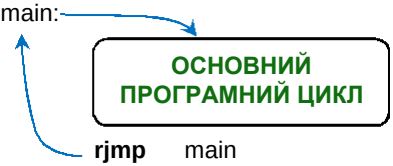
Лекція
Структура асемблерної програми.

Для зручного читання коду асемблерної програми, вона повинна мати перш за все чітку структурну організацію, повинна бути швидкою, не здійснювати затримок основного програмного циклу та має легко розширюватися. Ось приблизна структура для такої програми.

```
.include "m32Adef.inc" ; підключення бібліотечного файла МК ;========
Макроси ========================================
macro Clear
        clr         @0
.endmacro
;======== Макровизначення ==================================
.def      _temp = r16
.equ      Fig_0 = -64
;======== SRAM ==========================================
.DSEG
vector:  .byte     3
;======== FLASH =========================================
        .CSEG
        ;======== Вектор переривань ====================
        .org      $000
        jmp       reset
        .org      $028
        reti
        ;======== Підпрограми переривань ===============

reset:  ;======== Ініціалізація пам'яті та стеку ============

        ;======== Ініціалізація внутрішньої периферії =========

        ;======== Ініціалізація зовнішньої периферії ==========

        ;======== Видача дозволів на переривання ===========

main:
        rjmp      main
        ;======== Підпрограми ==========================

        ;======== Табличні дані
Table:  ========================== .db 25, 35, 45, 55
;======== EEPROM ========================================
        .ESEG
Einit:  .db         1,2,3
```



Про певні пункти структури програми ми вже в певній мірі дещо розказували. Тому зупинимося лише на окремих пунктах.

При старті програми МК, а особливо при її рестарті, регістри загального призначення та оперативна пам'ять не завжди є обнуленою. Часто там залишається сміття з попереднього запуску. І тому у програмі необхідно усі комірки пам'яті як оперативної, так і регістри загального призначення або встановлювати в необхідне значення, або обнулювати. Ну хоча б проініціалізувати ті комірки пам'яті, з якими ми будемо працювати.

За замовчуванням при старті МК значення стеку має мати значення кінця SRAM. Але все одно краще проініціалізувати його на початку програми, і бути впевненим, що стек розміщений у визначеному місці.

Ініціалізація внутрішньої периферії передбачає налаштування роботи різних таймерів, інтерфейсів, портів вводу/виводу і т.п.

Ініціалізація зовнішньої периферії передбачає налаштування роботи дисплеїв, зовнішньої пам'яті, ініціалізацію різних пристроїв, давачів, усього, що підключено ззовні до МК.

Коли основна ініціалізація виконана, тоді даються дозволи на переривання від периферії МК та загальний дозвіл (прапорець I регістру стану SREG). Після цього керування передається основному циклу програми.

Структурно підпрограми переривань розміщуються одразу після вектора переривань, а звичайні підпрограми після блоку основного циклу.

В кінці сегменту FLASH розміщуються таблиці з константами.

Паралельні порти вводу/виводу.

Паралельні порти вводу/виводу є основною складовою будь-якого МК AVR.

Ноги портів, окрім основного призначення – побайтного чи побітного вводу/виводу, можуть також використовуватися для роботи з певною внутрішньою периферією МК (у залежності від того, які альтернативні функції прикріплені до них).

Модель ATmega32A має в себе на борту 4 повних порти вводу/виводу: A, B, C та D. Налаштування та керування портом можемо виконувати як усім одразу, так і кожною його ногою окремо.

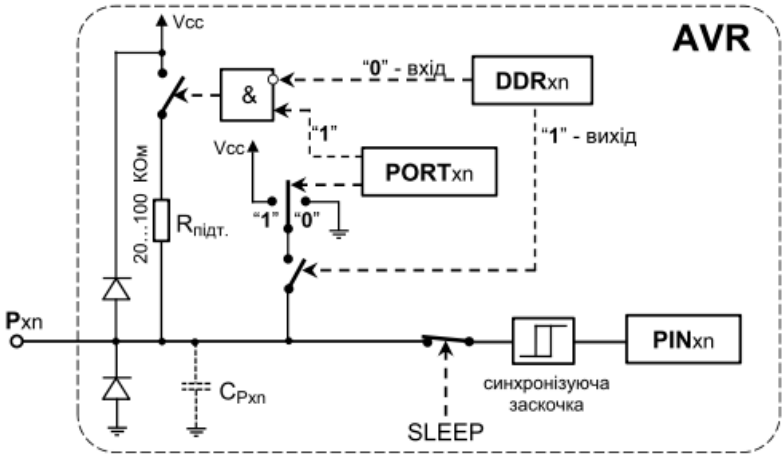


Рис. 1. Спрощена функціональна схема ноги порту вводу/виводу без врахування додаткових функцій

Усі ноги портів мають внутрішній діодний захист, який захищає від стрибків підвищеної напруги (верхній діод відкривається при напрузі вищій за напругу живлення МК, і з перепадом напруги вже борються фільтри БЖ) та від'ємної напруги (яка нейтралізується на землю нижнім діодом). Цей діодний захист призначений лише від імпульсних завад, і при перевищеній напрузі нога порту ймовірно вийде з ладу. Варто мати на увазі, що на кожній нозі присутня незначна внутрішня паразитна ємність.

За керування ногами портів МК AVR відповідає набір регістрів DDRx, PORTx та PINx (літера x відповідає назві регістра A, B, C чи D).

