## 1 Introduction

This assignment will feel somewhat familiar in that it is nearly identical to the preceding assignment: there is a coding problem and a puzzle-solving problem. The major change is that everything is at the assembly level:

* Problem 1 re-works the LCD clock functions in assembly rather than C
* Problem 2 involves analyzing a binary executable to provide it with the correct input to "defuse" the executable

Working with assembly will get you a much more acquainted with the low-level details of the x86-64 platform and give you a greater appreciation for "high-level" languages (like C).

## 2 Download Code and Setup

Download the code pack linked at the top of the page. Unzip this which will create a project folder. Create new files in this folder. Ultimately you will re-zip this folder to submit it.

| **File** | **State** | **Notes** |
| --- | --- | --- |
| Makefile | Provided | Problem 1 Build file |
| lcd.h | Provided | Problem 1 header file |
| lcd\_main.c | Provided | Problem 1 main() function |
| lcd\_clock.c | Provided | Problem 1 lcd clock drawing functions |
| lcd\_update.c | Create (?) | Problem 1 C functions, may copy from previous assignment |
| lcd\_update\_asm.s | Create | Problem 1 Assembly functions, re-code C in x86-64 |
|  |  |  |
| test\_lcd.c | Testing | Problem 1 binary tests for lcd\_upate.c |
| test\_lcd\_main.sh | Testing | Problem 1 shell tests for lcd clock |
| test\_lcd\_main\_data.sh | Testing | Problem 1 shell test data for lcd clock |
| bombNN.tar | Download | Problem 2 Debugging problem, [download from server](http://www-users.cs.umn.edu/~kauffman/2021/a3.html#bomb-download) |
| bombNN/bomb.c | Unpack | Problem 2 Unpack from .tar file, main() for bomb |
| bombNN/bomb | Unpack | Problem 2 Unpack from .tar executable to debug |
| bombNN/README | Unpack | Problem 2 Unpack from .tar contains "owner" of the bomb |
| input.txt | Edit | Problem 2 Input for bomb, fill this in |

## 3 Problem 1: LCD Assembly Functions

The functions in this problem are identical to a previous assignment in which code to support an LCD clock display was written. These functions are:

**int set\_tod\_from\_secs(int time\_of\_day\_sec, tod\_t \*tod)**

Given the number of seconds from the start of the day, set the fields of the struct pointed to by tod to have the correct hours, minutes, seconds, and pm indication.

**int set\_display\_bits\_from\_tod(tod\_t tod, int \*display)**

Given a tod\_t struct, reset and alter the bits pointed to by display to cause a proper clock display.

**int lcd\_update()**

Update global LCD\_DISPLAY\_PORT using the TIME\_OF\_DAY\_SECS. Call the previous two functions.

The big change in this iteration will be that **the functions must be written in x86-64 assembly code**. As C functions each of these is short, 30-50 lines maximum. The assembly versions will be somewhat longer as each C line typically needs 1-4 lines of assembly code to implement fully. Coding these functions in assembly give you real experience writing working assembly code and working with it in combination with C.

**The code setup and tests are identical** for this problem as for the previous C version of the problem. [Refer to original LCD Clock Problem description](http://www-users.cs.umn.edu/~kauffman/2021/a2.html#sec-3) for a broad overview of the clock simulator and files associated with it.

### 3.1 Write Your Assembly

As discussed in class, one can generate assembly code from C code with appropriate compiler flags. This can be useful for getting oriented and as a beginning to the code your assembly versions of the functions. However, code that is clearly compiler-generated with no hand-tweaking will

* Receive little credit on manual inspection
* Receive penalties on testing which lowers credit associated with that portion

Do not let that dissuade you from looking at compiler-generated assembly code from you C solution to the functions. Make sure that you take the following steps which are part of the manual inspection criteria.

#### Base your Assembly code on your C code

The files to be submitted for this problem include

* lcd\_update.c: C version of the functions
* lcd\_update\_asm.s: Assembly version of the functions

Graders will examine these for a correspondence between to the algorithm used in the C version to the Assembly version. Compiler generated assembly often does significant re-arrangements of assembly code with many intermediate labels that hand-written code will not have.

