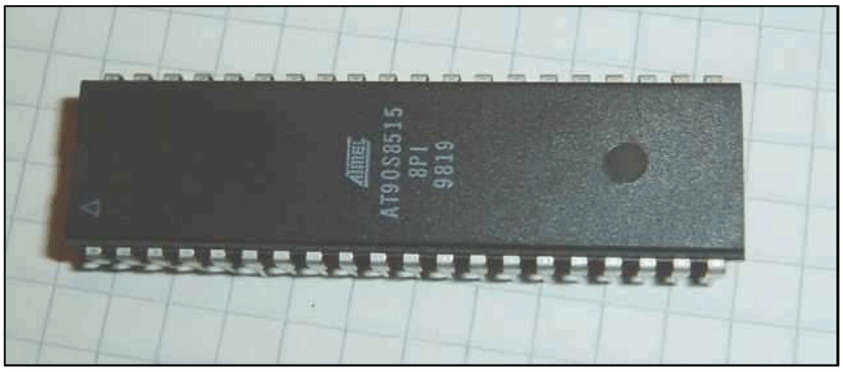


مقدمه‌ای بر زبان اسمبلی ریزپردازنده‌های AVR

by

Gerhard Schmidt

http://www.avr-asm-tutorial.net

December 2003

مترجم:

محمد مهدی‌پور

mo.mahdipour@gmail.com

تیر ۸۵ (جولای ۲۰۰۶)


© SAMAVi

فهرست

چرا از زبان اسمبلی استفاده کنیم؟

اسمبلی یا زبان‌های دیگر؟ سؤال این است که چرا باید زبان دیگری را یاد بگیرید، در حالی که قبلاً زبان‌های برنامه‌نویسی دیگر را فراگرفته‌اید؟ بهترین دلیل: هنگامی‌که شما در فرانسه زندگی می‌کنید قادر به گذران زندگی از طریق صحبت‌کردن به زبان انگلیسی هستید، اما در آنجا هرگز حس بودن در وطن را نمی‌کنید، و زندگی دشوار باقی می‌ماند. با این وضعیت می‌توانید زندگی را بگذرانید، اما به شکلی نسبتاً نامناسب. برای کارهای ضروری نیز باید زبان آن کشور را به‌کار ببرید.

آسان و مختصر

دستورات اسمبلی هر یک به طور مستقل به دستورات اجرایی ماشین ترجمه می‌شوند. پردازنده تنها لازم است آن چیزی را که شما از آن می‌خواهید و برای انجام وظیفه مورد نظر ضروری است اجرا کند. خبری از حلقه‌های اضافی و کارهای غیرضروری که حجم کد تولید شده را افزایش می‌دهند نیست. اگر حافظه ذخیره برنامهٔ شما کوچک و محدود بوده و مجبور به بهینه‌سازی برنامه برای گنجاندن آن در حافظه هستید، اسمبلی انتخاب اول شماست. اشکالزدایی برنامه‌های کوچک‌تر آسان تر بوده و هر مرحله از آن قابل فهم است.

تند و سریع

از آنجا که فقط مراحل ضروری کد اجرا می‌شوند، برنامه‌های اسمبلی تا حد امکان سریع هستند. مدت زمان اجرای هر مرحله مشخص است. برنامه‌های حساس به زمانی که باید با دقت عالی کار بکنند، مثلاً اندازه‌گیری زمان بدون استفاده از تایمر سخت‌افزاری، می‌بایست به زبان اسمبلی نوشته شوند. اگر سرعت اجرای برنامه برایتان مهم نیست و سپری شدن ۹۹٪ از وقت پردازنده در حالت‌های انتظار و بیکار برایتان اهمیتی ندارد، می‌توانید هر زبان دیگری را که می‌خواهید برای برنامه‌نویسی انتخاب کنید.