DDRx – ці регістри визначають напрямок роботи портів: на вхід чи на вихід. При цьому ми можемо налаштувати індивідуально кожну ногу вибраного порту. Якщо біт у регістрі, що відповідає потрібній нозі порту, дорівнює «0», тоді ця нога працює на вхід, якщо ж «1» – тоді на вихід. Візьмемо, наприклад, порт A:

	7	6	5	4	3	2	1	0
DDRA	1	1	0	0	0	1	1	0
	PA7 (OUT)	PA6 (OUT)	PA5 (IN)	PA4 (IN)	PA3 (IN)	PA2 (OUT)	PA1 (OUT)	PA0 (IN)

PORTx – значення бітів цих регістрів визначають налаштування виводів портів у залежності від значення, встановленого у регістрі DDRx. Якщо вивід працює на вихід, тобто біт у регістрі DDRx встановлений в «1», тоді значення відповідного біта регістра PORTx вказує на рівень напруги на нозі. Значення «1» встановлює на нозі напругу високого рівня (Vcc, напруга живлення МК), значення «0» встановлює низький рівень (цифрова земля).

Якщо вивід працює на вхід, тобто біт у регістрі DDRx встановлений в «0», тоді значення «1» відповідного біта регістра PORTx підключає внутрішній резистор (Rпідт ., рис. 3.2), який підтягує цей вивід до напруги живлення МК. Номінал цього підтягуючого резистора є не точним і знаходиться в межах 20-100 КОм. Значення «0» залишає цей вивід у високоімпедансному стані Hi-Z (без підтягуючого резистора), тобто повний опір входу є дуже високим та не впливає на зовнішнє підключення. У стані Hi-Z входи МК використовуються для підключення до сторонніх шин даних, не заважаючи при цьому їхній роботі.

PINx – ці регістри призначені лише для читання та, незалежно від налаштування портів вводу/виводу, завжди відображають поточні стани виводів МК. Для стабільності зчитування станів виводів МК є присутня синхронізуюча ланка, що складається з тригерної заскочки та розряду PINxn (рис. 1). Значення сигналу на виводі МК фіксується заскочкою при низькому рівні сигналу та перезаписується потім у розряд PINxn з наростаючим фронтом тактового сигналу. Відповідно, між операціями запису у порт та зчитуванням його стану є присутня затримка. Тому у програмах між командами out та in одного порту необхідно вставляти пустий оператор nop.

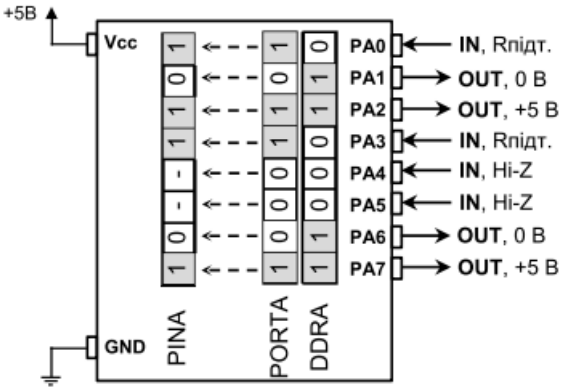


Рис. 2. Налаштування виводів порту А та значення регістра PINA

Значення окремих бітів регістрів PINx відповідають реальним рівням сигналів на виводах портів. «1» – при високому рівні, «0» – при низькому рівні. Якщо виводи налаштовані як високоімпедансні входи, та немає підключення зовнішніх сигналів, тоді значення відповідних бітів PINx будуть рівними «0». Однак, від найменших завад, наприклад дотик пальця до корпусу МК, на виході може з'явитися «1». Тому на рис. 3.3 значення бітів для таких виводів мають прочерки «-». Саме по цій причині непідключені входи до шин сигналів необхідно підтягувати через резистор до напруги живлення чи землі.

Поділ за рівнями вхідних зовнішніх сигналів, у залежності від напруги живлення МК, має вигляд як на рис. 3.

Рис. 3.4. Розподіл рівнів відносно напруги живлення МК з врахуванням гістерезисної петлі тригерної заскочки

Документацією визначені гарантовані рівні сигналів:

- від -0,5В до 0,3·Vcc – низький рівень;
- від 0,7·Vcc до Vcc+0.5В – високий рівень.

Нижчі чи вищі за вказані рівні напруг можуть вивести з ладу вивід МК. На границі рівнів (рис. 3) присутня так звана гістерезисна петля. Перехід від низького рівня до високого відбувається при одній напрузі , а зворотній перехід, від високого до низького рівня, при дещо нижчій напрузі. Ця гістерезисна петля є своєрідним механізмом захисту від переключення рівнів під дією завад. У документації розмах величини гістерезисної петлі визначений 0,05· Vcc. Середня ділянка рівнів дещо відрізняється для різних моделей та партії випуску.

Експериментальні зняття показів з декількох ATmega32A при напрузі живлення +5 В встановили такий усереднений діапазон розмаху гістерезисної петлі: 2,36-2,52 В. Тобто, перехід від низького до високого рівня встановлюється при напрузі 2,52 В, а від високого до низького при 2,36 В.

При підключенні різноманітних кнопок та клавіатур до МК для уникнення дії завад необхідно задіювати підтягуючі резистори до напруги живлення чи землі. Найпростіше налаштувати вхід на використання внутрішнього підтягуючого резистора до напруги живлення.

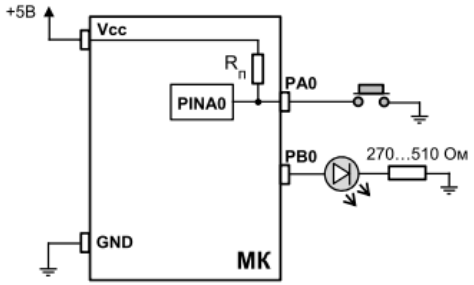


Рис. 4. Підключення кнопки та світлодіода до МК

Покажемо на прикладі (рис. 4) програмну реалізацію підключення кнопки та світлодіода до МК.

```
.include "m32Adef.inc"

        .CSEG
        ldi     r16, 0x00
        ldi     r17, 0xFF
        ;Порт A на вхід з підтягуючим резистором
        out     DDRA, r16
        out     PORTA, r17
        ;Порт B на вихід з низьким початковим рівнем
        out     DDRB, r17
        out     PORTB, r16

main:
        in      r16, PINA
        out     PORTB, r16
        rjmp    main
```

На початку програми ми виконуємо налаштування портів, до яких підключені зовнішні пристрої. Кнопка під'єднана до нульового виводу порту А, світлодіод до нульового виводу порту В. Увесь порт А ми налаштували на вхід та включили внутрішні підтягуючі резистори. Порт В налаштували на вихід. В основному програмному циклі ми зчитуємо стан входів усього порту А та заносимо цей байт стану у регістр загального призначення r16.

