ПРОГРАМУВАННЯ AVR МОВОЮ АСЕМБЛЕР

3.1. Приклад простої програми для AVR.

Знайомство з технікою програмування МК AVR мовою асемблер почнемо з розгляду структури простої програми. Усі приклади програм у конспекті лекцій наведені для програмного середовища AVR Studio.

```
.include "m32Adef.inc"    ; підключення бібліотечного файлу МК

        .DSEG             ; SRAM – сегмент даних (оперативна пам'ять)

        .CSEG             ; FLASH – сегмент програмного коду
        .org    $000      ┐
        jmp     reset     │
        ; .......         ├ вектор переривань
        .org    $028      │
        reti              ┘
reset:
        ;ініціалізація периферії
        ldi     r16, Low(RAMEND)
        out     SPL, r16
        ldi     r16, High(RAMEND)
        out     SPH, r16

        ldi     r16, 0x00        } Присвоєння регістрам
        ldi     r17, 0xFF        } загального призначення значень

        out     PORTA, r16       } Порт A -- працює на вихід
        out     DDRA, r17

        out     PORTB, r17       } Порт B -- працює на вхід
        out     DDRB, r16        } з підтягуючими резисторами

main:
        [ ОСНОВНИЙ ПРОГРАМНИЙ ЦИКЛ ]
        rjmp    main

        [ ПІДПРОГРАМИ ]

        .ESEG   ;EEPROM – сегмент для енергонезалежних даних
```

Оскільки кожна модель МК AVR має на борту певний визначений для неї набір периферії, то відповідно адреси регістрів, що відповідають за визначені пристрої периферії, відрізняються у кожної моделі. І тому, щоб не мати справи з адресами регістрів, а лише з їхніми іменами (визначеними у середовищі AVR Studio), усі відповідності між іменами та адресами регістрів винесені в окремі бібліотечні файли. Першим ділом при написані програми мовою асемблер у середовищі AVR Studio ми повинні підключити за допомогою директиви .include необхідний бібліотечний файл для вибраної моделі МК. У нашому випадку для моделі ATmega32A бібліотечний файл має назву "m32Adef.inc".

Компілятор AVR Studio має певний набір директив, які допомагають писати прості та ефективні програми на асемблері. Директиви .DSEG, .CSEG. та .ESEG умовно розбивають код програми на сегменти, в яких будуть міститися відповідні дані – для SRAM, для FLASH і для EEPROM. Директива .org вказує, що у цьому місці програми встановлюється конкретне значення адреси у пам’яті програми.

У наведеному прикладі програмний код (після директиви .CSEG) починається записом вектора визначених переривань. Перше переривання, скид МК, здійснює стрибок на програмну мітку reset, з якої починається ініціалізація наявної периферії та закінчується основним програмним циклом – програмною петлею на мітку main.

Навіщо зациклювати основну програму? Якщо ж немає в кінці написаного коду програми переходу на початок мітки main, тоді програма буде виконуватися далі, до кінця FLASH (а ATmega32A має аж 32 кБайти). У пустій області програмної пам’яті прописані значення FF, і відповідно МК пройдеться по них до кінця об’єму FLASH, ніби виконуючи пустий оператор, а потім продовжиться виконання з початку програми, тобто стрибок на мітку reset, ініціалізація периферії і т.п.

Оскільки наша робоча програма є зациклена, то одразу після команди переходу на мітку main можемо розміщувати необхідні підпрограми, до яких ми будемо звертатися з основного програмного циклу.

Введення чи зчитування значень з регістрів периферії ми можемо здійснювати лише через посередництво регістрів загального призначення R0-R31. Тобто, для того щоб записати значення у якийсь з периферійних регістрів, ми спершу присвоюємо це значення якомусь з регістрів загального призначення, а потім з цього регістра пересилаємо в периферійний регістр (рис. 3.1).

