Name: ______________________________

1. (10 points) For the following, Check T if the statement is true, or F if the statement is false.

   (a) [X] T [ ] F : If an X86 instruction modifies the data in the %al register in X86, then it also modifies the values in the %ax, %eax, and %rax registers.

   (b) [ ] T [X] F : If there is no width suffix (such as 'b', 'w', 'l', or 'q') associated with an X86 op-code, and none of the arguments of the instruction are registers, then the op-code will perform a 64 bit operation.

   If no width is specified, then the assembler flags the instruction as illegal. There is no default width in X86.

   (c) [X] T [ ] F : One of the reasons that X86-64 is so complicated is because it is downward compatible with over 40 years of X86 architecture development, including a version of X86 that ran on the very first personal computers in the 1970's.

   (d) [X] T [ ] F : If I execute the instruction "test -0x4(%rbp),$0x1", followed by the instruction "je .L5", then the jump to .L5 will occur only if the four bytes of memory at %rbp-4, interpretted as either a signed or unsigned number, is an exact multiple of some number times 2.

   (e) [X] T [ ] F : The X86 "mov" instruction handles initializing a register to a constant, initializing memory to a constant, copying data from one register to another, copying data from a register to memory, and copying data from memory to a register, but cannot copy data from memory to another location in memory.

   (f) [ ] T [X] F : If two computers have different microprocessor chips, then they require different Instruction Set Architectures (ISA's) in order to support the different hardware.

   Many different hardware implementations can support the same Instruction Set Architecture, and it is very common for very different microprocessor chips to support a single ISA.

   (g) [ ] T [X] F : The hardware required to add two unsigned integers to each other is different than the hardware required to add two signed integers to each other.

   The hardware is exactly the same. The only difference is how overflow is handled.

   (h) [X] T [ ] F : If you are debugging code that has been compiled by gcc without the -g flag, then the gdb "next" command will execute instructions until either the next breakpoint is reached, or the program either normally or abnormally ends.

   (i) [X] T [ ] F : In the X86 calling conventions, the caller pushes the return address on the stack in the "callq" instruction, and the callee pops the return address from the stack in the "retq" instruction. This violates stack ettiquette because the caller does not pop everything it pushed, but it still works because the return address will always be popped just before returning to the caller.

   (j) [ ] T [X] F : In the X86-64 ISA, while an instruction is executing in the ALU, the %rip register contains the address in memory of that instruction.

   The %rip register points to the *next* instruction as soon as the previous instruction is decoded.

Answer the following by checking all correct answers.

2. (6 points) Given the stack memory dump in figure 1 on page 9, and assuming that UNIX loaded the code at address 0x564ce4d02000, which callq instruction in either Listing 2 on page 8, Listing 3 on page 8, or Listing 4 on page 8 was run by the caller to generate the stack frame values?

   - [ ] 963: callq a31 <printBin>
   - [ ] 96d: callq 991 <leftBit>
   - [X] a0b: callq a31 <printBin>
   - [ ] a66: callq 740 <putchar@plt>
   - [ ] a98: callq aa0 <printStackInfo>
   - [ ] None of the above

   The return address is above where %rbp points, and subtracting the return address from the load location gets offset a10, which is the instruction after the callq instruction that generated the stack frame values. Note that the question should have read generate the **current** stack frame values. Or better yet, which callq instruction invoked the function executing in the current stack frame. Given the ambiguous question, "b" is an acceptable answer as well, because leftBit's return address of 0000564ce4d02972 has offset 972, which is the return from the callq at offset 96d. I gave 2 points partial credit for a98 printStackInfo because that is what prints the stack info (and it's in the call stack, but not printed).

3. (6 points) The declaration/assignment statement on line 21 in leftBit.c from Listing 1 on page 7 caused the gcc compiler to generate an instruction at which offset in Listing 3 on page 8?

