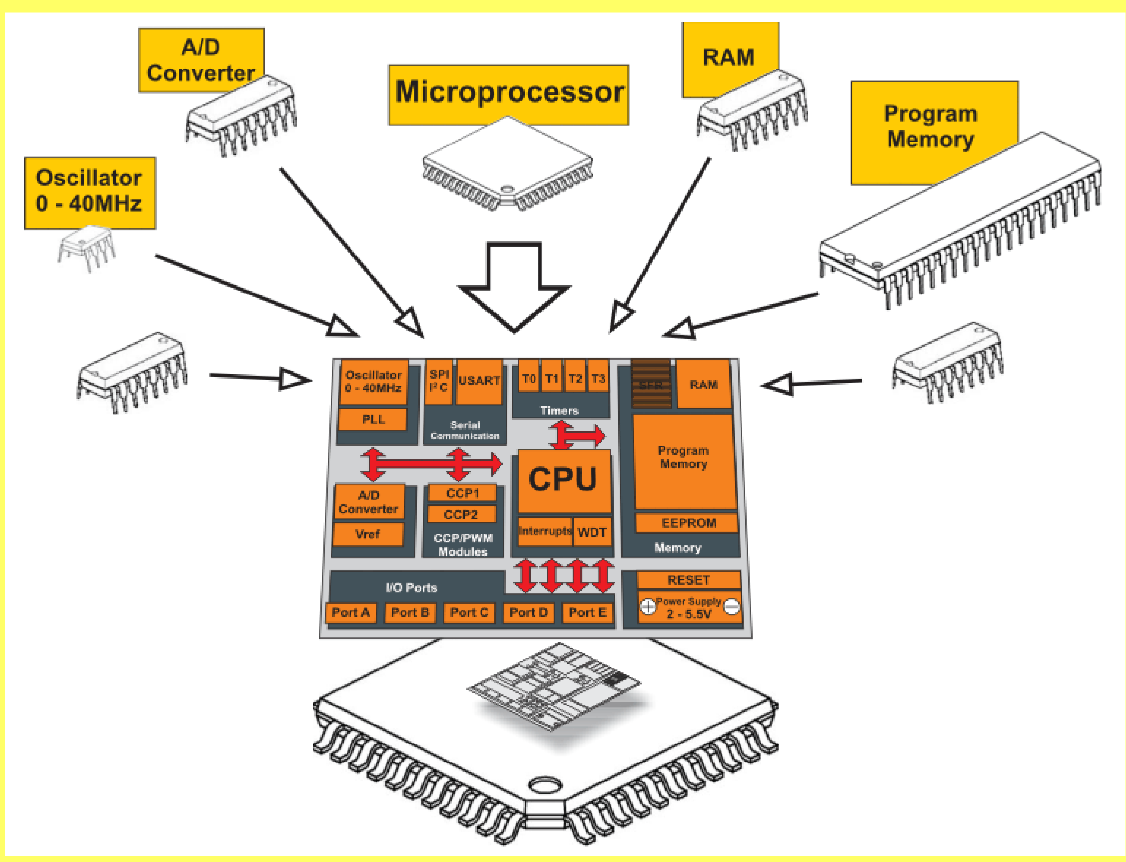
Микроконтролери



Архитектура и принцип на действие

Първо издание

Димо Петков

Микроконтролери

Архитектура и принцип на действие

Първо издание

София, 2015 г.

Всички права запазени. Нито една част от тази книга не може да бъде размножавана или предавана под никаква форма или начин, електронен или механичен, включително фотокопиране, записване или чрез каквито и да е системи за съхранение на информация, без предварително писмено разрешение от автора.

© Димо Петков Петков, автор

ISBN xxxx-xxxx-xxxx-xxxx

Кратко съдържание:

За автора 15
От автора 15
Предговор 16
За кого е предназначена тази книга 19
Как е организирана книгата 20
Как да четете тази книга 21

Част I: Въведение в цифровата електроника 23
1 Процесори, микропроцесори, микроконтролери и микрокомпютри 25
2 Представяне на информацията в компютрите 28
3 Бройни системи 33
4 Представяне на знакови цели числа в двоична бройна система. 45
5 Представяне на десетични дробни числа в двоична бройна система. 51
6 Цифрова аритметика 69

Част II: Градивни елементи на микроконтролерите 91
7 Логически елементи 93
8 Тригери 102
9 Регистри 118
10 Броячи 124
11 Шифратори и дешифратори 130
12 Мултиплексори и демултиплексори 132
13 Аритметично & Логическо устройство 136
14 CMOS технология 137

Част III: Архитектура на микроконтролерите 151
15 Общи сведения 153
16 Програма. Конструкция и изпълнение......................................165
17 Описание на компонентите на микроконтролера 168
18 Начално установяване на микроконтролера 210
19 Заключение 212

Част IV: Програмиране на микроконтролери на асемблер..........215
20 Въведение 217
21 Описание на процесора eZ8 219
22 Описание на инструкциите на eZ8 263
23 Описание на асемблера..415

24 Развойна среда Zilog Developer Studio II 449
25 Микроконтролери от серията ZF083A 461
26 Практически примери 591

Заключение...629

Съдържание:

За автора 15
От автора 15
Предговор 16
За кого е предназначена тази книга 19
Как е организирана книгата 20
Как да четете тази книга 21

Част I: Въведение в цифровата електроника 23
1 Процесори, микропроцесори, микроконтролери и микрокомпютри 25
2 Представяне на информацията в компютрите 28
3 Бройни системи 33
3.1 Въведение 33
3.2 Десетична бройна система 34
3.3 Двоична бройна система 36
3.4 Шестнадесетична бройна система 39
3.5 Заключение 43
4 Представяне на знакови цели числа в двоична бройна система. 45
4.1 Въведение 45
4.2 Представяне на знакови цели числа в прав код 45
4.3 Представяне на знакови цели числа в обратен код 46
4.4 Представяне на знакови цели числа в допълнителен код 47
4.5 Представяне на знакови цели числа в код с отместване 49
5 Представяне на десетични дробни числа в двоична бройна система. 51
5.1 Въведение 51
5.2 Дробни числа с фиксирана запетая 51
5.3 Дробни числа с плаваща запетая 58
5.3.1 Въведение 58
5.3.2 Стандартът IEEE 754 59
5.3.2.1 Общи сведения 59
5.3.2.2 Нормализирани числа 60
5.3.2.3 Представяне на нулата 64
5.3.2.4 Денормализирани числа 65
5.3.2.5 Безкрайност 66
5.3.2.6 NaNs 66
5.3.2.7 Обобщение 67
6 Цифрова аритметика 69
6.1 Цифрова аритметика с цели числа 69

6.2 Цифрова аритметика с числа с фиксирана запетая 87
6.3 Цифрова аритметика с числа с плаваща запетая 89

Част II: Градивни елементи на микроконтролерите 91
7 Логически елементи...93
7.1 Общи сведения...93
7.2 Логическо НЕ (NOT) 94
7.3 Логическо И (AND) 95
7.4 Логическо ИЛИ (OR)...96
7.5 Логическо Изключващо ИЛИ (XOR)..96
7.6 Логическо И-НЕ (NAND)..97
7.7 Логическо ИЛИ-НЕ (NOR) 98
7.8 Логическо Изключващо ИЛИ-НЕ (XNOR) 99
7.9 Логически повторител (буфер)..100
7.10 Логически елементи с три състояния 100
8 Тригери..102
8.1 Общи сведения...102
8.2 RS тригери 103
8.2.1 Асинхронни RS тригери 103
8.2.2 Асинхронен RS тригер с разрешаващ вход 106
8.2.3 Синхронен RS тригер..107
8.3 JK тригери 111
8.4 D тригери...113
8.5 T тригери 115
8.6 Тригер на Шмит..117
9 Регистри 118
9.1 Общи сведения...118
9.2 Преместващи регистри с последователно въвеждане..........120
9.3 Преместващи регистри с паралелно въвеждане 122
10 Броячи 124
10.1 Общи сведения...124
10.2 Асинхронни броячи..124
10.3 Синхронни броячи..127
11 Шифратори и дешифратори..130
12 Мултиплексори и демултиплексори 132
13 Аритметично & Логическо устройство 136
14 CMOS технология...137
14.1 Общи сведения...137
14.2 CMOS инвертор 139
14.3 CMOS NAND и NOR..141
14.4 CMOS буфер...142
14.5 CMOS буфер с три състояния...143
14.6 Изходи с отворен дрейн 144

14.7 Изтеглящи резистори 145
14.8 Понижаващи резистори..147
14.9 Захранващо напрежение 147
14.10 CMOS и TTL входни и изходни нива 148

Част III: Архитектура на микроконтролерите 151
15 Общи сведения 153
15.1 Обобщена архитектура..153
15.2 Типове микроконтролери..156
15.2.1 8-, 16- и 32-битови микроконтролери....................................157
15.2.2 Архитектура на микроконтролерите 158
15.2.3 Набор от инструкции 163
16 Програма. Конструкция и изпълнение....................................165
17 Описание на компонентите на микроконтролера 168
17.1 Базови компоненти на микроконтролера 168
17.1.1 Централен процесор 168
17.1.2 Програмна памет 171
17.1.3 Памет за данни..173
17.1.4 Контролер на прекъсванията....................................179
17.1.5 Генераторен блок..183
17.2 Периферия 186
17.2.1 Портове..186
17.2.2 Таймери 190
17.2.3 ШИМ (PWM - Pulse-Width Modulation) 193
17.2.4 Сравнение (Compare) 195
17.2.5 Прихващане (Capture) 196
17.2.6 Стражеви таймер (Watchdog) 199
17.2.7 Аналого-цифров преобразувател (АЦП)............................200
17.2.8 Аналогов компаратор..204
17.2.9 Комуникационни интерфейси....................................207
17.2.10 Други..209
18 Начално установяване на микроконтролера 210
19 Заключение 212

Част IV: Програмиране на микроконтролери на асемблер..........215
20 Въведение 217
21 Описание на процесора eZ8 219
21.1 Организация на паметта..219
21.2 Програмен модел 223
21.3 Стек 227
21.4 Прекъсвания 230
21.4.1 Общи сведения..230
21.4.2 Обработка на разрешено прекъсване....................................230

21.4.3 Обработка на неразрешено прекъсване......................233
21.4.4 Вложени прекъсвания 234
21.4.5 Генериране на софтуерни прекъсвания.......................236
21.4.6 Аварийни прекъсвания 237
21.5 Адресни режими 238
21.5.1 Регистрова адресация...240
12-битова регистрова адресация (ER) 240
8-битова регистрова адресация (R, RR) 241
4-битова регистрова адресация (r, rr) 243
21.5.2 Косвена адресация (IR, IRR, Ir, Irr)............................245
21.5.3 Escaped-адресация 249
21.5.4 Индексна адресация (X)...251
21.5.5 Директна адресация (DA) 252
21.5.6 Относителна адресация (RA) 253
21.6 Списък с инструкции...255
22 Описание на инструкциите на eZ8 263
ADC 264
ADCX 267
ADD 269
ADDX 272
AND 274
ANDX 277
ATM 279
BCLR 280
BIT 281
BRK 283
BSET 284
BSWAP...285
BTJ 287
BTJNZ..289
BTJZ 291
CALL 293
CCF 296
CLR 297
COM 299
CP 301
CPC 304
CPCX 307
CPX 309
DA 311
DEC 315
DECW...317
DI 319

DJNZ 320
EI 322
HALT 323
INC 324
INCW 326
IRET 328
JP 329
JP cc 331
JR 332
JR cc 333
LD 334
LDC 338
LDCI 340
LDE 342
LDEI 344
LDWX 346
LDX 348
LEA 353
MULT 355
NOP 357
OR 358
ORX 361
POP 363
POPX 365
PUSH 367
PUSHX 369
RCF 371
RET 372
RL 373
RLC 375
RR 377
RRC 379
SBC 381
SBCX 384
SCF 386
SRA 387
SRL 389
SRP 391
STOP 392
SUB 393
SUBX 396
SWAP 398
TCM 399

TCMX 401
TM 403
TMX 405
TRAP 407
WDT 409
XOR 410
XORX 413
23 Описание на асемблера...415
23.1 Общи сведения..415
23.2 Адресни пространства и сегменти 415
23.3 Изрази 418
23.3.1 Числово представяне...418
23.3.2 Аритметични оператори 419
23.3.3 Оператори за отношение 420
23.3.4 Битови оператори 420
23.3.5 Операторите HIGH и LOW..421
23.3.6 Операторите HIGH16 и LOW16..421
23.3.7 Оператор .FTOL..422
23.3.8 Оператор .LTOF..422
23.3.9 Приоритет на операторите..422
23.4 Директиви..423
23.4.1 Директиви за данни 423
BLKB 423
BLKW 424
BLKL 424
DB 425
DW 426
DW24 426
DL 426
DF...427
DD 427
23.4.2 ALIGN...428
23.4.3 SEGMENT 428
23.4.4 ORG..428
23.4.5 DEFINE...429
23.4.6 DS 430
23.4.7 END..431
23.4.8 EQU..431
23.4.9 VAR...432
23.4.10 INCLUDE 432
23.4.11 VECTOR 433
23.4.12 XDEF..434
23.4.13 XREF..434

23.4.14 Условни директиви 435
IF 435
IFDEF...437
IFSAME 438
IFMA..439
23.5 Макроси 440
23.5.1 Дефиниране и извикване на макрос 440
23.5.2 Локални макро-етикети 442
23.5.3 Макро-аргументи по избор..443
23.5.4 Излизане от макрос 444
23.6 Етикети 444
23.6.1 Анонимни етикети..445
23.6.2 Локални етикети 446
23.6.3 Импортиране и експортиране на етикети 446
23.6.4 Пространства на етикетите..447
23.6.5 Проверка на етикетите 447
24 Развойна среда Zilog Developer Studio II 449
24.1 Общи сведения..449
24.2 Създаване на асемблерен проект.......................................449
24.3 Отваряне на съществуващ проект......................................452
24.4 Конфигуриране на проекта 452
24.5 Описание на основните менюта и прозорци.....................455
24.5.1 Асемблиране на проекта...455
24.5.2 Тестване и дебъгване 455
24.6 Заключение..459
25 Микроконтролери от серията ZF083A 461
25.1 Основни характеристики 461
25.2 Описание на изводите 463
25.3 Адресно пространство..467
25.4 Карта на регистровата памет 468
25.5 Ресет и излизане от STOP-режим.......................................472
25.5.1 Източници и типове ресет 472
25.5.2 Описание на източниците на ресет...............................474
25.5.3 Излизане от STOP-режим 476
25.5.4 Описание на управляващите регистри 478
Регистър RSTSTAT (Reset Status)...478
25.6 Контролер на прекъсванията 481
25.6.1 Общи сведения...481
25.6.2 Описание на управляващите регистри 485
Регистър IRQ0 (Interrupt Request 0)..485
Регистър IRQ1 (Interrupt Request 1)..486
Регистър IRQ2 (Interrupt Request 2)..486
Регистър IRQ0ENH (IRQ0 Enable High Bit)............................487

Регистър IRQ0ENL (IRQ0 Enable Low Bit)....................................487
Регистър IRQ1ENH (IRQ1 Enable High Bit)..................................488
Регистър IRQ1ENL (IRQ1 Enable Low Bit)....................................488
Регистър IRQ2ENH (IRQ2 Enable High Bit)..................................488
Регистър IRQ2ENL (IRQ2 Enable Low Bit)....................................489
Регистър IRQES (Interrupt Edge Select) 489
Регистър IRQSS (Shared Interript Select) 490
Регистър IRQCTL (Interrupt Control)..490
25.7 Генераторен блок 492
25.7.1 Общи сведения...492
25.7.2 Детектиране и възстановяване на повреден системен тактов сигнал..494
25.7.3 Кварцов генератор..495
25.7.4 Вътрешен прецизен генератор 498
25.7.5 Описание на управляващите регистри 498
Регистър OSCCTL (Oscillator Control Register)...........................498
25.8 Режими с ниска консумация 500
25.8.1 Общи сведения...500
STOP-режим 500
HALT-режим...501
25.8.2 Описание на управляващите регистри 502
Регистър PWCTRL0 (Power Control Register 0)..........................502
25.9 Портове 504
25.9.1 Общи сведения...504
25.9.2 Алтернативни функции на изводите 504
25.9.3 Управляващи регистри 508
25.9.4 Описание на управляващите регистри 509
Регистър PxADDR (Port Address) 509
Регистър PxCTL (Port Control) 510
Регистър PxDD (Port Direction)...510
Регистър PxAF(Port Alternate Function).....................................511
Регистър PxOC (Port Output Control)..512
Регистър PxHDE (Port High Drive Enable) 512
Регистър PxSMRE (Port Stop Mode Recovery Enable)...............513
Регистър PxPUE (Port Pull-up enable)..513
Регистър PxAFS1 (Port Alternate Set 1) 514
Регистър PxAFS2 (Port Alternate Set 2) 514
Регистър PxIN (Port Input Data)..515
Регистър PxOUT (Port Output Data)..515
Регистър LEDEN (LED Drive Enable)..516
Регистър LEDLVLH (LED Drive Level High) 517
Регистър LEDLVLL (LED Drive Level Low) 517
25.10 WDT таймер 518

25.10.1 Общи сведения....518
25.10.2 Устройство и принцип на работа....518
25.10.3 Реакция при препълване на WDT-таймера....520
25.10.4 Запис в регистрите WDTU, WDTH и WDTL....521
25.10.5 Описание на управляващите регистри....522
Регистър WDTCTL (Watchdog Timer Control)....522
Регистър WDTU (Watchdog Timer Reload Upper Byte)....522
Регистър WDTH (Watchdog Timer Reload High Byte)....523
Регистър WDTL (Watchdog Timer Reload Low Byte)....523
25.11 Таймери....525
25.11.1 Общи сведения....525
25.11.2 Устройство и принцип на работа....525
25.11.3 Режими на работа....526
Еднократен режим (single-shot mode)....526
Непрекъснат режим (continuous mode)....528
Броячен режим....529
Компараторен броячен режим....531
Едноизходен PWM режим....532
Двуизходен PWM режим....536
Режим на прихващане (capture mode)....537
Режим на прихващане с рестартиране....539
Режим на сравнение (compare mode)....541
Режим на контролирано броене (gated mode)....543
Режим на прихващане/сравнение....545
25.11.4 Четене на текущата броячна стойност на таймера....546
25.11.5 Описание на управляващите регистри....547
Регистри TxH (Timer High Byte) и TxL (Timer Low Byte)....547
Регистри TxRH (Timer Reload High Byte) и TxRL (Timer Reload Low Byte)....548
Регистри TxPWMH (PWM High Byte) и TxPWML (PWM Low Byte)....549
Регистър TxCTL0 (Timer 0 Control)....550
Регистър TxCTL1 (Timer 1 Control)....551
25.12 Компаратор....555
25.12.1 Общи сведения....555
25.12.2 Устройство и принцип на работа....555
25.12.3 Описание на управляващите регистри....557
Регистър CMP0 (Comparator Control Register)....557
25.13 Аналого-цифров преобразувател....559
25.13.1 Общи сведения....559
25.13.2 Устройство и принцип на работа....559
25.13.3 Описание на управляващите регистри....561
Регистър ADCCTL0 (ADC Control 0)....561

Регистър ADCD_H (ADC Data High Byte) 562
Регистър ADCD_L (ADC Data Low Bits) 563
Регистър ADCSST (ADC Settling Time) 563
Регистър ADCST (ADC Sample Time) 564
Регистър ADCCP (ADC Clock Prescale) 564
25.14 Флаш-памет 565
25.14.1 Общи сведения 565
25.14.2 Флаш-информационна област 568
25.14.3 Флаш-контролер 568
25.14.4 Описание на управляващите регистри 573
Регистър FCTL (Flash Control) 573
Регистър FSTAT (Flash Status) 574
Регистър FPS (Flash Page Select) 575
Регистър FPROT (Flash Sector Protect) 575
Регистър FFREQH (Flash Frequency High) 576
Регистър FFREQL (Flash Frequency Low) 576
25.15 Флаш-конфигурационни битове 578
25.15.1 Общи сведения 578
25.15.2 Типове конфигурационни битове 579
25.16 Енерго-независима памет за данни (NVDS) 581
25.16.1 Общи сведения 581
25.16.2 Запис на байт в NVDS 581
25.16.3 Четене на байт от NVDS 582
25.17 Дебъгер 584
25.17.1 Общи сведения 584
25.17.2 Свързване на дебъгера към микроконтолера 585
25.17.3 Тестване/дебъгване с USB SmartCable 585
25.18 Електрически параметри 588
25.19 Информация за поръчка 589
26 Практически примери 591
26.1 С какво разполагаме 591
26.2 Пример 1 599
26.3 Пример 2 603
26.4 Пример 3 607
26.5 Пример 4 613
26.6 Пример 5 620

Заключение **629**

За автора

ДИМО ПЕТКОВ завършва Техническия Университет в Габрово през 2001г. Има магистърска степен по специалността „**Комуникационна Техника и Технологии“**. Започва кариерата си като сервизен инженер на аудио-видео техника. По-късно работи няколко години в белгийската фирма „**Мелексис“** като приложен инженер. През цялото това време се занимава с изучаване на програмирането на микроконтролери. Понастоящем работи като С-програмист в немската компания **КОСТАЛ**, занимаваща се с разработката на електронни модули за автомобили.

От автора

Липсата на подобна литература на българския пазар ме мотивира да създам тази книга. Като начинаещи програмисти на микроконтролери вероятно сте се сблъскали с въпросите "Откъде да започна?", "Коя книга е подходяща за мен?". Създадох тази книга като отговор на тези въпроси, за да спестя на начинаещите лутането в търсене на правилния път. Надявам се, че съм успял да постигна до голяма степен целта си. Може да изпращате Вашите въпроси и забележки на dimope@abv.bg.

Предговор

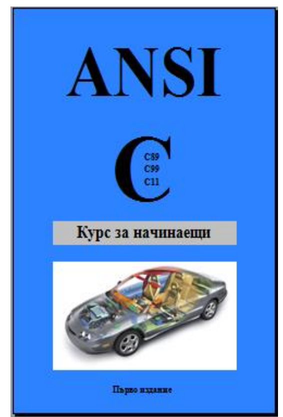
Развитието на микроелектрониката доведе до създаване на "интелигентни" интегрални схеми, наречени микроконтролери. Микроконтролерът е малък компютър, събран в един корпус и с ограничени ресурси в сравнение с един персонален компютър.

Микроконтролерите се използват във всички сфери на живота. Почти всички съвременни електронни устройства използват микроконтролери, от малки домакински уреди до печки, хладилници и т.н. Всички съвременни автомобили съдържат редица електронни модули с микроконтролери.

Терминът "интелигентен" е метафоричен, тъй като микроконтролерът се превръща в такъв само когато се програмира да върши определена задача, в противен случай

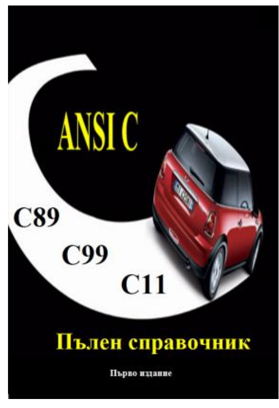
той е просто една безполезна интегрална схема. Езикът, който микроконтролерите разбират, се нарича **машинен език**. Той е набор от машинни инструкции. Една машинна инструкция е комбинация от единици (1) и нули (0). Директното използване на машинни инструкции от програмистите при създаване на програма е много труден и бавен процес. За улеснение на програмистите наборът от машинни инструкции на един микроконтролер се заменя с псевдо инструкции, които се състоят от по-лесни за разбиране и запомняне от човек символи. Псевдо инструкциите се наричат още **асемблерни инструкции**. Те се преобразуват в машинни инструкции чрез специална програма, наречена **Асемблер**. Различните микроконтролери имат различна архитектура и различен набор от машинни (съответно и асемблерни) инструкции. Например PIC микроконтролерите имат свой набор от инструкции, който се различава от набора инструкции на AVR микроконтролерите. За съжаление няма универсален асемблерен език. Това означава, че ако искате да прехвърлите програма, писана на асемблерен език за един микроконтролер - върху друг, ще трябва да я пренапишете наново, като използвате асемблерните инструкции на втория микроконтролер. Това е и един от големите недостатъци на асемблерния език. Освен това писането на асемблерен език изисква много добро познаване на архитектурата на микроконтролера. Също така, въпреки че асемблерните инструкции са по-разбираеми в сравнение с машинните, научаването и помненето им също е голямо предизвикателство. Тези недостатъци дават тласък в търсенето на други методи за създаване на по-големи и по-сложни програми. Така се появяват езици от по-високо ниво какъвто е езикът C. Езикът C предоставя едно абстрактно ниво на програмиране, което не зависи (или поне в по-голямата си част) от микроконтролера, за който се пише програмата. Именно в програмирането на микроконтролери езикът C остава незаменим, а това води и до нужда от C-програмисти.

Тази книга е третата от поредица книги, които ще Ви дадат пълна информация в максимално достъпен вид, която е необходима, за да се научите да програмирате микроконтролери на езика C. Процесът включва три стъпки. Първата стъпка е да се запознаете с елементите на езика C. За тази цел е предвидена книгата **„ANSI C. Курс за начинаещи”**.



Тя се фокусира върху базовите знания свързани с езика C. Освен, че ще придобиете добри познания за структурата на езика ще разберете и как той функционира. За максимално улеснение на абсолютно начинаещите C-програмисти е добавена и предварителна информация, необходима за разбирането на описания материал. Книгата разглежда почти всички аспекти на езика C, но не влиза в дълбоки детайли, които биха затруднили начинаещия програмист.

Предвидена е втора книга **„ANSI C. Пълен Справочник”**, която разглежда детайлно езика C и може да се използва за справочник дори от опитни програмисти.



Прочитайки книгата **„ANSI C. Пълен справочник”**, Вие ще станете истински C експерти.

След като усвоите езика C, втората стъпка е да се запознаете с устройството и принципа на действие на един микроконтролер. Това е и целта на настоящата книга.

Програмирането се учи най-вече с практика. Това е и третата стъпка. За целта е предвидена четвърта книга

„Програмиране на микроконтролери на езика С", която съдържа практически примери. Тя комбинира наученото от предходните две стъпки и ще Ви даде практически познания за софтуерните и хардуерни средства, необходими при написването, тестването и дебъгването (откриването и коригирането на грешки) на С код, на базата на конкретен микроконтролер.

За кого е предназначена тази книга

Тази книга е предназначена основно за хора, които искат да се занимават с програмиране на микроконтролери. Тя е подготвена така, че да Ви запознае с всички базови понятия и термини, които ще са Ви необходими при изучаването на устройството им. Книгата ще бъде и полезна за всеки електронен инженер, който иска да се запознае с устройството и принципа на работа на микроконтролерите. Ако не сте се занимавали досега с програмиране на микроконтролери и не знаете откъде да започнете, тази книга е точно за Вас. Книгата няма да Ви направи програмисти, но ще Ви даде всички необходими базови знания, които ще са Ви необходими при програмирането на микроконтролери.

Как е организирана книгата

Книгата е организирана в четири части:

Част I Ви запознава с предварителните знания, които ще са Ви необходими при изучаването на устройството и принципа на работа на един микроконтролер с помощта на тази книга. Тези знания включват представянето на информацията в микроконтролера, начина на представяне на числата в различни бройни системи (двоична, десетична, шестнадесетична), запознаване с аритметиката в двоична бройна система (цифрова аритметика).

Част II разглежда градивните елементи на микроконтролерите.

Част III разглежда базовите архитектури, по които се изграждат микроконтролерите и описва компонентите на един микроконтролер.

Част IV ще Ви запознае с програмирането на асемблерен език на базата на микроконтролера **ZF083A**, част от серията **Z8 Encore!** на компанията **ZiLOG**. Ще използвам развойния комплект **Z8F083A0128ZCOG**, който включва развойна платка, дебъгер и диск с развойната среда. Целта е да придобиете реална представа за устройството и начина на работа на един микроконтролер.

Причината да избера този микроконтролер е неговата простота, евтини софтуерни и хардуерни програмни средства. Развойната среда **Zilog Developer Studio II** осигурява текстов редактор, асемблер и C компилатор, и е напълно безплатна, изключително опростена и лесна за работа. Дебъгерът **USB Smart Cable** е от порядъка на 40-45 евро, което е напълно приемлива цена. Всички тези неща правят микроконтролерите от серията Z8 Encore!

изключително подходящи за начинаещите. Идеята ми е да отделите колкото се може по-малко време в изучаването на конкретна архитектура и развойните й средства и да се концентрирате върху разбирането на базовите принципи. Доброто познаване на тези принципи е предпоставка за по-лесно ориентиране в различните микроконтролерни архитектури.

Как да четете тази книга

Ако сте абсолютно начинаещ, ще трябва да прочетете цялата книга отначало докрай внимателно. Ако сте чели книгата **„ANSI C. Курс за начинаещи”**, голяма част от Част I ще Ви е позната. Въпреки това е добавен нов материал, който ще разшири познанията Ви. Ако сте на "ти" с цифровата електроника, може директно да преминете към Част III и IV.

Част I

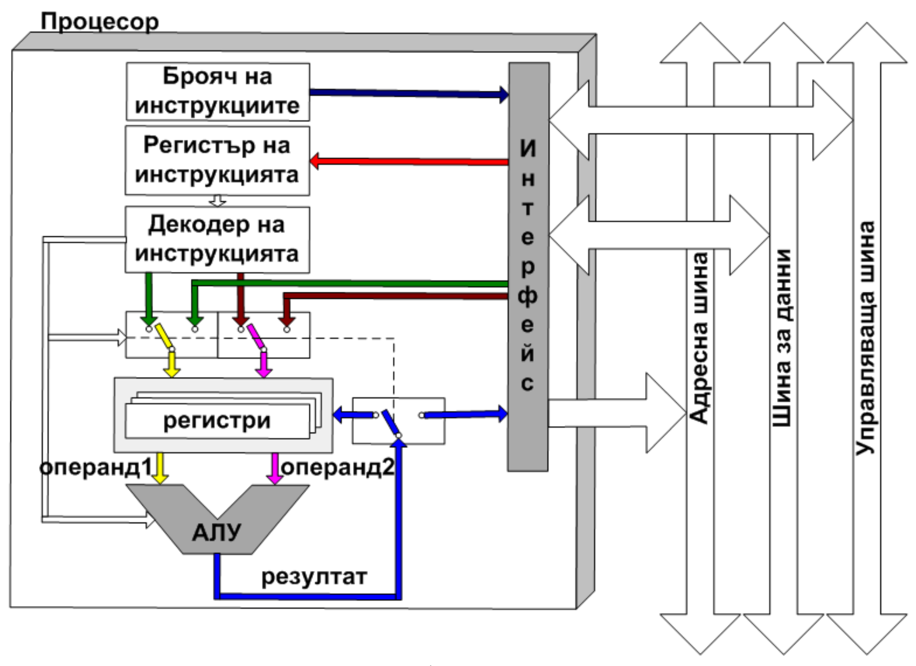
Въведение в цифровата електроника

1 Процесори, микропроцесори, микроконтролери и микрокомпютри

Когато започнете да се занимавате с програмиране на микроконтролери, ще се сблъскате с термини като процесор, микропроцесор, микрокомпютър и разбира се микроконтролер. Често тези термини се използват като еквивалентни. Въпреки това те обозначават различни неща. В тази глава ще направя кратък преглед какво означава всеки един от тези термини.

1) Процесор (**CPU - Central Processor Unit**)

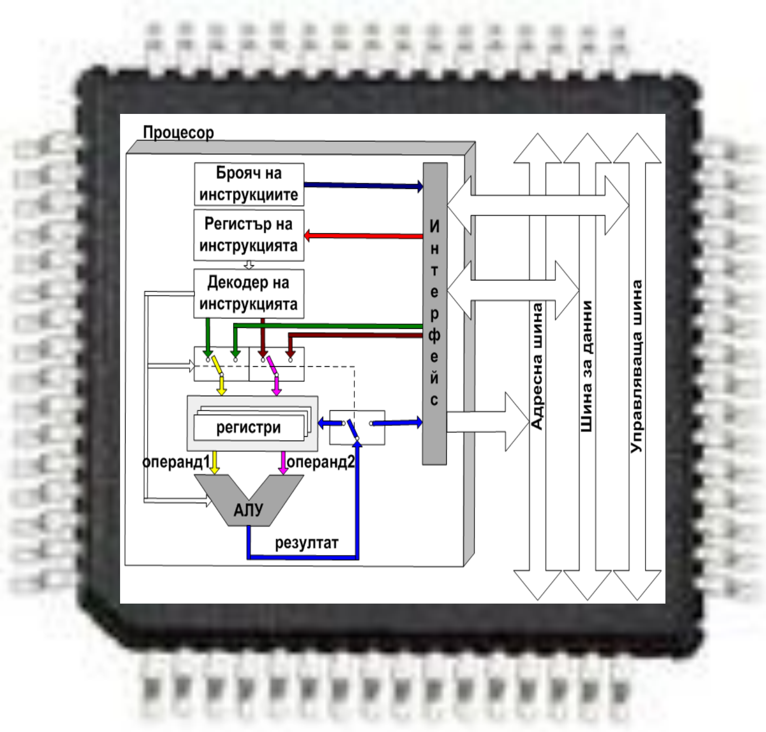
Процесорът е „мозъка” на една компютърна система. Това е главният компонент, който управлява работата на системата. Процесорът сам по себе си не съществува физически самостоятелно като електронен компонент. Също така процесорът не е завършена компютърна система, а само част от нея.



Фиг. 1 Процесор

2) Микропроцесор

Микропроцесорът е процесор, затворен в интегрална схема. Казано с други думи микропроцесорът е физическа реализация на процесор под формата на електронен компонент. Микропроцесорът също не е завършена компютърна система, а е само част от нея.



Фиг. 2 Микропроцесор

3) Микрокомпютър

Микрокомпютърът е микропроцесор със свързани към него допълнителни електронни компоненти (памети, генератор на тактов сигнал, портове и др.), необходими за изграждане на една компютърна система. Казано с други думи микрокомпютърът е вече завършена компютърна система.

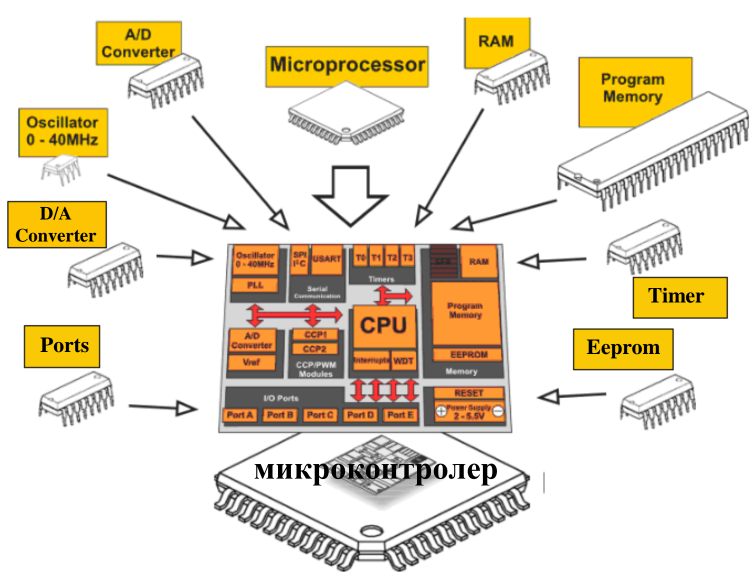
Терминът „микрокомпютър” идва от факта, че дадена

компютърна система използва микропроцесор. От тази гледна точка персоналният компютър също се явява микрокомпютър.

Фиг. 3 Персонален компютър

4) Микроконтролер

Микроконтролерът е микрокомпютър затворен в единична интегрална схема, т.е. . микроконтролерът също е завършена компютърна система.

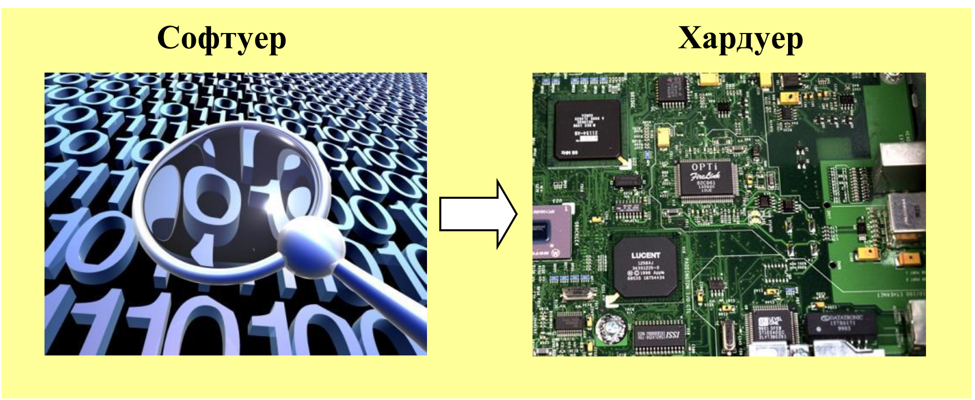


Фиг. 4 Микроконтролер

Фактът, че всички компоненти са събрани в един чип, означава, че микроконтролерът е с по-ограничени ресурси и възможности в сравнение с един микрокомпютър.

2 Представяне на информацията в компютрите

Най-общото определение за компютър е електронно устройство за въвеждане, обработка и извеждане на информация, състоящо се от два неразделни компонента: **хардуер** и **софтуер**. Хардуерът включва всички електронни елементи, от които е изграден компютърът. Софтуерът е програма, която "казва" на хардуера как да обработва информацията.

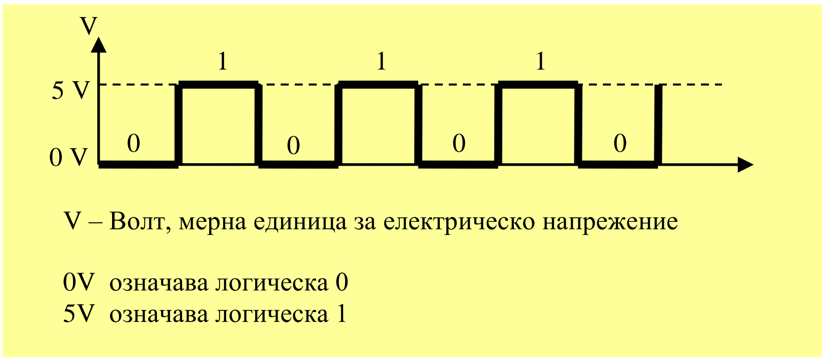


Фиг. 5 Главни компоненти на компютър

Съвременните компютри са цифрови. Независимо каква информация искаме да обработим, тя трябва да бъде подадена на компютъра в цифров (двоичен) вид. Цифровата информация е съвкупност от комбинации само от две стойности: нула (0) и единица (1). Причината информацията да се представя в цифров вид е, че тези две стойности се реализират от електронните компоненти много лесно само с помощта на два електрически сигнала: Например 0V и 5V (Фиг.6).

Тъй като стойностите 0 и 1 са абстрактни понятия, те се наричат още логическа 0 и логическа 1.

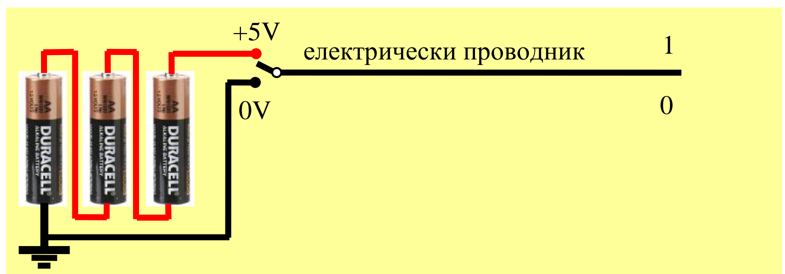
Най-малкото количество информация, която може да се пре-



Фиг. 6 Представяне на стойностите 0 и 1 с помощта на електрически сигнали

даде в цифров вид е 0 или 1 и се нарича **бит**, т.е. . един бит информация може да приема само две стойности: 0 или 1.

Фиг.7 показва как можем да предадем един бит информация по електрически проводник.

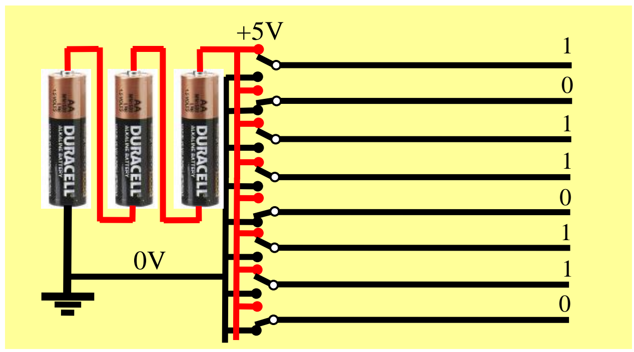


Фиг. 7 Предаване на бит по електрически проводник

Когато подадем на проводника напрежение 5V с помощта на ключето, това е еквивалентно предаване на лог.1 по него. Когато подадем на проводника 0V, това е еквивалентно предаване на лог.0.

За да представим повече цифрова информация наведнъж, ще трябва да използваме няколко бита в паралел, респективно няколко проводника (Фиг.8).

Съвкупността от 8 бита се нарича байт. Байтът е фундаментална мерна единица за количество информация в



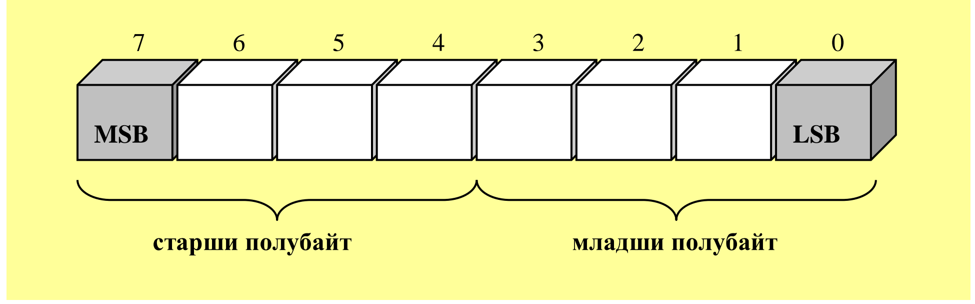
Фиг. 8 Предаване на 8 бита информация по електрически проводници

цифровата електроника. По-долу са дадени мерни единици и означения, които ще срещате често.

1 бит		[1bit]
1 байт	= 8 бита	[1B]
1 килобайт	= 1024 байта	[1kB]
1 мегабайт	= 1024 килобайта	[1MB]
1 гигабайт	= 1024 мегабайта	[1GB]
1 терабайт	= 1024 гигабайта	[1TB]

Група проводници за предаване на битове в паралел формират шина. Броят на проводниците в шината определя колко битова е тя. Например шина с 8 проводника се нарича 8-битова шина. Вероятно сте чували термините 8-битов, 16-битов, 32-битов, а напоследък и 64-битов компютър. Това е точно големината на шината, по която данните (битовете) се предават. Големината на шината за данни определя и количеството битове, което може да се обработва наведнъж от компютъра. Това количество битове се нарича още машинна дума на компютъра.

Преди да продължим напред, ще Ви запозная с още няколко термина. На следващата фигура е показан един байт с всичките 8 бита в него. Всеки бит си има номер. Най-десният бит има номер 0 и се нарича най-младши бит (**LSB – Least Significant Bit**) или само младши бит. Най-левият бит

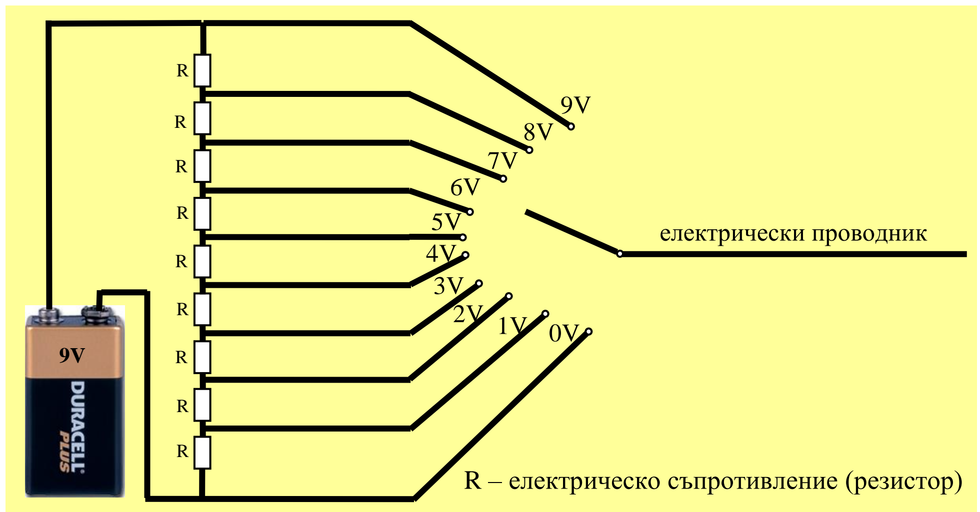


Фиг. 9 Битове в байт

има номер 7 и се нарича най-старши бит (**MSB –Most Significant Bit**) или само старши бит. Тази номерация на битовете е много важна, тъй като информацията, която се предава в даден бит, се интерпретира по различен начин, ако се предава в друг. Например байт със стойност 00000001 означава различна информация от байт със стойност например 10000000. Съвкупността от 4 бита се нарича още полубайт. Младшите четири бита (0÷3) в байт се наричат още младши полубайт, а старшите четири бита (4÷7) – старши полубайт.

В следващата глава ще разберете как точно се интерпретира двоичната информация (двоичните числа), закодирана в един или повече бита.

За да разберете защо представянето на информацията с двоични числа е по-лесна и удобна за обработка от хардуера в сравнение с десетичните числа, с които сме свикнали да работим в ежедневието, нека да разгледаме схема аналогична на Фиг.7, която позволява да генерираме всички десетични цифри. За по-лесно ще използваме източник на напрежение 9V, с който можем да генерираме всичките 10 десетични цифри 0÷9 под формата на напрежения 0V÷9V. В зависимост от позицията на ключа, по проводника могат да се предадат десет различни стойности под формата на десет различни напрежения. Сега сравнете Фиг.10 с Фиг.7. Дори и да не разбирате от електротехника и електроника, би трябва-



Фиг. 10 Предаване на информация по електрически проводник в десетичен формат

ло да може да оцените факта, че реализацията на Фиг.10 с помощта на електронни елементи би била доста по-сложна от тази на Фиг.7.

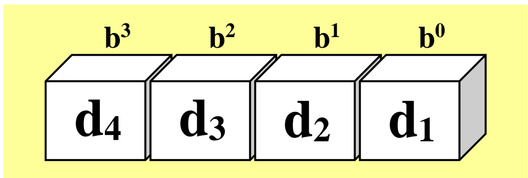
3 Бройни системи

3.1 Въведение

От предходната глава би трябвало вече да сте разбрали, че "езикът" на съвременните компютри се състои само от числа, които са комбинации от нули и единици. Тези числа се наричат двоични, тъй като се изграждат само с помощта на тези две цифри. Използването само на два символа (0 и 1) за представяне на числа се нарича **двоична бройна система**. Бройната система за представяне на числата в ежедневието на хората се нарича **десетична бройна система**. Както може би вече се досещате, тя се нарича така, защото използва 10 символа (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) за представяне на всякакви числа. Освен тези две бройни системи, в компютърната техника се използва широко още една бройна система – **шестнайсетична бройна система**. Тя използва 16 символа (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F).

Броят на символите, използвани от една бройна система, се нарича основа или база **b** (**radix** или **base**) на бройната система. Двоичната бройна система е с основа b = 2, десетичната – b = 10, и шестнайсетичната – b = 16.

Позицията на всяка цифра в едно число (важи и за трите бройни системи) има параметър, наричан тегло. Най-дясната цифра има тегло b^0, следващата цифра вляво от нея има тегло b^1 и т.н. Например, ако имаме едно четири цифрено число $d_4d_3d_2d_1$, отделните цифри имат тегло както е показано по-долу.



Изброените бройни системи се наричат още позиционни

бройни системи, тъй като позицията, в която се намира даден символ определя неговата стойност. Стойността на даден символ в едно число може да се изчисли като се умножи символа по неговото тегло. Стойността на едно число може да бъде изразена в десетична бройна система чрез сумата на стойностите на отделните символи.

Пример:

$$d_4d_3d_2d_1 = d_1*b^0 + d_2*b^1 + d_3*b^2 + d_4*b^3$$

Различните бройни системи използват общи символи, което може да доведе до грешно тълкуване в коя бройна система е дадено число. Когато е възможна такава ситуация, след най-дясната цифра се поставя долен индекс, който указва в коя бройна система е числото.

Пример:

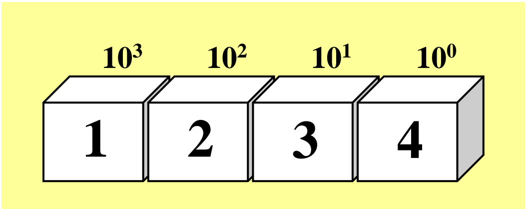
1010_{10} - числото е в десетична бройна система;
1010_{2} - числото е в двоична бройна система;
1010_{16} - числото е в шестнайсетична бройна система.

3.2 Десетична бройна система

Десетичната бройна система използва 10 символа (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) за представяне на всякакви числа и има основа b = 10. Съответно тегловните коефициенти са 10^0, 10^1, 10^2, 10^3, 10^4, 10^5, 10^6...

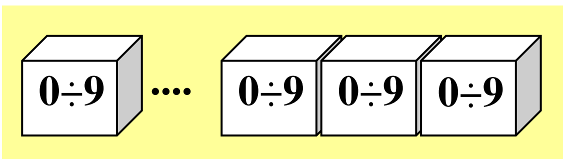
Пример:

Десетичното число 1234 може да се изрази и по следния начин:



$1234 = 4*10^0 + 3*10^1 + 2*10^2 + 1*10^3 = 4 + 30 + 200 + 1000 = 1234$

Въпреки че това е бройната система, която познаваме добре, ще разгледаме как става десетичното броене, тъй като това ще Ви помогне да разберете как се извършва броенето и в другите бройни системи. Всяка позиция в едно десетично число може да се променя от 0 до 9 (Фиг.11).



Фиг. 11 Стойности на десетичните позиции

Когато една позиция има стойност 9 и добавим 1 към нея, стойността в позицията се нулира, а следващата позиция се увеличава с единица. Табл.1 онагледява този процес.

0 + 1	90 + 1	990 + 1
1 + 1	91 + 1	991 + 1
2 + 1	92 + 1	992 + 1
3 + 1	93 + 1	993 + 1
4 + 1	94 + 1	994 + 1
5 + 1	95 + 1	995 + 1
6 + 1	96 + 1	996 + 1
7 + 1	97 + 1	997 + 1
8 + 1	98 + 1	998 + 1

9 + 1	99 + 1	999 + 1
10	100	1000
...	...	...

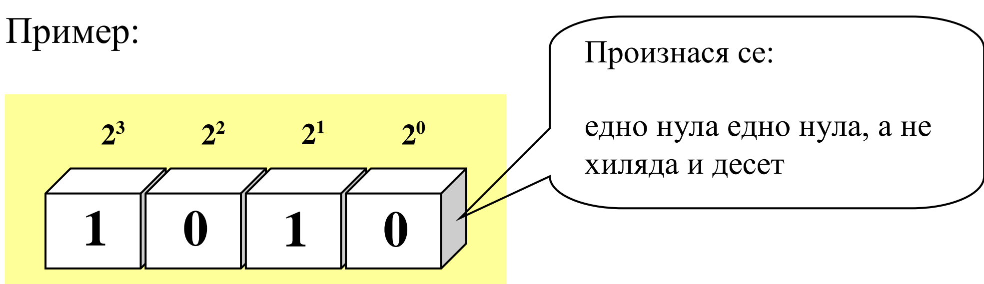
Табл. 1 Десетично броене

Максималният брой десетични числа, които могат да се формират от n- цифрено десетично число, е 10^n. Например с 3-цифрено десетично число могат да се представят 10^3 = 1000 на брой числа (от 0 до 999).

3.3 Двоична бройна система

Двоичната бройна система използва два символа (0 и 1) за представяне на всякакви числа и има основа b = 2. Съответно тегловните коефициенти са 2^0, 2^1, 2^2 , 2^3 , 2^4 , 2^5 , ...

Пример:



Двоичното число 1010 може да се изрази в десетична бройна система по следния начин:

$1010_2 = 0*2^0 + 1*2^1 + 0*2^2 + 1*2^3 = 0 + 2 + 0 + 8 = 10_{10}$

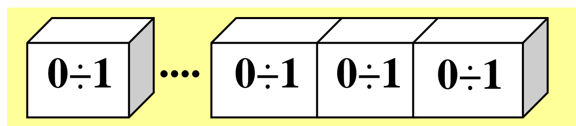
т.е. . двоичното число 1010 отговаря на десетичното число 10.

Табл.2 показва двоичните еквиваленти на десетичните цифри от 0 до 9.

десетична бройна система	двоична бройна система
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001

Табл. 2 Двоични еквиваленти на десетичните цифри от 0 до 9

Броенето в двоична бройна система е аналогично на десетичната. Всяка позиция в едно двоично число може да се променя от 0 до 1 (Фиг.12).



Фиг. 12 Стойности в двоичните позиции

Когато една позиция има стойност 1 и добавим 1 към нея, стойността в позицията се нулира, а следващата позиция се увеличава с единица. Табл.3 онагледява този процес.

0 + 1	1011 + 1
1 + 1	1100 + 1
10 + 1	1101 + 1
11 + 1	1110 + 1
100 + 1	1111 + 1
101 + 1	10000 + 1

110 + 1	10001 + 1
111 + 1	10010 + 1
1000 + 1	10011 + 1
1001 + 1	10100 + 1
1010 + 1	10101 + 1

Табл. 3 Двоично броене

Максималният брой двоични числа, които могат да се формират от n- цифрено двоично число, е 2^n. Например с 8-цифрено двоично число могат да се представят 2^8 = 256 на брой двоични числа (от 0 до 255 десетично). Всяка двоична цифра носи 1 бит информация. Термините **n-цифрено двоично число** и **n-битово число** са еквивалентни.

Обърнете внимание, че едно десетично число може да бъде представено като двоично чрез различен брой битове. Например числото 4 може да се кодира с минимум 3 бита, т.е. . като 100 (вижте Табл.2). Същият ще е резултатът, ако числото се кодира с 4, 8, 16 или повече бита, т.е. .

0100
00001000
00000000 00001000

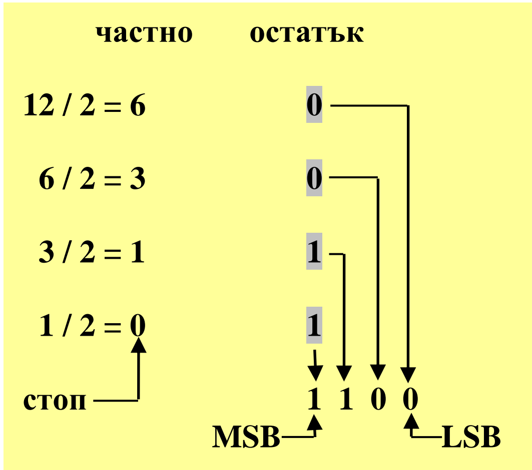
Това лесно може да го проверите, ако се опитате да преобразувате всяко едно от горните двоични числа в техния десетичен еквивалент по познатия вече начин. Например:

$0100 = 0*2^0 + 0*2^1 + 1*2^2 + 0*2^3 = 0 + 0 + 4 + 0 = 4$

Всички битове вляво от последния ненулев бит в едно двоично число не променят неговата стойност.

Едно десетично число може да бъде преобразувано в двоично чрез последователно деление на 2. Например, за да

преобразуваме десетичното число 12 в двоично, започваме с деление на 12 на 2. След това делим всяко получено частно на 2, докато частното стане 0. Остатъците, получени от всяко деление, формират двоичния еквивалент на десетичното число. Остатъкът, получен от първото деление, представлява младшия бит (LSB), а остатъкът, получен от последното деление, е старшият бит (MSB). Фиг.13 илюстрира този процес нагледно:



Фиг. 13 Преобразуване на десетично число в двоично

3.4 Шестнадесетична бройна система

Когато започнете да пишете C програми, ще се убедите, че използването на десетични числа в кода често е твърде неудобно. Тогава на помощ идва шестнайсетичната бройна система, която позволява големи числа да се запишат в по-компактен вид. Шестнайсетичната бройна система използва шестнайсет символа (0 ÷ 9 и A ÷ F или a ÷ f) за представяне на всякакви числа и има основа b = 16. Съответно тегловните коефициенти са 16^0, 16^1, 16^2 , 16^3 , 16^4 , 16^5 , 16^6...

Пример:

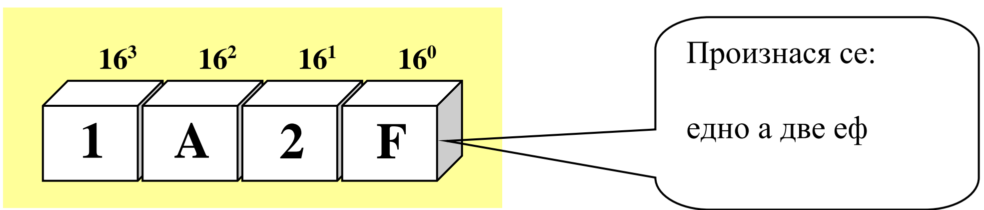


Табл.4 показва десетичните и двоичните еквиваленти на шестнайсетичните цифри 0 ÷ F.

шестнайсетична бройна система	десетична бройна система	двоична бройна система
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A (или a)	10	1010
B (или b)	11	1011
C (или c)	12	1100
D (или d)	13	1101
E (или e)	14	1110
F (или f)	15	1111

Табл. 4 Десетични и двоични еквиваленти на шестнайсетичните цифри от 0 ÷F

Шестнайсетичното число 1A2f може да се изрази в десетична бройна система по следния начин:

$1A2F = F*16^0 + 2*16^1 + A*16^2 + 1*16^3 = 15*16^0 + 2*16^1 +$

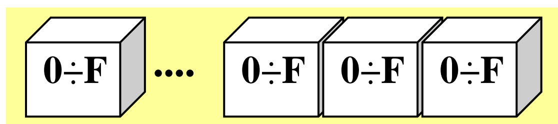
$10*16^2 + 1*16^3 = 6703_{10}$

т.е. . шестнайсетичното число 1A2F е еквивалентно на десетичното число 6703.

Шестнайсетичните числа дават по-ясна представа за битовото представяне на числото, в сравнение с десетичните. Например, знаейки двоичните еквиваленти на шестнайсетичните цифри, лесно може да определите, че в числото 1A2F младшите четири бита са 1111, следващите четири бита са 0010, следващите четири бита са 1010 и старшите четири бита са 0001.

Шестнайсетичните числа често се записват с представката 0x или 0X, например 1A2F е същото като **0x**1A2F или **0X**1A2F, или с наставката h или H, например 1A2F**h** или 1A2F**H.**

Броенето в шестнайсетична бройна система е аналогично на десетичната. Всяка позиция в едно шестнайсетично число може да се променя от 0 до F (или f) (Фиг.14).



Фиг. 14 Стойности в шестнайсетични позиции

Когато една позиция има стойност F и добавим 1 към нея, стойността в позицията се нулира, а следващата позиция се увеличава с единица. Табл.5 онагледява този процес.

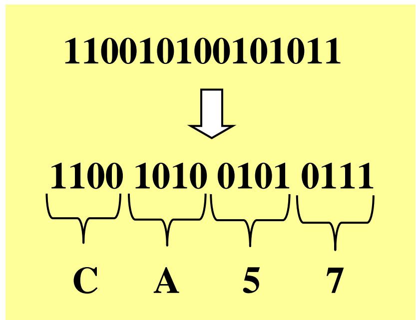
0 + 1	9 + 1	12 + 1	1B + 1
1 + 1	A + 1	13 + 1	1C + 1
2 + 1	B + 1	14 + 1	1D + 1
3 + 1	C + 1	15 + 1	1E + 1

4 + 1	D + 1	16 + 1	1F + 1
5 + 1	E + 1	17 + 1	20 + 1
6 + 1	F + 1	18 + 1	21 + 1
7 + 1	10 + 1	19 + 1	...
8 + 1	11 + 1	1A + 1	...

Табл. 5 Шестнайсетично броене

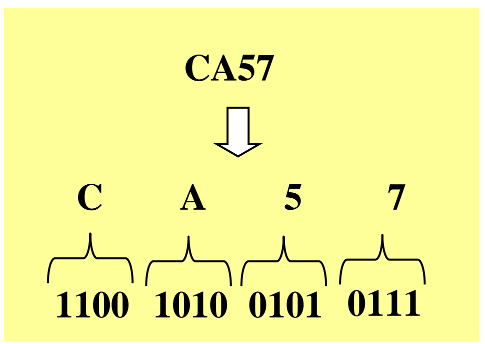
Използвайки информацията от Табл.4, лесно може да преобразувате едно шестнайсетично число в двоично и обратно.

За да преобразувате едно двоично число в шестнайсетично, разделете битовете на двоичното число на групи по 4 бита, започвайки от младшия бит. След това преобразувайте всяка група в шестнайсетичния еквивалент. Фиг.15 илюстрира този процес нагледно.



Фиг. 15 Преобразуване на двоично число в шестнайсетично

За да преобразувате едно шестнайсетично число в двоично, преобразувайте всяка шестнайсетична цифра в 4-битово двоично число (Фиг.16).



Фиг. 16 Преобразуване на шестнайсетично число в двоично

3.5 Заключение

Преди да продължим напред нека да направим едно обобщение по отношение на употребата на трите бройни системи. Хардуерът на един компютър е реализиран така, че да използва двоична бройна система (вижте отново Фиг.3, 4 и 5). Двоичната система обаче в повечето случаи е неудобна за употреба от човек, особено при големи числа. Затова когато се пишат програми, се използват десетични и шестнайсетични числа, които в крайна сметка се преобразуват в двоични преди програмата да се зареди в компютъра за изпълнение.

Следващата таблица показва десетичните числа от 1 до 15 и техните двоични и шестнайсетични еквиваленти. Научете я наизуст.

десетична бройна система	**двоична бройна система**	**шестнайсетична бройна система**
0	0000	0
1	0001	1
2	0010	2

3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A (или a)
11	1011	B (или b)
12	1100	C (или b)
12	1100	C (или c)
13	1101	D (или d)
14	1110	E (или e)
15	1111	F (или f)

Табл. 6 Сравнителна таблица на десетични, двоични и шестнайсетични числа

4 Представяне на знакови цели числа в двоична бройна система.

4.1 Въведение

В точка 3.3 Двоична бройна система разгледахме представяне само на беззнакови цели числа в двоична бройна система. Двоичната бройна система може да се използва и за представяне на знакови цели числа (отрицателни и положителни). Съществуват различни формати за представяне на знакови цели числа в двоична бройна система. По известните сред тях са прав код (**signed-bit magnitude**), обратен код (**1’s complement**), допълнителен код (**2’s complement**) и код с отместване (или излишък) (**biased** или **excess**).

4.2 Представяне на знакови цели числа в прав код

При правия код положителните и отрицателните числа се представят по същия начин както беззнаковите. Най-старшият бит се заделя за знаков бит. Стойност 0 в този бит указва, че числото е положително, а стойност 1 – отрицателно. Останалите битове се използват за кодиране на големината на числото. Недостатък на правия код е, че нулата има двойно представяне: +0 и -0.

Табл.7 показва представянето на 8-битови знакови числа в прав код.

+127	01111111
+126	01111110
+125	01111101
...	...

+2	00000010
+1	00000001
+0	00000000
-0	10000000
-1	10000001
-2	10000010
...	...
-125	11111101
-126	11111110
-127	11111111

Табл. 7 Представяне на 8-битови знакови числа в прав код

4.3 Представяне на знакови цели числа в обратен код

При обратния код положителните числа се представят по същия начин както беззнаковите. Отрицателните числа се представят чрез инвертиране на положителния еквивалент на числото (всички нули се заменят с 1, а единиците с 0). Например +9 се представя като 00001001, а -9 като 11110110. Както и при правия код, обратният код има недостатък, че нулата има двойно представяне: +0 и -0. Табл.8 показва представянето на 8-битови знакови числа в обратен код.

+127	01111111
+126	01111110
+125	01111101
...	...
+2	00000010
+1	00000001

+0	00000000
-0	11111111
-1	11111110
-2	11111101
...	...
-125	10000010
-126	10000001
-127	10000000

Табл. 8 Представяне на 8-битови знакови числа в обратен код

4.4 Представяне на знакови цели числа в допълнителен код

При допълнителния код положителните числа се представят по същия начин както беззнаковите. Отрицателните числа се представят като положителният еквивалент на числото се преобразува в обратен код и към резултата се добави 1. Например +9 се представя като 00001001, а -9 се представя като:

11110110 (обратен код на +9)
\+
00000001
____
11110111 (допълнителен код на -9)

Допълнителният код има следните предимства по отношение на правия и обратния код:

- нулата има само едно представяне (00000000);
- операцията изваждане може да се извърши с помощта на операцията събиране[1], т.е. . една и съща апаратна

част на процесора може да се използва за извършване на операциите изваждане и събиране.

Забележка[1]: Ако не разбирате това, не се притеснявайте. Компютърът извършва това скрито от Вас. За експеримент разгледайте следния пример:

7 – 3 = 7 + (-3) = 4

```
1 1 1 1     <— пренос
  0 1 1 1 (7)
 +
  1 1 0 1 (-3 в допълнителен код)
 -----------
1 0 1 0 0 (4)
^— преносът от най-старшите битове се игнорира
```

Както сами виждате, извършва се операция събиране, а резултатът е същият като изваждане на 7 – 3 (преносът, получен при сумирането на най-старшите битове, се изхвърля).

В 6.1 Цифрова аритметика с цели числа ще Ви запозная по-подробно с аритметиката с двоични числа.

Табл.9 показва представянето на 8-битови знакови числа в допълнителен код.

+127	01111111
+126	01111110
+125	01111101
...	...
+2	00000010
+1	00000001
0	00000000
-1	11111111
-2	11111110

...	...
-126	10000010
-127	10000001
-128	10000000

Табл. 9 Представяне на 8-битови знакови числа в допълнителен код.

Както се вижда от Табл.9, допълнителният код позволява в едно 8-битово число да се кодира пълния брой от 256 (2^8) възможни числа със знак (-128 ÷ +127).

Допълнителният код е най-предпочитаният код за представяне на отрицателни числа.

4.5 Представяне на знакови цели числа в код с отместване

В код с отместване (нарича се още код с излишък) всички числа се третират като беззнакови. Действителното число се получава, като от беззнаковото число се извади отместването. Например 8-битовият диапазон от знакови числа -128 ÷ +127 може да се представи в 8-битов код с отместване като за отместване се използва числото 128, т.е. . всички числа от диапазона се преместят с 128 позиции нагоре. Така всички отрицателни числа се превръщат в положителни. Например +9 се представя като 9 + 128 = 137 (10001001), а -9 се представя като -9 + 128 = 119 (01110111). Табл.10 показва представянето на 8-битови знакови числа в код с отместване.

8-битов диапазон от знакови числа	Код с излишък в десетичен формат	Код с излишък в двоичен формат
+127	255	11111111

+126	254	11111110
...	...	...
+2	130	10000010
+1	129	10000001
0	128	10000000
-1	127	01111111
-2	126	01111110
...	...	...
-127	1	00000001
-128	0	00000000

Табл. 10 Представяне на 8-битови знакови числа в код с отместване.

5 Представяне на десетични дробни числа в двоична бройна система.

5.1 Въведение

Освен цели десетични числа, двоичната бройна система може да представя и дробни десетични числа. Двоичните дробни числа могат да се представят по два начина: двоични дроби с фиксирана запетая[1] (**fixed-point**) и двоични дроби с плаваща запетая (**floating-point**).

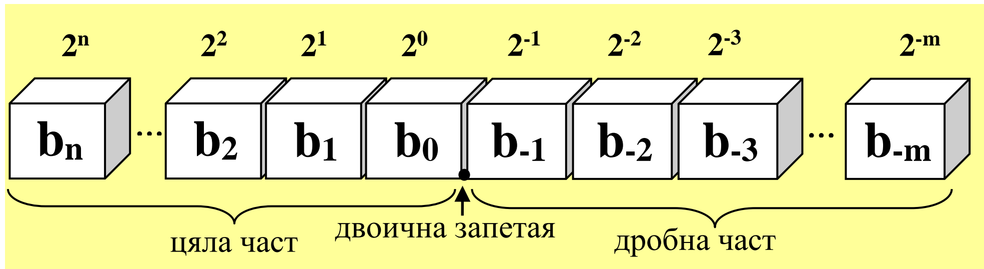
Забележка[1]: Въпреки че използвам термина "запетая", за разделителен символ между цялата и дробната част в англоезичната литература се използва символа точка. Затова въпреки че казвам запетая, ще използвам точка за разделител на цялата и дробната част.

5.2 Дробни числа с фиксирана запетая

В тази точка ще Ви покажа какво представляват двоичните дробни числа с фиксирана запетая. Този начин на представяне на десетичните дроби намира широко приложение в т.нар. **Цифрови Сигнални Процесори (DSP-Digital Signal Processor).** DSP-микропроцесорите са специализирани да извършват сложни аритметични изчисления възможно най-бързо. Те се използват широко в приложения за обработка на аудио и видео информация. Съществуват и микроконтролери, които също могат да извършват някои DSP-операции.

В системи, които използват фиксирана запетая, всички числа имат еднакъв брой цифри и запетаята винаги се намира на едно и също място, т.е. . позицията на запетаята е фиксирана. Например в десетична бройна система числата 2.34, 4.56, 0.98 се състоят от 3 цифри и запетаята е

разположена на две позиции вляво. Друг пример, но в двоична бройна система, са числата 11.10, 10.11, 01.01. Всички те се състоят от четири цифри, а двоичната запетая е разположена две позиции вляво. Фиг.17 показва тегловните коефициенти на цялата и дробната част на двоично число с фиксирана запетая.



Фиг. 17 Тегловни коефициенти на число с фиксирана запетая

За да преобразувате такова число в десетичен вид, е необходимо да съберете стойностите на всяка позиция, т.е. .

$b_n \ldots b_2 b_1 b_0 . b_{-1} b_{-2} b_{-3} \ldots b_{-m} = b_n x2^n + \ldots + b_2 x2^2 + b_1 x2^1 + b_0 x2^0$

$+ b_{-1} x2^{-1} + b_{-2} x2^{-2} + b_{-3} x2^{-3} + \ldots + b_{-m} x2^{-m}$

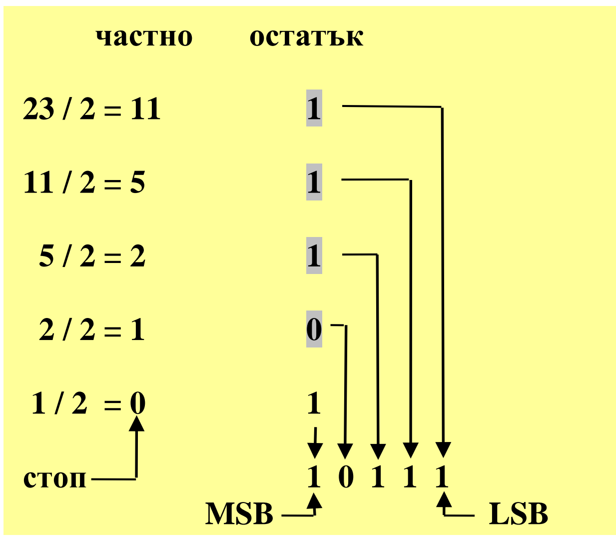
Например:

$10111.011 = 1x2^4 + 0x2^3 + 1x2^2 + 1x2^1 + 1x2^0 + 0x2^{-1} + 1x2^{-2} +$

$1x2^{-3} = 16 + 0 + 4 + 2 + 1 + 0 + 0.25 + 0.125 = 23.375$

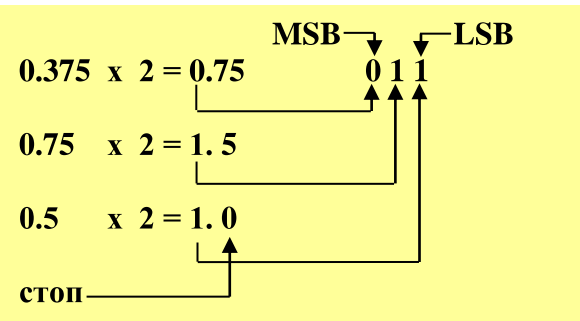
Най-лесния начин да преобразувате едно десетично дробно число в двоично дробно число с фиксирана запетая е да преобразувате поотделно цялата и дробната част. Нека да вземем числото 23.375 = 23 + 0.375. В точка 3.3 Двоична бройна система Ви показах как да преобразувате цели десетични числа в двоична бройна система (Фиг.18).

Дробната част на едно десетично число се преобразува като



Фиг. 18 Преобразуване на цялата част 23 в двоично число

се умножи по 2. След това дробната част на резултата отново умножаваме по 2 и така, докато дробната част на резултата стане 0. Целите части на всеки получен резултат формират дробния двоичен еквивалент на дробната десетична част на числото. Първата цяла част представлява най-старшия бит MSB, а последната цяла част представлява най-младшия бит LSB. Фиг.19 илюстрира този процес нагледно.



Фиг. 19 Преобразуване на дробната част 0.375 в двоично число

$23.375_{10} = 10111.011_2$

Обърнете внимание, че не всяка десетична дробна част може

да се представи точно в двоичен вид. Например нека да вземем числото 0.2 и да се опитаме да го преобразуваме в двоично:

0.2 x 2 = 0.4
0.4 x 2 = 0.8
0.8 x 2 = 1.6
0.6 x 2 = 1.2
0.2 x 2 = 0.4 → целият процес се повтаря отново
...

0.2 = 0.00110....≈ 0 + 0 + 0.125 + 0.0625 + 0 ≈ 0.1875

За такива числа, колкото повече битове се използват за дробната част, толкова резултатът ще бъде по-точен, но никога няма да може да представи абсолютно точно десетичната дробна част. Това означава, че в компютърните системи с фиксирана запетая, които използват краен брой битове за съхраняване на числата, представянето ще бъде приблизително.

В действителност целите двоични числа представляват числа с фиксирана запетая, където запетаята се намира след най-десния бит. Например десетичното число 9_{10} може да се представи като двоично число с фиксирана запетая по следния начин:

1001.

Когато преместваме двоичната запетая наляво с една позиция, това е равносилно на деление на 2, а при преместване надясно - на умножение по 2. Направете аналогия с десетичната запетая. При нея преместването наляво е равносилно на деление на 10, а надясно - на умножение по 10. Например:

1001. → (9)

100.1 → (9/2)
10.01 → (9/4)
1.001 → (9/8)
0.1001 → (9/16)

Обърнете внимание, че точността, с която могат да се представят двоичните числа с фиксирана запетая, е 2^{-m}, където m е броят на битовете след двоичната запетая. Например ако m = 1, точността, с която могат да се представят числата е 0.5, т.е. .

..., -1.5, -1.0, 0, +0.5, +1.0, +1.5,...

Важно е да разберете, че двоичната запетая не се представя по някакъв начин в една компютърна система, а се подразбира, че съществува. Най-честото обозначаване на формати с фиксирана запетая е **Qi.f**, където i отбелязва броя на битовете на цялата част, а f - броя на битовете на дробната част, например Q1.15.

За представяне на отрицателни числа с фиксирана запетая най-често се използва допълнителен код. Положителният двоичен еквивалент на числото се преобразува първо в обратен код, след което към резултата се добавя 1. Следващият пример илюстрира описания процес на базата на числото -23.375, използвайки формат Q8.4. Фиг.19 и 20 показват преобразуването на положителния еквивалент на това число в двоичен еквивалент с фиксирана запетая, т.е. . $23.375_{10} = 00010111.0110_2$. Обратният код на това число е 11101000.1001. Към най-младшия бит на това число добавяме 1 и получаваме допълнителния код на -23.375, т.е. .

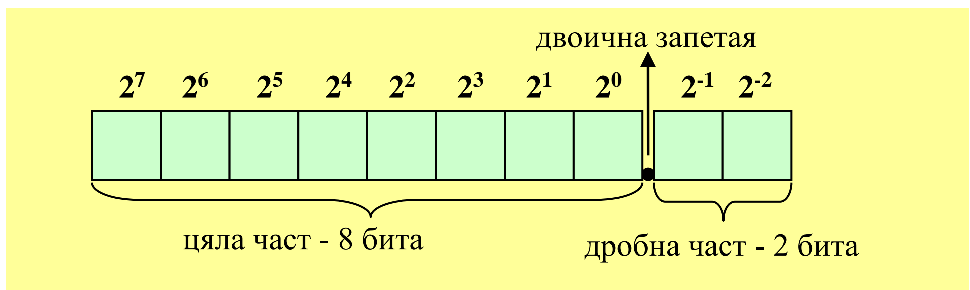
  11101000.1001 (обратен код на 23.375)
\+ 1
____
  11101000.1010 (-23.375 в допълнителен код)

Проверка:

```
 00010111.0110    (+23.375)
+
 11101000.1010    (-23.375)
 ______________________________
100000000.0000          (0)
```

(Първата цифра 1 на резултата е зачертана.)

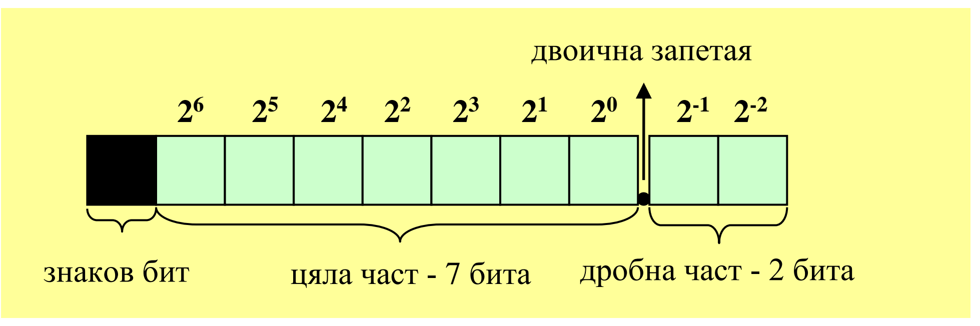
Диапазонът от стойности на беззнакови числа с фиксирана запетая е **$0 \div 2^i - 2^{-f}$ със стъпка 2^{-f}**. Следващата фигура показва беззнаково число във формат Q8.2. Такова число може да представи стойности в диапазона 0 ÷ 255.75 ($2^8 - 2^{-2}$) със стъпка 0.25 (2^{-2}), т.е. . 0.00, 0.25, 0.50, 0.75, 1.00, 1.25,..., 254.00, 254.25, 254.50, 254.75, 255.00, 255.25, 255.50, 255.75.



Фиг. 20 Беззнаково число с фиксирана запетая във формат Q8.2

Диапазонът от стойности на знакови числа с фиксирана запетая е **$-2^{i-1} \div 2^{i-1} - 2^{-f}$ със стъпка 2^{-f}**. Следващата фигура показва знаково число във формат Q8.2. Такова число може да представи стойности в диапазона -128 (-2^{8-1}) ÷ +127.75 ($2^{8-1} - 2^{-2}$) със стъпка 0.25 (2^{-2}), т.е. . -128.00, -127.75, -127.50, -127.25,..., -1.00, -0.75, -0.50, -0.25, 0.00, 0.25, 0.50, 0.75, 1.00,..., 126.00, 126.25, 126.50, 126.75, 127.00, 127.25, 127.50, 127.75.

Табл.11 показва стъпката (точността), с която могат да се представят числата с фиксирана запетая в зависимост от броя на битовете, заделени за дробната част:



Фиг. 21 Знаково число с фиксирана запетая във формат Q8.2

битове в дробната част	стъпка	
0	2^{0}	1
1	2^{-1}	0.5
2	2^{-2}	0.25
3	2^{-3}	0.125
4	2^{-4}	0.0625
5	2^{-5}	0.03125
6	2^{-6}	0.015625
7	2^{-7}	0.0078125
8	2^{-8}	0.00390625
9	2^{-9}	0.001953125
10	2^{-10}	0.0009765625
11	2^{-11}	0.00048828125
12	2^{-12}	0.000244140625
13	2^{-13}	0.0001220703125
14	2^{-14}	0.00006103515625
15	2^{-15}	0.000030517578125
16	2^{-16}	0.0000152587890625
17	2^{-17}	0.00000762939453125
18	2^{-18}	0.000003814697265625
19	2^{-19}	0.0000019073486328125

Табл. 11 Стъпка на представяне на числа с фиксирана запетая

5.3 Дробни числа с плаваща запетая

5.3.1 Въведение

Нека предположим, че е необходино да използваме много големи и много малки числа, такива като 5 432 678 123 456, 0.000 000 000 000 432 196 или 5 432 678 123 456.000 000 000 000 462 196. Ако трябва да съхраним тези числа в паметта на компютъра под формата на числа с фиксирана запетая (особено последното число), е необходимо да използваме голям брой битове. Такива числа се представят по-лесно като числа с плаваща запетая. Числото 5 432 678 123 456 може да се представи като 5.432 678 123 456 x 10^{12}, а числото 0.000 000 000 000 432 196 като 4.32 196 x 10^{-13}. Подобно числово преставяне използва две числа: мантиса (**mantissa** или **significand**) и експонента. Така числото 4.32 196 x 10^{-13} използва мантиса 4.32 196 и експонента -13. За да добиете представа какво се има предвид под "плаваща запетая", вземете десетичните числа +**6.0247x10^{23}**, +**3.7291x 10^{-27}**, **−1.0341x10^{2}** и т.н. Ние казваме, че всички тези числа имат 5 значещи цифри на точност. Скалиращият фактор 10^{23}, 10^{-27} и т.н., определя действителната позиция на десетичната запетая по отношение на значещите цифри, т.е. . запетаята „плава“ в зависимост от скалиращия фактор. Забележете начина, по който са представени числата. Това се нарича научна нотация. Научната нотация се използва от учени и инженери за представяне на много големи и много малки числа в компактен вид. Числата, представени по този начин, се състоят от следните компоненти:

- Знак **S (Sign)**: + или -
- Мантиса **M (Mantissa** или **Significand)**: 6.0247, 3.7291, 1.0341
- Експонента **E (Exponent):** 23, -27, 2;

Използвайки научна нотация, едно и също число може да се представи по няколко различни начина. Например:

$+60247x10^{19} = +6024.7x10^{20} = +602.47x10^{21} = +60.247x10^{22} = +6.0247x10^{23}$

Последното число се нарича **нормализирано**. Нормализирани числа са тези, които спазват условието

$$\mathbf{1 \leq M < b}$$

където

M – мантисата на числото (в нашия пример M = 6.0247)
b – база на използваната бройна система (в нашия пример b = 10)

Използването на нормализирани числа улеснява операциите с плаваща запетая.

Обобщената форма на запис на дестетични числа с плаваща запетая в научна нотация е:

$$\mathbf{Fp(10) = \pm M \times 10^{E}}$$

5.3.2 Стандартът IEEE 754

5.3.2.1 Общи сведения

Има много стандарти с плаваща запетая. Те се различават по начина на съхранение в паметта на компютъра на изброените по-горе компоненти на числата. Най-използван е стандартът **IEEE 754**. Той предоставя следните основни формати:

- единична точност (**single precision**);
- двойна точност (**double precision**).

IEEE 754 представя следните стойности:

- Нормализирани числа;
- Нула;
- Денормализирани числа;
- Безкрайност;
- NaNs (**Not a Number** – Не е число).

5.3.2.2 Нормализирани числа

Нормализираните двоични дробни числа се представят в следната форма:

$$\mathbf{Fp(2) = (-1)^{S} \times 1.F \times 2^{E}}$$

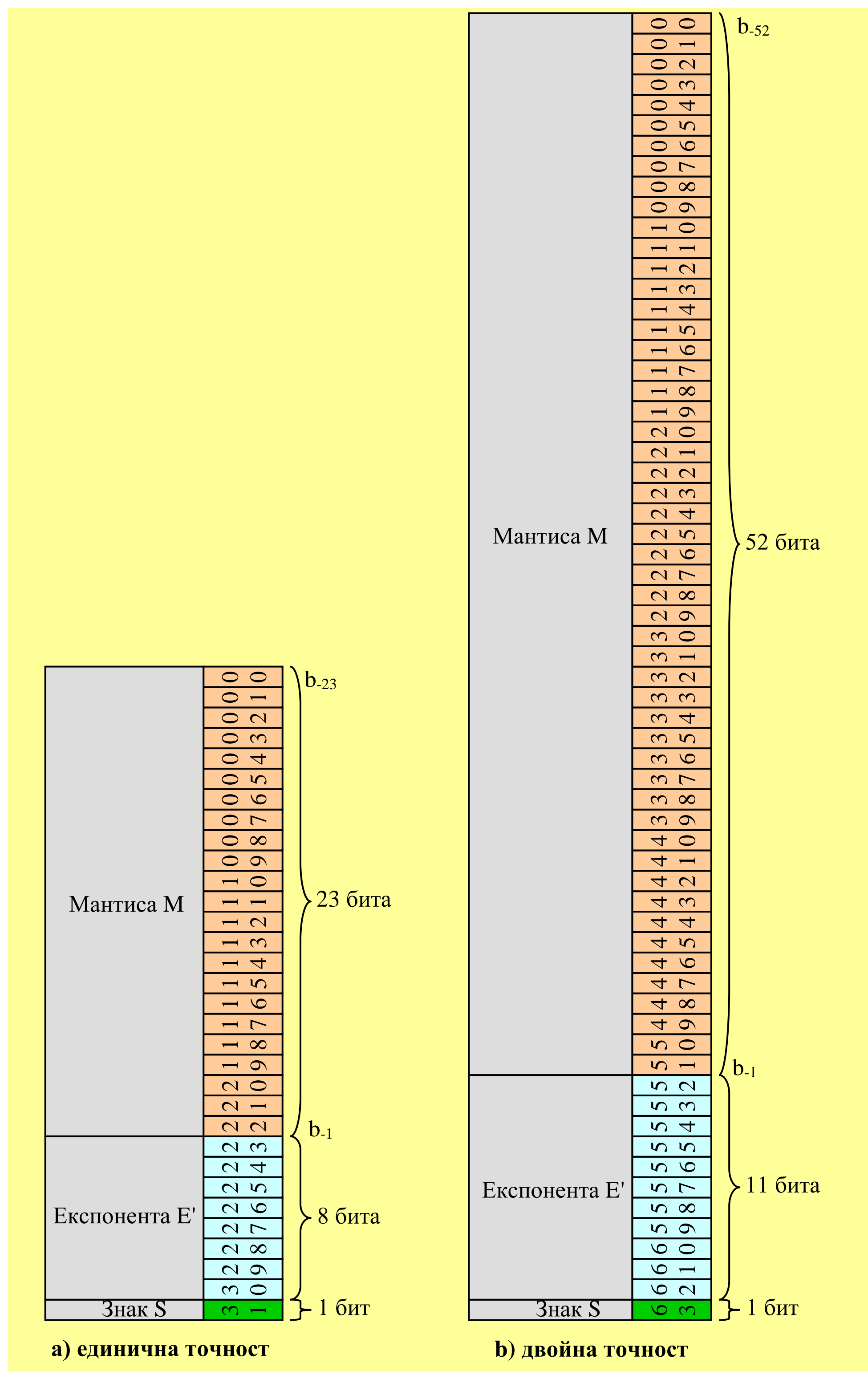
където

S – знак, определя дали числото е положително (S = 0) или отрицателно (S = 1).

F – дробна част на мантисата (**Fraction**).

E – експонента, която определя действителната позиция на двоичната запетая.

Фиг.22 илюстрира битовото представяне на числата с плаваща запетая с единична и двойна точност в IEEE 754



Фиг. 22 Формати с плаваща запетая в IEEE 754

стандарта.

Представянето на мантисата във формата $1.F = 1.\ b_{-1}\ b_{-2\dots}\ b_{-23}$ (b_{-52}) се нарича нормализирана форма. Стойността на значещите цифри се изчислява по обичайния начин:

$$\mathbf{M = 1x2^{0} + b_{-1}x2^{-1} + b_{-2}x2^{-2} + , \dots ,+ b_{-23}x2^{-23} (b_{-52}x2^{-52})}$$

Водещата единица и двоичната запетая след нея на мантисата съществуват винаги и затова не участват в действителното битово представяне, но се подразбират, т.е. . в паметта на компютъра се съхранява само дробната част F на мантисата.

Есkпонентата E се съхранява в паметта в код с излишък. Това позволява експонентата да се съхранява винаги като положително число. Действителната стойност на експонентата се получава като от съхранената в паметта експонента E' се извади добавеното отместване.

E = E' – 127 за числа с единична точност
E = E' – 1023 за числа с двойна точност

Следващият пример демонстрира преобразуване на десетично дробно число в двоично число с плаваща запетая с единична точност.

Пример: -12.4375

1. Преобразуване в двоичен вид

   Принципите на преобразуване в двоичен вид са разгледани в 5.2 Дробни числа с фиксирана запетая затова тук ще прескоча тези обяснения.

   $-12.4375_{10} = -1100.0111_{2} = -1100.0111_{2} \times 2^{0}$

2. Нормализиране на двоичното число

$-1100.0111 \times 2^{0} = -1.1000111 \times 2^{3} = (-1)^{1} \times 1.1000111 \times 2^{3}$

3. Представяне на числото в паметта на компютъра

Тъй като числото е отрицателно, знаковият бит S е 1. Изчислява се експонентата $E' = E + 127 = 3 + 127 = 130_{10} = 10000010_{2}$. Попълват се битовете на мантисата (водещата единица не се записва).

1	1	0	0	0	0	0	1	0	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Може да използвате онлайн конвертора от следния линк:

http://www.binaryconvert.com/convert_float.html?decimal=049048046048

и да проверите получения резултат.

Следва описание на параметрите на нормализираните числа. Разсъжденията се отнасят за числа с единична точност, като в скоби и сиво каре са посочени съответните стойности за формат с двойна точност.

E'min = 0 (0) – минимална отместена експонента
E'max = 127 (2047) – максимална отместена компонента

Крайните стойности на отместената експонента E' (E'min и E'max) се използват за представяне на ±0, ±∞ и NaNs. Останалите междинни стойности се използват за представяне на нормализирани числа. Ще означа тази експонента като E'norm, т.е. . E'min < E'norm < E'max. Знаейки това, може да определим минималното **Fpminnorm** и максималното **Fpmaxnorm** нормализирано число, което може да се представи с единична и двойна точност съответно.

$Fpminnorm(2) = (-1)^S \times 1.000...0 \times 2^{Eminnorm}$

$Fpmaxnorm(2) = (-1)^S \times 1.111...1 \times 2^{Emaxnorm}$

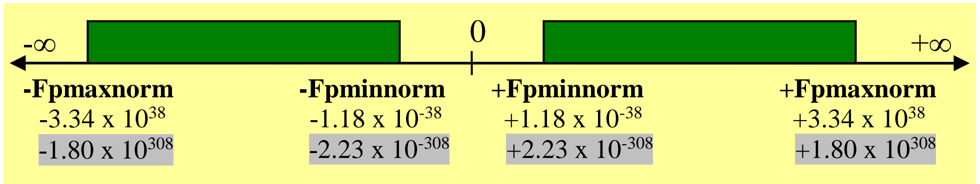
$Eminnorm = E'minnorm - 127\ (1023) = 1\ (1) - 127\ (1023) = -126\ (-1022)$

$Emaxnorm = E'maxnorm - 127\ (1023) = 254\ (2046) - 127\ (1023) = 127\ (1023)$

$Fpminnorm(2) = \pm 2^{-126(-1022)} \approx \pm 1.18\ (2.23) \times 10^{-38\ (-308)}$

$Fpmaxnorm(2) = \pm(2 - 2^{-23(-52)}) \times 2^{127(1023)} \approx \pm 3.4\ (1.80) \times 10^{38\ (308)}$

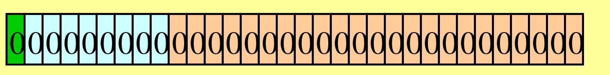
Фиг.23 показва диапазона на нормализираните числа с единична и двойна точност върху числовата ос.



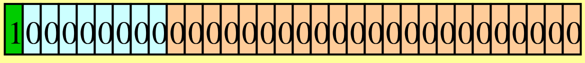
Фиг. 23 Диапазон на нормализираните числа

5.3.2.3 Представяне на нулата

За представяне на нулата се използва специална битова последователност: E' = 0 и F = 0. Заради знаковия бит S са възможни две нули: +0 и -0.



Фиг. 24 Представяне на +0 във формат с единична точност



Фиг. 25 Представяне на -0 във формат с единична точност

5.3.2.4 Денормализирани числа

Когато дробните числа станат много много малки (много близки до нулата), нормализираният числов формат не може повече да представя такива числа. Причината е, че експонентата не е достатъчно голяма да компенсира преместването на двоичната запетая надясно, за да се елиминират водещите нули. Такива числа се наричат денормализирани или субнормални.

Денормализираните двоични дробни числа се представят в следната форма:

$$\mathbf{Fp(2) = (-1)^{S} \times 0.F \times 2^{Edenorm}}$$

където

S – знак, определя дали числото е положително (S = 0) или отрицателно (S = 1).

F – дробната част на мантисата (**Fraction**).

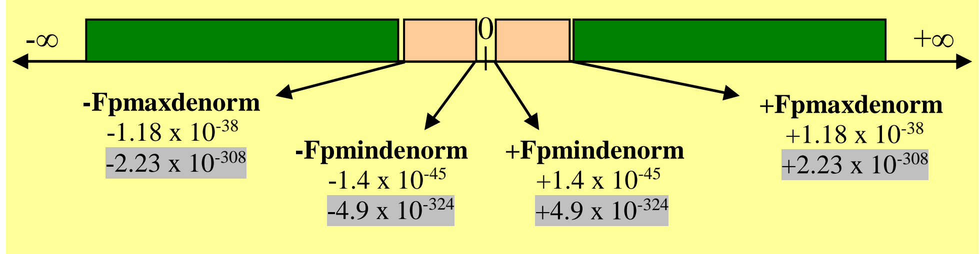
Edenorm – експонента, обозначаваща че числото е денормализирано.

Следва описание на параметрите на нормализираните числа. Разсъжденията се отнасят за числа с единична точност, като в скоби и сиво каре са посочени съответните стойности за формат с двойна точност.

$$\mathbf{Edenorm = E'denorm - 127(1023) = 0\ (0) - 127(1023) = -126(-1022)}$$

$$\mathbf{Fpmindenorm(2) = (-1)^{S} \times 0.000\ldots1 \times 2^{-126(-1022)} = \pm 2^{-149(-1074)} \approx \pm 1.4(4.9) \times 10^{--45(-324)}}$$

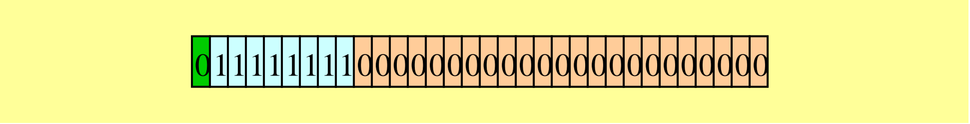
$$\mathbf{Fpmaxdenorm(2) = (-1)^{S} \times 0.111\ldots1 \times 2^{-126(-1022)} \approx \pm 2^{-126(-1022)} \approx \pm 1.18(2.23) \times 10^{-38\ (-308)}}$$



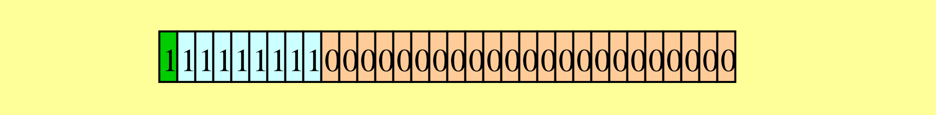
Фиг. 26 Диапазон на денормализираните числа

5.3.2.5 Безкрайност

За представяне на безкрайност се използва специална битова последователност: E' = 255(2047) и M = 0. Заради знаковия бит S са възможни две безкрайности: +∞ и -∞.



Фиг. 27 Представяне на +∞ във формат с единична точност



Фиг. 28 Представяне на -∞ във формат с единична точност

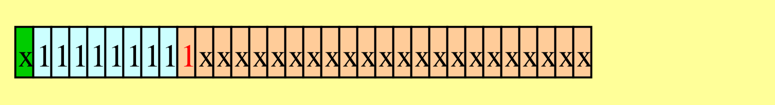
Следващата операция показва кога резултатът може да бъде ±∞.

±ненулево-число / 0 = ±∞

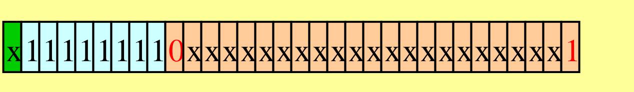
5.3.2.6 NaNs

Стойността NAN (Not a Number) се използва за представяне

на стойност, която не е число. Има две категории NaN: QNaN (Quiet NaN) и SNaN (Signaling NaN). За представянето им се използват специални битови последователности: E' = 255(2047) и M ≠ 0



Фиг. 29 Представяне на QNaN във формат с единична точност



Фиг. 30 Представяне на SNaN във формат с единична точност

QNaN отбелязва неопределени операции, докато SNaN отбелязва невалидни операции.

Следващите операции показват кога резултатът може да бъде NaN.

$$\pm 0 \,/\, \pm 0 = \mathbf{NaN}$$
$$\pm\infty \,/\, \pm\infty = \mathbf{NaN}$$
$$\pm\infty \times 0 = \mathbf{NaN}$$

5.3.2.7 Обобщение

Следващата таблица показва в резюме параметрите на числата с плаваща запетая по стандарта IEEE 754.

S	E'	F	Стойност
0	00..00	00..00	+0
0	00..00	00..01	Положително денормализирано число

		11..11	$+0.F \times 2^{-126(-1022)}$
0	00..01 11..10	xx..xx	Положително нормализирано число $+1.F \times 2^{E'-127(-1023)}$
0	11..11	00..00	$+\infty$
0	11..11	00..01 01..11	SNaN
0	11..11	10..00 11..11	QNaN
1	00..00	00..00	-0
1	00..00	00..01 11..11	Отрицателно денормализирано число $-0.F \times 2^{-126(-1022)}$
1	00..01 11..10	xx..xx	Отрицателно нормализирано число $-1.F \times 2^{E'-127(-1023)}$
1	11..11	00..00	$-\infty$
1	11..11	00..01 01..11	SNaN
1	11..11	10..00 11..11	QNaN

Табл. 12 IEEE 754 параметри

6 Цифрова аритметика

6.1 Цифрова аритметика с цели числа

Събиране

X + Y = Z

X – събираемо
Y – събираемо
Z – сума

Събирането на цели двоични числа е аналогично на събирането на цели десетични числа, дори е по-просто, тъй като двоичната бройна система се състои само от две цифри – 0 и 1. Следващата таблица описва всички възможни комбинации при събиране на двоични цифри.

комбинация		сума	пренос c_{out}
0 + 0	=	0	0
0 + 1	=	1	0
1 + 0	=	1	0
1 + 1	=	0	1

Табл. 13 Комбинации от събиране на двоични цифри

Нула, прибавена към коя да е цифра или число, дава самата цифра или число (вижте първите два реда на таблицата). Единица, прибавена към коя да е цифра или число, дава следващата по големина цифра или число (вижте последните два реда на таблицата). Ще разгледам последния ред малко по-подробно. При събиране на 1 с 1 резултатът е 10_2 (2 в десетична бройна ситема), т.е. . резулатният бит е 0 и се генерира пренос $\mathbf{c_{out}}$ = 1, който се добава към следващия по-старши бит (битът, който се намира вляво от текущия бит).

При събиране на битовете на две двоични числа се започва от най-младшите битове. Ако при това събиране се генерира пренос 1, той се добавя към сумата на следващите поред битове и т.н., докато се извърши събирането и на най-старшите битове на числата. Описаният процес може да се обобщи като $\mathbf{a_i + b_i + c_{in}}$ **и пренос** $\mathbf{c_{out}}$, където $\mathbf{a_i}$ и $\mathbf{b_i}$ са съответните битове на двоичните числа, $\mathbf{c_{in}}$ е преносът от сумата на предходните битове, а $\mathbf{c_{out}}$ е преносът от това събиране, който се явява $\mathbf{c_{in}}$ за следващите битове. При събиране на най-младшите битове $\mathbf{c_{in}}$ винаги е 0, т.е. . няма пренос. Табл.14 показва всички възможни комбинации при събирането на битовете на двоични числа и преноса $\mathbf{c_{in}}$.

a_i	b_i	c_{in}		сума	c_{out}
0	0	0	=	0	0
0	0	1	=	1	0
0	1	0	=	1	0
0	1	1	=	0	1
1	0	0	=	1	0
1	0	1	=	0	1
1	1	0	=	0	1
1	1	1	=	1	1

Табл. 14 Комбинации при събиране на битовете на двоични числа

Следващите примери илюстрират всичко описано дотук. За простота ще използвам 8-битови беззнакови числа, но принципите са абсолютно същите за 16- и 32-битови беззнакови числа.

Пример: 37 + 18 = 55

0 0 0 0 0 0 0 ← c_{out} (c_{in})
00100101 (37)
+

00010010 (18)

____

00110111 (55)

Пример: 55 + 100 = 155

1 1 0 0 1 0 0 ← c_{out} (c_{in})
00110111 (55)
+
01100100 (100)

____

10011011 (155)

Пример: 155 + 101 = 256

1 1 1 1 1 1 1 1 ← c_{out} (c_{in})
10011011 (155)
+
01100101 (101)

____

100000000 (0) Внимание!!!

Последният пример илюстрира ситуация, при която се генерира пренос от най-старшите битове, т.е. . сумата излиза извън обхвата от стойности, които могат да се представят в 8-битов беззнаков формат. Процесорите индицират това чрез специален бит C (**Carry**). Ако C = 0, няма препълване, в противен случай има препълване.

Изваждане

X - Y = Z

X – умаляемо
Y – умалител
Z – разлика

Изваждането на цели двоични числа е аналогично на изваждането на цели десетични числа, дори е по-просто, тъй като двоичната бройна система се състои само от две цифри – 0 и 1. Следващата таблица описва всички възможни комбинации при изваждане на двоични цифри.

комбинация		**разлика**	**заем b_{out}**
0 - 0	=	0	0
0 - 1	=	1	1
1 - 0	=	1	0
1 - 1	=	0	0

Табл. 15 Комбинации при изваждане на двоични цифри

Нула, извадена от коя да е цифра или число, дава самата цифра или число (вижте ред 1 и 3 на таблицата). Единица, извадена от коя да е цифра или число, дава предходната по големина цифра или число (вижте редове 2 и 4 на таблицата). Ще разгледам ред 2 малко по-подробно. При извършване на операцията 0 - 1 се взема заем 1 от следващия старши бит, т.е. . получава се 10_2 (2 десетично) - 1 = 1, т.е. . резулатният бит е 1 и се генерира заем $\mathbf{b_{out}}$ = 1, който се взема от следващия по-старши бит (битът, който се намира вляво от текущия бит).

При изваждане на битовете на две двоични числа се започва от най-младшите битове. Ако при това изваждане се генерира заем 1, той се изважда от разликата на следващите поред битове и т.н., докато се извърши изваждането и на най-старшите битове на числото. Описаният процес може да се обобщи като $\mathbf{a_i - b_i - b_{in}}$ и заем $\mathbf{b_{out}}$, където $\mathbf{a_i}$ и $\mathbf{b_i}$ са съответните битове на двоичното число, $\mathbf{b_{in}}$ е заемът от разликата на предходните битове, а $\mathbf{b_{out}}$ е заемът от това изваждане, който се явява $\mathbf{b_{in}}$ за следващите битове. Табл.16 показва всички възможни комбинации при изваждането на битовете на двоични числа и заема $\mathbf{b_{in}}$.

a_i	b_i	b_{in}		разлика	b_{out}
0	0	0	=	0	0
0	0	1	=	1	1
0	1	0	=	1	1
0	1	1	=	0	1
1	0	0	=	1	0
1	0	1	=	0	0
1	1	0	=	0	0
1	1	1	=	1	1

Табл. 16 Комбинации от изваждане на битовете на двоични числа

Следващите примери илюстрират всичко описано дотук. За простота ще използвам 8-битови беззнакови числа, но принципите са абсолютно същите за 16- и 32-битови беззнакови числа.

Пример: 37 - 18 = 19

0 0 1 0 1 0 0 ← b_{out} (b_{in})
00100101 (37)
-
00010010 (18)
____
00010011 (19)

Пример: 100 - 55 = 45

0 1 1 1 1 1 1 ← b_{out} (b_{in})
01100100 (100)
-
00110111 (55)
____
00101101 (45)

Пример: 4 - 10 = -6

1 1 1 1 1 0 1 0 $\leftarrow b_{out}$ (b_{in})
 00000100 (4)
-
 00001010 (10)
____
111111010 (-6) → резултатът е в допълнителен код

Последният пример илюстрира ситуация, при която се генерира заем при изваждане на най-старшите битове, т.е. . разликата е отрицателно число. Процесорите индицират това чрез специален бит B (**Borrow**). Ако B = 0, няма заем, в противен случай има заем.

Събиране и изваждане в допълнителен код

Съвременните компютърни системи използват допълнителен код за представяне на знакови числа. Допълнителният код има следните характеристики:

- Нулата има едно представяне (вижте отново Табл.9);
- Операцията изваждане се извършва чрез събиране на умаляемото и умалителя. Операцията изваждане може да се представи по следния начин:

  a - b = a + (-b) = (-b) + a

  -a - b = (-a) + (-b) = (-b) + (-a)

- Диапазонът от стойности, който може да се представи от n-битово знаково число е $\mathbf{-2^{n-1} \div +(2^{n-1}-1)}$.

  n = 8 → -128 ÷ +127
  n = 16 → -32768 ÷ +32767

n = 32 → -2 147 483 648 ÷ +2 147 483 647
n = 64 → -9 223 372 036 854 775 808 ÷
+9 223 372 036 854 775 807

Допълнителният код на положителните числа съвпада с обичайното двоично представяне на беззнаковите двоични числа. Например десетичното число +100 се представя по един и същ начин като беззнаково число и положително число в допълнителен код: 01100100. Отрицателните числа, както вече знаете, се представят в допълнителен код като положителният еквивалент на числото се преобразува в обратен код и към резултата се прибави 1.

Процесът на събиране (изваждане) в допълнителен код може да бъде обобщен по следния начин:

1. Двете числа, които ще се сумират (изваждат), се представят в допълнителен код.
2. Числата се сумират, използвайки базовите правила за двоично събиране, описани в Табл.13 и 14.
3. Преносът от най-старшите битове, ако има такъв, се изхвърля.
4. Полученият резултат е в допълнителен код.

Следващите примери илюстрират всичко описано дотук. За простота ще използвам 8-битови знакови числа, но принципите са абсолютно същите за 16- и 32-битови знакови числа.

Пример: (+37) + (+18) = 55

- Допълнителният код на +37 е 00100101.
- Допълнителният код на +18 е 00010010.

```
  00100101 (+37)
+
  00010010 (+18)
```

____

00110111 (+55)

Пример: (+37) + (-18) = +19

- Допълнителният код на +37 е 00100101.
- Допълнителният код на -18 е 11101110.

00100101 (+37)

\+

11101110 (-18)

____

~~1~~00010011 (+19)

Пример: (+18) + (-37) = -19

- Допълнителният код на +18 е 00010010.
- Допълнителният код на -37 е 11011011.

00010010 (+18)

\+

11011011 (-37)

____

11101101 (-19)

Пример: (-18) + (-37) = -55

- Допълнителният код на -18 е 11101110.
- Допълнителният код на -37 е 11011011.

11101110 (-18)

\+

11011011 (-37)

____

~~1~~11001001 (-55)

Пример: (-128) + (-1) = -129

- Допълнителният код на -128 е 10000000.
- Допълнителният код на -1 е 11111111.

10000000 (-128)
+
11111111 (-1)

~~1~~01111111 (+127) Внимание!!!

Последният пример илюстрира ситуация, при която се получава аритметично препълване, т.е. . резултатът излиза извън диапазона от представимите стойности. Резултатът от сумата на -128 и -1 е -129, но поради препълване резултатът, който се получава, е +127. Процесорите индицират това чрез специален бит V (**Overflow**). Ако V = 0, няма аритметично препълване, в противен случай има аритметично препълване.

Умножение

X * Y = Z

X – множимо
Y – множител
Z – произведение

Умножението на цели двоични числа е аналогично на умножението на цели десетични числа, дори е по-просто, тъй като двоичната бройна система се състои само от две цифри – 0 и 1. Следващата таблица описва всички възможни комбинации при умножение на двоични цифри.

комбинация		произведение
0 x 0	=	0
0 x 1	=	0
1 x 0	=	0
1 x 1	=	1

Табл. 17 Комбинации при умножение на двоични цифри

Нула, умножена с коя да е цифра или число, дава нула (вижте първите два реда на таблицата). Единица, умножена с коя да е цифра или число, дава самата цифра или число (вижте последните два реда на таблицата).

Класическият метод на умножение на беззнакови двоични числа е същият като при умножение на беззнакови десетични числа и се нарича алгоритъм с „многократно преместване наляво и сумиране” (**repeated left-shift and add**). При този метод умножението започва с най-младшия бит на множителя. Всяко частично произведение (без първото) се поставя под предишното и се премества с една позиция наляво. Този процес се повтаря, докато се изчерпят всички битове на множителя. Накрая така получените частични произведения се сумират и се получава крайното произведение.

Запомнете че при беззнаково умножаване на n-битово с m-битово число се получава (n+m)-битово беззнаково число, т.е. . ако умножите две 8-битови беззнакови числа резултатът ще бъде 16-битов.

Следващите примери илюстрират всичко описано дотук. За простота ще използвам 8-битови беззнакови числа, но принципите са абсолютно същите за 16- и 32-битови беззнакови числа.

Пример: 2 x 3 = 6

00000010 (2) множимо
x
00000011 (3) множител

00000010 x 1 = 0000000000000010
00000010 x 1 = 0000000000000100
00000010 x 0 = 0000000000000000

Частични произведения

00000010 x 0 = 0000000000000000
00000010 x 0 = 0000000000000000
00000010 x 0 = 0000000000000000
00000010 x 0 = 0000000000000000
00000010 x 0 = 0000000000000000

Частични произведения

0000000000000110 (6)

Частичните произведения се разширяват до 16-битови числа. Тъй като в случая те са положителни (множимото, от което те се получават, е положително), разширяването става чрез добавяне на съответния брой нули вляво на частичните произведения (нулите в червено). Вдясно на частичните произведения винаги се добавят нули (нулите в синьо).

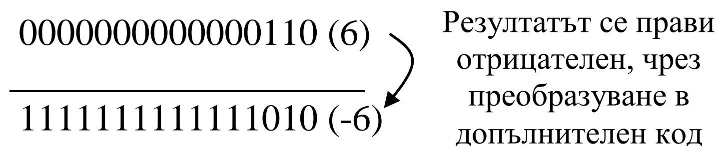
Най-простият начин за умножение на знакови цели числа е те да се преобразуват в положителни и да се извърши умножението по описания вече начин. Ако множимото и множителят имат различен знак, резултатът се отрицава, чрез преобразуване в допълнителен код.

Пример: (-2) x 3 = -6

00000010 (2) множимо
x
00000011 (3) множител

00000010 x 1 = 0000000000000010
00000010 x 1 = 0000000000000100
00000010 x 0 = 0000000000000000
00000010 x 0 = 0000000000000000
00000010 x 0 = 0000000000000000
00000010 x 0 = 0000000000000000
00000010 x 0 = 0000000000000000
00000010 x 0 = 0000000000000000

Частични произведения



Запомнете че при знаково умножаване на n-битово с m-битово число се получава (n+m)-битово знаково число, т.е. . ако умножавате две 8-битови знакови числа, резултатът ще бъде 16-битов (1 бит за знак и 15 бита за стойността).

Знаковите числа могат да се умножат по същия метод както и беззнаковите, но с лека модификация. Ако множителят е отрицателен, последното частично произведение се изважда, а не се добавя. За целта то се преобразува в допълнителен код и се добавя към останалите частични произведения.

Пример: (-2) x 3 = -6

```
                 11111110 (-2) множимо
                x
                 00000011 (3) множител
          ________________________

11111110 x 1 = 1111111111111110 \
11111110 x 1 = 1111111111111100 |
11111110 x 0 = 0000000000000000 |
11111110 x 0 = 0000000000000000 |  Частични
11111110 x 0 = 0000000000000000 >  произведения
11111110 x 0 = 0000000000000000 |
11111110 x 0 = 0000000000000000 |
11111110 x 0 = 0000000000000000 /
          ________________________
           1111111111111010 (-6)
```

Обърнете внимание на първите две частични произведения. Тъй като в случая те са отрицателни (множимото, от което те се получават, е отрицателно), разширяването става чрез добавяне на съответния брой единици вляво на частичните

произведения (единиците в червено).

Пример: 2 x (-3) = -6

```
                 00000010 (2) множимо
               x
                 11111101 (-3) множител
            ____________________
00000010 x 1 = 0000000000000010  \
00000010 x 0 = 0000000000000000  |
00000010 x 1 = 0000000000001000  |
00000010 x 1 = 0000000000010000  |  Частични
00000010 x 1 = 0000000000100000  >  произведения
00000010 x 1 = 0000000001000000  |
00000010 x 1 = 0000000010000000  |
00000010 x 1 = 0000000100000000  /  (зачеркнато)
            (1111111100000000)   }  Допълнителен код на
                                    последното частично
            ____________________       произведение
             1111111111111010 (-6)
```

Забележете че последното частично произведение (зачертаното с линия през средата) се заменя с допълителние си код .

Пример: (-2) x (-3) = 6

```
                 11111110 (-2) множимо
               x
                 11111101 (-3) множител
            ____________________

11111110 x 1 = 1111111111111110  \
11111110 x 0 = 0000000000000000  |
11111110 x 1 = 1111111111111000  |  Частични
11111110 x 1 = 1111111111110000  >  произведения
11111110 x 1 = 1111111111100000  |
11111110 x 1 = 1111111111000000  /
```

11111110 x 1 = 111111111000000
11111110 x 1 = ~~111111100000000~~
(000000100000000)

000000000000110 (6)

Частични произведения

Допълнителен код на последното частично произведение

Забележете че последното частично произведение (зачертаното с линия през средата) се заменя с допълнителния си код.

Освен описаните алгоритми за умножение съществуват и други алгоритми, които са по-удобни за реализация в компютърните системи, като алгоритъм с „многократно сумиране и преместване надясно" (**repeated add and right-shift**). При този алгоритъм първо се формира събираемо, състоящо се само от толкова на брой нули, колкото бита има в множимото. Другото събираемо се получава като най-младшия бит на множителя се умножи с множимото. Резултатът от първото събиране се премества една позиция надясно и изместеният бит се запомня. Освободената позиция на най-старшия бит се запълва с нула. Така формираната последователност се явява първото събираемо на следващото събиране. Второто събираемо се получава от умножението на следващия бит на множителя с множимото. Резултатната сума се премества надясно, като изместения бит отново се запомня. Процесът се повтаря до изчерпване на всички битове на множителя. Последната сума (след преместване надясно и запомняне на преместения бит) заедно с последователността от битове, получена при преместването на междинните суми надясно с една позиция, формират крайното произведение.

Следващият пример онагледява описания процес. За простота ще използвам 4-битови беззнакови числа.

Пример: 2 x 3 = 6

0010 (2) множимо
x
0011 (3) множител

Начална последователност → 0000
1 x 0010 = 0010

Първа междинна сума 0 → 0010 → 0

0001
1 x 0010 = 0010

Втора междинна сума 0 → 0011 → 10

0001
0 x 0010 = 0000

Трета междинна сума 0 → 0001 → 110

0000
0 x 0010 = 0000

Последна междинна сума 0 → 0000 → 0110

Произведение 00000110 (6)

Деление

X / Y = Q + R/Y

X – делимо
Y – делител
Q – частно
R – остатък

Ако R = 0, делението е точно, в противен случай делимото не се дели точно на делителя.

