Name:_____________________________________________________ Email:____________________@ncsu.edu

# ECE 461/561, Spring 2024: Test 1

This test is closed-computer, closed-book, closed-notes. You may use two 8.5" x 11" sheets of paper with anything you want written or printed on them. Each question is worth five points. ECE 461 students do not need to answer questions 7, 12, 17 and 22 but may do so for possible extra credit. Question 23 is extra credit for all students.
**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.**

**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 _________________________________________________________

| # | 1 | 2 | 3 | 4 | 5 | 6 | 7 (561) | 8 | 9 | 10 | 11 | 12 (561) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Score | | | | | | | | | | | | |
| # | 13 | 14 | 15 | 16 | 17 (561) | 18 | 19 | 20 | 21 | 22 (561) | 23 (XC) | Total |
| Score | | | | | | | | | | | | |

## Reference Information

| Instruction Type | Instruction Mnemonic |
|---|---|
| **Move** | MOV |
| **Load, Store (byte, halfword, signed, multiple)** | LDR, LDRB, LDRH, LDRSH, LDRSB, LDM, STR, STRB, STRH, STM |
| **Add, Subtract, Multiply** | ADD, ADDS, ADCS, ADR, SUB, SUBS, SBCS, RSBS, MULS |
| **Compare** | CMP, CMN |
| **Logical** | ANDS, EORS, ORRS, BICS, MVNS, TST |
| **Shift, Rotate (logic, arith.)** | LSLS, LSRS, ASRS, RORS |
| **Stack** | PUSH, POP |
| **Branch** | B, BX, B**{cond}** See table for **{cond}** → |
| **Branch & Link (sub. call)** | BL, BLX |
| **Extend (signed, unsigned)** | SXTH, SXTB, UXTH, UXTB |
| **Reverse** | REV, REV16, REVSH |
| **Processor State** | SVC, CPSID, CPSIE, SETEND, BKPT |
| **No Operation** | NOP |
| **Hint** | SEV, WFE, WFI, YIELD |
| **Barriers** | DMB, DSB, ISB |

| **Condition Flags** | | | |
|---|---|---|---|
| Z: 1 if result is zero | V: 1 if result overflowed | N: 1 if result is negative (MS bit is 1) | C: 1 if operation had carry or borrow |