   [ ] 999 [ ] 9a2 [X] 9ac [ ] 9b3 [ ] 9bf [ ] None of the above

   Line 21 initializes the local variable "w" to 32 or 0x20 after the if statement on line 20. If the if condition on line 20 is false, the code jumps to offset 9ac, which assigns 0x20 to -0x4(%rbp) which is a valid location for a local variable, so 9ac is the correct offset.

4. (6 points) The leftBit from Listing 1 on page 7 caused the gcc compiler to generate the object code in Listing 3 on page 8. In the leftBit function, which non-volatile (blue) registers are modified inside the leftBit function, and must be restored before leftBit returns? (Check all that apply.)

   [ ] %rbx [X] %rsp [X] %rbp [ ] %r12 [ ] %r13 [ ] %r14 [ ] %r15

   The %rbp value is pushed on the stack in the preamble and popped off the stack in the exit code. The %rsp value is not saved and restored in the stack, but it is logically saved and restored since we can derive it's value from %rbp. None of the other non-volatile registers are used in the leftBit function.

5. (6 points) The value "X= " referenced in the C instruction on line 14 in Listing 1 on page 7 is kept in which section of the ELF executable file generated by the compiler from the leftBit.c code?

   [ ] .text [ ] .plt_got [X] .rodata [ ] .data [ ] .bss [ ] None of the above

   Since the value does not fit in an instruction, the compiler needs to put it in a data section. It has an initial value, so it can't be in .bss, and there is no way to modify it, so .rodata is a better choice than .data.

6. (6 points) The x86 "sar" instruction at offset 9c6 in Listing 3 on page 8 shifts the value in the %eax register one bit to the right. In class, we learned that shifting one bit to the right is almost the same as dividing by two, but shifting to the right always rounds down, whereas dividing by two should round towards zero. With this in mind, what is the range of offsets of instructions generated by the gcc compiler to implement the C instruction on line 24 in Listing 1 on page 7?
☐ 9ac-9b3 ☐ 9bc-9c1 ☐ 9c6-9c8 ☐ 9bc-9c6 ☒ 9bc-9c8 ☐ None of the above
The gcc compiler checks to see if "w" is negative starting at offset 9bc by copying it to %edx, and shifting to the right by 31 bits, leaving just the sign bit in %edx. It adds the sign bit to the "w" value before shifting, so that if "w" is negative, it will round up instead of rounding down. The "sar" instruction at offset 9c6 does the divide, and the result is copied to "hw" at -0cx(%rbp) at offset 9c8.

7. (6 points) The C if condition on line 20 in leftBit.c from Listing 1 on page 7 caused the gcc compiler to generate a compare instruction at which offset in either Listing 2 on page 8, Listing 3 on page 8, or Listing 4 on page 8?
☐ 90f ☒ 99c ☐ a26 ☐ a54 ☐ a83 ☐ None of the above
Line 20 is in the leftBit function, and the if statement is the first instruction, so the compare right after the preamble of the leftBit function at offset 99c is the correct offset. Furthermore, it is clear that this instruction compares a zero value to the first parameter of leftBit.

8. (6 points) The C while condition on line 23 in leftBit.c from Listing 1 on page 7 caused the gcc compiler to generate a compare instruction at which offset in either Listing 2 on page 8, Listing 3 on page 8, or Listing 4 on page 8?
☐ 90f ☐ 99c ☒ a26 ☐ a54 ☐ a83 ☐ None of the above
The gcc compiler translates a while loop by putting the condition at the bottom of the loop, and branching down to that condition at loop entry, in this case, the jmp instruction at offset 9ba, which jumps to a26. The instruction at a26 compares a 1 value against -0x4(%rbp), which from the initializations, is a reference to the "w" local variable.

9. (6 points) Given the stack memory dump in figure 1 on page 9, what is the value of the caller's %rbp register?
☐ 0x564ce4d02a9d ☐ 0x564ce60c96e0 ☐ 0x564ce4d02c80 ☐ 0xffffffff33bdf170
☒ 7ffdc52e0070 ☐ None of the above
The %rbp register always points at the value of the caller's %rbp.