На наступному такті програми значення r16 виводимо на виходи порту B. Якщо кнопка не натиснута, тобто на вхід не під'єднана зовнішня земля, то стан виводу має значення «1», оскільки внутрішні підтягуючі резистори подають високий рівень. Натисканням кнопки ми шунтуємо підтягуючий резистор, подаючи безпосередньо землю на вхід, і отримуємо значення «0».

Електричні характеристики системи вводу-виводу. Максимальне абсолютне значення допустимого струму для однієї ноги системи вводу/виводу складає 40 мА. Але рекомендованим є значення, що не перевищує 20 мА при Vcc = 5 В та 10 мА при Vcc = 3 В. Однак загальний струм, що протікає через усі виводи МК не повинен перевищувати:

- 200 мА для МК AVR у DIP корпусі;
- 400 мА для МК AVR у TQFP та QFN корпусах.

Також обмеження накладаються на окремі порти. Для моделі ATmega32A ситуація є такою:

- DIP корпус (200 мА):
  - 100 мА для порту A;
  - 100 мА для решти B, C, D портів.
- TQFP чи QFN корпус (400 мА):
  - 100 мА для порту A;
  - 100 мА для виводів B0-B4
  - 100 мА для виводів B3-B7 та D0-D2; ○ 100 мА для виводів D3-D7;
  - 100 мА для порту C.

Отже при підключенні пристроїв до портів вводу/виводу необхідно чітко прораховувати кількість струму, що буде протікати через ноги МК та через які саме.

Оскільки виводи МК AVR забезпечують високу навантажувальну здатність при любому рівні сигналу , то до них можна безпосередньо підключати світлодіодну індикацію.

Вольт-амперна характеристика світлодіода така ж як у звичайних діодів (рис. 5). Для уникнення виходу з ладу ноги МК та світлодіода необхідно виконувати підключення світлодіодів через обмежувальні резистори. Визначення опору цього резистора здійснюють, виходячи зі струму, який має споживати світлодіод. Наприклад, для нашого FYL-3014LURC1A виберемо робочий струм 10 мА. На ВАХ цьому струму відповідає напруга 2,1 В.

$$R_{ОБМЕЖ} = (V_{CC} - V_{LED}) / I_{LED} = (5 - 2{,}1) / 0{,}01 = 290 \text{ Ом}$$

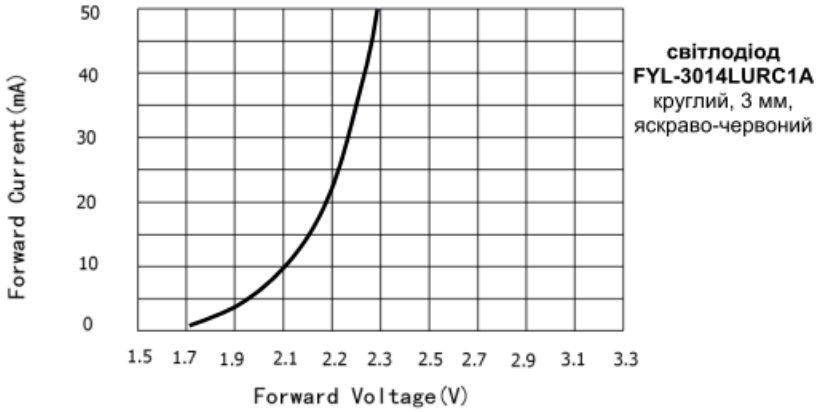


Рис. 5. Вольт-амперна характеристика світлодіода

Варіанти підключення кнопок та світлодіодів до МК. При підключенні світлодіодів необхідно чітко дотримуватися полярності: високий потенціал (+) до анода діода, низький потенціал (–) до катода. Відповідно до цього, ми можемо по різному підключати світлодіоди: або до зовнішньої додатної напруги та засвічувати світлодіод низьким рівнем («0») з виходу МК, або до зовнішньої землі та засвічувати високим рівнем з виходу МК (рис. 6 а). Вибір одного з цих двох способів підключення найчастіше обумовлюється зручністю підведення землі чи додатної напруги до світлодіода.

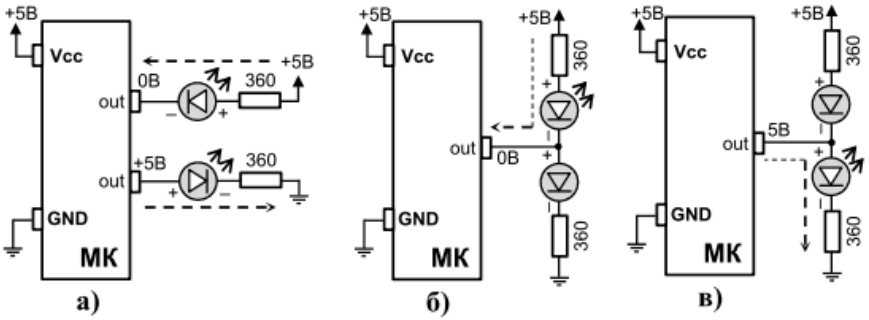


Рис. 6. Підключення світлодіодів

Для деяких задач необхідно почергово засвічувати два світлодіоди, наприклад, червоний світлодіод – канал передає дані, зелений –

канал відключений. Тоді їх можемо підключити разом до одного виводу МК, як на рис. 6. Подавши «0» на вихід – засвітимо верхній світлодіод (б), подавши «1» – засвітимо нижній (в). Якщо цей вивід МК налаштуємо як високоімпедансний вхід, тоді струм потече від додатної напруги до землі через обидва світлодіоди, і вони будуть обидва світитися, але дещо тьмяніше.

