1. Фрагменты кода на языке высокого уровня и соответствующие им коды на языке Ассемблер ARM

Пример 1.

- High-level language (C)

```
x = (a + b) - c;
```

- ARM:

```
ADR r4,a            ; get address for a
LDR r0,[r4]         ; get value of a
ADR r4,b            ; get address for b, reusing r4
LDR r1,[r4]         ; get value of b
ADD r3,r0,r1        ; compute a+b
ADR r4,c            ; get address for c
LDR r2,[r4]         ; get value of c
SUB r3,r3,r2        ; complete computation of x
ADR r4,x            ; get address for x
STR r3,[r4]         ; store value of x
```

Пример 2.

- High-level language (C)

```
y = a*(b+c);
```

- ARM:

```
ADR r4,b            ; get address for b
LDR r0,[r4]         ; get value of b
ADR r4,c            ; get address for c
LDR r1,[r4]         ; get value of c
ADD r2,r0,r1        ; compute partial result
ADR r4,a            ; get address for a
LDR r0,[r4]         ; get value of a
MUL r2,r2,r0        ; compute final value for y
ADR r4,y            ; get address for y
STR r2,[r4]         ; store y
```

Пример 3.

- High-level language (C)

```
z = (a << 2) | (b & 15);
```

- ARM:

```
ADR r4,a            ; get address for a
LDR r0,[r4]         ; get value of a
MOV r0,r0,LSL#2     ; perform shift
ADR r4,b            ; get address for b
LDR r1,[r4]         ; get value of b
AND r1,r1,#15       ; perform AND
ORR r1,r0,r1        ; perform OR
ADR r4,z            ; get address for z
STR r1,[r4]         ; store value for z
```

Пример 4. Condition Codes

- High-level language (C)

```
if (i == 0)
{
   i = i +10;
}
```

- ARM: (допустим, что i находится в R1)

```
SUBS R1, R1, #0
ADDEQ R1, R1, #10
```

Пример 5. Condition Codes

- High-level language (C)

```
for ( i = 0 ; i < 15 ; i++)
{
   j = j + j;
}
```

- ARM:

```
           SUB R0,R0,R0           ; i -> R0 and i = 0
start      CMP R0,#15             ; is i < 15?
           ADDLT R1,R1,R1         ; j = j + j
           ADDLT R0,R0,#1         ; i++
           BLT start
```

Пример 6.

- High-level language (C)

```
for ( i = 0 ; i < 15 ; i++)
{
   j = j + j;
}
```

- ARM:

```
          ; compute and test condition
          ADR r4,a          ; get address for a
          LDR r0,[r4]       ; get value of a
          ADR r4,b          ; get address for b
          LDR r1,[r4]       ; get value for b
          CMP r0,r1         ; compare a < b
          BGE fblock        ; if a >= b, branch to false block

          ; true block
          MOV r0,#5         ; generate value for x
          ADR r4,x          ; get address for x
          STR r0,[r4]       ; store x
          ADR r4,c          ; get address for c
          LDR r0,[r4]       ; get value of c
          ADR r4,d          ; get address for d
          LDR r1,[r4]       ; get value of d
          ADD r0,r0,r1      ; compute y
          ADR r4,y          ; get address for y
          STR r0,[r4]       ; store y
          B after           ; branch around false block

          ; false block
fblock    ADR r4,c          ; get address for c
          LDR r0,[r4]       ; get value of c
          ADR r4,d          ; get address for d
          LDR r1,[r4]       ; get value for d
          SUB r0,r0,r1      ; compute a-b
          ADR r4,x          ; get address for x
          STR r0,[r4]       ; store value of x
after ...
```

Пример 7. Использование более сложных условных команд (Conditional Instruction)

- High-level language (C)

Такой же код на языке высокого уровня (C), как в предыдущем примере; другой вариант реализации на ассемблере ARM

