ECE 785
TOPICS IN ADVANCED COMPUTER DESIGN
SPRING 2019 FINAL EXAM SOLUTIONS

Closed book, closed notes, one 8.5”x11” page of notes allowed. Calculator allowed. You are not permitted to have a computer or other electronic assistance. Show your work for each problem for full credit. Assume that C code is compiled for the ARM Cortex-A8 architecture and the TI AM335x processor on the Beaglebone Black Wireles using gcc according to the ARM Architecture procedure calling standard (AAPCS).

Question	Maximum Score	Points Off	Score
1	5		
2	10		
3	20		
4	20		
5	20		
6	25		
Total	100		

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 _____

Consider the following source code and corresponding object code, generated with –O3 optimization. The code encrypts input data from din and stores it in dout. Encryption is performed by exclusive-oring the input data with a series of pseudo-random values. These values are generated using a linear-feedback shift register which starts with the value of the seed argument.

Source Code:

```
void LFSR(uint16_t * din, uint16_t * dout, uint16_t seed) {
   uint16_t lfsr, n;
   uint8_t bit;

   lfsr = seed;
   for (n = 0; n<N; n++) {
      // source code from http://en.wikipedia.org/wiki/Linear_feedback_shift_register
      /* taps: 16 14 13 11; characteristic polynomial: x^16 + x^14 + x^13 + x^11 + 1 */
      bit = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5) ) & 1;
      lfsr = lfsr >> 1;
      if (bit)
         lfsr |= 0x8000;
      *dout++ = *(din++) ^ lfsr;
   }
}
```

Object Code:

```
        push    {r4, r5, r6, r7}
        subs    r1, r1, #2
        add     r7, r0, #2048
        mov     r3, r2
.L38:
        lsrs    r2, r3, #3
        lsrs    r5, r3, #1
        eor     r2, r2, r3, lsr #2
        eors    r2, r2, r3
        orr     r6, r5, #32768
        eor     r2, r2, r3, lsr #5
        tst     r2, #1
        ite     eq
        moveq   r3, r5
        movne   r3, r6
        ldrh    r4, [r0], #2
        eor     r2, r3, r4
        strh    r2, [r1, #2]!       @ movhi
        cmp     r0, r7
        bne     .L38
        pop     {r4, r5, r6, r7}
        bx      lr
```

1. Circle each basic block in the object code above. Label each basic block.

BB1:

push
subs
add
mov

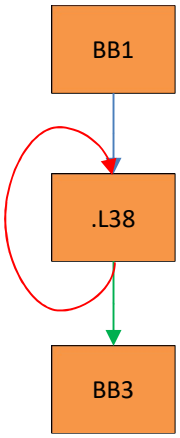
.L38:

lsrs
[all intermediate instructions]
bne .L38

BB3:

pop
bx

2. Draw the control-flow graph for the object code. Label each basic block. You do not need to include the instructions in the basic blocks.



BB1 -> .L38 -> {.L38, BB3}

3. The source code has a statement: **bit = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5)) & 1**
   a. Which object code instructions implement this source code? Mark them above with a 3.
   b. Explain how that object code works.

```
lsrs    r2, r3, #3
lsrs    r5, r3, #1
eor     r2, r2, r3, lsr #2
eors    r2, r2, r3
tst     r2, #1
eor     r2, r2, r3, lsr #5
```

The register r3 holds lfsr. The code first loads r2 with lfsr>>3 and r5 with lfsr>>1.
Then r2 (lfsr>>3) is exored with lfsr>>2 and then exored with r3 (lfsr>>0).
r2 is then exored with lfsr>>5 and the result is put in r2.
The tst r2, #1 instruction sets the condition code flags based on anding r2 with 1. The Z flag is set if the result is 0.

