Instructions:

- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page formula sheet. No calculators or other computing devices may be used. The use of any such device will be interpreted as an indication that you are finished with the test and your test form will be collected immediately.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 8 questions, some with multiple parts, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- Note that either failing to return this test, or discussing its content with a student who has not taken it is a violation of the Honor Code.

Do not start the test until instructed to do so!

Name Solution
printed

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

signed

1. [12 points] The hardware-level representation of an unsigned integer value X is $b_{31}b_{30}b_{29}\ldots b_2b_1b_0$. Suppose that X is a multiple of 4, but not a multiple of 8. What does that imply about the bits that represent X? Justify your answer carefully.

Recall that positional notation is used, so:

$$X = b_{31}2^{31} + b_{30}2^{30} + b_{29}2^{29} + \cdots + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0$$

If X is a multiple of 4, then both b_0 and b_1 must be 0, since we must be able to factor 4 out of X.

But, if X is not a multiple of 8, then b_2 must be 1, since otherwise we'd be able to factor 8 out of X.

---

2. [12 points] Two different executables, both named `runme`, exist in different directories on a Linux system. One executable is a text editor and the other is a game. An unsuspecting user types the command "`runme`" at the command prompt in a bash shell, and the game runs (rather than the text editor).

What does this tell you? Be precise and explain fully.

The shell searches the directories in the path variable, in the order they occur in the path, and runs the first executable file (with the correct name) that is found.

So, if the game is started, it must be the case that:

- **The directory containing the game executable runme is stored in a directory that is in the path.**
- **That directory occurs before the directory containing the editor executable, or the editor's directory is not in the path at all.**

Notes:

- **The question of whether the user is operating in the same directory as the executable for the game is entirely irrelevant… unless the current working directory is in the path, the shell will not look there.**

3. [12 points] Formulate a regular expression that could be used with grep to:

a) Search a text file for all strings that contain the character 'm', followed by one or two vowels, followed by the letter 't'. Matching strings would include "meet", "meat", "meeting", "remain", etc. The search should be case-sensitive.

```
m[aeiou]{1,2}t
```

b) Search a text file for all strings that are "words" in the grep sense, and consist of a sequence of digits representing an integer that is greater than or equal to 1234.

```
\<                      make it a "word"
0*                      allow leading zeros
(123[4-9]         |     1234 - 1239
 12[4-9][0-9]     |     1240 - 1299
 1[3-9][0-9]{2}   |     1300 - 1999
 [2-9][0-9]{3}    |     2000 - 9999
 [1-9][0-9]{4,})        >= 10000
>/
```

4. [12 points] I want a `void` C function that will enable me to complete the following code correctly:

```
int N = 5;
int A[5] = {1, 2, 3, 4, 5};
int Y = 3;

// My call to your void function will go here.

// Now, each element of A[] that is smaller than the original value of Y
// should be set to the original value of Y, and Y should equal the sum
// of the original values in the array A[].
//
// So, for the values given above, Y should now equal 15 and the first
// three elements in A[] should each equal 3.
```

The function can have any interface you think is appropriate, so long as its return type is `void` and calling it yields the desired effect.

Write your implementation of the function here:

Note: the function must be able to modify the caller's variable Y, so we must pass the function a pointer to Y.

```
void f(int L[], int Sz, int* pY) {

   if ( Sz <= 0 ) return;              // nothing to do

   int oldY = *pY;                     // copy original value of Y

   int Sum = 0;                        // local variable for sum

   for (int idx = 0; idx < Sz; idx++) {

      Sum = Sum + L[idx];              // accumulate sum of old values

      if ( L[idx] < oldY )             // replace list element, if smaller
         L[idx] = oldY;
   }

   *pY = Sum;                          // reset Y in caller to equal sum
}
```

Write the call I would need to insert into my code here: `f(A, N, &Y);`

5. [12 points] Consider the following C function:

```
////////////////////////////////////////////////// shiftLeftOne()
//  Shifts the elements of the given array left by one position,
//  "losing" the first element and not modifying the final one.
//  Pre:  A[] is of dimension Sz,
//        A[0:Sz-1] have been initialized
//  Post: for i = 0..Sz-1, A[i] = OLD A[i+1],
//        A[Sz-1] is unmodified
// void shiftLeftOne(int A[], int Sz) {

   int idx = 0;
   do {

      A[idx] = A[idx + 1];

      idx++;                        // correction announced at test
   } while ( idx < Sz - 1 );
}
```

The implementer of this function has assumed something that is not guaranteed by the preconditions. What? And, assuming all preconditions are satisfied, when will this error affect the results, and how?

The implementation uses a do-while loop, which means the body of the loop will be executed at least one time.

But, if the size of the array is 1, which is consistent with the stated preconditions, an attempt will be made to access a value that is one-past-the-end of the array.

This may result in a run-time error, or not, depending on the ownership of that memory location.

Since the access is only to read the value, no corruption of memory will occur.