If you were not able to complete the C functions for the LCD clock from the previous assignment, see a course staff member who will help you get them up and running quickly.

#### Annotate your Assembly Thoroughly

Comment your assembly code A LOT. While good C code can be quite self-explanatory with descriptive variable names and clear control structures, assembly is rarely so easy to understand. Include clear commentary on your assembly. This should include

* Division of functions into smaller blocks with comments describing what the blocks accomplish
* Descriptions of which "variables" from the C side are held in which registers
* Descriptions of most assembly lines and their effect on the variables held in the registers
* Descriptions of any data such as bitmasks stored in the assembly code.

#### Use Division

While it is a slow instruction that is cumbersome to set up, using division is the most human-readable means to compute several results needed in the required functions. Compiler generated code uses many tricks to avoid integer division so a lack of assembly instructions along this line will be a clear sign little effort has been put into the assembly code.

### 3.2 General Cautions when coding Assembly

1. Careful with constants: forgetting a $ in constants will lead to a bare, absolute memory address which will likely segfault your program. Compare:
2. movq $0,%rax # rax = 0
3. movq 0, %rax # rax = \*(0): segfault
4. # bare 0 is memory address 0 - out of bounds

Running your programs, assembly code included, in Valgrind can help to identify these problems. In Valgrind output, look for a line number in the assembly code which has absolute memory addresses or a register that has an invalid address.

1. Be disciplined about your register use: comment what "variables" are in which registers as it is up to you to keep track. Comments here are as helpful to you as to other readers.
2. Recognize that in x86-64 function parameters are passed in registers for up to 6 arguments. These are arranged as follows
   1. rdi / edi / di (arg 1)
   2. rsi / esi / si (arg 2)
   3. rdx / edx / dx (arg 3)
   4. rcx / ecx / cx (arg 4)
   5. r8 / r8d / r8w (arg 5)
   6. r9 / r9d / r9w (arg 6)

and the specific register corresponds to how argument sizes (64 bit args in rdi, 32 bit in edi, etc). The functions you will write have few arguments so they will all be in registers.

1. Use registers sparingly. The following registers (64-bit names) are "scratch" registers or "caller save." Functions may alter them freely (though some may contain function arguments).
2. rax rcx rdx rdi rsi r8 r9 r10 r11 # Caller save registers

No special actions need to be taken at the end of the function regarding these registers except that rax should contain the function return value.

Remaining registers are "callee save": if used, their original values must be restored before returning from the function.

rbx rbp r12 r13 r14 r15 # Callee save registers

This is typically done by pushing the callee registers to be used on the stack, using them, them popping them off the stack in reverse order. Avoid this if you can (and you probably can in our case).

1. Don't mess with the stack pointer rsp unless you know what you are doing.

### 3.3 Iterative Development Strategy

With working C versions of the three required functions, you should be able to employ the an iterative strategy while developing assembly versions: **focus on one assembly function while using working C functions for the remaining code**. The Makefile and support code are specifically set up for this and details iterative development are as follows.

1. In lcd\_update\_asm.s, comment all code/declarations of global symbols except for set\_tod\_from\_secs.
2. In lcd\_update.c, comment out the definition of the set\_tod\_from\_secs() function. This leaves working versions of the other two functions.
3. Write some assembly code for set\_tod\_from\_secs. Attempt to compile it on its own with
4. > make lcd\_update\_asm.o
5. gcc -Wall -g -c lcd\_update\_asm.s

which will report assembler errors if any are present

1. When there appear to be no assembler errors, create a **hybrid** main program which will combine the uncommented assembly and C functions
2. > make hybrid\_main
3. gcc -Wall -g -c lcd\_main.c
4. gcc -Wall -g -c lcd\_clock.c
5. gcc -Wall -g -c lcd\_update\_asm.s # compiles assembly ..
6. gcc -Wall -g -c lcd\_update.c # and C version to produce executable
7. gcc -Wall -g -o hybrid\_main lcd\_main.o lcd\_clock.o lcd\_update\_asm.o lcd\_update.o
8. One can then experiment with the hybrid\_main to see if the written assembly function is working correctly.
9. > ./hybrid\_main 10000
10. TIME\_OF\_DAY\_SEC set to: 10000
11. set\_tod\_from\_secs( 10000, &tod );
12. tod is {
13. .hours = -7
14. .minutes = 891231
15. .seconds = -256
16. .ispm = 0
17. }
18. ...