یادگیری اسمبلی آسان است

زبان اسمبلی خیلی پیچیده نیست و یادگیری آن سخت‌تر از زبان‌های دیگر نیست. فراگیری زبان اسمبلی مربوط به هر نوع سخت‌افزاری، شما را با مفاهیم پایه‌ای دیگر نسخه‌های زبان اسمبلی آشنا می‌کند. افزودن قواعد بعدی نیز آسان است. کد اسمبلی اولیه چندان جذاب و مناسب از آب در نمی‌آید؛ با افزودن هر ۱۰۰ خط کد، برنامه بهتر به نظر می‌رسد. برنامه‌های کامل احتیاج به چند هزار خط کد آزمایشی دارند و بهینه‌سازی آنها نیز کار زیادی می‌طلبد. با توجه به اینکه برخی قابلیت‌ها وابسته به سخت‌افزار هستند، نوشتن کد بهینه شده نیاز به آشنایی با مفاهیم سخت‌افزاری و قواعد دستوری دارد. مراحل اولیه یادگیری هر زبانی مشکل است. پس از چند هفته برنامه‌نویسی اگر به اولین کدهایی که نوشته‌اید نگاهی بیاندازید خواهید خندید. برخی دستورات اسمبلی نیاز به چند ماه تجربه دارند.

سری AT90Sxxxx بهترین انتخاب برای یادگیری زبان اسمبلی هستند

برنامه‌های اسمبلی قدری احمقانه هستند: تراشه هر چیزی را که به آن بگویید اجرا می‌کند و از شما نمی‌پرسد که آیا مطمئنید که می‌خواهید روی داده‌های قبلی بازنویسی کنید. تمام کارهای حفاظتی و کنترلی باید توسط خود شما برنامه‌نویسی شوند؛ تراشه فقط چیزی را که به آن گفته شده است انجام می‌دهد. هیچ بخشی به شما هشدار نمی‌دهد مگر آنکه قبلاً خودتان برنامه آن را نوشته باشید.

اشکالزدایی خطاهای پایه ای طراحی به همان پیچیدگی موجود در زبان‌های کامپیوتری دیگر است. اما آزمایش برنا مـه‌هـا روی تراشه‌های ATMEL بسیار ساده است. اگر برنامه شما کاری را که از آن انتظار دارید انجام نمی‌دهد می‌توانیـد بـه سادگی با افزودن چند خط اشکالزدایی به کد و برنامه‌ریزی دوبارهٔ تراشه آن را آزمایش کنید. دیگر زمان آن رسیده که بـا برنامه‌ریزهای EPROM، با لامپ‌های UV مورد استفاده برای پاک‌کردن برنامه‌ها، و با پین‌هایی که پس از چندین مرتبـه جداکردن و قراردادن روی سوکت خراب می‌شوند خداحافظی کنید.

تغییرات در یک چشم به هم زدن کامپایل شده و به سرعت برنامه‌ریزی می‌شوند، و می‌تـوان آن‌هـا را در محـیط Studio شبیه‌سازی کرده و یا روی مدار مورد بررسی قرار داد. دیگر نیازی به خارج کردن پین‌ها از مدار نیست، و دیگر هیچ لامپ UV وجود ندارد که در لحظات حساس، هنگامی‌که مشغول آزمایش یک ایدهٔ عالی برای رفع خطای برنامه‌تـان هـستید از کار بیفتد.

آن را بیازمایید!

در گذر از مراحل اولیه صبور باشید! اگر با زبان (سطح بالای) دیگری آشنایی دارید برای اولین بار به آن فکـر نکنیـد. در پشت هر زبان اسمبلی مفاهیم سخت‌افزاری خاصی وجود دارد. اکثر قابلیت‌های ویژهٔ دیگر زبان‌های کامپیوتری هیچ معنایی در اسمبلر ندارند.

یادگیری پنج دستور اولیه قدری مشکل است، ولی پس از آن سرعت یادگیری شما افزایش می‌یابد. پس از نوشتن نخستین کدهای اسمبلی اگر به لیست مجموعه دستورات نگاهی بیاندازید از مشابه بودن دیگر دستورات متعجب خواهید شد.

برای شروع سعی نکنید یک برنامه بزرگ و بسیار پیچیده بنویسید. این کار در هیچ زبان برنامه‌نویسی مناسب و مفید نبـوده و فقط موجب ناکامی‌می‌شود.

به زیرروال‌هایی که نوشته‌اید توضیحاتی اضافه کرده و پس از اشکالزدایی آنها را در محل ویژه‌ای ذخیـره کنیـد: پـس از مدت کوتاهی به آنها نیاز خواهید داشت.

موفق باشید!

سخت‌افزار مورد نیاز برای برنامه‌ریزی ریزپردازنده‌های AVR

فراگیری اسمبلی نیاز به تجهیزات سخت‌افزاری ساده‌ای برای آزمایش برنامه‌هایتان دارد تا صحت اجرای آنها را در عمل مشاهده کنید.

