Київський національний університет імені Тараса Шевченка


**Пархоменко Д. А., Смирнов Є. М.**

# Розробка радіоелектронних схем на основі мікроконтролерів (на прикладі AVR мікроконтролерів фірми Atmel)

**Методичний посібник до курсу "Проектування радіоелектронних схем" для студентів радіофізичного факультету**

Київ 2013

УДК 621.3

**Розробка радіолектронних схем на основі мікроконтролерів (на прикладі AVR мікроконтролерів фірми Atmel)**: методичний посібник до курсу "Проектування радіоелектронних схем" для студентів радіофізичного факультету / Пархоменко Д. А., Смирнов Є. М. – Київ: Радіофізичний факультет Київського національного університету імені Тараса Шевченка, 2013. – 74 с.

**Рецензент**: доцент кафедри радіотехніки та радіоелектронних систем Київського національного університету імені Тараса Шевченка, кандидат фізико-математичних наук Слюсаренко І. І.

Методичний посібник до практичних занять складений у відповідності з навчальною програмою дисципліни "Проектування радіоелектронних схем", що викладається на кафедрі квантової радіофізики радіофізичного факультету. В посібнику наведені короткі відомості про структуру та систему команд AVR мікроконтролерів фірми Atmel, а також вступ до застосування інтегрованого відлагоджувального середовища AVR Studio фірми Atmel. Наведений опис та принципова схема експериментального макета, що дозволяє студентам засвоїти операції по обробці сигналів вводу та динамічної індикації на основі мікроконтролера AT90S4433-8PI. В додатку для полегшення роботи студентів наведені приклади програм мовою Ассемблер. Посібник призначений для самостійного вивчення даної теми студентами, що обрали її як частину курсу за вибором.

## Вступ

На сьогоднішній день одним з найбільш потужних і гнучких засобів розробки електронних схем є мікроконтролери. При роботі з мікроконтролерами необхідно мати на увазі таку обставину - коли розробляється система на основі мікроконтролера, то створюються не тільки апаратні засоби, що реалізуються відповідним підключенням мікроконтролера до зовнішніх пристроїв. Окрім цього розробник повинен забезпечити виконання багатьох системних функцій, які в традиційних мікропроцесорних системах забезпечуються за допомогою операційної системи та спеціальних периферійних мікросхем. Це, з одного боку, дещо ускладнює задачу, з іншого ж боку, дозволяє суттєво оптимізувати проект - як його апаратну, так і програмну частину для конкретного застосування.

Відмінності в архітектурі процесорів можуть істотно позначитися на їхній продуктивності при виконанні різних завдань. Дискусії про порівняльні переваги різних комп'ютерних архітектур можна вести до нескінченності. Щоб уникнути участі в цих дискусіях, обмежимося коротким оглядом найбільш важливих особливостей різних архітектур та вказівками, для яких найкращим чином підходить та чи інша архітектура.

Запропонований посібник призначений для самостійного ознайомлення із структурою мікроконтролера та його програмуванням в рамках частини курсу "Проектування радіоелектронних схем", що читається студентам 1 курсу магістратури на кафедрі квантової радіофізики Київського національного університету імені Тараса Шевченка.

## Типи мікроконтролерів

Всі мікроконтролери можна розділити на наступні основні типи:

- Вбудовані 8-розрядні мікроконтролери
- 16- і 32-розрядні мікроконтролери
- Цифрові сигнальні процесори (DSP)

### *Вбудовувані мікроконтролери*

Промисловістю випускається дуже широка номенклатура вбудовуваних (embedded) мікроконтролерів. У цих мікроконтролерах всі необхідні ресурси (пам'ять, пристрої вводу-виводу і т.д.) розташовуються на одному кристалі з процесорним ядром. Все, що необхідно зробити - це подати живлення і тактові сигнали. Вбудовувані мікроконтролери можуть базуватися на існуючому мікропроцесорному ядрі або на процесорі, розробленому спеціально для даного мікроконтролера. Це означає, що існує велика різноманітність функціонування навіть серед пристроїв, що виконують однакові завдання.

Основне призначення вбудованих мікроконтролерів - забезпечити за допомогою недорогих коштів гнучке (програмне) управління об'єктами і зв'язок із зовнішніми пристроями. Ці мікроконтролери не призначені для реалізації комплексу складних функцій, але вони здатні забезпечити ефективне управління в багатьох галузях застосування.

Вбудовувані мікроконтролери містять значну кількість допоміжних пристроїв, завдяки чому забезпечується їх включення в систему з використанням мінімальної кількості додаткових компонентів. До складу цих мікроконтролерів зазвичай входять:

- Схема початкового запуску процесора (Reset)
- Генератор тактових імпульсів
- Центральний процесор
- Пам’ять програм (E(E)P)ROM і програмний інтерфейс
- Пам’ять даних RAM
- Засоби вводу-виводу даних
- Таймери, які фіксують кількість командних циклів

Загальна структура мікроконтролера показана на рис. 1. Ця структура дає уявлення про те, як мікроконтролер зв'язується із зовнішнім світом.

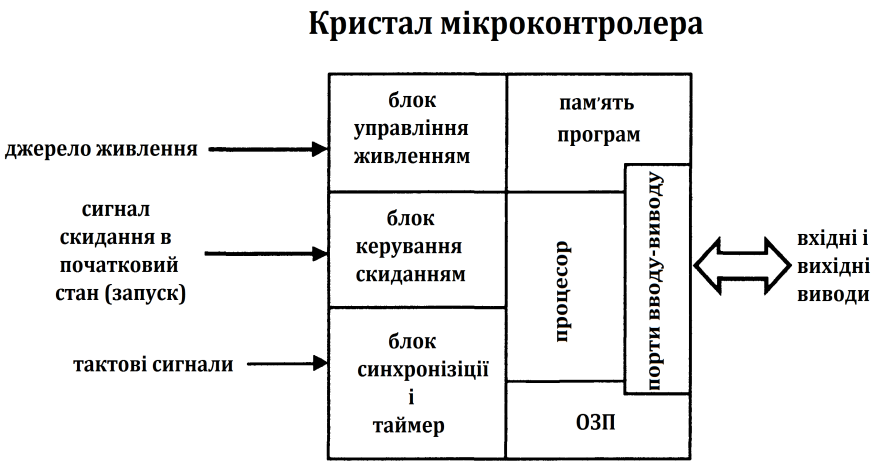



*Рис. 1. Структура мікроконтролера*

Більш складні вбудовувані мікроконтролери можуть додатково реалізувати наступні можливості:

- Вбудований монітор/наладчик програм
- Внутрішні засоби програмування пам’яті програм (ROM)
- Обробка переривань від різних джерел
- Аналоговий ввід-вивід
- Послідовний ввід-вивід (синхронний і асинхронний)
- Паралельний ввід-вивід (включаючи інтерфейс з комп’ютером)
- Підключення зовнішньої пам’яті (мікропроцесорний режим)

Всі ці можливості значно збільшують гнучкість застосування мікроконтролерів і роблять більш простим процес розробки систем на їх основі. Слід зауважити, що для реалізації цих можливостей в більшості випадків потрібне розширення функцій зовнішніх виводів.

Раніше мікроконтролери виготовлялися по біполярній або NMOS технології. Всі сучасні мікроконтролери виробляються за допомогою CMOS технології, яка дозволяє значно зменшити розмір кристала і розсіювану на ньому потужність.

Типові значення максимальної частоти тактових сигналів складають для різних мікроконтролерів 10 - 20 МГц. Головним чинником, що обмежує їх швидкість, є час доступу до пам'яті, яка використовується в мікроконтролерах. Однак для типових застосувань це обмеження не є суттєвим.

### *Мікроконтролер із зовнішньою пам’яттю*

Деякі мікроконтролери (особливо 16- і 32-розрядні) використовують тільки зовнішню пам’ять, яка включає в себе як пам’ять програм (ROM), так і деякий об’єм пам’яті даних (RAM), необхідний для даного застосування. Структура мікроконтролера із зовнішньою пам’яттю показана на рис. 2.

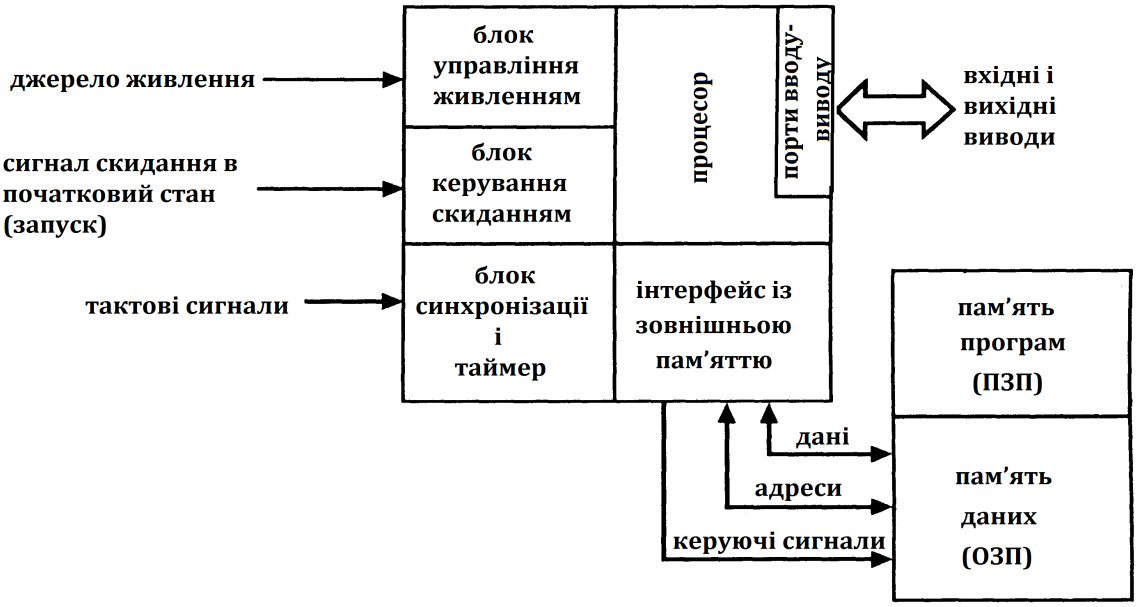



*Рис. 2. Блок схема мікроконтролера із зовнішньою пам'яттю.*

Класичним прикладом такого мікроконтролера є Intel 80188. По суті він являє собою мікропроцесор 8088, який використовувався в комп'ютерах IBM PC, інтегрований на загальному кристалі з додатковими схемами, що реалізують ряд стандартних функцій, таких як переривання і прямий доступ до пам'яті (DMA). Мета створення 80188 полягала в тому, щоб об'єднати в одному корпусі всі пристрої, необхідні інженеру для реалізації систем, в яких можуть використовуватися функціональні можливості та програмне забезпечення мікропроцесора 8088.

Аналогічні цілі досягаються при використанні мікроконтролера 80186, який має 16-розрядну зовнішню шину (80188 має 8-розрядну зовнішню шину) і являє собою 16-розрядний процесор 8086, інтегрований на загальному кристалі з додатковими периферійними схемами (такими ж, як у 80188). Так як мікропроцесор 8088 є спрощеною (8-розрядна зовнішня і 16-розрядна внутрішня шина) версією 8086 (16-розрядні зовнішня і внутрішня шини), так і мікроконтролер 80188 є спрощеною версією 80186.

Мікроконтролери з зовнішньою пам'яттю призначені для інших застосувань, ніж вбудовувані мікроконтролери. Ці застосування зазвичай вимагають великого обсягу пам'яті (RAM) і невеликої кількості пристроїв (портів) вводу-виводу. Для мікроконтролерів з зовнішньою пам'яттю найбільш підходящими є доповнення, у яких критичним ресурсом є пам'ять, а не число логічних вводів-виводів загального призначення, тоді як для вбудованих мікроконтролерів має місце протилежна ситуація. Типовим прикладом застосування для мікроконтролера з зовнішньою пам'яттю є контролер жорсткого диска з буферною кеш-пам'яттю, який забезпечує проміжне зберігання і розподіл великих обсягів даних (зазвичай вимірюються в мегабайтах). Зовнішня пам'ять дає можливість такому мікроконтролеру працювати з більш високою швидкістю, ніж вбудований мікроконтролер.

### *Цифрові сигнальні процесори*

Цифрові сигнальні процесори (DSP) - відносно нова категорія процесорів. Призначення DSP полягає в тому, щоб отримувати поточні дані від аналогової системи і формувати відповідний відгук. DSP і їх ALU (Arithmetic Logic Unit - арифметико-логічний пристрій, який є апаратним засобом для виконання обчислень) працюють з дуже високою швидкістю, що дозволяє здійснювати обробку даних в реальному масштабі часу. DSP часто використовуються в активних шумопригнічуючих мікрофонах, які встановлюються в літаках (другий мікрофон забезпечує сигнал навколишнього шуму, який вираховується з сигналу першого мікрофона, дозволяючи таким чином придушити шум і залишити тільки голос) або для придушення роздвоєння зображення в телевізійних сигналах.

Розробка DSP алгоритмів - це спеціальний розділ теорії керування. Викладка цієї теорії вимагає використання вельми поглибленої математики.

У різноманітних DSP можна знайти особливості, властиві як вбудовуваним мікроконтролерам, так і мікроконтролерам з зовнішньою пам'яттю. DSP не призначені для автономного застосування. Зазвичай вони входять до складу систем, в якості пристроїв керування зовнішнім устаткуванням, а також для обробки вхідних сигналів і формування відповідного відгуку.

## Архітектура процесорів

Як вказано було вище, не будемо вдаватися в дискусію щодо того, яка з архітектур краще - CISC чи RISC, Гарвардська чи Прінстонська. Розглянемо відмінності між цими архітектурами і покажемо, яке відношення вони мають до мікроконтролерів.

### *CISC проти RISC*

В даний час існує безліч RISC (Reduced Instruction Set Computers - комп'ютери із скороченою системою команд) процесорів, так як склалася думка, що RISC швидше ніж CISC (Complex Instruction Set Computers - комп'ютери зі складною системою команд) процесори. Така думка не зовсім вірна. Є багато процесорів, які називають RISC, хоча насправді вони відносяться до CISC. Більше того, в деяких додатках CISC-процесори виконують програмний код швидше, ніж це роблять RISC-процесори, або вирішують такі завдання, які RISC-процесори не можуть виконати.

Якою ж є насправді різниця між RISC і CISC? CISC-процесори виконують великий набір команд з розвиненими можливостями адресації (безпосередня, індексна і т.д.), даючи розробникові можливість вибрати найбільш підходящу команду для виконання необхідної операції. У RISC-процесорах набір виконуваних команд скорочений до мінімуму. При цьому розробник повинен комбінувати команди, щоб реалізувати більш складні операції.

Можливість рівноправного використання всіх регістрів процесора називається «ортогональністю» або «симетричністю» процесора. Це забезпечує додаткову гнучкість при виконанні деяких операцій. Розглянемо, наприклад, виконання умовних переходів в програмі. У CISC-процесорах умовний перехід звичайно реалізується відповідно до визначеного значення біта (флага) у регістрі стану. У RISC-процесорах умовний перехід може відбуватися при певному значенні біта, який знаходиться в будь-якому місці пам'яті. Це значно спрощує операції з флагами і виконання програм, які використовують ці флаги.

Успіх при використанні RISC-процесорів забезпечується завдяки тому, що їх більш прості команди вимагають для виконання значно меншу кількість машинних циклів. Таким чином досягається істотне підвищення продуктивності, що дозволяє RISC-процесорам ефективно вирішувати надзвичайно складні завдання.

### *ГАРВАРД проти ПРІНСТОНА*

Багато років тому уряд Сполучених Штатів дав завдання Гарвардському і Прінстонському університетам розробити архітектуру комп'ютера для військово-морської артилерії. Прінстонський університет розробив комп'ютер, який мав спільну пам'ять для зберігання програм і даних. Така архітектура комп'ютерів більше відома як архітектура фон Неймана, названа іменем наукового керівника цієї розробки (рис. 3).

У цій архітектурі блок інтерфейсу з пам'яттю виконує низку запитів до пам'яті, забезпечуючи вибірку команд, читання і запис даних, що розміщуються в пам'яті або у внутрішніх регістрах. Може здатися, що блок інтерфейсу є найбільш вузьким місцем між процесором і пам'яттю, так як одночасно з даними потрібно вибирати з пам'яті чергову команду. Однак у багатьох процесорах з Прінстонською архітектурою ця проблема вирішується шляхом вибірки наступної команди під час виконання попередньої. Така операція називається попередньою вибіркою («передвибірка»), і вона реалізується в більшості процесорів з такою архітектурою.

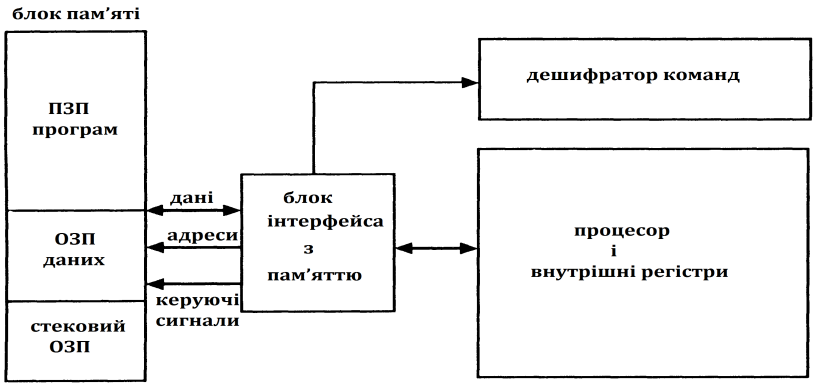



*Рис. 3. Структура комп'ютера з Прінстонською архітектурою.*

Гарвардський університет представив розробку комп'ютера, в якому для зберігання програм, даних і стека використовувалися окремі об'єми пам'яті (рис. 4).

Прінстонська архітектура виграла змагання, так як вона більше відповідала рівню технологій того часу. Використання загальної пам'яті виявилося кращим через ненадійність лампової електроніки (це було до широкого розповсюдження транзисторів) - при цьому виникало менше відмов.

Гарвардська архітектура майже не використовувалася до кінця 70-х років, допоки виробники мікроконтролерів зрозуміли, що ця архітектура надає перевагу пристроям, які вони розробляли.

Основною перевагою архітектури фон Неймана є те, що вона спрощує пристрій мікропроцесора, так як реалізує звернення тільки до однієї спільної пам'яті. Для мікропроцесорів найважливішим є те, що вміст оперативної пам'яті (RAM - Random Access Memory) може бути використано як для зберігання даних, так і для зберігання програм. У деяких додатках програмі необхідно мати доступ до вмісту стека. Все це надає велику гнучкість для розробника програмного забезпечення, перш за все в області операційних систем реального часу.

Гарвардська архітектура виконує команди за меншу кількість тактів, ніж архітектура фон Неймана. Це обумовлено тим, що у Гарвардській архітектурі більше можливостей для реалізації паралельних операцій. Вибірка наступної команди може відбуватися одночасно з виконанням попередньої команди, і немає необхідності зупиняти процесор на час вибірки команди.

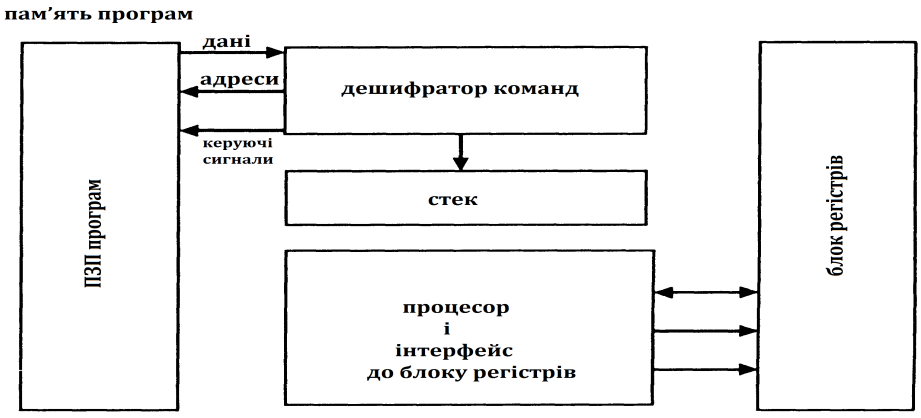



*Рис. 4. Структура комп'ютера з Гарвардською архітектурою*

Наприклад, якщо процесору з Прінстонською архітектурою необхідно рахувати байт і помістити його в акумулятор, то він виробляє послідовність дій показану на рис. 5. У першому циклі з пам'яті вибирається команда, в наступному циклі дані, які повинні бути поміщені в акумулятор, зчитуються з пам'яті.

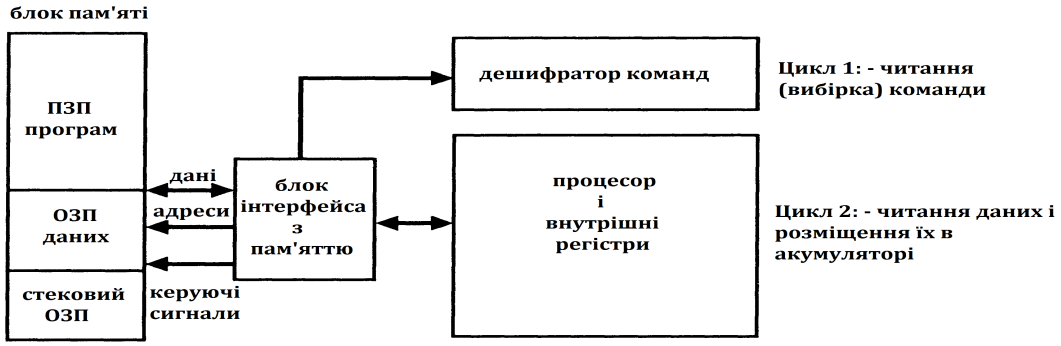



*Рис. 5. Виконання команди mov Ace, Reg в Прінстонській архітектурі*

У Гарвардській архітектурі, що забезпечує більш високий ступінь паралелізму операцій, виконання поточної операції може поєднуватися з вибіркою наступної команди (рис. 6). Команда також виконується за два цикли, але вибірка чергової команди проводиться одночасно з виконанням попередньої. Таким чином команда виконується всього за один цикл (під час читання наступної команди).

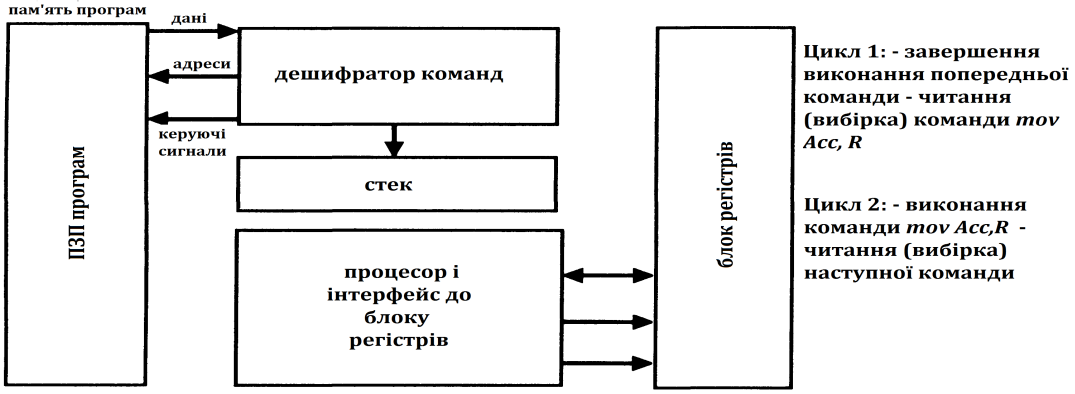



*Рис. 6. Виконання команди mov Ace,Reg в Гарвардській архітектурі.*

Цей метод реалізації операцій («паралелізм») дозволяє командам виконуватися за однакове число тактів, що дає можливість більш просто визначити час виконання циклів і критичних ділянок програми. Ця обставина є особливо важливою при виборі мікроконтролера для додатків, де потрібне строге забезпечення заданого часу виконання.

Наприклад, мікроконтролер AVR фірми Atmel виконує більшість команд за 1 такт (він же один машинний цикл), мікроконтролер PIC фірми Microchip виконує будь-яку команду, крім тих, які модифікують вміст програмного лічильника, за чотири такти (один цикл). Це спрощує реалізацію критичних по часу процедур в порівнянні з мікроконтролером Intel 8051, де для виконання команд

може знадобитися від 16 до 64 тактів. Через це часто не вдається підрахувати точний час виконання програми вручну і доводиться застосовувати симулятори або апаратні емулятори.

Слід зазначити, що такі загальні способи порівняння продуктивності не слід використовувати для всіх процесорів і мікроконтролерів, в яких реалізуються ці дві архітектури. Порівняння краще проводити відносно конкретних додатків. Різні архітектури й пристрої мають свої специфічні особливості, які дозволяють найкращим чином реалізувати ті чи інші програми. У деяких випадках конкретний додаток може бути виконаний тільки з використанням певної архітектури і специфічних особливостей мікроконтролера.

Після читання цього розділу багато хто, ймовірно, подумає, що Гарвардська архітектура - це єдино правильний вибір. Але Гарвардська архітектура є недостатньо гнучкою для деяких програмних процедур, які потрібні для реалізації ряду програм.

Не слід вважати, що одна архітектура краща, ніж інша. Кожна архітектура мікроконтролерів має свої переваги і недоліки в різних ситуаціях.

## Типи пам’яті мікроконтролерів

Можна виділити три основні види пам'яті, яка використовується в мікроконтролерах. Пам'ять програм являє собою постійну пам'ять, призначену для зберігання програмного коду і констант. Ця пам'ять не змінює свій вміст в процесі виконання програми. Пам'ять даних призначена для зберігання змінних в ході виконання програми. Регістри мікроконтролера - цей вид пам'яті включає внутрішні регістри процесора і регістри, які служать для керування периферійними пристроями.

Можливо, малий об'єм пам'яті мікроконтролерів здивує. Далі ви побачите, що це не є їх істотним недоліком. Але при першому знайомстві дана особливість дійсно викликає здивування, особливо, якщо порівнювати мікроконтролери з сучасними персональними комп'ютерами, які містять гігабайти пам'яті.

### *Пам’ять програм*

Для зберігання програм звичайно служить один з видів постійної пам'яті: PROM (однократно-програмований ПЗП), EPROM (електрично програмований ПЗП з ультрафіолетовим стиранням), EEPROM (ПЗП з електричним записом і стиранням, до цього виду відносяться також сучасні мікросхеми Flash-пам'яті) або ROM (масочно-програмований ПЗП). Всі ці види пам'яті є енергонезалежними - це означає, що вміст пам'яті зберігається після вимкнення живлення мікроконтролера. Така пам'ять необхідна, оскільки мікроконтролер не містить будь-яких пристроїв загальної пам'яті (магнітних дисків), з яких завантажуються програми на комп'ютерах. Програма постійно зберігається в мікроконтролері.

У процесі виконання програма зчитується з цієї пам'яті, а блок управління (дешифратор команд) забезпечує її декодування і виконання необхідних операцій. Вміст пам'яті програм не може змінюватися (перепрограмовуватися) під час виконання програми. Тому функціональне призначення мікроконтролера не може змінитися, поки вміст пам'яті його програми не буде стерто (якщо це можливо) і перепрограмовано (заповнено новими командами).

Слід звернути увагу, що розрядність мікроконтролера (8, 16 або 32 біт) вказується відповідно до розрядності його шини даних. У Гарвардській архітектурі команди можуть мати більшу розрядність, ніж дані, щоб дати можливість зчитувати за один такт цілу команду. Наприклад, мікроконтролери PIC залежно від моделі використовують команди з розрядністю 12, 14 або 16 біт.

