1. Operators – consider the following code:

(hint: you should know what an electrical short-circuit is, but do you know what a logic short-circuit is?)

```
#include <stdio.h>

int main()
{
  int a = 1;
 int b = 2;
 int c = !a || b;
 int d = a || --b;
 int e = a-- && --b;
 printf("a = %d, b = %d, c = %d, d = %d, e = %d", a, b, c, d, e);
 return 0;
}
```

What is the output of the program? (5 points)

a = 0, b = 1, c = 1, d = 1, e = 1

When determining “d”, a = 1, so the compiler will perform a logic reduction (1 || “anything” = 1) and assign 1 to d. The sub-expression (--b) will not be evaluated. This is called “short-circuiting” in the C language; a and b retain their original values after evaluating “d”.

The post-decrement of “a” means the original value is used when determining “e”. The pre-decrement of “b” means it is evaluated and b=1 is used when determining “e”.

2. Preprocessor directives – consider the following code:

```
#include <stdio.h>
#define square(num) num*num
int main()
{
  int val1, val2;
 val1 = 16 / square(4);
 val2 = 16 - square(4);
 printf("Value 1 = %d, Value 2 = %d", val1, val2);
```

return 0; } a) What is the output of this program? (2 points) Value 1 = 16, Value 2 = 0 b) How does preprocessing alter the lines calculating val1 and val2? (2 points) val1 = 16/4*4; val2 = 16-4*4; c) How could the define statement be improved to result in more reliable behaviour? (1 point) #define square(num) ((num)*(num))

3. Bitmasking – determine the output of the printf statements

a) printf("Ans: %d", 13 | (1 << 5)); (1 point)

Ans: 45

b) printf("Ans: %X", (0x4000D00D | 0xA)); (1 point)

Ans: 4000D00F

c) printf("Ans: %d", 80 & ~16); (1 point)

Ans: 64

d) Consider the following code and determine the output of the printf statement (2 points)

```
#include <stdio.h>
#define ASSIGN_R 0x4A4253FC
int main()
{
  long val1, val2;
 val1 = ASSIGN_R;
val1 &= ~0x77;
val2 = val1;
val2 ^= 0xFF;
printf("Value 1: %X, Value 2: %X", val1, val2);
return 0;
}
```

Value 1: 4A425388, Value 2: 4A425377

4. Pointers a) Consider the following code and determine the output of the printf statement (1 point) #include <stdio.h> void func(int *ptr) { } int main() {

*ptr = 100;

int val = 5;
func(&val);
printf("Ans: %d", val);
return 0;
}
Ans: 100

b) Consider the following code and determine the output of the printf statement (1 point)
#include <stdio.h>
void func(int *ptr, int num)
{

num = num + 15;
*ptr = *ptr + num;
return;
}
int main()
{

int val1 = 10, val2 = 25;
func(&val1,val2);
printf("Ans: %d", val1+val2);
return 0;
}
Ans: 75

c) Consider the following code and determine the output of the printf statement (2 points)

```
#include <stdio.h>
void func(int *ptr1, int *ptr2)
{

  ptr1 = ptr2;
 *ptr1 = 2;
return;
}
int main()
{

  int val1 = 10, val2 = 20;
 func(&val1,&val2);
printf("Value 1: %d, Value 2: %d", val1, val2);
return 0;
}
```
Value 1: 10 Value 2: 2

d) Consider the following code and determine the output of the printf statement (1 point)

```
#include <stdio.h>
int main()
{


  int *ptr;
 int val;
ptr = &val;
*ptr = 0;
*ptr += 5;
printf("Value 1: %d", val);
return 0;
}
```
Value 1: 5