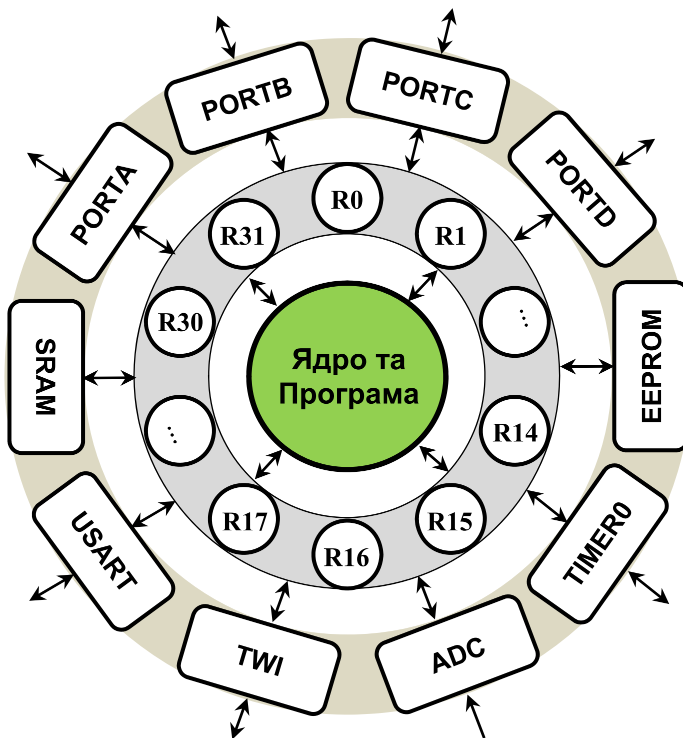

Рис. 3.1. Регістри загального призначення у ролі посередників

3.2. Директиви та функції асемблера AVR.

Асемблер МК AVR підтримує певне число директив. Ці директиви не транслюються безпосередньо в оперативний код програми. Натомість, вони використовуються для розмітки пам'яті, визначення макросів, ініціалізації пам'яті тощо. Вони є інструментом в руках програміста та спрощують йому життя. Розглянемо почергово основні директиви асемблера AVR.

.include – ця директива вказує компілятору зчитати вказаний файл та включити його вміст у поточну програму. Вміст зчитаного файлу вставляється у точку виклику цієї директиви. Включені файли також можуть містити .include директиви.

.exit – ця директива вказує компілятору на те, що досягнуто кінця файлу, і код, що розміщений після цієї директиви, ігнорується. Наприклад, ми включаємо за допомогою директиви .include певний файл з програмним кодом у наш проект і хочемо з метою уникнення

конфліктів імен, щоб включило не весь його вміст, а лише першу частину . Для цього у визначеному місці вставляємо директиву .exit, і тоді весь решта код після цієї директиви не буде включатися у наш проект.

У МК є в наявності три види пам'яті: пам'ять програм (FLASH), оперативна пам'ять (SRAM) та енергонезалежна пам'ять даних (EEPROM), і відповідно, в програмі передбачена можливість розмітки цих областей та при необхідності – ініціалізації значеннями.

```
.include "m32Adef.inc"

;============== SRAM – сегмент даних ==============================
          .DSEG

;============== FLASH – сегмент програмного коду =================
          .CSEG

;============== EEPROM – сегмент для енергонезалежних даних ======
          .ESEG
```

Це свого роду шаблон для будь-якого нового проекту.

.DSEG (Data segment) – визначає початок сегменту даних. У цьому сегменті ми можемо лише здійснювати розмітку області SRAM за допомогою директиви .byte (та міток). Іншими словами, ми резервуємо місце для наших даних в оперативній пам'яті, помічаючи ці зарезервовані області мітками, адреси яких визначає компілятор.

.CSEG (Code segment) – визначає початок сегменту, де розміщується наша основна виконавча програма.

.ESEG (EEPROM segment) – визначає початок сегменту, що відноситься до енергонезалежних даних. У цій області ми можемо виконувати розмітку пам'яті як для розміщення конкретних даних, так і резервування простору для програмного збереження наших даних.