У мікроконтролерах AVR команда завжди має розрядність 16 біт. Проте всі ці мікроконтролери мають шину даних розрядністю 8 біт.

У пристроях з Прінстонською архітектурою розрядність даних зазвичай визначає розрядність (кількість ліній) використовуваної шини. У мікроконтролерах Motorola 68HC05 24-розрядна команда розміщується в трьох 8-розрядних осередках пам'яті програм. Для повної вибірки такої команди необхідно зробити три цикли зчитування цієї пам'яті.

Коли говориться, що пристрій є 8-розрядним, це означає розрядність даних, які здатний обробляти мікроконтролер.

Так звана масочна пам'ять ROM (ПЗП) використовується тоді, коли програмний код заноситься в мікроконтролер на етапі його виробництва. Попередньо програма відлагоджується та тестується, після чого передається фірмі-виробнику, де програма перетвориться в малюнок маски на скляному фотошаблонів. Отриманий фотошаблон з маскою використовується в процесі створення з'єднань між елементами, з яких складається пам'ять програм.

Така ROM є найдешевшим типом постійної пам'яті для масового виробництва. Проте вона має ряд істотних недоліків, які призвели до того, що в останні роки цей тип пам'яті майже не використовується. Основними недоліками є значні витрати коштів і часу на створення нового комплекту фотошаблонів та їх впровадження у виробництво. Зазвичай такий процес займає близько десяти тижнів і є економічно вигідним при випуску десятків тисяч приладів. Тільки за таких обсягів виробництва забезпечується перевага ROM в порівнянні з E(E)PROM. Існує також обмеження, пов'язане з можливістю використання таких мікроконтролерів тільки в певній сфері застосування, так як його програма забезпечує виконання жорстко фіксованої послідовності операцій, і не може бути використана для вирішення будь-яких інших завдань.

Електрично програмована пам'ять EPROM складається з комірок, які програмуються електричними сигналами і стираються за допомогою ультрафіолетового світла. Пам'ять PROM може бути запрограмована тільки один раз. Ця пам'ять зазвичай містить плавкі перемички, які перепалюють під час програмування. В даний час така пам'ять використовується дуже рідко.

Комірка пам'яті EPROM представляє собою MOS-транзистор з плаваючим затвором, який оточений діоксидом кремнію (SiO2). Сток транзистора з'єднаний з «землею», а витік підключений до напруги живлення через резистор. У стертому стані (до запису) плаваючий затвор не містить заряду, і MOS-транзистор закритий. У цьому випадку на витоку підтримується високий потенціал, і при зверненні до осередку зчитується логічна одиниця. Програмування пам'яті зводиться до запису у відповідні поля логічних нулів.

Програмування здійснюється шляхом подачі на керуючий затвор високої напруги (рис.7). Цієї напруги має бути достатньо, щоб забезпечити пробій між керуючим і плаваючим затвором, після чого заряд з керуючого затвора переноситься на плаваючий. MOS-транзистор перемикається у відкритий стан, закорочуючи витік із землею. У цьому випадку при зверненні до осередку зчитується логічний нуль.

Щоб стерти вміст комірки, вона опромінюється ультрафіолетовим світлом, що дає заряду на плаваючому затворі достатню енергію, щоб він міг покинути затвор. Цей процес може займати від декількох секунд до декількох хвилин.

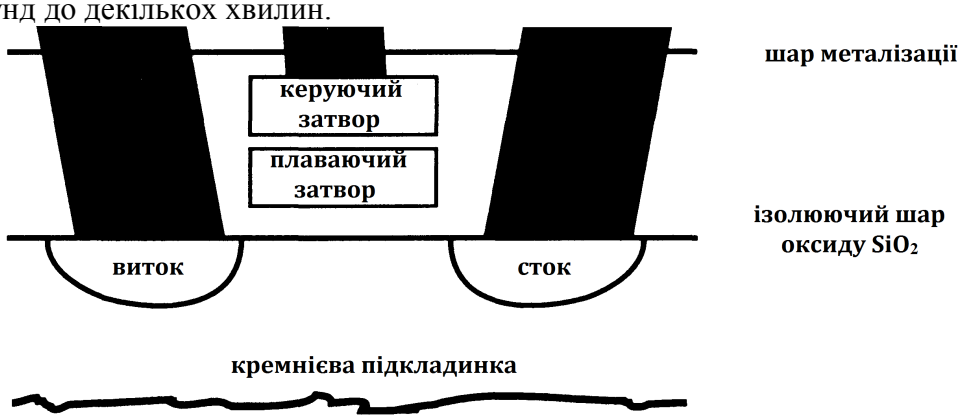



*Рис. 7. Комірка пам'яті EPROM*

Зазвичай, мікросхеми EPROM виробляються в керамічному корпусі з кварцовим віконцем для доступу ультрафіолетового світла. Такий корпус досить дорогий, що значно збільшує вартість мікросхеми. Для зменшення ціни мікросхеми EPROM укладають в корпус без віконця (версія EPROM з однократним програмуванням). Скорочення вартості при використанні таких корпусів може бути настільки значним, що ці версії EPROM в даний час часто використовуються замість масочних ROM.

Раніше мікроконтролери програмувались тільки за допомогою паралельних протоколів, досить складних для реалізації. В даний час протоколи програмування сучасної EPROM і EEPROM пам'яті істотно змінилося, що дозволило виконувати програмування мікроконтролера безпосередньо у складі системи, де він працює. Такий спосіб програмування одержала назву «in-system programming» або «ISP». ISP-мікроконтролери можуть бути запрограмовані після того, як їх припаяли на плату. При цьому скорочуються витрати на програмування, тому що немає необхідності у використанні спеціального обладнання - програматорів.

Пам'ять EEPROM (Electrically Erasable Programmable Memory - програмована пам'ять, що стирається електрично) можна вважати новим поколінням EPROM пам'яті. У такій пам'яті комірка стирається не ультрафіолетовим світлом, а шляхом електричного з'єднання плаваючого затвора з «землею». Використання EEPROM дозволяє стирати і програмувати мікроконтролер, не знімаючи його з плати. Таким способом можна періодично оновлювати його програмне забезпечення.

Пам'ять EEPROM дорожча, ніж EPROM (у два рази дорожче, ніж EPROM з однократним програмуванням). EEPROM працює трохи повільніше, ніж EPROM.

Основна перевага використання пам'яті EEPROM полягає в можливості її багаторазового перепрограмування без видалення з плати. Це дає величезний виграш на початкових етапах розробки систем на базі мікроконтролерів або в процесі їх вивчення, коли маса часу йде на багаторазовий пошук причин непрацездатності системи та виконання наступних циклів стирання-програмування пам'яті програм.

Функціонально Flash-пам'ять мало відрізняється від EEPROM. Основна відмінність полягає в способі стирання записаної інформації. У пам'яті EEPROM стирання проводиться окремо для кожного осередку, а в Flash-пам'яті стирання здійснюється цілими блоками. Якщо необхідно змінити вміст однієї клітинки Flash-пам'яті, то буде потрібно перепрограмувати цілий блок (або всю мікросхему). У мікроконтролерах з пам'яттю EEPROM можна змінювати окремі ділянки програми без необхідності перепрограмувати весь пристрій.

Часто вказується, що мікроконтролер має Flash-пам'ять, хоча насправді він містить EEPROM. В даний час між цими типами пам'яті є мало відмінностей, тому деякі виробники використовують такі терміни як еквівалентні.

### *Пам’ять даних*

При першому знайомстві з описом мікроконтролера багатьох здивує малий обсяг їх оперативної пам'яті даних RAM, який зазвичай складає десятки або сотні байт. Якщо мікроконтролер використовує для зберігання даних пам'ять EEPROM, то її обсяг також не перевищує кількох десятків байт.

Якщо Ви пишете програми для персонального комп'ютера (PC), то у Вас, напевно, виникне питання, що ж можна зробити з таким маленьким обсягом пам'яті. Ймовірно, Ваші програми для PC містять змінні, обсяг яких вимірюється в кілобайтах, не рахуючи використовуваних масивів даних. При використанні масивів необхідний обсяг пам'яті може становити сотні кілобайт. То що ж можна зробити, маючи об'єм ОЗП близько 250 байтів?

Справа в тому, що програмування мікроконтролера виконується трохи за іншими правилами, ніж програмування PC. Застосовуючи деякі нескладні правила, можна вирішувати багато задач з використанням невеликого обсягу пам'яті RAM. При програмуванні мікроконтролерів константи, якщо можливо, не зберігаються як змінні. Максимально використовуються апаратні можливості мікроконтролерів (такі як таймери, індексні регістри), щоб по можливості обмежити розміщення даних в RAM. Це означає, що при розробці прикладних програм необхідно заздалегідь подбати про розподіл ресурсів пам'яті. Прикладні програми повинні орієнтуватися на роботу без використання великих масивів даних.

### *Стек*

У мікроконтролерах RAM використовується для організації виклику підпрограм і обробки переривань. Під час цих операцій вміст програмного лічильника і основних регістрів (акумулятор,

регістр стану, індексні регістри і т.д.) зберігається і після цього відновлюється при поверненні до основної програми.

Стек - це електронна структура даних, яка функціонує аналогічно до своєї фізичної копії - стопки паперів. Коли що-небудь потрапляє у стек, то воно залишається там до тих пір, поки не буде викликане назад. Уявіть різнокольорові аркуші паперу, які укладаються в стопку один на інший. Коли листи видаляються, то відбувається їх переміщення у зворотному порядку. З цієї причини, стек часто називають чергою типу LIFO (Last In, First Out) - "останнім зайшов, першим вийшов".

У Прінстонській архітектурі RAM використовується для реалізації безлічі апаратних функції, включаючи функції стека. При цьому знижується продуктивність пристрою, так як для доступу до різних видів пам'яті потрібні багаторазові звернення, які не можуть виконуватися одночасно. З цієї ж причини Прінстонська архітектура зазвичай вимагає більшої кількості тактів на виконання команди, ніж Гарвардська.

Процесори Гарвардської архітектури можуть мати три області пам'яті, які адресують паралельно (в один і той же час) пам'ять програм, пам'ять даних, що включає простір вводу-виводу і стек.

У Гарвардській архітектурі стекові операції можуть проводитися в пам'яті, спеціально виділеній для цієї мети. Це означає, що при виконанні команди виклику підпрограми «call» процесор з Гарвардською архітектурою виконує кілька дій одночасно. В Прінстонській архітектурі при виконанні команди «call» наступна команда вибирається після того, як в стек буде поміщено вміст програмного лічильника.

Необхідно пам'ятати, що мікроконтролери обох архітектур мають обмежену місткість пам'яті для зберігання даних. Перевищення цієї межі може викликати проблеми при виконанні програми.

Якщо в процесорі виділений окремий стек і обсяг записаних у нього даних перевищує його місткість, то відбувається циклічна зміну вмісту покажчика стека і покажчик стека починає посилатися на раніше заповнену комірку стека. Це означає, що після занадто великої кількості команд «call» в стеку буде неправильна адреса повернення, яка була записана замість правильної адреси. Якщо мікропроцесор використовує загальну область пам'яті для розміщення даних і стека, то існує небезпека, що при переповненні стека відбудеться запис в область даних, або буде зроблена спроба запису завантажених у стек даних в область ROM.

Тепер розглянемо можливості збереження в стеці вмісту регістрів. У деяких архітектурах немає команд, що виконують завантаження вмісту регістрів в стек «push» і діставання зі стеку «pop». Однак команди «push» і «pop» можуть бути легко реалізовані за допомогою індексного регістра, який явно вказує на область стека. При цьому замість кожної з команд «push» і «pop» використовуються дві команди. Звичайно, таке рішення є менш ефективним, ніж використання спеціальних команд «push» і «pop», а використовуваний індексний регістр може знадобитися для інших цілей, проте це рішення забезпечує імітацію стека при використанні процесорів, у яких такі команди відсутні.

### *Регістри мікроконтролера і простір вводу-виводу*

Подібно до всіх комп'ютерних систем, мікроконтролери мають безліч регістрів, які використовуються для управлінням різними пристроями, підключеними до процесора. Це можуть бути регістри процесора (акумулятор, регістри стану, індексні регістри), регістри управління (регістри управління переривань, регістри управління таймером) або регістри, що забезпечують введення-виведення даних (регістри даних і регістри управління паралельним, послідовним або аналоговим вводом - виводом). Звернення до цих регістрів може вироблятися різними способами.

Способи звернення до регістрів, що реалізовані в мікроконтролері мають істотний вплив на їх продуктивність. Тому дуже важливо зрозуміти, як відбувається звернення до регістрів, щоб писати ефективні прикладні програми для мікроконтролерів. У процесорах з RISC-архітектурою всі регістри (часто і акумулятор) розташовуються за явно заданим адресами. Це забезпечує більш високу гнучкість при роботі процесора.

Розглянемо приклад процедури розгалуження програми за умови, що певний біт в регістрі порту вводу-виводу встановлений в 1. Для CISC-процесора відповідний псевдокод буде виглядати наступним чином:

```
Accumulator = IOPort                    ; Загрузити вміст регістра
                                      ; IOPort в акумулятор
Accumulator = Accumulator&(1<<Bit)    ; Маскувати всі біти акуму-
лятора,
                                      ; крім Bit
if ZeroFlag != 0                        ; Якщо не нуль, то біт Bit=1
goto Address
```

Асемблерний запис для мікроконтролерів Atmel AVR:

```
sbic IOPort, Bit              ; Пропустити наступну команду, якщо біт
Bit=0
rjmp Address
```

Більш ефективно ця процедура реалізується в мікроконтролері Intel 8051:

```
jb IOPort.Bit, Address      ;Перейти, якщо біт Bit=1
```

Ця послідовність операцій буде компілюватися із наступного коду на Сі:

```
if (IOPort.Bit == 1)
   run_function();
```

Використовуючи процесор, який може безпосередньо звертатися до будь-якого регістру, можна отримати перевагу при розробці простих прикладних програм. Наприклад, в мікроконтролері AVR вміст регістрів загального призначення і регістра стану не змінюється при передачі керування залежно від значення біта в регістрі порту IOPort.

Одним з важливих питань є розміщення регістрів в адресному просторі. У деяких процесорах всі регістри і RAM розташовуються в одному адресному просторі. Це означає, що пам'ять суміщена з регістрами. Такий підхід називається «відображенням пристроїв введення-виведення на пам'ять».

В інших процесорах адресний простір для пристроїв вводу-виводу відділений від загального простору пам’яті. Основна перевага розміщення регістрів введення-виведення в окремому просторі адрес полягає в тому, що при цьому спрощується схема підключення пам'яті програм і даних до загальної шини. Пристрої вводу-виводу зазвичай займають маленький блок адрес, що робить незручним декодування їх адреси спільно з великими блоками основної пам'яті. Окремий простір вводу-виводу дає деяку перевагу процесорам з Гарвардською архітектурою, забезпечуючи можливість зчитувати команду під час звернення до регістру введення-виведення.

Після всього вищесказаного, Ви, імовірно, вважаєте, що Гарвардська архітектура з регістрами і змінними, розташованими в різних окремих адресних просторах, є найбільш ефективною. І Ви маєте рацію. Проте є ряд причин, за якими використання мікроконтролерів Прінстонської архітектури з відображенням пристроїв введення-виведення на пам'ять може виявитися для деяких застосувань кращим.

### *Зовнішня пам’ять*

Незважаючи на величезні переваги використання внутрішньої вбудованої пам'яті, в деяких випадках необхідне підключення до мікроконтролера додаткової зовнішньої пам'яті (як пам'яті програм, так і даних). Існує два основних способи підключення зовнішньої пам'яті. Перший спосіб - підключення зовнішньої пам'яті до мікроконтролера, як до мікропроцесора. Багато мікроконтролерів містять спеціальні апаратні засоби для такого підключення. Другий спосіб полягає в тому, щоб підключити пам'ять до пристроїв вводу-виводу і реалізувати звернення до пам'яті через ці пристрої програмними засобами. Такий спосіб дозволяє використовувати прості пристрої вводу-виводу без реалізації складних шинних інтерфейсів. Вибір найкращого з цих способів залежить від конкретного додатка.

## Високопродуктивні RISC мікроконтролери сімейства AVR

На відміну від MICROCHIP, компанія ATMEL Corp. - один зі світових лідерів у виробництві широкого спектру мікросхем енергонезалежної пам'яті, FLASH-мікроконтролерів і мікросхем програмованої логіки, взяла старт з розробки RISC-мікроконтролерів в середині 90-х років, використовуючи всі свої технічні рішення, накопичені до цього часу.

Концепція нових швидкісних мікроконтролерів була розроблена групою розробників дослідного центру ATMEL в Норвегії, ініціали яких потім сформували марку AVR. Перші мікроконтролери AVR AT90S1200 з'явилися в середині 1997 р. і швидко здобули прихильність споживачів.

AVR-архітектура, на основі якої побудовані мікроконтролери сімейства AT90S, об'єднує потужний гарвардський RISC-процесор з роздільним доступом до пам'яті програм і даних, 32 регістри загального призначення, кожен з яких може працювати як регістр-акумулятор, і розвинену систему команд фіксованої довжини16-біт. Більшість команд виконується за один машинний такт з одночасним виконанням поточної і вибіркою наступної команди, що забезпечує продуктивність до 1 MIPS на кожен МГц тактової частоти.

32 регістри загального призначення утворюють регістровий файл швидкого доступу, де кожен регістр безпосередньо пов'язаний з АЛП. За один такт з реєстрового файлу вибираються два операнда, виконується операція, і результат повертається в регістровий файл. АЛП підтримує арифметичні та логічні операції з регістрами, між регістром і константою або безпосередньо з регістром.

Регістровий файл також доступний як частина пам'яті даних. 6 з 32-х регістрів можуть використовуватися як три 16-розрядних регістра-покажчика для непрямої адресації. Старші мікроконтролери сімейства AVR мають у складі АЛП апаратний помножувач.

Базовий набір команд AVR містить 120 інструкцій. Інструкції бітових операцій включають інструкції установки, очищення і тестування бітів.

Всі мікроконтролери AVR мають вбудовану FLASH ROM з можливістю внутрішньосхемного програмування через послідовний 4-провідний інтерфейс.

Периферія МК AVR включає: таймери-лічильники, широтно-імпульсні модулятори, підтримку зовнішніх переривань, аналогові компаратори, 10-розрядний 8-канальний АЦП, паралельні порти (від 3 до 48 ліній вводу/ виводу), інтерфейси UART і SPI, сторожовий таймер і пристрій скидання по включенню живлення. Всі ці якості перетворюють AVR-мікроконтролери на потужний інструмент для побудови сучасних, високопродуктивних і економічних контролерів різного призначення.

У рамках єдиної базової архітектури AVR-мікроконтролери підрозділяються на три підродини:

- **Classic AVR** — основна лінія мікроконтролерів з продуктивністю окремих модифікацій до 16 MIPS, FLASH ROM програм 2–8 Кбайт, EEPROM даних 64–512 байт, SRAM 128–512 байт;
- **mega AVR** з продуктивністю 4–6 MIPS для складних додатків, що потребують великого об’єму пам’яті, FLASH ROM програм 64–128 Кбайт, EEPROM даних 64–512 байт, SRAM 2–4 Кбайт, SRAM 4 Кбайт, вбудований 10-розрядний 8-канальний АЦП, апаратний помножувач 8x8;
- **tiny AVR** — низьковартісні мікроконтролери у 8-вивідному виконанні мають вбудовану схему контролю напруги живлення, що дозволяє обійтися без зовнішніх супервізорних мікросхем.

AVR-мікроконтролери підтримують сплячий режим і режим мікроспоживання. У сплячому режимі зупиняється центральне процесорне ядро, в той час як регістри, таймери-лічильники, сторожовий таймер і система переривань продовжують функціонувати. У режимі мікроспоживання зберігається вміст всіх регістрів, зупиняється тактовий генератор, забороняються всі функції мікроконтролера, поки не надійде сигнал зовнішнього переривання або апаратного скидання. Залежно від моделі, AVR-мікроконтролери працюють у діапазоні напруг

2,7-6 В або 4-6 В (виняток становить ATtiny12V з напругою живлення 1,2 В). Засоби налагодження. ATMEL пропонує програмне середовище AVR-studio для налагодження програм в режимі симуляції на програмному відлагоджувачу, а також для роботи безпосередньо з внутрішньосхемним емулятором. AVR-studio доступний з WEB-сторінки ATMEL, містить асемблер і призначений для роботи з емуляторами ICEPRO і MegaICE. Ряд компаній пропонують свої версії Сі-компіляторів, асемблер, редакторів зв'язків і завантажувачів для роботи з мікроконтролерами сімейства AVR.

Архітектура AVR буде розглядатися на прикладі контролера AT90S4433.

### *Мікроконтролер AT90S2333/4433*

- AVR® - висока продуктивність і RISC архітектура з низьким енергоспоживанням:
  - 118 потужних інструкцій - більшість із них виконується за один такт
  - 32 x 8 робочих регістрів загального призначення
  - Продуктивність, аж до 8 MIPS при 8 МГц
- Дані та енергонезалежна програмна пам'ять:
  - 2 Кбайт/4 Кбайт Flash - пам'ять з підтримкою внутрішньосистемного програмування (ISP), ресурс запису/стирання 1000 циклів
  - 128 байт SRAM
  - 128/ 256 байт EEPROM з підтримкою внутрішньосистемного програмування, ресурс запису/ стирання 100 000 циклів
  - Програмоване блокування для безпеки вмісту Flash і EEPROM пам'яті
- Периферія:
  - Один 8-ми розрядний таймер/ лічильник з окремим попереднім подільником частоти
  - Розширений 16-ти розрядний таймер/ лічильник окремим попереднім подільником частоти з режимами порівняння, захоплення і 8-ми/ 9-ти, чи 10-ти розрядною ШИМ
  - Вбудований аналоговий компаратор
  - Програмований сторожовий таймер з вбудованим тактовим генератором
  - Програмований UART
  - 6- ти канальний, 10-ти розрядний АЦП
  - SPI послідовний інтерфейс ведучий/ ведений
- Спеціальні функції мікроконтролера:
  - Система ініціалізації при аварійному вимкненні живлення
  - Вдосконалений ланцюг ініціалізації при ввімкненні живлення
  - Режими пониженого енергоспоживання:
  - Зовнішні і внутрішні джерела переривання
- Характеристики:
  - Низьке енергоспоживання, тех. процес швидкодіючої CMOS
  - Повністю статичний режим роботи
- Струм споживання при 4 МГц, 3 В, 25°C
  - Активний режим: 3.4 мА
  - Режим спокою: 1.4 мА
  - Режим відключення: <1 мкА
- I/O і тип корпуса:
  - 20 програмованих шин I/O
  - 28-pin PDIP і 32-pin TQFP
- Напруга живлення:
  - від 2.7 В до 6.0 В для AT90LS2333/AT90LS4433
  - від 4.0 В до 6.0 В для AT90S2333/AT90S4433
- Швидкодія: від 0 до 4 МГц (AT90LS2333/AT90LS4433)
- Швидкодія: від 0 до 8 МГц (AT90S2333/AT90S4433)

## Призначення виводів

Призначення виводів в залежності від корпуса наведено на рис. 8

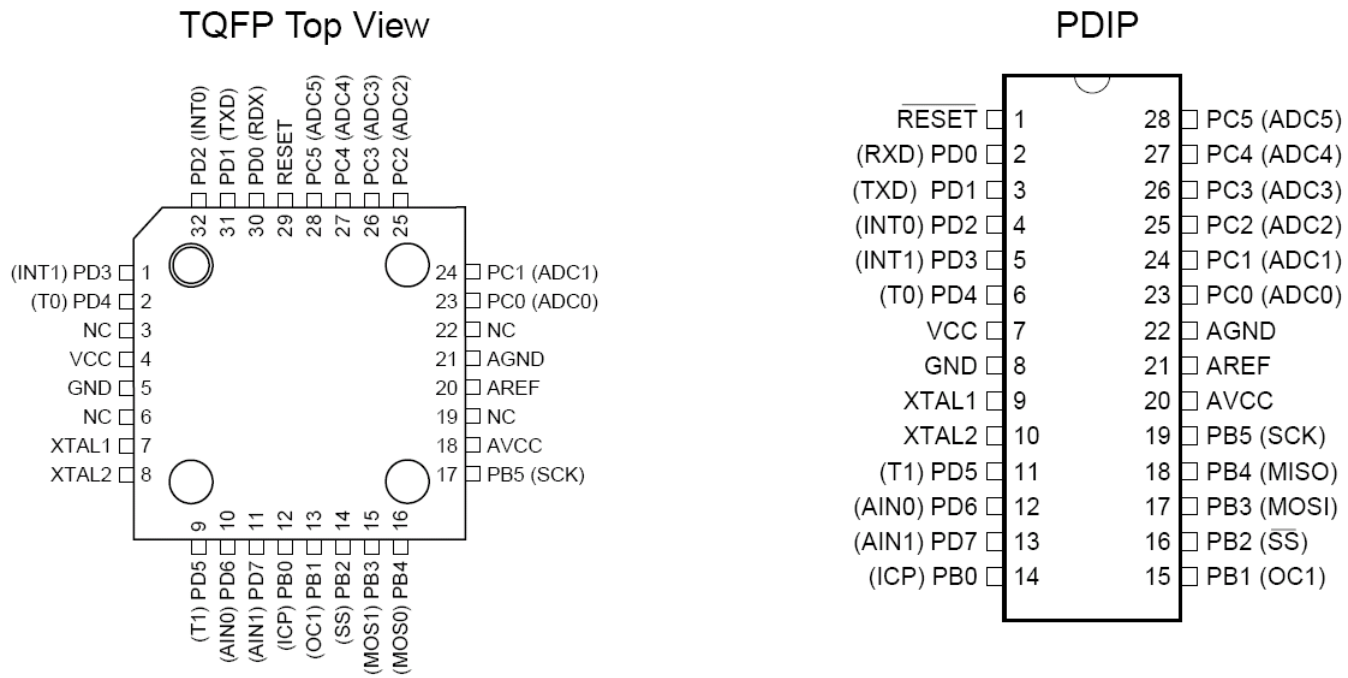



*Рис. 8 Призначення виводів*

## Структурна схема

Структурна схема контролера наведена на рис. 9.

