Name:____ Email:__________________@ncsu.edu

ECE 461/561, Spring 2023: Test 1 Solutions

This quiz is closed-computer, closed-book, closed-notes. You may use one 8.5" x 11" sheet of paper with anything you want written or printed on its two sides.

Assume the code is built using MDK-ARM (AC6 compiler, armlink linker, all settings for maximum optimization for time) for the Kinetis KL25Z128 MCU used on the FRDM-KL25Z evaluation board and the core clock frequency is fixed at 48 MHz. All questions are equally weighted. **4 points per question. 100% = 96 pts for ECE 461, 104 pts for ECE 561**

Please read and sign this statement: I have not received assistance from anyone nor assisted others while taking this test. I have also notified the test proctor of any violations of the above conditions.

Signature ____

#	Notes	Score
1		
2		
3		
4		
5		
6		
7		
8		
9	Extra Credit	

10		
11		
12		
13		
14		
15		
16		
17		
18		
19		

20		
21		
22		
23		
24		
25	561 Only	
26		
27	561 Only	
Total		

Possibly Useful Reference Information

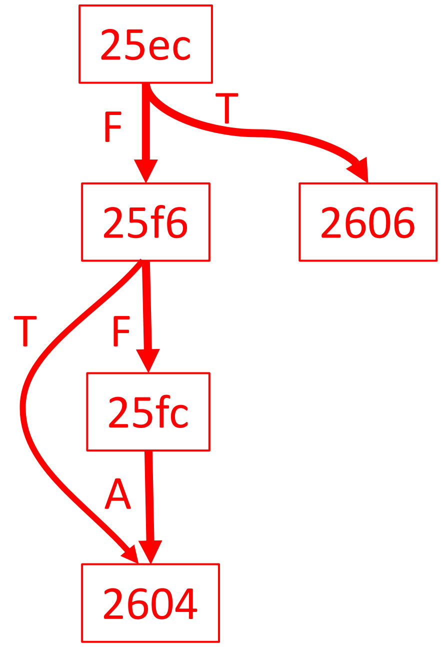
Condition Flag	Meaning if 0	Meaning if 1
Z Zero	Result not zero 0x0000	Result was zero
V Overflow	Result did not overflow	Result overflowed
N Negative	Not negative, MS bit of result is 0	Negative, MS bit of result is 1
C Cary	No carry out or borrow in	Carry out or borrow in occurred

Mnemonic Extension	Meaning	Condition Flags		Mnemonic Extension	Meaning	Condition Flags
EQ	Equal	Z == 1		VC	No overflow	V == 0
NE	Not equal	Z == 0		HI	Unsigned higher	C == 1 and Z == 0
CS	Carry set	C == 1		LS	Unsigned lower or same	C == 0 or Z == 1
CC	Carry clear	C == 0		GE	Signed greater than or equal	N == V
MI	Minus, negative	N == 1		LT	Signed less than	N != V
PL	Plus, positive or zero	N == 0		GT	Signed greater than	Z == 0 and N == V
VS	Overflow	V == 1		LE	Signed less than or equal	Z == 1 or N != V

