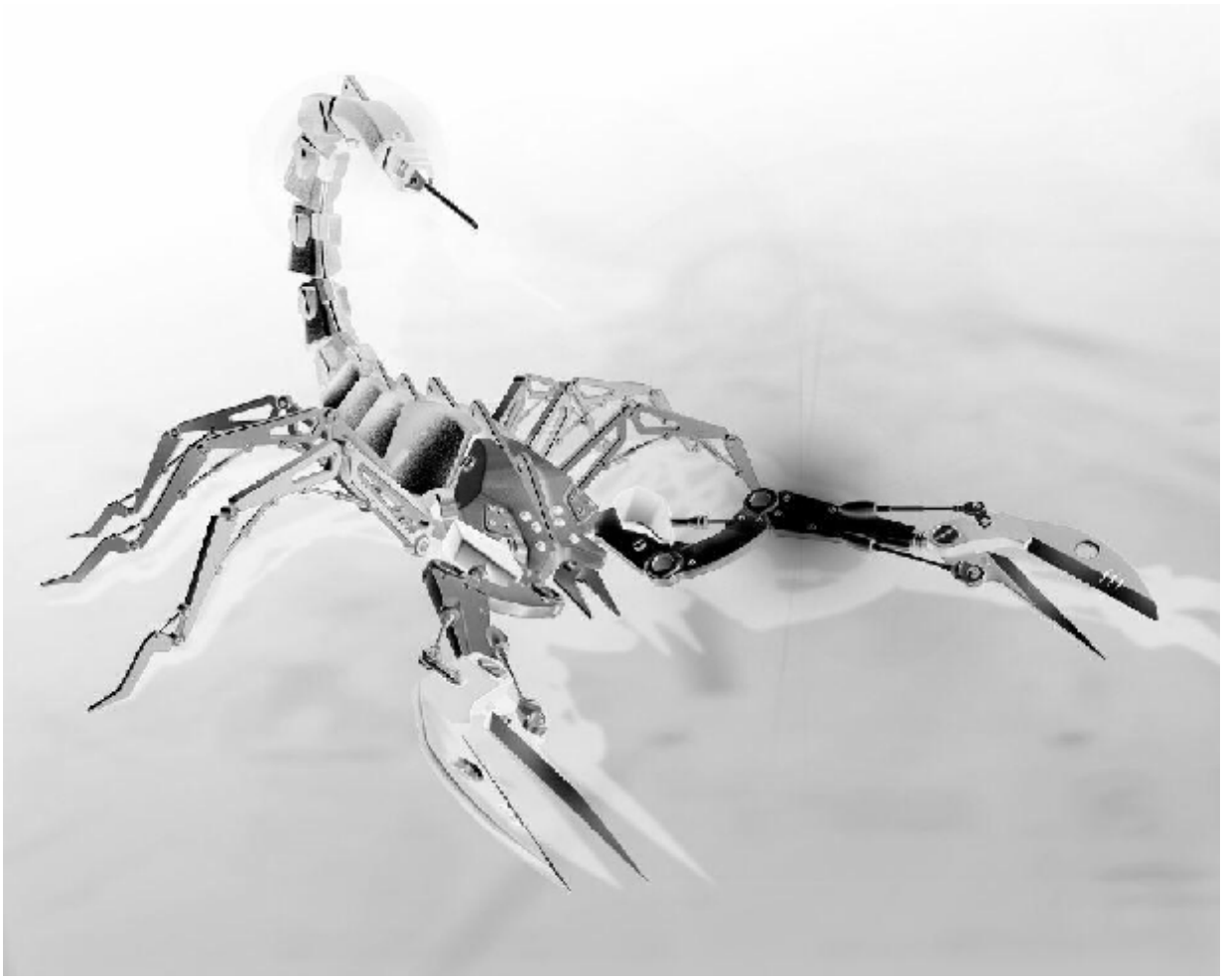
Ю.В. Дементьєв, В.К. Задорожний

Програмування AVR RISC мікроконтролерів



УДК 681.326
Д 30

Рецензенти:

В.А.Лужецький, доктор технічних наук, професор
С.М. Злепко, доктор технічних наук, професор
О.В. Грабчак, кандидат технічних наук, доцент

Рекомендовано до видання Вченою радою Вінницького національного технічного університету Міністерства освіти і науки України

Дементьєв Ю.В., Задорожний В.К.
Д 30 **Програмування AVR RISC мікроконтролерів**. Навчальний посібник. - Вінниця: ВНТУ, 2006. – 77 с.

В навчальному посібнику розглянуті основи організації, функціонування та програмування мікроконтролерів AVR, наведені характерні практичні задачі. Посібник розроблений у відповідності з планом кафедри та програмами дисциплін “Обчислювальні та мікропроцесорні засоби електронних апаратів ”, “Мікропроцесори в засобах зв'язку”, “Мікропроцесори в системах та пристроях”, “Проектування мікропроцесорних пристроїв”.

УДК 681.326

Міністерство освіти і науки України
Вінницький національний технічний університет

Ю.В. Дементьєв, В.К. Задорожний

Програмування AVR RISC мікроконтролерів

Затверджено Вченою радою Вінницького національного технічного університету як навчальний посібник для студентів напрямів підготовки 0924 – “Телекомунікації”, 0907 – “Радіотехніка”, 1601 – “Інформаційна безпека”. Протокол №12 від “29” червня 2006р.

Вінниця ВНТУ 2006

Зміст

Вступ 4
1 Архітектура мікроконтролера та фуціональне призначення його периферійних блоків 7
Завдання для самоперевірки 16
2 Система команд AVR-мікроконтролерів 16
2.1 Програмно доступні ресурси AVR-мікроконтролерів 16
2.2 Методи адресації інформації в командах 26
2.3 Система команд мікроконтролера 33
Завдання для самоперевірки 44
3 Вирази та директиви мови програмування асемблер 44
3.1 Вирази 44
3.2 Директиви 48
3.3 Створення та налагодження програм з використанням AVR Studio 56
Питання для самоперевірки 60
4 Компілятори мови програмування С та С++ 61
Питання для самоперевірки 64
5 Приклади програм роботи окремих блоків мікроконтролера 64
5.1 Модифікація окремих бітів комірок оперативної пам'яті чи змінних, які розміщуються в оперативній пам'яті 64
5.2 Реалізація програмних затримок при виконанні програми мікроконтролера 65
5.3 Реалізація програмного доступу до енергонезалежної пам’яті (EEPROM) 67
5.4 Реалізація програмного доступу до портів мікроконтролера 69
Питання для самоперевірки 69
6 Шляхи забезпечення надійності програмного забезпечення 70
Питання для самоперевірки 76

Література 76

ВСТУП

Впровадження цифрових методів обробки сигналів у радіотехнічні і телекомунікаційні системи забезпечило значне покращення характеристик апаратури. Системи керування та обробки сигналів на основі мікропроцесорів і мікроконтролерів швидко набули масового застосування, оскільки були сформульовані чіткі правила їхнього проектування. Так, наприклад, всі різновиди систем професійного зв'язку - радіорелейні, супутникові, оптоволоконні, а також абонентські - пейджингові, стільникові, високоякісне радіо і телевізійне мовлення використовують цифрові методи передачі інформації.

Як відомо, *мікропроцесор* - це програмно керований електронний пристрій, призначений для цифрової обробки інформації й керування процесом цієї обробки і реалізований у виді однієї чи декількох інтегральних мікросхем.

Мікропроцесорним пристроєм називається цифровий пристрій обробки інформації, що містить у своєму складі один чи кілька мікропроцесорів, модулі пам'яті, пристрої введення - виведення, блоки керування введенням - виведенням (контролери), зв'язані між собою внутрішньосистемною магістраллю, яка у загальному випадку складається із шин адреси, керування і даних.

Розвиток мікропроцесорних систем, з метою підвищення їх ефективності, йшов різними шляхами. Створення апаратної частини систем велося на основі магістрально-модульної структури, прикладні програми спочатку були нескладними і розроблялися відомими засобами. Як метод комплексного налагодження апаратури і програмного забезпечення фірмою Intel вже в середині 70-х років був запропонований метод внутрішньосхемної емуляції, який дозволяв уніфікувати проектування з використанням різних типів мікропроцесорів. При необхідності збільшити продуктивність системи, розробники йшли шляхом збільшення розрядності мікропроцесора - до 16-ти чи 32-двох розрядів, а також шляхом додаткового введення арифметичних співпроцесорів і різних типів контролерів, оскільки апаратна реалізація будь-якої функції забезпечувала її більш швидке виконання. Такі архітектурні рішення отримали назву CISC (Complex Instruction Set Computer) мікропроцесорів із комплексним набором інструкцій. До середини 90-х років збільшення розрядності означало, крім прямого результату, перехід в інший клас якості мікропроцесорних великих інтегральних схем (ВІС). Але це породжує ряд небажаних ефектів - збільшується довжина коду команди, число форматів команд, що у свою чергу ускладнює й сповільнює процес дешифрування коду команди і збільшує тривалість її обробки.

Помітним явищем наприкінці 80-х років стало переважне використання в нових системах керування восьми та шістнадцятирозрядних мікроконтролерів. До складу ВІС мікроконтролера входять всі складові мікро-ЕОМ: мікропроцесор, пам’ять програм і пам’ять даних, а також програмно керовані інтерфейсні схеми, що забезпечують зв’язок із зовнішніми компонентами системи.

В другій половині 90-х років ситуація істотно змінилася при проектуванні мікропроцесорних систем керування та обробки сигналів. У колишній магістрально-модульній структурі схеми обслуговування окремих вузлів і кінцевих пристроїв були драйверами без інтелекту. Вони виконували функції перетворення протоколів, служили підсилювачами потужності, але не мали можливості попередньо обробляти інформацію, що надходила. Функцію обробки виконував ведучий мікроконтролер, до якого від периферійних схем тягнулися джгути проводів. Наприклад, автомобілі на цьому етапі розвитку електроних систем керування містили до трьох миль проводів вагою більше 90 кг. Швидке здешевлення мікроконтролерів привело до доцільності заміни ними деяких периферійних схем. З'явилася можливість попередньої обробки локальних даних за допомогою так званих “інтелектуальних” сенсорів, з наступним пересиланням результатів ведучому мікроконтролеру, та й то лише в разі потреби. Це привело до широкого впровадження послідовних інтерфейсів, що існували раніше (RS-232, RS-485, SPI) та розробки нових (CAN, MicroLan). Топологія розвинутих систем керування отримала характер локальної мережі на основі послідовного інтерфейсу. В даний час використовуються послідовні канали як радіального, так і магістрального типів, синхронні й асинхронні.

Значне поліпшення характеристик власне мікроконтролерів, досягнуте в цей час, пов'язано з впровадженням RISC-архітектури, flash-пам'яті програм і EEPROM-пам'яті даних, прецизійних блоків АЦП і ЦАП сигналів. Гарвардська архітектура мікроконтролерів (з окремими блоками пам'яті програм і даних) дозволила оптимізувати формат команд і забезпечила вибір й виконання більшості з них за один машинний такт. Як відомо, основною ідеєю всіх RISC (Reduced Instruction Set Computer) мікропроцесорів є збільшення швидкодії за рахунок скорочення кількості операцій обміну з пам'яттю програм, а більшість операцій обробки даних реалізуються у форматах “регістр-регістр’’. Для цього кожну команду прагнуть вмістити в одну, максимум дві комірки пам'яті програм. При обмеженій розрядності комірки пам'яті це неминуче призводить до скорочення набору команд мікропроцесора. Використання технології flash-пам'яті забезпечило суттєве зниження вартості мікроконтролерів із пам'яттю програм, яку можна багаторазово перезаписувати. Це дозволило відмовитися від металокерамічних корпусів мікросхем із кварцовим склом, що були необхідні для пам'яті з ультрафіолетовим стиранням. Важливим аспектом підвищення надійності й мобільності систем стала також поява в структурі мікроконтролерів енергонезалежної пам'яті даних.

Необхідно відмітити, що передові структурні рішення в мікроелектроніці та збільшення ступеня інтеграції в даний час застосовуються в класі 8-розрядних мікроконтролерів, які призначені для реалізації нових функціональних модулів і структурних рішень.

Задачі керування в мікропроцесорних системах реалізуються як програмно, так і апаратно. Програмну частину відтворення функцій керування здійснює модуль центрального процесора, апаратну - спеціалізовані програмно керовані інтерфейсні модулі - послідовний порт, контролер прямого доступу до пам'яті, таймери/лічильники й інші, які виконують операції перетворення даних після одержання команд і даних від процесора, працюючи паралельно з ним та не гальмуючи виконання основної програми мікропроцесора.

Одним з представників 8-розрядних RISC-мікроконтролерів є сімейство мікроконтролерів AVR фірми Atmel. Сімейство мікроконтролерів з архітектурою AVR об'єднує 8–розрядні RISC-мікроконтролери загального призначення, які розроблені фірмою ATMEL, яка відома як світовий лідер у виробництві складних компонентів сучасної мікроелектроніки. Заснована в 1984 р. ATMEL Corp. (США) визначила такі сфери застосування своєї продукції: телекомунікації та мережі, обчислювальна техніка і вбудовані системи контролю та керування, побутова техніка та автомобілебудування. В даний час ATMEL випускає широкий спектр приладів енергонезалежної пам'яті високої швидкодії і мінімального енергоспоживання, мікроконтролери загального призначення та мікросхеми програмованої логіки.

Задум створення AVR та її назва зародилися в дослідницькому центрі ATMEL в Норвегії. Група фахівців (AV – від імен розробників Alf Bogen і Vergard Wollan, R – Risc) запропонували ряд ідей, які лягли в основу концепції AVR:

- використовувати швидку та економічну КМОП–технологію фірми ATMEL разом з перевагами RISC–архітектури для розробки і виробництва швидких 8-розрядних мікроконтролерів нового покоління, які за обчислювальною потужністю можна порівняти з 16–розрядними мікропроцесорами та мікроконтролерами та які переважають мікросхеми стандартної КМОП- логіки за швидкодією;

- розробити архітектуру та систему команд AVR відповідно до принципів мови С так, щоб апаратна частина мікроконтролера і його система команд були невід'ємними частинами одного цілого і використовувались з максимальною ефективністю;

- функціонально розширити можливості мікроконтролера опцією внутрішньосхемного програмування ISP (In System Programmable) шляхом об'єднання flash–технології фірми ATMEL зі стандартним швидким послідовним інтерфейсом SPI.

Ідея внутрішньої організації інформаційних шин нового мікроконтролера виникла в 1992р., через три роки були готові

експериментальні зразки. А перший промисловий варіант серійного мікроконтролера побачив світ в кінці 1996 р.

В результаті з'явилися мікроконтролери з низькою ціною, високою швидкодією, легкі в освоєнні та використанні, для яких на даний момент співвідношення ціна – швидкодія – енергоспоживання є одним з кращих на світовому ринку. Нова лінія мікроконтролерів розвивається дуже динамічно. Активно розширюються всі напрямки сімейства tiny (ATtiny13 має всього 6 ліній введення-виведення, 1 Кбайт па'мяті програм, 64 байти енергонезалежної пам'яті даних та 128 байтів оперативної пам'яті), classic (ATmega8515 має 35 ліній введення-виведення, 8 Кбайтів па'мяті програм, 512 байтів енергонезалежної пам'яті даних та 512 байтів оперативної пам'яті), mega (ATmega2561 має 54 лінії введення-виведення, 256 Кбайтів пам'яті програм, 8 Кбайтів енергонезалежної пам'яті даних та 4 Кбайти оперативної пам'яті).

В останній час намітилась тенденція створення спеціалізованих AVR мікроконтролерів: для керування електродвигунами (AT90PWM3), для роботи з рідкокристалічними індикаторами та в застосуваннях із зниженим енергоспоживанням.

Області застосування сімейства AVR багатогранні. Для ATtiny це інтелектуальні автомобільні сенсори різного призначення, іграшки, ігрові приставки, контролери захисту доступу в мобільних телефонах, зарядні пристрої, детектори диму і вогню, побутова техніка, різноманітні інфрачервоні пульти дистанційного керування. Classic використовуються в модемах різних типів, сучасних зарядних пристроях, супутникових навігаційних системах для визначення місця розташування автомобілів на дорозі, складної побутової техніки, пультах дистанційного керування, в промислових системах контролю і керування. Mega AVR призначені для принтерів, контролерів апаратів факсимільного зв'язку і ксероксів, контролерів сучасних дискових накопичувачів і CD-ROM, вимірювальних приладів обліку енергоресурсів.

Добре відомо, що розвинені засоби підтримки розробок при освоєнні і знайомстві з будь-яким новим сімейством мікроконтролерів мають не менше значення, ніж самі кристали. Фірма ATMEL потурбувалася і про це. Програмні та апаратні засоби для AVR розроблялися паралельно з самими мікроконтролерами і включають в себе компілятори, налагоджувачі, програматори, найпростіші налагоджувальні плати – конструктори.

1 АРХІТЕКТУРА МІКРОКОНТРОЛЕРА ТА ФУНКЦІОНАЛЬНЕ ПРИЗНАЧЕННЯ ЙОГО ПЕРИФЕРІЙНИХ БЛОКІВ

Під архітектурою мікропроцесорних пристроїв розуміють сукупність таких характеристик і параметрів:

- розрядність інформаційних шин адреси та даних;

- склад та призначення програмно доступних регістрів і блоків;
- система команд;
- режими адресації пам'яті;
- структура адресного простору;
- способи адресації периферійних пристроїв та засоби виконання операцій введення-виведення;
- класи переривань, особливості ініціювання й обробки переривання.

Розглянемо структурну схему 8-розрядного RISC мікроконтролера, що має швидкий процесор з Гарвардською архітектурою, пам'ять програм, пам'ять даних, порти введення-виведення та набір периферійних блоків. Структурна схема мікроконтролера наведена на рисунку 1.1. Гарвардська архітектура AVR реалізує повний логічний та фізичний поділ не тільки адресних просторів, але й інформаційних шин для окремого звертання до пам'яті програм і до пам'яті даних, причому способи адресації й доступу до цих масивів пам'яті також різні.

Подібна будова близька до структури цифрових сигнальних процесорів і забезпечує істотне підвищення продуктивності. Центральний процесор працює одночасно як із пам'яттю програм, так і з пам'яттю даних, розрядність шини пам'яті програм розширена до 16 бітів.

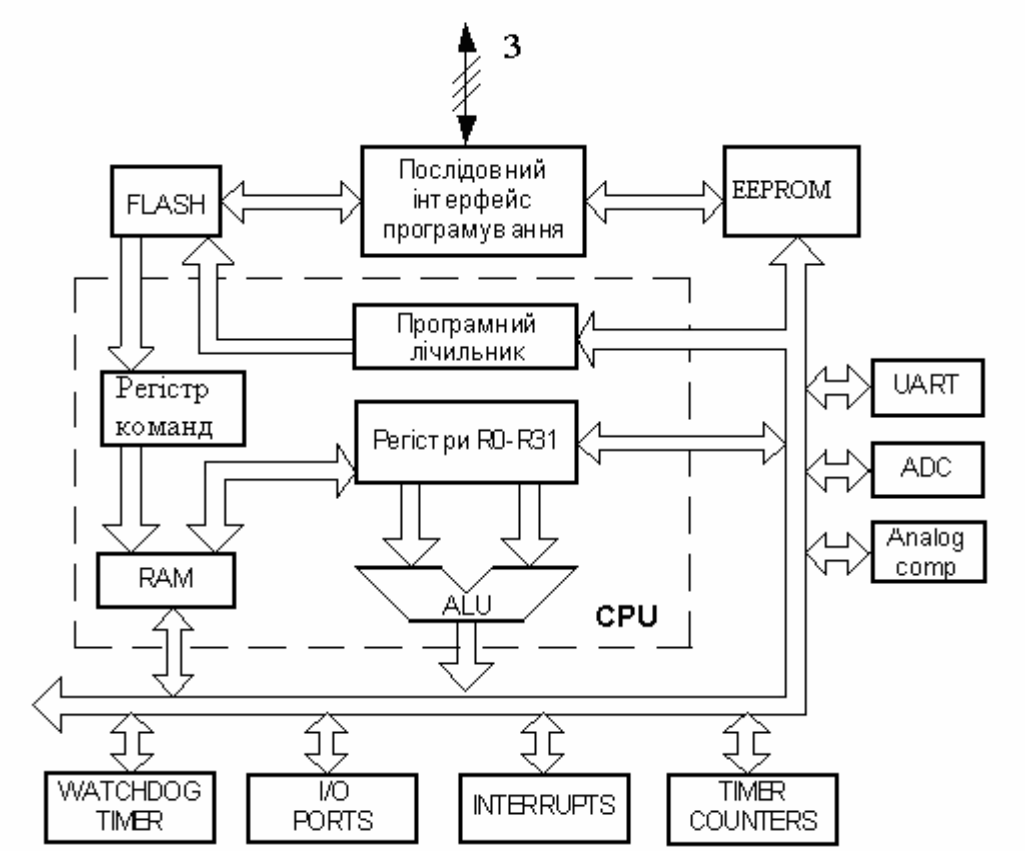


Рисунок 1.1 – Структурна схема AVR-мікроконтролера

Усі AVR-мікроконтролери мають flash-пам'ять програм, яка може бути завантажена за допомогою послідовного SPI-інтерфейсу. Крім програми у flash-пам'ять можуть бути записані постійні дані, що не

змінюються при функціонуванні мікропроцесорної системи. Це константи, таблиці знакогенераторів, таблиці лінеаризації датчиків і т.п. Перевагою технології flash є можливість оперативної зміни вмісту пам'яті програм, а недоліком – відсутність можливості стирати окремі комірки. Тому завжди виконується повне очищення всієї пам'яті програм. При цьому для AVR гарантується, як мінімум, 1000 циклів перезаписування комірок flash-пам'яті.