Answer the following questions by filling in the blanks.

10. (6 points) Based on the stack information in figure 1 on page 9, and the x86 assembler code derived from leftBit.c in Listing 2 on page 8, Listing 3 on page 8 and Listing 4 on page 8, and assuming the code is loaded at 0x564ce4d02000, what is the value of the parameter, `n`, in the stack frame of the currently executing function? (You may express your answer in hexadecimal.)
`0x0000ff00`
The current %rbp points at the top of the current stack frame. The return address above the frame is 0x564ce4d02a10, so the offset is 0xa10, which the instruction after the call to printBin, so the current function must be printBin. The instruction at offset a39 saves the parameter value at -0x14(%rbp), or, in this case, 0x7ffdc52e002c. The 4 byte value at that address is currently 0x0000ff00.

11. (6 points) Based on the stack information in figure 1 on page 9, and the x86 assembler code derived from leftBit.c in Listing 2 on page 8, Listing 3 on page 8 and Listing 4 on page 8, and assuming the code is loaded at 0x564ce4d02000, what is the value of the local variable, `x`, in the current function's caller's caller's stack frame? (You may express your answer in hexadecimal.)

`0x00001003`

The current %rbp points to the caller's %rbp or 0x7ffdc52e0070. The caller's %rbp is pointing at the caller's caller's %rbp or 0x7ffdc52e00a0. The return address above the caller's frame is 0x564ce4d02972, so the offset is 0x972, which is an instruction in the main function, so the caller's caller must be main. The instruction at offset 94a writes the return value from atoi(argv[1]) to the local variable at -0x4(%rbp), so x must be at -0x4(%rbp). In this case, that is, 0x7ffdc52e009c. The 4 byte value at that address is currently 0x00001003.

12. (6 points) Based on the stack information in figure 1 on page 9, and the x86 assembler code derived from leftBit.c in Listing 2 on page 8, Listing 3 on page 8 and Listing 4 on page 8, and assuming the code is loaded at 0x564ce4d02000, what is the value of the local variable, `mask`, in the current function's caller's stack frame? (You may express your answer in hexadecimal.)

`0x0000ff00`

The current %rbp points at the top of the current stack frame. The return address above the frame is 0x564ce4d02a10, so the offset is 0xa10, which the instruction in the leftBit function, so the caller must be leftBit. The caller's %rbp is the value that %rbp is pointing at, or 0x7ffdc52e0070. The instruction at offset a06 copies the first local variable at -0x10(%rbp) to what will become the argument to printBin, so mask must be at -0x10(%rbp). In this case, that is, 0x0x7ffdc52e0060. The 4 byte value at that address is currently 0x0000ff00.

13. (6 points) Based on the stack information in figure 1 on page 9, and the x86 assembler code derived from leftBit.c in Listing 2 on page 8, Listing 3 on page 8 and Listing 4 on page 8, and assuming the code is loaded at 0x564ce4d02000, what is the value of the parameter, `x`, in the current function's caller's stack frame? (You may express your answer in hexadecimal.)

`0x00001003`

The current %rbp points at the top of the current stack frame. The return address above the frame is 0x564ce4d02a10, so the offset is 0xa10, which the instruction in the leftBit function, so the caller must be leftBit. The caller's %rbp is the value that %rbp is pointing at, or 0x7ffdc52e0070. The instruction at offset 999 saves the parameter value at -0x14(%rbp), or, in this case, 0x0x7ffdc52e005c. The 4 byte value at that address is currently 0x00001003.

14. (6 points) Based on the stack information in figure 1 on page 9, and the x86 assembler code derived from leftBit.c in Listing 2 on page 8, Listing 3 on page 8 and Listing 4 on page 8, and assuming the code is loaded at 0x564ce4d02000, what is the value of the local variable, `w`, in the current function's caller's stack frame? (You may express your answer in hexadecimal.)