.byte – резервує вказану кількість байтів пам'яті в областях SRAM та EEPROM. Не може використовуватися у сегменті програмного коду.

```
          .DSEG
value:    .byte    1          ;резервує 1 байт для мітки value
vector:   .byte    3          ;резервує 3 байти для мітки vector
table:    .byte    10         ;резервує 10 байтів для мітки table
```

Значення адрес наших міток визначає компілятор. Якщо у моделі ATmega32A значення SRAM починається зі значення 0x060, то для цих міток компілятор визначить такі адреси:

```
value   = 0x060
vector  = 0x061
table   = 0x064
```

За замовчуванням значення пам'яті даних мають значення FF.

Для збереження табличних константних значень, тобто таких, які не змінюються в процесі виконання програми, наприклад таблиця значень синусів та косинусів, може використовувати як пам'ять EEPROM, так і пам'ять програм FLASH. Розміщення масивів даних в цих сегментах здійснюється за допомогою таких директив:

.db (define constant byte(s)) – вказує на розміщення масиву байтів. Кожне значення (вираз) масиву може приймати значення від -128 до 255. Якщо значення виражене від'ємним числом, тоді воно буде розміщене у пам'яті програм чи EEPROM як 8-ми бітне число у доповнюючому двійковому коді. Масив повинен містити щонайменше одне значення, два і більше значень відокремлюються між собою комами. Якщо директива .db вказана у сегменті .CSEG та масив містить більше ніж одне значення, тоді значення пакуються в пам'яті програм так, що два байти розміщуються в кожному слові пам'яті програм. Якщо масив містить непарне число значень, тоді останнє значення буде розміщене у своєму власному слові програмної пам'яті, а невикористана половинка програмного слова буде встановлена у нуль.

.dw (define constant word(s)) – вказує на розміщення масиву слів (2 байтних значень). Кожне значення масиву може приймати значення від -32768 до 65535. Якщо значення виражене від'ємним числом, тоді воно буде розміщене у пам'яті програм чи EEPROM як 16-ми бітне число у доповнюючому двійковому коді.

```
        .CSEG
v1:     .db     0, 255, 0b01010101, -125, 0xb3
v2:     .db      "Hello my dear friend"
v3:     .dw     -30125, 0xa0b5, 0b1010101010101010, 56, 65535

        .ESEG
V4:     .db     1, 2, 3
```

.dd (define constant doubleword(s)) та .dq (define constant quadword(s)) – ці директиви подібні до попередніх та використовуються для представлення масивів з 32 та 64-бітних значень.

.org (set program origin) – ця директива встановлює абсолютне значення адреси для комірки пам’яті. Може використовуватися у всіх трьох сегментах пам’яті: даних, програми та EEPROM. Є певні особливості щодо його використання. Якщо директива розміщується у сегментах SRAM та EEPROM, то необхідно враховувати, що адресація у них побайтна, а якщо розміщується у сегменті програми, тоді слід пам’ятати, що тут адресація послівна (по 2 байта). Ще один момент стосується пам’яті даних SRAM. Якщо вказати цю директиву з параметром 0, тоді ми будемо адресувати область, в якій розміщені регістри загального призначення. Тому для області даних розмітку слід робити, починаючи зі значення 0x60 чи навіть зі 0x100 (для МК з розширеною областю пам’яті для регістрів вводу/виводу). У прикладі простої програми ми зустрічали ці директиви при записі вектора переривань. Вони там необхідні, оскільки кожне переривання здійнює перехід на чітко встановлену адресу в межах цього вектора, що розміщений на початку області програмного коду.

```
        .DSEG
        .org    $150    ;встановлює адресу SRAM у значення 0x150
var:    .byte   1       ;резервує 1 байт у SRAM за адресою 0x150

        .CSEG
        .org    $40     ;встановлює Програмний лічильник у значення 0x40
        inc     r0      ;виконується якась робота
```