Examining Object Code

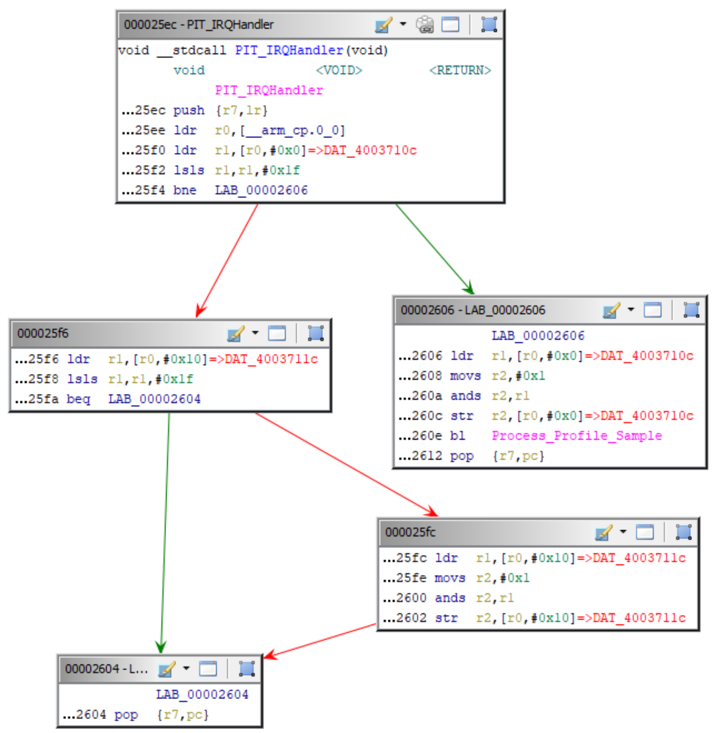
Consider the following source code.

```
#define PIT_TFLG_TIF_MASK (0x0001)
void PIT_IRQHandler() {
   // Which channel triggered the interrupt?
   if (PIT->CHANNEL[0].TFLG & PIT_TFLG_TIF_MASK) {
      // clear status flag timer channel 0
      PIT->CHANNEL[0].TFLG &= PIT_TFLG_TIF_MASK;
      // Do ISR work
      Process_Profile_Sample();
   } else if (PIT->CHANNEL[1].TFLG & PIT_TFLG_TIF_MASK) {
      // clear status flag for timer channel 1
      PIT->CHANNEL[1].TFLG &= PIT_TFLG_TIF_MASK;
   }
}
```



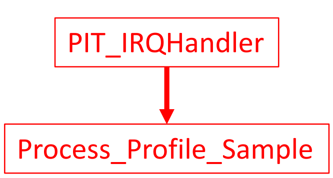
Compiling the code resulted in the disassembly listing below.

```
000025ec 80 b5       push   {r7,lr}
000025ee 09 48       ldr    r0,[pc,#36] ; @0x00002614
000025f0 01 68       ldr    r1,[r0,#0x0]
000025f2 c9 07       lsls   r1,r1,#0x1f
000025f4 07 d1       bne    0x00002606
000025f6 01 69       ldr    r1,[r0,#0x10]
000025f8 c9 07       lsls   r1,r1,#0x1f
000025fa 03 d0       beq    0x00002604
000025fc 01 69       ldr    r1,[r0,#0x10]
000025fe 01 22       movs   r2,#0x1
00002600 0a 40       ands   r2,r1
00002602 02 61       str    r2,[r0,#0x10]
00002604 80 bd       pop    {r7,pc}
00002606 01 68       ldr    r1,[r0,#0x0]
00002608 01 22       movs   r2,#0x1
0000260a 0a 40       ands   r2,r1
0000260c 02 60       str    r2,[r0,#0x0]
0000260e 00 f0 5f f8 bl     Process_Profile_Sample
00002612 80 bd       pop    {r7,pc}
00002614 0c 71 03 40
```



Use the object code listing to answer the following questions.

1. Identify each basic block of code. Draw a rectangle around each basic block in the object code listing above.
2. Draw the code's control flow graph next to the object code listing above.
   - Represent each basic block by a rectangle labeled with the last four digits of its starting address.
   - Use arrows to show the control flow edges (arcs) between basic blocks. Label each control flow edge with A, T, or F to indicate under which condition the edge is taken (always, condition true, or condition false). Do not include control flow edges for subroutine returns.
3. Draw the function call graph based on the object code.



4. Write the address for each prolog instruction.
   0x25ec
5. How much stack space (in bytes) does this function use, and what is it used for? Do not consider any subroutines it may call.
   Eight bytes of stack space are used to save r7 and lr (return address).
6. How many arguments does the function have, and which registers are used to pass them?
   There are no arguments.

7. List the address for each epilog instruction.
   0x2604 and 0x2612

8. What is the value (hexadecimal) of the word in memory starting at address 0x00002614, and what does it represent?
   The word is 0x4003710c, and it is the address of the PIT Channel 0 TFLG register (PIT_TFLG0)