`16 or 0x010`

The current %rbp points at the top of the current stack frame. The return address above the frame is 0x564ce4d02a10, so the offset is 0xa10, which the instruction in the leftBit function, so the caller must be leftBit. The caller's %rbp is the value that %rbp is pointing at, or 0x7ffdc52e0070. The instruction at offset 9ac initializes the first local variable at -0x4(%rbp) to 0x20, or 32, so w must be at -0x4(%rbp). In this case, that is, 0x0x7ffdc52e006c. The 4 byte value at that address is currently 0x00000010.

15. (6 points) Based on the stack information in figure 1 on page 9, and the x86 assembler code derived from leftBit.c in Listing 2 on page 8, Listing 3 on page 8 and Listing 4 on page 8, does the printBin function use the red zone? If not, what prevents gcc from using the red zone for the printBin function

printBin cannot use the red zone becuase it invokes lower level function printf

printBin is not a leaf function because it invokes a lower level function, and only leaf functions can use the red zone.

16. (6 points) Running the command `objdump -s -j.rodata leftBit` on the executable produced by compiling the code in Listing 1 on page 7 produces the following output...

```
leftBit:     file format elf64-x86-64

Contents of section .rodata:
 0d00 01000200 00000000 496e766f 6b652061  ........Invoke a
 0d10 73202573 203c6e3e 200a0977 68657265  s %s <n> ..where
 0d20 203c6e3e 20697320 616e2069 6e746567   <n> is an integ
 0d30 65720a00 583d2000 54686520 6c656674  er..X= .The left
 0d40 6d6f7374 20626974 206f6620 25642069  most bit of %d i
 0d50 73206174 20706f73 6974696f 6e202564  s at position %d
 0d60 0a002068 773d2564 206e3d25 64204d3d  .. hw=%d n=%d M=
 0d70 00000000 00000000 6261636b 74726163  ........backtrac
 0d80 655f7379 6d626f6c 73282900 00000000  e_symbols().....
 0d90 202d2d2d 2d2d2d2d 2d2d2d2d 2d2d2d2d   ---------------
 ...
```

What is the offset of the first argument to the printf function invoked on line 26? (You may express your answer in hexadecimal.)

0xd62

The first argument to the printf function is the literal string " hw=%d n=%d M=", which appears in the .rodata section of the executable file at offset d62 from above, or at offset 9f5, lea 0x366(%rip), where %rip is at offset 9fc, and 9fc+366=9dc.

Tear-off Page

Listing 1: leftBit.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

int leftBit(int x);
void printBin(int n);

int main(int argc, char **argv) {
    if (argc<2) {
        printf("Invoke as %s <n> \n\twhere <n> is an integer\n",argv[0]);
        return 1;
    }
    int x=atoi(argv[1]);
    printf("X= "); printBin(x);
    printf("The leftmost bit of %d is at position %d\n",x,leftBit(x));
    return 0;
}

int leftBit(int x) {
    if (x==0) return -1;
    int w=32; // Number of bits that might contain leftmost 1
    int n=0; // Rightmost bit that might contain leftmost 1
    while(w>1) { //Narrow down to a single bit
        int hw=w/2; // Look at half the range of bits
        int mask=((1<<hw)-1)<<(n+hw); // mask : nw ones in left half of range
        printf(" hw=%d n=%d M=",hw,n); printBin(mask);
        if (x&mask) n=n+hw; // If left half has a one bit, start at left half
        w=hw; // Ruled out either the left half or the right half
    }
    return n;
}

void printBin(int n) {
    int i;
    for(i=31;i>=0;i--) {
        printf("%c",(n&1<<i)?'1':'0');
        if (0==i%4) printf(" ");
    }
    printf("\n");
    printStackInfo();
}
```

Listing 2: leftBit.s(main)