6. Recall the discussion of the management of stack frames on an x86-32 Linux system.

a) [4 points] What is the role of the stack frame pointer? That is, what is it supposed to point to?

The stack frame pointer (which is register %ebp) is supposed to point to the first value in the stack frame for the currently-executing function.

Notes:

- **The stack frame pointer is stored in the register %ebp, and must not be confused with the stack pointer, which is stored in the register %esp. They are different and play different roles.**

b) [4 points] Why is it sometimes necessary to save the value of the stack frame pointer?

When a function call occurs, %ebp must be reset to point to the beginning of the stack frame for the called function; if we do not save the old value of %ebp, there will be no way to reset %ebp to the beginning of the caller's frame when the called function returns.

Notes:

- **Simplying saying that the old value must be backed up in case it is needed later doesn't really say anything about the matter at hand… that's ALWAYS the reason for backing up a value.**

c) [6 points] An x86-32 assembly procedure typically begins with the following two statements. Explain what each statement does.

```
pushl   %ebp        # 1
movl    %esp, %ebp  # 2
```

The first statement stores a copy of the frame pointer on the stack (decrements %esp by 4; writes current value in %ebp to target of %esp).

The second statement copies the value of the stack pointer %esp into the frame pointer %ebp, setting the frame pointer to point to the beginning of a new stack frame (and to the backed-up old value of the frame pointer).

7. [14 points] The following x86-32 assembly code was obtained by compiling a C function. Analyze the code and write a C function that is logically equivalent to the given assembly code. If the function takes any parameters, name them `Q1`, `Q2`, and so forth, in the order they occur on the stack. If the function uses any local variables, name them `A1`, `A2`, and so forth, in the order they occur on the stack. You may assume that any parameters and local variables are of type `unsigned int`.

```
        .file   "q7.c"
        .text
.globl f
        .type f, @function
f:
        pushl   %ebp                # these are stack frame setup code
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $0, -4(%ebp)        # assigns 0 to auto local variable A1
        movl    $1, -8(%ebp)        # assigns 1 to auto local variable A2
        jmp     .L2                 # jumps to loop test; indicates while loop
.L3:
        movl    -4(%ebp), %eax      # copies A1 into eax
        sall    $2, %eax            # multiplies value in eax by 2^2
        addl    -8(%ebp), %eax      # adds A2 to value in eax
        movl    %eax, -4(%ebp)      # writes eax to A1; A1 = 4*A1 + A2
        addl    $1, -8(%ebp)        # adds 1 to A2; A2++
.L2:
        movl    -8(%ebp), %eax      # copies A2 into eax
        cmpl    8(%ebp), %eax       # compares A2 to parameter Q1
        jbe     .L3                 # repeat loop body if A2 <= Q1 (unsigned)
        movl    -4(%ebp), %eax      # set return value to A1
        leave                       # return to caller
        ret
        .size    f, .-f
        .ident   "GCC: (Ubuntu/Linaro 4.5.2-8ubuntu4) 4.5.2"
        .section  .note.GNU-stack,"",@progbits
```

```
uint32_t f(uint32_t Q1) {

   uint32_t A1 = 0;
   uint32_t A2 = 1;

   while ( A2 <= Q1 ) {

      A1 = 4*A1 + A2;
      A2++;
   }

   return A1;
}
```

[This page is intentionally blank to provide more working space for question 7, if you need it.]

Analysis:

- **The code refers to three values on the stack, at addresses -4(%ebp), -8(%ebp) and 8(%ebp).**
- **The first two references are within the current stack frame (negative offsets from %ebp), so they are to auto local variables, which we'll call A1 and A2 respectively.**
- **The third is to a reference above the current stack frame, so it's to a parameter, which we'll call Q1.**
- **The conditional jump instruction is jbe, which indicates that unsigned values are being used; I used uint32_t in my solution, but you could also use unsigned int.**
- **The fact that the jbe branches to a preceding label indicates a loop, not an if or if..else.**
- **The fact that there is a jmp to the loop test code before the loop body is reached indicates a while loop rather than a do..while.**

8. Consider the following short x86-32 assembly fragment:

```
addl    $10, -8(%ebp)
imull   -8(%ebp), %eax
```

a) [4 points] Describe the difference between a *register* and *RAM*.

Registers are implemented (hardware) within the CPU, there are only a few of them, and they are much faster to access than anything outside the CPU.

RAM (random access memory) is implemented (hardware) outside the CPU, there is lots of it, and it is much slower to access than anything within the CPU.

Notes:

- **It is not correct to say that either is used for temporary vs permanent storage; register and RAM storage is ephemeral.**
- **It is not correct to say that only values in registers (or in RAM) can be operated upon; in the x86 architecture we have direct-memory operations (see imull and addl above).**

b) [8 points] With respect to the assembly fragment given above, which operands refer to registers and which refer to locations in RAM?

Register operands: %eax

Memory operands: -8(%ebp)

Notes:

- **The question was about operands; in the two given instructions there are three operands: %eax, $20 and -8(%ebp), twice.**