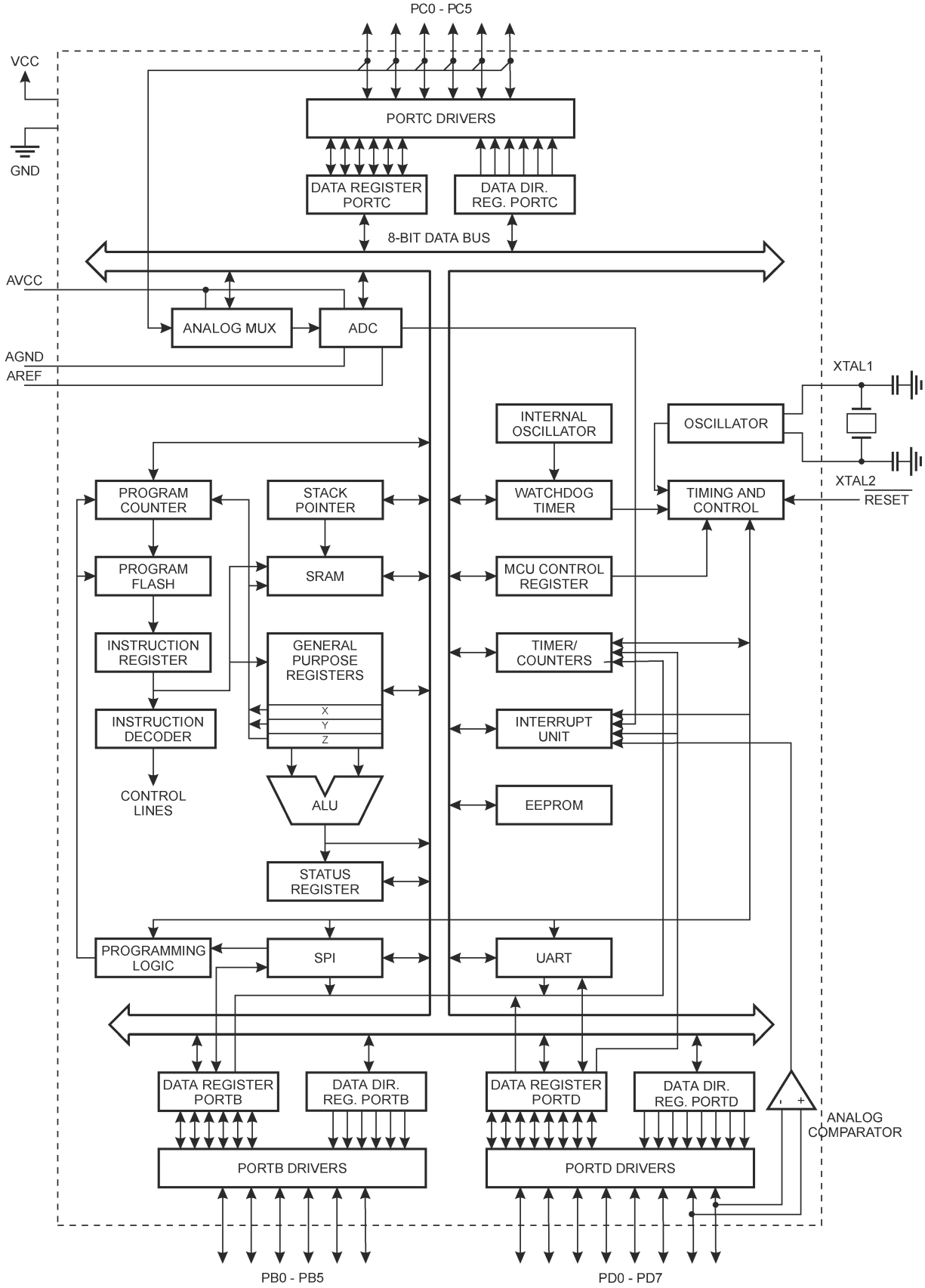



*Рис. 9 Структура мікроконтролера AVR 4433 фірми Atmel*

## Загальний опис

AT90S2333/4433 являються 8-ми розрядними CMOS мікроконтролерами з низьким рівнем енергоспоживання, які базуються на вдосконаленій AVR RISC архітектурі. Завдяки виконанню високопродуктивних інструкцій за один період тактового сигналу, AT90S2333/4433 досягають продуктивності, яка наближається до рівня 1 MIPS на МГц, забезпечуючи розробнику можливість оптимізувати рівень енергоспоживання у відповідності з необхідною обчислювальною продуктивністю.

Ядро AVR містить потужний набір інструкцій і 32 робочих регістри загального призначення. Всі 32 регістри напряму підключені до арифметико - логічного пристрою (АЛП), що забезпечує доступ до двох незалежних регістрів при виконанні однієї інструкції за один такт. В результаті, дана архітектура має більш високу ефективність коду, при підвищенні пропускної здатності, аж до 10 разів, в порівнянні зі стандартними мікроконтролерами CISC.

AT90S2333/4433 мають: 2 Кбайт/4 Кбайт Flash – пам’яті з підтримкою внутрішньосистемного програмування, 128/256 байт EEPROM, 20 ліній I/O загального призначення, 32 робочих регістрів загального призначення, два універсальних таймера/лічильника з режимами порівняння, внутрішні і зовнішні переривання, програмований послідовний UART, 6-ти канальний, 10-ти розрядний АЦП, програмований слідкуючий таймер з вбудованим тактовим генератором і програмований послідовний порт SPI для завантаження програм у Flash пам’ять, а також, два режими економії енергоспоживання, що вибираються програмно. Режим очікування «Idle» зупиняє CPU, але залишає функціонувати SRAM, таймер/лічильники, SPI порт і систему переривань. Режим економії енергоспоживання «Power Down» зберігає значення регістрів, але зупиняє тактовий генератор, відключаючи решту функцій мікроконтролера, аж до наступного зовнішнього переривання, чи до апаратної ініціалізації.

Пристрої виготовляються із застосуванням технології енергонезалежної пам’яті з високою щільністю розташування, розробленою в корпорації Atmel. Вбудована Flash - пам’ять з підтримкою внутрішньосистемного програмування забезпечує можливість перепрограмування програмного коду в складі системи, через SPI послідовного інтерфейсу, чи за допомогою стандартного програматора енергонезалежної пам’яті. Завдяки суміщенню вдосконаленого 8-ми розрядного RISC CPU з Flash- пам’яттю з підтримкою внутрішньосистемного програмування на одному кристалі були отримані високопродуктивні мікроконтролери AT90S2333/4433, які забезпечують гнучке і економічно високоефективне рішення для багатьох вбудовуваних систем управління.

AVR AT90S2333/4433 підтримуються повним набором програм і пакетів для розробки, включаючи: компілятори С, макроасемблери, відлагоджувачі / симулятори програм, внутрішньосхемні емулятори і набори для макетування.

*Таблиця 1*

| Модель | Flash пам’ять | EEPROM | SRAM | Напруга | Частота |
|---|---|---|---|---|---|
| AT90S2333 | 2Kbyte | 128byte | 128byte | 4.0-6.0В | 0-8MHz |
| AT90LS2333 | 2Kbyte | 128byte | 128byte | 2.7-6.0В | 0-4MHz |
| AT90S4433 | 4Kbyte | 256byte | 128byte | 4.0-6.0В | 0-8MHz |
| AT90LS4433 | 4Kbyte | 256byte | 128byte | 2.7-6.0В | 0-4MHz |

### *Архітектура AVR*

Файл регістрів швидкого доступу, містить 32 8-розрядних робочих регістри загального призначення пов’язаних безпосередньо з ALU. За один тактовий цикл із файлу регістрів вибираються два операнда, виконується операція і результат знову повертається в файл регістрів.

Шість із 32 регістрів можуть бути використані як три 16-розрядних регістра покажчика непрямої адресації адресного простору даних, які забезпечують ефективне обчислення адреси. Один із цих покажчиків адреси використовується, також, як покажчик адреси для функції

неперервного перегляду таблиць. Ці 16-розрядні додаткові регістри позначаються X-регістр, Y-регістр і Z-регістр.

ALU підтримує арифметичні і логічні операції між регістрами або між константою і регістром. Виконується в ALU і операції з окремими регістрами. На Рис. 10 показана AVR розширена RISC архітектура мікроконтролерів AT90S2333/4433.

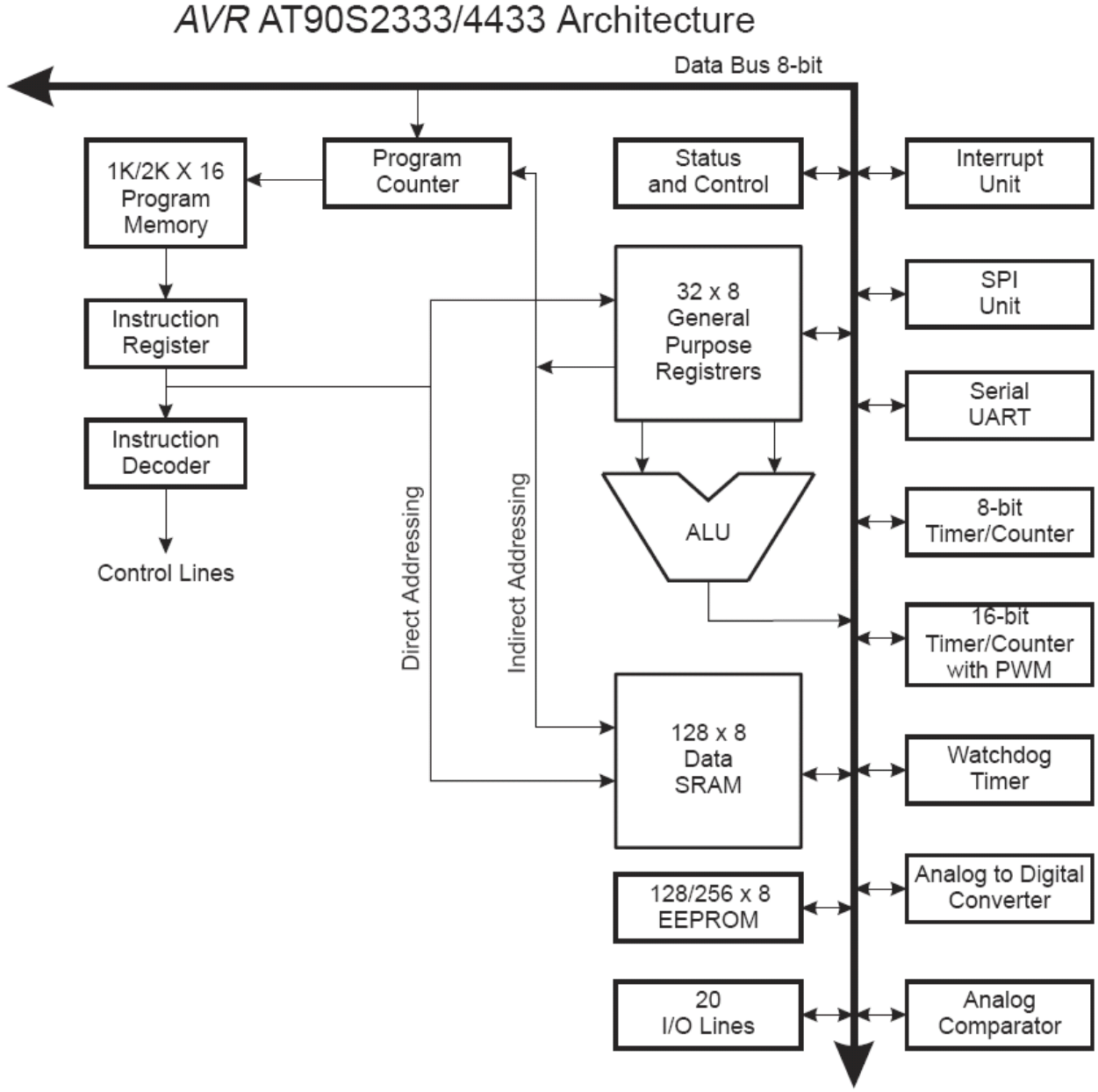



*Рис. 10. AVR RISC архітектура мікроконтролерів AT90S2333/4433*

В додаток до операцій з регістрами, регістровий файл може використовуватися і для звичайної адресації пам'яті. Це пояснюється тим, що файл регістрів розташовується по 32 наймолодшим адресам простору даних, і до них можна звертатися як до звичайних комірок пам'яті.

Простір пам'яті I/O містить 64 адреси периферійних функцій CPU таких як: регістри керування, таймери/лічильники, аналого-цифрові перетворювачі і інші I/O функції. До пам'яті I/O можна звертатися безпосередньо або як до комірок простору пам'яті відповідними адресами регістру файлів - F.

В мікроконтролерах AVR використані принципи Гарвардської архітектури - окремі пам'ять і шини для програм і даних. При роботі з пам'яттю програм використовується однорівневий конвеєр - в той час, як одна команда виконується, наступна команда вибирається із пам'яті програм, такий прийом дозволяє виконувати команду в кожному тактового циклі. Пам'яттю програм є внутрішньосистемно програмована Flash пам'ять. За малим винятком AVR команди

мають формат одного 16-розрядного слова, в зв'язку з чим кожна адреса пам'яті програм містить одну 16-разрядну команду.

В процесі обробки переривань і викликів підпрограм адреса повернення лічильника команд (PC) зберігається у стеку. Стек розміщується в SRAM даних і, відповідно розмір стеку обмежений лише загальним розміром SRAM і рівнем її використання. Всі програми користувача в підпрограмах повернення (раніше, ніж підпрограми або переривання будуть виконуватися) повинні ініціалізувати покажчик стеку (SP). 8-розрядний покажчик стеку, з можливістю читання/запису розташовується в просторі I/O.

AVR архітектура підтримує п'ять різних режимів адресації даних.

Гнучкий модуль обробки переривань має в просторі I/O свій керуючий регістр з додатковим бітом дозволу глобального переривання в регістрі статусу. Всі переривання мають свої вектори переривання в таблиці векторів переривання, розташованій на початку пам'яті програм. Пріоритети переривань відповідають положенню векторів переривань - переривання з найменшою адресою вектора має найвищий пріоритет.

Всі простори пам'яті AVR архітектури лінійні і регулярні.

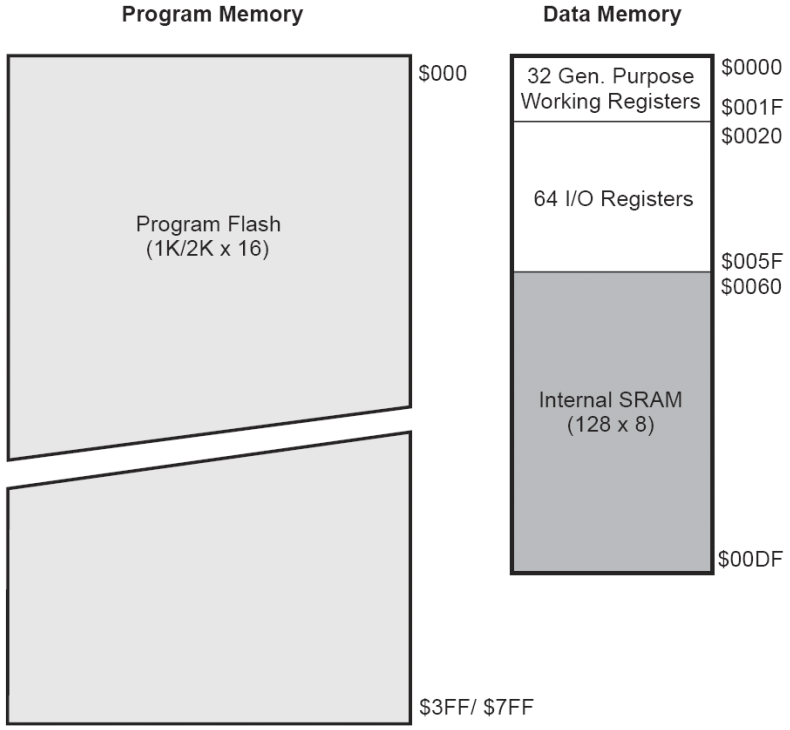



*Рис. 11 Організація пам'яті*

### *Файл регістрів загального призначення*

На Рис. 12 представлена структура 32 регістрів загального призначення.

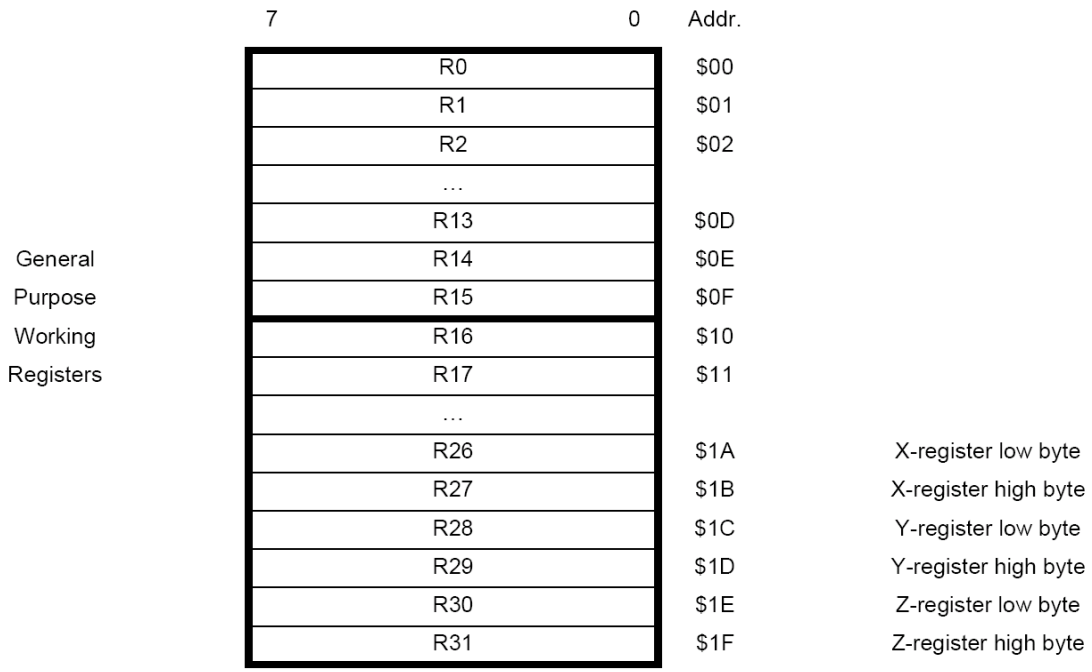



*Рис. 12. Регістри загального призначення мікроконтролерів AVR*

Всі регістрові команди звертаються безпосередньо до регістрів на протязі одного тактового циклу. Винятком є п'ять логічних і арифметичних операцій з константами (SBCI, SUBI, CPI і ANDI) і операція ORI між константою і вмістом регістра, і команда безпосередньої загрузки константи LDI. Ці команди використовують другу половину регістрів регістрового файлу - R16..R31.

Найбільш загальні команди SBC, SUB, CP, AND і OR і всі інші операції між двома регістрами чи з одним регістром використовують для запису результату регістровий файл.

Як показано на Рис. 12, кожному регістру відповідає адреса пам'яті даних, яка відображає їх в перших 32 комірках простору даних користувача. Хоча вони не використовуються як фізичні комірки SRAM, така організація пам'яті забезпечує гнучке звернення до регістрів, оскільки X,Y і Z регістри можуть бути використані для індексації будь-якого регістра в файлі.

### *Регістр X, регістр Y і регістр Z*

Шість регістрів (з R26 по R31) регістрового файлу, крім звичайних для інших регістрів функцій, виконують функцію 16-розрядних регістрів покажчиків адреси при непрямій адресації SRAM. Ці три регістри непрямої адресації визначаються як регістри X,Y і Z.

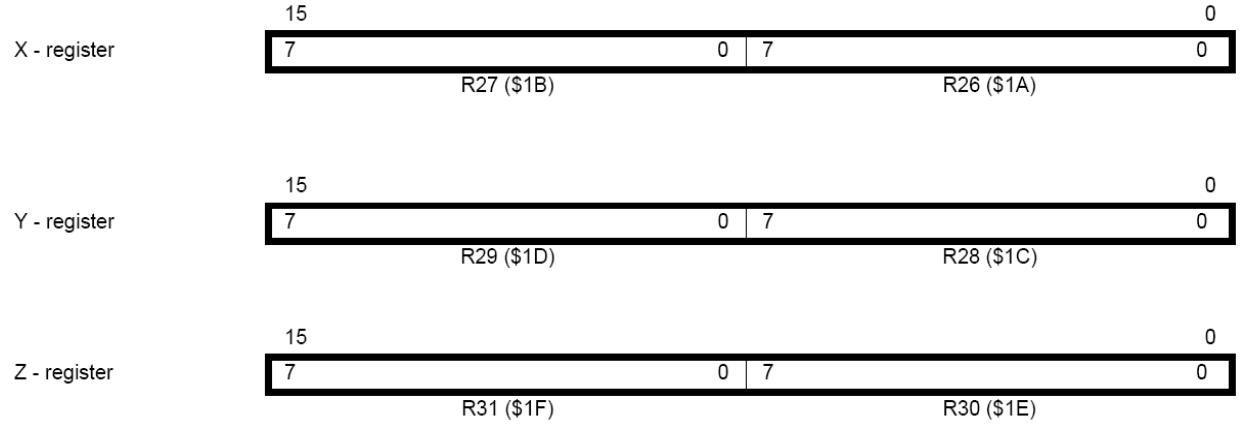



*Рис. 13. Регістри X, Y і Z*

В різних режимах адресації ці регістри виконують функції фіксованого зміщення, автоматичного інкременту і декременту (див. опис команд).

### *ALU - Арифметично-логічний пристрій*

Високопродуктивний AVR ALU з'єднаний безпосередньо зі всіма 32 швидкодіючими регістрами загального призначення. За один тактовий цикл ALU виконує операцію між регістрами цього регістрового файлу. Операції ALU поділяються на три основні категорії: арифметичні, логічні і операції над бітами.

### *Внутрішньосистемно-програмовна Flash пам'ять програм*

Коди програм мікроконтролерів ATmega603/103 записуються в 64/128 Кбайт вбудованої внутрішньосистемної програмованої Flash пам'яті. Оскільки всі команди мають формат одного або двох 16-розрядних слів, то і пам'ять програм має організацію 32/64Кх16. Flash пам'ять забезпечує не менше 1000 циклів стирання/запису.

Таблиці констант можуть бути розташовані в будь-якому місці всього простору пам'яті програм (див. опис команд LPM (Load Program Memory) - завантажити байт пам'яті програм і ELPM (Extended Load Program Memory) - завантажити байт пам'яті програм в розширеному режимі).

### *Конфігурація пам'яті*

На рис. 14 показано як організована SRAM в мікроконтролерах AT90S2333/4433.

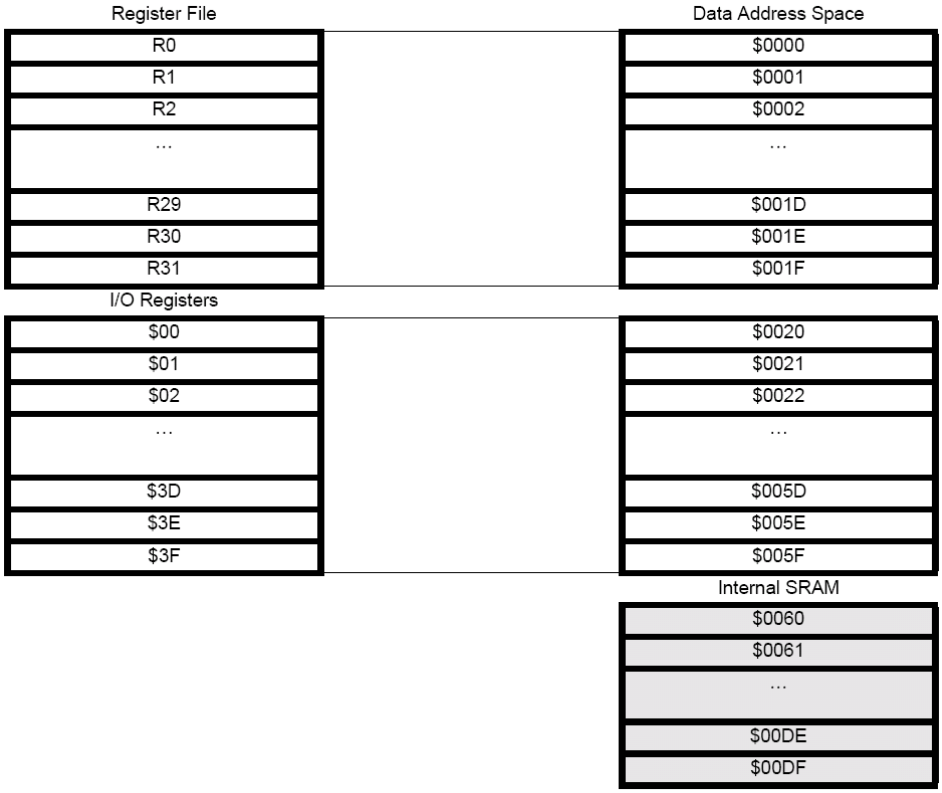



*Рис. 14. Конфігурація пам'яті*

По перших 224 адресах пам'яті даних розміщуються регістровий файл, простір пам'яті I/O і вбудована SRAM даних. Із них перші 96 адрес займають регістровий файл і простір пам'яті I/O, в наступних 128 адресах розміщується вбудована SRAM.

При адресації пам'яті даних використовуються п'ять режимів адресації: безпосередня адресація, непряма зі зміщенням, непряма, непряма з переддекрементом і непряма з постдекрементом. Регістри з R26 по R31 регістрового файлу працюють як X, Y і Z регістри покажчика непрямої адресації.

Пряма адресація покриває всю пам'ять. Непряма адресація зі зміщенням доступних 63 адрес відносно базових адрес, які знаходяться в регістрах Y або Z. При використанні непрямої адресації з автоматичним переддекрементом і постдекрементом автоматично декрементуються і інкрементуються адреси, записані в регістри X, Y і Z. Всіма цими режимами перекривається весь

адресний простір даних, включаючи 32 регістри загального призначення і 64 регістри I/O. Детальний опис всіх режимів адресації наведений в наступному розділі.

### *Режими адресації пам'яті програм і даних*

При зверненні до Flash пам'яті програм і пам'яті даних (SRAM, регістрового файлу і пам'яті I/O) AVR Enhanced RISC мікроконтролерами AT90S2333/4433 використовуються потужні і ефективні режими адресації. В даному розділі описуються режими адресації, які підтримуються AVR архітектурою. На рис. 15 OP означає частину слова команди, яка відповідає операційному коду.

#### Безпосередня адресація, одиночний регістр Rd

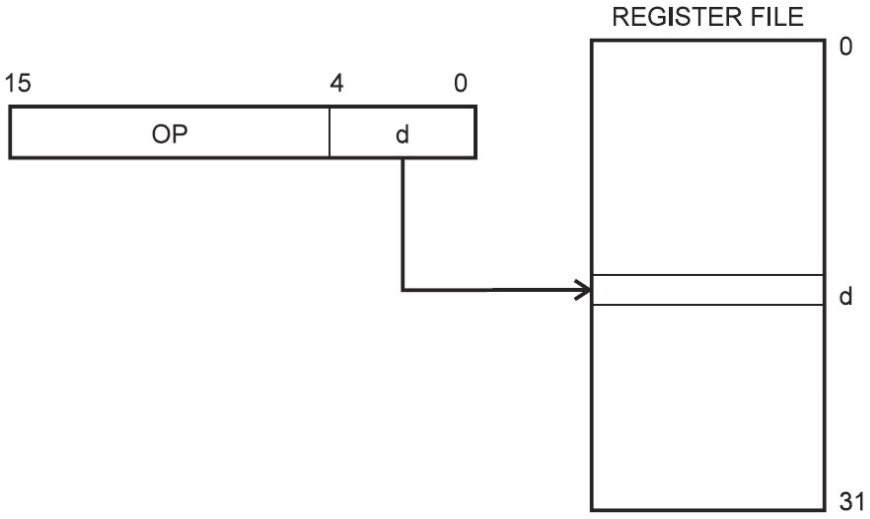



*Рис. 15. Безпосередня адресація одного регістра*

Операнд міститься в регістрі d (Rd).