```
900:  push   %rbp
901:  mov    %rsp,%rbp
904:  sub    $0x20,%rsp
908:  mov    %edi,-0x14(%rbp)
90b:  mov    %rsi,-0x20(%rbp)
90f:  cmpl   $0x1,-0x14(%rbp)
913:  jg     937 <main+0x37>
915:  mov    -0x20(%rbp),%rax
919:  mov    (%rax),%rax
91c:  mov    %rax,%rsi
91f:  lea    0x3e2(%rip),%rdi
926:  mov    $0x0,%eax
92b:  callq  780 <printf@plt>
930:  mov    $0x1,%eax
935:  jmp    98f <main+0x8f>
937:  mov    -0x20(%rbp),%rax
93b:  add    $0x8,%rax
93f:  mov    (%rax),%rax
942:  mov    %rax,%rdi
945:  callq  7a0 <atoi@plt>
94a:  mov    %eax,-0x4(%rbp)
94d:  lea    0x3e0(%rip),%rdi
954:  mov    $0x0,%eax
959:  callq  780 <printf@plt>
95e:  mov    -0x4(%rbp),%eax
961:  mov    %eax,%edi
963:  callq  a31 <printBin>
968:  mov    -0x4(%rbp),%eax
96b:  mov    %eax,%edi
96d:  callq  991 <leftBit>
972:  mov    %eax,%edx
974:  mov    -0x4(%rbp),%eax
977:  mov    %eax,%esi
979:  lea    0x3b8(%rip),%rdi
980:  mov    $0x0,%eax
985:  callq  780 <printf@plt>
98a:  mov    $0x0,%eax
98f:  leaveq
990:  retq
```

Listing 3: leftBit.s(leftBit)

```
991:  push   %rbp
992:  mov    %rsp,%rbp
995:  sub    $0x20,%rsp
999:  mov    %edi,-0x14(%rbp)
99c:  cmpl   $0x0,-0x14(%rbp)
9a0:  jne    9ac <leftBit+0x1b>
9a2:  mov    $0xffffffff,%eax
9a7:  jmpq   a2f <leftBit+0x9e>
9ac:  movl   $0x20,-0x4(%rbp)
9b3:  movl   $0x0,-0x8(%rbp)
9ba:  jmp    a26 <leftBit+0x95>
9bc:  mov    -0x4(%rbp),%eax
9bf:  mov    %eax,%edx
9c1:  shr    $0x1f,%edx
9c4:  add    %edx,%eax
9c6:  sar    %eax
9c8:  mov    %eax,-0xc(%rbp)
9cb:  mov    -0xc(%rbp),%eax
9ce:  mov    $0x1,%edx
9d3:  mov    %eax,%ecx
9d5:  shl    %cl,%edx
9d7:  mov    %edx,%eax
9d9:  lea    -0x1(%rax),%esi
9dc:  mov    -0x8(%rbp),%edx
9df:  mov    -0xc(%rbp),%eax
9e2:  add    %edx,%eax
9e4:  mov    %eax,%ecx
9e6:  shl    %cl,%esi
9e8:  mov    %esi,%eax
9ea:  mov    %eax,-0x10(%rbp)
9ed:  mov    -0x8(%rbp),%edx
9f0:  mov    -0xc(%rbp),%eax
9f3:  mov    %eax,%esi
9f5:  lea    0x366(%rip),%rdi
9fc:  mov    $0x0,%eax
a01:  callq  780 <printf@plt>
a06:  mov    -0x10(%rbp),%eax
a09:  mov    %eax,%edi
a0b:  callq  a31 <printBin>
a10:  mov    -0x14(%rbp),%eax
a13:  and    -0x10(%rbp),%eax
a16:  test   %eax,%eax
a18:  je     a20 <leftBit+0x8f>
a1a:  mov    -0xc(%rbp),%eax
a1d:  add    %eax,-0x8(%rbp)
a20:  mov    -0xc(%rbp),%eax
a23:  mov    %eax,-0x4(%rbp)
a26:  cmpl   $0x1,-0x4(%rbp)
a2a:  jg     9bc <leftBit+0x2b>
a2c:  mov    -0x8(%rbp),%eax
a2f:  leaveq
a30:  retq
```

Listing 4: leftBit.s(printBin)