Something clearly looks wrong in the above so some debugging of the assembly function seems in order.

1. When ready, run the "hybrid tests" which will do automated testing on the hybrid codes.
2. > make test-hybrid
3. gcc -Wall -g -c test\_lcd.c
4. ...
5. ===TESTS for Hybrid===
6. Running binary tests for hybrid
7. ./test\_hybrid
8. Test 1: set\_tod\_from\_secs() midnight : OK
9. Test 2: set\_tod\_from\_secs() after-midnight : OK
10. Test 3: set\_tod\_from\_secs() 1am : FAIL
11. ...
12. When all seems to be working correctly with the first assembly function, move on. Comment another C function and uncomment the corresponding assembly, write some code and repeat.

### 3.4 Structure of lcd\_update\_asm.s

Below is a rough outline of the structure of lcd\_updat\_asm.s. Consider copying this file as you get started and commenting parts of it out as needed.

.text

.global set\_tod\_from\_secs

## ENTRY POINT FOR REQUIRED FUNCTION

set\_tod\_from\_secs:

## assembly instructions here

### Data area associated with the next function

.data

my\_int: # declare location an single int

.int 1234 # value 1234

my\_array: # declare multiple ints in a row

.int 10 # for an array. Each are spaced

.int 20 # 4 bytes from each other

.int 30

.text

.global set\_display\_bits\_from\_tod

## ENTRY POINT FOR REQUIRED FUNCTION

set\_display\_bits\_from\_tod:

## assembly instructions here

## two useful techniques for this problem

movl my\_int(%rip),%eax # load my\_int into register eax

leaq my\_array(%rip),%edx # load pointer to beginning of my\_array into edx

.text

.global lcd\_update

## ENTRY POINT FOR REQUIRED FUNCTION

lcd\_update:

## assembly instructions here

## a useful technique for this problem

movl TIME\_OF\_DAY\_SEC(%rip), %eax # load global variable into eax

### 3.5 set\_tod\_from\_secs

int set\_tod\_from\_secs(int time\_of\_day\_sec, tod\_t \*tod);

// Accepts time of day in seconds as an argument and modifies the

// struct pointed at by tod to fill in its hours, minutes,

// etc. fields. If time\_of\_day\_sec is invalid (negative or larger

// than the number of seconds in a day) does not tod and returns 1 to

// indicate an error. Otherwise returns 0 to indicate success. This

// function DOES NOT modify any global variables

#### Assembly Implementation Notes set\_tod\_from\_secs

1. The function takes two arguments
   * an int which will be in register edi
   * a pointer which will be in rsi.

Note the 32-bit versus 64-bit registers.

1. Return values or functions are to be placed eax for 32 bit quantities as is the case here (int).
2. Use comparisons and jump to a separate section of code that is clearly marked as "error" if you detect a bad arguments.
3. Make use of division to "break down" the argument time\_of\_day\_secs. Keep in mind that the idivl instruction must have eax as the dividend, edx zeroed out. Any 32-bit register can contain the divisor. After the instruction, eax will hold the quotient and edx the remainder. With cleverness, you'll only need to do a couple divisions.
4. A pointer to a tod\_t struct can access its fields using the following offset table which assume that %reg holds a pointer to the struct (substitute an actual register name).

|  |  | **Destination** | **Assembly** |
| --- | --- | --- | --- |
| **C Field Access** | **Offset** | **Size** | **Assign 5 to field** |
| tod->hours | 0 bytes | 2 bytes | movw $5,0(%reg) |
| tod->minutes | 2 bytes | 2 bytes | movw $5,2(%reg) |
| tod->seconds | 4 bytes | 2 bytes | movw $5,4(%reg) |
| tod->ispm | 6 bytes | 1 byte | movb $5,6(%reg) |