Для забезпечення завадостійкості при підключенні кнопок до виводів МК необхідно використовувати зовнішній підтягуючий резистор, оскільки у деяких випадках внутрішній резистор може не справлятися із наявними завадами. При цьому вивід можемо підтягувати як до напруги живлення, так і до землі . Відповідно, кнопки мають бути підключеними до землі та напруги живлення (рис. 7 а). У другому випадку (з підтягуючим резистором до землі) логіка входу є прямою: при відпущеній кнопці – «0», при натисненій – «1».

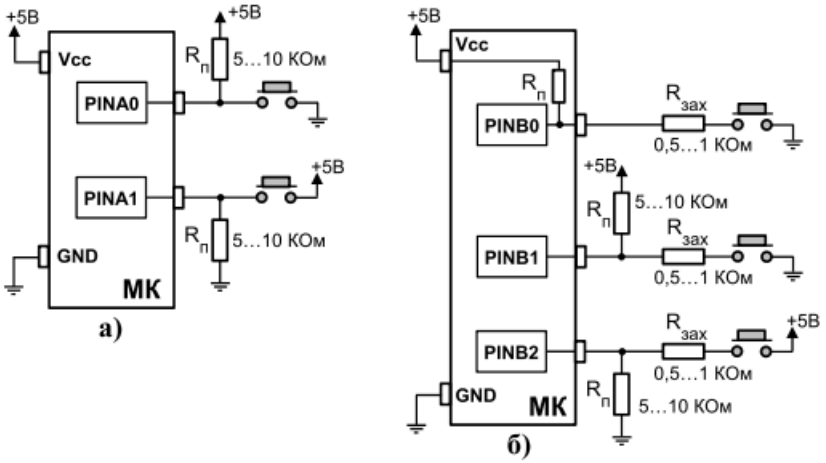


Рис. 7. Підключення кнопок

Для запобігання аварій через невірне налаштування виводів портів (на вихід, замість того, щоб на вхід), до яких підключені кнопки, необхідно підключати послідовно з кнопками захисні резистори (рис. 7 б).

Існує чимало різноманітних варіацій підключення кнопок та світлодіодів до виводів МК: 3 кнопки до 2 ніг, 6 чи 7 кнопок до 3 ніг, кнопка та світлодіод на 1 нозі, багато кнопок на 1 вхід АЦП і т.п.

Умовні та безумовні переходи та регістр стану SREG.

МК AVR мають три **команди безумовного переходу**:

jmp – абсолютний (прямий) перехід (direct jump);
rjmp – відносний перехід (relative jump);
ijmp – непрямий перехід (indirect jump to (Z)).

Покажемо на прикладі виконання команд безумовного переходу.

```
.include "m32Adef.inc"
        .CSEG
        .org    $085            ;встановлення програмної адреси
main:   nop                     ;пустий оператор
        ldi     r16, 0x25       ;завантаження в r16 значення
        jmp     main            ;абсолютний перехід

        nop                     ;пустий оператор
        rjmp    main            ;відносний перехід

        ldi     ZL, Low(main)   ;завантаження в регістрову пару Z
        ldi     ZH, High(main)  ;адреси переходу
        ijmp                    ;непрямий перехід
```

Згенерований дисасемблером код програми

```
+00000085: 0000     NOP                     No operation
+00000086: E205     LDI     R16,0x25        Load immediate
+00000087: 940C0085 JMP     0x00000085      Jump

+00000089: 0000     NOP                     No operation
+0000008A: CFFA     RJMP    PC-0x0005       Relative jump

+0000008B: E8E5     LDI     R30,0x85        Load immediate
+0000008C: E0F0     LDI     R31,0x00        Load immediate
+0000008D: 9409     IJMP                    Indirect jump to (Z)
```

адреса у пам'яті програм	*код команди*	*назва команди*	*аргументи*	*коментар*

Для команди абсолютного переходу **jmp** ми задаємо в її аргументі числове значення адреси у FLASH, туди, куди хочемо виконати перехід. Найпростіше у потрібному місці коду програми поставити мітку і вказати її в аргументі команди. Тоді компілятор собі сам визначить значення адреси, згідно вказаної мітки. У наведеному

прикладі ми переходимо командою jmp на адресу мітки main і, якщо зазирнути у згенерований дизасемблером код, ми бачимо, що там вже підставлене значення 0x00000085, яке відповідає адресі стрічки, на яку посилається мітка. Код команди jmp займає у програмі 4 байти (32 біти): 10 біт під саму команду переходу та 22 біти під адресу, куди має виконатися перехід. Команда jmp може адресувати у будь-яку точку програми, обсягом до 4М слів (макс. адреса 0x3FFFFF). Для усіх моделей AVR розмір FLASH наразі не має таких великих об'ємів. 2 байтів достатньо для адресації 128 Кбайтів FLASH. Варто зазначити, що старші біти адреси є частково перемішані з бітами коду команди, але 2 молодші байти є розташовані разом. Тому в дисасемблері ми бачимо загальний код для команди переходу з вказаною адресою як 940C0085, де 0085 і є адресою переходу. Команда jmp виконується за 3 такти МК.