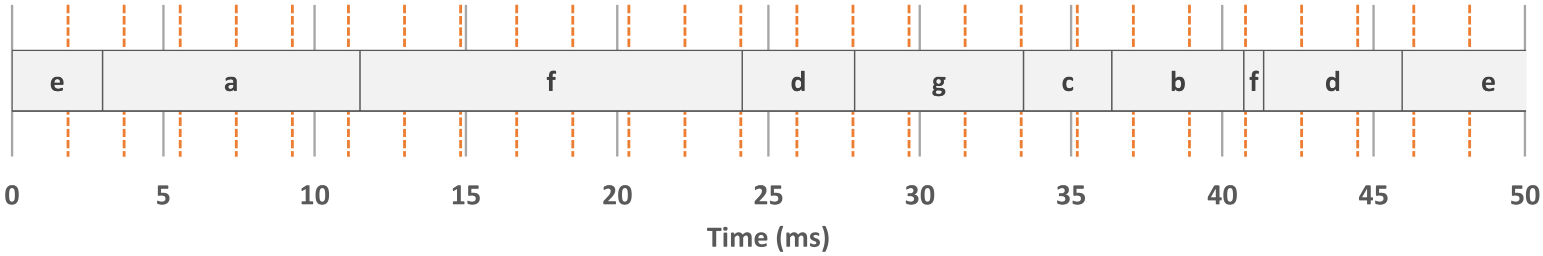
9. **Extra Credit:** How do the instructions "`lsls r1,r1,#0x1f` / `bne 0x00002606`" starting at 0x000025f4 implement the source "`if (PIT->CHANNEL[0].TFLG & PIT_TFLG_TIF_MASK)`"? Hint: lsls is "logical shift left, setting condition flags."
   Full credit: shift left 31 positions to zero out all bits but bit 0 (TIF flag), Z flag set if result is zero, bne branch is taken if result is zero (TIF wasn't set).
   3: minor error
   2: better understanding….
   1: bne checking Z flag or other minimal understanding

   lsls (logical shift left) shifts the register left by a certain number of bits (shifting in zeroes to the LS bit) and sets the condition code flags based on the final value of the destination register. So shifting r1 left by 0x1f = 31 positions will move bit 0 (the TFLAG register's TIF bit) to bit 31, and zero out bits 0 through 30.
   The **lsls** instruction (like other instructions ending in **s**) sets the Z flag to one if the result is zero. Otherwise it clears the Z flag to zero.
   In this code, a Z flag == 1 indicates the TIF bit was zero. The conditional branch **bne** branches if the **ne** (not equal) condition is true, represented by Z being 0. So the branch is taken if Z is 0, meaning the TIF bit was 1.

10. When the instruction at address 0x000025ee executes, where is the function's return address located? If in a register, specify the register name. If on the stack, specify the memory address relative to the stack pointer (e.g. SP+12).
    In memory at [SP+4] and in the link register **lr**

Profiling

The following diagram shows which function (a through g) a program is executing as time passes.



11. Complete the profile table below assuming sampling occurs at multiples of 5 ms (on the solid lines).

Assuming first sample is at T=0, last at T=50

Function Name	Sample count	Function Name	Sample count
(Total Samples)	11	**d**	2
a	2	**e**	2
b	1	**f**	2
c	1	**g**	1

Assuming first sample is at T=5, last at T=50

Function Name	Sample count	Function Name	Sample count
(Total Samples)	10	**d**	2
a	2	**e**	1
b	1	**f**	2
c	1	**g**	1

12. Use the results above to complete the profile table below, sorting functions with decreasing sample count.

Assuming first sample is at T=0, last at T=50

Function Name	Sample count	Function Name	Sample count
a/d/e/f	2	b/c/g	1
a/d/e/f	2	b/c/g	1
a/d/e/f	2	b/c/g	1
a/d/e/f	2		

Assuming first sample is at T=5, last at T=50

Function Name	Sample count	Function Name	Sample count
a/d/f	2	b/c/e/g	1
a/d/f	2	b/c/e/g	1
a/d/f	2	b/c/e/g	1
a/d/f	2		

13. How much could you speed up the program by optimizing only the top function?

    Assuming first sample is at T=0, last at T=50: Up to 2 samples could be removed, resulting in the code taking 9 instead of 11 samples. The speed-up is 2/11.

    Assuming first sample is at T=5, last at T=50: Up to 2 samples could be removed, resulting in the code taking 8 instead of 10 samples. The speed-up is 2/10.

14. Which functions (if any) were executed but were missed by sampling? None.
15. Complete the profile table assuming sampling occurs at multiples of 1.853 ms (on the dashed lines).

