. (5 points) Which x86 assembler code was generated from the pgmA.c in Listing 1 on page 5?
   ☐ Listing 5 on page 6 ☐ Listing 6 on page 6 ☐ Listing 7 on page 6 ☐ Listing 8 on page 6
   ☐ None of the above

3. (5 points) Which x86 assembler code was generated from the pgmB.c in Listing 2 on page 5?
   ☐ Listing 5 on page 6 ☐ Listing 6 on page 6 ☐ Listing 7 on page 6 ☐ Listing 8 on page 6
   ☐ None of the above

4. (5 points) Which x86 assembler code was generated from the pgmC.c in Listing 3 on page 5?
☐ Listing 5 on page 6 ☐ Listing 6 on page 6 ☐ Listing 7 on page 6 ☐ Listing 8 on page 6
☐ None of the above

5. (5 points) Which x86 assembler code was generated from the pgmD.c in Listing 4 on page 5?
☐ Listing 5 on page 6 ☐ Listing 6 on page 6 ☐ Listing 7 on page 6 ☐ Listing 8 on page 6
☐ None of the above

6. (5 points) The for loop instruction on line 6 in Listing 9 on page 7 generates x86 instructions at which of the following addresses in Listing 10 on page 7?
☐ 676 ☐ 67d ☐ 6ad ☐ 6b1, 6b4, 6b7 ☐ All of the above

7. (5 points) Given the stack data in figure 1 on page 8, what is the current value of the i variable in main?
☐ 0 ☐ 1 ☐ 2 ☐ 0x040060d ☐ None of the above

8. (5 points) Given the stack data in figure 1 on page 8, what is the current value of the j variable?
☐ 0 ☐ 1 ☐ 2 ☐ 0x555547ad ☐ None of the above

9. (5 points) Given the stack data in figure 1 on page 8, what is the value of main's caller's %rbp?
☐ 0x7fffffffec98 ☐ 0x7fffffffe900 ☐ 0x7fffffffe9e8 ☐ 0x7ffff7a5a2b1 ☐ None of the above

10. (5 points) The x86 instructions in Listing 10 on page 7 at addresses 6cd through 6d8 are typically called:
☐ func1 on-ramp ☐ funky chicken ☐ func1 preamble ☐ func1 frame create ☐ standard entry

11. (5 points) Given the stack data in figure 1 on page 8, what function is currently executing?
☐ The Operating System ☐ main ☐ func1 ☐ func2 ☐ None of the above

12. (5 points) In the code in Listing 10 on page 7, the following instructions are used to save and restore caller saved (red) registers:
☐ 660 and 6cb ☐ 6cd and 735 ☐ 737 and 75a ☐ All of the above ☐ None of the above

Answer the following questions by filling in the blanks.

13. (5 points) The instruction at address 70d in Listing 10 on page 7 performs a jne conditional jump, and therefore, uses the zero flag. What instruction sets the zero flag, and what are the conditions that cause the zero flag to get set on.

____

14. (5 points) Using Listing 10 on page 7, can you find where main's local variable `ans` kept? Express your answer as an offset from the %rbp register.

____

15. (5 points) In Listing 10 on page 7, the compiler has chosen to implement the comparison arg1==arg2 using the instructions at locations 745 and 749. However, the previous instruction at 742 just moved the value of the %al register into -0x8(%rbp) and then the compare instruction at 749 comapres the value in %al with the value in memory at -0x8(%rbp). How can these two ever be different?

____

16. (5 points) Using Listing 10 on page 7, what is the address of the x86 instruction that implements the comparison in main's for loop, specifically that determines if $i < argc$?

____

17. (5 points) Using Listing 10 on page 7, how much extra space does func1 reserve in its stack frame for local variables, copies of parameters, and alignment padding?

____

18. (5 points) Using Listing 10 on page 7 and figure 1 on page 8, what is the current value of the argv parameter in the main function?

____

19. (5 points) In Listing 10 on page 7, func2 keeps copies of the arguments arg1 and arg2 at %rbp-4 and %rbp-8, and since %rsp is the same as %rbp, that means that arg1 and arg2 values are stored *below* the bottom of the stack. What x86-64 ISA architecture feature lets the gcc compiler do this?

____

20. (5 points) In Listing 10 on page 7, the instructions at addresses 715 and 730 in func1 put either a one or a zero in the %eax register. The X86 calling conventions we learned stated that the %rax register should contain the return code. Why did the gcc compiler put a one or a zero into %eax instead of %rax?