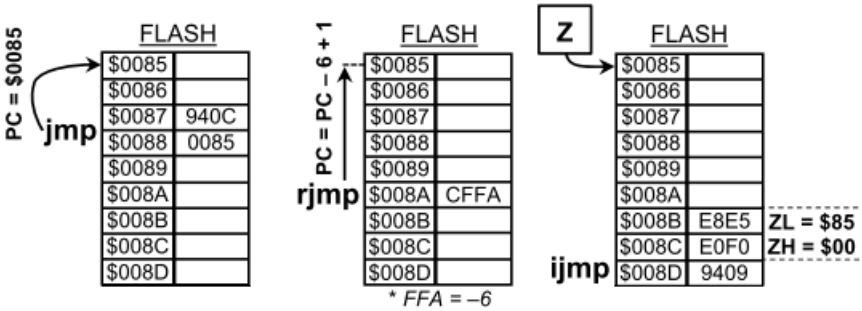


Рис. 8. Ілюстрація роботи команд для безумовного переходу

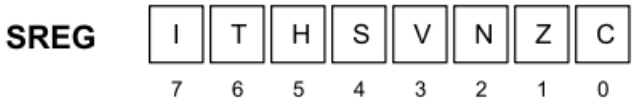
Команда відносного переходу **rjmp** виконує перехід, здійснюючи зміщення вперед чи назад у програмній пам'яті на вказане число відносно значення поточної адреси (програмного лічильника). Код команди зі значенням зміщення займає 2 байти (16 біт), 12 бітів з яких відведені під значення зміщення. Тому значення зміщення адреси програмного лічильника може приймати значення -2047…+2048 у пам'яті програм. У програмному коді як аргументи вказують мітки переходу. Компілятор сам обрахує необхідне значення величини зміщення та підставить у слово команди. Тому великої різниці між командами jmp та rjmp немає, лише в обмеженому діапазоні адресації переходу. Зате команда rjmp виконується за 2 такти.

Команда непрямого переходу **ijmp** виконує перехід за адресою, що розміщена в індексному регістрі Z. Код команди ijmp займає у програмі 2 байти, а команда виконується за 2 такти МК. Оскільки

індексний регістр Z двобайтний, то об'єм FLASH для цієї команди обмежується 128 кБайтами. Завдяки команді ijmp ми можемо

програмно змінювати нашу точку переходу, що робить програму гнучкішою.

Перед розглядом команд умовного переходу розберемо призначення окремих прапорців **регістру стану SREG**. Ці прапорці встановлюються у залежності від результату виконання арифметичних, логічних та порозрядних операцій над регістрами загального призначення в ALU.



C – прапорець переносу вказує на перенос чи позику при виконанні арифметичної чи логічної операції, тобто тоді, коли відбувся вихід за межі байта.

Z – прапорець нуля вказує на нульовий результат при виконані арифметичної чи логічної операції.

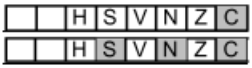
N – прапорець від'ємного результату вказує на від'ємний результат при виконані арифметичної чи логічної операції. Тобто в старшому розряді присутня «1». Цей прапорець має на увазі, що ми працюємо зі знаковими числами в діапазоні від -128 до 127. Ми вже раніше звертали увагу на те, що числа -25 та 231 для 8-ми розрядного асемблера є ідентичними, оскільки у двійковому представленні вони виглядають однаково: 0b11100111.

V – прапорець переповнення у доповняльному коді, тобто коли відбувається вихід за межі [-128; 127].

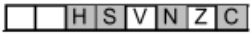
S – прапорець знаку є результатом суми за модулем 2 (виключне або) між прапорцями N та V ($S=N \oplus V$). Він з'являється, якщо при обчисленні чисел зі знаком результатом є від'ємне число, і при цьому не було переповнення у доповняльному коді, або отримуємо додатне число у результаті переповнення у доповняльному коді. В основному він використовується при порівнянні знакових чисел.

H – прапорець часткового переносу вказує на перенос чи позику між старшою половиною байта та молодшою при виконанні деяких арифметичних операцій. Частковий перенос є корисний у двійково-десятковій арифметиці.

```
ldi     r16, 120        ;r16=120
ldi     r17, 150        ;r17=150=-106
add     r16, r17        ;r16=120+150=120-106=14
subi    r16, 20         ;r16=14-20=-6=250
```



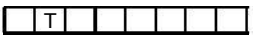

```
ldi     r16, 0b0000_0111        ;r16=7
ldi     r17, 0b0000_1100        ;r17=12
sub     r16, r17        ;r16=7-12=-5=251=0b1111_1011
```



T – користувацький біт для роботи з бітами регістрів загального призначення у командах bld (bit load) та bst (bit store). Команда bst копіює значення вказаного біта із зазначеного регістра загального призначення у біт T регістра стану SREG. За допомогою команди bld можна навпаки, з біта T встановити значення у вказаному біті зазначеного регістра загального призначення. Це є стандартний підхід для встановлення/скинення необхідного біта у регістрі загального призначення. Попереднього біт T встановлюється/скидається за допомогою призначених для цього команд set/clt.

```
set             ;встановлюємо біт T
bld     r20, 5  ;встановлюємо 5-й біт у r20
```


```
clt             ;очищуємо біт T
bld     r20, 5  ;очищуємо 5-й біт у r20
```



I – прапорець дозволу глобальних переривань. Коли він встановлений в «0» – будь-які переривання призупинені, коли в «1» – дозволені переривання знову активні . Команда sei – надає загальний дозвіл на переривання, cli – забороняє будь-які переривання.

Кожен з цих прапорців регістру стану SREG ми також можемо встановити/ скинути за допомогою призначених для цього команд. Для прапорця C – sec/clc; для Z – sez/cln; для N – sen/cln і т.д.

Команди умовного переходу виконують перехід, опираючись на значення одного з бітів регістра стану SREG. Тобто, для кожного прапорця стану є визначені свої команди переходу, у залежності від його значення. Якщо визначений прапорець відповідає зазначеній умові (встановлений чи обнулений), відбувається перехід у визначене місце у програмі, якщо ні – виконується наступна команда. Ці всі команди утворюють групу Branch, і мають однаковий принцип дії.

