ARM assembly

1) [25 pts] Convert the following C code fragment into ARM assembly. Register assignments are indicated in the comments (do not write assembly code for the variable declarations). You may use other general purpose registers as temporary storage. Use only the ARM instructions in the lecture notes and the ARM assembly summary (Pages/Datasheets section on Canvas). You do not need to use any of the advanced instruction formats, such as applying a shift to the second source operand.

```
uint32_t A[16], B[16];  // r0 = &A[0]; r1 = &B[0];
uint32_t i, m;          // r2 = i; r3 = m;

B[i] = (A[i] >> 1) + ((B[i] << 8) & m);
```

This version uses only the basic instruction formats (all you need for the exam):

```
lsl r4, r2, #2     ; r4 = i*4; // array offset
ldr r5, [r0, r4]   ; r5 = copy of A[i];
lsr r5, r5, #1     ; r5 = r5 >> 1;
ldr r6, [r1, r4]   ; r6 = copy of B[i];
lsl r6, r6, #8     ; r6 = r6 << 8;
and r6, r6, r3     ; r6 = r6 & m;
add r5, r5, r6     ; r5 = r5 + r6;
str r5, [r1, r4]   ; B[i] = r5; // write to B[i] in memory
```

The following solution takes advantage of more advanced instruction formats:

```
ldr r5, [r0, r2, lsl #2]  ; r5 = copy of A[i];
ldr r6, [r1, r2, lsl #2]  ; r6 = copy of B[i];
and r6, r3, r6, lsl #8    ; r6 = m & (r6 << 8);
add r5, r6, r5, lsr #1    ; r5 = r6 + (r5 >> 1);
str r5, [r1, r2, lsl #2]  ; B[i] = r5;
```

2) [25 pts] Convert the following C code fragment into ARM assembly. The rules are the same as for problem 1. Note: The short-circuit AND operator && does not evaluate the second condition if the first is false.

```
int32_t i, x;   // r0 = i; r1 = x;
int32_t A[32];  // r2 = &A[0];

if ((i < 31) && (A[i + 1] <= A[i])) {
    x--;
}
else {
    x = 0;
}
```

```
        cmp r0, #31
        bge else1         ; if i >= 31, goto else1
                          ; evaluate second condition only if i < 31
        lsl r3, r0, #2    ; r3 = i*4;
        add r3, r2, r3    ; r3 = &A[i];
        ldr r4, [r3, #4]  ; r4 = copy of A[i + 1];
        ldr r5, [r3]      ; r5 = copy of A[i];
        cmp r4, r5
        bgt else1         ; if A[i + 1] > A[i], goto else1

        sub r1, r1, #1    ; x--; // both conditions true
        b   done1         ; goto done1

else1   mov r1, #0        ; x = 0;

done1
```

3) [25 pts] Convert the C function `max4()` into ARM assembly. Your resulting assembly functions must be callable from C, so should adhere to the C register convention discussed in lecture (also in the ARM assembly summary on Canvas). You do not need to use any assembler directives or optimize your code. Hints: Carefully study which registers a function must preserve upon return, and which registers the caller can expect to be modified across a function call. You will need to save registers onto the stack to implement this function.

```
int32_t max2(int32_t a, int32_t b); // implemented elsewhere

int32_t max4(int32_t a, int32_t b, int32_t c, int32_t d)
{
    return max2(max2(a, b), max2(c, d));
}
```

```
; int32_t max4(int32_t a, int32_t b, int32_t c, int32_t d)
; upon entry (by the C convention):
;   r0 = argument #1 = a
;   r1 = argument #2 = b
;   r2 = argument #3 = c
;   r3 = argument #4 = d
;   note: r0-r3, r12 and lr do not need to be preserved upon return
; local variables (in r4-r11 - preserved across function calls):
;   r4 = c
;   r5 = d
;   r6 = max2(a,b) return value
; upon return (by the C convention):
;   r0 = return value
max4    push {r4-r6, lr} ; save registers on the stack
        mov  r4, r2      ; r4 = c;
        mov  r5, r3      ; r5 = d;

                         ; argument #1 is already a
                         ; argument #2 is already b
        bl   max2        ; call max2();// r0-r3, r12, lr not preserved
        mov  r6, r0      ; r6 = max2(a,b) return value;

        mov  r0, r4      ; argument #1 = c;
        mov  r1, r5      ; argument #2 = d;
        bl   max2        ; call max2();

        mov  r1, r0      ; argument #2 = max2(c,d) return value;
        mov  r0, r6      ; argument #1 = max2(a,b) return value;
        bl   max2        ; call max2(); // return value in r0

        pop  {r4-r6, pc} ; restore registers and return
```