.def – ця директива дозволяє встановити символьні псевдоніми для регістрів загального призначення. В процесі написання програми ми для роботи з нашими змінними вибираємо певні регістри загального призначення. Наприклад, r0 – секунди годинника, r1 – хвилини, r2 – години, r16 – тимчасові проміжні значення і т.п. В процесі написання програми ми можемо заплутатися в тому, які регістри вже використані, а які ще вільні. Якщо ж захочемо за якоюсь змінною величиною закріпити інший регістр загального призначення, тоді це теж викличе проблеми, оскільки потрібно буде виловити усі розміщення попередньо-вибраного нами регістра. Тому правильним підходом є на початку програми визначати таблицю відповідних символьних псевдонімів для наших регістрів загального призначення, і у програмі вже використовувати їх.

```
.def    _second   = r0
.def    _minute   = r1
.def    _hour     = r2
.def    _temp     = r16
.def    _temp2    = r17

        .CSEG

        ldi     _temp, 25         ;встановлює r16 у значення 25
        mov     _minute, _temp    ;пересилає значення з  r16 у r1
```

Для одного регістра загального призначення може бути одночасно визначено декілька псевдонімів. Про це звісно компілятор видасть повідомлення. Такий підхід буває зручним для розділення інформативних термінів. При цьому звертатися до вибраного регістра не має значення за яким псевдонімом.

```
.def    _bin      = r10
.def    _low      = r10
```

.equ – ця директива закріплює за символьними мітками константні числові значення чи вирази, які в процесі компіляції будуть підмінені у програмі замість міток. Наприклад, у нас в програмі часто використовується певна константа, і скажімо, в певний момент з’ясується, що ми її невірно вибрали. Доведеться дуже ретельно виловлювати усі її появи у програмі, щоб випадково не пропустити. Натомість, оголосивши на початку програми за допомогою директиви .equ мітку, проблема знімається сама собою.

```
.equ    XTAL = 8000000
.equ    BaudRate = 9600
.equ    BaudDiv = XTAL / (16 * BaudRate) - 1
```

.set – за своєю роботою ця директива, як і .equ, закріплює за міткою числове значення, але на відміну від попередньої, вона дозволяє перевстановлювати по ходу програми значення для цієї мітки.

```
.set    Foo = 5
loop:
        ldi     r16, Foo          ; r16 = Foo = 5 (!)
.set    Foo = 10
        ldi     r16, Foo          ; r16 = Foo = 10
.set    Foo = Foo + 10
        ldi     r16, Foo          ; r16 = Foo =20
        rjmp    loop
```

У наведеному прикладі директива .set розбиває програму на три підсегменти, і в кожному з них мітка Foo має своє визначене значення. Навіть у циклі, де Foo приймає нове значення, при переході на початок циклу там буде діяти інше значення для Foo, яке визначене ще перед початком циклу. Тому такі маніпуляції з директивою .set слід застосовувати у програмі з обережністю.

Для створення макросів передбачені директиви початку .macro та кінця макросу .endmacro або .endm. На відміну від виклику підпрограм, де здійснюється перехід за міткою та повернення у вихідну точку, виклик макроса означає, що компілятор виконає вставку коду макроса у точку виклику. Скільки раз макрос буде викликаний, на стільки ж і збільшиться об’єм вихідного програмного коду.

```
.macro  Addition
        clr     @0
        add     @0, @1
        add     @0, @2
.endmacro

        .CSEG
        ldi             r18, 5    ; r18 = 5
        ldi             r19, 8    ; r19 = 8
        Addition  r10, r18, r19   ; r10 = r18+r19 = 13
```

Макрос може приймати до 10 параметрів. Посилання на ці параметри позначаються в середині макроса як @0-@9. Порядок слідування визначається при виклику макроса. У наведеному прикладі @0 це r10, @1 – r18, а @2 – r19. Під час компіляції при підстановці коду макроса у точки виклику замість параметризованих змінних підставляються параметри, що вказані через кому після імені викликаного макроса.

Асемблер також підтримує умовну компіляцію, і для неї використовуються такі директиви: .else, .elif, .endif, .error, .if, .ifdef, .ifndef, .message. Це дає можливість писати програми одразу для цілого ряду моделей МК.