#### Безпосередня адресація, два регістра Rd і Rr

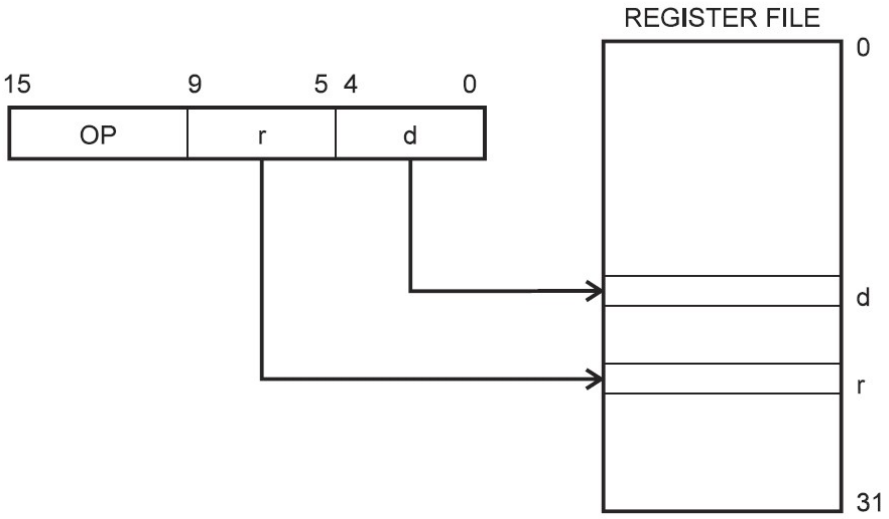



*Рис. 16. Безпосередня регістрова адресація двох регістрів*

Операнди містяться в регістрах r (Rr) і d (Rd).
Результат зберігається в регістрі d (Rd).

### Безпосередня адресація I/O

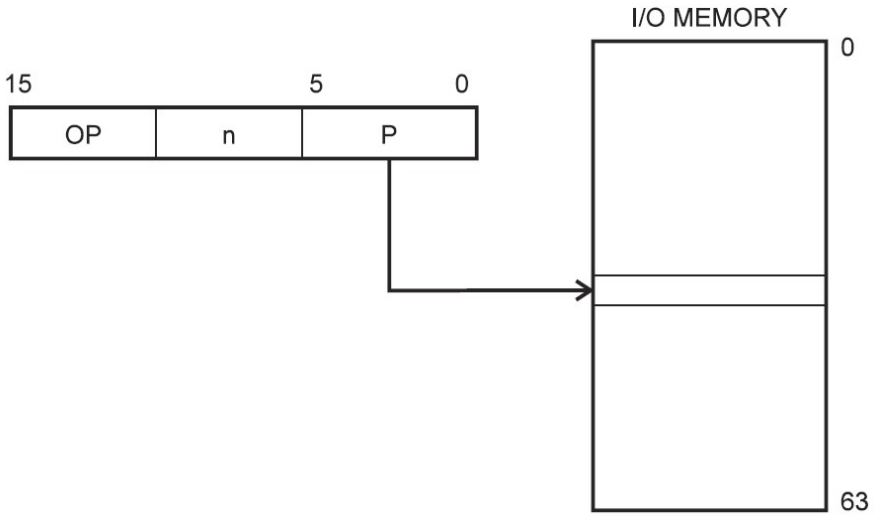



*Рис. 17. Безпосередня адресація I/O*

Адреса операнда міститься в 6 бітах слова команди. Величина n визначає адресу регістру джерела або регістру призначення.

### Безпосередня адресація даних

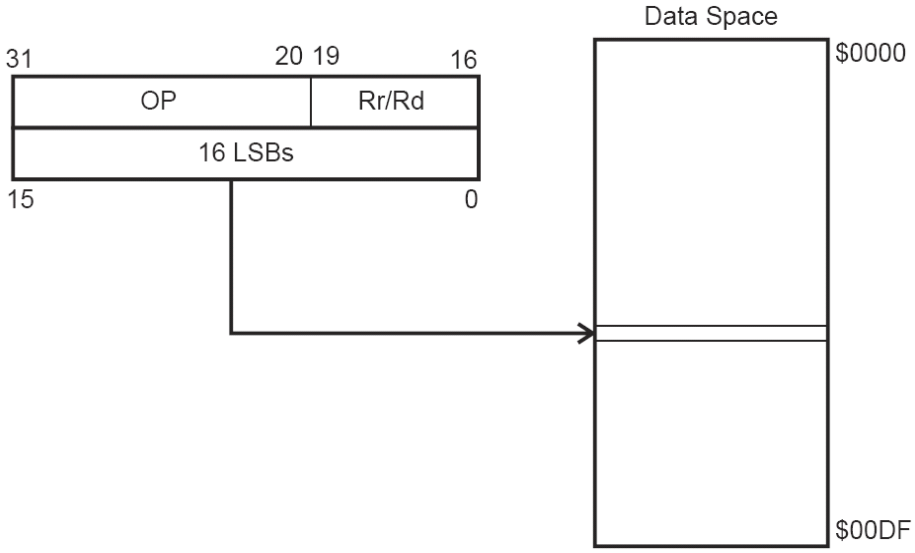



*Рис. 18. Безпосередня адресація даних*

16-розрядна адреса даних міститься в 16 молодших розрядах 32-розрядної команди. Rd/Rr визначають регістр джерело або регістр призначення.

### Непряма адресація даних зі зміщенням

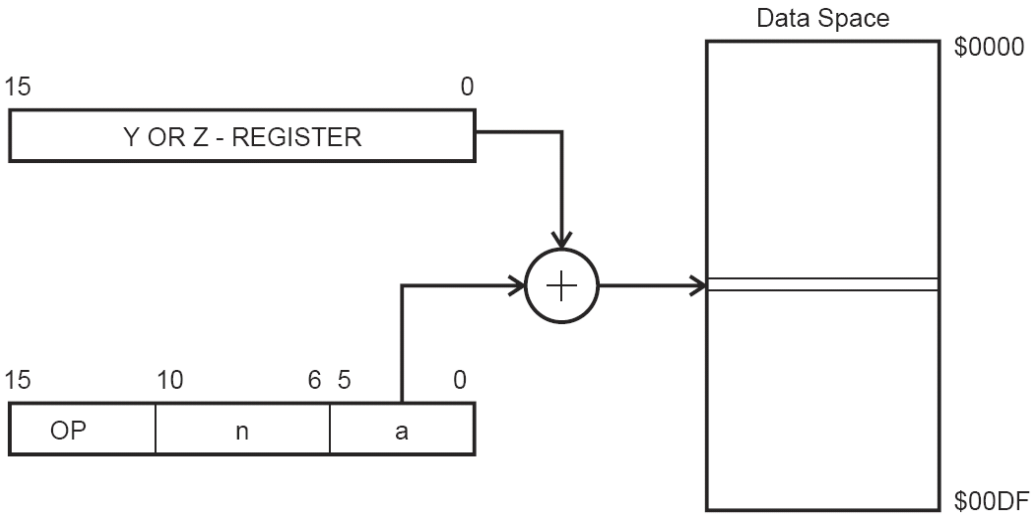



*Рис. 19. Непряма адресація даних зі зміщенням*

Адреса операнда вираховується підсумовуванням вмісту регістра Y або Z з 6 бітами адреси, які містяться в слові команди.

### Непряма адресація даних

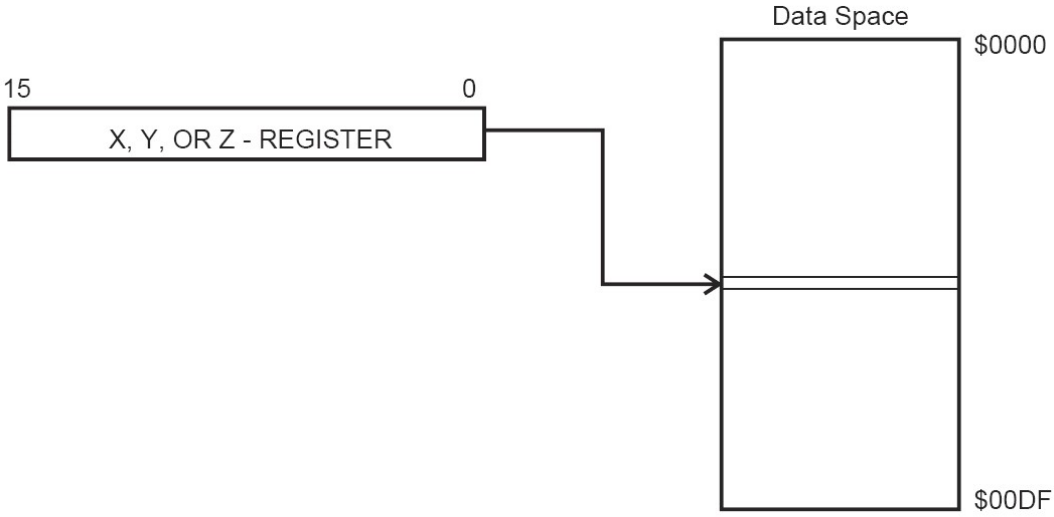



*Рис. 20. Непряма адресація даних*

Адреса операнда міститься в регістрі X, Y або Z.

### Непряма адресація даних з переддекрементом

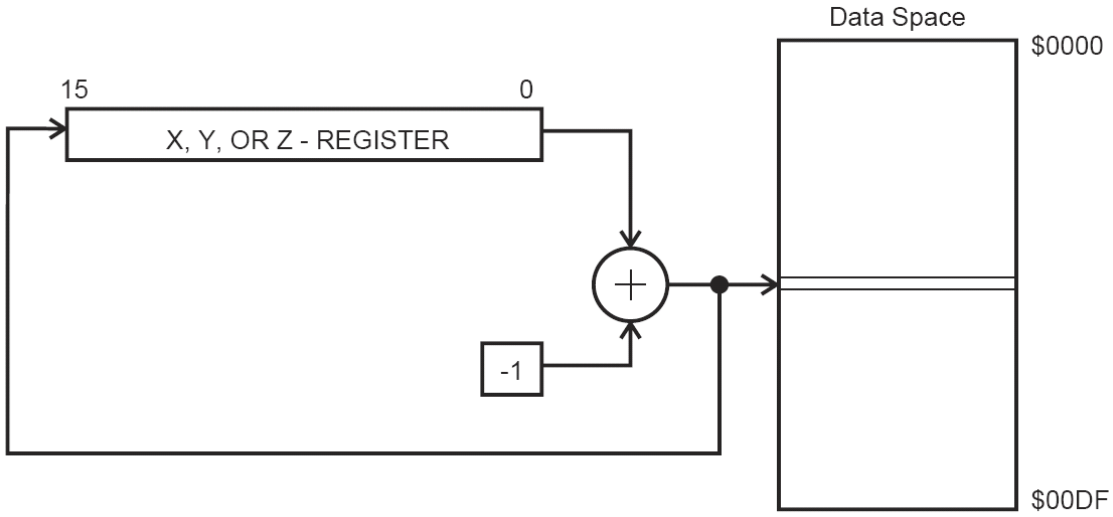



*Рис. 21. Непряма адресація даних з переддекрементом*

Перед виконанням операції регістр X, Y або Z декрементується. Декрементований вміст регістру X, Y або Z є адресою операнда.

### Непряма адресація даних з постінкрементом

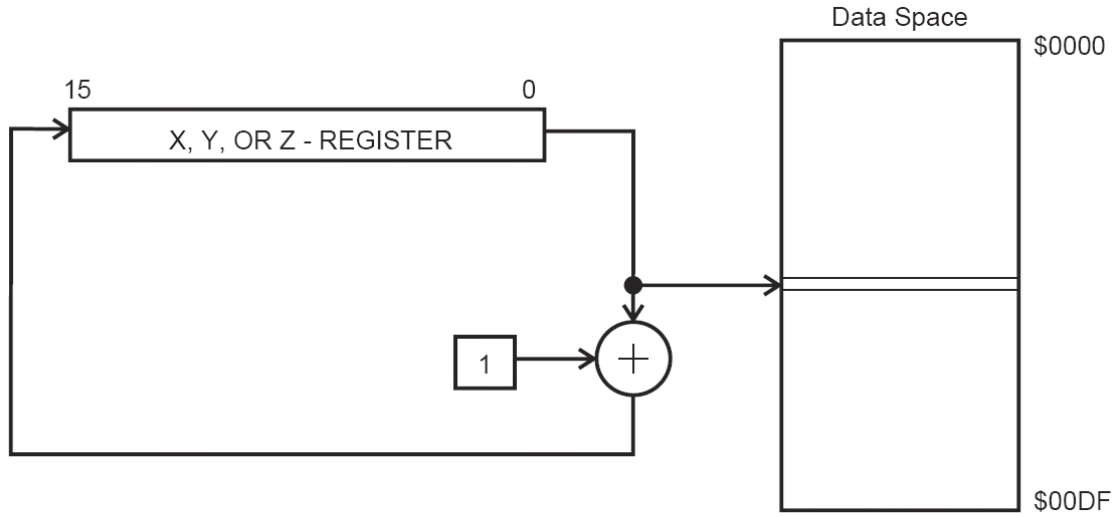



*Рис. 22. Непряма адресація даних з постінкрементом*

Після виконання операції регістр X, Y або Z інкрементується. Адресою операнда є вміст X, Y або Z регістра перед інкрементуванням.

### Адресація константи з використанням команд LPM

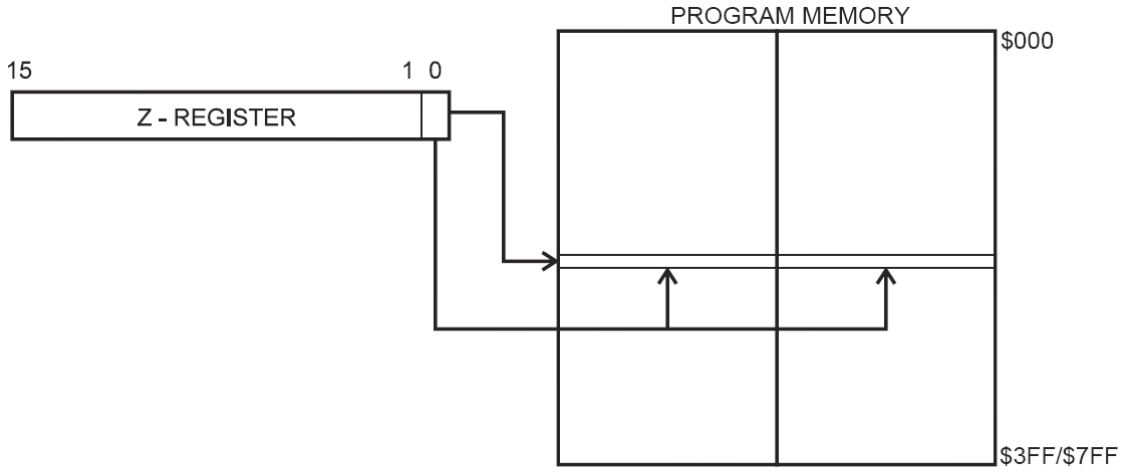



*Рис. 23. Адресація константи коду пам'яті*

Адреса байта константи визначається вмістом регістра Z. Старші 15 бітів визначають слово адреси (от 0 до 32К). Стан молодшого біта визначає вибір молодшого байта (LSB = 0) або старшого байта (LSB = 1).

### Непряма адресація пам'яті програм, команди IJMP і ICALL

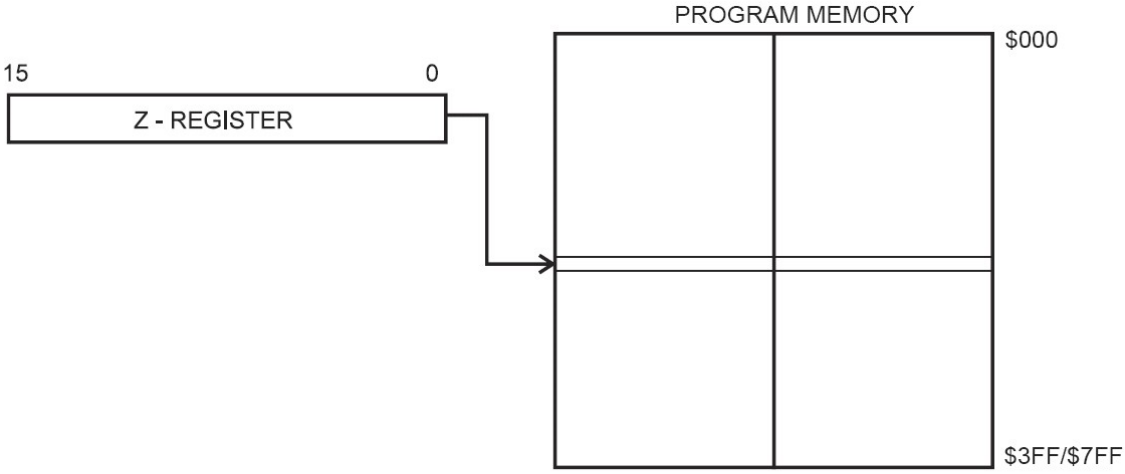



*Рис. 24. Непряма адресація пам'яті програм*

Виконання програми продовжується з адреси, яка міститься в регістрі Z (тобто лічильник команд загружається вмістом регістру Z).

### Непряма адресація пам'яті програм, команди RJMP і RCALL

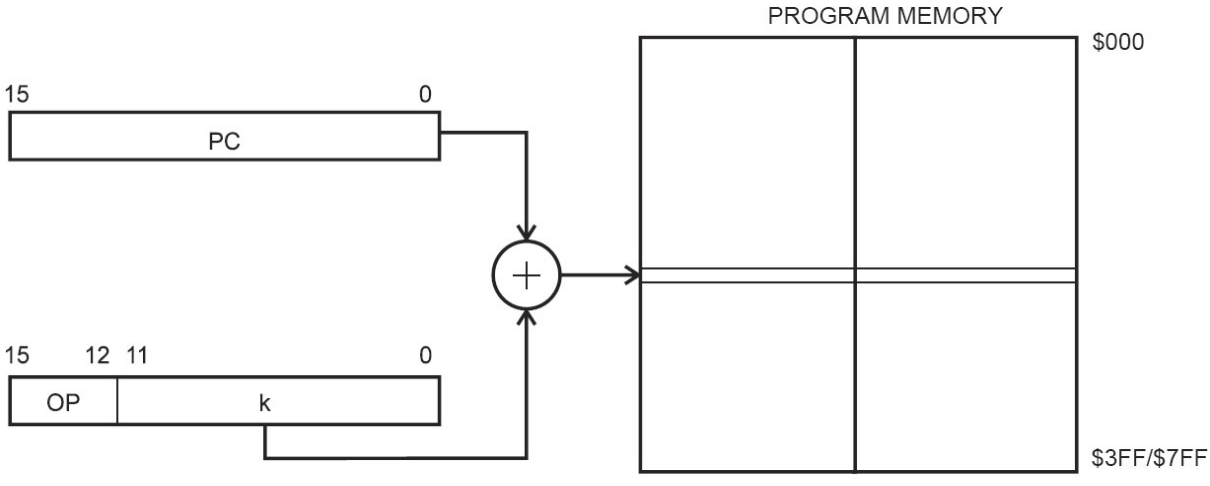


*Рис. 25. Відносна адресація пам'яті програм*

Виконання програми продовжується з адреси PC + k + 1. Значення відносної адреси може бути від -2048 до 2047.

### *EEPROM пам'ять даних*

EEPROM пам'ять даних організована як окремий простір даних з можливістю зчитування і запису окремого байта. EEPROM забезпечує 100000 циклів стирання/запису. Взаємодія між EEPROM і CPU визначається регістром адреси EEPROM, регістром даних EEPROM і регістром керування EEPROM.

### *Пам'ять вводу/виводу (I/O)*

Опис простору I/O мікроконтролерів AT90S2333/4433 представлений в таблиці 2.

*Таблиця 2*

| I/O Address (SRAM Address) | Name | Function |
|---|---|---|
| **$3F ($5F)** | SREG | Status REGister |
| **$3D ($5D)** | SP | Stack Pointer |
| **$3B ($5B)** | GIMSK | General Interrupt MaSK register |
| **$3A ($5A)** | GIFR | General Interrupt Flag Register |
| **$39 ($59)** | TIMSK | Timer/Counter Interrupt MaSK register |
| **$38 ($58)** | TIFR | Timer/Counter Interrupt Flag register |
| **$35 ($55)** | MCUCR | MCU general Control Register |
| **$34 ($54)** | MCUSR | MCU general Status Register |
| **$33 ($53)** | TCCR0 | Timer/Counter0 Control Register |
| **$32 ($52)** | TCNT0 | Timer/Counter0 (8-bit) |
| **$2F ($4F)** | TCCR1A | Timer/Counter1 Control Register A |
| **$2E ($4E)** | TCCR1B | Timer/Counter1 Control Register B |
| **$2D ($4D)** | TCNT1H | Timer/Counter1 High Byte |
| **$2C ($4C)** | TCNT1L | Timer/Counter1 Low Byte |
| **$2B ($4B)** | OCR1H | Timer/Counter1 Output Compare Register High Byte |
| **$2A ($4A)** | OCR1L | Timer/Counter1 Output Compare Register Low Byte |
| **$27 ($47)** | ICR1H | Timer/Counter1 Input Capture Register High Byte |
| **$26 ($46)** | ICR1L | Timer/Counter 1 Input Capture Register Low Byte |
| **$21 ($41)** | WDTCR | Watchdog Timer Control Register |
| **$1E ($3E)** | EEAR | EEPROM Address Register |
| **$1D ($3D)** | EEDR | EEPROM Data Register |
| **$1C ($3C)** | EECR | EEPROM Control Register |
| **$18 ($38)** | PORTB | Data Register, Port B |
| **$17 ($37)** | DDRB | Data Direction Register, Port B |
| **$16 ($36)** | PINB | Input Pins, Port B |
| **$15 ($35)** | PORTC | Data Register, Port C |
| **$14 ($34)** | DDRC | Data Direction Register, Port C |
| **$13 ($33)** | PINC | Input Pins, Port C |
| **$12 ($32)** | PORTD | Data Register, Port D |
| **$11 ($31)** | DDRD | Data Direction Register, Port D |

| | | |
|---|---|---|
| **$10 ($30)** | PIND | Input Pins, Port D |
| **$0F ($2F)** | SPDR | SPI I/O Data Register |
| **$0E ($2E)** | SPSR | SPI Status Register |
| **$0D ($2D)** | SPCR | SPI Control Register |
| **$0C ($2C)** | UDR | UART I/O Data Register |
| **$0B ($2B)** | USR | UART Status Register |
| **$0A ($2A)** | UCR | UART Control Register |
| **$09 ($29)** | UBRR | UART Baud Rate Register |
| **$08 ($28)** | ACSR | Analog Comparator Control and Status Register |
| **$07 ($27)** | ADMUX | ADC Multiplexer Select Register |
| **$06 ($26)** | ADCSR | ADC Control and Status Register |
| **$05 ($25)** | ADCH | ADC Data Register High |
| **$04 ($24)** | ADCL | ADC Data Register Low |
| **$03 ($23)** | UBRRHI | UART Baud Rate Register High |
| **Note:** | Reserved and unused locations are not shown in the table. | |

Примітка: Зарезервовані і комірки, що не використовуються, в таблиці не показані.

Всі засоби I/O і периферії мікроконтролерів AT90S2333/4433 розташовані в просторі I/O. Доступ до простору здійснюється посередництвом інструкцій IN і OUT, які передають дані між регістровим файлом (32 регістри загального призначення) і простором вводу/виводу $00 -. Регістри I/O в діапазоні адрес $00 - $1F мають побітову адресацію за допомогою інструкцій SBI і CBI. Стан кожного окремого біта цих регістрів може бути перевірено командами SBIS і SBIC. Оскільки регістри I/O представлені в адресному просторі SRAM, то до них можна адресуватися як до звичайних комірок SRAM з відповідними адресами. Адреса SRAM отримується простим додаванням $20 до безпосередньої адреси I/O. Адреса SRAM, по всьому документу, наведена в круглих дужках після безпосередньої адреси I/O. Більш детальний опис наведений в розділі Система команд 8-розрядних RISC мікроконтролерів сімейства AVR.

### *Регістр статусу - SREG*

Регістр статусу є частиною адресного простору вводу-виводу і в ньому встановлюються ознаки результату операцій ALU. Регістр статусу - SREG - розташований за адресою $3F ($5F) і його біти визначаються як:

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| $3F ($5F) | I | T | H | S | V | N | Z | C | SREG |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 - I: Global Interrupt Enable** - Дозвіл глобального переривання. Біт дозволу глобального переривання для дозволу переривання має бути встановлений в стан 1. Керування дозволом конкретного переривання виконується регістрами маски переривання GIMSK і TIMSK. Якщо біт глобального переривання очищений (в стані 0), то жодне із дозволених конкретних переривань, установлених в регістрах GIMSK і TIMSK, не діє. Біт I апаратно очищається після переривання і встановлюється для наступного дозволу глобального переривання командою RETI.
- **Bit 6 - T: Bit Copy Storage** - Біт збереження копії. Команди копіювання біта BLD (Bit LoaD) і BST (Bit STore) використовують біт T як біт джерело і біт призначення при операціях з

бітами. Командою BST біт регістра регістрового файлу копіюється в біт T, командою BLD біт T копіюється в регістр регістрового файлу.

- **Bit 5 - H: Half Carry Flag** - Флаг напівпереносу. Флаг напівпереносу вказує на напівперенос в ряді арифметичних операцій. Більш детальна інформація наведена в описі системи команд.
- **Bit 4 - S: Sign Bit, S = N ⊕ V** - Біт знака. Біт S завжди знаходиться в стані, який визначається логічним виключенням АБО (exclusive OR) між флагом від'ємного значення N і доповненням до двох флагу переповнення V. Більш детальна інформація наведена в описі системи команд.
- **Bit 3 - V: Two's Complement Overflow Flag** - Доповнення до двох флагу переповнення. Доповнення до двох флагу V підтримує арифметику доповнення до двох. Більш детальна інформація наведена в описі системи команд.
- **Bit 2 - N: Negative Flag** - Флаг від'ємного значення. Флаг від'ємного значення N вказує на від'ємний результат ряду арифметичних і логічних операцій. Більш детальна інформація наведена в описі системи команд.
- **Bit 1 - Z: Zero Flag** - Флаг нульового значення. Флаг нульового значення Z вказує на нульовий результат ряду арифметичних і логічних операцій. Більш детальна інформація наведена в описі системи команд.
- **Bit 0 - C: Carry Flag** - Флаг переносу. Флаг переносу C вказує на перенос в арифметичних і логічних операціях. Більш детальна інформація наведена в описі системи команд.

Основною ознакою в регістрі стану є флаг нуля, який встановлюється в 1, коли результатом операції є нуль. Коли будуть розглядатися команди процесора, ви побачите, що багато які команди можуть змінити цей Флаг, що робить його незручним для передачі параметрів. Замість нього краще використовувати флаг переносу C, який застосовується для цих цілей іншими процесорами в багатьох додатках. В ряді випадків більш зручним може виявитися використання тимчасового біта T, за допомогою якого можна задавати значення одного біта в якості умови розгалуження програми.

Як і в більшості мікроконтролерів і мікропроцесорів, флаг переносу/запозичення в AVR встановлюється після кожної операції додавання або віднімання. Він також використовується для тимчасового збереження старшого або молодшого біта операнда при операціях звичайних і циклічних зсувів, що типово для багатьох мікроконтролерів. При додаванні флаг переносу встановлюється в 1, якщо результат більше $FF, а при відніманні - якщо результат менше нуля.

Флаг переносу між тетрадами H (напівперенос) встановлюється після виконання додавання або віднімання, в процесі якого відбувся перенос із молодшої або запозичення в старшій тетраді (пів байта). Наприклад, якщо ви додасте 7 до вмісту регістра, в якому зберігалось число 9:

```
add R16, R17                    ; R16 = 9, R17 = 7
```

Регістр R17 містить число 7, і після додавання з вмістом R16, в якому зберігається 9, результатом буде число $10, яке не може бути розміщене в молодшій тетраді. В такому випадку буде встановлено флаг H в регістрі стану. Якщо молодша тетрада результату має значення менше, ніж 10, то флаг H буде скинутий в 0.