این بخش دو شماتیک ساده را معرفی می‌کند که شما را قادر به ساخت دستی سخت‌افزار مورد نیاز کرده و نکات ضروری را متذکر می‌شود. ساخت این سخت‌افزار واقعاً آسان است. برای آزمایش اولین مراحل برنامه‌نویسی، ساده‌تر از آن چیزی را نمی‌شناسم. اگر علاقه به انجام آزمایش‌های بیشتری دارید، مقداری فضای خالی برای توسعه‌های بعدی روی بورد آزمایشی نگه دارید.

اگر از لحیم کاری خوشتان نمی‌آید می‌توانید یک بورد آماده خریداری کنید. مشخصات بوردهای موجود در انتهای این بخش ذکر شده است.

رابط ISP برای خانواده پردازنده‌های AVR

پیش از آغاز به کار لازم است نکات مهمی را درباره ساز و کار برنامه‌ریزی سریال خانواده AVR یاد بگیریم. نیازی به سه ولتاژ متفاوت برای برنامه‌ریزی و خواندن حافظه فلش AVR ندارید. نیازی به یک ریزپردازنده دیگر برای برنامه‌ریزی AVR ندارید. نیازی به ۱۰ خط I/O برای ارتباط با تراشه ندارید؛ و نیز برای برنامه‌ریزی AVR مجبور به جداکردن پردازنده از بورد آزمایشی نیستید. این کار حتی ساده‌تر از این هم است.

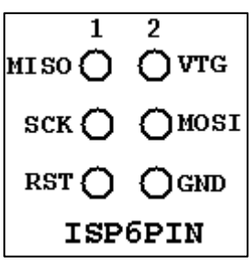
تمام کارها توسط یک رابط درونی تراشه AVR انجام می‌شود که شما را قادر به خواندن و نوشتن محتویات حافظه فلش برنامه و EEPROM داخلی می‌کند. این رابط به صورت سریال عمل کرده و نیاز به سه خط سیگنال دارد:

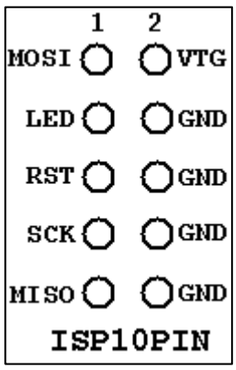
- SCK : سیگنال ساعت (Clock) که بیت‌های موردنظر برای نوشته شدن در حافظه را به داخل یک Shift-Register داخلی شیفت داده و بیت‌های موردنظر برای خواندن را از یک Shift-Register داخلی دیگر به طرف بیرون شیفت می‌دهد،
- MOSI : سیگنال داده که بیت‌ها را برای نوشته شدن در AVR ارسال می‌کند،
- MISO : سیگنال داده که بیت‌های خوانده شده از AVR را دریافت می‌کند.

این سه پایهٔ سیگنال تنها زمانی به عنوان برنامه‌ریز عمل می‌کنند که پایه RESET (که گاهی RST یا Restart نیز خوانده می‌شود) به صفر (GND) متصل باشد. در غیراینصورت، در حالت عادی این پایه‌ها خطوط ورودی- خروجی قابل برنامه‌ریزی مشابه دیگر خطوط ورودی- خروجی موجود در AVR هستند.

اگر می‌خواهید این پایه‌ها را برای مقاصد دیگری به کار برده، و نیز از آنها برای برنامه‌ریزی AVR استفاده کنید باید مراقب باشید که این دو حالت با هم تداخل نداشته باشند. معمولاً این دو را با استفاده از مقاومت‌ها و یا توسط یک مالتی پلکسر از هم جدا می‌کنند، اما نوع تمهیدات ضروری به نحوهٔ استفاده شما از این پایه‌ها در حالت عادی بستگی دارد. به هر حال بهتر است از این پایه‌ها به طور انحصاری برای برنامه‌ریزی-درمدار سیستم (In-System-Programming) استفاده کنید.