Однак останні версії кристалів сімейства AVR "mega" мають можливість змінювати вміст пам'яті програм під час роботи мікроконтролера. Таким чином, нові мікроконтролери AVR можуть змінювати алгоритми свого функціонування та програми, закладені в них, і далі працювати вже за зміненим алгоритмом чи новою програмою.

Наприклад, можна зберігати кілька робочих версій програми у зовнішній енергонезалежній пам'яті, а потім, залежно від зовнішніх чи внутрішніх логічних умов, програмно перезавантажувати робочі програми в той самий мікроконтролер AVR, не виймаючи його з друкованої плати. Для цього весь масив пам'яті програм поділяється на дві нерівні за обсягом області: невеликий програмний блок завантажувача (програма, що керує перезаписом flash-пам'яті програм) та блок для розміщення програмного коду. Програма-завантажувач створюється самим розробником і повинна бути запрограмована попередньо.

Усі мікроконтролери AVR також мають блок енергонезалежної пам'яті даних EEPROM, що електрично стирається. Цей тип пам'яті, доступний програмі мікроконтролера безпосередньо в ході її виконання, зручний для збереження проміжних даних, різних констант, таблиць перекодувань, каліброваних коефіцієнтів і т.п. EEPROM також може бути завантажена іззовні як через SPI інтерфейс, так і за допомогою звичайного програматора. Число циклів перезапису - не менше 100000. Внутрішня оперативна пам'ять SRAM присутня у всіх AVR сімейств "classic" і "mega" та в більшості кристалів сімейства "tiny". Для деяких мікроконтролерів передбачена можливість організації зовнішньої шини адреси та шини даних для під’єднання паралельної пам'яті даних обсягом до 64 Кбайтів.

Поділ шин доступу (див. рисунок 1.1) до flash-пам'яті і SRAM пам'яті дає можливість мати шини даних для пам'яті даних і пам'яті програм різної розрядності, а також використовувати технологію конвеєризації. Конвеєризація полягає в тому, що під час виконання поточної команди програмний код наступної вже вибирається з пам'яті та дешифрується.

Наступна відмінна риса архітектури мікроконтролерів AVR - регістровий файл швидкого доступу. Кожний з 32-х регістрів загального призначення R0-R31 розміром 1 байт безпосередньо зв'язаний з арифметико-логічним пристроєм (ALU) процесора, тобто у AVR існує 32 регістра - акумулятора для збереження результату арифметико-логічних операцій, що значно підвищує його швидкодію. Ця обставина дозволяє з наявністю конвеєрної обробки команд виконувати одну операцію в ALU за один машинний цикл. Так, два операнди вибираються з регістрового

файлу, виконується команда, а результат операції записується назад у регістровий файл протягом тільки одного циклу.

Шість із 32-х регістрів файлу можуть використовуватися як три 16-розрядних вказівники адреси при опосередкованій адресації даних. Один із цих вказівників Z застосовується також при доступі до даних, записаних у пам'яті програм мікроконтролера. Використання трьох 16-бітних вказівників істотно підвищує швидкість пересилання даних при роботі прикладної програми.

Регістровий файл займає молодші 32 байти в загальному адресному просторі SRAM AVR мікроконтролера. Таке архітектурне рішення дозволяє одержувати доступ до швидкої "регістрової" оперативної пам'яті мікроконтролера двома шляхами - безпосередньою адресацією в коді команди до будь-якого регістру, а також іншими доступними способами адресації середовища SRAM. У технічній документації фірми Atmel ця корисна властивість називається "швидке контекстне перемикання" і є ще однією відмінною рисою архітектури AVR, що підвищує ефективність роботи мікроконтролера і його продуктивність. Особливо помітна така особливість при реалізації процедур цілочисельної 16-бітної арифметики, коли виключаються багаторазові пересилки між різними комірками пам'яті даних при обробці арифметичних операндів в ALU.

Наступним кроком на шляху збільшення швидкодії AVR є використання технології конвеєризації, внаслідок чого цикл "вибір - виконання" команди помітно скорочений. Наприклад, у мікроконтролерів сімейства MCS-51 коротка команда виконується за 12 тактів генератора (один машинний цикл), протягом яких процесор послідовно зчитує код операції і виконує її. У PIC - контролерах фірми Microchip, де вже реалізований конвеєр, коротка команда виконується протягом 8 періодів тактової частоти (2 машинних цикли). За цей час послідовно дешифрується і зчитується код операції, виконується команда, фіксується результат і одночасно зчитується код наступної операції (однорівневий конвеєр). Тому в загальному потоці команд одна коротка команда реалізується за 4 періоди тактової частоти або за один машинний цикл.

У мікроконтролерах AVR теж використовується однорівневий конвеєр при звертанні до пам'яті програм і коротка команда в загальному потоці виконується, як і в PIC-контролерах, за один машинний цикл. Головна ж відмінність полягає в тому, що цей цикл у AVR складає усього один період тактової частоти. Для порівняння, на рисунку 1.2 наведено часові діаграми виконання типової команди для різних мікроконтролерних платформ.

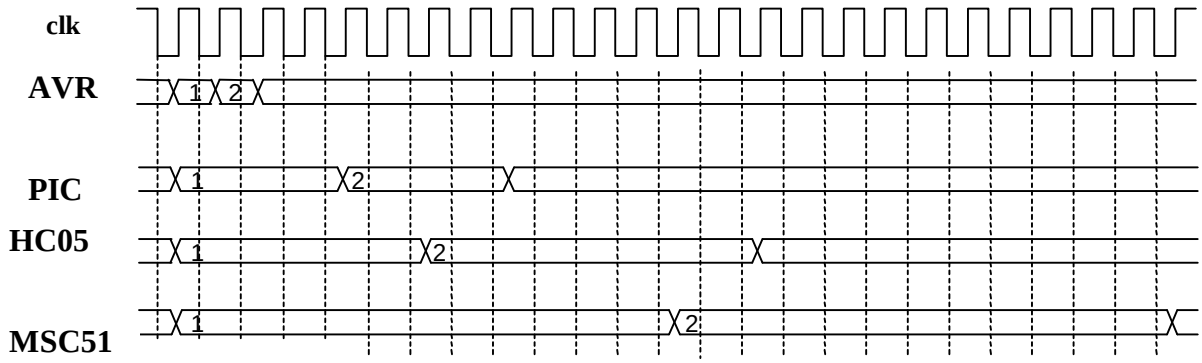


Рисунок 1.2 – Часові діаграми виконання типової команди деяких мікроконтролерних платформ

Внутрішній тактовий генератор AVR може запускатися від кількох джерел опорної частоти (зовнішній генератор, зовнішній кварцовий резонатор, внутрішній чи зовнішній RC-ланцюжок). Оскільки AVR-мікроконтролери цілком статичні, мінімальна частота тактового генератора знизу не обмежена (аж до покрокового режиму). Максимальна робоча частота визначається конкретним типом мікроконтролера. Цікаву апаратну особливість має мікроконтролер ATtiny15L. Він містить блок PLL для апаратного збільшення основної тактової частоти в 16 разів. При номінальному значенні останньої 1,6 мГц можна отримати частоту мікроконтролера 25,6 мГц, яку можна використати як джерело для таймера/лічильника мікроконтролера, значно підвищуючи роздільну здатність його роботи.

Сторожовий таймер (WatchDog Timer) призначений для захисту мікроконтролера від збоїв у процесі роботи. Він має свій власний RC-генератор, що працює на частоті 1 мГц. Ця частота є наближеною і залежить, насамперед, від величини напруги живлення мікроконтролера і від температури. Сторожовий таймер має у своєму складі власний попередній подільник вхідної частоти з програмованим коефіцієнтом ділення, що дозволяє встановлювати період переповнення таймера та перезапуску мікроконтролера. Роботу сторожового таймера можна дозволити чи заборонити програмним шляхом.

Мікроконтролери AVR мають у своєму складі від 1 до 4 таймерів/лічильників загального призначення розрядністю 8 або 16 біт, що можуть працювати як таймери від внутрішнього джерела опорної частоти і як лічильники зовнішніх подій із зовнішнім тактуванням. Загальні риси всіх таймерів/лічильників такі:

- наявність програмно-керованого попереднього подільника вхідної частоти з різними коефіцієнтами ділення. Відмінною рисою є можливість роботи таймерів/лічильників на основній тактовій частоті мікроконтролера без попереднього її зниження, що дає можливість формувати на виході таймерів/лічильників сигнали з малим значенням періоду;

- незалежне від режиму роботи процесорного ядра мікроконтролера функціонування (вміст таймера/лічильника можна як зчитати, так і завантажити його новим значенням у будь-який час);

- верхній частотний поріг сигналу лічильника визначається як половина основної тактової частоти мікроконтролера. Вибір зрізу або фронту зовнішнього сигналу програмується користувачем;

- наявність різних векторів переривань для декількох різних подій (переповнення, захоплення, порівняння).

Годинник реального часу (RTC) реалізовано у всіх мікроконтролерах сімейства "mega" та у деяких кристалах сімейства "classic". Таймер/лічильник RTC має свій власний попередній дільник частоти, який можна програмним шляхом підключити до основного внутрішнього джерела тактової частоти мікроконтролера чи до додаткового асинхронного джерела опорної частоти (кварцовий резонатор чи зовнішній синхросигнал). Для цієї мети зарезервовані два зовнішніх виводи мікроконтролера. Внутрішній осцилятор таймера/лічильника RTC оптимізовано для роботи із зовнішнім кварцовим резонатором 32,768 кГц, що дає змогу при коефіцієнті ділення частоти сигналу осцилятора 1/32768 отримати сигнал з періодом 1 секунда для підтримки роботи астрономічного годинника.

Порти введення-виведення AVR мають залежно від моделі мікроконтролера від 3 до 53 незалежних ліній "вхід/вихід". Кожен розряд порту може бути окремо запрограмований на введення чи на виведення інформації. Потужні вихідні драйвери забезпечують навантажувальну здатність за струмом до 20 мА на лінію порту (вхідний струм) при максимальному значенні 40 мА, що дозволяє, наприклад, безпосередньо підключати до мікроконтролера світлодіоди та біполярні транзистори. Загальне навантаження за струмом на всі лінії одного порту не повинні перевищувати 80 мА (усі значення наведені для напруги живлення 5 вольт).

Архітектурна особливість побудови портів введення-виведення в AVR полягає в тому, що для кожного фізичного виводу існує три біти контролю/керування, а не два, як у поширених 8-розрядних мікроконтролерах (Intel, Microchip, Motorola та ряду інших). Біти контролю/керування лініями портів введення-виведення розташовані в регістрах DDRx, PINx, PORTx. Спрощена структурна схема одного розряду порту введення-виведення AVR мікроконтролера наведена на рисунку 1.3. Для програмного керування бітом порту використовуються відповідні біти в керуючих регістрах DDRx - біт контролю напрямку передачі даних та можливого під’єднання виводу до шини живлення (VCC), PORTx - біт можливого під’єднання виводу до напруги Vcc та інформаційний біт на лінії порту, PINx - біт для відображення логічного рівня сигналу на фізичному виводі мікросхеми.

Саме три біта керуючих регістрів використовується для керування лінією порту введення-виведення, оскільки використання з цією метою

лише двох бітів породжує ряд проблем при операціях типу "читання-модифікація-запис". Наприклад, якщо мають місце дві послідовні операції "читання-модифікація-запис", то перший результат може бути загублений безповоротно, якщо лінія виведення порту працює на ємнісне навантаження і тому потрібен якийсь час для стабілізації рівня сигналу на зовнішньому виводі мікросхеми.

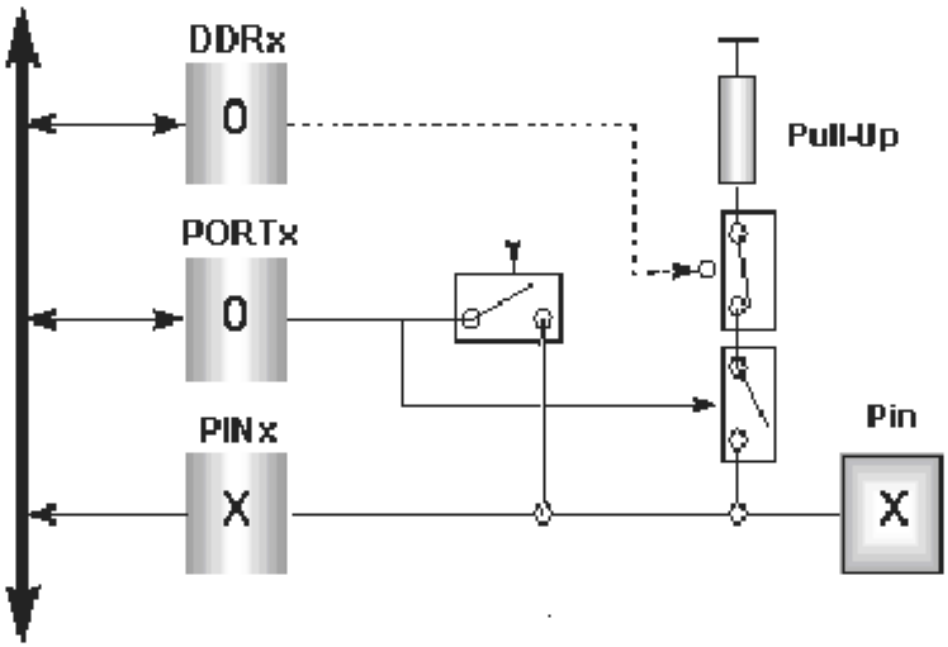


Рисунок 1.3 – Структурна схема одного розряду порту введення-виведення

Архітектура побудови портів введення-виведення AVR із трьома бітами контролю/керування дозволяє повністю контролювати процес введення-виведення даних. Якщо необхідно одержати реальне значення сигналу на фізичному виводі мікроконтролера – необхідно просто читати вміст лінії порту за адресою PINx. Якщо потрібно оновити вміст ліній виведення, то спочатку необхідно прочитати PORTx защіпку і потім модифікувати дані. Це дозволяє уникнути необхідності мати копію вмісту порту в пам'яті для безпеки і підвищує швидкість роботи мікроконтролера при роботі із зовнішніми пристроями.

Залежність режиму роботи ліній портів введення-виведення від вмісту керуючих регістрів наведена в таблиці 1.1

Таблиця 1.1

DDRXn	PORTXn	I/O	Pull-up	Примітка
0	0	Введення	Ні	Високоімпедансний (Z-стан) лінії PORTXn
0	1	Введення	Так	Лінія PORTXn - джерело струму
1	0	Виведення	Ні	На лінії PORTXn логічний “0”
1	1	Виведення	Ні	На лінії PORTXn логічна “1”

Аналоговий компаратор (АС) входить до складу більшості AVR-мікроконтролерів. Типовий рівень напруги зсуву нульового рівня дорівнює 10 мВ, час затримки поширення сигналу складає 500 нс і залежить від напруги живлення мікроконтролера. Так, наприклад, при напрузі живлення 2,7 вольт він дорівнює 750 нс. Аналоговий компаратор має свій власний вектор переривання в загальній системі переривань мікроконтролера. При цьому тип перепаду, що викликає запит на переривання при спрацьовуванні компаратора, може бути запрограмований користувачем як фронт чи зріз сигналу. Логічний вихід компаратора може бути програмним шляхом підключений до входу одного з 16-розрядних таймерів/лічильників, який працює в режимі захоплення. Це дає можливість вимірювати тривалість аналогових сигналів, а також максимально просто реалізувати аналого-цифровий перетворювач (АЦП) двотактного інтегрування.

Аналого-цифровий перетворювач – ADC, побудований за класичною схемою послідовних наближень із пристроєм вибірки/збереження (ПВЗ) на вході. Кожний з аналогових входів може бути з'єднаний із входом ПВЗ через аналоговий мультиплексор. Пристрій вибірки/збереження має свій власний підсилювач, що гарантує стабільне значення вимірюваного аналогового сигналу протягом усього часу перетворення. Розрядність АЦП складає 10 бітів при нормованій похибці ±2 молодших розряди. АЦП може працювати у двох режимах - однократне перетворення по будь-якому обраному каналу і послідовне циклічне опитування всіх каналів. Час перетворення вибирається програмно за допомогою вибору коефіцієнта ділення частоти спеціального дільника, що входить до складу блоку АЦП. Він складає 70...280 мкс для ATmega128 та 65...260 мкс для всіх інших мікроконтролерів, що мають у своєму складі АЦП. Важливою особливістю аналого-цифрового перетворювача є функція заглушення шуму при перетворенні. Користувач має можливість, виконавши короткий ряд програмних операцій, запустити АЦП у той час, коли центральний процесор знаходиться в одному з режимів зниженого енергоспоживання.

При цьому на точність перетворення не будуть впливати завади, що виникають при роботі процесорного ядра.

Мікроконтролери AVR можуть бути переведені, програмним шляхом, в один із шести режимів зниженого енергоспоживання. Для різних сімейств AVR і різних типів мікроконтролерів у межах кожного сімейства змінюються кількість і сполучення доступних режимів зниженого енергоспоживання:

- режим холостого ходу (IDLE), у якому припиняє роботу тільки процесор і фіксується вміст пам'яті даних, а внутрішній генератор синхросигналів, таймери, система переривань і WATCHDOG-таймер продовжують функціонувати;

- режим мікроспоживання (Power Down), у якому зберігається вміст регістрового файлу, але зупиняється внутрішній генератор синхросигналів. Вихід із режиму Power Down можливий або за загальним скиданням мікроконтролера, або за сигналом від зовнішнього джерела переривання. При включеному WATCHDOG-таймері струм споживання в цьому режимі складає близько 60...80 мкА, а при виключеному - менше 1 мкА для всіх типів AVR. Вище наведено значення, коректні для значення напруги живлення 5 В.

Режим збереження енергії (Power Save), реалізований тільки в тих AVR, що мають у своєму складі годинник реального часу. В основному режим Power Save ідентичний Power Down, але ньому допускається незалежна робота додаткового таймера/лічильника системи реального часу (RTC). Вихід із режиму Power Save можливий за перериванням, викликаним чи переповненням таймера/лічильника RTC, чи спрацьовуванням блоку порівняння цього лічильника. Струм споживання в цьому режимі складає 6...10 мкА при напрузі живлення 5 В на частоті 32,768 кГц.

Режим заглушення шуму при роботі аналого-цифрового перетворювача - Noise Reduction. Як уже відзначалося, у цьому режимі зупиняється процесорне ядро, але дозволена робота ЦАП, двопровідного інтерфейсу I2C і сторожового таймера.

Основний режим чекання (Standby) ідентичний режиму Power Down, але тут робота тактового генератора не припиняється. Це гарантує швидкий вихід мікроконтролера з режиму чекання, усього за 6 тактів генератора.

Додатковий режим чекання (Extended Standby) ідентичний режиму Power Save, але тут робота тактового генератора теж не припиняється. Це гарантує швидкий вихід мікроконтролера з режиму чекання усього за 6 тактів генератора.

Корпорація Atmel планує подальший розвиток перспективної лінії AVR-мікроконтролерів. Найбільш цікаві рішення реалізовані в сімействі "mega", де розпочатий серійний випуск цілого ряду кристалів за технологією 0,35 мкм. Обсяг Flash-пам'яті програм з функціями ISP(можливість програмування мікроконтролера в системі при основній

напрузі живлення) і SPM (store program memory) - функція самопрограмування пам'яті мікроконтролера в системі без участі зовнішнього програматора у нових "mega" становить від 8 до 128 кілобайтів. Усі нові мікроконтролери сімейства "mega" мають JTAG - інтерфейс (за винятком mega 8), апаратний помножувач 8×8, що дає 16-розрядний результат, схему захисту від збоїв, двопровідний послідовний інтерфейс, аналого-цифровий перетворювач і ряд інших апаратних особливостей. Крім цього, удвічі підвищена швидкість роботи всіх периферійних вузлів (SPI, PWM, UART і інших), поліпшена робота схеми тактування, а також спрощений доступ до зовнішньої пам'яті даних.

Завдання для самоперевірки

1. Вкажіть особливості, які притаманні AVR RISC архітектурі.
2. Назвіть блоки, які входять до складу AVR мікроконтролера.
3. Як визначити час виконання команди в AVR мікроконтролері?
4. Назвіть особливості портів введення-виведення AVR мікроконтролера та вкажіть перелік регістрів, які забезпечують до них програмний доступ?
5. Наведіть структурну схему та вкажіть призначення блоку сторожового таймера в AVR мікроконтролері.
6. Наведіть сруктуру можливої програми AVR мікроконтролера, яка використовує блок сторожового таймера.