Делението е най-трудната от всички предходни операции. Класическият метод на деление на беззнакови двоични числа е същият като при деление на беззнакови десетични числа и се нарича алгоритъм с „многократно преместване надясно и изваждане” (**repeated right-shift and subract**). За по-лесно ще използвам следващия пример, за да опиша как работи този алгоритъм.

Пример: 38 / 12 = 3 и остатък 2

- 8-битовият код на 38 е 00100110.
- 8-битовият код на 12 е 00001100.

1. Водещите нули на делимото и делителя се игнорират.

   ~~00~~100110 / ~~0000~~1100

2. Делителят се поставя под делимото и се сравнява със същия брой битове на делимото (битовете в сиво). Ако делителят е по-голям (както е в случая), в частното се записва 0 и делителят се премества с една позиция надясно и отново се извършва сравнение.

   100110 / 1100= 0   100110 / = 0
   1100   1100

3. Ако делителят е по-малък от делимото (битовете в сиво), в частното се записва 1 и делимото се изважда от делителя.

   100110 / 1100 = 01
   \-   1100
   ______
      111 → Частичен остатък

4. Поредният бит на делимото се сваля до частичния остатък.

```
  100110  / 1100 = 01
-  1100
  ______
    1110
```

5. Делителят се поставя под така формираното число и се сравнява. Ако делителят е по-малък (както е в случая), в частното се записва 1 и делителят се изважда, в противен случай в частното се записва 0, поредният бит на делимото се сваля и делителят се премества една позиция надясно.

```
  100110  / 1100 = 011
-  1100
  ______
    1110
-   1100
  ______
      10
```

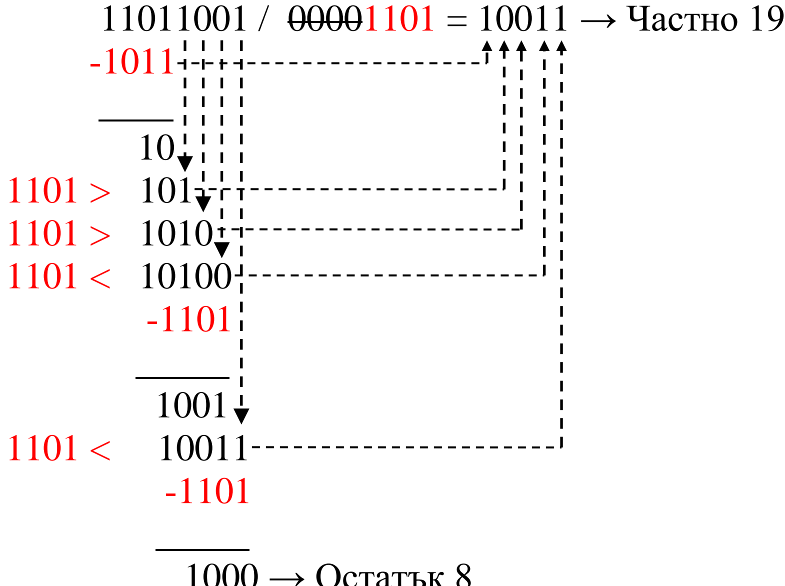
6. Точка 4 и 5 се повтарят, докато се изчерпят всички битове на делимото.

```
  100110  / 1100 = 011 → Частно 3
-  1100
  ______
    1110
-   1100
  ______
      10 → Остатък 2
```

Нека да разгледаме още един пример.

Пример: 217 / 11 = 19 и остатък 8

- 8-битовият код на 217 е 11011001.
- 8-битовият код на 11 е 00001011.



Най-простия начин за деление на знакови числа е те да се преобразуват в положителни, да се извърши беззнаково деление и да се добави знака на частното и остатъка съгласно следните условия:

Знак-частно = Знак-делимо XOR Знак-делител
Знак-остатък = Знак-делимо

Ако знакът на частното и остатъкът е отрицателен (1), те се отрицават, като се преобразуват в допълнителен код.

Пример: -6 / 4 = -1 и остатък -2

- 8-битовият код на -6 е 11111010.
- 8-битовият положителния код на -6 е 00000110 (+6).
- 8-битовият код на 4 е 00000100.

~~00000~~110 / ~~00000~~100 = 1 → Частно 1

- 100

$\overline{}$

10 → Остатък 2

Знак-частно = 1 XOR 0 = 1
Знак-остатък = 1

Частно = $-1_{10}(11111111_2$ в допълнителен код)
Остатък = $-2_{10}(11111110_2$ в допълнителен код)

Освен описаните алгоритми за деление съществуват и други, които са по-удобни за реализация в компютърните системи.

6.2 Цифрова аритметика с числа с фиксирана запетая

Спомнете си, че в 5.2 Дробни числа с фиксирана запетая Ви обясних, че целите двочни числа също са числа с фиксирана запетя, като запетаята е разположена най-вдясно на всички битове на числото. Това означава, че правилата, описани в предходната точка, са в сила и тук. Следващите примери използват формата Q8.3.

Събиране (изваждане)

Пример: 15.375 + 20.75 = 36.125

- Двоичният формат Q8.3 на 15.375 е 00001111.011
- Двоичният формат Q8.3 на 20.75 е 00010100.110

0 0 1 1 1 1 1 1 1 0 ← c_{out} (c_{in})
 00001111.011 (15.375)
\+
 00010100.110 (20.75)

 00100100.001 (36.125)

Пример: +15.375 + (-20.75) = -5.375

- Двоичният формат Q8.3 на 15.375 е 00001111.011
- Двоичният формат Q8.3 на -20.75 е 11101011.010 (в допълнителен код)

0 0 0 1 1 1 1 0 1 0 ← c_{out} (c_{in})

00001111.011 (+15.375)

\+

11101011.010 (-20.75)

____

11111010.101 (-5.375)

Пример: (-15.375) + (-20.75) = -36.125

- Двоичният формат Q8.3 на 15.375 е 00001111.011
- Двоичният формат Q8.3 на -20.75 е 11101011.010 (в допълнителен код)

0 0 0 0 0 0 0 0 0 0 ← c_{out} (c_{in})

11110000.101 (-15.375)

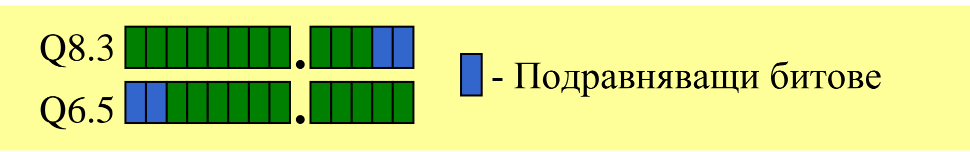
\+

11101011.010 (-20.75)

____

~~1~~11011011.111 (-36.125)

При събиране (изваждане) на числа с фиксирана запетая, представени в различен формати, е необходимо двоичната запетая да бъде подравнена преди това. Например ако имаме две числа във форматите Q8.3 и Q6.5, те трябва да се подравнят по следния начин:



Резултатното число ще бъде във формат Q8.5.

Умножение

$Qi_1.f_1$ x $Qi_1.f_1 = Q(i_1+i_2).(f_1+f_2)$

Пример: 6.125 x 3.75 = 22.96875

- Двоичният формат Q8.3 на 6.125 е 00000110.001
- Двоичният формат Q8.3 на 3.75 е 00000011.110

```
00000110.001 x 00000011.110
___________________________
              00000000000
             00000110001
            00000110001
           00000110001
          00000110001
         ...................
      00000000000
___________________________
   0000000000010110.111110
```

Деление

Делението на числа с фиксирана запетая, чиято дробна част е различна от нула, е достатъчно сложна задача, поради тази причина то няма да бъде разгледано тук.

6.3 Цифрова аритметика с числа с плаваща запетая

Ако Fp_1 и Fp_2 са две числа с плаваща запетая, представени като

$Fp_1 = M_1 \times 2^{E_1}$

$Fp_2 = M_2 \times 2^{E_2}$,

то са в сила следните аритметични операции: +, –, *, /.

Преди да се извърши операцията + или – е необходимо експонентите на двете числа да бъдат изравнени, т.е. . $E_1 = E_2 = E$.

$Fp_1 + Fp_2 = M_1 \times 2^E + M_2 \times 2^E = (M_1 + M_2) \times 2^E$

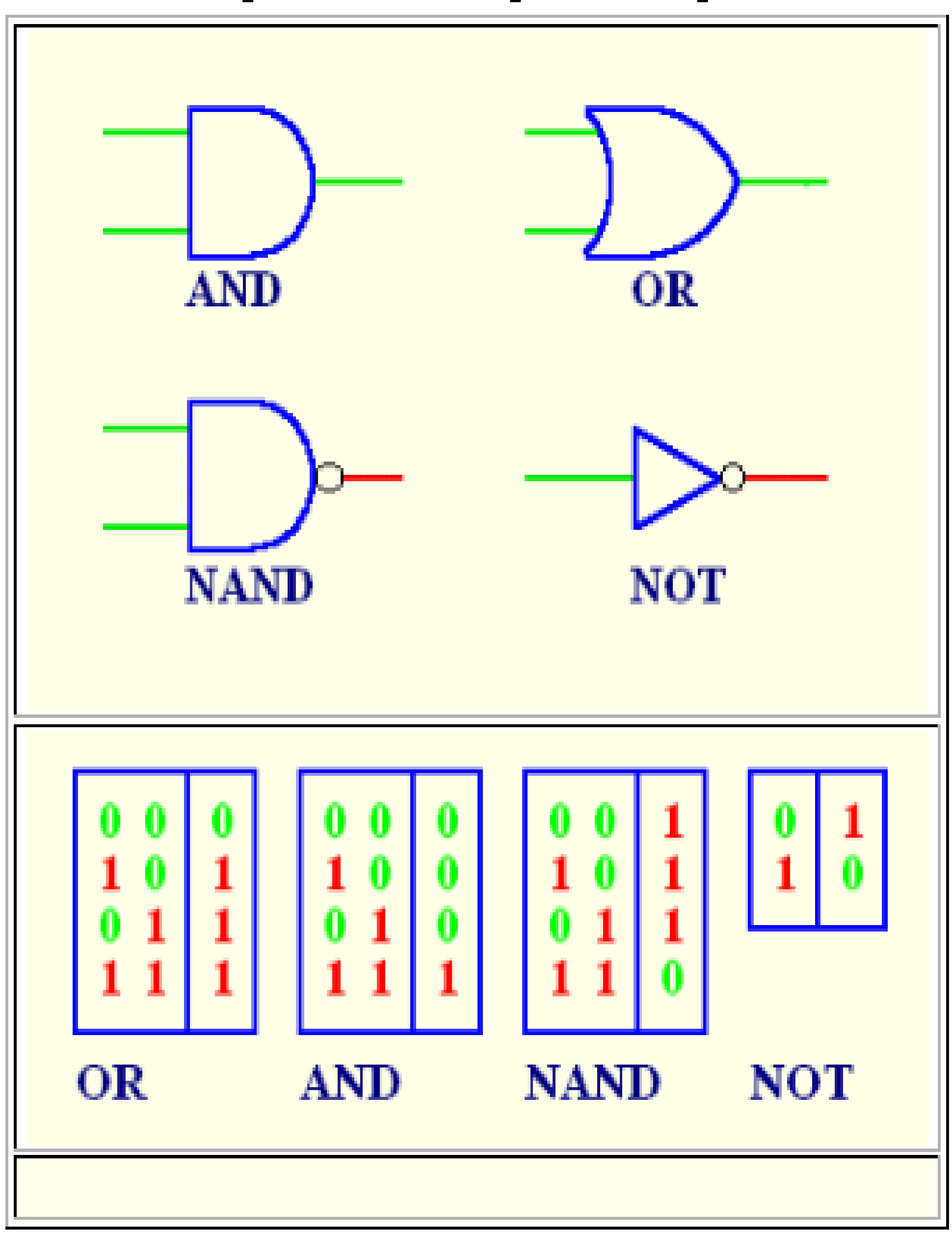
$Fp_1 – Fp_2 = M_1 \times 2^E – M_2 \times 2^E = (M_1 – M_2) \times 2^E$

$Fp_1 \times Fp_2 = (M_1 \times 2^{E_1}) \times (M_2 \times 2^{E_2}) = (M_1 \times M_2) \times 2^{(E_1 + E_2)}$

$Fp_1 / Fp_2 = (M_1 \times 2^{E_1}) / (M_2 \times 2^{E_2}) = (M_1 / M_2) \times 2^{(E_1 – E_2)}$

Част II

Градивни елементи на микроконтролерите

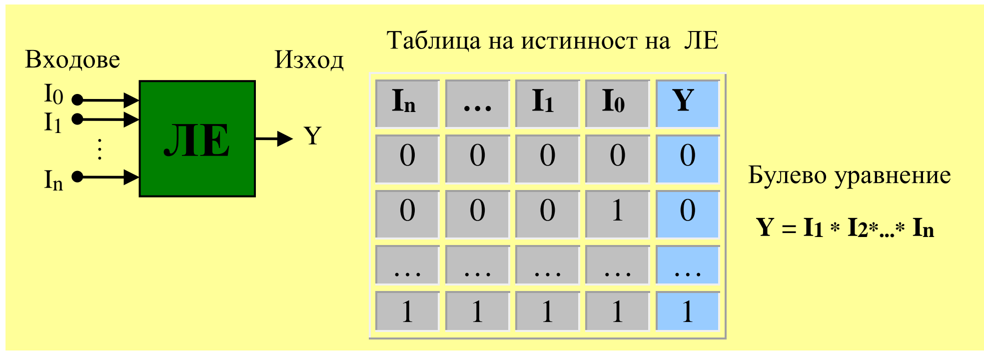


7 Логически елементи

7.1 Общи сведения

Логическите елементи (ЛЕ) са най-простите електронни цифрови схеми, с чиято помощ се реализират цифрови схеми с различна степен на сложност. Има три базови логически елементи: **NOT (Логическо НЕ)**, **OR (Логическо ИЛИ)** и **AND (Логическо И)**. Останалите логически елементи, като **NOR (Логическо ИЛИ-НЕ)**, **NAND (Логическо И-НЕ)**, **XOR (Логическо Изключващо ИЛИ)**, **XNOR (Логическо Изключващо ИЛИ-НЕ)** и т.н., се получават от базовите елементи.

Логическите елементи са базовите градивни елементи на всяка цифрова система, включително и на микроконтролерите. Един логически елемент се състои от един или повече входове и един изход. Връзката между резултата, получен на изхода, и стойностите на входовете на логическия елемент може да бъде описана с помощта на **таблица на истинност** или **логическо (булево) уравнение**. Таблицата на истинност изброява вляво всички възможни комбинации от стойности на входовете и вдясно резултата на изхода срещу всяка входна комбинация. Булевото уравнение е математически израз, използващ двоични стойности (Фиг.31).



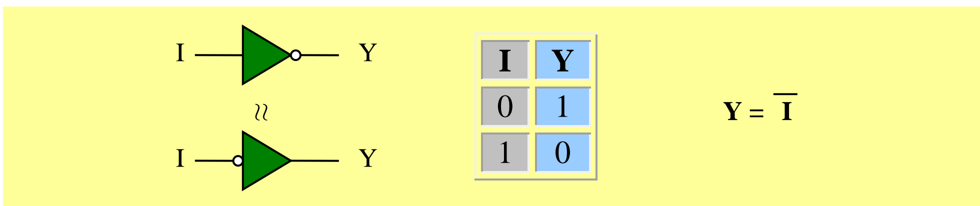
I_n	…	I_1	I_0	Y
0	0	0	0	0
0	0	0	1	0
…	…	…	…	…
1	1	1	1	1

Фиг. 31 Логически елемент

Логическите елементи и изградените с тях цифрови схеми могат да се разглежда като „черни кутии" с определен брой входове и изходи. Освен входове и изходи те имат и изводи за захранване и маса, които обикновено не се отбелязват, а се подразбира, че ги има. В следващите няколко подточки ще Ви запозная с базовите логически елементи. Ще разгледаме как с тяхна помощ се създават по-сложни цифрови схеми като тригери, с чиято помощ се създават още по-сложни цифрови схеми като регистри, броячи, делители на честота и други. По принцип не е необходимо да познавате в детайли вътрешната структура на една цифрова схема, за да програмирате микроконтролери. Например не е необходимо да знаете как точно е изграден един брояч в микроконтролера, за да го използвате. Достатъчно е да знаете какво е брояч и какво прави, а не как го прави. Въпреки това тези знания няма да Ви бъдат излишни. Много често в документацията на микроконтролерите се използват опростени блокови схеми, използващи логическите елементи, разгледани в тази част, които описват как работи дадена част на микроконтролера. Освен това ще придобиете реална представа как работи микроконтролера на най-ниско ниво.

7.2 Логическо НЕ (NOT)

Логическият елемент NOT има един вход и един изход. Фиг.32 показва символа за обозначаване на този логически елемент, таблицата му на истинност и булевото уравнение.



I	Y
0	1
1	0

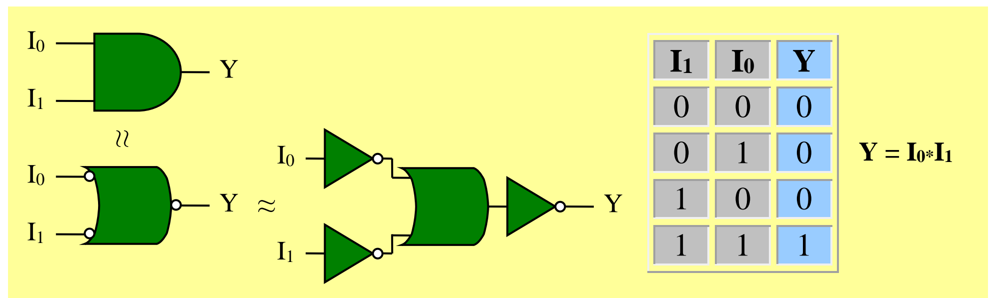
Фиг. 32 Логическо НЕ (NOT)

Когато стойността на входа I е лог.0, стойността на изхода Y е лог.1 и обратно, когато стойността на входа I е лог.1, стойността на изхода Y е лог.0, т.е. . стойността на изхода Y винаги е инвертирана спрямо стойността на входа I. Кръгчето в изхода (входа) на логическия елемент и чертата над I в булевото уравнение указват именно това. По тази причина логическият елемент NOT се нарича още инвертор. Булевото уравнение се чете като „Y е равно на **НЕ** I”.

Обръщам Ви отново внимание, че изводите за захранване и маса се подразбират.

7.3 Логическо И (AND)

Логическият елемент AND има два или повече входа и един изход. Фиг.33 показва символа за обозначаване на този логически елемент с два входа, таблицата му на истинност и булевото уравнение.



I_1	I_0	Y
0	0	0
0	1	0
1	0	0
1	1	1

Фиг. 33 Логическо И (AND)

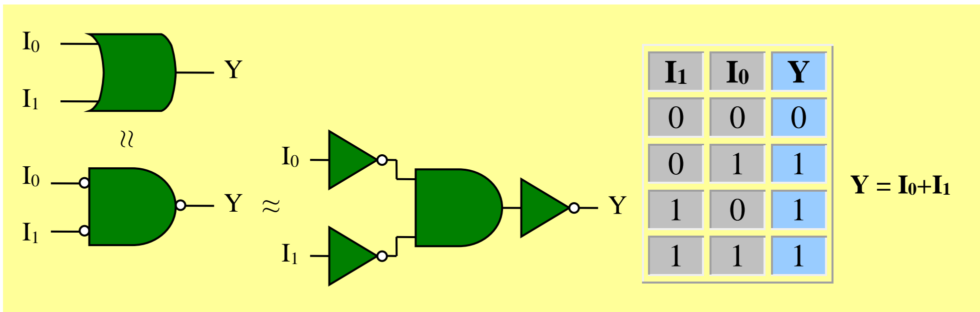
Както се вижда от таблицата на истинност, изходът е 1, само когато **И** двата входа са 1. Булевото уравнение се чете като „Y е равно на I_0 **И** I_1”, а не „Y е равно на I_0 **по** I_1”.

На фигурата е показано също и алтернативно обозначение на логическия елемент И, което се получава с помощта на

инвертори и логическия елемент ИЛИ (вижте следващата подточка).

7.4 Логическо ИЛИ (OR)

Логическият елемент OR има два или повече входа и един изход. Фиг.34 показва символа за обозначаване на този логически елемент с два входа, таблицата му на истинност и булевото уравнение.



I_1	I_0	Y
0	0	0
0	1	1
1	0	1
1	1	1

Фиг. 34 Логическо ИЛИ (OR)

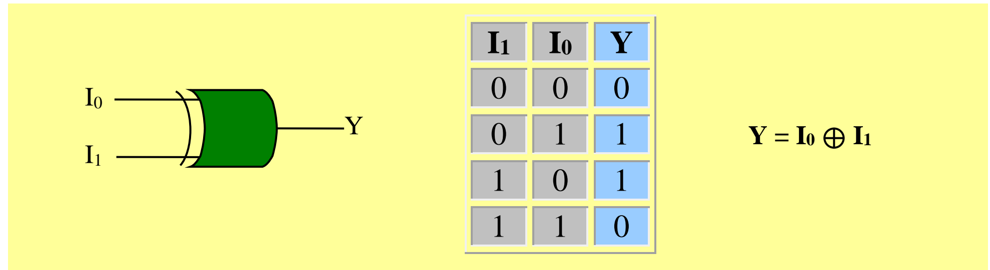
Както се вижда от таблицата на истинност, изходът е 1, когато поне единият от входовете е 1. Булевото уравнение се чете като „Y е равно на I_0 **ИЛИ** I_1”, а не „Y е равно на I_0 **плюс** I_1”.

На фигурата е показано също и алтернативно обозначение на логическия елемент ИЛИ, което се получава с помощта на инвертори и логическия елемент И (вижте предходната подточка).

7.5 Логическо Изключващо ИЛИ (XOR)

Логическият елемент XOR има два или повече входа и един

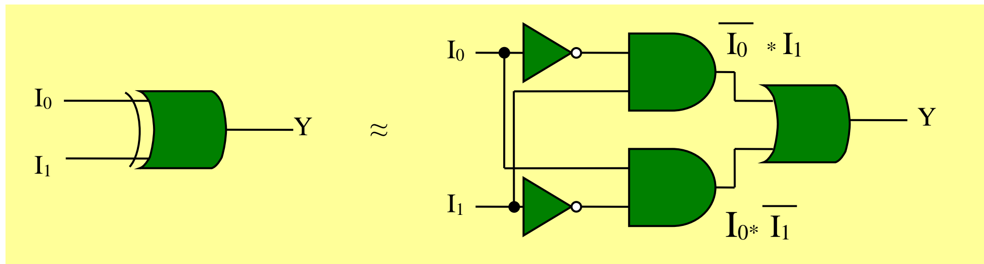
изход. Фиг.35 показва символа за обозначаване на този логически елемент с два входа, таблицата му на истинност и булевото уравнение.



I_1	I_0	Y
0	0	0
0	1	1
1	0	1
1	1	0

Фиг. 35 Логическо Изключващо ИЛИ (XOR)

Логическата операцията, която се извършва от XOR, е еквивалентна на: $Y = I_0 \oplus I_1 = \overline{I_0} * I_1 + I_0 * \overline{I_1}$. Тоест XOR може да се конструира от трите базови логически елементи както е показано на следващата фигура.

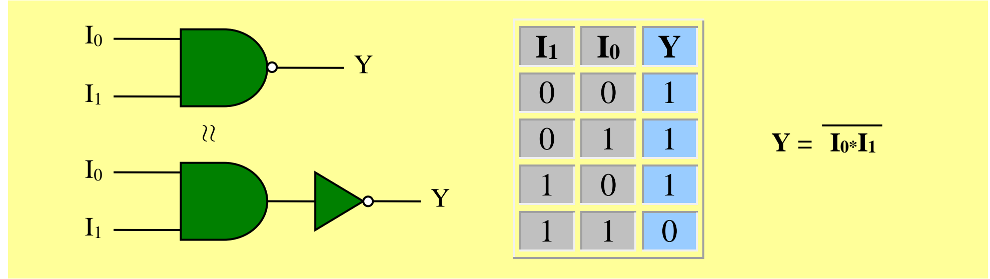


Фиг. 36 Еквивалентна схема на XOR

7.6 Логическо И-НЕ (NAND)

Логическият елемент NAND има два или повече входа и един изход. Фиг.37 показва символа за обозначаване на този логически елемент с два входа, таблицата му на истинност и булевото уравнение.

Както се вижда от фигурата, NAND се получава като в изхо-

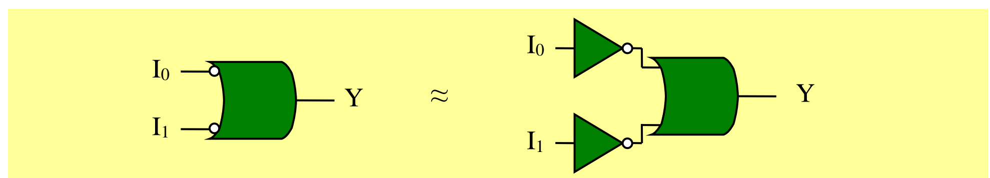


I_1	I_0	Y
0	0	1
0	1	1
1	0	1
1	1	0

Фиг. 37 Логическо И-НЕ (NAND)

да на AND се свърже инвертор.

Следващата фигура показва алтернативно обозначение на логическия елемент И-НЕ, което се получава с помощта на инвертори и логическия елемент ИЛИ.



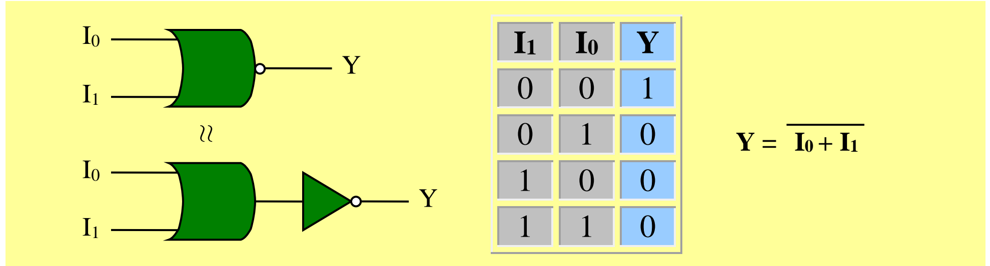
Фиг. 38 Логическо И-НЕ (NAND) – алтернативен символ

7.7 Логическо ИЛИ-НЕ (NOR)

Логическият елемент NOR има два или повече входа и един изход. Фиг.39 показва символа за обозначаване на този логически елемент с два входа, таблицата му на истинност и булевото уравнение.

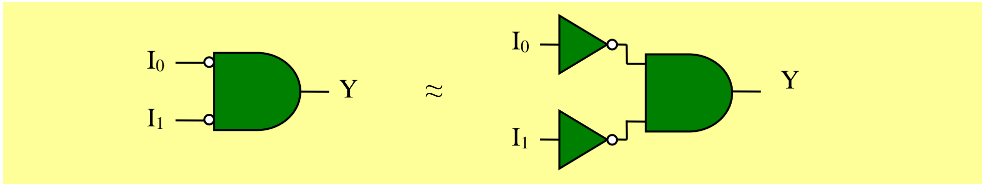
Както се вижда от фигурата, NOR се получава като в изхода на OR се свърже инвертор.

Фиг.40 показва алтернативно обозначение на логическия елемент ИЛИ-НЕ, което се получава с помощта на инвертори и логическия елемент И.



I_1	I_0	Y
0	0	1
0	1	0
1	0	0
1	1	0

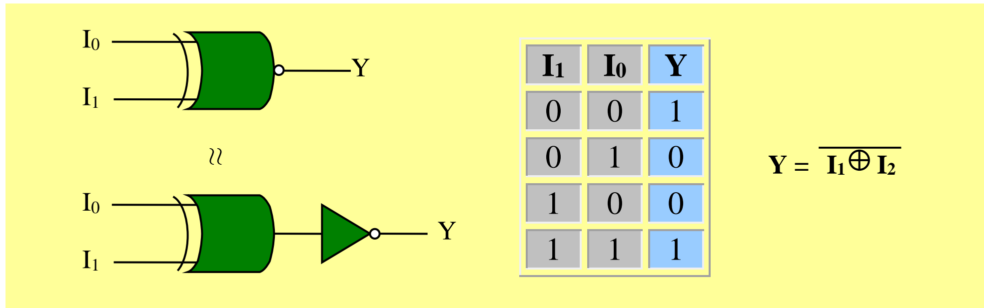
Фиг. 39 Логическо ИЛИ-НЕ (NOR)



Фиг. 40 Логическо ИЛИ-НЕ (NOP) – алтернативен символ

7.8 Логическо Изключващо ИЛИ-НЕ (XNOR)

Логическият елемент XNOR има два или повече входа и един изход. Фиг.41 показва символа за обозначаване на този логически елемент с два входа, таблицата му на истинност и булевото уравнение.



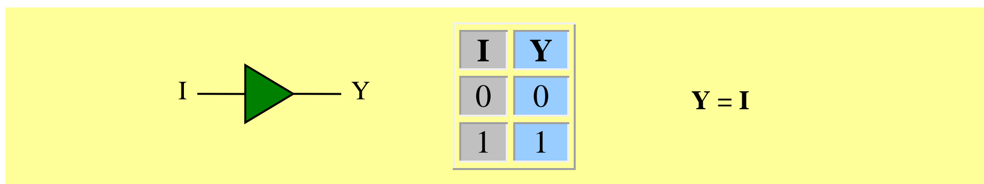
I_1	I_0	Y
0	0	1
0	1	0
1	0	0
1	1	1

Фиг. 41 Логическо Изключващо ИЛИ (XOR)

Както се вижда от фигурата, XNOR се получава като в изхода на XOR се свърже инвертор.

7.9 Логически повторител (буфер)

Логическият повторител има един вход и един изход. Фиг.42 показва символа за обозначаване на този логически елемент, таблицата му на истинност и булевото уравнение.



I	Y
0	0
1	1

Фиг. 42 Логически повторител

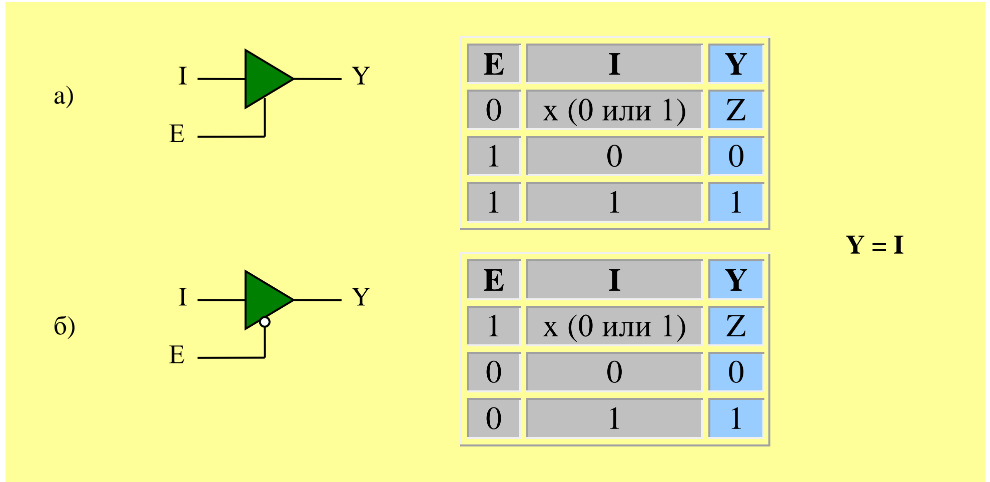
Както се вижда от таблицата на истинност, изходът просто повтаря стойността на входа.

7.10 Логически елементи с три състояния

Всеки логически елемент може да имат допълнителен разрешаващ вход E (Enable), който активира или деактивира изхода. Когато изходът е активиран, той може да има стойност 0 или 1 в зависимост от стойностите на входа. Когато изходът е деактивиран, той се намира в т.нар. **високо-импедансно състояние**, което се обозначава като Hi-Z или само Z. В това трето състояние нивото на изхода не е нито лог.0, нито лог.1. Изходът може да се разглежда като „плаваща" жица, която не е свързана с нищо. Следващата фигура показва буфер с три състояния с активен висок (Фиг.43а) и активен нисък (Фиг.43б) разрешаваш сигнал

Когато изходът е деактивиран, той се намира в състояние Z, независимо от стойността на входа.

Фиг.44 показва абстрактно представяне на буфер с три състояния и активен висок разрешаващ сигнал.



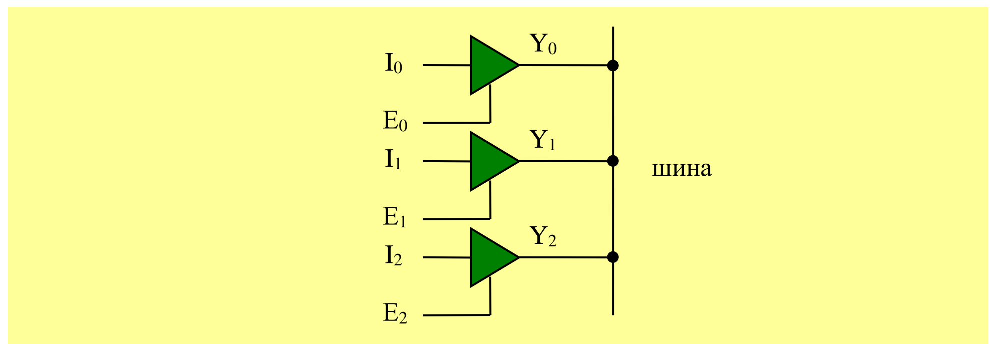
E	I	Y
0	x (0 или 1)	Z
1	0	0
1	1	1

E	I	Y
1	x (0 или 1)	Z
0	0	0
0	1	1

Фиг. 43 Буфер с три състояния

Фиг. 44 Буфер с три състояния – ключова схема

Логическите елементи с три състояния позволяват свързването на входовете или изходите на няколко такива елементи към обща шина (Фиг.45).



Фиг. 45 Свързване на буфер с три състояния към обща шина

Само входната стойност на елемента, чийто изход е активиран, ще се появи на шината.

8 Тригери

8.1 Общи сведения

Цифровите схеми се делят на два основни типа:

- Комбинационни цифрови схеми;

  Изходите на комбинационните цифровите схеми зависят от текущите стойности на входовете. Логическите елементи, разгледани в предната глава, са най-простите комбинационни цифрови схеми.

- Схеми с последователна логика.

  Изходите на схемите с последователна логика зависят както от текущите стойности на входовете, така и от предходните. Схемите с последователна логика се наричат още схеми с памет.

Тригерите принадлежат към втората група цифрови схеми. Те имат две устойчиви състояния: ВИСОКО (HIGH) и НИСКО (LOW). В зависимост дали използват тактов сигнал тригерите се делят на две групи:

- Асинхронни тригери (**latches**);

- Синхронни тригери (**flip-flops**).

В зависимост от принципа на работа тригерите се делят на:

- RS тригери;

- JK тригери;

- D тригери;

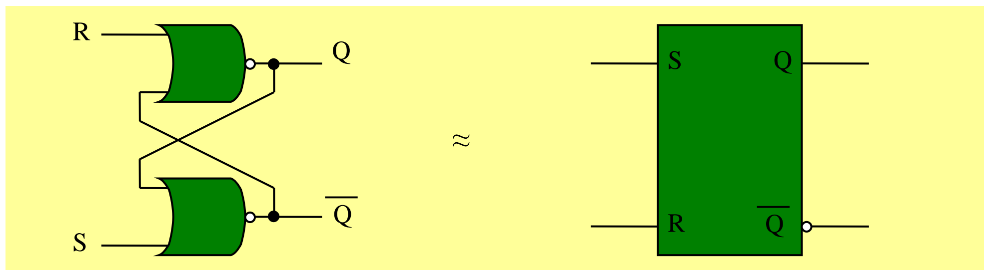
- Т тригери.

Тригерите могат да се използват самостоятелно или в комбина за изграждането на по-сложни цифрови схеми като регистри, броячи и делители на честота (регистрите и броячите са разгледани в следващите глави). Следващите подточки описват накратко принципа на работа на различните видове тригерите.

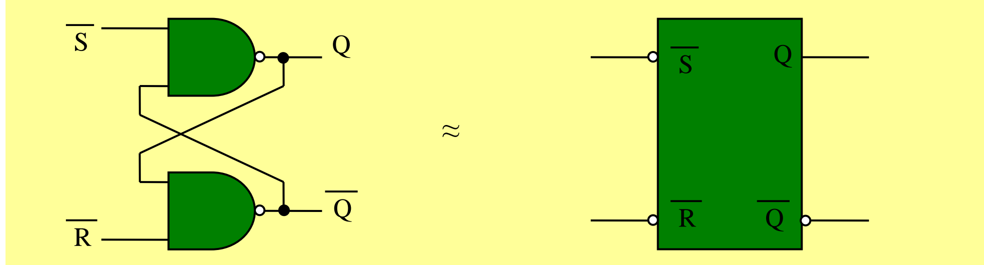
8.2 RS тригери

8.2.1 Асинхронни RS тригери

Фиг.46 и 47 показват вътрешното устройство на RS тригер и символите, с които се обозначават. Активните входни нива могат да бъдат както високи (лог.1) (Фиг.46), така и ниски (лог.0)(Фиг.47).



Фиг. 46 RS тригер с активни високи входове



Фиг. 47 RS тригер с активни ниски входове

Когато Q изходът е 1, се казва че тригерът е установен. Когато Q изходът е 0, се казва че тригерът е нулиран. Входът S (**SET**) се използва за установяване на тригера, а входа R (**RESET**) – за нулиране. Следващите таблици на истинност описват работата на тригер с високи активни и ниски активни входове съответно.

S	**R**	**Q**	$\overline{\mathbf{Q}}$	**Коментар**
0	0	Q	$\overline{Q}$	Съхраняване на предишното състояние на изходите
0	1	0	1	Нулиране на тригера
1	0	1	0	Установяване на тригера
1	1	0	0	Забранена комбинация

Табл. 18 Таблица на истинност на RS тригер с активни високи входове

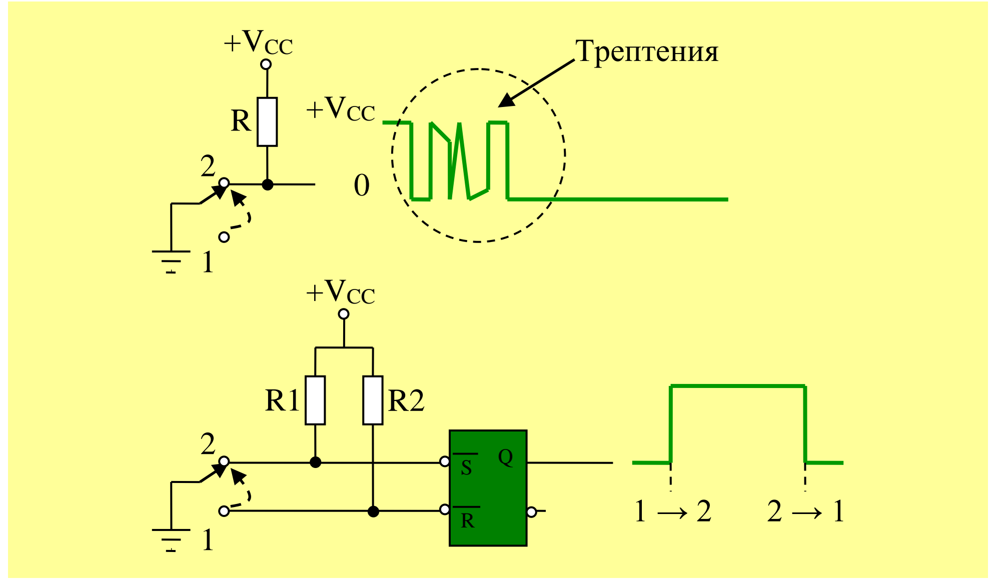
$\overline{\mathbf{S}}$	$\overline{\mathbf{R}}$	**Q**	$\overline{\mathbf{Q}}$	**Коментар**
0	0	1	1	Забранена комбинация
0	1	1	0	Установяване на тригера
1	0	0	1	Нулиране на тригера
1	1	Q	$\overline{Q}$	Съхраняване на предишното състояние на изходите

Табл. 19 Таблица на истинност на RS тригер с активни ниски входове

Обърнете внимание, че когато двата входа са активни едновременно това се приема за забранена комбинация и не трябва да се използва. Например ако R = 1 и S = 1 при тригер с активни високи входове, двата изхода са 0. Причината тази комбинация да е забранена е не толкова, че правият и инвертираният изход имат еднакви стойности, а че ако след това входовете сменят стойностите си едновременно в ниско ниво, нивото на изходите е непредсказуемо.

Следващата фигура показва едно типично приложение на RS тригер в ролята на „гасяща" схема на трептенията, получени

при превключването на ключ или при натискане/отпускане на бутон (Фиг.48).

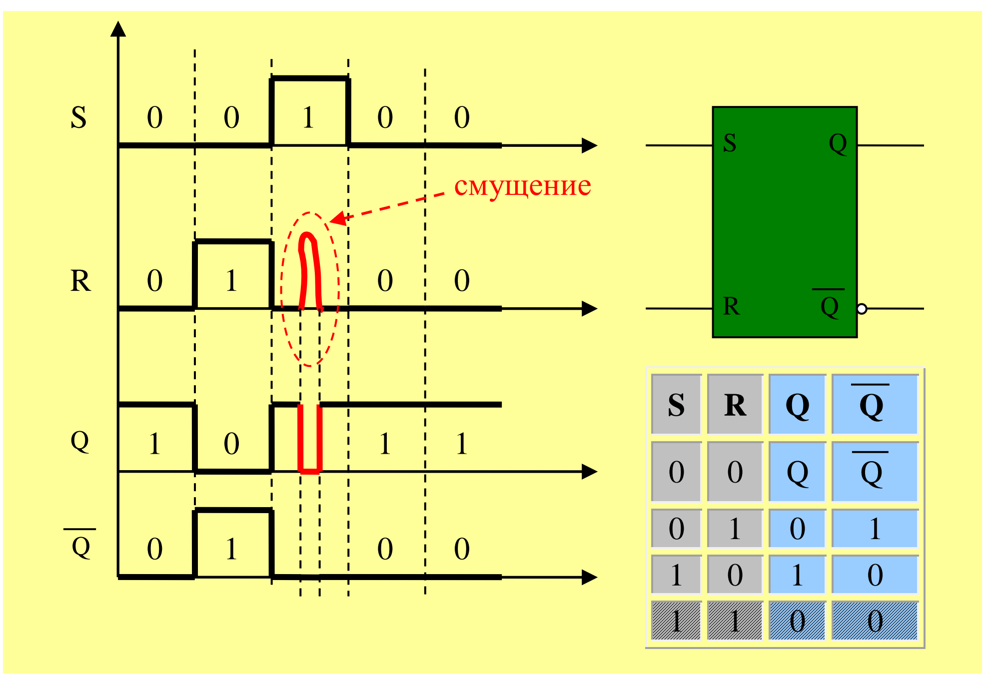


Фиг. 48 "Гасене" на трептения с RS тригер

Приема се, че ключът е в позиция 1. При това положение на извод $\overline{R}$ се подава ниско ниво и тригерът е нулиран. При превключване на ключа в позиция 2 извод $\overline{R}$ преминава във високо ниво, заради изтеглящия резистор R2, а извод $\overline{S}$ преминава в ниско ниво при първият контакт с ключа. Въпреки че $\overline{S}$ остава в ниско ниво за много кратко време поради трепкането на ключа, това е достатъчно да установи изхода на тригера. Трепканията не променят това, тъй като при отделяне на ключа от позиция 2 на двата извода се подава високо ниво (заради изтеглящите резистори R1 и R2), а при тази входна комбинация тригерът „помни" предишното си състояние (вижте отново Табл.19) и остава установен. Аналогичен е и резултатът при последващо превключване на ключа в позиция 1.

8.2.2 Асинхронен RS тригер с разрешаващ вход

RS тригерите, описани в предходната подточка, имат един съществен недостатък, а именно не са устойчиви на смущения във входните сигнали. Разгледайте диаграмата показана на следващата фигура.

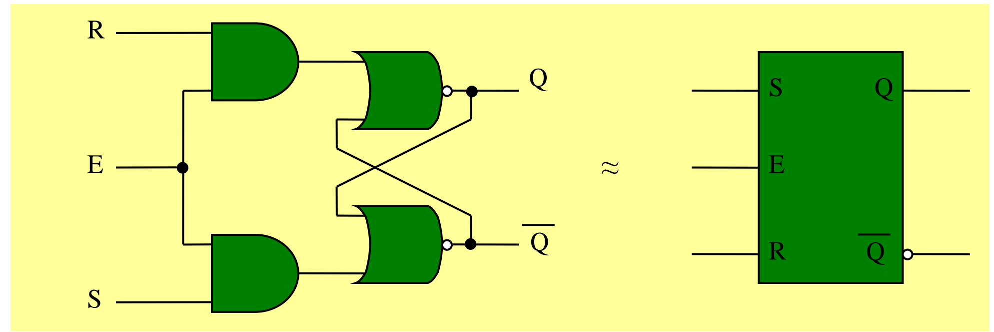


S	R	Q	$\overline{Q}$
0	0	Q	$\overline{Q}$
0	1	0	1
1	0	1	0
1	1	0	0

Фиг. 49 Недостатък на асинхронен RS тригер

Смущението показано на горната фигура, води до подаване на непозволена комбинация на входовете на тригера (S = 1, R = 1). При тази комбинация и двата изхода на тригера се нулират. Ефектът от появата на смущения във входните сигнали може да бъде намален чрез добавяне на разрешаващ вход (Фиг.50).

Входните сигнали се възприемат само ако са разрешени (E = 1). Ако входните сигнали са забранени (E = 0), те не оказват влияние на тригера и той запазва последните си стойности на изхода (Табл.20).



Фиг. 50 Асинхронен RS тригер с активни високи входове и разрешаващ вход

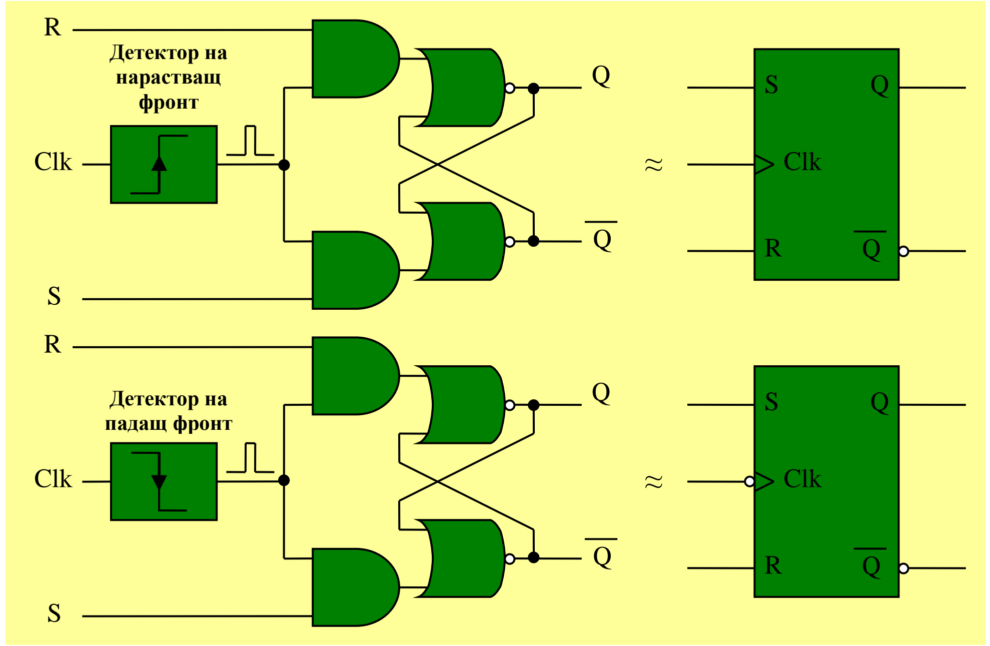
E	S	R	Q	$\overline{Q}$	Коментар
0	x	x	Q	$\overline{Q}$	Съхраняване на предишното състояние на изходите. Стойностите на входовете се игнорират.
1	0	0	Q	$\overline{Q}$	Съхраняване на предишното състояние на изходите
1	0	1	0	1	Нулиране на тригера
1	1	0	1	0	Установяване на тригера
1	1	1	0	0	Забранена комбинация

Табл. 20 Таблица на истинност на Асинхронен RS тригер с активни високи входове и разрешаващ вход

8.2.3 Синхронен RS тригер

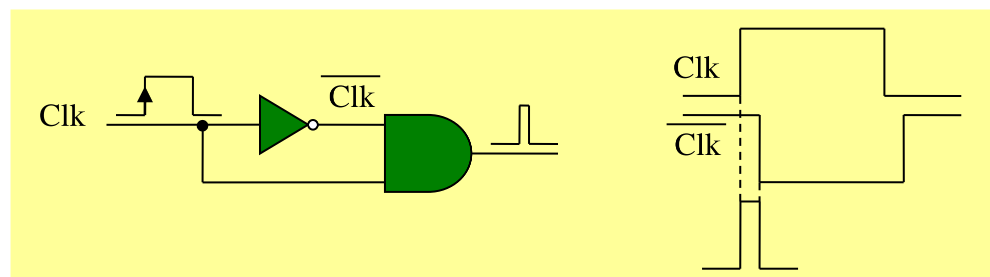
Разрешаващият вход не отстранява напълно ефекта от смущения във входните сигнали. Ако разрешаващият сигнал е активен за по-дълго време, всяко смущение във входните сигнали, което се появи през това време, ще се отрази на изхода на тригера. Този недостатък може да бъде минимизиран чрез използването на тактов (синхронизиращ) вход Clk. Входните сигнали се възприемат само при наличие на нарастващ или падащ фронт на тактовия вход. Следващата фигура показва вътрешната структура и символа за обозначаване на синхронен RS тригер с

положителен и отрицателен тактов вход съответно. Тактовият сигнал постъпва на детектор на фронт (положителен или отрицателен). При детектирането на съответния фронт детекторът генерира кратък положителен импулс, който действа като разрешаващ сигнал за входните сигнали на тригера.

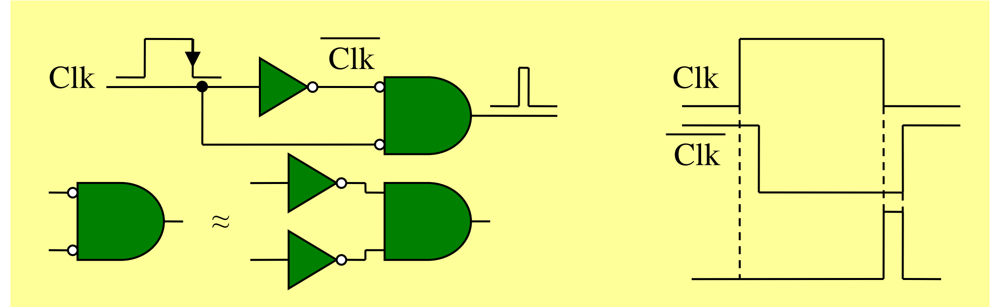


Фиг. 51 Синхронен RS тригер с активни високи входове и тактов вход

Фиг.52 и 53 показват вътрешната структура на детектора на положителен и отрицателен фронт съответно.



Фиг. 52 Детектор на нарастващ фронт



Фиг. 53 Детектор на падащ фронт

Показаните схеми използват факта, че между изходния и входния сигнал на инвертора има известно дефазиране (закъснение).

Следващите таблици на истинност описват работата на синхронните RS тригери с положителен и отрицателен тактов вход съответно.

Clk	S	R	Q	$\overline{Q}$	Коментар
-	x	x	Q	$\overline{Q}$	Съхраняване на предишното състояние на изходите. Стойностите на входовете се игнорират.
↑	0	0	Q	$\overline{Q}$	Съхраняване на предишното състояние на изходите
↑	0	1	0	1	Нулиране на тригера
↑	1	0	1	0	Установяване на тригера
↑	1	1	0	0	Забранена комбинация

Табл. 21 Таблица на истинност на синхронен RS тригер с активни високи входове и положителен тактов вход

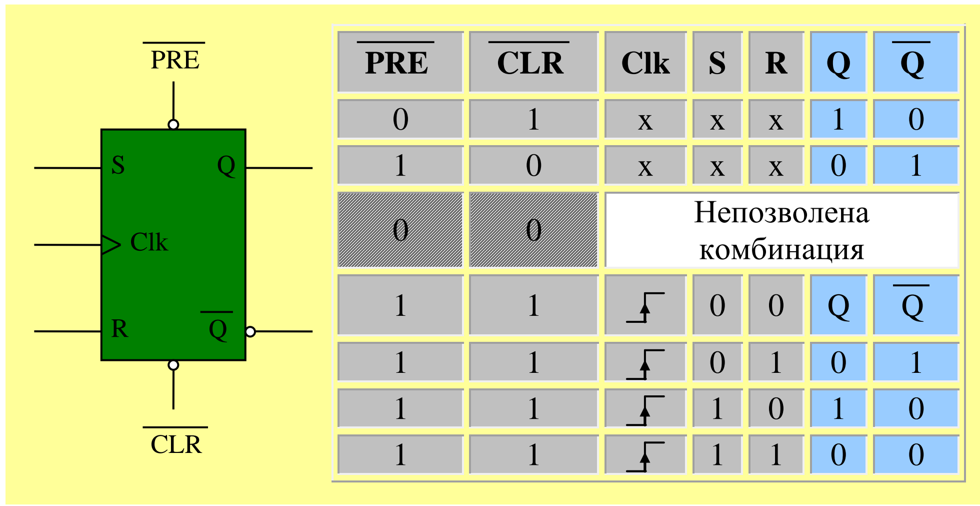
Clk	S	R	Q	$\overline{Q}$	Коментар
-	x	x	Q	$\overline{Q}$	Съхраняване на предишното състояние на изходите. Стойностите на входовете се игнорират.
↓	0	0	Q	$\overline{Q}$	Съхраняване на предишното състояние на изходите

↓	0	1	0	1	Нулиране на тригера
↓	1	0	1	0	Установяване на тригера
↓	1	1	0	0	Забранена комбинация

Табл. 22 Таблица на истинност на синхронен RS тригер с активни високи входове и отрицателен тактов вход

Синхронната работа на тригерите е от голямо значение при компютрите, където всички действия трябва да се извършват в строго определен ред и момент.

Входовете R и S на синхронния RS тригер се наричат още синхронни входове, тъй като те оказват влияние на изхода само при наличие на синхронизиращ сигнал на вход Clk. За по-голяма гъвкавост тригерите могат да имат допълнителни асинхронни входове за нулирани и установяване на тригера, работещи независимо от тактовия вход и синхронните входове. С тяхна помощ тригерът може да бъде установен или нулиран във всеки един момент независимо от стойностите на другите входове. Тези входове обикновено се обозначават като PRE (PRESET) за установяване и CLR (CLEAR) за нулиране (Фиг.54).



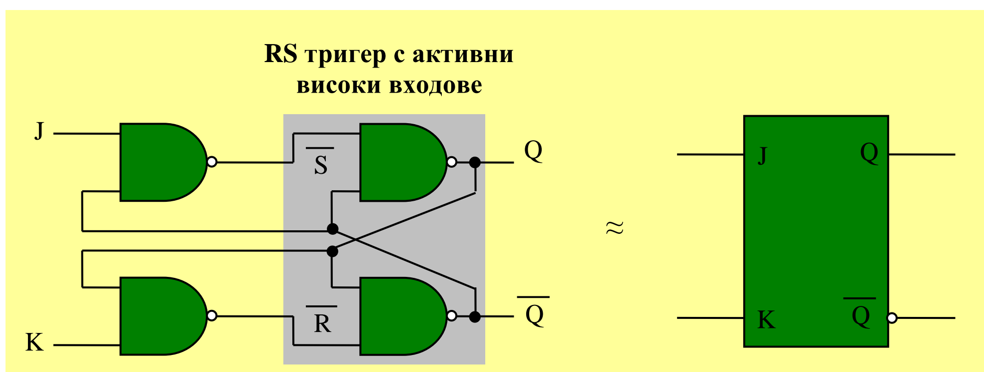
$\overline{PRE}$	$\overline{CLR}$	Clk	S	R	Q	$\overline{Q}$
0	1	x	x	x	1	0
1	0	x	x	x	0	1
0	0	Непозволена комбинация				
1	1	↑	0	0	Q	$\overline{Q}$
1	1	↑	0	1	0	1
1	1	↑	1	0	1	0
1	1	↑	1	1	0	0

Фиг. 54 Синхронен RS тригер с асинхронни входове

На горната фигура асинхронните входове са активни при подаване на ниско ниво на съответния извод. Това е подчертано и с чертата над имената на изводите, както и с кръгчетата на изводите. По тази причина подаването на лог.0 едновременно на тези изводи не е позволено.

8.3 JK тригери

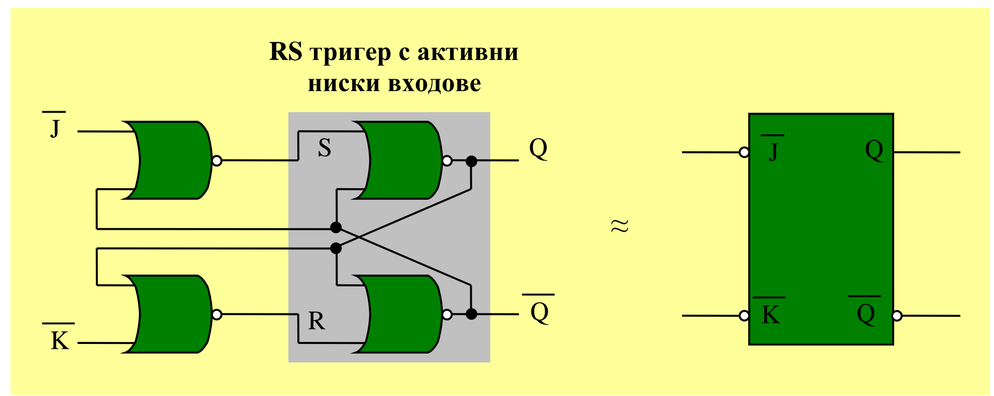
Този тригер е най-универсалния от всички типове тригери. Фиг.55 и 56 показват вътрешната структура на асинхронен JK тригер с активни високи и активни ниски входове съответно и символите, с които се обозначават, а Табл.17 и Табл.18 са таблиците на истинност на тези тригери.



Фиг. 55 Асинхронен JK тригер с активни високи входове

J	K	Q	$\overline{Q}$	Коментар
0	0	Q	$\overline{Q}$	Съхраняване на предишното състояние на изходите
0	1	0	1	Нулиране на тригера
1	0	1	0	Установяване на тригера
1	1	$\overline{Q}$	Q	Превключване на тригера: 0 → 1 или 1→ 0

Табл. 23 Таблица на истинност на асинхронен JK тригер с активни високи входове



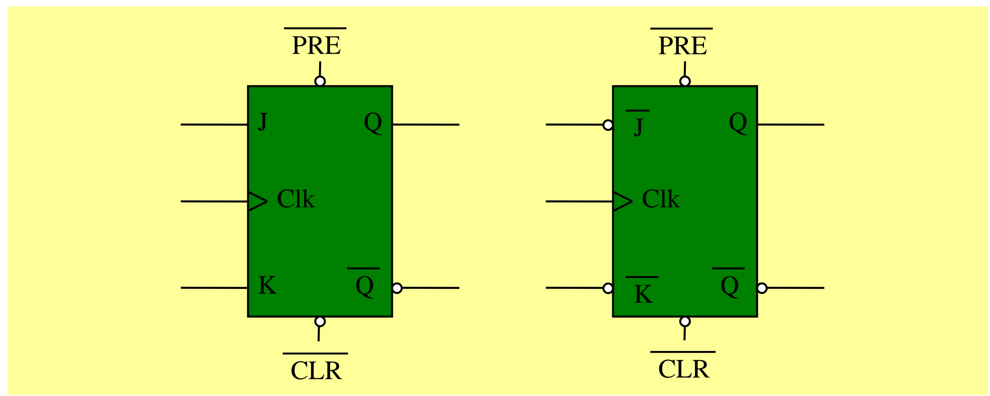
Фиг. 56 Асинхронен JK тригер с активни ниски входове

$\overline{J}$	$\overline{K}$	Q	$\overline{Q}$	Коментар
0	0	$\overline{Q}$	Q	Превключване на тригера: 0 → 1 или 1→ 0
0	1	0	1	Установяване на тригера
1	0	1	0	Нулиране на тригера
1	1	Q	$\overline{Q}$	Съхраняване на предишното състояние на изходите

Табл. 24 Таблица на истинност на асинхронен JK тригер с активни ниски входове

JK тригерът работи по същия начин като RS тригер, но премахва недостатъка със забранената входна комбинация. При тази входна комбинация JK тригерът се превключва (ако преди тази комбинация Q е 0, Q се превключва в 1 и обратно).

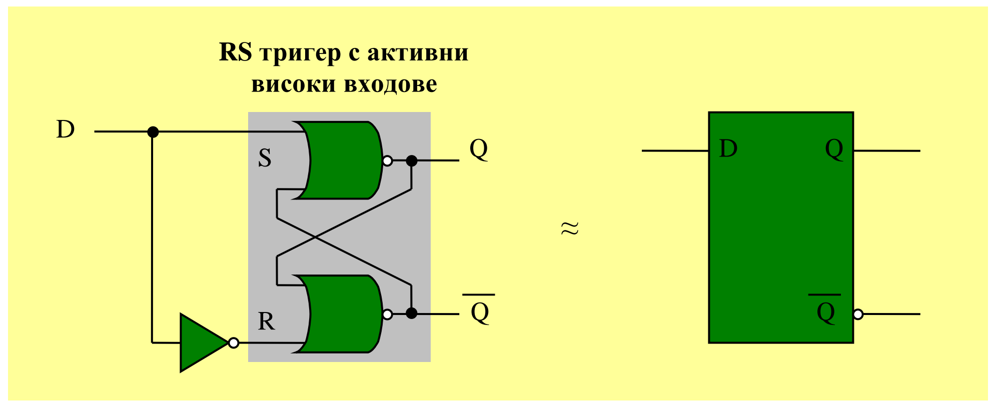
Както и RS тригерите, JK тригерите могат да имат тактов вход Clk (положителен или отрицателен) и асинхронни входове PRE и CLR (Фиг.57)



Фиг. 57 Синхронни JK тригери с асинхронни входове

8.4 D тригери

Фиг.58 показва вътрешната структура и символа за обозначаване на асинхронен D тригер.



Фиг. 58 Асинхронен D тригер

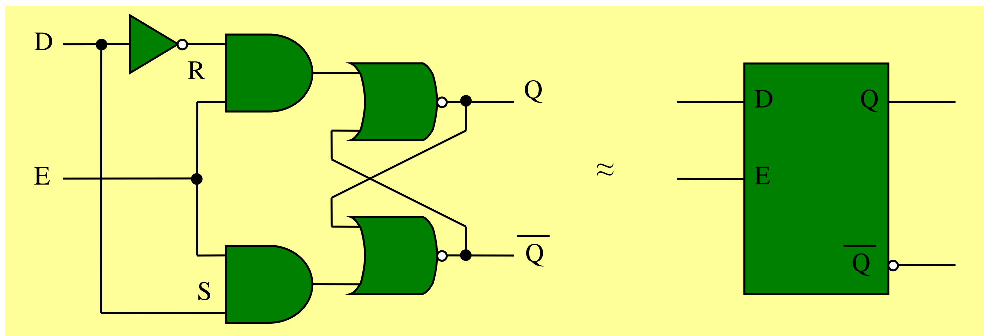
Както се вижда от фигурата, D тригерът лесно се получава от RS тригер с активни ниски входове.

D тригерът работи подобно на буфер. Стойността на входа D се появява на изхода Q, но с известно закъснение. Оттам идва и името на тригера – D (**delay** - закъснение) (Табл.25).

D	Q	$\overline{Q}$
0	0	1
1	1	1

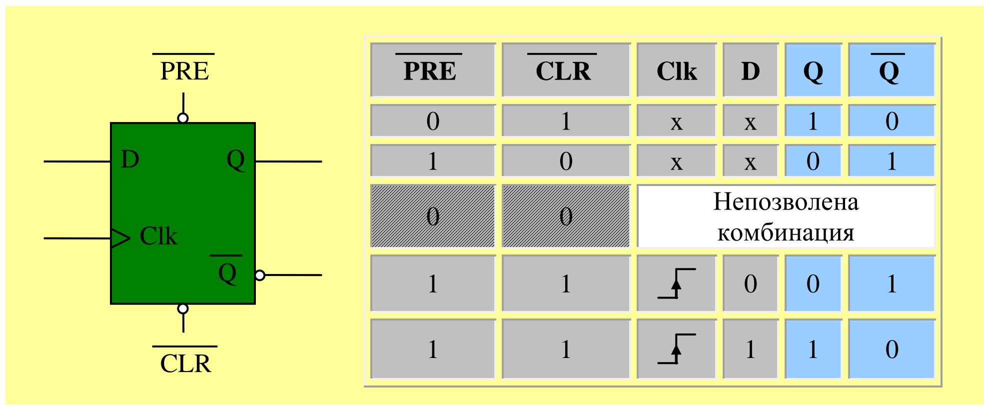
Табл. 25 Таблица на истинност на D тригер

Аналогично може да получите и асинхронен D тригер с разрешаващ вход (Фиг.59). Стойността на входа D се появява на изхода само когато разрешаващият сигнал е активен (E = 1).



Фиг. 59 Асинхронен D тригер с разрешаващ вход

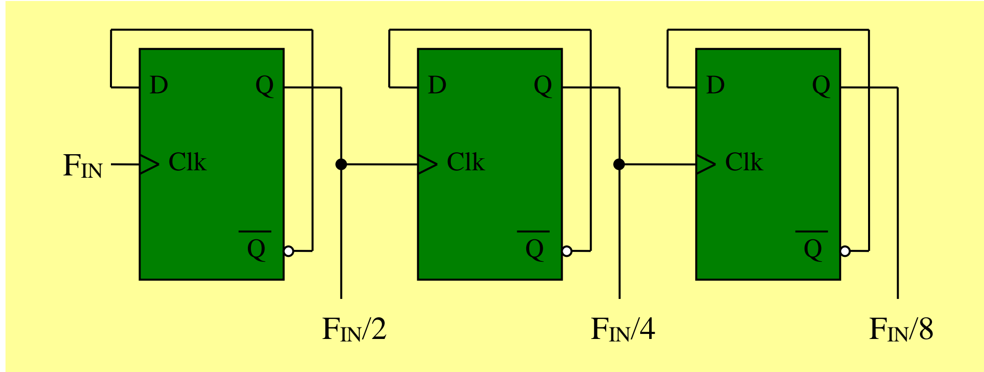
Фиг.60 описва D тригер с положителен тактов вход и асинхронни входове. Стойността на входа D се появява на изхода само по време на нарастващ фронт на тактовия сигнал.



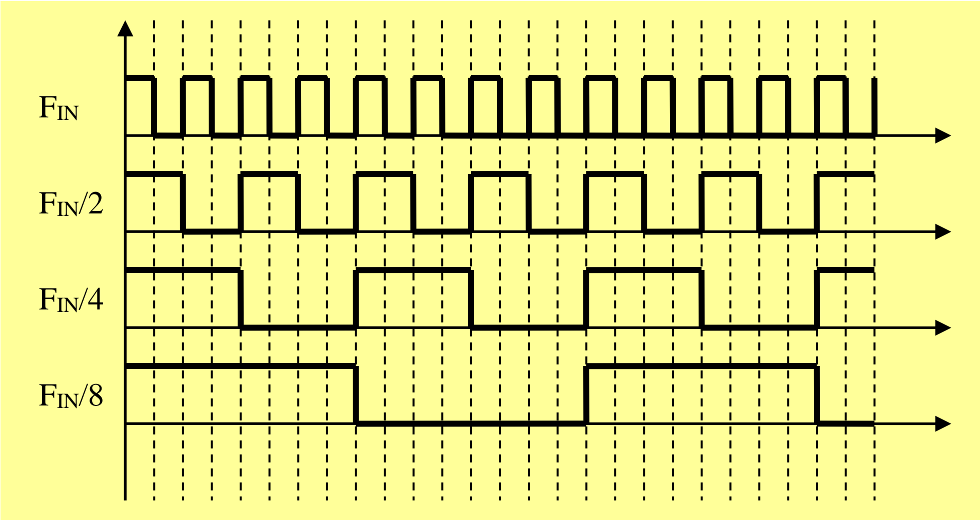
$\overline{PRE}$	$\overline{CLR}$	Clk	D	Q	$\overline{Q}$
0	1	x	x	1	0
1	0	x	x	0	1
0	0	Непозволена комбинация			
1	1	↑	0	0	1
1	1	↑	1	1	0

Фиг. 60 Синхронен D тригер с асинхронни входове

Фиг.61 показва реализирането на делител на честота със синхронен D тригер, а Фиг.62 показва времедиаграмите.



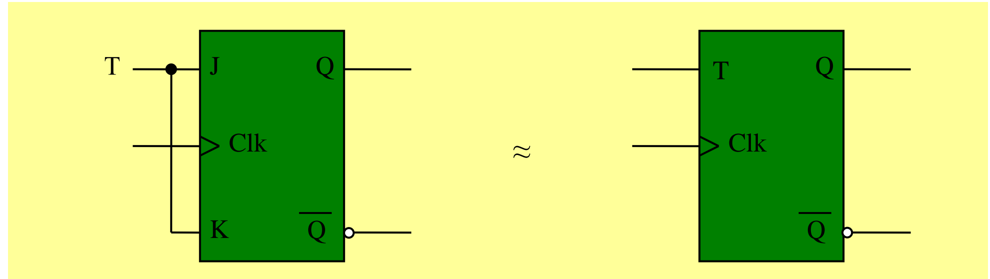
Фиг. 61 Делител на честота с D тригер



Фиг. 62 Делител на честота с D тригер - времедиаграма

8.5 T тригери

Фиг.63 показва вътрешната структура и символа за обозначаване на T (**Toggle**) тригер. Табл.26 описва принципа на работа на T тригера.

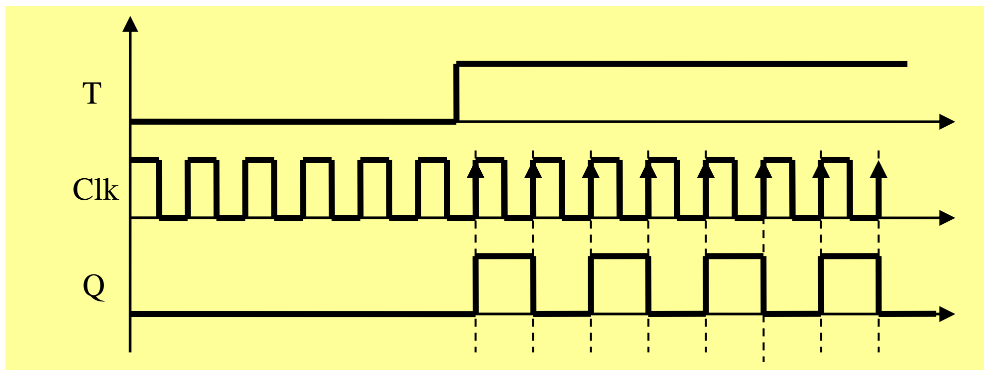


Фиг. 63 Т тригер

Clk	T	Q	Коментар
x	0	Q	Изходите запазват предишното си състояние
↑	1	$\overline{Q}$	Превключване на тригера: 0 → 1 или 1→ 0

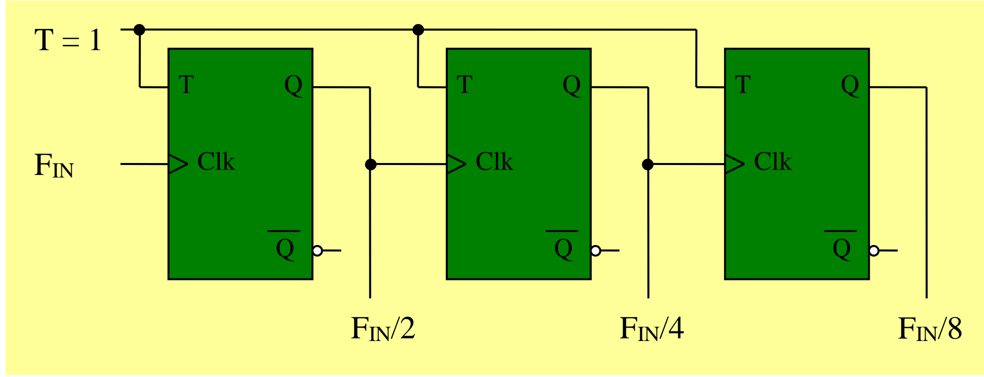
Табл. 26 Таблица на истинност на Т тригер

Както се вижда, Т тригерът се получава чрез свързване на изводите J и K на JK тригер. Ако погледнете таблицата на този тригер, лесно ще разберете как работи така формирания Т тригер. Когато на вход Т има лог.0, изходите запазват предишното си състояние, независимо от тактовия вход. Когато на вход Т има лог.1, изходите се превключват на всеки нарастващ фронт на тактовия вход (ако Q е бил 0, той се превключва в 1 и обратно). Времедиаграмите на следващата фигура описват описаният процес нагледно.



Фиг. 64 Времедиаграми на Т тригер

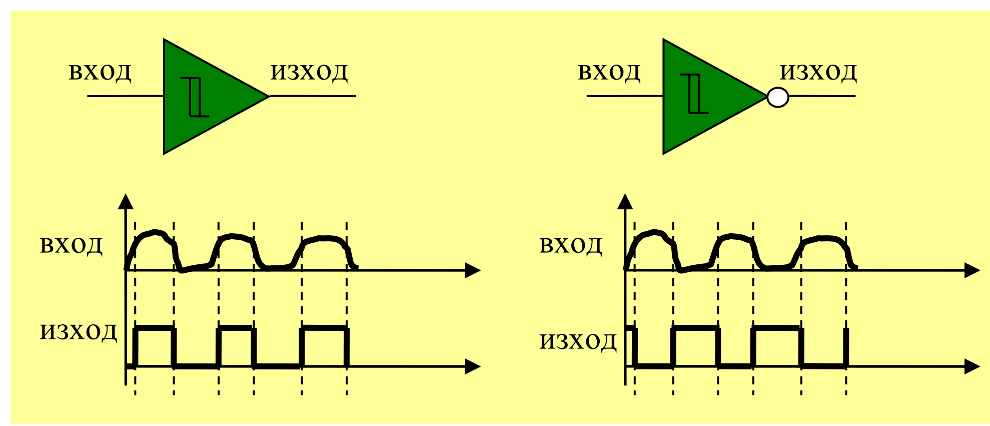
Забележете, че когато тригерът е в режим на превключване (T = 1), сигналът в Q изхода е с двойно по-ниска честота от тази на тактовия вход. Ето защо T тригерът може да се използва за реализиране на делител на честота (Фиг.65).



Фиг. 65 Делител на честота с T тригер

8.6 Тригер на Шмит

Тригерът на Шмит е устройство с един вход и един изход. Неговото предназначение е да преобразува плавно изменящи се входни сигнали в правоъгълни (Фиг.66).

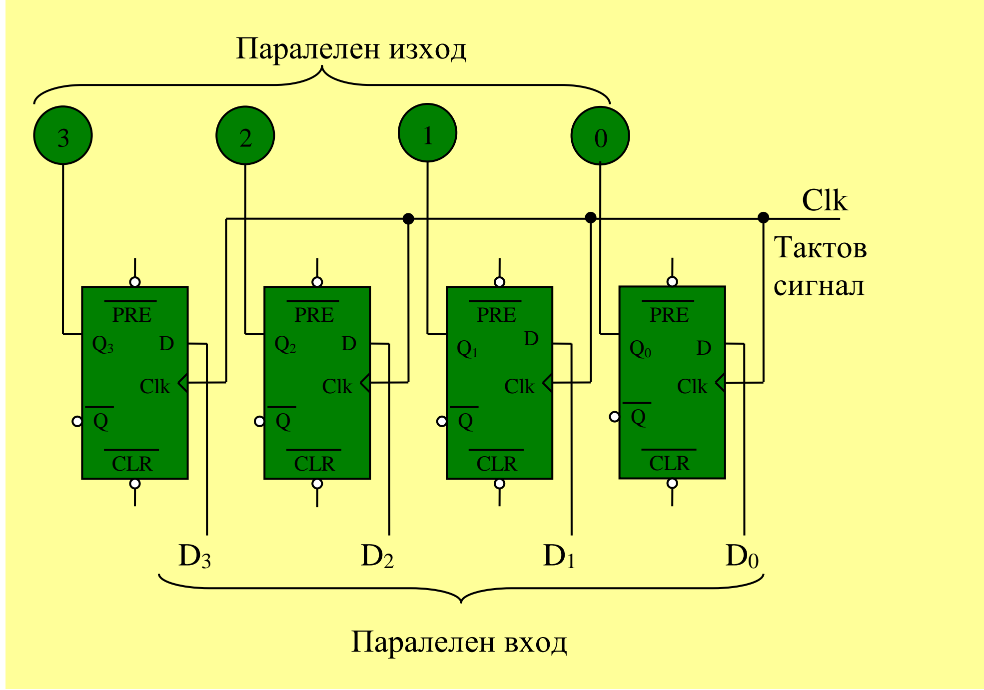


Фиг. 66 Тригер на Шмит с прав и инвертиран изход

9 Регистри

9.1 Общи сведения

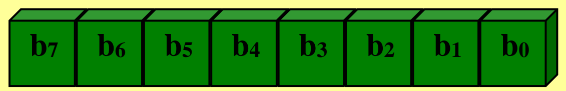
Регистърът е съвкупност от тригери с общ тактов вход. Всеки тригер съхранява 1 бит информация, т.е. . 8-битов регистър се състои от 8 тригера, 16-битов регистър се състои от 16 тригера и т.н. Следващата фигура показва проста схема на 4-битов регистър от D тригери.



Фиг. 67 4-битов регистър

Данните на входовете $D_0 \div D_3$ едновременно се появяват в изходите при появата на активен фронт (в случая нарастващ) на тактовия вход Clk.

От гледна точка на един програмист регистърът може да се разглежда като съвкупност от кутийки, всяка от които може да съхранява един бит информация (Фиг.68). Всяка кутийка представлява тригер.



Фиг. 68 8-битов регистър - абстрактен изглед

Освен да съхраняват данни, регистрите могат да извършват дейности като преместване на данните и броене. Първият тип регистри се наричат **преместващи регистри**, а вторият тип – **броячни регистри** или просто **броячи**.

В зависимост от начина на въвеждане на данните, преместващите регистри могат да се класифицират като:

- Преместващи регистри с последователно въвеждане;
- Преместващи регистри с паралелно въвеждане.

В зависимост от начина на извеждане на данните, те могат да се класифицират като:

- Преместващи регистри с последователно извеждане;
- Преместващи регистри с паралелно извеждане.

В зависимост от посоката на преместване на данните, те могат да се класифицират като:

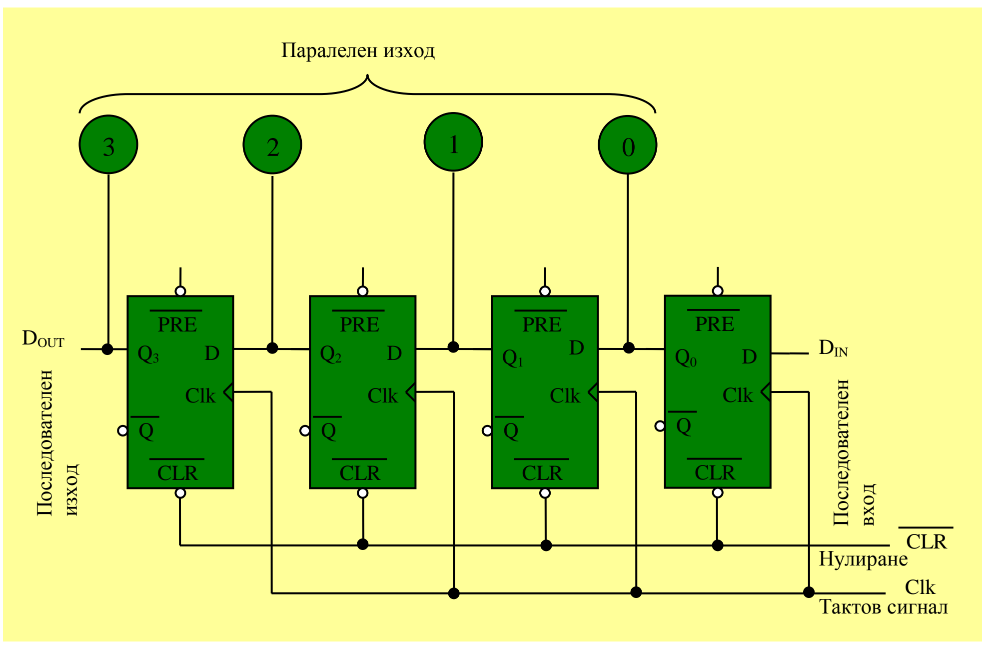
- Преместващи регистри наляво;
- Преместващи регистри надясно;

Преместващи регистри, които обединяват в себе си всички по-горе функционалности, се наричат универсални преместващи регистри.

Следващите подточки разглеждат накратко преместващите регистри. Броячите са разгледани в 11 Броячи.

9.2 Преместващи регистри с последователно въвеждане

Фиг.69 показва 4-битов преместващ регистър с последователно въвеждане, изграден с D тригери.



Фиг. 69 Преместващ регистър наляво с последователно въвеждане

Данните се въвеждат чрез последователния вход D_{IN}. На всеки тактов импулс се въвежда само по един бит. Табл.27 показва таблицата на истинност.

№	входове			изходи			
	$\overline{CLR}$	D_{IN}	такт	Q_0	Q_1	Q_2	$Q_3(D_{OUT})$
1	0	x	x	0	0	0	0
2	1	1	-	0	0	0	0
3	1	1	↑	1	0	0	0
4	1	1	↑	1	1	0	0
5	1	1	↑	1	1	1	0
6	1	1	↑	1	1	1	1

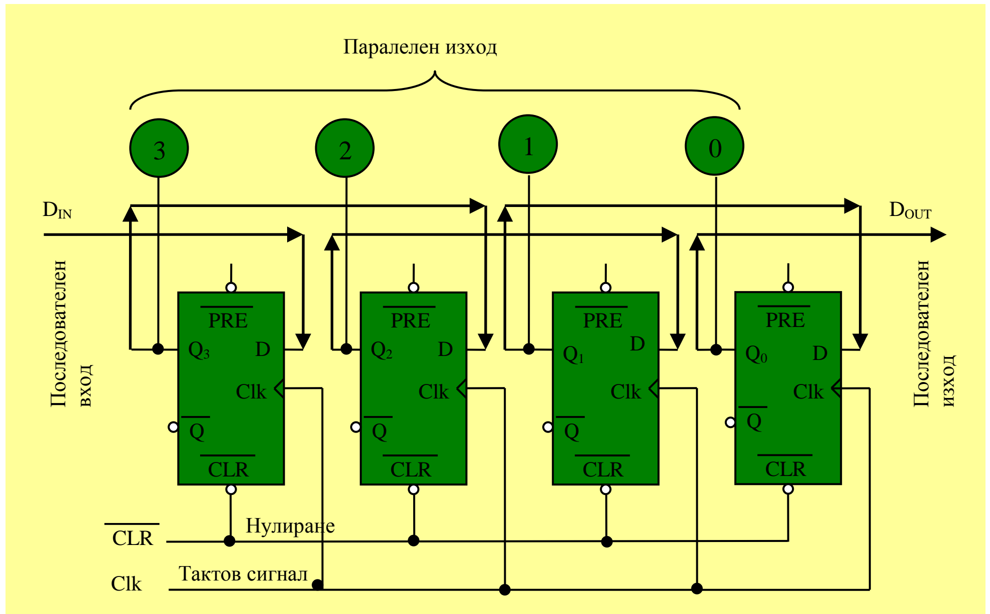
7	1	0	↑	0	1	1	1
8	1	0	↑	0	0	1	1
9	1	0	↑	0	0	0	1
10	1	1	↑	1	0	0	0
11	1	0	↑	0	1	0	0
12	1	0	↑	0	0	1	0
13	1	1	↑	1	0	0	1
14	1	1	↑	1	1	0	0
15	1	1	↑	1	1	1	0

Табл. 27 Таблица на истинност на преместващ регистър с последователно въвеждане

Първият ред от таблицата показва нулиране на всички тригери чрез подаване на лог.0 на вход $\overline{CLR}$. Вторият ред показва подаване на 1 на последователния вход. Поради липса на активен тактов фронт на вход Clk изходите си остават нулирани. При постъпване на нарастващ фронт данните на входа D_{IN} се появяват в изхода Q_0 (ред 3). Ако проследите таблицата по-нататък, ще видите, че при всеки нарастващ тактов фронт данните на входа D_{IN} се появяват на изход Q_0, данните на Q_0 се появяват на Q_1, тези на Q_1 се появяват на Q_2 и т.н., т.е. . данните се преместват на всеки тактов сигнал.

Ако данните се вземат от четирите изхода едновременно, регистърът е с паралелно извеждане, а ако данните се вземат само от последния изход (Q_3), регистърът е с последователно извеждане.

Горният регистър премества данните наляво. За да се превърне в преместващ регистър надясно е необходимо изходите да се пресвържат както е показано на следващата фигура.

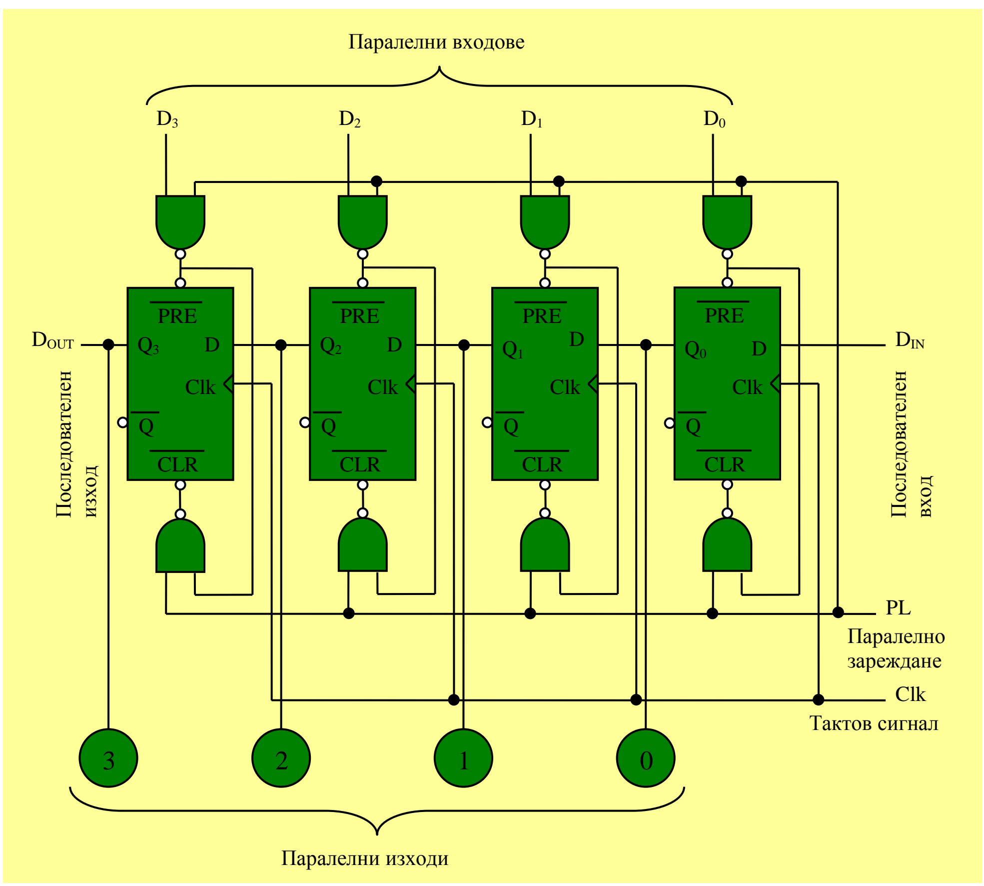


Фиг. 70 Преместващ регистър надясно с последователно въвеждане

Чрез добавяне на допълнителни логически елементи, преместващият регистър може да се направи да премества наляво или надясно в зависимост от управляващ посоката сигнал.

9.3 Преместващи регистри с паралелно въвеждане

Фиг.71 показва 4-битов преместващ регистър с паралелно въвеждане, изграден с D тригери. Данните се зареждат чрез паралелните входове $D_3 \div D_0$, свързани към асинхронните входове $\overline{PRE}$ и $\overline{CLR}$. Когато разрешаващият сигнал за паралелно зареждане PL е 1, данните на входовете $D_3 \div D_0$ се зареждат в преместващия регистър. Ако данните се вземат от четирите изхода едновременно, регистърът е с паралелно извеждане, а ако данните се вземат само от последния изход (Q_3), регистърът е с последователно извеждане.



Фиг. 71 Преместващ регистър наляво с паралелно въвеждане

Преместващите регистри с паралелно въвеждане/извеждане и последователно въвеждане/извеждане намират широко приложение в създаването на серийни комуникационни интерфейси в микроконтролерите. Типично данните с големина един байт се зареждат паралелно в преместващ регистър и след това на всеки активен фронт на тактовия сигнал данните се изпращат бит по бит (серийно) по линията за данни. При четене данните постъпват по линията за данни на последователния вход, а от там на всеки активен фронт на тактовия сигнал данните се зареждат в преместващия регистър. В 17.2.9 Комуникационни интерфейси ще намерите обобщена блокова схема на сериен комуникационен интерфейс, изграден с помощта на преместващи регистри.

10 Броячи

10.1 Общи сведения

Броячът е регистър, който брои импулси. Освен това броячите могат да изпълняват не толкова очевидни функции, например делене на честота. Основните градивни елементи на броячите са тригерите.

В зависимост от начина на превключване на тригерите броячите се делят на:

- асинхронни;
- синхронни.

В зависимост от посоката на броене броячите се делят на:

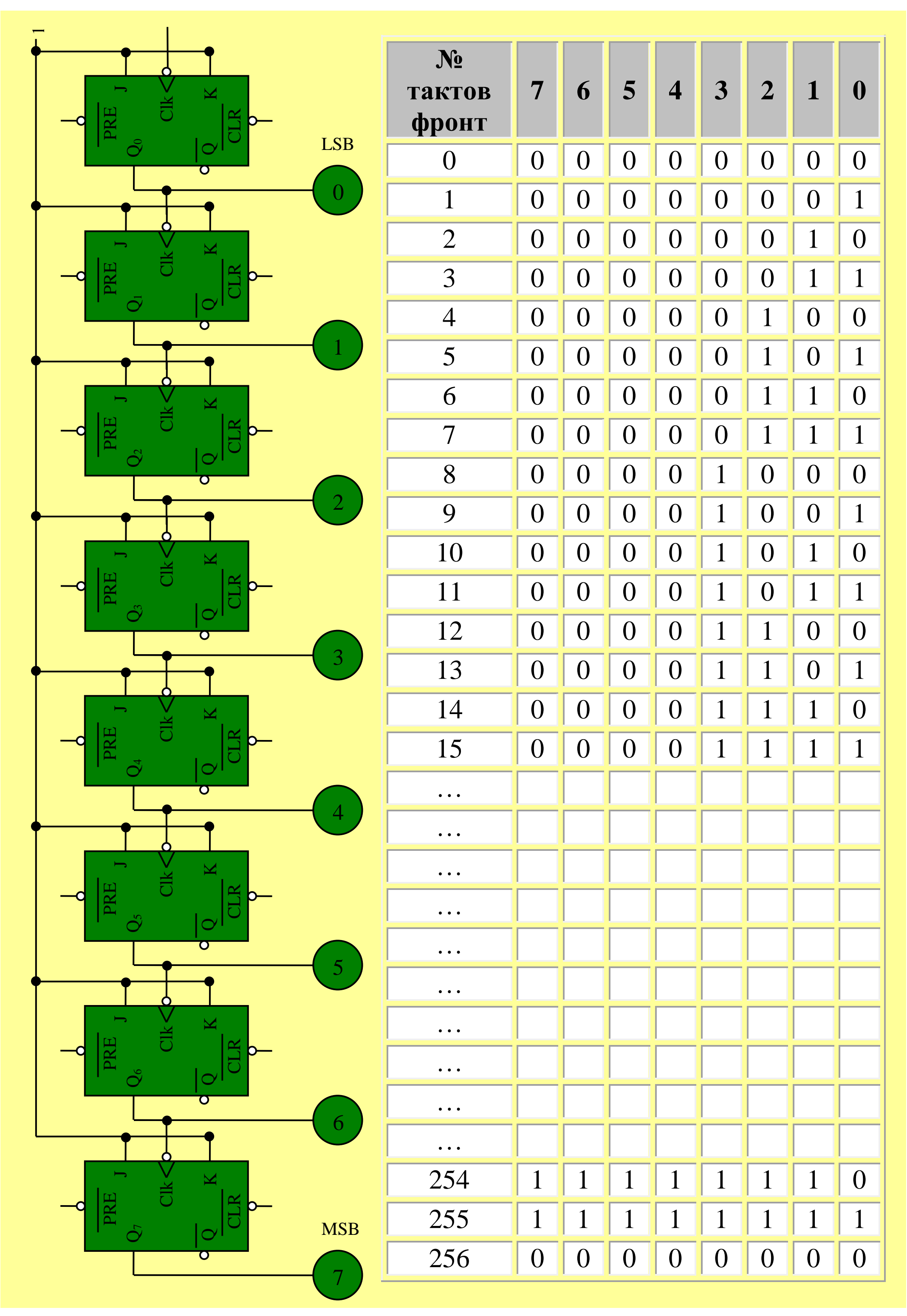
- прави;
- обратни (изваждащи).

Броят на различните състояния, през които преминава един брояч, се нарича **модул на брояча**.

Следващите подточки разглеждат накратко броячите.

10.2 Асинхронни броячи

На следващата фигура е показан 8-битов асинхронен прав брояч от JK тригери. Всички тригери работят в режим на превключване (J и K входовете са свързани към лог.1). Тактовият сигнал се подава само на тактовия вход на тригер 0 (първият отгоре надолу). Q изходът на всеки тригер е свързан към тактовия вход на следващия тригер.

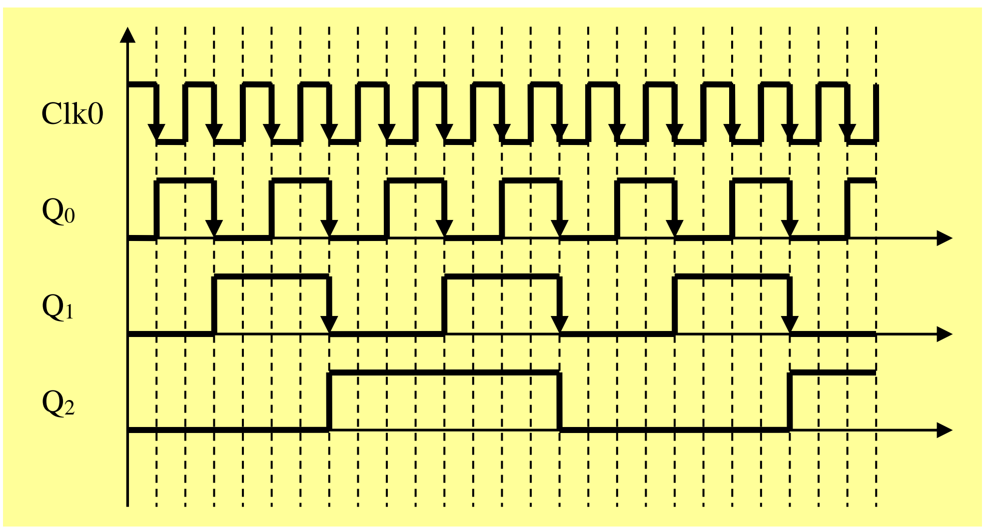


№ тактов фронт	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
10	0	0	0	0	1	0	1	0
11	0	0	0	0	1	0	1	1
12	0	0	0	0	1	1	0	0
13	0	0	0	0	1	1	0	1
14	0	0	0	0	1	1	1	0
15	0	0	0	0	1	1	1	1
…								
…								
…								
…								
…								
…								
…								
…								
…								
…								
254	1	1	1	1	1	1	1	0
255	1	1	1	1	1	1	1	1
256	0	0	0	0	0	0	0	0

Фиг. 72 8-битов асинхронен брояч

Тригерите се превключват по отрицателния фронт на тактовия сигнал. Нека предположим, че всички тригери са нулирани (с помощта на асинхронните входове $\overline{CLR}$). При постъпване на първия отрицателен фронт на тактовия сигнал, Q изходът на тригер 0 се превключва в 1. Изходите на всички останали тригери не се променят (стойността на брояча е 00000001 – 1 десетично). При постъпване на втория отрицателен фронт на тактовия сигнал, Q изходът на тригер 0 се превключва в 0, т.е. . на тактовия вход на тригер 1 се получава отрицателен фронт, което превключва тригер 1 в 1 (стойността на брояча става 00000010 – 2 десетично). При постъпване на третия отрицателен фронт на тактовия сигнал, Q изходът на тригер 0 се превключва в 1. Изходът на тригер 1 не се изменя поради липсата на отрицателен фронт на тактовия му вход (стойността на брояча става 00000011 – 3 десетично). При постъпване на четвъртия отрицателен фронт на тактовия сигнал, Q изходът на тригер 0 се превключва в 0. Това води до превключване на тригер 1 в 0, което пък води до превключване на тригер 2 в 1 ((стойността на брояча става 00000100 – 4 десетично). Този процес продължава, докато броячът достигне максималната си стойност 255 (всички Q изходи са в 1 → 11111111). При постъпване на 256-я отрицателен фронт на тактовия сигнал, изходът на тригер 0 се превключва в 0, което води до превключване на изхода на тригер 1 в 0 и т.н., докато всички тригери последователно се превключват в 0 (броячът се нулира → 00000000). Описаните по-горе процеси са илюстрирани на следващата фигура. Забележете, че сигналът на изход Q_0 е с двойно по-ниска честота от тази на тактовия сигнал, сигналът на изход Q_1 е с четири пъти по-ниска честота от тази на тактовия сигнал и т.н., т.е. . броячът работи като делител на честота с коефициенти на делене 2, 4, 8, 16, 32, 64, 128 и 256.

Асинхронните броячи се наричат още броячи с последователен пренос, тъй като всеки тригер превключва



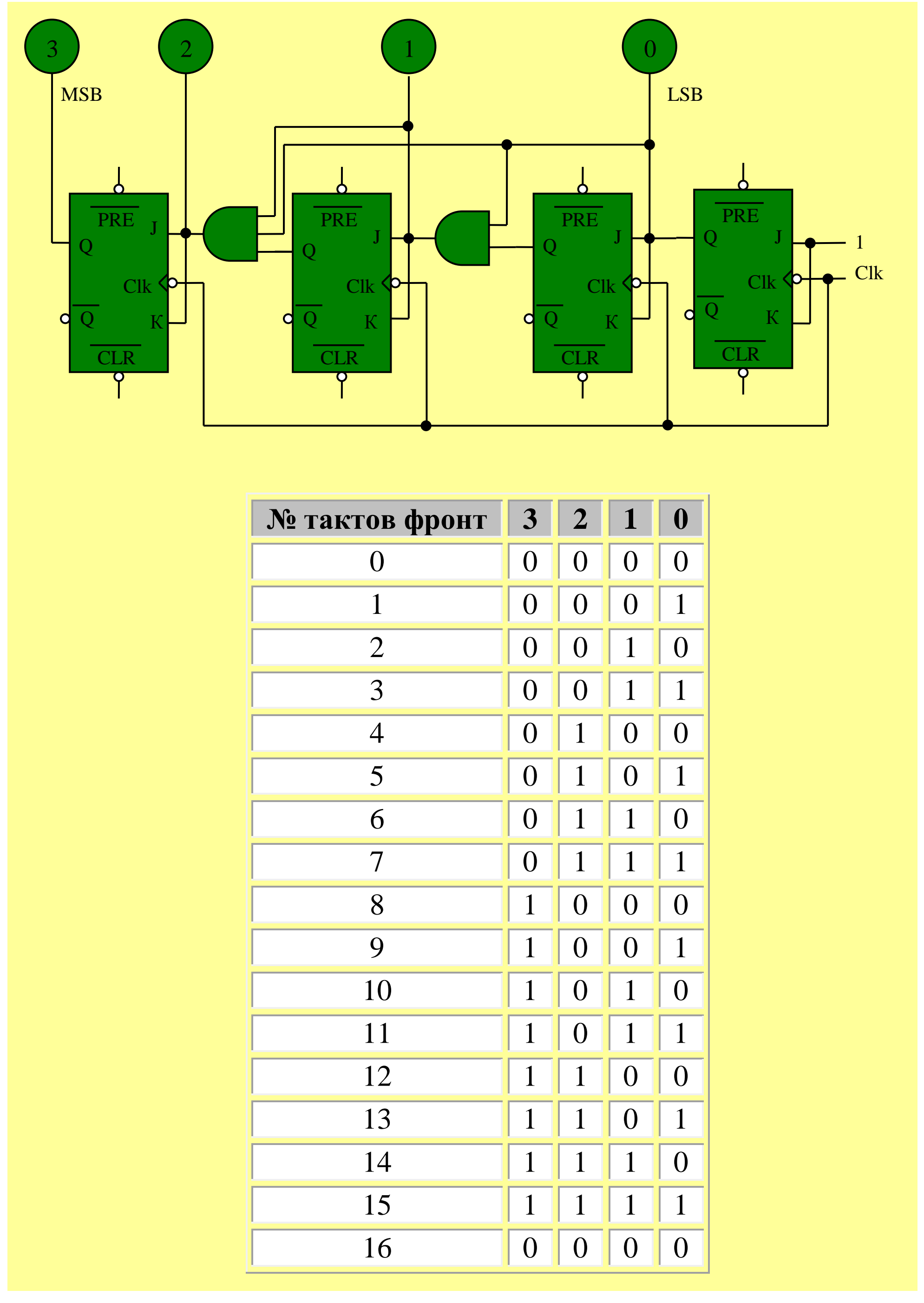
Фиг. 73 8-битов асинхронен брояч - времедиаграма

следващия, който пък превключва по-следващия и т.н. Превключването на един тригер отнема известно време. Това може да бъде проблем при високи честоти на тактовия сигнал.

Ако промените схемата на брояча от Фиг.62, така че свържете $\overline{Q}$ изхода на всеки тригер към тактовия вход на следващия, ще получите обратен (изваждащ) брояч.

10.3 Синхронни броячи

При синхронните броячи всички тригери се превключват едновременно (синхронно) при постъпването на активен фронт на тактовия сигнал. Фиг.74 показва 4-битов прав синхронен брояч и таблицата му на истинност. Забележете, че тактовите входове на всички тригери са свързани заедно. Няма да разглеждам подробно как работи този брояч. Важното е да знаете, че на всеки отрицателен фронт на тактовия сигнал броячът се увеличава с 1.



№ тактов фронт	3	2	1	0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1
16	0	0	0	0

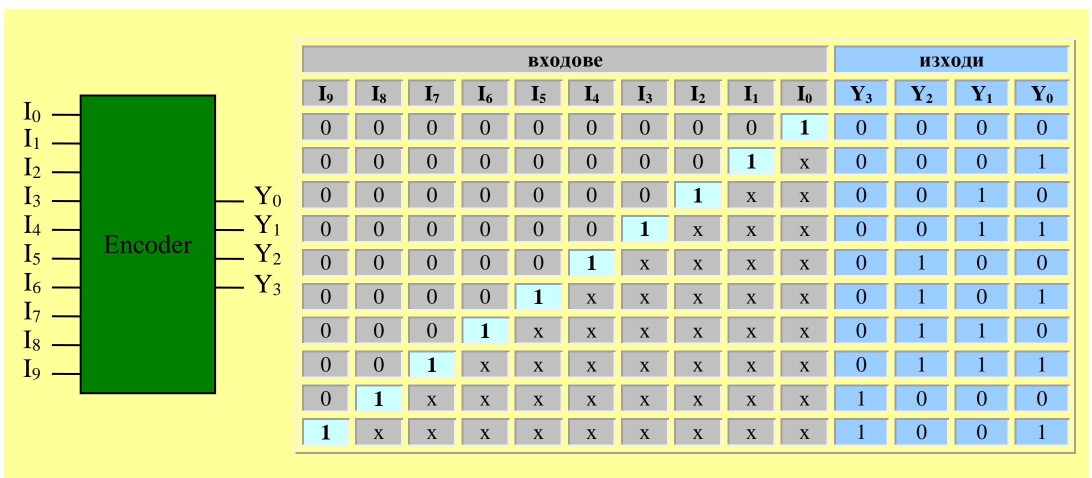
Фиг. 74 4-битов синхронен брояч

Броячите се използват в микроконтролерите за изграждане на таймерни модули с най-разнообразни функционалности.

В 17.2.2 Таймери, 17.2.3 ШИМ, 17.2.4 Сравнение , 17.2.5 Прихващане и 17.2.6 Стражеви таймер ще научите повече за употребата на броячите в микроконтролерите.

11 Шифратори и дешифратори

Шифраторът (**Enoder**) е устройство, което преобразува единичен сигнал на един от входовете си в n-битов двоичен код. Фиг.75 показва обозначаването на шифратор с 10 входа и 4 изхода и таблицата му на истинност.

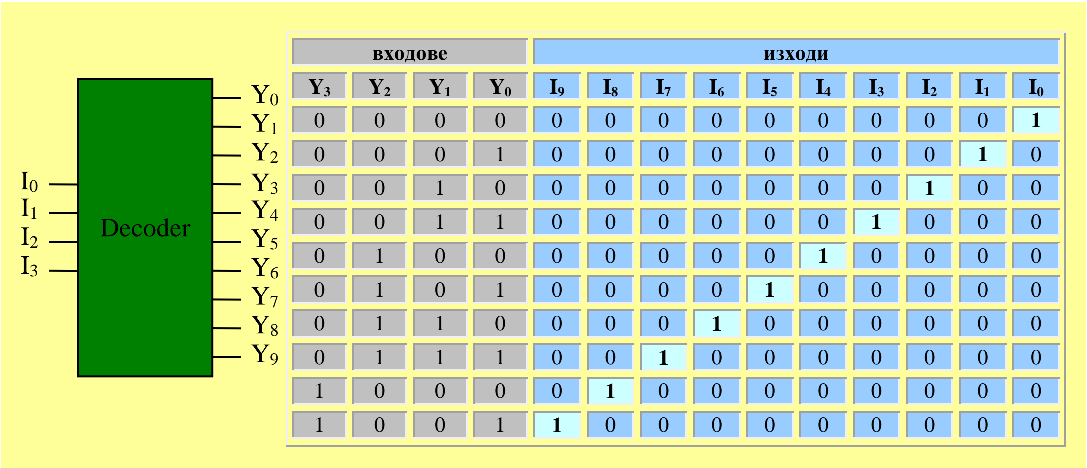


входове										изходи			
I_9	I_8	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	**1**	0	0	0	0
0	0	0	0	0	0	0	0	**1**	x	0	0	0	1
0	0	0	0	0	0	0	**1**	x	x	0	0	1	0
0	0	0	0	0	0	**1**	x	x	x	0	0	1	1
0	0	0	0	0	**1**	x	x	x	x	0	1	0	0
0	0	0	0	**1**	x	x	x	x	x	0	1	0	1
0	0	0	**1**	x	x	x	x	x	x	0	1	1	0
0	0	**1**	x	x	x	x	x	x	x	0	1	1	1
0	**1**	x	x	x	x	x	x	x	x	1	0	0	0
1	x	x	x	x	x	x	x	x	x	1	0	0	1

Фиг. 75 Шифратор 10x4

При подаване на лог.1 на даден вход на изхода се получава двоичния код, отговарящ на номера на входа. Например ако на вход I_0 (вход с номер нула) се подаде лог.1, на изхода се получава 0000_2 (0 десетично), или ако на вход I_8 (вход с номер осем) се подаде лог.1, на изхода се получава 1000_2 (8 десетично). Показаният шифратор се нарича още приоритетен шифратор, тъй като всеки вход си има приоритет. Колкото е по-голям номера на входа, толкова по-висок приоритет има той. Приоритетът се използва в случай, че няколко входа се активират едновременно. В този случай само входът с най-висок приоритет са взема под внимание. Например ако на входове I_0 и I_8 се подава едновременно лог.1, на изхода се получава 1000_2 (8 десетично), т.е. . единицата на вход I_0 се игнорира. Тъй като схемата на приоритетния шифратор е сравнително сложна тя няма да бъде представена тук.

Дешифраторът (**Decoder**) е устройство, което работи обратно на шифратора, т.е. . подаването на двоична стойност на входа води до активирането на определен изход. Фиг.76 показва обозначаването на дешифратор с 4 входа и 10 изхода и таблицата му на истинност.



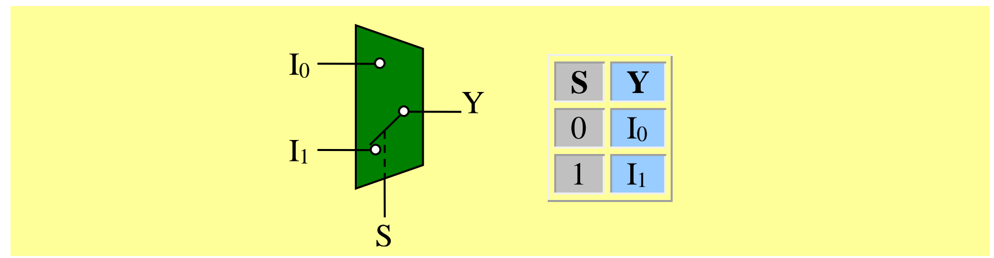
входове				изходи									
Y_3	Y_2	Y_1	Y_0	I_9	I_8	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
0	0	0	0	0	0	0	0	0	0	0	0	0	**1**
0	0	0	1	0	0	0	0	0	0	0	0	**1**	0
0	0	1	0	0	0	0	0	0	0	0	**1**	0	0
0	0	1	1	0	0	0	0	0	0	**1**	0	0	0
0	1	0	0	0	0	0	0	0	**1**	0	0	0	0
0	1	0	1	0	0	0	0	**1**	0	0	0	0	0
0	1	1	0	0	0	0	**1**	0	0	0	0	0	0
0	1	1	1	0	0	**1**	0	0	0	0	0	0	0
1	0	0	0	0	**1**	0	0	0	0	0	0	0	0
1	0	0	1	**1**	0	0	0	0	0	0	0	0	0

Фиг. 76 Дешифратор 4x10

Подаване на стойност 0000_2 (0 десетично) на входа води до активирането на изход Y_0, подаването на стойност 1000_2 (8 десетично) води до активирането на изход Y_8 и т.н.

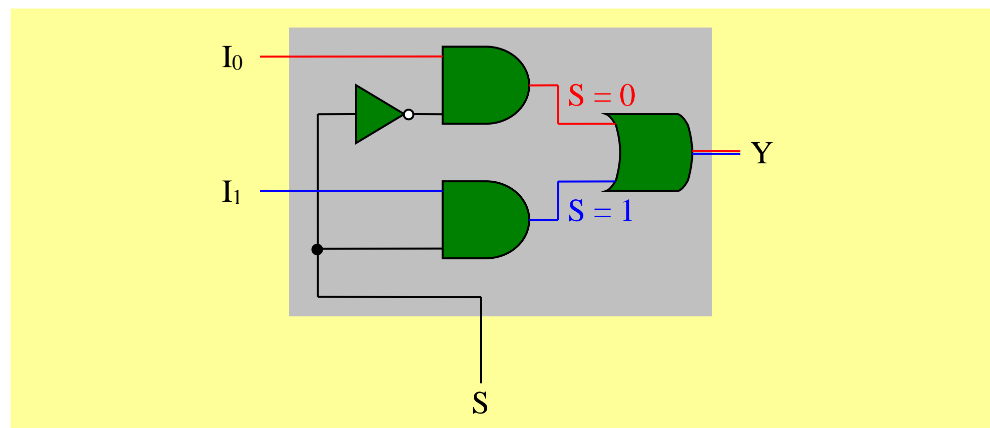
12 Мултиплексори и демултиплексори

Мултиплексорът е устройство с няколко входа за данни и един изход. С помощта на селекторни входове се избира кой от входовете за данни да се подаде на изхода, т.е. . мултиплексорът е многовходов цифров ключ. Фиг.77 показва абстрактен изглед на мултиплексор с два входа. В зависимост от селекторния вход S изходът Y е свързан с вход I_0 (S = 0) или I_1 (S = 1).



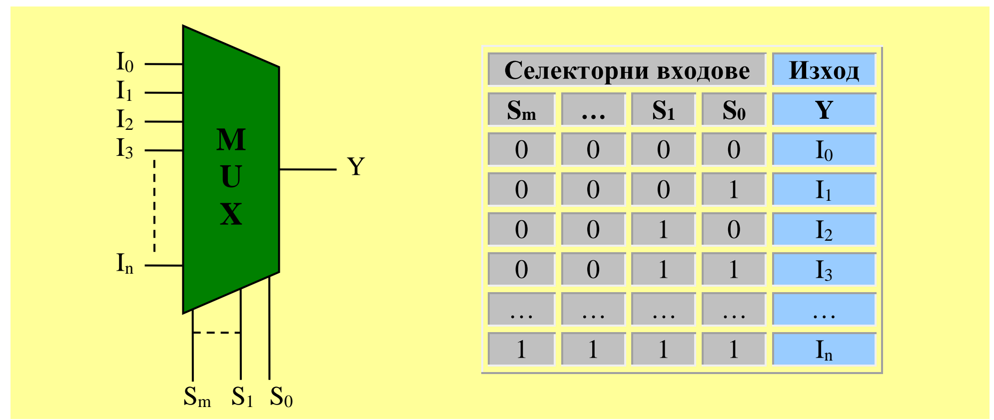
Фиг. 77 Мултиплексор с два входа

Мултиплексор с два входа може да бъде реализиран както е показано на следващата фигура.



Фиг. 78 Реализация на мултиплексор с два входа

Следващата фигура показва символа за обозначаването на мултиплексор и таблицата му на истинност. Селекторните входове $S_0 \div S_m$ определят кой от входовете $I_0 \div I_n$, ще се пода-

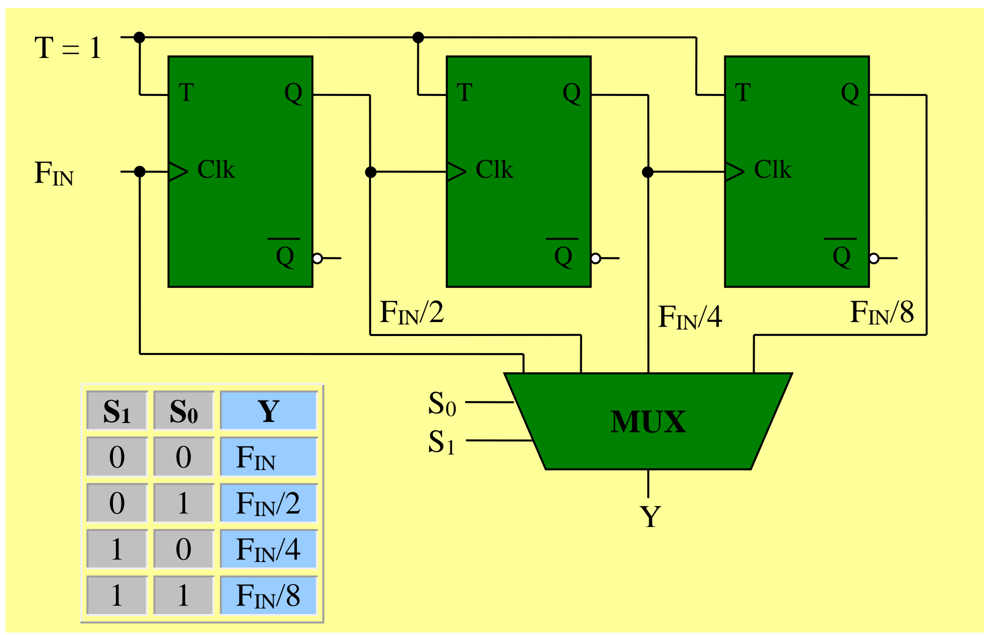


Селекторни входове				Изход
S_m	…	S_1	S_0	Y
0	0	0	0	I_0
0	0	0	1	I_1
0	0	1	0	I_2
0	0	1	1	I_3
…	…	…	…	…
1	1	1	1	I_n

Фиг. 79 Мултиплексор

де на изхода Y.