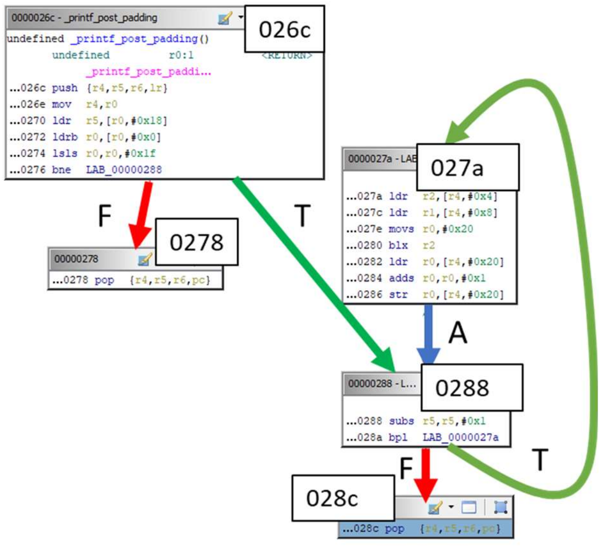
**Conditional Branch Types**

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

## Examining Object Code

Consider the following object code (obtained from Ghidra) for a function with unknown source code.

```
0000026c 70 b5   push {r4,r5,r6,lr}
0000026e 04 46   mov  r4,r0
00000270 85 69   ldr  r5,[r0,#0x18]
00000272 00 78   ldrb r0,[r0,#0x0]
00000274 c0 07   lsls r0,r0,#0x1f
00000276 07 d1   bne  00000288 ; branch
00000278 70 bd   pop  {r4,r5,r6,pc} ; return
0000027a 62 68   ldr  r2,[r4,#0x4] ; target
0000027c a1 68   ldr  r1,[r4,#0x8]
0000027e 20 20   movs r0,#0x20
00000280 90 47   blx  r2
00000282 20 6a   ldr  r0,[r4,#0x20]
00000284 40 1c   adds r0,r0,#0x1
00000286 20 62   str  r0,[r4,#0x20]
00000288 6d 1e   subs r5,r5,#0x1 ; target
0000028a f6 d5   bpl  0000027a ; branch
0000028c 70 bd   pop  {r4,r5,r6,pc} ; return
```



```
undefined _printf_post_padding()
      undefined         r0:1          <RETURN>
            _printf_post_paddi...
...026c push {r4,r5,r6,lr}
...026e mov   r4,r0
...0270 ldr   r5,[r0,#0x18]
...0272 ldrb r0,[r0,#0x0]
...0274 lsls r0,r0,#0x1f
...0276 bne   LAB_00000288
```

```
...0278 pop   {r4,r5,r6,pc}
```

```
...027a ldr   r2,[r4,#0x4]
...027c ldr   r1,[r4,#0x8]
...027e movs r0,#0x20
...0280 blx   r2
...0282 ldr   r0,[r4,#0x20]
...0284 adds r0,r0,#0x1
...0286 str   r0,[r4,#0x20]
```

```
...0288 subs r5,r5,#0x1
...028a bpl   LAB_0000027a
```

```
...028c pop   {r4,r5,r6,pc}
```

1. Draw a rectangle around each basic block in the 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 calls or returns.
3. Write the address of each prolog instruction.

   026c

4. How much stack space (in bytes) does this function use? Do not consider any subroutines it may call.

   16 bytes. 4 registers pushed * 4 bytes/register = 16 bytes

5. When the instruction at address 0x00000282 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+24).

   Return address was in link register (LR) but was pushed onto stack in prolog. Stack is full-descending, so SP points to last used value on stack. So, after this instruction executes, these four registers are at the address range SP to SP+15. Registers are pushed so the smallest register number goes to the smallest address. LR is register number 14, so it goes in the highest address: the word starting at SP+12.

6. Write the address of each epilog instruction.

   0278, 028c

7. **ECE 561 Only:** How many arguments does this function have, and which register/s are used to pass it/them?
   There is one argument, and it is passed through r0. The other possible argument registers are either not used (r3) or are overwritten without being preserved and restored (r1 by instrction at 0x27c, r2 at 0x027a).

Consider the blx r2 instruction at address 0x00000280.

8. Where does the value of r2 come from? Be as descriptive and specific as possible.

This is loaded by the instruction at 0x027a from memory at the address [r4+0x04]. r4 got its value from r0 (the argument) by the intsruction at 0x026e. So r2 comes from memory address [parameter 1 + 0x04]. Parameter 1 must be a pointer to something (structure, array, union…) with a word value located starting 4 bytes from its start.

(Because this value is used as a subroutine call address, it must be the address of a function.)

9. What does the blx r2 instruction do?

It calls a subroutine which starts at the address in r2 (see above).

Consider the use of register r5 by the instructions at addresses from 0x0000026e through 0x0000028a.

10. Where does the initial value of r5 come from? Be as descriptive and specific as possible.

This is loaded by the instruction at 0x0270 from memory at the address [r0+0x18], aka [r0+24]. So r5 comes from memory address [parameter 1 + 0x18]. Parameter 1 must be a pointer to something (structure, array, union…) with a word value located starting 0x18 (aka 24) bytes from its start.

11. What is register r5 used for in this function, and how is it related to the execution (control) flow?

r5 is used as a loop iteration counter, and controls how many times the loop starting at 0x027a executes.

**12. ECE 561 Only:** Explain why this function pushes and pops register r6. Identify which instruction in the object code makes it necessary to push and pop r6 (hint: it's not a push or pop instruction).

r6 is never used in this code, apart from the prolog and epilog. So it must be pushed and popped because it's possible for this function to call a subroutine (blx r2 at address 0x0280), and that function may need to use r6.

## Profiling

A program was run for about one second and was sampled every 1 ms, so there were **1000 samples** taken. The table below shows the number of samples for eight functions in the program.

13. Complete the table by filling in the blank cells.

| Function Name | Sample count | Order (#1 = most samples, #8 = fewest samples) | Percentage of program's time |
|---|---|---|---|
| **a** | 34 | #7 | 3.4% |
| **b** | 76 | #5 | 7.6% |
| **c** | 120 | #4 | 12% |
| **d** | 180 | #2 | 18% |
| **e** | 190 | #1 | 19% |
| **f** | 140 | #3 | 14% |
| **g** | 22 | #8 | 2.2% |
| **h** | 38 | #6 | 3.8% |

14. What percentage of the program's time do these eight functions account for?

80%

15. How many samples would the program take to run if you could cut the top (#1) function's execution time to half of its current value, but changed nothing else?