Назва команди розпочинається 2-ма літерами br_{xx}, а дві останні літери команди визначають логіку переходу.

```
        ldi     r16, 0b00000001
shift:  lsl     r16         ;логічний зсув вліво бітів r16
        brhs    one         ;якщо прапор H=1 → перехід на мітку one
if_cc:  brcc    shift       ;якщо прапор C=0 → перехід на мітку shift
        rjmp    finish      ;перехід на мітку finish

one:    set                 ;встановлюємо біт T=1
        bld     r16, 0      ;встановлюємо 0-й біт r16 в «1»
        rjmp    if_cc       ;перехід на мітку if_cc

finish: nop
        nop
```

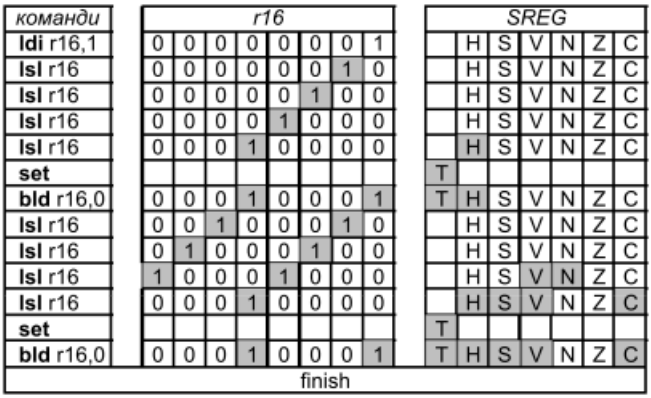
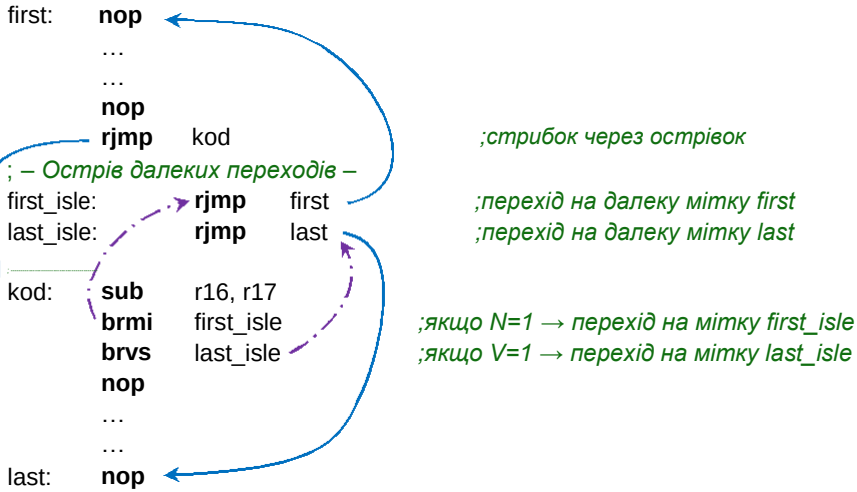


Рис. 9. Ілюстрація роботи умовних переходів

Команди переходу групи Branch подібні до команди rjmp тим, що виконують перехід, здійснюючи зміщення вперед чи назад у програмній пам'яті на вказане число відносно значення поточної адреси (програмного лічильника). Коди команд групи Branch займають 2 байти (16 бітів), 7 бітів з яких відведені під значення зміщення, яке, відповідно, може приймати значення -64…63. Команди групи Branch виконують переходи на близькі віддалі, і тому для збільшення їхньої дистанції розміщують неподалік посеред програмного коду проміжні острівки з командами jmp чи rjmp для переходу на віддалені адреси у пам'яті програм. Команди групи Branch виконуються за 1 такт, якщо умова не справджується, і продовжує виконання наступна в програмі команда, за 2 такти, якщо виконується перехід.



Для організації умов для команд умовного переходу , наприклад чи рівні між собою 2 регістри загального призначення, використовуються **команди групи Compare**. Їхня дія подібна команді віднімання: вони віднімають одне значення від іншого, виставляють прапорці регістра стану SREG, але отриманий числовий результат нікуди не заносять. Таким чином, ми можемо виконати перехід Branch на основі порівняння двох значень: вони рівні, одне більше за інше і т.п. Асемблер містить 4 команди групи Compare:

cp – порівняння (compare) → виставляє прапорці на основі результату Rd–Rr;

cpc – порівняння з врахуванням переносу (compare with carry) → виставляє прапорці на основі результату Rd–Rr–C;

cpi – порівняння регістра з константою (compare register with Immediate) → виставляє прапорці на основі результату Rd–K;

cpse – порівнює Rd=Rr, якщо вони рівні, тоді пропускає наступну команду (compare, skip if equal);

```
        cp    r16, r17    ;r16-r17 → виставлення прапорців SREG
        brcs  first       ;якщо r16<r17 (C=1) → перехід на мітку first

        cpi   r16, 40     ;r16-40  → виставлення прапорців SREG
        breq  second      ;якщо r16=40 (Z=1)→ перехід на мітку second

first:  nop
second: nop
```

Перші три команди групи Compare виконуються за 1 такт, остання команда cpse – за 1 такт, якщо умова не виконується, а якщо виконується, тоді: 2 такти, при розмірі команди, що пропускається, 1 слово, та 3 такти, при розмірі команди, що пропускається, 2 слова.

На основі поєднання команд Compare та Branch можна утворювати **умовні конструкції** типу if(умова) … else, як у мові Сі.