Следващата фигура описва примерна употреба на мултиплексор.

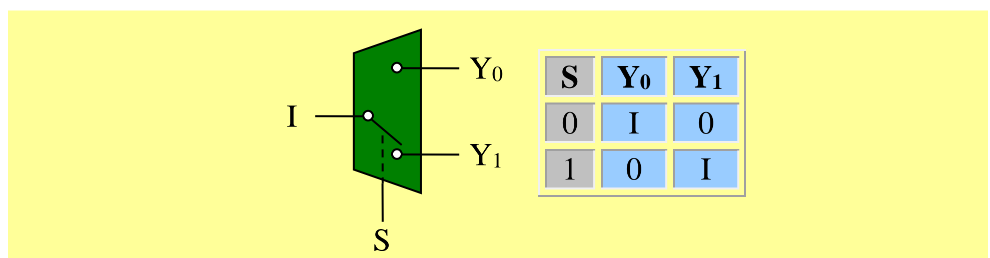


S_1	S_0	Y
0	0	F_{IN}
0	1	$F_{IN}/2$
1	0	$F_{IN}/4$
1	1	$F_{IN}/8$

Фиг. 80 Използване на мултиплексор

Фиг.80 представлява делител на честота, чийто изходи са свързани към мултиплексор. С негова помощ се избира кой изход на делителя да се подаде на изхода Y.

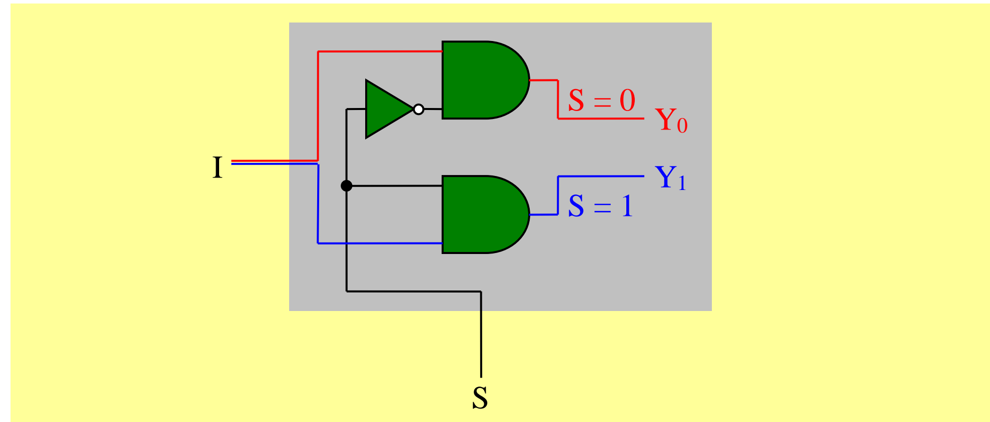
Демултиплексорът е устройство с един вход за данни и няколко изхода. С помощта на селекторни входове се избира на кой от изходите да се подаде входа за данни, т.е. . демултиплексорът е многоизходен цифров ключ. Фиг.81 показва абстрактен изглед на демултиплексор с два изхода. В зависимост от селекторния вход S входът I е свързан с изход Y_0 (S = 0) или Y_1 (S = 1).



S	Y_0	Y_1
0	I	0
1	0	I

Фиг. 81 Демултиплексор с два изхода

Демултиплексор с два изхода може да бъде реализиран както е показано на следващата фигура.



Фиг. 82 Реализация на демултиплексор с два изхода

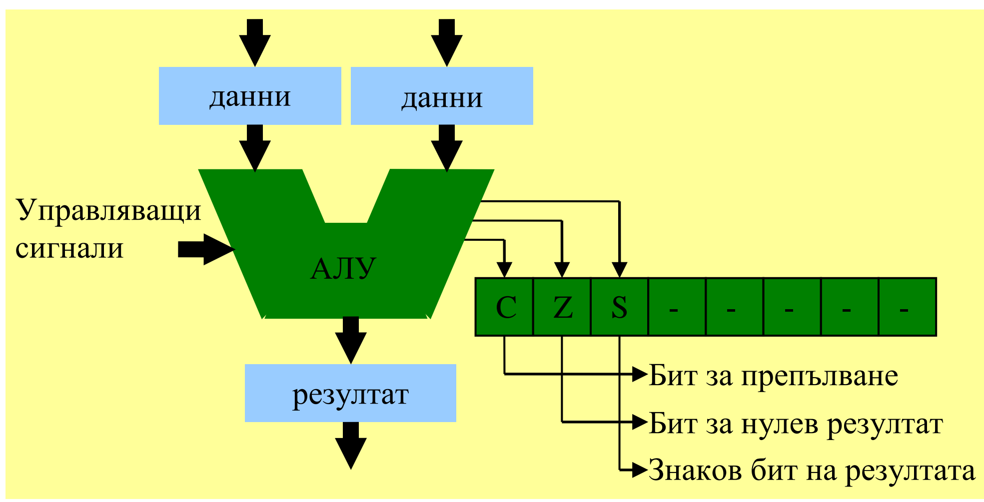
Следващата фигура показва символа за обозначаването на демултиплексор и таблицата му на истинност. Селекторните входове $S_0 \div S_m$ определят на кой от изходите $Y_0 \div Y_n$ ще се подаде входа I.

Селекторни входове				Изходи					
S_m	…	S_1	S_0	Y_n	…	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	**I**
0	0	0	1	0	0	0	0	**I**	0
0	0	1	0	0	0	0	**I**	0	0
0	0	1	1	0	0	**I**	0	0	0
…	…	…	…	…	…	…	…		
1		1	1	**I**	0	0	0	0	0

Фиг. 83 Демултиплексор

13 Аритметично & Логическо устройство

АЛУ (**ALU - Aritmetic & Logic Unit**) е основната цифрова схема в един микроконтролер. Както подсказва името, АЛУ извършва аритметични и логически операции (събиране, изваждане, умножение, преместване наляво или надясно, логически операци И, ИЛИ и т.н.). Фиг.84 показва обобщената блокова схема на АЛУ.



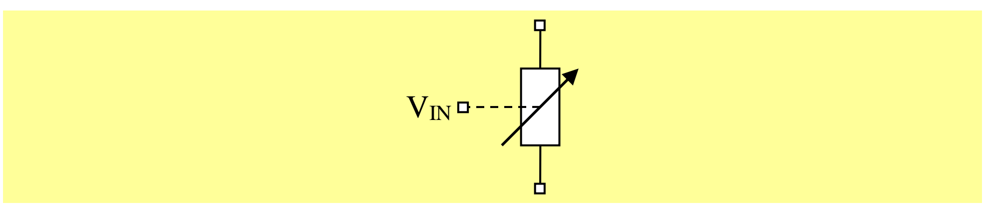
Фиг. 84 АЛУ

С помощта на логическите елементи и схеми, разгледани в предходните глави, се създават суматори (извършват събиране), субтрактори (извършват изваждане), умножители (извършват умножение) и други устройства, които изграждат функционалността на АЛУ. Операцията, която АЛУ трябва да извърши върху данните, се определя от подаваните към него управляващи сигнали. При извършване на операциите е възможно да възникне препълване, нулев резултат или да се получи отрицателен резултат. АЛУ индицира това като установява в 1 съответните битове на специален регистър. Потребителят може да провери стойностите на тези битове, за да определи дали е възникнала някоя от горните ситуации, и да предприеме съответните действия.

14 CMOS технология

14.1 Общи сведения

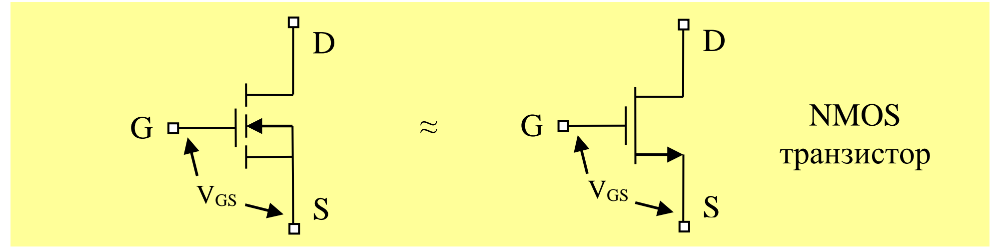
Съвременните цифрови устройства, в това число микроконтролерите, се произвеждат по т.нар. CMOS (**Complementary Metal-Oxid-Semiconductor**) технология. Градивните елементи, използвани в тази технология, са MOS транзистори. MOS транзисторът може да се разглежда като 3-изводно устройство, което работи като управлявано с напрежение съпротивление. Напрежението, приложено към единия извод, управлява съпротивлението между другите два (Фиг.85).



Фиг. 85 MOS транзистор като управлявано с напрежение съпротивление

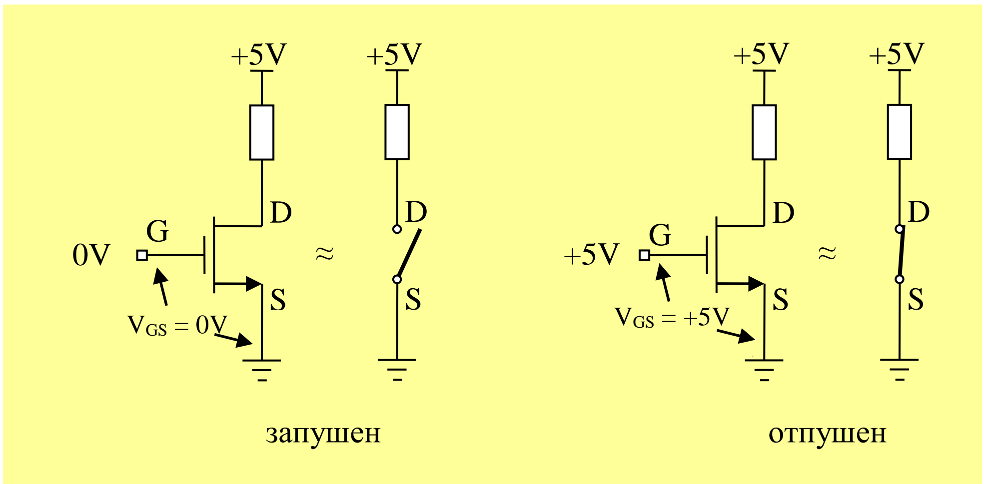
В цифровите схеми, където се използват напрежения с две нива, ниско (0V) и високо (например 5V), MOS транзисторите работят в ключов режим. В ключов режим MOS транзисторът е или включен (съпротивлението е много ниско, клонящо към 0Ω), или изключен (съпротивлението е много високо, от порядъка на 1MΩ или повече).

Има два типа MOS транзистори: n-канален и p-канален. Символите за обозначаване на n-канален MOS транзистор са показани на Фиг.86. Символът вляво обикновено се използва в електронни схеми с дискретни транзистори, докато този вдясно обикновено се използва в електронни схеми в интегрално изпълнение. Изводите се обозначават като гейт (**G - Gate**), дрейн (**D - Drain**) и сорс (**S - Source**). Напрежението, подадено между гейта и сорса V_{GS}, нормално е нула или положително. Ако $V_{GS} = 0V$, съпротивлението



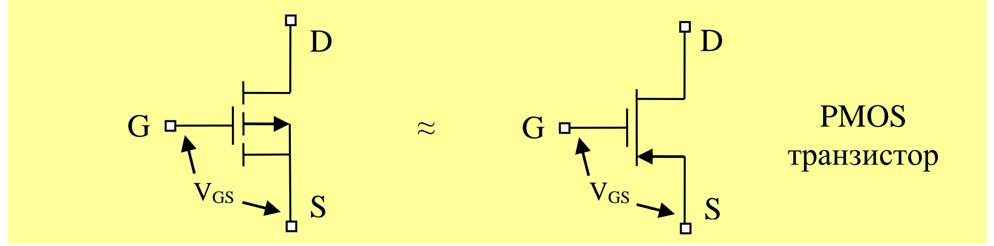
Фиг. 86 n-канален MOS транзистор (NMOS транзистор)

между дрейна и сорса R_{DS} е много голямо, т.е. . транзисторът е изключен (нарича се още запушен). Ако почнем да увеличаваме V_{GS}, R_{DS} започва да намалява. При определена стойност на V_{GS} съпротивлението R_{DS} става много ниско, от порядъка на няколко ома, т.е. . транзисторът е включен (нарича се още отпушен). Следващата фигура показва описаните принципи на работа на NMOS транзистор.



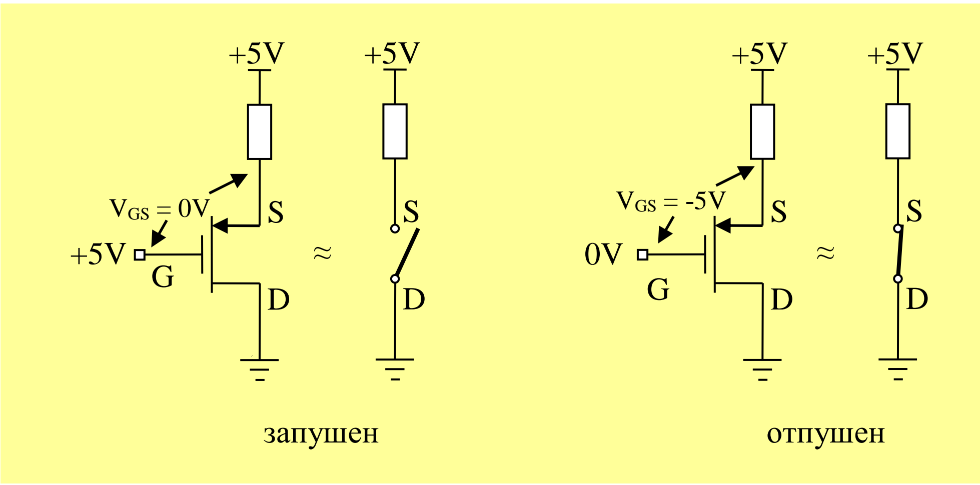
Фиг. 87 Принцип на работа на NMOS транзистор в ключов режим

Символите за обозначаване на p-канален MOS транзистор са показани на на Фиг.88. PMOS транзисторът работи аналогично на NMOS с тази разлика, че напрежението V_{GS} е нула или отрицателно. Ако V_{GS} = 0V, съпротивлението между дрейна и сорса R_{DS} е много голямо, т.е. . транзисторът е изключен. Ако почнем да намаляваме V_{GS}, R_{DS} започва да намалява. При определена стойност на V_{GS} съпротивлението



Фиг. 88 p-канален MOS транзистор (PMOS транзистор)

R_{DS} става много ниско, от порядъка на няколко ома, т.е. . транзисторът е включен. Следващата фигура показва описаните принципи на работа на PMOS транзистор.

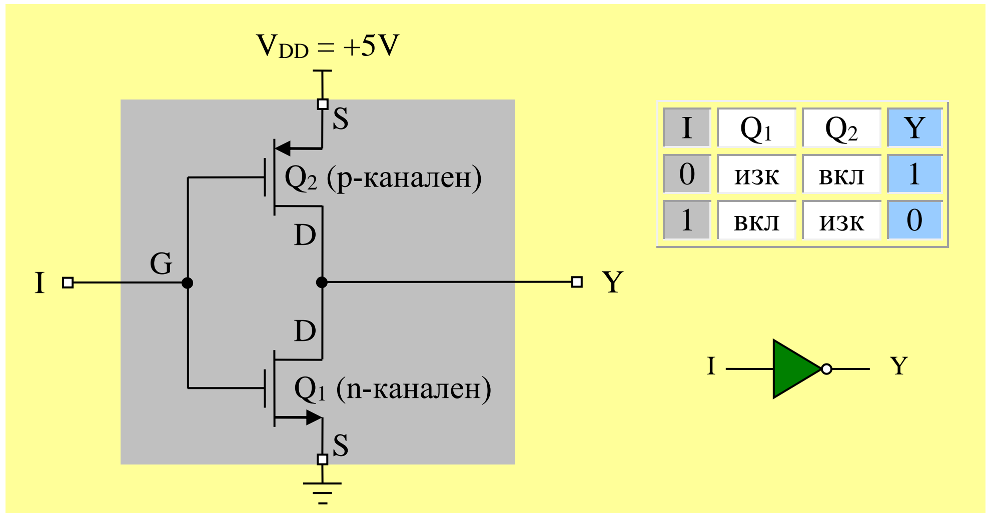


Фиг. 89 Принцип на работа на PMOS транзистор в ключов режим

14.2 CMOS инвертор

CMOS технологията използва NMOS и PMOS транзисторите в комбинация, за да формира CMOS логика. Най-простата CMOS логическа схема е инверторът. Следващата фигура показва реализирането на CMOS инвертор.

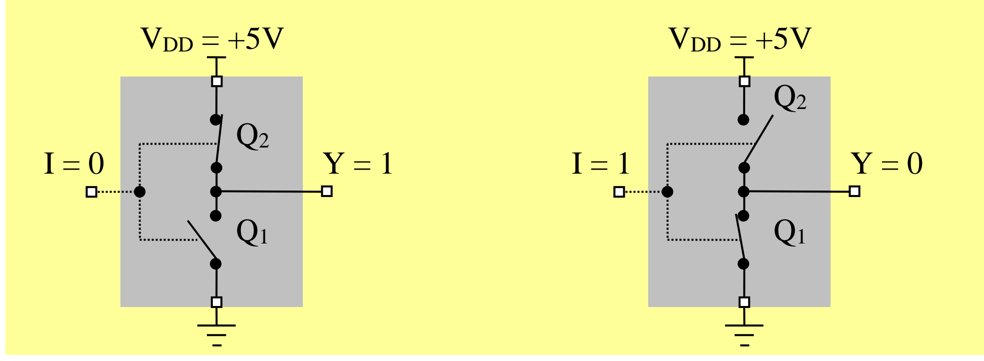
Когато I = 0 (0V), Q_1 изключен (запушен), а Q_2 е включен (отпушен). При тези условия съпротивлението на Q_1 е много



Фиг. 90 CMOS инвертор – основна схема

малко, а на Q_2 много голямо и Y = 1 (+5V). Когато I = 1 (+5V), Q_1 включен (отпушен), а Q_2 е изключен (запушен). При тези условия съпротивлението на Q_1 е много голямо, а на Q_2 много малко и Y = 0 (0V).

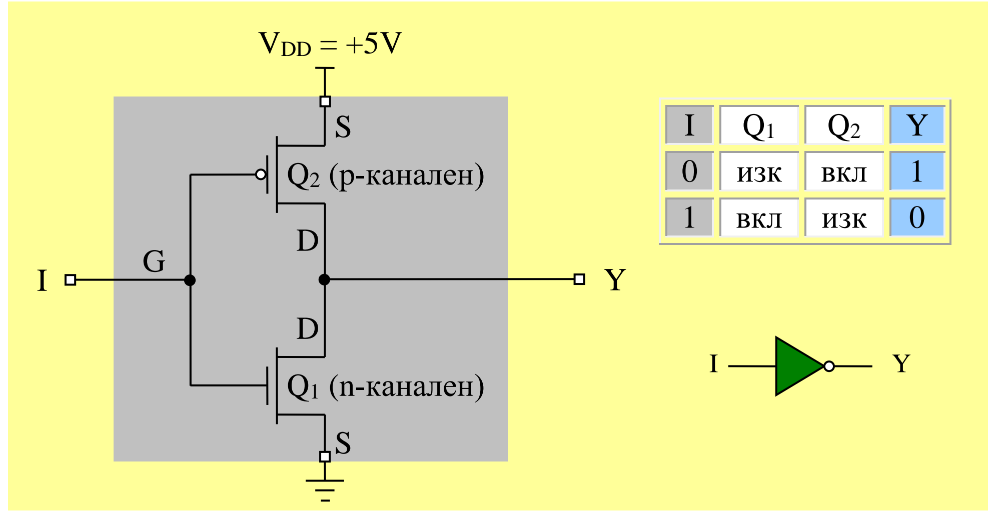
Тъй като в цифровите схеми MOS транзисторите работят в ключов режим, те могат да се представят като ключове (Фиг.91).



Фиг. 91 CMOS инвертор – ключова схема

Ключовият модел, показан на Фиг.91, позволява CMOS схемите да се чертаят по начин, който прави тяхното логическо поведение по-лесно за разбиране. Фиг.92 показва

други символи за обозначаване на PMOS и NMOS транзистори в цифровите схеми, които отразяват тяхното логическо поведение.



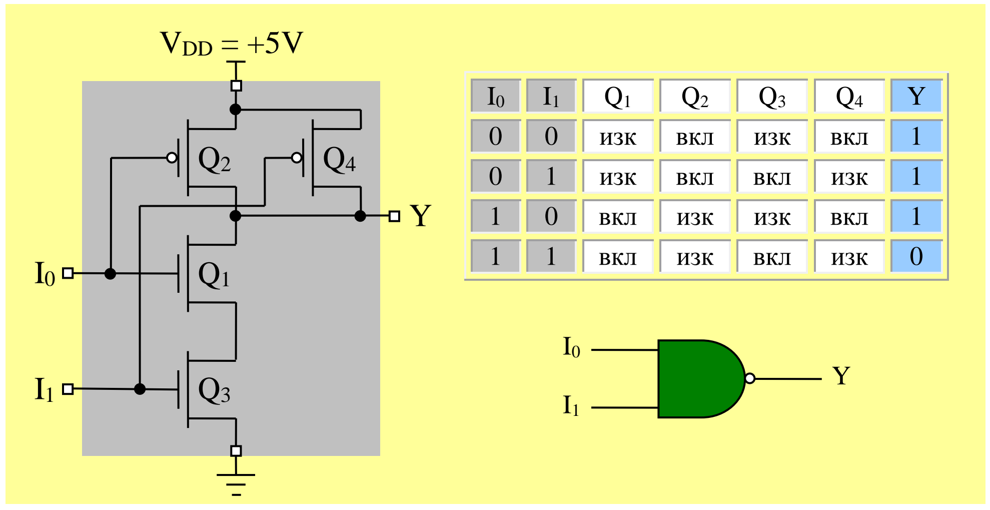
I	Q1	Q2	Y
0	изк	вкл	1
1	вкл	изк	0

Фиг. 92 CMOS инвертор – логическа схема

Кръгчето на гейта на Q_2 подсказва, че транзисторът е отпушен когато $I = 0$, а липсата на кръгче пред Q_1 подсказва, че транзисторът е отпушен когато $I = 1$.

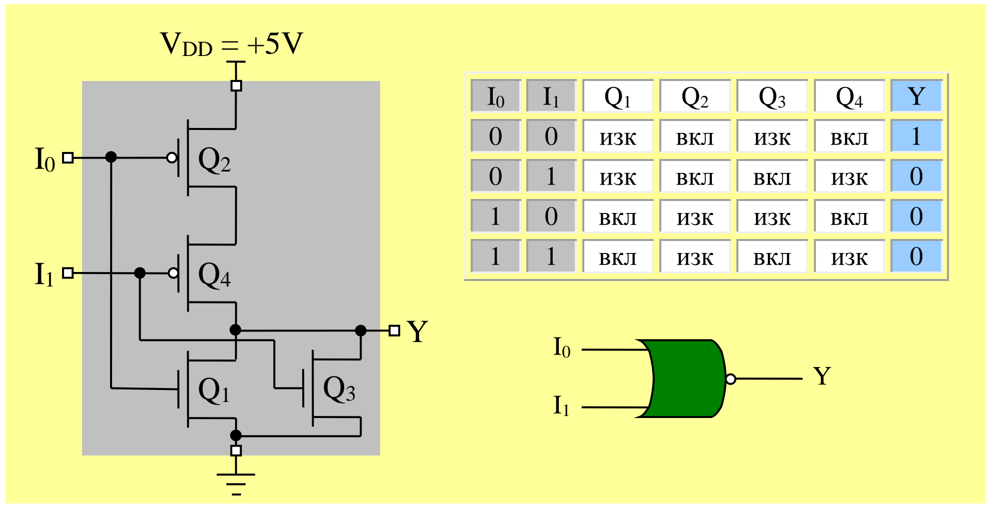
14.3 CMOS NAND и NOR

Фиг.93 показва вътрешната логическа схема на двувходов CMOS NAND логически елемент, а Фиг.94 показва вътрешната логическа схема на двувходов CMOS NOR логически елемент. Ако в изхода на показаните схеми добавите CMOS инвертора от Фиг.92, ще получите вътрешните логически схеми на CMOS AND и CMOS OR логическите елементи съответно.



I_0	I_1	Q_1	Q_2	Q_3	Q_4	Y
0	0	изк	вкл	изк	вкл	1
0	1	изк	вкл	вкл	изк	1
1	0	вкл	изк	изк	вкл	1
1	1	вкл	изк	вкл	изк	0

Фиг. 93 CMOS NAND – логическа схема

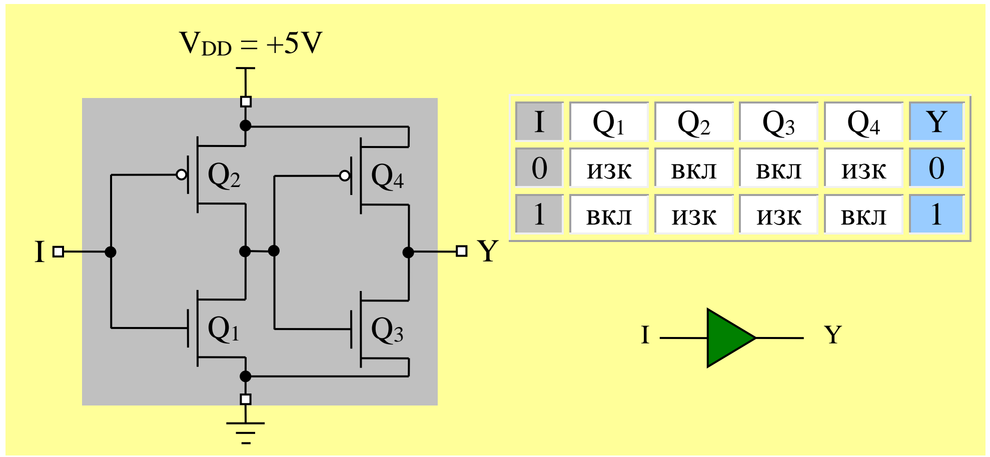


I_0	I_1	Q_1	Q_2	Q_3	Q_4	Y
0	0	изк	вкл	изк	вкл	1
0	1	изк	вкл	вкл	изк	0
1	0	вкл	изк	изк	вкл	0
1	1	вкл	изк	вкл	изк	0

Фиг. 94 CMOS NOR – логическа схема

14.4 CMOS буфер

Фиг.95 показва вътрешната логическа схема на CMOS буфер. Ако се вгледате внимателно, ще видите, че тя се състои от два инвертора, свързани последователно.

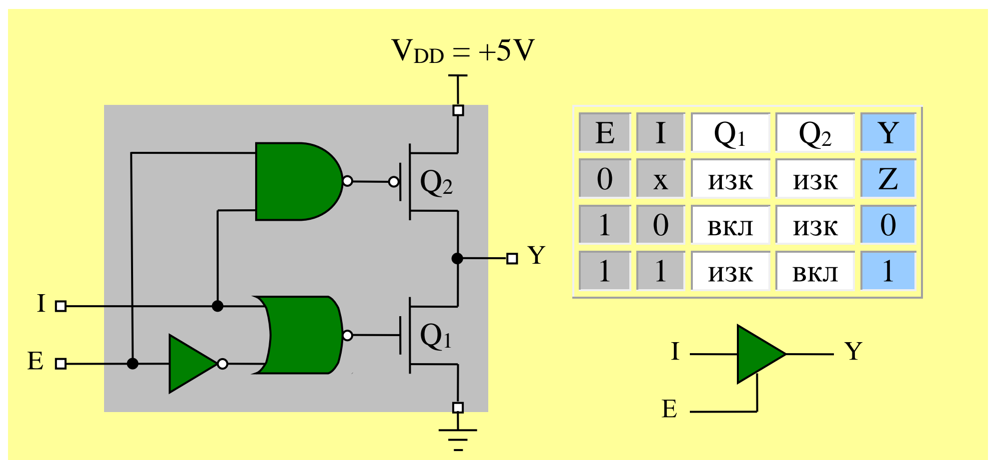


I	Q1	Q2	Q3	Q4	Y
0	изк	вкл	вкл	изк	0
1	вкл	изк	изк	вкл	1

Фиг. 95 CMOS буфер – логическа схема

14.5 CMOS буфер с три състояния

Фиг.96 показва опростена вътрешна логическа схема на CMOS буфер с три състояния и активно високо ниво на разрешаващия сигнал.



E	I	Q1	Q2	Y
0	x	изк	изк	Z
1	0	вкл	изк	0
1	1	изк	вкл	1

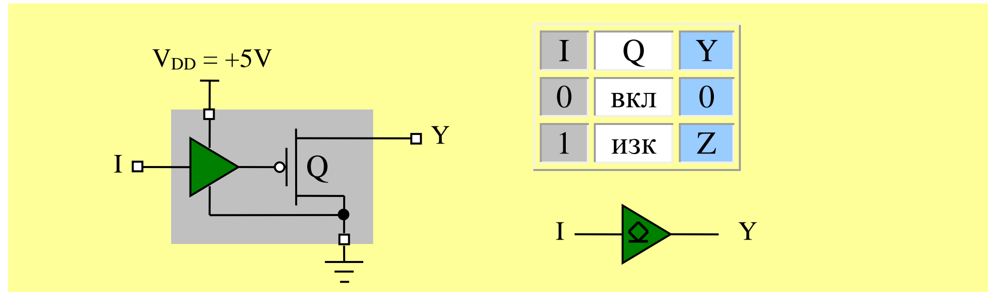
Фиг. 96 CMOS буфер с три състояния

Когато E = 0 и двата транзистора са изключени независимо от стойността на входа I. В това състояние изходът Y се държи все едно не е свързан към схемата, т.е. . все едно

„плава” свободно. Лесно може да си го представите, ако замените транзисторите с два отворени ключа. Както вече знаете, това състояние се нарича високоимпедансно състояние.

14.6 Изходи с отворен дрейн

Когато дрейнът на един MOS транзистор не е свързан към захранване и директно е изведен като изход, се казва че изходът е с отворен дрейн. Фиг.97 показва опростена вътрешна схема на буфер с отворен дрейн.

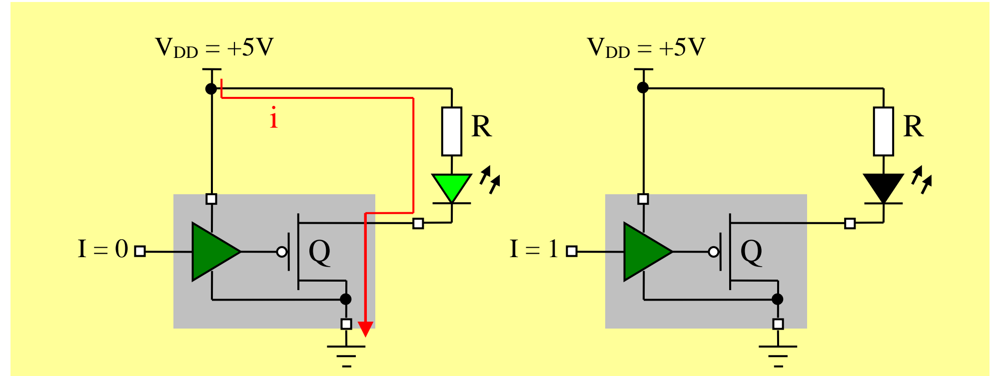


I	Q	Y
0	вкл	0
1	изк	Z

Фиг. 97 Изход с отворен дрейн

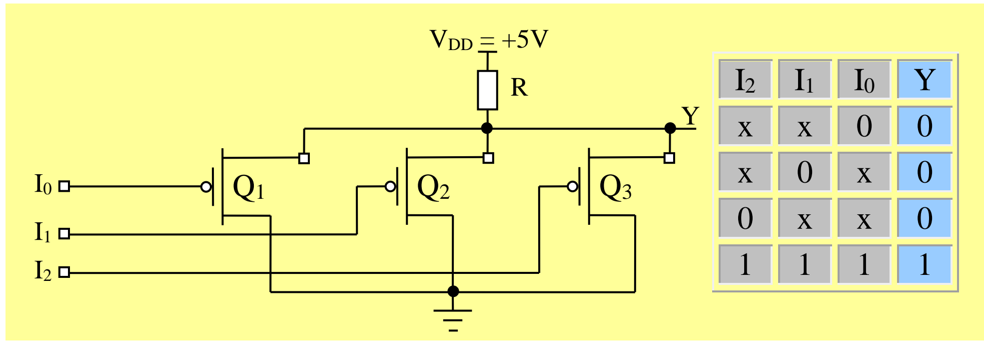
В изхода на буфера е включен p-канален MOS транзистор. Когато I = 0, транзисторът е включен и изходът Y е свързан към маса, т.е. . Y = 0. Когато I = 1, транзисторът е изключен и изходът Y не е свързан към нищо, т.е. . Y се намира във високоимпедансно състояние Z. Обърнете внимание на символа ⌂, който типично се използва за обозначаване на изходи с отворен дрейн.

От приницпа на работа става ясно, че изходите с отворен дрейн могат само да поемат ток. Фиг.98 показва примерно приложение. Когато I = 0, транзисторът е включен, през светодиода протича ток и той светва. Когато I = 1, транзисторът е изключен и светодиодът не свети.



Фиг. 98 Приложение на изход с отворен дрейн

Друго типично приложение е създаването на жична-И (**wired-AND**) конфигурация (Фиг.99).



I_2	I_1	I_0	Y
x	x	0	0
x	0	x	0
0	x	x	0
1	1	1	1

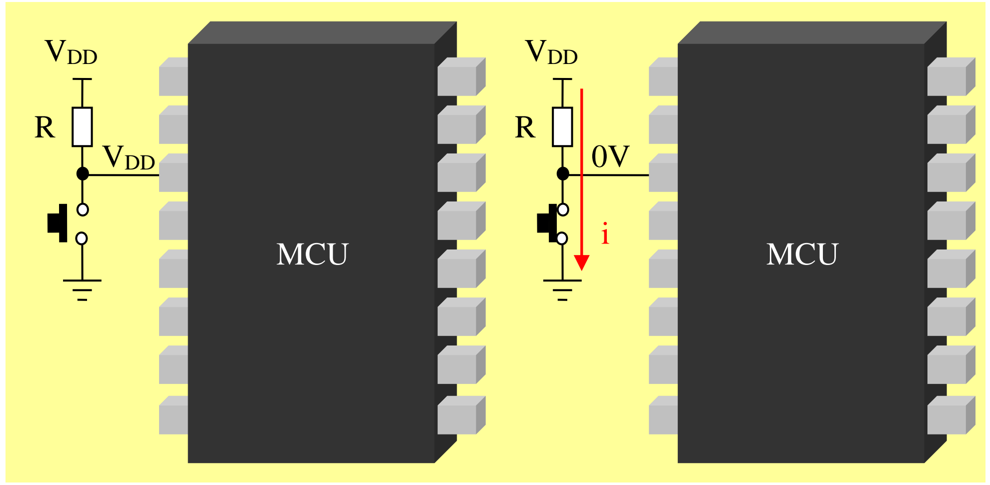
Фиг. 99 Жична-И конфигурация

Линията Y е 1 само когато всички транзистори са изключени. Линията Y е 0, когато поне един от транзисторите е включен.

14.7 Изтеглящи резистори

Изтеглящият (**pull up**) резистор е резистор, използван в цифровите устройства, за да осигури ясно определено логическо ниво на един извод (или на даден проводник) при всякакви условия. Единият край на резистора е свързан към

захранване, а другият към извода. Ние вече използвахме изтеглящи резистори в предходните подточки (вижте отново Фиг.48 и 99). Следващата фигура показва друга типична употреба на изтеглящ резистор.



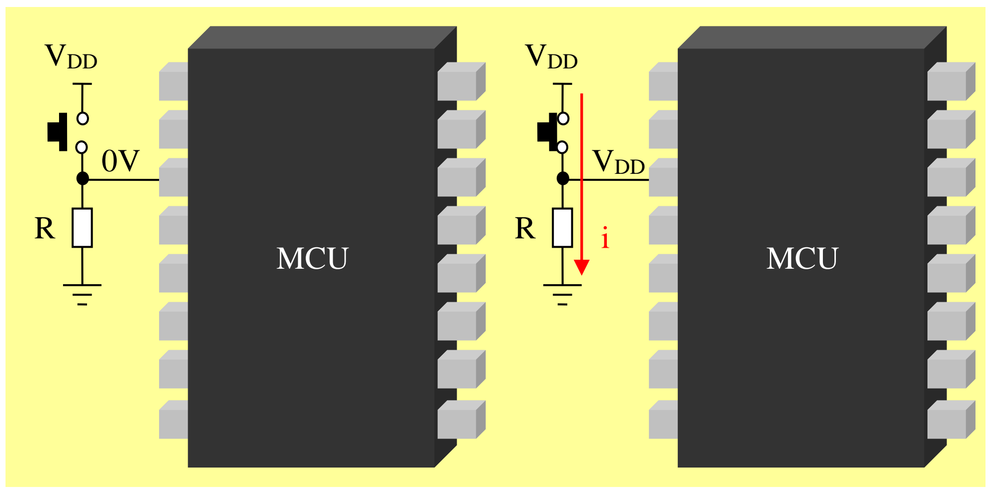
Фиг. 100 Пример за използване на изтеглящ резистор

Резисторът е свързан към входен извод на микроконтролер. Когато бутонът е отпуснат, напрежението на извода се повдига (изтегля) до нивото на захранващото напрежение V_{DD}, заради изтеглящия резистор R. При натискане на бутона изводът се свързва към маса, т.е. . напрежението пада до 0V. Ако махнете резистора R и бутонът е отпуснат, изводът ще бъде в плаващо състояние, т.е. . нито лог.0, нито лог.1. Интерпретирането на стойността на плаващ вход от микроконтролера е непредсказуемо.

Ниска стойност на R се нарича силен изтеглящ резистор (**strong pull up**), заради по-големия ток, който протича през него. Висока стойност на R се нарича слаб изтеглящ резистор (**weak pull up**), заради по-малкия ток, който протича през него.

14.8 Понижаващи резистори

Понижаващият (**pull down**) резистор е подобен на изтеглящия резистор с тази разлика, че единият му край е свързан към маса, а другият към извода. Следващата фигура показва типична употреба на понижаващ резистор.



Фиг. 101 Пример за използване на понижаващ резистор

Когато бутонът е отпуснат, напрежението на извода се понижава до 0V, заради понижаващия резистор R. При натискане на бутона изводът се свързва към захранването, т.е. . напрежението се вдига до V_{DD}.

14.9 Захранващо напрежение

CMOS устройствата използват различни захранващи напрежения: +5V, +3.3V, +2.5V и +1.2V. Типично микроконтролерите използват първите две захранващи напрежения (+5V и +3.3V).

14.10 CMOS и TTL входни и изходни нива

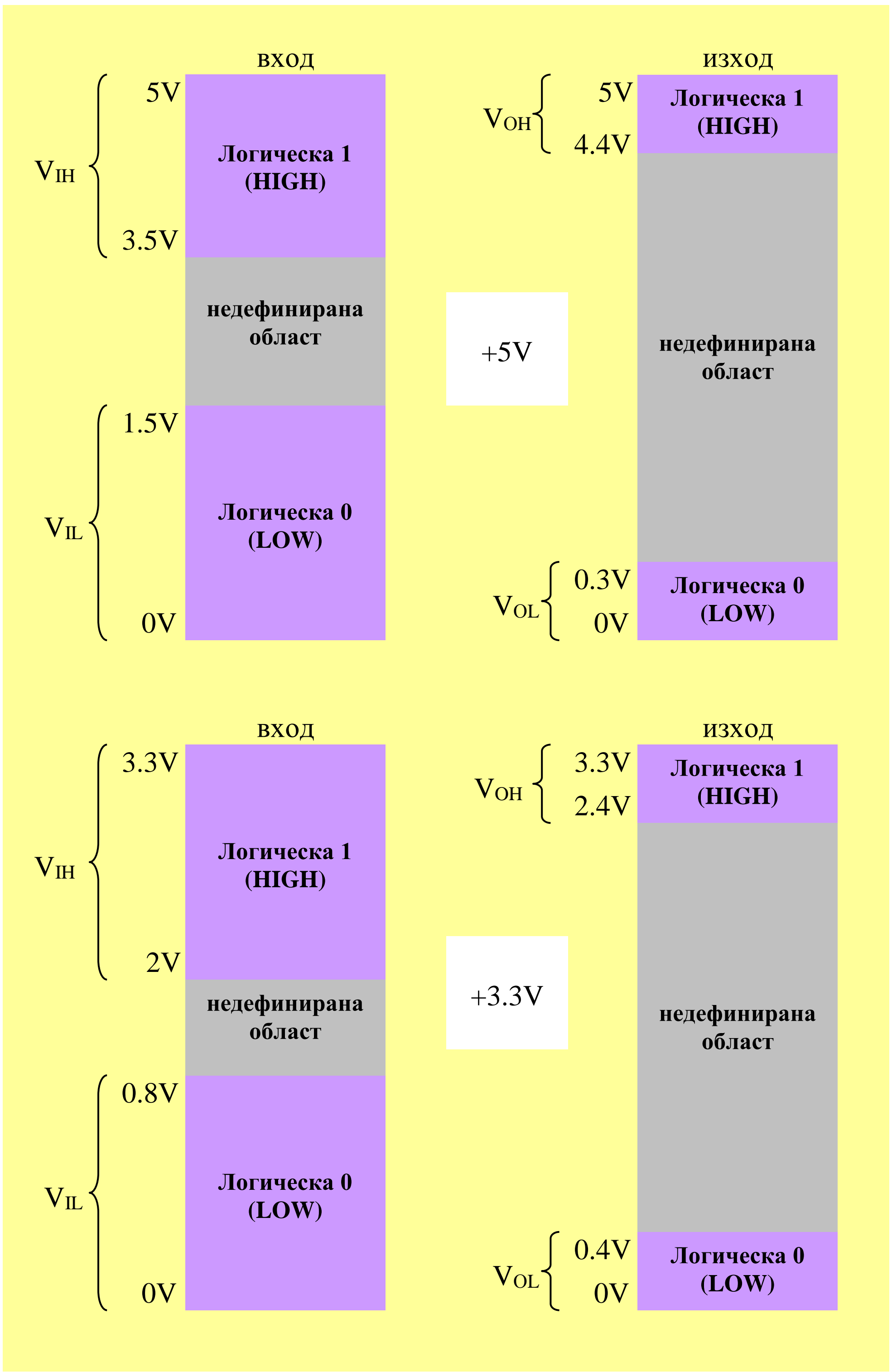
Досега за входни и изходни нива на входовете и изходите на цифровите устройства използвахме абстрактните понятия логическа 0 (или ниско ниво) и логическа 1 (или високо ниво). Зад тези понятия обаче се крият реални стойности на напрежения, които зависят от технологията, по която са изградени цифровите устройства. Входните и изходните нива на тези напрежения се обозначават по следния начин:

- V_{IL} – входна логическа нула;
- V_{IH} – входна логическа единица;
- V_{OL} – изходна логическа нула;
- V_{OH} – изходна логическа единица.

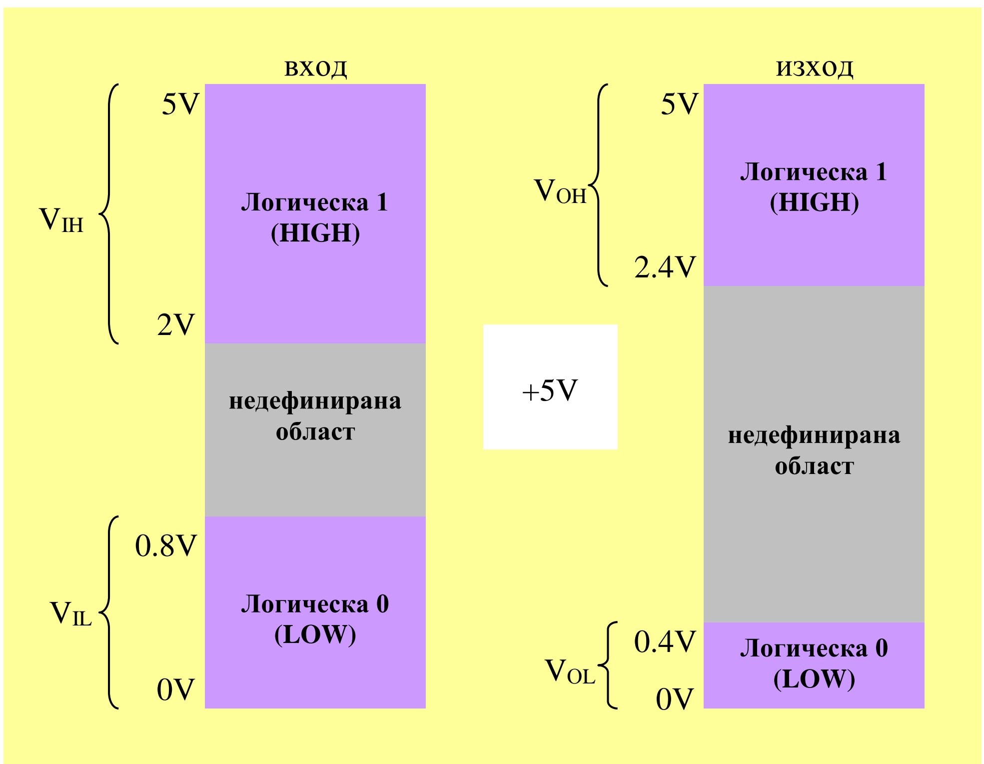
Фиг.102 показва нагледно стойностите на напреженията, които стоят зад тези обозначения, за CMOS устройства, използващи захранващи напрежения +5V и +3.3V.

TTL (**Transistor-Transistor Logic**) е по-стара технология, използвана за производство на цифрови интегрални схеми. Градивните елементи, използвани в тази технология, са биполярни транзистори (**BJT – Bipolar Junction Transistor**). Има два типа биполярни транзистори: NPN и PNP. Няма да навлизам в детайли, тъй като не е и необходимо. Причината да спомена тази технология е, че често ще срещата понятията „TTL логически нива". TTL цифровите интегрални схеми използват захранващо напрежение +5V. Фиг.103 показва входните и изходните TTL логически нива.

Някои цифрови интегрални схеми са прозиведени по CMOS технология, но входните и/или изходните нива на изводите използват TTL съвместими нива.



Фиг. 102 CMOS входни и изходни нива

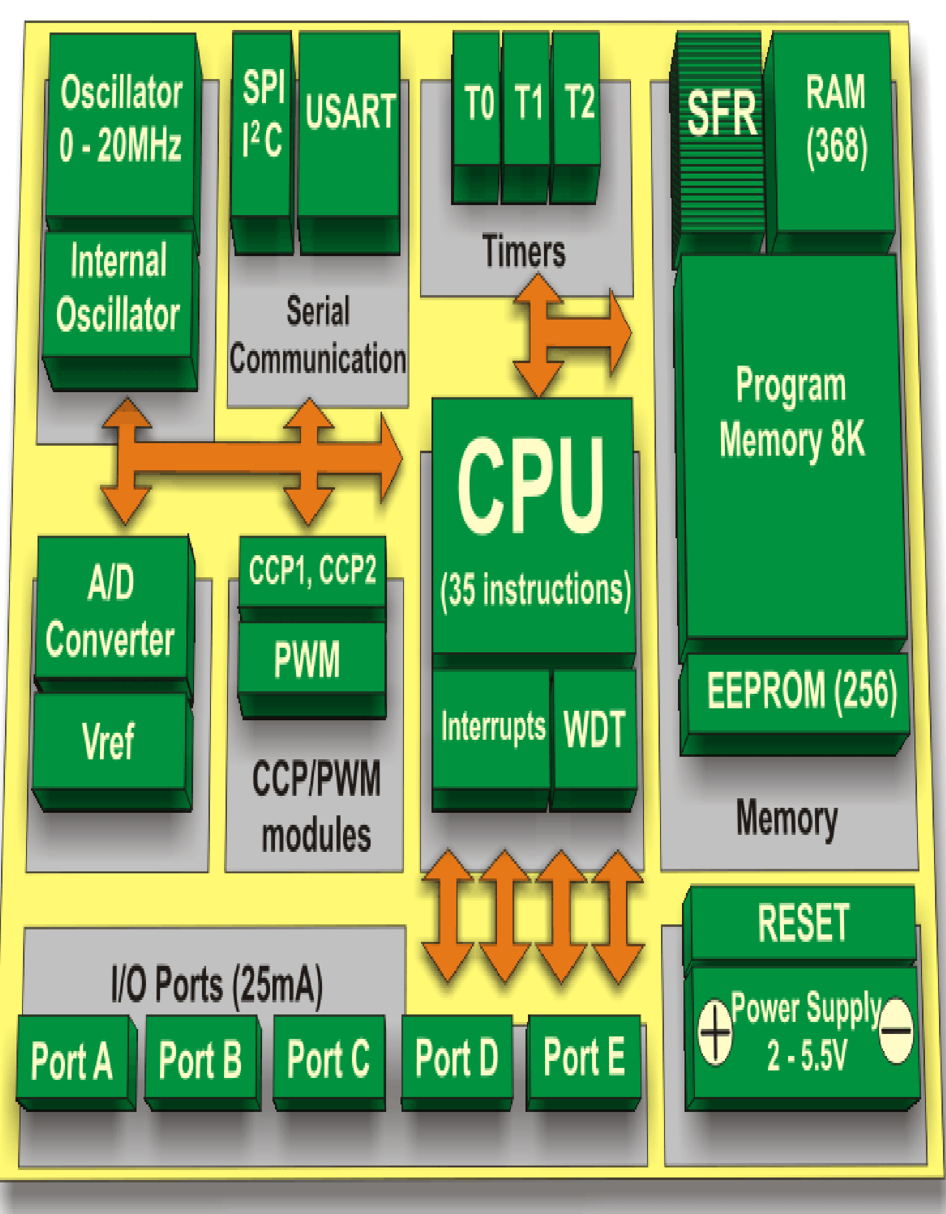


Фиг. 103 TTL входни и изходни нива

Преди да свържете цифрова интегрална схема към микроконтролер уверете се, че входно/изходните им нива са съвместими.

Част III

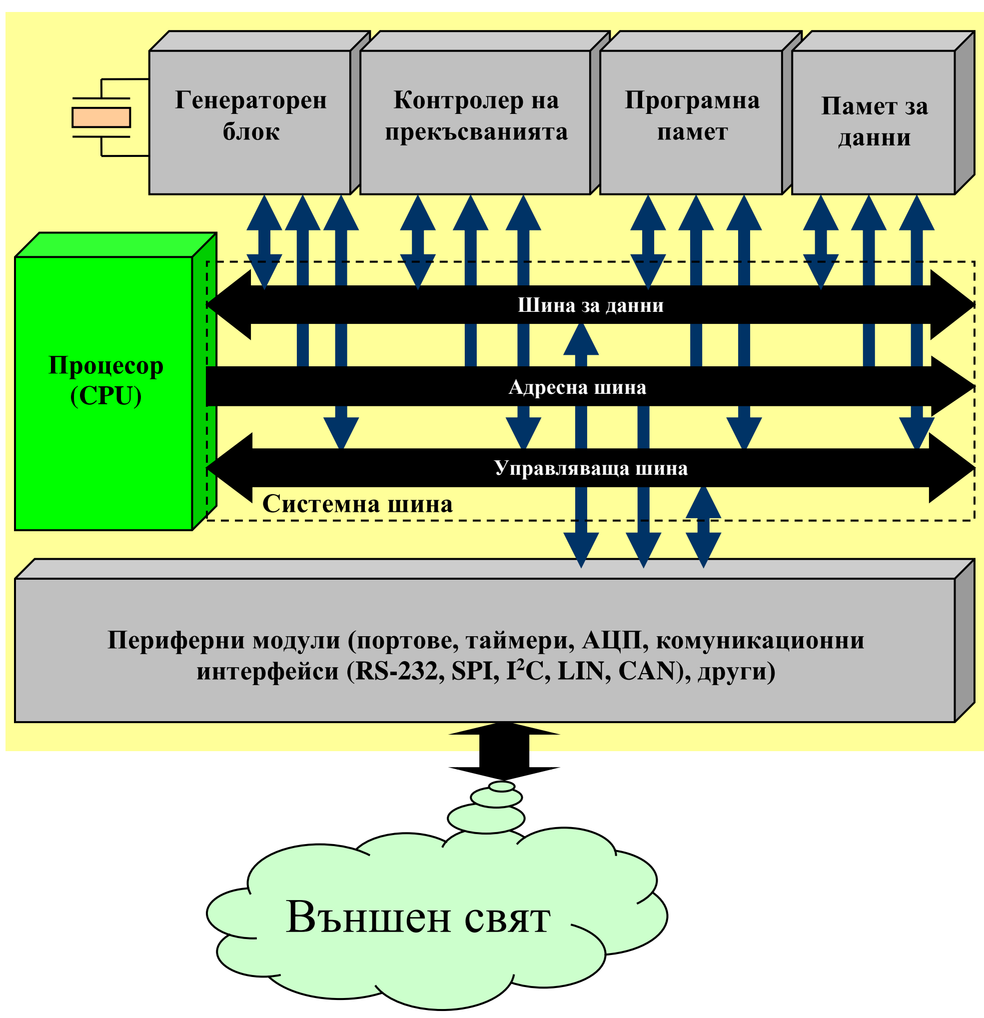
Архитектура на микроконтролерите



15 Общи сведения

15.1 Обобщена архитектура

Най-общо архитектурата на микроконтролерите може да се сведе до показаната на Фиг.104.



Фиг. 104 Обобщена архитектура на микроконтролер

- Процесор

Основния компонент на един микроконтролер е

процесорът (**CPU – Central Processor Unit**). Това е „мозъкът” на всяка компютърна система. Процесорът изпълнява инструкциите на компютърната програма. В контекста на микроконтролерите програмата се нарича фърмуеър (**firmware**).

- Програмна памет

Програмната памет служи за съхранение на инструкциите на програмата, но също може да съхранява и константни данни.

- Памет за данни

Паметта за данни се използва за съхранение на променливи. Програмите използват променливи да съхраняват стойности в тях, които могат да се променят в процеса на изпълнение на програмата, както и да съхраняват резултати от аритметични и логически операции.

- Контролер на прекъсванията

Контролерът на прекъсванията приема сигналите за прекъсване, получени от периферните модули, и известява процесора. Прекъсванията са средство, чрез което периферните модули уведомяват процесора, че е възникнало някакво събитие например нивото на извод се е променило от високо в ниско (или обратно), че се е препълнил таймер или Аналого-цифров Преобразувател (АЦП) е завършил преобразуване на аналогов сигнал и т.н. Всяко прекъсване има своя подпрограма, която отговаря за неговата обработка (**ISR – Interrupt Service Routine**). При получаване на сигнал за прекъсване, процесорът спира изпълнението на текущата програма и стартира ISR-подпрограмата, отговаряща за прекъсването. След като ISR-

подпрограмата завърши изпълнението на задачата, за която отговаря, процесорът отново се връща към изпълнението на предишната си програма.

- Генераторен блок

Генераторният блок служи да генерира всички тактови сигнали, необходими за работата на микроконтролера.

- Периферни модули

Периферните модули, осъществяват връзката на микроконтролера с външния свят. Микроконтролерите имат най-различни периферни модули:

  - Портове
  - Таймери
  - АЦП
  - Серийни комуникационни модули
  - Други

Всички компоненти на микроконтролера са свързани към процесора посредством системна шина. Шината е съвкупност от проводници, по които се предава най-различна информация. Системната шина се състои от три подшини:

- Адресна шина

Информацията, предавана по тази шина, се явява адрес. Този адрес определя източника или приемника на данните, предавани по шината за данни. Всички устройства, с

които процесорът може да обменя информация, си имат адрес.

- Шина за данни

Информацията, предавана по тази шина, се явява полезните данни, които микроконтролерът ще обработва по някакъв начин. Типично шината за данни е широка 8, 16 или 32 бита и определя разредеността на процесора, съответно на микроконтролера (дали е 8-, 16-, или 32-битов). Количеството данни, които един процесор може да обработва наведнъж, се нарича още машинна дума или само дума.

- Управляваща шина

Информацията, предавана по тази шина, има управляващ характер. Например дали процесорът ще чете данни от дадено устройство, или ще изпраща данни към него. Тактовите сигнали, получени от генераторния блок и сигналите за прекъсване, получени от периферните модули, също са част от управляващата шина.

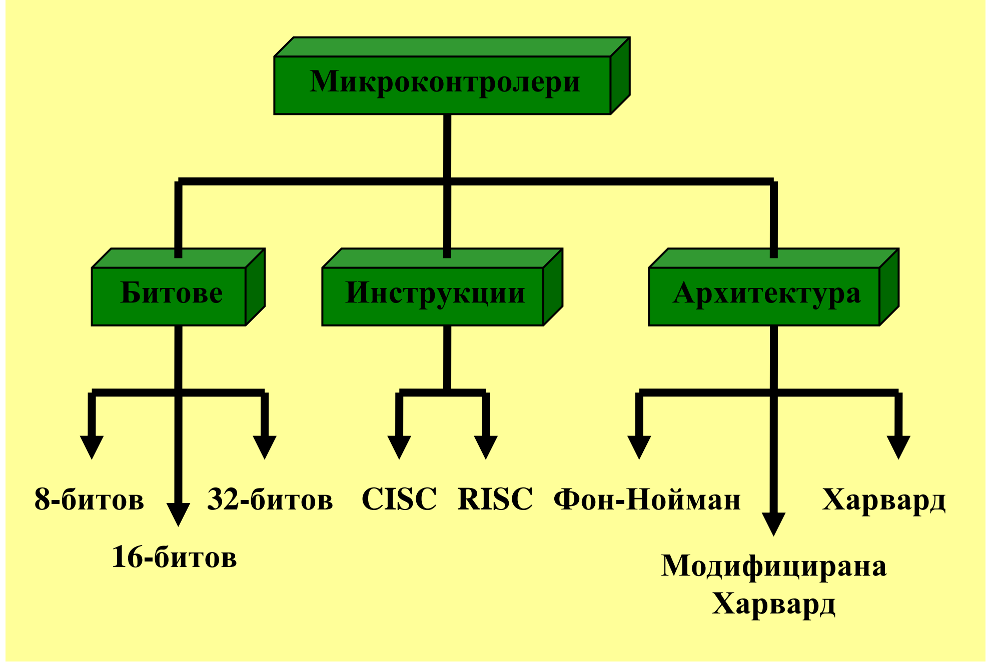
15.2 Типове микроконтролери

Микроконтролерите могат да се класифицират по следните признаци:

- Широчина на шината за данни в битове;
- Тип на архитектурата;
- Набор от инструкции.

Следващата фигура показва в резюме класификацията на

микроконтролерите.



Фиг. 105 Класификация на микроконтролерите

15.2.1 8-, 16- и 32-битови микроконтролери

8-БИТОВИ МИКРОКОНТРОЛЕРИ

Когато процесорът извършва аритметични и логически операции върху 8-битови данни, микроконтролерът е 8-битов. Широчината на вътрешната шина, по която се пренасят данните, е 8-бита. Примери за 8-битови микроконтролери са: **Z8Encore!**, **PIC10/12/16/18**, **AVR8**, **8051-базирани** и много други.

16-БИТОВИ МИКРОКОНТРОЛЕРИ

Когато процесорът извършва аритметични и логически операции върху 16-битови данни, микроконтролерът е 16-

битов. Широчината на вътрешната шина, по която се пренасят данните, е 16-бита. Примери за 16-битови микроконтролери са: **ZNEO**, **PIC24**, **MSP430** и много други.

32-БИТОВИ МИКРОКОНТРОЛЕРИ

Когато процесорът извършва аритметични и логически операции върху 32-битови данни, микроконтролерът е 32-битов. Широчината на вътрешната шина, по която се пренасят данните, е 32-бита. Примери за 32-битови микроконтролери са: **PIC32**, **AVR32**, **ARM-базирани**, **PowerPC-базирани** и много други.

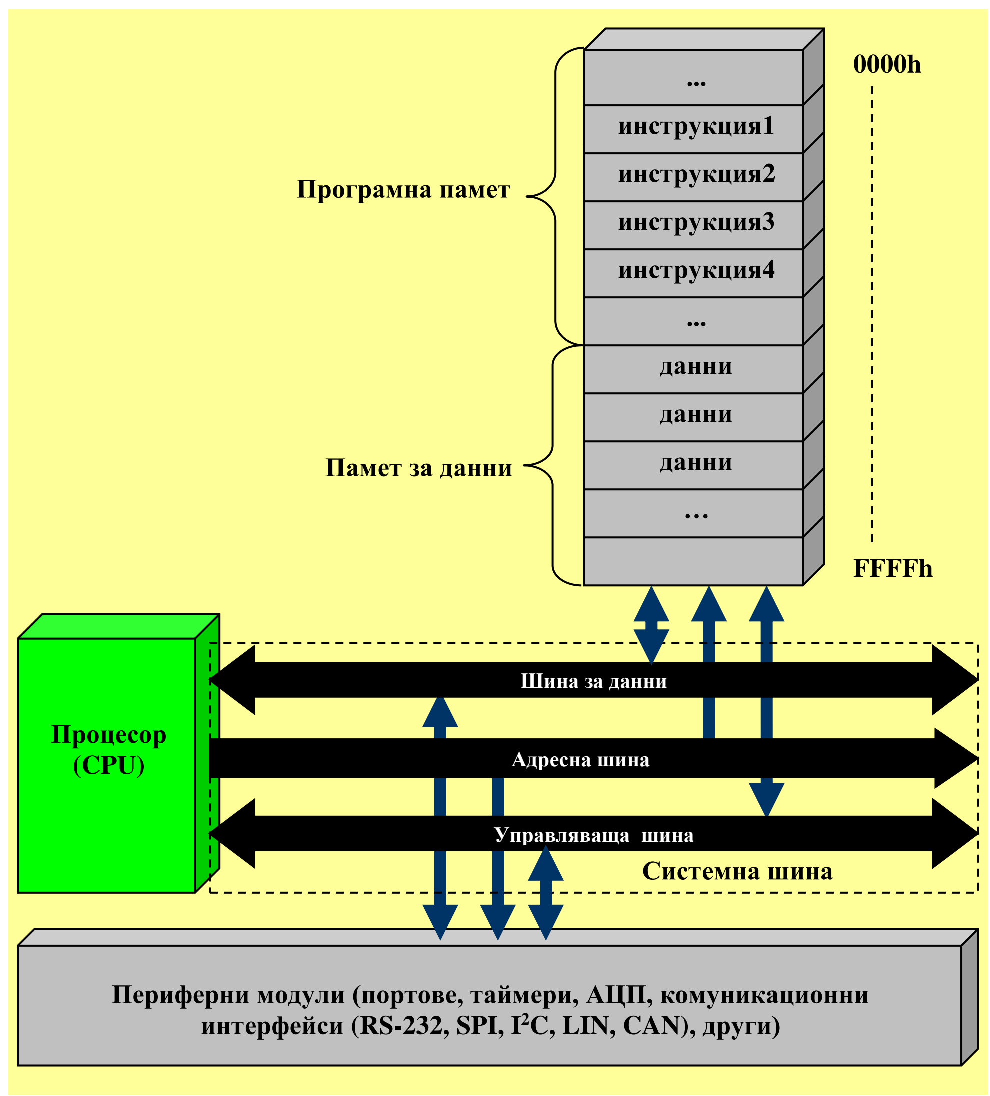
Тенденцията в бъдещото развитие на микроконтролерите е не в разширяване на шината за данни, а в увеличаване на броя на процесорните ядра в един микроконтролер.

15.2.2 Архитектура на микроконтролерите

Има три основни типа архитектури, по които се изграждат микроконтролерите.

- Фон-Нойман (**Von-Neuman**);

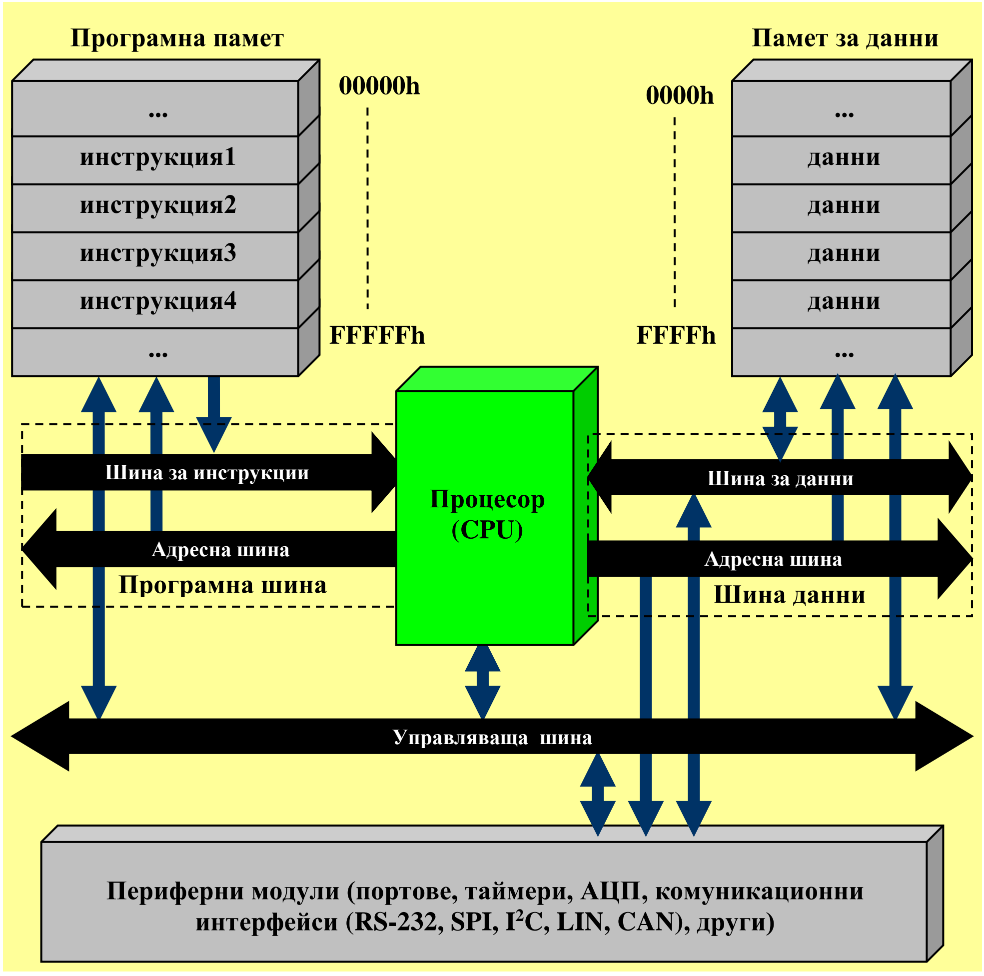
Фон Нойман архитектурата, известна още като Принстън (**Princeton**), е описана през 1945г от Джон фон Нойман (**John von Neumann**). Основната идея на тази архитектура е използването на общо адресно пространство за програмната памет и паметта за данни. т.е. . инструкциите и данните се извличат по една и съща шина (Фиг.106). Това означава, че процесорът първо извлича инструкцията, след което извлича данните, изисквани от тази инструкция. Това е и основния недостатък на тази архитектура.



Фиг. 106 Фон Нойман архитектура

- Харвард (**Harvard**);

Харвард архитектурата отстранява недостатъка на Фон Нойман архитектурата, като използва отделни адресни пространства за програмната памет и паметта за данни. На практика това означава използване на отделни шини за извличане на инструкциите и данните (Фиг.107).



Фиг. 107 Харвард архитектура

Това дава следните две предимства:

- Конвейрезиране на инструкциите (**pipelining**). Докато процесорът изпълнява поредната инструкция, се извлича следващата. Това позволява също и едновременно извличане на инструкции и данни, което е невъзможно при архитектурата фон Нойман. Това значително увеличава скоростта на изпълнение на програмата.

- Програмната памет може да бъде организирана в думи с различен размер, обикновено по-голям, от този на

паметта за данни. Например ако инструкциите са широки 16-бита, програмната памет също може да бъде организирана като 16-битови клетки. Това позволява инструкциите да се извличат наведнъж, което също води до увеличаване на бързодействието на микроконтролера.

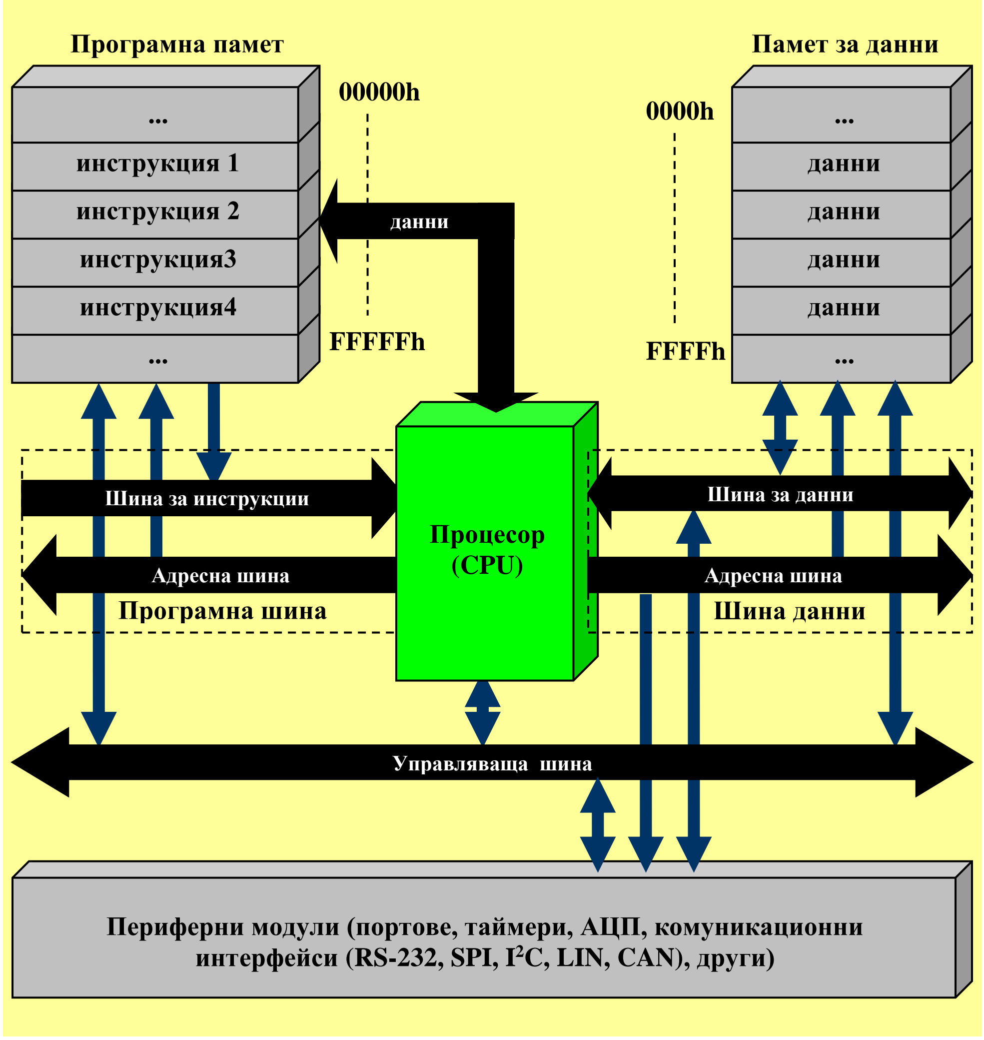
- Модифицирана Харвард (**Modified-Harvard**).

Доста рових из интернет да намеря точно и ясно описание на това какво точно представлява този тип архитектура. За съжаление се натъкнах на различна и не съвсем ясна информация. Типичното определение е, че това е вариант на Харвард архитектурата, който позволява програмната памет да се достъпва като памет за данни. За целта се осигурява допълнителна шина за данни между процесора и програмната памет (Фиг.108).

Съгласно това определение повечето микроконтролери, които са документирани, че използват Харвард архитектура, всъщност използват модифицирана Харвард архиитектура, тъй като могат да четат/записват данни от/в програмната си памет.

Според друго определение модифицираната Харвард архитектура използва кеш-памети както е показано на Фиг.109. Когато процесорът изпълнява инструкции от кеш-паметта[1], той се държи като Харвард-процесор, а когато изпълнява от главната памет, той се държи като фон Нойман-процесор.

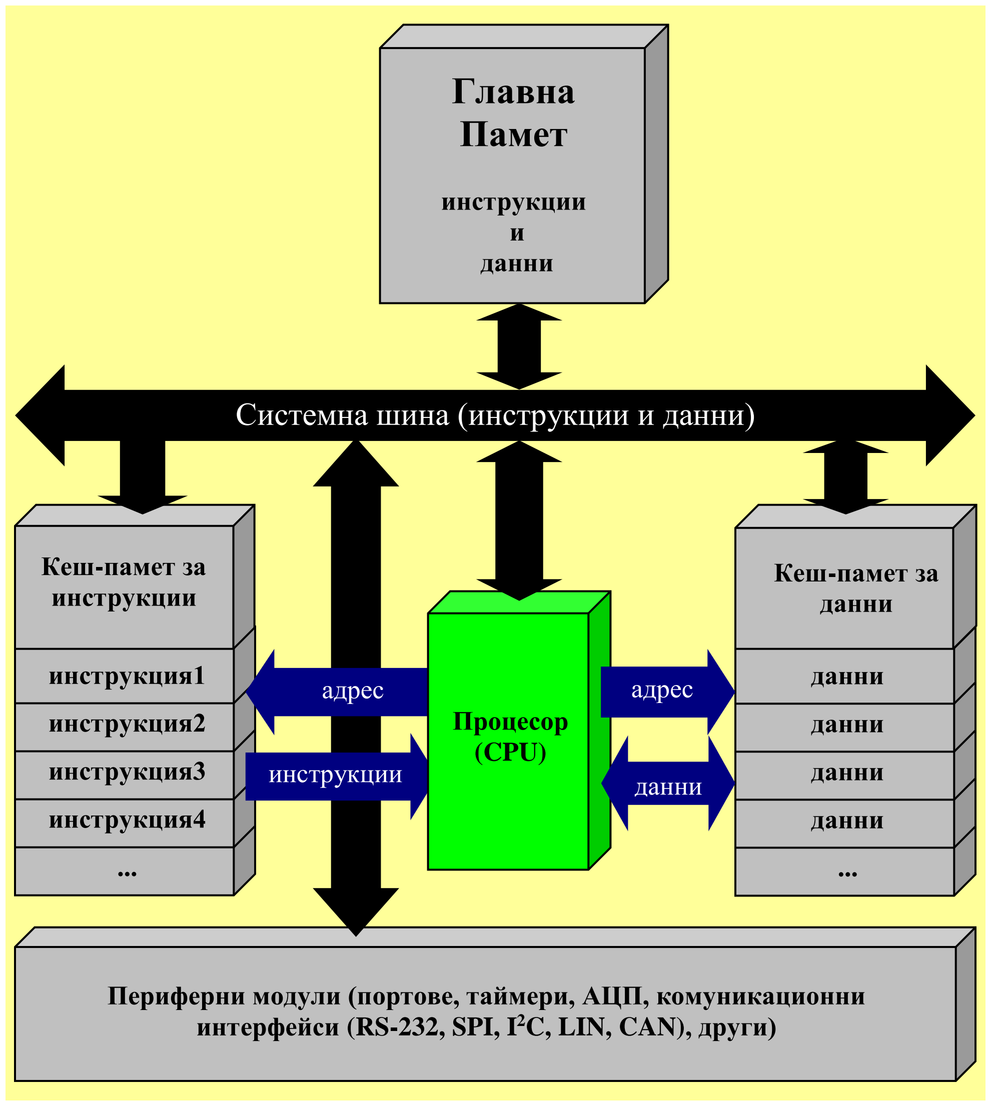
Забележка[1]: Кеш-паметта е вид RAM памет, достъпа до която е по-бърз от достъпа до обичайната RAM памет.



Фиг. 108 Модифицирана Харвард архитектура-вариант 1

Типични представители на този тип архитектура са ARM9 и x86 процесорите.

Според мен определението хибридна архитектура би било по-подходящо в случая.



Фиг. 109 Модифицирана Харвард архитектура-вариант 2

15.2.3 Набор от инструкции

Инструкциите са важна характеристика на всеки процесор. Те влияят на размера на кода, т.е. . върху обема на заеманата от програмата памет. Една инструкция се състои от опкод и операнди. Опкодът указва каква операция извършва инструкцията (събиране, изваждане, преместване, преход и

т.н.), а операндите са обектите, върху които се прилага операцията. Операндите могат да бъдат регистри на процесора, клетки от паметта за данни и други. Основните метрики, които характеризират инструкциите са:

- Размер на инструкцията;
- Скорост на изпълнение;
- Налични инструкции;
- Адресни режими[1].

Забележка[1]: Адресният режим определя начина, по който инструкцията достига операндите си. За съжаление няма единно обозначаване на адресните режими.

В зависимост от набора инструкции, които микроконтролерите могат да изпълняват, те се разделят на:

- CISC (Complex Instruction Set Computer);
- RISC (Reduced Instruction Set Computer).

CISC-микроконтролерите, както подсказва самото име, се характеризират с голям брой инструкции и адресни режими, съответно по-голяма степен на сложност в сравнение с RISC-микроконтролерите.

16 Програма. Конструкция и изпълнение

Преди да разгледаме компонентите на микроконтролера е необходимо да се запознаем с конструкцията и изпълнението на една програма. Програмата се съхранява в програмната памет на микроконтролера под формата на машинни инструкции. Машинните инструкции са комбинации от нули и единици и се състоят от **опкод** и **операнди**. Опкодът указва каква операция извършва инструкцията (събиране, изваждане, преместване, преход и т.н.), а операндите са обектите, върху които се прилага операцията. Програмата се състои от:

- основна програма;
- подпрограми;
- подпрограми за обработка на прекъсвания.

Подпрограмата е парче код, което извършва специфична задача. Това парче код се изпълнява, когато подпрограмата се извика със специална инструкция (типично обозначавана като CALL на асемблерен език). Една подпрограма може да извиква други подпрограми и т.н. При завършване на изпълнението на подпрограмата, изпълнението на програмата се връща към инструкцията, която се намира непосредствено след инструкцията за извикване на подпрограмата.

Подпрограмата за обработка на прекъсване (ISR – Interrupt Service Routine) е подобна на обикновена подпрограма с тази разлика, че тя се извиква автоматично от хардуера при възникване на прекъсване. Прекъсванията са сигнали, които се генерират от периферните модули при възникване на определени събития (промяна на нивото на извод и т.н.). При завършване на изпълнението на ISR-подпрограмата,

изпълнението на програмата се връща към инструкцията, която е била наред за изпълнение точно преди да възникне прекъсването и преди хардуера да извиква ISR-подпрограмата.

Фиг.110 илюстрира описаните процеси.

Фиг. 110 Конструкция и изпълнение на програма

Процесорът изпълнява инструкциите на основната програма. При достигане на инструкцията **CALL подпрограма** процесорът спира изпълнението на инструкциите на основната програма и се прехвърля за изпълнение на инструкциите на подпрограмата. По време на изпълнение на

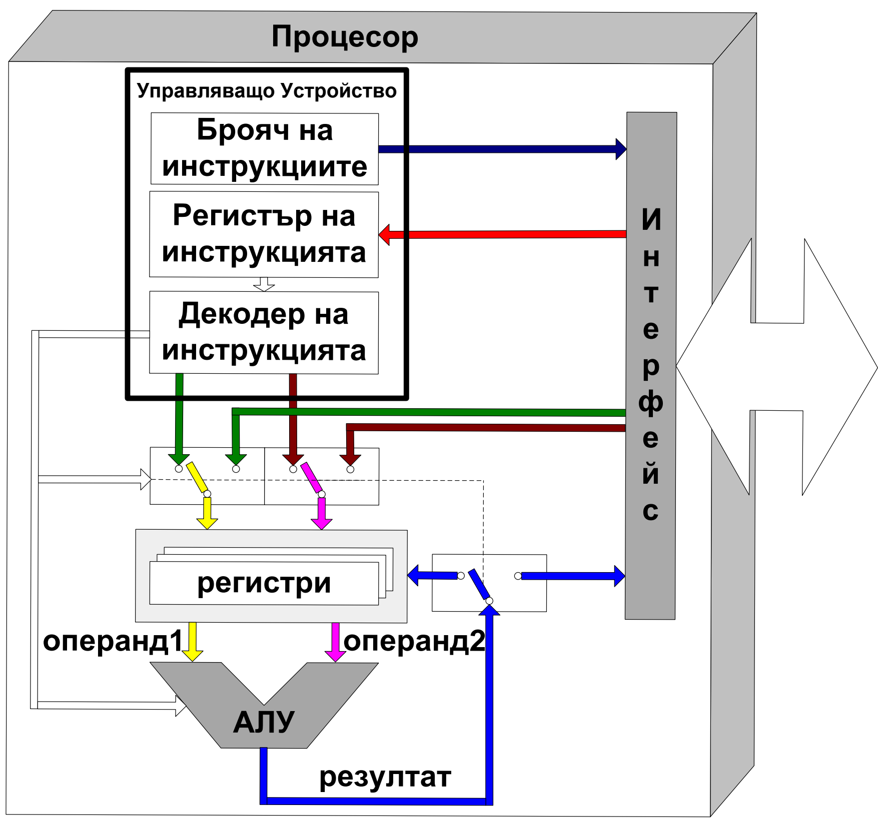
тези инструкции процесорът получава сигнал за прекъсване (моментът на възникване на прекъсването е отбелязан с). Процесорът спира изпълнението на инструкциите на подпрограмата и се прехвърля за изпълнение на инструкциите на ISR-подпрограмата. При завършване на ISR-подпрограмата процесорът възобновява изпълнението на инструкциите на подпрограмата оттам откъдето е бил стигнал преди да възникне прекъсването. При завършване на инструкциите на подпрограмата, процесорът се връща към изпълнението на първата инструкция след инструкцията **CALL subroutine** в основната програма.

17 Описание на компонентите на микроконтролера

17.1 Базови компоненти на микроконтролера

17.1.1 Централен процесор

Фиг.111 показва обобщената блокова схема на един процесор.



Фиг. 111 Централен процесор

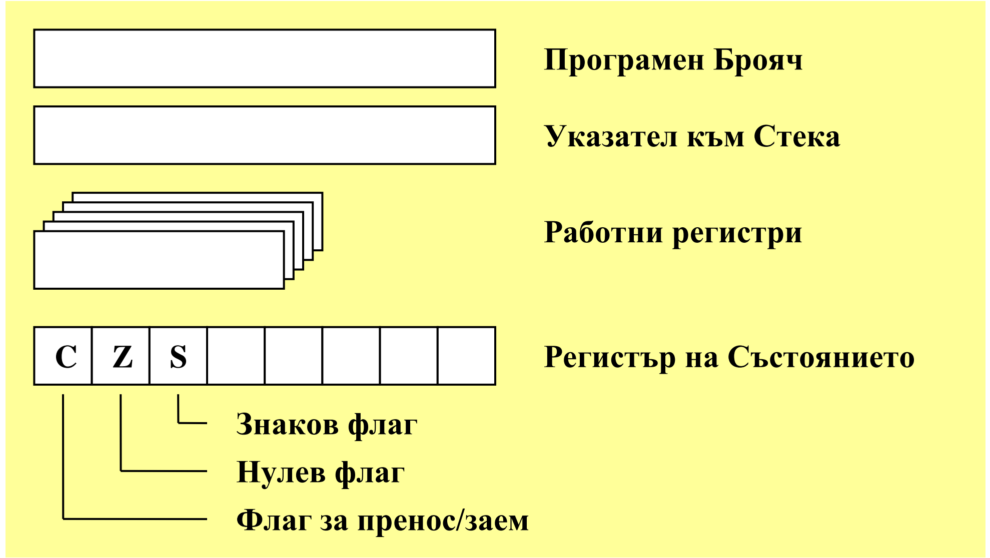
Той се състои от **Аритметично & Логическо устройство (АЛУ)**, **Регистри** и **Управляващо Устройство (УУ)**. Процесорът е отговорен за обработката на данни. Как ще се обработват данните, зависи от програмата, която му е зададена да изпълнява. От гледна точка на процесора,

програмата е набор от инструкции, които му указват какво да прави. Типичните инструкции, които един процесор изпълнява, са събиране и изваждане на числа, сравняване на числа, по-сложните процесори извършват умножение и деление на числа и много други. УУ отговаря за извличането на инструкциите на програмата и декодирането им. АЛУ, както подсказва неговото име, извършва аритметични и логически операции (като събиране, изваждане, умножение, преместване наляво или надясно и др.) върху данните. Регистрите на процесора са специални клетки памет, до които достъпът е най-бърз. Част от тези регистри имат строго специализирано предназначение, а друга част могат да се използват за временно съхранение на данни и се наричат регистри с общо предназначение. Данните, върху които АЛУ извършва някаква операция, се наричат най-общо операнди.

При подаване на захранване на микроконтролера, **Броячът на инструкциите** се зарежда с адреса в програмната памет, където се намира първата инструкция на програмата. Този адрес се нарича още **ресет вектор**. Инструкцията се чете от програмната памет и се копира в **Регистъра на Инструкциите**, а в **Брояча на инструкциите** автоматично се зарежда адреса на следващата за изпълнение инструкция. Извлечената инструкция се декодира от **Декодера на Инструкциите**, който в резултат на това "казва" на АЛУ каква операция трябва да извърши. В зависимост от инструкцията данните могат да се съдържат в самата инструкция или да се извлекат от паметта за данни. Също така в зависимост от инструкцията резултатът от операцията може да се съхрани отново в някой от регистрите на процесора или в паметта за данни. Накратко казано, изпълнението на една програма представлява извличане на инструкциите, от които тя се състои, тяхното декодиране и изпълнението им.

От гледна точка на един програмист процесорът може да се

представи по следния начин:



Фиг. 112 Програмен модел на процесор

Ролята на Програмния Брояч (**PC - Program Counter**) вече би трябвало да е ясна. Той сочи към поредната за изпълнение инструкция, т.е. . съдържа адреса в програмната памет на поредната за изпълнение инструкция.

Указателят към Стека (**SP – Stack Pointer**) е специален регистър, който се използва за работа със стека. Стекът е специална памет, използвана за съхранение на съдържанието на Програмния Брояч (адреса на връщане) при извикване на подпрограми и ISR-подпрограми. Стекът е разгледан по-подробно в 17.1.3 Памет за данни.

Работните регистри са регистри за обща употреба, използвани за временно съхранение на данни и резултати от аритметични и логически операции. Такива регистри обикновено се наричат акумулатори. Типично микроконтролерите имат поне един акумулатор.

Регистърът на състоянието (**PSW – Program Status Word**) е

специален регистър, който съдържа различни флагове (битове), отразяващи резултата от изпълнението на всяка инструкция. Типично този регистър съдържа следните флагове:

- Флаг за пренос/заем C

  Този флаг показва дали има пренос от най-старшите битове на операндите при операция събиране или заем при операция изваждане.

- Нулев флаг Z

  Този флаг показва дали резултатът от аритметична или логическа операция е нула.

- Знаков флаг S

  Този флаг показва дали резултатът от аритметична или логическа операция е положителен или отрицателен.

В зависимост от микроконтролера, Регистърът на Състоянието може да съдържа и други флагове.

17.1.2 Програмна памет

Програмната памет, както подсказва името, служи за съхранение на инструкциите, които изграждат програмата. Съвременните технологии позволяват в програмната памет да се съхраняват и константни данни. Програмната памет се нарича още ROM (**Read Only Memory**). ROM паметта може да се изгражда по различни технологии:

- Masked-ROM

При този тип памет фърмуерът се програмира директно от производителя при изработката на паметта, чрез фотолитографски метод и не може повече да бъде променян.

- PROM (Programmable ROM) или OTP (One-time programmable ROM);

  Този тип памет може да бъде програмирана с помощта на специално устройство, наречено **програматор**. В непрограмирано състояние всички битове имат стойност 1. Програмирането на даден бит в 0 става чрез прогаряне с помощта на електрически ток на специална връзка, наречена бушон (**fuse**). След прогарянето връзката не може да бъде възстановена.

- EPROM (Erasable Programmable ROM)

  Този тип памет също както и PROM, позволява да бъде програмирана с програматор. Предимството на EPROM обаче е възможността съдържанието да се изтрива. Изтриването става като паметта се изложи на ултравиолетово лъчение за няколко минути. За целта корпусът на паметта е снабден със специално прозорче (Фиг.113).

Фиг. 113 EPROM

Поради високата цена (заради специалното стъклено прозорче) този тип памет се е използвал само по време на развойния процес. Финалните устройства са използвали микроконтролери с masked-ROM или PROM.

- EEPROM (Electrically Erasable Programmable ROM)

  EEPROM паметта вътрешно е подобна на EPROM, с тази разлика, че съдържанието може да бъде изтрито с електрическо напрежение, вместо с ултравиолетова светлина. Този тип памет позволява изтриване и програмиране на индивидуални байтове. Като недостатък на EEPROM може да се посочи относително дългото време за изтриване/програмиране на отделните байтове. По тази причина EEPROM паметите не се използват като програмна памет (за съхранение на инструкции), а като енергонезависима памет за съхранение на калибрационни данни например или някаква друга информация.

- Flash EEPROM (или просто Flash)

  Флаш паметта е подобна на EEPROM с тази разлика, че изтриването става на блокове (страници), а не на индивидуални байтове. Има два основни типа флаш памети: NAND и NOR. NAND-флаш паметта може да бъде записана/четена на блокове (страници). NOR-флаш паметта позволява запис/четене на индивидуални байтове, което ги прави подходящи за съхранение на инструкции.

  NOR-флаш паметта (или нейни разновидности) е предпочитаният тип програмна памет в съвременните микроконтролери.

17.1.3 Памет за данни

Паметта за данни, както подсказва името, служи за съхранение на временни данни, използвани в програмата и

резултати, получени от изпълнението й. Паметта за данни се нарича още памет с произволен достъп (**RAM** - **Random Access Memory**). Информацията в RAM се пази само докато е налично захранване. При изключване на захранването, данните в RAM се губят.

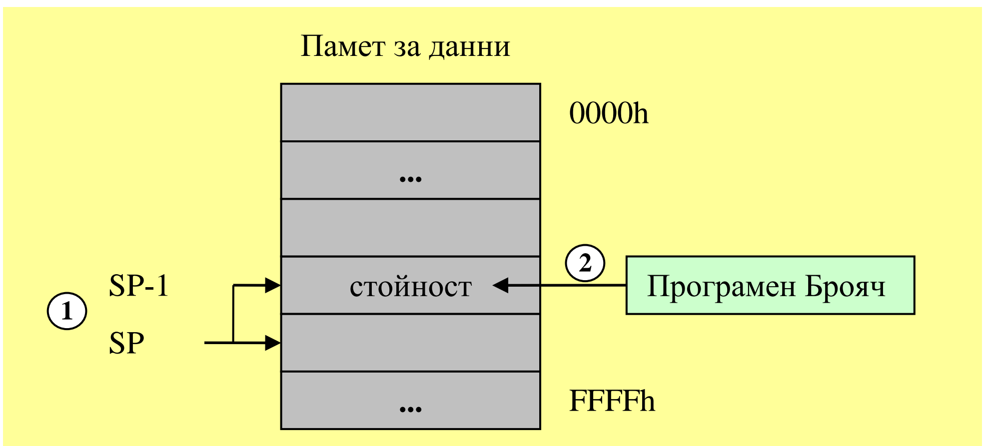
RAM паметта се дели основно на два типа: статична RAM (**SRAM**) и динамична RAM (**DRAM**). SRAM запазва съдържанието си, докато се подава електрическо захранване. DRAM паметта освен наличието на захранващо напрежение изисква и специален контролер, който периодично да опреснява данните. Ако това не се прави, съдържанието на DRAM се губи след определено време дори и при наличие на захранващо напрежение. При изключване на захранването съдържанието в SRAM и DRAM се губи.

Предимството на SRAM пред DRAM е, че достъпът до SRAM е по-бърз и реализацията е по-проста. Като недостатък може да се посочи по-високата цена.

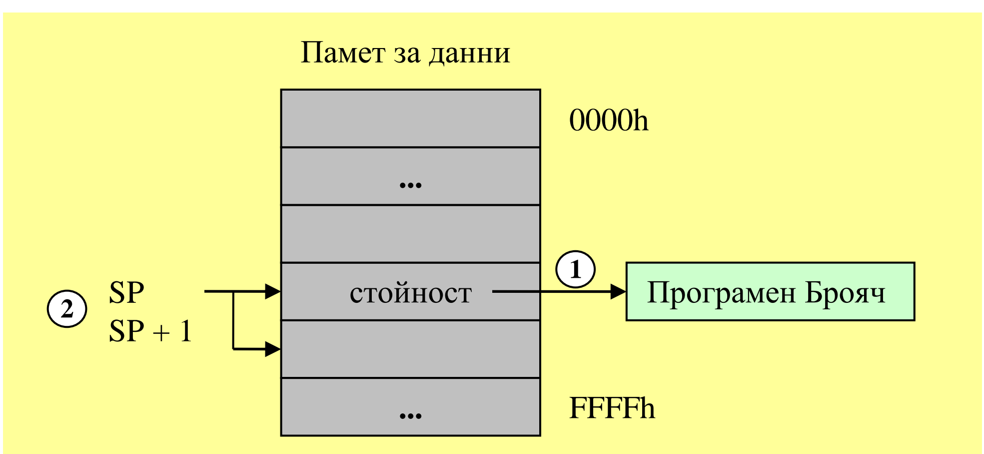
Типично RAM паметта, освен за съхранение на временни данни, се използва и за разполагане на стека. Стекът се използва за съхранение на Програмния Брояч (адреса на връщане) при извикване на подпрограма или ISR-подпрограма. Допълнително при извикване на ISR-подпрограма, в стека се съхранява и Регистъра на Състоянието. Указателят към Стека е специален регистър, който се използва при запис и четене на стека и винаги сочи към върха на стека. Стекът работи на принципа последен влязъл първи излязъл (**LIFO –Last Input First Output**). Някои микроконтролери имат специални инструкции за запис/четене на потребителски данни в/от стека. В зависимост от микроконтролера операциите със стека могат да се изпълняват по един от следните механизми:

- предекремент-постинкремент

При този механизъм при запис в стека Указателят към Стека първо се намалява, след което се извършва запис в стека. При четене от стека данните първо се извличат, след което Указателят към Стека се увеличава към следващата клетка. Следващите две фигури илюстрират процесите на запис и четене на стека по механизма предекремент-постинкремент.



Фиг. 114 Запис в стека по механизма предекремент-постинкремент

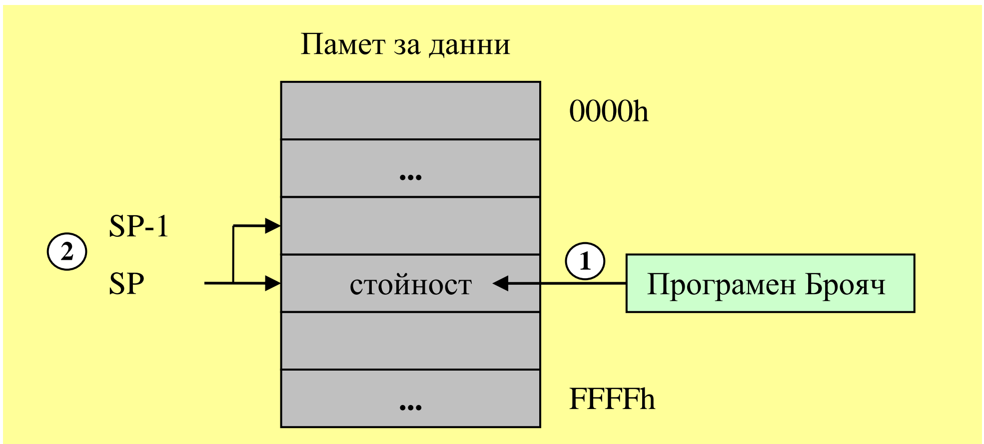


Фиг. 115 Четене от стека по механизма предекремент-постинкремент

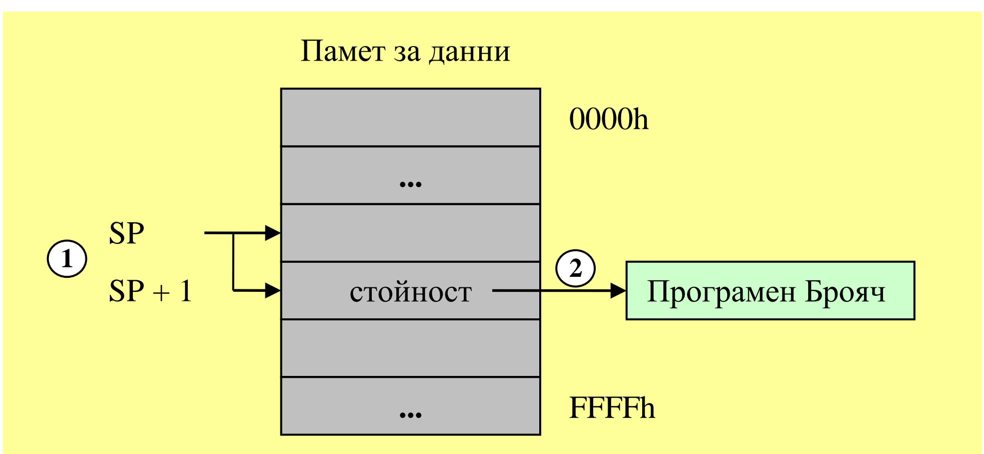
- постдекремент-преинкремент

При този механизъм при запис в стека първо се

извършва записа, след което Указателят към Стека се намалява. При четене Указателят към Стека първо се увеличава, след което се извличат данните от стека. Следващите две фигури илюстрират процесите на запис и четене на стека по механизма постдекремент-преинкремент.



Фиг. 116 Запис в стека по механизма постдекремент-преинкремент

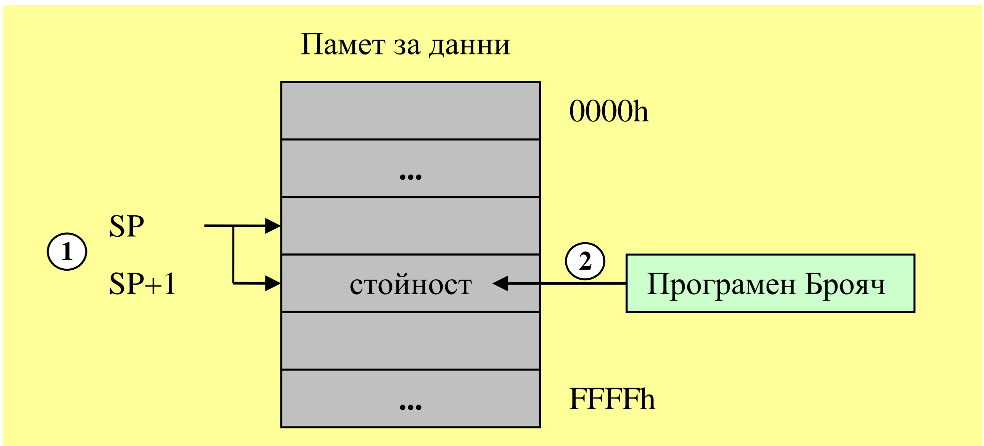


Фиг. 117 Четене от стека по механизма постдекремент-преинкремент

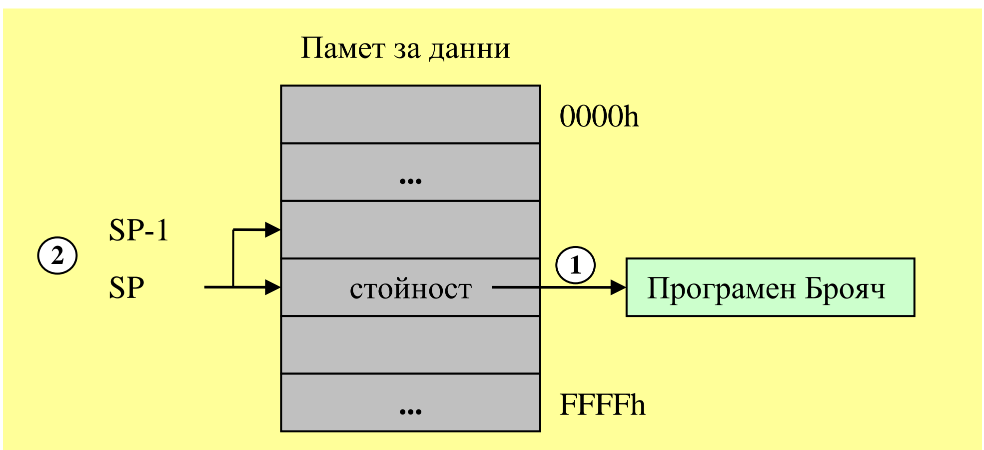
- преинкремент-постдекремент

При този механизъм при запис в стека Указателят към Стека първо се увеличава, след което данните се

записват в стека. При четене от стека, данните първо се извличат, след което Указателят към Стека се намалява. Следващите две фигури илюстрират процесите на запис и четене на стека по механизма преинкремент-постдекремент.



Фиг. 118 Запис в стека по механизма преинкремент-постдекремент

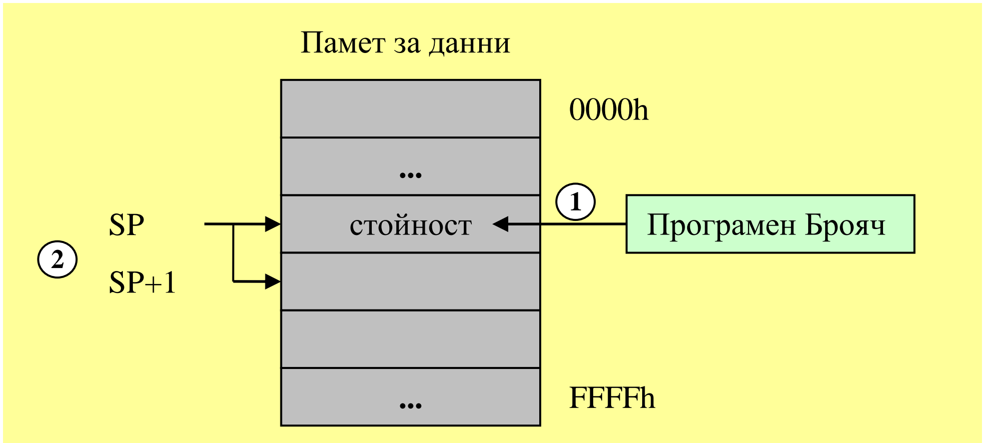


Фиг. 119 Четене от стека по механизма преинкремент-постдекремент

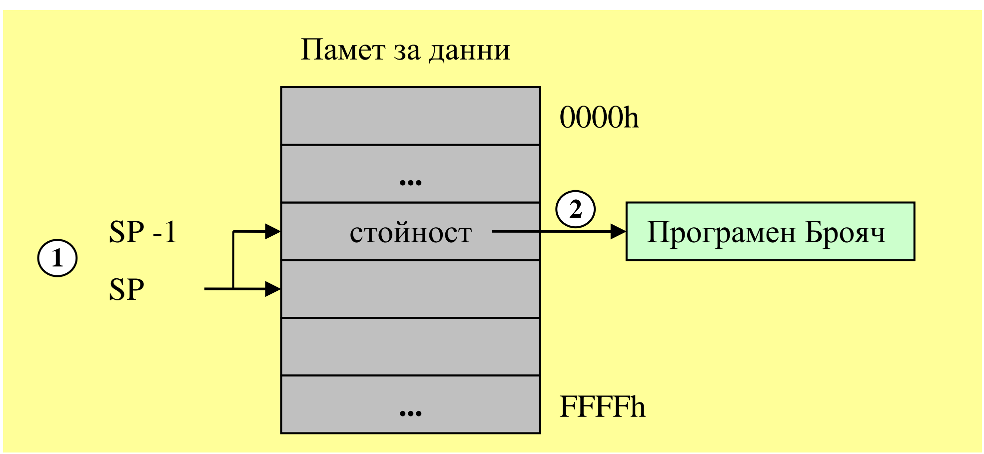
- постинкремент-предекремент

При този механизъм при запис в стека данните първо се записват в стека, след което Указателят към Стека се увеличава. При четене от стека Указателят към

Стека първо се намалява, след което данните се извличат от стека. Следващите две фигури илюстрират процесите на запис и четене на стека по механизма постинкремент-предекремент.



Фиг. 120 Запис в стека по механизма постинкремент-предекремент



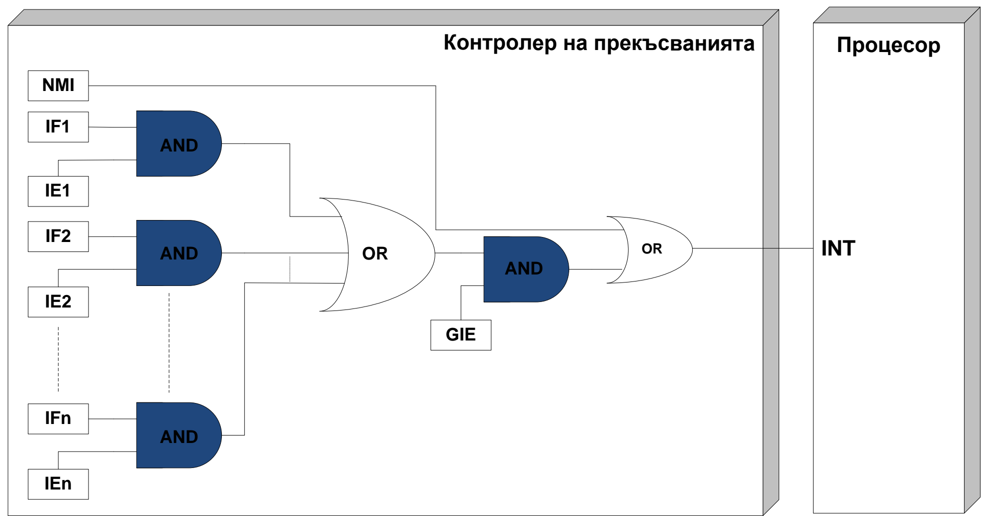
Фиг. 121 Четене от стека по механизма постинкремент-предекремент

Стек, който използва паметта за данни, се нарича софтуерен стек. При софтуерния стек отговорност на програмиста е да конфигурира началото на стека чрез запис на съответния адрес в Указателя към Стека. Има микроконтролери, които реализират хардуерен стек, т.е. . за стек се използва специална отделна памет, която не е част от паметта за

данни. При този тип реализация на стека Указателят към Стека обикновено се управлява автоматично от хардуера и не е достъпен за програмиста.

17.1.4 Контролер на прекъсванията

Фиг.122 показва обобщена блокова схема на контролера на прекъсванията.



Фиг. 122 Контролер на прекъсванията

Както вече споменах по-рано, контролерът на прекъсванията приема сигналите за прекъсване, получени от периферните модули, и известява процесора, че е възникнало някакво събитие (промяна на нивото на даден извод, препълване на таймер и т.н.). Всеки източник на прекъсване се характеризира със следните атрибути:

- Флаг на прекъсване IF (**Interrupt Flag**);

  Флагът на прекъсване е просто бит, който показва наличието или отсъствието на прекъсване. Логическа

1 означава, че е възникнало прекъсване.

- Локален бит за разрешаване на прекъсването IE (**Interrupt Enable**);

  Този бит служи за локално забраняване/разрешаване на дадено прекъсване. Ако прекъсването е забранено (IE = 0), то няма да бъде обработено от процесора.

- Подпрограма за обработка на прекъсването (**ISR – Interrupt Service Routine**).

  На всеки източник на прекъсване се назначава специална подпрограма, която се извиква при възникване на това прекъсване (при условие, че прекъсването е разрешено). При получаване на сигнал за прекъсване процесорът спира изпълнението на текущата програма и стартира ISR-подпрограмата, отговаряща за прекъсването. След като ISR-подпрограмата завърши изпълнението на задачата, за която отговаря, процесорът отново се връща към изпълнението на предишната си програма.

Много често изпълнението на даден участък от една програма не трябва да се прекъсва от други подпрограми. За улеснение всички прекъсвания могат да се забраняват с помощта на глобален бит GIE (**Global Interrupt Enable**). Когато GIE = 0 всички прекъсвания са забранени. Това означава, че едно прекъсване ще бъде разрешено само ако съответния локален разрешаващ бит IEx е 1 и глобалният бит GIE е 1.

Някои микроконтролери имат източници на прекъсване, които не могат да бъдат забранявани. Такива прекъсвания се наричат немаскируеми (**non-maskable**). Най-често някой от изводите на микроконтролера служи за генериране на такова прекъсване по нарастващ или падащ фронт. Обикновено

изводът се означава като NMI (**Non-Maskable Interrupt**). Противоположно на немаскируемите прекъсвания, прекъсванията, които могат да се забраняват, се наричат маскируеми (**maskable**).

Тук е момента да Ви запозная със следните термини:

- Ресет вектор

  Ресет векторът е адреса в програмната памет откъдето започва изпълнението на програмата.

- Вектор на прекъсване

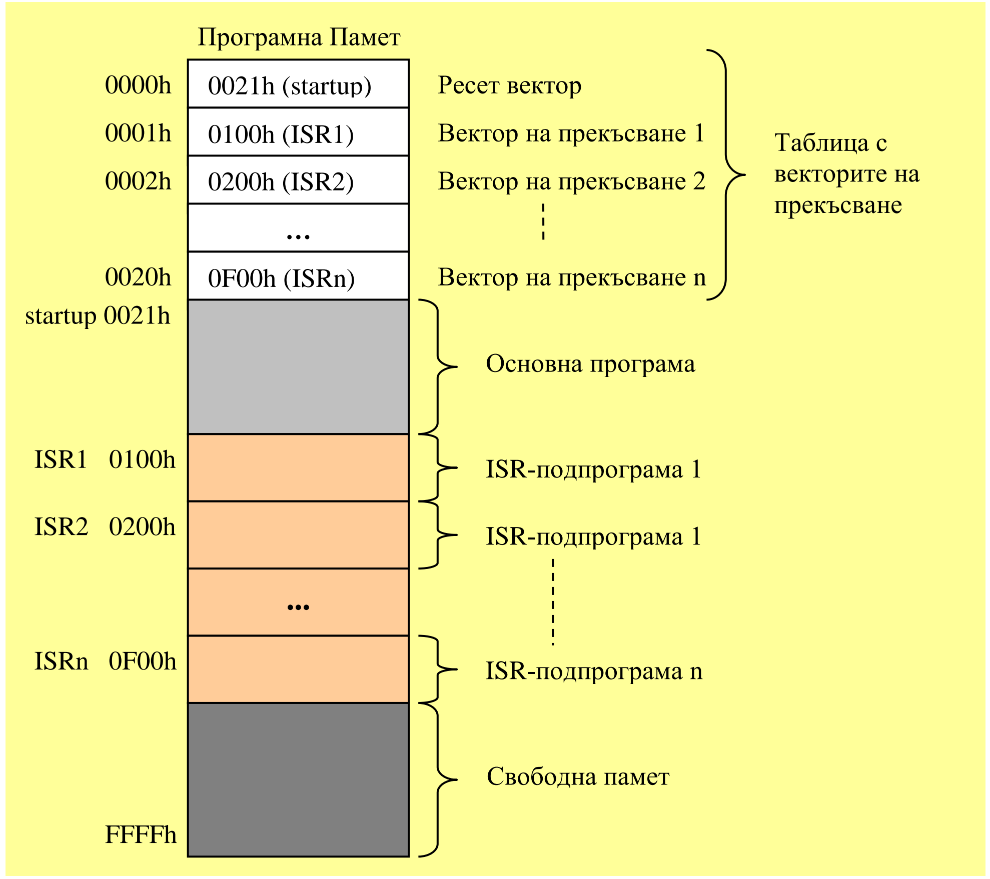
  Векторът на прекъсване се нарича адреса в програмната памет, където се разполага ISR-подпрограмата на даден източник на прекъсване.

- Таблица с векторите на прекъсване

  Таблицата с векторите на прекъсване се нарича част от програмната памет, където се съхраняват векторите на прекъсване (адресите на ISR-подпрограмите) и ресет вектора. Отговорност на програмиста е да попълни тази таблица коректно. При възникване на прекъсване или системен ресет, съответният вектор автоматично се извлича от таблицата с векторите на прекъсване и се зарежда в Програмния Брояч.

Следващата фигура дава по-ясна представа за описаните по-горе термини. Използваните адреси са примерни. Обикновено таблицата с векторите на прекъсване заема началото на програмната памет.

Прекъсванията имат приоритет. Ако възникнат две или повече прекъсвания едновременно, процесорът ги обработ-



Фиг. 123 Таблица с векторите на прекъсване

ва, започвайки от прекъсването с най-висок приоритет. Има различни механизми за задаване на приоритет на прекъсванията. Например един от начините е използването на реда на векторите на прекъсване в таблицата с векторите на прекъсване. Източник на прекъсване, чийто вектор на прекъсване се намира по-напред в таблицата, има по-висок приоритет. Това се нарича още хардуерен приоритет. Недостатък на хардуерния приоритет е, че не позволява конфигуриране на приоритетите на източниците.

Освен хардуерния приоритет някои микроконтролери предоставят и задаване на т.нар. софтуерен приоритет. На всеки източник на прекъсване с помощта на конфигурационни битове се задава определен приоритет.

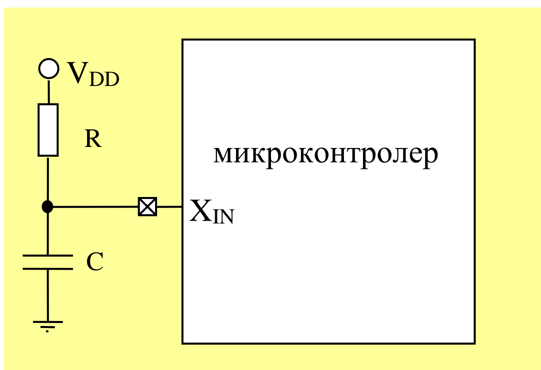
Ако два или повече източника имат еднакъв софтуерен приоритет, под внимание се взима хардуерния приоритет.

17.1.5 Генераторен блок

Генераторният блок служи да генерира всички необходими тактови сигнали, необходими за работата на компонентите на микроконтролера. Типично в микроконтролерите се използват следните видове генератори:

- RC генератор;

  Фиг.124 показва свързването на RC група към извод на микроконтролера.

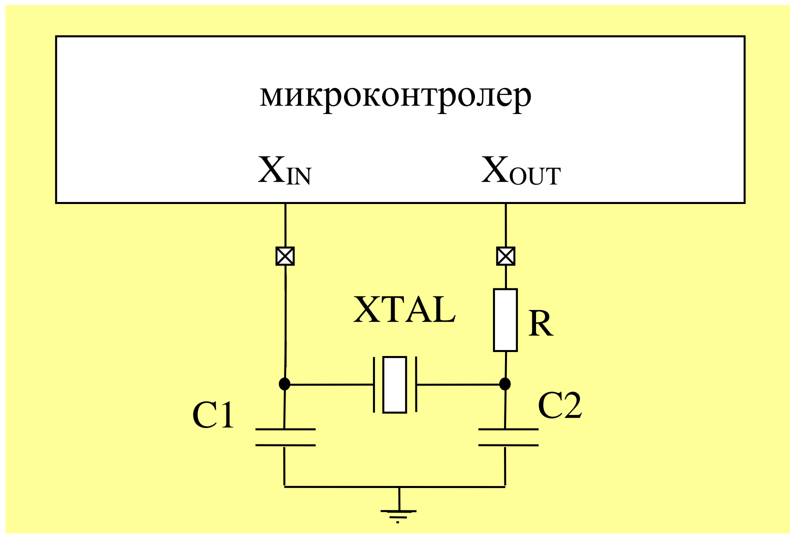


Фиг. 124 RC генератор

  RC генераторите се характеризират с кратко време за стартиране и ниска стойност. Като недостатък може да се посочи ниската стабилност на генерирания сигнал. Честотата на изходния сигнал при RC генераторите много се влияе от промените в захранващото напрежение, температурата и стареенето на елементите.