4. The source code has a statement: **if (bit) lfsr |= 0x8000**
   a. Which object code instructions implement this source code? Mark them above with a 4.
   b. Explain how that object code works.

```
orr     r6, r5, #32768

ite     eq
moveq   r3, r5
movne   r3, r6
```

The tst r2, #1 instruction sets the Z flag to 1 if the least-significant bit is 0. If equal (Z=1, bit = 0), move r5 into r3, else move r6 into r3. r5 holds lfsr, while r6 holds lfsr | 0x8000

5. Imagine that you profile the code using perf.
    a. Which object code instruction do you expect to dominate execution time, and why?

eor r2, r3, r4 would dominate the time because it can't execute until the load to r4 (ldrh r4, [r0], #2) completes, and this will cause occasional data cache misses.

    b. How can you change the code to minimize this problem? Explain why the change helps, and under what conditions it completely eliminates the problem.

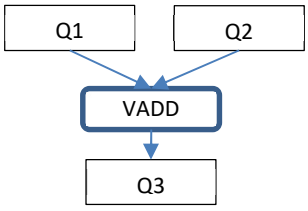
Move the ldrh instruction up to the top of the loop to hide the load latency of a data cache miss. If the load latency is shorter than the time from the load to the use (the last eor instruction), then the delay will be completely eliminated.

6. We would like to make **LFSR** run faster by unrolling its loop and then using the Advanced SIMD instructions for vectorization.
    a. Explain how to unroll the loop in the function **LFSR** by a factor of eight. Write pseudocode or C code to show your approach.

This is straightforward, just modifying the loop test, copying the loop body and adjusting the variables and offsets.

```
void LFSR(uint16_t * din, uint16_t * dout, uint16_t seed) {
    uint16_t lfsr, n;
    uint8_t bit;
    lfsr = seed;

    for (n = 0; n<N-8; n+=8) {
        // Iteration 1
        bit = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5) ) & 1;
        lfsr = lfsr >> 1;
        if (bit)
            lfsr |= 0x8000;
        *dout++ = *(din++) ^ lfsr;

        // Iteration 2
        bit = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5) ) & 1;
        lfsr = lfsr >> 1;
        if (bit)
            lfsr |= 0x8000;
        *dout++ = *(din++) ^ lfsr;

        // Iterations 3-7 go here

        // Iteration 8
        bit = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5) ) & 1;
        lfsr = lfsr >> 1;
        if (bit)
            lfsr |= 0x8000;
        *dout++ = *(din++) ^ lfsr;
    }

    // Clean-up loop (epilog) for remaining iterations
    for (; n<N; n++) {
        bit = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5) ) & 1;
    lfsr = lfsr >> 1;
    if (bit)
            lfsr |= 0x8000;
    *dout++ = *(din++) ^ lfsr;
  }
}
```

b. Vectorize the unrolled loop. Sketch out a diagram showing the dataflow with Advanced SIMD registers and instructions (please see the example below). Hint: rather than compute **bit** with so many shifts and exclusive-ors, you may wish to use the VCNT instruction, which counts the number of set bits (equal to 1) in each element.
Example dataflow notation for VADDQ Q3, Q1, Q2:



Each loop iteration depends on the previous iteration's value of lfsr; this is a loop-carried true data dependency. However, these can be calculated ahead of time.

- *Before the first iteration of the vectorized loop can execute, each lane needs its lfsr. So insert code which precalculates the value of lfsr for unrolled iteration (0, 1, 2, 3, … 7) based on seed. lfsr_0 = seed, lfsr_1 = f(lfsr_0), lfsr_2 = f(lfsr_1), … lfsr_7 = f(lfsr_6).*
- *For each subsequent vectorized loop iteration m, lane n will execute original loop iteration number (8*m) + n. Lane n needs an lfsr value which has been advanced by 8 steps.*
    - *One way is to sequentially compute it for each lane. Each lane does eight lfsr update steps on its previous value, for a total of 64 update steps.*

- *Another way is to use lane 7's lfsr value: lane 0 advances it by 1 step, lane 1 by two steps, etc. With this approach there will be 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = 36 update steps.*
- *There are other solutions possible.*

There are other approaches possible (e.g. large look-up table).