1000 – 190/2 = 1000 – 95 = 905 samples

16. How many samples would the program take to run if you could cut the second (#2) function's execution time to one tenth of its current value, but changed nothing else?

1000 – 180*(9/10) = 1000 – 162 = 838 samples

17. **ECE 561 Only:** Normally we would start by optimizing function #1. Give the **three** best reasons why we might NOT start with function #1.

a: #1 is a library function, which is likely already optimized. We wrote #2 and know it has a lot of room for optimization, so it's probably going to be easier to optimize.

b: We looked at the missing 200 samples, and at least 191 of them are from a single function X. So X dominates execution time, not #1.

c: We have already spent a lot of time optimizing function #1 doing the easy things.

d: We don't have the source code for #1.

## Analysis Without a Profile

Consider the source code below. This function is used in HW2 to rotate the arrow. It reads an array of points (ipt), rotates them around another point (center) and writes the updated coordinates to an output array of points (opt).

Assume that each floating-point math operation takes the time shown in the table. Use your own judgement based on class experience for times taken by other operations.

| Operation | Time (µs) |
|---|---|
| sinf | 75.0 us |
| cosf | 75.0 us |
| float / | 5.7 us |
| float * | 3.2 us |
| float + or – | 2.9 us |

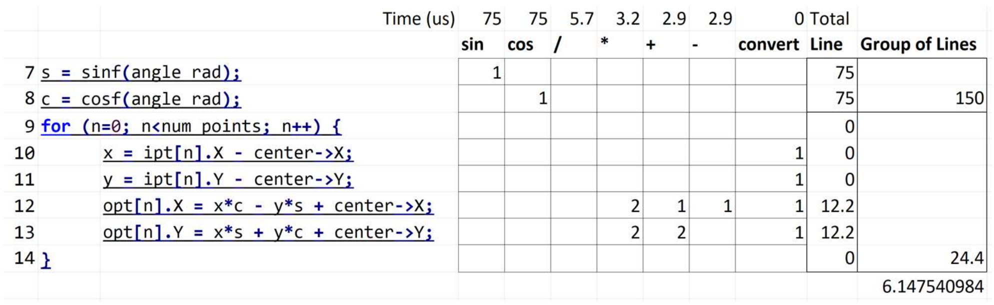
```
1  typedef struct { int32_t X, Y; } PT_T;
2
3  void Rotate_Points(PT_T * ipt, int num_points, PT_T * center, float angle_rad, PT_T * opt) {
4         float x, y;
5         int n;
6         float s, c;
7         s = sinf(angle_rad);
8         c = cosf(angle_rad);
9         for (n=0; n<num_points; n++) {
10               x = ipt[n].X - center->X;
11               y = ipt[n].Y - center->Y;
12               opt[n].X = x*c - y*s + center->X;
13               opt[n].Y = x*s + y*c + center->Y;
14        }
15 }
```

18. Why does the code use sinf and cosf instead of sin and cos? What's the difference between sinf/cosf and sin/cos that causes this?

sinf and cosf are the single-precision floating-point versions of sin and cos, which are double-precision. A single-precision operation is faster than its double-precision equivalent because the single-precision format is 32 bits long, while the double-precision format is 64 bits long.

19. How much time do you expect it will take from starting line 7 to finishing line 8?

75 + 75 = 150 us. We can neglect the time for the other operations (move angle_rad argument to r0 before calling sinf, cosf, store results to memory) since they are so much faster.



| | | Time (us) | 75 | 75 | 5.7 | 3.2 | 2.9 | 2.9 | 0 | Total | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | sin | cos | / | * | + | - | convert | Line | Group of Lines |
| 7 | s = sinf(angle_rad); | | 1 | | | | | | | 75 | |
| 8 | c = cosf(angle_rad); | | | 1 | | | | | | 75 | 150 |
| 9 | for (n=0; n<num_points; n++) { | | | | | | | | | 0 | |
| 10 | x = ipt[n].X - center->X; | | | | | | | | 1 | 0 | |
| 11 | y = ipt[n].Y - center->Y; | | | | | | | | 1 | 0 | |
| 12 | opt[n].X = x*c - y*s + center->X; | | | | | 2 | 1 | 1 | 1 | 12.2 | |
| 13 | opt[n].Y = x*s + y*c + center->Y; | | | | | 2 | 2 | | 1 | 12.2 | |
| 14 | } | | | | | | | | | 0 | 24.4 |
| | | | | | | | | | | | 6.147540984 |

20. How much time do you expect it will take from starting line 9 to finishing line 14, assuming num_points = 8?

We can neglect the time for lines 9 and 15, as they are integer operations and therefore much faster than loop body which has floating-point operations. We will also neglect time for conversion between integer and floating-point types, since the table doesn't provide a time for it.