Як і флаги напівпереноса для всіх інших контролерів, біт H змінюється при переносі/запозиченні із молодшого шістнадцяткового розряду (тетради). Його зміна не базується на десятковому результаті операції. Але якщо потрібно буде контролювати декадний перенос при операціях з двійково-десятковими числами, біт напівпереносу може бути безпосередньо використаний при виконанні операції додавання, як це показано вище. Якщо результат операції більше або рівний 10, то встановлюється значення флагу H = 1, яке показує, що правильний двійково-десятковий результат може бути отриманий шляхом додавання числа 6.

Використання флагів від'ємного результату N, переповнення V і знаку S може здатися досить важким, якщо розглядати, як вони встановлюються в 1 або скидаються в 0 при арифметичних операціях. Слід зрозуміти їх призначення, і тоді їх застосування стане простим і логічним. Ці біти треба перевіряти тільки після операції додавання або віднімання, в процесі яких використовуються або отримуються відповідні числа, представлені в додатковому коді.

Флаг від'ємного результату N встановлюється в 1 якщо старший (знаковий) розряд результату (біт 7) рівний 1. Коли знаковий розряд встановлений в 1, це часто означає, що результат від'ємний. Слід, однак, відмітити, що знаковий розряд може бути встановлений в 1 і в результаті логічної операції. В цьому випадку знаковий розряд не повинен використовуватися в яких-небудь інших цілях, крім вказівки на те, що він був встановлений в 1 після відповідної операції.

Флаг переповнення V встановлюється при додаванні або відніманні двох чисел зі знаком, представлених в додатковому коді (доповнення до двох), у випадку, якщо значення отриманого результату виходить за границі допустимого діапазону, котрий для восьмибітового числа зі знаком складає від -128 до +127. При додаванні двох додатніх чисел флаг V встановлюється в 1, якщо сума більше 127, у випадку додавання двох від'ємних чисел — якщо результат менше -128. Цей флаг показує, що результат не є правильним 8-розрядним числом зі знаком.

Ви можете сказати, що на справді результат є правильним, тільки його не можна розмістити у 8-розрядному регістрі, і будете праві. Для вирішення цієї ситуації служить флаг знаку S, котрий дозволяє представити результат додавання або віднімання 8-розрядних чисел в додатковому коді у вигляді 9-розрядного числа зі знаком. Якщо після операцій додавання або віднімання чисел зі знаком флаг переповнення V встановлений в 1, то результатом буде 9-розрядное число зі знаком, старшим (знаковим) розрядом числа є флаг S, а вісім бітів результату будуть зберігатися в 8-розрядному регістрі-приймачі.

Біт T — тимчасовий біт, котрий використовується для зберігання результату команд «BST» і «BLD» або для передачі одне бітових параметрів. Хоча біт T не змінюється ніякими іншими командами, він повинен бути збережений разом з другими бітами регістра стану при виконанні переривань або підпрограм, котрі можуть змінити вміст цього регістра.

Останнім є загальний флаг дозволу переривань I. Коли він встановлений в 1, запроси переривань будуть обслуговуватися. Якщо флаг I скинутий в 0, то обслуговування переривань буде відкладено до тих пір, поки цей флаг не буде встановлений в 1.

### *Покажчик стеку - Stack Pointer - SP*

Мікроконтролери AT90S2333/4433 оснащені 8-розрядним покажчиком стеку, розміщеним в регістрі простору I/O за адресою $3D ($5D).

| $3D ($5D) | SP7 | SP6 | SP5 | SP4 | SP3 | SP2 | SP1 | SP0 | SP |
|---|---|---|---|---|---|---|---|---|---|
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Покажчик стеку вказує на область в SRAM даних, в якій розміщуються стеки підпрограм і переривань. Об'єм стеку в SRAM даних повинен бути заданий програмою перед будь-яким викликом підпрограми або обробкою дозволеного переривання. Покажчик стеку повинен вказувати на точку з адресою вище $60. Покажчик стеку декрементується на одиницю, при кожному збереженні даних в стек командою PUSH, і на дві одиниці при занесенні в стек адреси повернення у випадку виклику підпрограми командою RCALL або виникнення переривання.

Покажчик стеку інкрементується на одиницю, при видобуванні даних із стеку командою POP, і на дві одиниці при видобуванні адреси зі стеку при поверненні із підпрограми (RET) або із переривання (RETI).

### *Обробка переривань і скидання*

Мікроконтролери AT90S2333/4433 мають 13 джерел переривань. Ці переривання і вектор скидання мають окремі програмні вектори в просторі пам'яті програм. Кожному перериванню присвоєний свій біт дозволу, який повинен бути встановлений разом з бітом I регістра статусу.

Молодші адреси простору пам'яті програм автоматично визначаються як вектори скидання і переривань.

Повний перелік векторів представлений в Таблиці 3. Перелік представляє також рівень пріоритету для кожного переривання. Переривання з молодшими адресами мають більший рівень пріоритету. RESET має найвищий рівень пріоритету, наступним є INT0 - Запит зовнішнього переривання 0 і т.д.

*Таблиця 3*

| Vector No. | Program Address | Source | Interrupt Definition |
|---|---|---|---|
| **1** | $000 | RESET | External Pin, Power-On Reset, Brown-Out Reset and Watchdog Reset. |
| **2** | $001 | INT0 | External Interrupt Request 0 |
| **3** | $002 | INT1 | External Interrupt Request 1 |
| **4** | $003 | TIMER1 CAPT | Timer/Counter1 Capture Event |
| **5** | $004 | TIMER1 COMP | Timer/Counter1 Compare Match |
| **6** | $005 | TIMER1 OVF | Timer/Counter1 Overflow |
| **7** | $006 | TIMER0 OVF | Timer/Counter0 Overflow |
| **8** | $007 | SPI, STC | Serial Transfer Complete |
| **9** | $008 | UART, RX | UART, Rx Complete |
| **10** | $009 | UART, UDRE | UART Data Register Empty |
| **11** | $00A | UART, TX | UART, Tx Complete |
| **12** | $00B | ADC | ADC Conversion Complete |
| **13** | $00C | EE_RDY | EEPROM Ready |
| **14** | $00D | ANA_COMP | Analog Comparator |

Приклад:

```
Address Labels    Code                 Comments
$000            rjmp RESET          ; Reset Handler
$001            rjmp EXT_INT0       ; IRQ0 Handler
$002            rjmp EXT_INT1       ; IRQ1 Handler
$003            rjmp TIM1_CAPT      ; Timer1 Capture Handler
$004            rjmp TIM1_COMP      ; Timer1 compare Handler
$005            rjmp TIM1_OVF       ; Timer1 Overflow Handler
$006            rjmp TIM0_OVF       ; Timer0 Overflow Handler
$007            rjmp SPI_STC        ; SPI Transfer Complete Handler
$008            rjmp UART_RXC       ; UART RX Complete Handler
$009            rjmp UART_DRE       ; UDR Empty Handler
$00a            rjmp UART_TXC       ; UART TX Complete Handler
$00b            rjmp ADC            ; ADC Conversion Complete Handler
$00c            rjmp EE_RDY         ; EEPROM Ready Handler
$00d            rjmp ANA_COMP       ; Analog Comparator Handler
;
$00e   MAIN:    ldi r16,low(RAMEND); Main program start
$00f            out SP,r16;
$010            <instr> xxx ;
  …      …          …                        …
```

Мікроконтролери AT90S2333/4433 містять два спеціальних 8-розрядних регістра маски переривань: регістр маски зовнішніх переривань GIMSK (General Interrupt Mask) і регістр маски переривань по таймеру/лічильнику TIMSK (Timer/Counter Interrupt Mask). Крім того, в регістрах управління периферією можуть бути організовані й інші біти дозволу і біти маски.

При виникненні переривання біт I дозволу глобального переривання (Global Interrupt Enable) очищується і всі інші переривання забороняються. ПО користувача, з тим, щоб дозволити вкладення переривання, може встановити біт I всередині підпрограми обробки переривання. Вихід із підпрограми обробки переривання відбувається по команді RETI, при цьому біт I встановлюється в стан 1.

Коли програмний лічильник вказує на вектор підпрограми-обробки переривання, відповідний флаг переривання апаратно очищається. Деякі флаги переривань можна очистити, записавши у відповідний біт(и) такого флагу логічну одиницю.

Якщо переривання відбулося в момент коли відповідний біт дозволу переривання (маски) очищений (в нулі), флаг переривання буде встановлений і буде зберігатися в пам'яті до тих пір, поки переривання не буде дозволено або поки флаг не очистять програмно.

Якщо одне або більше переривань відбулося коли біт глобального дозволу переривань I скинутий (в нулі), відповідні флаги переривань будуть встановлені і залишаться в пам'яті до тих пір, поки біт глобального дозволу переривань не буде встановлений (в одиницю) і будуть оброблені у відповідності з пріоритетом.

Увага: при вході в переривання регістр статусу автоматично не зберігається в пам'яті. Це покладається на програму-обробника переривання.

### Регістр маски зовнішніх переривань

General Interrupt Mask Register – GIMSK, знаходиться в I/O області за адресою $3B($5B).

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| $3B ($5B) | INT1 | INT0 | - | - | - | - | - | - | GIMSK |
| Read/Write | R/W | R/W | R | R | R | R | R | R | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 - INT1: External Interrupt Request 1 Enable** – При встановленому біті INT1 і встановленому біті I в регістрі статусу (SREG) дозволяються переривання по входу INT1 мікроконтролера. Визначає переривання (по наростаючому, спадаючому фронту або по логічному рівню) регістр MCUCR (MCU general Control Register). Обробник переривання (вектор) запускається за адресою $002.
- **Bit 6 - INT0: External Interrupt Request 0 Enable** – При встановленому біті INT0 і встановленому біті I в регістра статусу (SREG) дозволяються переривання по входу INT0 мікроконтролера. Визначає переривання (по наростаючому, спадаючому фронту або по логічному рівню) регістр MCUCR (MCU general Control Register). Обробник переривання (вектор) запускається за адресою $001.
- **Bits 5..0 - Res: Reserved bits**.

### Регістр флагів зовнішніх переривань

General Interrupt Flag Register – GIFR, знаходиться в I/O області за адресою $3A($5A).

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| $3A ($5A) | INTF1 | INTF0 | - | - | - | - | - | - | GIFR |
| Read/Write | R/W | R/W | R | R | R | R | R | R | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 - INTF1: External Interrupt Flag 1** – У випадку виникнення запиту переривання на входах INT1, INTF1 буде встановлена 1. Якщо біт I регістра SREG і відповідний біт дозволу INT1 в GIMSK будуть встановлені, то MCU перейде до вектору переривання $002.

При запуску програми-обробника переривання флаг очищається. Крім того, його можна очистити, записавши в нього логічну 1.

- **Bit 6 - INTF0: External Interrupt Flag 0** – У випадку виникнення запиту переривання на входах INT0, INTF0 буде встановлена 1. Якщо біт I регістра SREG і відповідний біт дозволу INT0 в GIMSK будуть встановлені, то MCU перейде до вектору переривання $001. При запуску програми-обробника переривання флаг очищається. Крім того, його можна очистити, записавши в нього логічну 1.
- **Bits 5..0 - Res: Reserved bits**.

### Регістр маски переривань таймера

Timer/Counter Interrupt Mask Register – TIMSK, знаходиться в I/O області за адресою $39($59).

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| $39 ($59) | TOIE1 | OCIE1 | - | - | TICIE1 | - | TOIE0 | - | TIMSK |
| Read/Write | R/W | R/W | R | R | R/W | R | R/W | R | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 - TOIE1: Timer/Counter1 Overflow Interrupt Enable** – При встановленому біті TOIE1 і встановленому біті I регістра статусу дозволяється переривання по переповненню таймера/лічильника1. Відповідне переривання (з вектором $005) виконується якщо відбудеться переповнення таймера/лічильника1, тобто в регістрі флагів TIFR встановиться флаг переповнення таймера/лічильника1 TOV1.
- **Bit 6 - OCIE1: Timer/Counter1 Output Compare Match Interrupt Enable** – При встановленому біті OCIE1 і встановленому біті I регістра статусу дозволяється переривання в порівнянні з вмістом регістрів порівняння і таймера/лічильника1. Відповідне переривання (з вектором $004) виконується якщо відбудеться співпадіння при порівнянні з вмістом регістрів порівняння і таймера/лічильника1, тобто в регістрі флагів переривання TIFR встановиться флаг співпадіння таймера/лічильника1 OCF1.
- **Bits 5..4 - Res: Reserved bits.**
- **Bit 3 - TICIE1: Timer/Counter1 Input Capture Interrupt Enable** – При встановленому біті TICIE1 і встановленому біті I регістра статусу дозволяється переривання по захопленню таймера/лічильника1. Відповідне переривання (з вектором $003) виконується якщо відбудеться захоплення по входу 14, PB0(ICP), тобто в регістрі флагів TIFR встановиться флаг захоплення таймера/лічильника1 ICF1.
- **Bit 2 - Res: Reserved bits**.
- **Bit 1 - TOIE0: Timer/Counter0 Overflow Interrupt Enable** – При встановленому біті TOIE0 і встановленому біті I регістра статусу дозволяється переривання по переповненню таймера/лічильника0. Відповідне переривання (з вектором $006) виконується якщо відбудеться переповнення таймера/лічильника0, тобто в регістрі флагів TIFR встановиться флаг переповнення таймера/лічильника0 TOV0.
- **Bit 0 - Res: Reserved bits**.

### Регістр флагів переривань таймера

Timer/Counter Interrupt Flag Register – TIFR, знаходиться в I/O області за адресою $39($59).

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| $38 ($58) | TOV1 | OCF1 | - | - | ICF1 | - | TOV0 | - | TIFR |
| Read/Write | R/W | R/W | R | R | R/W | R | R/W | R | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 - TOV1: Timer/Counter1 Overflow Flag** – Біт TOV1 встановлюється при переповненні таймера/лічильника1. Він апаратно очищається при обробці відповідного вектора переривання. Можлива очистка біта записом логічної 1.
- **Bit 6 - OCF1: Output Compare Flag 1** – Біт OCF1 встановлюється при співпадінні стану таймера/лічильника1 з вмістом регістра OCR1 (Output Compare Register 1). Флаг OCF1 апаратно очищається при обробці відповідного вектора переривання. Можлива очистка біта записом у флаг логічної 1.
- **Bits 5..4 - Res: Reserved bits.**
- **Bit 3 - ICF1: Input Capture Flag 1** – Біт ICF1встановлюється у випадку захоплення входу і показує, що стан таймера/лічильника1 записано на вхідний регістр захоплення ICR1. Флаг очищається апаратно при обробці відповідного вектора переривання. Можлива очистка біта записом у флаг логічної 1.
- **Bit 2 - Res: Reserved bits**.
- **Bit 1 - TOV0: Timer/Counter0 Overflow Flag** – Біт TOV0 встановлюється при переповненні таймера/лічильника0. Він апаратно очищається при обробці відповідного вектора переривання. Можлива очистка біта записом логічної 1.
- **Bit 0 - Res: Reserved bits**.

Примітка: Більш детальний опис функціонування і настройки системи переривань мікроконтролера AT90S4433 можна знайти на www.atmel.com.

### *8-бітний Таймер/лічильник 0*

8-бітний Таймер/лічильник 0 може бути підключений безпосередньо до тактового генератора (CK), через перед подільник (прескалер), до зовнішнього виводу, а також зупинений. Переключення режимів роботи таймера здійснюється за допомогою регістру TCCR0. Таймер може викликати переривання при переповненні. Настройка переривань здійснюється за допомогою регістрів TIMSK і TIFR. Структурна схема таймера наведена на рис. 26

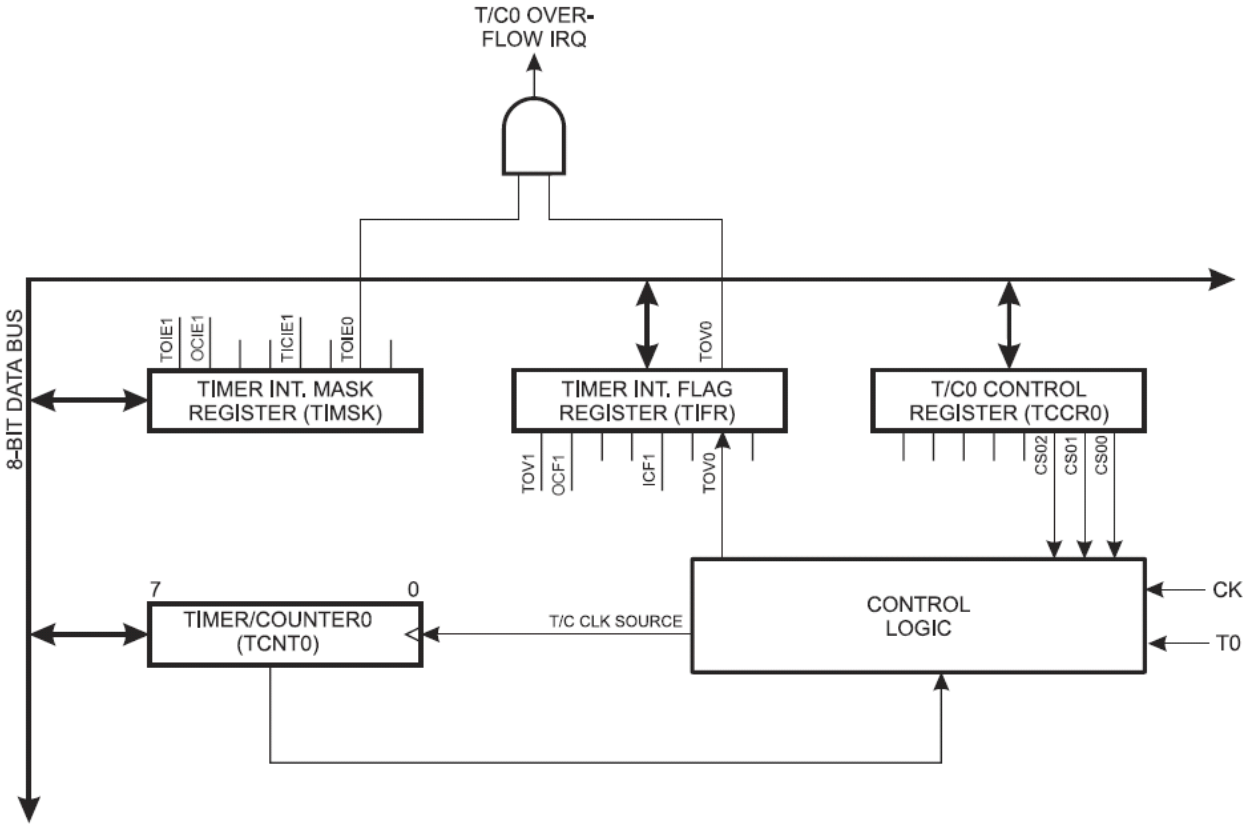
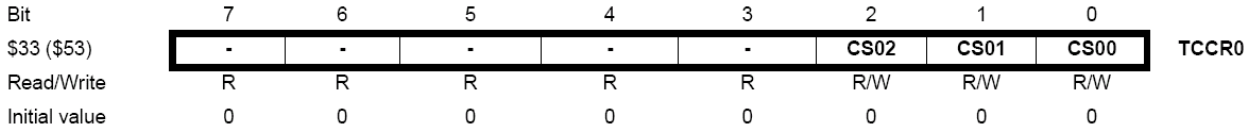



*Рис. 26. Структурна схема таймера/лічильника 0*

### Регістр керування таймером/лічильником 0 – TCCR0

Timer/Counter0 Control Register - TCCR0, знаходиться в I/O області за адресою $33($53).



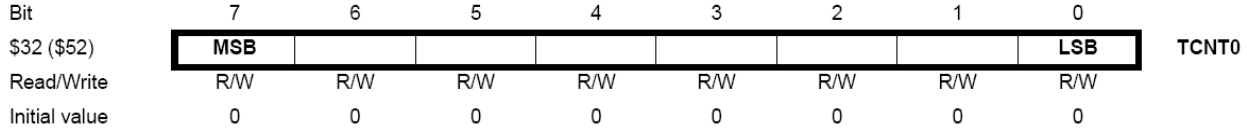

- **Bits 7..3 - Res: Reserved bits**.
- **Bits 2,1,0 - CS02, CS01, CS00: Clock Select0, bit 2,1 and 0** – Біти вибору джерела тактів згідно наведеної нижче таблиці 4.

*Таблиця 4*

| CS02 | CS01 | CS00 | Description |
|---|---|---|---|
| 0 | 0 | 0 | Stop, Timer/Counter0 is stopped. |
| 0 | 0 | 1 | CK |
| 0 | 1 | 0 | CK / 8 |
| 0 | 1 | 1 | CK / 64 |
| 1 | 0 | 0 | CK / 256 |
| 1 | 0 | 1 | CK / 1024 |
| 1 | 1 | 0 | External Pin T0, falling edge |
| 1 | 1 | 1 | External Pin T0, rising edge |

### Регістр керування таймера/лічильника 0 – TCNT0

Timer Counter 0 Register - TCNT0, знаходиться в I/O області за адресою $32($52).




Вміст цього 8-розрядного регістра власне і є таймером/лічильником. Таймер реалізований як лічильник по наростанню (вверх). Регістр доступний в режимі читання/запису. Якщо в таймер/лічильник записано деяке значення і вибране джерело тактового сигналу, то він продовжить рахувати із записаного значення з тактовою частотою джерела після завершення операції запису.

### *Порти вводу/виводу*

Всі порти в архітектурі AVR мають так звану Read-Modify-Write функціональність. Це значить, що в будь-який момент часу будь-яка ніжка порту може бути переключена на ввід або на вивід без будь-якого впливу на інші ніжки порту за допомогою інструкцій CBI і SBI.

Схема підключення зовнішнього виводу, показана на рис 33 10, дає уявлення про його роботу з кожним набором ліній (який називається «портом») зв'язано три адреси вводу-виводу, які дозволяють визначати значення даних, записаних в порт, напрямок передачі даних(«1» — вивід, «0» — ввід) і реальне значення сигналу на зовнішньому виводі. Внаслідок цього є можливість «підтягнути» виводи порту до високого потенціалу для роботи в режимі вводу даних, читання даних може бути виконано або безпосередньо з зовнішнього виводу, або з виходу регістру даних порту. Така можливість є важливою відмінністю роботи порту. Якщо зовнішня лінія перевантажена або випадково закорочена на «землю», то стан зовнішнього виходу ніколи не буде

мінятися. Ось чому в деяких випадках дуже важливо мати можливість прочитати вміст регістру порту і порівняти його з реальним станом зовнішнього виводу.

Хоча лінії вводу-виводу AVR працюють подібно до аналогічних ліній в інших мікроконтролерах, вони мають одну суттєву відмінність. Вона полягає в тому, що «підтягування» зовнішнього виводу до високого потенціалу управляється не окремими бітами регістра, а для цього використовується спеціальна комбінаційна схема. Ця схема дозволяє «підтягування» тільки тоді, коли зовнішній вивід працює в режимі вводу даних.

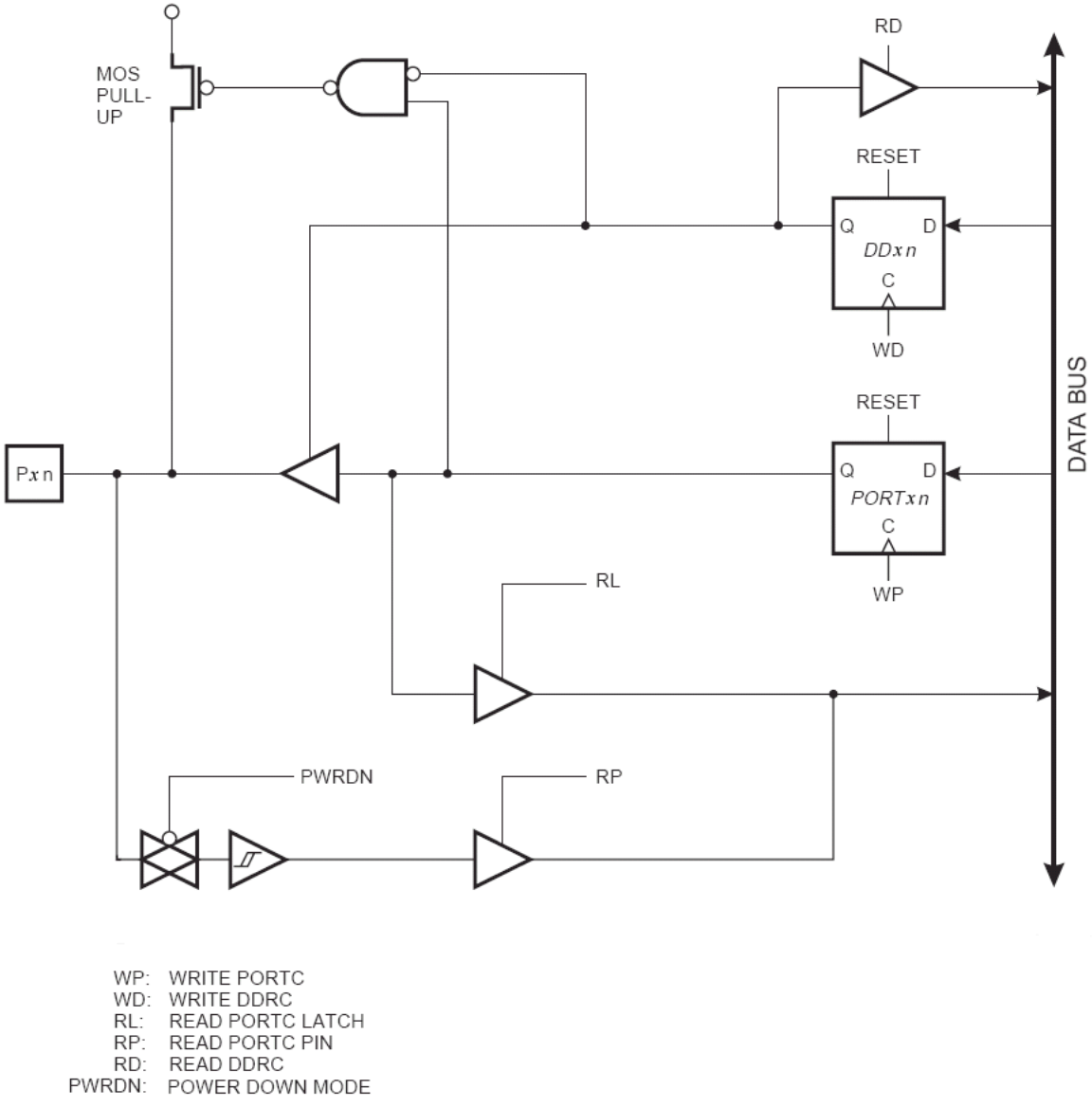



*Рис. 27. Схема підключення зовнішнього виводу порту AVR*

Взаємодія з портом здійснюється за допомогою трьох розташованих в просторі I/O регістрів: регістру даних - PORTx, регістру напрямку передачі даних -, і регістру читання стану входу - PINx. Регістр вводу PINx доступний тільки для читання, регістри даних PORTx і напрямку передачі даних DDRx доступні як для читання, так і для запису. Всі виводи порту оснащені індивідуально підключеними вбудованими підтягуючими резисторами.