2 СИСТЕМА КОМАНД AVR-МІКРОКОНТРОЛЕРІВ

2.1 Програмно доступні ресурси мікроконтролерів AVR

При створенні програми для мікроконтролера мовою Асемблер розробник оперує лише програмно доступними ресурсами мікропроцесорної системи. У мікроконтролера ці ресурси складають: програмно доступні регістри загального призначення, внутрішня оперативна пам'ять, зовнішня оперативна пам'ять даних (для мікроконтролерів, які мають можливість формувати зовнішню паралельну системну шину), пам'ять регістрів введення-виведення, енергонезалежна пам'ять даних, пам'ять програм.

На рисунку 2.1 наведена загальна програмна модель мікроконтролерів AVR, яка є зображенням його програмно-доступних ресурсів. Центральним блоком в такій моделі є регістровий файл, розміром тридцять два восьмирозрядних регістри (R0 - R31). Особливістю регістрового файлу є відсутність спеціального регістра-акумулятора для збереження результату при арифметико-логічних операціях. В мікроконтролерах AVR регістри R0 - R31 безпосередньо доступні при арифметико-логічних операціях на ALU.

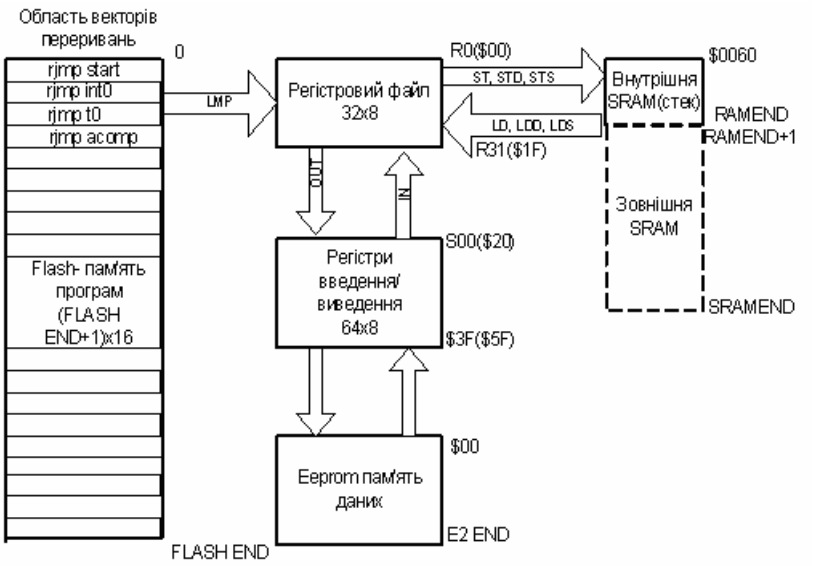


Рисунок 2.1 – Програмна модель AVR мікроконтролера

Старші регістри (рисунок 2.2) об'єднані у пари й утворюють три адресні шістнадцятирозрядні регістри-вказівники, призначені для опосередкованої (непрямої) адресації комірок пам'яті (мікроконтролери AVR без SRAM мають тільки один 16-бітний регістр Z).

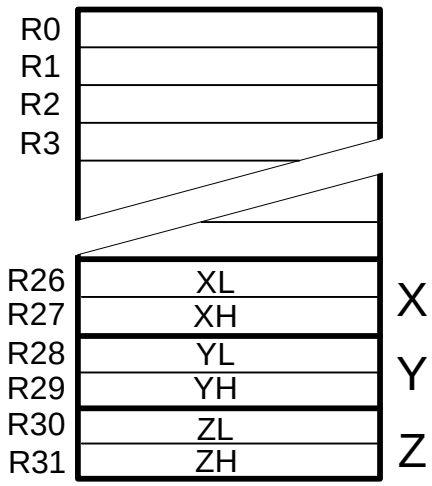


Рисунок 2.2 – Регістровий файл мікроконтролерів AVR

Наступні 64 адреси за регістрами загального призначення займають регістри вводу/виводу. У цій області згруповані всі регістри даних, керування й статусу внутрішніх програмно керованих периферійних блоків мікроконтролера.

При використанні команд IN і OUT використовуються адреси вводу-виводу з 00h по 3Fh. Але до регістрів введення-виведення можна

звертатися і як до комірок внутрішньої пам'яті. При цьому до безпосередньої адреси вводу/виводу додається зміщення 0x20, наприклад, наступними командами читається одна й та ж сама комірка пам'яті:

```
in    r16,0x00;
ldd   r16,(0x00+0x20);
```

Регістри введення-виведення мають програмно доступні біти. Звертання до них здійснюється командами SBI і CBI, а перевірка стану - командами SBIS і SBIC. У таблиці 2.1 наведено список можливих регістрів введення-виведення мікроконтролера, що може змінюватися залежно від його типу в сімействі AVR.

Таблиця 2.1 – Регістри введення-виведення мікроконтролера

Позначення I/O регістра	Функція, що реалізується
SREG	Регістр статусу
SPH	Старший байт вказівника стеку
SPL	Молодший байт вказівника стеку
XDIV	Регістр керування значенням тактової частоти
RAMPZ	Регістр вибору сторінки пам'яті
IECR	Регістр керування зовнішніми перериваннями
EIMSK	Регістр масок зовнішніх переривань
EIFR	Регістр прапорців зовнішніх переривань
TIMSK	Регістр масок переривань таймерів/лічильників
TIFR	Регістр прапорців переривання таймера/лічильника
MCUCR	Регістр керування процесором
MCUSR	Регістр статусу процесора
TCCR 0	Регістр керування таймером/лічильником 0
TCNT 0	Таймер/лічильник 0
OCR 0	Регістр порівняння виходу таймера/лічильника 0
ASSR	Регістр статусу асинхронного режиму
TCCR1 A	Регістр керування А - таймера/лічильника 1
TCCR1 B	Регістр керування В - таймера/лічильника 1
TCNT1 H	Старший байт таймера/лічильника 1
TCNT1 L	Молодший байт таймера/лічильника 1
OCR1A H	Старший байт регістра А - порівняння виходу таймера/лічильника 1

Продовження таблиці 2.1

Позначення I/O регістра	Функція, що реалізується
OCR1A L	Молодший байт регістра A - порівняння виходу таймера/лічильника 1
OCR1B H	Старший байт регістра B- порівняння виходу таймера/лічильника 1
OCR1B L	Молодший байт регістра B - порівняння виходу таймера/лічильника 1
ICR1 H	Старший байт регістра захоплення таймера/лічильника 1
ICR1 L	Молодший байт регістра захоплення таймера /лічильника 1
TCCR 2	Регістр керування таймера/лічильника 2
TCNT 2	Таймер/лічильник 2
OCR 2	Регістр порівняння виходу таймера/лічильника 2
WDTCR	Регістр керування сторожового таймера
EEAR H	Старший байт регістра адреси EEPROM
EEAR L	Молодший байт регістра адреси EEPROM
EEDR	Регістр даних EEPROM
EECR	Регістр керування EEPROM
PORT A	Регістр даних порту A
DDR A	Регістр напрямку даних порту A
PIN A	Виводи входів порту A
PORT B	Регістр даних порту B
DDR B	Регістр напрямку даних порту B
PIN B	Виводи входів порту B
PORT C	Регістр даних порту C
PORT D	Регістр даних порту D
DDR D	Регістр напрямку даних порту D
PIND	Виводи входів порту D
SPDR	Регістр даних порту SPI
SPSR	Регістр статусу порту SPI
SPCR	Регістр керування порту SPI
UDR	Регістр даних порту UART
USR	Регістр статусу порту UART

Продовження таблиці 2.1

Позначення I/O регістра	Функція, що реалізується
UCR	Регістр керування порту UART
UBRR	Регістр керування швидкістю порту UART
ACSR	Регістр керування і статусу аналогового компаратора
ADMUX	Регістр мультиплексора АЦП
ADCSR	Регістр керування і статусу АЦП
ADC H	Старший байт регістра даних АЦП
ADC L	Молодший байт регістра даних АЦП
PORT E	Регістр даних порту E
DDR E	Регістр напрямку даних порту E
PIN E	Виводи входів порту E
PIN F	Виводи входів порту F

У просторі адрес регістрів введення-виведення знаходяться і регістри керування мікроконтролера: регістр стану, вказівник стеку, регістр вибору сторінки, регістр керування процесором, регістр керування коефіцієнтом поділу частоти тактового генератора.

Розглянемо детально формати й значення окремих бітів цих регістрів.

Під час виконання арифметичних і логічних операцій або операцій роботи з бітами, ALU формує ті чи інші ознаки результату операції, тобто встановлює або скидає біти в регістрі стану SREG (Status Register), функціональне значення яких відображено на рисунку 2.3.

I	T	H	S	V	N	Z	C

Рисунок 2.3 – Регістр стану SREG (Status Register)

Ознаки результату операції можуть бути потім використані в програмі при виконанні подальших арифметично-логічних операцій або команд умовних переходів.

Призначення окремих бітів регістра стану SREG таке:

Біт C (carry) - встановлюється, якщо під час виконання операції був перенос із старшого розряду результату.

Біт Z (zero) - встановлюється, якщо результат операції дорівнює нулю.

Біт N - встановлюється, якщо MSB (Most Significant Bit) старший біт результату дорівнює 1(правильно показує знак результату, якщо не було переповнення розрядної сітки знакового числа).

Біт V - встановлюється, якщо під час виконання операції було переповнення розрядної сітки знакового результату.

Біт S = N xor V - правильно показує знак результату і при переповненні розрядної сітки знакового числа.

Біт H - встановлюється, якщо під час виконання операції був перенос з 3-го розряду результату (перенос з молодшої тетради).

Для ознак результату операції, що є бітами регістра введення-виведення SREG, наявний цілий набір команд установлення й скидання. Команди умовних переходів у якості своїх операндів можуть мати як біти-ознаки результату операції, так і окремі розряди побітно адресованих регістрів введення-виведення.

Мікроконтролери AVR мають 16-розрядний вказівник стеку SP, розміщений у двох регістрах введення-виведення (SPH та SPL), які зображені на рисунку 2.4.

.

SPH	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8
SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
Початк.код	0	0	0	0	0	0		

Рисунок 2.4 Розміщення бітів вказівника стеку SP

Оскільки мікроконтролери підтримують ОЗП обсягом до 64 Кбайтів, то використовуються всі 16 розрядів вказівника стеку, значення якого містить адресу області ОЗП даних, у якій розміщується стек підпрограм та процедур переривань. Початкова адреса вказівника повинна задаватися програмно, перед викликом підпрограм та дозволом переривань.

Початкове значення адреси повинно бути більше 60h. Вказівник стеку декрементується на одиницю при кожному програмному занесенні даних командою PUSH у стек і на дві одиниці при апаратному занесенні в стек адреси повернення до основної програми при виконанні процедури переривання. Вказівник стеку інкрементується на одиницю при програмному відновленні даних із стеку командою POP і на дві одиниці при апаратному відновленні адреси із стеку при поверненні з підпрограми командою RET чи поверненні з процедури переривання (RETI).

Таким чином, стек зростає від старших адрес до молодших, тому з огляду на те, що початкове значення вказівника стеку після скидання дорівнює нулю, програміст обов'язково повинен подбати про початкове встановлення вмісту вказівника стеку на початку програми, якщо він планує використовувати хоча б одну підпрограму чи використовувати переривання.

Мікроконтролери, що не мають SRAM, містять трирівневий апаратний стек.

Розмір стека, що організовується в оперативній пам'яті, обмежений лише розміром цієї пам'яті. Якщо мікроконтролер містить на кристалі 128 байтів внутрішньої SRAM і не має можливості підключення зовнішньої SRAM, то як вказівник вершини стеку використовується регістр SPL. Якщо ж є можливість підключення зовнішньої пам'яті чи внутрішньої, яка має розміри 256 байтів і більше, то вказівник стеку буде складатися з двох регістрів SPL і SPH.

Якщо пам’ять програм мікроконтролера має об’єм 128 Кбайтів, то для читання пам'яті програм за допомогою 16-розрядного вказівника Z використовується керуючий регістр RAMPZ для визначення сторінки пам’яті, до якої можливе звертання за допомогою команди ELPM/SPM (рисунок 2.5).

7	6	5	4	3	2	1	0
							RAMPZ0

Рисунок 2.5 – Регістр вибору сторінки пам’яті RAMPZ

Можливі такі варіанти використання біта RAMPZ0:

RAMPZ0 = 0, команді ELPM доступна пам'ять програм з адресами від 0000h до 7FFFh (молодші 64 Кбайт);

RAMPZ0 = 1, команді ELPM доступна пам'ять програм з адресами від 8000h до FFFFh (старші 64 Кбайт).

На команду LPM установки регістру RAMPZ не впливають.

Наступним розглянемо регістр керування процесором – MCUCR, окремі біти якого наведені на рисунку 2.6.

7	6	5	4	3	2	1	0
SRE	SRW	SE	SM1	SMO	-	-	-

Рисунок 2.6 – Регістр керування процесором MCUCR

Біти регістру керування MCUCR керують виконанням основних функцій процесора:

MCUCR 7 - дозвіл роботи зовнішнього ОЗП. Встановлений біт SRE (1) дозволяє звертання до зовнішньої пам'яті даних і переводить роботу виводів AD0 - AD 7 (Порт A) для формування зовнішньої шини даних та молодшої частини шини адреси, A8 - A15 (Порт C) для формування старшої частини шини адреси, WR і RD на виконання альтернативної функції із синхронізації записування та читання зовнішньої пам’яті. Біт SRE також переналаштовує установки напрямків у відповідних регістрах

напрямку даних. Очищення біта SRE (0) забороняє звертання до зовнішньої пам'яті даних і відновлює стандартні установки напрямків портів.

MCUCR 6 - режим стану чекання. При встановленому біті SRW (1) до циклу звертань до зовнішньої пам'яті даних додається один цикл чекання. При скинутому біті SRW (0) звертання до зовнішньої пам'яті виконується за трицикловою схемою.

MCUCR 5 - дозвіл режиму енергозбереження. Встановлений в одиницю біт SE дозволяє переведення MCU у “сплячий” режим по команді SLEEP. Щоб виключити випадкове переведення MCU у режим sleep, рекомендується встановлювати біт SE безпосередньо перед виконанням команди SLEEP.

MCUCR 4,3 - Біти вибору режиму енергозбереження. Ці біти дозволяють вибрати один із трьох можливих режимів енергозбереження як показано в таблиці 2.2.

Таблиця 2.2 - Встановлення режимів енергозбереження

SM1	SM0	Режим
0	0	Режим Idle
0	1	Зарезервований
1	0	Режим Power Down
1	1	Режим Power Save

MCUCR 2…0 - зарезервовані біти регістра.

Наступний регістр XDIV використовується для установлення коефіцієнта ділення частоти кварцового генератора (рисунок 2.7). Ця можливість може бути використана для зменшення енергоспоживання, величина якого пропорційна тактовій частоті генератора.

XDIVEN	XDIV6	XDIV5	XDIV4	XDIV3	XDIV2	XDIV1	XDIV0

Рисунок 2.7 – Регістр XDIV керування частотою кварцового генератора

біт 7 (XDIVEN) - дозвіл ділення частоти XTAL. При встановленому біті XDIVEN (1) тактова частота для процесора і всієї периферії визначається як тактова частота генератора Fosc поділена на значення двійкового коду, встановленого бітами XDIV6 – XDIV0, які є бітами вибору коефіцієнта ділення. Ці біти встановлюють коефіцієнт ділення тактової частоти Fosc (при встановленому біті XDIVEN). Якщо десяткове

значення цих семи бітів позначити через d, то тактова частота процесора буде визначатися за формулою

$$FCLK=XTAL/(129-d).$$

Стан цих битів можна змінити тільки при скинутому біті XDIVEN. При встановленому біті XDIVEN, записане одноразово у біти XDIV 6..XDIV 0 значення буде визначати коефіцієнт ділення. При скиданні біта XDIVEN значення бітів XDIV6..XDIV0 ігноруються.

Для збереження оперативних даних програміст, крім регістрового файлу, може використовувати внутрішню і зовнішню пам'ять, якщо вона присутня - блоки SRAM (див.рис. 2.1). Робота з зовнішньою SRAM може бути програмно дозволена/заборонена - установленням/скиданням біта SRE у регістрі введення-виведення MCUSR.

Операції обміну з внутрішньою оперативною пам'яттю мікроконтролер виконує за два машиних цикли. Доступ до зовнішньої SRAM вимагає одного додаткового циклу на кожен байт у порівнянні з доступом до внутрішньої пам'яті. Крім того, установкою біта SRW у регістрі введення-виведення MCUSR можна програмно збільшити час обміну з зовнішньої SRAM ще на один додатковий машиний цикл чекання.

Оскільки внутрішня і зовнішня SRAM входять до єдиного адресного простору (разом з оперативними регістрами і регістрами введення-виведення), то для доступу до комірок внутрішньої і зовнішньої пам'яті використовуються ті ж самі команди.

Регістрами введення-виведення є також регістри, що керують системою переривання мікроконтролера, роботою енергонезалежної EEPROM пам'яті, сторожовим таймером, портами введення-виведення й іншими периферійними вузлами. Вивчення даних регістрів зручно виконувати одночасно з вивченням конкретного периферійного вузла.

Регістровий файл, блок регістрів введення-виведення й оперативна пам'ять, як показано на рисунку 2.1, утворюють єдиний адресний простір, що дає можливість при програмуванні звертатися до 32 оперативних регістрів і до регістрів введення-виведення як до комірок пам'яті, використовуючи команди доступу до SRAM (у тому числі і з непрямою адресацією). Внутрішня SRAM користувача всіх AVR мікроконтролерів починається з адреси 60h.

Слід зазначити, що регістри введення-виведення не повністю використовують відведені для них 64 адреси. Невикористані адреси зарезервовані для розширення мікроконтролера новими периферійними блоками у майбутньому, додаткових комірок пам'яті за цими адресами не існує.

Варто також мати на увазі, що в різних типах AVR мікроконтролерів одні й ті самі регістри введення-виведення можуть мати різні адреси. Для того, щоб забезпечити сумісність програмного забезпечення з одного типу кристала на інший, варто використовувати в програмі стандартні, прийняті в оригінальній фірмовій документації символічні імена регістрів введення-виведення, а відповідність цих імен реальним адресам задавати на початку

своєї програми за допомогою директиви асемблера .INCLUDE, яка підключає до програмного проекту файл визначення адрес регістрів введення-виведення з розширенням .inc. Такі файли вже створені розробниками фірми ATMEL і вільно поширюються разом з документацією на мікроконтролери AVR. У цих файлах задається відповідність між символічними іменами та адресами регістрів введення-виведення. Якщо для звертання до регістра введення-виведення використовуються команди обміну з пам'ятю SRAM, то до символічного імені необхідно додати зсув у вигляді числа 20h.

Крім оперативної пам'яті програмно доступними ресурсами мікроконтролера є енергонезалежні, електрично програмовані FLASH і EEPROM блоки пам'яті, що мають окремі адресні простори. Всі команди AVR є 16-розрядними словами, тому flash-пам'ять організована як послідовність 16-розрядних комірок і має ємність від 512 слів до 64 Кбайтів слів залежно від типу кристала.

У flash-пам'ять, крім програми, можуть бути записані постійні дані, що не змінюються під час функціонування мікропроцесорної системи. Дані з FLASH пам'яті можуть бути програмним шляхом зчитані в регістровий файл за допомогою команд LPM, ELPM (дивись групу команд передачі даних).

Молодші адреси пам'яті програм мають спеціальне призначення. Адреса 0000h є вектором переривання за сигналом RESET - адресою, з якої починає виконуватися програма після скидання процесора. Починаючи з наступнї адреси 0001h, комірки пам'яті програм містять вектори переривань від інших сигналів мікроконтролера. У цій області для кожного можливого джерела переривання відведена своя адреса, за якою (у випадку використання даного переривання) розміщують команду відносного переходу RJMP на підпрограму обробки переривання (див. рисунок 2.1).

Варто пам'ятати, що адреси векторів переривання одних і тих самих апаратних вузлів для різних типів AVR можуть мати різне значення. Тому для забезпечення сумісності програмного забезпечення, так само як і у випадку з регістрами введення-виведення, зручно використовувати символічні імена адрес векторів переривання, що визначені у відповідному .inc-файлі.

EEPROM блок пам'яті даних, що електрично стирається, призначений для збереження енергонезалежних даних, які можуть змінюватися безпосередньо на об'єкті. Це калібровані коефіцієнти, різні уставки, конфігураційні параметри системи і т.п. Таким чином, EEPROM-пам'ять даних може бути програмним шляхом як прочитана, так і записана. Однак спеціальних команд звертання до EEPROM-пам'яті немає. Читання і записування окремих комірок EEPROM виконується через регістри введення-виведення - EEAR (регістр адреси), EEDR (регістр даних) і EECR (регістр керування).