____

Tear-off Page

Listing 1: pgmA.c

```
int main(int argc, char **argv) {
    int i; int ans=0;
    for(i=0;i<argc; i++) {
        ans+=i;
    }
    return ans;
}
```

Listing 2: pgmB.c

```
int main(int argc, char **argv) {
    int ans=0;
    switch(argc) {
        case 1: ans=100; break;
        case 2: ans=1; break;
        case 3: ans=20; break;
        case 4: ans=40; break;
        case 5: ans=50; break;
        default: ans=-1; break;
    }
    return ans;
}
```

Listing 3: pgmC.c

```
int main(int argc, char **argv) {
    int ans=0;
    if (argc > 5) ans=-1;
    else if (argc < 1) ans=-1;
    else ans=100;
    return ans;
}
```

Listing 4: pgmD.c

```
int main(int argc, char **argv) {
    int ans=0;
    while(argc >= 5) {
        ans++;
        argc=argc - 2;
    }
    return ans;
}
```

Listing 5: pgm?.objdump.txt

```
0000000000000660 <main>:
 660: push   %rbp
 661: mov    %rsp,%rbp
 664: mov    %edi,-0x14(%rbp)
 667: mov    %rsi,-0x20(%rbp)
 66b: movl   $0x0,-0x4(%rbp)
 672: cmpl   $0x5,-0x14(%rbp)
 676: jle    681 <main+0x21>
 678: movl   $0xffffffff,-0x4(%rbp)
 67f: jmp    697 <main+0x37>
 681: cmpl   $0x0,-0x14(%rbp)
 685: jg     690 <main+0x30>
 687: movl   $0xffffffff,-0x4(%rbp)
 68e: jmp    697 <main+0x37>
 690: movl   $0x64,-0x4(%rbp)
 697: mov    -0x4(%rbp),%eax
 69a: pop    %rbp
 69b: retq
```

Listing 6: pgm?.objdump.txt

```
0000000000000660 <main>:
 660:    push    %rbp
 661:    mov     %rsp,%rbp
 664:    mov     %edi,-0x14(%rbp)
 667:    mov     %rsi,-0x20(%rbp)
 66b:    movl    $0x0,-0x4(%rbp)
 672:    cmpl    $0x5,-0x14(%rbp)
 676:    ja      6c9 <main+0x69>
 678:    mov     -0x14(%rbp),%eax
 67b:    lea     0x0(,%rax,4),%rdx
 683:    lea     0xda(%rip),%rax
 68a:    mov     (%rdx,%rax,1),%eax
 68d:    movslq  %eax,%rdx
 690:    lea     0xcd(%rip),%rax
 697:    add     %rdx,%rax
 69a:    jmpq    *%rax
 69c:    movl    $0x64,-0x4(%rbp)
 6a3:    jmp     6d1 <main+0x71>
 6a5:    movl    $0x1,-0x4(%rbp)
 6ac:    jmp     6d1 <main+0x71>
 6ae:    movl    $0x14,-0x4(%rbp)
 6b5:    jmp     6d1 <main+0x71>
 6b7:    movl    $0x28,-0x4(%rbp)
 6be:    jmp     6d1 <main+0x71>
 6c0:    movl    $0x32,-0x4(%rbp)
 6c7:    jmp     6d1 <main+0x71>
 6c9:    movl    $0xffffffff,-0x4(%rbp)
 6d1:    mov     -0x4(%rbp),%eax
 6d4:    pop     %rbp
 6d5: retq
```

Listing 7: pgm?.objdump.txt

```
0000000000000660 <main>:
 660:    push    %rbp
 661:    mov     %rsp,%rbp
 664:    mov     %edi,-0x14(%rbp)
 667:    mov     %rsi,-0x20(%rbp)
 66b:    movl    $0x0,-0x4(%rbp)
 672:    jmp     67c <main+0x1c>
 674:    addl    $0x1,-0x4(%rbp)
 678:    subl    $0x2,-0x14(%rbp)
 67c:    cmpl    $0x4,-0x14(%rbp)
 680:    jg      674 <main+0x14>
 682:    mov     -0x4(%rbp),%eax
 685:    pop     %rbp
686: retq
```

Listing 8: pgm?.objdump.txt

```
0000000000000660 <main>:
 660:    push    %rbp
 661:    mov     %rsp,%rbp
 664:    mov     %edi,-0x14(%rbp)
 667:    mov     %rsi,-0x20(%rbp)
 66b:    movl    $0x0,-0x8(%rbp)
 672:    movl    $0x0,-0x4(%rbp)
 679:    jmp     685 <main+0x25>
 67b:    mov     -0x4(%rbp),%eax
 67e:    add     %eax,-0x8(%rbp)
 681:    addl    $0x1,-0x4(%rbp)
 685:    mov     -0x4(%rbp),%eax
 688:    cmp     -0x14(%rbp),%eax
 68b:    jl      67b <main+0x1b>
 68d:    mov     -0x8(%rbp),%eax
 690:    pop     %rbp
691: retq
```

Listing 9: reflexive.c

```c
int func1(int arg1, char * arg2);
int func2(char arg1, char arg2);
int main(int argc,char **argv) {
    int i;
    int ans=0;
    for(i=1;i<argc;i++) {
        if (func1(i,argv[i])>0) { ans=i; }
    }
    if (ans>0) { return 0; }
    return 1;
}
int func1(int arg1, char *arg2) {
    int j=1;
    while(arg2[j]!=0x00) {
        if (0==func2(arg2[0],arg2[j])) { j++; }
        else { return 1; }
    }
    return 0;
}
int func2(char arg1, char arg2) {
    if (arg1==arg2) return 1;
    return 0;
}
```