Вихідні буфери виводів порту здатні забезпечити струм до 20 мА, що достатньо для прямого керування LED індикатором. Якщо виводи порту використовуються в якості входів і зовнішні 37підтягнуті до низького рівня (нуля), то це може викликати зайве енергоспоживання у випадку активних внутрішніх підтягуючи резисторів до високого рівню (одиниці). Виводи портів можуть також виконувати альтернативні функції, які представлені в таблицях:

*Таблиця 5*

| DDxn | PORTxn | I/O | Pull Up | Comment |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| 0 | 0 | Input | No | Tri-state (Hi-Z) |
| 0 | 1 | Input | Yes | PBn will source current if ext. pulled low. |
| 1 | 0 | Output | No | Push-Pull Zero Output |
| 1 | 1 | Output | No | Push-Pull One Output |

Note n: 5…0, pin number

## Порт B

Port B Pins Alternate Functions *Таблиця 6*

| Port Pin | Alternate Functions |
|---|---|
| PB0 | ICP (Timer/Counter 1 input capture pin) |
| PB1 | OC1 (Timer/Counter 1 output compare match output) |
| PB2 | SS (SPI Slave Select input) |
| PB3 | MOSI (SPI Bus Master Output/Slave Input) |
| PB4 | MISO (SPI Bus Master Input/Slave Output) |
| PB5 | SCK (SPI Bus Serial Clock) |

Примітка: коли використовуються альтернативні функції, регістри DDRB і PORTB повинні бути встановлені як вказано в описі альтернативних функцій.

**Port B Data Register - PORTB**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| $18 ($38) | - | - | PORTB5 | PORTB4 | PORTB3 | PORTB2 | PORTB1 | PORTB0 | PORTB |
| Read/Write | R | R | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

**Port B Data Direction Register - DDRB**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| $17 ($37) | - | - | DDB5 | DDB4 | DDB3 | DDB2 | DDB1 | DDB0 | DDRB |
| Read/Write | R | R | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

**Port B Input Pins Address - PINB**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| $16 ($36) | - | - | PINB5 | PINB4 | PINB3 | PINB2 | PINB1 | PINB0 | PINB |
| Read/Write | R | R | R | R | R | R | R | R | |
| Initial value | 0 | 0 | Hi-Z | Hi-Z | Hi-Z | Hi-Z | Hi-Z | Hi-Z | |

## Порт C

Альтернативна функція – входи 10-бітного вбудованого АЦП. Якщо деякі виводи порту сконфігуровані як виходи, бажано щоби і під час аналого-цифрового перетворення вони не переключались. Це може призвести до похибки результату.

**Port C Data Register - PORTC**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| $15 ($35) | - | - | PORTC5 | PORTC4 | PORTC3 | PORTC2 | PORTC1 | PORTC0 | PORTC |
| Read/Write | R | R | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

**Port C Data Direction Register - DDRC**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| $14 ($34) | - | - | DDC5 | DDC4 | DDC3 | DDC2 | DDC1 | DDC0 | DDRC |
| Read/Write | R | R | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

**Port C Input Pins Address - PINC**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| $13 ($33) | - | - | PINC5 | PINC4 | PINC3 | PINC2 | PINC1 | PINC0 | PINC |
| Read/Write | R | R | R | R | R | R | R | R | |
| Initial value | Q | Q | Hi-Z | Hi-Z | Hi-Z | Hi-Z | Hi-Z | Hi-Z | |

## Порт D

Port D Pins Alternate Functions *Таблиця 7*

| Port Pin | Alternate Function |
|---|---|
| PD0 | RXD (UART Input line) |
| PD1 | TXD (UART Output line) |
| PD2 | INT0 (External interrupt 0 input) |
| PD3 | INT1 (External interrupt 1 input) |
| PD4 | T0 (Timer/Counter 0 external counter input) |
| PD5 | T1 (Timer/Counter 1 external counter input) |
| PD6 | AIN0 (Analog comparator positive input) |
| PD7 | AIN1 (Analog comparator negative input) |

**Port D Data Register - PORTD**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| $12 ($32) | PORTD7 | PORTD6 | PORTD5 | PORTD4 | PORTD3 | PORTD2 | PORTD1 | PORTD0 | PORTD |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

**Port D Data Direction Register - DDRD**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| $11 ($31) | DDD7 | DDD6 | DDD5 | DDD4 | DDD3 | DDD2 | DDD1 | DDD0 | DDRD |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

**Port D Input Pins Address - PIND**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| $10 ($30) | PIND7 | PIND6 | PIND5 | PIND4 | PIND3 | PIND2 | PIND1 | PIND0 | PIND |
| Read/Write | R | R | R | R | R | R | R | R | |
| Initial value | Hi-Z | Hi-Z | Hi-Z | Hi-Z | Hi-Z | Hi-Z | Hi-Z | Hi-Z | |

## AVR-мікроконтролери: засоби розробки

В процесі вибору елементної бази для створення нового пристрою розробник розглядає не тільки технічні характеристики того або іншого мікроконтролера, але також велику увагу присвячує засобам підтримки — як апаратним (стартові набори, програматори, внутрішньосхемні емулятори), так і програмним (мови низького і високого рівня, симулятори). Природно, до уваги береться не тільки зручність роботи і функціональні можливості конкретного пакета програм, але і його вартість. Розглянемо програмні засоби для розробки пристроїв з використанням мікроконтролерів сімейства AVR, що випускаються фірмою ATMEL.

Зараз AVR-мікроконтролери фірми ATMEL завоювали широку популярність на ринку. І це недивно — по кількості моделей в сімействі вони займають перше місце в світі серед Flash-мікроконтролерів і по сукупності своїх характеристик випереджають більшість аналогічних виробів, займаючи одне із перших місць в світі по співставленню ціна/продуктивність. Висока продуктивність досягнута не в останню чергу завдяки потужному і зручному набору команд, який суттєво підвищує ефективність коду в порівнянні з мікроконтролерами класичної архітектури.

Як стане ясно із нижчевикладеного матеріалу, основним інструментом програміста є інтегроване середовище розробки (IDE — INTEGRATED DEVELOPMENT ENVIRONMENT). Ця оболонка включає в себе текстовий редактор, менеджер проектів, відлагоджувач і представляє наступні можливості:

- створення і редагування вихідного коду на асемблері або Сі;
- символьна відлагодження у вихідних кодах;
- перегляд з вмістом FLASH-ROM, EEPROM, SRAM, регістрів і портів вводу/виводу;
- необмежена кількість точок переривання;
- буфер трасування;
- перегляд і модифікація змінних з підтримкою механізму Drag-and-Drop;
- модифікація стану активності виводів портів вводу/виводу;
- спільна робота з усіма внутрішньосхемними емуляторами фірми Atmel.

Архітектура AVR-мікроконтролерів спроектована під компілятори з мов високого рівня. Зокрема, велика кількість регістрів загального призначення зручна для зберігання «регістрових» змінних при написанні програми на С. Цьому сприяє також висока швидкодія мікроконтролерів (час виконання команди складає 100–150 наносекунд) і практично необмежений об’єм пам’яті програм (мікросхема ATmega103 має Flash програм об’ємом 128 Кбайт). Фірми, що випускають С-компілятори для мікроконтролерів, не змушують себе чекати і скоро запропонують відповідні пакети. Із всього різноманіття найбільш цікавий компілятор від шведської фірми IAR Systems.

### *Інтегроване відлагоджувальне середовище AVR Studio фірми Atmel*

AVR Studio 4 - нове професійне інтегроване середовище розробки (Integrated Development Environment - IDE), призначене для написання і відлагодження прикладних програм для AVR мікропроцесорів в середовищі Windows 9x/NT/2000. AVR Studio 4 містить асемблер і симулятор. Також IDE підтримує такі засоби розробки для AVR як: ICE50, ICE40, JTAGICE, ICE200, STK500/501/502 і AVRISP.

AVR Studio підтримує COFF як формат вихідних даних для символьної відлагодження. Інші програмні засоби третіх фірм також можуть бути сконфігуровані для роботи з AVR Studio.

### Вікно вихідного тексту програм

Менеджер проекту (project manager) об’єднує в проект групу файлів і забезпечує інтерфейс для підключення зовнішнього асемблера/компілятора. Таким чином, є можливість написати програму на вибраній мові і компілювати її вибраним компілятором. Потужний текстовий редактор, що входить в AVR Studio, забезпечує «безшовну» стиковку із зовнішнім компілятором і редактором зв’язків. Вихідний код можна редагувати безпосередньо у відлагоджувальному вікні. Підтримується кольорове виділення відповідних фрагментів вихідного коду.

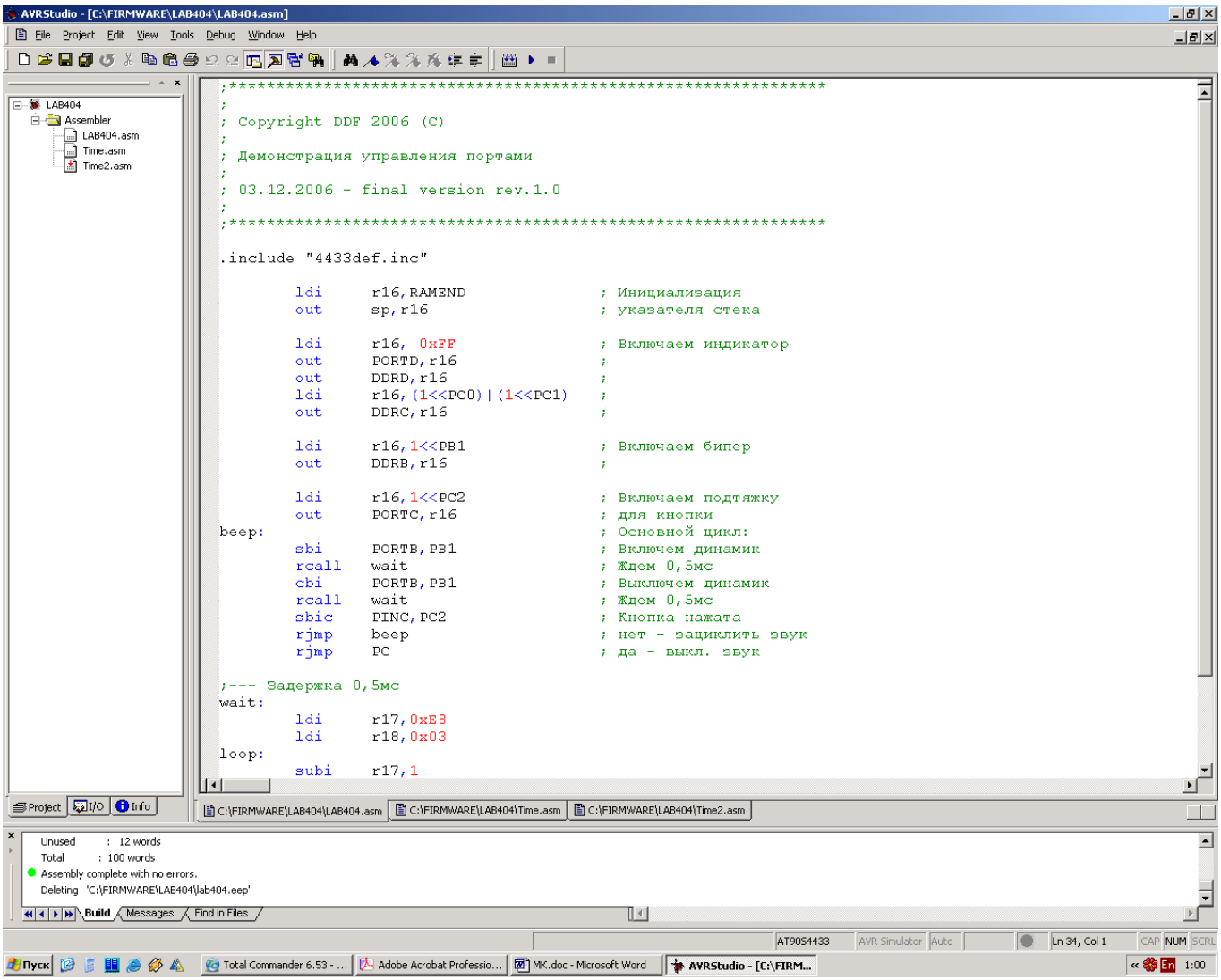



*Рис. 28. Вікно AVR Studio*

Інтерфейс спеціально розроблений для полегшення роботи користувача. Інструментальні панелі (toolbars) і клавіші швидкого доступу забезпечують зручний доступ до всіх ресурсів AVR-мікроконтролера. Установка точок переривання і переключення на вікно вихідного тексту здійснюється одним натиском кнопки миші.

Настройки робочого оточення зберігаються при виході. При першому запускові потрібно налагодити вікна для керування і виводу необхідної інформації і час наступної загрузки автоматично відновлюються.

## Відлагоджувач/симулятор

AVR Studio має засоби для програмної і апаратної відлагодження проектів. У вікні відображається код, який виконується у оточенні відлагоджувача (емуляторі або програмному симуляторі), а текстовий маркер завжди знаходиться на рядку, який буде виконаний в наступному циклі.

Користувач може виконувати програму повністю в покроковому режимі, трасуючи блоки функцій, або виконуючи програму до місця, де стоїть курсор. Можна визначати необмежену кількість точок зупинки, кожна з яких може бути ввімкнена або вимкнена. Точки зупинки зберігаються між сесіями роботи.

У вікні вихідного тексту програми виводиться інформація про процес виконання програми. До того ж, AVR Studio має багато інших вікон, які дозволяють управляти і відображати інформацію про будь-який елемент мікроконтролера.

Список доступних вікон:

- Watch window: Вікно показує значення певних символів. В цьому вікні користувач може проглядати значення і адреси змінних.
- Trace window: Вікно показує хронологію програми, яка виконується в теперішньому часі.
- Register window: Вікно показує вміст регістрів. Регістри можна змінювати під час зупинки програми.
- Memory windows: Вікна показують вміст пам'яті програм, даних, портів вводу/виводу і енергонезалежного ПЗП. Пам'ять можна продивлятися в HEX, двійковому або десятковому форматах. Вміст пам'яті можна змінювати під час зупинки програми.
- I/O window: Показує вміст різних регістрів вводу/виводу:
  - EEPROM
  - I/O порти
  - Таймери і т.д.
- Message window: Вікно показує повідомлення від AVR Studio.
- Processor window: У вікні відображається важлива інформація про ресурси мікроконтролера, включаючи програмний лічильник, покажчик стеку, регістр статусу і лічильник циклу. Ці параметри можуть модифікуватися під час зупинки програми.

Однією із важливих характеристик AVR Studio є вбудована підтримка роботи з внутрішньосхемним емулятором. При запуску програми проводиться запит COM-портів комп'ютера на предмет наявності підключеного емулятора. Якщо на якому-небудь COM-порті виявляється емулятор (в загальному випадку допускається спільна робота кількох емуляторів), AVR Studio стартує в режимі апаратної відлагодження (emulator mode), з'являється напис «emulator» в нижній частині основного вікна, інакше активізується режим симулятора. Інтерфейс користувача в обох випадках ідентичний.

Емулятори AVR ICE 200, ICE PRO і AVR ICE30 мають можливість оновлення конфігурації. Відповідне програмне забезпечення входить до складу AVR Studio. При ініціалізації емулятора відбувається перевірка поточної версії емулятора і при необхідності виводиться вікно з пропозицією «обновити» версію.

В AVR Studio включена підтримка засобів відлагодження фірми Atmel:

- Внутрішньосхемний емулятор Atmel ICEPRO
- Внутрішньосхемний емулятор Atmel MegaICE
- Внутрішньосхемний емулятор Atmel AVRICE
- Внутрішньосхемний емулятор Atmel ICE200
- Внутрішньосхемний емулятор Atmel AsicICE
- Внутрішньосхемний емулятор Atmel ICE10
- Внутрішньосхемний емулятор Atmel ICE30

С AVR Studio також суміщені будь-які програматори і засоби відлагодження, які підтримують мікроконтролери фірми Atmel.

### *Інтегроване середовище розробки IAR Embedded Workbench*

Фірма IAR Systems відома своєю продуктивністю, її продукти підтримують близько двадцяти типів мікроконтролерів різних фірм-виробників. В комплект поставки входить середовище розробки IAR Embedded Workbench і відлагоджувач IAR C-Spy. Девіз фірми «Різні архітектури. Одне рішення». Такий підхід має неоднозначну оцінку користувачів. Єдине середовище розробки полегшує перехід до нового типу мікроконтролера.

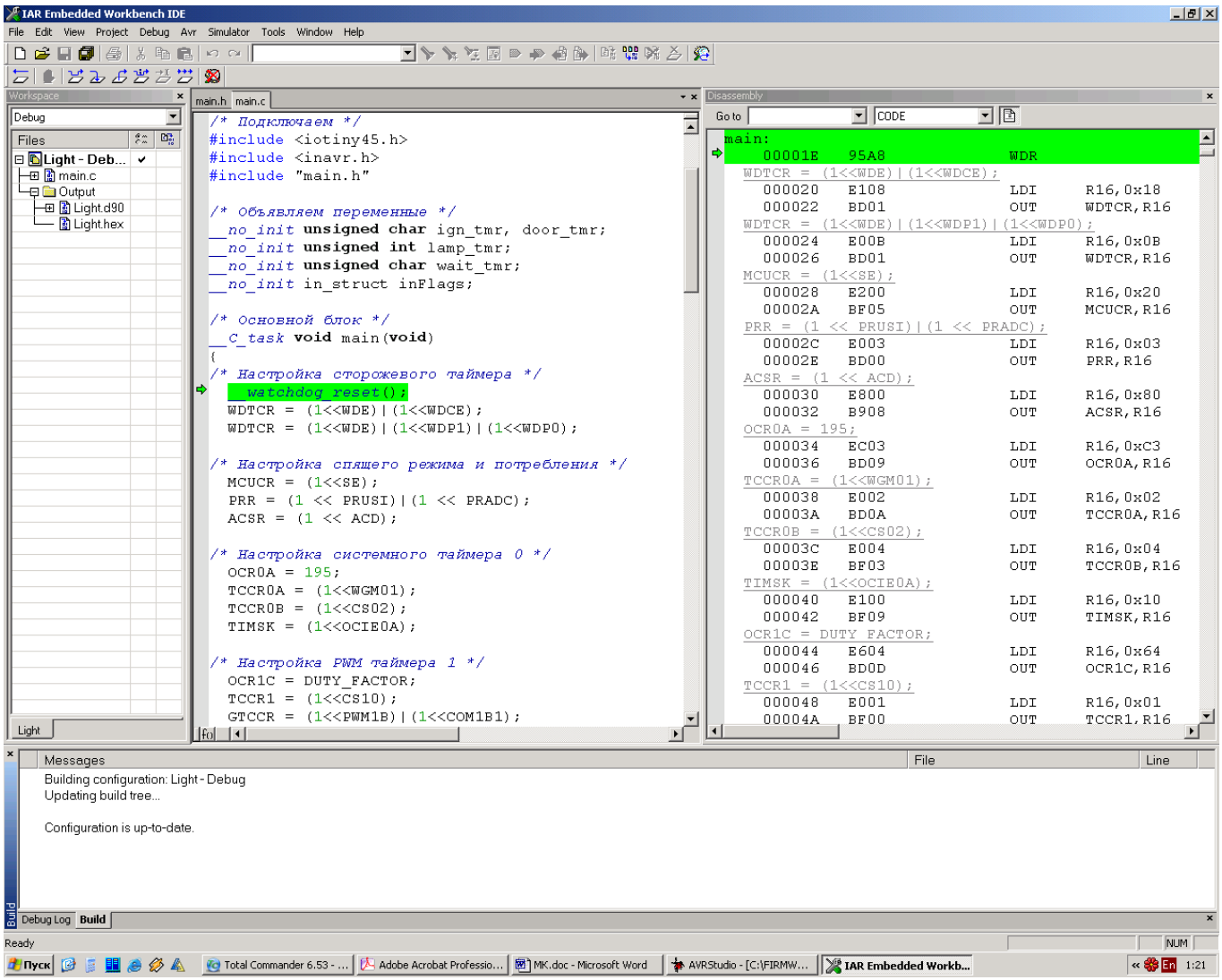



*Рис. 29. С-компілятор фірми IAR*

IAR Systems створила дуже потужний і зручний інструмент програмування мікроконтролерів фірми Atmel сімейства AVR, включаючи Mega. Це середовище для відлагодження, яке працює під керуванням Windows. В неї входять компілятор з мови Сі, асемблер, компонувач, і відлагоджувач, при цьому можлива взаємодія з зовнішніми програмами типу AVR Studio. Вбудований редактор спеціально настроєний на синтаксис мови Сі, а додаткові утиліти і хороша вбудована система допомоги додатково полегшують написання програм.

- Редактор вихідного тексту:
- Зручний інтерфейс користувача
- Автоматичне виділення помилок
- Налаштовувана панель інструментів
- Виділення директив Сі
- Розвиток засобу пошуку
- Компілятор з мови Сі
- Один із кращих компіляторів по ефективності коду
- Повна сумісність з ANSI C
- Кілька моделей для ефективного розподілу пам’яті
- Алгоритми оптимізації спеціально для AVR-мікроконтролерів
- Розширення мови для вбудованих систем
- Асемблер
- Інтегрований макроасемблер для додатків реального часу
- Включає препроцесор для компілятора Сі

- Компонувач
- Підтримує повну компоновку, розташування, і створення формату
- Підтримує більше 30 стандартних вихідних форматів для використання спільно з внутрішньосхемними емуляторами
- Завантаження модулів тільки при необхідності
- Вихідний формат повністю сумісний з AVR Studio
- Симулятор і відлагоджувач
- Відлагодження в кодах Сі і асемблера
- Різні точки зупинки
- Мова опису периферії і операцій вводу/виводу
- Перегляд областей CODE, DATA, EEPROM і регістрів вводу/виводу.
- Обробка переривань з передбаченням
- Контроль будь-яких змінних і стеку
- Комплексні типи даних

## Додаток 1

### *Загальна інформація*

Компілятор транслює вихідні коди з мова асемблера в об'єктний код. Отриманий об'єктний код можна використовувати в симуляторі ATMEL AVR Studio, або в емуляторі ATMEL AVR In-Circuit Emulator. Компілятор також генерує код, який може бути безпосередньо запрограмований в мікроконтролери AVR.

Компілятор генерує код, який не потребує редагування зв'язків.

Компілятор працює під Microsoft Windows 3.11, Microsoft Windows95 і Microsoft Windows NT. Крім цього є консольна версія для MS-DOS.

Набір інструкцій сімейства мікроконтролерів AVR описаний в даному документі коротко, для більш повної інформації по інструкціях звертайтесь до повного опису інструкцій і документації по конкретному мікроконтролеру.

### *Вихідні коди*

Компілятор працює з вихідними файлами, які містять інструкції, мітки і директиви. Інструкції і директиви, як правило, мають один або кілька операндів.

Рядок коду не має бути довше 120 символів.

Будь-який рядок може починатися з мітки, яка є набором символів, що закінчуються двокрапкою. Мітки використовуються для вказівки місця, в яке передається керування при переходах, а також для надання імен змінних.

Вхідний рядок може мати одну із чотирьох форм:

```
[мітка:] директива [операнди] [Коментарій]
[мітка:] інструкція [операнди] [Коментарій]
Коментарій
Пустий рядок
```

Коментарі мають наступну форму:

```
; [Текст]
```

Позиції в квадратних дужках необов'язкові. Текст після крапки з комою (;) і до кінця строки ігнорується компілятором. Мітки, інструкції і директиви більш детально описуються нижче.

### *Приклади:*

```
label:    .EQU var1=100  ; Встановлює var1 рівним 100 (Це директива)
          .EQU var2=200  ; встановлює var2 рівним 200
test:     rjmp test           ; нескінченний цикл (Це інструкція)
                         ; Рядок з лише одним коментарем
                         ; Ще один рядок з коментарем
```

Компілятор не потребує щоб мітки, директиви, коментарії або інструкції находились в певній колонці рядка.

### *Інструкції процесорів AVR*

Нижче наведений набір команд процесорів AVR, більш детальний опис їх можна знайти в AVR Data Book.