2.2 Методи адресації інформації в командах

При створенні програми мовою Асемблер, розробник оперує з програмно доступними ресурсами мікропроцесорної системи. За складом та кількістю реалізованих інструкцій AVR більше схожі на CISC мікроконтролери. Наприклад, у PIC-контролерів система команд нараховує до 75 різних інструкцій, а в MCS 51 вона складає 111 інструкцій.

Система команд AVR досить розвинута і нараховує до 133 різних інструкцій, але блок керування мікроконтролера AVR апаратно реалізує лише частину з їх числа, інші ж інструкції емулюються програмно. Майже всі команди мають фіксовану довжину в одне слово (16 біт), що дозволяє в більшості випадків поєднувати в одній команді код операції і операнди. Лише деякі команди мають розмір у два слова (32 біт) і відносяться до групи команд виклику підпрограми - CALL, довгих переходів у межах всього адресного простору - JMP, повернення з підпрограм RET і команд роботи з пам'яттю програм - LPM. Розрізняють п'ять груп команд:

– команди пересилання даних,
– команди арифметичних і логічних операцій,
– команди роботи з окремими бітами,
– команди умовних та безумовних переходів.

В останніх версіях кристалів AVR сімейства "mega" реалізована функція апаратного множення, що дає реалізовувати досить швидкі обчислювальні алгоритми. Тому прогресивна архітектура RISC у поєднанні з наявністю регістрового файлу і розширеної системи команд дозволяє в короткий термін створювати програми з ефективним кодом як за величиною, так і за швидкістю виконання.

При переході від молодших до старших моделей AVR існує сумісність системи команд, розподілення пам'яті, але необхідно пам'ятати, що адреси векторів переривання одних і тих самих периферійних вузлів у різних типів AVR різні, що вимагає внесення відповідних змін у програму при її перенесенні на інший тип AVR. Така відповідність між ресурсами мікропроцесора та адресами пам'яті описується, наприклад, для мікроконтролера AT90S2313 при використанні Асемблера директивою .include "1200def.inc", а при використанні мови Сі включенням в проект відповідного файлу, наприклад, #include <io2313.h>.

Кожна команда процесора кодує поточну операцію, методи доступу до операндів та містить пряму або опосередковану адресу наступної команди. Командний рядок Асемблера містить мітку (символічна адреса), мнемоніку (символічне ім'я) команди, поле операндів, коментар. Особливістю системи команд є те, що арифметико-логічні операції можна виконувати лише на регістрах регістрового файлу, а команди обробки даних типу регістр-пам'ять або пам'ять-пам'ять заборонені. Між оперативною пам'ятю та регістрами дозволені лише команди пересилання даних.

Методи адресації – це набір механізмів доступу до операндів. Одні з них прості та короткі і тому приводять до компактного формату команди і швидкого доступу до операнда, але обсяг доступних з їхньою допомогою ресурсів обмежений. Інші методи адресації дозволяють оперувати з усіма наявними в системі ресурсами, але команда виходить задовгою, на її завантаження та виконання витрачається більше часу.

Відмітимо, що при наявності в команді адрес двох операндів та адреси результату, має місце триадресна команда, де кожна адреса формується з використанням власного методу адресації. Але, як правило, результат операції розміщується за місцем першого операнда, а тому такі команди є двоадресними.

Мікроконтролери AVR підтримують такі способи адресації операндів в командах.

1. Регістрова адресація.

а) Регістрова адресація з одним регістром.

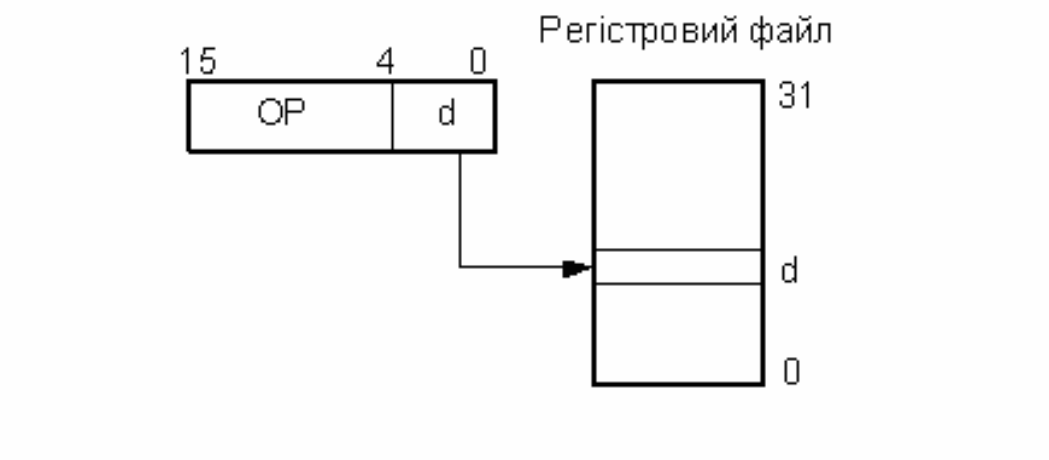


Рисунок 2.8 – Регістрова адресація (один регістр загального призначення)

Адреса регістра знаходиться в полі d команди, наприклад:

```
clr     r16;   r16=0
```

б) Регістрова адресація з двома регістрами.

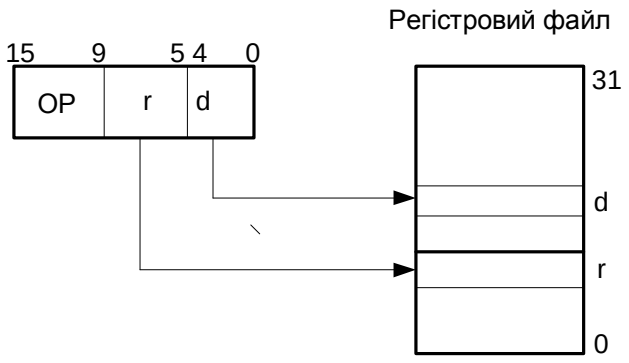


Рисунок 2.9 – Регістрова адресація (два регістри загального призначення)

Адреси регістрів знаходяться в полі r та d команди, наприклад:

mov r16,r17; r16 = r17

в) Регістрова адресація з регістром введення - виведення.

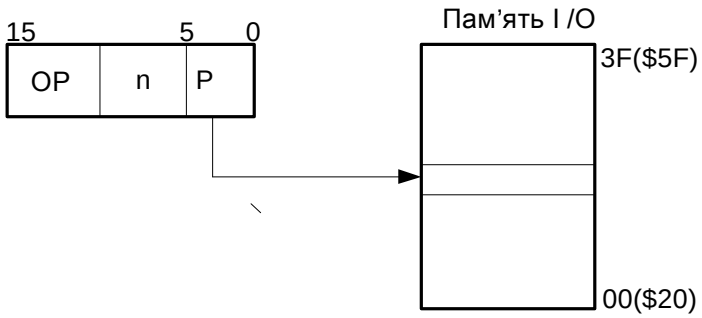


Рисунок 2.10 – Регістрова адресація (регістри введення - виведення)

Адреса регістра знаходиться в полі P команди, наприклад:

out portB,r16; portB =r16

2. Безпосередня адресація даних.

Передбачає присутність операнду безпосередньо в самій команді, наприклад:

ldi r16,0x66; r16=0x66.

Наведений приклад ілюструє наявність одночасно двох способів адресації даних в команді – регістрова адресація для приймача та безпосередня адресація для передавача.

3. Пряма адресація.

а) Пряма адресація даних.

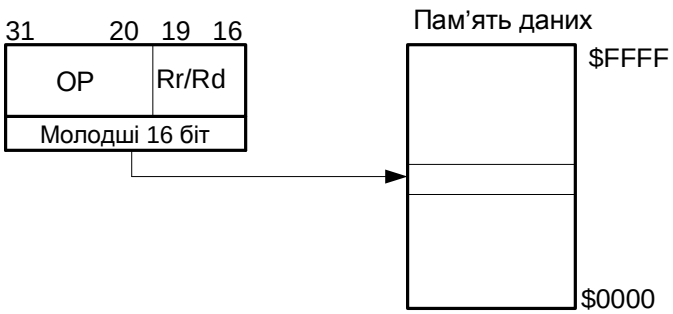


Рисунок 2.11 – Пряма адресація даних

В полі команди безпосередньо розміщується пряма адреса комірки пам’яті, наприклад:

```
lds     r16,0x60;    r16 = (0x60)
sts     0x70,r18;    (0x70) = r18
```

б) Пряма адресація команд.

В полі команди міститься пряма адреса команди при безумовному переході (jmp) чи адреса першої команди процедури (call), наприклад:

```
jmp   marker;
call  name;
```

4. Непряма (опосередкована) адресація даних.

а) Непряма адресація даних із зміщенням.

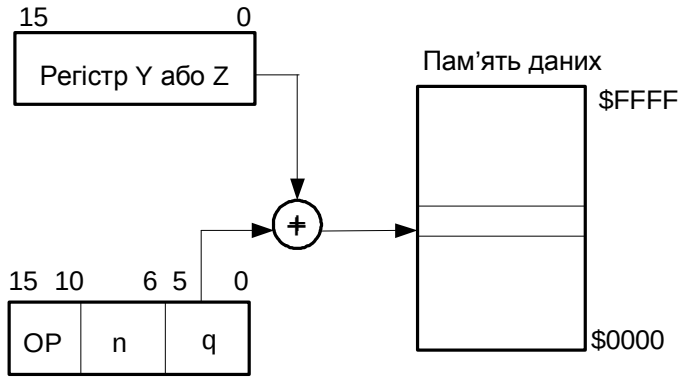


Рисунок 2.12 – Непряма адресація даних із зсувом

Адреса пам'яті визначається як сума адресного регістра-вказівника та зміщення, яке безпосередньо розміщено в полі команди q, наприклад

```
ldd   r16,Y+q;      r16 = (Y+q)
std   Z+q,r17;      (Z+q) = r17
```

б) Непряма адресація даних.

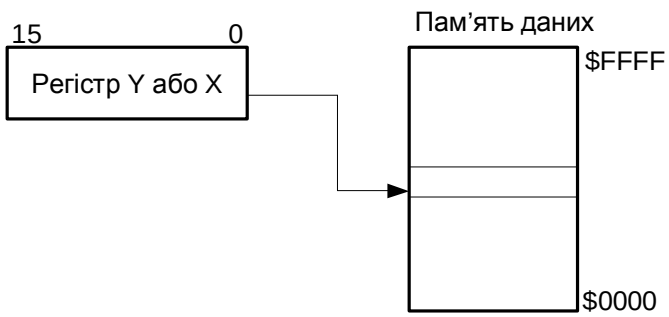


Рисунок 2.13 – Непряма адресація даних

Адреса пам'яті визначається вмістом адресного регістра-вказівника, який використовується в команді, наприклад:

```
ldd     r16,Y;        r16 = (Y)
std     Z,r17;        (Z) = r17
```

в) Непряма адресація даних з переддекрементом.

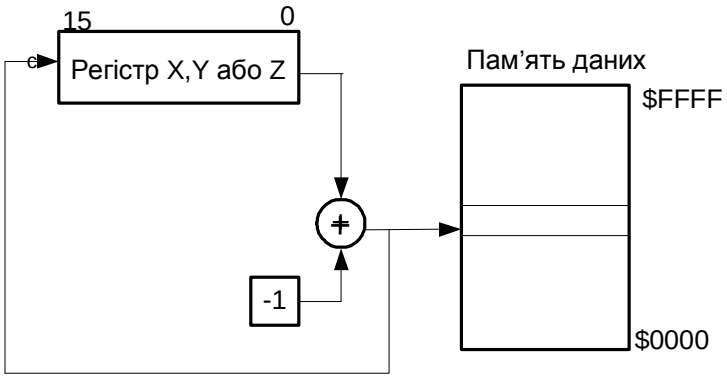


Рисунок 2.14 – Непряма адресація даних із переддекрементом

Перед виконанням команди вміст адресного регістра-вказівника зменшується на 1 (декрементується), а отримана адреса використовується для доступу до оперативної пам’яті, наприклад:

```
ld      r16,-Y;       Y=Y-1,  r16 = (Y)
st      -Z,r17;       Z=Z-1,   (Z) = r17
```

г) Непряма адресація даних з постінкрементом.

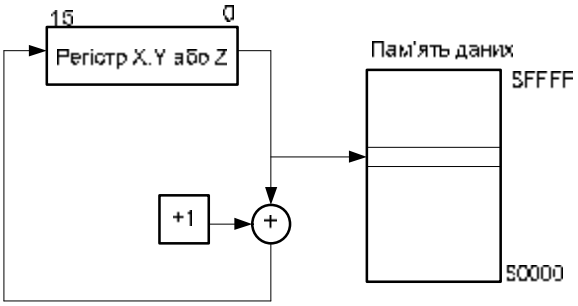


Рисунок 2.15 – Непряма адресація даних з постінкрементом

Вміст регістра-вказівника використовується для доступу до оперативної пам’яті, а після виконання операції вміст адресного регістра-вказівника збільшується на 1 (інкрементується), наприклад:

```
ld      r16,Y+;       r16 = (Y), Y=Y+1
st      Z+,r17;       (Z) =  r17, Z=Z+1
```

д) Непряма адресація констант програмної пам’яті.

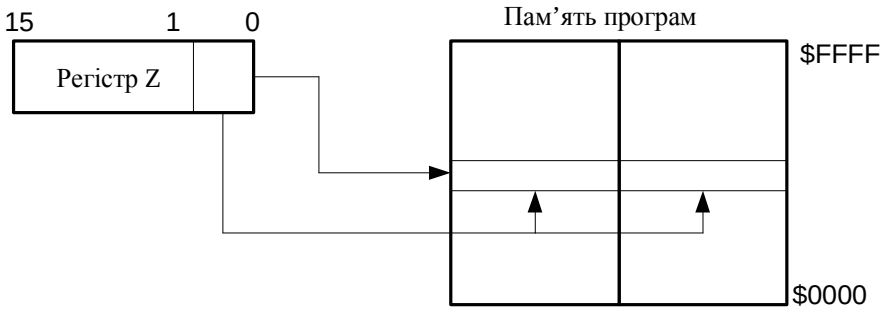


Рисунок 2.16 – Адресація константи в пам'яті програм в командах LPM та ELPM

Адреса константи - байту міститься у вказівнику Z. Старші 15 бітів визначають адресу слова, а молодший біт – значення молодшого чи старшого байту константи пам'яті програм (0 – вибір молодшого байту, 1 – старшого байту), наприклад:

```
lpm ;       r0=(Z)
```

е) Непряма адресація пам'яті програм у командах ijmp та icall.

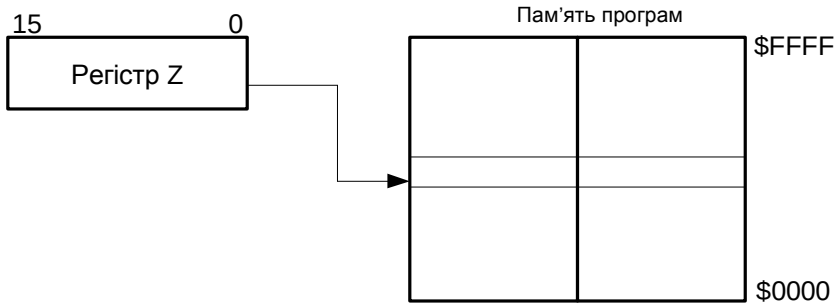


Рисунок 2.17 – Непряма адресація пам'яті програм у командах ijmp та icall

В програмний лічильник завантажується команда, адреса якої міститься в регістрі-вказівнику Z, наприклад:

```
ijmp;       PC =  (Z)
icall;      PC =  (Z)
```

5. Відносна адресація пам'яті програм у командах rjmp та rcall.

Після виконання команди rjmp та rcall виконання наступної команди продовжується з адреси (PC+k+1). Відносна адреса k може змінюватись від -2048 до 2047.

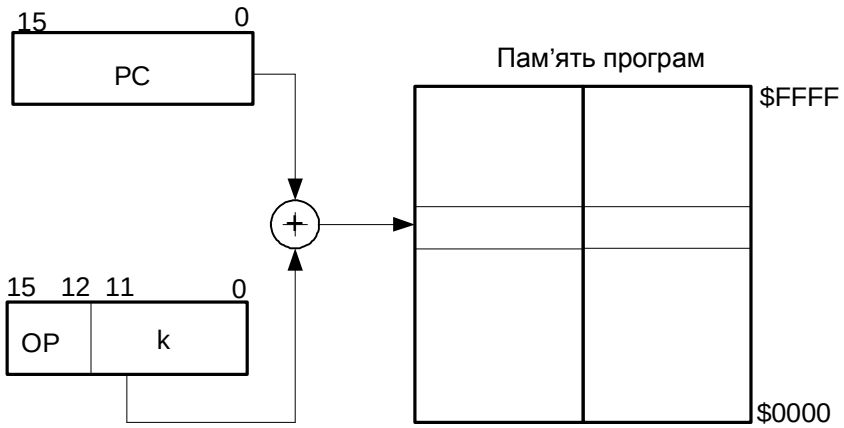


Рисунок 2.18 – Відносна адресація пам'яті програм

6. Бітова адресація даних в команді передбачає звертання лише до окремого вказаного в команді біта, наприклад:

```
sbi    PORTB,6;   PORTB.6=1
clt;              T=0
```

Молодші моделі AVR не мають деяких команд. Основна відмінність полягає в тому, що мікроконтролери, у яких відсутня SRAM, (наприклад, AT90S1200, ATtiny10/11) не мають і відповідних команд роботи з оперативною пам'ятю. Крім того, AT90S1200 не має команд ADI, SBI, IJMP, ICALL, LPM, а ATtiny10/11 - команд ADI, SBI, IJMP, ICALL.

Тільки мікроконтролери MegaAVR мають двослівні команди абсолютних переходів JMP і CALL, які виконуються за три такти. Всім іншим типам AVR (Tiny та Classic) такі повільні команди не потрібні, оскільки весь адресний простір обсягом до 4 Кбайтів слів доступний за допомогою команд відносних переходів RJMP, RCALL.

Також окремо знаходиться команда ELPM сторінкового читання flash-пам'яті, що необхідна тільки для ATmega128, через збільшення розмірів пам'яті програм у цього мікроконтролера до 128 Кбайтів.

Варто звернути увагу на те, що ряд різних за мнемокодом команд із групи арифметичних операцій транслюються в той самий двійковий код, наприклад, ANDI та CBR, ORI та SBR. Розробник може довільно виконувати вибір між ними, залежно від контексту програми.

Спеціальна директива асемблера .device <typeAVR> забезпечує контроль відповідності команд, використаних у тексті програми, типу зазначеного процесора.

При переході від молодших до старших моделей AVR існує сумісність на рівні системи команд, але адреси векторів переривання одних і тих самих периферійних вузлів у різних типах AVR різні, що вимагає внесення відповідних змін у програму при її перенесенні на новий тип мікроконтролера сімейства.

2.3 Система команд мікроконтролера

Розглянемо особливості окремих груп команд за їх функціональним призначенням.

Команди передачі даних наведені в поданій нижче таблиці 2.3.