Таблиця 1. Умовні конструкції на основі команди cp

Умова	**Знакові числа** [-128; 127]		**Беззнакові числа** [0; 255]	
	SREG	**Команди**	**SREG**	**Команди**
Rd = Rr	Z = 1	**cp** Rd, Rr **breq** <мітка>	Z = 1	**cp** Rd, Rr **breq** <мітка>
Rd ≠ Rr	Z = 0	**cp** Rd, Rr **brne** <мітка>	Z = 0	**cp** Rd, Rr **brne** <мітка>
Rd < Rr	S = 1	**cp** Rd, Rr **brlt** <мітка>	C = 1	**cp** Rd, Rr **brcs** <мітка>
Rd >= Rr	S = 0	**cp** Rd, Rr **brge** <мітка>	C = 0	**cp** Rd, Rr **brcc** <мітка>
Rd > Rr *альтерн.ум.* *(Rr < Rd)***	Z = 0 та S = 0 *(S = 1)***	**cp** Rd, Rr **breq** next **brge** <мітка> next: ***cp*** *Rr, Rd* ***brlt*** *<мітка>*	Z = 0 та C = 0 *(C = 1)***	**cp** Rd, Rr **breq** next **brcc** <мітка> next: ***cp*** *Rr, Rd* ***brcs*** *<мітка>*
Rd <= Rr *альтерн.ум.* *(Rr >= Rd)***	Z = 1 або S = 1 *(S = 0)***	**cp** Rd, Rr **breq** <мітка> **brlt** <мітка> ***cp*** *Rr, Rd* ***brge*** *<мітка>*	Z = 1 або C = 1 *(C = 0)***	**cp** Rd, Rr **breq** <мітка> **brcs** <мітка> ***cp*** *Rr, Rd* ***brcc*** *<мітка>*

* $S=N \oplus V$

** *Для переходу за цією умовою операнди команди порівняння повинні бути записані у зворотному порядку, тобто замість* ***cp*** *Rd, Rr →* ***cp*** *Rr, Rd*

Варто мати на увазі, що умови порівняння для знакових та беззнакових типів використовують різні прапорці регістру стану SREG. Наприклад, порівнюючи числа - 46(210) та 36(36), у нас перше число як знакове є меншим, бо -46 < 36, але як беззнакове воно більше, бо 210 > 36.

Покажемо на прикладі асемблерну реалізацію умовної конструкції з умовою (r16<r17), де вхідні числа є знаковими. Для цього виберемо з таблиці 3.7 необхідний нам варіант.

Rd < Rr	S = 1	**cp** Rd, Rr **brlt** <мітка>

```
        ldi     r16, -46
        ldi     r17, 36
;----- if(r16<r17) ---знакові-------
        cp      r16, r17
        brlt    then_block
        rjmp    else_block
;----- then ----------------------
then_block:
        nop
        nop
        rjmp    end_if
;----- esle ----------------------
else_block:
        nop
        nop
;----- end if --------------------
end_if:
        nop
```

;якщо r16<r17 → перехід на мітку then_block

У табл. 3.7 дві останні умови мають альтернативні варіанти рішення за рахунок перестановки операндів. Однак для команди cpi (порівняння регістра з константою) таку перестановку зробити не можна. У такому випадку слід реалізовувати основний варіант.

```
        ldi     r16, 55
;----- if(r16>const)  --беззнакові-
        cpi     r16, 50
        breq    else_block
        brcc    then_block
;----- esle ----------------------
else_block:
        nop
        nop
        rjmp    end_if
;----- then ----------------------
then_block:
        nop
        nop
;----- end if block --------------
end_if:
        nop
```

;якщо r16=K → перехід на мітку else_block
;якщо r16>=K → перехід на мітку then_block

Ще одним різновидом умовних команд є **команди групи Skip**. Вони призначені для перевірки окремих бітів регістрів загального призначення R0-R31 та молодших 32 регістрів вводу виводу з адресами у межах $0-$1F (Це регістри портів A, B, C та D, SPI, TWI, EEPROM; деякі з регістрів USART, АЦП, аналогового компаратора).

Принцип роботи команд групи Skip полягає у перевірці вказаних бітів у зазначених регістрах , і якщо відповідний біт є встановлений чи очищений, у залежності від умови команди skip, тоді наступна коман-да пропускається. Є 4 таких команди:

sbrc – пропустити наступну команду , якщо вказаний біт в регістрі заг. призначення дор. «0» (skip if bit in register cleared);

sbrs – пропустити наступну команду , якщо вказаний біт в регістрі заг. призначення дор. «1» (skip if bit in register is set);

sbic – пропустити наступну команду, якщо вказаний біт в регістрі вводу/виводу дор. «0» (skip if bit in I/O register cleared);

sbis – пропустити наступну команду, якщо вказаний біт в регістрі вводу/виводу дор. «1» (skip if bit in I/O register is set).

Перед тим ми ще розглядали команду cpse з групи Compare, яка за своєю дією подібна до них.

Якщо команди Compare та Branch використовувалися для порівняння числових значень, то команди Skip використовуються для організації логіки на основі значень окремих бітів. Наприклад, ми можемо на основі одного з регістрів загального призначення організувати собі регістр програмних прапорців.

```
.def    _flags = r10          ; _flags  f7 f6 f5 f4 f3 f2 f1 f0
                                        7  6  5  4  3  2  1  0
        .CSEG
        clt
        bld     _flags, 0
        set
        bld     _flags, 1     ; _flags  f7 f6 f5 f4 f3 f2 f1 f0
        …                               7  6  5  4  3  2  1  0
;---------- if (f0==1) ----------------
        sbrs    _flags, 0
        rjmp    f0_end
;---------- then ----------------------
        nop
f0_end:
;---------- if (f1==1) ----------------
        sbrs    _flags, 1
        rjmp    f1_end
;---------- then ----------------------
        nop
f1_end:
```