- Кварцов или керамичен генератор;

Фиг.125 показва типичната схема на свързване на външните елементи на кварцов/керамичен генератор.

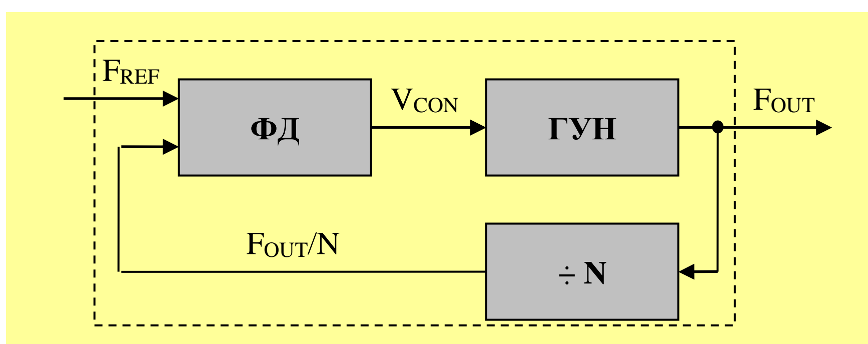


Фиг. 125 Кварцов/керамичен генератор

Използването на кварцов/керамичен резонатор гарантира висока стабилност на честотата на генерирания сигнал.

- PLL (Phase-locked loop) генератор.

Фиг.126 показва блоковата схема на PLL генератор.



Фиг. 126 PLL генератор

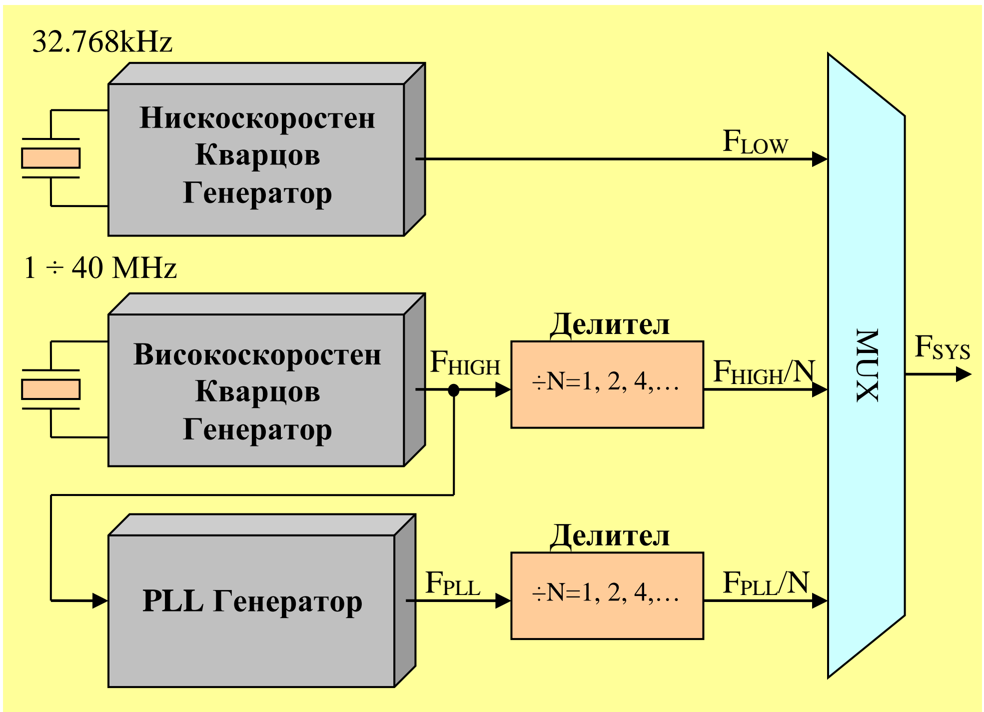
Основният елемент на PLL генератора е генератор,

управляван с напрежение ГУН (**VCO – Voltage Controlled Oscillator**). Честотата на ГУН се регулира с напрежение, подавано от фазов детектор ФД. Фазовият детектор генерира управляващо напрежение V_{CON}, пропорционално на фазовата разлика на изходния сигнал F_{OUT} (минал евентуално през делител на честота ÷N) и опорен (референтен) сигнал F_{REF}. Сигналът F_{REF} обикновено се получава от кварцов генератор.

PLL генераторите се характеризират с висока стабилност на честотата на генерирания сигнал.

Някои микроконтролери предоставят вътрешни генератори, които работят без използването на външни компоненти.

Фиг.127 показва типичен генераторен блок на един микроконтролер.



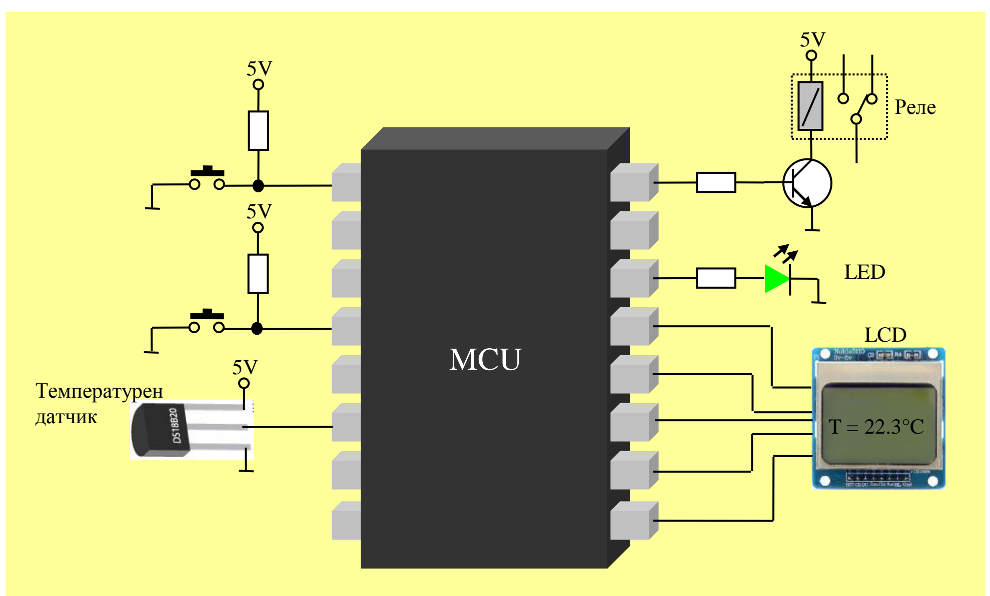
Фиг. 127 Блокова схема на генераторен блок

Системният тактов сигнал F_{SYS} може да се получи от различни източници: F_{LOW}, F_{HIGH}/N или F_{PLL}/N, където N е коефициент на делене 1, 2, 4 и т.н.

17.2 Периферия

17.2.1 Портове

Една от най-важните характеристика на микроконтролерите е броят на изводите. Изводите осъществяват физическата връзка на периферните модули на микроконтролера с външния свят. Най-общо казано те се използват за извеждане/въвеждане на информация към/от външния свят (Фиг.128).



Фиг. 128 Връзка на микроконтролера с външния свят

Изводите на микроконтролера са групирани в портове. Броят на изводите в един порт е максимум броят на битовете на шината за данни, т.е. . в 8-битов микроконтролер един порт може да съдържа максимум 8 извода. Портовете в

микроконтролерите обикновено се обозначават с букви или цифри например PA, PB, PC и т.н., или P0, P1, P2 и т.н., а изводите PA0 (извод 0 на PA), PA1 (извод 1 на PA) и т.н. или P00 (извод 0 на P0), P01 (извод 1 на P0) и т.н. Различните производители използват различни обозначения.

Един порт се характеризира със следните управляващи регистри:

- Регистър за определяне на посоката на данните PxDDR[1] (Port x Data Direction Register);

  Този регистър определя посоката на извода, дали изводът ще работи като изход или вход. Обикновено 0 означава, че изводът е изход, а 1 – вход.

- Изходен регистър за данни PxOUT[1] (Port x Data Out Register);

  Данните в този регистър се извеждат на съответните изводи (при условие, че изводите са конфигурирани като изходи).

- Входен регистър за данни PxIN[1] (Port x Data in Register).

  Този регистър съдържа стойностите на съответните изводи (при условие, че изводите са конфигурирани като входове).

Забележка[1]: Използваното обозначение на регистъра е примерно. Различните производители използват различни обозначения.

Някои микроконтролери използват общ регистър за извеждане/въвеждане на данни. При запис в регистъра

данните се извеждат на изводите (при условие, че изводите са конфигурирани като изходи), а при четене на регистъра се четат стойностите на изводите (при условие, че изводите са конфигурирани като входове). Не е задължително изводите на един порт да се конфигурират само като входове или само като изходи. Всеки извод на порта може индивидуално да се конфигурира като вход или изход.

По практически съображения изводите на един микроконтролер могат да изпълняват различни функции.

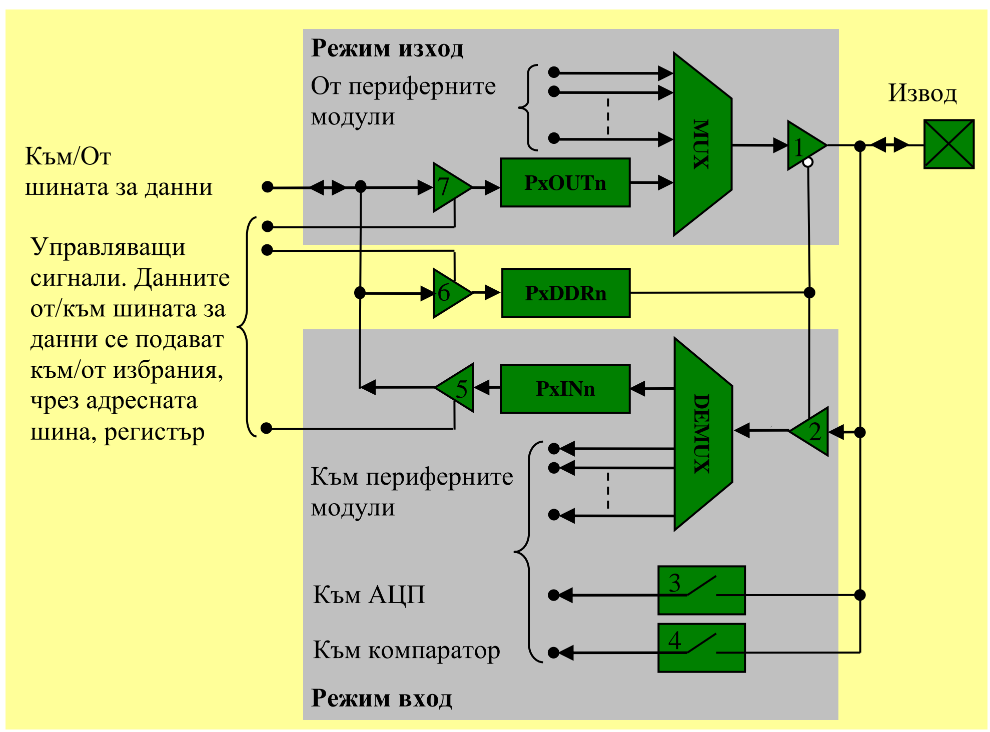
- Цифров вход-изход за обща употреба (**GPIO – General Purpose Input Output**);
- Аналогов вход (вход към АЦП);
- Вход на таймер;
- Изход на таймер;
- Вход на компаратор;
- Изход на компаратор;
- Други.

Фиг.129 описва функционалната блокова схема на един извод на порт.

Когато изводът е конфигуриран като цифров изход (бит PxDDRn = 0), при запис в регистъра за данни PxOUT данните постъпват по шината за данни в съответния бит PxOUTn (x е номера на порта, а n е номера на съответния бит), а от там, през съответния вход на мултиплексора и изходния буфер 1, на извода.

Когато изводът е конфигуриран като изход на компаратор

например, стойността на изхода на компаратора, през съответния вход на мултиплексора и изходния буфер 1, се подава на извода.



Фиг. 129 Функционална блокова схема на извод на порт

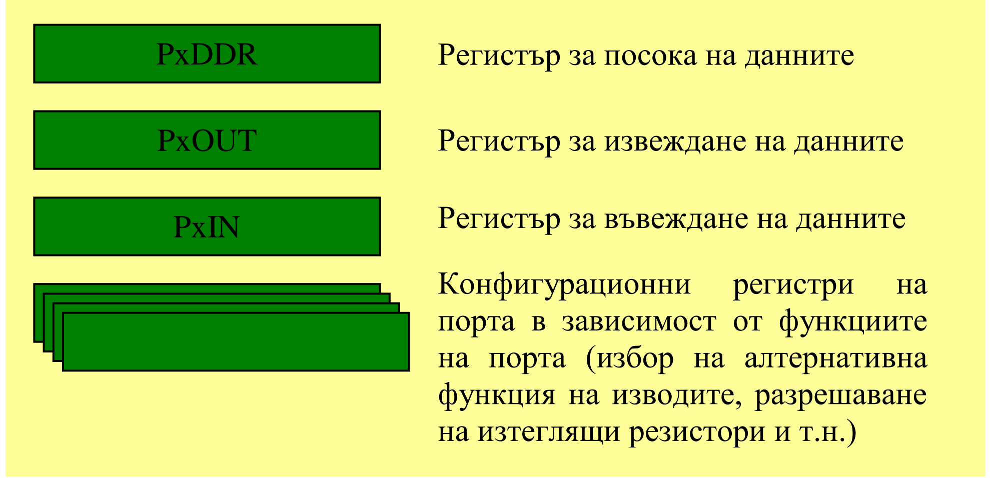
Когато изводът е конфигуриран като цифров вход (бит PxDDRn = 1), стойността на извода, през входния буфер 2 и съответния изход на демултиплексора, постъпва в съответния бит PxINn на входния регистър за данни PxIN. При четене на регистър PxIN данните се подават на шината за данни, а от там към приемника на данните.

Когато изводът е конфигуриран като вход на компаратор например, стойността на извода, през входния буфер 4, се подава към компаратора.

Не е задължително всички изводи да имат еднаква функционалност. Възможно е някои изводи да работят само

като входове или само като изходи. Също така изводите на даден порт могат да имат допълнителни свойства, като вътрешен изтеглящ резистор (**pull up**), който може да се включва и изключва. Някои изводи могат да работят в режим на изход с отворен дрейн и т.н.

От гледна точка на един програмист портът може да се представи последния начин (Фиг.130):



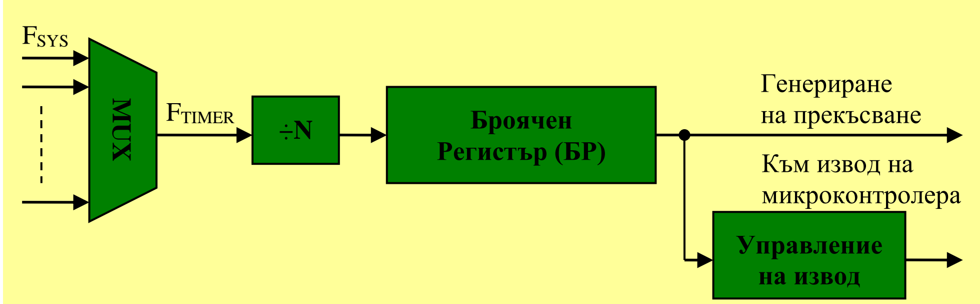
Фиг. 130 Програмен модел на порт

Отговорност на програмиста е да конфигурира тези регистри коректно съгласно функцията, която ще се изпълнява от даден извод в приложението.

17.2.2 Таймери

Фиг.131 показва функционална блокова схема на един таймер. Основният елемент на таймера е броячен регистър БР. БР се увеличава на всеки активен фронт на тактовия сигнал на таймера F_{TIMER}. Тактовият сигнал на таймера типично се получава от системния тактов сигнал F_{SYS}, но може да се получава и от други източници например външен

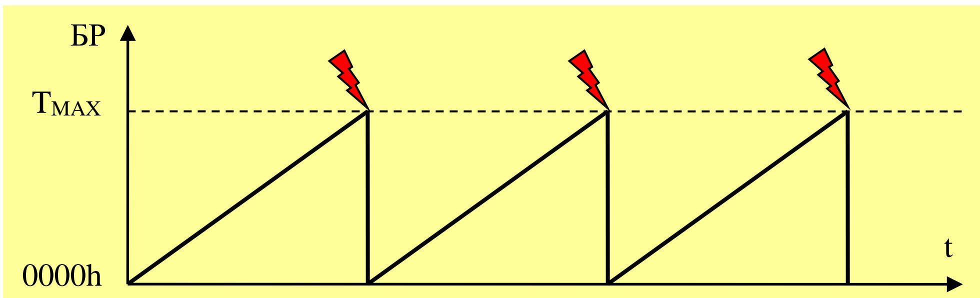
тактов сигнал от извод на микроконтролера, от изход на компаратор и други.



Фиг. 131 Функционална блокова схема на таймер

Обикновено когато тактовият сигнал идва от вътрешния системен такт на микроконтролера, се казва, че таймерът работи в **таймерен режим**. Когато тактовият сигнал идва от извод на микроконтролера, се казва, че таймера работи в **броячен режим**.

Следващата фигура описва принципа на работа на таймера.



Фиг. 132 Принцип на работа н атаймера

Броячният регистър се увеличава на всеки активен фронт на тактовия сигнал F_{TIMER}. При достигане на максималната броячна стойност T_{MAX} Броячният Регистър се нулира и се генерира сигнал за прекъсване. Обикновено при препълване на таймера се генерира и сигнал, който управлява извод на микроконтролера например нивото на извода се превключва от ниско във високо ниво или обратно.

Обикновено в таймерен режим таймерът се използва за генериране на времезадръжка. Времезадръжката може да се изчисли от следното уравнение:

$$t[s] = \frac{T_{MAX} - T_0}{F_{TIMER}[Hz]} * N$$

където

T_0 – началната стойност заредена в брояча
T_{MAX} – максимална стойност на брояча
N – коефициент на деление на честотата на тактовия сигнал
F_{TIMER} – тактов сигнал на таймера

Например ако F_{TIMER} = 10MHz, T_0 = 0, T_{MAX} = 65535 (16-битов брояч) и N = 1, то

$$t = \frac{65535 - 0}{10 * 10^6 [Hz]} * 1 = 6553.5 * 10^{-6} [s] = 6553.5[\mu s]$$

От гледна точка на един програмист таймерът може да се представи последния начин (Фиг.133):

Фиг. 133 Програмен модел на таймер

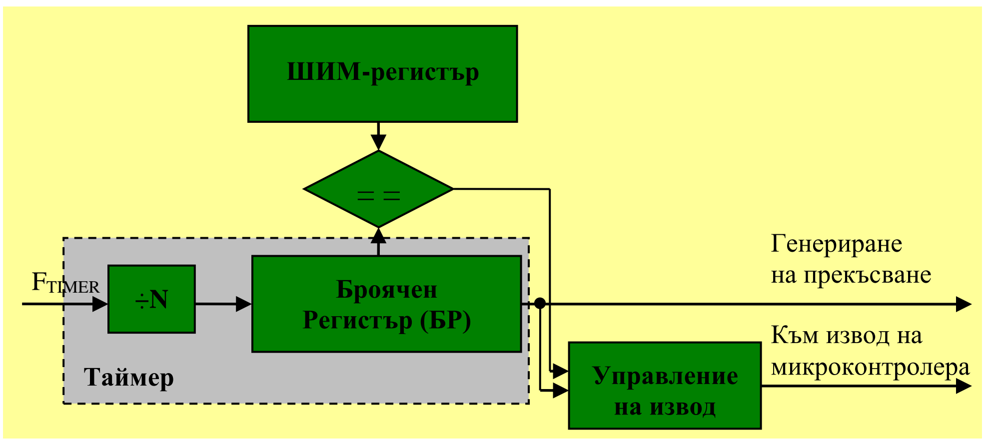
Отговорност на програмиста е да конфигурира тези регистри коректно съгласно функцията, която ще се изпълнява от таймера.

Обикновено таймерите могат да работят и в други режими,

като режим на генериране на широчинно-импулсни сигнали (**PWM – Pusle-With Modulation**), режим на прихващане (**Capture**), режим на сравнение (**Compare**) и др. Следващите подточки разглеждат гореспоменатите режими.

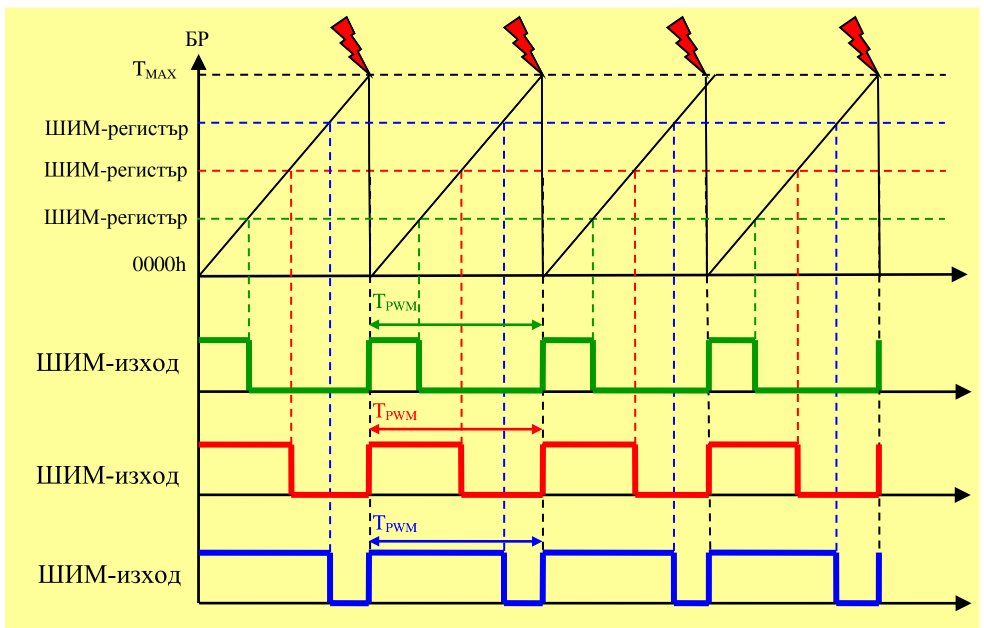
17.2.3 ШИМ (PWM - Pulse-Width Modulation)

ШИМ (Широчинно-импулсен Модулатор) се използва за генериране на правоъгълни сигнали с променлива широчина на импулса. Фиг.134 показва функционалната блокова схема на ШИМ.



Фиг. 134 Функционална блокова схема на ШИМ

Фиг.135 описва принципа на работа на ШИМ. Приема се, че началното ниво на извода е високо. Обикновено началното ниво на ШИМ-изхода може да се конфигурира. Стойността на БР се сравнява със стойността, съхранена в ШИМ-регистъра. Когато БР достигне стойността, съхранена в ШИМ-регистъра, ШИМ-изходът се превключва в ниско ниво и БР продължава да брои. Когато БР достигне максималната си броячна стойност, ШИМ-изходът се превключва отново във високо ниво, БР се нулира, генерира се прекъсване и цикълът се повтаря. На Фиг.135 е показан



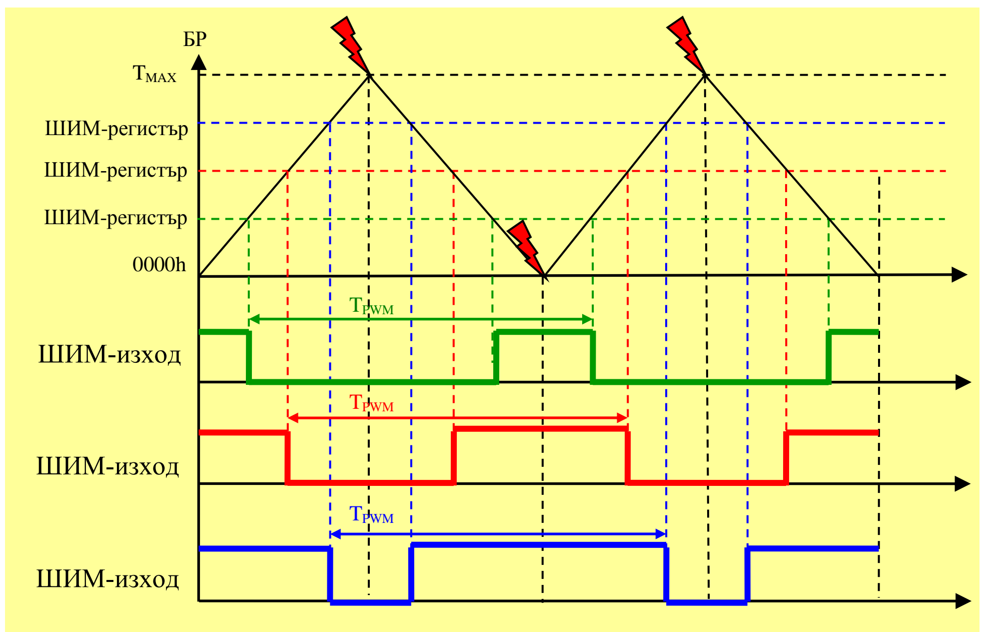
Фиг. 135 Принцип на работа на ШИМ (режм edge-aligned)

резултата на ШИМ-изхода при три различни стойности на ШИМ-регистъра. Както може да видите, променяйки стойността в ШИМ-регистъра може да променяте широчината на генерирания импулс. Периода на изходния сигнал зависи от времето за препълване на брояча, респективно от честотата на тактовия сигнал. Отношението между високото ниво на импулса t_H и периода на T_{PWM} се нарича **коефициент на запълване k** на сигнала.

$$k[\%] = \frac{t_H}{T_{PWM}} * 100$$

Описаният режим на работа на ШИМ-модула се нарича фронтално-подравнен (**edge-aligned**).

Някои микроконтролери могат да използват и т.нар. централно-подравнен (**center-aligned**) режим на генериране на ШИМ-сигнал. На Фиг.136 е показан резултата на ШИМ-изход, работещ в централно-подравнен режим, при три раз-



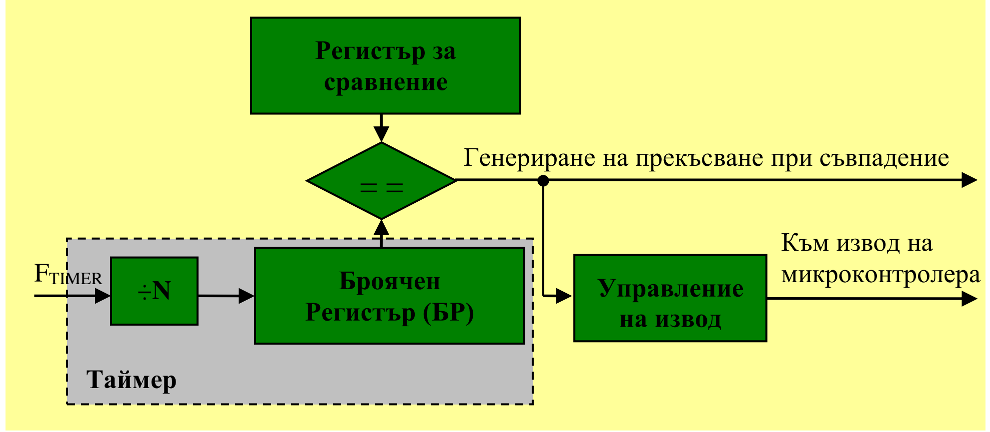
Фиг. 136 Принцип на работа на ШИМ (режим center-aligned)

лични стойности на ШИМ-регистъра. В този режим БР първо брои от нула нагоре. Когато БР достигне стойността, съхранена в ШИМ-регистъра, ШИМ-изходът се превключва в ниско ниво и БР продължава да брои. Когато БР достигне максималната си броячна стойност, на следващия тактов сигнал броенето започва в обратна посока. Когато БР достигне стойността, съхранена в ШИМ-регистъра, ШИМ-изходът се превключва във високо ниво и описания цикъл се повтаря.

Обикновено ШИМ не се реализира като отделен компонент, а е част от функционалността на таймерния модул.

17.2.4 Сравнение (Compare)

Фиг.137 показва функционалната блокова схема на модул за Сравнение. Стойността на Регистър за Сравнение постоянно се сравнява с Броячен Регистър БР.



Фиг. 137 Функционална блокова схема на модул за Сравнение

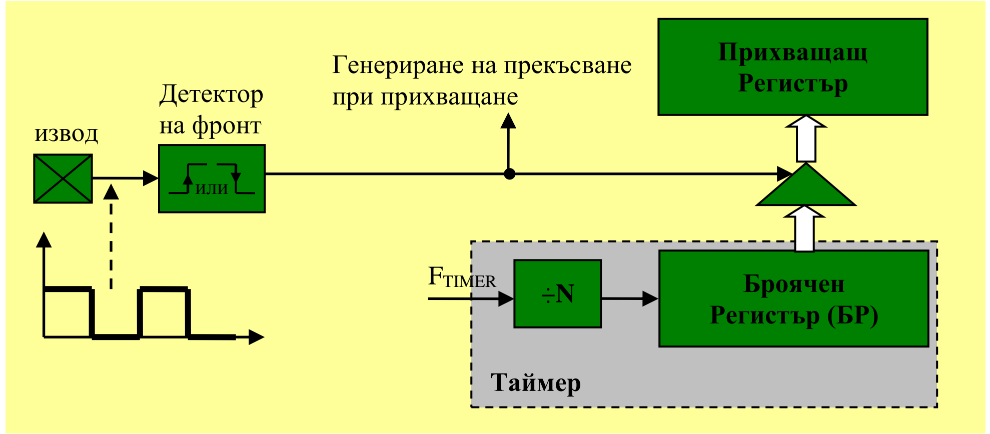
Когато стойностите се изравнят, са възможни различни ситуации:

- Генериране на прекъсване;
- Генериране на прекъсване и превключване на нивото на извод (от високо в ниско или обратно);
- Генериране на прекъсване и превключване на извод във високо ниво;
- Генериране на прекъсване и превключване на извод в ниско ниво.

Обикновено модулът за Сравнение не се реализира като отделен компонент, а е част от функционалността на таймерния модул.

17.2.5 Прихващане (Capture)

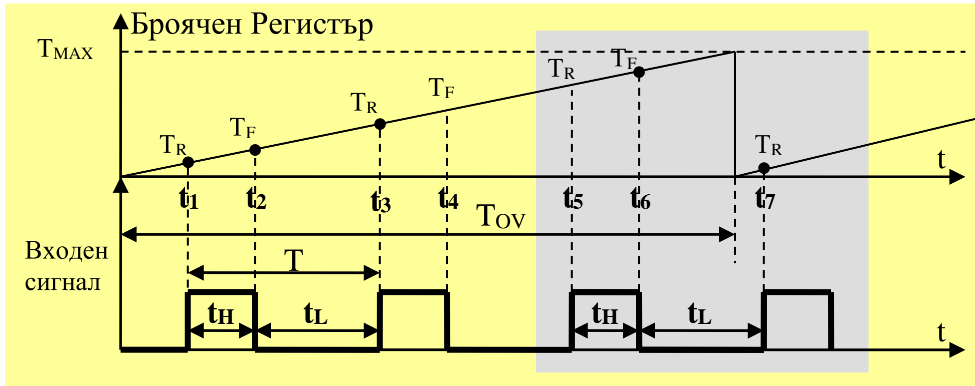
Фиг.138 показва функционалната блокова схема на модул за Прихващане.



Фиг. 138 Функционална блокова схема на модул за Прихващане

Основният елемент в модула на прихващане е детектор на фронт. Този детектор следи нивото на извод и може да се конфигурира да детектира нарастващ фронт, падащ фронт или и двата. При детектиране на активен фронт се генерира сигнал за прекъсване и стойността на Броячния Регистър се копира автоматично в Прихващащ Регистър.

Модулът на прихващане типично се използва за измерване на продължителността на високото или ниското ниво, съответно и периода на входен сигнал. т.е. . може да се използва за реализиране на честотомер. Фиг.139 описва принципа на измерване на коефициента на запълване на входен сигнал ($к = t_H / (t_H + t_L)$).



Фиг. 139 Измерване на коефициент на запълване на входен сигнал

Ако периодът на входния сигнал е известен, за по-лесно измерване, периодът на препълване на таймера T_{OV} се избира да е по-голям от периода T на входния сигнал. При това условие, Броячният Регистър на таймера ще се препълва най-много веднъж в рамките на един период на входния сигнал (вижте сивото каре на Фиг.128).

Стойността на Броячния Регистър се прихваща при всяко възникване на нарастващ и падащ фронт на входния сигнал. Прихванатите стойности са обозначени като T_R при нарастващ (**rising**) фронт и T_F при падащ (**falling**) фронт. Максималната стойност на броене на таймера е означена като T_{MAX}.

При тези допускания ако прихванатата стойност е по-голяма от предходната, високото и ниското ниво на входния сигнал се изчисляват по следния начин:

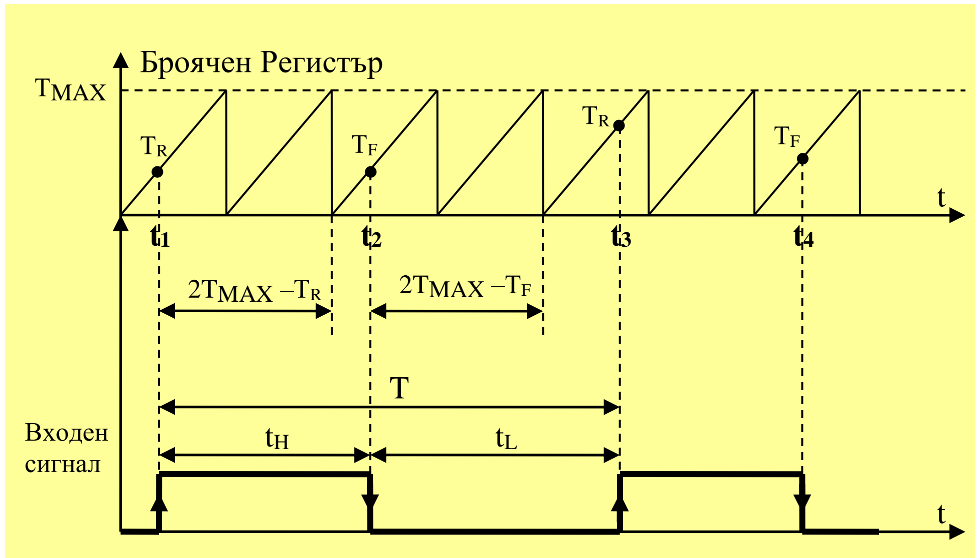
$t_H = T_F(t_2) - T_R(t_1)$
$t_L = T_R(t_3) - T_F(t_2)$

Ако прихванатата стойност е по-малка от предходната, това означава, че е възникнало препълване на таймера. На Фиг.139 тази ситуация е показана в сивото каре ($T_R(t_7) < T_F(t_6)$). Това трябва да се вземе на предвид при изчисляване на t_H и t_L.

$t_H = T_F(t_6) - T_R(t_5)$
$t_L = (T_{MAX} - T_F(t_6)) + T_R(t_7)$

Ако периодът на входния сигнал е неизвестен, е възможно таймерът да се препълва повече от веднъж в рамките на един период на входния сигнал. Това трябва да се вземе предвид при изчисление на t_H и t_L. Фиг.140 описва такава ситуация.

$t_H = (2T_{MAX} - T_R(t_1)) + T_F(t_2)$
$t_L = (2T_{MAX} - T_F(t_2)) + T_R(t_3)$

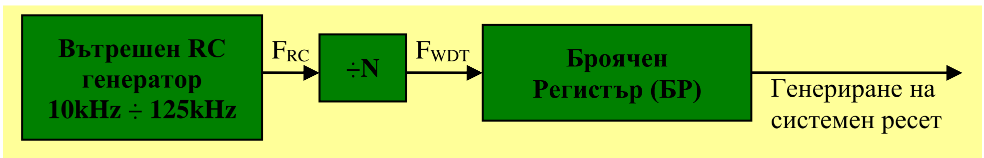


Фиг. 140 Измерване на коефициент на запълване на входен сигнал

Обикновено модулът за Прихващане не се реализира като отделен компонент, а е част от функционалността на таймерния модул.

17.2.6 Стражеви таймер (Watchdog)

Стражевият таймер (WDT-таймер) е таймер със специална функция. Фиг.141 показва функционалната блокова схема.

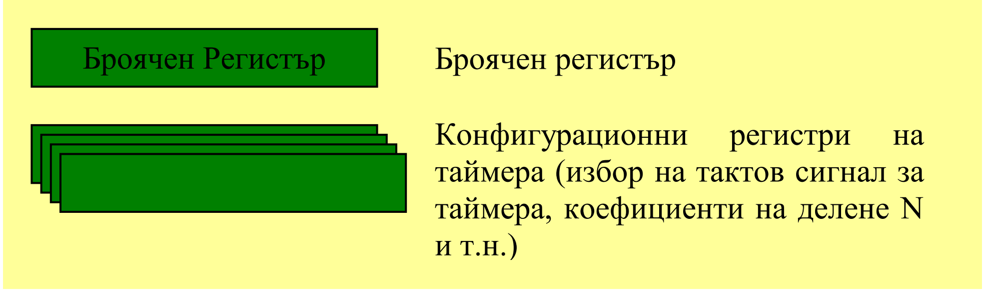


Фиг. 141 Функционална блокова схема на WDT-таймер

Обикновено тактовият сигнал на WDT-таймера се получава от собствен RC-генератор. При препълване на таймера се генерира системен ресет, което води до рестартиране на микроконтролера и изпълняваната от него програма.

WDT-таймерът трябва да се нулира периодично преди да се препълни и да генерира ресет. В правилно работеща програма той не трябва да се препълва, с изключение на някои ситуации, при които умишлено се генерира ресет с негова помощ. Ролята на WDT-таймера е да извежда микроконтролера в познато състояние в случай, че изпълнението на програмата изпадне в непредвидена ситуация поради някаква причина (например космическа частица прасне микроконтролера и промени стойността на даден бит ☺, или просто причината е в „задклавиатурното устройство", което е по-често срещаната и по-вероятна ситуация ☺).

От гледна точка на един програмист WDT-таймерът може да се представи по следния начин (Фиг.142):

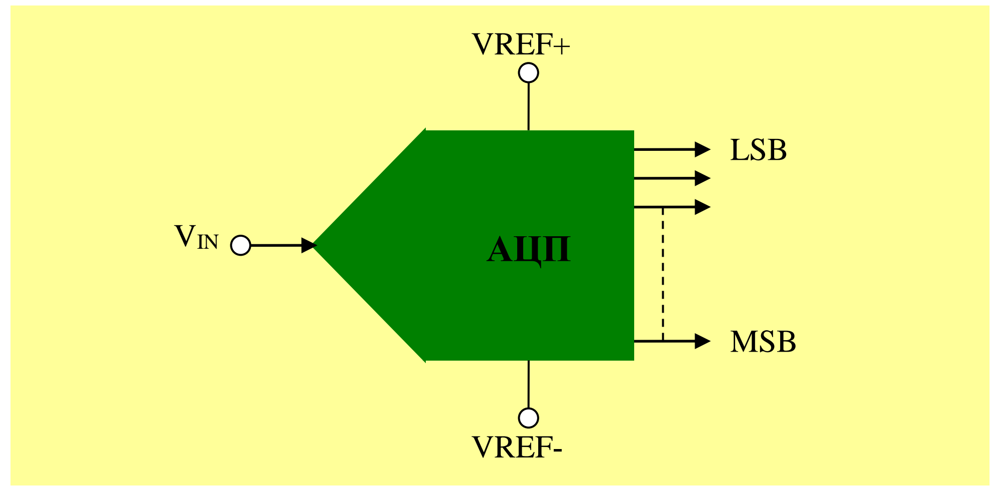


Фиг. 142 Програмен модел на WDT-таймер

Отговорност на програмиста е да конфигурира тези регистри коректно.

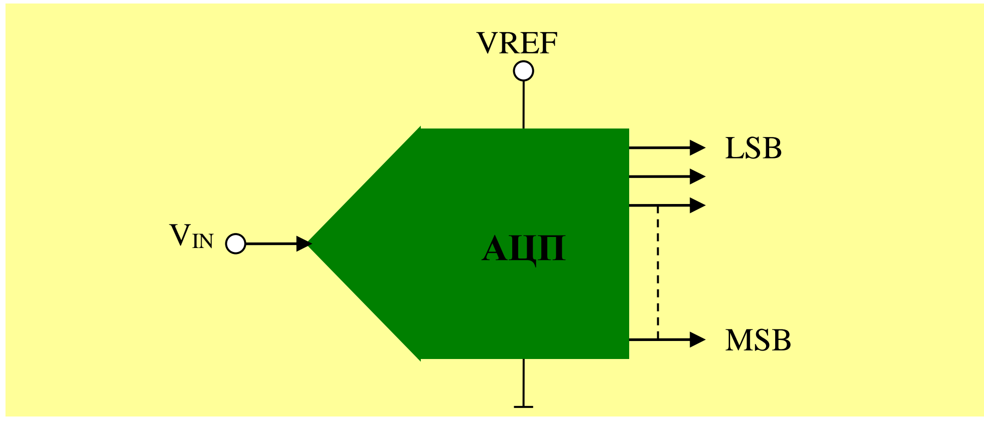
17.2.7 Аналого-цифров преобразувател (АЦП)

АЦП (**ADC – Analog-to-Digital Convertor**) преобразува входен аналогов сигнал в цифров сигнал. Типично резултатът е 10- или 12-битов. Фиг.143 показва стандартния символ за обозначаване на АЦП.



Фиг. 143 Символ за обозначаване на АЦП

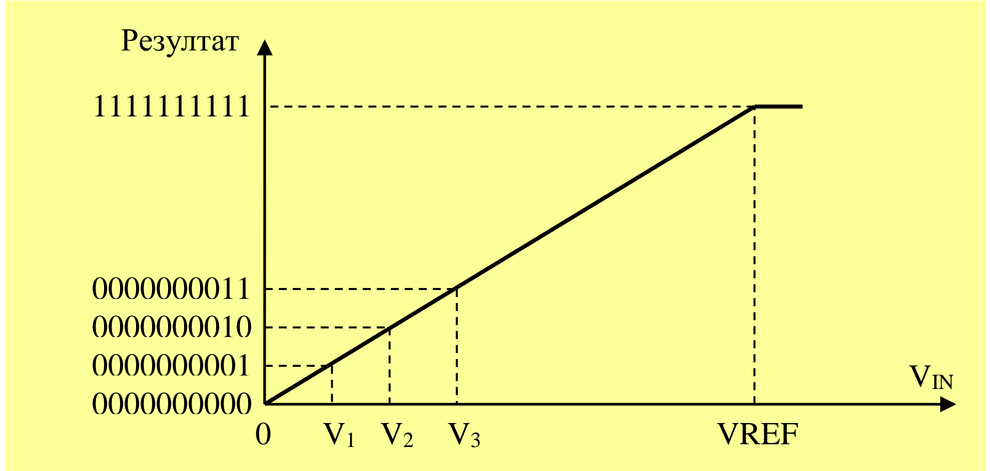
Обхватът на входното напрежение, което може да се измери от АЦП, се определя от опорните напрежения VREF+ и VREF-, т.е. . VREF- $\leq V_{IN} \leq$ VREF+. Ако АЦП може да преобразува отрицателни и положителни входни напрежение, той се нарича биполярен, докато АЦП, който преобразува само положителни входни напрежения се нарича еднополярен. Типично микроконтролерите използват еднополярни АЦП-та (Фиг.144).



Фиг. 144 Еднополярен АЦП

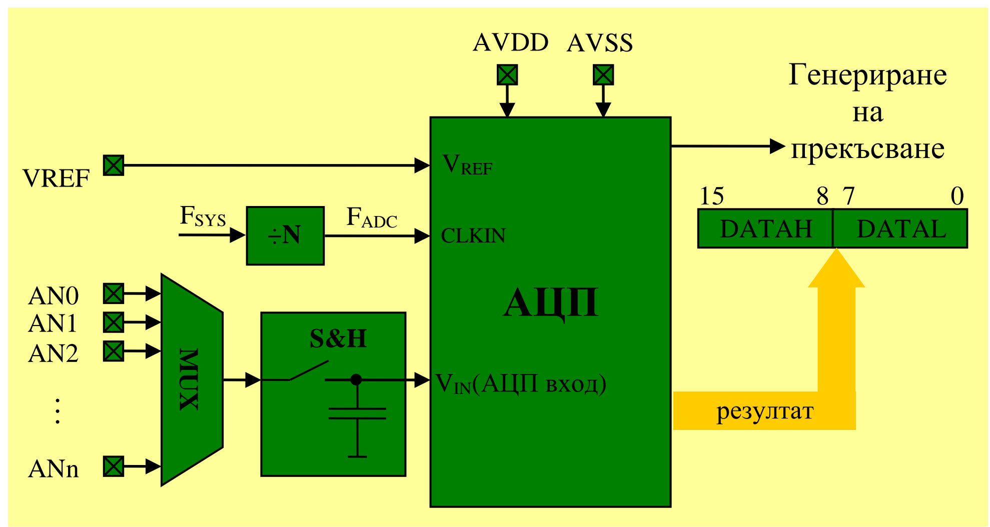
Входният диапазон от напрежения (0V ÷ VREF) се разделя на под обхвати. Броят на под обхватите е 2^n, където n е броят

на битовете, с които се представя резултата от преобразуването. На всеки под обхват съответства определена цифрова (двоична) стойност. Един под обхват се нарича още LSB (Least Significant Bit). Броят на тези под обхвати се нарича **резолюция** на АЦП. Например 10-битов АЦП има резолюция 2^{10} = 1024, т.е. . целия входен диапазон от напрежения може да се представи от 1024 на брой 10-битови двоични кодове. Следващата фигура показва трансферната характеристика на АЦП.



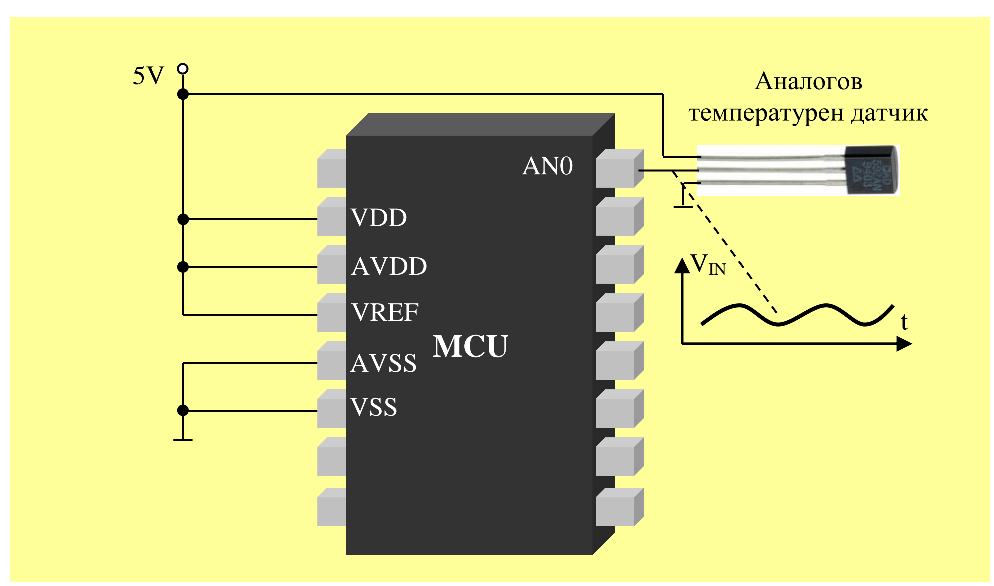
Фиг. 145 Трансферна характеристика на АЦП

Фиг.146 показва функционалната блокова схема на АЦП модул. При подаване на стартов сигнал към АЦП да започне преобразуване, избраният сигнал през мултиплексора се подава на схема **Sample&Hold**. Основният елемент на тази схема е кондензатор, който се зарежда до стойността на преобразуваното напрежение. Времето необходимо за зареждане на кондензатора се нарича време за семплиране t_S (**sample time**). След изтичане на това време, напрежението на кондензатора се подава към входа на АЦП за преобразуване. Времето необходимо на АЦП да преобразува напрежението се нарича време за задръжка t_H (**hold time**). Пълното време за преобразуване е $t_{CONV} = t_S + t_H$ и е основният параметър, определящ бързодействието на АЦП.



Фиг. 146 Функционална блокова схема на АЦП модул

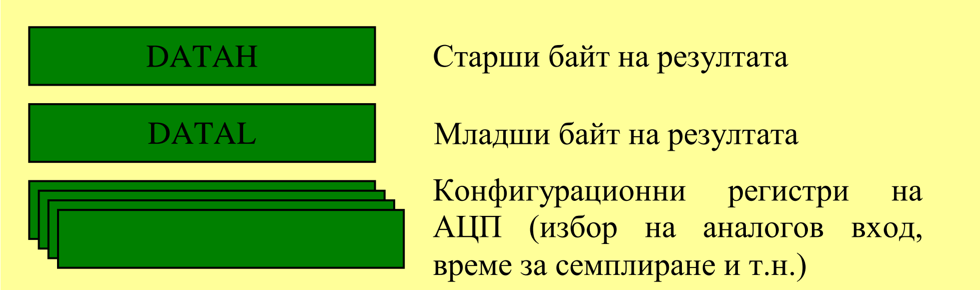
Следващата фигура демонстрира типична схема на измерване на аналогов изход на температурен датчик.



Фиг. 147 Примерна схема на измерване с АЦП

AVDD и AVSS са изводите за захранване на АЦП. Обикновено за тях са предвидени отделни изводи на микроконтролера.

От гледна точка на един програмист АЦП модулът може да се представи последния начин (Фиг.148):

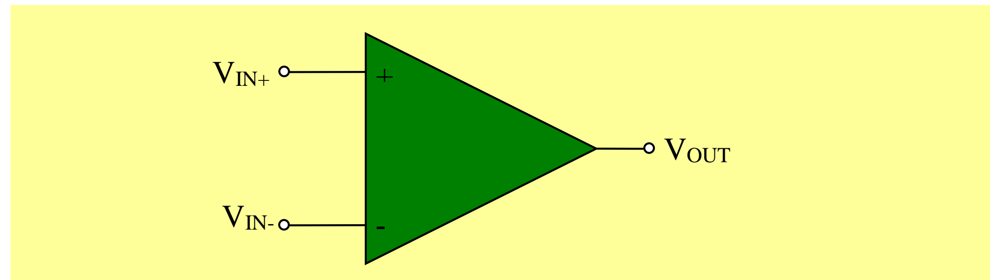


Фиг. 148 Програмен модел на АЦП модул

Отговорност на програмиста е да конфигурира тези регистри коректно.

17.2.8 Аналогов компаратор

Аналоговият компаратор е устройство с два аналогови входа и един цифров изход (Фиг.149).

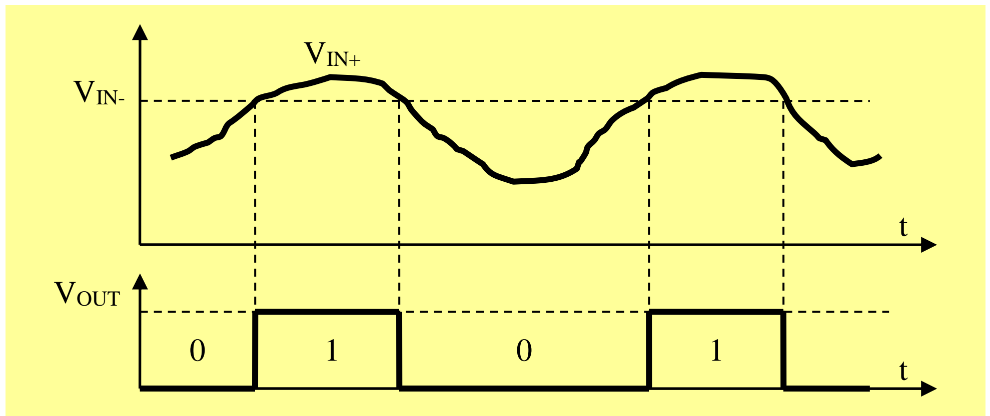


Фиг. 149 Аналогов компаратор

Напреженията подадени на аналоговите входове V_{IN+} и V_{IN-} се сравняват. Нивото на изхода V_{OUT} се променя съгласно следните условия:

$$V_{OUT} = \begin{cases} 1, \text{ ако } V_{IN+} > V_{IN-} \\ 0, \text{ ако } V_{IN+} < V_{IN-} \end{cases}$$

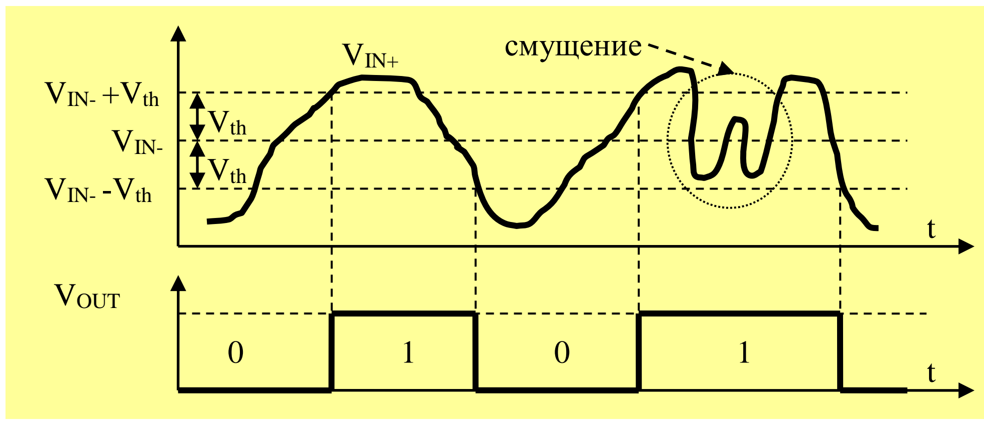
Фиг.150 описва принципа на работа на компаратора.



Фиг. 150 Принцип на работа на аналогов компаратор

Когато V_{IN+} надвиши V_{IN-}, изходът V_{OUT} се превключва в лог.1, а когато V_{IN+} падне под V_{IN-}, изходът V_{OUT} се превключва в лог.0.

За устойчивост срещу смущения повечето компараторите имат хистерезис. За целта се използва прагово напрежение V_{th}. Изходът се превключва в лог.1 само ако $V_{IN+} > V_{IN-} + V_{th}$, и се превключва в лог.0 само ако $V_{IN+} < V_{IN-} - V_{th}$ (Фиг.151).

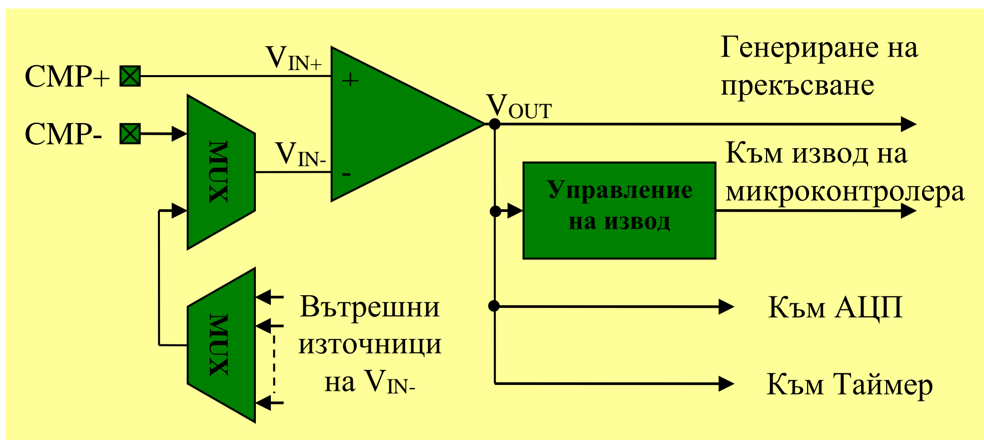


Фиг. 151 Принцип на работа на аналогов компаратор с хистерезис

Обърнете внимание на смущението от Фиг.151. Ако компараторът не използваше хистерезис (както на Фиг.150),

това смущение би предизвикало няколкократно превключване на изхода на компаратора.

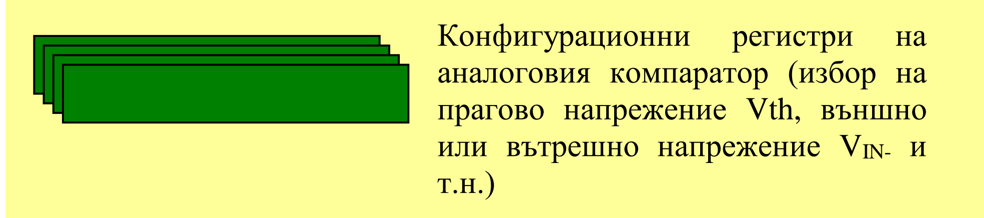
Следващата фигура показва функционалната блокова схема на компараторен модул на микроконтролер.



Фиг. 152 Функционална блокова схема на компараторен модул

При превключване на изхода се генерира прекъсване. Обикновено изходът на компаратора може да се изведе на извод на микроконтролера. Също така изходът на компаратора може да се използва като стартов сигнал за АЦП или тактов сигнал за таймер.

От гледна точка на един програмист компараторният модул може да се представи по следния начин (Фиг.153):



Фиг. 153 Програмен модел на компараторен модул

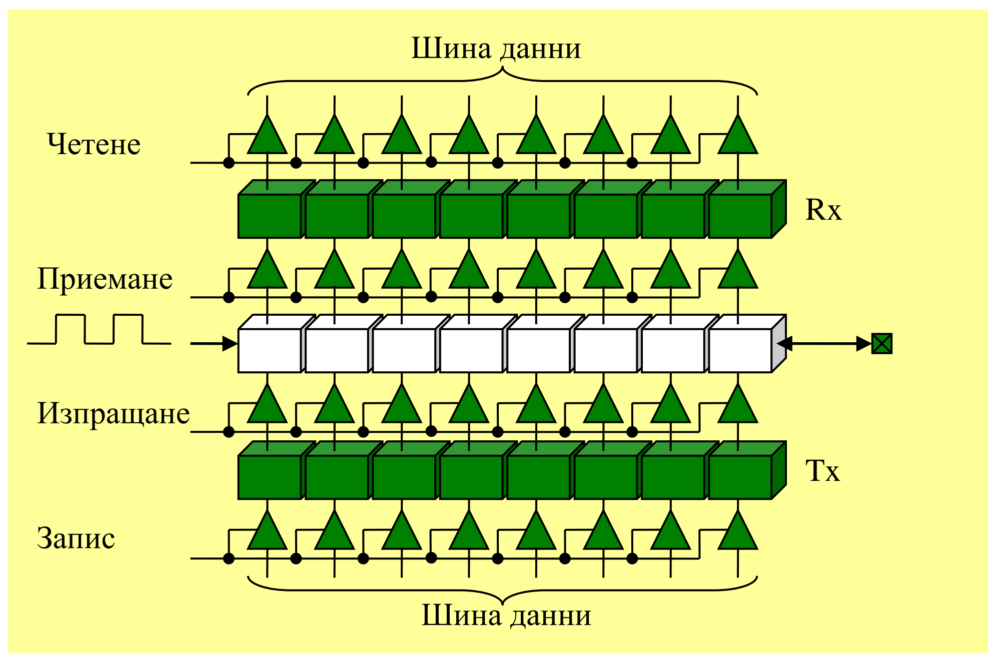
Отговорност на програмиста е да конфигурира тези регистри коректно.

17.2.9 Комуникационни интерфейси

Микроконтролерите предоставят различни комуникационни интерфейси. Най-разпространените комуникационни интерфейси са RS-232, SPI, I²C, LIN и CAN. Всички тези интерфейси са серийни, т.е. . предаването на данните става бит по бит. Описанието на тези интерфейси излиза извън обхвата на тази книга. Тук ще разгледам съвсем обобщено серийните комуникационни интерфейси. Комуникационните интерфейси могат да се разделят на:

- Полудуплексни

Докато се извършва предаване, не може да се извършва приемане и обратно. Следващата фигура показва обобщена структура на полудуплексен сериен комуникационен интерфейс.



Фиг. 154 Полудуплексен сериен комуникационен интерфейс

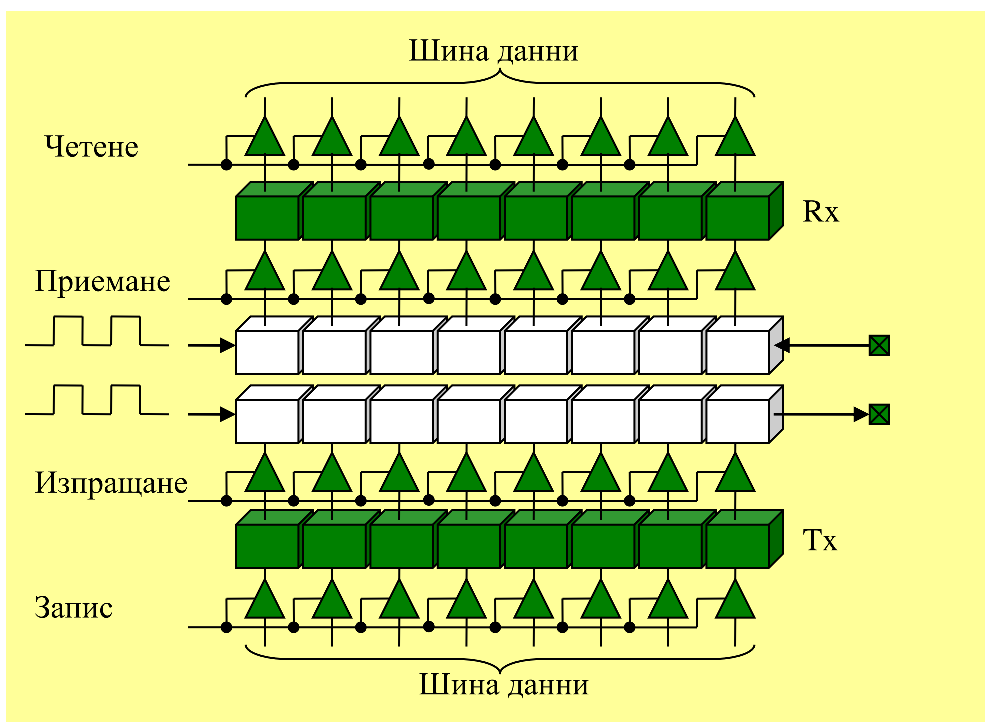
При изпращане данните постъпват от шината за данни в

регистър Tx. При стартиране на изпращането данните в регистър Tx се зареждат в приемо-предавателен преместващ регистър и започват да се изпращат бит по бит към извода на микроконтролера на всеки активен фронт на тактовия сигнал.

При приемане данните от извода на микроконтролера постъпват бит по бит в приемо-предавателния регистър. При приемане на всички битове съдържанието на приемо-предавателния регистър се копира в регистър Rx, откъдето може да бъде прочетено от потребителя.

- Дуплексни

При дуплексните комуникационни интерфейси е възможно едновременно приемане и предаване на данни. Следващата фигура показва обобщена структура на дуплексен сериен комуникационен интерфейс.



Фиг. 155 Дуплексен сериен комуникационен интерфейс

За целта се използват два отделни преместващи регистъра – един за приемане и един за предаване. Това позволява едновременно приемане и изпращане на данни.

17.2.10 Други

Различните микроконтролери могат да предоставят голямо разнообразие от периферни модули с най-различна функционалност.

18 Начално установяване на микроконтролера

Началното установяване (нарича се още системен ресет или просто ресет) на микроконтролера представлява извличане на ресет вектора от таблицата с векторите на прекъсване и зареждането му в Програмния Брояч. С други думи казано, ресетът води до рестартиране на микроконтролера и изпълняване на програмата отначало. След ресет обикновено битовете на RAM-паметта имат произволни стойности, докато повечето битове на управляващите регистри имат ясно дефинирани начални стойности, които са описани в документацията на микроконтролера. Източниците на ресет могат да бъдат различни. Типично микроконтролерът се установява в начално състояние от следните източници:

- Включване на захранването (**POR – Power-On Reset**);

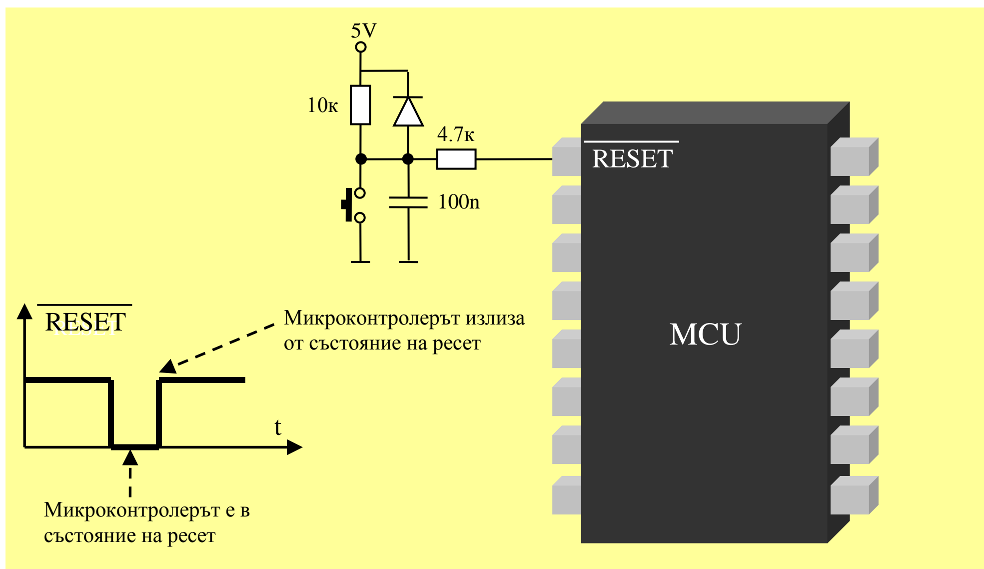
  Всеки микроконтролер се рестартира, когато захранващото напрежение V_{DD} надвиши определена прагова стойност V_{POR}.

- Пропадане на захранването (**BOR – Brown-Out Reset**);

  Когато захранващото напрежение V_{DD} падне под определено прагово напрежение V_{BOR}, се генерира системен ресет и микроконтролерът остава в това състояние, докато захранващото напрежение не надвиши отново напрежението V_{BOR}.

- Извод на микроконтролера

  Всеки микроконтролер има специален извод, чрез който микроконтролерът може да бъде рестартиран. Фиг.156 показва типичната схема на свързване на външните елементи към извода.



Фиг. 156 Начално установяване на микроконтролер от извод

При натискане на бутона на извод $\overline{\text{RESET}}$ се подава ниско ниво, което води до ресет на микроконтролера. При отпускане на бутона микроконтролерът излиза от състояние на ресет и стартира изпълнение на програмата, записана в Програмната Памет.

- Стражеви таймер

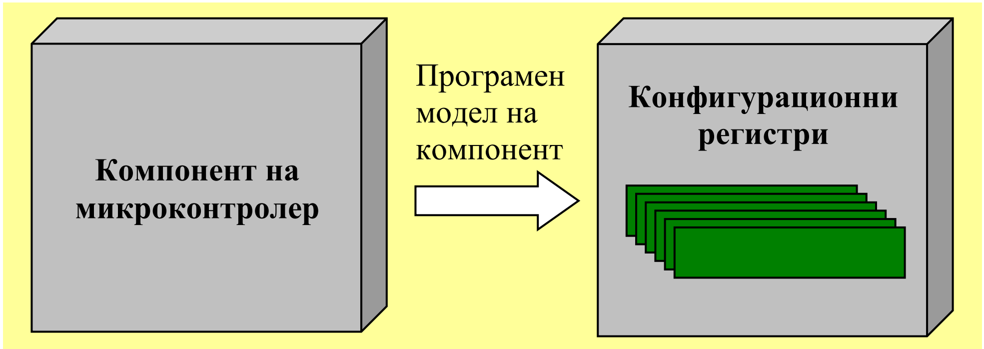
В точка 17.2.6 Стражеви таймер Ви обясних какво представлява този таймер. При препълване на таймера се генерира системен ресет.

Различните микроконтролери могат да предоставят и други източници на системен ресет.

Обикновено всеки микроконтролер предоставя регистър с флагове, които указват източника предизвикал системен ресет.

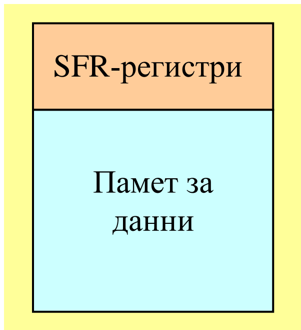
19 Заключение

В предходните две глави разгледахме какво представляват класическите компонентите на един микроконтролер и как работят. Различните производители могат да добавят допълнителни периферни модули с най-разнообразна функционалност. От гледна точка на един програмист всеки компонент на микроконтролера може да се разглежда като набор от конфигурационни регистри (Фиг.157).



Фиг. 157 Програмен модел на компонент на микроконтролер

Конфигурационните регистри обикновено използват същото адресно пространство като паметта за данни и се обозначават най-общо като SFRs (**Special Function Registers**) (Фиг.158).



Фиг. 158 SFR-регистри

Програмите, които се занимават с конфигурирането на SFR-

регистрите на микроконтролера, се наричат **драйвери**. Това са програми от най-ниско ниво. Независимо на какъв език се пишат тези програми (асемблер или C) те са тясно свързани с хардуера. Програмите, които се занимават с решаването на конкретна задача, се наричат **приложения**. Приложенията използват драйверите, за постигане на крайната цел, т.е. . те са програми от по-високо ниво (Фиг.159).

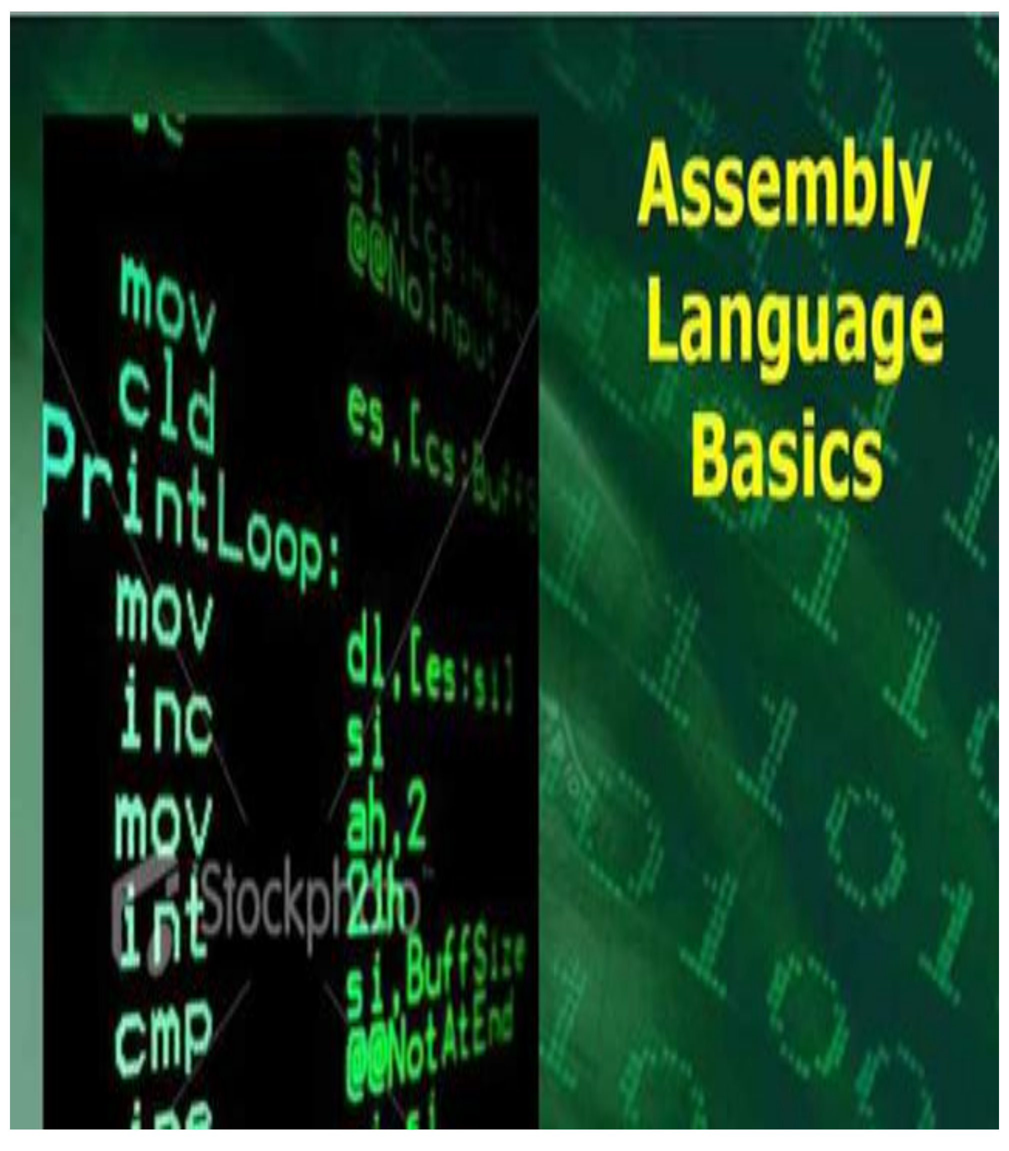
Фиг. 159 Архитектура на програмното осигуряване

Приложенията, написани на езика C, в една или друга степен са независими от архитектурата на микроконтролера. Това означава, че ако по някаква причина трябва да подмените микроконтролера с друг (с различна архитектура), ще трябва да пренапишете драйверите за новия микроконтролер и може да преизползвате същите приложение директно или с малки промени. Това е и едно от предимствата на езика C в сравнение с асемблерния език и се нарича **преносимост**.

В заключение искам просто да Ви обърна внимание, че описаните в тази част устройство и принцип на работа на микроконтролерите са обобщени. Не бива да очаквате, че описаното тук ще отговаря точно на устройството и принципа на работа на конкретен микроконтролер. Познаването обаче на базовите принципи ще Ви помогне по-лесно да се ориентирате в различните архитектури на микроконтролерите.

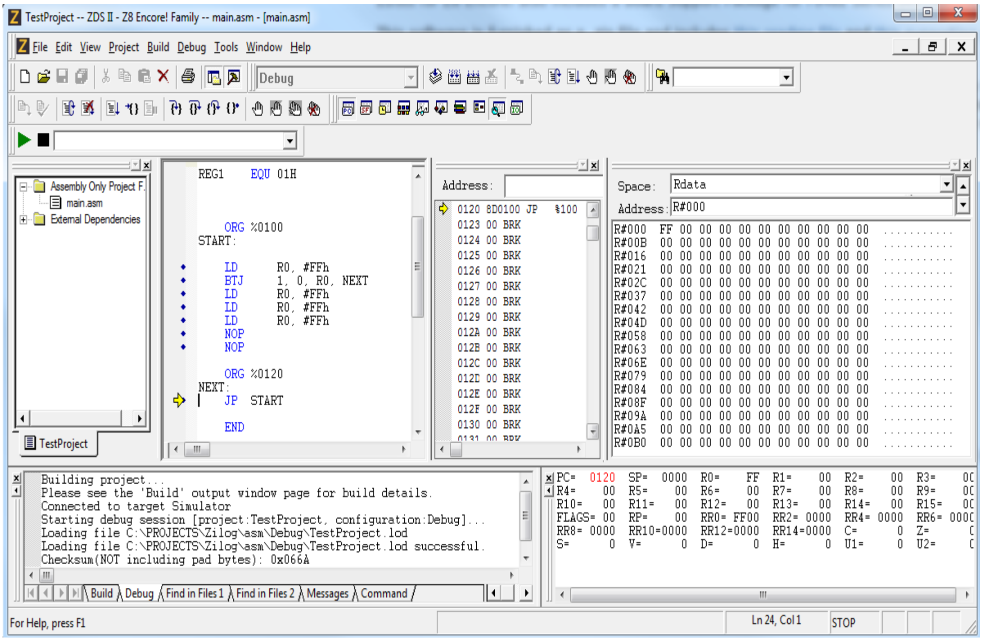
Част IV

Програмиране на микроконтролери на асемблер



20 Въведение

Асемблерният език е езикът от най-ниско ниво, използван за програмиране на микроконтролерите. По мое мнение всеки начинаещ програмист на микроконтролери трябва да се запознае с програмирането на асемблерен език преди да премине към език от по-високо ниво какъвто е С. В тази част ще Ви запозная с микроконтролера **ZF083A**, част от фамилията **Z8Encore!ZF083A** на фирмата ZiLOG. Дълго обмислях на коя архитектура да се спра. Има голямо разнообразие от архитектури, но повечето не са подходящи за начинаещи. И така, докато един ден във форума http://www.mcu-bg.com един от участниците с прозвище **Н'бабане Гт'муан'га** ми подхвърли идеята за фамилията микроконтролери **Z8Encore!** на фирмата ZiLOG. Предимствата на тази фамилия микроконтролери е тяхната простота, напълно безплатна среда за програмиране **Zilog Developer Studio II** (включително и на езика С без никакви ограничения в размера и оптимизацията на кода),



Фиг. 160 Zilog Developer Studio II

евтин дебъгер **USB Smart Cable**.

Фиг. 161 USB Smart Cable

Ето защо реших, че това е микроконтролерът, отговарящ най-близко на моите изисквания. Благодарности на **Н'бабане Гт'муан'га** за идеята ☺.

Ако сте абсолютно начинаещ, силно Ви препоръчвам да отделите време (според мен между 4 и 6 седмици, ако четете редовно) и да изучите материала описан в тази част. Целта е не толкова да изучите дадената архитектура (има десетки архитектури, едва ли е възможно да ги изучите всичките), а по-скоро да добиете практическа представа как работи един микроконтролер. Така ще си изградите твърди основи, на които да стъпите, и които ще продължите да надграждате във Вашия бъдещ професионален път като програмисти на микроконтролери.

В основата на фамилията микроконтролери **Z8Encore!** стои процесорът **eZ8**. В следващата точка ще Ви запозная с архитектурата и възможностите на този процесор.

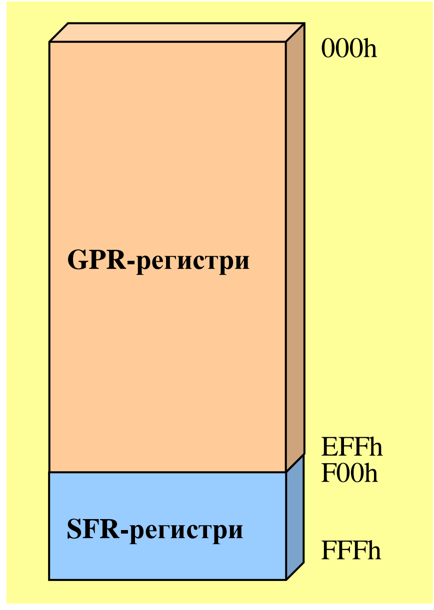
21 Описание на процесора eZ8

21.1 Организация на паметта

Процесорът **eZ8** може да осъществява достъп до три различни типа памети. Всяка памет има свое отделно адресно пространство.

- Регистрова Памет (РП)

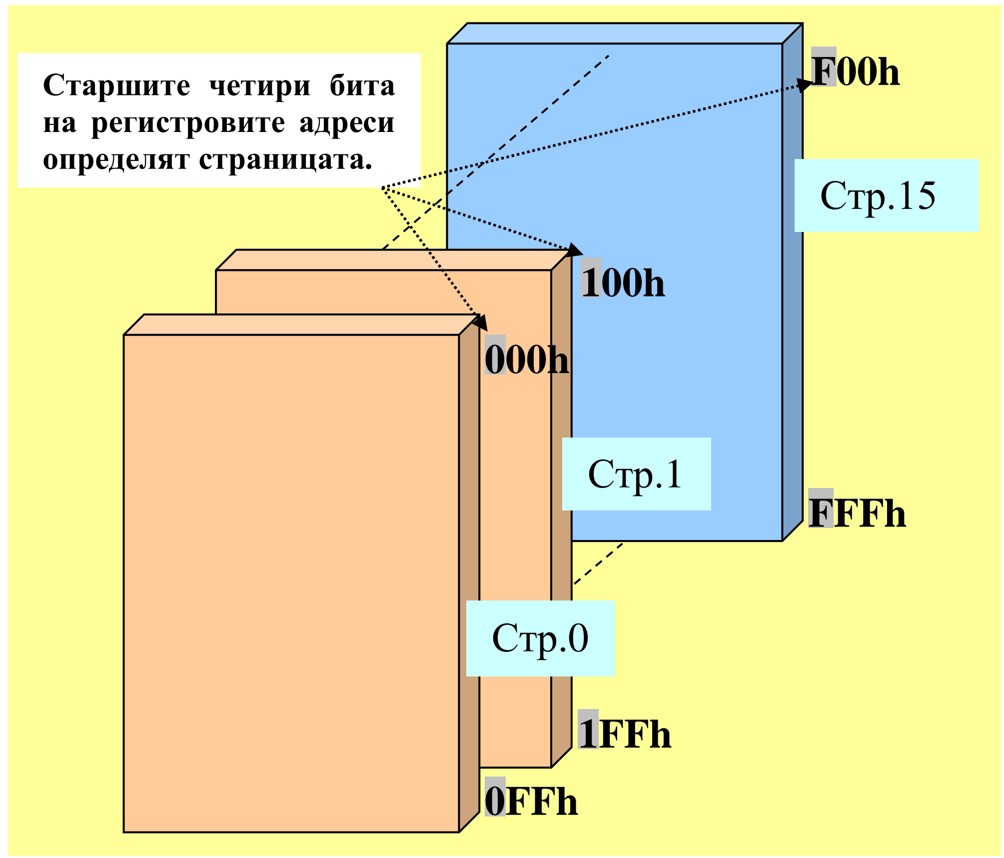
Регистровата Памет се състои от два типа регистри: регистри за обща употреба (**GPR – General Purpose Register**) и управляващи регистри, които се наричат още регистри със специална функция (**SFR – Special Function register**). GPR-регистрите могат да се използват като акумулатори, адресни указатели, индексни регистри, за стек или временно съхранение на данни. SFR-регистрите служат за управление на микроконтролера. Процесорът **eZ8** може да адресира до 4096 байта (4kB) регистри. Фиг.162 показва организацията на РП.



Фиг. 162 Регистрова Памет

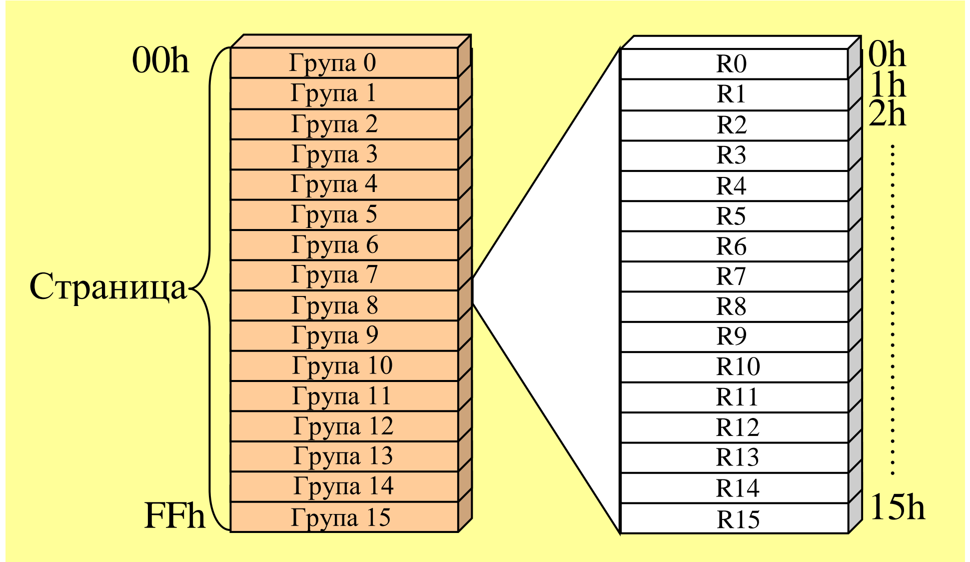
Както се вижда от фигурата, SFR-регистрите заемат последните 256 байта от Регистровата Памет.

Регистровата Памет се разделя логически на 16 страници, всяка състояща се от 256 регистри (Фиг.163).



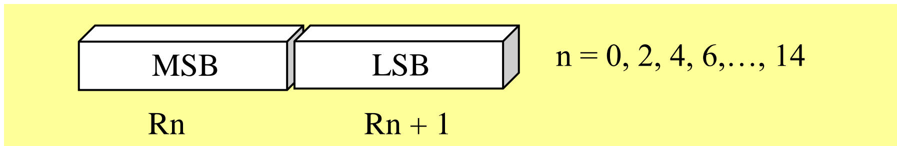
Фиг. 163 Страниране на Регистровата Памет

Всяка страница от своя страна се разделя на 16 групи, всяка състояща се от 16 регистъра, наричани още **работни регистри**. Регистрите във всяка група се обозначават като R0, R1, R2,..., R15 (Фиг.164).



Фиг. 164 Разделяне на страница на 16 групи по 16 регистъра

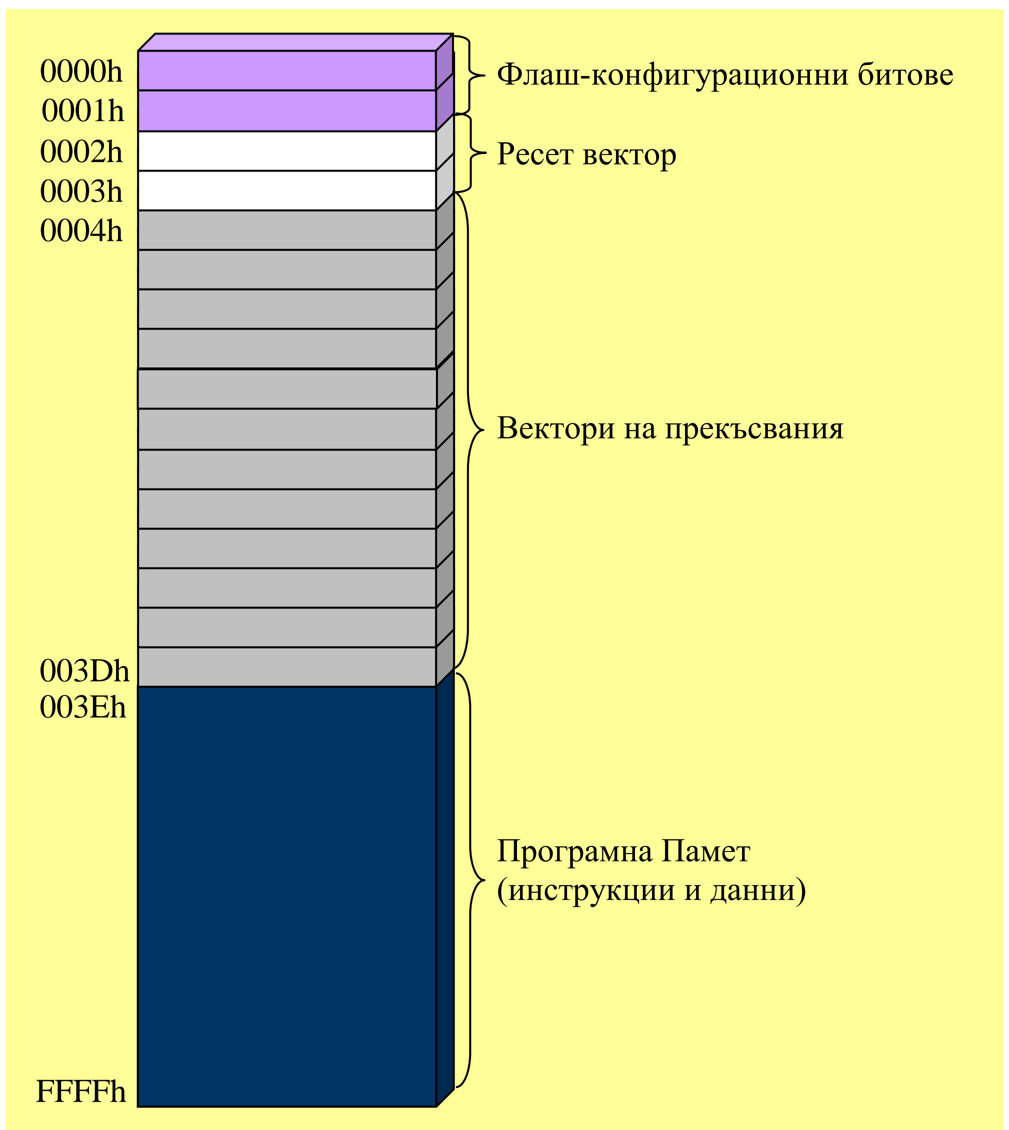
Работните регистри могат да се адресират и като 16-битови думи (двойка работни регистри). За целта два съседни регистъра се свързват, за да формират 16 битова дума (Фиг.165).



Фиг. 165 16 битова регистрова дума

- Програмна Памет (ПП)

Процесорът **eZ8** може да адресира 65 536 байта (64 kB) Програмна Памет. Програмната Памет се използва за съхранение на инструкциите на програмата и за съхранение на данни. Следващата фигура показва организацията на ПП. Първите два байта са заети от флаш-конфигурационни битове (вижте 25.15 Флаш-конфигурационни битове), след тях е разположена таблицата с векторите на прекъсване, и най-накрая паметта за инструкциите и данни.



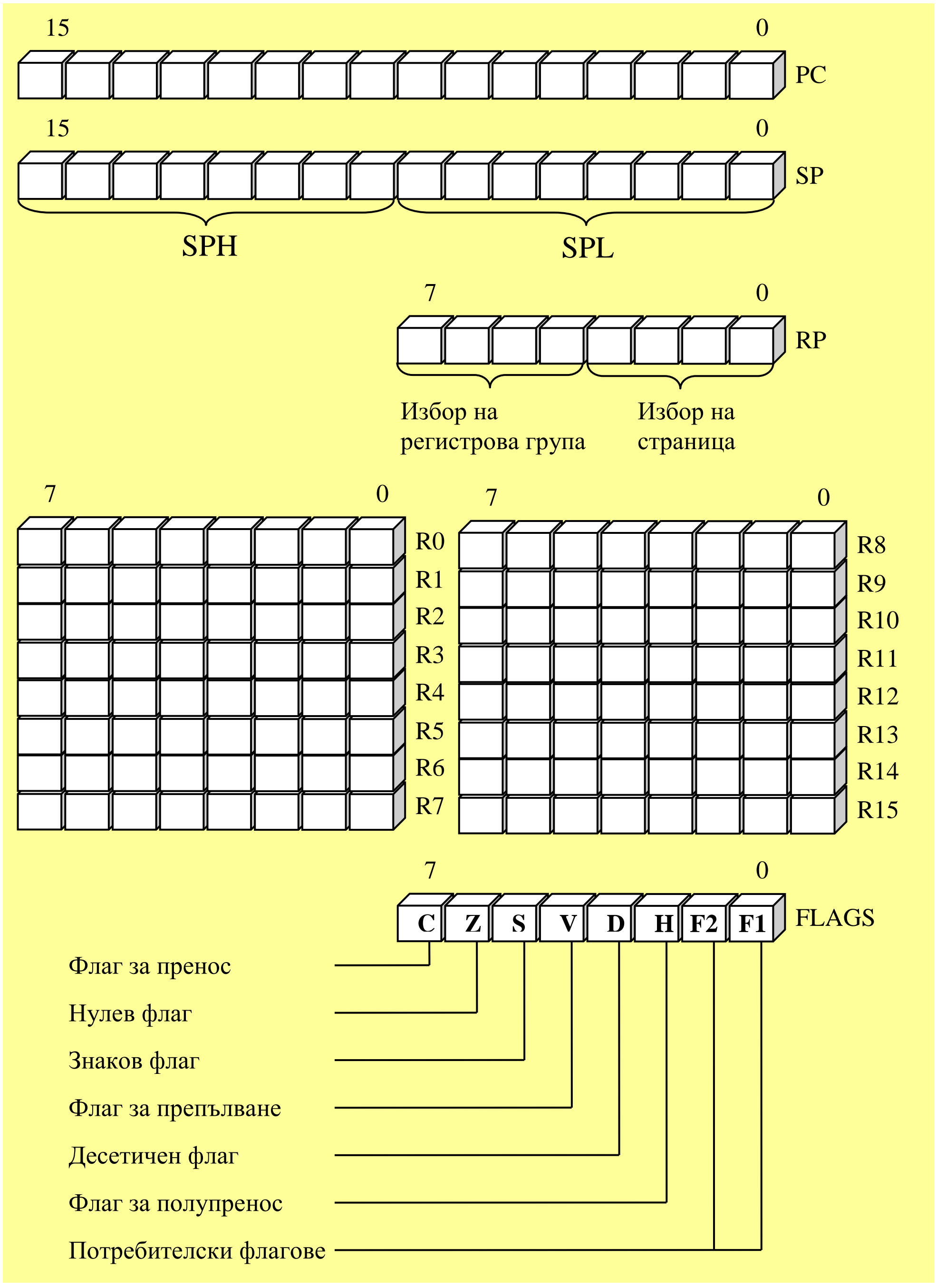
Фиг. 166 Програмна Памет

- Памет за Данни (ПД)

Процесорът **eZ8** също така може да адресира и 65 536 байта (64 kB) Памет за Данни. Достъпът до нея става с помощта на инструкциите **LDE** и **LDEI**. Тази памет заема адресно пространство с адреси 0000h ÷ FFFFh.

21.2 Програмен модел

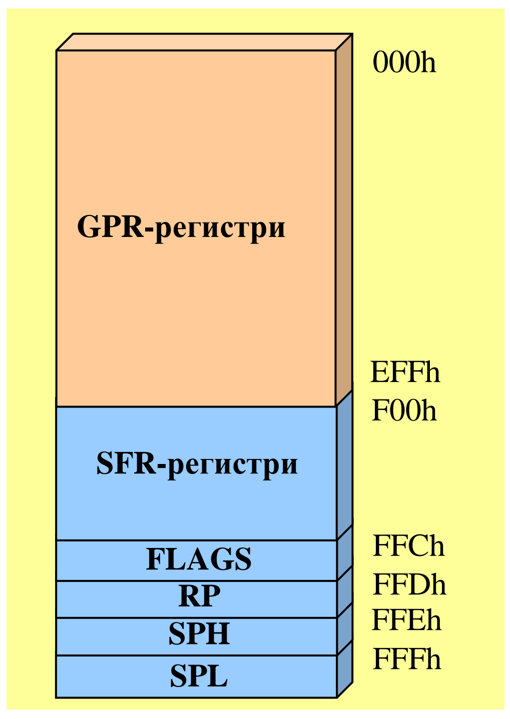
Фиг.167 описва програмния модел на процесора eZ8.



Фиг. 167 Програмен модел на процесора eZ8

Програмният Брояч PC (**Program Counter**) е 16-битов и съдържа адреса в Програмната Памет на поредната инструкция за изпълнение. Указателят на Стека SP (**Stack Pointer**) също е 16-битов и сочи към последната записана стойност в стека. SP се състои от два 8-битови регистъра: SPH и SPL. Указателят към Регистрите RP (**Register Pointer**) е 8 битов и се използва за избор на страница (RP[3:0]) и регистрова група (RP[7:4]) от Регистровата Памет. R0÷R15 е текущата избрана регистрова група. Регистърът на Флаговете FLAGS е 8-битов и съдържа флагове, указващи различни ситуации (като препълване при сумиране, нулев резултат и др.) от последната изпълнена от процесора инструкция.

С изключение на Програмния Брояч всички останали регистри за управление на процесора са част от Регистровата Памет и винаги се разполагат на едни и същи адреси (Фиг.168).



Фиг. 168 Процесорни регистри

Флаг за пренос C

Флагът за пренос C е 1 когато резултатът от аритметична операция генерира пренос от или заем в бит 7 на данните. В противен случай е 0. Инструкциите за ротация **RRC**/**RLC** и преместване **SRA**/**SRL** също влияят върху флага за пренос. Освен тези инструкции има три инструкции, които директно могат да променят стойността на този флаг: **CCF**/**RCF**/**SCF**.

Нулев Флаг Z

Ако резултатът от аритметична или логическа операция е нула, този флаг е 1, в противен случай е 0.

Знаков флаг S

Знаковият флаг S съдържа стойността на най-старшия бит на резултата от аритметична или логическа операция. Процесорът eZ8 използва допълнителен код **(2's complement**) за представяне на знакови числа. При този код ако най-старшия бит на числото е 0, то е положително (S = 0), а ако е 1, то е отрицателно (S = 1).

Флаг за препълване V

Флагът за препълване V е 1, ако резултатът от знакови аритметични, за преместване и ротация операции е по-голям от максималната възможна стойност (> 127) или по-малък от минималната възможна стойност (< -128), която може да се представи от 8-битово число в допълнителен код, т.е. . резултатът е извън диапазона [-128 ÷ 127]. В противен случай V = 0.

Десетичен флаг D

Флагът D се използва за BCD (**Binary-Coded Decimal**) аритметични операции. Тъй като алгоритъмът за коригиране

на BCD операции е различен за събиране и изваждане, този флаг специфицира типа на последната изпълнена инструкция, позволявайки последваща операция за десетична настройка. Обикновено този флаг не се използва като тест-условие. След изваждане D = 1. След сумиране D = 0.

Флаг за полупренос H

Флагът за полупренос H е 1, когато операция събиране генерира пренос от бит 3 или операция изваждане генерира заем в бит 4. Инструкцията **DA** използва този бит, за да преобразува резултата от предходна операция за събиране или изваждане в правилен BCD резултат. Подобно на флаг D, този флаг не се използва директно от потребителите.

Потребителски флагове F1/F2

Тези флагове не се използват от процесора и могат да се използват от потребителя свободно за собствени цели.

Тест-условия

Флаговете C, Z, S и V управляват работата на инструкциите за преход **JP cc** и **JR cc**. Табл.28 описва всички тест-условия **cc**, използвани от тези инструкции.

cc	описание	тест-условие
F	Винаги ЛЪЖА (FALSE)	-
LT	По-малко от	(S XOR V) = 1
LE	По-малко от или равно	(Z OR (S XOR V)) = 1
ULE	Беззнаково по-малко от или равно	(C OR Z) = 1
OV	Препълване	V = 1
MI	Минус	S = 1
Z	Нула	Z = 1
EQ	Равно	Z = 1

C	Пренос	C = 1
ULT	Беззнаково по-малко от	C = 1
T	Винаги ИСТИНА (TRUE)	-
GE	По-голямо или равно	(S XOR V) = 0
GT	По-голямо от	(Z OR (S XOR V)) = 0
UGT	Беззнаково по-голямо от	(C = 0 AND Z = 0)
NOV	Няма препълване	V = 0
PL	Плюс	S = 0
NZ	Различно от нула	Z = 0
NE	Не е равно	Z = 0
NC	Няма пренос	C = 0
UGE	Беззнаково по-голямо или равно	C = 0

Табл. 28 Тест-условия

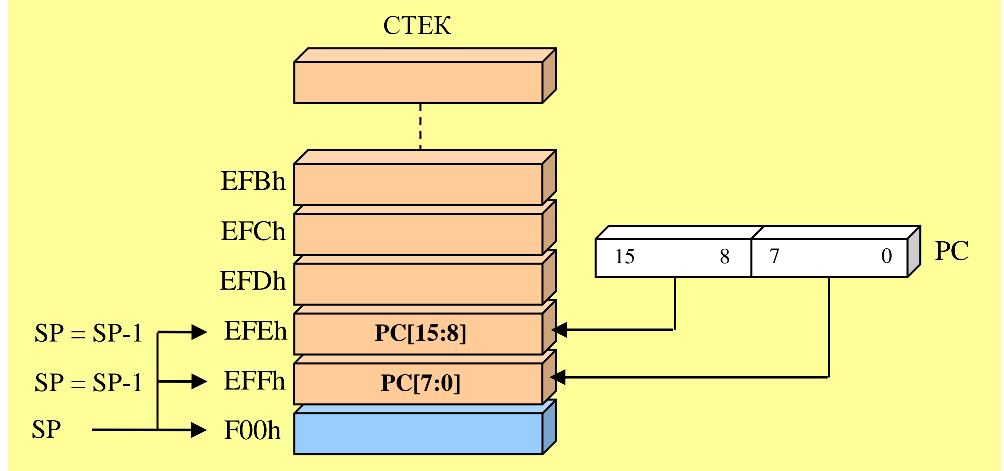
В 22 Описание на инструкциите ще намерите подробна информация за влиянието на инструкциите върху всички тези флагове.

21.3 Стек

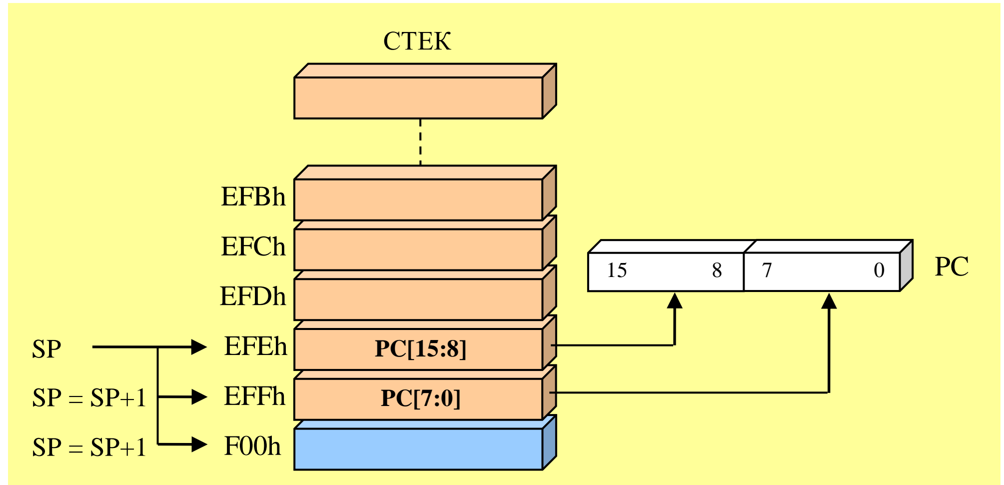
Стекът в eZ8 се разполага в Регистровата Памет. Указателят на стека е 16-битов и се състои от регистрите SPH (старши байт на SP) и SPL (младши байт на SP), т.е. . SP = SPH:SPL. SP сочи винаги към последно записаната стойност в стека и се изменя по механизма предекремент-постинкремент (при запис SP първо се намалява и след това стойността се записва в стека, при четене стека се прочита и след това SP се увеличава).

При извикване на инструкцията CALL (извикване на подпрограма) съдържанието на PC се съхранява в стека автоматично. При изпълнение на инструкцията RET (връщане от подпрограма) съхранената в стека стойност се

връща обратно в PC. Следващите фигури илюстрират този процес. Приема се, че стекът е разположен в края Регистровата Памет, заемана от GPR-регистрите, и указателят на стека SP сочи към първия SFR-регистър. Това означава, че първата записана стойност в стека ще се разположи в последния GPR-регистър (адрес EFFh).



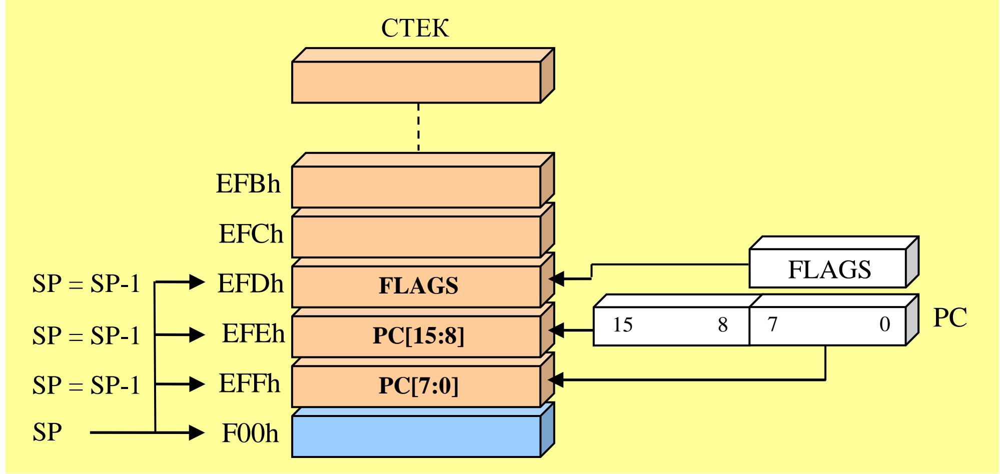
Фиг. 169 Стек и инструкцията CALL



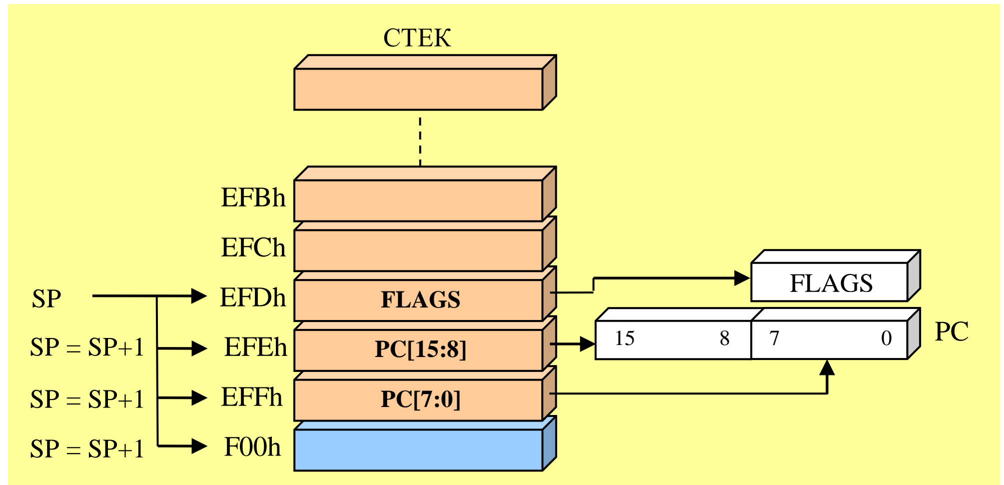
Фиг. 170 Стек и инструкцията RET

При възникване на прекъсване освен PC в стека се съхранява и Регистъра на Флаговете FLAGS. Изпълнението на

инструкцията IRET (връщане от прекъсване) възстановява стойностите на тези регистри обратно. Следващите фигури илюстрират този процес. Приема се, че стекът е разположен в края Регистровата Памет, заемана от GPR-регистрите и указателят на стека SP сочи към първия SFR-регистър. Това означава, че първата записана стойност в стека ще се разположи в последния GPR-регистър (адрес EFFh).



Фиг. 171 Стек и възникване на прекъсване



Фиг. 172 Стек и инструкцията IRET

Процесорът eZ8 поддържа и две инструкции за работа със

стека: PUSH (запис в стека) и POP (четене от стека). Тези инструкции позволяват в стека да се съхраняват и потребителски данни.

21.4 Прекъсвания

21.4.1 Общи сведения

Прекъсванията позволяват на периферните модули на микроконтролера да прекъсват текущо изпълняваната от процесора програма и да го превключват към изпълнение на подпрограма, обслужваща това прекъсване (**ISR – Interrupt Service Routine**). Когато тази подпрограма завърши изпълнението си, процесорът отново се връща към изпълнение на инструкциите на прекъснатата програма. Всички прекъсвания в eZ8 могат да бъдат разрешени с инструкцията EI (установява в 1 глобалния разрешаващ бит IRQE в регистър IRQCTL) и да бъдат забранени с инструкцията DI (нулира глобалния разрешаващ бит IRQE в регистър IRQCTL). Също така източниците на прекъсване имат приоритет, който определя кое прекъсване ще се обработи първо в случай, че възникнат няколко прекъсвания едновременно. Броят на източниците на прекъсване е различен за различните микроконтролери на фамилията **eZ8 Encore!**. Този въпрос, както и приоритетите на източниците на прекъсвания, ще бъде разгледан по-надолу, когато разгледаме микроконтролера ZF083A.

21.4.2 Обработка на разрешено прекъсване

Всеки източник на прекъсване има собствен адрес в Програмната Памет, на който се съхранява **векторът на прекъсването**. Векторът на прекъсването[1] заема два байта и

представлява адреса на подпрограмата, обработваща това прекъсване.

Забележка[1]: Отговорност на програмиста е да запише адреса (вектора на прекъсване) на дадена ISR-подпрограма на съответния адрес в Програмната Памет.

Когато едно прекъсване възникне, се извършват следните действия:

1. Бит IRQE се нулира (всички прекъсвания се забраняват).

2. PC[7:0] се записва в стека;

3. PC[15:8] се записва в стека;

4. FLAGS се записва в стека;

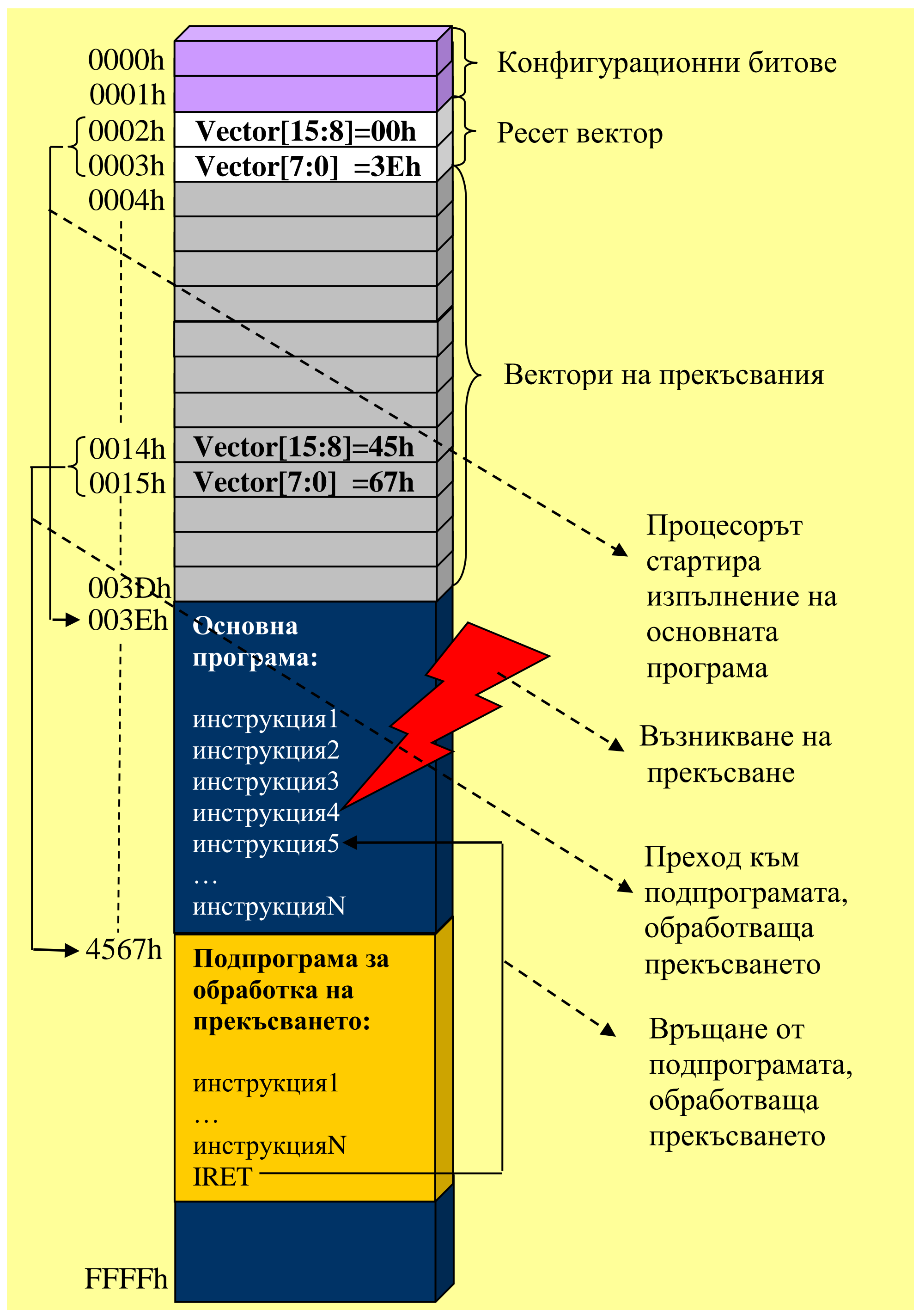
5. Старшият байт на вектора на прекъсване се записва в PC[15:8];

6. Младшият байт на вектора на прекъсване се записва в PC[7:0];

   След стъпки 4 и 5 PC сочи към подпрограмата, обработваща прекъсването.

7. Извършва се преход към подпрограмата, обработваща прекъсването.

Следващата фигура илюстрира обработката на прекъсване. Приема се, че векторът на прекъсване се съхранява на адреси 0014h и 0015h в Програмната Памет.



Фиг. 173 Обработка на прекъсване

Фиг.173 описва следната ситуация: ресет векторът съдържа

адреса на основната програма и процесорът изпълнява нейните инструкции. По време на изпълнение на **инструкция4** възниква прекъсване. След като процесорът завърши изпълнението на тази инструкция, съдържанието на PC (който вече сочи към **инструкция5**), се съхранява в стека. Съдържанието на FLAGS също се съхранява в стека и в PC се зарежда адреса на подпрограмата, обработваща това прекъсване. В нашия пример този вектор (4567h) е разположен на адреси 0014h и 0015h. След това процесорът преминава към изпълнение на инструкциите на ISR-подпрограмата. При достигане на инструкцията IRET (връщане от прекъсване), регистрите PC и FLAGS възстановяват стойностите си от стека и процесорът продължава с изпълнението на следващата инструкция от основната програма (**инструкция5**).

21.4.3 Обработка на неразрешено прекъсване

Флаговете на прекъсване се установяват в 1 при всяко генериране на прекъсване, независимо дали прекъсването е разрешено или забранено. Ако едно прекъсване е забранено, Вие може периодично да проверявате съответния флаг на прекъсване дали е установен в 1 и да извършите извикване на подпрограма, която обработва това прекъсване.

```
        …
        TM      IRQ1, #00010000b ;Проверка на флаг 4 в IRQ1
        JP      Z, NEXT ;Ако флагът е 0, прескочи следващата
                        ;инструкция
        CALL    InterruptHandler ;Извикай подпрограмата, която
                                 ;обработва това прекъсване
NEXT:
        …
InterruptHandler:
        …
        RET ;Връщане от подпрограмата
```

21.4.4 Вложени прекъсвания

Когато процесорът стартира обработка на прекъсване, всички останали прекъсвания се забраняват (бит IRQE се нулира). Това означава, че обработката на високоприоритетно прекъсване може да бъде забавена от ниско приоритетно прекъсване, текущо обработвано от процесора. Този проблем може да се разреши най-лесно ако в подпрограмата, обработваща ниско приоритетното прекъсване, разрешите отново прекъсванията, като установите в 1 бит IRQE.

```
InterruptHandler:
        EI ;Разрешаване на прекъсванията
        …
        DI ;Забраняване на прекъсванията
        RET ;Връщане от подпрограмата
```

Този подход има и един съществен недостатък, а именно, че и по-ниско приоритетни прекъсвания ще могат да прекъсват подпрограмата. Този недостатък може да се избегне, ако изпълните следните стъпки в подпрограмата:

1. Запишете регистрите IRQCTL (Interrupt Control), IRQnENH (Interrupt Enable High, n = 0 ÷ 2) и IRQnENL (Interrupt Enable Low, n = 0 ÷ 2) в стека.

2. Конфигурирайте регистрите IRQnENH и IRQnENL, така че да забраните прекъсванията с по-нисък приоритет.

3. Изпълнете инструкцията EI, за да разрешите прекъсванията.

4. Продължете с обработката на текущото ниско приоритетно прекъсване.

5. Възстановете обратно регистрите IRQCTL, IRQnENH и

IRQnENL от стека.

6. Изпълнете инструкцията DI, за да забраните прекъсванията.