Компілятор AVR Studio має у своєму арсеналі окрім директив ще також і набір функцій, які обчислюються в процесі компіляції (табл. 3.1). Ці функції мають суто допоміжний характер та застосовуються для спрощення обчислень необхідних величин, що використовуються як константи для нашої основної програми. Вони не мають відношення до математичних бібліотек, які використовуються для обчислень при роботі МК.

Таблиця 3.1. Основні функції компілятора AVR Studio

Low(*вираз*)	Повертає молодший байт від числового виразу
High(*вираз*)	Повертає другий байт від числового виразу
Byte2(*вираз*)	Аналогічна функції high()
Byte3(*вираз*)	Повертає третій байт від числового виразу
Byte4(*вираз*)	Повертає четвертий байт від числового виразу
Lwrd(*вираз*)	Повертає 0-15 біти від числового виразу
Hwrd(*вираз*)	Повертає 16-31 біти від числового виразу
Page(*вираз*)	Повертає 16-21 біти від числового виразу
Exp2(*вираз*)	Повертає значення 2 до степені виразу
Log2(*вираз*)	Повертає цілу частину від двійкового логарифма

```
.equ      foo   =0x1234
.equ      foo2 =0xABCDEF89

          .CSEG
          ldi         r16, Low(foo)               ; r16 = 0x34
          ldi         r17, High(foo)              ; r17 = 0x12
          ldi         r18, Low(foo2)              ; r18 = 0x89
          ldi         r19, Byte2(foo2)            ; r19 = 0xEF
          ldi         r20, Byte3(foo2)            ; r20 = 0xCD
          ldi         r21, Byte4(foo2)            ; r21 = 0xAB
          ldi         r22, Exp2(3)                ; r22 = 8
          ldi         r23, Log2(17)               ; r23 = 4

table:    .dw         Lwrd(foo2), Hwrd(foo2)  ; 0xEF89, 0xABCD
```

Також компілятор AVR Studio має ряд арифметичних, порозрядних, логічних та умовних операцій. Вони, як і функції, мають допоміжний характер і спрощують обчислення необхідних величин, та обробляються на етапі компіляції.

Таблиця 3.2. Перелік операцій компілятора AVR Studio

<table>
<tr><td>Арифметичні</td><td>+ - * /</td></tr>
<tr><td>Порозрядні</td><td>~ & | ^ << >></td></tr>
<tr><td>Логічні</td><td>! && ||</td></tr>
<tr><td>Умовні</td><td>< > <= >= == != ?</td></tr>
</table>

Наведені операції за своєю дією ідентичні аналогічним операціям мови C++, за винятком одного моменту, що стосується значення результату логічних та умовних операцій. Воно є числом 1 або 0, у залежності від результату операції. Наприклад, логічне && повертає 1, якщо обидва значення є більшими за нуль, та повертає 0, якщо

значення рівні 0. Оператор рівності == повертає 1, якщо порівнювані значення рівні, інакше поверне число 0.

```
    ldi     r16, 25*(c1==c2) + 1     ;якщо c1==c2, тоді  r16 = 26
                                     ; інакше r16 = 1
```

Коментарі в асемблері можуть записуватися як в класичному виді за допомогою ‘;’, так і в Сі-стилі.

```
; класичний асемблер ний коментар, діє до кінця стрічки
// Сішний рядковий коментар, діє до кінця стрічки
/* Сішний блоковий коментар,
 обгороджений колючками текст
компілятором ігнорується */
```

Довжина рядка в асемблерному коді обмежена 120 символами.

3.3. Представлення чисел.

Основним типом чисел в асемблері є цілочисельний 8-бітний тип. Асемблер підтримує різні представлення чисел:

- Десяткове (за замовчуванням): 26, 255;
- Шістнадцяткове (C та Pascal нотації): 0x1A, $FF, $1A, 0xFF;
- Двійкове: 0b00011010, 0b11111111;
- Вісімкове (нуль спереду): 032, 0377.