Приклад 1. Розрахунок програмної затримки. Для реалізації найпростішої затримки завантажуємо процесор МК циклічною роботою з умовою та рахуємо такти. Найлегше виконувати віднімання 1 від якогось числа та перевіряти на нульовий результат. Якщо необхідно більшу затримку, тоді задіюють декілька регістрів та використовують операції віднімання з врахуванням переносу.

```
        ldi   r16, 2          ; 1 такт
        ldi   r17, 2          ; 1 такт

delay:  subi  r16, 1          ; 1 такт
        sbci  r17, 0          ; 1 такт
        brne  delay           ; 2 такт (1 – коли нема переходу)
        nop
```

Таблиця 2. Покрокова трансляція програми затримки

	команди	*R16*	R17	*прапорці*		*такти*	
	ldi r16,2	2		Z	C	1	1
	ldi r17,2	2	2	Z	C	1	2
delay:	**subi** r16,0	1	2	Z	C	1	3
	sbci r17,0	1	2	Z	C	1	4
	brne delay			Z	C	2	6
delay:	**subi** r16,0	0	2	Z	C	1	7
	sbci r17,0	0	2	Z	C	1	8
	brne delay			Z	C	2	10
delay:	**subi** r16,0	255	2	Z	C	1	11
	sbci r17,0	255	1	Z	C	1	12
	brne delay			Z	C	2	14
	…	254…2					…
delay:	**subi** r16,0	1	1	Z	C	1	1027
	sbci r17,0	1	1	Z	C	1	1028
	brne delay			Z	C	2	1030
delay:	**subi** r16,0	0	1	Z	C	1	1031
	sbci r17,0	0	1	Z	C	1	1032
	brne delay			Z	C	2	1034
delay:	**subi** r16,0	255	1	Z	C	1	1035
	sbci r17,0	255	0	Z	C	1	1036
	brne delay			Z	C	2	1038
	…	254…2					…
delay:	**subi** r16,0	1	0	Z	C	1	2051
	sbci r17,0	1	0	Z	C	1	2052
	brne delay			Z	C	2	2054
delay:	**subi** r16,0	0	0	Z	C	1	2055
	sbci r17,0	0	0	Z	C	1	2056
	brne delay			Z	C	1	2057
	nop						

У регістрах R16 та R17 ми записали число 0x0202 (514). Один цикл віднімання займає у нас 4 такти. Отже розрахунок отримується так:

$$SumCycle = Ncycle \times Value - 1 + Nldi$$

де Cycle – кількість тактів у циклі; Value – число, записане у регістрах; Nldi – кількість команд ldi для запису значень у регістри; SumCycle – сумарна кількість тактів затримки.

$$SumCycle = 4 \times 514 - 1 + 2 = 2057$$

Зворотна задача, визначення необхідного числа, виглядає так:

$$Value = (SumCycle - Nldi + 1)/ Ncycle$$

Обчислимо необхідну кількість тактів для 1 мсек:

$$SumCycle = час * частоту\ тактування = 0{,}001 * 8*10^6 = 8000$$

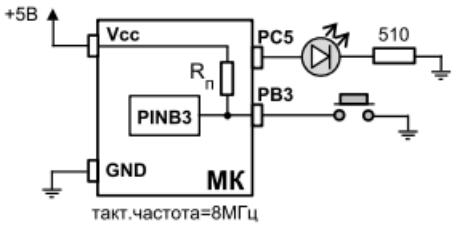
$$Value = (8000 - 2 + 1)/ 4 = 1999{,}75 = \sim 2000 = 0x07D0$$

Оскільки ми виконали заокруглення, то реальна кількість тактів становить 8001, а тривалість затримки 0,001000125 сек., що є цілком прийнятно.

Таблиця 3.9. Вибір кількості байтів для розрахунку тривалості

К-сть байтів	Макс. к-сть тактів	Макс. тривалість
1	765	95,625 мкСек
2	262 141	~32,76 мСек
3	83 886 077	~10,48 Сек
4	25 769 803 773	~3 221 Сек

Приклад 2. Блимаючий світлодіод. У пасивному режимі світлодіод блимає з інтервалом приблизно 1 сек., тобто 1 сек. світиться, а на 1 сек. гасне. При натисненні кнопки, світлодіод починає блимати у 2 рази частіше.



Розрахуємо значення числа для затримки в 1 сек. з використанням 3 регістрів загального призначення.

$$Value_{1сек} = (8 \cdot 10^6 - 3 + 1)/5 = \sim 1{,}6 \cdot 10^6 = 0x186A00$$

Значення числа для затримки в 0,5 сек. буде у 2 рази меншим

$$Value_{0,5сек} = 0{,}8 \cdot 10^6 = 0x0C3500.$$

```
.include "m32Adef.inc"
.def    _temp1 = r16
.def    _temp2 = r17
.def    _temp3 = r18

        .CSEG
        ldi     _temp1, 0x00
        ldi     _temp2, 0xFF
        ;Порт B на вхід з підтягуючим резистором
        out     DDRB, _temp1
        out     PORTB, _temp2
        ;Порт C на вихід
        out     DDRC, _temp2
        out     PORTC, _temp1
main:
        ;Інверсія виходу PC5 світлодіода
        sbic    PORTC, 5
        rjmp    elsePC5
        sbi     PORTC, 5
        rjmp    skipPC5end
elsePC5:
        cbi     PORTC, 5
skipPC5end:
        ;Вибір значення затримки в залежності чи натиснута кнопка PB3
        sbis    PINB, 3
        rjmp    elsePB3
        ldi     _temp1, 0x00
        ldi     _temp2, 0x6A
        ldi     _temp3, 0x18
        rjmp    delay
elsePB3:
        ldi     _temp1, 0x00
        ldi     _temp2, 0x35
        ldi     _temp3, 0x0C

delay:  subi    _temp1, 1
        sbci    _temp2, 0
        sbci    _temp3, 0
        brne    delay

        rjmp    main
```