7. Изпълнете инструкцията IRET, за да излезете от подпрограмата.

```
InterruptHandler:
        ;Стъпка 1
        PUSH   IRQCTL
        PUSH   IRQ0ENH
        PUSH   IRQ1ENH
        PUSH   IRQ2ENH
        PUSH   IRQ0ENL
        PUSH   IRQ1ENL
        PUSH   IRQ2ENL

        ;Стъпка 2
        LDX   IRQ0ENH, # НоваСтойност;
        LDX   IRQ1ENH, #НоваСтойност;
        LDX   IRQ2ENH, #НоваСтойност;
        LDX   IRQ0ENL, #НоваСтойност;
        LDX   IRQ1ENL, #НоваСтойност;
        LDX   IRQ2ENL, #НоваСтойност;

        ;Стъпка 3
        EI ;Разрешаване на прекъсванията

        ;Стъпка 4
        ...
        ; Кода на подпрограмата се разполага тук
        ...
        ;Стъпка 5
        POP   IRQ2ENL
        POP   IRQ1ENL
        POP   IRQ0ENL
        POP   IRQ2ENH
        POP   IRQ1ENH
        POP   IRQ0ENH
        POP   IRQCTL
```

```
;Стъпка 6
DI ;Забраняване на прекъсванията

;Стъпка 7
RET ;Връщане от подпрограмата
```

Ако изпълните тези стъпки, само прекъсвания с по-висок приоритет ще могат да прекъсват обработката на текущото по-ниско приоритетно прекъсване.

Не се безпокойте, ако инструкциите и имената на регистрите не Ви говорят нищо. Когато разгледаме описанието на инструкциите (22 Описание на инструкциите на eZ8) и контролерът на прекъсванията на микроконтролера ZF083A (25.6 Контролер на прекъсванията), всичко ще Ви се изясни напълно.

21.4.5 Генериране на софтуерни прекъсвания

Флаговете на прекъсване могат да бъдат установявани в 1 и от софтуера с помощта на инструкции. Контролерът на прекъсванията обработва софтуерните прекъсвания по същия начин както и хардуерните.

Пример:

```
OR      IRQ1, #00010000b ; IRQ1[4] = 1
```

Тази инструкция установява в 1 бит 4 на регистър IRQ1. Ако прекъсванията са разрешени и няма друго по-високоприоритетно прекъсване, процесорът ще стартира процедура по обработка на това прекъсване.

21.4.6 Аварийни прекъсвания

Аварийното прекъсване (**trap**) е вид прекъсване, което се генерира само в определени аварийни ситуации, т.е. . то не трябва да се генерира при нормално изпълнение на програмата. Например микроконтролерът ZF083A може да генерира аварийно прекъсване при възникване на една от следните ситуации:

- Изпълнение на грешна инструкция;
- Грешка в основния генератор на тактовия сигнал;
- Грешка в генератора на WDT-таймера.

Процесорът обработва аварийните прекъсвания по същия начин както и обичайните прекъсвания. Аварийното прекъсване, генерирано от изпълнение на грешна инструкция, обаче има следните особености: процесорът съхранява регистър FLAGS и PC в стека както и при обичайните прекъсванията, с тази разлика, че PC не се увеличава, а остава да сочи към невалидната инструкция. След това векторът на аварийното прекъсване от грешна инструкция се зарежда в PC.

	Подпрограмата, обработваща аварийното прекъсване от изпълнение на грешна инструкция, не трябва да завършва с инструкцията IRET. Причината е, че PC ще се зареди отново с адреса на невалидната инструкция, което на свой ред ще предизвика повторно генериране на същото аварийно прекъсване.

21.5 Адресни режими

Адресният режим представлява начина, по който дадена инструкция на процесора формира адреса на регистър от Регистровата Памет или адрес в Програмната Памет. Процесорът eZ8 използва следните адресни режими за адресиране на Регистровата Памет:

- **Регистрова адресация (ER, R, RR, r, rr)**
  - 12-битова (разширена) регистрова адресация (ER)
  - 8-битова регистрова адресация (R)
  - 8-битова регистрова адресация на двойка регистри (RR)
  - 4-битова регистрова адресация на работен регистър (r)
  - 4-битова регистрова адресация на двойка работни регистри (rr)
- **Косвена адресация (IR, IRR, Ir, Irr)**
  - Косвена адресация на регистър (IR)
  - Косвена адресация на двойка регистри (IRR)
  - Косвена адресация на работен регистър (r)
  - Косвена адресация на двойка работни регистри (Irr)
- **Индексна адресация (X)**

и следните адресни режими за достъп до Програмната Памет:

- **Директна адресация (DA)**

- **Относителна адресация (RA)**

Съществува още един вид адресация, наречена непосредствена адресация на данни (**IM**). При тази адресация използвания операнд се явяват 8-битови данни, които са част от самата инструкция

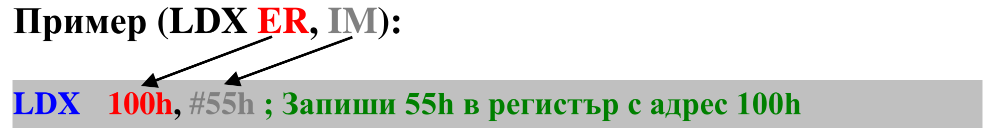
Адресните режими ER, R, RR, IR и IRR, при които операндът е 12- или 8-битов адрес на регистър, могат да използват т.нар. **Escaped-адресация**.

Следващите подточки описват всички тези адресни режими подробно. Ако сте абсолютно начинаещ, поднесената информацията може да Ви затрудни на моменти и да Ви се стори, че нещата изглеждат по-сложни, отколкото са в действителност. Ако стигнете до подобно заключение, това изобщо не бива да Ви отчайва. Просто продължете да четете книгата, докато стигнете до описанието на самите инструкции, поддържани от процесора eZ8. Докато четете описанието на дадена инструкция, връщайте се периодично към описанието на адресните режими. Така постепенно нещата ще Ви се изяснят, което, съм убеден, ще Ви направи много доволни от себе си ☺.

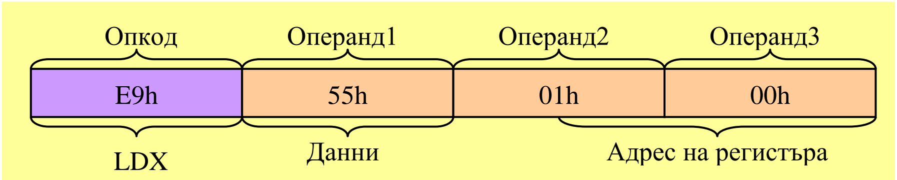
21.5.1 Регистрова адресация

12-битова регистрова адресация (ER)

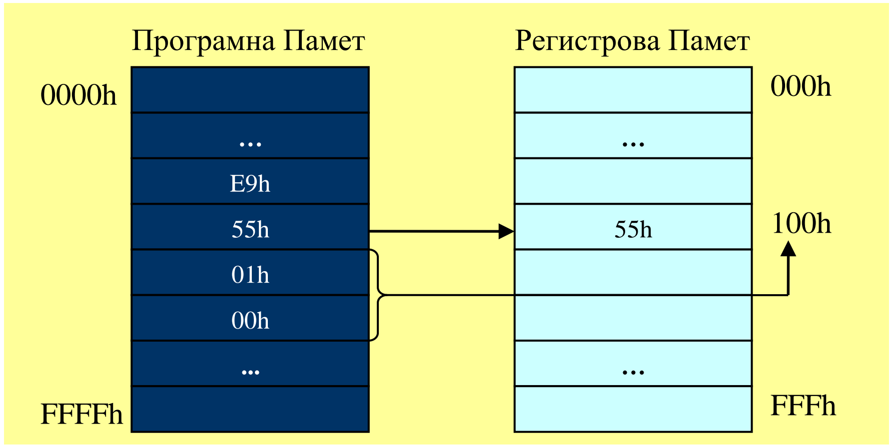
При тази адресация инструкцията съдържа пълния 12-битов адрес на регистър от Регистровата Памет.



Фиг.174 и 175 показват вътрешната структура и разполагането в Програмната Памет на тази инструкция.



Фиг. 174 Структура на инструкцията LDX 100h, #55h



Фиг. 175 LDX 100h, #55h

Както се вижда от фигурите, пълният 12-битов дрес се съдържа в самата инструкция (старшите четири бита 11:8 на адреса са разположени в младшия полубайт на операнд2, а младшите осем бита 7:0 са разположени в операнд3).

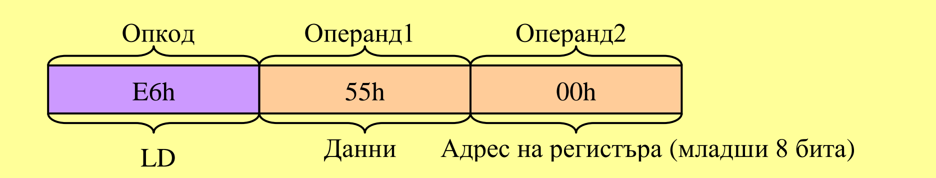
8-битова регистрова адресация (R, RR)

При тази адресация инструкцията съдържа младшите 8 бита на адреса на регистър или двойка регистри от Регистровата Памет. Старшите 4 бита се съдържат в битове RP[3:0]. Казано по друг начин, тази адресация позволява директно адресиране на регистрите в текущата страница от Регистровата Памет. Текущата страница се определя от битовете RP[3:0].

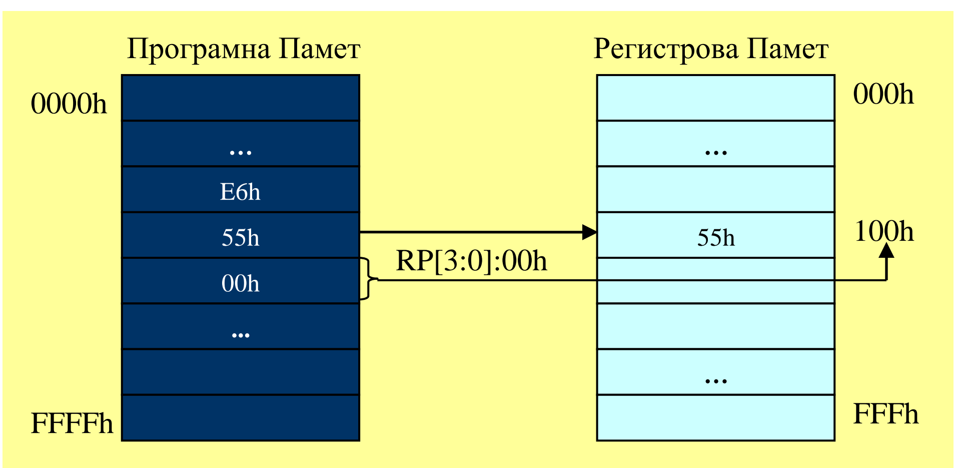
Пример (LD R, IM): Приема се, че RP[3:0] = 1h

```
LD    00h, #55h ;Запиши 55h в регистър с адрес 00h на текущата
                ;страница
```

Фиг.176 и 177 показват вътрешната структура и разполагането в програмната памет на тази инструкция.



Фиг. 176 Структура на инструкцията LD 00h, #55h

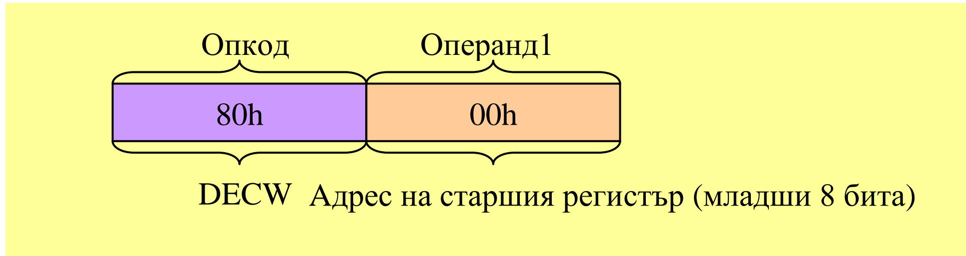


Фиг. 177 LD 00h, #55h (RP[3:0] = 1h)

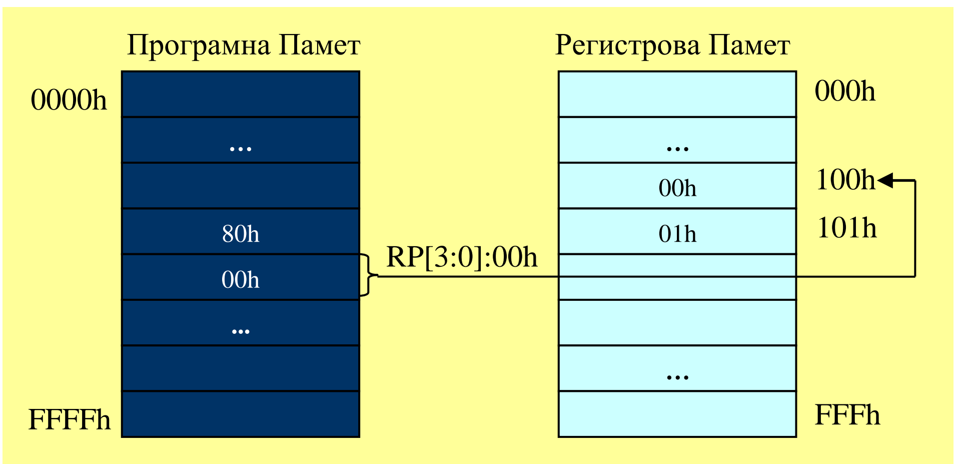
Пример (DECW RR): Приема се, че RP[3:0] = 1h, регистър 100h = 00h и регистър 101h = 02h

```
DECW        00h ; Намали съдържанието на двойката работни
                ; регистри 00h:01h с 1
```

Фиг.178 и 179 показват вътрешната структура и разполагането в Програмната Памет на тази инструкция.



Фиг. 178 Структура на инструкцията DECW 00h



Фиг. 179 DECW 00h (RP[3:0] = 1h, Рег. 100h = 00h, Рег. 101h = 02h)

Както подсказва и коментара, тази инструкция намалява съдържанието на двойка регистри. Инструкцията съдържа само младшите 8 бита на старшия регистър (в случая 00h), но асемблерът знае, че трябва да вземе под внимание и следващия регистър (в случая регистър 01h). Битове RP[3:0] съдържат старшите 4 бита на адресите на тези регистри (това може да се каже и по друг начин: битове RP[3:0]

определят страницата, в която са разположени тези регистри). В нашия пример RP[3:0] = 01h, което означава, че инструкцията адресира двойката регистри 100h:101h, които съдържат стойност 0002h преди нейното изпълнение. След изпълнението на инструкцията, стойността в тези регистри се намалява до 0001h.

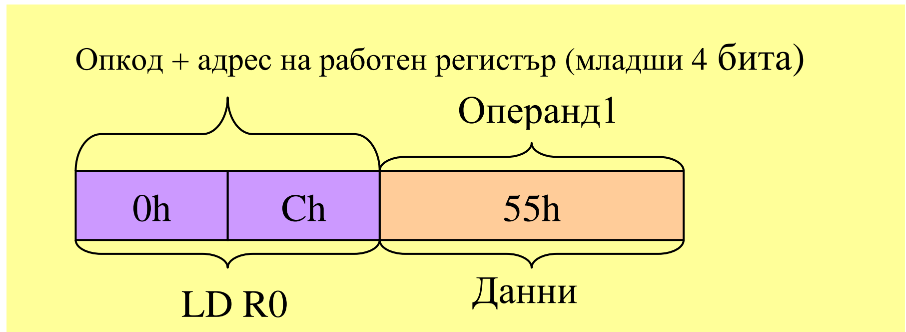
4-битова регистрова адресация (r, rr)

При тази адресация инструкцията съдържа младшите 4 бита на адреса на регистър от Регистровата Памет. Тази адресация се използва за адресация на работен регистър или двойка работни регистри от Регистровата Памет. Текущата група работни регистри се определя от битове RP[7:4], а текущата страница се определя от битове RP[3:0]. Пълният 12-битов адрес на регистъра е **RP[3:0]:RP[7:4]:[младши четири бита 3:0].**

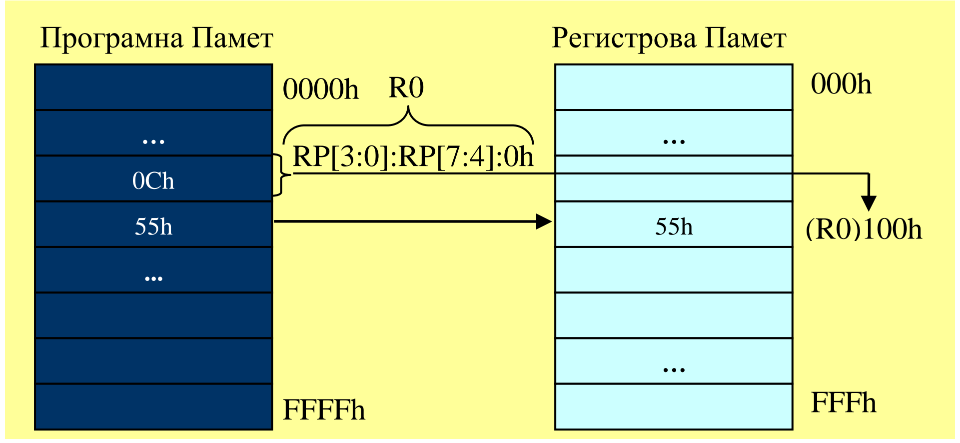
Пример (LD r, IM): Приема се, че RP[3:0] = 1h, RP[7:4] = 0h

```
LD      R0, #55h ;Запиши 55h в регистър  R0
```

Фиг.180 и 181 показват вътрешната структура и разполагането в Програмната Памет на тази инструкция.



Фиг. 180 Структура на инструкцията LD R0, #55h



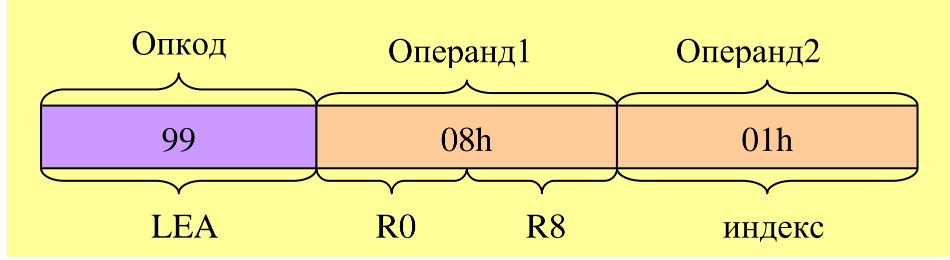
Фиг. 181 LD R0, #55h (RP[3:0] = 1h, RP[7:4] = 0h)

Пример (LEA rr, X(rr)): Приема се, че RP[3:0] = 0h, RP[7:4] = 0h, R8 = 55h и R9 = 54h

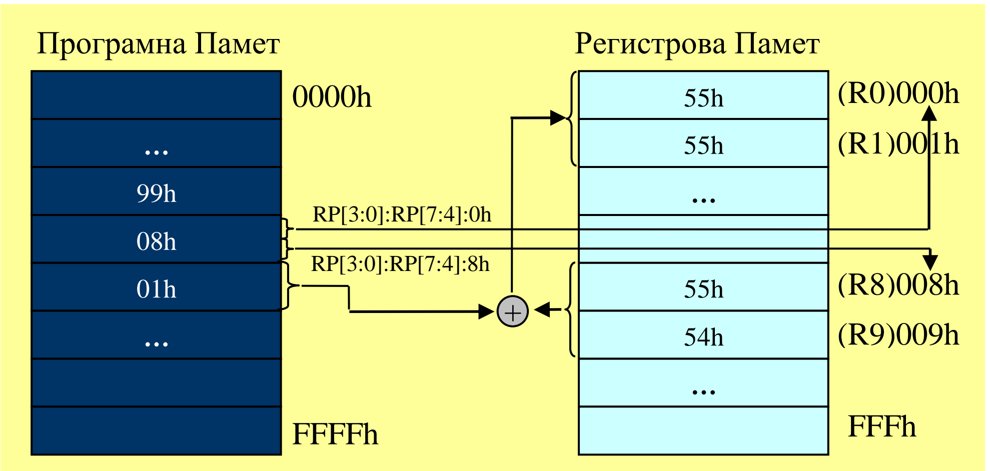
```
LEA   RR0, 1(RR8) ;Запиши стойността, съхранена в двойката
                  ; работни регистри R8:R9 плюс индекс 1 в
                  ; двойката работни регистри R0:R1
```

Фиг.182 и 183 показват вътрешната структура и разполагането в Програмната Памет на тази инструкция.

Тази инструкция записва стойността, съхранена в двойката регистри R8:R9 плюс 1, в двойката работни регистри R0:R1.



Фиг. 182 Структура на инструкцията LEA RR0, 1(RR8)



Фиг. 183 LEA RR0, 1(RR8) (RP[3:0] = 0h, RP[7:4] = 0h, R8 = 55h, R9 = 54h)

21.5.2 Косвена адресация (IR, IRR, Ir, Irr)

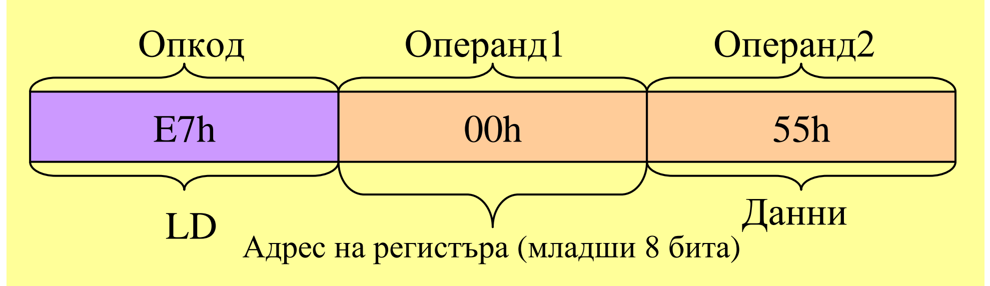
При тази адресация съдържанието на специфицирания в инструкцията регистър (или двойка регистри) или работен регистър (или двойка работни регистри) е адрес. В зависимост от инструкцията това може да бъде адрес на регистър от Регистровата Памет, Програмната Памет или Паметта за Данни.

Пример (LD IR, IM): Приема се, че RP[3:0] = 1h и регистър 100h = FFh

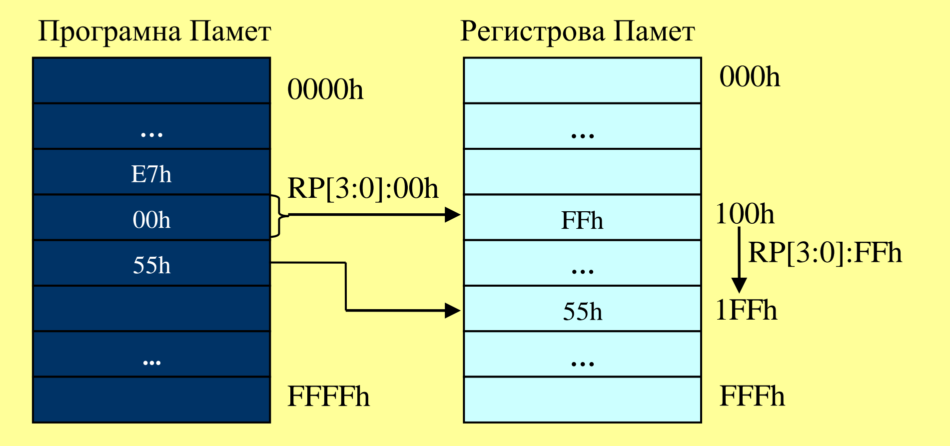
```
LD    @00h, #55h ;Запиши стойността 55h в регистъра, сочен от
                 ;регистър 00h на текущата страница
```

Фиг.184 и 185 показват вътрешната структура и разполагането в Програмната Памет на тази инструкция.

Тази инструкция записва 55h в регистъра, сочен от специфицирания в инструкцията регистър.



Фиг. 184 Структура на инструкцията LD @00h, #55h



Фиг. 185 LD @00h, #55h (RP[3:0] = 1h, 100h = FFh)

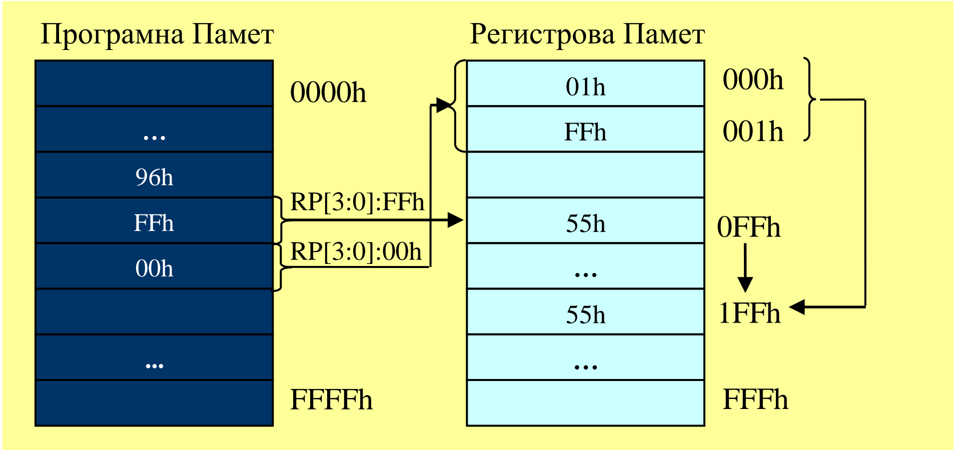
Пример (LDX IRR, R): Приема се, че RP[3:0] = 0h, двойката регистри 000h:001h = 01FFh и регистър 0FFh = 55h

```
LDX   @00h, FFh ;Запиши стойността в регистър FFh в регистъра,
                ;сочен от двойката регистри 00h:01h
```

Фиг.186 и 187 показват вътрешната структура и разполагането в Програмната Памет на тази инструкция.

Стойността, съхранена в регистър FF на текущата страница, се записва в регистъра, сочен от двойката регистри 00h:01h.

Фиг. 186 Структура на инструкцията LD @00h, FFh



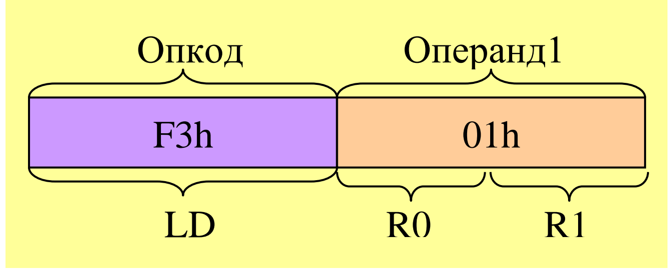
Фиг. 187 LD @00h, FFh (RP[3:0] = 0h, 0FFh = 55h)

Пример (LD Ir, r): Приема се, че RP[3:0] = 0h, RP[7:4] = 0h, R0 = FFh и R1 = 55h

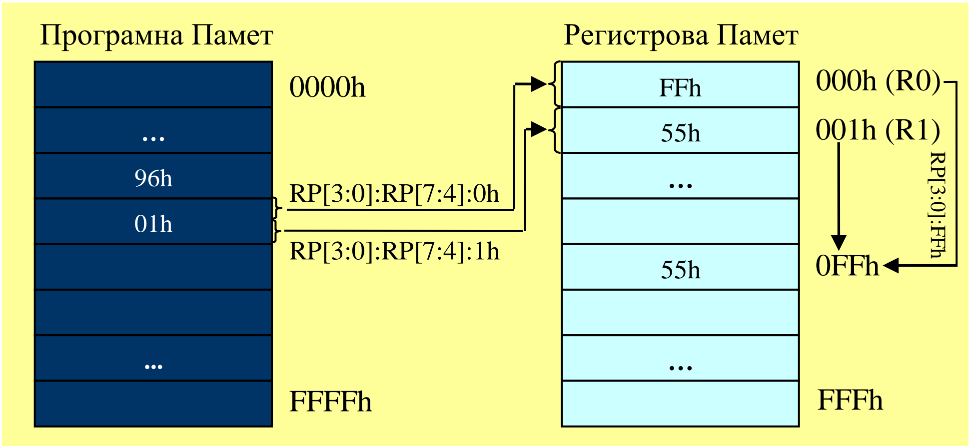
```
LD    @R0, R1 ;Запиши стойността на R1 в регистъра, сочен от
              ;регистър R0
```

Фиг.188 и 189 показват вътрешната структура и разполагането в Програмната Памет на тази инструкция.

Стойността, съхранена в работния регистър R1, се записва в регистъра, сочен от работния регистър R0. В нашия пример стойността 55h се записва в регистър 0FFh.



Фиг. 188 Структура на инструкцията LD @R0, R1



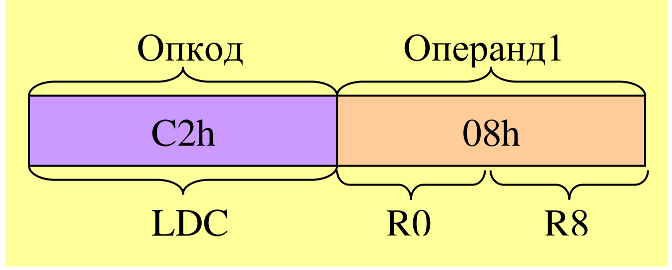
Фиг. 189 LD @R0, R1 (RP[3:0] = 0h, RP[7:4] = 0h, R0 = FFh, R1 = 55h)

Пример (LDC r, Irr): Приема се, че SP[3:0] = 0h, SP[7:4] = 0h, R8:R9 = 1234h и клетката с адрес 1234h в Програмната Памет съдържа 55h

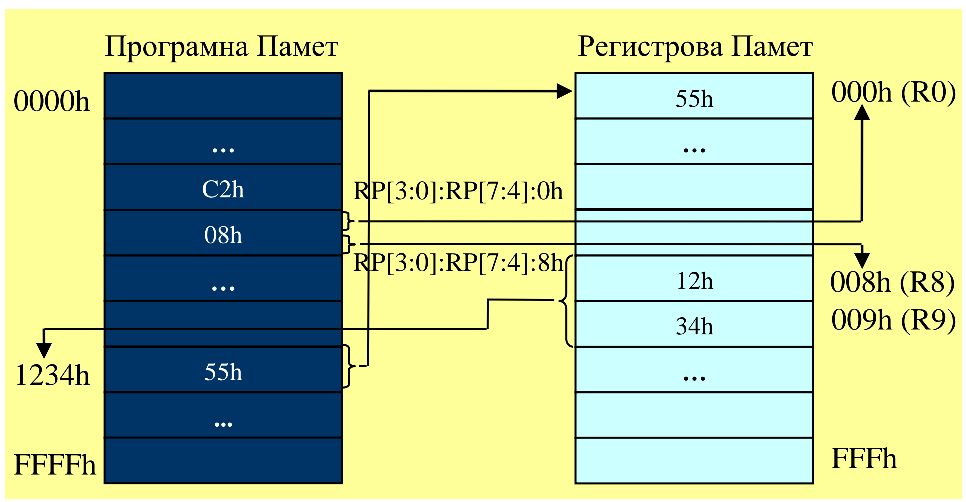
```
LDC R0, @RR8 ;Копирай стойността в програмната памет, сочена
             ;от двойката работни регистри R8:R9 в работния
             ;регистър R0
```

Фиг.190 и 191 показват вътрешната структура и разполагането в Програмната Памет на тази инструкция.

Тази инструкция копира стойността от Програмната Памет, сочена от двойката работни регистри R8:R9, в работния регистър R0.



Фиг. 190 Структура на инструкцията LD R3, @RR8



Фиг. 191 LDC R0, @RR8 (RP[3:0] = 0h, RP[7:4] = 0h, R8:R9 = 1234h)

21.5.3 Escaped-адресация

Този адресен режим се използва за директен достъп до всеки работен регистър в Регистровата Памет, използвайки 8- или 4-битова адресна методология.

За съжаление така и не ми стана много ясно какъв е смисълът на тази адресация.

8-битова Escaped-адресация

Спомнете си, че при регистровите адресации R и RR, и

косвените адресации IR и IRR, инструкцията съдържа младшите 8 бита на регистров адрес. Ако старшият полубайт на този адрес е Eh (1110b), младшия полубайт указва работен регистър и пълният 12-битов адрес е **RP[3:0]:RP[7:4]:[младши полубайт 3:0]**

Нека да обобщим:

Ако адресът е	се използва
00h ÷ DFh	8-битова регистрова/косвена адресация
E0h ÷ EFh	**8-битова Escaped-адресация**
F0h ÷ FFh	8-битова регистрова/косвена адресация

Табл. 29 Тип адресация според използвания адрес

Тъй като адресите E0h ÷ EFh се използват за 8-битова Escaped-адресация, за достъп до регистрите с тези адреси запишете RP[7:4] = Eh или използвайте косвена адресация (вижте 21.5.2 Косвена адресация).

12-битова Escaped-адресация

Спомнете си, че при 12-битовата регистрова адресация (ER) инструкцията съдържа пълния 12-битов регистров адрес.

<u>4-битова адресна методология</u>

Ако старшият байт на този адрес е EEh (11101110b), младшият полубайт указва работен регистър и пълният 12-битов адрес е **RP[3:0]:RP[7:4]:[младши полубайт 3:0]**

<u>8-битова адресна методология</u>

Ако старшият полубайт на този адрес е Eh (1110b), младшият байт указва регистър и пълният 12-битов адрес е **RP[3:0]:[младши полубайт 7:0]**.

Нека да обобщим:

Ако адресът е	се използва
000h ÷ DFFh	12-битова регистрова адресация
E00h ÷ EDFh	**12-битова Escaped-адресация с 8-битова адресна методология**
EE0h ÷ EEFh	**12-битова Escaped-адресация с 4-битова адресна методология**
EF0h ÷ EFFh	**12-битова Escaped-адресация с 8-битова адресна методология**
F00h ÷ FFFh	12-битова регистрова адресация

Табл. 30 Тип адресация според използвания адрес

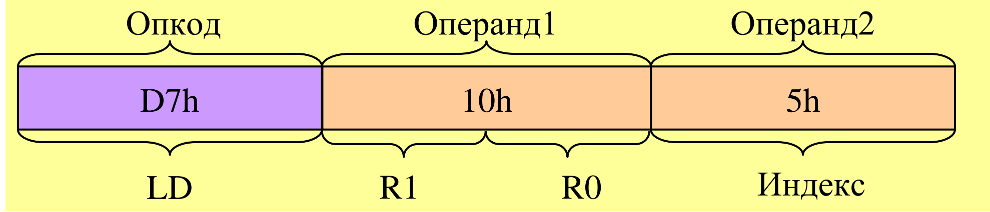
21.5.4 Индексна адресация (X)

При тази адресация адресът се формира като към стойността на работен регистър (или двойка регистри), указан в инструкцията, се добави знакова стойност (индекс) в диапазона -127 ÷ +128.

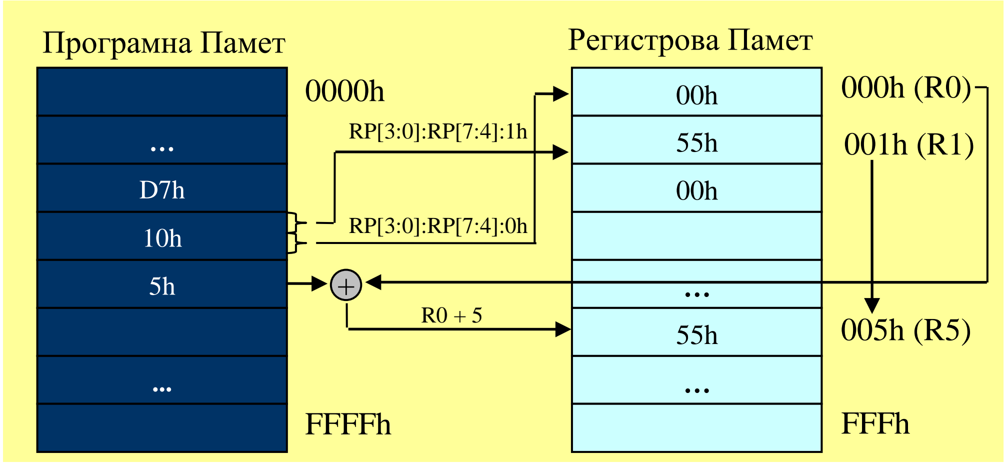
Пример (LD X(r), r): Приема се, че R0 = 00h, R1 = 55h, RP[3:0] = 0h, RP[7:4] = 0h

```
LD      5(R0), R1; Запиши стойността в R1 на адрес R0 + 5
```

Фиг.192 и 193 показват вътрешната структура и разполагането в Програмната Памет на тази инструкция.



Фиг. 192 Структура на инструкцията LD 5(R0), R1



Фиг. 193 LD 5(R0), R1 (RP[3:0] = 0h, RP[7:4] = 0h R0 = 00h, R1 = 55h)

21.5.5 Директна адресация (DA)

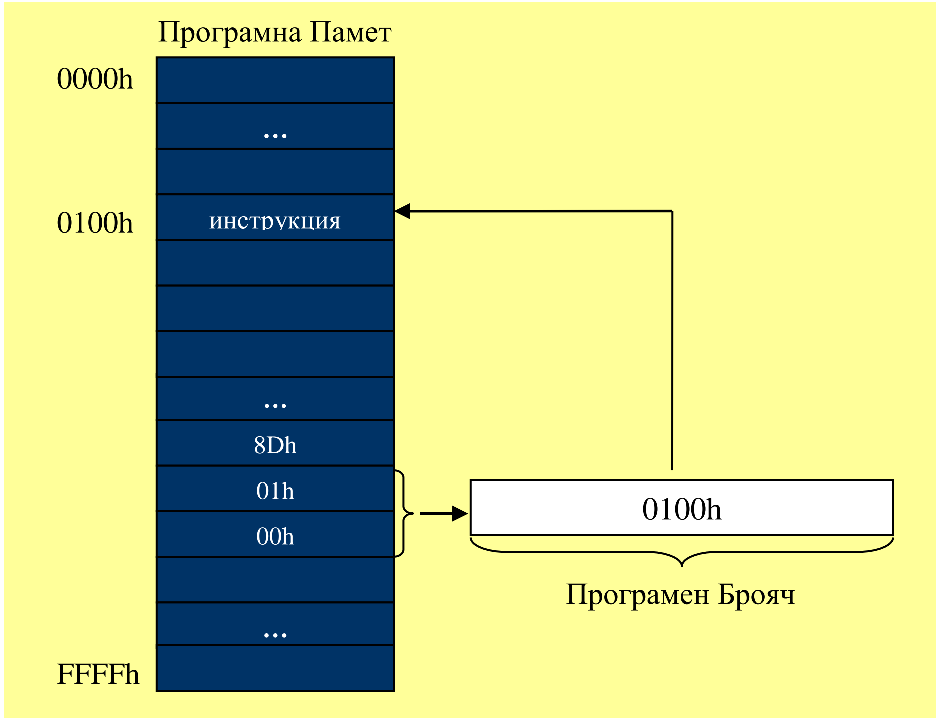
Тази адресация специфицира адреса на следващата за изпълнение инструкция. Само инструкциите **JP**, **JP cc** и **CALL** използват тази адресация. 16-битовия адрес, указан в инструкцията, се записва в Програмния Брояч.

Пример (JP DA):

```
JP      0100h ;Преход на адрес 0100h в Програмната Памет
```

Фиг.194 и 195 показват вътрешната структура и разполагането в Програмната Памет на тази инструкция.

Фиг. 194 Структура на инструкцията JP 0100h



Фиг. 195 JP 0100h

21.5.6 Относителна адресация (RA)

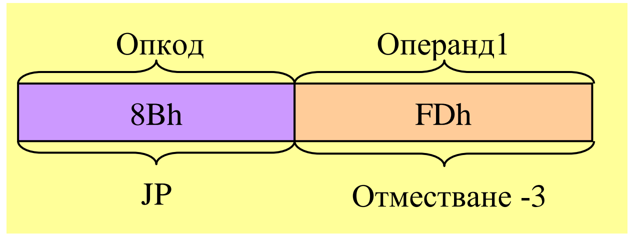
Инструкциите, използващи тази адресация, съдържат отместване в допълнителен код в обхвата -128 ÷ +127. Това отместване се добавя към текущото съдържание на Програмния Брояч. Програмата прави преход към инструкцията, намираща се на така формирания адрес. Преди изпълнението на инструкцията Програмният Брояч съдържа адреса на инструкцията, която се намира непосредствено след нея. Тази адресация се използва само от инструкциите **JR** и **DJNZ**.

Пример (JR RA):

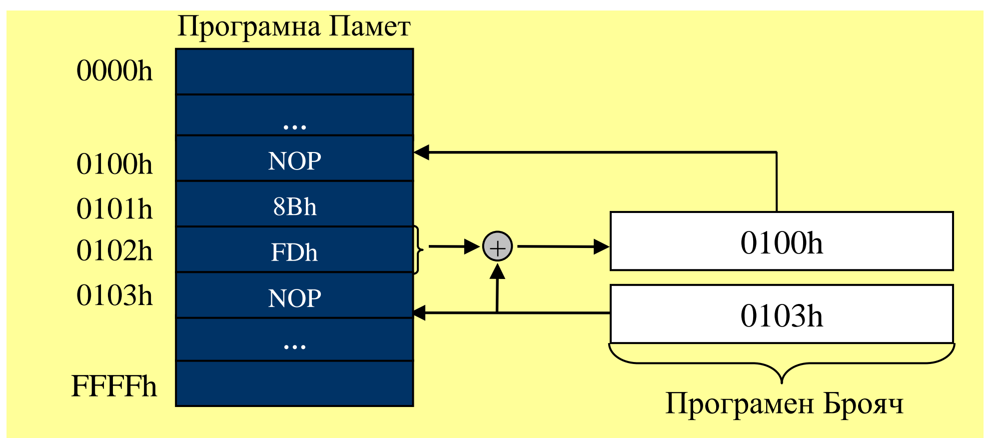
Адрес:
0100h:NOP

```
0101h:JR      0100h ;Преход към адрес 0100h
0103h:NOP
```

Фиг.196 и 197 показват вътрешната структура и разполагането в Програмната Памет на тази инструкция при допуснатите условия



Фиг. 196 Структура на инструкцията JR 0100h



Фиг. 197 Структура на инструкцията JR 0100h

Обърнете внимание, че инструкцията JR специфицира директно адреса, към който се извършва преход. Асемблерът автоматично изчислява отместването X като изважда стойността на Програмния Брояч от указания адрес. При нашите допускания отместването е:

X = 0100h – PC = 0100h – 0103h = FDh (-3 в допълнителен код).

По време на изпълнение на инструкцията **JR 0100h**

стойността на Програмния Брояч е 0103h. Отместването FDh (-3 в допълнителен код) се добавя към Програмния Брояч и програмата прави преход към резултатния адрес, т.е. . 0103h (259_{10}) + FDh (-3_{10}) = 0100h (256_{10}).

21.6 Списък с инструкции

Табл.31 и 32 описват символите и обозначенията, използвани в инструкциите.

Символ	Описание
dst	Дестинация
src	Източник
@	Префикс за косвена адресация
C	Флаг за пренос
SP	Указател към Стека
PC	Програмен Брояч
FLAGS	Регистър на Флаговете
RP	Регистров Указател
#	Префикс за данни и векторен адрес
b	Суфикс за двоични числа
%	Префикс за шестнадесетични числа
h	Суфикс за шестнадесетични числа

Табл. 31 Символи, използвани в инструкциите

Оцветените редове в следващата таблица подчертават адресните режими, които могат да използват Escaped-адресация.

Обозначение	Значение	Операнд	Описание
bit	Бит	bit	Номер на бит, 0 ÷ 7
cc	Тест-условия	cc	Вижте Табл.28
DA	Директен адрес	Addrs	16-битов адрес в

<table>
<tr><td></td><td></td><td></td><td>Програмната Памет,
0000h ÷ FFFFh</td></tr>
<tr><td rowspan="2">ER</td><td rowspan="2">Разширена регистрова адресация</td><td>Reg</td><td>12-битов адрес в Регистровата Памет, 000h ÷ FFFh</td></tr>
<tr><td>Rn</td><td>Rn (n = 0 ÷ 15) е работен регистър (при Escaped-адресация)</td></tr>
<tr><td>IM</td><td>Данни</td><td>#Data</td><td>8-битови данни,
00h ÷ FFh</td></tr>
<tr><td>Ir</td><td>Косвена адресация (работен регистър)</td><td>@Rn</td><td>Rn (n = 0 ÷ 15) е работен регистър</td></tr>
<tr><td rowspan="2">IR</td><td rowspan="2">Косвена адресация (регистър)</td><td>@Reg</td><td>Reg е 8-битов адрес на регистър от текущата страница, 00h ÷ FFh</td></tr>
<tr><td>@Rn</td><td>Rn (n = 0 ÷ 15) е работен регистър (при Escaped-адресация)</td></tr>
<tr><td>Irr</td><td>Косвена адресация (двойка работни регистри)</td><td>@RRp</td><td>RRp е старшия байт на двойка работни регистри,
p = 0, 2, 4,…,14</td></tr>
<tr><td rowspan="2">IRR</td><td rowspan="2">Косвена адресация (двойка регистри)</td><td>@Reg</td><td>Reg е 8-битов адрес на старшия байт на двойка регистри от текущата страница, 00h ÷ FEh</td></tr>
<tr><td>@Rn</td><td>Rn (n = 0 ÷ 15) е работен регистър (при Escaped-адресация)</td></tr>
<tr><td>p</td><td>Полярност</td><td>p</td><td>p е еднобитова двоична стойност,
0 или 1</td></tr>
</table>

<table>
<tr><td>r</td><td>Работен регистър</td><td>Rn</td><td>Rn (n = 0 ÷ 15) е работен регистър</td></tr>
<tr><td rowspan="2">R</td><td rowspan="2">Регистър</td><td>Reg</td><td>Reg е 8-битов адрес на регистър от текущата страница, 00h ÷ FFh</td></tr>
<tr><td>Rn</td><td>Rn (n = 0 ÷ 15) е работен регистър (при Escaped-адресация)</td></tr>
<tr><td>RA</td><td>Относителна адресация</td><td>X</td><td>Индекс в обхвата -128 ÷ +127, който е отместване относно адреса на следващата инструкция</td></tr>
<tr><td>rr</td><td>Двойка работни регистри</td><td>RRp</td><td>RRp е старшия байт на двойка работни регистри, p = 0, 2, 4,…,14</td></tr>
<tr><td rowspan="2">RR</td><td rowspan="2">Двойка регистри</td><td>Reg</td><td>Reg е 8-битов адрес на старшия байт на двойка регистри от текущата страница, 00h ÷ FEh</td></tr>
<tr><td>RRp</td><td>RRp е старшия байт на двойка работни регистри, p = 0, 2, 4,…,14 (при Escaped-адресация)</td></tr>
<tr><td>Vector</td><td>Векторен адрес</td><td>#Vector</td><td>00h ÷ FFh</td></tr>
<tr><td>X</td><td>Индекс</td><td>#Index</td><td>Регистър или двойка регистри, който се индексират със знакова стойност -128 ÷ +127</td></tr>
</table>

Табл. 32 Обозначения, използвани в инструкциите

Следващата таблица описва в резюме всички инструкции,

поддържани от процесора eZ8. На този етап може само да прегледате инструкциите и да се опитате да разберете доколкото успеете какво прави всяка от тях. Инструкциите са описани подробно в следващата глава. Обърнете внимание, че последните две колони съдържат броя на тактовите сигнали необходими за извличане на инструкцията от паметта и изпълнението й.

* - означава, че стойността на флага зависи от резултата от операцията
\- - означава, че стойността на флага не зависи от резултата от операцията
x - означава, че стойността на флага е недефинирана

асемблерна инструкция	**операция**	**Адресни режими**		**C**	**Z**	**S**	**V**	**D**	**H**	**изв**	**изп**
		dst	**src**								
ADC dst, src	dst ← dst + src + C	r	r	*	*	*	*	0	*	2	3
		r	Ir							2	4
		R	R							3	3
		R	IR							3	4
		R	IM							3	3
		IR	IM							3	4
ADCX dst, src	dst ← dst + src + C	ER	ER	*	*	*	*	0	*	4	3
		ER	IM							4	3
ADD dst, src	dst ← dst + src	r	r	*	*	*	*	0	*	2	3
		r	Ir							2	4
		R	R							3	3
		R	IR							3	4
		R	IM							3	3
		IR	IM							3	4
ADDX dst, src	dst ← dst + src	ER	ER	*	*	*	*	0	*	4	3
		ER	IM							4	3
AND dst, src	dst ← dst AND src	r	r	-	*	*	0	-	-	2	3
		r	Ir							2	4
		R	R							3	3
		R	IR							3	4
		R	IM							3	3
		IR	IM							3	4
ANDX dst, src	dst ← dst AND src	ER	ER	-	*	*	0	-	-	4	3
		ER	IM							4	3
ATM	Блокира всички прекъсвания и DMA заявки по време на изпълнението на следващите 3 инструкции			-	-	-	-	-	-	1	2
BCLR bit, dst	dst[bit] ← 0	r		-	-	-	-	-	-	2	2
BIT p, bit, dst	dst[bit] ← p	r		-	-	-	-	-	-	2	2
BRK	Стоп от дебъгера			-	-	-	-	-	-	1	2
BSET bit, dst	dst[bit] ← 1	r		-	-	-	-	-	-	2	2
BSWAP dst	dst[7:0] ← dst[0:7]	R		x	*	*	0	-	-	2	2
BTJ p, bit, src, dst	Ако src[bit] = p PC ← PC + X, X = dst – PC = -128 ÷ + 127		r	-	-	-	-	-	-	3	3
			Ir							3	4
BTJNZ bit, src, dst	Ако src[bit] = 1		r	-	-	-	-	-	-	3	3

	PC ← PC + X, X = dst – PC = -128 ÷ + 127		Ir							3	4
BTJZ bit, src, dst	Ако src[bit] = 0 PC ← PC + X, X = dst – PC = -128 ÷ + 127		r	-	-	-	-	-	-	3	3
			Ir							3	4
CALL dst	SP ← SP – 2 @SP ← PC PC ← dst	IRR		-	-	-	-	-	-	2	6
		DA								3	3
CCF	C ← ~C			*	-	-	-	-	-	1	2
CLR dst	dst ← 00h	R		-	-	-	-	-	-	2	2
		IR								2	3
COM dst	dst ← ~dst	R		-	*	*	0	-	-	2	2
		IR								2	3
CP dst, src	dst – src	r	r	*	*	*	*	-	-	2	3
		r	Ir							2	4
		R	R							3	3
		R	IR							3	4
		R	IM							3	3
		IR	IM							3	4
CPC dst, src	dst – src – C	r	r	*	*	*	*	-	-	3	3
		r	Ir							3	4
		R	R							4	3
		R	IR							4	4
		R	IM							4	3
		IR	IM							4	4
CPCX dst, src	dst – src – C	ER	ER	*	*	*	*	-	-	5	3
		ER	IM							5	3
CPX dst, src	dst – src	ER	ER	*	*	*	*	-	-	4	3
		ER	IM							4	3
DA dst	dst ← DecAdjust (dst)	R		*	*	*	x	-	-	2	2
		IR								2	3
DEC dst	dst ← dst – 1	R		-	*	*	*	-	-	2	2
		IR								2	3
DECW dst	dst ← dst – 1	RR		-	*	*	*	-	-	2	5
		IRR								2	6
DI	IRQCTL[7] ← 0			-	-	-	-	-	-	1	2
DJNZ dst, RA	dst ← dst – 1 Ако dst ≠ 0 PC ← PC + X, X = RA – PC = -128 ÷ + 127	r		-	-	-	-	-	-	2	Z/NZ 3/4
EI	IRQCTL[7] ← 1			-	-	-	-	-	-	1	2
HALT	Влизане в HALT-режим			-	-	-	-	-	-	1	2
INC dst	dst ← dst + 1	R		-	*	*	-	-	-	2	2
		IR								2	3
		r								1	2
INCW dst	dst ← dst + 1	RR		-	*	*	-	-	-	2	5
		IRR								2	6
IRET	FLAGS ← @SP SP ← SP + 1 PC ← @SP SP ← SP + 2 IRQCTL[7] ← 1			*	*	*	*	*	*	1	5
JP dst	PC ← dst	DA		-	-	-	-	-	-	3	2
		IRR								2	3
JP cc, dst	Ако cc е ИСТИНА PC ← dst	DA		-	-	-	-	-	-	3	2
JR dst	PC ← PC + X,	DA		-	-	-	-	-	-	2	2

	X = dst – PC = -128 ÷ + 127										
JR cc, dst	Ако cc е ИСТИНА PC ← PC + X, X = dst – PC = -128 ÷ + 127	DA		-	-	-	-	-	-	2	2
LD dst, src	dst ← src	r	IM	-	-	-	-	-	-	2	2
		r	X(r)							3	3
		X(r)	r							3	4
		r	Ir							2	3
		R	R							3	2
		R	IR							3	3
		R	IM							3	2
		IR	IM							3	3
		Ir	r							2	3
		IR	R							3	3
LDC dst, src	dst ← src	r	Irr	-	-	-	-	-	-	2	5
		Ir	Irr							2	9
		Irr	r							2	5
LDCI dst, src	dst ← src Rn ← Rn + 1 RRn ← RRn + 1	r	Irr	-	-	-	-	-	-	2	9
		Irr	r							2	9
LDE dst, src	dst ← src	r	Irr	-	-	-	-	-	-	2	5
		Irr	r							2	5
LDEI dst, src	dst ← src Rn ← Rn + 1 RRn ← RRn + 1	Ir	Irr	-	-	-	-	-	-	2	9
		Irr	Ir							2	9
LDWX dst, src	dst ← src	ER	ER	-	-	-	-	-	-	5	4
LDX dst, src	dst ← src	r	ER	-	-	-	-	-	-	3	2
		Ir	ER							3	3
		R	IRR							3	4
		IR	IRR							3	5
		r	X(rr)							3	4
		X(rr)	r							3	4
		ER	r							3	2
		ER	Ir							3	3
		IRR	R							3	4
		IRR	IR							3	5
		ER	ER							4	2
		ER	IM							4	2
LEA dst, X(src)	dst ← src + X X = -128 ÷ +127	r	X(r)	-	-	-	-	-	-	3	3
		rr	X(rr)							3	5
MULT dst	dst[15:0] ← dst[15:8]*dst[7:0]	RR		-	-	-	-	-	-	2	8
NOP				-	-	-	-	-	-	1	2
OR dst, src	dst ← dst OR src	r	r	-	*	*	0	-	-	2	3
		r	Ir							2	4
		R	R							3	3
		R	IR							3	4
		R	IM							3	3
		IR	IM							3	4
ORX dst, src	dst ← dst OR src	ER	ER	-	*	*	0	-	-	4	3
		ER	IM							4	3
POP dst	dst ← @SP SP ← SP + 1	R		-	-	-	-	-	-	2	2
		IR								2	3
POPX dst	dst ← @SP SP ← SP + 1	ER		-	-	-	-	-	-	3	2
PUSH src	SP ← SP - 1 @SP ← src		r	-	-	-	-	-	-	2	2
			IR							2	3
			IM							3	2

PUSHX src	SP ← SP – 1 @SP ← src		ER	-	-	-	-	-	-	3	2
RCF	C ← 0			0	-	-	-	-	-	1	2
RET	PC ← @SP SP ← SP + 2			-	-	-	-	-	-	1	4
RL dst	C D7 D6 D5 D4 D3 D2 D1 D0 dst	R		*	*	*	*	-	-	2	2
		IR								2	3
RLC dst	C D7 D6 D5 D4 D3 D2 D1 D0 dst	R		*	*	*	*	-	-	2	2
		IR								2	3
RR dst	D7 D6 D5 D4 D3 D2 D1 D0 C dst	R		*	*	*	*	-	-	2	2
		IR								2	3
RRC dst	D7 D6 D5 D4 D3 D2 D1 D0 C dst	R		*	*	*	*	-	-	2	2
		IR								2	3
SBC dst, src	dst ← dst – src – C	r	r	*	*	*	*	1	*	2	3
		r	Ir							2	4
		R	R							3	3
		R	IR							3	4
		R	IM							3	3
		IR	IM							3	4
SBCX dst, src	dst ← dst – src – C	ER	ER	*	*	*	*	1	*	4	3
		ER	IM							4	3
SCF	C ← 1			1	-	-	-	-	-	1	2
SRA dst	D7 D6 D5 D4 D3 D2 D1 D0 C dst	R		*	*	*	0	-	-	2	2
		IR								2	3
SRL dst	0 D7 D6 D5 D4 D3 D2 D1 D0 C dst	R		*	*	0	*	-	-	3	2
		IR								3	3
SRP src	RP ← src		IM	-	-	-	-	-	-	2	2
STOP	Влизане в STOP-режим			-	-	-	-	-	-	1	2
SUB dst, src	dst ← dst – src	r	r	*	*	*	*	1	*	2	3
		r	Ir							2	4
		R	R							3	3
		R	IR							3	4
		R	IM							3	3
		IR	IM							3	4
SUBX dst, src	dst ← dst – src	ER	ER	*	*	*	*	1	*	4	3
		ER	IM							4	3
SWAP dst	dst[7:4] ↔ dst[4:0]	R		x	*	*	x	-	-	2	2
		IR								2	3
TCM dst, src	(NOT dst) AND src	r	r	-	*	*	0	-	-	2	3
		r	Ir							2	4
		R	R							3	3
		R	IR							3	4
		R	IM							3	3
		IR	IM							3	4

TCMX dst, src	(NOT dst) AND src	ER	ER	-	*	*	0	-	-	4	3
		ER	IM							4	3
TM dst, src	dst AND src	r	r	-	*	*	0	-	-	2	3
		r	Ir							2	4
		R	R							3	3
		R	IR							3	4
		R	IM							3	3
		IR	IM							3	4
TMX dst, src	dst AND src	ER	ER	-	*	*	0	-	-	4	3
		ER	IM							4	3
TRAP Vector	SP ← SP – 2 @SP ← PC SP ← SP – 1 @SP ← FLAGS PC ← @Vector		#Vect or	-	-	-	-	-	-	2	6
WDT	Рестартиране на WDT-таймера			-	-	-	-	-	-	1	2
XOR dst, src	dst ← dst XOR src	r	r	-	*	*	0	-	-	2	3
		r	Ir							2	4
		R	R							3	3
		R	IR							3	4
		R	IM							3	3
		IR	IM							3	4
XORX dst, src	dst ← dst XOR src	ER	ER	-	*	*	0	-	-	4	3
		ER	IM							4	3

Табл. 33 Списък с инструкции на процесора eZ8

Най-добре ще научите инструкциите като ги пробвате как работят на практика. Затова Ви препоръчвам да прескочите следващите две глави и да прочетете глава 24 Развойна среда Zilog Developer Studio II. В нея ще Ви покажа как да си създадете проект и ще Ви запозная с основните менюта и прозорци. След това се върнете отново към пропуснатите глави и използвайте наученото, за да тествате инструкциите, използвайки симулатора на средата Zilog Developer Studio II.

Препоръчвам Ви да си разпечатате Табл.31, 32 и 33, за да са Ви под ръка, тъй като често ще се налага да се обръщате към тях, докато свикнете с инструкциите.

22 Описание на инструкциите на eZ8

Преди да премина към описанието на инструкциите ще се опитам да Ви обясня използването на Escaped-адресацията. Спомнете си, че когато използвате 8-битови регистрови адреси Ex (x = 0÷F) или 12-битови регистрови адреси EEx (x = 0÷F), младшият полубайт на този адрес указва работен регистър (R0, R1, …, R15). Вместо да използвате тези адреси в инструкции, които изискват 8- или 12-битови регистрови адреси, Вие може директно да използвате имената на работните регистри, т.е. . вместо адрес E0 може да използвате R0, или вместо адрес EE0 може да използвате R0. Асемблерът автоматично заменя името на работния регистър със съответния адрес. Ще илюстрирам описаното с на базата на инструкцията **ADD R, R**. Тази инструкция сумира стойностите на регистрите, указани като операнди. Регистрите използват 8-битова регистрова адресация (R).

```
ADD E0h, 12h ;Съдържанието на регистър 12h се добавя към
             ;съдържанието на работния регистър R0
             ;(използва се Escaped-адресация) и резултатът
             ;се съхранява в R0.
```

Тази инструкция е еквивалентна на следното:

```
ADD R0, 12h ;Съдържанието на регистър 12h се добавя към
            ;съдържанието на работния регистър R0
            ;(използва се Escaped-адресация) и резултатът
            ;се съхранява в R0.
```

Същото правило е валидно и за инструкции, използващи 12-битови регистрови адреси. Например следните две инструкции са еквивалентни.

```
ADCX    EE4h, B12h    ≈    ADCX    R4    , B12h
```

ADC

Синтаксис

ADC dst, src

Операция

dst ← dst + src + C

Описание

Събиране с пренос. Стойността на операнда src и флага C се добавя към стойността на операнда dst, и резултатът се съхранява в операнда dst. Събирането се извършва в допълнителен код.

Флагове

C 1 ако има пренос от бит 7, иначе е 0
Z 1 ако резултатът е 0, иначе е 0
S 1 ако резултатът е отрицателен, иначе е 0
V 1 ако има аритметично препълване, иначе е 0
D 0
H 1 ако има пренос от бит 3, иначе е 0

абревиатура	dst	src	опкод	o1	o2	o3
ADC	r1	r2	12h	{r1, r2}	-	-
ADC	r1	@r2	13h	{r1, r2}	-	-
ADC	R1	R2	14h	R2	R1	-
ADC	R1	@R2	15h	R2	R1	-
ADC	R1	IM	16h	R1	IM	-
ADC	@R1	IM	17h	R1	IM	

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

ADC R3, R11

Ако работният регистър R3 съдържа стойност 16h, флаг C е 1 и работният регистър R11 съдържа стойност 20h, горната инструкция записва стойност 37h в R3 и нулира флаговете C, Z, S, V, D и H.

Пример:

ADC R15, @R10

Ако работният регистър R15 съдържа стойност 16h, флаг C е 0 и работният регистър R10 съдържа стойност 20h, и регистър 20h съдържа стойност 11h, горната инструкция записва стойност 27h в R15 и нулира флаговете C, Z, S, V, D и H.

Пример:

ADC 34h, 12h

Ако регистър 34h съдържа стойност 2Eh, флаг C е 1 и регистър 12h съдържа стойност 1Bh, горната инструкция записва стойност 4Ah в регистър 34h, установява флаг H и нулира флаговете C, Z, S, V и D.

Пример:

ADC E4h, 12h

Използването на адрес E4h активира Escaped-адресация, т.е. . младшият полубайтбайт на адреса указва работен регистър. В нашия случай това е R4. Ако работният регистър R4 съдържа стойност 2Eh, флаг C е 1 и регистър 12h съдържа стойност 1Bh, горната инструкция записва стойност 4Ah в R4, установява флаг H и нулира флаговете C, Z, S, V и D.

Пример:

ADC 4Bh, @R3

Ако регистър 4Bh съдържа стойност 82h, флаг C е 1 и работният регистър R3 (използва се Escaped-адресация) съдържа стойност 10h, и регистър 10h съдържа стойност 01h, горната инструкция записва стойност 84h в регистър 4Bh, установява флаг S и нулира флаговете C, Z, V, D и H.

Пример:

ADC 6Ch, #03h

Ако регистър 6Ch съдържа стойност 2Ah, флаг C е 0, горната инструкция записва стойност 2Dh в регистър 6Ch, и нулира флаговете C, Z, S, V, D и H.

Пример:

ADC @D4h, #02h

Ако регистър D4h съдържа стойност 55h и регистър 55h съдържа стойност 4Ch, и флаг C е 1, горната инструкция записва стойност 4Fh в регистър 55h и нулира флаговете C, Z, S, V, D и H.

ADCX

Синтаксис

ADCX dst, src

Операция

dst ← dst + src + C

Описание

Събиране с пренос. Стойността на операнда src и флага C се добавя към стойността на операнда dst, и резултатът се съхранява в операнда dst. Събирането се извършва в допълнителен код. Операндите dst и src използват разширена (12 битова) адресация за достъп до всеки регистър в Регистровата Памет.

Флагове

C 1 ако има пренос от бит 7, иначе е 0
Z 1 ако резултатът е 0, иначе е 0
S 1 ако резултатът е отрицателен, иначе е 0
V 1 ако има аритметично препълване, иначе е 0
D 0
H 1 ако има пренос от бит 3, иначе е 0

абревиатура	dst	src	опкод	o1	o2	o3
ADCX	ER1	ER2	18h	ER2[11:4]	{ ER2[4:3], ER1[11:8]}	ER1[7:0]
ADCX	ER	IM	19h	src	{0h, ER1[11:8]}	ER1[7:0]

Адресният режим ER може да използва Escaped-адресация. Ако старшият байт на адреса е EEh, младшият полубайт указва работен регистър.

Пример:

ADCX 634h, B12h

Ако регистър 634h съдържа стойност 2Eh, флаг C е 1 и регистър B12h съдържа стойност 1Bh, горната инструкция записва стойност 4Ah в регистър 634h, установява флаг H и нулира флаговете C, Z, S, V и D.

Пример:

ADCX EE4h, B12h

Използването на адрес EE4h активира Escaped-адресация, т.е. . младшият полубайт на адреса указва работен регистър. В нашия случай това е R4. Ако работният регистър R4 съдържа стойност 2Eh, флаг C е 1 и регистър B12h съдържа стойност 1Bh, горната инструкция записва стойност 4Ah в R4, установява флаг H и нулира флаговете C, Z, S, V и D.

Пример:

ADCX 46Ch, #03h

Ако регистър 46Ch съдържа стойност 2Ah и флаг C е 0, горната инструкция записва стойност 2Dh в регистър 46Ch и нулира флаговете C, Z, S, V, D и H.

ADD

Синтаксис

ADD dst, src

Операция

dst ← dst + src

Описание

Събиране. Стойността на операнда src се добавя към стойността на операнда dst и резултатът се съхранява в операнда dst. Събирането се извършва в допълнителен код.

Флагове

C 1 ако има пренос от бит 7, иначе е 0
Z 1 ако резултатът е 0, иначе е 0
S 1 ако резултатът е отрицателен, иначе е 0
V 1 ако има аритметично препълване, иначе е 0
D 0
H 1 ако има пренос от бит 3, иначе е 0

абревиатура	dst	src	опкод	o1	o2	o3
ADD	r1	r2	02h	(r1, r2)	-	-
ADD	r1	@r2	03h	(r1, r2)	-	-
ADD	R1	R2	04h	R2	R1	-
ADD	R1	IR	05h	R2	R1	-
ADD	R	IM	06h	R1	IM	-
ADD	@R1	IM	07h	R1	IM	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

ADD R3, R11

Ако работният регистър R3 съдържа стойност 16h и работният регистър R11 съдържа стойност 20h, горната инструкция записва стойност 36h в R3 и нулира флаговете C, Z, S, V, D и H.

Пример:

ADD R15, @R10

Ако работният регистър R15 съдържа стойност 16h и работният регистър R10 съдържа стойност 20h, и регистър 20h съдържа стойност 11h, горната инструкция записва стойност 27h в R15 и нулира флаговете C, Z, S, V, D и H.

Пример:

ADD 34h, 12h

Ако регистър 34h съдържа стойност 2Eh и регистър 12h съдържа стойност 1Bh, горната инструкция записва стойност 49h в регистър 34h, установява флаг H и нулира флаговете C, Z, S, V и D.

Пример:

ADD E4h, 12h

Използването на адрес E4h активира Escaped-адресация, т.е. . младшият байт на адреса указва работен регистър. В нашия случай това е R4. Ако работният регистър R4 съдържа стойност 2Eh и регистър 12h съдържа стойност 1Bh, горната инструкция записва стойност 49h в R4, установява флаг H и нулира флаговете C, Z, S, V и D.

Пример:

ADD 4Bh, @R3

Ако регистър 4Bh съдържа стойност 82h, и работният регистър R3 (използва се Escaped-адресация) съдържа стойност 10h и регистър 10h съдържа стойност 01h, горната инструкция записва стойност 83h в регистър 4Bh, установява флаг S и нулира флаговете C, Z, V, D и H.

Пример:

ADD 6Ch, #03h

Ако регистър 6Ch съдържа стойност 2Ah, горната инструкция записва стойност 2Dh в регистър 6Ch, и нулира флаговете C, Z, S, V, D и H.

Пример:

ADD @D4h, #02h

Ако регистър D4h съдържа стойност 5Fh и регистър 5Fh съдържа стойност 4Ch, горната инструкция записва стойност 4Eh в регистър 5Fh и нулира флаговете C, Z, S, V, D и H.

ADDX

Синтаксис

ADDX dst, src

Операция

dst ← dst + src

Описание

Събиране. Стойността на операнда src се добавя към стойността на операнда dst, и резултатът се съхранява в операнда dst. Събирането се извършва в допълнителен код. Операндите dst и src използват разширена (12 битова) адресация за достъп до всеки регистър в Регистровата Памет.

Флагове

C 1 ако има пренос от бит 7, иначе е 0
Z 1 ако резултатът е 0, иначе е 0
S 1 ако резултатът е отрицателен, иначе е 0
V 1 ако има аритметично препълване, иначе е 0
D 0
H 1 ако има пренос от бит 3, иначе е 0

абревиатура	dst	src	опкод	o1	o2	o3
ADDX	ER	ER	08h	ER2[11:4]	{ER2[3:0], ER1[11:8]}	ER1[7:0]
ADDX	ER	IM	09h	IM	{0h, ER1[11:8]}	ER1[7:0]

Адресният режим ER може да използва Escaped-адресация. Ако старшият байт на адреса е EEh, младшият полубайт указва работен регистър.

Пример:

ADDX 634h, B12h

Ако регистър 634h съдържа стойност 2Eh и регистър B12h съдържа стойност 1Bh, горната инструкция записва стойност 49h в регистър 634h, установява флаг H и нулира флаговете C, Z, S, V и D.

Пример:

ADDX EE4h, B12h

Използването на адрес EE4h активира Escaped-адресация, т.е. . младшият байт на адреса указва работен регистър. В нашия случай това е R4. Ако работният регистър R4 съдържа стойност 2Eh и регистър B12h съдържа стойност 1Bh, горната инструкция записва стойност 49h в R4, установява флаг H и нулира флаговете C, Z, S, V и D.

Пример:

ADDX 46Ch, #03h

Ако регистър 46Ch съдържа стойност 2Ah, горната инструкция записва стойност 2Dh в регистър 46Ch и нулира флаговете C, Z, S, V, D и H.

AND

Синтаксис

AND dst, src

Операция

dst ← dst AND src

Описание

Логическо И. Операндът src се AND-ва с операнда dst и резултатът се съхранява в операнда dst.

Флагове

C -
Z 1 ако резултатът е 0, иначе е 0
S 1 ако бит 7 на резултата е 1, иначе е 0
V 0
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
AND	r1	r2	52h	(r1, r2)	-	-
AND	r1	@r2	53h	(r1, r2)	-	-
AND	R1	R1	54h	R2	R1	-
AND	R1	@R2	55h	R2	R1	-
AND	R1	IM	56h	R1	IM	-
AND	@R1	IM	57h	R1	IM	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

AND R1, R14

Ако работният регистър R1 съдържа стойност F9h (11111001b) и работният регистър R14 съдържа стойност 8Dh (10001101b), горната инструкция записва стойност 08h (00001000b) в R1 и нулира флаговете Z, V, и S.

Пример:

AND R4, @R13

Ако работният регистър R4 съдържа стойност 38h (00111000b), работният регистър R13 съдържа стойност 7Bh и регистър 7Bh съдържа стойност 6Ah (01101010b), горната инструкция записва стойност 68h (01101000b) в R4 и нулира флаговете Z, V и S.

Пример:

AND 3Ah, 42h

Ако регистър 3Ah съдържа стойност F5h (11110101b) и регистър 42h съдържа стойност 0Ah (00001010b), горната инструкция записва стойност 00h в регистър 3Ah, установява флаг Z и нулира флаговете V и S.

Пример:

AND E4h, 42h

Използването на адрес E4h активира Escaped-адресация, т.е. . младшият байт на адреса указва работен регистър. В нашия случай това е R4. Ако работният регистър R4 съдържа стойност F5h (11110101b) и регистър 42h съдържа стойност 0Ah (00001010b), горната инструкция записва стойност 00h в регистър R4, установява флаг Z и нулира флаговете V и S.

Пример:

AND R5, @45h

Ако регистър R5 (използва се Escaped-адресация) съдържа стойност F0h (11110000b), регистър 45h съдържа стойност 3Ah и регистър 3Ah съдържа стойност 7Fh (01111111b), горната инструкция записва стойност 70h в R5 и нулира флаговете Z, V и S.

Пример:

AND 7Ah, #F0h

Ако регистър 7Ah съдържа стойност F7h (11110111b), горната инструкция записва стойност F0h в регистър 7Ah, установява флаг S и нулира флаговете Z и V.

Пример:

AND @R3, #05h

Ако регистър R3 (използва се Escaped-адресация) съдържа стойност 3Eh и регистър 3Eh съдържа стойност ECh (11101100b), горната инструкция записва стойност 04h (00000100b) в регистър 3Eh и нулира флаговете Z,V и S.

ANDX

Синтаксис

ANDX dst, src

Операция

dst ← dst AND src

Описание

Логическо И. Операндът src се AND-ва с операнда dst и резултатът се съхранява в операнда dst. Операндите dst и src използват разширена (12 битова) адресация за достъп до всеки регистър в Регистровата Памет.

Флагове

C -
Z 1 ако резултатът е 0, иначе е 0
S 1 ако бит 7 на резултата е 1, иначе е 0
V 0
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
ANDX	ER1	ER2	58h	ER2 [11:4]	{ ER2[3:0], ER1[11:8]}	ER1[7:0]
ANDX	ER1	IM	59h	src	{0h, ER1 [11:8]}	ER1[7:0]

Адресният режим ER може да използва Escaped-адресация. Ако старшият байт на адреса е EEh, младшият полубайт указва работен регистър.

Пример:

ANDX 93Ah, 142h

Ако регистър 93Ah съдържа стойност F5h (11110101b) и

регистър 142h съдържа стойност 0Ah (00001010b), горната инструкция записва стойност 00h (00000000b) в регистър 93Ah, установява флага Z и нулира флаговете V и S.

Пример:

ANDX D7Ah, #F0h

Ако регистър D7Ah съдържа стойност F7h (11110111b), горната инструкция записва стойност F0h (11110000b) в регистър D7Ah, установява флага S и нулира флаговете Z и V.

ATM

Синтаксис

ATM

Операция

Блокира всички прекъсвания и DMA[1] заявки през следващите 3 инструкции.

Описание

Тази инструкция принуждава процесора eZ8 да изпълни следващите 3 инструкции като една операция (**ATM – atomic (неделим)**). По време на изпълнение на тези инструкции всички прекъсвания и DMA заявки са блокирани

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
ATM	-	-	2Fh	-	-	-

Забележка[1]: **DMA (Direct Memory Access)** е метод на трансфер на данни от RAM-паметта до друга част на микроконтролера и обратно, без участието на процесора. Този процес се контролира от DMA-контролер. Микроконтролерите от серията Z8Encore!ZF803A нямат DMA-контролер.

BCLR

Синтаксис

BCLR bit, dst

Операция

dst[bit] $\leftarrow$ 0

Описание

Нулира бит на операнда dst.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	бит	dst	опкод	o1	o2	o3
BCLR	bit	r1	E2h	{0b, bit, r1}	-	-

Пример:

BCLR 4, R7

Ако работният регистър R7 съдържа стойност 38h (00111000b), горната инструкция нулира бит 4 и стойността в R7 става 28h (00101000b).

BIT

Синтаксис

BIT p,bit, dst

Операция

dst[bit] ← p

Описание

Нулира/Установява бит на операнда dst.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	полярност	бит	dst	опкод	o1	o2	o3
BIT	p	bit	r1	E2h	{p, bit, r1}	-	-

Пример:

BIT 0,4, R7

Ако работният регистър R7 съдържа стойност 38h (00111000b), горната инструкция нулира бит 4 и стойността в R7 става 28h (00101000b).

Пример:

BIT 1,2, R7

Ако работният регистър R7 съдържа стойност 38h (00111000b), горната инструкция установява бит 2 и

стойността в R7 става 3Ch (00111100b).

BRK

Синтаксис

BRK

Операция

-

Описание

Тази инструкция изпълнява спиране на вградения дебъгер на специфициран адрес.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	dst	опкод	o1	o2	o3
BRK	-	-	00h	-	-	-

BSET

Синтаксис

BSET bit, dst

Операция

dst[bit] ← 1

Описание

Установява бит на операнда dst.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	бит	dst	опкод	o1	o2	o3
BSET	bit	r1	E2h	{1b, bit, r1}	-	-

Пример:

BSET 2, R7

Ако работният регистър R7 съдържа стойност 38h (00111000b), горната инструкция установява бит 2 и стойността в R7 става 3Ch (00111100b).

BSWAP

Синтаксис

BSWAP dst

Операция

dst[7:0] ← dst[0:7]

Описание

Тази инструкция разменя местата на битовете на операнда dst:

dst[7] ↔ dst[0]
dst[6] ↔ dst[1]
dst[5] ↔ dst[2]
dst[4] ↔ dst[3]

Флагове

C	-
Z	1 ако резултатът е 0, иначе е 0
S	1 ако резултатът е отрицателен, иначе е 0
V	0
D	-
H	-

абревиатура	src	dst	опкод	o1	o2	o3
BSWAP	-	R1	D5h	R1	-	-

Адресният режим R може да използва Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

BSWAP 00h

Ако регистър 00h съдържа стойност 80h (10000000b), след изпълнение на инструкцията регистърът ще съдържа стойност 01h (00000001).

BTJ

Синтаксис

BTJ p, bit, src, DA

Операция

Ако src[bit] = p, то PC ← PC + X, където X = DA - PC

Описание

Тази инструкция тества бит bit в операнда src дали е равен на p. Ако това е така, към Програмния Брояч (който сочи вече към следващата инструкция) се добавя знаково отместване X = DA – PC, което трябва да е в диапазона -128 ÷ +127, и се извършва преход към новия адрес. В противен случай процесорът продължава с изпълнение на инструкцията, която се намира непосредствено след тази инструкция.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	полярност	бит	src	адрес	опкод	o1	o2	o3
BTJ	p	bit	r2	DA	F6h	{p, bit[2:0], r2}	X	-
BTJ	p	bit	@r2	DA	F7h	{p, bit[2:0], r2}	X	-

Пример:

BTJ 0, 5, R7, NEXT

```
        HALT
NEXT:
        LD          R0, @R2
```

Ако работният регистър R7 съдържа стойност 20h (00100000b), то следващата инструкция проверява бит 5 на R7 дали е 0. Тъй като това не е така, програмата преминава към изпълнение на следващата инструкция, която в нашия пример е HALT.

Обърнете внимание, че вместо директен адрес, в инструкцията се използва идентификатор NEXT. Този идентификатор се нарича етикет. На всяка инструкция може да бъде поставен етикет, който е идентичен с адреса в Програмната Памет, където се разполага инструкцията. Това освобождава програмиста от използването на директни адреси в програмата. В нашия пример етикетът NEXT маркира адреса, на който е разположена инструкцията LD R0, @R2. Етикетът и инструкцията могат да бъдат разположени на един ред или на отделни редове както е показано в примера.

Пример:

```
        BTJ         1, 5, R7, NEXT
        HALT
NEXT:
        LD          R0, @R2
```

Ако работният регистър R7 съдържа стойност 20h (00100000b), то следващата инструкция проверява бит 5 на R7 дали е 1. Тъй като това е така, програмата прескача следващата инструкция и преминава към изпълнение на инструкцията LD R0, @R2.

BTJNZ

Синтаксис

BTJNZ bit, src, DA

Операция

Ако src[bit] = 1, то PC ← PC + X, където X = DA - PC

Описание

Тази инструкция тества бит bit в операнда src дали е равен на 1. Ако това е така, към Програмния Брояч (който сочи вече към следващата инструкция) се добавя знаково отместване X = DA – PC, което трябва да е в диапазона -128 ÷ +127, и се извършва преход към новия адрес. В противен случай процесорът продължава с изпълнение на инструкцията, която се намира непосредствено след тази инструкция.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	бит	src	адрес	опкод	o1	o2	o3
BTJNZ	bit	r2	DA	F6h	{1b, bit, r2}	X	-
BTJNZ	bit	@r2	DA	F7h	{1b, bit, r2}	X	-

Пример:

```
        BTJNZ       5, R7, NEXT
        HALT
NEXT:
        LD          R0, @R2
```

Ако работният регистър R7 съдържа стойност 20h (00100000b), то следващата инструкция проверява бит 5 на R7 дали е 1. Тъй като това е така, програмата прескача следващата инструкция и преминава към изпълнение на инструкцията LD R0, @R2.

Обърнете внимание, че вместо директен адрес, в инструкцията се използва идентификатор NEXT. Този идентификатор се нарича етикет. На всяка инструкция може да бъде поставен етикет, който е идентичен с адреса в Програмната Памет, където се разполага инструкцията. Това освобождава програмиста от използването на директни адреси в прогрмата. В нашия пример етикетът NEXT маркира адреса, на който е разположена инструкцията LD R0, @R2. Етикетът и инструкцията могат да бъдат разположени на един ред или на отделни редове както е показано в примера.

Пример:

```
        BTJNZ       3, @R7, NEXT
        HALT
NEXT:
        LD          R0, @R2
```

Ако работният регистър R7 съдържа стойност A5h и регистър A5h съдържа стойност 20h (00100000b), то следващата инструкция проверява бит 3 на регистър 20h дали е 1. Тъй като това не е така, програмата преминава към изпълнение на следващата инструкция, която в нашия пример е HALT.

BTJZ

Синтаксис

BTJZ bit, src, DA

Операция

Ако src[bit] = 0, то PC ← PC + X

Описание

Тази инструкция тества бит bit в операнда src дали е равен на 0. Ако това е така, към Програмния Брояч (който сочи вече към следващата инструкция) се добавя знаково отместване X = DA – PC, което трябва да е в диапазона -128 ÷ +127, и се извършва преход към новия адрес. В противен случай процесорът продължава с изпълнение на инструкцията, която се намира непосредствено след тази инструкция.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	бит	src	адрес	опкод	o1	o2	o3
BTJZ	bit	r2	DA	F6h	{0b, bit, r2}	X	-
BTJZ	bit	@r2	DA	F7h	{0b, bit, r2}	X	-

Пример:

```
        BTJZ    3, R7, NEXT
        HALT
NEXT:
        LD      R0, @R2
```

Ако работният регистър R7 съдържа стойност 20h (00100000b), то следващата инструкция проверява бит 5 на R7 дали е 0. Тъй като това е така, програмата прескача следващата инструкция и преминава към изпълнение на инструкцията LD R0, @R2.

Обърнете внимание, че вместо директен адрес, в инструкцията се използва идентификатор NEXT. Този идентификатор се нарича етикет. На всяка инструкция може да бъде поставен етикет, който е идентичен с адреса в Програмната Памет, където се разполага инструкцията. Това освобождава програмиста от използването на директни адреси в програмата. В нашия пример етикетът NEXT маркира адреса, на който е разположена инструкцията LD R0, @R2. Етикетът и инструкцията могат да бъдат разположени на един ред или на отделни редове както е показано в примера.

Пример:

```
        BTJZ        5, @R7, NEXT
        HALT
NEXT:
        LD          R0, @R2
```

Ако работният регистър R7 съдържа стойност A5h и регистър A5h съдържа стойност 20h (00100000b), то следващата инструкция проверява бит 5 на регистър 20h дали е 0. Тъй като това не е така, програмата преминава към изпълнение на следващата инструкция, която в нашия пример е HALT.

CALL

Синтаксис

CALL dst

Операция

SP ← SP – 2
@SP ← PC
PC ← dst

Описание

Извикване на подпрограма. Указателят към Стека SP се намалява с 2 и Програмният Брояч PC (който съдържа адреса на първата инструкция след инструкцията CALL) се записва в стека и указания с dst адрес се записва в PC. Сега PC сочи към първата инструкция на подпрограмата.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	опкод	o1	o2	o3
CALL	@RR1	D4h	RR1	-	-
CALL	DA	D6h	DA[15:8]	DA[7:0]	-

Адресният режим IR може да използва Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

CALL @00h

Ако двойката регистри 00h:01h съдържа стойност 1234h, горната инструкция съхранява текущата стойност на Програмния Брояч в стека и го зарежда със стойността 1234h. Програмата се прехвърля за изпълнение на подпрограмата, разположена на този адрес.

Пример:

CALL @E0h

Ако двойката работни регистри R0:R1 (използва се Escaped-адресация) съдържа стойност 1234h, горната инструкция съхранява текущата стойност на Програмния Брояч в стека и го зарежда със стойността 1234h. Програмата се прехвърля за изпълнение на подпрограмата, разположена на този адрес.

Тази инструкция е еквивалентна на: CALL @RR0

Пример:

```
        CALL        MySubroutine
        …
MySubroutine:
        инструкция1
        …
        инструкцияN
        RET
```

Обърнете внимание, че вместо директен адрес, в инструкцията се използва идентификатор MySubroutine. Този идентификатор се нарича етикет. На всяка инструкция може да бъде поставен етикет, който е идентичен с адреса в Програмната Памет, където се разполага инструкцията. Това освобождава програмиста от използването на директни адреси в програмата. В нашия пример етикетът

MySubroutine се асоциира с адреса, на който е разположена първата инструкция на извикваната подпрограма. Етикетът и инструкцията могат да бъдат разположени на един ред или на отделни редове както е показано в примера.

CCF

Синтаксис

CCF

Операция

C ← ~C

Описание

Инвертиране на флага C. Ако C = 1, флагът се нулира. Ако C = 1, флагът се установява.

Флагове

C ~C
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
CCF	-	-	EFh	-	-	-

Пример:

CCF

Ако флагът C е 0, горната инструкция установява флага C.

CLR

Синтаксис

CLR dst

Операция

dst ← 0

Описание

Нулиране на операнда dst.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	опкод	o1	o2	o3
CLR	R1	B0h	R1	-	-
CLR	@R1	B1h	R1	-	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

CLR FFh

Ако регистър FFh съдържа стойност AFh, горната инструкция нулира регистър FFh.

Пример:

CLR   R6

Ако работният регистър R6 (използва се Escaped-адресация) съдържа стойност AFh, горната инструкция нулира R6.

Пример:

CLR   @A5h

Ако работният регистър A5h съдържа стойност 23h и регистър 23h съдържа стойност FCh, горната инструкция нулира регистър 23h.

COM

Синтаксис

COM dst

Операция

dst ← ~dst

Описание

Инвертиране на операнда dst.

Флагове

C	-
Z	1 ако резултатът е 0, иначе е 0
S	1 ако бит 7 на резултата е 1, иначе е 0
V	0
D	-
H	-

абревиатура	dst	опкод	o1	o2	o3
COM	R1	60h	R1	-	-
COM	@R1	61h	R1	-	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

COM 08h

Ако регистър 08h съдържа стойност 24h (00100100b), горната инструкция записва DBh (11011011b) в регистър 08h, установява флаг S и нулира флаговете Z и V.

Пример:

COM R8

Ако работният регистър R8 (използва се Escaped-адресация) съдържа стойност 24h (00100100b), горната инструкция записва DBh (11011011b) в R8, установява флаг S и нулира флаговете Z и V.

Пример:

COM @08h

Ако регистър 08h съдържа стойност 24h и регистър 24h съдържа стойност FFh (11111111b), горната инструкция записва 00h (00000000h) в регистър 24h, установява флаг Z и нулира флаговете V и S.

CP

Синтаксис

CP dst, src

Операция

dst - src

Описание

Сравнение на операнди. Операндът src се сравнява с (се изважда от) операнда dst и флаговете се установяват/нулират в зависимост от резултата. Може да използвате инструкциите **JP cc** или **JR cc**, за да извършите преход към определена част на програмата в зависимост от резултата от инструкцията CP.

Флагове

C 1 ако има заявка за заем от бит 7, иначе е 0
Z 1 ако резултатът е 0, иначе е 0
S 1 ако бит 7 на резултата е 1, иначе е 0
V 1 ако има аритметично препълване, иначе е 0
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
CP	r1	r2	A2h	(r1, r2)	-	-
CP	r1	r2	A3h	(r1, r2)	-	-
CP	R1	R2	A4h	R2	R1	-
CP	R1	@R2	A5h	R2	R1	-
CP	R1	IM	A6h	R1	IM	-
CP	@R1	IM	A7h	R1	IM	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

CP   R3, R11

Ако работният регистър R3 съдържа стойност 16h и работният регистър R11 съдържа стойност 20h, горната инструкция установява флагове C и S, и нулира флаговете Z и V.

Пример:

CP   R15, @R10

Ако работният регистър R15 съдържа стойност 16h, работният регистър R10 съдържа стойност 20h и регистър 20h съдържа стойност 11h, горната инструкция нулира флаговете C, Z, S и V.

Пример:

CP   34h, 12h

Ако регистър 34h съдържа стойност 2Eh и регистър 12h съдържа стойност 1Bh, горната инструкция нулира флаговете C, Z, S и V.

Пример:

CP   4Bh, @R3

Ако регистър 4Bh съдържа стойност 82h, работният регистър R3 съдържа стойност 10h и регистър 10h съдържа стойност 01h, горната инструкция установява флаг S и нулира флаговете C, Z и V.

Пример:

CP   6Ch, #2Ah

Ако регистър 6Ch съдържа стойност 2Ah, горната инструкция установява флаг Z и нулира флаговете C, S и V.

Пример:

CP @D4h, #FFh

Ако регистър D4h съдържа стойност FCh и регистър FCh съдържа стойност 8Fh, горната инструкция установява флаг V и нулира флаговете C, Z и S.

CPC

Синтаксис

CPC dst, src

Операция

dst - src - C

Описание

Сравнение на операнди с пренос. Операндът src и флагът C се сравняват с (се изваждат от) операнда dst и флаговете се установяват/нулират в зависимост от резултата. Тази инструкция може да се използва за сравняване на многобайтови променливи. Флаг Z се установява в 1 само ако първоначалното състояние на флага е 1 и резултатът от сравнението е 0. Може да използвате инструкциите **JP cc** или **JR cc**, за да извършите преход към определена част на програмата в зависимост от резултата от инструкцията CPC.

Флагове

C 1 ако има заявка за заем от бит 7, иначе е 0

Z 1 ако резултатът е 0 и първоначалното състояние на флага Z е 1, иначе е 0

S 1 ако резултатът е отрицателен, иначе е 0

V 1 ако има аритметично препълване, иначе е 0

D -

H -

абревиатура	dst	src	опкод	o1	o2	o3
CPC	r1	r2	1FA2h	(r1, r2)	-	-
CPC	r1	@r2	1FA3h	(r1, r2)	-	-
CPC	R1	R2	1FA4h	R2	R1	-
CPC	R1	@R2	1FA5h	R2	R1	-
CPC	R1	IM	1FA6h	R1	IM	-
CPC	@R1	IM	1FA7h	R1	IM	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

CPC R3, R11

Ако работният регистър R3 съдържа стойност 16h, работният регистър R11 съдържа стойност 20h и флагът C е 1, горната инструкция установява флагове C и S, и нулира флаговете Z и V.

Пример:

CPC R15, @R10

Ако работният регистър R15 съдържа стойност 16h, работният регистър R10 съдържа стойност 20h, регистър 20h съдържа стойност 11h и флагът C е 0, горната инструкция нулира флаговете C, Z, S и V.

Пример:

CPC 34h, 12h

Ако регистър 34h съдържа стойност 2Eh, регистър 12h съдържа стойност 1Bh и флагът C е 1, горната инструкция нулира флаговете C, Z, S и V.

Пример:

CPC 4Bh, @R3

Ако регистър 4Bh съдържа стойност 82h, работният регистър R3 съдържа стойност 10h, регистър 10h съдържа стойност 81h, флаг C е 1 и флаг Z е 0, горната инструкция

установява флаг Z и нулира флаговете C, S и V.

Пример:

CPC 6Ch, #2Ah

Ако регистър 6Ch съдържа стойност 2Ah, флагът C е 0, и флагът Z е 1, горната инструкция нулира флаговете C, Z, S и V.

Пример:

CPC @D4h, #FFh

Ако регистър D4h съдържа стойност FCh, регистър FCh съдържа стойност 8Fh и флаг C е 0, горната инструкция установява флаг V и нулира флаговете C, Z и S.

CPCX

Синтаксис

CPCX dst, src

Операция

dst - src - C

Описание

Сравнение на операнди с пренос с използване на разширена (12-битови адреси) адресация. Операндът src и флагът C се сравняват с (се изваждат от) операнда dst и флаговете се установяват/нулират в зависимост от резултата. Тази инструкция може да се използва за сравняване на многобайтови променливи. Флаг Z се установява в 1 само ако първоначалното състояние на флага е 1 и резултатът от сравнението е 0. Може да използвате инструкциите **JP cc** или **JR cc**, за да извършите преход към определена част на програмата в зависимост от резултата от инструкцията CPCX.

Флагове

- C 1 ако има заявка за заем от бит 7, иначе е 0
- Z 1 ако резултатът е 0 и първоначалното състояние на флага Z е 1, иначе е 0
- S 1 ако резултатът е отрицателен, иначе е 0
- V 1 ако има аритметично препълване, иначе е 0
- D -
- H -

абревиатура	dst	src	опкод	o1	o2	o3
CPCX	ER1	ER2	1FA8h	ER2[11:4]	{ER2[3:0], ER1[11:8]}	ER1[7:0]
CPCX	ER1	IM	1FA9h	IM	{0, ER1[11:8]}	ER1[7:0]

Адресният режим ER може да използва Escaped-адресация.

Ако старшият байт на адреса е EEh, младшият полубайт указва работен регистър.

Пример:

CPCX AB3h, 911h

Ако регистър AB3h съдържа стойност 16h, регистър 911h съдържа стойност 20h и флаг C е 1, горната инструкция установява флаговете C и S, и нулира флаговете Z и V.

Пример:

CPCX EE0h, 911h

Ако работният регистър R0 (използва се Escaped-адресация) съдържа стойност 16h, регистър 911h съдържа стойност 20h и флаг C е 1, горната инструкция установява флаговете C и S, и нулира флаговете Z и V.

CPCX 26Ch, #2Ah

Ако регистър 26Ch съдържа стойност 2Ah, флаг C е 0 и флаг Z е 0, горната инструкция установява флага Z, и нулира флаговете C, S и V.

CPX

Синтаксис

CPX dst, src

Операция

dst - src

Описание

Сравнение на операнди с използване на разширена (12-битови адреси) адресация. Операндът src се сравнява с (се изважда от) операнда dst и флаговете се установяват/нулират в зависимост от резултата. Може да използвате инструкциите **JP cc** или **JR cc**, за да извършите преход към определена част на програмата в зависимост от резултата от инструкцията CPX.

Флагове

C 1 ако има заявка за заем от бит 7, иначе е 0
Z 1 ако резултатът е 0, иначе е 0
S 1 ако резултатът е отрицателен, иначе е 0
V 1 ако има аритметично препълване, иначе е 0
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
CPX	ER1	ER2	A8h	ER2[11:4]	{ER2[3:0], ER1[11:8]}	ER1[7:0]
CPX	ER1	IM	A9h	IM	{0, ER1[11:8]}	ER1[7:0]

Адресният режим ER може да използва Escaped-адресация. Ако старшият байт на адреса е EEh, младшият полубайт указва работен регистър.

Пример:

CPX AB3h, 911h

Ако регистър AB3h съдържа стойност 16h и регистър 911h съдържа стойност 20h, горната инструкция установява флаговете C и S, и нулира флаговете Z и V.

Пример:

CPX EE0h, 911h

Ако работният регистър R0 (използва се Escaped-адресация) съдържа стойност 16h и регистър 911h съдържа стойност 20h, горната инструкция установява флаговете C и S, и нулира флаговете Z и V.

CPX 26Ch, #2Ah

Ако регистър 26Ch съдържа стойност 2Ah, горната инструкция установява флага Z и нулира флаговете C, S и V.

DA

Синтаксис

DA   dst

Операция

dst ← DA(dst)

Описание

Операндът dst се коригира, за да се формират две 4-битови BCD[1] цифри, получени след сумиране/изваждане на BCD кодирани байтове. Следващата таблица описва операцията, която се извършва при изпълнение на тази инструкция. Ако стойността в операнда dst не е резултат от валидно събиране/изваждане на BCD цифри, резултатът от тази инструкция е недефиниран.

инструкция	Флаг C преди DA	Битове 7-4 (hex)	Флаг H преди DA	Битове 3-0 (hex)	Число добавено към резултата (hex)	Флаг C след DA
ADD/ADC	0	0-9	0	0-9	00	0
	0	0-8	0	A-F	06	0
	0	0-9	1	0-3	06	0
	0	A-F	0	0-9	60	1
	0	9-F	0	A-F	66	1
	0	A-F	1	0-3	66	1
	1	0-2	0	0-9	60	1
	1	0-2	0	A-F	66	1
	1	0-3	1	0-3	66	1
SUB/SBC	0	0-9	0	0-9	00	0
	0	0-8	1	6-F	FA	0
	1	7-F	0	0-9	A0	1
	1	6-F	1	6-F	9A	1

Флагове

C   1 ако има пренос от бит 7, иначе е 0

Z 1 ако резултатът е 0, иначе е 0
S 1 ако бит 7 на резултата е 0, иначе е 0
V x
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
DA	R1	-	40h	R1	-	-
DA	@R1	-	41h	R1	-	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

Нека имаме две BCD стойности 15h и 27h, които искаме да съберем.

```
 0001 0101 = 15h
+
 0010 0111 = 27h
 ---------------------
 0011 1100 = 3Ch
```

Резултатът трябва да е 42h, но както се вижда от горния пример резултатът е 3Ch, заради стандартното двоично събиране (процесорът разглежда двоичните числа 0001 0101 и 0011 1100 като стандартни двоични числа със стойности 21 и 39 съответно, а не като BCD числа. Резултатът от сумирането на тези числа е 60 (3Ch)). За да се получи правилен BCD резултат след събирането, е необходимо да се изпълни инструкцията DA. Нека допуснем, че резултатът от събирането се намира в регистър 5Fh.

```
DA    5Fh
```

Горната инструкция коригира резултата, така че след изпълнението й в регистър 5Fh се получава правилния BCD резултат 42h и флаговете C, Z и S се нулират.

```
  0011 1100 = 3Ch
+
  0000 0110 = 06h
  ---------------------
  0100 0010 = 42h
```

Забележка[1]: **BCD (Binary Coded Decimal)** е двоичен код, в който всяка цифра на десетично цяло число се представя с 4-битов двоичен код.

десетична бройна система	двоична бройна система	BCD код
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	0101
6	0110	0110
7	0111	0111
8	1000	1000
9	1001	1001
10	1010	0001 0000
11	1011	0001 0001
12	1100	0001 0010
13	1101	0001 0011
14	1110	0001 0100
15	1111	0001 0101
...	...	...

255	11111111	0010 0101 0101

Аритметика с BCD-числа не е разгледана в тази книга.

DEC

Синтаксис

DEC dst

Операция

dst ← dst - 1

Описание

Съдържанието на операнда dst се намалява с 1 и резултатът отново се съхранява в dst.

Флагове

C -
Z 1 ако резултатът е 0, иначе е 0
S 1 ако бит 7 на резултата е 1, иначе е 0
V 1 ако има аритметично препълване, иначе е 0
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
DEC	R1	-	30h	R1	-	-
DEC	@R1	-	31h	R1	-	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

DEC 00h

Ако регистър 00h съдържа стойност 2Ah, горната инструкция записва стойност 29h в регистър 00h и нулира флаговете Z, V и S.

Пример:

DEC R0

Ако работният регистър R0 (използва се Escaped-адресация) съдържа стойност 2Ah, горната инструкция записва стойност 29h в R0 и нулира флаговете Z, V и S.

Пример:

DEC @B3h

Ако регистър B3h съдържа стойност CBh и регистър CBh съдържа стойност 01h, горната инструкция записва стойност 00h в регистър CBh, установява флаг Z и нулира флаговете V и S.

DECW

Синтаксис

DECW dst

Операция

dst ← dst - 1

Описание

Съдържанието на 16 битовия операнд dst се намалява с 1 и резултатът отново се съхранява в dst.

Флагове

C	-
Z	1 ако резултатът е 0, иначе е 0
S	1 ако бит 7 на резултата е 1, иначе е 0
V	1 ако има аритметично препълване, иначе е 0
D	-
H	-

абревиатура	dst	src	опкод	o1	o2	o3
DECW	RR1	-	80h	RR1	-	-
DECW	@R1	-	81h	R1	-	-

Адресните режими RR и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

DECW 30h

Ако двойката регистри 30h:31h съдържа стойност 0AF2h, горната инструкция записва стойност 0AF1h в регистрите 30h:31h и нулира флаговете Z, V и S.

Пример:

DECW RR0

Ако двойката работни регистри R0:R1 (използва се Escaped-адресация) съдържат стойност 0AF2h, горната инструкция записва стойност 0AF1h в R0:R1 и нулира флаговете Z, V и S.

Пример:

DECW @80h

Ако регистър 80h съдържа стойност 30h, и двойката регистри 30h:31h съдържат стойност FAF3h, горната инструкция записва стойност FAF2h в регистрите 30h:31h, установява флаг S и нулира флаговете Z и V.

Пример:

DECW @R0

Ако работният регистър R0 (използва се Escaped-адресация) съдържа стойност 30h, и двойката регистри 30h:31h съдържат стойност FAF3h, горната инструкция записва стойност FAF2h в регистрите 30h:31h, установява флаг S и нулира флаговете Z и V.

DI

Синтаксис

DI

Операция

IRQCTL[7] ← 0

Описание

Бит 7 на регистър IRQCTL се нулира. Това забранява всички прекъсвания.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2o	o3
DI	-	-	8Fh	-	-	-

Пример:

```
DI      ; IRQCL[7] = 0, прекъсванията са забранени
```

DJNZ

Синтаксис

DJNZ dst, RA

Операция

dst ← dst – 1

Ако dst ≠ 0, то PC ← PC + X, където X = RA - PC

Описание

Операндът dst се намалява с 1 и резултатът се съхранява отново в dst. Ако резултатът е различен от 0, към Програмния Брояч (който сочи вече към следващата инструкция) се добавя знаково отместване X = RA – PC, което трябва да е в диапазона -128 ÷ +127, и се извършва преход към новия адрес. В противен случай процесорът продължава с изпълнение на инструкцията, която се намира непосредствено след тази инструкция.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	адрес	опкод	о1	о2	о3
DJNZ	r0÷r15	RA	0Ah÷FAh	X	-	-

Пример:

loop: DJNZ R0, loop

Ако работният регистър R0 съдържа стойност 02h, горната инструкция записва стойност 01h в R0 и отново се изпълнява същата инструкция. При повторното изпълнение на инструкцията стойността в R0 става 0 и програмата продължава със следващата инструкция, която се намира непосредствено след тази инструкция.

EI

Синтаксис

EI

Операция

IRQCTL[7] ← 1

Описание

Бит 7 на регистър IRQCTL се установява. Това разрешава прекъсванията.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
EI	-	-	9Fh	-	-	-

Пример:

```
EI      ; IRQCL[7] = 1, прекъсванията са разрешени
```

HALT

Синтаксис

HALT

Операция

Влизане в HALT-режим

Описание

Тази инструкция поставя процесора eZ8 в HALT-режим (вижте 25.8 Режими с ниска консумация).

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
HALT	-	-	7Fh	-	-	-

Пример:

```
HALT        ; процесорът eZ8 влиза в HALT-режим
```

INC

Синтаксис

INC dst

Операция

dst ← dst + 1

Описание

Съдържанието на операнда dst се увеличава с 1 и резултатът отново се съхранява в dst.

Флагове

C -
Z 1 ако резултатът е 0, иначе е 0
S 1 ако бит 7 на резултата е 1, иначе е 0
V 1 ако има аритметично препълване, иначе е 0
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
INC	R1	-	20h	R1	-	-
INC	@R1	-	21h	R1	-	-
INC	r0÷r15	-	0E÷FE	-	-	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

INC 00h

Ако регистър 00h съдържа стойност 2Ah, горната инструкция записва стойност 2Bh в регистър 00h и нулира флаговете Z, V и S.

Пример:

INC B3h

Ако регистър B3h съдържа стойност CBh, горната инструкция записва стойност CCh в регистър CBh, установява флаг S и нулира флаговете Z и V.

Пример:

INC @B3h

Ако регистър B3h съдържа стойност CBh и регистър CBh съдържа стойност FFh, горната инструкция записва стойност 00h в регистър CBh, установява флаг Z и нулира флаговете V и S.

Пример:

INC @R0

Ако регистър R0 (Използва се Escaped-адресация) съдържа стойност CBh и регистър CBh съдържа стойност FFh, горната инструкция записва стойност 00h в регистър CBh, установява флаг Z и нулира флаговете V и S.

Пример:

INC R0

Ако работният регистър R0 съдържа стойност 2Ah, горната инструкция записва стойност 2Bh в R0 и нулира флаговете Z, V и S.

INCW

Синтаксис

INCW dst

Операция

dst ← dst + 1

Описание

Съдържанието на 16-битовия операнд dst се увеличава с 1 и резултатът отново се съхранява в dst.

Флагове

C	-
Z	1 ако резултатът е 0, иначе е 0
S	1 ако бит 7 на резултата е 1, иначе е 0
V	1 ако има аритметично препълване, иначе е 0
D	-
H	-

абревиатура	dst	src	опкод	o1	o2	o3
INCW	RR1	-	A0h	RR1	-	-
INCW	@R1	-	A1h	R1	-	-

Адресните режими RR и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

INCW 30h

Ако двойката регистри 30h:31h съдържа стойност 0AF2h, горната инструкция записва стойност 0AF3h в регистрите 30h:31h и нулира флаговете Z, V и S.

Пример:

INCW   RR0

Ако двойката работни регистри R0:R1 (използва се Escaped-адресация) съдържат стойност 0AF1h, горната инструкция записва стойност 0AF2h в R0:R1 и нулира флаговете Z, V и S.

Пример:

INCW   @80h

Ако регистър 80h съдържа стойност 30h, и двойката регистри 30h:31h съдържат стойност FAF2h, горната инструкция записва стойност FAF3h в регистрите 30h:31h, установява флаг S и нулира флаговете Z и V.

Пример:

INCW   @R0

Ако работният регистър R0 (използва се Escaped-адресация) съдържа стойност 30h, и двойката регистри 30h:31h съдържат стойност FAF2h, горната инструкция записва стойност FAF3h в регистрите 30h:31h, установява флаг S и нулира флаговете Z и V.

IRET

Синтаксис

IRET

Операция

FLAGS ← @SP
SP ← SP + 1
PC ← @SP
SP ← SP + 2
IRQCTL[7] ← 1

Описание

Тази инструкция се използва за излизане от подпрограма за обработка на прекъсване. Изпълнението й води до възстановяване на регистрите FLAGS и PC от стека и разрешаване на прекъсванията, чрез установяване на бит 7 в регистър IRQCTL. За повече информация вижте също 21.3 Стек.

Флагове

FLAGS ← @SP

абревиатура	dst	src	опкод	o1	o2	o3
IRET	-	-	BFh	-	-	-

```
;Подпрограма за обработка на прекъсване
IrqHandler:
        инструкция1
        ...
        инструкцияN
        IRET ;връщане в точката на прекъсване
```

JP

Синтаксис

JP dst

Операция

PC ← dst

Описание

Инструкция за безусловен преход. Тази инструкция записва Програмния Брояч PC с адреса указан с dst. Програмата се прехвърля за изпълнение на инструкцията, намираща се на този адрес.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
JP	DA	-	8Dh	DA[15:8]	DA[15:7]	-
JP	@RR1	-	C4h	RR1	-	-

Адресният режим RR може да използва Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

JP 0100h

Програмният Брояч се зарежда със стойност 0100h. Програмата се прехвърля за изпълнение на инструкцията,

разположена на този адрес.

Пример:

JP @00h

Ако двойката регистри 00h:01h съдържа стойност 0100h, горната инструкция зарежда Програмния Брояч със стойността 0100h и програмата се прехвърля за изпълнение на инструкцията, разположена на този адрес.

JP @E0h

Ако двойката работни регистри R0:R1 (използва се Escaped-адресация) съдържа стойност 0100h, горната инструкция зарежда Програмния Брояч със стойността 0100h и програмата се прехвърля за изпълнение на инструкцията, разположена на този адрес.

Тази инструкция е еквивалентна на : JP @RR0

Пример:

JP @RR2

Ако двойката работни регистри R2:R3 (използва се Escaped-адресация) съдържа стойност 3F45h, горната инструкция зарежда Програмния Брояч със стойността 3F45h и програмата се прехвърля за изпълнение на инструкцията, намираща се на този адрес.

JP cc

Синтаксис

JP cc, dst

Операция

Ако тест-условието cc е ИСТИНА (1), то PC ← dst

Описание

Тази инструкция записва Програмния Брояч PC с адреса указан с dst, ако тест-условието cc е 1 (вижте Табл.28). Програмата се прехвърля за изпълнение на инструкцията, намираща се на този адрес. В противен случай програмата преминава към изпълнение на следващата инструкция, намираща се непосредствено след тази инструкция.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	тест-условие	адрес	опкод	o1	o2	o3
JP	cc	DA	0Dh÷FDh	DA[15:8]	DA[7:0]	-

Пример:

```
        JP      Z, NEXT ;преход към инструкция1, ако Z = 1
        инструкция1
NEXT:
        инструкция2
```

JR

Синтаксис

JR DA

Операция

PC ← PC + X, където X = DA - PC

Описание

Тази инструкция добавя към Програмния Брояч (който сочи вече към следващата инструкция) знаково отместване X = DA – PC, което трябва да е в диапазона -128 ÷ +127, и се извършва преход към новия адрес.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	адрес	src	опкод	o1	o2	o3
JR	DA	-	8Bh	X		

```
NEXT:
        инструкция1
        ...
        инструкцияN
        JR      NEXT ;преход към инструкция1
```

JR cc

Синтаксис

JR cc, DA

Операция

Ако тест-условието cc е ИСТИНА (1), то PC ← PC + X, където X = DA - PC

Описание

Тази инструкция добавя към Програмния Брояч (който сочи вече към следващата инструкция) знаково отместване X = DA – PC, което трябва да е в диапазона -128 ÷ +127, ако тест-условието cc е 1 (вижте Табл.28), и се извършва преход към новия адрес. В противен случай програмата преминава към изпълнение на следващата инструкция, намираща се непосредствено след тази инструкция.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	тест-условие	адрес	опкод	o1	o2	o3
JR	cc	DA	0Bh÷FBh	X	-	-

NEXT:

```
	инструкция1
	...
	инструкцияN
	JR	Z, NEXT ;преход към инструкция1, ако Z = 1
```

LD

Синтаксис

LD dst, src

Операция

dst ← src

Описание

Съдържанието на операнда src се записва в операнда dst.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
LD	r1	@r2	E3h	{r1, r2}	-	-
LD	R1	R2	E4h	R2	R1	-
LD	R1	@R2	E5h	R2	R1	-
LD	R1	IM	E6h	R1	IM	-
LD	@R1	IM	E7h	R1	IM	-
LD	@r1	r2	F3h	{r1, r2}	-	-
LD	@R1	R2	F5h	R2	R1	-
LD	r1	X(r2)	C7h	{r1, r2}	X	-
LD	X(r1)	r2	D7h	{r1, r2}	X	-
LD	r0÷r15	IM	0Ch÷FCh	IM	-	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

```
LD    R15, #34h
```

Горната инструкция записва стойност 34h в работния регистър R15.

Пример:

```
LD    R14, 34h
```

Ако регистър 34h съдържа стойност FCh, горната инструкция записва стойността FCh в работния регистър R14.

Пример:

```
LD    34h, R14
```

Ако работният регистър R14 съдържа стойност 45h, горната инструкция записва стойността 45h в регистър 34h.

Пример:

```
LD    R13, @R12
```

Ако работният регистър R12 съдържа стойност 34h и регистър 34h съдържа стойност FFh, горната инструкция записва стойността FFh в работния регистър R13.

Пример:

```
LD    @R13, R12
```

Ако работният регистър R13 съдържа стойност 45h и работният регистър R12 съдържа стойност 00h, горната инструкция записва стойността 00h в регистър 45h.

Пример:

LD 34h, 45h

Ако регистър 45h съдържа стойност CFh, горната инструкция записва стойността CFh в регистър 34h.

Пример:

LD 34h, @45h

Ако регистър 45h съдържа стойност CFh и регистър CFh съдържа стойност FFh, горната инструкция записва стойността FFh в регистър 34h.

Пример:

LD 34h, #A4h

Горната инструкция записва стойността A4h в регистър 34h.

Пример:

LD @R14, #FCh

Ако работният регистър R14 съдържа стойност 7Fh, горната инструкция записва стойността FCh в регистър 7Fh.

Пример:

LD @34h , 45h

Ако регистър 34h съдържа стойност CFh и регистър 45h съдържа стойност FFh, горната инструкция записва стойността FFh в регистър CFh.

Пример:

LD R10, 24h (R0)

Ако работният регистър R0 съдържа стойност 08h и регистър 2Ch (24h + 08h = 2Ch) съдържа стойност 4Fh, горната инструкция записва стойността 4Fh в работния регистър R10.

Пример:

LD F0h (R0), R10

Ако работният регистър R0 съдържа стойност 08h и работният регистър R10 съдържа стойност 83h, горната инструкция записва стойността 83h в регистър F8h (F0h + 08h = F8h).

LDC

Синтаксис

LDC dst, src

Операция

dst ← src

Описание

Зареждане на константен байт от Програмната Памет в работен регистър и обратно. Адресът в Програмната Памет се задава чрез двойка работни регистри.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
LDC	r1	@rr2	C2h	{r1, rr2}	-	-
LDC	@r1	@rr2	C5h	{r1, rr2}	-	-
LDC	@rr1	r2	D2h	{r2, rr1}	-	-

Пример:

LDC R2, @RR6

Ако двойката работни регистри R6:R7 съдържа стойност 30A2h и адресът в Програмната Памет 30A2h съдържа стойност 22h, горната инструкция копира стойността 22h в работния регистър R2.

Пример:

LDC @R3, @RR8

Ако работният регистър R3 съдържа стойност A5h, двойката работни регистри R8:R9 съдържа стойност 2A72h и адресът в Програмната Памет 2A72h съдържа стойност 42h, горната инструкция копира стойността 42h в регистър A5h.

Пример:

LDC @RR6, R2

Ако работният регистър R2 съдържа стойност 22h и двойката работни регистри R6:R7 съдържа стойност 10A2h, горната инструкция записва стойността 22h на адрес 10A2h в Програмната Памет.

LDCI

Синтаксис

LDCI dst, src

Операция

dst ← src

r ← r + 1

rr ← rr + 1

Описание

Тази инструкция се използва за извършване на трансфер на данни между Програмната Памет и Регистровата Памет. Адресът в Програмната Памет се задава чрез двойка работни регистри, а адресът в Регистровата Памет се задава чрез работен регистър. Адресите в работните регистри автоматично се увеличават към следващите адреси след изпълнение на инструкцията.

Флагове

C -

Z -

S -

V -

D -

H -

абревиатура	dst	src	опкод	o1	o2	o3
LDCI	@r1	@rr2	C3h	{r1, rr2}	-	-
LDCI	@rr1	@r2	D3h	{r2, rr1}	-	-

Пример:

LDCI @R2, @RR6

Ако двойката работни регистри R6:R7 съдържа стойност 30A2h, адресът 30A2h в Програмната Памет съдържа стойност 22h и работният регистър R2 съдържа стойност 20h, горната инструкция копира стойността 22h в регистър 20h. Двойката работни регистри R6:R7 се увеличава до 30A3h и работният регистър R2 се увеличава до 21h. Многократното изпълнение на тази инструкция ще доведе до копиране на блок от данни от Програмната Памет в Регистровата Памет.

Пример:

LDCI @RR6, @R2

Ако работният регистър R2 съдържа стойност 20h, регистър 20h съдържа стойност 22h и двойката работни регистри R6:R7 съдържа стойност 30A2h, горната инструкция записва стойността 22h на адрес 30A2h в Програмната Памет. Работният регистър R2 се увеличава до 21h и двойката работни регистри R6:R7 се увеличава до 30A3h. Многократното изпълнение на тази инструкция ще доведе до копиране на блок от данни от Регистровата Памет в Програмната Памет.