Таблиця 2.3 – Команди передачі даних

Мне-моні-ка	Операнди	Опис команди	Операція	Прапорці	Цикл
ELPM		Розширене завантаження пам'яті програм в R0	R0= (RAMPZ:Z)	Hi	3
MOV	Rd, Rr d=0-31 r=0-31	Копіювати регістр	Rd=Rr	Hi	1
LDI	Rd, k 1 d=16-31 k=0-0xff	Завантажити регістр безпосередньою константою	Rd=k	Hi	1
LDS	Rd, k d=0-31 k=0-0xffff	Записування в регістр вмісту прямої адреси пам’яті	Rd = (k)	Hi	3
LD	Rd, X d= 0-31	Завантажити регістр вмістом пам’яті з непрямою адресацією	Rd = (X)	Hi	2
LD	Rd, X+ d=0-31	Завантажити регістр вмістом пам’яті з непрямою адресацією та постінкрементом	Rd = (X), X=X+1	Hi	2
LD	Rd, -X d=0-31	Преддекремент вказівника і завантаження регістра вмістом пам’яті з непрямою адресацією	X=X-1, Rd= (X)	Hi	2
LD	Rd, Y d=0-31	Завантажити регістр вмістом пам’яті з непрямою адресацією	Rd = (Y)	Hi	2
LD	Rd, Y+ d=0-31	Завантажити регістр вмістом пам’яті з непрямою адресацією та постінкрементом	Rd = (Y), Y=Y+1	Hi	2
LD	Rd, -Y d=0-31	Переддекремент вказівника і завантаження регістра вмістом пам’яті	Y=Y-1, Rd= (Y)	Hi	2

Продовження таблиці 2.3

Мне-моні-ка	Операнди	Опис команди	Операція	Прапорці	Цикл
LDD	Rd, Y+q d=0-31 q=0- 63	Завантажити регістр вмістом пам’яті з непрямою адресацією та зсувом	Rd = (Y+q)	Hi	2
LD	Rd, Z d=0-31	Завантажити регістр вмістом пам’яті з непрямою адресацією	Rd = (Z)	Hi	2
LD	Rd, Z+ d=0-31	Завантажити регістр вмістом пам’яті з непрямою адресацією та постінкрементом	Rd = (Z), Z=Z+1	Hi	2
LD	Rd, -Z D=0-31	Переддекремент вказівника і завантаження регістра вмістом пам’яті з непрямою адресацією	Z=Z- 1, Rd = (Z)	Hi	2
LDD	Rd, Z+q d=0-31 q=0- 63	Завантажити регістр вмістом пам’яті з непрямою адресацією та зсувом	Rd= (Z+q)	Hi	2
STS	k, Rr d=0-31 k=0-0xffff	Записування в пряму адресу пам’яті з регістра	(k) = Rr	Hi	3
ST	X, Rr r=0-31	Записування в пам’ять з регістра через вказівник	(X) = Rr	Hi	2
ST	X+, Rr r=0-31	Записування з регістра в пам’ять з непрямою адресацією та постінкрементом	(X)=Rr, X=X+ 1	Hi	2
ST	-X, Rr R=0-31	Переддекремент вказівника і запис з регістра в пам’ять з непрямою адресацією	X=X- 1, (X) = Rr	Hi	2
ST	Y, Rr r=0-31	Завантажити в пам’ять з регістра через вказівник	(Y) = Rr	Hi	2

Продовження таблиці 2.3

Мне-моні-ка	Операнди	Опис команди	Операція	Прапорці	Цикл
ST	Y+, Rr r=0-31	Завантажити з регістра в пам’ять з непрямою адресацією та постінкрементом	(Y)= Rr, Y=Y+1	Hi	2
ST	-Y, Rr r=0-31	Переддекремент вказівника і Завантажити з регістра в пам’ять з непрямою адресацією	Y=Y-1, (Y) = Rr	Hi	2
STD	Y+q, Rr d=0-31 q=0- 63	Завантажити регістр вмістом пам’яті з непрямою адресацією та зсувом	(Y+q)= Rr	Hi	2
ST	Z, Rr r=0-31	Записування в пам’ять з регістра через вказівник	(Z) = Rr	Hi	2
ST	Z+, Rr r=0-31	Завантажити з регістра в пам’ять з непрямою адресацією та постінкрементом	(Z)= Rr, Z=Z+ 1	Hi	2
STD	Z+q, Rr d=0-31 q=0- 63	Завантажити регістр вмістом пам’яті з непрямою адресацією та зсувом	Z+q= Rr	Hi	2
LPM		Завантажити байт із пам'яті програм	R0= (Z)	Hi	3
IN	Rd, P d=0-31 P=0-63	Завантажити в регістр дані з порту I/O	Rd= P	Hi	1
OUT	P, Rr d=0-31 P=0-63	Записати в порт I/O дані з регістра	P=Rr	Hi	1
PUSH	Rr r=0-31	Зберегти вміст регістра у стеку	STACK = Rr	Hi	2
POP	Rd d=0-31	Відновити вміст регістр із стека	Rd = STACK	Hi	2

Команда MOV копіює вміст одного регістра загального призначення в інший. Команди ELPM і LPM виконують завантаження з пам'яті програм.

Команда LDI завантажує регістр загального призначення безпосередньо константою (тільки регістри R16 - R31), а команди LD виконують завантаження регістра з комірки пам'яті з використанням непрямої адресації через вказівники адреси X, Y, Z, а також варіанти з постінкрементом та переддекрементом вказівника адреси.

Команда LDD виконує аналогічні операції за допомогою непрямої адресації із зсувом.

Команда LDS завантажує в регістр загального призначення вміст комірки пам'яті даних за допомогою прямої адреси, яка міститься в команді.

Команда ST виконує обернені операції відносно команди LD - заносить в комірку пам'яті даних вміст регістра з використанням адреси комірки у вказівнику адреси. Доступні також варіанти з постінкрементом і переддекрементом вказівника адреси.

Команда STD виконує операції записування в комірку пам'яті даних вміст регістра з використанням опосередкованої адресації із зсувом.

Команда STS виконує операції записування в комірку пам'яті даних вміст регістра з використанням прямої адресації.

Команда IN дає змогу зчитувати вміст регістра введення-виведення в регістр загального призначення, а команда OUT виконує обернену операцію.

Команда PUSH зберігає вміст регістра у стеку, а команда POP виконує обернену операцію відновлення вмісту регістра. При виконанні цих команд крім програмного лічильника змінюється і значення вказівника стека SP.

Необхідно звернути увагу, що команди пересилання даних, як і розглянуті нижче команди переходів та передачі керування, значення прапорців регістра SREG не модифікують.

Організація виконання циклів, зміна напрямку обчислень, робота з підпрограмами виконується за допомогою команд передачі керування (переходів), які наведені у таблиці 2.4.

Таблиця 2.4 - Команди переходів і передачі керування

Мне-моніка	Операнди	Опис	Операція	Пра-порці	Цикли
RJMP	K	Відносний перехід	PC= PC + k + 1	Ні	2
LJMP		Перейти непрямо за вмістом вказівника Z	PC=Z	Ні	2
JMP	k	Перейти за адресою	PC=k	Ні	3

Продовження таблиці 2.4

Мне-моніка	Операнди	Опис	Операція	Пра-порці	Цикли
RCALL	K	Відносний виклик підпрограми	PC= PC + k + 1	Ні	3
ICALL		Виклик підпрограми за вказівником Z	PC=Z	Ні	3
CALL	k k0-64K	Виконати довгий виклик підпрограми	PC=k	Ні	4
RET		Повернення з підпрограми	PC = STACK	Ні	4
RETI		Повернення з переривання та його дозвіл	PC = STACK	Ні	4
CPSE	Rd, Rr d=0-31, r=0-31	Порівняти і пропустити, якщо регістри рівні	if (Rd= =Rr) PC = PC + 2 (ore 3)	Ні	1/2/3
SBRC	Rr,b r=0-31, b=0-7	Пропустити, якщо біт у регістрі очищений	if (Rr(b)= =0) PC = PC + 2 (ore 3)	Ні	1/2/3
SBRS	Rr,b r=0-31, b=0-7	Пропустити, якщо біт у регістрі встановлений	if (Rr(b)= =1) PC = PC + 2 (ore 3)	Ні	1/2/3
SBIC	P,b r=0-31, b=0-7	Пропустити, якщо біт у регістрі P I/O очищений	if (P(b)= =0) PC= PC + 2 (ore 3)	Ні	1/2/3
SBIS	P,b r=0-31, b=0-7	Пропустити, якщо біт у регістрі P I/O установлений	if (P(b)= =1) PC = PC + 2 (ore 3)	Ні	1/2/3
BRBS	S,k s=0-7 k=-64 - +63	Перейти, якщо біт у регістрі статусу встановлений	if (SREG(s)= =1) PC = PC + k + 1	Ні	1/2
BRBC	S,k s=0-7 k=-64 - +63	Перейти, якщо біт у регістрі статусу очищено	if (SREG(s)= =0) PC = PC + k + 1	Ні	1/2
BREQ	K k=-64 - +63	Перейти, якщо дорівнює	if (SREG(Z)= =1) PC = PC + k + 1	Ні	1/2
BRNE	K k=-64 - +63	Перейти, якщо не дорівнює	if (SREG(Z)= =0) PC= PC + k + 1	Ні	1/2
BRCS	k k=-64 - +63	Перейти, якщо біт переносу встановлено	if(C= =1) PC = PC + k + 1	Ні	1/2

Продовження таблиці 2.4

Мне-моніка	Операнди	Опис	Операція	Пра-порці	Цикли
BRCC	k k=-64 - +63	Перейти, якщо біт переносу обнулено	if(C= =0) PC = PC + k + 1	Hi	1/2
BRSH	k k=-64 - +63	Перейти, якщо дорівнює чи більше (без знаку)	if (Rd ≥Rr) (C= =0) PC = PC + k + 1	Hi	1/2
BRLO	k k=-64 - +63	Перейти, якщо менше (без знаку)	if (Rd < Rr) (C= =1) PC = PC + k + 1	Hi	1/2
BRMI	k k=-64 - +63	Перейти, якщо мінус	if(N= =1) PC = PC + k + 1	Hi	1/2
BRPL	k k=-64 - +63	Перейти, якщо плюс	if(N= =0) PC = PC + k + 1	Hi	1/2
BRGE	k k=-64 - +63	Перейти, якщо більше або дорівнює (із знаком)	if(Rd≥Rr) (N xor V= =0) PC = PC + k + 1	Hi	1/2
BRLT	k k=-64 - +63	Перейти, якщо менше (із знаком)	if (Rd<Rr) (N xor V= =1) PC = PC + k +1	Hi	1/2
BRHS	k k=-64 - +63	Перейти, якщо прапорець півпереносу встановлений	if (H= =1) PC =PC + k +1	Hi	1/2
BRHC	k k=-64 - +63	Перейти, якщо прапорець півпереносу очищений	if(H= =0) PC =PC + k + 1	Hi	1/2
BRTS	k k=-64 - +63	Перейти, якщо прапорець T встановлений	if (T= =1) PC = PC + k + 1	Hi	1/2
BRTC	k k=-64 - +63	Перейти, якщо прапорець T очищений	if (T= =0) PC = PC + k + 1	Hi	1/2
BRVS	k k=-64 - +63	Перейти, якщо прапорець переповнення встановлений	if (V= =1) PC = PC + k + 1	Hi	1/2
BRVC	k k=-64 - +63	Перейти, якщо прапорець переповнення очищений	if (V= =0) PC = PC + k + 1	Hi	1/2

Продовження таблиці 2.4

Мне-моніка	Операнди	Опис	Операція	Пра-порці	Цикли
BRIE	k k=-64 - +63	Перейти, якщо біт глобального дозволу переривань встановлено	if (I= =1) PC = PC + k + 1	Ні	1/2
BRID	k k=-64 - +63	Перейти, якщо біт глобального дозволу переривань очищено	if (I= =0) PC =PC + k + 1	Ні	1/2

Для обробки даних мікроконтролер використовує групу команд, що реалізують арифметичні і логічні операції, а також операції зсуву. Важливу і досить велику групу утворюють команди, що виконують операції над окремими бітами.

Всі арифметичні та логічні операції, а також частина операцій роботи з бітами (див. таблицю 2.4) виконуються в ALU тільки над вмістом регістрів. Варто звернути увагу, що команди, які другим операндом мають константу (SUBI, SBCI, ANDI, ORI, SBR, CBR), можуть використовувати першим операндом тільки регістри з другої половини регістрового файлу (R16-R31). Команди 16-розрядного додавання з константою ADI і віднімання константи SBI, першим операндом використовують тільки регістри R24, R26, R28, R30.

Арифметичні операції є операціями над 8-розрядними цілими числами. Для виконання операцій над багатобайтовими словами використовуються команди додавання і віднімання з урахуванням прапорця переносу. Для виконання операцій над числами з плаваючою комою, реалізації тригонометричних функцій і т.п. слугують бібліотеки програм, що входять до складу інтегрованої системи програмування «Асемблер-AVR».

Арифметичні і логічні команди, а також команди зсуву та операції з бітами наведені в наступних двох таблицях 2.5 та 2.6.

Таблиця 2.5 - Арифметичні та логічні команди

Мне-моніка	Операнди	Опис	Операція	Пра-порці	Цикли
ADD	Rd, Rr d=0 - 31 r=0 - 31	Додати без переносу	Rd = Rd + Rr	Z,C, N,V, H	1
ADC	Rd, Rr d=0 - 31 r=0 - 31	Додати з переносом	Rd = Rd + Rr + C	Z, C, N,V, H	1

Продовження таблиці 2.5

Мне-моніка	Операнди	Опис	Операція	Пра-порці	Цикли
ADIW	Rd, K d{24,26,28,30} K=0 - 63	Додати безпосереднє значення із словом	Rdh:Rdl = Rdh:Rdl + K	Z,C, N,V	2
SUB	Rd, Rr d=0 - 31 r=0 - 31	Відняти без позички	Rd = Rd - Rr	Z,C, N,V, H	1
SUBI	Rd, K d=16 - 31 K=0 - 255	Відняти безпосереднє значення	Rd = Rd - K	Z,C, N,V, H	1
SBC	Rd, Rr d=0 - 31 r=0 - 31	Відняти з позичкою	Rd =Rd - Rr-C	Z, C, N,V, H	1
SBCI	Rd, K d=16 - 31 K=0 - 255	Відняти безпосереднє значення з позичкою	Rd =Rd - K- C	Z,C, N,V, H	1
SBIW	Rd, K d{24,26,28,30} K=0 - 63	Відняти безпосереднє значення від слова	Rdh:Rdl = Rdh:Rdl -K	Z,C, N, V	2
AND	Rd, Rr d=0 - 31 r=0 - 31	Виконати логічне множення	Rd = Rd & Rr	Z, N,V	1
ANDI	Rd, K d=16 - 31 K=0 - 255	Логічне множення з безпосереднім значенням	Rd = Rd & K	Z, N,V	1
OR	Rd, Rr d=0 - 31 r=0 - 31	Логічне додавання	Rd = Rd v Rr	Z, N,V	1
ORI	Rd, K d=16 - 31 K=0 – 255	Логічне додавання з безпосереднім значенням	Rd = Rd v K	Z, N,V	1
EOR	Rd, Rr d=0 - 31 r=0 – 31	Виключальне АБО	Rd = Rd xor Rr	Z, N,V	1
COM	Rd d=0 - 31	Доповнення до одиниці	Rd = 0xff - Rd	Z,C, N, V	1

Продовження таблиці 2.5

Мне-моніка	Операнди	Опис	Операція	Пра-порці	Цикли
NEG	Rd d=0 - 31	Доповнення до двох	Rd = 0x00 -Rd	Z,C, N, V,H	1
SBR	Rd, K d=16 - 31 K=0 – 255	Установити біт(и) в регістрі	Rd = Rd v K	Z, N,V	1
CBR	Rd, K d=16 - 31 K=0 – 255	Очистити біти в регістрі	Rd=Rd &(0xff - K)	Z,N,V	1
INC	Rd d=0 - 31	Інкрементувати вміст регістра	Rd = Rd + 1	Z,N,V	1
DEC	Rd d=0 - 31	Декрементувати вміст регістра	Rd = Rd - 1	Z, N,V	1
TST	Rd d=0 – 31	Перевірити на нуль чи мінус	Rd = Rd & Rd	Z, I,M,V	1
CLR	Rd d=0 - 31	Очистити регістр	Rd = Rd xor Rd	Z, N,V	1
SER	Rd d=0 - 31	Установити всі біти регістра	Rd = 0xff	Hi	1
CP	Rd, Rr d=0 - 31 r=0 – 31	Порівняти	Rd-Rr	Z,C, N,V, H	1
CPC	Rd, Rr d=0 - 31 r=0 – 31	Порівняти з врахуванням переносу	Rd - Rr - C	Z,C, N, V, H	1
CPI	Rd, K d=16 - 31 K=0 – 255	Порівняти з константою	Rd-K	Z,C, N, V, H	1

Таблиця 2.6 - Команди зсуву і операції з окремими бітами

Мне-моніка	Операнди	Опис	Операція	Пра-порці	Цикли
LSL	Rd d=16 - 31	Логічно зсунути ліворуч	Rd(n+1)=Rd(n), Rd(0) =0, C = Rd(7)	Z,C,N,V, H	1
LSR	Rd d=16 - 31	Логічно зсунути праворуч	Rd(n)=Rd(n+1), Rd(7) =0, C = Rd(0)	Z,C,N,V	1
ROL	Rd d=16 - 31	Зсунути ліворуч через перенос	Rd(0) =C, Rd(n+1)=Rd(n), C = Rd(7)	Z,C,N,V, H	1
ROR	Rd d=16 - 31	Зсунути праворуч через перенос	Rd(7) =C, Rd(n)=Rd(n+1), C = Rd(0)	Z,C,N,V	1
ASR	Rd d=16 - 31	Арифметично зсунути праворуч	Rd(n)=Rd(n+1), C=Rd(0), n=0 - 6	Z,C,N,V	1
SWAP	Rd d=16 - 31	Поміняти тетради місцями	Rd (3 - 0) = Rd (7 - 4)	Hi	1
BSET	s s =0 - 7	Встановити прапорець	SREG(s)= 1	SREG(s)	1
BCLR	s s =0 - 7	Очистити прапорець	SREG(s)= 0	SREG(s)	1
SBI	P, b P =0 - 31 b =0 - 7	Встановити біт в регістрі I/O	l/0(P,b)= 1	Hi	2
CBI	P,b P =0 - 31 b =0 – 7	Очистити біт в регістрі I/O	l/0(P,b)= 0	Hi	2
BST	Rd, b d= 0 - 31 b =0 - 7	Переписати біт з регістра в прапорець T	T= Rd(b)	T	1
SEC		Встановити прапорець переносу	C=1	C	1
SEN		Встановити прапорець від’ємного значення	N=1	N	1

Продовження таблиці 2.6

Мне-моніка	Операнди	Опис	Операція	Пра-порці	Цикли
CLN		Очистити прапорець N від'ємного значення	N=0	N	1
SEZ		Встановити прапорець нульового значення	Z=1	Z	1
CLZ		Очистити прапорець нульового значення	Z=0	Z	1
SEI		Встановити прапорець глобального переривання	l= 1	I	1
CLI		Очистити прапорець глобального переривання	l=0	I	1
SES		Встановити прапорець знаку	S=1	S	1
CLS		Очистити прапорець знаку	S=0	S	1
SEV		Встановити прапорець переповнення	V=1	V	1
CLV		Очистити прапорець переповнення	V=0	V	1
SET		Встановити прапорець T	T=1	T	1
CLT		Очистити прапорець T	T=0	T	1
SEH		Встановити прапорець півпереносу	H=1	Hi	1
CL		Очистити прапорець півпереносу	H=0	Hi	1
NOP		Виконати холосту команду		Hi	1

Продовження таблиці 2.6

Мне-моніка	Операнди	Опис	Операція	Пра-порці	Цикли
SLEEP		Встановити режим SLEEP		Ні	1
WDR		Очистити лічильник сторожового таймера WDT		Ні	1

Завдання для самоперевірки

1. Наведіть структуру програмно доступних регістрів мікроконтролера.
2. Вкажіть призначення та склад регістрів введення-виведення мікроконтролера.
3. Які регістри мікроконтролера використовуються як вказівники адреси оперативної пам’яті?
4. Вкажіть призначення регістра статусу мікроконтролера SREG та сформулюйте умови модифікації його окремих бітів.
5. Що дає програмне керування коефіцієнтом ділення частоти тактового генератора мікроконтролера?
6. Які команди мікроконтролера модифікують стан регістра статусу мікроконтролера?
7. Яким чином можна класифікувати команди мікроконтролера по групах?

3 ВИРАЗИ ТА ДИРЕКТИВИ МОВИ ПРОГРАМУВАННЯ АСЕМБЛЕР

3.1 Вирази

Асемблер дозволяє використовувати в тексті програми вирази. Вони можуть містити операнди, операції і функції [3].

Операнди

Можуть бути використані такі операнди:

- визначені програмістом мітки, що відмічають певне місце програми;
- змінні, значення яких на початку програми визначені за допомогою директиви SET;
- константи, визначені за допомогою директиви EQU;
- цілі константи:
  - десяткові (за замовчуванням): 10, 255,
  - шістнадцятіркові (різні види запису):0x0a, $0a, 0ah, 0xff, $ff, 0ffh,

- двійкові: 0b00001010, 0b11111111;
- коди символів ASCII: 'A', 'a';
- рядки ASCII (без нуль-термінатора в кінці рядка): “String”;
- PC - поточне значення лічильника команд в пам'яті програм.

Функції

low(вираз) - повертає молодший байт виразу;

high(вираз) - повертає старший байт виразу;

exp2(вираз) - повертає значення цілої частини 2^(вираз);

log2(вираз) - повертає значення цілої частини log2(вираз);

Операції

Асемблер підтримує різні оператори, описані нижче. При їх використанні можна застосовувати дужки.

Логічне заперечення, позначення: !

Опис: унарний оператор, повертає 1, якщо вираз рівний нулю і 0, якщо вираз був не рівний нулю.

Пріоритет: 14. Приклад:

```
ldi    r16, ! 0xf 0;     завантажити в r16    0x00.
```

Побітова інверсія, позначення: ~

Опис: унарний оператор, який повертає побітно інвертований початковий вираз.

Пріоритет: 14. Приклад:

```
 ldi    r16,~0xf0;      завантажити в r16  0x0f
```

*Унарний мінус, п*означення: -

Опис: повертає число із знаком, зміненим на протилежний.

Множення, позначення: *

Опис: повертає результат множення двох чисел.

Пріоритет: 13. Приклад:

```
  ldi  r30,label*2 ;  завантажити в регістр   r16   label*2
```

Ділення, позначення: /

Опис: повертає цілу частину від ділення лівого операнда на правий.

Пріоритет: 13. Приклад:

```
ldi r30, label/2 ;  завантажити в регістр r30   label/2
```

Додавання, позначення: +

Опис: повертає суму двох чисел.

Пріоритет: 12. Приклад:

```
ldi        r30, c1+c2    ; завантажити в регістр r30  c1+c2
```

Віднімання, позначення: -

Опис: повертає результат віднімання правого числа від лівого.