1. You will need to use these offsets to set the fields of the struct near the end of the routine.

### 3.6 set\_display\_bits\_from\_tod

int set\_display\_bits\_from\_tod(tod\_t tod, int \*display);

// Accepts a tod and alters the bits in the int pointed at by display

// to reflect how the LCD clock should appear. If any fields of tod

// are negative or too large (e.g. bigger than 12 for hours, bigger

// than 59 for min/sec), no change is made to display and 1 is

// returned to indicate an error. Otherwise returns 0 to indicate

// success. This function DOES NOT modify any global variables

//

// May make use of an array of bit masks corresponding to the pattern

// for each digit of the clock to make the task easier.

#### Assembly Implementation Notes set\_display\_bits\_from\_tod

1. Arguments will be
   * a packed tod\_t struct in rdi
   * an integer pointer in rsi
2. The packed tod\_t struct is entirely in the 64-bit rdi register which has the following layout.

|  | **Bits** | **Shift** |  |
| --- | --- | --- | --- |
| **C Field Access** | **in rdi** | **Required** | **Size** |
| tod.hours | 00-15 | None | 2 bytes |
| tod.minutes | 16-31 | Right by 16 | 2 bytes |
| tod.seconds | 32-47 | Right by 32 | 2 bytes |
| tod.ispm | 48-57 | Right by 48 | 1 byte |

1. To access individual fields of the struct, you will need to do shifting and masking to extract the values from the rdi register.
2. Use comparisons and jump to a separate section of code that is clearly marked as "error" if you detect bad fields in the tod struct argument.
3. As was the case in the C version of the problem, it is useful to create a table of bit masks corresponding to the bits that should be set for each clock digit (e.g. digit "1" has bit patter 0b0000110). In assembly this is easiest to do by using a data section with successive integers. An example of how this can be done is below.
4. .section .data
5. array: # an array of 3 ints
6. .int 200 # array[0] = 200
7. .int 300 # array[1] = 300
8. .int 400 # array[3] = 400
9. const:
10. .int 17 # special constant
11. .section .text
12. .globl func
13. func:
14. leaq array(%rip),%r8 # r8 points to array, rip used to enable relocation
15. movq $2,%r9 # r9 = 2, index into array
16. movl (%r8,%r9,4),%r10d # r10d = array[2], note 32-bit movl and dest reg
17. movl const(%rip),%r11d # r11d = 17 (const), rip used to enable relocation

Adapt this example to create a table of useful bit masks for digits. The GCC assembler understands binary constants specified with the 0b0011011 style syntax.

1. Make use of division again to compute "digits" for the ones and tens place of the hours and minutes for the clock. Use these digits to reference into the table of digit bit masks you create to progressively build up the correct bit pattern for the clock display.
2. Use shifts and ORs to combine the digit bit patterns to create the final clock display bit pattern.

### 3.7 lcd\_update

int lcd\_update();

// Examines the TIME\_OF\_DAY\_SEC global variable to determine hour,

// minute, and am/pm. Sets the global variable LCD\_DISPLAY\_PORT bits

// to show the proper time. If TIME\_OF\_DAY\_SEC appears to be in error

// (to large/small) makes no change to LCD\_DISPLAY\_PORT and returns 1

// to indicate an error. Otherwise returns 0 to indicate success.

//

// Makes use of the set\_tod\_from\_secs() and

// set\_display\_bits\_from\_tod() functions.