LDE

Синтаксис

LDE dst, src

Операция

dst ← src

Описание

Зареждане на байт от Паметта за Данни в работен регистър и обратно. Адресът в Паметта за Данни се задава чрез двойка работни регистри.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
LDE	r1	@rr2	82h	{r1, rr2}	-	-
LDE	@rr1	r2	92h	{r2, rr1}	-	-

Пример:

LDE R2, @RR6

Ако двойката работни регистри R6:R7 съдържа стойност 40A2h и адресът в Паметта за Данни 40A2h съдържа стойност 22h, горната инструкция копира стойността 22h в работния регистър R2.

Пример:

LDE @RR6, R2

Ако работният регистър R2 съдържа стойност 22h и двойката работни регистри R6:R7 съдържа стойност 404Ah, горната инструкция записва стойността 22h на адрес 404Ah в Паметта за Данни.

LDEI

Синтаксис

LDEI dst, src

Операция

dst ← src

r ← r + 1

rr ← rr + 1

Описание

Тази инструкция се използва за извършване на трансфер на данни между Паметта за Данни и Регистровата Памет. Адресът в Паметта за Данни се задава чрез двойка работни регистри, а адресът в Регистровата Памет се задава чрез работен регистър. Адресите в работните регистри автоматично се увеличават към следващите адреси след изпълнение на инструкцията.

Флагове

C -

Z -

S -

V -

D -

H -

абревиатура	dst	src	опкод	o1	o2	o3
LDEI	@r1	@rr2	83h	{r1, rr2}	-	-
LDEI	@rr1	@r2	93h	{r2, rr1}	-	-

Пример:

LDEI @R2, @RR6

Ако двойката работни регистри R6:R7 съдържа стойност 404Ah, адресът 404Ah в Паметта за Данни съдържа стойност ABh и работният регистър R2 съдържа стойност 22h, горната инструкция копира стойността ABh в регистър 22h. Двойката работни регистри R6:R7 се увеличава до 404Bh и работният регистър R2 се увеличава до 23h. Многократното изпълнение на тази инструкция ще доведе до копиране на блок от данни от Паметта за Данни в Регистровата Памет.

Пример:

LDEI @RR6, @R2

Ако работният регистър R2 съдържа стойност 22h, регистър 22h съдържа стойност ABh и двойката работни регистри R6:R7 съдържа стойност 404Ah, горната инструкция записва стойността ABh на адрес 404Ah в Паметта за Данни. Работният регистър R2 се увеличава до 23h и двойката работни регистри R6:R7 се увеличава до 404Bh. Многократното изпълнение на тази инструкция ще доведе до копиране на блок от данни от Регистровата Памет в Паметта за Данни.

LDWX

Синтаксис

LDWX dst, src

Операция

dst ← src

Описание

Тази инструкция се използва за копиране на 16-битови думи от един адрес на друг в Регистровата Памет, чрез използване на разширена (12-битова) регистрова адресация. Адресите на операндите src и dst трябва да бъдат четни (най-младшия бит на адреса трябва да е 0).

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
LDWX	ER1	ER2	1FE8h	ER2[11:4]	{ ER2[3:0], ER1[11:8]}	ER1[7:0]

Адресният режим ER може да използва Escaped-адресация. Ако старшият байт на адреса е EEh, младшият полубайт указва работен регистър.

Пример:

LDWX 000h, 002h

Ако двойката регистри 002h:003h съдържат стойност 1234h, горната инструкция копира стойността 1234h в двойката регистри 000h:001h.

Пример:

LDWX	RR0, RR2

Ако двойката работни регистри R2:R3 (използва се Escaped-адресация) съдържа стойност 1234h, горната инструкция копира стойността 1234h в двойката работни регистри R0:R1 (използва се Escaped-адресация).

LDX

Синтаксис

LDX dst, src

Операция

dst ← src

Описание

Тази инструкция копира съдържанието на операнда src в операнда dst. Основната цел на тази инструкция е прехвърляне на данни от една страница в друга на Регистровата Памет.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
LDX	r1	ER2	84h	{r1, ER2[11:8]}	ER2[7:0]	-
LDX	@r1	ER2	85h	{r1, ER[11:8]}	ER2[7:0]	-
LDX	R1	@RR2	86h	RR2	R1	-
LDX	@R1	@.ER(RR2)	87h[1]	RR2	R1	-
LDX	r1	X(rr2)	88h	{r1, rr2}	X	-
LDX	X(rr1)	r2	89h	{r1, rr2}	X	-
LDX	ER1	r2	94h	{r2, ER1[11:8]}	ER1[7:0]	-
LDX	ER1	@r2	95h	{r2, ER1[11:8]}	ER1[7:0]	-
LDX	@RR1	R2	96h	R2	RR1	-
LDX	@.ER(RR1)	@R2	97h[1]	R2	RR1	-
LDX	ER1	ER2	E8h[2]	ER2[11:4]	{ER2[3:0],	ER1[7:0]

					ER1[11:8]}	
LDX	ER1	IM	E9h[2]	IM	{0, ER1[11:8]}	ER1[7:0]

Забележка[1]: Ако разгледате таблицата с всички инструкции на процесора eZ8, ще забележите, че LDX е единствената инструкция, която използва едновременно адресните режими IR и IRR, т.е. . **LDX IR, IRR** и **LDX IRR, IR**, където двата операнда използват косвена адресация. За асемблера обаче и двете инструкции изглеждат по един и същ начин, т.е. . като **LDX @Reg, @Reg**, поради което асемблерът няма как да направи разлика между двете инструкции. За да се избегне този проблем, операндът IRR за тази инструкция използва @.ER(RR) нотация вместо @Reg, т.е. .

LDX @Reg, @.ER(RR) и **LDX @.ER(RR), @Reg**

По този начин асемблерът може да разпознае, кой операнд използва адресния режим IRR и кой IR.

Забележка[2]: Само тези LDX инструкции могат да използват Escaped-адресацията.

Пример:

LDX R1, 702h

Ако регистър 702h съдържа стойност B3h, горната инструкция записва стойността B3h в работния регистър R1.

Пример:

LDX @R1, 702h

Ако работният регистър R1 съдържа стойност B3h и регистър 702h съдържа стойност 46h, горната инструкция записва стойността 46h в регистър B3h.

Пример:

LDX 96h, @22h

Ако двойката регистри 22h:23h съдържа стойност 0655h и регистър 0655h съдържа стойност 1Ch, горната инструкция записва стойността 1Ch в регистър 96h.

Пример:

LDX @20h, @.ER(F2h)

Ако регистър 20h съдържа стойност 28h, двойката регистри F2h:F3h съдържа стойност 0167h и регистър 0167h съдържа стойност 9Bh, горната инструкция записва стойността 9Bh в регистър 28h.

Обърнете внимание, че .ER() нотацията позволява на асемблера да определи кой от двата операнда трябва да се третира като двойка регистри и кой като единичен регистър.

Пример:

LDX R1, 7(RR10)

Ако двойката работни регистри R10:R11 съдържа стойност 0529h и регистър 0530h (0529h + 7h = 0530h) съдържа стойност C1h, горната инструкция записва стойността C1h в работния регистър R1.

Пример:

LDX 7(RR10), R2

Ако двойката работни регистри R10:R11 съдържа стойност 0529h и работният регистър R2 съдържа стойност E8h, горната инструкция записва стойността E8h в регистър

0530h (0529h + 7h = 0530h).

Пример:

LDX 700h, R6

Ако работният регистър R6 съдържа стойност B4h, горната инструкция записва стойността B4h в регистър 700h.

Пример:

LDX 700h, @R6

Ако работният регистър R6 съдържа стойност B4h и регистър B4h съдържа стойност 6Ah, горната инструкция записва стойността 6Ah в регистър 700h.

Пример:

LDX @F0h, 21h

Ако двойката регистри F0h:F1h съдържа стойност 0296h и регистър 21h съдържа стойност 78h, горната инструкция записва стойността 78h в регистър 0296h

Пример:

LDX @.ER(20h), @F2h

Ако двойката регистри 20h:21h съдържа стойност 0456h, регистър F2h съдържа стойност BBh и регистър BBh съдържа стойност 5Fh, горната инструкция записва стойността 5Fh в регистър 0456h.

Обърнете внимание, че .ER() нотацията позволява на асемблера да определи кой от двата операнда трябва да се третира като двойка регистри и кой като единичен регистър.

Пример:

LDX 702h, 29Ch

Ако регистър 29Ch съдържа стойност 22h, горната инструкция записва стойността 22h в регистър 702h.

Пример:

LDX 703h, #56h

Горната инструкция записва стойността 56h в регистър 703h.

LEA

Синтаксис

LEA dst, X(src)

Операция

dst ← src + X, където X = -128 ÷ +127

Описание

Стойността на операнда src + X се копира в операнда dst.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
LEA	r1	X(r2)	98h	{r1,r2}	X	-
LEA	rr1	X(rr2)	99h	{rr1,rr2}	X	-

Пример:

LEA R11, %15(R3)

Ако работният регистър R3 съдържа стойност 16h, горната инструкция записва стойност 2Bh (15h + 16h = 2Bh) в работния регистър R11.

Пример:

LEA RR14, %79(RR8)

Ако двойката работни регистри R8:R9 съдържа стойност 22ABh, горната инструкция записва стойност 2324h (22ABh + 79h = 2324h) в двойката работни регистри R14:R15.

MULT

Синтаксис

MULT dst

Операция

dst[15:0] ← dst[15:8] * dst[7:0]

Описание

Тази инструкция извършва умножение на два беззнакови 8-битови операнди.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
MULT	RR1	-	F4h	RR1	-	-

Адресният режим RR може да използва Escaped-адреасция. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

MULT 00h

Ако регистър 00h има стойност FFh и регистър 01h има стойност 10h, горната инструкция записва стойност 0FF0 (FFh * 10h = 0FF0h) в двойката регистри 00h:01h (00h=0Fh и 01h=F0h).

Пример:

MULT RR0

Ако работният регистър R0 има стойност FFh и работният регистър R1 има стойност 10h (използва се Escaped-адресация), горната инструкция записва стойност 0FF0 (FFh * 10h = 0FF0h) в двойката работни регистри R0:R1 (R0=0Fh и R1=F0h).

NOP

Синтаксис

NOP

Операция

Празна инструкция

Описание

Празна инструкция. Не извършва никакви действия.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	опкод	o1	o2	o3
NOP	0Fh	-	-	-

OR

Синтаксис

OR dst, src

Операция

dst ← dst OR src

Описание

Логическо ИЛИ. Операндът src се OR-ва с операнда dst и резултатът се съхранява в операнда dst.

Флагове

C -
Z 1 ако резултатът е 0, иначе е 0
S 1 ако бит 7 на резултата е 1, иначе е 0
V 0
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
OR	r1	r2	42h	{r1, r2}	-	-
OR	r1	@r2	43h	{r1, r2}	-	-
OR	R1	R2	44h	R2	R1	-
OR	R1	@R2	45h	R2	R1	-
OR	R1	IM	46h	R1	IM	-
OR	@R1	IM	47h	R1	IM	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

OR R1, R14

Ако работният регистър R1 съдържа стойност 38h

(00111000b) и работният регистър R14 съдържа стойност 8Dh (10001101b), горната инструкция записва стойност BDh (10111101b) в R1, установява флаг S и нулира флаговете Z и V.

Пример:

OR R4, @R13

Ако работният регистър R4 съдържа стойност F9h (11111001b), работният регистър R13 съдържа стойност 7Bh и регистър 7Bh съдържа стойност 6Ah (01101010b), горната инструкция записва стойност FBh (11111011b) в R4, установява флаг S и нулира флаговете Z и V.

Пример:

OR 3Ah, 42h

Ако регистър 3Ah съдържа стойност F5h (11110101b) и регистър 42h съдържа стойност 0Ah (00001010b), горната инструкция записва стойност FFh в регистър 3Ah, установява флаг S и нулира флаговете Z и V.

Пример:

OR E4h, 42h

Използването на адрес E4h активира Escaped-адресация, т.е. . младшият полубайт на адреса указва работен регистър. В нашия случай това е R4. Ако работният регистър R4 съдържа стойност F5h (11110101b) и регистър 42h съдържа стойност 0Ah (00001010b), горната инструкция записва стойност FFh в регистър R4, установява флаг S и нулира флаговете Z и V.

Пример:

OR R5, @45h

Ако регистър R5 (използва се Escaped-адресация) съдържа стойност 70h (01110000b), регистър 45h съдържа стойност 3Ah и регистър 3Ah съдържа стойност 7Fh (01111111b), горната инструкция записва стойност 7Fh (01111111b) в R5 и нулира флаговете Z, V и S.

Пример:

OR 7Ah, #F0h

Ако регистър 7Ah съдържа стойност F7h (11110111b), горната инструкция записва стойност F7h (11110111b) в регистър 7Ah, установява флаг S и нулира флаговете Z и V.

Пример:

OR @R3, #05h

Ако регистър R3 (използва се Escaped-адресация) съдържа стойност 3Eh и регистър 3Eh съдържа стойност 0Ch (00001100b), горната инструкция записва стойност 0Dh (00001101b) в регистър 3Eh и нулира флаговете Z,V и S.

ORX

Синтаксис

ORX dst, src

Операция

dst ← dst OR src

Описание

Логическо ИЛИ. Операндът src се OR-ва с операнда dst и резултатът се съхранява в операнда dst. Операндите dst и src използват разширена (12-битова) адресация за достъп до всеки регистър в Регистровата Памет.

Флагове

C -
Z 1 ако резултатът е 0, иначе е 0
S 1 ако бит 7 на резултата е 1, иначе е 0
V 0
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
ORX	ER1	ER2	48h	ER2[11:4]	{ER2[3:0], ER1[11:8]}	ER1[7:0]
ORX	ER1	IM	49h	IM	{0h, ER1[11:8]}	ER1[7:0]

Адресният режим ER може да използва Escaped-адресация. Ако старшият байт на адреса е EEh, младшият полубайт указва работен регистър.

Пример:

ORX 93Ah, 142h

Ако регистър 93Ah съдържа стойност F5h (11110101b) и регистър 142h съдържа стойност 0Ah (00001010b), горната инструкция записва стойност FFh (11111111b) в регистър 93Ah, установява флага S и нулира флаговете Z и V.

Пример:

ORX D7Ah, #01100000b

Ако регистър D7Ah съдържа стойност 07h (00000111b), горната инструкция записва стойност 67h (01100111b) в регистър D7Ah и нулира флаговете S, Z и V.

POP

Синтаксис

POP dst

Операция

dst ← @SP
SP ← SP + 1

Описание

Стойността на регистъра, сочен от Указателя към Стека SP, се копира в операнда dst и SP се увеличава с 1. Казано с други думи, върхът на стека се копира в операнда dst.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
POP	R1	-	50h	R1	-	-
POP	@R1	-	51h	R1	-	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

POP 34h

Ако SP сочи към регистър 70h и регистър 70h съдържа

стойност 44h, горната инструкция копира стойността 44h в регистър 34h и SP се увеличава с 1 (сочи към регистър 71h).

POPX

Синтаксис

POPX dst

Операция

dst ← @SP
SP ← SP + 1

Описание

Стойността на регистъра, сочен от Указателя към Стека SP, се копира в операнда dst и SP се увеличава с 1. Казано с други думи, върхът на стека се копира в операнда dst. Операндът dst използва разширена (12-битова) адресация.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
POPX	ER1	-	D8h	ER1[11:4]	{ ER1[3:0], 0h }	-

Адресният режим ER може да използва Escaped-адресация. Ако старшият байт на адреса е EEh, младшият полубайт указва работен регистър.

Пример:

POPX 345h

Ако SP сочи към регистър D70h и регистър D70h съдържа

стойност 44h, горната инструкция копира стойността 44h в регистър 345h и SP се увеличава с 1 (сочи към регистър D71h).

PUSH

Синтаксис

PUSH src

Операция

SP ← SP - 1

@SP ← src

Описание

Указателят към Стека SP се намалява с 1. Стойността на операнда src се копира в регистъра, сочен от SP. Казано с други думи, стойността на операнда src се записва в стека.

Флагове

C -

Z -

S -

V -

D -

H -

абревиатура	dst	src	опкод	o1	o2	o3
PUSH	-	R2	70h	R2	-	-
PUSH	-	@R2	71h	R2	-	-
PUSH	-	IM	1F70h	IM	-	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

PUSH FCh

Ако SP сочи към регистър D20h, горната инструкция записва стойността FCh в регистър D1Fh (преди операцията SP се намалява с 1).

PUSHX

Синтаксис

PUSHX src

Операция

SP ← SP - 1
@SP ← src

Описание

Указателят към Стека SP се намалява с 1. Стойността на операнда src се копира в регистъра, сочен от SP. Казано с други думи, стойността на операнда src се записва в стека. Операндът src използва разширена (12-битова) адресация.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
PUSHX	-	ER2	C8h	ER2[11:4]	{ ER2[3:0], 0h}	-

Адресният режим ER може да използва Escaped-адресация. Ако старшият байт на адреса е EEh, младшият полубайт указва работен регистър.

Пример:

PUSHX FCAh

Ако SP сочи към регистър D24h и регистър FCAh съдържа

стойност 55h, горната инструкция записва стойността 55h в регистър D23h (преди операцията SP се намалява с 1).

RCF

Синтаксис

RCF

Операция

C ← 0

Описание

Нулиране на флаг C.

Флагове

C 0
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
RCF	-	-	CFh	-	-	-

Пример:

RCF

Ако флаг C е 1, горната инструкция нулира флаг C.

RET

Синтаксис

RET

Операция

PC ← @SP

SP ← SP + 2

Описание

Тази инструкция се използва за излизане от подпрограма. Изпълнението й води до възстановяване на Програмния Брояч PC от стека. Указателят към Стека SP се намалява с 2. За повече информация вижте също 21.3 Стек.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
RET	-	-	AFh	-	-	-

```
;Подпрограма
MySubroutine:
        инструкция1
        ...
        инструкцияN
        RET   ;връщане от подпрограмата
```

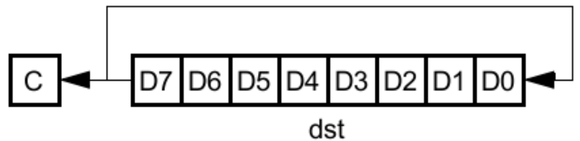
RL

Синтаксис

RL dst

Операция

Ротация наляво.



Описание

Съдържанието на операнда dst се премества с една позиция наляво. Стойността на бит 7 се премества в бит 0 и флаг C.

Флагове

C	dst[7]
Z	1 ако резултатът е 0, иначе е 0
S	1 ако бит 7 на резултата е 1, иначе е 0
V	1 ако има аритметично препълване, иначе е 0
D	-
H	-

абревиатура	dst	src	опкод	o1	o2	o3
RL	R1	-	90h	R1	-	-
RL	@R1	-	91h	R1	-	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

RL C6h

Ако регистър C6h съдържа стойност 88h (10001000b), горната инструкция оставя стойност 11h (00010001b) в регистър C6h, установява флаговете C и V и нулира флаговете S и Z.

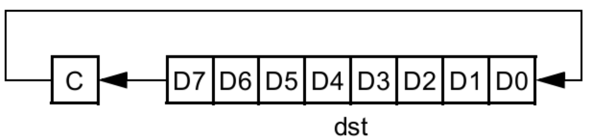
RLC

Синтаксис

RL dst

Операция

Ротация наляво през флаг C.



Описание

Съдържанието на операнда dst се премества с една позиция наляво. Стойността на бит 7 се премества във флаг C, а стойността на флаг C се премества в бит 0 на dst.

Флагове

C dst[7]
Z 1 ако резултатът е 0, иначе е 0
S 1 ако бит 7 на резултата е 1, иначе е 0
V 1 ако има аритметично препълване, иначе е 0
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
RLC	R1	-	10h	R1	-	-
RLC	@R1	-	11h	R1	-	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

RLC C6h

Ако регистър C6h съдържа стойност 8Fh (10001111b) и флаг C е 0, горната инструкция оставя стойност 1Eh (00011110b) в регистър C6h, установява флаговете C и V и нулира флаговете S и Z.

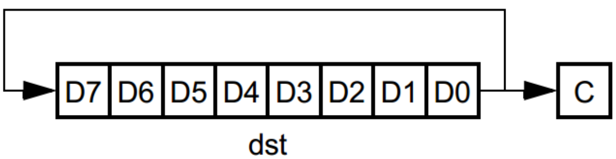
RR

Синтаксис

RLR dst

Операция

Ротация надясно.



Описание

Съдържанието на операнда dst се премества с една позиция надясно. Стойността на бит 0 се премества в бит 7 и флаг C.

Флагове

C dst[0]
Z 1 ако резултатът е 0, иначе е 0
S 1 ако бит 7 на резултата е 1, иначе е 0
V 1 ако има аритметично препълване, иначе е 0
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
RR	R1	-	E0h	R1	-	-
RR	@R1	-	E1h	R1	-	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

```
RR    R6
```

Ако работният регистър R6 (използва се Escaped-адресация) съдържа стойност 31h (00110001b), горната инструкция оставя стойност 98h (10011000b) в R6, установява флаговете C, Vи S, и нулира флага Z.

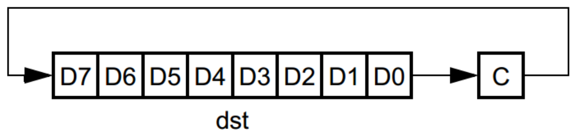
RRC

Синтаксис

RRC dst

Операция

Ротация надясно през флаг C.



Описание

Съдържанието на операнда dst се премества с една позиция надясно. Стойността на бит 0 се премества във флаг C, а стойността на флаг C се премества в бит 7 на dst.

Флагове

C	dst[0]
Z	1 ако резултатът е 0, иначе е 0
S	1 ако бит 7 на резултата е 1, иначе е 0
V	1 ако има аритметично препълване, иначе е 0
D	-
H	-

абревиатура	dst	src	опкод	o1	o2	o3
RRC	R1	-	C0h	R1	-	-
RRC	@R1	-	C1h	R1	-	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

RRC C6h

Ако регистър C6h съдържа стойност DDh (11011101b) и флаг C е 0, горната инструкция оставя стойност 6Eh (01101110b) в регистър C6h, установява флаговете C и V и нулира флаговете S и Z.

SBC

Синтаксис

SBC dst, src

Операция

dst ← dst - src - C

Описание

Изваждане със заем. Стойността на операнда src и флага C се изважда от стойността на операнда dst, и резултатът се съхранява в операнда dst.

Флагове

C	1 ако има заявка за заем от бит 7, иначе е 0
Z	1 ако резултатът е 0, иначе е 0
S	1 ако бит 7 на резултата е 1, иначе е 0
V	1 ако има аритметично препълване, иначе е 0
D	1
H	1 ако има заявка за заем от бит 3, иначе е 0

абревиатура	dst	src	опкод	o1	o2	o3
SBC	r1	r2	32h	{r1,r2}	-	-
SBC	r1	@r2	33h	{r1,r2}	-	-
SBC	R1	R2	34h	R2	R1	-
SBC	R1	@R2	35h	R2	R1	-
SBC	R1	IM	36h	R1	IM	-
SBC	@R1	IM	37h	R1	IM	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

SBC R3, R11

Ако регистър R3 съдържа стойност 16h (00010110b), флаг C е 1 и работният регистър R11 съдържа стойност 20h, горната инструкция оставя стойност F5h (11110101b) в регистър R3, установява флаговете C, S и D, и нулира флаговете Z, V и H.

Пример:

SBC R15, @R10

Ако работният регистър R15 съдържа стойност 16h, флаг C е 0, работният регистър R10 съдържа стойност 20h и регистър 20h съдържа стойност 11h, горната инструкция оставя 05h в работния регистър R15, установява флаг D, и нулира флаговете C, Z, S, V и H.

Пример:

SBC 34h, 12h

Ако регистър 34h съдържа стойност 2Eh, флаг C е 1 и регистър 12h съдържа стойност 1Bh, горната инструкция оставя стойност 12h в регистър 34h, установява флаг D, и нулира флаговете C, Z, S, V и H.

Пример:

SBC 4Bh, @R3

Ако регистър 4Bh съдържа стойност 82h, флаг C е 1, работният регистър R3 съдържа стойност 10h и регистър 10h съдържа стойност 01h, горната инструкция оставя стойност 80h в регистър 4Bh, установява флаговете D и S, и нулира флаговете C, Z, V и H.

Пример:

SBC 6Ch, #03h

Ако регистър 6Ch съдържа стойност 2Ah и флаг C е 0, горната инструкция оставя стойност 27h в регистър 6Ch, установява флаг D, и нулира флаговете C, Z, S, V и H.

Пример:

SBC @D4h, #02h

Ако регистър D4h съдържа стойност 5Fh, регистър 5F съдържа стойност 4Ch и флаг C е 1, горната инструкция оставя стойност 49h в регистър 5Fh, установява флаг D, и нулира флаговете C, Z, S, V и H.

SBCX

Синтаксис

SBCX dst, src

Операция

dst ← dst - src - C

Описание

Изваждане със заем. Стойността на операнда src и флага C се изважда от стойността на операнда dst, и резултатът се съхранява в операнда dst. Операндите dst и src използват разширена (12 битова) адресация за достъп до всеки регистър в Регистровата Памет.

Флагове

C 1 ако има заявка за заем от бит 7, иначе е 0
Z 1 ако резултатът е 0, иначе е 0
S 1 ако бит 7 на резултата е 1, иначе е 0
V 1 ако има аритметично препълване, иначе е 0
D 1
H 1 ако има заявка за заем от бит 3, иначе е 0

абревиатура	dst	src	опкод	o1	o2	o3
SBCX	ER1	ER2	38h	ER2[11:8]	{ ER2[3:0], ER1[11:8]}	ER1[7:0]
SBCX	ER1	IM	39h	IM	{ 0h, ER1[11:8]}	ER1[7:0]

Адресният режим ER може да използва Escaped-адресация. Ако старшият байт на адреса е EEh, младшият полубайт указва работен регистър.

Пример:

SBCX 346h, 129h

Ако регистър 346h съдържа стойност 2Eh, флаг C е 1 и регистър 129h съдържа стойност 1Bh, горната инструкция оставя стойност 12h в регистър 346h, установява флаг D и нулира флаговете C, Z, S, и H.

Пример:

SBCX C6Ch, #03h

Ако регистър C6Ch съдържа стойност 2Ah и флаг C е 0, горната инструкция оставя стойност 27h в регистър C6Ch, установява флаг D и нулира флаговете C, Z, S, и H.

SCF

Синтаксис

SCF

Операция

C ← 1

Описание

Установяване на флаг C.

Флагове

C 1
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
SCF	-	-	DFh	-	-	-

Пример:

SCF

Ако флагът C е 0, горната инструкция установява флага C.

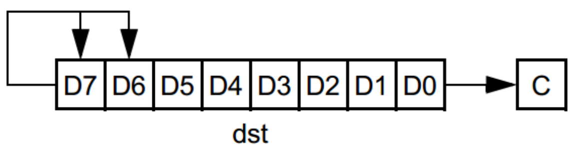
SRA

Синтаксис

SRA dst

Операция

Знаково преместване надясно.



Описание

Съдържанието на операнда dst се премества с една позиция надясно. Стойността на бит 0 се премества във флаг C, а стойността на бит 7 (знаков бит) се запазва и се копира в бит 6.

Флагове

C dst[0]
Z 1 ако резултатът е 0, иначе е 0
S 1 ако бит 7 на резултата е 1, иначе е 0
V 0
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
SRA	R1	-	D0h	R1	-	-
RL	@R1	-	D1h	R1	-	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

SRA R6

Ако работният регистър R6 (използва се Escaped-адресация) има стойност 31h (00110001b), горната инструкция оставя стойност B8h (10111000b) в R6, установява флаг C и нулира флаговете Z, V и S.

Пример:

SRA @C6h

Ако регистър C6h има стойност DFh и регистър DFh има стойност B8h (10111000b), горната инструкция оставя стойност DCh (11011100b) в регистър DFh, установява флаг S и нулира флаговете C, Z и V.

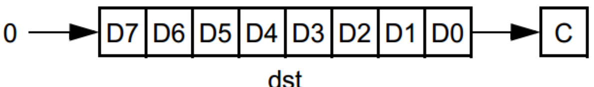
SRL

Синтаксис

SRL dst

Операция

Логическо преместване надясно.



Описание

Съдържанието на операнда dst се премества с една позиция надясно. Стойността на бит 0 се премества във флаг C, а стойността на бит 7 се попълва с 0.

Флагове

C dst[0]
Z 1 ако резултатът е 0, иначе е 0
S 0
V 1 ако има аритметично препълване, иначе е 0
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
SRL	R1	-	1FC0h	R1	-	-
SRL	@R1	-	1FC1h	R1	-	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

SRL R6

Ако работният регистър R6 (използва се Escaped-адресация) има стойност B1h (10110001b), горната инструкция оставя стойност 58h (01011000b) в R6, установява флаг C и нулира флаговете Z, V и S.

Пример:

SRL @C6h

Ако регистър C6h има стойност DFh и регистър DFh има стойност F8h (11111000b), горната инструкция оставя стойност 7Ch (01111100b) в регистър DFh и нулира флаговете C, Z, S и V.

SRP

Синтаксис

SRP src

Операция

RP ← src

Описание

Стойността на операнда src се записва в Указателя към Регистрите RP. Старшият полубайт RP[7:4] указва текущата регистрова група, а младшият полубайт RP[3:0] указва текущата регистрова страница.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
SRP	-	IM	01h	IM	-	-

Пример:

```
SRP   #01h ;RP = 01h
```

STOP

Синтаксис

STOP

Операция

Влизане в STOP-режим

Описание

Тази инструкция поставя процесора eZ8 в STOP-режим (вижте 25.8 Режими с ниска консумация).

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
STOP	-	-	6Fh	-	-	-

Пример:

```
STOP        ; процесорът eZ8 влиза в STOP-режим
```

SUB

Синтаксис

SUB dst, src

Операция

dst ← dst - src

Описание

Изваждане. Стойността на операнда src се изважда от стойността на операнда dst и резултатът се съхранява в операнда dst. Използва се допълнителен код.

Флагове

C 1 ако има заявка за заем от бит 7, иначе е 0
Z 1 ако резултатът е 0, иначе е 0
S 1 ако резултатът е отрицателен, иначе е 0
V 1 ако има аритметично препълване, иначе е 0
D 1
H 1 ако има заявка за заем от бит 3, иначе е 0

абревиатура	dst	src	опкод	o1	o2	o3
SUB	r1	r2	22h	{r1,r2}	-	-
SUB	r1	@r2	23h	{r1,r2}	-	-
SUB	R1	R2	24h	R2	R1	-
SUB	R1	@R2	25h	R2	R1	-
SUB	R1	IM	26h	R1	IM	-
SUB	@R1	IM	27h	R1	IM	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

SUB R3, R11

Ако работният регистър R3 съдържа стойност 16h и работният регистър R11 съдържа стойност 20h, горната инструкция записва стойност F6h в R3, установява флаговете C, S и D, и нулира флаговете Z,V, D и H.

Пример:

SUB R15, @R10

Ако работният регистър R15 съдържа стойност 16h и работният регистър R10 съдържа стойност 20h, и регистър 20h съдържа стойност 11h, горната инструкция записва стойност 05h в R15, установява флаг D и нулира флаговете C, Z, S, V и H.

Пример:

SUB 34h, 12h

Ако регистър 34h съдържа стойност 2Eh и регистър 12h съдържа стойност 1Bh, горната инструкция записва стойност 13h в регистър 34h, установява флаг D и нулира флаговете C, Z, S, V и H.

Пример:

SUB E4h, 12h

Използването на адрес E4h активира Escaped-адресация, т.е. . младшият полубайт на адреса указва работен регистър. В нашия случай това е R4. Ако работният регистър R4 съдържа стойност 2Eh и регистър 12h съдържа стойност 1Bh, горната инструкция записва стойност 13h в R4, установява флаг D и нулира флаговете C, Z, S, V и H.

Пример:

SUB 4Bh, @R3

Ако регистър 4Bh съдържа стойност 82h, и работният регистър R3 (използва се Escaped-адресация) съдържа стойност 10h и регистър 10h съдържа стойност 01h, горната инструкция записва стойност 81h в регистър 4Bh, установява флаговете D и S и нулира флаговете C, Z, V и H.

Пример:

SUB 6Ch, #03h

Ако регистър 6Ch съдържа стойност 2Ah, горната инструкция записва стойност 27h в регистър 6Ch, установява флаг D и нулира флаговете C, Z, S, V и H.

Пример:

SUB @D4h, #02h

Ако регистър D4h съдържа стойност 5Fh и регистър 5Fh съдържа стойност 4Ah, горната инструкция записва стойност 4Eh в регистър 5Fh, установява флаг D и нулира флаговете C, Z, S, V и H.

SUBX

Синтаксис

SUBX dst, src

Операция

dst ← dst - src

Описание

Изваждане. Стойността на операнда се изважда от стойността на операнда dst, и резултатът се съхранява в операнда dst. Използва се допълнителен код. Операндите dst и src използват разширена (12-битова) адресация за достъп до всеки регистър в Регистровата Памет.

Флагове

C 1 ако има заявка за заем от бит 7, иначе е 0
Z 1 ако резултатът е 0, иначе е 0
S 1 ако резултатът е отрицателен, иначе е 0
V 1 ако има аритметично препълване, иначе е 0
D 1
H 1 ако има заявка за заем от бит 3, иначе е 0

абревиатура	dst	src	опкод	o1	o2	o3
SUBX	ER1	ER2	28h	ER2[11:4]	{ER2[3:0], ER1[11:8]}	ER1[7:0]
SUBX	ER1	IM	29h	IM	{0h, ER1[11:8]}	ER1[7:0]

Адресният режим ER може да използва Escaped-адресация. Ако старшият байт на адреса е EEh, младшият полубайт указва работен регистър.

Пример:

SUBX 234h, 912h

Ако регистър 234h съдържа стойност 2Eh и регистър 912h съдържа стойност 1Bh, горната инструкция записва стойност 13h в регистър 234h, установява флаг D и нулира флаговете C, Z, S, V и H.

SWAP

Синтаксис

SWAP dst

Операция

dst[7:4] ↔ dst[3:0]

Описание

Тази инструкция разменя местата на старшия и младшия полубайт на операнда dst.

Флагове

C x
Z 1 ако резултатът е 0, иначе е 0
S 1 ако бит 7 на резултата е 0, иначе е 0
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
SWAP	R1	-	F0h	R1	-	-
SWAP	@R1	-	F1h	R1	-	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

SWAP BCh

Ако регистър BCh съдържа стойност B3h (10110011b), горната инструкция оставя стойност 3Bh (00111011b) в регистър BCh.

TCM

Синтаксис

TCM dst, src

Операция

(NOT dst) AND src

Описание

Тази инструкция тества избрани битове в операнда dst за логическа 1. Операндът src служи за маска. Специфицирайте съответните битове в маската да са 1. Операндът dst първо се инвертира (нулите се заменят с единици и обратно) и след това се AND-ва с маската src. Ако флаг Z е 1, тестваните битове са 1. Операндите dst и src не се променят.

Флагове

C -
Z 1 ако резултатът е 0, иначе е 0
S 1 ако бит 7 на резултата е 1, иначе е 0
V 0
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
TCM	r1	r2	62h	{r1,r2}	-	-
TCM	r1	@r2	63h	{r1,r2}	-	-
TCM	R1	R2	64h	R2	R1	-
TCM	R1	@R2	65h	R2	R1	-
TCM	R1	IM	66h	R1	IM	-
TCM	@R1	IM	67h	R1	IM	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

TCM R3, R7

Ако работният регистър R3 съдържа стойност 45h (01000101b) и работният регистър R7 съдържа стойност 01h (00000001b) (бит 0 в R3 се тества дали е 1), горната инструкция установява флаг Z (което означава, че бит 0 в R3 е 1) и нулира флаговете V и S.

TCMX

Синтаксис

TCMX dst, src

Операция

(NOT dst) AND src

Описание

Тази инструкция тества избрани битове в операнда dst за логическа 1. Операндът src служи за маска. Специфицирайте съответните битове в маската да са 1. Операндът dst първо се инвертира (нулите се заменят с единици и обратно) и след това се AND-ва със маската src. Ако флаг Z е 1, тестваните битове са 1. Операндите dst и src не се променят. Операндите dst и src използват разширена (12 битова) адресация за достъп до всеки регистър в Регистровата Памет.

Флагове

C -
Z 1 ако резултатът е 0, иначе е 0
S 1 ако бит 7 на резултата е 1, иначе е 0
V 0
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
TCMX	ER1	ER2	68h	ER2[11:4]	{ER2[3:0], ER1[11:8]}	ER1[7:0]
TCMX	ER1	IM	69h	IM	{0h, ER1[11:8]}	ER1[7:0]

Адресният режим ER може да използва Escaped-адресация. Ако старшият байт на адреса е EEh, младшият полубайт указва работен регистър.

Пример:

TCMX DD4h, 420h

Ако регистър DD4h съдържа стойност 04h (00000100b) и регистър 420h съдържа стойност 80h (10000000b) (бит 7 в регистър DD4h се тества дали е 1), горната инструкция нулира флаг Z (бит 7 в регистър DD4h не е 1) и установява флаг S и нулира флаг V.

TM

Синтаксис

TM dst, src

Операция

dst AND src

Описание

Тази инструкция тества избрани битове в операнда dst за логическа 0. Операндът src служи за маска. Специфицирайте съответните битове в маската да са 1. Операндът dst се AND-ва с маската src. Ако флаг Z е 1, тестваните битове са 0. Операндите dst и src не се променят.

Флагове

C -
Z 1 ако резултатът е 0, иначе е 0
S 1 ако бит 7 на резултата е 1, иначе е 0
V 0
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
TM	r1	r2	72h	{r1,r2}	-	-
TM	r1	@r2	73h	{r1,r2}	-	-
TM	R1	R2	74h	R2	R1	-
TM	R1	@R2	75h	R2	R1	-
TM	R1	IM	76h	R1	IM	-
TM	@R1	IM	77h	R1	IM	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

TM R3, R7

Ако работният регистър R3 съдържа стойност 45h (01000101b) и работният регистър R7 съдържа стойност 02h (00000010b) (бит 1 в R3 се тества дали е 0), горната инструкция установява флаг Z (което означава, че бит 1 в R3 е 0) и нулира флаговете V и S.

TMX

Синтаксис

TMX dst, src

Операция

dst AND src

Описание

Тази инструкция тества избрани битове в операнда dst за логическа 0. Операндът src служи за маска. Специфицирайте съответните битове в маската да са 1. Операндът dst се AND-ва със маската src. Ако флаг Z е 1, тестваните битове са 0. Операндите dst и src не се променят. Операндите dst и src използват разширена (12 битова) адресация за достъп до всеки регистър в Регистровата Памет.

Флагове

C	-
Z	1 ако резултатът е 0, иначе е 0
S	1 ако бит 7 на резултата е 1, иначе е 0
V	0
D	-
H	-

абревиатура	dst	src	опкод	o1	o2	o3
TMX	ER1	ER2	78h	ER2[11:4]	{ER2[3:0], ER1[11:8]}	ER1[7:0]
TMX	ER1	IM	79h	IM	{0h, ER1[11:8]}	ER1[7:0]

Адресният режим ER може да използва Escaped-адресация. Ако старшият байт на адреса е EEh, младшият полубайт указва работен регистър.

Пример:

TMX 789h, 246h

Ако регистър 789h съдържа стойност 45h (01000101b) и регистър 246h съдържа стойност 02h (00000010b) (бит 1 в регистър 789h се тества дали е 0), горната инструкция установява флаг Z (бит 1 в регистър DD4h е 0) и нулира флаговете V и S.

TRAP

Синтаксис

TRAP Vector

Операция

Генериране на софтуерно прекъсване.

SP ← SP - 2
@ SP ← PC
SP ← SP - 1
@ SP ← FLAGS
PC ← @Vector

Описание

Тази инструкция генерира софтуерно прекъсване. Програмният Брояч PC и регистър FLAGS се съхраняват в стека. Процесорът eZ8 зарежда 16-битовия Програмен Брояч със стойността, съхранена на адреса в Програмната Памет, указан с Vector. Физическият адрес в Програмната Памет се получава чрез умножаване на Vector*2. Vector има 256 възможни стойности: от 0 до 255. Базовият адрес започва от 0000h и завършва на 01FEh (510 десетично). Използвайте инструкцията IRET за връщане от прекъсването.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	Vector	опкод	o1	o2	o3

TRAP	Vector	F2h	Vector	-	-

Vector	Адрес в Програмната Памет (Vector * 2)
0[1]	0000h:0001h
1	0002h:0003h
2	0004h:0005h
3	0006h:0007h
4	0008h:0009h
5	000Ah:000Bh
6	000Ch:000Dh
7	000Eh:000Dh
…	…
255	01FEh:01FFh

Забележка[1]: Въпреки че в оригиналната документация нищо не е споменато, използването на този вектор е безсмислен, тъй като в Програмният Брояч ще се зареди стойността, съхранена на адреси 0000h:0001h в Програмната Памет. Z8Encore! микроконтролерите използват тези адреси за съхраняване на специални флаш-конфигурационни битове. В 25.15 Флаш-конфигурационни битове ще научите повече затова.

Пример:

TRAP #%34h

Ако адрес 68h (получава се като посоченият в инструкцията адрес се умножи по 2) съдържа стойност A0h и адрес 69h съдържа стойност 2Fh, горната инструкция съхранява регистрите FLAGS и PC в стека, PC се зарежда със стойността A02Fh и програмата се прехвърля за изпълнение на инструкцията, разположена на този адрес.

WDT

Синтаксис

WDT

Операция

Презареждане на WDT-таймера.

Описание

Първото изпълнение на тази инструкция стартира WDT-таймера. Всяко последващо изпълнение на инструкцията води до презареждане на WDT-таймера, предпазвайки го от нулиране.

Флагове

C -
Z -
S -
V -
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
WDT	-	-	52h	-	-	-

XOR

Синтаксис

XOR dst, src

Операция

dst ← dst XOR src

Описание

Логическо изключващо ИЛИ. Операндът src се XOR-ва с операнда dst и резултатът се съхранява в операнда dst.

Флагове

C -
Z 1 ако резултатът е 0, иначе е 0
S 1 ако бит 7 на резултата е 1, иначе е 0
V 0
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
XOR	r1	r2	B2h	{r1,r2}	-	-
XOR	r1	@r2	B3h	{r1,r2}	-	-
XOR	R1	R2	B4h	R2	R1	-
XOR	R1	@R2	45h	R2	R1	-
XOR	R1	IM	B6h	R1	IM	-
XOR	@R1	IM	B7h	R1	IM	-

Адресните режими R и IR могат да използват Escaped-адресация. Ако старшият полубайт на адреса е Eh, младшият полубайт указва работен регистър.

Пример:

XOR R1, R14

Ако работният регистър R1 съдържа стойност 38h (00111000b) и работният регистър R14 съдържа стойност 8Dh (10001101b), горната инструкция записва стойност B5h (10110101b) в R1, установява флаг S и нулира флаговете Z и V.

Пример:

XOR R4, @R13

Ако работният регистър R4 съдържа стойност F9h (11111001b), работният регистър R13 съдържа стойност 7Bh и регистър 7Bh съдържа стойност 6Ah (01101010b), горната инструкция записва стойност 93h (10010011b) в R4, установява флаг S и нулира флаговете Z, V.

Пример:

XOR 3Ah, 42h

Ако регистър 3Ah съдържа стойност F5h (11110101b) и регистър 42h съдържа стойност 0Ah (00001010b), горната инструкция записва стойност FFh в регистър 3Ah, установява флаг S и нулира флаговете Z и V.

Пример:

XOR E4h, 42h

Използването на адрес E4h активира Escaped-адресация, т.е. . младшият полубайт на адреса указва работен регистър. В нашия случай това е R4. Ако работният регистър R4 съдържа стойност F5h (11110101b) и регистър 42h съдържа стойност 0Ah (00001010b), горната инструкция записва стойност FFh в регистър R4, установява флаг S и нулира флаговете Z и V.

Пример:

XOR R5, @45h

Ако регистър R5 (използва се Escaped-адресация) съдържа стойност F0h (11110000b), регистър 45h съдържа стойност 3Ah и регистър 3Ah съдържа стойност 7Fh (01111111b), горната инструкция записва стойност 8Fh (10001111b) в R5, установява флаг S и нулира флаговете Z и V.

Пример:

XOR 7Ah, #F0h

Ако регистър 7Ah съдържа стойност F7h (11110111b), горната инструкция записва стойност 07h в регистър 7Ah и нулира флаговете Z, V и S.

Пример:

XOR @R3, #05h

Ако регистър R3 (използва се Escaped-адресация) съдържа стойност 3Eh и регистър 3Eh съдържа стойност 6Ch (01101100b), горната инструкция записва стойност 69h (01101001b) в регистър 3Eh и нулира флаговете Z,V и S.

XORX

Синтаксис

XORX dst, src

Операция

dst ← dst XOR src

Описание

Логическо изключващо ИЛИ. Операндът src се XOR-ва с операнда dst и резултатът се съхранява в операнда dst. Операндите dst и src използват разширена (12-битова) адресация за достъп до всеки регистър в Регистровата Памет.

Флагове

C -
Z 1 ако резултатът е 0, иначе е 0
S 1 ако бит 7 на резултата е 1, иначе е 0
V 0
D -
H -

абревиатура	dst	src	опкод	o1	o2	o3
XORX	ER1	ER2	B8h	ER2[11:4]	{ER2[3:0], ER1[11:8]}	ER1[7:0]
XORX	ER1	IM	B9h	IM	{0h, ER1[11:8]}	ER1[7:0]

Адресният режим ER може да използва Escaped-адресация. Ако старшият байт на адреса е EEh, младшият полубайт указва работен регистър.

Пример:

ORX 93Ah, 142h

Ако регистър 93Ah съдържа стойност F5h (11110101b) и регистър 142h съдържа стойност 0Ah (00001010b), горната инструкция записва стойност 9Fh (10011111b) в регистър 93Ah, установява флага S и нулира флаговете Z и V.

Пример:

XORX D7Ah, #01100110b

Ако регистър D7Ah съдържа стойност 07h (00000111b), горната инструкция записва стойност 61h (01100001b) в регистър D7Ah и нулира флаговете S, Z и V.

23 Описание на асемблера

23.1 Общи сведения

Асемблерът е специална програма, която преобразува асемблерния код в обектен код (машинни инструкции). Когато асемблирате една асемблерна програма на една компютърна система (в случая на персонален компютър), а генерираният код е предназначен за изпълнение на друга компютърна система (в случая на микроконтролера ZF083A), по-правилно е да се използва термина Макроасемблер вместо Асемблер. За по-кратко аз обаче ще използвам термина Асемблер.

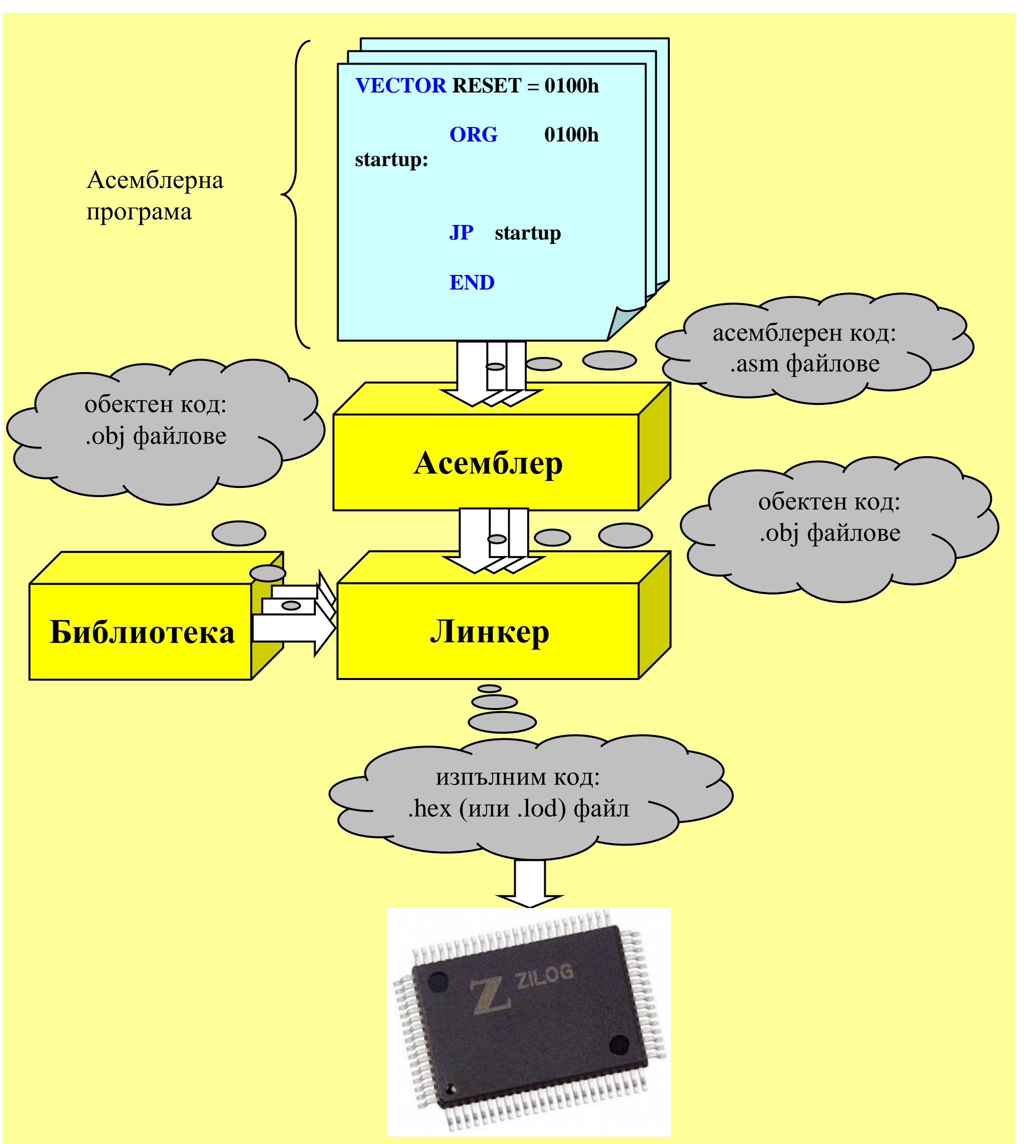
Асемблерната програма се състои от сорс-файлове с разширение **.asm**. Асемблерът преобразува асемблерните файлове в преместваеми обектни файлове с разширение **.obj**. Тези обектни файлове се подават на друга програма, наречена **Линкер**, която генерира крайния изпълним файл (файл с разширение **.hex** и/или **.lod**), готов за зареждане в програмната памет на микроконтролера. Фиг.198 илюстрира описания процес.

23.2 Адресни пространства и сегменти

ZDS II развойната среда разделя паметите на Z8 Encore! микроконтролерите на няколко адресни пространства (**address spaces**):

- ROM (Read Only Memory)

ROM-пространството се използва за съхранение на код и константни данни. Това пространство се разполага в Програмната Памет на адреси 0000h÷XXXXh, където XXXXh е последния адрес в Програмната Памет и е разли-



Фиг. 198 Асемблиране на асемблерна програма

чен за различните микроконтролери.

- RData (Register Data)

RData-пространството заема адреси 00h÷FFh в Регистровата Памет. То се използва за съхранение на неконстантни данни (променливи).

- EData (Extended Data)

EData-пространството заема адреси 100h÷EFFh в Регистровата Памет. То се използва за съхранение на неконстантни данни (променливи). Някои микроконтролери имат по-малко памет, ето защо горната граница на това адресно пространство ще бъде под EFFh. Микроконтролери, които имат 256 байта RAM памет или по-малко, нямат EData адресно пространство.

Забележка: Адресите F00h÷FFFh в Регистровата Памет са резервирани за управляващите регистри (SFRs).

Всяко адресно пространство се състои от сегменти. Сегментите са именувани логически части от данни или код. Сегментите с еднакви имена се съединяват по време на свързване. Всеки сегмент се асоциира само с едно адресно пространство. Има два типа сегменти:

- абсолютни сегменти

Абсолютният сегмент е сегмент с фиксирано начало. Началото на сегмента се задава с директивата ORG (директивите са разгледани в 23.4 Директиви). Данните и кодът в един абсолютен сегмент се разполагат на специфицирани физически адреси.

- преместваеми сегменти

Преместваемият сегмент е сегмент без фиксирано начало. Линкерът е отговорен за разполагането на кода и данните в съответните адресни пространства.

Следващата таблица описва сегментите, предефинирани от Асемблера.

сегмент	пространство	съдържание	тип по подразбиране
near_bss	RData	неинициализирани данни	Преместваем
far_bss	EData	неинициализирани данни	Преместваем
near_data	RData	инициализирани данни	Преместваем
far_data	EData	инициализирани данни	Преместваем
rom_data	ROM	инициализирани данни	Преместваем
near_txt	RData	низови константи	Преместваем
far_txt	EData	низови константи	Преместваем
rom_text	ROM	низови константи	Преместваем
code	ROM	код	Преместваем
text	RData	инициализирани данни	Преместваем
__vectors_*nnn*	ROM	Вектори на прекъсване	Абсолютен

Табл. 34 Предефинирани сегменти

Потребителят може да дефинира нов сегмент чрез използване на директивата DEFINE.

23.3 Изрази

23.3.1 Числово представяне

Числата се представят вътрешно като 32-битови знакови цели числа или 32-битови числа с плаваща запетая по стандарта IEEE 754 с единична точност.

Десетични числа

1234 ; Положително цяло десетични число
-123_456 ; Отрицателно цяло десетични число

12E-45 ; Положително десетично число с плаваща запетая
-123.456 ; Отрицателно десетично число с плаваща запетая
123.45E6 ;Положително десетично число с плаваща запетая

Шестнайсетични числа

ABCDEFFF**h** ; Положително шестнайсетично число
ABCDEFFF**H** ; Положително шестнайсетично число
%ABCDEFFF ; Положително шестнайсетично число
%ABCD_EFFF ; Положително шестнайсетично число

Двоични числа

0101b
1111B
0010_1010B

Символни константи (използват ASCII кодиране)

'A'
'3'

В повечето случаи, където операндът е цяло число или число с плаваща запетая, може да се използва и израз. Следващите подточки описват всички оператори, които могат да се използват в изразите.

23.3.2 Аритметични оператори

Оператор	Описание
<<	Преместване на ляво

>>	Аритметично преместване надясно
**	Повдигане на степен
*	Умножение
/	Деление
%	Деление по модул[1]
+	Събиране
-	Изваждане

Табл. 35 Аритметични оператори

Забележка[1]: Операторът % трябва да бъде разделен с интервали отпред и отзад от останалата част на израза.

23.3.3 Оператори за отношение

Оператор	Описание
==	Равно
!=	Различно
>	По-голямо
<	По-малко
>=	По-голямо или равно
<=	По-малко или равно

Табл. 36 Оператори за отношение

23.3.4 Битови оператори

Оператор	Описание
&	Битово И (AND)
|	Битово ИЛИ (OR)
^	Битово изключващо ИЛИ (XOR)

~	Битово инвертиране
!	Отрицание

Табл. 37 Битови оператори

23.3.5 Операторите HIGH и LOW

Тези оператори се използват за извличане на байтове от целочислени изрази. Операторът LOW извлича байта, започващ от бит 0 на израза, докато HIGH операторът извлича байта, започващ от бит 8 на израза.

Синтаксис:

LOW/HIGH израз

Пример:

```
LOW 1234h ; Връща като резултат 34h
HIGH 1234h ; Връща като резултат 12h
```

Тези оператори са удобни за инициализиране на указателя към стека SP.

Пример:

```
LDX SPL, #LOW(0100h)   ; 00h → SPL
LDX SPH, #HIGH(0100h)  ; 01h → SPH
```

23.3.6 Операторите HIGH16 и LOW16

Тези оператори се използват за извличане на 16-битови думи от целочислени изрази. Операторът LOW16 извлича думата,

започваща от бит 0 на израза, докато HIGH16 операторът извлича думата, започваща от бит 16 на израза.

Синтаксис:

LOW16/HIGH16 израз

Пример:

LOW16 12345678h ; Връща като резултат 5678h
HIGH16 12345678h ; Връща като резултат 1223h

23.3.7 Оператор .FTOL

Преобразува число с плаваща запетая в цяло число.

Пример:

LD r0, #.FTOL(12.34) ; 12 → r0

23.3.8 Оператор .LTOF

Преобразува целочислено число в число с плаваща запетая.

23.3.9 Приоритет на операторите

Оператор	Приоритет
( )	висок ↓
~ унарен - ! low high	
** * / %	

<table>
<tr><td>+</td><td>-</td><td>&</td><td>|</td><td>^</td><td>>></td><td><<</td><td rowspan="2">нисък</td></tr>
<tr><td><</td><td>></td><td><=</td><td>>=</td><td>==</td><td>!=</td><td></td></tr>
</table>

Табл. 38 Приоритет и асоциативност на операторите

Операторите, които се намират в една група имат еднакъв приоритет. Операторите с еднакъв приоритет се изчисляват от ляво надясно. Редът на изчисление на операторите може да се променя с помощта на скоби.

23.4 Директиви

Директивите управляват процеса на асемблиране, като осигуряват на асемблера допълнителна информация. Те улесняват писането на асемблерни програми. За абсолютно начинаещия директивите може да изглеждат като инструкции на процесора, но в действителност те са инструкции към асемблера и не се преобразуват в машинен код.

23.4.1 Директиви за данни

BLKB

Заделяне на блок-памет за 8 битови данни.

Синтаксис:

BLKB брой-данни [, <начална стойност>]

Пример:

```
BLKB        16 ; Заделя памет за 16 неинициализирани байта
```

```
BLKB        16, -1 ; Заделя памет за 16 байта и ги
                   ; инициализира в -1
```

BLKW

Заделяне на блок-памет за 16 битови данни.

Синтаксис:

BLKW брой-данни [, <начална стойност>]

Пример:

```
BLKW        16 ; Заделя памет за 16 неинициализирани 16-
               ; битови данни
BLKW        16, -1 ; Заделя памет за 16 16-битови данни и
                   ; ги инициализира в -1
```

BLKL

Заделяне на блок-памет за 32 битови данни.

Синтаксис:

BLKL брой-данни [, <начална стойност>]

Пример:

```
BLKL        16 ; Заделя памет за 16 неинициализирани 32-
               ; битови данни
BLKL        16, -1 ; Заделя памет за 16 32-битови данни и
                   ; ги инициализира в -1
```

DB

Заделяне и инициализиране на памет за 8-битови данни.

Синтаксис:

DB "Последователност-символи"

DB стойност [, стойност, ..., стойност]

Пример:

```
DB    "Hello World" ; Заделя и инициализира 11 байта
DB    1,2 ; Заделя 2 байта. Инициализира първия с 1, а
          ; втория с 2.
DB    %12 ; Заделя 1 байт и го инициализира с 0x12
```

Тази директива е особено подходяща за конфигуриране на флаш-конфигурационните байтове.

Пример:

```
DEFINE ___flash_option_segment, SPACE=ROM, ORG=0
SEGMENT ___flash_option_segment

flash_option_byte1: DB %FF
flash_option_byte2: DB %FF
```

Директивите DEFINE и SEGMENT са описани по-надолу. Най-общо казано в нашия пример се дефинира сегмент **___flash_option_segment**, който се разполага в ROM-адресното пространство, започвайки от адрес 0000h (точно където са разположени флаш-конфигурационните байтове). Директивата SEGMENT прави този сегмент активен, след което с директивата DB се заделят и инициализират първите два байта в сегмента.

DW

Заделяне и инициализиране на памет за 16-битови данни.

Синтаксис:

DW стойност [, стойност, …, стойност]

Пример:

```
DW    1, 2 ; Заделя памет за две 16-битови цели числа.
           ; Инициализира първото с 1, а второто с 2.
```

DW24

Заделяне и инициализиране на памет за 24-битови данни.

Синтаксис:

DW24 стойност [, стойност, …, стойност]

Пример:

```
DW24        1, 2 ; Заделя памет за две 24-битови цели
                 ; числа. Инициализира първото с 1, а второто
                 ; с 2.
```

DL

Заделяне и инициализиране на памет за 32-битови данни.

Синтаксис:

DL стойност [, стойност, …, стойност]

Пример:

```
DL    1, 2 ; Заделя памет за две 32-битови цели числа.
           ; Инициализира първото с 1, а второто с 2.
```

DF

Заделяне и инициализиране на памет за 32-битови дробни числа с единична точност по стандарта IEEE-754.

Синтаксис:

DF   стойност [, стойност, …, стойност]

Пример:

```
DF    0.1, 0.2 ; Заделя памет за две 32-битови дробни
               ; числа. Инициализира първото с 0.1, а
               ; второто с 0.2
```

DD

Заделяне и инициализиране на памет за 64-битови дробни числа с двойна точност по стандарта IEEE-754.

Синтаксис:

DD   стойност [, стойност, …, стойност]

Пример:

```
DD    0.1, -16.42 ; Заделя памет за две 64-битови дробни
                  ; числа. Инициализира първото с 0.1, а
                  ; второто с -16.42
```

23.4.2 ALIGN

Принуждава обектът, разположен след директивата, да бъде разположен на адрес кратен на **<стойност>**.

Синтаксис:

ALIGN <стойност>

Пример:

```
ALIGN    2
DW   1234h ; Заделяне на памет за 16-битови данни на адрес
           ; кратен на 2. Паметта се инициализира с 1234h.
```

23.4.3 SEGMENT

Указва начало на по-рано дефиниран сегмент (вижте директивта DEFINE).

Синтаксис:

SEGMENT име-сегмент

Пример:

```
SEGMENT code ; Начало на сегмент в ROM-адресното
             ; пространство за разполагане на
             ; инструкции
```

23.4.4 ORG

Задава адреса в текущия сегмент, на обекта, който се намира след тази директива.

Синтаксис:

ORG <адрес>

Пример:

```
SEGMENT near_data ; Избор на RData-адресно пространство
ORG  030h
DB   1,2,3 ; Заделяне и инициализиране на три байта в
           ; RData-адресното пространство, започвайки от
           ; адрес 030h

SEGMENT code ; Избор на ROM-адресно пространство
ORG  003Eh
LD   R0, #55h ; Тази инструкция ще се разположи на адрес
              ; 003Eh в Програмната Памет
```

23.4.5 DEFINE

Дефиниране на сегмент.

Синтаксис:

DEFINE име-сегмент [,<space>] [,<align>] [,<org>]

където

<space>

Дефинира адресното пространство, към което принадлежи сегмента и има следния синтаксис:

SPACE = адресно-пространство

Пример:

DEFINE fdata,SPACE = EData

<align>

Дефинира подравняването на сегмента. Подравняването, изразено в байтове, трябва да бъде степен на 2 (1, 2, 4, 8 и т.н.) и има следния синтаксис:

ALIGN = подравняване

Пример:

DEFINE fdata,SPACE = EData, ALIGN = 2

<org>

Дефинира адреса, на който да се разположи сегмента и има следния синтаксис:

ORG = адрес

Пример:

DEFINE near_code,ORG = %FFF8

23.4.6 DS

Заделяне на неинициализирана памет.

Синтаксис:

DS брой-байтове

Пример:

```
DS      10 ; Заделя памет за 10 байта
```

Тази директива е особено полезна за дефиниране на променливи в RAM паметта.

Пример:

```
SEGMENT near_bss ; RData-адресното пространство
var1:  DS     1 ;  Резервиране на 1 байт памет

SEGMENT code ; ROM-адресно пространсво
LD     var1, #55h  ; 55h → var1
```

23.4.7 END

Информира асемблера за край на сорс-файла.

Синтаксис:

```
END
```

Всичко след тази директива се игнорира от асемблера.

23.4.8 EQU

Назначаване на символни имена на числови или низови стойности.

Синтаксис:

```
<етикет>   EQU  <израз>
```

Пример:

```
lenght     EQU     6
```

```
width        EQU        11
area         EQU        length*width
```

Пример:

```
CONST_VAL   EQU    55h

LD    var1, # CONST_VAL  ; 55h → var1
```

23.4.9 VAR

Тази директива е подобна на EQU с тази разлика, че позволява стойността на етикета да бъде променяна.

Синтаксис:

<етикет> VAR <израз>

Пример:

```
lenght       VAR        6
width        EQU        lenght/2 ; 6/2
lenght       VAR        8
area         EQU        lenght*width ; 8 * 6/2
```

23.4.10 INCLUDE

Вмъква съдържанието на друг файл в текущия файл.

Синтаксис:

INCLUDE "име-файл.inc"

или

INCLUDE "път-към-файла\име-файл.inc"

Пример:

```
INCLUDE   "calc.inc"
INCLUDE   "\test\calc.inc"
```

23.4.11 VECTOR

Инициализира таблицата с векторите на прекъсване с векторите на прекъсване на обработващите подпрограми.

Синтаксис:

VECTOR <име-вектор> = <име-подпрограма>

където

<име-вектор>

RESET	-	ресет вектор
WDT	-	WDT-таймер
POTRAP	-	Основен генератор
WOTRAP	-	RC генератор на WDT-таймера
TRAP	-	Грешна инструкция
TIMER1	-	Таймер1
TIMER0	-	Таймер0
ADC	-	АЦП
P0AD	-	Извод PA0
P1AD	-	Извод PA1
P2AD	-	Извод PA2
P3AD	-	Извод PA3
P4AD	-	Извод PA4
P5AD	-	Извод PA5

P6AD	-	Извод PA6
P7AD	-	Извод PA7
C3	-	Извод PC3
C2	-	Извод PC2
C1	-	Извод PC1
C0	-	Извод PC0

<име-подпрограма>

Име на подпрограмата за обработка на прекъсването.

Пример:

```
VECTOR  RESET   = startup
VECTOR  WDT     = watchdog_irq_handler
VECTOR  TIMER0  = timer0_irq_handler
```

23.4.12 XDEF

Прави етикети, дефинирани някъде в асемблерния файл, видими за други файлове.

Синтаксис:

XDEF етикет1, етикет2, …, етикетN

Пример:

```
XDEF label1, label2, label3
```

23.4.13 XREF

Прави етикети, дефинирани в други асемблерни файлове, видими в текущия файл. Директивата също може да укаже и адресното пространство, в което се намира етикета.

Синтаксис:

XREF етикет1[:<простр>],…, етикетN [:<простр>]

Пример:

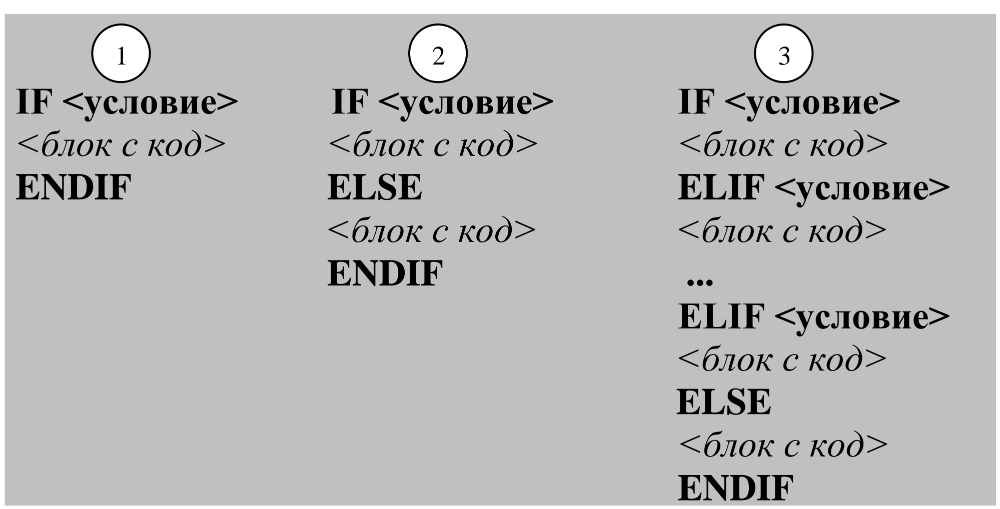
```
XREF    label
XREF    label1, label2, label3
XREF    label:ROM
```

23.4.14 Условни директиви

Условните директиви се използват за управление на асемблирането на различни части от кода. Цели блокове с код могат да бъдат разрешени или забранени за асемблиране, чрез използване на условните асемблерни директиви.

IF

Синтаксис:



1	2	3
IF <условие>	**IF <условие>**	**IF <условие>**
<блок с код>	*<блок с код>*	*<блок с код>*
ENDIF	**ELSE**	**ELIF <условие>**
	<блок с код>	*<блок с код>*
	ENDIF	**...**
		ELIF <условие>
		<блок с код>
		ELSE
		<блок с код>
		ENDIF

Тази директива работи по следния начин:

1. Асемблерът изчислява <условие>.

2. Ако резултатът е различен от 0, блокът с код, заключен между IF и ENDIF (за форма 1), IF и ELSE (за форма 2) и IF и ELIF (за форма 3), се асемблира, а всички останали блокове с код се игнорират.

3. Ако резултатът е 0, се извършват следните действия в звисимост от използваната форма на директивата:

    a. Форма 1: <блок с код> между IF и ENDIF се игнорира.

    b. Форма 2: <блок с код> между IF и ELSE се игнорира, а <блок с код> между ELSE и ENDIF се подава на асемблера.

    c. Форма 3: <блок с код> между IF и първият ELIF се игнорира и се преминава към изчисляване на <условие> на първия ELIF. Ако резултатът е различен от 0, <блок с код>, заключен между първия ELIF и следващия ELIF, се подава на асмеблера, а останалите <блок с код> се игнорират. Ако резултатът е 0, се преминава към изчисляване на <условие> на следващия ELIF и т.н. Ако всички условия се изчисляват до 0, <блок с код>, заключен между ELSE и ENDIF се подава на асемблера.

Пример:

```
IF      XYZ != 3
;…
;някакъв код, който се асемблира ако XYZ е различно от 3
;…
```

```
ELSE
;…
;някакъв код, който се асемблира ако XYZ е  3
;…
ENDIF
```

IFDEF

Синтаксис:

Тази директива работи по следния начин:

1. Асемблерът проверява дали <етикет> е дефиниран.

2. Ако <етикет> е дефиниран, <блок с код>, заключен между IFDEF и ENDIF (за форма 1), IFDEF и ELSE (за форма 2), се подава на асемблера, а <блок с код>, заключен между ELSE и ENDIF се игнорира.

3. Ако <етикет> не е дефиниран, <блок с код>, заключен между IFDEF и ENDIF (за форма 1), IFDEF и ELSE (за форма 2), се игнорира, а <блок с код>, заключен между ELSE и ENDIF се подава на асемблера.

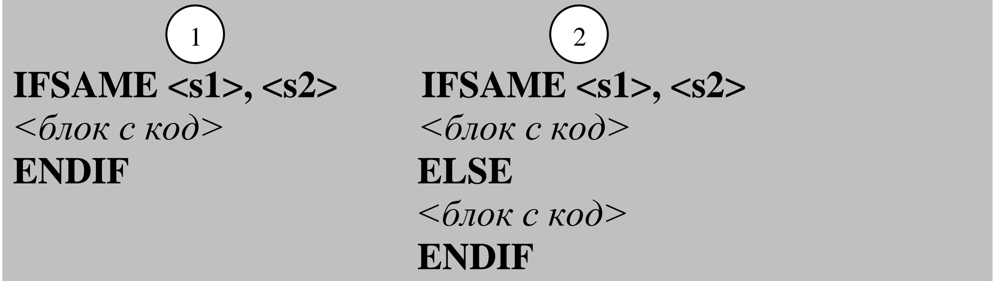
Пример:

```
IFDEF        XYZ
```

```
;…
;някакъв код, който се асемблира ако XYZ е дефиниран
;…
ELSE
;…
;някакъв код, който се асемблира ако XYZ не е дефиниран
;…
ENDIF
```

IFSAME

Синтаксис:



Тази директива работи по следния начин:

1. Асемблерът проверява дали стринговете <s1> и <s2> са еднакви.

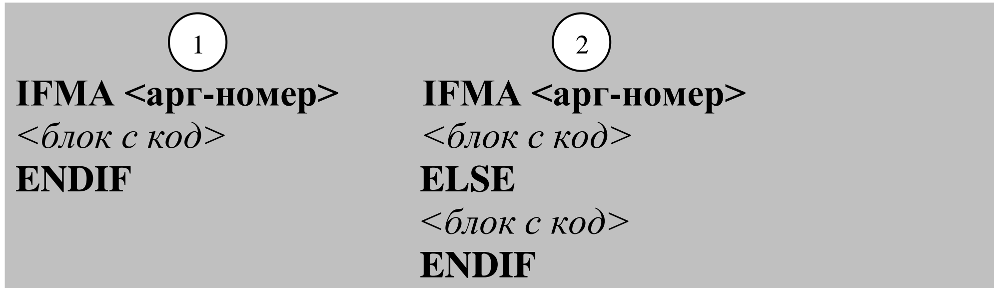
2. Ако <s1> и <s2> са еднакви, <блок с код>, заключен между IFSAME и ENDIF (за форма 1), IFSAME и ELSE (за форма 2), се подава на асемблера, а <блок с код>, заключен между ELSE и ENDIF се игнорира.

3. Ако <s1> и <s2> не са еднакви, <блок с код>, заключен между IFSAME и ENDIF (за форма 1), IFSAME и ELSE (за форма 2), се игнорира, а <блок с код>, заключен между ELSE и ENDIF се подава на асемблера.

IFMA

Тази директива може да се използва само в макроси (вижте 23.5 Макроси).

Синтаксис:



Тази директива работи по следния начин:

1. Асемблерът проверява дали аргументът, указан с <арг-номер> е дефиниран.

2. Ако <арг-номер> е дефиниран, <блок с код>, заключен между IFMA и ENDIF (за форма 1), IFMA и ELSE (за форма 2), се подава на асемблера, а <блок с код>, заключен между ELSE и ENDIF се игнорира.

3. Ако <арг-номер> не е дефиниран, <блок с код>, заключен между IFMA и ENDIF (за форма 1), IFMA и ELSE (за форма 2), се игнорира, а <блок с код>, заключен между ELSE и ENDIF се подава на асемблера.

Пример:

Вижте 23.5 Макроси

23.5 Макроси

23.5.1 Дефиниране и извикване на макрос

Макросът е програмно средство, което позволява последователност от асемблерни инструкции да бъде представена от един асемблерен символ. Макросите могат да имат и параметри. Макросът трябва да бъде дефиниран преди да се използва в кода.

Дефиниране на макрос:

```
<име-макрос>[:]   MACRO      [<пар1>,...,<парN>]
<тяло на макроса>
                  ENDMAC[RO] <име-макрос>
```

Пример:

```
mymacro:   MACRO
LD    R0, #55h
LD    R1, #44h
           ENDMACRO mymacro
```

Обърнете внимание, че двете точки след името на макроса могат да се пропуснат, т.е. . следния пример е напълно валиден:

Пример:

```
mymacro    MACRO
LD    R0, #55h
LD    R1, #44h
           ENDMACRO mymacro
```

Също така дефиницията на макрос може да завършва с директивата ENDMAC или ENDMACRO.

Пример:

```
mymacro    MACRO
LD     R0, #55h
LD     R1, #44h
           ENDMAC mymacro
```

Както се вижда от дефиницията, макросите могат да имат и параметри. Параметрите се разделят със запетая един от друг.

Пример:

```
mymacro:   MACRO     val1, val2
LD     R0, #val1
LD     R1, #vla2
           ENDMACRO mymacro
```

Макросът се извиква чрез своето име. Ако макросът има и параметри, при извикването след името на макроса се изброяват и аргументите, разделени със запетая един от друг.

Извикване на макрос:

<име-макрос> [<арг1> [,<арг2>]...[,<аргN>]]

Извикването на макроса в кода се заменя от асемблера с неговото тяло. Параметрите, изполвани в тялото на макроса, се заменят със съответните аргументи. Процесът на замяна на макрос с неговото тяло се нарича разгъване на макрос.

Пример:

```
mymacro    55h, 44h
```

Асемблерът ще замени горният ред с тялото на макроса:

```
LD    R0, #55h
LD    R1, #44h
```

Аргументите на макрос могат да бъдат разграничавани един от друг, чрез използване на разграничителни символи, дефинирани с директивата MACDELIM. По подразбиране за разграничители се използват фигурни скоби, но [] и () също са позволени.

Пример:

```
;Дефиниране на макроса
BRA: MACRO ARG1
JP ARG1
      ENDMACRO

;Извикване на макроса
LAB: BRA {NE, LAB} ; Използване на разграничителите по
                   ; подразбиране
```

Пример:

```
MACDELIM     [ ; Задаване на нови разграничители

;Дефиниране на макроса
BRA: MACRO ARG1
JP    ARG1
      ENDMACRO

;Извикване на макроса
LAB: BRA [NE, LAB] ; Използване на разграничителите []
```

23.5.2 Локални макро-етикети

Локалните макро-етикети позволяват използването на

етикети в многократно разгъване на макроси без дублиране. Символи, предшествани от $$, се разглеждат като локални етикети с област на видимост само в тялото на макроса. Асемблерът автоматично заменя символите $$ с долна черта, последвана от номера на извикване на макроса.

Пример:

```
;Дефиниране на макрос
LJMP: MACRO cc, label
JR      cc,$$lab
JP      label
$$lab: ENDMAC

;Извикване на макроса
LJMP        Z, NEXT
```

Това извикване се разгъва подобно на показания по-долу код:

```
JR      Z,_1NEXT
JP      NEXT
_1NEXT:
```

Повторно извикване на макроса ще се разгъне до:

```
JR      Z,_2NEXT
JP      NEXT
_2NEXT:
```

23.5.3 Макро-аргументи по избор

Макросите с параметри мога да пропускат подаването на аргументи при извикването си. Директивата IFMA (вижте

IFMA) може да се използва за определяне дали даден аргумент е подаден при извикване на макрос. Аргументите се номерират от едно нагоре.

Пример:

```
; Дефиниране на макрос
MISSING_ARG:  MACRO     ARG0,ARG1,ARG2
IFMA 2 ; Проверка дали аргумент 2 е подаден
LD ARG0,ARG1 ; аргумент 2 е подаден
ELSE
LD ARG0,ARG2 ; аргумент 2 не е подаден
ENDIF
              ENDMACRO MISSING_ARG

;Извикване на макроса с липсващ аргумент 2
MISSING_ARG R1, ,@R2
```

Извикването на макроса ще се разгъне до инструкцията, заключена между директивите ELSE и ENDIF:

```
LD    R1, @R2
```

23.5.4 Излизане от макрос

Може да използвате директивата MACEXIT за незабавно излизане от макрос. Типично тази директива се използва за излизане от рекурсивни макроси.

23.6 Етикети

Етикетите са символно представяне на адреси в паметта. Те трябва да спазват следните условия:

- Трябва да завършват с двуеточие;

- Трябва да започват от началото на реда. Предхождащи интервали не са позволени;

- Могат да бъдат дефинирани с директивата EQU по следния начин:

  <етикет>   EQU   <израз>

- Първият символ на етикет може да бъде буква или един от следните знаци: _, **$**, **?**, **.**, #. Следващите символи могат да включват букви, цифри или горните знаци. Един етикет може да бъде дефиниран само веднъж. Максималната дължина на етикета е 129 символа;

- Етикети, които се интерпретират като шестнадесетични числа, не са позволени.

23.6.1 Анонимни етикети

Анонимният етикет се състои от два знака за долар ($). Анонимните етикети могат да бъдат дефинирани произволен брой пъти. Референцията към анонимен етикет се осъществява с помощта на други два етикета: $B (backward reference) и $F (forward reference).

Етикетът $B се използва за референция към последния анонимен етикет, дефиниран преди референцията.

Пример:

```
$$:   NOP  <---+
      JP   $B--+
```

```
$$:     NOP
```

Етикетът $F се използва за референция към следващия анонимен етикет, дефиниран след референцията.

Пример:

```
$$:     NOP
        JP      $F ──┐
$$:     NOP  ◄───────┘
```

23.6.2 Локални етикети

Всеки етикет, започващ с $ или завършващ с ? се счита за локален етикет. Областта на видимост на локален етикет завършва когато се срещне директивата SCOPE, позволявайки преизползване на името на етикета. Локалните етикети не могат да бъдат експортирани.

Пример:

```
$LOOP:      JP $LOOP ; Безкраен преход към $LOOP
LAB?:       JP LAB? ; Безкраен преход към LAB?
SCOPE ; Нова област на видимост
$LOOP:      JP $LOOP ; Преизползване на  $LOOP
LAB?:       JP LAB? ; Преизползване на LAB?
```

23.6.3 Импортиране и експортиране на етикети

Етикетите могт да бъдат импортирани от други модули с директивата XREF. Адресното пространство, в което е разположен етикета, също може да се укаже с директивата, в противен случй се използва адресното пространство на

текущия сегмент, в който е рзположена директивата XREF.

Етикетите могат да бъдат експортирани към други модули с директивата XDEF.

23.6.4 Пространства на етикетите

Асемблерът използва адресното пространство на етикетите когато проверява валидността на операндите на инструкциите. Някои операнди изискват етикета да бъде разположен в определено адресно пространство, тъй като инструкцията може да работи с данни, разположени в това адресно пространство. Един етикет се назначава на адресно пространство чрез един от следните методи:

- Адресното пространство на сегмента, в който е дефиниран етикета;
- Адресното пространство, указано в директивата XREF;
- Ако адресното пространство не е указано в директивата XREF, се използва адресното пространство на сегмента, в който се намира директивата XREF.

23.6.5 Проверка на етикетите

Асемблерът извършва проверка на адресното пространство, в което се намира етикета, когато етикетът се използва като операнд на инструкция. Ето защо, ако етикетът не е разположен в подходящо адресно пространство, асемблерът ще генерира предупреждение.

```
XREF label1:ROM
JP  label1 ; Валиден операнд
LD r0, label1 ; Невалиден операнд
```

24 Развойна среда Zilog Developer Studio II

24.1 Общи сведения

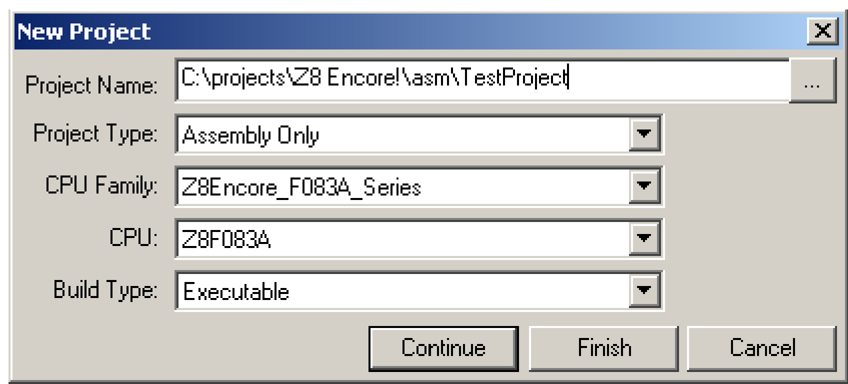
Zilog предоставя напълно безплатно развойната среда Zilog Developer Studio II (ZDS II) за програмиране на микороконтролерите Z8Encore!. Тази среда предоставя всичко необходимо (текстов редактор, асемблер и С компилатор) за написването, тестването и дебъгването на асемблерни и С програми.

В следващите подточки ще Ви покажа как да си създадете проект за писане на асемблерни програми и ще Ви запозная с основните менюта и прозорци, които ще използвате за тестване и дебъгване на тези програми.

24.2 Създаване на асемблерен проект

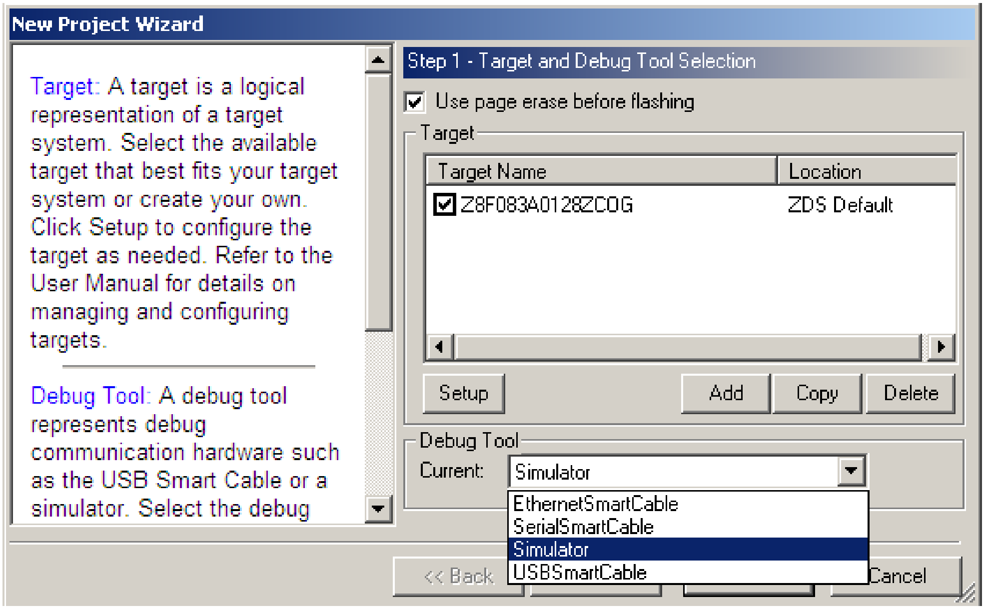
Създаването на проект в средата ZDS II е много лесно и отнема няколко минути.

За да създадете проект изберете **File → New Project**. Това води до отваряне на прозореца **New Project** (Фиг.199).



Фиг. 199 New Project

Задайте показаните настройки (в полето **Project Name** може да зададете път и име на проекта по свой избор) и натиснете Continue. Отваря се прозореца **New Project Wizard** (Фиг. 200).

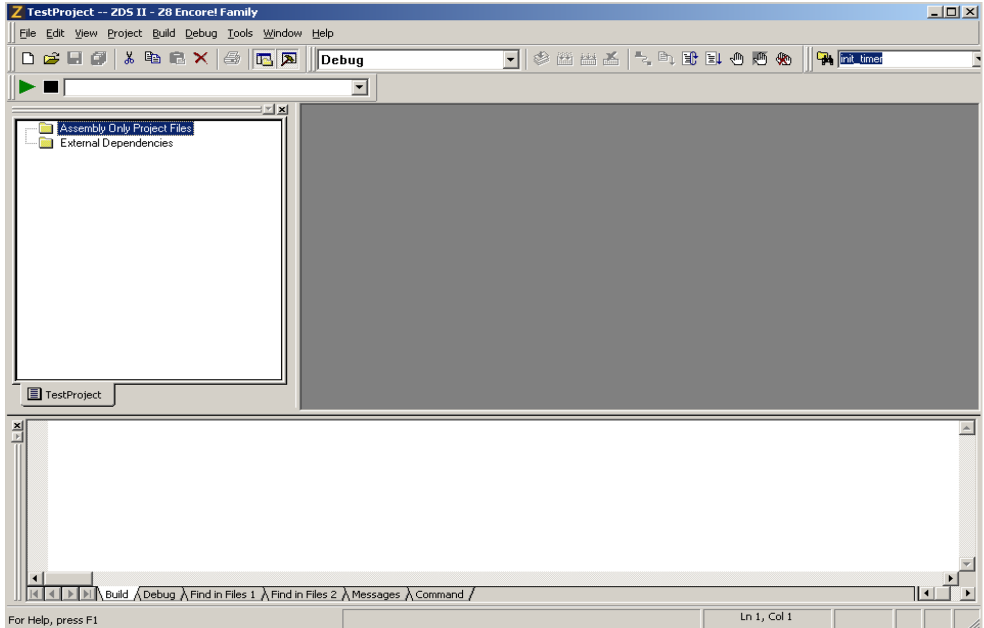


Фиг. 200 New Project Wizard

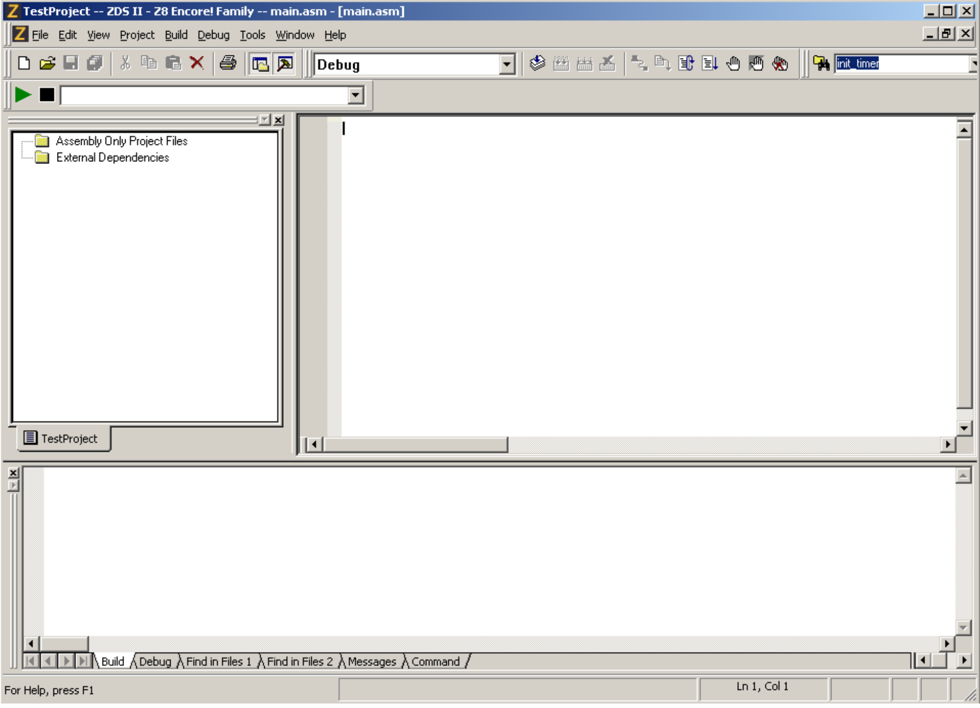
В полето **Debug Tool: Curent** изберете **Simulator** и натиснете Finish. Вашият проект е създаден (Фиг.201).

Това което остава да направите е да създадете сорс-файл (или сорс-файлове), в който да пишете Вашия асемблерен код. За целта изберете **File → New File** (Фиг.202). Съхранете файла (с разширение **.asm**) в директорията на проекта като изберете **File → Save As**.

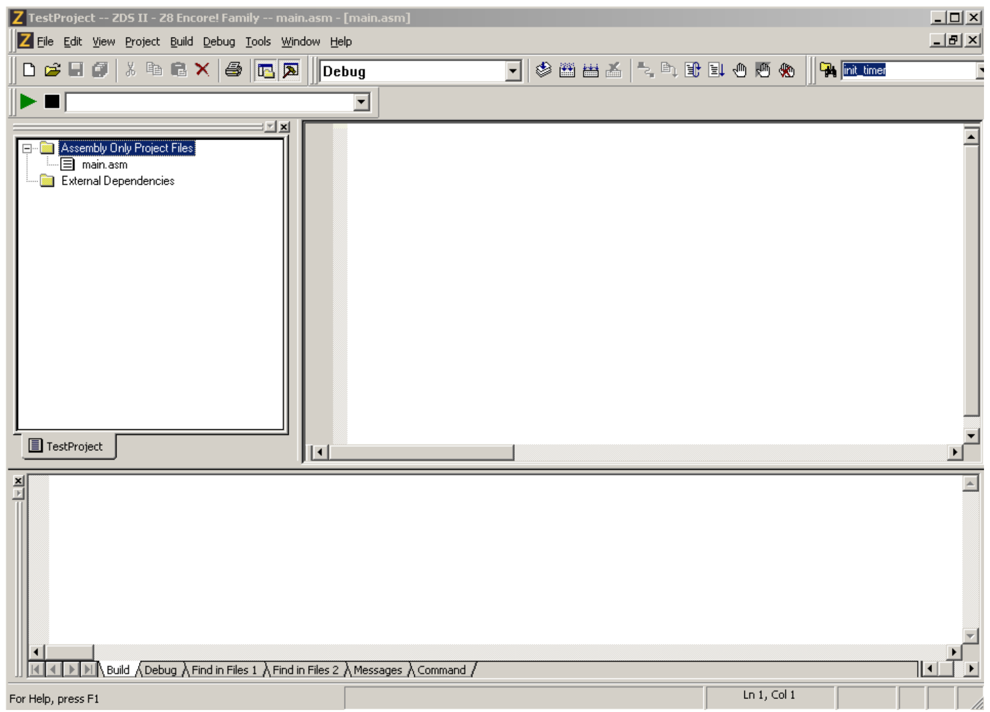
Вече имате създаден сорс-файл, който трябва да бъде добавен в проекта. За целта изберете **Project → Add Files…** . Намерете файла, избрете го и натиснете Add. Вашият файл е добавен в проекта (Фиг.203) и може да пишете код в него.



Фиг. 201 TestProject



Фиг. 202 Създаване на асемблерен файл



Фиг. 203 Добавяне на сорс-файл към проекта

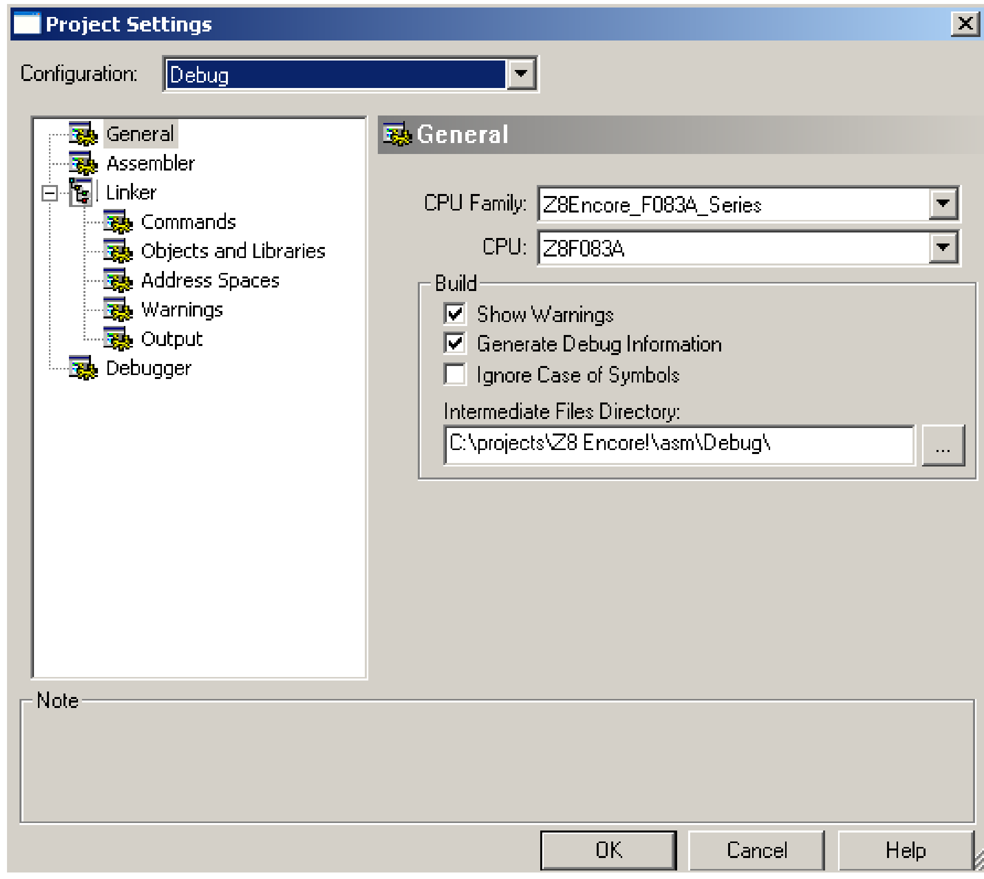
24.3 Отваряне на съществуващ проект

За да отворите вече съществуващ проект изберете **File** → **Open Project**, намерете проектния файл (файла с разширение **.zdsproj**), изберете го и натиснете Open.

24.4 Конфигуриране на проекта

За да конфигурирате проекта, изберете **Project** → **Settings…**. Отваря се прозореца **Project Settings** (Фиг.204). В лявата част на прозореца са изброени няколко категории настройки:

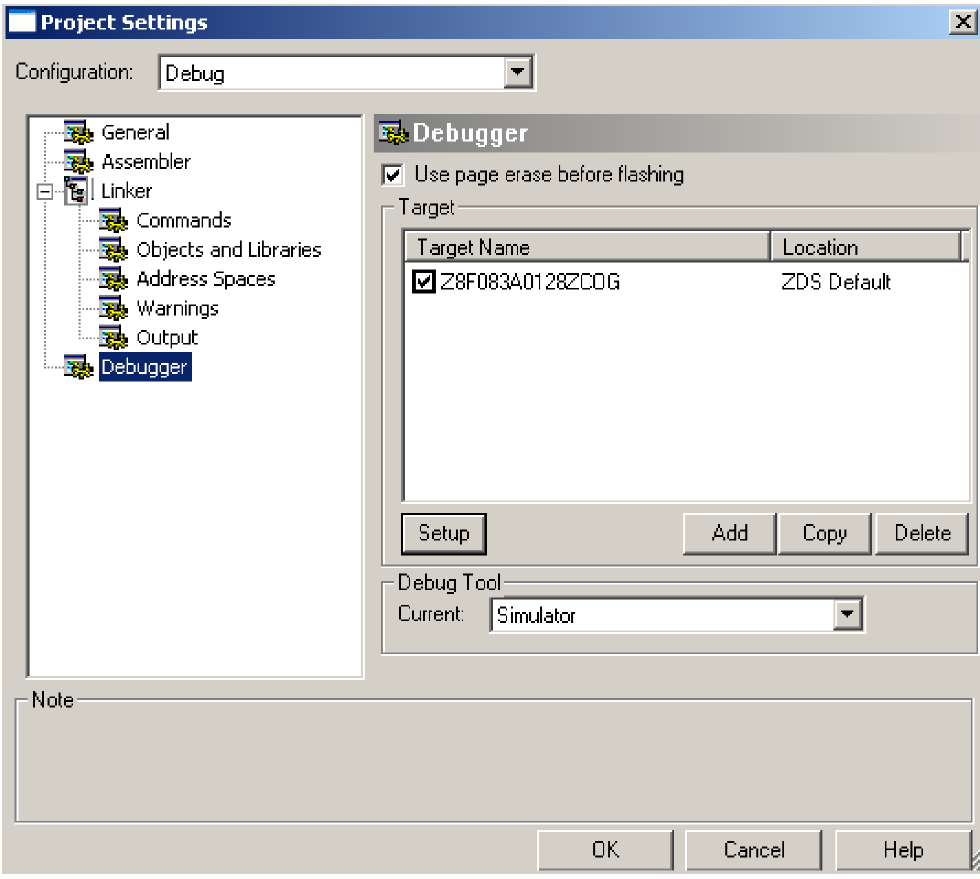
- General



Фиг. 204 Project Settings

В категорията General може да зададете нов микроконтролер.

▪ Debugger



Фиг. 205 Project Setting - Debugger

В категорията Debugger може да зададете нов инструмент за тестване и дебъгване (например USBSmartCable вместо симулатора).

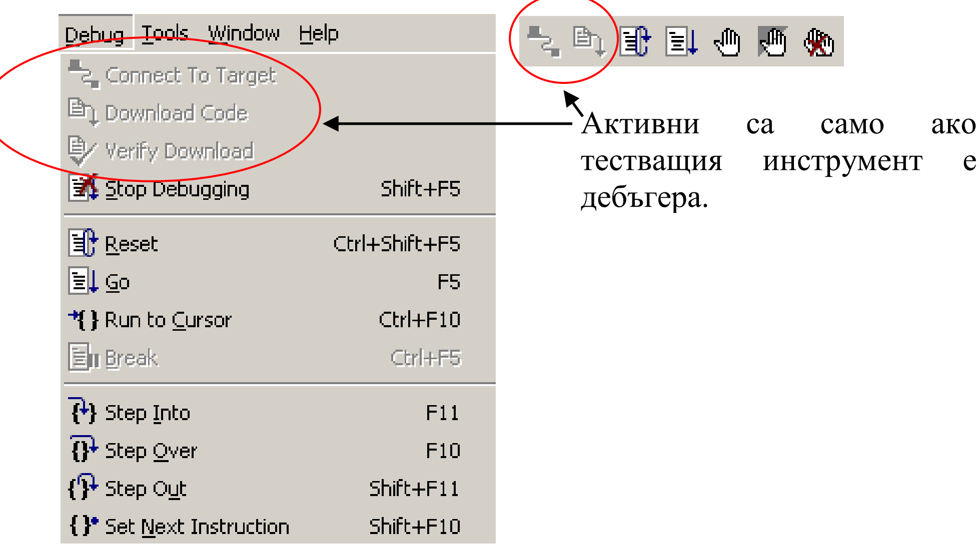
24.5 Описание на основните менюта и прозорци

24.5.1 Асемблиране на проекта

| | |_ Компилиране/Асемблиране на целия проект (всички файлове)
| |_ Компилиране/Асемблиране само на последно променените файлове
|_ Компилиране/Асемблиране само на текущия файл

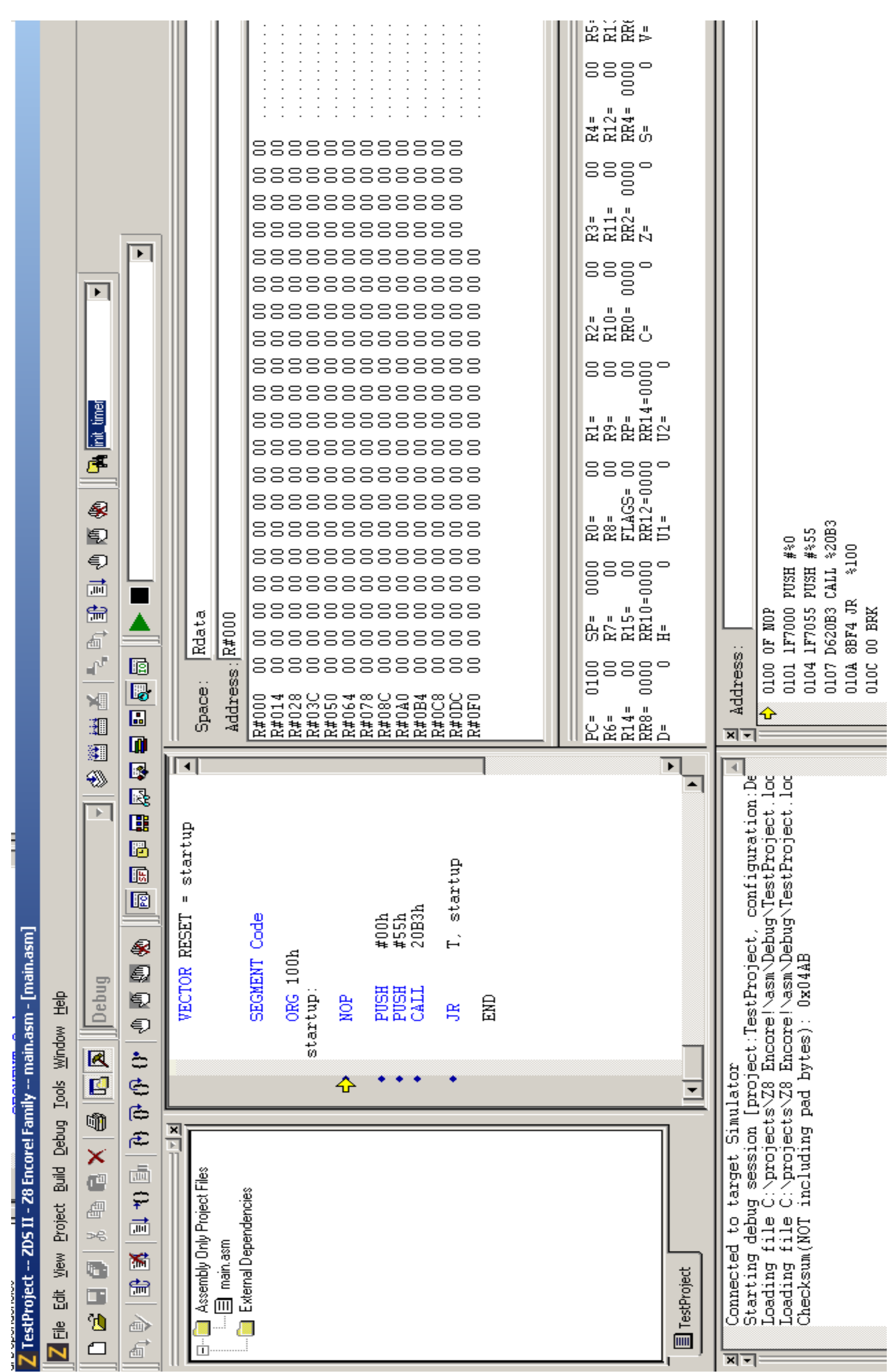
24.5.2 Тестване и дебъгване

По-долу са показани менюто и лентата с инструменти, които се използват за тестване/дебъгване на кода.



Фиг. 206 Меню DEBUG

В зависимост от избрания инструмент за тестване/дебъгване (симулатор или дебъгер) някои бутони може да се неактивни. Влизането в тестов режим става просто с натискане на ресет бутона (Фиг.207). Забележете, че вляво на прозореца с програмния код се появява жълта стрелка, която сочи към поредната за изпълнение



Фиг. 207 Влизане в режим на тестване/дебъгване

инструкция. Следва описание на различни бутони, които ще използвате, за да тествате/дебъгвате Вашия код.

```
|   |   |   |   |   |   |   |   |_ Задаване на следващата за изпълнение
|   |   |   |   |   |   |   |      инструкция
|   |   |   |   |   |   |   |_ Излизане от подпрограма
|   |   |   |   |   |   |_ Изпълняване на поредната инструкция
|   |   |   |   |   |      (без влизане в подпрограма)
|   |   |   |   |   |_ Изпълняване на поредната инструкция
|   |   |   |   |      (с влизане в подпрограма)
|   |   |   |   |_Спиране на изпълнението на програмата
|   |   |   |_Стартиране на програмата до мястото където е разположен курсора
|   |   |_Стартиране на програмата
|   |_Излизане от режим тестване/дебъгване
|_Ресет
```

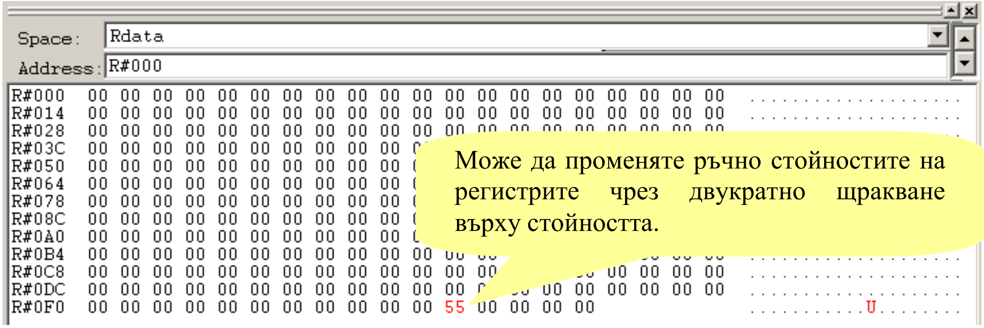
```
|   |   |   |_ Премахване на всички точки на прекъсване[1]
|   |   |_ Забраняване на всички точки на прекъсване
|   |_ Забраняване на текущата точка на прекъсване
|_ Поставяне на точка на прекъсване
```

Забележка[1]: Точката на прекъсване (**breakpoint**) е механизъм за умишлено спиране на програмата при достигане до определена инструкция. Може да поставяте точки на прекъсване само на инструкции.

Значението на тези бутони ще стане най-ясно, когато започнете да ги използвате на практика.

Вижте 25.17.3 Тестване/дебъгване с USB SmartCable, за да разберете как да тествате/дебъгвате кода с помощта на дебъгера.

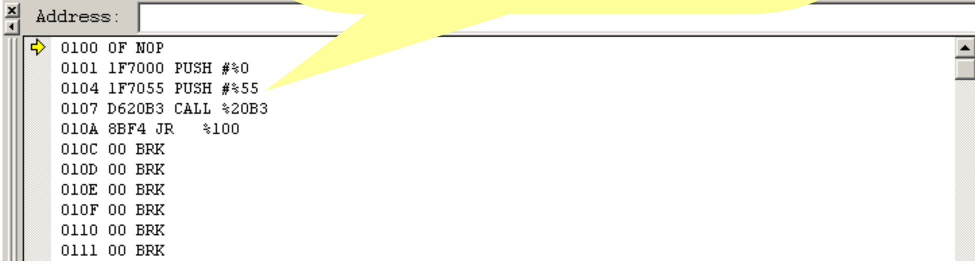
Следващите фигури илюстрират няколко от най-често използваните прозорци в средата ZDS II по време на тестване/дебъгване на кода.



Фиг. 208 Памет (Регистрова Памет, Програмна Памет, NVDS)

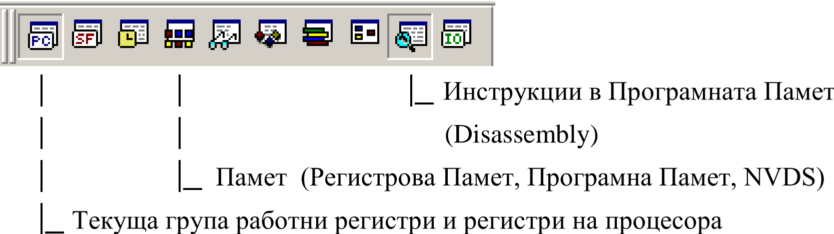

```
PC=  0100  SP=  0000  R0=     00  R1=     00  R2=     00  R3=     00  R4=     00  R5=     00
R6=    55  R7=    00  R8=     00  R9=     00  R10=    00  R11=    00  R12=    00  R13=    00
R14=   00  R15=   00  FLAGS=  00  RP=     00  RR0=  0000  RR2=  0000  RR4=  0000  RR6=  5500
RR8= 0000  RR10=0000  RR12=0000  RR14=0000  C=       0  Z=       0  S=       0  V=       0
D=      0  H=      0  U1=      0  U2=      0
```

Фиг. 209 Текуща група работни регистри и регистри на процесора



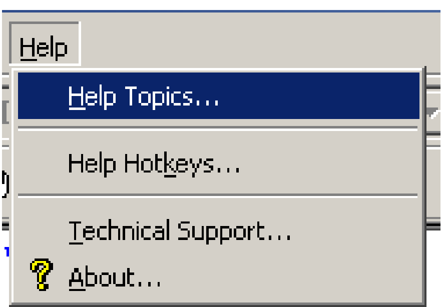
Фиг. 210 Инструкции в Програмната Памет (Disassembly)

Използвайте следната лента за показване на тези прозорци.



24.6 Заключение

В горните няколко подточки Ви описах най-важните неща за програмната среда ZDS II, с които да започнете. Експериментирайте колкото се може повече, за да добиете ясна представа кой прозорец каква информация Ви носи. За повече подробности използвайте вграденото в средата помощно меню.

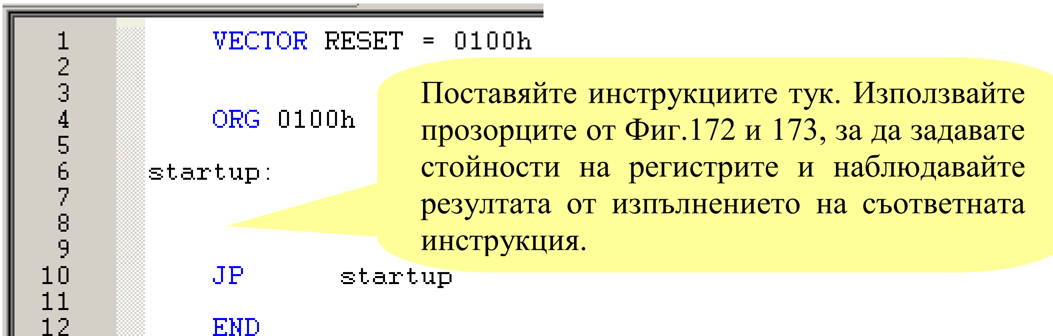


Фиг. 211 Помощно меню

Ако сте следвали съвета ми и сте прескочилите предните две глави, сега вече може да се върнете към тяхното изучаване.

За да тествате инструкциите така както са описани в 22 описание на инструкциите, конфигурирайте проекта да използва микроконтролер с повече RAM памет (вижте 24.4 конфигуриране на проекта). Например може да използвате микроконтролера Z8F6482XT.

Използвайте следната кратка програма като шаблон и експериментирайте с инструкциите, за да разберете как работят.