Instruction-level performance analysis

4) [25 pts] The following is a block copy function that copies data one word (4 bytes) at a time. Determine the number of clock cycles needed to execute the assembly version of this function on a Cortex-M4, assuming `count=8` and no memory access stalls. Use the instruction timing summary from lecture notes or the ARM assembly summary on Canvas. If you are interested, detailed timing information is in the ARM Cortex-M4 Technical Reference Manual section 3.3. Hint: The assembly code is not a one-to-one conversion from C, but the loop runs for the same number of iterations.

```
void block_copy4(uint32_t dst[], uint32_t src[], uint32_t count)
{
    uint32_t i;
    for (i = 0; i < count; i++) {
        dst[i] = src[i];
    }
}
```

```
; void block_copy4(uint32_t dst[], uint32_t src[], uint32_t count);
; Arguments:
; r0 = dst = destination pointer
; r1 = src = source pointer
; r2 = count = number of 4-byte words to copy
; Local variables:
; r2 = number of bytes to copy
; r3 = offset = i * 4  (does not follow the C code exactly)
; r12 = temp
block_copy4
        lsl   r2, r2, #2     ; r2 = count * 4; // # of bytes to copy
        mov   r3, #0         ; offset = 0;

loop1   cmp   r3, r2
        bhs   done1          ; if (offset >= total bytes), done

        ldr   r12, [r1, r3]  ; load word from source
        str   r12, [r0, r3]  ; store word to destination
        add   r3, r3, #4     ; offset += 4;
        b     loop1          ; continue loop

done1   bx    lr             ; return
```

```
block_copy4
        lsl   r2, r2, #2      ; data operation: 1 cycle
        mov   r3, #0          ; data operation: 1 cycle

loop1   cmp   r3, r2          ; data operation: 1 cycle
        bhs   done1           ; branch to label: 1 cycle if not taken
                              ;                  2 cycles if taken

        ldr   r12, [r1, r3]   ; LDR, 1st in sequence: 2 cycles
        str   r12, [r0, r3]   ; STR with register offset
                              ;    2nd in sequence: 1 cycle
        add   r3, r3, #4      ; data operation: 1 cycle
        b     loop1           ; branch to label, always taken: 2 cycles

done1   bx    lr              ; branch to register, always taken:
                              ;    3 cycles
```

The loop is executed 8 times (`count = 8`), `bhs` is not taken in these iterations.

cycles each iteration = 1 (cmp) + 1 (bhs) + 2 (ldr) + 1 (str) + 1 (add) + 2 (b) = 8

In the 9th iteration, `bhs` is taken.

cycles for 9th iteration = 1 (cmp) + 2 (bhs) = 3

The remaining instructions are executed once.

cycles for instructions executed once = 1 (lsl) + 1 (mov) + 3 (bx) = 5

Total clock cycles = 8 (cycles/iteration) * 8 (iterations) + 3 (9th iteration) + 5 = 72

Note that on the exam you may need to interpret assembly language to determine the number of iterations. In this problem, the following instructions determine the number of iterations. It may help to write out the value of `r3` each time it is used in the `cmp` instruction.

```
        lsl   r2, r2, #2      ; r2 = 8 << 2 = 32;
        mov   r3, #0          ; r3 = 0;

loop1   cmp   r3, r2          ; r3 = 0,4,8,12,16,20,24,28,32 (last)
        bhs   done1           ; if r3 >= 32, done

        add   r3, r3, #4      ; r3 += 4;
        b     loop1           ; continue loop
done1
```