#### Assembly Implementation Notes for lcd\_update

1. No arguments come into the function.
2. To access global symbols/variables which are not defined in the assembly file, use the relative position from the instruction pointer register which allows the linker to handle the task. Specifically relevant examples are
3. movl TIME\_OF\_DAY\_SEC(%rip), %edx # copy global var to reg edx
4. movl %r8d,LCD\_DISPLAY\_PORT(%rip) # copy reg r8d to global var
5. Call the two previous functions to create the struct and manipulate the bits of an the display. Calling a function requires that the stack be aligned to 16-bytes; there is always an 8-byte quantity on the stack (previous value of the rsp stack pointer). This means the stack must be extended with a pushq instruction before any calls. A typical sequence is
6. pushq %rdx # push any 64-bit register onto the stack
7. call some\_func # stack aligned, call function
8. ## return val from func in rax or eax
9. popq %rdx # restore the stack
10. If several function calls will be made, a single push is all that is needed as in the below
11. pushq %rdx # push any 64-bit register onto the stack
12. call some\_func1 # stack aligned, call function
13. ## return val from func in rax or eax
14. ## do some more stuff
15. call some\_func2 # stack aligned, call function
16. ## return val from func in rax or eax
17. popq %rdx # restore the stack
18. In order to call the set\_tod\_from\_secs() function, this function will need to allocate space on the stack for a tod\_t. As described previously, this struct can be packed to fit in 8 bytes so a pushq $0 will put a "zero" tod\_t struct on the stack and %rsp is then a pointer to it which can be copied to other registers.
19. Similarly, to call the set\_display\_bits\_from\_tod() function, one will need a packed tod\_t in a register. If the preceding set\_tod\_from\_secs() call succeeded, this packed struct can be read from memory into a register with a movq instruction. That stack space can re-used if needed.
20. Keep in mind that you will need to do error checking of the return values from the two functions: if they return non-zero values jump to a clearly marked "error" section and return a 1. If an error occurs, **don't forget to pop any values off the stack that have been pushed before returning.**

### 3.8 Manual Inspection Criteria for lcd\_update\_asm.c (30%)   GRADING

| **Weight** | **Criteria** |
| --- | --- |
| 10 | set\_tod\_from\_secs() |
|  | Clear signs of hand-crafted assembly are present. |
|  | Detailed documentation/comments are provided showing the algorithm used in the assembly |
|  | There is a clear relation of the code to the C algorithm used in lcd\_updat.c |
|  | High-level variables and registers they occupy are described. |
|  | Error checking on the input values is done with a clear "error" section/label |
|  |  |
|  | Division is used to compute quotients and remainders that are needed. |
|  | There is a clearly documented section which updates struct fields in memory |
|  | No function calls are made that would alter the stack contents |
| 10 | set\_display\_bits\_from\_tod() |
|  | Clear signs of hand-crafted assembly are present. |
|  | Detailed documentation/comments are provided showing the algorithm used in the assembly |
|  | There is a clear relation of the code to the C algorithm used in lcd\_updat.c |
|  | High-level variables and registers they occupy are described. |
|  | Error checking on the input values is done with a clear "error" section/label |
|  |  |
|  | There is a clearly documented data section setting up useful tables of bitmasks |
|  | Division is used to compute quotients and remainders that are needed. |
|  | Struct fields are unpacked from an argument register using shift operations |
|  | No function calls are made that would alter the stack contents |
| 10 | lcd\_update() |
|  | Clear signs of hand-crafted assembly are present. |
|  | Detailed documentation/comments are provided showing the algorithm used in the assembly |
|  | There is a clear relation of the code to the C algorithm used in lcd\_updat.c |
|  | High-level variables and registers they occupy are described. |
|  | Error checking on the return values is done with a clear "error" section/label |
|  |  |
|  | Memory is pushed onto the stack for local variables that must be passed by reference |
|  | Function calls to the earlier two functions are made with appropriate arguments passed |

### 3.9 Tests for LCD Clock (20%)

Both binary and shell tests can be run with make test-p1. This will make use of the lcd\_update\_asm.s file only. During development, you may run the tests with a mixture of C and assembly as described in the [Iterative Development Section](http://www-users.cs.umn.edu/~kauffman/2021/a3.html#iter-dev) but testing will be done with assembly only.