Накратко тази програма прави следното:

Редът **VECTOR RESET = 0100h** зарежда адресите, където е разположен ресет вектора (вижте отново Фиг.166 и 173), със стойността 0100h, т.е. . адресите 0002h:0003h ще се заредят със стойност 0100h.

```
Space:   Rom
Address: C#0000
C#0000  00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C#0013  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C#0026  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C#0039  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Директивата **ORG** указва адреса в Програмната Памет, който се асоциира с етикета startup (в случая 0100h). При стартиране програмата ще започне с изпълнението на инструкцията, разположена на този адрес.

Инструкцията **JP startup** извършва преход към инструкцията, разположена след етикета startup.

Директивата **END** указва, че това е края на сорс-файла. Всичко след тази директива ще се игнорира от Асемблера.

25 Микроконтролери от серията ZF083A

25.1 Основни характеристики

Z8Encore! F083A серията микроконтролери има следните характеристики:

- 20MHz процесор eZ8;
- До 8kB Флаш памет;
- До 256 байта Регистрова Памет (**RAM-памет**);
- 100 байта енергонезависима памет за данни (**NVDS - Nonvolatile Data Storage**);
- До 23 Входни-Изходни изводи;
- Вътрешен прецизен генератор;
- Два 16 битови многофункционални таймери;
- WDT-таймер със собствен вътрешен RC генератор;
- Еднопроводна двупосочна връзка с дебъгер;
- Бърз 8-канален, 10 битов АЦП;
- Аналогов компаратор;
- До 17 източника на прекъсване;
- Защита от временно понижаване на захранването (**VBO – Voltage Brown-Out**);
- Ресет при включване на захранването (**POR – Power - On Reset**);

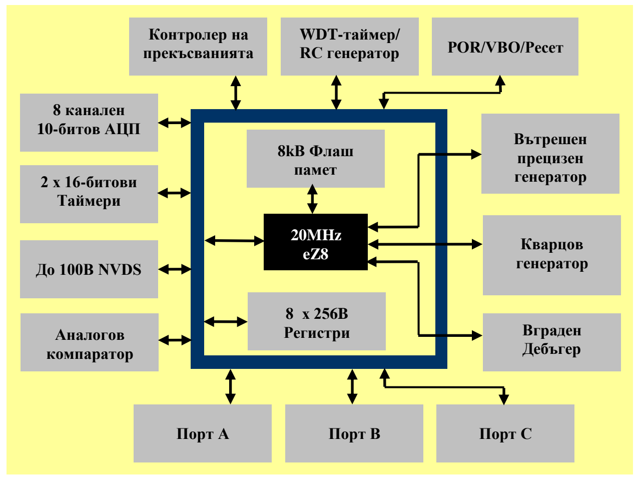
- Работно напрежение 2.6 ÷ 3.6V;
- До тринадесет 5V-толерантни изводи;
- 20- и 28-изводни корпуси;
- Работен температурен обхват:
    - Стандартен 0 ÷ 70 °C;
    - Разширен -40 ÷ 105 °C.

Табл.39 показва основните характеристики на микроконтролерите от серията Z8Encore! F083A.

Микроконтролер	Флаш (kB)	RAM (B)	NVDS (100B)	ADC	Входни/Изходни изводи
Z8F083A	8	256	Да	Да	17/23
Z8F043A	4	256	Да	Да	17/23

Табл. 39 Микроконтролери от серията Z8 Encore! F083A

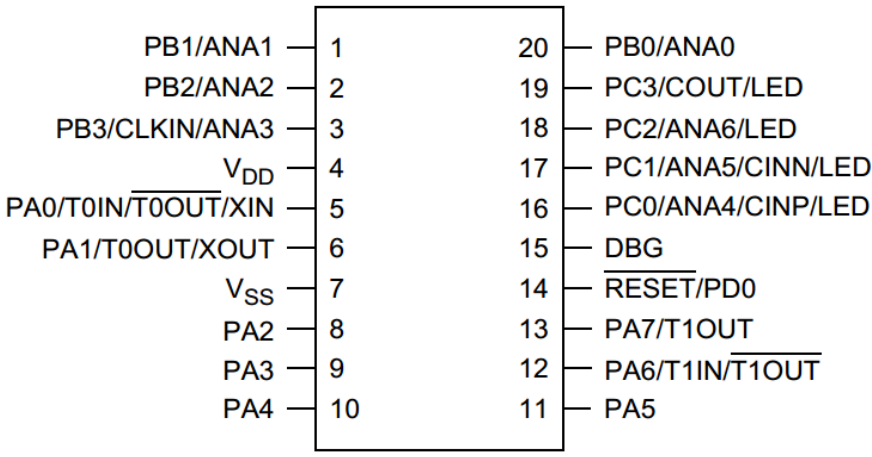
Следващата фигура показва блоковата схема на Z8Encore! F083A.



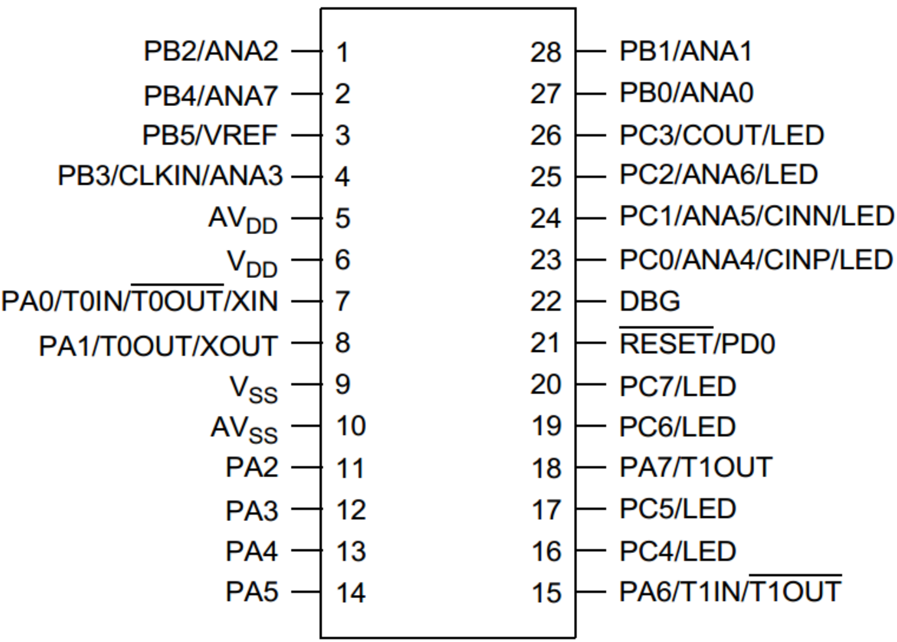
Фиг. 212 Блокова схема на Z8Encore! F083A

25.2 Описание на изводите

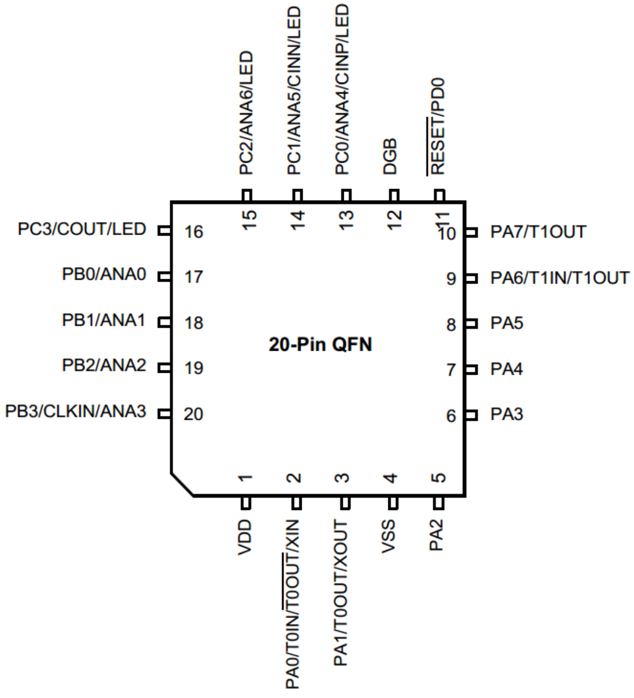
Следващите няколко фигури показва наличните корпуси, а Табл.40 описва функциите на всички изводи. Обърнете внимание, че някои изводи имат няколко различни функции.



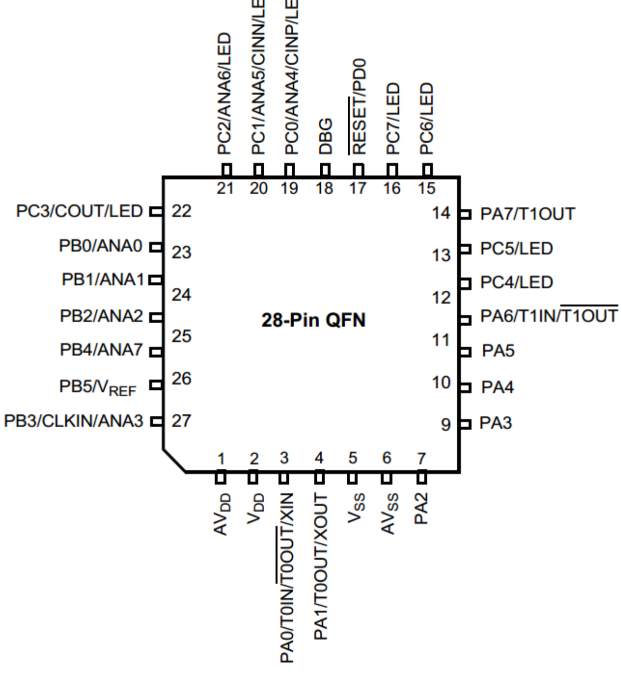
Фиг. 213 20-изводен SOIC, SSOP, PDIP корпус



Фиг. 214 28-изводен SOIC и SSOP корпус



Фиг. 215 20-изводен QFN корпус



Фиг. 216 28-изводен QFN корпус

В/И – Вход/Изход

Функция	В/И	Описание
		Портове
PA[7:0]	В/И	Порт A. Тези изводи се използват като GPIO[1]
PB[5:0]	В/И	Порт B. Тези изводи се използват като GPIO[1]
PC[7:0]	В/И	Порт C. Тези изводи се използват като GPIO[1]
PD[0]	И	Порт D. Този извод се използват само като изход
		Таймери
T0OUT/ T1OUT	И	Таймерен изход 0-1
$\overline{\text{T0OUT}}$/ $\overline{\text{T1OUT}}$	И	Инвертиран таймерен изход 0-1
T0IN/T1IN	В	Таймерен вход 0-1
		Компаратор
CINP/CINN	В	Компараторни входове (положителен и отрицателен)
COUT	И	Компараторен изход
		АЦП
ANA[7:0]	В	Аналогови входове за АЦП
VREF	В	Вход за опорно напрежение за АЦП
		Генератори
XIN	В	Вход за външен кварцов кристал
XOUT	И	Иход за външен кварцов кристал
CLKIN	В	Вход за външен TTL системен тактов сигнал
		LED драйвери
LED	И	LED изход. Всички изводи на порт C имат способността да управляват LED директно без допълнителни компоненти.
		Дебъгер
DBG	В/И	Вход/Изход за дебъгер. Този извод е с отворен дрейн и изисква външен изтеглящ (**pull up**) резистор.
		Ресет
$\overline{\text{RESET}}$	В/И	Вход за ресет. Също служи и за ресет индикатор.
		Захранване

V_{DD}	B	Захранване на цифровата част на устройството
AV_{DD}	B	Захранване на аналоговата част на устройството
V_{SS}	B	Маса на цифровата част на устройството
AV_{SS}	B	Маса на аналоговата част на устройството

Табл. 40 Описание на функциите на изводите

Забележка[1]**: GPIO (General Purpose Input/Output)** – Цифров Вход/Изход за Обща Употреба.

25.3 Адресно пространство

Регистрова памет

Z8Encore!F083A серията устройства имат до 256 байта RAM. Четене от регистър извън наличната RAM памет връща недефинирана стойност, а записът няма никакъв ефект.

Програмна памет

Z8Encore!F083A серията устройства имат до 8kB флаш памет. Четене от адрес извън наличната флаш памет връща FFh, а записът няма никакъв ефект. Табл.41 описва картата на паметта на наличните устройства.

Устройство	Адрес	Функция
ZF083A	0000h-0001h	Флаш-конфигурационни битове
	0002h-0003h	Ресет вектор
	0004h-003Dh	Таблица с вектори на прекъсванията
	003Eh-1FFFh	Програмна памет
ZF043A	0000h-0001h	Флаш-конфигурационни битове
	0002h-0003h	Ресет вектор
	0004h-003Dh	Таблица с вектори на прекъсванията
	003Eh-0FFFh	Програмна памет

Табл. 41 Карта на програмната памет

Памет за данни

Z8 Encore!F083A серията устройства не използва 64kB адресно пространство на процесора eZ8.

25.4 Карта на регистровата памет

Табл.42 описва всички регистри на Регистровата Памет, адресите, които заемат и стойностите им след ресет.

xx – недефинирана стойност

Адрес	Обозначение	Описание	Ресет
GPR-регистри			
000h-0FFh	-	RAM	xx
100h-1FFh	-	Резервиран	xx
SFR-регистри			
Таймер0			
F00h	T0H	Timer 0 High Byte	00
F01h	T0L	Timer 0 Low Byte	01
F02h	T0RH	Timer 0 Reload High Byte	FF
F03h	T0RL	Timer 0 Reload Low Byte	FF
F04h	T0PWMH	Timer 0 PWM High Byte	00
F05h	T0PWML	Timer 0 PWM Low Byte	00
F06h	T0CTL0	Timer 0 Control 0	00
F07h	T0CTL1	Timer 0 Control 1	00
Таймер1			
F08h	T1H	Timer 1 High Byte	00
F09h	T1L	Timer 1 Low Byte	01
F0Ah	T1RH	Timer 1 Reload High Byte	FF
F0Bh	T1RL	Timer 1 Reload Low Byte	FF
F0Ch	T1PWMH	Timer 1 PWM High Byte	00
F0Dh	T1PWML	Timer 1 PWM Low Byte	00
F0Eh	T1CTL0	Timer 1 Control 0	00
F0Fh	T1CTL1	Timer 1 Control 1	00
F10h-F6Fh	-	Резервиран	xx
АЦП			
F70h	ADCCTL0	ADC Control 0	00
F71h	-	Резервиран	xx
F72h	ADCD_H	ADC Data High Byte	xx

F73h	ADCDL	ADC Data Low Bits	xx
F74h	ADCSST	ADC Sample Settling Time	0F
F75h	ADCST	ADC Sample Time	3F
F76h	ADCCP	ADC Clock Prescale	00
F77h-F7Fh	-	Резервиран	xx
LPC (Low Power Control)			
F80h	PWRCTL0	Power Control 0	88
F81h	Резервиран	-	xx
LED контролер			
F82h	LEDEN	LED Drive Enable	00
F83h	LEDLVLH	LED Drive Level High	00
F84h	LEDLVLL	LED Drive Level Low	00
F85h	-	Резервиран	xx
Генератор			
F86h	OSCCTL	Oscillator Control	A0
F87h-F8Fh	-	Резервиран	xx
Компаратор0			
F90h	CMP0	Comparator 0 Control	14
F91h-FBFh	-	Резервиран	xx
Контролер на прекъсванията			
FC0h	IRQ0	Interrupt Request 0	00
FC1h	IRQ0ENH	IRQ0 Enable High Bit	00
FC2h	IRQ0ENL	IRQ0 Enable Low Bit	00
FC3h	IRQ1	Interrupt Request 1	00
FC4h	IRQ1ENH	IRQ1 Enable High Bit	00
FC5h	IRQ1ENL	IRQ1 Enable Low Bit	00
FC6h	IRQ2	Interrupt Request 2	00
FC7h	IRQ2ENH	IRQ2 Enable High Bit	00
FC8h	IRQ2ENL	IRQ2 Enable Low Bit	00
FC9h-FCCh	-	Резервиран	xx
FCDh	IRQES	Interrupt Edge Select	00
FCEh	IRQSS	Shared Interrupt Select	00
FCFh	IRQCTL	Interrupt Control	00
Порт A			

FD0h	PAADDR	Port A Address	00
FD1h	PACTL	Port A Control	00
FD2h	PAIN	Port A Input Data	xx
FD3h	PAOUT	Port A Output Data	00
Порт B			
FD4h	PBADDR	Port B Address	00
FD5h	PBCTL	Port B Control	00
FD6h	PBIN	Port B Input Data	xx
FD7h	PBOUT	Port B Output Data	00
Порт C			
FD8h	PCADDR	Port C Address	00
FD9h	PCCTL	Port C Control	00
FDAh	PCIN	Port C Input Data	xx
FDBh	PCOUT	Port C Output Data	00
Порт D			
FDCh	PDADDR	Port C Address	00
FDDh	PDCTL	Port C Control	00
FDEh	-	Резервиран	xx
FDFh	PDOUT	Port D Output Data	00
FE0h-FEFh	-	Резервиран	xx
WDT таймер			
FF0h	RSTSTAT	Reset Status	xx
	WDTCTL	WDT Control	xx
FF1h	WDTU	WDT Reload upper byte	FF
FF2h	WDTH	WDT Reload High Byte	FF
FF3h	WDTL	WDT Reload Low Byte	FF
FF4h-FF5h	-	Резервиран	xx
Флаш контролер			
FF6h	TRMADR	Trim Bit Address	00
FF7h	TRMDR	Trim Data	xx
FF8h	FCTL	Flash Control	00
	FSTAT	Flash Status	00
FF9h	FPS	Flash Page Select	00
	FPROT	Flash Sector Protect	00

FFAh	FFREQH	Flash Programming Frequency High Byte	00
FFBh	FFREQL	Flash Programming Frequency Low Byte	00
Процесорни регистри			
FFCh	FLAGS	Flags	xx
FFDh	RP	Register Pointer	xx
FFEh	SPH	Stack Pointer High Byte	xx
FFFh	SPL	Stack Pointer Low Byte	xx

Табл. 42 Карта на регистровата памет

25.5 Ресет и излизане от STOP-режим

25.5.1 Източници и типове ресет

Серията микроконтролери Z8Encore!F083A има следните източници на начално установяване (системен ресет) на микроконтролера:

- Включване на захранването (**POR – Power-On Reset**);
- Понижаване на захранването (**VBO – Voltage Brown-Out Reset**);
- Препълване на WDT-таймера[1];
- Ресет сигнал от външния извод $\overline{\text{RESET}}$.

Забележка[1]: WDT-таймерът ще предизвика ресет само ако тази функция е разрешена от флаш-конфигурационния бит **WDT_RES** (вижте 25.15 Флаш-конфигурационни битове).

Излизане от STOP-режим[1] се разглежда като форма на ресет на микроконтролера. Следващата таблица изброява типовете ресет и тяхната продължителност.

тип ресет	продължителност
системен ресет	66 такта на вътрешния прецизен генератор
системен ресет с разрешен кварцов генератор	5000 такта на вътрешния прецизен генератор
излизане от STOP-режим	66 такта на вътрешния прецизен генератор
излизане от STOP-режим с	5000 такта на вътрешния прецизен генератор

разрешен кварцов генератор	

Табл. 43 Типове ресет

Забележка[1]: Процесорът eZE8 има два режима на ниска консумация: STOP-режим и HALT-режим. Те са разгледани подробно в 25.8 Режими с ниска консумация

При ресет процесорът eZ8 и всички периферни модули са спрени, кварцовият генератор и вътрешният генератор на WDT-таймера продължават да работят, SFR-регистрите се зареждат с началните си стойности (вижте Табл.42), всички GPIO изводи се конфигурират като входове със забранени изтеглящи резистори. Извод PD0 е изключение и се конфигурира като двупосочен $\overline{\text{RESET}}$ извод с отворен дрейн. След излизане от ресет стойността на ресет вектора се зарежда в програмния брояч PC и програмата започва изпълнението си от този адрес. След ресет системният тактов сигнал винаги идва от вътрешния прецизен генератор. Табл.44 описва източниците на ресет на микроконтролера в различни работни режими на процесора.

Работен режим	Източник на ресет	Специални условия
Нормален или HALT-режим	POR/VBO	-
	Препълване на WDT-таймера	-
	$\overline{\text{RESET}}$ извод	Продължителността на импулса трябва да е минимум 4 системни такта.
	Дебъгер	-
STOP-режим	POR/BOR	-
	$\overline{\text{RESET}}$ извод	Продължителността на импулса трябва да е минимум 12ns.

	Ниско ниво на извод DBG	-

Табл. 44 Ресет на микроконтролера в различни работни режими

25.5.2 Описание на източниците на ресет

Включване на захранването (POR)

Всяко устройство от серията Z8 Encore! F083A има вътрешна POR схема, която наблюдава захранването на цифровата част и държи микроконтролера в ресет състояние, докато захранването превиши определено прагово напрежение V_{POR}, при което се стартира брояч. Микроконтролерът се държи в ресет състояние до препълване на брояча. Времето за препълване на брояча е по-дълго, ако кварцовият генератор е разрешен (вижте Табл.43). След излизане от ресет стойността на ресет вектора се зарежда в програмния брояч PC и програмата започва изпълнението си от този адрес, а бит POR в регистър RSTSTAT се установява в 1.

Понижаване на захранването (VBO)

Всяко устройство в серията Z8Encore!F083A има вътрешна VBO схема, която наблюдава захранването на цифровата част. Когато захранването падне под определено прагово напрежение V_{VBO} ($V_{VBO} < V_{POR}$), микроконтролерът влиза в ресет състояние и остава така, докато напрежението е по-малко от V_{POR}. След като напрежението превиши V_{POR}, се стартира брояч. Микроконтролерът се държи в ресет състояние до препълване на брояча. Времето за препълване на брояча е по-дълго, ако кварцовият генератор е разрешен (вижте Табл.43). След излизане от ресет стойността на ресет вектора се зарежда в програмния брояч PC и програмата

започва изпълнението си от този адрес, а бит POR в регистър RSTSTAT се установява в 1.

VBO схемата може да бъде изключена в STOP-режим с помощта на флаш-конфигурационния бит VBO_AO (вижте 25.15 Флаш-конфигурационни битове).

Препълване на WDT-таймера

Препълването на WDT-таймера инициира системен ресет, ако флаш-конфигурационният бит WDT_RES е 1. Ако флаш конфигурационният бит WDT_RES е 0, препълването на WDT-таймера генерира прекъсване, но не и системен ресет. При ресет от WDT-таймера бит WDT в регистър RSTSTAT се установява в 1.

Извод $\overline{\text{RESET}}$

Този извод има входен тригер на Шмит и вътрешен изтеглящ резистор. Преход от високо в ниско ниво на този извод, за минимум четири такта на системния генератор, инициира системен ресет. Ако извод $\overline{\text{RESET}}$ е в ниско ниво и след изтичане на периода на ресета, микроконтролерът остава в ресет, докато нивото на извода стане високо. След излизане от ресет бит EXT в регистър RSTSTAT се установява в 1.

Освен като вход за външен ресет на микроконтролера, този извод служи и за индикатор, че процесорът е в ресет състояние. След като настъпи ресет от вътрешен източник (POR, BOR, WDT-таймер) този извод се държи в ниско ниво от вътрешна схема до изтичане на ресет периода както е описано в Табл.43. Тази особеност позволява на микроконтролерите да установяват в начално състояние външни елементи, свързани към него, и по този начин да се синхронизират.

Дебъгер

Вградения в микроконтролера дебъгер също може да инициира системен POR ресет чрез установяване в 1 на бит RST в регистър OCDCTL. При излизане от ресет този бит автоматично се нулира и бит POR в регистър RSTSTAT се установява в 1.

25.5.3 Излизане от STOP-режим

Когато процесорът eZ8 изпълни инструкцията STOP, микроконтролерът влиза в режим на ниска консумация[1,2], наречен STOP-режим.

Забележка[1]: Често на жаргон вкарването на микроконтролера в режим с ниска консумация се нарича „заспиване" на микроконтролера.

Забележка[2]: Често на жаргон изкарването на микроконтролера от режим с ниска консумация се нарича „събуждане" на микроконтролера.

Излизането на микроконтролера от STOP-режим не влияе на регистрите с изключение на регистрите RSTSTAT и OSCCTL. При всяко излизане от STOP-режим системният тактов сигнал винаги идва от вътрешния прецизен генератор. Стойността съхранена на ресет вектора се зарежда в програмния брояч PC и програмата започва изпълнението си от този адрес, а бит STOP в регистър RSTSTAT се установява в 1.

Табл.45 описва източниците, които водят до излизане от STOP режим и действията, които съпровождат този процес.

източник	действие

Препълване от WDT-таймера, конфигуриран за ресет	Излизане от STOP-режим
Препълване от WDT-таймера, конфигуриран за прекъсване	Излизане от STOP-режим, последвано от прекъсване (ако прекъсването е разрешено)
Промяна на нивото на извод	Излизане от STOP-режим
$\overline{\text{RESET}}$ извод	Системен ресет
Ниско ниво на извод DBG	Системен ресет

Табл. 45 Източници на излизане от STOP-режим

Излизане от STOP-режим при препълване на WDT-таймера

Ако WDT-таймерът се препълни по време на STOP-режим, микроконтролерът преминава в нормален режим на работа. Битове WDT и STOP в регистър RSTSTAT се установяват в 1. Ако WDT-таймерът е конфигуриран за прекъсване, и микроконтролерът е конфигуриран да го обработи, то след излизане от STOP-режим процесорът обработва това прекъсване.

Излизане от STOP-режим при промяна на нивото на извод

Всеки от GPIO изводите може да се конфигурира да буди процесора при промяна на нивото на извода (↑ или ↓). Бит STOP в регистър RSTSTAT се установяват в 1.

В STOP-режим входните регистри за данни PxIN са забранени. Стойността на входен извод се записва в съответния бит за данни само ако сигналът остане на извода до края на процедурата по събуждане на процесора. Ето защо кратки импулси на такъв извод инициират събуждане на процесора, но не се записват в регистъра за данни, нито се стартира прекъсване (ако е разрешено за този извод).

Излизане от STOP-режим от извод $\overline{\text{RESET}}$

Ниско ниво на този извод по време на STOP-режим инициира системен ресет. Минималната широчина на импулса трябва да бъде 12ns. Бит EXT в регистър RSTSTAT се установява в 1.

Излизане от STOP-режим по ниско ниво на извод DBG

По време на STOP-режим дебъгерът инициира системен ресет при детектиране на определени грешни сигнали на извод DBG (ниско ниво с определена продължителност в зависимост от ситуацията). Дебъгерната част не се инициализира, докато останалата част на микроконтролера преминава през нормална процедура на системен ресет. Бит POR в регистър RSTSTAT се установява в 1.

25.5.4 Описание на управляващите регистри

Регистър RSTSTAT (Reset Status)

Табл.46 описва регистър RSTSTAT. Този регистър е само за четене. Четене на RSTSTAT нулира старшите четири бита. Този регистър използва същия адрес като регистър WDTCTL (Watchdog Timer Control Register), който е само за запис.

Ч/З – Четене/Запис

Бит	7	6	5	4	3	2	1	0
Име	POR	STOP	WDT	EXT				
Ресет	вижте Табл.47				0	0	0	0
Ч/З	Ч	Ч	Ч	Ч	Ч	Ч	Ч	Ч
Адрес	FF0h							

Бит	Описание
7	**Индикатор на POR ресет** Установява се в 1 ако настъпи POR ресет и се нулира, ако настъпи препълване от WDT-таймера или излизане от STOP-режим. Четене на регистъра също нулира този бит.
6	**Индикатор на излизане от STOP-режим** Установява се в 1 ако настъпи излизане от STOP-режим и се нулира при POR ресет или ресет от препълване на WDT-таймера, докато устройството не е в STOP-режим. Четене на регистъра също нулира този бит.
5	**Индикатор на препълване на WDT-таймера** Установява се в 1 ако настъпи препълване от WDT-таймера. POR ресет нулира този бит. Излизане от STOP-режим при промяна на нивото на извод също нулира този бит. Четене на регистъра нулира този бит.
4	**Индикатор на ресет от извод $\overline{\text{RESET}}$** Установява с в 1 при ресет от извод $\overline{\text{RESET}}$. POR ресет или излизане от STOP-режим при промяна на нивото на извод нулира този бит. Четене на регистъра също нулира този бит.
3 - 0	Тези битове са резервирани и трябва да бъдат 0.

Табл. 46 RSTSTAT

Събитие	POR	STOP	WDT	EXT
Ресет от POR	1	0	0	0
Ресет от извод $\overline{\text{RESET}}$	0	0	0	1
Ресет от препълване на WDT-таймера	0	0	1	0
Ресет от дебъгера (OCTCTL[1] = 1)	1	0	0	0
Ресет от STOP-режим при ниско ниво на извод DBG	1	0	0	0
Излизане от STOP-режим при промяна на нивото на извод	0	1	0	0

Излизане от STOP-режим при препълване на WDT-таймера	0	1	1	0

Табл. 47 Индикатори на източника на ресет

25.6 Контролер на прекъсванията

25.6.1 Общи сведения

Z8Encore!F083A серията микроконтролери има 17 източници на прекъсване, използващи 16 вектора на прекъсване, и 3 източника на аварийни прекъсвания.

- 12 прекъсвания от GPIO-изводи;
- 5 прекъсвания от периферни модули;
- Аварийно прекъсване от изпълнение на грешна инструкция;
- Аварийно прекъсване от грешка в основния генератор;
- Аварийно прекъсване от грешка в генератора на WDT-таймера.

Прекъсването от PA6 и Компаратора използва общ адрес, на който се разполага вектора на прекъсване.

Табл.48 описва всички източници на прекъсване, адресите в програмната памет, където се разполагат векторите на прекъсване и хардуерния приоритет на прекъсване. Един вектор на прекъсване се разполага с най-старшия байт на четен адрес и най-младшия байт на нечетен (big-endian).

Приоритет	Адрес	Източник на прекъсване
най-висок	0002h	Ресет (не е прекъсване)
	0004h	WDT-таймер
	0006h	Грешна инструкция (аварийно прекъсване)
	0008h	-
	000Ah	Таймер1
	000Ch	Таймер0

	000Eh	-
	0010h	-
	0012h	-
	0014h	-
	0016h	АЦП
	0018h	PA7 (избираем нарастващ или падащ фронт)
	001Ah	PA6 (избираем нарастващ или падащ фронт, или компараторен изход)
	001Ch	PA5 (избираем нарастващ или падащ фронт)
	001Eh	PA4 (избираем нарастващ или падащ фронт)
	0020h	PA3 (избираем нарастващ или падащ фронт)
	0022h	PA2 (избираем нарастващ или падащ фронт)
	0024h	PA1 (избираем нарастващ или падащ фронт)
	0026h	PA0 (избираем нарастващ или падащ фронт)
	0028h	-
	002Ah	-
	002Ch	-
	002Eh	-
	0030h	PC3 (по двата входни фронта)
	0032h	PC2 (по двата входни фронта)
	0034h	PC1 (по двата входни фронта)
	0036h	PC0 (по двата входни фронта)
	0038h	-
	003Ah	Основен генератор (аварийно прекъсване)
най-нисък	003Ch	RC генератор на WDT-таймера (аварийно прекъсване)

Табл. 48 Източници на прекъсване в ZF083A

Прекъсванията се разрешават с глобалния бит IRQE в регистър IRQCTL.

Бит IRQE се установява в 1 (прекъсванията са разрешени) при едно от следните действия:

- Изпълняване на инструкцията EI;

- Изпълняване на инструкцията IRET;
- Запис на 1 в бит IRQE.

Бит IRQE се установява в 0 (прекъсванията са забранени) от следните действия:

- Изпълняване на инструкцията DI;
- Приемане на прекъсване за обработка от eZ8 процесора;
- Запис на 0 в бит IRQE;
- Ресет;
- Изпълнение на инструкцията TRAP;
- Генериране на аварийно прекъсване от изпълнение на грешна инструкция;
- Генериране на аварийно прекъсване от грешка в главния генератор;
- Генериране на аварийно прекъсване от грешка в генератора на WDT-таймера.

Всеки източник на прекъсване има локален механизъм за разрешаване. Този механизъм също се използва за задаване и на софтуерен приоритет на прекъсванията. Процесорът eZ8 има 3 нива на софтуерен приоритет: ниво 3 (най-високо), ниво 1 (най-ниско). Ако възникнат няколко прекъсвания с еднакъв софтуерен приоритет, процесорът взема под внимание хардуерния приоритет. Прекъсването с най-висок хардуерен приоритет се обработва първо.

Ресет контролерът, прекъсването от WDT-таймера и трите

аварийни прекъсвания винаги имат най-високо софтуерно ниво на прекъсване. Всички останали източници на прекъсване имат по два бита, чрез които се извършва локално разрешаване и се задава софтуерно ниво на прекъсване. Табл.49 описва възможните комбинации на тези битове.

IRQnENH[x]	IRQnENL[x]	Софтуерен приоритет	Описание
0	0	-	Забранено
0	1	Ниво 1	Ниско
1	0	Ниво 2	Средно
1	1	Ниво 3	Високо

n = 0-2, x = 0-7

Табл. 49 Разрешаване и задаване на софтуерен приоритет

За повече информация вижте описанието на регистрите IRQnENH и IRQnENL (n = 0 - 2).

Когато процесорът стартира обработка на прекъсване, съответния флаг за заявка на прекъсването автоматично се нулира. Запис на 0 във флага също го нулира.

ZiLOG препоръчва следния начин за нулиране на флаг на прекъсване:

ANDX IRQ0, MASK

където

MASK съдържа 0 в позицията, която трябва да се нулира в регистър IRQ0.

Флаговете за заявка за прекъсване освен хардуерно, могат да се установяват в 1 и софтуерно. Софтуерните прекъсвания се обработват по същия начин както и хардуерните.

Пример:

```
OR    IRQ1, #0010000b
```

Тази инструкция установява бит 5 (заявка за прекъсване от PA5) в регистър IRQ1. Ако прекъсването е разрешено и няма друго активно прекъсване с по-висок приоритет, процесорът стартира неговата обработка.

25.6.2 Описание на управляващите регистри

Регистър IRQ0 (Interrupt Request 0)

Бит	7	6	5	4	3	2	1	0
Име		T1I	T0I					ADCI
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FC0h							

Бит	**Описание**
7	Този бит е резервиран и трябва да бъде 0.
6	**Заявка за прекъсване от Таймер1** 0 – няма заявка за прекъсване 1 – има заявка за прекъсване
5	**Заявка за прекъсване от Таймер0** 0 – няма заявка за прекъсване 1 – има заявка за прекъсване
4-1	Тези битове са резервирани и трябва да бъдат 0.
0	**Заявка за прекъсване от АЦП** 0 – няма заявка за прекъсване 1 – има заявка за прекъсване

Табл. 50 IRQ0

Регистър IRQ1 (Interrupt Request 1)

Бит	7	6	5	4	3	2	1	0
Име	PA7I	PA6CI	PA5I	PA4I	PA3I	PA2I	PA1I	PA0I
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FC3h							

Бит	**Описание**
7	**Заявка за прекъсване от PA7** 0 – няма заявка за прекъсване 1 – има заявка за прекъсване
6	**Заявка за прекъсване от PA6 или Компаратора** 0 – няма заявка за прекъсване 1 – има заявка за прекъсване
5-0	**Заявка за прекъсване от PAx (x = 5-0)** 0 – няма заявка за прекъсване 1 – има заявка за прекъсване

Табл. 51 IRQ1

Регистър IRQ2 (Interrupt Request 2)

Бит	7	6	5	4	3	2	1	0
Име					PC3I	PC2I	PC1I	PC0I
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FF6h							

Бит	**Описание**
7-4	Тези битове са резервирани и трябва да бъдат 0.
3-0	**Заявка за прекъсване от PCx (x = 3-0)** 0 – няма заявка за прекъсване 1 – има заявка за прекъсване

Табл. 52 IRQ2

Регистър IRQ0ENH (IRQ0 Enable High Bit)

Бит	7	6	5	4	3	2	1	0
Име		T1ENH	T0ENH					ADCENH
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FC1h							

Бит	Описание
7	Този бит е резервиран и трябва да бъде 0.
6	Старши бит за разрешаване на прекъсване от Таймер1
5	Старши бит за разрешаване на прекъсване от Таймер0
4-1	Тези битове са резервирани и трябва да бъдат 0.
0	Старши бит за разрешаване на прекъсване от АЦП

Табл. 53 IRQ0ENH

Регистър IRQ0ENL (IRQ0 Enable Low Bit)

Бит	7	6	5	4	3	2	1	0
Име		T1ENL	T0ENL					ADCENL
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч	Ч/З	Ч/З	Ч/З	Ч/З	Ч	Ч	Ч/З
Адрес	FC2h							

Бит	Описание
7	Този бит е резервиран и трябва да бъде 0.
6	Младши бит за разрешаване на прекъсване от Таймер1
5	Младши бит за разрешаване на прекъсване от Таймер0
4-1	Тези битове са резервирани и трябва да бъдат 0.
0	Младши бит за разрешаване на прекъсване от АЦП

Табл. 54 IRQ0ENL

Регистър IRQ1ENH (IRQ1 Enable High Bit)

Бит	7	6	5	4	3	2	1	0
Име	PA7ENH	PA6CENH	PA5ENH	PA4ENH	PA3ENH	PA2ENH	PA1ENH	PA0ENH
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FC4h							

Бит	Описание
7	Старши бит за разрешаване на прекъсване от PA7
6	Старши бит за разрешаване на прекъсване от PA6/Компаратор
5	Старши бит за разрешаване на прекъсване от PAx (x = 5-0)

Табл. 55 IRQ1ENH

Регистър IRQ1ENL (IRQ1 Enable Low Bit)

Бит	7	6	5	4	3	2	1	0
Име	PA7ENL	PA6CENL	PA5ENL	PA4ENL	PA3ENL	PA2ENL	PA1ENL	PA0ENL
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FC5h							

Бит	Описание
7	Младши бит за разрешаване на прекъсване от PA7
6	Младши бит за разрешаване на прекъсване от PA6/Компаратор
5	Младши бит за разрешаване на прекъсване от PAx (x = 5-0)

Табл. 56 IRQ1ENL

Регистър IRQ2ENH (IRQ2 Enable High Bit)

Бит	7	6	5	4	3	2	1	0

Име					C3ENH	C2ENH	C1ENH	C0ENH
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FC7h							

Бит	Описание
7-4	Тези битове са резервирани и трябва да бъдат 0.
3-0	Старши бит за разрешаване на прекъсване от PCx (x = 3-0)

Табл. 57 ERQ2ENH

Регистър IRQ2ENL (IRQ2 Enable Low Bit)

Бит	7	6	5	4	3	2	1	0
Име					C3ENL	C2ENL	C1ENL	C0ENL
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FC8h							

Бит	Описание
7-4	Тези битове са резервирани и трябва да бъдат 0.
3-0	Младши бит за разрешаване на прекъсване от PCx (x = 3-0)

Табл. 58 IRQ2ENL

Регистър IRQES (Interrupt Edge Select)

Бит	7	6	5	4	3	2	1	0
Име	IES7	IES6	IES5	IES4	IES3	IES2	IES1	IES0
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FCDh							

Бит	Описание

7-0	**Избор на активен фронт на прекъсване** 0 – Прекъсване по падащ фронт на PAx или PDx (x = 7-0) 1 – Прекъсване по нарастващ фронт на PAx или PDx (x = 7-0)

Табл. 59 IRQES

Регистър IRQSS (Shared Interript Select)

Бит	7	6	5	4	3	2	1	0
Име		PA6CS						
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FCEh							

Бит	**Описание**
7	Този бит е резервиран и трябва да бъде 0.
6	**Избор на източник на прекъсване от PA6/Компаратор** 0 – Прекъсване от PA6 1 – Прекъсване от Компаратора
5-0	Тези битове са резервирани и трябва да бъдат 0.

Табл. 60 IRQSS

Регистър IRQCTL (Interrupt Control)

Бит	7	6	5	4	3	2	1	0
Име	IRQE							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч	Ч	Ч	Ч	Ч	Ч	Ч
Адрес	FCFh							

Бит	**Описание**
7	**Глобален бит за разрешаване на прекъсванията** 0 – Прекъсванията са забранени 1 – Прекъсванията са разрешени

6-0	Тези битове са резервирани и трябва да бъдат 0.

Табл. 61 IRQCTL

25.7 Генераторен блок

25.7.1 Общи сведения

Z8Encore!F083A серията микроконтролери има 4 източника на тактови сигнали, които могат да се използват за системен тактов сигнал:

- Вътрешен прецизен настройваем RC генератор;
- Вътрешен генератор с външен кварцов или керамичен резонатор, или външна RC верига;
- Тактов сигнал от външен генератор;
- Ниско прецизен генератор на WDT-таймера.

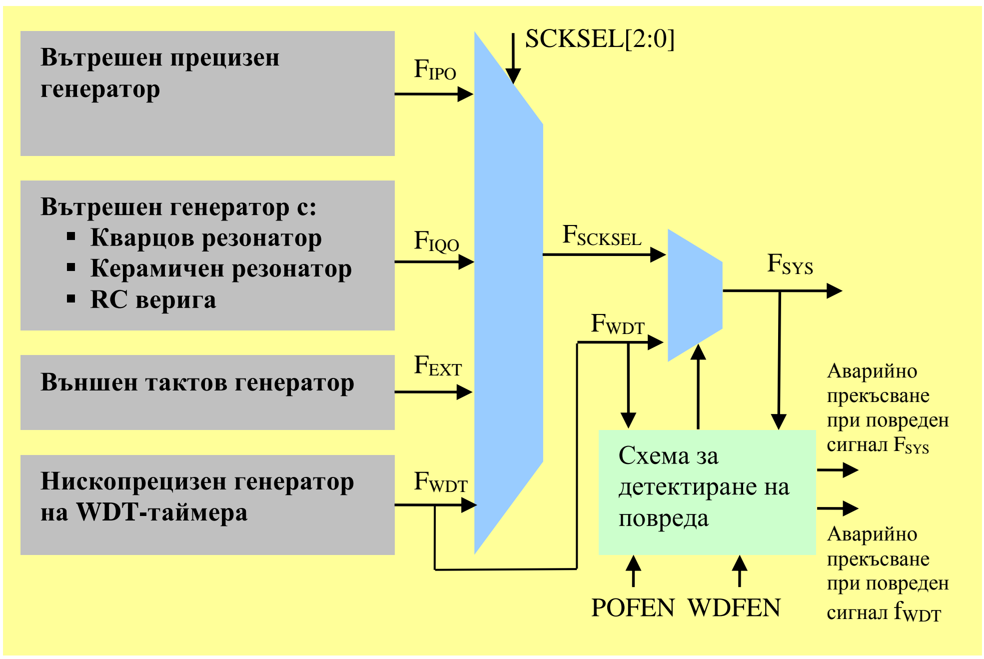
Табл.62 описва характеристиките на описаните по-горе източници на тактови сигнали.

Генератор	Характеристики
Вътрешен прецизен RC генератор	▪ 119kHz или 20MHz ▪ ±4% точност, когато е настроен ▪ не изисква външни компоненти
Вътрешен генератор с външен кварцов или керамичен резонатор	▪ 32kHz ÷ 20MHz ▪ много висока точност (зависи от използвания кварцов или керамичен резонатор) ▪ изисква външни компоненти
Вътрешен генератор с външна RC верига	▪ 32kHz ÷ 4MHz ▪ точността зависи от външни компоненти
Тактов сигнал от външен	▪ 0 ÷ 20MHz

генератор	▪ точността зависи от външния генератор
Нископрецизен генератор на WDT-таймера	▪ 10kHz ▪ ±40% точност ▪ не изисква външни компоненти ▪ ниска консумация-

Табл. 62 Генератори на тактови сигнали

В допълнение Z8Encore!F083A серията микроконтролери има схема за детектиране и възстановяване на повреден системен тактов сигнал. Следващата фигура илюстрира обобщения изглед на генераторния блок.



Фиг. 217 Генераторен блок

25.7.2 Детектиране и възстановяване на повреден системен тактов сигнал

Повреда в генератора на системния тактов сигнал

Z8Encore!F083A серията микроконтролери генерира немаскируемо прекъсване в случай на детектиране на повреден системен тактов сигнал. За да поддържа устройството работоспособно, детектиращата схема автоматично превключва RC генератора на WDT-таймера да бъде източник на системния тактов сигнал (този генератор трябва да бъде разрешен преди това). Тази функционалност не е налична, ако RC генераторът на WDT-таймера е забранен. Функционалността не е налична също, ако RC генераторът на WDT-таймера се използва като източник на системния тактов сигнал.

Повреда в системния тактов сигнал се детектира, ако честотата му падне по 1kHz ± 50%. Фалшива повреда може да се детектира и ако системният тактов сигнал се генерира от външен генератор с честота под допустимата. В този случай детектиращата схема може да бъде изключена (бит POFEN = 0 в регистър OSCCTL).

Повреда в RC генератора на WDT-таймера

Z8Encore!F083A серията микроконтролери генерира немаскируемо прекъсване в случай на детектиране на повреден тактов сигнал на RC генератора на WDT-таймера. Това не води до превключване на системния тактов сигнал. След детектиране на такава повреда, детектиране на повреда в системния тактов сигнал е невъзможна. Детектиращата схема не работи, ако RC генераторът на WDT-таймера е забранен или се използва като източник на системния тактов сигнал. Детектиращата схема може да бъде изключена (бит WDFEN = 0 в регистър OSCCTL).

Схемата за детектиране на грешка в RC генератора на WDT-таймера отброява 8004 системни такта, докато следи тактовия сигнал на RC генератора преди да определи дали има повреда. Скоростта на системния такт определя скоростта на детектиране на грешка в RC генератора на WDT-таймера. Много бавен системен такт води до много дълго време за детектиране.

25.7.3 Кварцов генератор

Кварцовият генератор използва външен кварцов резонатор, за да генерира честоти в обхвата 32KHz ÷ 20MHz. Може да се използва също и керамичен резонатор с честота до 8MHz или RC верига с честота до 4MHz. В допълнение входният извод XIN[1] също може да приема CMOS-тактов сигнал с честоти 32KHz ÷ 20MHz. Ако се използва външен тактов генератор, извод XOUT трябва да бъде оставен несвързан.

Забележка[1]: Въпреки че X_{IN} може да се използва като вход за външен тактов сигнал, извод CLKIN е по-подходящ за тази цел.

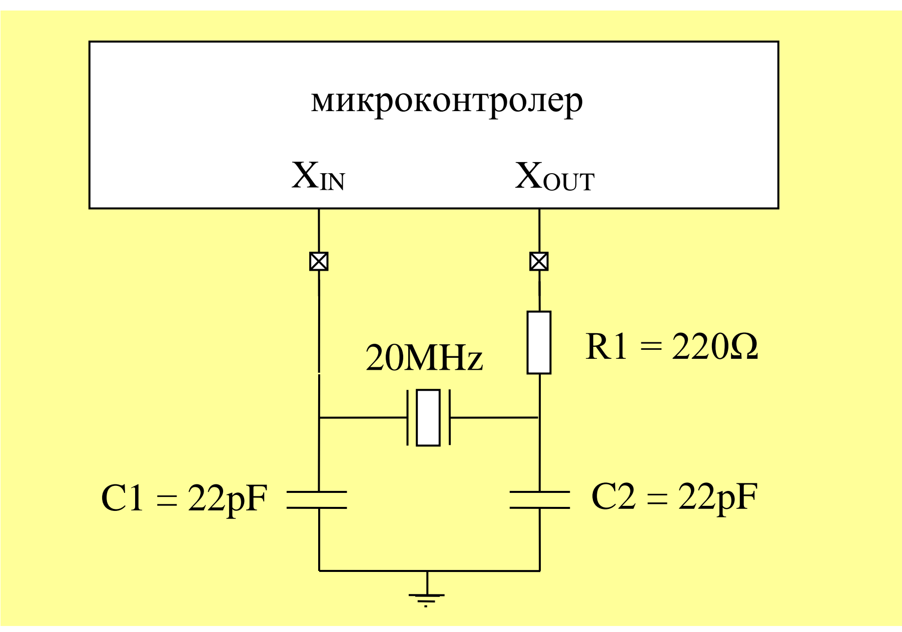
Кварцовият генератор поддържа 4 режима на работа:

- режим с ниска консумация с кварцови резонатори 32kHz ÷ 1MHz;
- режим със средна консумация с кварцови резонатори 0.5MHz ÷ 8MHz;
- режим с висока консумация с кварцови резонатори 8MHz ÷ 20MHz;
- външна RC верига (< 4MHz).

Режимът на работа се избира с флаш-конфигурационните битове (вижте 25.15 Флаш-конфигурационни битове).

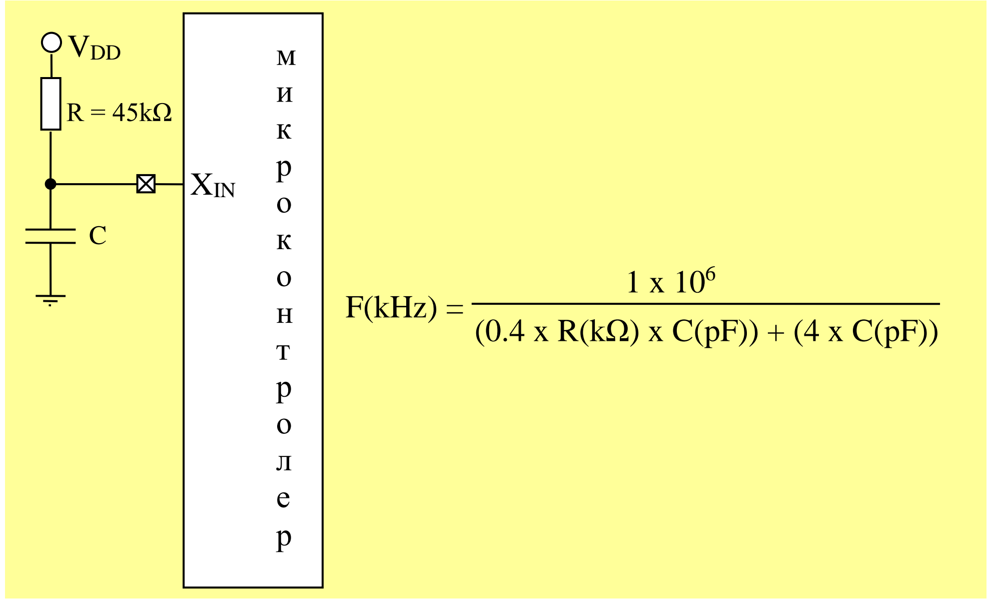
Кварцовият генератор може да бъде разрешен автоматично след ресет посредством флаш-конфигурационният бит XTLDIS или ръчно чрез бит XTLEN в регистър OSCCTL. В последния случай потребителският код трябва да изчака поне 5000 цикъла на вътрешния прецизен генератор за стабилизация на тактовия сигнал на кварцовия генератор. След изтичане на това време кварцовият генератор може да бъде избран за системен тактов генератор.

Следващата фигура показва препоръчителните стойности на външните компоненти на кварцовия генератор с 20MHz кварцов резонатор.



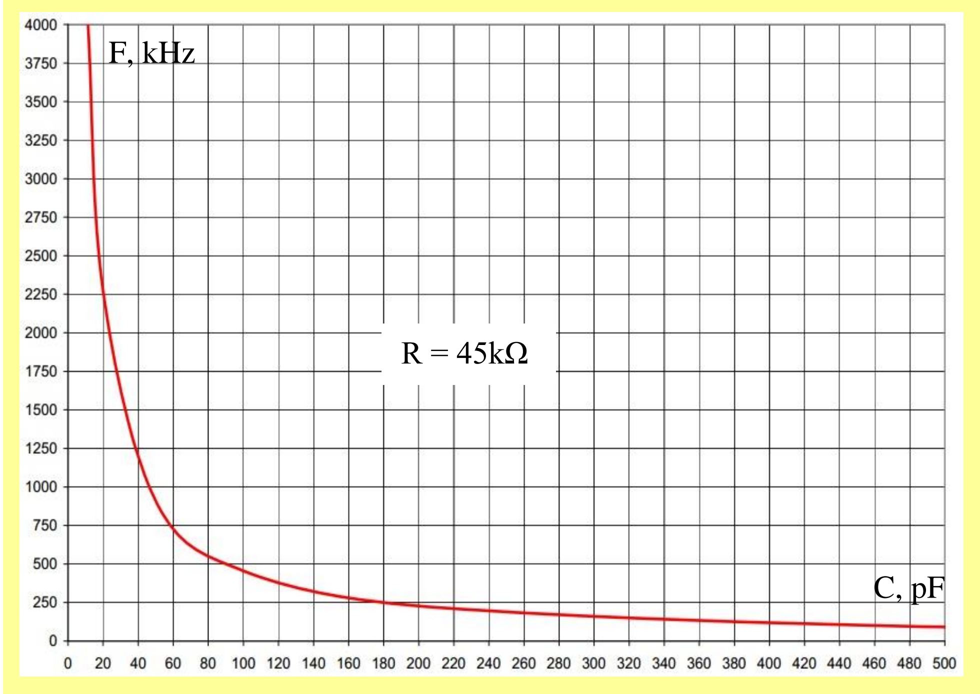
Фиг. 218 Външни компоненти на кварцов генератор

Следващата фигура показва използването на външна RC група.



Фиг. 219 Външни компоненти на RC генератор

Фиг.209 показва типични честоти при V_{DD} = 3.3V, T = 25°C и R = 45kΩ.



Фиг. 220 Типични честоти при R=25kΩ и различни стойнсти на C

	В RC режим на работа генераторът спира да осцилира, ако захранващото напрежение падне под 2.7V, но преди захранването да падне до V_{VBO}. Генераторът възстановява работата си, когато захранващото напрежение превиши 2.7V.

25.7.4 Вътрешен прецизен генератор

Вътрешният прецизен генератор (**IPO - Internal Precision Oscillator**) работи без външни компоненти с честота 20MHz или 119 kHz. Максималното време за стартиране на IPO е 25µs. IPO-генераторът е фабрично настроен на горните честоти.

25.7.5 Описание на управляващите регистри

Регистър OSCCTL (Oscillator Control Register)

Бит	7	6	5	4	3	2	1	0
Име	INTEN	XTLEN	WDTEN	POFEN	WDFEN	SCKSEL		
Ресет	1	0	1	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F86h							

Бит	**Описание**
7	**Разрешаване на IPO генератора** 0 – IPO е забранен 1 – IPO е разрешен
6	**Разрешаване на кварцовия генератор** 0 – Кварцовият генератор е забранен 1 – Кварцовият генератор е разрешен
5	**Разрешаване на RC генератора на WDT-таймера** 0 – RC генераторът е забранен 1 – RC генераторът е разрешен

4	**Разрешаване на детектиращата схема на системния такт** 0 – детектиращата схема е забранена 1 – детектиращата схема е разрешена
3	**Разрешаване на детектиращата схема на RC генератора на WDT-таймера** 0 – детектиращата схема е забранена 1 – детектиращата схема е разрешена
2-0	**Избор на източник на системен тактов сигнал** 000 – IPO с честота 20MHz 001 – IPO с честота 119kHz 010 – Кварцов генератор (кварцов или керамичен резонатор, RC) 011 – Генератор на WDT-таймера 100 – Външен тактов генератор 111 – Резервирана комбинация

Табл. 63 OSCCTL

25.8 Режими с ниска консумация

25.8.1 Общи сведения

Z8Encore!F083A серията микроконтролери има два режима на ниска консумация:

- STOP-режим;
- HALT-режим.

	Потребителят не трябва да разрешава изтеглящите резистори на неизползваните GPIO изводи.

STOP-режим

Микроконтролерът влиза в STOP-режим при изпълнение на инструкцията STOP. Работните характеристика на този режим са следните:

- Кварцовият и вътрешният прецизен генератор са спрени;
- Системният тактов сигнал е спрян;
- Процесорът eZ8 е спрян (не изпълнява инструкции, програмният брояч не се увеличава);
- RC генераторът на WDT-таймера продължава да работи (ако е разрешен);
- WDT-таймера продължава да работи (ако е разрешен);
- VBO схемата продължава да работи (ако е разрешена);
- Всички останали периферни модули са спрени.

За допълнително намаляване на консумацията, всички GPIO изводи, които са конфигурирани като цифрови входове трябва да бъдат отведени до V_{DD}, когато изтеглящите резистори са разрешени, или към една от захранващите шини (V_{DD} или GND), когато изтеглящите резистори са забранени.

За излизане от този режим вижте 25.5.3 Излизане от STOP-режим.

HALT-режим

Микроконтролерът влиза в HALT-режим при изпълнение на инструкцията HALT. Работните характеристика на този режим са следните:

- Системният тактов сигнал продължава да работи;
- Процесорът eZ8 е спрян (не изпълнява инструкции, програмният брояч не се увеличава);
- RC генераторът на WDT-таймера продължава да работи;
- WDT-таймера продължава да работи (ако е разрешен);
- Всички останали периферни модули продължават да работят.

За допълнително намаляване на консумацията, всички GPIO изводи, които са конфигурирани като цифрови входове, трябва да бъдат отведени до V_{DD}, когато изтеглящите резистори са разрешени, или към една захранващите шини (V_{DD} или GND), когато изтеглящите резистори са забранени.

Микроконтролерът излиза от този режим при възникване на

някое от следните събития:

- Прекъсване;
- Препълване на WDT-таймера (ресет или прекъсване);
- POR ресет;
- VBO ресет;
- Ресет от извод $\overline{\text{RESET}}$.

25.8.2 Описание на управляващите регистри

Регистър PWCTRL0 (Power Control Register 0)

Всеки бит в този регистър забранява даден периферен модул или чрез изключване на тактовия сигнал или чрез изключване на захранването към този модул.

Бит	7	6	5	4	3	2	1	0
Име				VBO			COMP	
Ресет[1]	1	0	0	0	1	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F80h							

Бит	**Описание**
7-5	Тези битове са резервирани и трябва да бъдат 0.
4	**Забраняване на VBO схемата** 0 – VBO схемата е разрешена 1 – VBO схемата е забранена
3-2	Тези битове са резервирани и трябва да бъдат 0.
1	**Забраняване на Компаратора** 0 – Компараторът е разрешен

	1 – Компараторът е забранен
0	Този бит е резервиран и трябва да бъде 0.

Табл. 64 PWRCTL0

Забележка[1]: Този регистър се установява в начално състояние само от POR ресет. Други видове ресет не оказват влияние върху него.

25.9 Портове

25.9.1 Общи сведения

Z8Encore!F083A серията микроконтролери имат максимум 23 входнi/изходни изводi за обща употреба (**GPIO – General Purpose Input Output**). Тези изводи са групирани в портове (Порт A ÷ Порт D). Всеки порт има набор от конфигурационни регистри, които настройват функцията на всеки извод (цифров вход/изход, аналогов вход, таймерен вход/изход и т.н.) и други (вижте 25.2 Описание на изводите).

Следващата таблица описва портовете и изводите в тях за различните устройства от Z8Encore!F083A серията.

Устройство	Корпус	АЦП	Порт A	Порт B	Порт C	Порт D	Брой изводи
Z8F083A Z8F043A	20-извода	Да	7:0	3:0	3:0	0	17
Z8F083A Z8F043A	28-извода	Да	7:0	5:0	7:0	0	23

Табл. 65 Брой налични изводи според корпуса

25.9.2 Алтернативни функции на изводите

Ако се върнете отново към точка 25.2 Описание на изводите, ще видите, че повечето изводи могат да изпълняват по няколко функции, а не само изводи за цифров вход/изход. Когато един извод е конфигуриран да работи в някой от алтернативните си режими, управлението му се поема от съответния периферен модул.

PD0 / $\overline{\text{RESET}}$

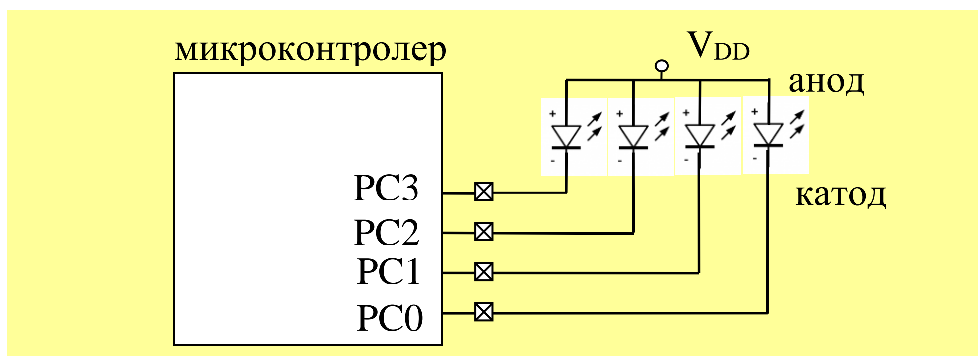
PD0 и $\overline{\text{RESET}}$ използват общ извод. За разлика от изводите на другите портове този извод не работи като GPIO след ресет, а като двупосочен (вход/изход) ресет извод с отворен-дрейн с вътрешен изтеглящ резистор. В GPIO-режим PD0 може да работи само като изход и трябва да бъде конфигуриран като такъв. PD0 може да работи като изход с повишена мощност, но не може да се използва да извежда микроконтролера от STOP-режим.

PA0/XIN и PA1/XOUT

Функционалността на кварцовия генератор не се управлява от GPIO блока. Когато този генератор е разрешен, GPIO функционалността на изводи PA0 и PA1 е изключена и те работят като XIN и XOUT съответно.

Директно управление на светодиоди

Изводите на Порт C могат да управляват директно светодиоди (**LED – Light Emitting Diode**). Изходите поемат ток с програмируеми нива 3mA, 7mA, 13mA и 20mA. Анодът на светодиода трябва да бъде свързан към V_{DD}, а катодът към извода (Фиг.221)



Фиг. 221 Свързване на светодиоди към Порт C

5V-толерантни изводи

Всички изводи, с изключение на изводите, които работят като аналогови входове, изводите на кварцовия генератор, изводите на компаратора, PA[1:0], PB[5:0] и PC[2:0], издържат на напрежение 5V.

Външен тактов сигнал

За устройства, които работят с външен TTL тактов сигнал, извод PB3 служи за тактов вход. В такъв случай конфигурирайте извода да използва алтернативната си функция CLKIN и чрез регистър OSCCTL изберете външния тактов сигнал за системен такт.

Прекъсвания от изводите

Много от изводите могат да генерират прекъсвания по нарастващ, падащ или и по двата фронта. За повече информация вижте 25.6 Контролер на прекъсванията.

Следващите таблици описват всички изводи на Порт A, B, C и D, алтернативните функции, които могат да изпълняват и управляващите битове, с които се избира съответната алтернативна функция.

Порт А			
Извод	**Алтернативна Функция**	**Описание**	**Регистър**
PA0	T0IN/ $\overline{\text{T0OUT}}$	Таймерен вход 0/ Инвертиран таймерен изход 0	PAAF[0] = 1
PA1	T0OUT	Таймерен изход 0	PAAF[1] = 1
PA2	-	-	-
PA3	-	-	-
PA4	-	-	-
PA5	-	-	-
PA6	T1IN/ $\overline{\text{T1OUT}}$	Таймерен вход 1/ Инвертиран	PAAF[6] = 1

		таймерен изход 1	
PA7	T1OUT	Таймерен изход 1	PAAF[7] = 1

Табл. 66 Алтернативни функции на Порт А

Порт B			
Извод	**Алтернативна Функция**	**Описание**	**Регистър**
PB0	-	-	AFS1[0] = 0
	ANA0	Аналогов вход 0	AFS1[0] = 1
PB1	-	-	AFS1[1] = 0
	ANA1	Аналогов вход 1	AFS1[1] = 1
PB2	-	-	AFS2[2] = 0
	ANA2	Аналогов вход 2	AFS2[2] = 1
PB3	CLKIN	Вход за външен тактов сигнал	AFS1[3] = 0
	ANA3	Аналогов вход 3	AFS1[3] = 1
PB4	-	-	AFS1[4] = 0
	ANA7	Аналогов вход 7	AFS1[4] = 1
PB5	-	-	AFS1[5] = 0
	VERF	Вход за опорно напрежение	AFS1[5] = 1
PB6	-	-	AFS1[6] = 0
	-	-	AFS1[6] = 1
PB7	-	-	AFS1[7] = 0
	-	-	AFS1[7] = 1

Табл. 67 Алтернативни функции на Порт B

Порт C			
Извод	**Алтернативна Функция**	**Описание**	**Регистър**
PC0	-	-	AFS2[0] = 0
	ANA4/CINP	Аналогов вход 4/Компараторен вход +	AFS2[0] = 1
PC1	-	-	AFS2[1] = 0
	ANA5/CINN	Аналогов вход 5/Компараторен вход -	AFS2[1] = 1
PC2	-	-	AFS2[2] = 0
	ANA6	Аналогов вход 6	AFS2[2] = 1
PC3	COUT	Компараторен изход	AFS2[3] = 0
	-	-	AFS2[3] = 1

PC4	-	-	AFS2[4] = 0
	-	-	AFS2[4] = 1
PC5	-	-	AFS2[5] = 0
	-	-	AFS2[5] = 1
PC6	-	-	AFS2[6] = 0
	-	-	AFS2[6] = 1
PC7	-	-	AFS2[7] = 0
	-	-	AFS2[7] = 1

Табл. 68 Алтернативни функции на Порт C

Порт D			
Извод	**Алтернативна Функция**	**Описание**	**Регистър**
PD0	$\overline{\text{RESET}}$	Външен ресет. По подразбиране работи като ресет извод.	PDAF[0] = 1

Табл. 69 Алтернативни функции на Порт D

25.9.3 Управляващи регистри

Всеки порт има 4 управляващи регистри (Табл.70).

Регистър	Описание
PxADDR	Адресен регистър (избира подрегистър)
PxCTL	Управляващ регистър (осъществява достъп до подрегистър)
PxIN	Входен регистър
PxOUT	Изходен регистър

x = A,B,C и D

Табл. 70 Управляващи регистри

Освен тези регистри всеки порт има и няколко управляващи подрегистри (Табл.71).

Подрегистър	Адрес	Описание
PxDD	01h	Управление на посоката на данните (вход или изход)
PxAF	02h	Разрешаване на алтернативна функция
PxOC	03h	Изходен контрол (отворен-дрейн)
PxHDE	04h	Разрешаване на мощно управление
PxSMRE	05h	Излизане от STOP-режим
PxPUE	06h	Разрешаване на изтеглящите резистори
PxAFS1	07h	Алтернативна функция регистър 1
PxAFS2	08h	Алтернативна функия регистр 2

x = A,B,C и D

Табл. 71 Управляващи подрегистри

Подрегистрите не са част от Регистровата Памет като всички останали управляващи регистри, а се адресират чрез регистър PxADDR (x = A, B, C и D). Достъпът до подрегистър става чрез регистър PxCTL (x = A, B, C и D). Например ако PAADDR = 01h, достъпът до регистър PACTL осъществява достъп до подрегистър PADD.

25.9.4 Описание на управляващите регистри

Регистър PxADDR (Port Address)

PxADDR регистъра служи като указател към управляващите подрегистри на всеки порт.

Бит	7	6	5	4	3	2	1	0
Име	PADDR[7:0]							
Ресет	00h							
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FD0h, FD4h, FD8h, FDCh							
Бит	**Описание**							

7-0	**Адрес на подрегистър** 00 – 01 – PxDD 02 – PxAF 03 – PxOC 04 – PxHDE 05 – PxSMRE 06 – PxPUE 07 – PxAFS1 08 – PxAFS2

Табл. 72 PxADDR

Регистър PxCTL (Port Control)

PxCTL регистърът служи за достъп до управляващите подрегистри на всеки порт. Регистър PxADDR определя подрегистъра.

Бит	7	6	5	4	3	2	1	0
Име	PCTL[7:0]							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FD1h, FD5h, FD9h, FDDh							
Бит	**Описание**							
7-0	Осъществява достъп до подрегистъра, сочен от PxADDR							

Табл. 73 PxCTL

Регистър PxDD (Port Direction)

Този регистър управлява посоката на изводите, когато изводът е конфигуриран като GPIO. В алтернативен режим на работа, посоката на извода се управлява от съответния периферен модул.

Бит	7	6	5	4	3	2	1	0
Име	DD[7:0]							
Ресет	1	1	1	1	1	1	1	1
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	PxADDR = 01h							

Бит	Описание
7-0	**Посока на данните** 0 – Цифров Изход. Данните в регистър PxOUT се извеждат на извода. 1 – Цифров вход. Стойността на извода се чете и се записва в PxIN.

Табл. 74 PxDD

Регистър PxAF(Port Alternate Function)

Този регистър разрешава алтернативната функция на съответния извод. Конкретната алтернативна функция на извода се избира с регистрите PxAFS1 и PxAFS2 (вижте Табл.80 и 81). Изключение са Порт A и D, при които алтернативната функция на извода се разрешава и избира директно чрез регистър PxAF (вижте Табл.66 и 69).

Бит	7	6	5	4	3	2	1	0
Име	AF[7:0]							
Ресет	00h (Порт A-C), 01h (Порт D)							
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	PxADDR = 02h							

Бит	Описание
7-0	**Разрешаване на алтернативна функция на извод** 0 – Алтернативната функция е забранена. Изводът работи като GPIO (цифров вход или изход). 1 – Алтернативната функция е разрешена.

Табл. 75 PxAF

Регистър PxOC (Port Output Control)

Този регистър конфигурира специфицирания извод като отворен-дрейн независимо дали извода работи в GPIO или алтернативен режим.

Бит	7	6	5	4	3	2	1	0
Име	POC[7:0]							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	PxADDR = 03h							

Бит	**Описание**
7:0	**Управление на изхода** 0 – Режим отворен-дрейн забранен 1 – Режим отворен-дрейн разрешен

Табл. 76 PxOC

Регистър PxHDE (Port High Drive Enable)

Този регистър конфигурира специфицирания извод за мощни изходни токови операции независимо дали извода работи в GPIO или алтернативен режим.

Бит	7	6	5	4	3	2	1	0
Име	PHDE[7:0]							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	PxADDR = 04h							

Бит	**Описание**
7:0	**Разрешаване на мощно управление** 0 – Изводът е конфигуриран за стандартни изходни токови операции

	1 – Изводът е конфигуриран за мощни изходни токови операции

Табл. 77 PxHDE

Регистър PxSMRE (Port Stop Mode Recovery Enable)

Този регистър разрешава специфицирания извод да извежда процесора от STOP-режим Всеки преход на извода (нарастващ или падащ фронт) стартира процедура по извеждане на процесора от STOP-режим (вижте 25.5.3 Излизане от STOP-режим).

Бит	7	6	5	4	3	2	1	0
Име	PSMRE[7:0]							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	PxADDR = 05h							

Бит	**Описание**
7-0	0 – Излизането от STOP-режим забранено 1 – Излизането от STOP-режим разрешено

Табл. 78 PxSMRE

Регистър PxPUE (Port Pull-up enable)

Този регистър разрешава вътрешния изтеглящ резистор на изводите на Порта A÷D.

Бит	7	6	5	4	3	2	1	0
Име	PPUE[7:0]							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	PxADDR = 06h							

Бит	**Описание**
7:0	0 – Изтеглящият резистор е забранен 1 – Изтеглящият резистор е разрешен

Табл. 79 PxPUE

Регистър PxAFS1 (Port Alternate Set 1)

Този регистър избира алетрнативната функция на специфицирания извод на Порт B (вижте Табл.67). Алтернативната функция на извода също трябва да бъде разрешена в регистър PxAF.

Бит	7	6	5	4	3	2	1	0
Име	PAFS1[7:0]							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	PxADDR = 07h							
Бит	**Описание**							
7:0	0 – Вижте Табл. 67 1 – Вижте Табл. 67							

Табл. 80 PxAFS1

Регистър PxAFS2 (Port Alternate Set 2)

Този регистър избира алетрнативната функция на специфицирания извод на Порт C (вижте Табл.68). Алтернативната функция на извода също трябва да бъде разрешена в регистър PxAF.

Бит	7	6	5	4	3	2	1	0
Име	PAFS2[7:0]							

Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	PxADDR = 08h							
Бит	**Описание**							
7:0	0 – Вижте Табл. 68 1 – Вижте Табл. 68							

Табл. 81 PxAFS2

Регистър PxIN (Port Input Data)

Когато специфицираният извод е конфигуриран като цифров вход, нивото на този извод (лог.0 или 1) се записва в регистър PxIN (x = A, B и C). Извод PD0 не може да работи като цифров вход.

Бит	7	6	5	4	3	2	1	0
Име	PIN[7:0]							
Ресет	x	x	x	x	x	x	x	x
Ч/З	Ч	Ч	Ч	Ч	Ч	Ч	Ч	Ч
Адрес	FD2h, FD6h, FDAh							
Бит	**Описание**							
7:0	0 – Логическа 0 на входа 1 – Логическа 1 на входа							

Табл. 82 PxIN

Регистър PxOUT (Port Output Data)

Данните в този регистър се извеждат на съответните изводи, когато са конфигурирани като цифрови изходи.

Бит	7	6	5	4	3	2	1	0
Име	POUT[7:0]							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FD3h, FD7h, FDBh, FDFh							

Бит	Описание
7:0	0 – Извеждане на лог.0 на извода 1 – Извеждане на лог.1 на извода. Логическата 1 не се извежда на извода, ако той е конфигуриран в режим отворен-дрейн (вижте Регистър PxOC).

Табл. 83 PxOUT

Регистър LEDEN (LED Drive Enable)

Този регистър включва вътрешните схеми на изводите на Порт C, които директно могат да управляват светодиоди.

Бит	7	6	5	4	3	2	1	0
Име	LEDEN[7:0]							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F82h							

Бит	Описание
7:0	0 – Вътрешната схема за управление на LED е изключна 1 – Вътрешната схема за управление на LED е включена

Табл. 84 LEDEN

Когато извод PCx (x = 0÷7) е конфигуриран да работи в LED режим, битовете в регистрите LEDLVLH и LEDLVLL конфигурират големината на тока (Табл.85).

LEDLVLHx	LEDLVLLx	LED ток
0	0	3mA
0	1	7mA
1	0	13mA
1	1	20mA

x = 0÷7

Табл. 85 LED големина на тока

Регистър LEDLVLH (LED Drive Level High)

Бит	7	6	5	4	3	2	1	0
Име	LEDLVLH[7:0]							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F83h							
Бит	**Описание**							
7:0	Вижте Табл.85							

Табл. 86 LEDLVLH

Регистър LEDLVLL (LED Drive Level Low)

Бит	7	6	5	4	3	2	1	0
Име	LEDLVLL[7:0]							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F84h							
Бит	**Описание**							
7:0	Вижте Табл.85							

Табл. 87 LEDLVLL

25.10 WDT таймер

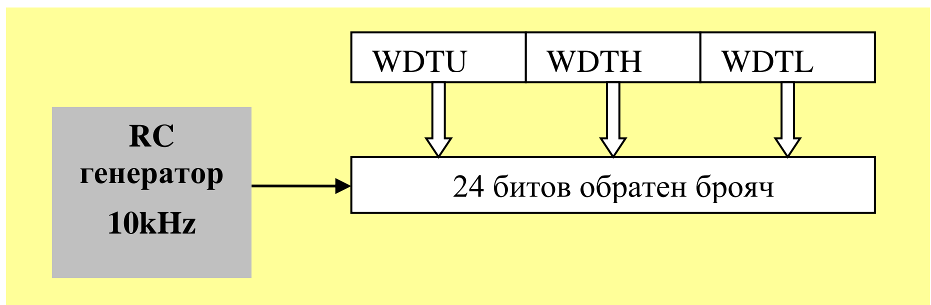
25.10.1 Общи сведения

WDT-таймерът защитава програмата от „зависване”. Той има следните свойства:

- Собствен вътрешен RC генератор на тактов сигнал;
- 24 битов програмируем период на препълване;
- Избираема реакция при препълване: ресет или прекъсване.

25.10.2 Устройство и принцип на работа

Фиг.222 показва блоковата схема на WDT-таймера.



Фиг. 222 Блокова схема на WDT-таймера

Веднъж разрешен таймерът брои постоянно и трябва да бъде обновяван периодично, за да се избегне препълване. WDT-таймерът се разрешава чрез изпълнение на инструкцията WDT или чрез флаш-конфигурационния бит WDT_AO. Този флаш-конфигурационен бит включва таймера веднага след излизане на процесора от ресет състояние, дори инструкцията WDT да не е била изпълнена.

Таймерът работи в режим на обратно броене. При разрешаването му 24 битовата стойност, формирана от 8 битовите регистри WDTU, WDTH и WDTL, се зарежда в брояча. На всеки тактов сигнал от RC генератора броячът се намалява с 1. Броячът се намалява до 000000h, освен ако не се изпълни инструкцията WDT. Изпълнението на тази инструкция принуждава презареждане на брояча с 24 битовата стойност от регистрите WDTU, WDTH и WDTL и броенето се възобновява. По този начин се избягва изтичането на периода на брояча и генериране на ресет или прекъсване от WDT-таймера.

Времето за изтичане на периода на таймера се изчислява по следната формула:

$$T_{WDT}(ms) = \frac{Стойност \quad за \quad презареждане}{10}$$

където *Стойност за презареждане* е 24 битовата десетична стойност формирана от 8 битовите регистри WDTU, WDTH и WDTL.

WDT-таймерът не може да се презареди след като достигне 000002h. Презареждащата стойност не трябва да бъде под 000004h. Табл.88 дава информация за приблизителното време за препълване на таймера при минимална и максимална презареждаща стойност.

Презареждаща стойност	**Време за препълване**	**Описание**
000004h	400μs	минимално
000400h	102ms	по подразбиране
FFFFFFh	28 минути	максимално

Табл. 88 Времена на препълване на таймера

WDT-таймерът се препълва (изтича периодът на броене),

когато броячът достигне 000000h. Реакцията при препълване може да бъде ресет или прекъсване. Това зависи от флаш-конфигурационния бит WDT_RES.

25.10.3 Реакция при препълване на WDT-таймера

Генериране на прекъсване в нормален режим

Ако е конфигуриран да генерира прекъсване по препълване, WDT-таймерът изпраща заявка за прекъсване към Контролерът на прекъсванията и вдига флаг WDT в регистър RSTSTAT. Ако прекъсванията са разрешени, процесорът извлича вектора на прекъсване и изпълнява кода, намиращ се на този адрес. WDT-таймерът се зарежда с максималната си стойност FFFFFFh и продължава да брои. Таймерът не се презарежда автоматично с презареждащата стойност в регистрите WDTU, WDTH и WDTL.

Регистър RSTSTAT трябва да бъде прочетен преди изчистване на прекъсването от таймера. Това четене изчиства флага WDT.

Генериране на прекъсване в STOP-режим

Ако е конфигуриран да генерира прекъсване по препълване и процесорът е в STOP-режим, WDT-таймерът автоматично инициира излизане от STOP-режим, генерира заявка за прекъсване и вдига флаговете WDT и STOP в регистър RSTSTAT. За повече детайли относно излизане от STOP-режим вижте 25.5.3 Излизане от STOP-режим.

Ако прекъсванията са разрешени, след излизане от STOP-режим процесорът извлича вектора на прекъсване и изпълнява кода, намиращ се на този адрес.

Генериране на ресет в нормален режим

Ако е конфигуриран да генерира ресет по препълване, WDT-таймерът вкарва процесорът в ресет състояние и вдига флага WDT в регистър RSTSTAT. За повече информаци относно ресет състоянието вижте 25.5 Ресет и излизане от STOP-режим.

Генериране на ресет в STOP-режим

Ако е конфигуриран да генерира ресет по препълване и процесорът е в STOP-режим, WDT-таймерът автоматично инициира излизане от STOP-режим и вдига флаговете WDT и STOP в регистър RSTSTAT. За повече детайли относно излизане от STOP-режим вижте 25.5.3 Излизане от STOP-режим.

25.10.4 Запис в регистрите WDTU, WDTH и WDTL

Директният записът в тези регистри е забранен и изисква специална отключваща последователност.

1. Запишете 55h в регистър WDTCTL;
2. Запишете AAh в регистър WDTCTL;
3. Запишете стойност в WDTU;
4. Запишете стойност в WDTH;
5. Запишете стойност в WDTL.

Презареждащите регистри трябва да бъдат записани в указания по-горе ред. Не извършвайте други междинни записи между тези операции, в противен случай

заключващият механизъм се активира отново. Презареждащата стойност в тези регистри се зарежда в брояча при първото разрешаване на таймера и всеки път при изпълняване на инструкцията WDT.

25.10.5 Описание на управляващите регистри

Регистър WDTCTL (Watchdog Timer Control)

Този регистър е само за запис. Запис на отключващата последователност 55h AAh в него отключва достъпа до презареждащите регистри WDTU, WDTH и WDTL. Операциите за запис в регистър WDTCTL нямат ефект върху неговите битовете.

Бит	7	6	5	4	3	2	1	0
Име	WDTUNLK							
Ресет	x	x	x	x	x	x	x	x
Ч/З	З	З	З	З	З	З	З	З
Адрес	FF0h							

Бит	**Описание**
7:0	Потребителският софтуер трябва да запише правилната отключваща последователност в този регистър преди да запише презареждащите регистри

Табл. 89 WDTCTL

Регистър WDTU (Watchdog Timer Reload Upper Byte)

Бит	7	6	5	4	3	2	1	0
Име	WDTU							
Ресет	x	x	x	x	x	x	x	x
Ч/З	Ч[1]/З	Ч[1]/З	Ч[1]/З	Ч[1]/З	Ч[1]/З	Ч[1]/З	Ч[1]/З	Ч[1]/З

Адрес	FF1h
Бит	**Описание**
7:0	Битове 23:16 на презареждащата стойност

Табл. 90 WDTU

Забележка[1]: Четенето на този регистър връща текущата стойост на брояча.

Регистър WDTH (Watchdog Timer Reload High Byte)

Бит	7	6	5	4	3	2	1	0
Име	WDTH							
Ресет	x	x	x	x	x	x	x	x
Ч/З	Ч[1]/З	Ч[1]/З	Ч[1]/З	Ч[1]/З	Ч[1]/З	Ч[1]/З	Ч[1]/З	Ч[1]/З
Адрес	FF2h							

Бит	Описание
7:0	Битове 15:8 на презареждащата стойност

Табл. 91 WDTH

Забележка[1]: Четенето на този регистър връща текущата стойност на брояча.

Регистър WDTL (Watchdog Timer Reload Low Byte)

Бит	7	6	5	4	3	2	1	0
Име	WDTL							
Ресет	x	x	x	x	x	x	x	x
Ч/З	Ч[1]/З	Ч[1]/З	Ч[1]/З	Ч[1]/З	Ч[1]/З	Ч[1]/З	Ч[1]/З	Ч[1]/З

Адрес	FF3h
Бит	**Описание**
7:0	Битове 7:0 на презареждащата стойност

Табл. 92 WDTL

Забележка[1]: Четенето на този регистър връща текущата стойност на брояча.

25.11 Таймери

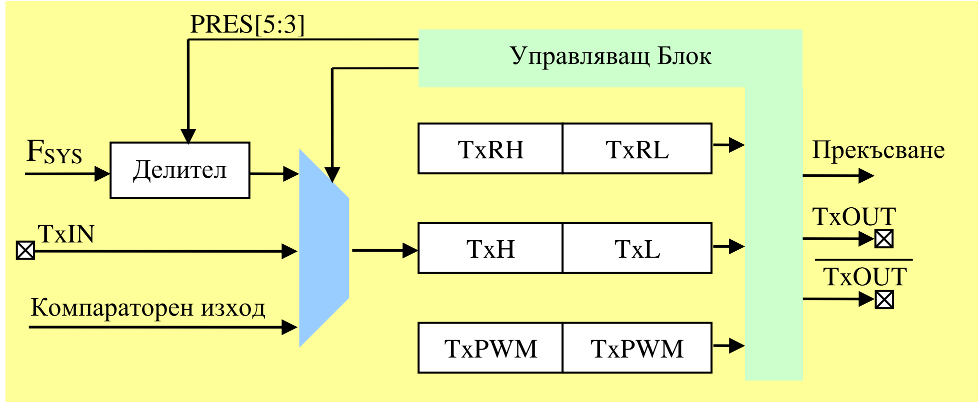
25.11.1 Общи сведения

Z8Encore!F083A серията микроконтролери съдържа два идентични 16 битови мнгофункционални таймери (Таймер 0 и Таймер 1) за отмерване на време, броене на събития, генериране на импулси, прихващане (**capture**) и сравнение (**compare**). Таймерите имат следните характеристики:

- 16 битови таймери;
- Програмируем входен делител с коефициенти на делене от 1 до 128;
- Генериране на PWM (**Pusle-Width Modulation**);
- Режим на прихващане;
- Режим на сравнение;
- Извод за таймерен вход, прихващане или контрол на броенето. Честотата на външния входен сигнал може да е максимум ¼ от честотата на системния тактов сигнал;
- Извод за таймерен изход;
- Генериране на прекъсване.

25.11.2 Устройство и принцип на работа

Фиг.223 показва блоковата схема на Таймерите.



Фиг. 223 Блокова схема на Таймер x (x = 0 или 1)

Таймерите са 16-битови прави броячи, състоящи се от два 8-битови броячни регистри TxH и TxL, два 8-битови регистри TxRH и TxRL, съдържащи максималната броячна стойност и два 8-битови регистри TxPWMH и TxPWML, които се използват в различните режими на работа на таймерите.

Принципът на работа на таймера зависи от избрания режим. Следващата точка описва всички режими на работа на таймера.

25.11.3 Режими на работа

Еднократен режим (single-shot mode)

В този режим таймерът брои до 16-битовата броячна стойност, съхранена в 8-битовите регистри TxRH и TxRL. Системният тактов сигнал служи за тактов сигнал на таймера. При достигане на броячната стойност таймерът генерира прекъсване, броячните регистри се презареждат с 0001h, таймерът автоматично се забранява и спира да брои. Ако таймерният изход TxOUT е разрешен, нивото на извода се превключва (от ниско във високо или обратно) при презареждането на таймера в 0001h. Бит TPOL в регистър

TxCTL1 определя началното ниво на извода.

Следвайте следните стъпки когато конфигурирате таймера в еднократен режим:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:

   - Забраните таймера;
   - Конфигурирате таймера в еднократен режим;
   - Конфигурирате входния делител;
   - Конфигурирате началното ниво, ако използвате таймерния изход TxOUT.

2. Запишете начална стойност на таймера в броячните регистри TxH и TxL.
3. Запишете максималната броячната стойност в регистрите TxRH и TxRL.
4. Ако е необходимо, разрешете прекъсването от таймера.
5. Ако е необходимо, конфигурирайте съответния извод в режим на таймерен изход.
6. Разрешете таймера.

Периодът на препълване на таймера може да се определи от следното уравнение:

$$T(s) = \frac{(\text{Максимална Броячна Стойност} - \text{Начална Броячна Стойност}) * \text{Делител}}{\text{Системен Такт}(Hz)}$$

Непрекъснат режим (continuous mode)

В този режим таймерът брои до 16-битовата броячна стойност, съхранена в 8-битовите регистри TxRH и TxRL. Системният тактов сигнал служи за тактов сигнал на таймера. При достигане на броячната стойност таймерът генерира прекъсване, броячните регистри се презареждат с 0001h и броенето продължава. Ако таймерният изход TxOUT е разрешен, нивото на извода се превключва (от ниско във високо или обратно) при презареждането на таймера в 0001h. Бит TPOL в регистър TxCTL1 определя началното ниво на извода.

Следвайте следните стъпки когато конфигурирате таймера в непрекъснат режим:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:

    - Забраните таймера;

    - Конфигурирате таймера в непрекъснат режим;

    - Конфигурирате входния делител;

    - Конфигурирате началното ниво, ако използвате таймерния изход TxOUT.

2. Запишете начална стойност на таймера в броячните регистри TxH и TxL (обикновено 0001h).

3. Запишете максималната броячната стойност в регистрите TxRH и TxRL.

4. Ако е необходимо, разрешете прекъсването от таймера.

5. Ако е необходимо, конфигурирайте съответния извод в режим на таймерен изход.

6. Разрешете таймера.

Периодът на препълване на таймера може да се определи от следното уравнение:

$$T(s) = \frac{Максимална\ Броячна\ Стойност \quad * \quad Делител}{Системен\ Такт(Hz)}$$

Ако началната стойност в броячните регистри е различна от 0001h, използвайте уравнението за еднократен режим, за да определите първия период на препълване на таймера.

Броячен режим

В този режим таймерът брои преходите на таймерния вход TxIN. Бит TPOL в регистър TxCTL1 определя активния фронт на входния сигнал, по който става броенето (падащ или нарастващ фронт). В броячен режим входният делител е забранен.

	Честотата на входния сигнал трябва да бъде максимум ¼ от честотата на системния тактов сигнал.

При достигане на броячната стойност, съхранена в регистрите TxRH и TxRL, се генерира прекъсване, таймерните регистри се презареждат с 0001h и броенето продължава. Ако таймерният изход TxOUT е разрешен, нивото на извода се превключва (от ниско във високо или обратно) при презареждането на таймера в 0001h. Бит TPOL в регистър TxCTL1 определя началното ниво на извода.

Следвайте следните стъпки, когато конфигурирате таймера в броячен режим:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:

   - Забраните таймера;

   - Конфигурирате таймера в броячен режим;

   - Изберете активния входен фронт (падащ или нарастващ). Тази настройка задава също и началното ниво на таймерния изход (ако е разрешен).

2. Запишете начална стойност на таймера в броячните регистри TxH и TxL. Тази настройка има ефект само при първото препълване на таймера. След първото препълване на таймера броенето винаги започва от 0001h.

3. Запишете максималната броячна стойност в регистрите TxRH и TxRL.

4. Ако е необходимо, разрешете прекъсването от таймера.

5. Конфигурирайте съответния извод в режим на таймерен вход.

6. Ако е необходимо, конфигурирайте съответния извод в режим на таймерен изход.

7. Разрешете таймера.

Броят на активните фронтове на таймерния вход може да се определи от следното уравнение:

$$N = \text{Текуща Броячна Стойност} - \text{Начална Броячна Стойност}$$

Компараторен броячен режим

В този режим таймерът брои преходите от компараторния изход. Бит TPOL в регистър TxCTL1 определя дали броенето се извършва по нарастващия или падащия фронт на компараторния сигнал. В компараторен броячен режим входният делител е забранен.

	Честотата на компараторния сигнал трябва да бъде максимум ¼ от честотата на системния тактов сигнал.

При достигане на броячната стойност, съхранена в регистрите TxRH и TxRL, се генерира прекъсване, броячните регистри се презареждат с 0001h и броенето продължава. Ако таймерният изход TxOUT е разрешен, нивото на извода се превключва (от ниско във високо или обратно) при презареждането на таймера в 0001h. Бит TPOL в регистър TxCTL1 определя началното ниво на извода.

Следвайте следните стъпки, когато конфигурирате таймера в компараторен броячен режим:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:

   - Забраните таймера;

   - Конфигурирате таймера в компараторен броячен режим;

   - Изберете активния входен фронт на компараторния сигнал (падащ или нарастващ). Тази настройка задава също и началното ниво на таймерния изход (ако е разрешен).

2. Запишете начална стойност на таймера в броячните

регистри TxH и TxL. Тази настройка има ефект само при първото препълване на таймера. След първото препълване на таймера, броенето винаги започва от 0001h.

3. Запишете максималната броячната стойност в регистрите TxRH и TxRL.

4. Ако е необходимо, разрешете прекъсването от таймера.

5. Ако е необходимо, конфигурирайте съответния извод в режим на таймерен изход.

6. Разрешете таймера.

Броят на активните фронтове от компараторния изход може да се определи от следното уравнение:

$$N = Текуща\ Броячна\ Стойност - Начална\ Броячна\ Стойност$$

Едноизходен PWM режим

В този режим таймерът генерира широчинно-модулирани импулси (**PWM – Pulse-Width Modulation**) на таймерния изход TxOUT. Системният тактов сигнал служи за тактов сигнал. Таймерът първо брои до 16-битовата широчинна стойност съхранена в 8-битовите регистри TxPWMH и TxPWML. Когато таймерът достигне тази стойност, таймерният изход се превключва (от ниско във високо ниво или обратно). Таймерът продължава да брои, докато достигне максималната 16-битова броячна стойност в 8-битовите регистри TxRH и TxRL. При достигане на тази стойност таймерът генерира прекъсване, таймерните регистри се презареждат с 0001h, таймерният изход се превключва отново и процесът се повтаря отначало. Бит TPOL в регистър TxCTL1 определя началното ниво на

извода TxOUT.

Ако TPOL = 0, началното ниво на извода е ниско и се превключва във високо, когато TxH:TxL == TxPWMH:TxPWML. Когато TxH:TxL == TxRH:TxRL, изходът отново се превключва в ниско ниво.

Ако TPOL = 1, началното ниво на извода е високо и се превключва в ниско, когато TxH:TxL == TxPWMH:TxPWML. Когато TxH:TxL == TxRH:TxRL, изходът отново се превключва във високо ниво.

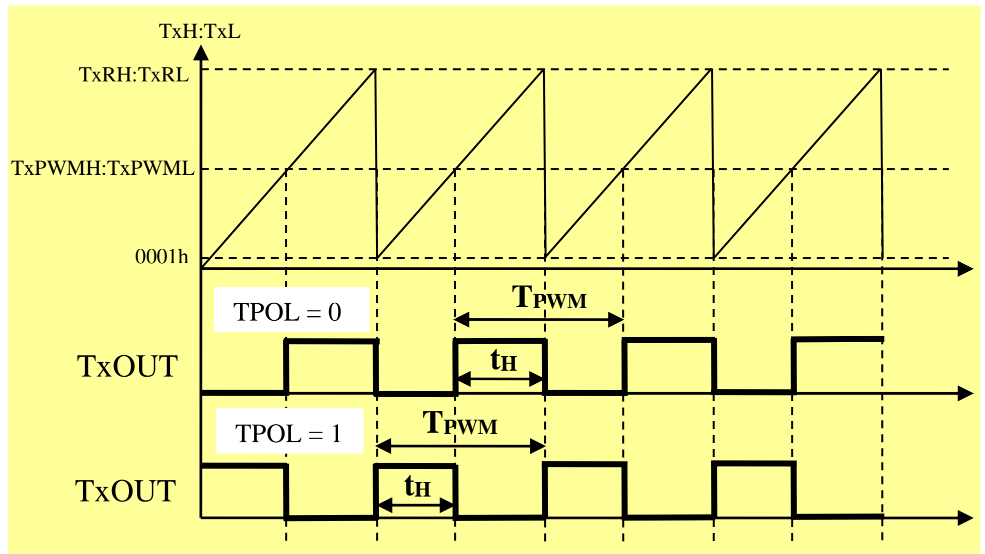
Периодът T_{PWM} на изходния сигнал се определя от максималната броячната стойност в TxRH:TxRL. По-голяма стойност съответства на по-голям период.

Коефициентът на запълване $k = t_H/T_{PWM}$ се определя от стойността в регистрите TxPWMH:TxPWML.

При TPOL = 1 по-голяма стойност съответства на по-голям коефициент на запълване. Когато TxPWMH:TxPWML = 0001h, k = 0. Когато TxPWMH:TxPWML = TxRH:TxRL, k = 1.

При TPOL = 0 по-малка стойност съответства на по-голям коефициент на запълване. Когато TxPWMH:TxPWML = 0001h, k = 1. Когато TxPWMH:TxPWML = TxRH:TxRL, k = 0.

Фиг.224 илюстрира описания процес.



Фиг. 224 Едноизходен PWM режим

Следвайте следните стъпки когато конфигурирате таймера в едноизходен PWM режим:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:

   - Забраните таймера;
   - Конфигурирате таймера в едноизходен PWM режим;
   - Конфигурирате входния делител;
   - Конфигурирате началната ниво на таймерния изход TxOUT.

2. Запишете начална стойност на таймера в броячните регистри TxH и TxL (обикновено 0001h). Тази настройка има ефект само при първото препълване на таймера. След първото препълване на таймера, броенето винаги започва от 0001h.

3. Запишете широчинната стойност в регистрите TxPWMH и TxPWML.

4. Запишете максималната броячна стойност в регистрите TxRH и TxRL. Тази стойност трябва да е по-голяма от широчинната стойност.

5. Ако е необходимо, разрешете прекъсването от таймера.

6. Конфигурирайте съответния извод в режим на таймерен изход.

7. Разрешете таймера.

Периодът на PWM сигнала се определя от следното уравнение:

$$T_{PWM}\,(s) = \frac{Максимална\ Броячна\ Стойност \quad * \quad Делител}{Системен\ Такт\,(Hz)}$$

Ако TPOL = 0, коефициентът на запълване се определя от следното уравнение:

$$к\ (\%) = \left(1 \quad - \quad \frac{Широчинна\ Стойност}{Максимална\ Броячна\ Стойност}\right) \quad * \quad 100$$

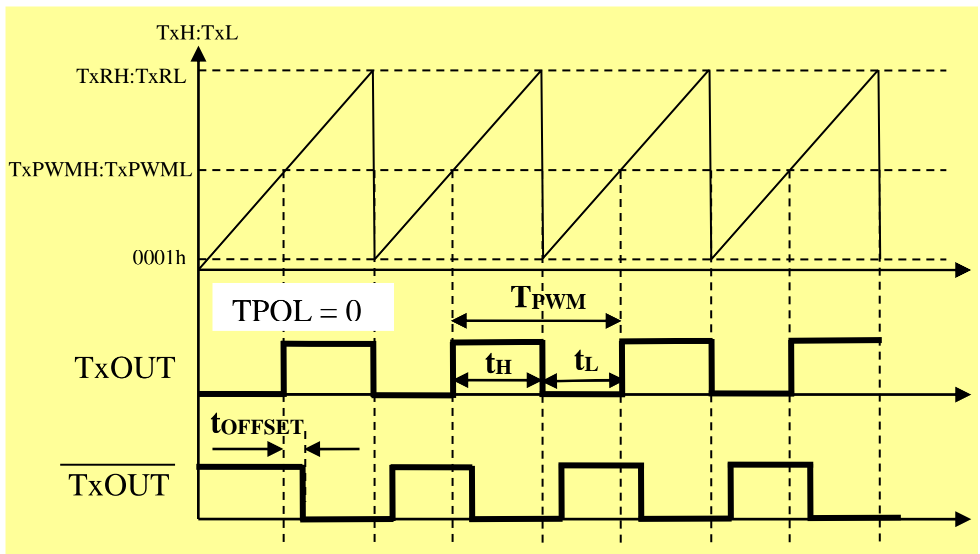
Ако TPOL = 1, коефициентът на запълване се определя от следното уравнение:

$$k\ (\%) = \frac{Широчинна\ Стойност}{Максимална\ Броячна\ Стойност} \quad * \quad 100$$

Двуизходен PWM режим

Този режим е като предходния с тази разлика, че таймерът генерира два широчинно-импулсни сигнала, един нормален сигнал на изход TxOUT и втори инвертиран сигнал на изход $\overline{\text{TxOUT}}$. Между двата сигнала може да бъде добавено и конфигурируемо дефазиране t_{OFFSET} (от 0 до 128 такта на системния тактов сигнал).

Фиг.225 илюстрира описания процес при TPOL = 0.



Фиг. 225 Двуизходен PWM режим

Следвайте следните стъпки когато конфигурирате таймера в двуизходен PWM режим:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:

   - Забраните таймера;

   - Конфигурирате таймера в двуизходен PWM режим;

- Конфигурирате входния делител;

- Конфигурирате началнота ниво на таймерния изход TxOUT.

2. Запишете начална стойност на таймера в броячните регистри TxH и TxL (обикновено 0001h). Тази настройка има ефект само при първото препълване на таймера. След първото препълване на таймера, броенето винаги започва от 0001h.

3. Запишете широчинната стойност в регистрите TxPWMH и TxPWML.

4. Конфигурирайте дефазирането между двата таймерни изхода. Дефазирането трябва да бъде по-малко от t_H и t_L.

5. Запишете максималната броячната стойност в регистрите TxRH и TxRL. Тази стойност трябва да е по-голяма от широчинната стойност.

6. Ако е необходимо, разрешете прекъсването от таймера.

7. Конфигурирайте съответния извод в режим на таймерен изход.

8. Разрешете таймера.

Режим на прихващане (capture mode)

В този режим при появата на активен фронт (нарастващ или падащ) на таймерния вход TxIN, текущата броячна стойност в регистрите TxH:TxL се копира (прихваща) в регистрите TxPWMH:TxPWML. Системният тактов сигнал служи за тактов сигнал на таймера. Бит TPOL в регистър TxCTL1

определя по кой фронт (нарастващ или падащ) на таймерния вход да става прихващането на текущата броячна стойност на таймера. При появата на активен фронт на таймерния вход се генерира прекъсване и таймерът продължава да брои. Флаг INTCAP в регистър TxCTL1 се вдига, за да укаже, че прекъсването се дължи на прихващащо събитие, а не от препълване на таймера.

Таймерът продължава да брои, докато стигне броячната стойност в регистрите TxRH:TxRL. При достигане на тази стойност, таймерът генерира прекъсване, броячните регистри се презареждат с 0001h[1] и броенето се възобновява. Флаг INTCAP в регистър TxCTL1 се нулира, за да укаже, че прекъсването е поради препълване на таймера, а не поради прихващащо събитие.

Забележка[1]: В оригиналната документация е описано, че при тази ситуация таймерът продължава да брои, но по-вероятно е пропуснато да се каже, че преди това броячните регистри TxH:TxL се презареждат с 0001h.

Следвайте следните стъпки когато конфигурирате таймера в режим на прихващане:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:

   - Забраните таймера;

   - Конфигурирате таймера в режим на прихващане;

   - Конфигурирате входния делител;

   - Изберете активния фронт на таймерния вход.

2. Запишете начална стойност на таймера в броячните регистри TxH и TxL (обикновено 0001h).

3. Запишете максималната броячна стойност в регистрите TxRH и TxRL.

4. Запишете 0000h в регистрите TxPWMH:TxPWML. Нулирането на тези регистри позволява на софтуера да определи дали прекъсването е генерирано от прихващащо събитие или от препълване на таймера. Ако стойността в TxPWMH:TxPWML е все още нула, значи прекъсването е от препълване.

5. Ако е необходимо, разрешете прекъсването от таймера. По подразбиране прекъсване се генерира при прихващащо събитие и при препълване на таймера. Ако е необходимо, конфигурирайте таймера да генерира прекъсване само при прихващане или само при препълване чрез бит TICONFIG в регистър TxCTL1.

6. Конфигурирайте съответния извод в режим на таймерен вход.

7. Разрешете таймера.

Времето от стартирането на таймера до появата прихващащо събитие може да се определи от уравнението:

$$T_{CAPTURE}\ (s) = \frac{(Прихваната\ Броячна\ Стойност\ -\ Начална\ Броячна\ Стойност) * Делител}{СистемнаТактоваЧестота\ (Hz)}$$

Режим на прихващане с рестартиране

В този режим при появата на активен фронт (нарастващ или падащ) на таймерния вход TxIN, текущата броячна стойност в регистрите TxH:TxL се копира (прихваща) в регистрите

TxPWMH:TxPWML. Системният тактов сигнал служи за тактов сигнал на таймера. Бит TPOL в регистър TxCTL1 определя по кой фронт (нарастващ или падащ) на таймерния вход да става прихващането на текущата броячна стойност на таймера.

При появата на активен фронт на таймерня вход, се генерира прекъсване и броячните регистри TxH:TxL се презареждат с 0001h и броенето се възобновява. Флаг INTCAP в регистър TxCTL1 се вдига, за да укаже, че прекъсването се дължи на прихващащо събитие.

Ако не възникне прихващащо събитие, таймерът продължава да брои, докато стигне броячната стойност в регистрите TxRH:TxRL. При достигане на тази стойност, таймерът генерира прекъсване и броячните регистри се презареждат с 0001h и броенето се възобновява. Флаг INTCAP в регистър TxCTL1 се нулира, за да укаже, че прекъсването е поради препълване на таймера.

Следвайте следните стъпки, когато конфигурирате таймера в режим на прихващане с рестартиране:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:

   - Забраните таймера;

   - Конфигурирате таймера в режим на прихващане с рестартиране;

   - Конфигурирате входния делител;

   - Изберете активния фронт на таймерния вход.

2. Запишете начална стойност на таймера в броячните

регистри TxH и TxL (обикновено 0001h).

3. Запишете максималната броячната стойност в регистрите TxRH и TxRL.

4. Запишете 0000h в регистрите TxPWMH:TxPWML. Нулирането на тези регистри позволява на софтуера да определи дали прекъсването е генерирано от прихващащо събитие или от препълване на таймера. Ако стойността в TxPWMH:TxPWML е все още нула, значи прекъсването е от препълване.

5. Ако е необходимо, разрешете прекъсването от таймера. По подразбиране прекъсване се генерира при прихващащо събитие и при препълване на таймера. Ако е необходимо, конфигурирайте таймера да генерира прекъсване само при прихващане или само при препълване чрез бит TICONFIG в регистър TxCTL1.

6. Конфигурирайте съответния извод в режим на таймерен вход.

7. Разрешете таймера.

Времето от стартирането на таймера до появата прихващащо събитие може да се определи от уравнението:

$$T_{CAPTURE}\,(s) = \frac{(Прихваната\ Броячна\ Стойност \quad - \quad Начална\ Броячна\ Стойност) * Делител}{Системна Тактова Честота\ (Hz)}$$

Режим на сравнение (compare mode)

В този режим таймерът брои до 16-битовата стойност съхранена в регистрите TxRH:TxRL (извършва се сравнение на стойността в броячните регистри TxH:TxL с броячната стойност в регистрите TxRH:TxRL). Системният тактов

сигнал служи за тактов сигнал на таймера. При достигане до тази стойност, таймерът генерира прекъсване и броенето продължава (броячните регистри TxH:TxL не се презареждат с 0001h). Допълнително, нивото на таймерния изход TxOUT (ако е разрешен) се превключва (от ниско във високо или обратно). Бит TPOL в регистър TxCTL1 определя началното ниво на таймерния изход.

Ако таймерът достигне стойност FFFFh, броячните регистри се презареждат с 0000h и броенето продължава.

Следвайте следните стъпки когато конфигурирате таймера в режим на сравнение:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:

   - Забраните таймера;

   - Конфигурирате таймера в режим на сравнение;

   - Конфигурирате входния делител;

   - Изберете начално ниво на таймерния изход.

2. Запишете начална стойност на таймера в броячните регистри TxH и TxL.

3. Запишете сравняващата броячната стойност в регистрите TxRH и TxRL.

4. Разрешете прекъсването от таймера.

5. Ако е необходимо, конфигурирайте съответния извод в режим на таймерен изход.

6. Разрешете таймера.

Времето от стартирането на таймера до възникване на сравнението може да се определи от уравнението:

$$T_{COMPARE}\,(s) = \frac{(Сравняваща\ Броячна\ Стойност\ -\ Начална\ Броячна\ Стойност) * Делител}{СистемнаТактоваЧестота\,(Hz)}$$

Режим на контролирано броене (gated mode)

В този режим активен сигнал (ниско или високо ниво) на таймерния вход TxIN служи като разрешаващ сигнал за таймера, т.е. . таймерът брои само когато на таймерния вход има валиден разрешаващ сигнал. Бит TPOL в регистър TxCTL1 определя активното ниво на разрешаващия сигнал. При деактивиране на разрешаващия сигнал се генерира прекъсване. Системният тактов сигнал служи за тактов сигнал на таймера

Таймерът брои до 16-битовата броячна стойност в регистрите TxRH:TxRL. При достигане на тази стойност таймерът генерира прекъсване, броячните регистри TxH:TxL се презареждат с 0001h и броенето се възобновява (ако разрешаващият сигнал е все още активен). Допълнително, ако таймерният изход е разрешен, нивото на извода се превключва (от високо в ниско или обратно) при презареждане на таймера.

За да определите причината за прекъсването (дали идва от препълване на таймера или деактивиране на разрешаващия сигнал), прочетете нивото на таймерния вход TxIN и го сравнете със стойността на бит TPOL.

Следвайте следните стъпки, когато конфигурирате таймера в режим на контролирано броене:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:

   - Забраните таймера;

   - Конфигурирате таймера в контролиран режим на броене;

   - Конфигурирате входния делител.

2. Запишете начална стойност на таймера в броячните регистри TxH и TxL. Тази настройка има ефект само до първото препълване на таймера. След първото препълване на таймера, броенето винаги започва от 0001h.

3. Запишете максималната броячната стойност в регистрите TxRH и TxRL.

4. Разрешете прекъсването от таймера. По подразбиране прекъсване се генерира при деактивиране на разрешаващия сигнал и при препълване на таймера. Вие може да конфигурирате прекъсването да се генерира само при деактивиране на разрешаващия сигнал или само при препълване на таймера, чрез бит TICONFIG в регистър TxCTL1.

5. Конфигурирайте съответния извод като таймерен вход.

6. Ако е необходимо, конфигурирайте съответния извод в режим на таймерен изход.

7. Разрешете таймера.

8. Активирайте разрешаващия сигнал, за да стартирате броенето.

Режим на прихващане/сравнение

В този режим таймерът започва да брои по първия активен фронт (нарастващ или падащ) на таймерния вход TxIN. Бит TPOL в регистър TxCTL1 определя активния фронт. Системният тактов сигнал служи за тактов сигнал на таймера.

При всеки следващ активен фрон на входа текущатаброячната стойност на таймера се копира (прихваща) в регистрите TxPWMH:TxPWML, генерира се прекъсване, броячните регистри TxH:TxL се презареждат в 0001h и броенето продължава. Флаг INTCAP в регистър TxCTL1 се вдига, за да укаже, че прекъсването се дължи на прихващащо събитие.

Ако не възникне прихващащо събитие (няма активен фронт на таймерния вход), таймерът брои до 16-битовата сравняваща стойност в регистрите TxRH:TxRL. При достигане на тази стойност таймерът генерира прекъсване, броячните регистри TxH:TxL се презареждат с 0001h и броенето се възобновява. Флаг INTCAP в регистър TxCTL1 се нулира, за да укаже, че прекъсването не се дължи на прихващащо събитие.

Следвайте следните стъпки когато конфигурирате таймера в режим на прихващане/сравнение:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:

   - Забраните таймера;

   - Конфигурирате таймера в режим на прихващане/сравнение;

- Конфигурирате входния делител;

- Изберете активен фронт на таймерния вход.

2. Запишете начална стойност на таймера в броячните регистри TxH и TxL (обикновено 0001h).

3. Запишете сравняващата броячната стойност в регистрите TxRH и TxRL.

4. Разрешете прекъсването от таймера. По подразбиране прекъсване се генерира при прихващащо събитие и препълване на таймера. Може да конфигурирате прекъсването да се генерира само при прихващащо събитие или само при препълване на таймера, чрез бит TICONFIG в регистър TxCTL1.

5. Конфигурирайте съответния извод в режим на таймерен вход.

6. Разрешете таймера.

7. Броенето започва по първия активен фронт на таймерния вход без да се генерира прекъсване.

Времето от стартирането на таймера до възникване на прихващане може да се определи от уравнението:

$$T_{COMPARE}\,(s) = \frac{(Прихваната\ Броячна\ Стойност\ -\ Начална\ Броячна\ Стойност) * Делител}{СистемнаТактоваЧестота\ (Hz)}$$

25.11.4 Четене на текущата броячна стойност на таймера

Текущата броячна стойност може да бъде прочетена, докато

таймерът брои. Когато таймерът е разрешен, четенето на регистър TxH води и до копиране на TxL във временен регистър. Последващо четене на TxL, връща стойността, съхранена във временния регистър.

Тази операция позволява точно изчитане на 16 битовата стойност на таймера, докато е разрешен. Когато таймерът е забранен, четене на TxL връща действителната стойност в този регистър.

25.11.5 Описание на управляващите регистри

Регистри TxH (Timer High Byte) и TxL (Timer Low Byte)

Тези регистри съдържат текущата броячна стойност на таймера. Когато таймерът е разрешен, четене на TxH води и до копиране на TxL във временен регистър. Последващо четене на TxL връща стойността, съхранена във временния регистър. Когато таймерът е забранен, четене на TxL, връща действителната стойност в този регистър.

Бит	7	6	5	4	3	2	1	0
Име	TH							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F00h, F08h							
Бит	**Описание**							
7-0	Старши байт на 16-битовия брояч							

Табл. 93 TxH

Бит	7	6	5	4	3	2	1	0
Име	TL							

Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F01h, F09h							
Бит	**Описание**							
7-0	Младши байт на 16-битовия брояч							

Табл. 94 TxL

Регистри TxRH (Timer Reload High Byte) и TxRL (Timer Reload Low Byte)

Тези регистри съдържат максималната 16-битова броячна стойност до която брои таймера. Запис в TxRH води до запис във временен регистър. Запис в TxRL води и до копиране на временния регистър в TxRH. Тази операция позволява едновременно обновяване на 16-битовата броячната стойност в тези регистри. В режим на сравнение тези регистри съдържат 16-битовата сравняваща стойност.

Бит	7	6	5	4	3	2	1	0
Име	TRH							
Ресет	1	1	1	1	1	1	1	1
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F02h, F0Ah							
Бит	**Описание**							
7-0	Старши байт на 16 битовата броячна/сравняваща стойност							

Табл. 95 TxRH

Бит	7	6	5	4	3	2	1	0
Име	TRL							
Ресет	1	1	1	1	1	1	1	1
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F03h, F0Bh							

Бит	Описание
7-0	Младши байт на 16 битовата броячна/сравняваща стойност

Табл. 96 TxRL

Регистри TxPWMH (PWM High Byte) и TxPWML (PWM Low Byte)

Тези регистри съдържат 16-битовата стойност, определяща продължителността на високото ниво на импулсите в едноизходен и двуизходен PWM режим на работа. В режим на прихващане и прихващане/сравнение тези регистри съдържат прихванатата броячна стойност при възникване на прихващащо събитие.

Бит	7	6	5	4	3	2	1	0
Име	TPWMH							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F04h, F0Ch							

Бит	Описание
7-0	Старши байт на 16-битовата широчинна стойност в PWM режими или старши байт на прихванатата броячна стойност в режими на прихващане.

Табл. 97 TxPWMH

Бит	7	6	5	4	3	2	1	0
Име	TPWML							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F05h, F0Dh							

Бит	Описание
7-0	Младши байт на 16-битовата широчинна стойност в PWM режими или младши байт на прихванатата броячна стойност в режими на прихващане.

Табл. 98 TxPWML

Регистър TxCTL0 (Timer 0 Control)

Бит	7	6	5	4	3	2	1	0
Име	TMODEHI	TICONFIG			PWMD			INCAP
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F06h, F0Eh							

Бит	Описание
7	**Работен режим на таймера** Този бит заедно с битове TMODE[2:0] в регистър TxCTL1 определят работния режим на таймера. **TMODEHI:TMODE[2:0]** – **Режим** 0000 – Еднократен 0001 – Непрекъснат 0010 – Броячен 0011 – Едноизходен PWM 0100 – Прихващане 0101 – Сравнение 0110 – Контролирано броене 0111 – Прихващане/Сравнение 1000 – Двуизходен PWM режим 1001 – Прихващане с рестартиране 1010 – Компараторен
6-5	**Конфигуриране на прекъсването от таймера** 0x – Прекъсване при презареждане (препълване), сравнение и прихващане 10 – Прекъсване при прихващане/деактивиране 11 – Прекъсване при презареждане (препълване)/сравнение

4	Резервиран
3-1	**Конфигуриране на t_{OFFSET}** 000 – 0 цикъла 001 – 2 цикъла 010 – 4 цикъла 011 – 8 цикъла 100 – 16 цикъла 101 – 32 цикъла 110 – 64 цикъла 111 – 128 цикъла
0	**Индикатор на източник на прекъсване** 0 – Прекъсването не се дължи на прихващащо събитие 1 – Прекъсването се дължи на прихващащо събитие

Табл. 99 TxCTL0

Регистър TxCTL1 (Timer 1 Control)

Бит	7	6	5	4	3	2	1	0
Име	TEN	TPOL	PRES			TMODE		
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F07h, F0Fh							

Бит	**Описание**
7	**Разрешаване на таймера** 0 – Таймерът е забранен 1 – Таймерът е разрешен
6	**Полярност на входа/изхода** **<u>Еднократен режим</u>** Когато таймерът е забранен, таймерният изход извежда стойността на този бит. Когато таймерът е разрешен, таймерният изход се превключва при презареждане (препълване) на таймера. **<u>Непрекъснат режим</u>**

Когато таймерът е забранен, таймерният изход извежда стойността на този бит. Когато таймерът е разрешен, таймерният изход се превключва при всяко презареждане (препълване) на таймера.

Броячен режим

Когато таймерът е забранен, таймерният изход извежда стойността на този бит. Когато таймерът е разрешен, таймерният изход се превключва при всяко презареждане (препълване) на таймера.
0 – Броене по нарастващ фронт
1 – Броене по падащ фронт

Компараторен броячен режим

Когато таймерът е забранен, таймерният изход извежда стойността на този бит. Когато таймерът е разрешен, таймерният изход се превключва при всяко презареждане (препълване) на таймера.

Когато таймерният изход TxOUT е разрешен, той винаги извежда стойността съхранена в бит TPOL, независимо дали таймерът е разрешен или забранен. Промяна на TPOL, докато таймерът е разрешен и брои, не променя веднага стойността на TxOUT.

Еднозиходен PWM режим

0 – Извод TxOUT е в ниско ниво, когато таймерът е забранен. TxOUT се превключва във високо ниво, когато таймерът е разрешен и TxH:TxL == TxPWMH:TxPWML. Когато TxH:TxL == TxRH:TxRL, TxOUT отново се превключва в ниско ниво.
1 – Извод TxOUT е във високо ниво, когато таймерът е забранен. TxOUT се превключва в ниско ниво, когато таймерът е разрешен и TxH:TxL == TxPWMH:TxPWML. Когато TxH:TxL == TxPH:TxPL, TxOUT отново се превключва във високо ниво.

Двуизходен PWM режим

0 – Извод TxOUT е в ниско ниво и извод $\overline{\text{TxOUT}}$ е във високо ниво, когато таймерът е забранен. TxOUT се превключва във високо ниво, когато таймерът е разрешен и TxH:TxL == TxPWMH:TxPWML. Когато TxH:TxL == TxRH:TxRL, TxOUT отново се превключва в ниско ниво.
1 – Извод TxOUT е във високо ниво и извод $\overline{\text{TxOUT}}$ е в ниско ниво, когато таймерът е забранен. TxOUT се превключва в ниско ниво, когато таймерът е разрешен и TxH:TxL == TxPWMH:TxPWML. Когато TxH:TxL == TxRH:TxRL, TxOUT отново се превключва във високо ниво.

Режим на прихващане

0 – Прихващане по нарастващ входен фронт
1 – Прихващане по падащ входен фронт

Режим на прихващане с рестартиране

0 – Прихващане по нарастващ входен фронт
1 – Прихващане по падащ входен фронт

Режим на сравнение

Когато таймерът е забранен, таймерният изход извежда стойността на този бит. Когато таймерът е разрешен и TxH:TxL == TxRH:TxRL, таймерният изход се превключва.

Режим на контролирано броене

0 – Таймерът брои, когато нивото на входа TxIN е високо и прекъсване се генерира по падащ фронт на входния сигнал.
1 – Таймерът брои, когато нивото на входа TxIN е ниско и прекъсване се генерира по нарастващ фронт на входния сигнал.

Режим на прихващане/сравнение

0 – Броенето се стартира по първия нарастващ фронт на входа

<table>
<tr><td></td><td>TxIN. При всеки следващ нарастващ фронт броячната стойност се прихваща.
1 – Броенето се стартира по първия падащ фронт на входа TxIN. При всеки следващ падащ фронт броячната стойност се прихваща.</td></tr>
<tr><td>5-3</td><td>Коефициент на деление на Fsys
000 – 1
001 – 2
010 – 4
011 – 8
100 – 16
101 – 32
110 – 64
111 – 128</td></tr>
<tr><td>2-0</td><td>Работен режим на таймера
Тези битове заедно с бит TMODEHI в регистър TxCTL0 определят работния режим на таймера.

<table>
<tr><th>TMODEHI:TMODE[2:0]</th><th>Режим</th></tr>
<tr><td>0000</td><td>Еднократен</td></tr>
<tr><td>0001</td><td>Непрекъснат</td></tr>
<tr><td>0010</td><td>Броячен</td></tr>
<tr><td>0011</td><td>Едноизходен PWM</td></tr>
<tr><td>0100</td><td>Прихващане</td></tr>
<tr><td>0101</td><td>Сравнение</td></tr>
<tr><td>0110</td><td>Контролирано броене</td></tr>
<tr><td>0111</td><td>Прихващане/Сравнение</td></tr>
<tr><td>1000</td><td>Двуизходен PWM режим</td></tr>
<tr><td>1001</td><td>Прихващане с рестартиране</td></tr>
<tr><td>1010</td><td>Компараторен</td></tr>
</table></td></tr>
</table>

Табл. 100 TxCTL1

25.12 Компаратор

25.12.1 Общи сведения

Компараторът в Z8Encore!F083A серията микроконтролери има следните характеристики:

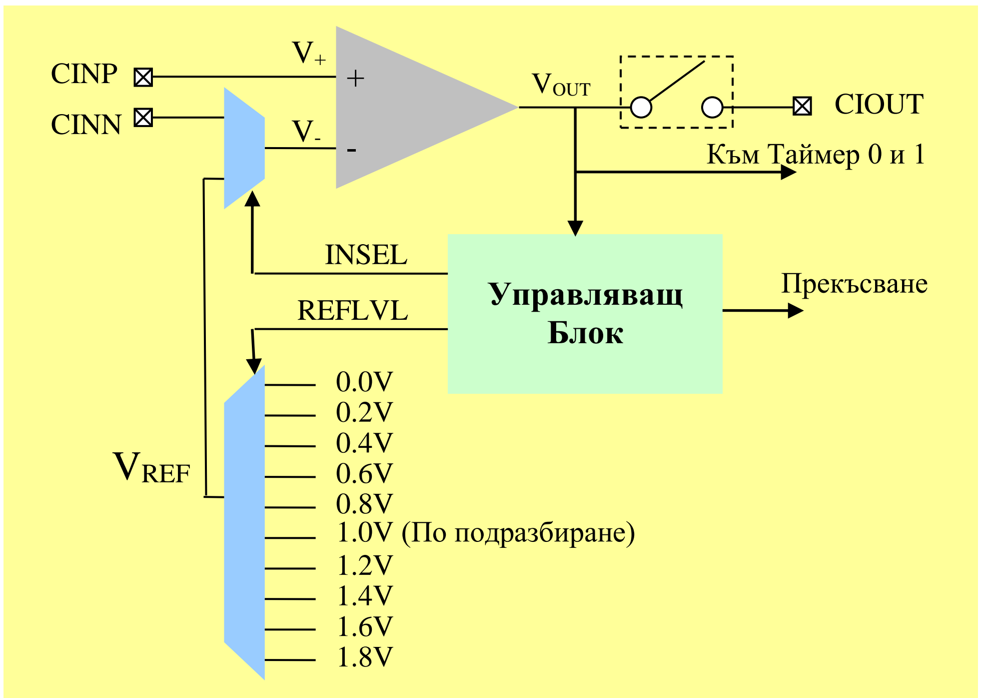
- Положителен вход свързан към извод CINP;
- Отрицателен вход CINN свързан към извод CINN или към вътрешно конфигурируемо опорно напрежение V_{REF};
- Изходът на компаратора може да се изведе на извод COUT;
- Време за отговор 100ns;
- Входен хистерезис 8mV.

25.12.2 Устройство и принцип на работа

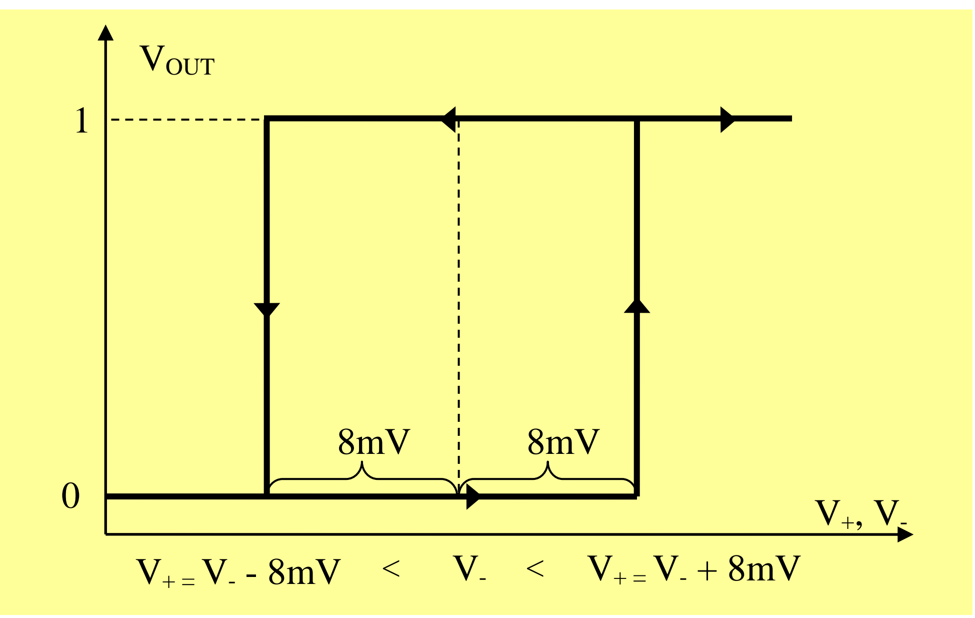
Фиг.226 показва блоковата схема на компаратора, а Фиг.227 описва принципа на работа.

Когато напрежението V_+ е по-голямо от V_-, V_{OUT} се превключва във високо ниво и се генерира прекъсване. Когато напрежението V_+ е по-малко от V_-, V_{OUT} се превключва в ниско ниво и се генерира прекъсване.

За намаляване на консумацията, захранването на компаратора може да бъде изключено. За повече детайли вижте Регистър PWCTL0.



Фиг. 226 Блокова схема на компаратора



Фиг. 227 V_{OUT} като функция на V_+ и V_-

	В резултат на времето за отговор на компаратора, Zilog не препоръчва разрешаване на компаратор без първо забраняване на прекъсванията и изчакване на изхода да се установи. Това закъснение предпазва от фалшиви прекъсвания след разрешаване на компаратора.

Следващият пример показва как безопасно да разрешите компаратора.