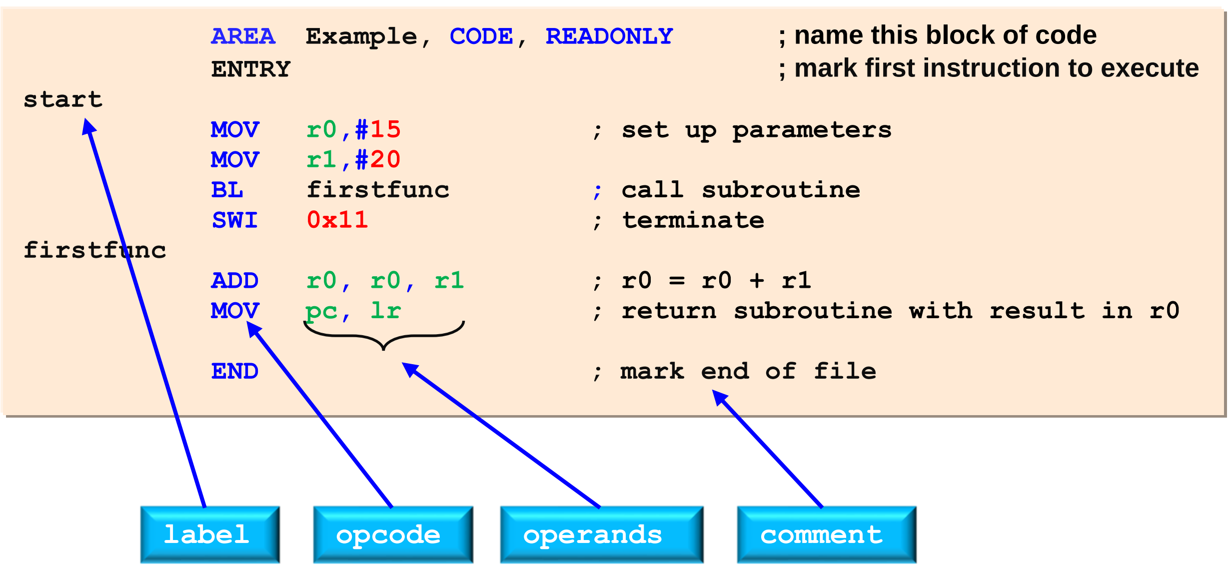
- ARM:

```
; compute and test condition
ADR r4,a        ; get address
for a
LDR r0,[r4]     ; get value of
a
ADR r4,b        ; get address

ADRLT r4,x      ; get address for x
STRLT r0,[r4]   ; store x
ADRLT r4,c      ; get address for c
LDRLT r0,[r4]   ; get value of c
ADRLT r4,d      ; get address for d
LDRLT r1,[r4]   ; get value of d
ADDLT r0,r0,r1  ; compute y
ADRLT r4,y      ; get address for y
STRLT r0,[r4]   ; store y
; false block
ADRGE r4,c      ; get address for c

LDRGE r0,[r4]   ; get value of c
ADRGE r4,d      ; get address for d
LDRGE r1,[r4]   ; get value for d
SUBGE r0,r0,r1  ; compute a-b
ADRGE r4,x      ; get address for x
STRGE r0,[r4]   ; store value of x
```

2. ARM Assembler

2.1. Assembly Language Basics

- The following is a simple example which illustrates some of the core constituents of an ARM assembler module:



2.2. General Layout

- The general form of lines in an assembler module is:

```
label <space> opcode <space> operands <space>    ; comment
```

- Each field must be separated by one or more **<whitespace>** (such as a space or a tab).
- Actual instructions never start in the first column, since they must be preceded by **whitespace,** even if there is no label.
- All three sections are optional and the assembler will also accept blank lines to improve the clarity of the code.

2.3. Simple Example Description

- The main routine of the program (labelled **start**) loads the values 15 and 20 into registers `r0` and `r1`.
- The program then calls the subroutine `firstfunc` by using a branch with link instruction (`BL`).
- The subroutine adds together the two parameters it has received and places the result back into `r0`.
- It then returns by simply restoring the program counter to the address, which was stored in **link register** (`r14`) on entry.
- Upon return from subroutine, the main program simply terminates using software interrupt (`SWI`) 11. This instructs the program to exit cleanly and return control to the debugger.

2.4. Assembly Directives

- Directives are instructions to the assembler program, NOT to the microprocessor.
- `AREA` Directive – specifies chunks of data or code that are manipulated by the linker.
  - A complete application will consist of one or more areas. The example above consists of a single area, which contains code, and is marked as being read-only. A single `CODE` area is the minimum required to produce an application.
- `ENTRY` Directive – marks the first instruction to be executed within an application
  - An application can contain only a single entry point and so in multisource-module application, only a single module will contain an ENTRY directive.
- `END` Directive – marks the end of the module.

2.5. Sum of n numbers

2.5.1. First version

```
            AREA        SUM, CODE, READONLY
            EXPORT      sum1
            ; r0 = input variable n
            ; r0 = output variable sum

sum1
            MOV    r1,#0              ; set sum = 0

sum_loop
            ADD    r1,r1,r0           ; set sum =sum + n
            SUBS   r0,r0,#1           ; set n = n - 1
            BNE    sum_loop
sum_rtn
            MOV    r0,r1              ; set return value
            MOV    pc,lr

            END
```

2.5.2. Second version

```
                AREA        SUM, CODE, READONLY
                EXPORT      sum
                ; r0 = input variable n
                ; r0 = output variable sum

sum
                MLA    r1,r0,r0,r0         ; n*(n+1) = n*n + n
                MOV    r0,r1,LSR#1         ; divide by 2

sum_rtn
                MOV    pc,lr

                END
```

2.6. Compute k ($k = \log_2 n$ or $n = 2^k$)

```
                AREA        LOG, CODE, READONLY
                EXPORT      log
                ; r0 = input variable n
                ; r0 = output variable m (0 by default)
                ; r1 = output variable k (n <= 2^k)
log
                MOV    r2, #0              ; set m = 0
                MOV    r1, #-1             ; set k = -1
log_loop
                TST    r0, #1              ; test LSB(n) == 1
                ADDNE r2, r2, #1           ; set m = m+1 if true
                ADD    r1, r1, #1          ; set k = k+1
                MOVS   r0, r0, LSR #1      ; set n = n>>1
                BNE    log_loop            ; continue if n != 0
                CMP    r2, #1              ; test m ==1
                MOVEQ r0, #1               ; set m = 1 if true
log_rtn
                MOV    pc,lr
                END
```