Assuming first sample is at T=0, last at T=50

Function Name	Sample count	Function Name	Sample count
(Total Samples)	28	**d**	4
a	5	**e**	5
b	2	**f**	8
c	1	**g**	3

Assuming first sample is at T=0, last at T=50-T_{sample}

Function Name	Sample count	Function Name	Sample count
(Total Samples)	27	**d**	4
a	5	**e**	4
b	2	**f**	8
c	1	**g**	3

Assuming first sample is at T=T_{sample}, last at T=50

Function Name	Sample count	Function Name	Sample count
(Total Samples)	27	**d**	4
a	5	**e**	4
b	2	**f**	8
c	1	**g**	3

Assuming first sample is at T=T_{sample}, last at T=50-T_{sample}

Function Name	Sample count	Function Name	Sample count
(Total Samples)	26	**d**	4
a	5	**e**	3
b	2	**f**	8
c	1	**g**	3

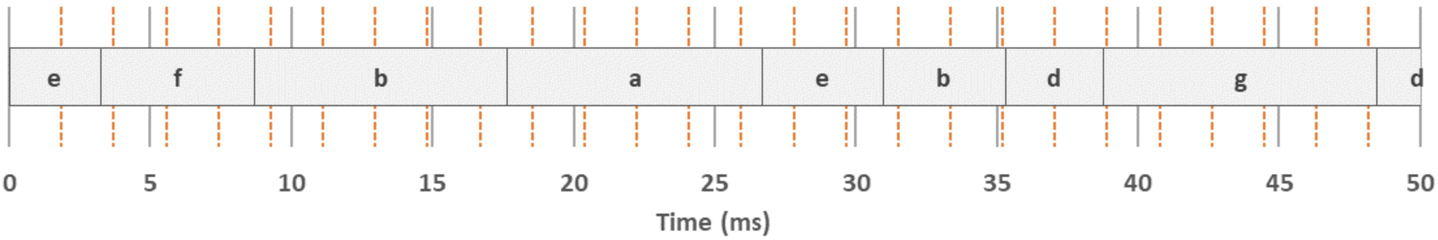
16. Which function had its fraction of total samples change the most between these two sampling periods? Be sure to include the fractions from both sampling periods (e.g. 3/7 and 9/20).

    Assuming first sample is at T=0, last at T=50: *Function f went from 2/11 to 8/26*

    Assuming first sample is at T=T_{sample}, last at T=50: *Function f went from 2/10 to 8/25*

Profiling (alternate version)

The following diagram shows which function (a through g) a program is executing as time passes.



1. Complete the profile table below assuming sampling occurs at multiples of 5 ms (on the solid lines).
   ½ pt per box

Assuming first sample is at T=0, last at T=50

Function Name	Sample count	Function Name	Sample count
(Total Samples)	11	**d**	1
a	2	**e**	2
b	3	**f**	1
c	0	**g**	2

Assuming first sample is at T=5, last at T=50

Function Name	Sample count	Function Name	Sample count
(Total Samples)	10	**d**	1
a	2	**e**	1
b	3	**f**	1
c	0	**g**	2

2. Use the results above to complete the profile table below, sorting functions with decreasing sample count.
   ½ pt per box

Assuming first sample is at T=0, last at T=50

Function Name	Sample count	Function Name	Sample count
b	*3*	d/f	1
a/e/g	*2*	d/f	1
a/e/g	*2*	c	0
a/e/g	*2*		

Assuming first sample is at T=5, last at T=50

Function Name	Sample count	Function Name	Sample count
b	*3*	d/e/f	1
a/g	*2*	d/e/f	1
a/g	*2*	c	0
d/e/f	1		

3. How much could you speed up the program by optimizing only the top function?
   Assuming first sample is at T=0, last at T=50: *Up to 3 samples could be removed, resulting in the code taking 8 instead of 11 samples. The speed-up is 3/11.*
   Assuming first sample is at T=5, last at T=50: *Up to 3 samples could be removed, resulting in the code taking 7 instead of 10 samples. The speed-up is 3/10.*

4. Which functions (if any) were executed but were missed by sampling? Function d was missed.

5. Complete the profile table assuming sampling occurs at multiples of 1.853 ms (on the dashed lines).

Assuming first sample is at T=0, last at T=50

Function Name	Sample count	Function Name	Sample count
(Total Samples)	28	**d**	2
a	5	**e**	4
b	8	**f**	3
c	0	**g**	6