```
DI      ; Забраняване на прекъсванията
LD      CMP0,R0 ;Зареждане на нова конфигурация
NOP
NOP ; Изчакване на установяване на изхода
CLR   IRQ0   ;Изчистване на всички фалшиви чакащи
             ;прекъсвания
EI      ; Разрешаване на прекъсванията
```

25.12.3 Описание на управляващите регистри

Регистър CMP0 (Comparator Control Register)

Бит	7	6	5	4	3	2	1	0
Име		INNSEL	REFLVL					
Ресет	0	0	0	1	0	1	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F90h							

Бит	**Описание**
7	Този бит е резервиран и трябва да се поддържа в 0.
6	**Избор на V-** 0 – CINN е източник на опорно напрежение 1 – V_{REF} е източник на опорно напрежение
5-2	**Избор на вътрешно опорно напрежение V_{REF}** 0000 – 0.0V 0001 – 0.2V 0010 – 0.4V

	0011 – 0.6V 0100 – 0.8V 0101 – 1.0V (По подразбиране) 0110 – 1.2V 0111 – 1.4V 1000 – 1.6V 1001 – 1.8V 1010 – 1111 (Резервирани)
1-0	Тези битове са резервирани и трябва да се поддържат в 0.

Табл. 101 CMP0

25.13 Аналого-цифров преобразувател

25.13.1 Общи сведения

Аналого-цифровият преобразувател в Z8 Encore!F083A серията микроконтролери има следните характеристики:

- 8 аналогови входа;
- Време за преобразуване под 2.8μs (тактовият сигнал за АЦП трябва да е под 10MHz);
- Програмируемо време за семплиране на входното напрежение;
- Генериране на прекъсване при завършване на преобразуването;
- Генератор на вътрешно опорно напрежение;
- Извод V_{REF} за подаване на външно опорно напрежение.

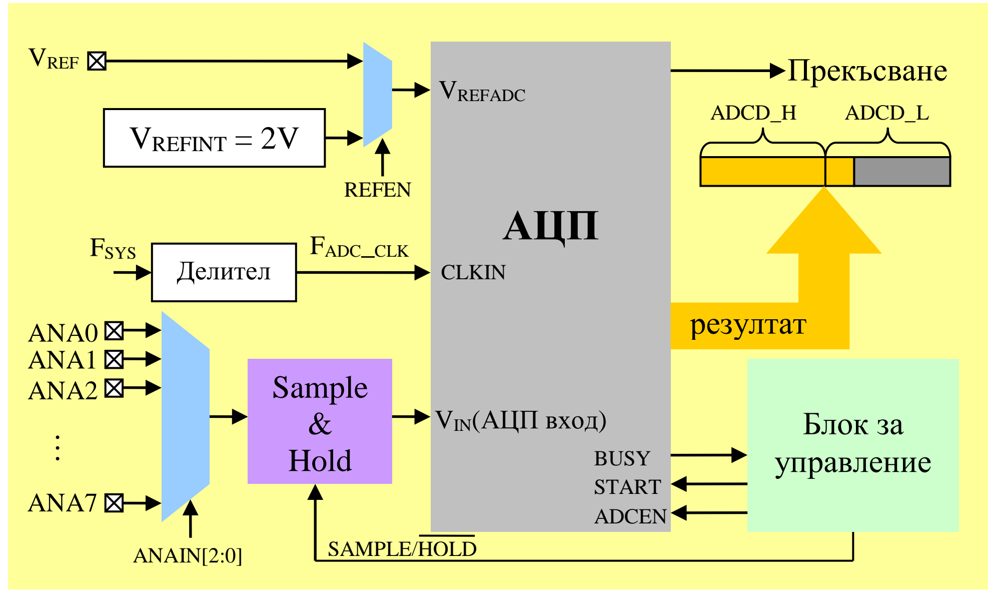
25.13.2 Устройство и принцип на работа

Фиг.228 показва блоковата схема на АЦП. АЦП преобразува избрания аналогов вход ANAx (x = 0 - 7) в 10-битов цифров код. Резултатът от преобразуването може да се изчисли от следното уравнение:

$$АЦПрезултат = \frac{V_{IN}}{V_{REFADC}} \times 1024$$

където,

V_{IN} – Напрежение на измервания аналогов вход
V_{REFADC} – Опорно напрежение на АЦП



Фиг. 228 Блокова схема на АЦП

Опорното напрежение V_{REFADC} може да се получи от вътрешен източник или от извод V_{REF}:

V_{REFADC} = { 2V от вътрешния източник V_{REFINT} ; V_{DD} от извод V_{REF} }

Максималното входно напрежение, което се преобразува от АЦП е:

V_{INTMAX} = { **V_{REFINT}**, ако се използва вътрешното опорно напрежение ; **$0.9V_{DD}$**, ако се използва външно опорно напрежение (от извод V_{REF}, свързан към V_{DD}) }

Преобразуването се стартира с бит START в регистър ADCCTL0. Този бит служи и за индикатор дали текущото преобразуване е в прогрес или е завършило (прочитане на 1 в този бит указва, че в момента се извършва преобразуване).

Стартиране на ново преобразуване води до прекратяване на текущото и започване на ново.

При завършване на преобразуването, АЦП генерира прекъсване. Чакащи заявки за прекъсване, когато АЦП е забранено, не се нулират автоматично.

Времето за преобразуване се определя от следното уравнение:

$$T_{CONV} = T_S + T_H + 13*T_{ADC_CLK}$$

където

T_S – Време за семплиране (**sample time**). Това време може да се конфигурира чрез регистър ADCST и трябва да е минимум 1μs.

T_H – Време за задържане (**hold time**). Това време може да се конфигурира чрез регистър ADCSST и трябва да е минимум 0.5μs.

T_{ADC_CLK} – Период на тактовия сигнал за АЦП. Максималната честота на тактовия сигнал F_{ADC_CLK} е 10MHz.

25.13.3 Описание на управляващите регистри

Регистър ADCCTL0 (ADC Control 0)

Бит	7	6	5	4	3	2	1	0
Име	START		REFEN	ADCEN		ANAIN		
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F70h							

Бит	Описание
7	**START/BUSY** 0 – Прочитане на 0 означава, че АЦП е свободен. Запис на 0 няма ефект. 1 – Прочитане на 1 означава, че АЦП извършва преобразуване в момента. Запис на 1 стартира преобразуване.
6	Този бит е резервиран и трябва да се поддържа в 0.
5	**Избор на опорно напрежение** 0 – Извод V_{REF} е източник на опорно напрежение. 1 – V_{REFIN} е източник на опорно напрежение. Стойността на V_{REFIN} се появява на извод V_{REF}.
4	**Разрешаване на АЦП** 0 – АЦП е забранен. 1 – АЦП е разрешен.
3	Този бит е резервиран и трябва да се поддържа в 0.
2-0	**Избор на аналогов канал** 000 – ANA0 е избран да се преобразува. 001 – ANA1 е избран да се преобразува. 010 – ANA2 е избран да се преобразува. 011 – ANA3 е избран да се преобразува. 100 – ANA4 е избран да се преобразува. 101 – ANA5 е избран да се преобразува. 110 – ANA6 е избран да се преобразува. 111 – ANA7 е избран да се преобразува.

Табл. 102 ADCCTL0

Регистър ADCD_H (ADC Data High Byte)

Бит	7	6	5	4	3	2	1	0
Име	ADCDH							
Ресет	x							
Ч/З	Ч	Ч	Ч	Ч	Ч	Ч	Ч	Ч
Адрес	F72h							

Бит	Описание
7-0	00h-FFh: Старши байт на резултата от преобразуването.

Табл. 103 ADCD_H

Регистър ADCD_L (ADC Data Low Bits)

Бит	7	6	5	4	3	2	1	0
Име	ADCDL							
Ресет	x		x					
Ч/З	Ч	Ч	Ч	Ч	Ч	Ч	Ч	Ч
Адрес	F73h							

Бит	Описание
7-6	00h-11h: Младши битове на резултата от преобразуването.
5-0	Тези битове са резервирани и трябва да се поддържат в 0.

Табл. 104 ADCD_L

Регистър ADCSST (ADC Settling Time)

Бит	7	6	5	4	3	2	1	0
Име					SST			
Ресет	0				1	1	1	1
Ч/З	Ч				Ч/З			
Адрес	F74h							

Бит	Описание
7-4	Тези битове са резервирани и трябва да се поддържат в 0.
3-0	**Избор на време T_H** Измерва се в ситемни тактови периода. Трябва да бъде минимум 0.5μs.

Табл. 105 ADCDSST

Регистър ADCST (ADC Sample Time)

Бит	7	6	5	4	3	2	1	0
Име			ST					
Ресет	0		1	1	1	1	1	1
Ч/З	Ч/З		Ч/З					
Адрес	F75h							

Бит	Описание
7-6	Тези битове са резервирани и трябва да се поддържат в 0.
5-0	**Избор на време T_S** Измерва се в ситемни тактови периода. Трябва да бъде минимум 1μs.

Табл. 106 ADCST

Регистър ADCCP (ADC Clock Prescale)

Бит	7	6	5	4	3	2	1	0
Име						DIV8	DIV4	DIV2
Ресет	0					0	0	0
Ч/З	Ч/З							
Адрес	F76h							

Бит	Описание
7-3	Тези битове са резервирани и трябва да се поддържат в 0.
2	**F_{SYS}/8 (Най-висок приоритет)** 0 – Системният такт не се разделя. 1 – Системният такт се разделя на 8.
1	**F_{SYS}/4** 0 – Системният такт не се разделя. 1 – Системният такт се разделя на 4.
0	**F_{SYS}/2 (Най-нисък приоритет)** 0 – Системният такт не се разделя. 1 – Системният такт се разделя на 2.

Табл. 107 ADCSP

25.14 Флаш-памет

25.14.1 Общи сведения

Микроконтролерите от серията Z8Encore!F083A имат 4kB (4096 байта) или 8kB (8192 байта) флаш-памет за четене/запис/изтриване. Паметта може да се програмира и изтрива, докато микроконтолера е на платката, с помощта на потребителски код или с дебъгер/програматор.

Флаш-паметта е разделена на страници по 512 байта. Флаш-страницата е минималният блок памет, който може да бъде изтрит. Всяка страница е разделена на 8 реда всеки с по 64 байта.

За защита на програмата/данните, флаш-паметта е разделена също и на сектори. В **ZF043A** секторът заема една флаш-страницата, а в **ZF083A** секторът заема две флаш-страници.

Първите два байта на флаш-паметта се заемат от флаш-конфигурационни битове (вижте 25.15 Флаш-конфигурационни битове).

Табл.108 описва конфигурацията на Програмната Памет за всяко устройство от серията Z8Encore! F083A.

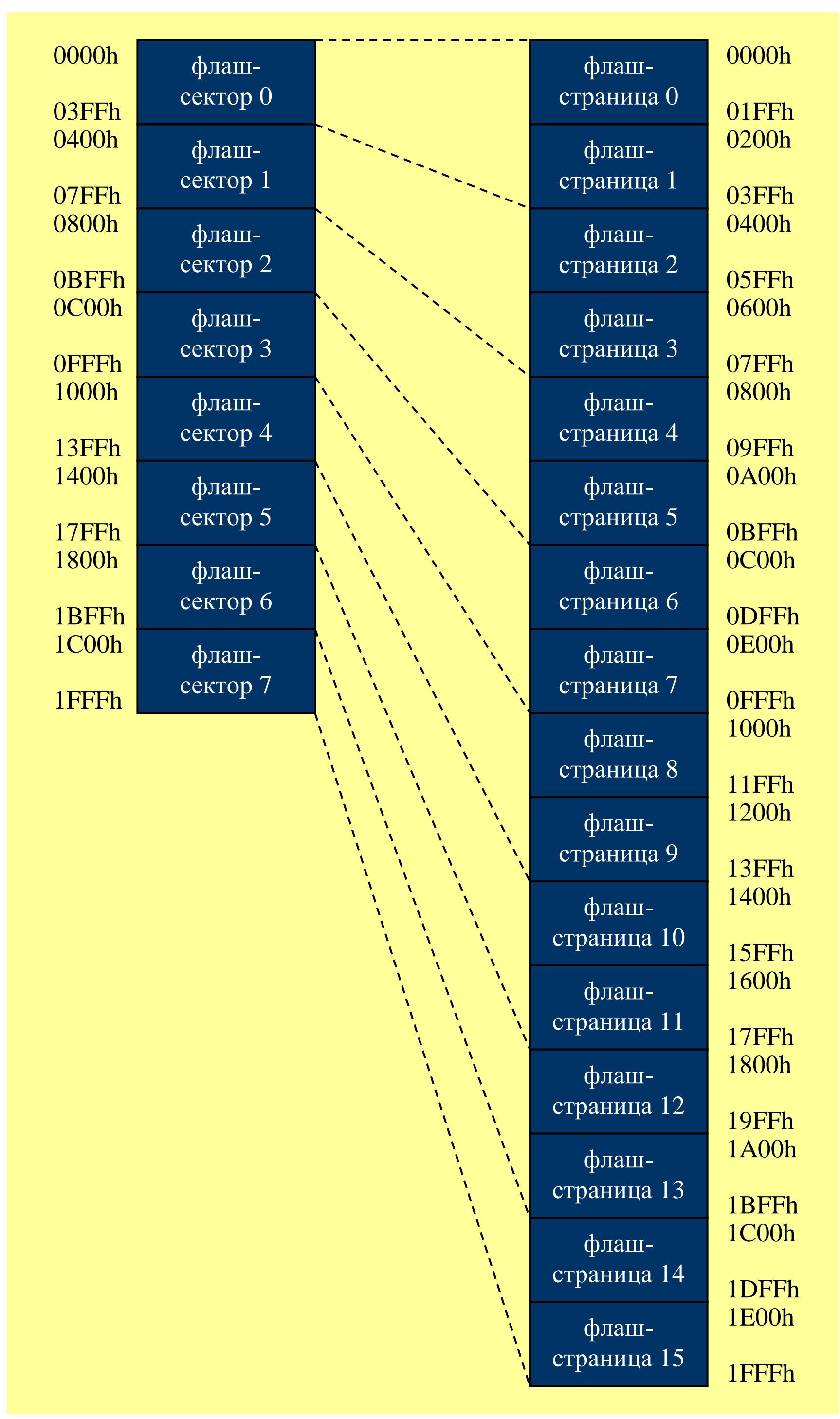
	Флаш-памет kB (байтове)	**Флаш-страници**	**адреси**	**Флаш-сектор (байтове)**
ZF083A	8 (8196)	16	0000h-1FFFh	1024
ZF043A	4 (4096)	8	0000h-0FFFh	512

Табл. 108 Конфигурация на флаш-паметта на серията Z8Encore!F083A

Следващите фигури показват разделянето на флаш-паметта на сектори и страници.

0000h 01FFh	флаш-сектор 0		флаш-страница 0	0000h 01FFh
0200h 03FFh	флаш-сектор 1		флаш-страница 1	0200h 03FFh
0400h 05FFh	флаш-сектор 2		флаш-страница 2	0400h 05FFh
0600h 07FFh	флаш-сектор 3		флаш-страница 3	0600h 07FFh
0800h 09FFh	флаш-сектор 4		флаш-страница 4	0800h 09FFh
0A00h 0BFFh	флаш-сектор 5		флаш-страница 5	0A00h 0BFFh
0C00h 0DFFh	флаш-сектор 6		флаш-страница 6	0C00h 0DFFh
0E00h 0FFFh	флаш-сектор 7		флаш-страница 7	0E00h 0FFFh

Фиг. 229 Флаш-памет на ZF043A



Фиг. 230 Флаш-памет на ZF083A

25.14.2 Флаш-информационна област

Флаш-информационната област е физически разделена от Програмната Памет и се разполага на адреси FE00h ÷ FE7Fh. В тази област се съхраняват фабрични настройки на VBO-модула и вътрешния прецизен генератор, и фабрични калибрационни данни на АЦП-модула (Табл.109).

Адрес	Описание
FE00h – FE3Fh	Zilog-конфигурационни битове
FE40h – FE53h	Номер на устройството. 20-символен ASCII буквено-цифров код
FE54h – FE5Fh	-
FE60h – FE7Fh	Zilog-калибрациони данни

Табл. 109 Флаш-информационна област

Флаш-информационната област е достъпна когато бит 7 на регистър FPS е 1. Когато достъпът е разрешен, флаш-информационната област се „наслагва” върху Програмната Памет с адреси FE00h ÷ FE7Fh и четене от тези адреси води до четене на флаш-информационната област вместо от Програмната Памет. Флаш-информационната област е достъпна само за четене.

25.14.3 Флаш-контролер

Флаш-контролерът програмира и изтрива флаш-паметта. Той съдържа няколко защитни механизми срещу случайно програмиране или изтриване. Преди извършване на операции програмиране или изтриване, потребителят трябва да конфигурира регистрите FFREQH и FFREQL (**Flash Frequence High/Low**). Тези регистри формират 16-битова честотна стойност FFREQ, която задава времевите характеристики на операциите програмиране и изтриване.

$$FFREQ[15:0] = \frac{СистемнаТактоваЧестота(Hz)}{1000}$$

	Изтриването и програмирането на флаш-паметта изисква системният тактов сигнал да бъде в диапазона 10кHz ÷ 20MHz.

Защита на кода от външен достъп

Потребителският код може да бъде защитен от прочитане с дебъгер/програматор чрез нулиране на флаш-конфигурационния бит FRP (вижте 25.15 Флаш-конфигурационни битове).

Защита на кода от случайно програмиране и изтриване

Z8Encore!F083A микроконтролерите имат няколко нива на защита от случайно програмиране и изтриване на флаш-паметта.

- Защита чрез флаш-конфигурационния бит FWP.

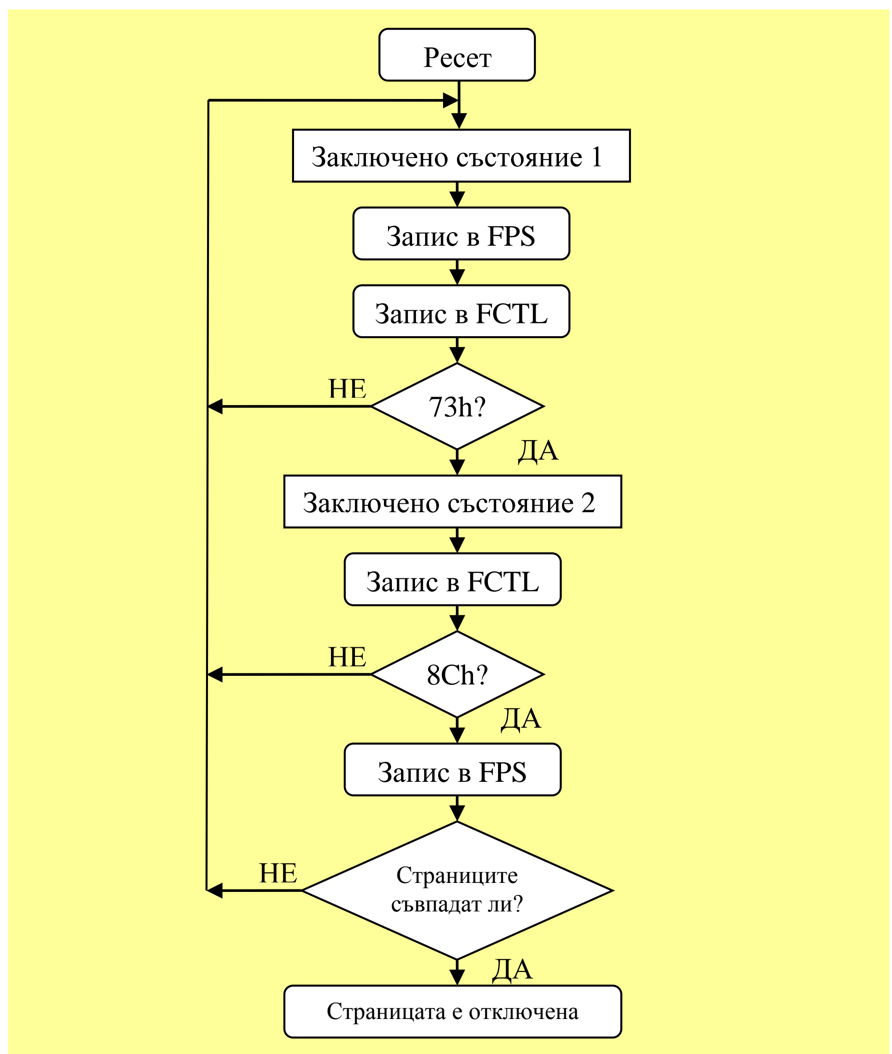
FWP	Описание
0	Програмирането и изтриването е забранено за цялата флаш-памет. В потребителския код програмирането, изтриването на страница и на цялата памет също е забранено. Цялото изтриване на паметта е възможно чрез дебъгера.
1	Програмирането, изтриването на страница и на цялата памет е разрешено.

Табл. 110 Защита на кода с флаш-конфигурационния бит FWP

- Защита чрез специална последователност от действия.

При ресет флаш-контролерът е заключен, за да предпази флаш-паметта от случайно изтриване и програмиране. За да програмирате или изтриете паметта, първо запишете номера

на страницата в регистър FPS. Отключете флаш-контролера чрез последователен запис в регистър FCTL на стойностите 73h и 8Ch. Запишете отново същата страница в регистър FPS. Ако страниците се различават, флаш-контролерът отново се връща в заключено състояние. Ако двете страници съвпадат, избраната страница става активна (фиг. 231)



Фиг. 231 Отключване на флаш-страница

След отключването на дадена страница, потребителят трябва да разреши програмирането или изтриването ѝ. Запис на стойност 95h в регистър FCTL води до изтриване на

страница само ако отключената страница се намира в незащитен сектор. Запис на друга стойност в регистър FCTL заключва флаш-контролера. Цяло изтриване напаметта не е позволено чрез потребителски код, а само чрез дебъгера.

След като отключи дадена страница, потребителят може да запише байт в нея. След като байтът е записан страницата остава отключена, позволявайки последващи записи на байтове. Последващи записи в регистър FCTL водят до заключване на страницата.

- Сектор-базирана защита

Финалният защитен механизъм е имплементиран на базата на сектори. Флаш-паметта е разделена на 8 сектори (вижте фиг.229 и 230).

Регистър FPROT съдържа битове за разрешаване на програмирането или изтриването на секторите. След като един сектор е защитен (съответния бит в регистър FPROT се запише в 1) той не може да бъде отключен от потребителския код (съответният бит не може да бъде нулиран с помощта на инструкции). Регистър FPROT се нулира след ресет.

Регистрите FPROT и FPS използват един и същ регистров адрес FF9h. Достъпът до FPROT се разрешава чрез запис на 5Eh в регистър FCTL. Запис на стойност различна от 5Eh в FCTL забранява достъпа до регистър FPROT и разрешава достъпа до регистър FPS.

Спазвайте следната процедура, когато конфигурирате регистър FPROT:

1. Запишете 00h във FCTL, за да нулирате флаш-контролера;

2. Запишете 5Eh във FCTL, за да разрешите достъпа до регистър FPROT;

3. Конфигурирайте регистър FPROT;

4. Запишете 00h във FCTL, за да нулирате флаш-контролера.

Регистър FPROT се инициализира в 0 след ресет, поставяйки всеки сектор в незащитено състояние. Когато бит в регистър FPROT се запише в 1, съответният сектор не може да бъде записван или изтриван. След установяване на бит в регистър FPROT, битът не може да бъде нулиран от потребителя.

Изтриване на страница

Флаш-страницата е минималния блок памет, който може да бъде изтрит. Изтрити байтове имат стойост FFh. Само страници, които се намират в незащитени сектори могат да бъдат изтрити. За да изриете страница, първо трябва да я отключите (Фиг.231) и след това да запишете стойност 95h в регистър FCTL. По време на изтриването процесорът спира да изпълнява инструкции, но тактовият сигнал и периферийните модули продължават да работят.

Изтриване на цялата памет

Цялата памет може да бъде изтрита наведнъж само с дебъгер.

Програмиране на байт

Записът на байт във флаш-паметта от потребителския код става с помощта на инструкциите LDC и LDCI. Записът е разрешен само след отключване на съответната страница, в която ще се извършва записа (фиг.231). Изтрит байт има стойност FFh. Операцията програмиране се използва само да

промени даден бит от 1 в 0. Един или повече битове могат да бъдат променени от 0 в 1 само чрез операция изтриване.

	Един байт не може да бъде записан повече два пъти преди страницата да бъде изтрита.

25.14.4 Описание на управляващите регистри

Регистър FCTL (Flash Control)

Флаш-контролерът трябва да бъде отключен преди флаш-паметта да може да бъде програмирана или изтриване. Запис на стойностите 73h и 8Ch последователно в регистър FCTL отключва флаш-контролера. След като флаш-контролерът е отключен, флаш-паметта е разрешена за пълно изтриване[1] или изтриване на страници чрез запис на съответните команди в регистър FCTL.

Забележка[1]: Цялото изтриване на флаш-паметта е възможно само чрез дебъгера.

Запис на невалидна стойност или невалидна последователност от стойности във FCTL води до заключване на флаш-контролера. Регистър FCTL и регистър FSTAT използват един и същ регистров адрес. При запис се използва регистър FCTL, а при четене FSTAT.

Бит	7	6	5	4	3	2	1	0
Име	FCMD							
Ресет	0	0	0	0	0	0	0	0
Ч/З	З	З	З	З	З	З	З	З
Адрес	FF8h							

Бит	Описание
7-0	**Флаш-команда** 73h – Първа команда за отключване 8Ch – Втора команда за отключване 95h – Команда за изтриване на страница 63h – Команда за изтриване на цялата памет 5Eh – Разрешаване на достъпа до регистър FPROT

Табл. 111 FCTL

Регистър FSTAT (Flash Status)

Регистър FSTAT указва текущото състояние на флаш-контролера. Регистър FSTAT и регистър FCTL използват един и същ регистров адрес. При запис се използва регистър FCTL, а при четене FSTAT.

Бит	7	6	5	4	3	2	1	0
Име			FSTAT					
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч	Ч	Ч	Ч	Ч	Ч	Ч	Ч
Адрес	FF8h							

Бит	Описание
7-6	Тези битове са резервирани и трябва да бъдат 0.
5-0	**Състояние на флаш-контролера** 000000 – Флаш-контролерът е заключен 000001 – Първата команда за отключване е приета 000010 – Втората команда за отключване е приета 000011 – Флаш-контролерът е отключен 000100 – Регистър FPROT е избран 001xxx – Операция програмиране е в прогрес 010xxx – Операция изтриване на страница е в прогрес 100xxx – Операция пълно изтриване е в прогрес

Табл. 112 FSTAT

Регистър FPS (Flash Page Select)

Регистър FPS избира флаш-страница, която ще бъде програмирана или изтривана. Регистър FPS и регистър FPROT използват един и същ регистров адрес. FPROT е достъпен когато флаш-контролерът е заключен и регистър FCTL е записан с 5Eh. Във всички други случаи е достъпен регистър FPS.

Бит	7	6	5	4	3	2	1	0
Име	INFO_EN	PAGE						
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FF9h							

Бит	**Описание**
7	**Разрешаване на флаш-информационната област** 0 – Флаш-информационната област е забранена 1 – Флаш-информационната област е разрешена
6-0	**Избор на страница** Това 7 битово поле указва флаш-страница за изтриване и отключване. Тези битове се явяват старшите битове [15-9] на адресите във флаш-паметта. За Z8F04xx микроконтролерите старшите 4 бита трябва да бъдат 0.

Табл. 113 FPS

Регистър FPROT (Flash Sector Protect)

Регистър FPROT съдържа разрешаващи битове за всеки флаш-сектор. Тези битове могат да бъдат установявани с инструкции, но не могат да бъдат нулирани с инструкции.

Регистър FPROT и регистър FPS използват един и същ регистров адрес. FPROT е достъпен когато флаш-контролерът е заключен и регистър FCTL е записан с 5Eh.

Във всички други случаи е достъпен регистър FPS.

Бит	7	6	5	4	3	2	1	0
Име	SPROT7	SPROT6	SPROT5	SPROT4	SPROT3	SPROT2	SPROT1	SPROT0
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FF9h							

Бит	**Описание**
7-0	**Защита на сектор** 0 – Секторът е незащитен 1 – Секторът е защитен

Табл. 114 FPROT

Регистър FFREQH (Flash Frequency High)

Бит	7	6	5	4	3	2	1	0
Име	FFREQH							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FFAh							

Бит	**Описание**
7-0	**Старши байт на 16 битовата флаш-честотна стойност FFREQ**

Табл. 115 FFREQH

Регистър FFREQL (Flash Frequency Low)

Бит	7	6	5	4	3	2	1	0
Име	FFREQL							
Ресет	0	0	0	0	0	0	0	0

Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FFBh							
Бит	**Описание**							
7-0	**Младши байт на 16 битовата флаш-честотна стойност FFREQ**							

Табл. 116 FFREQL

25.15 Флаш-конфигурационни битове

25.15.1 Общи сведения

Флаш-конфигурационните битове позволяват на потребителя да управлява следните функции на микроконтролера:

- Генериране на прекъсване или ресет при препълване на WDT-таймера;
- Разрешаване на WDT-таймера след ресет;
- Защитаване на потребителския код от прочитане;
- Защитаване на части или на целия потребителски код от случайно програмиране или изтриване;
- Изключване на VBO схемата в STOP-режим;
- Конфигуриране на кварцовия генератор:
    - кварцови резонатори 32kHz ÷ 1MHz;
    - кварцови резонатори 0.5MHz ÷ 5MHz;
    - кварцови резонатори 5MHz ÷ 20MHz;
    - външна RC верига (< 4MHz).
- Настройка на вътрешния прецизен генератор.

След всяко препрограмиране на флаш-конфигурационните битове, микроконтролерът трябва да се ресетне, за да може новите настройки да станат активни. При всеки тип ресет (системен или излизане от STOP-режим) флаш-конфигурационните битове се изчитат от Програмната

Памет и се записват в конфигурационни регистри. Тези регистри не са част от Регистровата Памет и не са достъпни за запис или четене.

25.15.2 Типове конфигурационни битове

Потребителски флаш-конфигурационни битове

Тези битове заемат първите два байта на Програмната памет. Инфомацията в тези байтове се губи при изтриване на флаш-страница 0 на Програмната Памет.

Бит	7	6	5	4	3	2	1	0
Име	WDT_RES	WDT_AO	OSC_SEL[1:0]		VBO_AO	FRP		FWP
Ресет	Н	Н	Н	Н	Н	Н	Н	Н
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	Програмна Памет 0000h							
Н - непроменен								

Бит	**Описание**
7	**Реакция на WDT-таймера при препълване** 0 – WDT-таймерът генерира прекъсване 1 – WDT-таймерът генерира системен ресет
6	**Включване на WDT-таймера** 0 – WDT-таймерът е включен и не може да бъде забранен 1 – WDT-таймерът не е включен. WDT-таймерът се включва при изпълнение на инструкцията WDT. Веднъж включен таймерът може да бъде забранен само от ресет.
5-4	**Режим на генератора** 00 – Външна RC верига (< 4MHz) 01 – кварцов резонатор (32kHz ÷ 1MHz) 10 – кварцов (керамичен) резонатор (0.5MHz ÷ 5MHz) 11 – кварцов резонатор (5MHz ÷ 20MHz)
3	**Разрешаване на VBO защитата** 0 – VBO защитата е забранена в STOP-режим 1 – VBO защитата винаги е разрешена (дори и в STOP-режим)

2	**Защита на флаш-паметта (програмния код) от прочитане** 0 – Програмния код е защитен 1 – Програмния код е незащитен
1	Този бит е резервиран и трябва да бъде 1.
0	**Защита на флаш-паметта от програмиране и изтриване** 0 – Програмирането и изтриването е забранено за цялата флаш-памет. В потребителския код програмирането, изтриването на страница и на цялата памет също е забранено. Цялото изтриване на паметта е възможно чрез дебъгера. 1 – Програмирането, изтриването на страница и на цялата памет е разрешено.

Табл. 117 Потребителски флаш-конфигурационен байт 0

Бит	7	6	5	4	3	2	1	0
Име	VBO_RES			XTLDIS				
Ресет	Н	Н	Н	Н	Н	Н	Н	Н
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	Програмна Памет 0001h							
Н – непроменен								

Бит	**Описание**
7	1 – VBO-модула генерира системен ресет
6-5	Тези битове са резервирани и трябва да бъдат 1.
4	**Включване на кварцовия генератор при ресет** 0 – Кварцовият генератор се включва при ресет 1 – Кварцовият генератор не се включва при ресет
3-0	Тези битове са резервирани и трябва да бъдат 1.

Табл. 118 Потребителски флаш-конфигурационен байт 1

Фабрични флаш-конфигурационни битове

Тези флаш-конфигурационни битове са настроени фабрично и не се нуждаят от потребителска намеса. По тази причина тази тема няма да бъде разгледана в книгата.

25.16 Енерго-независима памет за данни (NVDS)

25.16.1 Общи сведения

Z8Encore!F083A микроконтролерите съдържат енергонезависима памет NVDS (**Non-Volatile Data Storage**) с обем до 100 байта, която издържа на 100 000 цикъла за запис.

За достъп до NVDS се използват две подпрограми: подпрограма за запис и подпрограма за четене. Тези подпрограми са предефинирани и се разполагат на адреси извън Програмната Памет, използвана от потребителския код. Достъпът до тях става с инструкцията CALL.

За да не разрушават потребителския код, тези подпрограми съхраняват в стека текущия набор от работни регистри R0 ÷ R15. При излизане от тези подпрограми съдържанието на работните регистри се възстановява.

При достъпа до NVDS прекъсванията не са забранени. Всяко прекъсване, което възниква по време на достъп до NVDS, не трябва да променя работните регистри и текущото съдържание на стека, в противен случай съдържанието на NVDS може да бъде повредено. Zilog препоръчва прекъсванията да бъдат забранени преди достъп до NVDS.

За правилна работа на NVDS, флаш-честотните регистри FFREQH:FFREQL трябва да бъдат програмирани в зависимост от стойността на системната тактова честота (вижте 25.14 Флаш-памет).

25.16.2 Запис на байт в NVDS

За да запише байт в NVDS, потребителският код трябва

първо да запише адреса (00h ÷ 63h) и след това данните (00h ÷ FFh) в стека. Записът на данните в NVDS се осъществява чрез извикване на подпограмата за запис на адрес 20B3h. При връщане от подпрограмата състоянието на резултата от записа се съхранява в работния регистър R0 (Табл.119). Потребителският код трябва да махне адреса и байта с данни от стека.

Запис на невалиден адрес няма никакъв ефект.

Бит	7	6	5	4	3	2	1	0
Име						FE	IGADDR	WE
Адрес	Работен регистър R0							

Бит	**Описание**
7-3	-
2	**Флаш-грешка** 0 – Няма флаш-грешка 1 – Възниканала е флаш-грешка
1	**Невалиден адрес** 0 – Валиден адрес 1 – Невалиден адрес
0	**Грешка при запис** 0 – Няма грешка 1 – Има грешка. При запис записаната стойност автоматично се изчита и ако прочетената стойност се различава от записваната, този бит се установява.

Табл. 119 Статус байт при NVDS запис

25.16.3 Четене на байт от NVDS

За да прочете байт от NVDS, потребителският код трябва първо да запише адреса (00h ÷ 63h) в стека. Четенето на данните от NVDS се осъществява чрез извикване на подрпограмата за четене на адрес 2000h. При връщане от

подпрограмата прочетеният байт са намира в работния регистър R0, а състоянието на резултата от четенето се съхранява в работния регистър R1 (Табл.120). Потребителският код трябва да махне адреса от стека.

Четене от невалиден адрес връща стойност FFh.

Бит	7	6	5	4	3	2	1	0
Име				DE		FE	IGADDR	
Адрес	Работен регистър R1							

Бит	**Описание**
7-5	-
4	**Грешка на данните** 0 – Няма грешка 1 – Възниканала е грешка. Когато се чете NVDS адрес, ако се намери грешка в най-последните данни, съответстващи на този адрес, този бит се установява.
3	-
2	**Флаш-грешка** 0 – Няма флаш-грешка 1 – Възниканала е флаш-грешка
1	**Невалиден адрес** 0 – Валиден адрес 1 – Невалиден адрес
0	-

Табл. 120 Статус байт при NVDS запис

25.17 Дебъгер

25.17.1 Общи сведения

Z8 Encore! фамилията микроконтролери съдържа вграден дебъгер (**OCD – On-Chip Debugger**), който има следните характеристики:

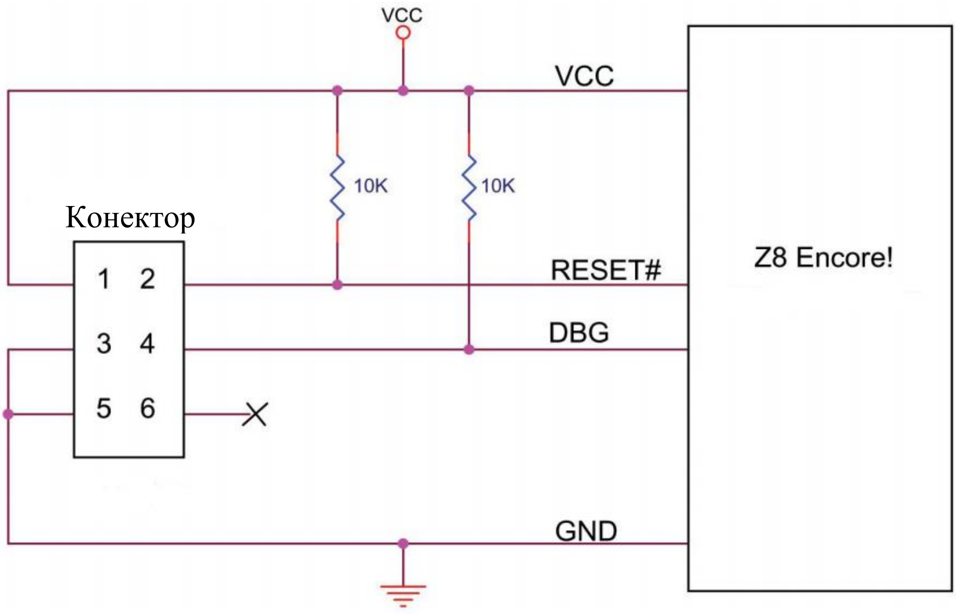
- Четене и запис на Регистровата Памет;
- Четене и запис на Програмната Памет;
- Четене и запис на Паметта за Данни;
- Задаване на точки на прекъсване (**breakpoints**);
- Изпълняване на инструкции на процесора eZ8.

Връзката между вградения дебъгер на микроконтролера и компютър става посредством интерфейса **USB SmartCable** (Фиг. 232)

Фиг. 232 USB SmartCable интерфейс

25.17.2 Свързване на дебъгера към микроконтолера

Следващата фигура показва начина на свързване на **USB SmarCable** интерфейса с микроконтролера.



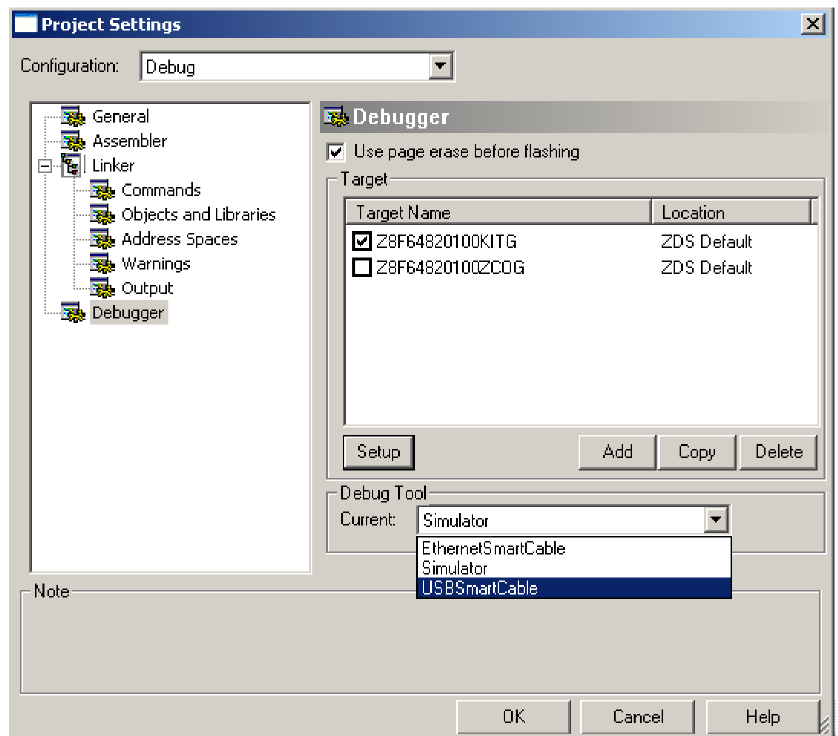
Фиг. 233 Свързване на USB SmartCable интерфейс с Z8 Encore! микроконтролер

25.17.3 Тестване/дебъгване с USB SmartCable

За да използвате дебъгера, трябва да го изберете като инструмент за тестване/дебъгване. За целта изберете **Project → Settings…**. Отваря се прозореца **Project Settings** (Фиг.234). Изберете категорията **Debugger** и от менюто **Debug Tool:Current** изберете **USBSmartCable.**

Свържете дебъгера към компютъра и платката (Фиг.235).

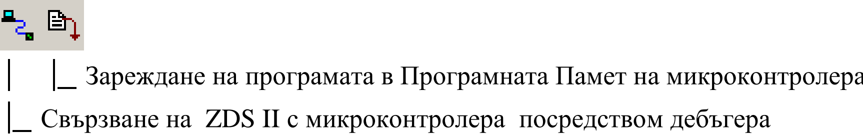
	Захранването на развойната платка трябва да бъде изключено преди да свържете/откачите дебъгера към/от нея.



Фиг. 234 Project Ssettings

Фиг. 235 Свързване на дебъгера към платката и компютъра

Използвайте следните бутони в средата ZDS II, за да се свържете с микроконтролера на платката и да заредите програмата в Програмната Памет.



Може да използвате и ресет бутона . ZDS II средата първо ще се свърже с микроконтролера на платката, след което ще зареди програмата в Програмната Памет.

Допълнителна информация може да намерите в 24.5.2 Тестване и дебъгване.

25.18 Електрически параметри

Пълно описание на електрическите параметри на Z8Encore!F083A серията микроконтролери може да намерите в оригиналната документация Z8Encore!®F083A Series. Product Specification (PS026310-1212).

25.19 Информация за поръчка

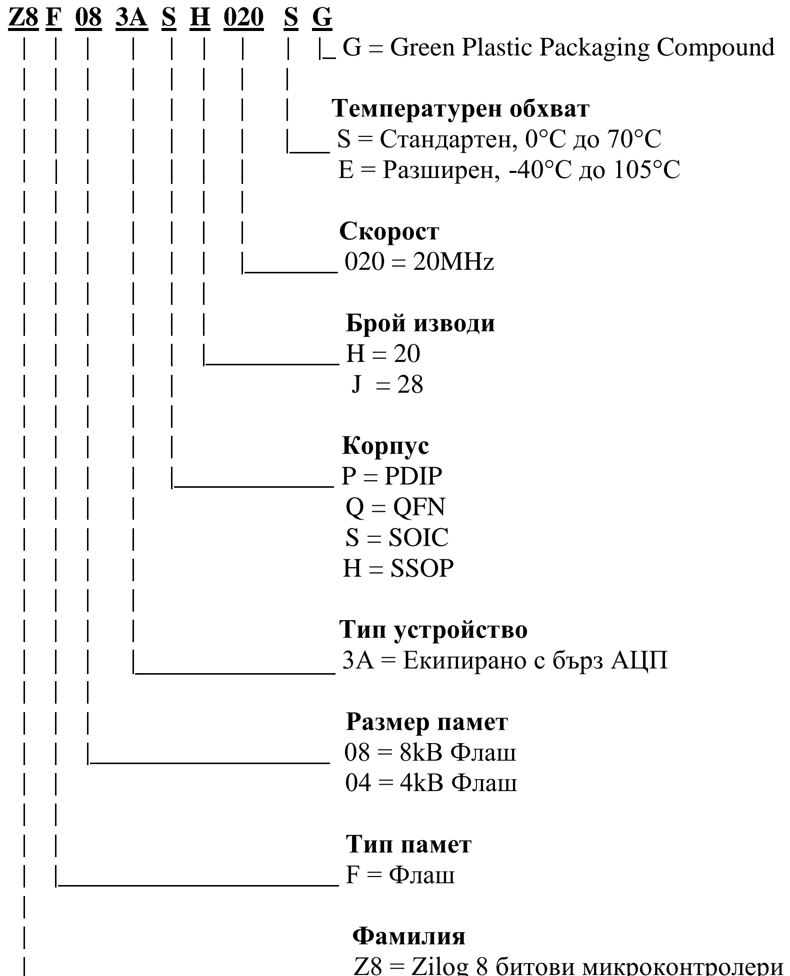
Може да използвате следващата таблица при избора си на микроконтролер от серията Z8 Encore!F083A.

Номер на устройството	Флаш	RAM	NVDS	АЦП канали	Описание
Z8 Encore!F083A с 8kB Флаш (0°C до 70°C)					
Z8F083ASH020SG	8kB	256B	100B	7	SOIC-20
Z8F083AHH020SG				7	SSOP -20
Z8F083APH020SG				7	PDIP-20
Z8F083AQH020SG				7	QFN-20
Z8F083ASJ020SG				8	SOIC-28
Z8F083AHJ020SG				8	SSOP-28
Z8F083AQJ020SG				8	QFN-28
Z8 Encore!F083A с 8kB Флаш (-40°C до 105°C)					
Z8F083ASH020EG	8kB	256B	100B	7	SOIC-20
Z8F083AHH020EG				7	SSOP -20
Z8F083APH020EG				7	PDIP-20
Z8F083AQH020EG				7	QFN-20
Z8F083ASJ020EG				8	SOIC-28
Z8F083AHJ020EG				8	SSOP-28
Z8F083AQJ020EG				8	QFN-28
Z8 Encore!F083A с 4kB Флаш (0°C до 70°C)					
Z8F043ASH020SG	4kB	256B	100B	7	SOIC-20
Z8F043AHH020SG				7	SSOP -20
Z8F043APH020SG				7	PDIP-20
Z8F043AQH020SG				7	QFN-20
Z8F043ASJ020SG				8	SOIC-28
Z8F043AHJ020SG				8	SSOP-28
Z8F043AQJ020SG				8	QFN-28
Z8 Encore!F083A с 4kB Флаш (-40°C до 105°C)					
Z8F043ASH020EG	4kB	256B	100B	7	SOIC-20
Z8F043AHH020EG				7	SSOP -20

Z8F043APH020EG				7	PDIP-20
Z8F043AQH020EG				7	QFN-20
Z8F043ASJ020EG				8	SOIC-28
Z8F043AHJ020EG				8	SSOP-28
Z8F043AQJ020EG				8	QFN-28

Табл. 121 Z8 Encore!F083A серия микроконтролери

Значение на символите в номера на устройството:

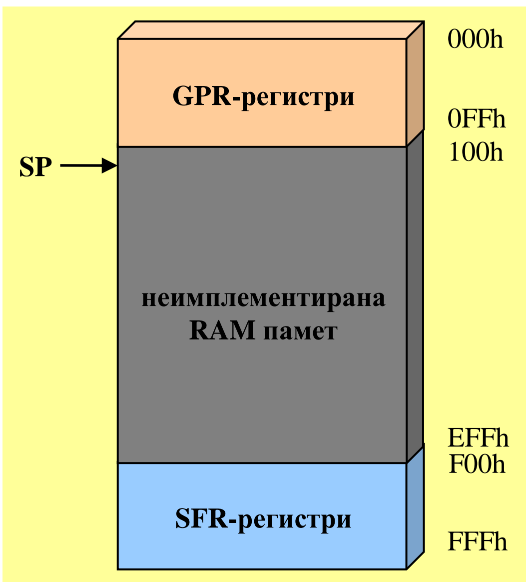


26 Практически примери

26.1 С какво разполагаме

Преди да преминем към практическите примери, нека да направим един бърз преглед на ресурсите на микроконтролера ZF083A и на развойната платка, която ще използваме.

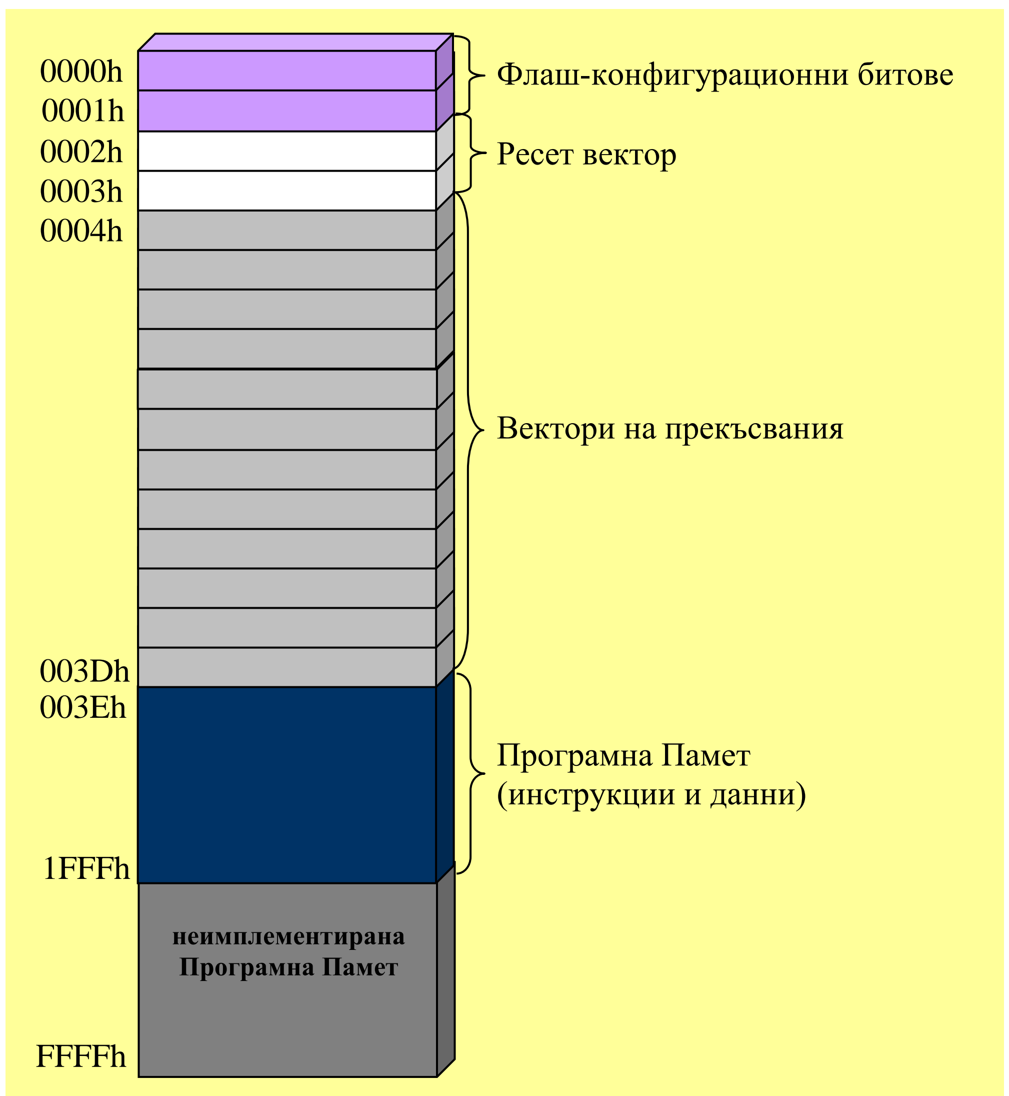
ZF083A има само една страница с RAM памет (000h÷0FFh).



Фиг. 236 Регистрова Памет на ZF083A

Това означава, че няма да ни се налага да променяме указателя на страниците RP[3:0]. Ще конфигурираме указателя към стека SP да сочи към първия адрес на следващата неимплементирана страница. Така първия запис в стека ще започне от адрес 0FFh. Ако сте забравили как работи стека, прочетете още веднъж 21.3 Стек.

Следващата фигура описва Програмната Памет на ZF083A. Програмната Памет за инструкции и константни данни се разполага на адреси 003E÷1FFFh.



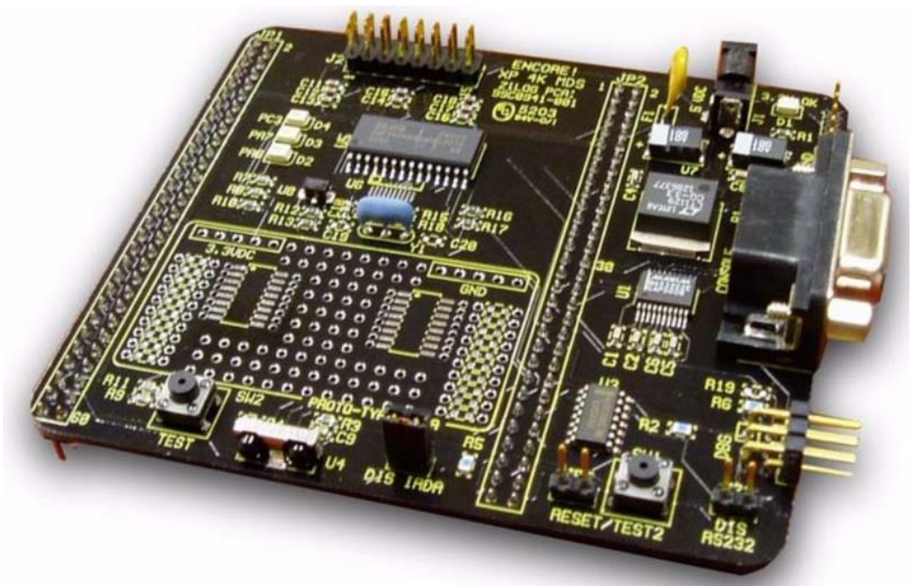
Фиг. 237 Програмна Памет на ZF083A

Следващата фигура показва развойната платка, част от развойния комплект Z8F083A0128ZCOG. Основните елементи, монтирани на платката са:

- Регулатор на напрежение U7 (3.3V);
- Керамичен резонатор Y1 (20MHz);
- Три светодиода D2, D3 и D4, свързани съответно към

изводи PA6, PA7 и PC3;

- Бутон SW2, свързан към извод PA2;
- Ресет бутон SW1.

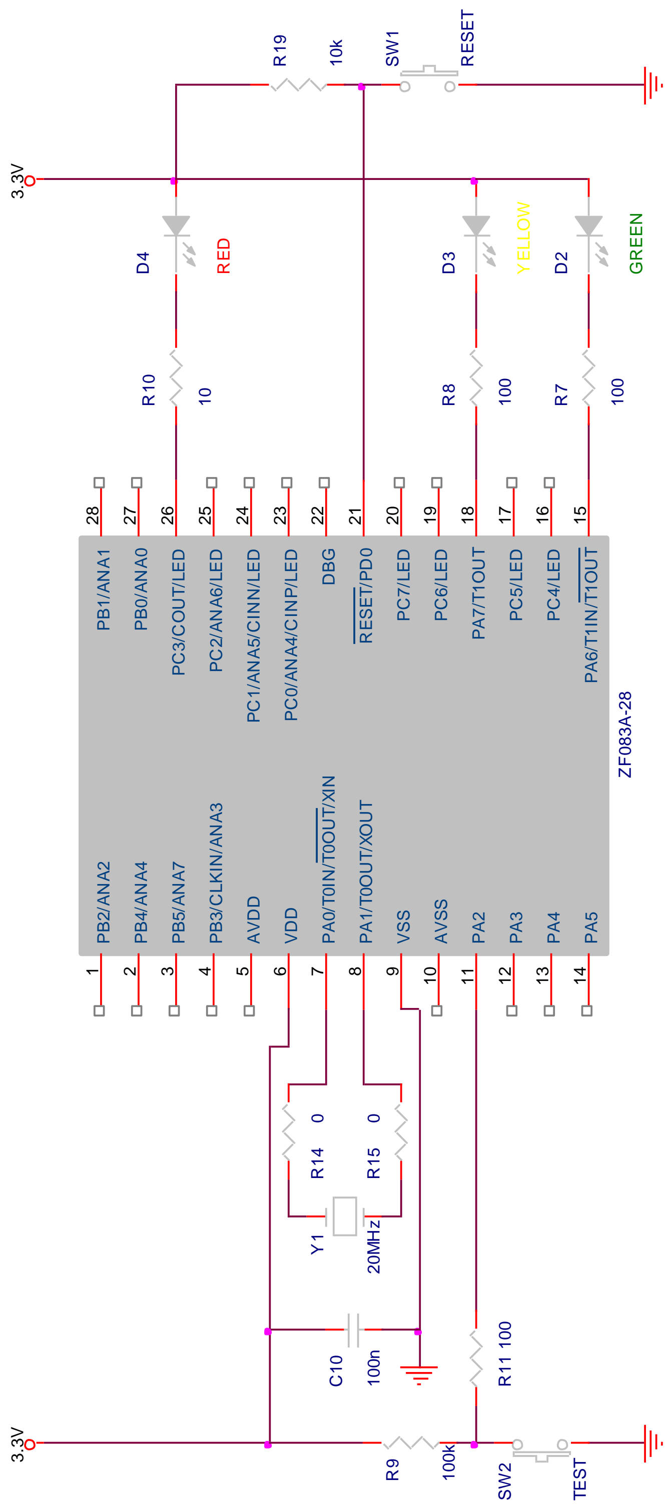


Фиг. 238 ZF083A развойна платка

Фиг.239 показва в опростен вид част от принципната електрическа схема на развойната платка, която ще използваме в нашите примери.

След прицпината електрическа схема ще намерите шаблон на асемблерен сорс-файл, който ще използваме. Разучете внимателно този файл. Не всички елементи в него са задължителни, но показаното ще Ви научи да си структурирате програмите професионално.

Може да си създадете шаблонен проект и да добавите в него този файл, като го преименувате примерно main.asm, и да го асемблирате, за да видите че всичко е наред. След това може да използвате този шаблонен проект като основа на всеки Ваш нов проект.



Фиг. 239 Принципна електрическа схема

```
;************************************************************
; File: template.asm
; Description:
;
;
;------------------------------------------------------------------------------------
;
; Modified: dd mm yy
;************************************************************
; Copyright (C) All Rights Reserved.
;************************************************************


;************************************************************
;  Include files
;************************************************************
        INCLUDE "ez8.inc"
```

Съдържа имената на всички SFR-регистри. Това позволява директното използване на тези имена вместо адресите на регистрите в инструкциите. Тук може да включвате и .inc файлове създадени от Вас.

```
;************************************************************
; Local Constant Defitions
;************************************************************

; Stack starting address
NEAR_STACK      EQU     %0100
```

Дефиниране на константа, която представлява начало на стека.

```
;************************************************************
;Default User Flash Options
;Change values of flash_option_byte1 and flash_option_byte2 if user option
;bits are used.
;************************************************************
        DEFINE ___flash_option_segment,SPACE=ROM, ORG=0
        SEGMENT ___flash_option_segment

flash_option_byte1: DB %FF
flash_option_byte2: DB %FF
```

Заделяне и инициализиране на адреси 0000h и 0001h в Програмната Памет. Може да задавате стойности по свой избор, за да управлявате определени функции на микроконтролера (вижте **26.15 Флаш-конфигурационни битове**).

```
;************************************************************
; Vector table
;************************************************************
; USED vectors:
        VECTOR RESET       = startup        ; RESET
```

```
;UNUSED vectors
        VECTOR WDT      = isr_dummy; Watch-Dog Timer
        VECTOR POTRAP   = isr_dummy; Primary Oscillator Fail Trap
        VECTOR WOTRAP   = isr_dummy; Watchdog Oscillator Fail Trap
        VECTOR TRAP     = isr_dummy; Illegal Instruction Trap
        VECTOR TIMER1   = isr_dummy; Timer1
        VECTOR TIMER0   = isr_dummy; Timer0
        VECTOR ADC      = isr_dummy; ADC
        VECTOR P0AD     = isr_dummy; PA0
        VECTOR P1AD     = isr_dummy; PA1
        VECTOR P2AD     = isr_dummy; PA2
        VECTOR P3AD     = isr_dummy; PA3
        VECTOR P4AD     = isr_dummy; PA4
        VECTOR P5AD     = isr_dummy; PA5
        VECTOR P6AD     = isr_dummy; PA6
        VECTOR P7AD     = isr_dummy; PA7
        VECTOR C3       = isr_dummy; PC3
        VECTOR C2       = isr_dummy; PC2
        VECTOR C1       = isr_dummy; PC1
        VECTOR C0       = isr_dummy; PC0
```

Конфигуриране на таблицата с векторите на прекъсвания. Добра практика е неизползваните адреси да се инициализират с името на фиктивна ISR-подпрограма. Така ако по невнимание разрешите някое прекъсване, при възникване на заявка за прекъсване програмата ще извърши преход към фиктивната подпрограма, а не към произволно място в Програмната Памет.

```
;RESERVED VECTORS are declared here to prevent linker from filling
;these locations with other code segments

        DEFINE __VECTORS_008, SPACE = ROM, ORG = %008
        SEGMENT __VECTORS_008

__VECTOR_008:   DW isr_dummy

        DEFINE __VECTORS_00E, SPACE = ROM, ORG = %00E
        SEGMENT __VECTORS_00E

__VECTOR_00E:   DW isr_dummy
__VECTOR_010:   DW isr_dummy
__VECTOR_012:   DW isr_dummy
__VECTOR_014:   DW isr_dummy
```

Инициализиране на неиплементираните адреси от таблицата с векторите на прекъсванията с адреса на фиктивна ISR-подпрограма (вижте **Табл.32 Източници на прекъсване в ZF083A**).

```
        DEFINE __VECTORS_028, SPACE = ROM, ORG = %028
        SEGMENT __VECTORS_028

__VECTOR_028:   DW isr_dummy
__VECTOR_02A:   DW isr_dummy
__VECTOR_02C:   DW isr_dummy
```

```
__VECTOR_02E:     DW isr_dummy

        DEFINE __VECTORS_038, SPACE = ROM, ORG = %038
        SEGMENT __VECTORS_038

__VECTOR_038:     DW isr_dummy

;*************************************************************
;Program starts here
;*************************************************************

        DEFINE main_user_program_segment, SPACE = ROM
        SEGMENT main_user_program_segment

startup:
   DI                                    ; Prevent interruption
   SRP    #%00                           ; Working registers 00-0f
   LDX    SPL, #LOW(NEAR_STACK)          ; Initialize stack pointer
   LDX    SPH, #HIGH(NEAR_STACK)
   CALL main
exit:
   JR exit

;*************************************************************
; Subroutine: Main routine
;*************************************************************
main:




   RET

;*************************************************************
; Subroutine:
; Decription:
;
;*************************************************************
```

Инициализиране на регистър RP (**Register Pointer**) и на указателя към стека SP. След тази инициализация програмата извършва преход към началото на Вашия код.

Тук се разполага основния потребителски код.

Тук се разполагат подпрограмите, извиквани от основния код или от други потребителски подпрограми.

```
;**************************************************************
; If PC stays here, code execute an expected vector interrupt
;**************************************************************
isr_dummy:
   JP    $
```

Фиктивна ISR-подпрограма, която съдържа просто един безкраен цикъл. Правилно работеща програма не трябва никога да достига до тук.

```
;**************************************************************
;        GLOBAL SYMBOLS
;**************************************************************
```

Ако програмата се състои от няколко сорс-файла, тук може да експортирате всички етикети с директивата XDEF, които ще се използват и в другите сорс-файлове.

```
;**************************************************************
;        EXTERNAL SYMBOLS
;**************************************************************
```

Ако програмата се състои от няколко сорс-файла, тук може да импортирате всички етикети с директивата XREF, дефинирани в други сорс-файлове, и които ще се използват и в текущия сорс-файл.

```
   END
```

По подразбиране процесорът използва вътрешния прецизен генератор 20MHz като източник на системен тактов сигнал. Това означава, че периодът на тактовия сигнал е 1/20MHz = 0.05µs. Всички примери в книгата ще използват този източник на тактов сигнал, но Вие може да експериментирате като конфигурирате кварцовия генератор с 20MHz кварцов резонатор, като източник на системен тактов сигнал.

Във всички следващи примери потребителският код ще бъде заграждан в сиви карета за по лесното му отделяне от шаблонния код.

26.2 Пример 1

Първият пример ще бъде съвсем елементарен. Ще накараме някой от светодиодите да светне, например D3.

```
;*****************************************************************
; File: main.asm
; Description:
;         Turn on LED D3
;-----------------------------------------------------------------------------------------
;
; Modified: 24 07 2014
;*****************************************************************
; Copyright (C) All Rights Reserved.
;*****************************************************************

;*****************************************************************
;  Include files
;*****************************************************************
        INCLUDE "ez8.inc"

;*****************************************************************
; Local Constant Defitions
;*****************************************************************

; Stack starting address
NEAR_STACK      EQU    %0100

; Port A bits position
PA7     EQU    80h
PA6     EQU    40h

;PxADDR subregisters
NO_REG        EQU    00h
PxDD          EQU    01h
PxAF          EQU    02h
PxOC          EQU    03h
PxHDE         EQU    04h
PxSMRE        EQU    05h
PxPUE         EQU    06h
PxAFS1        EQU    07h
PxAFS2        EQU    08h
```

```
;***************************************************************
;Default User Flash Options
;Change values of flash_option_byte1 and flash_option_byte2 if user option
;bits are used.
;***************************************************************
        DEFINE ___flash_option_segment,SPACE=ROM, ORG=0
        SEGMENT ___flash_option_segment

flash_option_byte1: DB %FF
flash_option_byte2: DB %FF

;***************************************************************
; Vector table
;***************************************************************
; USED vectors:
        VECTOR RESET        = startup       ; RESET

;UNUSED vectors
        VECTOR WDT          = isr_dummy; Watch-Dog Timer
        VECTOR POTRAP       = isr_dummy; Primary Oscillator Fail Trap
        VECTOR WOTRAP       = isr_dummy; Watchdog Oscillator Fail Trap
        VECTOR TRAP         = isr_dummy; Illegal Instruction Trap
        VECTOR TIMER1       = isr_dummy; Timer1
        VECTOR TIMER0       = isr_dummy; Timer0
        VECTOR ADC          = isr_dummy; ADC
        VECTOR P0AD         = isr_dummy; PA0
        VECTOR P1AD         = isr_dummy; PA1
        VECTOR P2AD         = isr_dummy; PA2
        VECTOR P3AD         = isr_dummy; PA3
        VECTOR P4AD         = isr_dummy; PA4
        VECTOR P5AD         = isr_dummy; PA5
        VECTOR P6AD         = isr_dummy; PA6
        VECTOR P7AD         = isr_dummy; PA7
        VECTOR C3           = isr_dummy; PC3
        VECTOR C2           = isr_dummy; PC2
        VECTOR C1           = isr_dummy; PC1
        VECTOR C0           = isr_dummy; PC0

;RESERVED VECTORS are declared here to prevent linker from filling
;these locations with other code segments

        DEFINE __VECTORS_008, SPACE = ROM, ORG = %008
        SEGMENT __VECTORS_008
```

```
__VECTOR_008:     DW isr_dummy

        DEFINE __VECTORS_00E, SPACE = ROM, ORG = %00E
        SEGMENT __VECTORS_00E

__VECTOR_00E:     DW isr_dummy
__VECTOR_010:     DW isr_dummy
__VECTOR_012:     DW isr_dummy
__VECTOR_014:     DW isr_dummy

        DEFINE __VECTORS_028, SPACE = ROM, ORG = %028
        SEGMENT __VECTORS_028

__VECTOR_028:     DW isr_dummy
__VECTOR_02A:     DW isr_dummy
__VECTOR_02C:     DW isr_dummy
__VECTOR_02E:     DW isr_dummy

        DEFINE __VECTORS_038, SPACE = ROM, ORG = %038
        SEGMENT __VECTORS_038

__VECTOR_038:     DW isr_dummy

;*************************************************************
;Program starts here
;*************************************************************

        DEFINE main_user_program_segment, SPACE = ROM
        SEGMENT main_user_program_segment

startup:
   DI                                   ; Prevent interruption
   SRP   #%00                           ; Working registers 00-0f
   LDX   SPL, #LOW(NEAR_STACK)          ; Initialize stack pointer
   LDX   SPH, #HIGH(NEAR_STACK)
   CALL main
exit:
   JR exit

;*************************************************************
; Subroutine: Main routine
;*************************************************************
main:
```

```
  LDX    PAOUT, #~PA7    ; Set low lewel on pin PA7
  LDX    PAADDR,#PxDD ; Select Port A data direction subregister
  LDX    PACTL, #~PA7     ; Make pin PA7 output

  JR       $ ; Stay here
  RET

;*************************************************************
; Subroutine:
; Decription:
;
;*************************************************************


;*************************************************************
; If PC stays here, code execute an expected vector interrupt
;*************************************************************
isr_dummy:
  JP    $

;*************************************************************
;         GLOBAL SYMBOLS
;*************************************************************

;*************************************************************
;         EXTERNAL SYMBOLS
;*************************************************************

   END
```

26.3 Пример 2

Този пример прави същото като предходния, но структурира програмата по друг начин.

```
;*************************************************************
; File: main.asm
; Description:
;         Turn on LED D3 ver.2
;-----------------------------------------------------------------------------------
;
; Modified: 25 07 2014
;*************************************************************
; Copyright (C) All Rights Reserved.
;*************************************************************

;*************************************************************
;  Include files
;*************************************************************
        INCLUDE "ez8.inc"

;*************************************************************
; Local Constant Defitions
;*************************************************************

; Stack starting address
NEAR_STACK      EQU    %0100

; Port A bits position
PA7     EQU    80h
PA6     EQU    40h

;PxADDR subregisters
NO_REG      EQU    00h
PxDD        EQU    01h
PxAF        EQU    02h
PxOC        EQU    03h
PxHDE       EQU    04h
PxSMRE      EQU    05h
PxPUE       EQU    06h
PxAFS1      EQU    07h
PxAFS2      EQU    08h
```

```
;*************************************************************
; Macro Defitions
;*************************************************************
YellowLedOn:        MACRO
LDX   PAOUT, #~PA7  ; Set low lewel on pin PA7
                    ENDMACRO

;*************************************************************
;Default User Flash Options
;Change values of flash_option_byte1 and flash_option_byte2 if user option
;bits are used.
;*************************************************************
        DEFINE ___flash_option_segment,SPACE=ROM, ORG=0
        SEGMENT ___flash_option_segment

flash_option_byte1: DB %FF
flash_option_byte2: DB %FF

;*************************************************************
; Vector table
;*************************************************************
; USED vectors:
        VECTOR RESET      = startup     ; RESET

;UNUSED vectors
        VECTOR WDT        = isr_dummy; Watch-Dog Timer
        VECTOR POTRAP     = isr_dummy; Primary Oscillator Fail Trap
        VECTOR WOTRAP     = isr_dummy; Watchdog Oscillator Fail Trap
        VECTOR TRAP       = isr_dummy; Illegal Instruction Trap
        VECTOR TIMER1     = isr_dummy; Timer1
        VECTOR TIMER0     = isr_dummy; Timer0
        VECTOR ADC        = isr_dummy; ADC
        VECTOR P0AD       = isr_dummy; PA0
        VECTOR P1AD       = isr_dummy; PA1
        VECTOR P2AD       = isr_dummy; PA2
        VECTOR P3AD       = isr_dummy; PA3
        VECTOR P4AD       = isr_dummy; PA4
        VECTOR P5AD       = isr_dummy; PA5
        VECTOR P6AD       = isr_dummy; PA6
        VECTOR P7AD       = isr_dummy; PA7
        VECTOR C3         = isr_dummy; PC3
        VECTOR C2         = isr_dummy; PC2
        VECTOR C1         = isr_dummy; PC1
```

```
        VECTOR C0            = isr_dummy; PC0

;RESERVED VECTORS are declared here to prevent linker from filling
;these locations with other code segments

        DEFINE __VECTORS_008, SPACE = ROM, ORG = %008
        SEGMENT __VECTORS_008

__VECTOR_008:    DW isr_dummy
        DEFINE __VECTORS_00E, SPACE = ROM, ORG = %00E
        SEGMENT __VECTORS_00E

__VECTOR_00E:    DW isr_dummy
__VECTOR_010:    DW isr_dummy
__VECTOR_012:    DW isr_dummy
__VECTOR_014:    DW isr_dummy

        DEFINE __VECTORS_028, SPACE = ROM, ORG = %028
        SEGMENT __VECTORS_028

__VECTOR_028:    DW isr_dummy
__VECTOR_02A:    DW isr_dummy
__VECTOR_02C:    DW isr_dummy
__VECTOR_02E:    DW isr_dummy

        DEFINE __VECTORS_038, SPACE = ROM, ORG = %038
        SEGMENT __VECTORS_038

__VECTOR_038:    DW isr_dummy

;****************************************************************
;Program starts here
;****************************************************************

        DEFINE main_user_program_segment, SPACE = ROM
        SEGMENT main_user_program_segment

startup:
   DI                                    ; Prevent interruption
   SRP   #%00                            ; Working registers 00-0f
   LDX   SPL, #LOW(NEAR_STACK)           ; Initialize stack pointer
   LDX   SPH, #HIGH(NEAR_STACK)
   CALL main
```

```
exit:
   JR exit

;************************************************************
; Subroutine: Main routine
;************************************************************
main:

   CALL    PortInit ; Ports initialization

   YellowLedOn    ; Turn on yellow led

   JR        $ ; Stay here
   RET

;************************************************************
; Subroutine: PortInit
; Decription: Initializes port pins
;
;************************************************************
PortInit:
   LDX    PAOUT, #PA7       ; Set high lewel on pin PA7
   LDX    PAADDR,#PxDD ; Select Port A data direction subregister
   LDX    PACTL, #~PA7     ; Make pin PA7 output
   RET

;************************************************************
; If PC stays here, code execute an expected vector interrupt
;************************************************************
isr_dummy:
   JP    $

;************************************************************
;          GLOBAL SYMBOLS
;************************************************************

;************************************************************
;          EXTERNAL SYMBOLS
;************************************************************


   END
```

26.4 Пример 3

В този пример ще разнообразим малко нещата, като накараме светодиода да мига през 1 секунда.

```
;*************************************************************
; File: main.asm
; Description:
;         LED D3 blinking
;-------------------------------------------------------------------------------
;
; Modified: 25 07 2014
;*************************************************************
; Copyright (C) All Rights Reserved.
;*************************************************************

;*************************************************************
;  Include files
;*************************************************************
          INCLUDE "ez8.inc"

;*************************************************************
; Local Constant Defitions
;*************************************************************

; Stack starting address
NEAR_STACK      EQU    %0100

; Port A bits position
PA7     EQU    80h
PA6     EQU    40h

;PxADDR subregisters
NO_REG        EQU    00h
PxDD          EQU    01h
PxAF          EQU    02h
PxOC          EQU    03h
PxHDE         EQU    04h
PxSMRE        EQU    05h
PxPUE         EQU    06h
PxAFS1        EQU    07h
PxAFS2        EQU    08h
```

```
;**************************************************************
; Variable Defitions
;**************************************************************

        SEGMENT    NEAR_DATA ; RData space

Counter1_16bits:      DS      2 ; Reserve 2  bytes for Counter1_16bits
Counter2_16bits:      DS      2 ; Reserve 2  bytes for Counter2_16bits

;**************************************************************
; Macro Defitions
;**************************************************************
YellowLedOn:          MACRO
LDX   PAOUT, #~PA7  ; Set low lewel on pin PA7
                      ENDMACRO

YellowLedOff:         MACRO
LDX   PAOUT, #PA7   ; Set high lewel on pin PA7
                      ENDMACRO

;**************************************************************
;Default User Flash Options
;Change values of flash_option_byte1 and flash_option_byte2 if user option
;bits are used.
;**************************************************************
        DEFINE ___flash_option_segment,SPACE=ROM, ORG=0
        SEGMENT ___flash_option_segment

flash_option_byte1: DB %FF
flash_option_byte2: DB %FF

;**************************************************************
; Vector table
;**************************************************************
; USED vectors:
        VECTOR RESET      = startup       ; RESET

;UNUSED vectors
        VECTOR WDT        = isr_dummy; Watch-Dog Timer
        VECTOR POTRAP     = isr_dummy; Primary Oscillator Fail Trap
        VECTOR WOTRAP     = isr_dummy; Watchdog Oscillator Fail Trap
        VECTOR TRAP       = isr_dummy; Illegal Instruction Trap
        VECTOR TIMER1     = isr_dummy; Timer1
```

```
        VECTOR TIMER0     = isr_dummy; Timer0
        VECTOR ADC        = isr_dummy; ADC
        VECTOR P0AD       = isr_dummy; PA0
        VECTOR P1AD       = isr_dummy; PA1
        VECTOR P2AD       = isr_dummy; PA2
        VECTOR P3AD       = isr_dummy; PA3
        VECTOR P4AD       = isr_dummy; PA4
        VECTOR P5AD       = isr_dummy; PA5
        VECTOR P6AD       = isr_dummy; PA6
        VECTOR P7AD       = isr_dummy; PA7
        VECTOR C3         = isr_dummy; PC3
        VECTOR C2         = isr_dummy; PC2
        VECTOR C1         = isr_dummy; PC1
        VECTOR C0         = isr_dummy; PC0

;RESERVED VECTORS are declared here to prevent linker from filling
;these locations with other code segments

        DEFINE __VECTORS_008, SPACE = ROM, ORG = %008
        SEGMENT __VECTORS_008

__VECTOR_008:   DW isr_dummy

        DEFINE __VECTORS_00E, SPACE = ROM, ORG = %00E
        SEGMENT __VECTORS_00E

__VECTOR_00E:   DW isr_dummy
__VECTOR_010:   DW isr_dummy
__VECTOR_012:   DW isr_dummy
__VECTOR_014:   DW isr_dummy

        DEFINE __VECTORS_028, SPACE = ROM, ORG = %028
        SEGMENT __VECTORS_028

__VECTOR_028:   DW isr_dummy
__VECTOR_02A:   DW isr_dummy
__VECTOR_02C:   DW isr_dummy
__VECTOR_02E:   DW isr_dummy

        DEFINE __VECTORS_038, SPACE = ROM, ORG = %038
        SEGMENT __VECTORS_038

__VECTOR_038:   DW isr_dummy
```

```
;*****************************************************************
;Program starts here
;*****************************************************************

          DEFINE main_user_program_segment, SPACE = ROM
          SEGMENT main_user_program_segment

startup:
   DI                                        ; Prevent interruption
   SRP   #%00                                ; Working registers 00-0f
   LDX   SPL, #LOW(NEAR_STACK)               ; Initialize stack pointer
   LDX   SPH, #HIGH(NEAR_STACK)
   CALL main
exit:
   JR exit

;*****************************************************************
; Subroutine: Main routine
;*****************************************************************
main:

  CALL    PortInit   ; Ports initialization

main_loop:
  YellowLedOn       ; Turn on yellow led
  CALL  Delay_1s  ; Wait 1s
  YellowLedOff      ; Turn off yellow led
  CALL  Delay_1s  ; Wait 1s
  JR    main_loop    ; Repeat the cycle

 RET

;*****************************************************************
; Subroutine: PortInit
; Decription: Initializes port pins
;
;*****************************************************************
PortInit:
   LDX    PAOUT, #PA7       ; Set high lewel on pin PA7
   LDX    PAADDR,#PxDD ; Select Port A data direction subregister
   LDX    PACTL, #~PA7     ; Make pin PA7 output
   RET
```

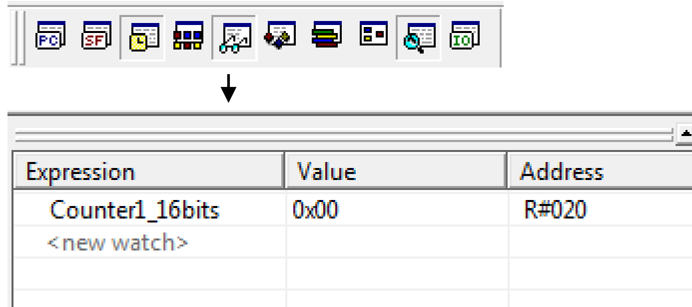
```
;*****************************************************************
; Subroutine: Delay_1s
; Decription: Provides 1 second delay (20 000 000 clock cycles)
;
;*****************************************************************
Delay_1s:

   LDX    Counter2_16bits,     #00h  ; Load counter2 high byte
   LDX    Counter2_16bits+1, #1Fh ; Load counter2 low byte

reload_counter1:
   LDX    Counter1_16bits,     #FCh  ; Load counter1 high byte
   LDX    Counter1_16bits+1, #FFh  ; Load counter1 low byte

delay_loop:
   DECW   Counter1_16bits ; Decrement Counter1 by 1
   JR         NZ, delay_loop   ; Repeat if Counter1 is not zero

   DECW    Counter2_16bits       ; Decrement Counter2 by 1
   JR          NZ, reload_counter1; Exit if Counter2 value is zero

   RET

;*****************************************************************
; If PC stays here, code execute an expected vector interrupt
;*****************************************************************
isr_dummy:
   JP    $

;*****************************************************************
;         GLOBAL SYMBOLS
;*****************************************************************

;*****************************************************************
;         EXTERNAL SYMBOLS
;*****************************************************************

    END
```

В този пример съм използвал прозореца **Clock** за да наглася подпрограмата **Delay_1s** да се изпълнява за приблизително около 20 000 000 системни тактови цикъла (1 секунда при

период на тактовия сигнал 0.05μс).

Обърнете внимание, че този прозорец показва тактовете само в режим на симулатор.

Друг важен прозорец, който може да използвате е **Watch**. В него може да поставяте (с влачене&пускане – drag&drop) променливите, които сте си дефинирали (например с директивата DS) и да наблюдавате как се променят стойностите им по време на изпълнение на програмата.



Expression	Value	Address
Counter1_16bits	0x00	R#020
<new watch>		

26.5 Пример 4

В този пример ще накараме и трите светодиода да светват последователно (**GREEN**, **YELLOW**, **RED**), след което да изгасват в същия ред. За целта ще използваме Таймер 0, който ще работи в непрекъснат режим. При всяко прекъсване от таймера ще се извършва едно от горните действия. Избрал съм максималния коефициент на деление на тактовия сигнал на Таймер 0. Това води до време за препълване на таймера около 419ms.

$F_{TIMER0} = F_{SYS}/128 = 20MHz/128 = 0.15625MHz$
$T_{TIMER0} = 1/ F_{TIMER0} = 1/0.15625MHz = 6.4\mu s$
$T_{0OVERFLOW} = T_{TIMER0} * T0RH:T0RL = 6.4\mu s * 65535 = 419424\mu s \approx 419ms$

```
;*****************************************************************
; File: main.asm
; Description:
;         LEDs blinking
;-----------------------------------------------------------------
;
; Modified: 27 07 2014
;*****************************************************************
; Copyright (C) All Rights Reserved.
;*****************************************************************

;*****************************************************************
;  Include files
;*****************************************************************
        INCLUDE "ez8.inc"

;*****************************************************************
; Local Constant Defitions
;*****************************************************************

; Stack starting address
NEAR_STACK      EQU    %0100

;PxADDR subregisters
NO_REG      EQU    00h
PxDD        EQU    01h
```

```
PxAF        EQU   02h
PxOC        EQU   03h
PxHDE       EQU   04h
PxSMRE      EQU   05h
PxPUE       EQU   06h
PxAFS1      EQU   07h
PxAFS2      EQU   08h
```

```
; Port A bits position
PA7    EQU   80h
PA6    EQU   40h

; Port C bits position
PC3    EQU   08h

;  T0CTL1 bits position
TEN         EQU   80h

;Timer 0 interrupt flag mask
TMR0_INT_FLAG_MASK  EQU   00100000b
```

```
;****************************************************************
; Variable Defitions
;****************************************************************
```

```
        SEGMENT    NEAR_DATA ; RData space

LedsCounter   DS      1 ; Reserve 1  bytes for LEDs counter
```

```
;****************************************************************
; Macro Defitions
;****************************************************************
YellowLedOn:        MACRO
ANDX  PAOUT, #~PA7  ; Set low lewel on pin PA7
                    ENDMACRO
```

```
YellowLedOff:       MACRO
ORX   PAOUT, #PA7   ; Set high lewel on pin PA7
                    ENDMACRO
```

```
GreenLedOn:         MACRO
ANDX  PAOUT, #~PA6  ; Set low lewel on pin PA6
                    ENDMACRO
```

```
GreenLedOff:        MACRO
ORX   PAOUT, #PA6   ; Set high lewel on pin PA6
                    ENDMACRO

RedLedOn:           MACRO
ANDX  PCOUT, #~PC3  ; Set low lewel on pin PC3
                    ENDMACRO

RedLedOff:          MACRO
ORX   PCOUT, #PC3   ; Set high lewel on pin P3
                    ENDMACRO

;*************************************************************
;Default User Flash Options
;Change values of flash_option_byte1 and flash_option_byte2 if user option
; bits are used.
;*************************************************************
        DEFINE ___flash_option_segment,SPACE=ROM, ORG=0
        SEGMENT ___flash_option_segment

flash_option_byte1: DB %FF
flash_option_byte2: DB %FF

;*************************************************************
; Vector table
;*************************************************************
; USED vectors:
        VECTOR RESET        = startup       ; RESET
        VECTOR TIMER0       = isr_tmr0      ; Timer0

;UNUSED vectors
        VECTOR WDT          = isr_dummy; Watch-Dog Timer
        VECTOR POTRAP       = isr_dummy; Primary Oscillator Fail Trap
        VECTOR WOTRAP       = isr_dummy; Watchdog Oscillator Fail Trap
        VECTOR TRAP         = isr_dummy; Illegal Instruction Trap
        VECTOR TIMER1       = isr_dummy; Timer1
        ;VECTOR TIMER0       = isr_dummy; Timer0
        VECTOR ADC          = isr_dummy; ADC
        VECTOR P0AD         = isr_dummy; PA0
        VECTOR P1AD         = isr_dummy; PA1
        VECTOR P2AD         = isr_dummy; PA2
        VECTOR P3AD         = isr_dummy; PA3
        VECTOR P4AD         = isr_dummy; PA4
```

```
        VECTOR P5AD        = isr_dummy; PA5
        VECTOR P6AD        = isr_dummy; PA6
        VECTOR P7AD        = isr_dummy; PA7
        VECTOR C3          = isr_dummy; PC3
        VECTOR C2          = isr_dummy; PC2
        VECTOR C1          = isr_dummy; PC1
        VECTOR C0          = isr_dummy; PC0

;RESERVED VECTORS are declared here to prevent linker from filling
;these locations with other code segments

        DEFINE __VECTORS_008, SPACE = ROM, ORG = %008
        SEGMENT __VECTORS_008

__VECTOR_008:    DW isr_dummy

        DEFINE __VECTORS_00E, SPACE = ROM, ORG = %00E
        SEGMENT __VECTORS_00E

__VECTOR_00E:    DW isr_dummy
__VECTOR_010:    DW isr_dummy
__VECTOR_012:    DW isr_dummy
__VECTOR_014:    DW isr_dummy

        DEFINE __VECTORS_028, SPACE = ROM, ORG = %028
        SEGMENT __VECTORS_028

__VECTOR_028:    DW isr_dummy
__VECTOR_02A:    DW isr_dummy
__VECTOR_02C:    DW isr_dummy
__VECTOR_02E:    DW isr_dummy

        DEFINE __VECTORS_038, SPACE = ROM, ORG = %038
        SEGMENT __VECTORS_038

__VECTOR_038:    DW isr_dummy

;**************************************************************
;Program starts here
;**************************************************************

        DEFINE main_user_program_segment, SPACE = ROM
        SEGMENT main_user_program_segment
```

```
startup:
   DI                                     ; Prevent interruption
   SRP    #%00                            ; Working registers 00-0f
   LDX    SPL, #LOW(NEAR_STACK)           ; Initialize stack pointer
   LDX    SPH, #HIGH(NEAR_STACK)
   CALL main
exit:
   JR exit

;*************************************************************
; Subroutine: Main routine
;*************************************************************
main:

   LD       LedsCounter, #00h ; Reset LEDs counter
   CALL     PortInit          ; Ports initialization
   CALL     Tmr0Init          ; Timer 0 initialization
   EI                         ; Enable interrupts

   JR       $                 ; Stay here
   RET

;*************************************************************
; Subroutine: PortInit
; Decription: Initializes port pins
;
;*************************************************************
PortInit:
   LDX    PAOUT, #PA7|PA6     ; Set high  lewel  on pin  PA7 and PA6
   LDX    PAADDR,#PxDD        ; Select Port A data direction subregister
   LDX    PACTL, #~(PA7|PA6); Make pin PA7 and PA6 outputs

   LDX    PCOUT, #PC3      ; Set high level on pin PC3
   LDX    PCADDR,#PxDD ; Select Port C data direction subregister
   LDX    PCCTL, #~PC3     ; Make pin PC3 output

   RET

;*************************************************************
; Subroutine: Tmr0Init
; Decription: Timer0 initialization
;
;*************************************************************
```

```
Tmr0Init:
   LDX        T0CTL0, #00000000b ; See Table 0 above (in ZDSII project)
   LDX        T0CTL1, #00111001b ; See Table 1 above (in ZDSII project)

  LDX         T0H, #00h   ; Load Timer High Byte
  LDX         T0L, #01h   ; Load Timer Low Byte
  LDX         T0RH, #FFh ; Load Timer Reload High Byte
  LDX         T0RL, #FFh  ; Load Timer Reload Low Byte

  LDX         IRQ0ENH, #00h;\ Set Timer 0 interrupt low priority
  LDX         IRQ0ENL, #20h;/

 ORX          T0CTL1, #TEN; Enable Timer 0

 RET
```

```
;*****************************************************************
; Subroutine: isr_tmr0
; Decription: Timer 0 interrupt handler
;
;*****************************************************************
isr_tmr0:
   ANDX  IRQ0, #~TMR0_INT_FLAG_MASK ; Clear the interrupt flag

   INC    LedsCounter          ; Increment LEDs counter
   CP     LedsCounter, #01h ; LedsCounter == 1?
   JP     Z, green_led_on     ; If yes, go to turn on green LED
   CP     LedsCounter, #02h ; LedsCounter == 2?
   JP     Z, yellow_led_on   ; If yes, go to turn on yellow LED
   CP     LedsCounter, #03h ; LedsCounter == 3?
   JP     Z, red_led_on        ; If yes, go to turn on red LED
   CP     LedsCounter, #04h ; LedsCounter == 4?
   JP     Z, green_led_off    ; If yes, go to turn off green LED
   CP     LedsCounter, #05h ; LedsCounter == 5?
   JP     Z, yellow_led_off   ; If yes, go to turn off yellow LED
   CP      LedsCounter, #06h ; LedsCounter == 6?
   JP      Z, red_led_off       ; If yes, go to turn off red LED
   JR      Tmr0IsrExit          ; Otherwise exit Timer 0 ISR

green_led_on:
   GreenLedOn          ; Turn on green LED
   JR   Tmr0IsrExit     ; Exit Timer 0 ISR
```

```
yellow_led_on:
   YellowLedOn         ; Turn on yellow LED
   JR   Tmr0IsrExit    ; Exit Timer 0 ISR

red_led_on:
   RedLedOn            ; Turn on red LED
   JR   Tmr0IsrExit    ; Exit Timer 0 ISR

green_led_off:
   GreenLedOff         ; Turn off green LED
   JR   Tmr0IsrExit    ; Exit Timer 0 ISR

yellow_led_off:
   YellowLedOff        ; Turn off yellow LED
   JR   Tmr0IsrExit    ; Exit Timer 0 ISR

red_led_off:
   RedLedOff                ; Turn off red LED
   LD   LedsCounter, #00h ; Reset LEDs counter

Tmr0IsrExit;
   IRET

;**********************************************************
; If PC stays here, code execute an expected vector interrupt
;**********************************************************
isr_dummy:
   JP   $

;**********************************************************
;        GLOBAL SYMBOLS
;**********************************************************

;**********************************************************
;        EXTERNAL SYMBOLS
;**********************************************************

   END
```

26.6 Пример 5

В този пример ще използваме бутона SW2, свързан към извод PA2. При натискане на бутона ще се включва Таймер 0 и светодиодите ще мигат както и в предходния пример, а при отпускане на бутона Таймер 0 ще се изключва.

Въпреки че не го виждате или усещате, при натискане на бутона се получава трептене на вътрешните пластини, при което се получава многократно затваряне и отваряне на контактите му. Ето защо след като детектирате, че бутона е натиснат (прочетете лог. 0 на извод PA2) е необходимо да изчакате поне 10ms, за да затихнат трепетенията на пластините, след което отново трябва да проверите дали нивото на извод PA2 е все още лог. 0, след което може да разрешите Таймер 0. Същото важи и когато детектирате, че бутона е отпуснат (прочетете лог. 1 на извод PA2), преди да забраните Таймер 0.

```
;*************************************************************
; File: main.asm
; Description:
;         LEDs blinking 2
;-------------------------------------------------------------------------------
;
; Modified: 30 07 2014
;*************************************************************
; Copyright (C) All Rights Reserved.
;*************************************************************

;*************************************************************
;  Include files
;*************************************************************
        INCLUDE "ez8.inc"

;*************************************************************
; Local Constant Defitions
;*************************************************************
```

```
; Stack starting address
NEAR_STACK      EQU   %0100

;PxADDR subregisters
NO_REG       EQU   00h
PxDD         EQU   01h
PxAF         EQU   02h
PxOC         EQU   03h
PxHDE        EQU   04h
PxSMRE       EQU   05h
PxPUE        EQU   06h
PxAFS1       EQU   07h
PxAFS2       EQU   08h

; Port A bits position
PA7   EQU   80h
PA6   EQU   40h

; Port C bits position
PC3   EQU   08h

;  T0CTL1 bits position
TEN          EQU   80h

;Timer 0 interrupt flag mask
TMR0_INT_FLAG_MASK  EQU   00100000b

;************************************************************
; Variable Defitions
;************************************************************

          SEGMENT    NEAR_DATA ; RData space

Counter1_16bits:      DS      2 ; Reserve 2  bytes for Counter1_16bits
Counter2_16bits:      DS      2 ; Reserve 2  bytes for Counter2_16bits
LedsCounter           DS      1 ; Reserve 1  bytes for LEDs counter

;************************************************************
; Macro Defitions
;************************************************************
YellowLedOn:          MACRO
ANDX  PAOUT, #~PA7  ; Set low lewel on pin PA7
                      ENDMACRO
```

```
YellowLedOff:          MACRO
ORX   PAOUT, #PA7   ; Set high lewel on pin PA7
                       ENDMACRO
```

```
GreenLedOn:            MACRO
ANDX  PAOUT, #~PA6  ; Set low lewel on pin PA6
                       ENDMACRO
```

```
GreenLedOff:           MACRO
ORX   PAOUT, #PA6   ; Set high lewel on pin PA6
                       ENDMACRO
```

```
RedLedOn:              MACRO
ANDX  PCOUT, #~PC3  ; Set low lewel on pin PC3
                       ENDMACRO
RedLedOff:             MACRO
ORX   PCOUT, #PC3   ; Set high lewel on pin P3
                       ENDMACRO
```

```
EnableTimer0:          MACRO
ORX          T0CTL1, #TEN; Enable Timer 0
                       ENDMACRO
```

```
DisableTimer0:         MACRO
ANDX         T0CTL1, #~TEN; Disable Timer 0
                       ENDMACRO
```

```
;*************************************************************
;Default User Flash Options
;Change values of flash_option_byte1 and flash_option_byte2 if user option
; bits are used.
;*************************************************************
        DEFINE ___flash_option_segment,SPACE=ROM, ORG=0
        SEGMENT ___flash_option_segment

flash_option_byte1: DB %FF
flash_option_byte2: DB %FF

;*************************************************************
; Vector table
;*************************************************************
; USED vectors:
        VECTOR RESET      = startup      ; RESET
```

```
        VECTOR TIMER0      = isr_tmr0     ; Timer0

;UNUSED vectors
        VECTOR WDT         = isr_dummy; Watch-Dog Timer
        VECTOR POTRAP      = isr_dummy; Primary Oscillator Fail Trap
        VECTOR WOTRAP      = isr_dummy; Watchdog Oscillator Fail Trap
        VECTOR TRAP        = isr_dummy; Illegal Instruction Trap
        VECTOR TIMER1      = isr_dummy; Timer1
        ;VECTOR TIMER0     = isr_dummy; Timer0
        VECTOR ADC         = isr_dummy; ADC
        VECTOR P0AD        = isr_dummy; PA0
        VECTOR P1AD        = isr_dummy; PA1
        VECTOR P2AD        = isr_dummy; PA2
        VECTOR P3AD        = isr_dummy; PA3
        VECTOR P4AD        = isr_dummy; PA4
        VECTOR P5AD        = isr_dummy; PA5
        VECTOR P6AD        = isr_dummy; PA6
        VECTOR P7AD        = isr_dummy; PA7
        VECTOR C3          = isr_dummy; PC3
        VECTOR C2          = isr_dummy; PC2
        VECTOR C1          = isr_dummy; PC1
        VECTOR C0          = isr_dummy; PC0

;RESERVED VECTORS are declared here to prevent linker from filling
;these locations with other code segments

        DEFINE __VECTORS_008, SPACE = ROM, ORG = %008
        SEGMENT __VECTORS_008

__VECTOR_008:    DW isr_dummy

        DEFINE __VECTORS_00E, SPACE = ROM, ORG = %00E
        SEGMENT __VECTORS_00E

__VECTOR_00E:    DW isr_dummy
__VECTOR_010:    DW isr_dummy
__VECTOR_012:    DW isr_dummy
__VECTOR_014:    DW isr_dummy

        DEFINE __VECTORS_028, SPACE = ROM, ORG = %028
        SEGMENT __VECTORS_028

__VECTOR_028:    DW isr_dummy
```

```
__VECTOR_02A:    DW isr_dummy
__VECTOR_02C:    DW isr_dummy
__VECTOR_02E:    DW isr_dummy

        DEFINE __VECTORS_038, SPACE = ROM, ORG = %038
        SEGMENT __VECTORS_038

__VECTOR_038:    DW isr_dummy

;*************************************************************
;Program starts here
;*************************************************************
        DEFINE main_user_program_segment, SPACE = ROM
        SEGMENT main_user_program_segment

startup:
  DI                                ; Prevent interruption
  SRP   #%00                        ; Working registers 00-0f
  LDX   SPL, #LOW(NEAR_STACK)       ; Initialize stack pointer
  LDX   SPH, #HIGH(NEAR_STACK)
  CALL main
exit:
  JR exit

;*************************************************************
; Subroutine: Main routine
;*************************************************************
main:

  LD       LedsCounter, #00h ; Reset LEDs counter
  CALL     PortInit          ; Ports initialization
  CALL     Tmr0Init          ; Timer 0 initialization
  EI                         ; Enable interrupts

check:;Check for pressed button
  LDX    R0, PAIN            ; Read Port A
  TM     R0, #00000100b      ; Check the button (PA2)
  JR     NZ, check           ; If the button is not pressed, check again
  CALL   Delay_10ms          ; Wait 10ms
  LDX    R0, PAIN            ; Read Port A again
  TM     R0, #00000100b      ; Check the button (PA2) again
  JR     NZ, check           ; If the button is not pressed, check again
  EnableTimer0               ; Otherwise enable Timer 0
```

```
release: ;Check for released button
   LDX     R0, PAIN          ; Read Port A
   TM      R0, #00000100b  ; Check the button (PA2)
   JR      Z, release        ; If the button is still pressed, check again
   CALL  Delay_10ms        ; Wait 10ms
   DisableTimer0           ; Disable Timer 0
   JR      NZ, check         ; Repeat the cycle
   RET
;***********************************************************
; Subroutine: PortInit
; Decription: Initializes port pins
;
;***********************************************************
PortInit:
   LDX    PAOUT, #PA7|PA6    ; Set high  lewel  on pins  PA7 and PA6
   LDX    PAADDR,#PxDD       ; Select Port A data direction subregister
   LDX    PACTL, #~(PA7|PA6); Make pins PA7 and PA6 outputs

   LDX    PCOUT, #PC3      ; Set high level on pin PC3
   LDX    PCADDR,#PxDD ; Select Port C data direction subregister
   LDX    PCCTL, #~PC3     ; Make pin PC3 output

   RET

;***********************************************************
; Subroutine: Tmr0Init
; Decription: Timer0 initialization
;
;***********************************************************
Tmr0Init:
  LDX    T0CTL0, #00000000b ; See Table 0 above (in ZDII project)
  LDX    T0CTL1, #00111001b ; See Table 1 above (in ZDII project)

  LDX    T0H, #00h    ; Load Timer High Byte
  LDX    T0L, #01h     ; Load Timer Low Byte
  LDX    T0RH, #FFh ; Load Timer Reload High Byte
  LDX    T0RL, #FFh  ; Load Timer Reload Low Byte

  LDX    IRQ0ENH, #00h;\ Set Timer 0 interrupt low priority
  LDX    IRQ0ENL, #20h;/

  RET
```

```
;*****************************************************************
; Subroutine: Delay_10ms
; Decription: Provide delay 10ms (200 000 clock cycles)
;
;*****************************************************************
Delay_10ms
   LDX   Counter2_16bits,   #00h   ; Load counter2 high byte
   LDX   Counter2_16bits+1, #02h ; Load counter2 low byte
reload_counter1:
   LDX   Counter1_16bits,   #27h   ; Load counter1 high byte
   LDX   Counter1_16bits+1, #10h ; Load counter1 low byte

delay_loop:
   DECW   Counter1_16bits  ; Decrement Counter1 by 1
   JR         NZ, delay_loop  ; Repeat if Counter1 is not zero

   DECW   Counter2_16bits         ; Decrement Counter2 by 1
   JR         NZ, reload_counter1   ; Exit if Counter2 value is zero

   RET

;*****************************************************************
; Subroutine: isr_tmr0
; Decription: Timer 0 interrupt handler
;
;*****************************************************************
isr_tmr0:
   ANDX  IRQ0, #~TMR0_INT_FLAG_MASK ; Clear the interrupt flag

   INC    LedsCounter          ; Increment LEDs counter
   CP     LedsCounter, #01h ; LedsCounter == 1?
   JP     Z, green_led_on      ; If yes, go to turn on green LED
   CP     LedsCounter, #02h ; LedsCounter == 2?
   JP     Z, yellow_led_on    ; If yes, go to turn on yellow LED
   CP     LedsCounter, #03h ; LedsCounter == 3?
   JP     Z, red_led_on          ; If yes, go to turn on red LED
   CP     LedsCounter, #04h ; LedsCounter == 4?
   JP     Z, green_led_off      ; If yes, go to turn off green LED
   CP     LedsCounter, #05h ; LedsCounter == 5?
   JP     Z, yellow_led_off    ; If yes, go to turn off yellow LED
   CP      LedsCounter, #06h ; LedsCounter == 6?
   JP      Z, red_led_off         ; If yes, go to turn off red LED
   JR      Tmr0IsrExit            ; Otherwise exit Timer 0 ISR
```

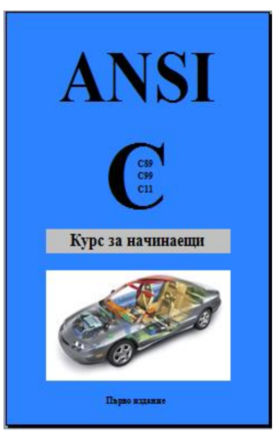
```
green_led_on:
   GreenLedOn          ; Turn on green LED
   JR   Tmr0IsrExit    ; Exit Timer 0 ISR

yellow_led_on:
   YellowLedOn         ; Turn on yellow LED
   JR   Tmr0IsrExit    ; Exit Timer 0 ISR

red_led_on:
   RedLedOn            ; Turn on red LED
   JR   Tmr0IsrExit    ; Exit Timer 0 ISR

green_led_off:
   GreenLedOff         ; Turn off green LED
   JR   Tmr0IsrExit    ; Exit Timer 0 ISR

yellow_led_off:
   YellowLedOff        ; Turn off yellow LED
   JR   Tmr0IsrExit    ; Exit Timer 0 ISR

red_led_off:
   RedLedOff                ; Turn off red LED
   LD  LedsCounter, #00h ; Reset LEDs counter

Tmr0IsrExit;
   IRET

;*************************************************************
; If PC stays here, code execute an expected vector interrupt
;*************************************************************
isr_dummy:
   JP   $

;*************************************************************
;        GLOBAL SYMBOLS
;*************************************************************

;*************************************************************
;        EXTERNAL SYMBOLS
;*************************************************************

   END
```

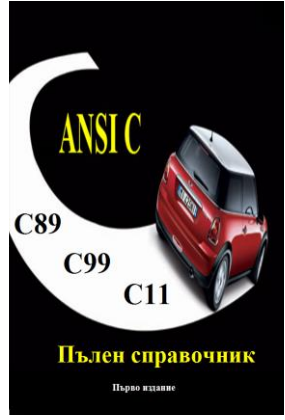
С този пример ще приключим програмирането на асемблер. Въпреки че примерите бяха кратки, съм убеден, че вече имате ясна представа как работи микроконтролера на ниско ниво.

Заключение

Поздравления🙂! Вече имате представа какво е микроконтролер и как работи. Придобитите знания ще Ви помогнат да се ориентирате по-лесно в различните процесорни архитектури.

Ако не сте запознати с езика С, следващата стъпка е да го направите. За целта съм предвидил книгите **ANSI C. Курс за начинаещи** и **ANSI C. Пълен справочник**.



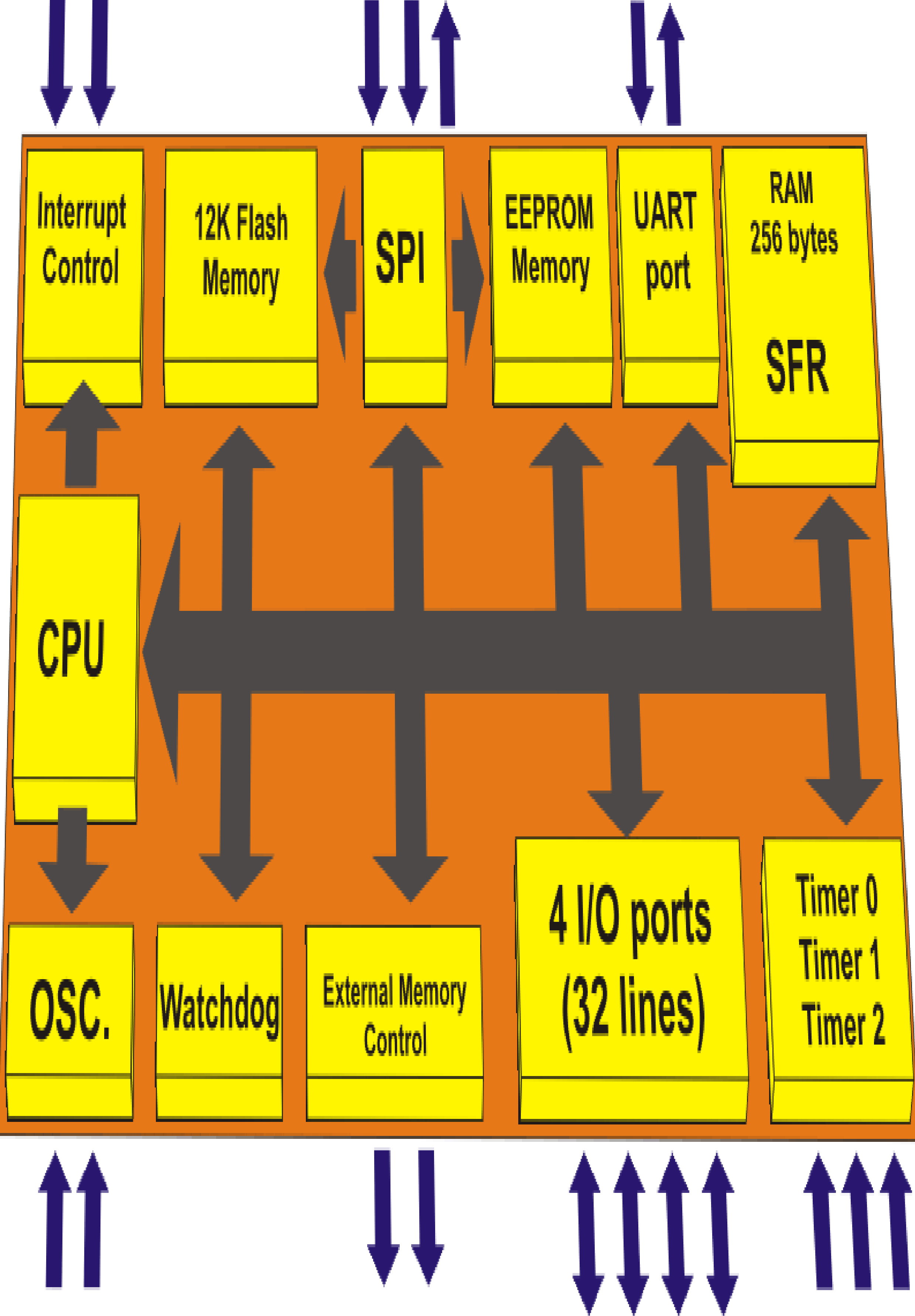


За тези, които са минали вече тази стъпка следващото ниво е да приложат знанията си на практика. За тази цел съм предвидил книгата **Програмиране на микроконтролери на езика С**.

Тази книга е продължение на настоящата. В нея ще Ви запозная с особеностите на C компилатора за Z8Encore! микроконтролерите и ще разгледаме различни примери на C.

Очаквайте следващата книга от поредицата.

Тази книга е продължение на настоящата книга. Тя комбинира наученото от предходните две стъпки (езикът С и принципа на действие на микроконтролерите) и ще Ви научи как да използвате езика С в програмирането на микроконтролери.



Interrupt Control
12K Flash Memory
SPI
EEPROM Memory
UART port
RAM
256 bytes
SFR
CPU
OSC.
Watchdog
External Memory Control
4 I/O ports
(32 lines)
Timer 0
Timer 1
Timer 2