| **Weight** | **Criteria** |
| --- | --- |
| 15 | test\_lcd.c |
|  | Provides 30 tests for functions in lcd\_update.c |
|  | Compile and run using make test-p1 |
|  | 1 point per 2 tests passed |
|  | Deductions for memory problems identified by Valgrind |
| 5 | test\_lcd\_main.sh |
|  | 5 Tests of the lcd\_main which uses functions from lcd\_update.c |
|  | 1 point per test passed |
|  | Deductions for memory problems identified by Valgrind |

## 4 Problem 2: The Binary Bomb

### 4.1 Overview

The nature of this problem is similar to the previous assignment's puzzlebox: there is a program called bomb which expects certain inputs from a parameter file or typed as input. If the inputs are "correct", a phase will be "defused" earning points and allowing access to a subsequent phases. The major change is that the bomb program is in binary so must be debugged in assembly.

Below is a summary of useful information concerning the binary bomb.

**Bombs are Individual**

The bomb you will download contains subtle variations so that the solution to yours will not work on other bombs. Feel free to discuss general techniques with classmates but know that you'll need to ultimately defuse your own bomb.

**Bombs are Binary**

A small amount of C code with the main() function is included but the bulk of the code is binary which will require using gdb to debug the assembly code.

**Bombs only Run on Lab Machines**

To stay in contact with the scoring server, bombs won't run on your laptop. You'll need to work on them on lab machines.

**Bombs Take Input**

Similar to puzzlebox, create an input.txt file which will contain your answers. Run bombs with this input file. Note that if the bomb runs out of input, you can type input directly into the bomb though this may look a little funny in the debugger.

**Defusing Phases Earns Points**

As with the earlier puzzlebox, points for this problem are earned based on how many phases are completed. Each phase that is completed will automatically be logged with the scoring server

**Bomb Explosions Lose Points**

If incorrect input is entered and the bomb runs to completion, it will "explode" which causes credit to be deducted. See the scoring system for details. This can be prevented by setting breakpoints prior to the explosion sequence and restarting the bomb when those breakpoints are hit.

### 4.2 Machines on which bombs run

The binary bomb makes frequent contact with a scoring server so you can only run it on a list of prescribed machines. These comprise most of the valid CSELabs machines and are listed in the table below.

| **Machine** | **Login Address** | **Location** |
| --- | --- | --- |
| apollo | csel-apollo.cselabs.umn.edu | Machine Room |
| atlas | csel-atlas.cselabs.umn.edu | Machine Room |
| Vole | csel-vole-01.cselabs.umn.edu | [Virtual](http://vole.cse.umn.edu/) |
|  | csel-vole-02.cselabs.umn.edu |  |
|  | … |  |
|  | csel-vole-48.cselabs.umn.edu |  |
| 4-250 Lab | csel-kh4250-01.cselabs.umn.edu | Keller 4-250 |
|  | … |  |
|  | csel-kh4250-49.cselabs.umn.edu |  |
| 4-240 Lab | csel-kh4240-01.cselabs.umn.edu | Keller 4-240 |
|  | … |  |
|  | csel-kh4240-10.cselabs.umn.edu |  |
| Lind Lab | csel-lind40-01.cselabs.umn.edu | Lind Hall 40 |
|  | … |  |
|  | csel-lind40-43.cselabs.umn.edu |  |

Attempting to run a bomb on an un-authorized machine will error out immediately as in

> ./bomb

Initialization error: illegal host 'ck-laptop'.

Legal hosts are as follows:

csel-apollo

csel-atlas

csel-vole-01

csel-vole-02

...