Listing 10: reflexive.objdump.txt

```
0000000000000660 <main>:
 660: push   %rbp
 661: mov    %rsp,%rbp
 664: sub    $0x20,%rsp
 668: mov    %edi,-0x14(%rbp)
 66b: mov    %rsi,-0x20(%rbp)
 66f: movl   $0x0,-0x8(%rbp)
 676: movl   $0x1,-0x4(%rbp)
 67d: jmp    6b1 <main+0x51>
 67f: mov    -0x4(%rbp),%eax
 682: cltq
 684: lea    0x0(,%rax,8),%rdx
 68c: mov    -0x20(%rbp),%rax
 690: add    %rdx,%rax
 693: mov    (%rax),%rdx
 696: mov    -0x4(%rbp),%eax
 699: mov    %rdx,%rsi
 69c: mov    %eax,%edi
 69e: callq  6cd <func1>
 6a3: test   %eax,%eax
 6a5: jle    6ad <main+0x4d>
 6a7: mov    -0x4(%rbp),%eax
 6aa: mov    %eax,-0x8(%rbp)
 6ad: addl   $0x1,-0x4(%rbp)
 6b1: mov    -0x4(%rbp),%eax
 6b4: cmp    -0x14(%rbp),%eax
 6b7: jl     67f <main+0x1f>
 6b9: cmpl   $0x0,-0x8(%rbp)
 6bd: jle    6c6 <main+0x66>
 6bf: mov    $0x0,%eax
 6c4: jmp    6cb <main+0x6b>
 6c6: mov    $0x1,%eax
 6cb: leaveq
 6cc: retq

00000000000006cd <func1>:
 6cd: push   %rbp
 6ce: mov    %rsp,%rbp
 6d1: sub    $0x20,%rsp
 6d5: mov    %edi,-0x14(%rbp)
 6d8: mov    %rsi,-0x20(%rbp)
 6dc: movl   $0x1,-0x4(%rbp)
 6e3: jmp    71c <func1+0x4f>
 6e5: mov    -0x4(%rbp),%eax
 6e8: movslq %eax,%rdx
 6eb: mov    -0x20(%rbp),%rax
 6ef: add    %rdx,%rax
 6f2: movzbl (%rax),%eax
 6f5: movsbl %al,%edx
 6f8: mov    -0x20(%rbp),%rax
 6fc: movzbl (%rax),%eax
 6ff: movsbl %al,%eax
 702: mov    %edx,%esi
 704: mov    %eax,%edi
 706: callq  737 <func2>
 70b: test   %eax,%eax
 70d: jne    715 <func1+0x48>
 70f: addl   $0x1,-0x4(%rbp)
 713: jmp    71c <func1+0x4f>
 715: mov    $0x1,%eax
 71a: jmp    735 <func1+0x68>
```

```
 71c: mov    -0x4(%rbp),%eax
 71f: movslq %eax,%rdx
 722: mov    -0x20(%rbp),%rax
 726: add    %rdx,%rax
 729: movzbl (%rax),%eax
 72c: test   %al,%al
 72e: jne    6e5 <func1+0x18>
 730: mov    $0x0,%eax
 735: leaveq
 736: retq

0000000000000737 <func2>:
 737: push   %rbp
 738: mov    %rsp,%rbp
 73b: mov    %edi,%edx
 73d: mov    %esi,%eax
 73f: mov    %dl,-0x4(%rbp)
 742: mov    %al,-0x8(%rbp)
 745: movzbl -0x4(%rbp),%eax
 749: cmp    -0x8(%rbp),%al
 74c: jne    755 <func2+0x1e>
 74e: mov    $0x1,%eax
 753: jmp    75a <func2+0x23>
 755: mov    $0x0,%eax
 75a: pop    %rbp
 75b: retq
```

Figure 1: Stack representation, %rsp and %rbp are identified, and all multi-byte data is represented in big endian format.

Address	Value (64 bit)	Comments
0x7fffffffe908	0x00007ffff7a5a2b1	
0x7fffffffe900	0x0000555555554760	
0x7fffffffe8f8	0x00000001 0x00000002	Two 4-byte values
0x7fffffffe8f0	0x00007fffffffe9e0	
0x7fffffffe8e8	0x55554530 0x00000004	Two 4-byte values
0x7fffffffe8e0	0x00007fffffffe9e8	
0x7fffffffe8d8	0x00005555555546a3	
0x7fffffffe8d0	0x00007fffffffe900	← %rbp
0x7fffffffe8c8	0x555547ad 0x00000001	Two 4-byte values
0x7fffffffe8c0	0x00000001 0x00000000	Two 4-byte values
0x7fffffffe8b8	0xf7ad2fe0 0x00000002	Two 4-byte values
0x7fffffffe8b0	0x00007fffffffec98	← %rsp
0x7fffffffe8a8	0x000055555555470b	