Пріоритет: 12. Приклад:

```
ldi r17,c1-c2;  завантажити в регістр r17 c1-c2
```

Зсув ліворуч, позначення: <<

Опис: повертає значення лівого числа, зсунуте ліворуч на число розрядів, рівне правому числу.

Пріоритет: 11. Приклад:

```
 ldi     r17,1<<3 ;  завантажує в регістр r17 число 8
```

Зсув праворуч, позначення: >>

Опис: повертає значення лівого числа, зсунуте праворуч на число розрядів, рівне правому числу.

Пріоритет: 11. Приклад:

```
      ldi r17,32>>2 ;  завантажує в регістр r17 число 8
```

Менше, позначення: <

Опис: повертає 1, якщо перше число менше другого, інакше 0.

Пріоритет 10. Приклад:

```
   ldi r18,(c1 <  c2);
```

Менше або дорівнює, позначення: <=

Опис: повертає 1, якщо перше число менше або рівне другому, інакше 0.

Пріоритет 10. Приклад:

```
   ldi r18,(c1 <=  c2);
```

Більше, позначення: >

Опис: повертає 1, якщо перше число більше другого, інакше 0.

Пріоритет 10. Приклад:

```
   ldi r18,(c1 >  c2);
```

Більше або дорівнює, позначення: >=

Опис: повертає 1, якщо перше число більше або рівне другому, інакше 0.
Пріоритет 10. Приклад:

```
ldi r18,(c1 >= c2);
```

Дорівнює, позначення: = =

Опис: повертає 1, якщо перше число рівне другому, інакше 0.
Пріоритет: 9. Приклад:

```
ldi     r16, (c1 = = c2);
```

Не дорівнює, позначення: ! =

Опис: повертає 1, якщо перше число не рівне другому, інакше 0.
Пріоритет: 9. Приклад:

```
ldi     r16, (c1 != c2);
```

Побітова кон'юнкція, позначення: &

Опис: повертає результат побітової операції «І» між операндами.
Пріоритет: 8. Приклад:

```
ldi r18, (c1 & c2);
```

Побітова диз'юнкція, позначення: |

Опис: повертає результат побітової операції «АБО» між операндами.
Пріоритет: 7. Приклад:

```
ldi r18, (c1 | c2);
```

Побітове виключальне АБО (нерівнозначність), позначення: ^

Опис: повертає результат побітової операції «виключальне АБО» між операндами.
Пріоритет: 7. Приклад:

```
ldi r18, (c1 ^ c2);
```

Логічне І, Позначення: &&

Опис: повертає 1, якщо результат логічного множення двох операндів не дорівнює нулю, інакше - 0.

Пріоритет: 5. Приклад:

```
ldi r18,(c1 && c2);
```

Логічне АБО, Позначення: ||

Опис: повертає 1, якщо результат логічного додавання двох операндів не дорівнює нулю, інакше - 0.

Пріоритет: 4. Приклад:

```
ldi r18,(c1 || c2);
```

Питання для самоперевірки

1. Яке значення буде завантажено в регістр командою

```
ldi  r16,~(0x66^0xcd)?
```

2. Яке значення буде завантажено в регістр командою

```
ldi  r17,~(1<<4)?
```

3. Яке значення буде завантажено в регістр командою

```
ldi  r18,(0x12  &&  0x24)?
```

4. Яке значення буде завантажено в регістр командою

```
ldi  r19,(0x21 |  12)?
```

3.2 Директиви

При написанні програм на асемблері транслятор допускає використання директив – указання транслятору на виконання певних дій.

Нижче наведений перелік директив транслятора асемблера.

BYTE - резервує 1 байт для змінної;

CSEG - сегмент програми;

DB - визначає байт-константу;

DEF - визначає символічне ім'я для регістра;

DEVICE - задає тип мікроконтролера в сімействі;

DSEG - сегмент даних;

DW - визначає слово-константу;

MACRO - визначення макросу;

ENDMACRO - кінець визначення макросу;

EQU – ставить у відповідність символьному імені арифметичний вираз;

ESEG - сегмент EEPROM;

EXIT - вийти з файлу (кінець тексту програми);

INCLUDE - завантажити текст з іншого файлу;

LIST - включити генерацію лістингу;

LISTMAC - включити зміст макросів до лістингу;

NOLIST - вимкнути генерацію лістингу;

ORG - встановити розташування;

SET – поставити у відповідність символу вираз.

Всі директиви повинні починатися з крапки.

.BYTE - резервує місце (або декілька місць) розміром 1 байт для змінної.

Директива BYTE резервує один байт в пам'яті SRAM для змінної. Для того, щоб мати змогу звертатись до цієї змінної, перед директивою BYTE повинна стояти мітка. Директива має один параметр - кількість байтів для резервування. Директива може використовуватися тільки для резервування місця в пам'яті даних. Синтаксис:

Мітка: .BYTE числове значення;

Приклад:

```
varl:   .BYTE 1          ;резервуємо 1 байт для змінної
table.  .BYTE  tab_size ; резервуємо tab_size байтів

.CSEG
  Idi  r30,low(var1)      ;Завантажуємо молодший байт Z
  Idi  r31,high(var1)     ;Завантажуємо старший байт Z
  Id   r1,Z               ; Завантажити вміст змінної var1 в r 1
```

.CSEG - сегмент коду.

Директива CSEG визначає початок сегменту коду (програм). У тексті програми може бути декілька сегментів коду. Транслятор асемблера в процесі компіляції програми об'єднує всі сегменти коду в один. Директива BYTE не може бути використана в сегменті коду. Якщо в програмі немає явного указання назви сегменту, за замовчуванням вважається, що це сегмент коду. Директива CSEG не має параметрів. Сегмент коду має свій лічильник слів. Директива .ORG може бути використана для розміщення коду або констант у визначеному програмістом місці пам'яті програм. Синтаксис:

.CSEG

Приклад:

```
.DSEG                    ; Початок сегменту даних
```

```
vartab: .BYTE 4    ; Резервуємо 4 байти в SRAM
,CSEG              ; Початок сегменту коду
  const:  .DW 2    ; Число 0x0002 в пам'ять програм
   mov r1,r0       ; Що-небудь зробимо
```

.DB - визначити байти-константи в пам'яті.

Директива DB резервує місце в пам'яті програми або EEPROM. Для того, щоб мати змогу звертатись до зарезервованих констант, перед цією директивою слід ставити мітку. Директива DB може бути розташована тільки в сегменті коду або EEPROM. Параметрами директиви DB є список виразів.

Список виразів - один або декілька виразів, розділених комами. Кожний вираз може бути числом від -128 до 255. Якщо вираз є від'ємним числом, він буде розміщений в пам'яті програми або EEPROM у додатковому коді.

Якщо директива DB розташована в сегменті коду і має більше одного виразу в списку параметрів, вирази пакуються таким чином, що два байти розташовуються в одному слові пам'яті програми. Якщо число виразів непарне, останній вираз буде поміщений в окреме слово пам'яті програми, навіть якщо після директиви DB розташована ще одна директива DB. Синтаксис:

Мітка: .DB список виразів

Приклад:

```
.CSEG
  const:   .DB   0, 255, 0b01011100, -128, 0xaa
.ESEG
  EEconst: .DB   0xf f
```

.DEF - призначити регістру символьне ім'я**.**

Директива DEF дозволяє призначити регістру символьне ім'я, що дозволяє зробити програму набагато зрозумілішою. Можна призначити одному регістру декілька символьних імен. Символьне ім'я регістру може бути перевизначене в тексті програми. Синтаксис:

.DEF символьне ім'я = регістр

Приклад:

```
.DEF temp = r16
.DEF ior = r0
```

```
.CSEG
ldi     temp,0xf0; Завантажити в регістр temp число 0xf0
in      ior,SREG; Прочитати в ior вміс  SREG
```

.DEVICE - визначає тип мікроконтролера сімейства.

Директива DEVICE дозволяє програмісту вказати, на якому мікроконтролері виконувати програму. Якщо в тексті програми вказана ця директива, транслятор асемблера перевірятиме текст програми на наявність недійсних операцій, які не підтримуються вибраним мікроконтролером. При спробі використання більшого розміру SRAM або EEPROM пам'яті, ніж у вибраного мікроконтролера, також буде видано попередження. Якщо директива DEVICE відсутня в тексті програми, дозволені всі команди сімейства мікроконтролерів AVR, а розміри пам'яті не перевіряються. Синтаксис:

```
.DEVICE AT90S1200 | AT9OS2313 | AT90S2323 | AT90S2343 |
        AT90S4414 | AT90S8515 | ATMEGA128
```

Примітка. З'являються нові моделі мікроконтролерів, тому при необхідності використовувати новіший мікроконтролер слід самостійно відстежувати дозволені інструкції в тексті програми (відповідно не застосовуючи директиву DEVICE) або скористатися оновленою версією транслятора асемблера.

Приклад:

```
.DEVICE AT90S1200; Використовується мікроконтролер AT90S1200
.CSEG
  push   r30   ; Цей запис викликає повідомлення про те, що
               ; вибраний мікроконтролер не підтримує цю
               ; інструкцію
```

.DSEG - сегмент даних.

Директива DSEG визначає початок сегмента даних. У початковому тексті програми на асемблері може бути декілька сегментів даних. В процесі трансляції всі вони будуть об'єднані в один. Звичайно сегмент даних містить тільки директиви BYTE з мітками. Сегмент даних має свій лічильник байтів. Директива ORG може бути використана для розташування змінних в конкретних місцях SRAM. Директива DSEG не має параметрів.

Синтаксис:

. DSEG

Приклад:

```
.DSEG                          ; Початок сегмента даних
   var1:     .BYTE  1          ; Резервуємо 1 байт для змінної var1
   table:    .BYTE  size       ; Резервуємо size байт

.CSEG
   ldi  r30,low(var1)          ;  Завантажуємо молодший байт Z-регістру
   ldi  r31, high(var1)        ;  Завантажуємо старший байт Z-регістру
   Id   r1,Z                   ;  Завантажити вміст змінної var1   в  r1
```

.DW - визначення слів-констант в пам'яті програми або EEPROM.

Директива DW резервує місце в пам'яті програми або EEPROM. Для того, щоб мати змогу звертатись до зарезервованого простору, перед цією директивою слід ставити мітку. Директива DW повинна бути розташована в сегменті коду або EEPROM. Параметрами директиви DW є список виразів.

Список виразів - це один або декілька виразів, розділені комами. Кожний вираз може бути рівний числу від -32768 до 65535. Якщо вираз є від'ємним числом, його буде поміщено в пам'ять програми або EEPROM в додатковому коді.

Синтаксис:

Мітка: .DW список виразів

Приклад:

```
.CSEG
varlist:  .DW   0, 56255, 0b0101110011101011,  -12128, 0xaaff
.ESEG
EEvar:    .DW    0x7766
```

. **MACRO** - початок визначення макрокоманди.

Директива MACRO вказує транслятору асемблера на початок визначення макрокоманди. Параметром директиви MACRO є ім'я визначеної макрокоманди. Надалі при виявленні в тексті програми імені макрокоманди транслятор асемблера замінюватиме це ім'я на текст макрокоманди. Макрокоманда може мати до 10 параметрів. Ці параметри мають фіксовані імена: @0...@9. При виклику макрокоманди параметри повинні бути у вигляді списку, розділені комами. Визначення макрокоманди завершується директивою ENDMACRO.

При визначенні нової макрокоманди не можна використовувати інші

макрокоманди (забороняється використовувати вкладені макрокоманди).

Макрокоманда повинна бути визначена в тексті програми до того, як її використовують.

За замовчуванням в лістингу генерується тільки виклик макрокоманди. Щоб одержати в лістингу текст макрокоманд, слід використовувати директиву LISTMAC. Текст макрокоманди в лістингу помічено символом «+».

Синтаксис: .MACRO

Приклад:

```
 MACRO sub16              ; Початок визначення макрокоманди
    subi    r16,low(@0)   ; Віднімаємо молодший байт
    sbci    r17,high(@0)  ; Віднімаємо старший байт
. ENDMACRO                ; Кінець визначення макрокоманди

.CSEG                     ; Початок сегмента коду
   sub16      0x4321      ; Відняти  0x4321 з r17 та r16
```

Примітка:

r17, r16 в даному випадку - пара регістрів, що містить 16-розрядне число.

.ENDMACRO - кінець опису макрокоманди.

.EQU - присвоїти символьному позначенню вираз.

Директива EQU присвоює символьному позначенню значення виразу. Надалі це символьне позначення може бути використане у виразах. Присвоєне значення - константа. У тексті програми цей символьний вираз не може бути перевизначений або змінений.

Синтаксис:

.EQU Символьний вираз = вираз

Приклад:

```
.EOU offset = 0x23
 .EQU porta = offset + 2
.CSEG                     ; Початок сегмента коду
   clr r2                 ; Очистити регістр r 2
   out porta,r2           ; Записати в порт A
```

.ESEG

Директива ESEG визначає початок сегмента EEPROM. У тексті програми може бути декілька сегментів EEPROM. Транслятор асемблера в процесі

компіляції програми об'єднує всі сегменти EEPROM в один. Директива BYTE не може бути використана в сегменті EEPROM. Директива ESEG не має ніяких параметрів. Сегмент EEPROM має свій лічильник байтів. Директива .ORG може бути використана для розміщення коду або констант у визначеному програмістом місці пам'яті EEPROM.

Синтаксис:

.ESEG

Приклад:

```
        .DSEG           ; Початок сегмента даних
vartab: .BYTE   4       ; Резервуємо 4 байти в SRAM
        .ESEG
Eevar:  .DW Oxff67      ;  Ініціалізуємо одне слово в EPROM
        .CSEG           ; Початок сегмента коду
const:  .DW 2           ;Число 0x02 в пам'яті програми
    mov     r1,r0       ; Що-небудь зробимо
```

.EXIT- кінець тексту програми.

Директива EXIT вказує транслятору, що слід завершити трансляцію програми. За відсутності такої директиви транслятор асемблера працює до тих пір, поки початковий файл не завершиться (EOF). Якщо директива EXIT зустрічається у файлі, який підключається в текст директивою INCLUDE, транслятор асемблера продовжить роботу з рядка, наступного після відповідної директиви INCLUDE.

Синтаксис:

.EXIT

.INCLUDE - підключити файл до проекту.

Директива INCLUDE вказує транслятору асемблера на необхідність вставити в текст програми інший файл. Реально при обробці цієї директиви транслюється файл, вказаний в директиві INCLUDE, після завершення його обробки (досягши кінця файлу або директиви .EXIT) продовжується обробка основного файлу. Вкладені файли, в свою чергу, можуть мати директиви INCLUDE.

Синтаксис:

.INCIUDE “ім'я файлу"

Приклад:

```
.INCLUDE    “spi.h"   ; включити в проект для трансляції файл spi.h
```

.LIST - включити генерацію лістингу.

Директива LIST включає генерацію лістингу. Асемблер генерує лістинг, що містить текст на асемблері, адреси і коди операцій. За замовчуванням генерація лістингу включена. У комбінації з директивою NOLIST можна організувати лише друк потрібних фрагментів програми.

Синтаксис:

```
.LIST
```

Приклад:

```
.NOLIST                 ;       Відключити генерацію лістингу
INCLUDE "macro.inc"     ;       Файли, що включаються, не будуть
INCLUDE "const.def "    ;       показані в лістингу
.LIST                   ;       Включити генерацію лістингу
```

.LISTMAC - включити розкриття макрокоманд.

Директива LISTMAC вказує транслятору асемблера на необхідність показу в лістингу вмісту макрокоманд. За замовчуванням в лістингу приводиться лише назва макрокоманди.

Синтаксис:

```
.LISTMAC
```

ORG - установлення значення лічильника розташування.

Директива ORG встановлює абсолютне значення лічильника розташування. Параметром директиви є значення, яке повинно бути присвоєно лічильнику. При використанні директиви ORG в сегменті даних буде визначено значення розташування в оперативній пам'яті SRAM. Директивою ORG в сегменті коду буде визначено значення програмного лічильника. Директива ORG в сегменті EEPROM буде визначати значення, розташуване в пам'яті EEPROM.

Якщо перед директивою розташована мітка (на цьому ж рядку), мітка набуде значення параметра директиви. Значення за замовчуванням для сегмента коду і EEPROM рівне 0, а для SRAM - 32 (оскільки регістри займають простір з 0 до 31). Необхідно підкреслити, що для EEPROM і SRAM лічильник розташування вказує на байти, тоді як в пам'яті програм – на слова.

Синтаксис:

.ORG вираз

Приклад:

```
.DSEG                    ; Початок сегмента даних (SRAM)
.ORG 0x37                ; Встановити адресу SRAM
  variable:   .BYTE 1    ; Зарезервувати 1 байт за адресою 0x37 SRAM

.ESEG                    ; Початок сегмента EEPROM
.ORG 0x20                ; Встановити значення лічильника розташування
  eevar:   .DW 0xf77a    ; Ініціалізувати слово в пам'яті EEPROM

  .CSEG
  .ORG     0x10          ; Встановити лічильник на значення 0x10
     mov     r18,r1       ; Ця команда буде розташована в пам'яті
                          ; програм за адресою 0x10
```

.SET - присвоїти символьному позначенню вираз.

Директива SET присвоює символьному позначенню значення виразу. В подальшому таке символьне позначення може бути використане у виразах. У тексті програми символьне значення виразу може бути змінено.

Синтаксис:

.SET символьне позначення = вираз

Приклад:

```
.SET   offset = 0x66
.CSEG        ; початок сегмента коду
sts  offset,r2; запис в пряму адресу
```

3.3 Створення та налагодження програм з використанням AVR Studio

Система **AVR Studio** включає редактор тексту, компілятор із редактором зв'язків, бібліотеку стандартних функцій і символічний налагоджувач. Такі системи дозволяють різко скоротити витрати часу на створення і корекцію програмного забезпечення, користуючись єдиним інтегрованим середовищем створення програм.

Створення першої програми в AVR Studio 3.55

1. Запустіть програму **AVR Studio 3.55**.
2. Для створення нового проекту виберіть у головному меню **Project - New...** . В вікні, що з’явилося, в рядку **Project name** введіть ім'я нового проекту, наприклад “new”. Далі, натиснувши лівою кнопкою мишки на

рядку **AVR Assembler,** виберіть тип проекту (рисунок 3.1) . Натисніть на кнопці **OK**.

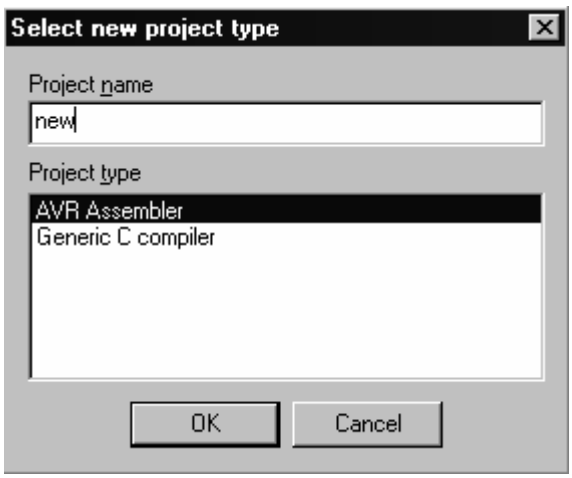


Рисунок 3.1

3. В вікні, що утворилось, натисніть правою кнопкою мишки на рядку **Assembler Files**, в меню, що з'явилося (рисунок 3.2) натисніть лівою кнопкою мишки на рядку **Add File...** .

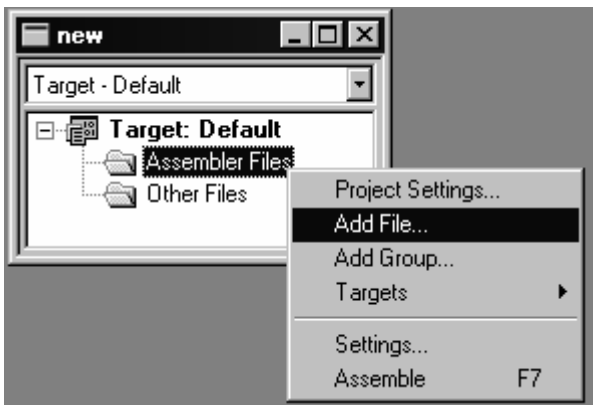


Рисунок 3.2

4. З'явиться вікно введення імені файлу майбутньої програми, якому необхідно надати розширення **.asm**. Таким чином, буде створений пустий файл робочої програми. Якщо у вас вже існує файл програми (з розширенням **.asm**) потрібно вибрати цей файл.

5. Для того, щоб зробити створений файл основним, необхідно натиснути правою кнопкою мишки на ім’я створеного файлу в вікні проекту, а потім в меню, що з'явилося, натисніть лівою кнопкою мишки на рядку **Assembler entry file** (рисунок 3.3). Тепер необхідно відкрити створений файл, двічі натиснувши на ньому лівою кнопкою мишки у вікні проекту. В вікні, що з'явилось можна набирати текст програми.