Assuming first sample is at T=0, last at T=50-T_{sample}

Function Name	Sample count	Function Name	Sample count
(Total Samples)	27	**d**	1
a	5	**e**	4
b	8	**f**	3
c	0	**g**	6

Assuming first sample is at T=T_{sample}, last at T=50

Function Name	Sample count	Function Name	Sample count
(Total Samples)	27	**d**	2
a	5	**e**	3
b	8	**f**	3
c	0	**g**	6

Assuming first sample is at T=T_{sample}, last at T=50-T_{sample}

Function Name	Sample count	Function Name	Sample count
(Total Samples)	26	**d**	1
a	5	**e**	3
b	8	**f**	3
c	0	**g**	6

6. Which function had its fraction of total samples change the most between these two sampling periods? Be sure to include the fractions from both sampling periods (e.g. 3/7 and 9/20).
   Assuming first sample is at T=0, last at T=50: *Function b went from 3/11 to 8/26*
   Assuming first sample is at T=T_{sample}, last at T=50: *Function b went from 3/10 to 8/25*

Analysis Without a Profile

Consider the following source code. The profiler uses it to create a list of region numbers called SortedRegions[], which is sorted from most to least frequent. **Assume NumProfileRegions is 200.**

```
1   volatile int RegionCount[NumProfileRegions];
2   int SortedRegions[NumProfileRegions];
3
4   void Sort_Profile_Regions(void) {
5       unsigned int i, j, temp;
6       // Copy unsorted region numbers
7       for (i = 0; i < NumProfileRegions; i++) {                              A C b
8           SortedRegions[i] = i;                                              W
9       }                                                                      B
10      // Sort those region numbers
11      for (i = 0; i < NumProfileRegions; i++) {                              A C b
12          for (j = i + 1; j < NumProfileRegions; j++) {                      A C b
13              if (RegionCount[SortedRegions[i]] < RegionCount[SortedRegions[j]]) {   R R C b?
14                  temp = SortedRegions[i];                                   R
15                  SortedRegions[i] = SortedRegions[j];                       R W
16                  SortedRegions[j] = temp;                                   W
17              }                                                              B
18          }                                                                  B
19      }                                                                      B
20  }                                                                          B
21
```

17. How many times do you expect the loop starting at line 7 to execute?

    NumProfileRegions times = 200 times.

18. How many times do you expect the loop starting at line 11 to execute?

    NumProfileRegions times = 200 times.

19. How many times do you expect the loop starting at line 12 to execute?

    (199 + 198 … + 101 + 100 + 99 … + 2 + 1) times

    First time (i=0) through loop 11: 199 times,

    Second time (i=1) through loop 11: 198 times

    …

    Second-to-last time (i=198) through loop 11: 1 time,

    Last time (i=199) through loop 11: 0 times

    $$n = \sum_{i=1}^{199} i = \left(200 * \frac{200-1}{2}\right) = 200 * \frac{199}{2} = 19{,}900$$

    NumProfileRegions * (NumProfileRegions – 1) / 2 times

20. What is the range of times which you expect code at lines 14 to 16 to execute? Include both the minimum and maximum values, and explain what input data triggers each extreme.

    Minimum

    If the list is already sorted, then lines 14-16 will never execute, so 0 is the minimum.

    2 pts: 0 times and list already sorted

    Maximum

    Bound the worst case. If lines 14-16 execute each time through the loop, they will execute 19,900 times. This happens if the list is initially reverse-sorted.

    2 pts: 19,900 times and reverse-sorted list

    1.5 pts: same as 19 and reverse-sorted list

21. **How many** load register (**ldr**) instructions do you expect to execute each time that line 13 runs? **Which address** does each instruction load from (e.g. r4+8, r5+r2)? Assume these registers have already been loaded with the following contents: r0: &RegionCount, r1: &SortedRegions, r2: i, r3: j.
Four ldr instructions. OK to use , instead of +
- load from [SortedRegions[i]]: rni
  - ldr r4, [r1 + r2*4].
- load from RegionCount[SortedRegions[i]]: rc[rni]
  - ldr r5, [r0 + r4*4] *This must be register loaded in previous instruction, this register doesn't matter*
- load from [SortedRegions[j]]: rnj
  - ldr r6, [r1 + r3*4]
- load from RegionCount[SortedRegions[j]]: rc[rnj]
  - ldr r7, [r0 + r6*4] *This must be register loaded in previous instruction, , this register doesn't matter*

Points off:

- Number of ldr instructions != 4: **-1 pt**
- Offset not multiplied by 4 bytes/word: **-1 pt** (max once)
- Incorrect registers per color codes above: **-1 pt** (max once)

22. Which line of source code will dominate (account for the most) execution time? Explain why.
2 pts: Line 13.
2 pts: That line of code reads from memory four times and performs a comparison. It is executed each time through the most deeply-nested loop.