Асемблер може працювати як з беззнаковими цілими, так і зі знаковими числами. Для роботи з від’ємними числами використовується їхнє представлення у доповнюючому коді. Для представлення 8-бітного від’ємного числа від значення 256 віднімається це число без знаку, наприклад, -10 у доповнюючому 8-бітному числі це 256-10=246. Для асемблера значення -10 та 246 є ідентичними, тому аналіз значень покладається суто на програміста. Використання доповнюючого коду є природним для обчислювальної техніки, бо при цьому спрощується сам механізм арифметичних операцій з числами. У таблиці 3.3 наведено порівняльне співвідношення між знаковими значеннями та їхніми представленнями у 8-бітному форматі в регістрах.

Таблиця 3.3. Представлення 8-бітних знакових чисел

-4	-3	-2	-1	0	1	2	3
252	253	254	255	0	1	2	3
11111100	11111101	11111110	11111111	00000000	00000001	00000010	00000011

Ознакою від'ємного числа у двійковому виді є наявність «1» у старшому розряді. Про отриманий від'ємний результат сигналізує відповідний прапорець регістру стану SREG. І знову ж таки, МК не відрізняє знакових чисел від беззнакових, тому контроль покладається на програміста.

У МК AVR також підтримується апаратне множення чисел у дробовому форматі 1.7. Дамо пояснення про формати такого типу.

Формат «n.q» позначає дробове число з n розрядами (двійковими цифрами) зліва від десяткової крапки та q розрядами справа від неї. Дробові числа для 8-бітного МК AVR представляються у форматі 1.7, тобто 1 розряд під цілу частину чи знак та 7 розрядів під дробову частину.

Для беззнакового дробового числа у форматі 1.7 діапазон можливих значень лежить в межах [0; 2>. Для знакового дробового числа єдиний розряд для цілого значення відведений суто під знак, і тому діапазон можливих значень лежить в межах [-1; 1>.

Таблиця 3.4. Представлення беззнакових дробових чисел 1.7

2-ве	**10-ве**	**1.7**	**2-ва**	**10-ве**	**1.7**
00000000	0	0.0	…	…	…
00000001	1	0.0078125	11111101	253	1.9765625
00000010	2	0.015625	11111110	254	1.984375
00000011	3	0.0234375	11111111	255	1.9921875

Таблиця 3.5. Представлення знакових дробових чисел 1.7

2-ве	**10-ве**		**1.7**	**2-ве**	**10-ве**		**1.7**
	знак.	**беззн.**			**знак.**	**беззн.**	
00000000	0	0	0.0	10000000	-128	128	-1.0
00000001	1	1	0.0078125	10000001	-127	129	-0.9921875
…	…		…	…	…		…
01111110	126	126	0.984375	11111110	-2	254	-0.015625
01111111	127	127	0.9921875	11111111	-1	255	-0.0078125

Щоб отримати реальне значення числа у форматі 1.7, необхідно його десяткове цілочисельне представлення поділити на число 128.

При множенні чисел у форматі «n.q», наприклад n1.q1× n2.q2, кінцевий результат буде мати представлення (n1+n2).(q1+q2). Однак, для того щоб отриманий результат був у зручному форматі, апаратне множення виконує для результату ще й порозрядний зсув вліво на 1 біт, після чого результат може бути заокруглений до 1.7.

1.7× 1.7 = (2.14) <<1 = 1.15

У AVR передбачені три команди апаратного множення 1.7:
- **fmul** → беззнаковий 1.7 × беззнаковий 1.7;
- **fmuls** → знаковий 1.7 × знаковий 1.7;
- **fmulsu** → знаковий 1.7 × беззнаковий 1.7.

Для спрощення формування дробових чисел у форматі 1.7 та 1.15 передбачені спеціальні функції для компілятора AVR (табл. 3.6).