### 4.3 Bomb Download and Setup

* Download your bomb from the following web address
  + <http://apollo.cselabs.umn.edu:15213/>
* **This site must be accessed from UMN machines** as it is behind the campus firewall. Using a browser on [Vole](http://vole.cse.umn.edu/) is the easiest way get a bomb onto your CSELabs account (and will let you tell friends "I've used a browser inside a browser.").
* Enter your UMN information in the required fields. If you fail to enter your official information, you may not get a grade for this portion.
* The bomb will download as a .tar file, an archive format. On Unix machines, extract the contents using the command untar as in
* > ls
* bomb10.tar
* > tar xfv bomb10.tar
* bomb10/README
* bomb10/bomb.c
* bomb10/bomb
* > ls
* bomb10.tar bomb10/
* > cd bomb10
* > ls
* bomb\* bomb.c README
* The resulting bomb is unique for the downloader and the owner is in the README and logged on the download server.
* The file bomb (sometimes listed with a \* to indicate it is executable) is a compiled binary so employ your assembly gdb skills to cracking it.
* Create a file input.txt. The bomb can be run with it as in
* > ./bomb input.txt

but you'll likely want to do this in gdb to avoid exploding the bomb.

* Unlike previous puzzles, if input.txt runs out of input, the bomb will prompt for you to type input. This can be a way to explore ahead a little bit in the bomb after solving a phase.

### 4.4 Scoring and Scoreboard (50%)   GRADING

Scoring is done according to the following table.

| **Pts** | **Phase** |
| --- | --- |
| 8 | Phase 1 |
| 8 | Phase 2 |
| 9 | Phase 3 |
| 9 | Phase 4 |
| 8 | Phase 5 |
| 8 | Phase 6 |
| 50 | Total |

**Explosion Penalty**: 0.5 points are deducted for each explosion up to 20 explosions (maximum -10 points).

On successfully defusing stages, the bomb will contact a server which tracks scores by number. The scoreboard is here:

* <http://apollo.cselabs.umn.edu:15213/scoreboard>

You'll need to know your bomb number to see your score but can also see the scores of others.

#### Examples of Scoring

| **Phases** |  |  | **Final** |  |
| --- | --- | --- | --- | --- |
| **Defused** | **Explosions** | **Computation** | **Score** | **Notes** |
| 6 | 1 | 50 - floor(0.5\*1) | 50 | 1 explosion for free |
| 6 | 4 | 50 - floor(0.5\*4) | 48 |  |
| 6 | 10 | 50 - floor(0.5\*10) | 40 |  |
| 5 | 7 | 42 - floor(0.5\*7) | 39 | Round down for penalty |
| 4 | 4 | 34 - floor(0.5\*4) | 32 |  |
| 1 | 0 | 8 - floor(0.5\*0) | 8 |  |
| 0 | 20 | 0 - (floor(0.5\*20) | -10 |  |
| 0 | 30 | 0 - (floor(0.5\*20) | -10 | Max 20 explosions counted |

#### Getting Credit for the Problem

* Ensure that the score listed on the [Scoreboard site](http://www-users.cs.umn.edu/~kauffman/2021/a3.html#scoreboard) reflects your progress.
* Ensure your input.txt along with your bombNN/ directory are in your project directory with the rest of your code.

### 4.5 Advice

* If you accidentally run the bomb from the command line, you can kill it with the Unix interrupt key sequence Ctrl-c (hold control, press C key).
* > ./bomb
* Welcome to my fiendish little bomb. You have 6 phases with
* which to blow yourself up. Have a nice day!
* ^C
* So you think you can stop the bomb with ctrl-c, do you?
* Well...OK. :-)
* >
* Most of the time you should run the bomb in gdb as in
* > gdb ./bomb

Refer to the [Quick Guide to GDB](http://www-users.cs.umn.edu/~kauffman/2021/gdb) if you have forgotten how to use gdb and pay particular attention to the sections on debugging assembly.

* Figure out what the explosion routine is called and **always set a breakpoint there**. This will allow you to stop the bomb
* Make use of other tools to analyze the binary bomb aside from the debugger. Some of these are described at the end of the [Quick Guide to GDB](http://www-users.cs.umn.edu/~kauffman/2021/gdb.html#binary-tools). They will allow you to search for "interesting" data in the executable bomb. The author of the bomb is encoded in the binary as a string somewhere which may be relevant to inputs for some phases.
* Feel free to do some internet research. The "bomb lab" assignment has a long history and there are some useful guides out there that can help you through rough patches. Keep in mind that your bomb will differ but the techniques to defuse it may be similar to others.