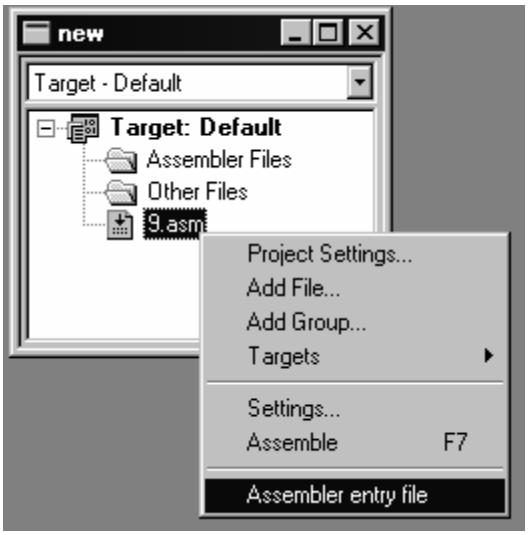


Рисунок 3.3

6. Після введення тексту програми необхідно натиснути клавішу **F7,** щоб провести компіляцію програми. Після цього з'являється вікно (рисунок 3.4), в якому міститься результат компіляції програми. При наявності помилок повідомляється про всі рядки, в яких виявлено помилки. Натиснувши на рядок, в якому повідомляється про помилку, ми потрапимо в робочій програмі в те місце, де виявлено помилку.

При завершенні компіляції без помилок, тобто в вікні (рисунок 3.4) нижній рядок буде мати вигляд **Assembly complete with no errors**, можна приступити до наочного перегляду (налагодження) роботи програми.

7. Натиснувши клавішу **F11**, почнемо процес налагодження. Для перегляду результатів роботи програми із меню **View** (Вид) викличемо вікна **Registers (**Регістри), **Processor** (Процесор), **New Memory View** (Новий вид пам'яті) . Вікно **Registers** відображає поточний вміст регістрів загального призначення **R0-R31**. Вікно **Processor** відображає поточне значення регістрових пар (X, Y, Z), значення програмного лічильника, значення вказівника стека та всіх восьми прапорців регістру стану програми.

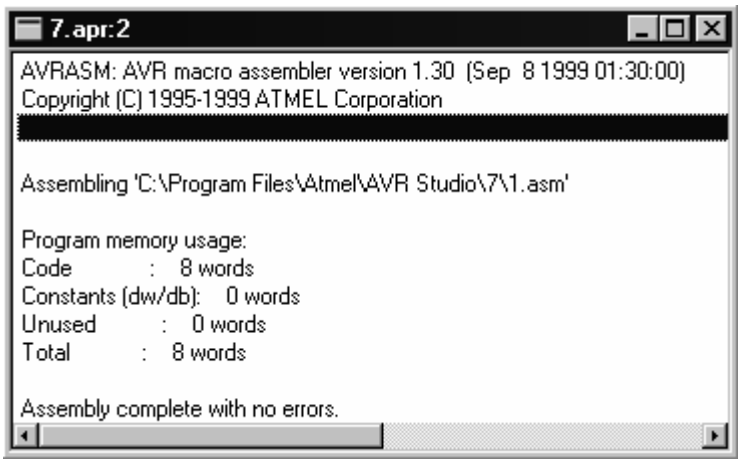


Рисунок 3.4

Вікно **New Memory View**, залежно від обраного режиму, відображає поточний зміст пам'яті даних, пам'яті програм, пам'яті введення-виведення, пам'яті EEPROM. Робоче вікно програми набуде вигляду (рисунок 3.5):

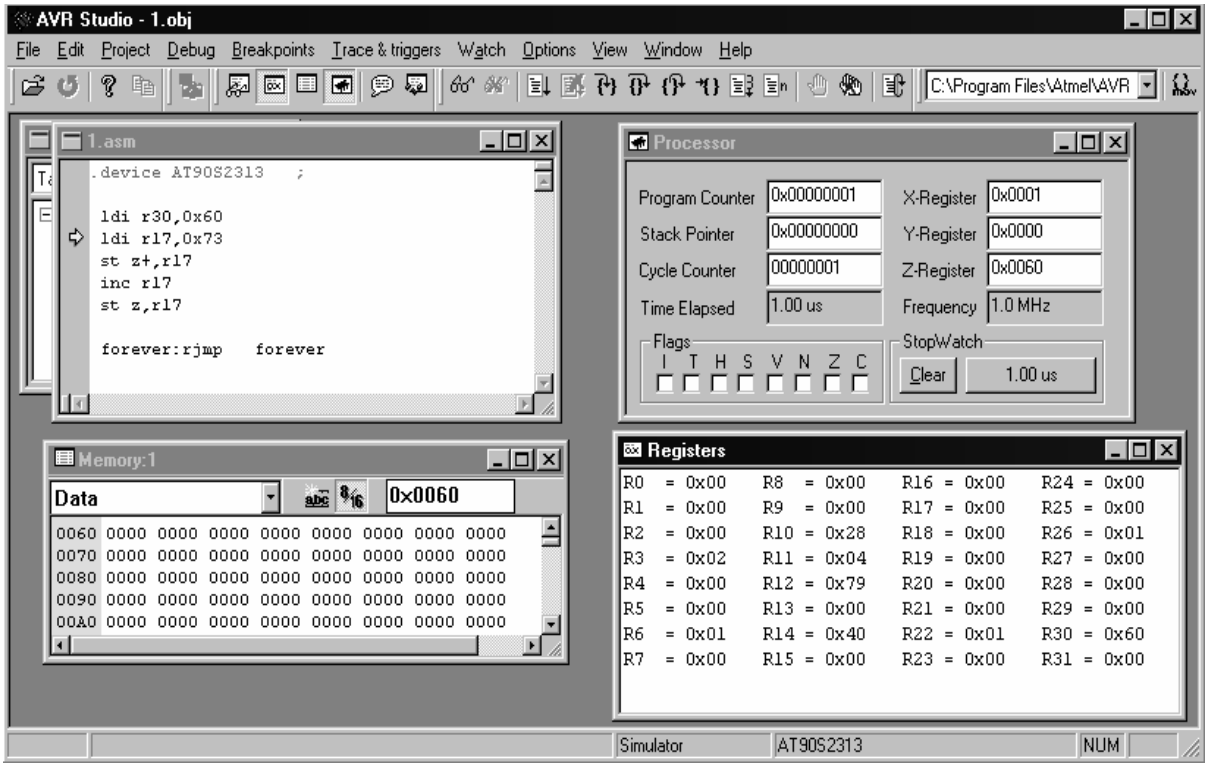


Рисунок 3.5

8. Тепер, натискаючи клавішу **F11 (Trace into)**, можна прослідкувати зміни у вікнах. Кожне натискання клавіші **F11** приводить до виконання поточних команд.

9. Компілятор працює з вихідними файлами, що містять інструкції, мітки і директиви. Інструкції і директиви, як правило, мають один чи декілька операндів.

Рядок програми не повинен бути довшим 120 символів. Будь-який рядок може починатися з мітки, що є набором символів та закінчується двокрапкою. Мітки використовуються для фіксації місця, у яке передається керування при переходах, а також для опису імен змінних.

Вхідний рядок може мати одну з чотирьох форм:

– [мітка:] директива [операнди] [Коментар];

– [мітка:] інструкція [операнди] [Коментар];

– Коментар;

– Порожній рядок.

Коментар має таку форму:

команда; [коментар]

Позиції в квадратних дужках необов'язкові. Текст після крапки з комою (;) і до кінця рядка ігнорується компілятором.

Питання для самоперевірки

1. Чим використання директиви .MACRO відрізняється від виклику підпрограми?
2. Які дії виконує директива .ORG та в яких випадках її доцільно використовувати?
3. Які дії виконує директива .BYTE?
4. Які дії виконує директива .DEF?

4 КОМПІЛЯТОРИ МОВИ ПРОГРАМУВАННЯ С++

Одними з найбільш популярних компіляторів Сі для сімейства мікроконтролерів AVR є компілятори IAR Embedded Workbench (http://www.iar.com) фірми "IAR Systems" (Швеція) чи компілятор CodeVision AVR (Румунія) фірми "HP Info Tech" (http://www.hpinfotech.com), оскільки останній розповсюджується як “freeware”.

CodeVision включає компілятор С відповідно до стандарту ANSI, графічну оболонку та автоматичний генератор програм для мікроконтролерів AVR сімейства [5].

Об’єктні файли компілятора дозволяють здійснювати налагодження та прогляд вмісту змінних за допомогою налагоджувача AVR Studio, починаючи з версії 3.55.

В складі компонентів CodeVision AVR присутня дуже корисна термінальна програма Terminal, яка дозволяє тестувати програми, що використовують передачу даних по послідовній лінії.

Компілятор CodeVision AVR може генерувати шаблони програм для роботи з зовнішніми компонентами, які можуть бути вказані як опції при створенні проекту програми для CodeVision AVR, а саме:

- стандартний рідкокристалічний індикатор із вбудованим контролером,
- передача даних по послідовній двопроводній шині I2C,
- робота з сенсором температури LM75 фірмы National Semiconductor,
- робота з годинниками реального часу PCF8583 фірми Philips, DS1302 чи DS1307 фірми Dallas Semiconductor,
- робота з однопровідним інтерфейсом фірми Dallas Semiconductor,
- робота з сенсорами температури DS1820 чи DS1822 фірми National Semiconductor,
- робота з сенсором температури/термостатом DS1621 фірми National Semiconductor,
- робота з енергонезалежною пам'яттю EEPROM DS2430, DS2433 фірмы National Semiconductor,

- передача даних по послідовній синхронній трипровідній шині SPI,
- керування режимами зниженого енергоспоживання ,
- формування програмних модулів для реалізації затримок в програмі.

Компілятор CodeVision AVR дозволяє автоматично генерувати програми для виконання таких функцій:

- ініціалізація портів введення-виведення,
- ініціалізація зовнішніх переривань,
- ініціалізація доступу до зовнішньої пам'яті,
- визначення джерела переривання від сигнала RESET,
- ініціалізація таймерів-лічильників,
- ініціалізація сторожового таймера,
- ініціалізація блоку послідовного обміну даними UART,
- ініціалізація аналогового компаратора.

На рисунку 4.1 наведено фрагмент екрана, який дозволяє налаштувати опції програмного проекту, що дає змогу отримати відповідній шаблон програми для роботи, наприклад, з послідовним асинхронним інтерфейсом UART, послідовним інтерфейсом I2C, стандартним рідкокристалічним LCD індикатором, аналоговим компаратором, таймерами/лічильниками, портами введення-виведення, організувати переривання програми як від зовнішніх так і від внутрішніх джерел запиту переривання.

Рисунок 4.1 Налаштування опцій програмного проекту для отримання відповідного шаблону програми

Фрагмент екрана, який ілюструє створений шаблон програми з вибором опції таймера-лічильника при створенні проекту наведено на рисунку 4.2.

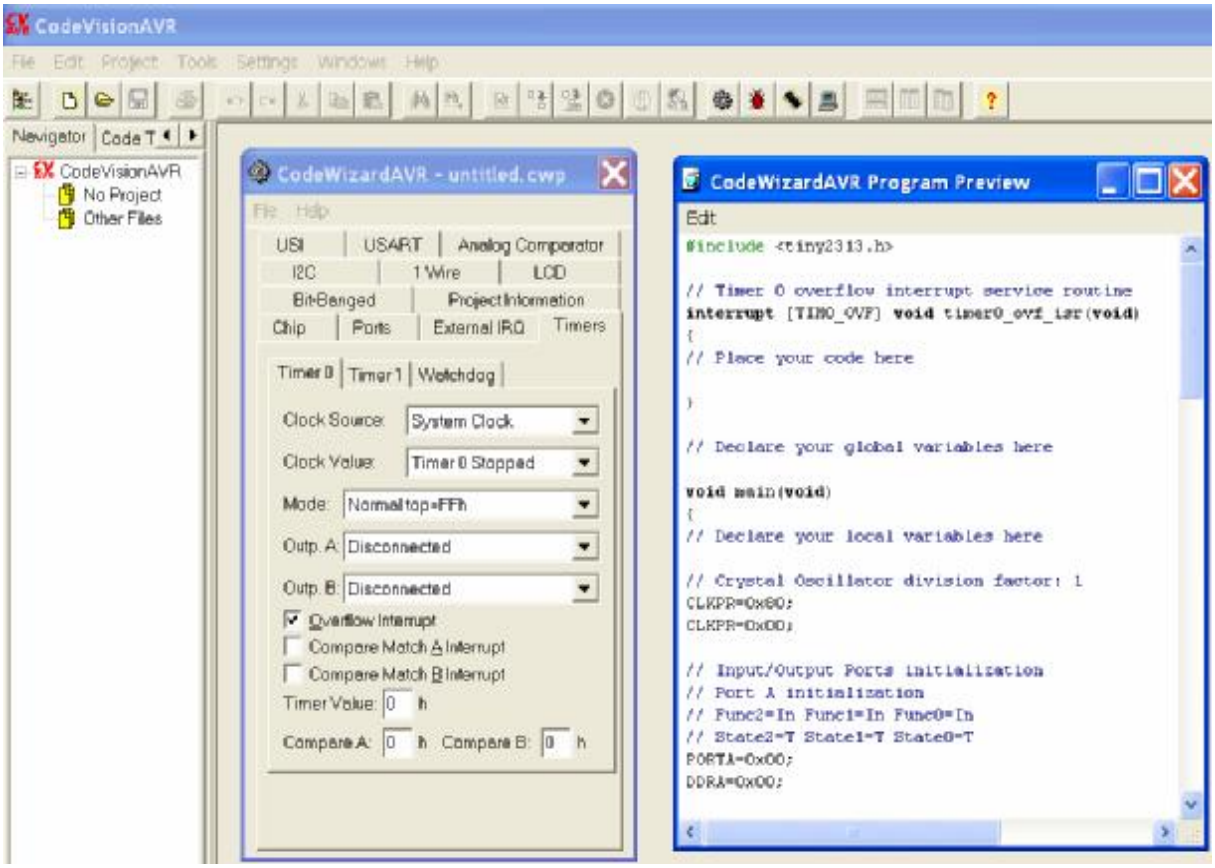


Рисунок 4.2 Шаблон екрана для роботи з таймером-лічильником

Для прикладу розглянемо просту програму, створену в CodeVision, яка вмикає та вимикає світлодіод з періодом близько однієї секунди, який підключено до лінії 7 порту B. Текст програми має такий вигляд:

```
/*****************************************************
Author  : Freeware, for evaluation and non-commercial use only
Company :
Comments:
Chip type           : AT90S2313
Clock frequency     : 4.000000 MHz
Memory model        : Tiny
External SRAM size  : 0
Data Stack size     : 32
*****************************************************/

#include <90s2313.h>
```

```
 unsigned int  count_delay=0;
 unsigned char delay_flag=0;

// Timer 0 overflow interrupt service routine
//  Fov=500 000 Hz/256 = 1953 Hz   Tov=0.512 msec
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
 {
  count_delay++;
 }

void delay_msec(unsigned int msec)  //return 1 if yes delay, return 0 if no
 {
    count_delay=0;
    while(count_delay < 2*msec) #asm("nop");
 }


void main(void)
{
// Input/Output Ports initialization
// Port B initialization
PORTB=0x00;
DDRB |= (1<<7);                                 //PORTB,7 output

// Port D initialization
PORTD=0x00;
DDRD=0x00;
// Timer/Counter 0 initialization
// Clock source: System Clock
// Clock value:
TCCR0=0x02;                                      //Fclk= 4 000 kHz/8 = 500 kHz
TCNT0=0x00;


// External Interrupt(s) initialization
// INT0: Off
// INT1: Off
GIMSK=0x00;
MCUCR=0x00;

// Timer(s)/Counter(s) Interrupt(s) initialization
TIMSK=0x02;

// Analog Comparator initialization
```

```
// Analog Comparator: Off
// Analog Comparator Input Capture by Timer/Counter 1: Off
ACSR=0x80;

// Global enable interrupts
#asm("sei")

while (1)
    {
      PORTB &= ~(1<<7);          // "0" on PORTB,7 -  “on”
      delay_msec(1000);
      PORTB |= (1<<7);           // "1" on PORTB,7  -  “off”
      delay_msec(1000);

    };

}    //end main
```

Питання для самоперевірки

1. Які можливі шаблони програм генерує компілятор CodeVision AVR?

2. Які функції може виконувати термінальна програма “Terminal” компілятора CodeVision AVR?

3. В чому різниця між С компілятором фірми IAR Systems та CodeVision AVR?

5 ПРИКЛАДИ ПРОГРАМ РОБОТИ ОКРЕМИХ БЛОКІВ МІКРОКОНТРОЛЕРА

5.1 Модифікація окремих бітів комірок оперативної пам'яті чи змінних, які розміщуються в оперативній пам'яті

Для модифікації окремих бітів комірки пам’яті чи змінної можна використовувати логічні операції з константою для накладання маски на комірку пам’яті чи змінну.

Наприклад, потрібно очистити третій біт комірки пам'яті з адресою 0x60, а потім встановити цей самий біт не модифікуючи інших бітів цієї комірки пам'яті.

1) При використанні мови асемблера фрагмент програми може мати такий вигляд:

a) Модифікація біта з використанням логічної операції та маски.

```
ldi      r30,0x60;
```

```
ldi     r31,0;              set   Z=0x60
ld       r16,Z;
andi     r16,0xf7;           clear bit 3
st       Z,r16;              store date in ram,  adr=0x60
ld       r16,Z;
ori      r16,0x08;           set  bit 3
st       Z,r16;              store  date in ram, adr=0x60
```

б) Модифікація біта з використанням логічної операції та операторів зсуву, які підтримує асемблер.

```
ldi     r30,0x60;
ldi     r31,0;              set   Z=0x60
ld       r16,Z;
andi     r16, ~(1<<3);      clear bit 3
st       Z,r16;              store date in ram,  adr=0x60
ld       r16,Z;
ori      r16,(1<<3);         set  bit 3
st       Z,r16;              store  date in ram, adr=0x60
```

2) При використанні мови Сі та маски фрагмент програми може мати такий вигляд.

```
unsigned char    var;
var &= 0xf7;                    //clear bit 3
 var  |=  0x08;                 //set     bit 3
```

Але краще користуватись логічними операціями та операторами зсуву, що дає змогу символічно описувати біт, який має бути модифіковано:

```
unsigned char    var;
unsigned char    bit=3;
    var  &= ~(1 <<  bit );          //clear bit 3
    var  |=     (1 << bit3);        //set   bit 3
```

5.2 Реалізація програмних затримок при виконанні програми мікроконтролера

Для реалізації затримок в програмі можна використовувати два підходи - програмний та апаратний.

Програмний підхід передбачає використання в якості затримки час виконання певного числа команд мікроконтролера, оскільки час виконання кожної команди відомий. Недоліком такого підходу є неможлиівсть виконання мікроконтролером під час програмної затримки програми основної задачі, а також залежність величини затримки від частоти роботи його тактового генератора.

Наприклад, реалізація програмної затримки на асемблері може мати такий вигляд.

```
                        //for Fosc=5 Mhz, call= 4 clock
delay_5mksec:
          push      r21;          2   clock
           ldi      r21,8;        1    clock
l_5mksec:
          dec       r21;          1     clock
           brne     l_5mksec;  1/2  clock
           pop      r21;          2     clock
           ret;                    4     clock
              ;------------------------

delay_1msec:
          push     r21
          ldi      r21,1000/5
loop_1msec:
           call      delay_5mksec
            dec      r21
           brne      loop_100mksec
           pop       r21
             ret
             ;------------------------
```

Приклад реалізації програмної затримки на мові Сі.

```
void delay_mksec (unsigned int pause)    // pause N mksec
{
 unsigned int j;   char i;

   for (j=0; j<=pause; j++)
   for (i=0; i<1; i++) __no_operation();   //for  7 MHz
  //for (i=0; i<2; i++) __no_operation();  //for 14 MHz
}
            //-------------------------------------
void delay_msec (unsigned int pause)     // pause - N msec
{
  unsigned int i;

   for (i=0; i<=pause; i++)
    delay_mksec(1000);
}
          //-------------------------------------
```

Приклад іншої реалізації програмної затримки на мові Сі з врахуванням частоти роботи тактового генератора мікроконтролера.

```
#define F_clock  7372800  //CPU Clock

#define delay_mksec(del)
__delay_cycles((unsigned long)((double)F_clock*del/1000000))

#define delay_msec(del)
__delay_cycles((unsigned long)((double)F_clock*del/1000))
//-------------------------------------
```

Апаратний підхід передбачає, як правило, використання внутрішніх апаратно реалізованих блоків таймерів-лічильників.

При налаштуванні таймера-лічильника в якості лічильника зовнішній сигнал відомої частоти поступає на вхід лічильника. Тоді відрізок часу буде визначатись кількістю сигналів, які зафіксовані лічильником.

При налаштуванні таймера-лічильника в якості таймера, внутрішній сигнал мікроконтролера відомої частоти поступає на вхід лічильника. Відрізок часу буде визначатись кількістю сигналів, які будуть фіксовані лічильником.

При такому підході мікроконтролер може виконувати основну програму паралельно з роботою таймера-лічильника.