Таблиця 3.6. Додаткові функції компілятора

Q7(*вираз*)	Конвертує число з плаваючою комою до знакового формату 1.7 (для fmul/fmuls/ fmulsu команд)
Q15(*вираз*)	Конвертує число з плаваючою комою до знакового формату 1.15
Int(*вираз*)	Обрізає число з плаваючою комою до цілого числа (відкидає дробову частину)
Frac(*вираз*)	Отримує дробову частину числа з плаваючою комою (відкидає цілу частину)
Abs(*вираз*)	Повертає абсолютне значення з числа

Оскільки функція Q7() формує лише знакове дробове число 1.7, то для отримання беззнакового числа за допомогою цієї функції формуємо дробову частину числа та додаємо 1 у старший розряд.

```
        ldi     r16, Q7(-0.5)           ; r16=192(-64)   → -64/128=-0.5
        ldi     r17, Q7(0.5)            ; r17=64         → 64/128=0.5
        ldi     r18, (1<<7) | Q7(0.5)   ; r18=192        → 192/128=1.5
        ldi     r19, (1<<7) | Q7(0.4)   ; r19=179        → 179/128=1.3984375
        ldi     r20, Q7(-0.9)           ;r20=140(-116) → -116/128=-0.90625

; (беззнаковий 1.7) × (беззнаковий 1.7)
        fmul    r17, r18        ;res=24576      → 24576/32768=0.75
        fmul    r18, r19        ;res=3200       → 3200/32768=0.09765625
                                                        переповнення
; (знаковий 1.7) × (знаковий 1.7)
        fmuls   r16, r17        ;res=57344(-8192) → -8192/32768=-0.25
; (знаковий 1.7) × (беззнаковий 1.7)
        fmulsu  r16, r18        ;res=40960 (-24576) → -24576/32768=-0.75
        fmulsu  r20, r18        ;res=20992          →20992 /32768=0,640625
                                                        переповнення
```

При множенні дробових беззнакових чисел потрібно слідкувати, щоб не було переповнення, тобто результат не виходив за визначені межі (не був більшим за значення 2). Аналогічно, при множенні знакового на беззнаковий також потрібно слідкувати, щоб не було вихід за межі діапазону [-1;1>.

Щоб отримати реальне значення числа у форматі 1.15, необхідно його десяткове цілочисельне представлення поділити на число 2^{15}=32768.

Замість використання функції Q7() ми можемо просто необхідне число з плаваючою комою помножити на 128. Наведений нижче код ідентичний попередньому, з тією лиш різницею, що може бути відмінність в один розряд для чисел, що точно не відображаються на цілочисельний тип та мусять бути заокруглені.

```
ldi     r16, -0.5*128       ; Q7(-0.5)
ldi     r17, 0.5*128        ; Q7(0.5)
ldi     r18, 1.5*128        ; (1<<7) | Q7(0.5)
ldi     r19, 1.4*128        ; (1<<7) | Q7(0.4)
ldi     r20, -0.9*128       ; Q7(-0.9)
```

Якщо ж потрібно множити дробові числа, що виходять за межі, вказані для формату 1.7, наприклад 4.5×6.25, тоді необхідно умовно перейти до іншого формату. Виберемо для цього випадку беззнаковий формат 3.5 з діапазоном значень [0; 8>. У результаті множення беззнакових дробових чисел 3.5 отримаємо результат у форматі 6.10, а якщо врахувати ще зсув при множенні вліво, то кінцевий формат має бути у форматі 5.11.

Для запису у регістр загального призначення нашого числа з плаваючою комою його необхідно помножити на 2^5=32 (у степені кількість розрядів після коми). Для зворотного відображення у число з плаваючою комою необхідно отриманий результат у форматі 5.11 поділити на 2^{11}=2048.

4.5×6.25 = 28.125

```
ldi     r16, 4.5*32         ; r16=144       → 144 / 32 = 4.5
ldi     r17, 6.25*32        ; r17=200       → 200 / 32 = 6.25

; (беззнаковий 3.5) × (беззнаковий 3.5)<<1=(беззнаковий 5.11)
fmul    r16, r17        ;res= 57600     → 57600 / 2048 = 28.125
```

Аналогічні маніпуляції виконуються і при множенні знакових дробових чисел, а також й інших форматів 2.6, 4.4, 5.3, 6.2, 7.1. Формат 8.0 представляє собою цілі числа, і для нього слід використовувати команди апаратного множення mul, muls, mulsu, в яких вже не відбувається зсув вліво, а кінцевий результат представляється у форматі 16.0.