### Арифметичні і логічні інструкції

| Мнемоніка | Опе-ранди | Опис | Операція | Флаги | Ци-кли |
|---|---|---|---|---|---|
| **ADD** | Rd,Rr | Підсумовування без переносу | Rd = Rd + Rr | Z,C,N,V,H,S | 1 |
| **ADC** | Rd,Rr | Підсумовування з переносом | Rd = Rd + Rr + C | Z,C,N,V,H,S | 1 |
| **SUB** | Rd,Rr | Віднімання без переносу | Rd = Rd - Rr | Z,C,N,V,H,S | 1 |
| **SUBI** | Rd,K8 | Віднімання константи | Rd = Rd - K8 | Z,C,N,V,H,S | 1 |
| **SBC** | Rd,Rr | Віднімання з переносом | Rd = Rd - Rr - C | Z,C,N,V,H,S | 1 |
| **SBCI** | Rd,K8 | Віднімання константи з переносом | Rd = Rd - K8 - C | Z,C,N,V,H,S | 1 |
| **AND** | Rd,Rr | Логічне І | Rd = Rd · Rr | Z,N,V,S | 1 |
| **ANDI** | Rd,K8 | Логічне І з константою | Rd = Rd · K8 | Z,N,V,S | 1 |
| **OR** | Rd,Rr | Логічне АБО | Rd = Rd V Rr | Z,N,V,S | 1 |
| **ORI** | Rd,K8 | Логічне АБО з константою | Rd = Rd V K8 | Z,N,V,S | 1 |
| **EOR** | Rd,Rr | Логічне виключне АБО | Rd = Rd EOR Rr | Z,N,V,S | 1 |
| **COM** | Rd | Побітна інверсія | Rd = $FF - Rd | Z,C,N,V,S | 1 |
| **NEG** | Rd | Зміна знаку (Дод. код) | Rd = $00 - Rd | Z,C,N,V,H,S | 1 |
| **SBR** | Rd,K8 | Установити біт (біти) в регістрі | Rd = Rd V K8 | Z,C,N,V,S | 1 |
| **CBR** | Rd,K8 | Скинути біт (біти) в регістрі | Rd = Rd · ($FF - K8) | Z,C,N,V,S | 1 |
| **INC** | Rd | Інкрементувати значення регістра | Rd = Rd + 1 | Z,N,V,S | 1 |
| **DEC** | Rd | Декрементувати значення регістра | Rd = Rd -1 | Z,N,V,S | 1 |
| **TST** | Rd | Перевірка на алель або від'ємність | Rd = Rd · Rd | Z,C,N,V,S | 1 |
| **CLR** | Rd | Очистити регістр | Rd = 0 | Z,C,N,V,S | 1 |
| **SER** | Rd | Установити регістр | Rd = $FF | None | 1 |
| **ADIW** | Rdl,K6 | Додати константу і слово | Rdh:Rdl = Rdh:Rdl + K6 | Z,C,N,V,S | 2 |
| **SBIW** | Rdl,K6 | Відняти константу із слова | Rdh:Rdl = Rdh:Rdl - K 6 | Z,C,N,V,S | 2 |
| **MUL** | Rd,Rr | Множення чисел без знаку | R1:R0 = Rd * Rr | Z,C | 2 |
| **MULS** | Rd,Rr | Множення чисел зі знаком | R1:R0 = Rd * Rr | Z,C | 2 |
| **MULSU** | Rd,Rr | Множення числа зі знаком з числом без знаку | R1:R0 = Rd * Rr | Z,C | 2 |
| **FMUL** | Rd,Rr | Множення дробових чисел без знаку | R1:R0 = (Rd * Rr) << 1 | Z,C | 2 |
| **FMULS** | Rd,Rr | Множення дробових чисел зі знаком | R1:R0 = (Rd *Rr) << 1 | Z,C | 2 |
| **FMULSU** | Rd,Rr | Множення дробового числа зі знаком з числом без знаку | R1:R0 = (Rd * Rr) << 1 | Z,C | 2 |

### Інструкції розгалуження

| Мнемоніка | Опе-ранди | Опис | Операція | Флаги | Цикли |
|---|---|---|---|---|---|
| **RJMP** | k | Відносний перехід | PC = PC + k +1 | None | 2 |
| **IJMP** | Нема | Непрямий перехід на (Z) | PC = Z | None | 2 |
| **EIJMP** | Нема | Розширений непрямий перехід на (Z) | STACK =PC+1, PC(15:0) = Z, PC(21:16) = EIND | None | 2 |
| **JMP** | k | Перехід | PC = k | None | 3 |
| **RCALL** | k | Відносний виклик підпрограми | STACK=PC+1, PC=PC+k +1 | None | 3/4* |
| **ICALL** | Нема | Непрямий виклик (Z) | STACK = PC+1, PC = Z | None | 3/4* |
| **EICALL** | Нема | Розширений непрямий виклик (Z) | STACK =PC+1, PC(15:0) = Z, PC(21:16) =EIND | None | 4* |
| **CALL** | k | Виклик підпрограми | STACK = PC+2, PC = k | None | 4/5* |
| **RET** | Нема | Повернення із підпрограми | PC = STACK | None | 4/5* |
| **RETI** | Нема | Повернення із переривання | PC = STACK | I | 4/5* |
| **CPSE** | Rd,Rr | Порівняти, пропустити якщо рівні | if (Rd ==Rr) PC = PC 2 or 3 | None | 1/2/3 |
| **CP** | Rd,Rr | Порівняти | Rd -Rr | Z,C,N,V, H,S | 1 |
| **CPC** | Rd,Rr | Порівняти з переносом | Rd - Rr - C | Z,C,N,V, H,S | 1 |
| **CPI** | Rd,K8 | Порівняти з константою | Rd - K | Z,C,N,V, H,S | 1 |
| **SBRC** | Rr,b | Пропустити якщо біт в регістрі очищений | if(Rr(b)==0) PC = PC + 2 or 3 | None | 1/2/3 |
| **SBRS** | Rr,b | Пропустити якщо біт в регістрі встановлений | if(Rr(b)==1) PC = PC + 2 or 3 | None | 1/2/3 |
| **SBIC** | P,b | Пропустити якщо біт в порту очищений | if(I/O(P,b)==0) PC=PC+2 or 3 | None | 1/2/3 |
| **SBIS** | P,b | Пропустити якщо біт в порту встановлений | if(I/O(P,b)==1) PC=PC+2 or 3 | None | 1/2/3 |
| **BRBC** | s,k | Перейти якщо флаг в SREG очищений | if(SREG(s)==0) PC=PC+k +1 | None | 1/2 |
| | | | | | |
| **Мнемоніка** | Опе-ранди | Опис | Операція | Флаги | Цикли |
| **BRBS** | s,k | Перейти якщо флаг в SREG встановлений | if(SREG(s)==1) PC=PC+k+1 | None | 1/2 |
| **BREQ** | k | Перейти якщо рівні | if(Z==1) PC = PC + k + 1 | None | 1/2 |
| **BRNE** | k | Перейти якщо не рівні | if(Z==0) PC = PC + k + 1 | None | 1/2 |
| **BRCS** | k | Перейти якщо перенос встановлений | if(C==1) PC = PC + k + 1 | None | 1/2 |
| **BRCC** | k | Перейти якщо перенос очищений | if(C==0) PC = PC + k + 1 | None | 1/2 |
| **BRSH** | k | Перейти якщо рівні або більше | if(C==0) PC = PC + k + 1 | None | 1/2 |
| **BRLO** | k | Перейти якщо менше | if(C==1) PC = PC + k + 1 | None | 1/2 |
| **BRMI** | k | Перейти якщо мінус | if(N==1) PC = PC + k + 1 | None | 1/2 |
| **BRPL** | k | Перейти якщо плюс | if(N==0) PC = PC + k + 1 | None | 1/2 |

| | | | | | |
|---|---|---|---|---|---|
| **BRGE** | k | Перейти якщо більше або рівні (зі знаком) | if(S==0) PC = PC + k + 1 | None | 1/2 |
| **BRLT** | k | Перейти якщо менше (зі знаком) | if(S==1) PC = PC + k + 1 | None | 1/2 |
| **BRHS** | k | Перейти якщо флаг внутрішнього переносу встановлений | if(H==1) PC = PC + k + 1 | None | 1/2 |
| **BRHC** | k | Перейти якщо флаг внутрішнього переносу очищений | if(H==0) PC = PC + k + 1 | None | 1/2 |
| **BRTS** | k | Перейти якщо флаг T встановлений | if(T==1) PC = PC + k + 1 | None | 1/2 |
| **BRTC** | k | Перейти якщо флаг T очищений | if(T==0) PC = PC + k + 1 | None | 1/2 |
| **BRVS** | k | Перейти якщо флаг переповнення встановлений | if(V==1) PC = PC + k + 1 | None | 1/2 |
| **BRVC** | k | Перейти якщо флаг переповнення очищений | if(V==0) PC = PC + k + 1 | None | 1/2 |
| **BRIE** | k | Перейти якщо переривання дозволено | if(I==1) PC = PC + k + 1 | None | 1/2 |
| **BRID** | k | Перейти якщо переривання заборонено | if(I==0) PC = PC + k + 1 | None | 1/2 |

* Для операцій доступу до даних кількість циклів вказано при умові доступу до внутрішньої пам’яті даних, і не коректно при роботі з зовнішнім ОЗП. Для інструкцій CALL, ICALL, EICALL, RCALL, RET і RETI, необхідно додати три цикли плюс по два цикли для кожного чекання в контролерах з PC менше 16 біт (128KB пам’яті програм). Для пристроїв з пам’яттю програм вище 128KB , додайте п’ять циклів плюс по три цикли на кожне чекання.

### Інструкції передачі даних

| Мнемоніка | Операнди | Опис | Операція | Флаги | Цикли |
|---|---|---|---|---|---|
| **MOV** | Rd,Rr | Скопіювати регістр | Rd = Rr | None | 1 |
| **MOVW** | Rd,Rr | Скопіювати пару регістрів | Rd+1: Rd=Rr+1:Rr, r,d even | None | 1 |
| **LDI** | Rd,K8 | Завантажити константу | Rd = K | None | 1 |
| **LDS** | Rd,k | Пряме завантаження | Rd = (k) | None | 2* |
| **LD** | Rd,X | Непряме завантаження | Rd = (X) | None | 2* |
| **LD** | Rd,X+ | Непряме завантаження з пост-інкрементом | Rd = (X), X=X+1 | None | 2* |
| **LD** | Rd,-X | Непряме завантаження з пре-декрементом | X=X-1, Rd = (X) | None | 2* |
| **LD** | Rd,Y | Непряме завантаження | Rd = (Y) | None | 2* |
| **LD** | Rd,Y+ | Непряме завантаження з пост-інкрементом | Rd = (Y), Y=Y+1 | None | 2* |
| **LD** | Rd,-Y | Непряме завантаження з пре-декрементом | Y=Y-1, Rd = (Y) | None | 2* |
| **LDD** | Rd,Y+q | Непряме завантаження з заміщенням | Rd = (Y+q) | None | 2* |
| **LD** | Rd,Z | Непряме завантаження | Rd = (Z) | None | 2* |
| **LD** | Rd,Z+ | Непряме завантаження з пост-інкрементом | Rd = (Z), Z=Z+1 | None | 2* |
| **LD** | Rd,-Z | Непряме завантаження з пре-декрементом | Z=Z-1, Rd = (Z) | None | 2* |
| **LDD** | Rd,Z+q | Непряме завантаження з заміщенням | Rd = (Z+q) | None | 2* |
| **STS** | k,Rr | Пряме збереження | (k) = Rr | None | 2* |
| **ST** | X,Rr | Непряме збереження | (X) = Rr | None | 2* |
| **ST** | X+,Rr | Непряме збереження з пост-інкрементом | (X) = Rr, X=X+1 | None | 2* |
| **ST** | -X,Rr | Непряме збереження з пре-декрементом | X=X-1, (X)=Rr | None | 2* |
| **ST** | Y,Rr | Непряме збереження | (Y) = Rr | None | 2* |
| **ST** | Y+,Rr | Непряме збереження з пост-інкрементом | (Y) = Rr, Y=Y+1 | None | 2 |
| **ST** | -Y,Rr | Непряме збереження з пре-декрементом | Y=Y-1, (Y) = Rr | None | 2 |
| **ST** | Y+q,Rr | Непряме збереження з заміщенням | (Y+q) = Rr | None | 2 |
| **ST** | Z,Rr | Непряме збереження | (Z) = Rr | None | 2 |
| **ST** | Z+,Rr | Непряме збереження з пост-інкрементом | (Z) = Rr, Z=Z+1 | None | 2 |
| **ST** | -Z,Rr | Непряме збереження з пре-декрементом | Z=Z-1, (Z) = Rr | None | 2 |
| **ST** | Z+q,Rr | Непряме збереження з заміщенням | (Z+q) = Rr | None | 2 |
| **LPM** | Нет | Завантаження із програмної пам'яті | R0 = (Z) | None | 3 |
| **LPM** | Rd,Z | Завантаження із програмної пам'яті | Rd = (Z) | None | 3 |
| **Мнемоніка** | Операнди | Опис | Операція | Флаги | Цикли |
| **LPM** | Rd,Z+ | Завантаження із програмної пам'яті з пост-інкрементом | Rd = (Z), Z=Z+1 | None | 3 |
| **ELPM** | Нема | Розширене завантаження із | R0 = (RAMPZ:Z) | None | 3 |

| | | | | | |
|---|---|---|---|---|---|
| | | програмної пам'яті | | | |
| **ELPM** | Rd,Z | Розширене завантаження із програмної пам'яті | Rd = (RAMPZ:Z) | None | 3 |
| **ELPM** | Rd,Z+ | Розширене завантаження із програмної пам'яті з пост-інкрементом | Rd = (RAMPZ:Z), Z = Z+1 | None | 3 |
| **SPM** | Нема | Збереження в програмній пам'яті | (Z) = R1:R0 | None | - |
| **ESPM** | Нема | Розширене збереження в програмній пам'яті | (RAMPZ:Z) = R1:R0 | None | - |
| **IN** | Rd,P | Читання з порта | Rd = P | None | 1 |
| **OUT** | P,Rr | Запис в порт | P = Rr | None | 1 |
| **PUSH** | Rr | Занесення регістру в стек | STACK = Rr | None | 2 |
| **POP** | Rd | Видобування регістру із стеку | Rd = STACK | None | 2 |

* Для операцій доступу до даних кількість циклів вказано при умові доступу до внутрішньої пам'яті даних, і не коректне при роботі з зовнішнім ОЗП. Для інструкцій LD, ST, LDD, STD, LDS, STS, PUSH і POP, необхідно додати один цикл плюс по одному циклу для кожного чекання.

### Інструкції роботи з бітами

| Мнемо-ніка | Опе-ранди | Опис | Операція | Флаги | Ци-кли |
|---|---|---|---|---|---|
| **LSL** | Rd | Логічний зсув вліво | Rd(n+1)=Rd(n), Rd(0)=0, C=Rd(7) | Z,C,N,V,H,S | 1 |
| **LSR** | Rd | Логічний зсув вправо | Rd(n)=Rd(n+1), Rd(7)=0, C=Rd(0) | Z,C,N,V,S | 1 |
| **ROL** | Rd | Циклічний зсув вліво через C | Rd(0)=C, Rd(n+1)=Rd(n), C=Rd(7) | Z,C,N,V,H,S | 1 |
| **ROR** | Rd | Циклічний зсув вправо через C | Rd(7)=C, Rd(n)=Rd(n+1), C=Rd(0) | Z,C,N,V,S | 1 |
| **ASR** | Rd | Арифметичний зсув вправо | Rd(n)=Rd(n+1), n=0,...,6 | Z,C,N,V,S | 1 |
| **SWAP** | Rd | Перестановка тетрад | Rd(3..0) = Rd(7..4), Rd(7..4) = Rd(3..0) | None | 1 |
| **BSET** | s | Установка флагу | SREG(s) = 1 | SREG(s) | 1 |
| **BCLR** | s | Очистка флагу | SREG(s) = 0 | SREG(s) | 1 |
| **SBI** | P,b | Встановити біт в порту | I/O(P,b) = 1 | None | 2 |
| **CBI** | P,b | Очистити біт в порту | I/O(P,b) = 0 | None | 2 |
| **Мнемо-ніка** | Опе-ранди | Опис | Операція | Флаги | Ци-кли |
| **BST** | Rr,b | Зберегти біт із регістра в T | T = Rr(b) | T | 1 |
| **BLD** | Rd,b | Завантажити біт із T в регістр | Rd(b) = T | None | 1 |
| **SEC** | Нема | Встановити флаг переносу | C =1 | C | 1 |
| **CLC** | Нема | Очистити флаг переносу | C = 0 | C | 1 |
| **SEN** | Нема | Встановити флаг від'ємного числа | N = 1 | N | 1 |
| **CLN** | Нема | Очистити флаг від'ємного числа | N = 0 | N | 1 |
| **SEZ** | Нема | Встановити флаг нуля | Z = 1 | Z | 1 |

| | | | | | |
|---|---|---|---|---|---|
| **CLZ** | Нема | Очистити флаг нуля | Z = 0 | Z | 1 |
| **SEI** | Нема | Встановити флаг переривань | I = 1 | I | 1 |
| **CLI** | Нема | Очистити флаг переривань | I = 0 | I | 1 |
| **SES** | Нема | Встановити флаг числа зі знаком | S = 1 | S | 1 |
| **CLN** | Нема | Очистити флаг числа зі знаком | S = 0 | S | 1 |
| **SEV** | Нема | Встановити флаг переповнення | V = 1 | V | 1 |
| **CLV** | Нема | Очистити флаг переповнення | V = 0 | V | 1 |
| **SET** | Нема | Встановити флаг T | T = 1 | T | 1 |
| **CLT** | Нема | Очистити флаг T | T = 0 | T | 1 |
| **SEH** | Нема | Встановити флаг внутрішнього переносу | H = 1 | H | 1 |
| **CLH** | Нема | Очистити флаг внутрішнього переносу | H = 0 | H | 1 |
| **NOP** | Нема | Немає операції | Нема | None | 1 |
| **SLEEP** | Нема | Спати (зменшити енергоспоживання) | Див. опис інструкції | None | 1 |
| **WDR** | Нема | Скидання сторожового таймера | Див. опис інструкції | None | 1 |

Асемблер не розрізняє регістр символів.

Операнди можуть бути таких видів:

- Rd: Результуючий (і вихідний) регістр в регістровому файлі
- Rr: Вихідний регістр в регістровому файлі
- b: Константа (3 біта), може бути константний вираз
- s: Константа (3 біта), може бути константний вираз
- P: Константа (5-6 біт), може бути константний вираз
- K6; Константа (6 біт), може бути константний вираз
- K8: Константа (8 біт), може бути константний вираз
- k: Константа (розмір залежить від інструкції), може бути константний вираз
- q: Константа (6 біт), може бути константний вираз
- Rdl: R24, R26, R28, R30. Для інструкцій ADIW і SBIW
- X,Y,Z: Регістри непрямої адресації (X=R27:R26, Y=R29:R28, Z=R31:R30)

### Директиви асемблера

Компілятор підтримує ряд директив. Директиви не транслюються безпосередньо в код. Замість цього вони використовуються для вказівки положення в програмній пам’яті, визначення макросів, ініціалізації пам’яті і т.д. Список директив наведений в наступній таблиці.

| Директива | Опис |
|---|---|
| BYTE | Зарезервувати байти в ОЗП |
| CSEG | Програмний сегмент |
| DB | Обмежити байти і флеш або EEPROM |
| DEF | Назначити регістру символічне ім’я |
| DEVICE | Обмежити пристрій для якого компілюється програма |
| DSEG | Сегмент даних |
| DW | Обмежити слова і флеш або EEPROM |
| ENDM, ENDMACRO | Кінець макросу |

| | |
|---|---|
| EQU | Встановити постійний вираз |
| ESEG | Сегмент EEPROM |
| EXIT | Вийти із файлу |
| INCLUDE | Вкласти інший файл |
| LIST | Ввімкнути генерацію лістингу |
| LISTMAC | Ввімкнути розвертання макросів в лістингу |
| MACRO | Початок макросу |
| NOLIST | Вимкнути генерацію лістингу |
| ORG | Встановити положення в сегменті |
| SET | Встановити змінний символічний еквівалент виразу |

Всі директиви відділяються точкою.

**BYTE - Зарезервувати байти в ОЗП**

Директива BYTE резервує байти в ОЗП. Якщо Ви хочете мати можливість посилатися на виділену область пам'яті, то директива BYTE має бути відділена міткою. Директива приймає один обов'язковий параметр, який вказує кількість виділених байт. Ця директива може використовуватися тільки в сегменті даних(див. директиви CSEG і DSEG). Виділені байти не ініціалізуються.

**Синтаксис:**

```
MITKA: .BYTE вираз
```

**Приклад:**

```
.DSEG
var1:  .BYTE 1              ; резервує 1 байт для var1
table: .BYTE tab_size       ; резервує tab_size байт
.CSEG
               ldi r30,low(var1)  ; Загружає молодший байт регістра
Z
               ldi r31,high(var1) ; Загружає старший байт регістра Z

               ld r1,Z            ; Загружає VAR1 в регістр 1
```

**CSEG - Програмний сегмент**

Директива CSEG визначає початок програмного сегмента. Вихідний файл може складатися з кількох програмних сегментів, які об'єднуються в один програмний сегмент при компіляції. Програмний сегмент є сегментом за замовчуванням. Програмні сегменти мають свої власні лічильники положення, які рахують не побайтово, а по словах. Директива ORG може бути використана для розташування коду і констант в необхідному місці сегменту. Директива CSEG не має параметрів.

**Синтаксис:**

```
.CSEG
```

**Приклад:**

```
.DSEG                       ; Початок сегмента даних
vartab:      .BYTE 4        ; Резервує 4 байта в ОЗП
.CSEG                       ; Початок кодового сегменту
const:       .DW 2          ; Розмістити константу 0x0002 в пам'яті
```

```
програм
                              mov r1,r0      ; Виконати дії
```

**DB - Обмежити байти і флеш або EEPROM**

Директива DB резервує необхідну кількість байт в пам'яті програм або в EEPROM. Якщо Ви хочете мати можливість посилатися на виділену область пам'яті, то директива DB повинна бути помічена міткою. Директива DB повинна мати хоча би один параметр. Дана директива може бути розміщена тільки в сегменті програм (CSEG) або в сегменті EEPROM (ESEG).

Параметри, які передаються директиві - це послідовність виразів розділених комами. Кожен вираз моє бути або числом в діапазоні (-128..255), або в результаті обчислення має давати результат в цьому ж діапазоні, в протилежному випадку число скорочується до байта, причому без видачі попереджень.

Якщо директива отримує більше одного параметра і поточним є програмний сегмент, то параметри упаковуються в слова (перший параметр - молодший байт), і якщо число параметрів непарне, то останній вираз буде скорочений до байта і записаний як слово зі старшим байтом рівним нулю, навіть якщо далі йде ще одна директива DB.

**Синтаксис:**

```
МІТКА: .DB список_виразів
```

**Приклад:**

```
.CSEG
consts: .DB 0, 255, 0b01010101, -128, 0xaa
.ESEG
const2: .DB 1,2,3
```

**DEF - Назначити регістру символічне ім'я**

Директива DEF дозволяє посилатися на регістр через деяке символічне ім'я. Визначене ім'я можуть використовувати і всі наступні частини програми для звернення до даного регістру. Регістр може мати кілька різних імен. Символічне ім'я може бути переназначено пізніше в програмі.

**Синтаксис:**

```
.DEF Символічне_ім'я = Регістр
```

**Приклад:**

```
.DEF temp = R16
.DEF ior  = R0
.CSEG
  ldi temp,0xf0       ; Завантажити 0xf0 в регістр temp (R16)
  in  ior,0x3f        ; Прочитати SREG в регістр ior (R0)
  eor temp, ior       ; Регістри temp і ior додаються по виключному
або
```

**DEVICE - Обмежити пристрій, для якого компілюється програма**

Директива DEVICE дозволяє вказати для якого пристрою компілюється програма. При використанні даної директиви компілятор видає попередження, якщо буде знайдена інструкція, яку не підтримує даний мікроконтролер. Також буде видано попередження, якщо програмний сегмент, або сегмент EEPROM перевищать розмір, допустимий пристроєм. Якщо ж директива не використовується, то всі інструкції вважаються допустимими, і відсутні обмеження на розмір сегментів.

**Синтаксис:**

```
.DEVICE AT90S1200 |AT90S2313 | AT90S2323 | AT90S2333 | AT90S2343 |
AT90S4414 | AT90S4433 | AT90S4434 | AT90S8515 | AT90S8534 |
AT90S8535 | ATtiny11 |  ATtiny12 | ATtiny22 | ATmega603 | ATmega103
```

**Приклад:**

```
.DEVICE AT90S1200       ; Використовується AT90S1200
.CSEG
                push r30        ; Ця інструкція викликає
попередження
                                                ; оскільки AT90S1200
її не має
```

**DSEG - Сегмент даних**

Директива DSEG визначає початок сегменту даних. Вихідний файл може складатися із кількох сегментів даних, які об'єднуються в один сегмент при компіляції. Сегмент даних зазвичай складається тільки із директив BYTE і міток. Сегменти даних мають свої власні побайтні лічильники положення. Директива ORG може бути використана для розташування змінних в необхідному місці ОЗУ. Директива не має параметрів.

**Синтаксис:**

```
.DSEG
```

**Приклад:**

```
.DSEG                   ; Початок сегмента даних
var1:    .BYTE 1        ; зарезервувати 1 байт для var1
table:   .BYTE tab_size      ; зарезервувати tab_size байт.
.CSEG
    ldi r30,low(var1)   ; Завантажити молодший байт регістра Z
    ldi r31,high(var1)  ; Завантажити старший байт регістра Z
    ld r1,Z             ; Завантажити var1 в регістр r1
```

**DW - Обмежити слова і флеш або EEPROM**

Директива DW резервує необхідну кількість слів в пам'яті програм або в EEPROM. Якщо Ви хочете мати можливість посилатися на виділену область пам'яті, то директива DW повинна бути відділена міткою. Директива DW повинна мати хоча би один параметр. Дана директива може бути розміщена тільки в сегменті програм (CSEG) або в сегменті EEPROM (ESEG).

Параметри, які передаються директиві - це послідовність виразів, розділених комам. Кожен вираз має бути або числом в діапазоні (-32768..65535), або в результаті обчислення повинен давати результат в цьому ж діапазоні, в протилежному випадку число усікається до слова, причому без видачі попереджень.

**Синтаксис:**

```
MITKA:    .DW expressionlist
```

**Приклад:**

```
.CSEG
varlist:  .DW 0, 0xffff, 0b1001110001010101, -32768, 65535
.ESEG
eevarlst:      .DW 0,0xffff,10
```

**ENDMACRO - Кінець макросу**

Директива визначає кінець макросу, і не приймає ніяких параметрів. Для інформації по визначенню макросів див. директиву MACRO.

**Синтаксис:**

```
.ENDMACRO
```

**Приклад:**

```
.MACRO SUBI16            ; Початок визначення макросу
       subi r16,low(@0)      ; Вирахувати молодший байт першого
параметра
       sbci r17,high(@0)     ; Вирахувати старший байт першого
параметра
.ENDMACRO
```

**EQU - Встановити постійний вираз**

Директива EQU присвоює мітці значення. Ця мітка може пізніше використовуватися в виразах. Мітка, якій присвоєне значення даною директивою, не може бути перевизначена і її значення не може бути змінене.

**Синтаксис:**

```
.EQU мітка = вираз
```

**Приклад:**

```
.EQU  io_offset = 0x23
.EQU  porta       = io_offset + 2
.CSEG               ; Початок сегмента даних
    clr r2          ; Очистити регістр r2
    out porta,r2    ; Записати в порт A
```

**ESEG - Сегмент EEPROM**

Директива ESEG визначає початок сегмента EEPROM. Вихідний файл може складатися із кількох сегментів EEPROM, які об'єднуються в один сегмент при компіляції. Сегмент EEPROM зазвичай складається лише із директив DB, DW і міток. Сегменти EEPROM мають свої власні побайтні лічильники стану. Директива ORG може бути використана для розташування змінних в необхідному місці EEPROM. Директива не має параметрів.

**Синтаксис:**

```
.ESEG
```

**Приклад:**

```
.DSEG                   ; Початок сегмента даних
var1:    .BYTE 1        ; зарезервувати 1 байт для var1
table:   .BYTE tab_size     ; зарезервувати tab_size байт.
.ESEG
eevar1:  .DW 0xffff         ; проініціалізувати 1 слово в EEPROM
```

**EXIT - Вийти із файлу**

Натрапивши на директиву EXIT компілятор завершує компіляцію даного файлу. Якщо директива використана у вкладеному файлі (див. директиву INCLUDE), то компіляція

продовжується зі строки, наступної після директиви INCLUDE. Якщо ж файл не є вкладеним, то компіляція припиняється.

**Синтаксис:**

```
.EXIT
```

**Приклад:**

```
.EXIT     ; Вийти із даного файлу
```

**INCLUDE - Вкласти інший файл**

Натрапивши на директиву INCLUDE компілятор відкриває вказаний в ній файл, компілює його, поки файл не закінчиться або не зустрінеться директива EXIT, після цього продовжує компіляцію початкового файлу зі строки, наступної після директиви INCLUDE. Вкладений файл може також містити директиви INCLUDE.

**Синтаксис:**

```
.INCLUDE "ім'я_файлу"
```

**Приклад:**

```
; файл iodefs.asm:
.EQU sreg      = 0x3f     ; Регістр статусу
.EQU sphigh = 0x3e  ; Старший байт покажчика стеку
.EQU splow     = 0x3d     ; Молодший байт покажчика стеку
; файл incdemo.asm
.INCLUDE iodefs.asm       ; Вкласти визначення портів
          in r0,sreg      ; Прочитати регістр статусу
```

**LIST - Включити генерацію лістингу**

Директива LIST вказує компілятору на необхідність створення лістингу. Лістинг представляє собою комбінацію асемблерного коду, адрес і кодів операцій. По дефолту генерація лістингу включена, однак дана директива використовується разом з директивою NOLIST для отримання лістингів окремих частих вихідних файлів.

**Синтаксис:**

```
.LIST
```

**Приклад:**

```
.NOLIST              ; Вимкнути генерацію лістингу
.INCLUDE "macro.inc"     ; Вкладені файли не будуть
.INCLUDE "const.def"     ; відображені в лістингу
.LIST                ; Включити генерацію лістингу
```

**LISTMAC - Включити розвертання макросів у лістингу**

Після директиви LISTMAC компілятор буде показувати в лістингу вміст макросу. За замовчуванням в лістингу показуються тільки виклик макросу і параметри, які передаються.

**Синтаксис:**

```
.LISTMAC
```

**Приклад:**

```
.MACRO MACX       ; Визначення макросу
     add r0,@0         ; Тіло макросу
     eor r1,@1
.ENDMACRO              ; Кінець визначення макросу
.LISTMAC             ; Включити розгортання макросів
      MACX r2,r1    ; Виклик макросу (в лістингу буде показане тіло
мак роса)
```

**MACRO - Початок мак роса**

З директиви MACRO починається визначення мак роса. В якості параметра директиві передається ім'я мак роса. При знайденні імені мак роса пізніше в тексті програми, компілятор замінює це ім'я на тіло мак роса. Макрос може мати до 10 параметрів, до яких в його тіло звертаються через @0-@9. При виклику параметри перераховуються через коми. Визначення мак роса закінчується директивою ENDMACRO.