```
a31:  push   %rbp
a32:  mov    %rsp,%rbp
a35:  sub    $0x20,%rsp
a39:  mov    %edi,-0x14(%rbp)
a3c:  movl   $0x1f,-0x4(%rbp)
a43:  jmp    a83 <printBin+0x52>
a45:  mov    -0x4(%rbp),%eax
a48:  mov    -0x14(%rbp),%edx
a4b:  mov    %eax,%ecx
a4d:  sar    %cl,%edx
a4f:  mov    %edx,%eax
a51:  and    $0x1,%eax
a54:  test   %eax,%eax
a56:  je     a5f <printBin+0x2e>
a58:  mov    $0x31,%eax
a5d:  jmp    a64 <printBin+0x33>
a5f:  mov    $0x30,%eax
```

```
a64:  mov    %eax,%edi
a66:  callq  740 <putchar@plt>
a6b:  mov    -0x4(%rbp),%eax
a6e:  and    $0x3,%eax
a71:  test   %eax,%eax
a73:  jne    a7f <printBin+0x4e>
a75:  mov    $0x20,%edi
a7a:  callq  740 <putchar@plt>
a7f:  subl   $0x1,-0x4(%rbp)
a83:  cmpl   $0x0,-0x4(%rbp)
```

```
a87:  jns    a45 <printBin+0x14>
a89:  mov    $0xa,%edi
a8e:  callq  740 <putchar@plt>
a93:  mov    $0x0,%eax
a98:  callq  aa0 <printStackInfo>
a9d:  nop
a9e:  leaveq
a9f:  retq
```

Figure 1: Contents of Stack Memory

Address	64-bit Value (big-endian)	Value (32 bit) +0	+4	Comments
0x7ffdc52e00a0	0000564ce4d02c80	e4d02c80	0000564c	← main's %rbp
0x7ffdc52e0098	0000100300000000	00000000	00001003	
0x7ffdc52e0090	00007ffdc52e0180	c52e0180	00007ffd	
0x7ffdc52e0088	00000002e4d027d0	e4d027d0	00000002	
0x7ffdc52e0080	00007ffdc52e0188	c52e0188	00007ffd	
0x7ffdc52e0078	0000564ce4d02972	e4d02972	0000564c	← leftBit's ret addr & main's %rsp
0x7ffdc52e0070	00007ffdc52e00a0	c52e00a0	00007ffd	← leftBit's %rbp
0x7ffdc52e0068	0000001000000000	00000000	00000010	
0x7ffdc52e0060	000000080000ff00	0000ff00	00000008	
0x7ffdc52e0058	0000100300000000	00000000	00001003	
0x7ffdc52e0050	00007ffdc52e0180	c52e0180	00007ffd	
0x7ffdc52e0048	0000564ce4d02a10	e4d02a10	0000564c	← ret addr & leftBit's %rsp
0x7ffdc52e0040	00007ffdc52e0070	c52e0070	00007ffd	← %rbp
0x7ffdc52e0038	ffffffff33bdf170	33bdf170	ffffffff	
0x7ffdc52e0030	0000564ce4d02c80	e4d02c80	0000564c	
0x7ffdc52e0028	0000ff0000000000	00000000	0000ff00	
0x7ffdc52e0020	0000564ce60c96e0	e60c96e0	0000564c	
0x7ffdc52e0018	0000564ce4d02a9d	e4d02a9d	0000564c	← %rsp

Question:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	Total
Points:	10	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	100
Bonus Points:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0