5.3 Реалізація програмного доступу до енергонезалежної пам’яті (EEPROM)

Мікроконтролери містять область енергонезалежної памяті (EEPROM), інформація в якій електрично стирається, а сама пам'ять забезпечує до 100 000 циклів перезаписування. В такій пам'яті зручно зберігати тимчасові константи, інформацію про конфигурацію мікропроцесорного пристрою.

Основні операції, які виконує така пам'ять, – це операції записування та читання. Робота з такою пам'яттю забезпечується програмним доступом до трьох керуючих регістрів – адреси (EEAR), даних (EEDR) та регістра керування (EECR).

Варіант програми на асемблері, який підтримує роботу з пам'яттю EEPROM може мати такий вигляд:

```
                    ; EEPROM_write:
EEPROM_write:
    sbic   EECR,EEWE        ; Wait for completion of previous write
    rjmp   EEPROM_write
```

```
 out    EEARH, r18          ; Set up address (r18:r17) in address register
out     EEARL, r17
                            ; Write data (r16) to data register
Out     EEDR,r16
                            ; Write logical one to EEMWE
sbi     EECR,EEMWE
                            ; Start eeprom write by setting EEWE
sbi     EECR,EEWE
ret

                ;EEPROM_read:
sbic EECR,EEWE          ; Wait for completion of previous write
rjmp EEPROM_read
                            ; Set up address (r18:r17) in address register
out EEARH, r18
out EEARL, r17
                            ; Start eeprom read by writing EERE
sbi EECR,EERE
                            ; Read data from data register
in r16,EEDR
ret
```

Варіант програми на мові Сі, який підтримує роботу з пам'яттю EEPROM може мати такий вигляд:

```
void EEPROM_write(unsigned int uiAddress, unsigned char ucData)
{
 while(EECR & (1<<EEWE));         // Wait for completion of previous write

  EEAR = uiAddress;                  // Set up address and data registers
  EEDR = ucData;
  EECR |= (1<<EEMWE);                // Write logical one to EEMWE
  EECR |= (1<<EEWE);                 // Start eeprom write by setting   EEWE
}

unsigned char EEPROM_read(unsigned int uiAddress)
{
 while(EECR & (1<<EEWE));         //Wait for completion of previous write

 EEAR = uiAddress;                   //  Set up address register
 EECR |= (1<<EERE);                  // Start eeprom read by writing EERE
   return EEDR;                      // Return data from data register
}
```

5.4 Реалізація програмного доступу до портів мікроконтролера

Кожному двонаправленому порту X введення-виведення поставлено у відповідність три регістри - регістр напрямку DDRX, регістр даних PORTX та регістр логічного стану зовнішніх ліній порту PINX.

Тому робота з портами введення-виведення зводиться до програмного доступу до регістрів порту.

Приклад програми на асемблері, яка встановлює високий рівень на лініях 0-3 порту, налаштовує лінії 0-3 порту в режим виводу, читає стан ліній 0 -3 порту.

```
...
ldi     r16,(1<<PB7) | (1<<PB6) | (1<<PB1) | (1<<PB0)
ldi     r17,(1<<DDB3) | (1<<DDB2) | (1<<DDB1) | (1<<DDB0)
out     PORTB,r16
out     DDRB,r17
        ; Insert pause nop for synchronization
nop
        ; Read port pins
in      r16,PINB
...
```

Аналогічні дії над лініями порту, які описані на мові Сі, мають такий вигляд:

```
unsigned char i;
...
                    // Define pull-ups and set outputs high
                    // Define directions for port pins
PORTB = (1<<PB7) | (1<<PB6) | (1<<PB1) | (1<<PB0);
DDRB = (1<<DDB3) | (1<<DDB2) | (1<<DDB1) | (1<<DDB0);
                    // Insert pause for synchronization
_NOP();
                    // Read port pins
i = PINB;
...
```

Питання для самоперевірки

1. Навести фрагмент програми на асемблері, яка налаштовує порт B мікроконтролера на режим введення і очікує код 0x64 на вході порту протягом 1 секунди, а результат виводить в порт D лінії 7.

2. Навести фрагмент програми мовою Сі, яка налаштовує порт B мікроконтролера на режим введення і очікує код 0x64 на вході порту протягом 1 секунди.

3. Навести фрагмент програми на асемблері, який би виконував сортування елементів масиву в порядку зростання.

4. Навести фрагмент програми иовою Сі, який би виконував сортування елементів масиву в порядку зростання.

6 ШЛЯХИ ЗАБЕЗПЕЧЕННЯ НАДІЙНОСТІ ПРОГРАМ

Основні фактори, котрі визначають надійність функціонування програмного забезпечення (ПЗ), можна об'єднати в три групи [2]. До першої групи відносяться фактори, які безпосередньо визначають надійність програм, а саме:

а) суб’єктивні особливості користувачів програм ;

б) спотворення початкових даних - спотворення даних, що надходять від людини-оператора; даних, що надходять телекомунікаційними каналами зв'язку, даних в процесі накопичення і зберігання в пам'яті;

в) помилки в програмах та їх прояв - статистичні характеристики помилок і спотворень програм, масивів даних та обчислювального процесу.

До другої групи факторів належать методи проектування коректних програм:

а) структурне проектування програм і даних, структурне проектування програмних модулів та взаємодії модулів, структурування даних;

б) тестування програм - детерміноване та статистичне тестування, динамічне тестування і контроль пропускної здатності в реальному часі.

Третя група факторів включає методи контролю і забезпечення надійності програм:

а) методи використання часової, інформаційної та програмної надлишковості;

б) методи контролю програм, даних і обчислювального процесу; передпусковий та оперативний контроль;

в) методи програмного відновлення тестів програм, виправлення даних, коректування обчислювального процесу;

г) методи випробування на надійність - випробування в нормальних умовах експлуатації, форсовані випробування, розрахунково-експериментальні методи визначення надійності;

д) методи забезпечення надійності при супроводженні - забезпечення зберігання еталонних версій програм та забезпечення коректності внесення змін і розвитку версій.

Охарактеризуємо більш детально третю групу факторів забезпечення надійності ПЗ. Для підвищення надійності функціонування ПЗ, захисту інформації програмно-алгоритмічними методами використовують часову, інформаційну та програмну надлишковість.

Часова надлишковість полягає у використанні певної частини апаратних засобів для контролю виконання програм і відновлення обчислювального процесу. Вона реалізується шляхом передбачення певного резерву продуктивності, котрий буде використано потім для контролю та підвищення надійності функціонування ПЗ. Резерв часу використовується на контроль і виявлення спотворень, їх діагностування і прийняття рішень щодо відновлення обчислювального процесу чи інформації, а також на реалізацію операцій відновлення.

Інформаційна надлишковість полягає у дублюванні накопичених початкових та проміжних даних. Вона використовується для збереження достовірності даних, котрі потребують значного часу для відновлення чи найбільше впливають на нормальне функціонування ПЗ. Зазначені дані характеризують певні інтегральні відомості про зовнішній керований процес і в разі їх руйнування керування згаданим процесом та відповідна обробка інформації можуть надовго припинитися. Інформаційна надлишковість дає можливість не тільки виявляти спотворення даних, а й усувати помилки.

Програмна надлишковість дає можливість контролювати та забезпечувати достовірність найбільш важливих рішень з обробки інформації та керування, її суть полягає у застосуванні кількох варіантів програм, котрі відрізняються методами розв'язання певної задачі або програмною реалізацією одного і того самого методу. Це дає можливість виключати спотворення результатів, зумовлені програмними помилками чи збоями.

Часто у зв'язку з невиправданим оптимізмом розробників введенням надлишковості для забезпечення надійності ПЗ нехтують. Це знижує показники надійності програм, що розробляються.

Небажані наслідки спотворень обчислювального процесу та даних викликають необхідність контролю функціювання програм. Кожний метод контролювання орієнтований на виявлення певного типу наслідків спотворень чи їх певної сукупності. Це зумовлює застосування ієрархічних підходів, на основі котрих використовується послідовність методів, що дає можливість досягти заданої глибини пошуку спотворень в програмах.

Щодо процесу експлуатації, то контроль надійності функціонування ПЗ може проводитись на трьох етапах:

а) у неробочому режимі системи керування чи обробки інформації при проведенні регламентних профілактичних робіт (профілактичний контроль);

б) при підготовці до включення нормального робочого режиму функціонування ПЗ (передпусковий контроль);

в) в процесі розв'язання основних функціональних задач системою у нормальному робочому режимі (оперативний контроль).

Під час профілактичного контролю припиняється розв'язування задач в нормальному робочому режимі, детально аналізуються характеристики і виконується значний об'єм профілактичних і відновлювальних робіт. При

цьому аналізуються дані про зареєстровані спотворення інформації і обчислювального процесу, що накопичились при робочому функціонуванні. За цими даними та діагностичними засобами встановлюються причини спотворень, які надалі намагаються усунути. За допомогою контрольної та діагностичної апаратури і діагностичних програм джерела ненадійності можуть бути локалізовані автоматизовано. Після чого проводиться ремонт апаратури чи виправлення програм. Заміна версії програм після усунення в них помилок, забезпечення достовірності і збереження кожної версії приводять до підвищення надійності функціонування ПЗ.

На етапі передпускового функційного контролю за обмежений час здійснюється необхідний мінімум перевірок надійності функціонування ПЗ для підготовки до включення нормального режиму роботи. Визначається перелік контрольних операцій та послідовність їх проведення. Головна задача при цьому полягає у перевірці збереження програм та масивів даних, необхідних для початкового запуску системи. Передпусковий контроль може також проводитись без безпосереднього оброблення реальної інформації і видання інформації зовнішнім користувачам.

На третьому етапі у процесі розв'язання основних функціональних задач здійснюється досить глибокий оперативний контроль ПЗ і максимально швидке автоматизоване відновлення за будь-яких спотворень даних чи обчислювального процесу. Методи оперативного контролю, в свою чергу, розділяють на методи контролю стану і збереження програм у різних видах пам'яті, методи контролю динаміки проходження програм, збереження зв'язків і взаємодії модулів при робочому функціонуванні ПЗ, методи контролю стану та даних у пам'яті.

Хоча етапи контролю ПЗ розглядають як взаємодоповнюючі, але найбільший вплив на надійність мають оперативний та передпусковий контроль.

Метод оперативного відновлення ПЗ передбачає певні заходи щодо усунення наслідків спотворень. Типовими серед них є:

а) ігнорування виявленого спотворення у випадку, коли воно слабо впливає на вихідні результати та сам процес обробки інформації;

б) повторення розв'язання задачі чи виконання програм при тих самих вхідних даних;

в) виключення повідомлення з оброблення за причиною спотворення або важкості продовження обчислювального процесу;

г) короткочасне скорочення розв'язання даної задачі до оновлення вихідних даних чи усунення джерела спотворювання;

д) переналагодження режиму роботи ПЗ через появу чинників спотворень;

е) відновлення інформації за рахунок її дублювання;

є) відновлення процесу обробки інформації чи процесу керування з режиму початкового пуску.

Основна мета випробувань ПЗ на надійність полягає у встановленні відповідності розроблених програм вимогам технічного завдання та іншим документам, зокрема державним стандартам. Базовим документом при цьому повинен бути план проведення серії детермінованих та статистичних експериментів. Для цього визначаються об'єм і послідовність кожного експерименту за критеріями мінімізації їх об'єму та погодженої із замовником достовірності одержуваних результатів.

Крім наведених вище методів підвищення надійності програмного забезпечення, для створення надійних програм необхідно дотримуватись певної послідовності [6].

1. Створення будь-якої програми починається з *постановки задачі*. На цьому етапі необхідно предметну задачу перевести в терміни більш близькі до програмування. Постановка задачі завершується створенням технічного завдання, а потім зовнішньої специфікації програми, яка включає в себе опис даних та результатів, опис обчислювальної частини задачі, спосіб звернення до програми, опис можливих аварійних ситуацій та помилок користувача. На цьому етапі програма розглядається як чорний ящик, для якого визначена його функція та вхідні і вихідні дані.
2. Алгоритми для розв’язання задачі та точність її розв’язку залежить від типів та структури даних, тому на даному етапі *розробляється внутрішня структура даних*.
3. *Визначення загальної структури та взаємодії модулів програми*. На даному етапі необхідно розбити задачу на менш складні автономні модулі, тому важливим елементом є специфікація інтерфейсів, які визначають способи взаємодії окремих модулів
4. *Структурне програмування*. Спочатку створюється логічний скелет програми, який потім наповнюється програмним кодом. Для цього пишуться програмні модулі верхнього рівня та складаються тестові програми для їх налагодження, а програмні модулі наступного рівня заміняються вставками, які спрощено імітують їх наявність.
5. *Тестування програми*. Для повного тестування програми необхідно перевіряти кожну гілку алгоритму. Тести повинні перевіряти роботу програми при граничних умовах даних. Окремо необхідно перевіряти реакцію програми на помилкові вхідні дані.

Головною метою, до якої необхідно наближатись, є отримання легкозрозумілої програми з максимально простою структурою. Тому в більшості випадків справедливий девіз “Better simpler than clever”. Тому всі технології програмування направлені для досягнення саме цієї мети, що дає можливість легко модифікувати та супроводжувати програму у майбутньому.

При створенні ж самого тексту програми бажано дотримуватись таких рекомендацій:

1. Якщо алгоритм можна розбити на послідовність закінчених дій, то кожну дію необхідно оформляти у вигляді функції.

2. Імена змінних бажано вибирати, виходячи з їх функціонального призначення.

3. Локальні змінні більш бажані, ніж глобальні. Глобальні змінні краще оголошувати з атрибутом “static”.

4. Всю необхідну для функції інформацію необхідно намагатись передавати їй в як параметри, а не через глобальні змінні, зміну яких важко відслідкувати.

5. При створенні ж самого тексту програми слід уникати використання в програмі чисел у явному вигляді.

6. В ключових місцях програму слід супроводжувати коментарями.

Як приклад розглянемо реалізацію контролю достовірності передачі інформаційного блоку по лінії зв’язку чи контролю цілісності інформації в пам’яті мікроконтролера з використанням інформаційної надлишковості. Наведений нижче на мові програмування Сі програмний модуль виконує згортку інформаційного блоку через таблицю та формує 16-розрядну контрольну суму `zModem_CRC_H`, `zModem_CRC_L`, яка передається та зберігається разом з інформаційним блоком. Критерієм достовірності інформаційного блоку є виконання згортки над інформаційним блоком та співпадання отриманої контрольної суми з контрольною сумою, яка зберігається разом з інформаційним блоком.

Текст програмного модуля має такий вигляд:

```
//========== Table for counting zModem CRC ================
/* Table of CRC values for high-order byte */
char __flash auchCRCHi[] =
{
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,
0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,
0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40,
0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,
0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40,
0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
```

```
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40,
0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,
0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40
} ;

/* Table of CRC values for low-order byte */

char __flash auchCRCLo[] =
{
0x00, 0xC0, 0xC1, 0x01, 0xC3, 0x03, 0x02, 0xC2, 0xC6, 0x06,
0x07, 0xC7, 0x05, 0xC5, 0xC4, 0x04, 0xCC, 0x0C, 0x0D, 0xCD,
0x0F, 0xCF, 0xCE, 0x0E, 0x0A, 0xCA, 0xCB, 0x0B, 0xC9, 0x09,
0x08, 0xC8, 0xD8, 0x18, 0x19, 0xD9, 0x1B, 0xDB, 0xDA, 0x1A,
0x1E, 0xDE, 0xDF, 0x1F, 0xDD, 0x1D, 0x1C, 0xDC, 0x14, 0xD4,
0xD5, 0x15, 0xD7, 0x17, 0x16, 0xD6, 0xD2, 0x12, 0x13, 0xD3,
0x11, 0xD1, 0xD0, 0x10, 0xF0, 0x30, 0x31, 0xF1, 0x33, 0xF3,
0xF2, 0x32, 0x36, 0xF6, 0xF7, 0x37, 0xF5, 0x35, 0x34, 0xF4,
0x3C, 0xFC, 0xFD, 0x3D, 0xFF, 0x3F, 0x3E, 0xFE, 0xFA, 0x3A,
0x3B, 0xFB, 0x39, 0xF9, 0xF8, 0x38, 0x28, 0xE8, 0xE9, 0x29,
0xEB, 0x2B, 0x2A, 0xEA, 0xEE, 0x2E, 0x2F, 0xEF, 0x2D, 0xED,
0xEC, 0x2C, 0xE4, 0x24, 0x25, 0xE5, 0x27, 0xE7, 0xE6, 0x26,
0x22, 0xE2, 0xE3, 0x23, 0xE1, 0x21, 0x20, 0xE0, 0xA0, 0x60,
0x61, 0xA1, 0x63, 0xA3, 0xA2, 0x62, 0x66, 0xA6, 0xA7, 0x67,
0xA5, 0x65, 0x64, 0xA4, 0x6C, 0xAC, 0xAD, 0x6D, 0xAF, 0x6F,
0x6E, 0xAE, 0xAA, 0x6A, 0x6B, 0xAB, 0x69, 0xA9, 0xA8, 0x68,
0x78, 0xB8, 0xB9, 0x79, 0xBB, 0x7B, 0x7A, 0xBA, 0xBE, 0x7E,
0x7F, 0xBF, 0x7D, 0xBD, 0xBC, 0x7C, 0xB4, 0x74, 0x75, 0xB5,
0x77, 0xB7, 0xB6, 0x76, 0x72, 0xB2, 0xB3, 0x73, 0xB1, 0x71,
0x70, 0xB0, 0x50, 0x90, 0x91, 0x51, 0x93, 0x53, 0x52, 0x92,
0x96, 0x56, 0x57, 0x97, 0x55, 0x95, 0x94, 0x54, 0x9C, 0x5C,
0x5D, 0x9D, 0x5F, 0x9F, 0x9E, 0x5E, 0x5A, 0x9A, 0x9B, 0x5B,
0x99, 0x59, 0x58, 0x98, 0x88, 0x48, 0x49, 0x89, 0x4B, 0x8B,
0x8A, 0x4A, 0x4E, 0x8E, 0x8F, 0x4F, 0x8D, 0x4D, 0x4C, 0x8C,
0x44, 0x84, 0x85, 0x45, 0x87, 0x47, 0x46, 0x86, 0x82, 0x42,
0x43, 0x83, 0x41, 0x81, 0x80, 0x40
} ;
//-----------------------------------------------------

 char zModem_CRC_H;  // high CRC byte
 char zModem_CRC_L;  // low CRC byte

//-----------------------------------------------------
void zModem_CRC_Clear(void)
{
 zModem_CRC_H = 0xFF;
 zModem_CRC_L = 0xFF;
}
//-----------------------------------------------------
```

```
//-------------------------------------------------------
void zModem_CRC_Add(char byte)
{
 char uIndex;

 uIndex       = zModem_CRC_H ^ byte;     //calculate the CRC
 zModem_CRC_H = zModem_CRC_L ^ auchCRCHi[uIndex] ;
 zModem_CRC_L = auchCRCLo[uIndex] ;
}
//-------------------------------------------------------
```

Питання для самоперевірки

1. Які фактори впливають на надійність функціонування програмного забезпечення?

2. В чому полягає використання інформаційної надлишковості для підвищення надійності програмного забезпечення?

3. Сформулюйте етапи контролю надійності функціонування програмного забезпечення.

Література

1. Голубцов М.С. Микроконтроллеры AVR: от простого к сложному / –М.: СОЛОН-Пресс, 2003. - 288 с.
2. Локазюк В.М. Надійність, помилки і тестування програмного забезпечення комп’ютерних пристроїв та систем: Навчальний посібник. – Хмельницький: ТУП, 2003. -74с.
3. http:// www.atmel.com
4. http:// www.IAR.com
5. http://www.hpinfotech.com
6. C/C++. Программирование на языке высокого уровня/ Т.А. Паловская. – СПб. : Питер, 2002. - 464 с.: Ил.
7. Профессиональное программирование на языке Си: От Turbo C к Borland C++: Справ. пособие под ред. А.И. Касаткина. – Мн.: Выш. Шк., 1992. - 240 с.
8. Трамперт В. AVR-RISC микроконтроллеры.: Пер. с нем. – К.: “МК-Пресс”, 2006. – 464 с., ил.

Навчальне видання

Дементьєв Юрій Вікторович, Задорожний Віталій Констянтинович

Програмування AVR RISC мікроконтролерів

Навчальний посібник

Оригінал-макет підготовлено авторами

Редактор Т.О. Старічек

Науково-методичний відділ ВНТУ
Свідоцтво Держкомінформу України
Серія ДК № 746 від 25.12.2001
21021, м. Вінниця, Хмельницьке шосе, 95, ВНТУ

Підписано до друку
Формат 29.7 × 42 1/4
Друк різографічний
Тираж прим.
Зам. №

Гарнітура Times New Roman
Папір офсетний
Ум. друк. арк.

Віддруковано в комп'ютерному інформаційно-видавничому центрі
Вінницького національного технічного університету
Свідоцтво Держкомінформу України
Серія ДК № 746 від 25.12.2001

21021, м. Вінниця, Хмельницьке шосе, 95, ВНТУ