По дефолту в лістинг включається тільки виклик мак роса, для розгортки мак роса необхідно використовувати директиву LISTMAC.

Макрос в лістингу показується знаком +.

**Синтаксис:**

```
.MACRO ім'я макросу
```

**Приклад:**

```
.MACRO SUBI16          ; Початок визначення макросу
     subi @1,low(@0)       ; Вирахувати молодший байт параметра 0 із
параметра 1
     sbci @2,high(@0)    ; Вирахувати старший байт параметра 0 із
параметра 2
.ENDMACRO                  ; Кінець визначення макросу
.CSEG                      ; Початок програмного сегменту
  SUBI16 0x1234,r16,r17    ; Вирахувати 0x1234 із r17:r16
```

**NOLIST - Вимкнути генерацію лістингу**

Директива NOLIST вказує компілятору на необхідність зупинки генерації лістингу. Лістинг представляє собою комбінацію асемблерного коду, адрес і кодів операцій. За замовчуванням генерація лістингу ввімкнена, однак може бути відключена даною директивою. Крім того, дана директива може бути використана разом з директивою LIST для отримання лістингів окремих частин вихідних файлів.

**Синтаксис:**

```
.NOLIST
```

**Приклад:**

```
.NOLIST              ; Вимкнути генерацію лістингу
.INCLUDE "macro.inc"     ; Вкладені файли не будуть
.INCLUDE "const.def"     ; відображені в лістингу
.LIST                ; Ввімкнути генерацію лістингу
```

**ORG - Встановити  положення в сегменті**

Директива ORG встановлює лічильник положення рівним заданій величині, яка передається як параметр. Для сегмента даних вона встановлює лічильник положення в SRAM (ОЗУ), для сегмента програм це програмний лічильник, а для сегмента EEPROM це положення в EEPROM. Якщо директиві передує мітка (в тому ж рядку), то мітка розміщується за адресою, вказаною в

параметрі директиви. Перед початком компіляції програмний лічильник і лічильник EEPROM рівні нулю, а лічильник ОЗП рівний 32 (оскільки адреси 0-31 зайняті регістрами). Зверніть увагу, що для ОЗП і EEPROM використовуються побайтні лічильники, для програмного сегмента - послівний.

**Синтаксис:**

```
.ORG вираз
```

**Приклад:**

```
.DSEG               ; Початок сегмента даних
.ORG 0x37               ; Встановити  адрес SRAM рівним 0x37
variable:      .BYTE 1   ; Зарезервувати байт за адресою 0x37H
.CSEG
.ORG 0x10               ; Встановити  програмний лічильник рівним
0x10
          mov r0,r1       ; Дана команда буде розміщена за адресою
0x10
```

**SET - Встановити  змінний символічний еквівалент виразу**

Директива SET присвоює імені деяке значення. Це ім'я пізніше може бути використане в виразах. Причому, на відміну від директиви EQU значення імені може бути змінене іншою директивою SET.

**Синтаксис:**

```
.SET ім'я = вираз
```

**Приклад:**

```
.SET io_offset = 0x23
.SET porta        = io_offset + 2
.CSEG               ; Початок кодового сегмента
    clr r2          ; Очистити регістр 2
    out porta,r2    ; Записати в порт A
```

## Вирази

Компілятор дозволяє використовувати в програмі вирази, які можуть складатися з операндів, операторів і функцій. Всі вирази являються 32-бітними.

**Операнди**

Можуть бути використані наступні операнди:

- Мітки, визначені користувачем (дають значення свого положення).
- Змінні, визначені директивою SET
- Константи, визначені директивою EQU
- Числа, задані в форматі:
  - Десятковому (прийнятий за замовчуванням): 10, 255
  - Шістнадцятковому (два варіанти запису): 0x0a, $0a, 0xff, $ff
  - Двійковому: 0b00001010, 0b11111111
  - Вісімковому (починаються з нуля): 010, 077
- PC - поточні значення програмного лічильника (Programm Counter)

**Оператори**

Компілятор підтримує ряд операторів, які перераховані в таблиці (чим вище положення в таблиці, тим вище пріоритет оператора). Вирази можуть братися в круглі дужки, такі вирази вираховуються перед виразами за дужками.

| Пріоритет | Символ | Опис |
|---|---|---|
| **14** | ! | Логічне заперечення |
| **14** | ~ | Побітове заперечення |
| **14** | - | Мінус |
| **13** | * | Множення |
| **13** | / | Ділення |
| **12** | + | Підсумовування |
| **12** | - | Віднімання |
| **11** | << | Зсув вліво |
| **11** | >> | Зсув вправо |
| **10** | < | Менше ніж |
| **10** | <= | Менше або дорівнює |
| **10** | > | Більше ніж |
| **10** | >= | Більше або дорівнює |
| **9** | == | Рівні |
| **9** | != | Не рівні |
| **8** | & | Побітове І |
| **7** | ^ | Побітове виключне АБО |
| **6** | \| | Побітове АБО |
| **5** | && | Логічне І |
| **4** | \|\| | Логічне АБО |

**Логічне заперечення**

**Символ:** !
**Опис:** Повертає 1 якщо вираз рівні 0, і навпаки
**Пріоритет:** 14
**Приклад:**
```
ldi r16, !0xf0      ; В r16 Завантажити 0x00
```

**Побітове заперечення**

**Символ:** ~
**Опис:** Повертає вираз в якому всі біти проінвертовані
**Пріоритет:** 14
**Приклад:** ldi r16, ~0xf0 ; В r16 Завантажити 0x0f

**Мінус**

**Символ:** -
**Опис:** Повертає арифметичне заперечення виразу
**Пріоритет:** 14
**Приклад:** ldi r16,-2 ; Завантажити -2(0xfe) в r16

**Множення**

**Символ:** *
**Опис:** Повертає результат множення двох виразів
**Пріоритет:** 13
**Приклад:** ldi r30, label*2

**Ділення**

**Символ:** /
**Опис:** Повертає цілу частину результату ділення лівого виразу на правий
**Пріоритет:** 13
**Приклад:** ldi r30, label/2

**Підсумовування**

**Символ:** +
**Опис:** Повертає суму двох виразів
**Пріоритет:** 12
**Приклад:** ldi r30, c1+c2

**Віднімання**

**Символ:** -
**Опис:** Повертає результат віднімання правого виразу із лівого
**Пріоритет:** 12
**Приклад:** ldi r17, c1-c2

**Зсув вліво**

**Символ:** <<
**Опис:** Повертає лівий вираз, зсунутий вліво на число біт, вказаних справа
**Пріоритет:** 11
**Приклад:** ldi r17, 1<<bitmask ; В r17 Завантажити 1 зсунуту вліво bitmask разів

**Зсув вправо**

**Символ:** >>
**Опис:** Повертає лівий вираз, зсунутий вправо на число біт, вказаних справа
**Пріоритет:** 11
**Приклад:** ldi r17, c1>>c2 ; В r17 Завантажити c1 зсунуте вправо c2 разів

**Менше ніж**

**Символ:** <
**Опис:** Повертає 1 якщо лівий вираз менший, ніж правий (враховується

знак), і 0 в протилежному випадку
**Пріоритет:** 10
**Приклад:** ori r18, bitmask*(c1<c2)+1

**Менше або рівні**

**Символ:** <=
**Опис:** Повертає 1 якщо лівий вираз менше ніж правий, або рівний(враховується знак), і 0 в протилежному випадку
**Пріоритет:** 10
**Приклад:** ori r18, bitmask*(c1<=c2)+1

**Більше ніж**

**Символ:** >
**Опис:** Повертає 1 якщо лівий вираз більший, ніж правий (враховується знак), і 0 в протилежному випадку
**Пріоритет:** 10
**Приклад:** ori r18, bitmask*(c1>c2)+1

**Більше або рівні**

**Символ:** >=
**Опис:** Повертає 1 якщо лівий вираз більший або дорівнює правому (враховується знак), і 0 в протилежному випадку
**Пріоритет:** 10
**Приклад:** ori r18, bitmask*(c1>=c2)+1

**Рівні**

**Символ:** ==
**Опис:** Повертає 1 якщо лівий вираз рівний правому (враховується знак), і 0 в протилежному випадку
**Пріоритет:** 9
**Приклад:** andi r19, bitmask*(c1==c2)+1

**Не рівні**

**Символ:** !=
**Опис:** Повертає 1 якщо лівий вираз не рівний правому (враховується знак), і 0 в протилежному випадку
**Пріоритет:** 9
**Приклад:** .SET flag = (c1!=c2)  ;Встановити  flag рівним 1 або 0

**Побітове И**

**Символ:** &
**Опис:** Повертає результат побітового і виразів
**Пріоритет:** 8
**Приклад:** ldi r18, High(c1&c2)

**Побітове виключне АБО**

**Символ:** ^
**Опис:** Повертає результат побітового виключного АБО виразів
**Пріоритет:** 7
**Приклад:** ldi r18, Low(c1^c2)

**Побітове АБО**

**Символ:** |
**Опис:** Повертає результат побітового АБО виразів

**Пріоритет:** 6
**Приклад:** ldi r18, Low(c1|c2)

**Логічне И**

**Символ:** &&
**Опис:** Повертає 1 якщо обидва вирази не рівні нулю, і 0 в протилежному випадку
**Пріоритет:** 5
**Приклад:** ldi r18, Low(c1&&c2)

**Логічне АБО**

**Символ:** ||
**Опис:** Повертає 1 якщо хоча би один із виразів не рівний нулю, і 0 в протилежному випадку
**Пріоритет:** 4
**Приклад:** ldi r18, Low(c1||c2)

**Функції**

Визначені наступні функції:

- LOW(вираз) Повертає молодший байт виразу
- HIGH(вираз) Повертає другий байт виразу
- BYTE2(вираз) те саме, що і функція HIGH
- BYTE3(вираз) Повертає третій байт виразу
- BYTE4(вираз) Повертає четвертий байт виразу
- LWRD(вираз) Повертає біти 0-15 виразу
- HWRD(вираз) Повертає біти 16-31 виразу
- PAGE(вираз) Повертає біти 16-21 виразу
- EXP2(вираз) Повертає 2 в степені (вираз)
- LOG2(вираз) Повертає цілу частину log2(вираз)

**Додаток 2**

***Схема програматора STK 200***

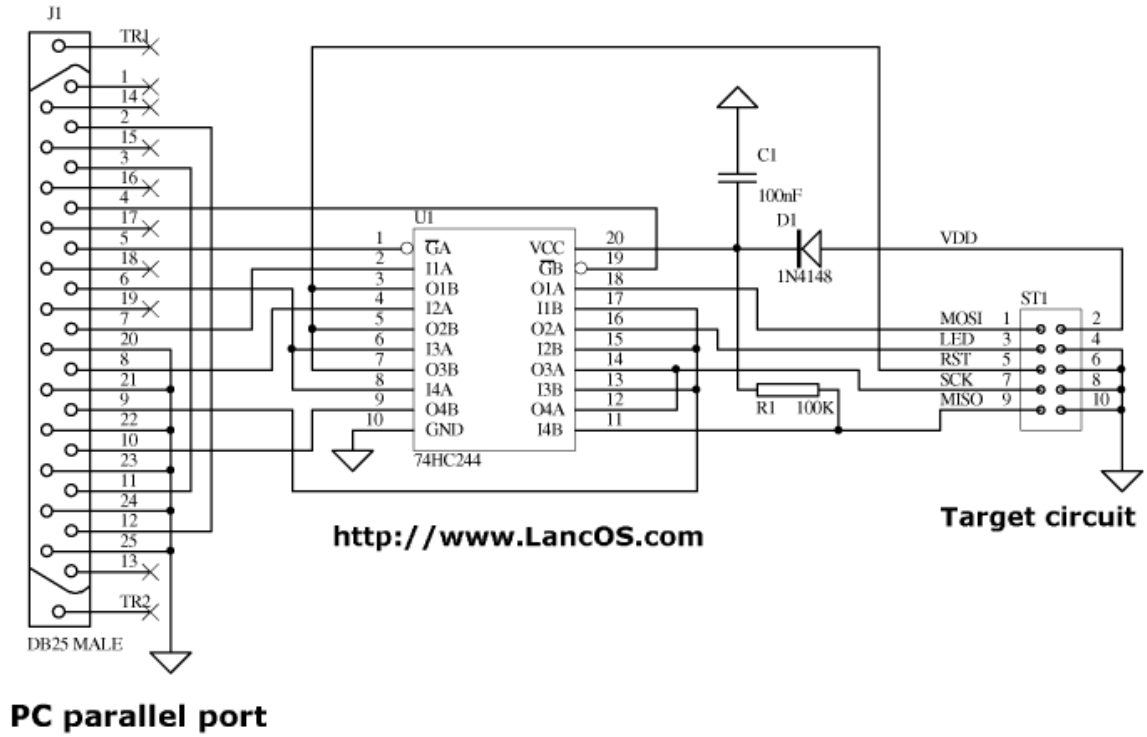
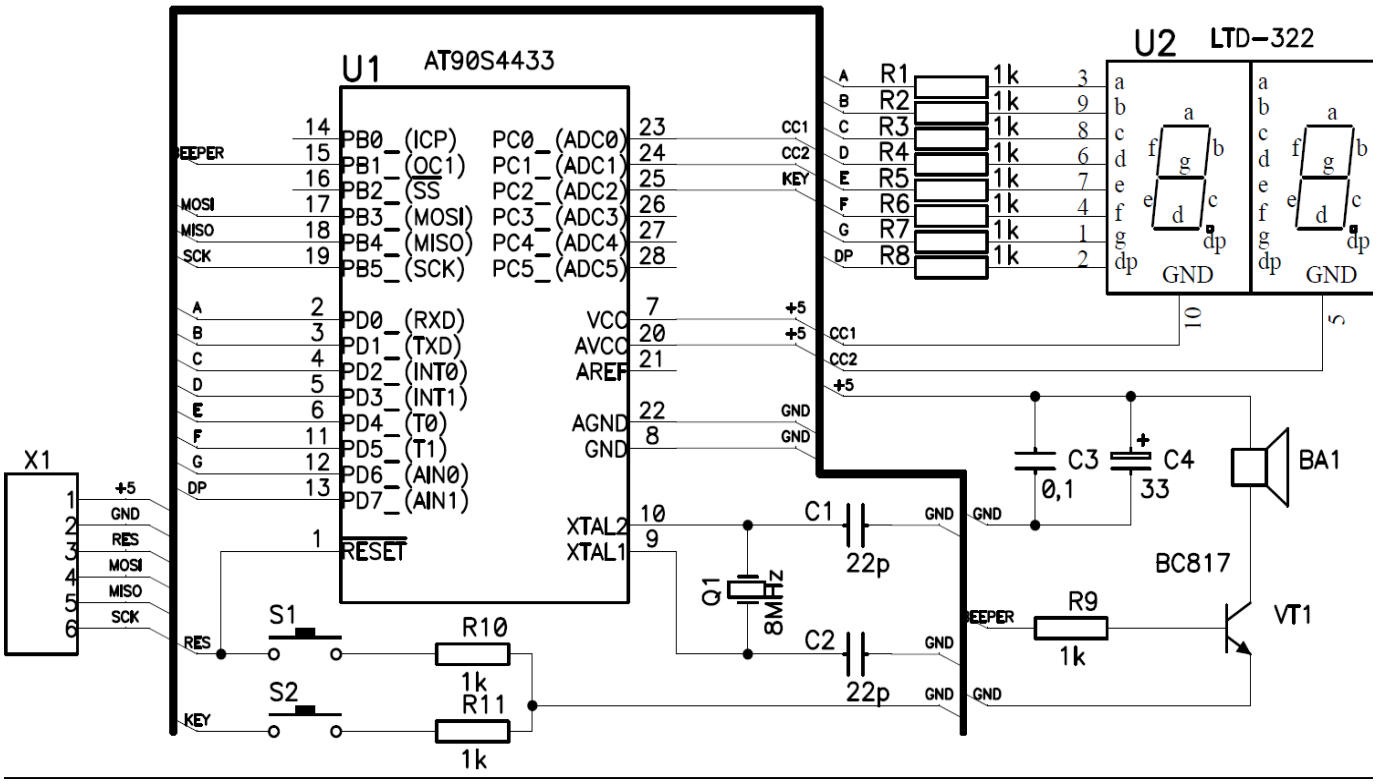



***Схема макета***



Додаток 3

***Приклад 1***

```
;***************************************************************
;
; Copyright DDF 2011 (C)
;
; Демонстрація керування портами
;
; 03.12.2011 - final version rev.1.0
;
;***************************************************************

.include "4433def.inc"

     ldi  r16,RAMEND          ; Ініціалізація
     out  sp,r16              ; покажчика стеку

     ldi  r16, 0xFF      ; Вмикаємо індикатор
     out  PORTD,r16      ;
     out  DDRD,r16       ;
     ldi  r16,(1<<PC0)|(1<<PC1)    ;
     out  DDRC,r16       ;

     ldi  r16,1<<PB1          ; Вмикаємо динамік
     out  DDRB,r16       ;

     ldi  r16,1<<PC2          ; Вмикаємо підтяжку
     out  PORTC,r16      ; для кнопки
beep:                         ; Зупиняємо цикл:
     sbi  PORTB,PB1      ; Вмикаємо динамік
     rcall     wait           ; Чекаємо 0,5мс
     cbi  PORTB,PB1      ; Вимикаємо динамік
     rcall     wait           ; Чекаємо 0,5мс
     sbic      PINC,PC2       ; Кнопка натиснута
     rjmp beep           ; ні - зациклити звук
     rjmp PC             ; так - вимикаємо звук

;--- Затримка 0,5мс
wait:
     ldi  r17,0xE8
     ldi  r18,0x03
loop:
     subi r17,1
     sbci r18,0
     brne      loop
     ret

.exit
```

***Приклад 2***

```
;***************************************************************
```

```
;
; Copyright DDF 2011 (C)
;
; Таймер зворотного відліку
;
; 03.12.2011 - final version rev.1.0
;
;*************************************************************

.equ _0 = 0x3F
.equ _1 = 0x06
.equ _2 = 0x5B
.equ _3 = 0x4F
.equ _4 = 0x66
.equ _5 = 0x6D
.equ _6 = 0x7D
.equ _7 = 0x07
.equ _8 = 0x7F
.equ _9 = 0x6F


.include "4433def.inc"

    ldi  r16,RAMEND          ; Ініціалізація
    out  sp,r16              ; покажчика стеку

    ldi  r16, _0             ; Вмикаємо "0"
    out  PORTD,r16      ;
    ldi  r16,0xFF       ; Порт D налаштовуємо
    out  DDRD,r16       ; на вивід -> LED

    ldi  r16,1<<PB1          ; Порт B -> PB1 налаштовуємо
    out  DDRB,r16       ; на вивід (BEEPER)

    ldi  r16,1<<PC2          ; Вмикаємо підтяжку на
    out  PORTC,r16      ; кнопку PORTC -> PC2

    ldi  r16,(1<<PC1)        ; Порт налаштовуємо
    out  DDRC,r16       ; на вивід PC1 -> OK LED

key:                    ; Основний цикл:
    sbic      PINC,PC2       ; Чекаємо натиску
    rjmp key            ; на кнопку

    ldi  r16, _9             ; Вмикаємо "9"
    out  PORTD,r16      ;
    rcall     beep           ; Видаємо звук
    ldi  r16, _8             ; Вмикаємо "8"
    out  PORTD,r16      ;
    rcall     beep           ; Видаємо звук
    ldi  r16, _7             ; Вмикаємо "7"
    out  PORTD,r16      ;
    rcall     beep           ; Видаємо звук
```

```
    ldi  r16, _6              ; Вмикаємо "6"
    out  PORTD,r16      ;
    rcall     beep            ; Видаємо звук
    ldi  r16, _5              ; Вмикаємо "5"
    out  PORTD,r16      ;
    rcall     beep            ; Видаємо звук
    ldi  r16, _4              ; Вмикаємо "4"
    out  PORTD,r16      ;
    rcall     beep            ; Видаємо звук
    ldi  r16, _3              ; Вмикаємо "3"
    out  PORTD,r16      ;
    rcall     beep            ; Видаємо звук
    ldi  r16, _2              ; Вмикаємо "2"
    out  PORTD,r16      ;
    rcall     beep            ; Видаємо звук
    ldi  r16, _1              ; Вмикаємо "1"
    out  PORTD,r16      ;
    rcall     beep            ; Видаємо звук
    ldi  r16, _0              ; Вмикаємо "0"
    out  PORTD,r16      ;
    rcall     beep            ; Видаємо звук
    rjmp key            ; Переходимо знову до запиту кнопки...

;--- Пікаємо 1кГц по 50мс з паузою 1с
beep:
    ldi  r19,0X32
beep_loop:
    sbi  PORTB,PB1
    rcall     wait
    cbi  PORTB,PB1
    rcall     wait
    dec  r19
    brne      beep_loop
    rcall     wait_1s
    ret

;--- Затримка 0,5мс
wait:
    ldi  r17,0xE8
    ldi  r18,0x03
loop:
    subi r17,1
    sbci r18,0
    brne      loop
    ret

;--- Підпрограма затримки на 1с
wait_1s:
    ldi  r17,0x00
    ldi  r18,0x6A
    ldi  r19,0x18
loop_1s:
    subi r17,1
```

```
    sbci r18,0
    sbci r19,0
    brne      loop_1s
    ret

.exit
```

***Приклад 3***

```
;***************************************************************
;
; Copyright DDF 2011 (C)
;
; Таймер зворотного відліку
; з динамічною індикацією
;
; 03.12.2011 - final version rev.1.0
;
;***************************************************************

.equ _0 = 0x3F
.equ _1 = 0x06
.equ _2 = 0x5B
.equ _3 = 0x4F
.equ _4 = 0x66
.equ _5 = 0x6D
.equ _6 = 0x7D
.equ _7 = 0x07
.equ _8 = 0x7F
.equ _9 = 0x6F

.def      DISPLAY = r22

.include "4433def.inc"

;--- Таблиця векторів переривань
.org 0x00
    rjmp RESET
.org 0x06
    rjmp TIM_INT

.org 0x0e
;--- Основна програма
RESET:
    ldi  r16,RAMEND          ; Ініціалізація
    out  sp,r16              ; покажчика стеку

    ldi  r16,0xFF       ; Порт D налаштовуємо
    out  DDRD,r16       ; на вивід -> LED

    ldi  r16,1<<PB1          ; Порт B -> PB1 налаштовуємо
    out  DDRB,r16       ; на вивід (BEEPER)
```

```
    ldi  r16,1<<PC2          ; Вмикаємо підтяжку на
    out  PORTC,r16      ; кнопку PORTC -> PC2

    ldi  r16,(1<<PC1)|(1<<PC0)    ; Порт налаштовуємо
    out  DDRC,r16       ; на вивід PC0 і PC1 -> OK LED

    ldi  r16,(1<<CS02)       ; Вмикаємо таймер 0
    out  TCCR0,r16      ; з перед подільником (прескалером) 1/256
    ldi  r16,(1<<TOIE0)      ; Дозволяємо переривання
    out  TIMSK,r16      ; від таймера 0

    ldi  DISPLAY,0      ; Ініціалізуємо індикатор
    sei                 ; Дозволяємо переривання

key:                    ; Основний цикл:
    sbic      PINC,PC2       ; Чекаємо натиску
    rjmp key            ; на кнопку

    ldi  DISPLAY,99          ; При натиску заносимо число 99 в
індикатор
COUNT:                       ; Цикл відліку:
    rcall     beep           ; Видаємо звуковий Сигнал
    dec  DISPLAY             ; Декрементуємо лічильник
    brne      COUNT               ; Якщо не "0" - зациклюємо
    rcall     beep           ;
    rjmp key            ; Переходимо знову до запиту кнопки...

;--- Пікаємо 1кГц по 50мс з паузою 1с
beep:
    ldi  r19,0X32
beep_loop:
    sbi  PORTB,PB1
    rcall     wait
    cbi  PORTB,PB1
    rcall     wait
    dec  r19
    brne      beep_loop
    rcall     wait_1s
    ret

;--- Затримка 0,5мс
wait:
    ldi  r17,0xE8
    ldi  r18,0x03
loop:
    subi r17,1
    sbci r18,0
    brne      loop
    ret

;--- Підпрограма затримки на 1с
wait_1s:
```

```
    ldi  r17,0x00
    ldi  r18,0x6A
    ldi  r19,0x18
loop_1s:
    subi r17,1
    sbci r18,0
    sbci r19,0
    brne     loop_1s
    ret

;--- Переривання від таймера 0
TIM_INT:
    push r0             ; Зберігаємо в стек:
    in   r0,SREG            ; Регістр статусу
    push r0             ; Регістр r0
    push r17            ; Регістр r17
    push r18            ; Регістр r18

    mov  r18,DISPLAY        ; Робимо розділення
    clr  r17            ; на одиниці і десятки:
DIG_BCD:
    subi r18,10             ; Віднімемо 10
    inc  r17            ; Інкрементуємо лічильник десятків
    brcc DIG_BCD            ; Повторюємо поки результат > 10
    subi r18,-10            ; Додаємо назад 10
    dec  r17            ; і декрементуємо лічильник десятків

    ldi  ZL,low(2*TABLE)          ; Загружаємо базову адресу
    ldi  ZH,high(2*TABLE)   ; таблиці дешифратора...
    clr  r0             ; Гасимо індикацію,
    out  PORTD,r0       ; щоби не було видно переключення
    sbic PORTC,PC1      ; Горить молодший розряд?
    rjmp NEXT_DIG       ; ні - переходимо на NEXT_DIG
    sbi  PORTC,PC1      ; так: гасимо його
    cbi  PORTC,PC0      ; і вмикаємо старший
    add  ZL,r17             ; Додаємо до базової адреси таблиці
    rjmp DECODER            ; цифру яку потрібно показати...
NEXT_DIG:               ; Другий розряд:
    sbi  PORTC,PC0      ; Гасимо старший розряд
    cbi  PORTC,PC1      ; і вмикаємо молодший
    add  ZL,r18             ; Додаємо до базової адреси таблиці
DECODER:                ; цифру яку потрібно показати...
    adc  ZH,r0              ; Дешифратор:
    lpm                 ; Загружаємо з таблиці цифру
    out  PORTD,r0       ; і виводимо її...
                        ; Видобуваємо зі стеку:
    pop  r18            ; Регістр r18
    pop  r17            ; Регістр r17
    pop  r0             ; Регістр r0
    out  SREG,r0            ; Регістр статусу
    pop  r0             ;
    reti                ; Виходимо із переривання
```

```
;--- Таблиця дешифратора
TABLE:
.db _0,_1,_2,_3,_4,_5,_6,_7,_8,_9

.exit
```

## Рекомендована література


1. Баранов В.Н.  Применение микроконтроллеров AVR: схемы, алгоритмы, программы Издательство: Додэка, 289 с., 2004
2. В.В. Гребнев  Микроконтроллеры семейства AVR фирмы Atmel, Издательство: РадиоСофт, 174 с., 2002
3. Джон Мортон  Микроконтроллеры AVR. Вводный курс Издательство: Издательский дом Додэка-XXI, 272 с.,  2006
4. Atmel Flash Microcontrollers. Product Portfolio Atmel corporation, 28 p., 2012
5. Kristian Saether, Ingar Fredriksen, Introducing a New Breed of Microcontrollers for 8/16-bit Applications http://www.atmel.com/images/doc7926.pdf

## ЗМІСТ