So lines 9 through 14 will take approximately 24.4 us for each iteration of the loop. There are eight iterations, so the time taken by executing the loop eight times will be 8 * 24.4 us = 195.2 us.

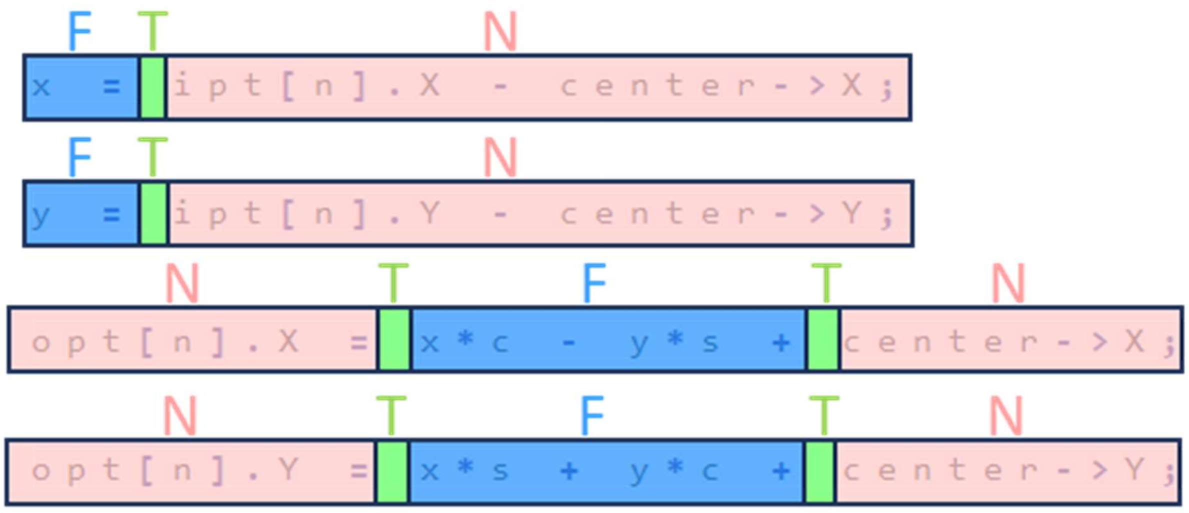
21. Which two lines of code do you expect to dominate the loop's execution time? Explain why.

Lines 12 and 13 will dominate the loop's execution time, as they have many floating-point operations (two multiplies, an add, and either an add or subtract). Lines 10 and 11 only need to do type conversions on the results of integer additions.

22. **ECE 561 Only:** The code listing shows the loop body of the Rotate_Points function. Mark each operation (+ – * / =) with:

- **F** if performed with floating-point math.
- **N** if performed with integer math.

Insert a **T** wherever type conversions (floating-point <-> integer) are needed. Assume the compiler performs no optimizations.



```
x = ipt[n].X - center->X;
y = ipt[n].Y - center->Y;
opt[n].X = x*c - y*s + center->X;
opt[n].Y = x*s + y*c + center->Y;
```

23. **Extra Credit**: What is the break-even value for num_points which makes lines 7-8 take about as long as lines 9-14? (Please write the equation if you don't have a calculator). Explain how this value affects your approach to optimization.

They are equal when 150 us = num_points*24.4 us

Solving for num_points gives us num_points = 150/24.4 = 6.148

So…

- If num_points is always 6 or less, then the sinf and cosf operations dominate the function's execution time and we should optimize them first.
- If num_points is always 7 or more, then the loop dominates the function's execution time and we should optimize it first.
- If num_points varies above and below the 6.148 boundary, then we have a more complex problem and need to consider other factors:
  - the distribution of num_points for a typical run of the program
  - the timing model for the function (t = 150 us + num_points*24.4 us).