23. Which line of source code will account for the **second most** execution time? Explain why.
2pts: Line 12.
1 pt: It executes each time through the most deeply-nested loop.
1 pt: On all iterations after the first, it increments j, compares j with the limit, and decides whether to exit the loop. This takes three instructions. The other code in the loop body (lines 14, 15, 16) is is shorter than line 12, since SortedRegions[i] and SortedRegions[j] were already loaded into registers by line 13. Furthermore, lines 14, 15, 16 are not executed on every loop iteration unless the list is reverse-sorted.

24. The array RegionCount is declared as **volatile**. How does this affect the assembly code generated for lines 13 through 16?
2 pts: RegionCount is volatile, so nothing read from RegionCount is allowed to be reused, so the code will be slower. There will always be memory loads from RegionCount for line 13: one from RegionCount[SortedRegions[i]] and one from RegionCount[SortedRegions[j]].

[However, if RegionCount were not volatile, the compiler could optimize by reusing previously read values. The variable i usually stays the same from execution of line 13 to the next, only changing with each iteration of the line 11 for loop. In line 13, this means that SortedRegions[i] would be the same as the previous iteration of the line 12 loop, unless the previous iteration swapped it with SortedRegions[j]. If SortedRegions[i] was the same as the previous iteration, then RegionCount[SortedRegions[i]] would be the same as the previous iteration and could be reused, eliminating the need to reload it.]

2 pts: There is no impact on the code for lines 14-16, as they don't access RegionCount.

25. **ECE 561 Only**: Revise lines 13-16 of the code to improve performance by removing the effect from the previous question. Assume that profiling is disabled when Sort_Profile_Regions() runs, so RegionCount will not change then. Write your code here:

4 pts:

The index SortedRegions[i] stays the same each time through the loop at line 12 (though it changes with each time through the loop at line 11). So RegionCount[SortedRegions[i]] can be reused after the first loop iteration.

```
unsigned int rc_sr_i;
// Sort those region numbers
    for (i = 0; i < NumProfileRegions; i++) {
        int rc_sr_i = RegionCount[SortedRegions[i]];
        for (j = i + 1; j < NumProfileRegions; j++) {
            if (rc_sr_i < RegionCount[SortedRegions[j]]) {
                temp = SortedRegions[i];
                SortedRegions[i] = SortedRegions[j];
                SortedRegions[j] = temp;
                rc_sr_i = temp;
            }
        }
    }
```

Other optimizations are possible but need to be evaluated in detail case by case.

General Optimization

26. Write two small loops in C code such that applying loop unrolling (e.g. by the compiler) would speed up the first loop significantly, but the second loop only minimally. The loops can do anything that you want.

4 pts. Ratio of second loop body time divided by second loop increment+test+branch time should be larger than ratio of first loop body time divided by first loop increment+test+branch time

Example code:

```
int a, i=0;
float f=0;

for (i = 0; i < N; i++) {
   a += i; // loop body is very fast compared to test and branch
}
for (i = 0; i < N; i++) {
   f += (0.317*i)*a; // loop body is much slower than loop test and branch
}
```

27. **ECE 561 Only:** Some numerical approximations use range reduction, even though it takes additional time. Give an example and explain why range reduction is used.

2 pts: Explanation: Range reduction allows **reuse of a more accurate portion of an approximation** because of **duplication** from **symmetry** or **periodicity**. Range reduction requires processing the input value (range) to fit within the approximation's preferred range.

2 pts: For example, sin(2000.73*π) = sin(0.73*π), as sin is periodic, repeating with period 2π. So range reduction in this case is converting 2000.73*π to 0.73*π with a remainder or modulo operation: reduced_range = range % (2*π). (Actually we need to use fmod instead of %, which is the integer modulus operator.)