در روش برنامه‌ریزی-درمدار سیستم (In-System-Programming) توصیه می‌شود که سخت‌افزار برنامه‌ریز از منبع ولتاژ سیستم شما تغذیه شود. این کار ساده‌ایست و نیاز به دو خط اضافی بین برنامه‌ریز و بورد AVR دارد. GND همان زمین عمومی و VTG (Target Voltage) ولتاژ ورودی (معمولاً ۵ ولت) است. با انجام این کار مجموعاً ٦ خط بین برنامه‌ریز و بورد AVR وجود خواهد داشت. اتصال ISP6 نهایی، براساس تعریف شرکت ATMEL، در سمت چپ نشان داده شده است.



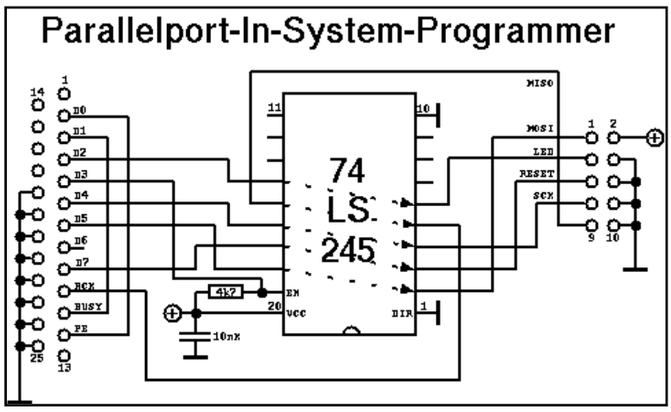


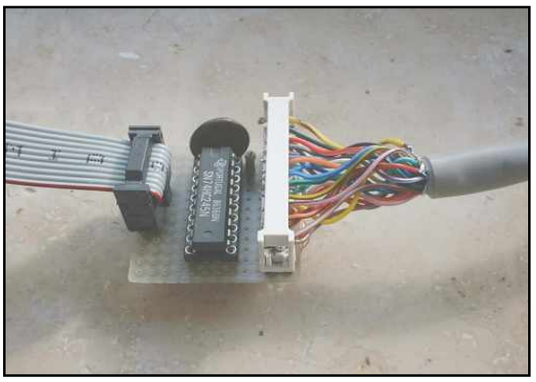
استانداردها همواره دارای استانداردهایی جانبی هستند که پیش از آنها به کار برده می‌شدند. این حقیقت، اساس تشکیل‌دهنده سازگاری در صنعت است. در مورد اخیر، استاندارد جانبی با عنوان ISP10 طراحی و روی بورد STK200 به کار گرفته شده است. این استاندارد هنوز هم یک استاندارد بسیار متداول بوده، و حتی در STK500 هم از آن استفاده شده است. ISP10 یک خط سیگنال اضافی برای روشن کردن یک دیود نورانی (LED) قرمز دارد. روشن بودن این دیود نورانی نشان‌دهندهٔ مشغول بودن برنامه‌ریز است(یک ایده خوب). تنها کافی است که دیود نورانی را به یک مقاومت متصل کرده و آن را به ولتاژ ورودی مثبت ببندید.

برنامه‌ریز AVR برای پورت موازی کامپیوتر شخصی

اکنون هویهٔ خود را داغ کرده و برنامه‌ریز خود را بسازید. این یک مدار کاملاً ساده است که با قطعات استاندارد موجود در جعبه ابزار آزمایش‌هایتان به راحتی قابل ساخت است.

بله، این تمام چیزی است که برای برنامه‌ریزی یک AVR نیاز دارید. رابط ۲۵ پایه به پورت موازی کامپیوتر شما متصل شده و ISP ده پایه به بورد آزمایشی AVR متصل می‌شود. اگر در جعبه قطعات، قطعه 74LS245 موجود نیست می‌توانید از یک 74HC245 یا یک 74HC244/74LS244 (با تغییر بعضی پایه‌ها و سیگنال‌ها) استفاده کنید. اگر از مدل HC استفاده می‌کنید فراموش نکنید که ورودی‌های استفاده نشده را به GND یا ولتاژ تغذیه متصل کنید، در غیراینصورت ممکن است به دلیل وجود ظرفیت‌های خازنیِ بسیار کوچک بین بافرها، نویز زیادی تولید شود.[1]





عملیات برنامه‌ریزی توسط نرم‌افزار ISP انجام می‌شود. این نرم‌افزار در صفحه دانلود نرم‌افزار شرکت ATMEL (بر روی وب) موجود است.

بورد آزمایشی با AT90S2313

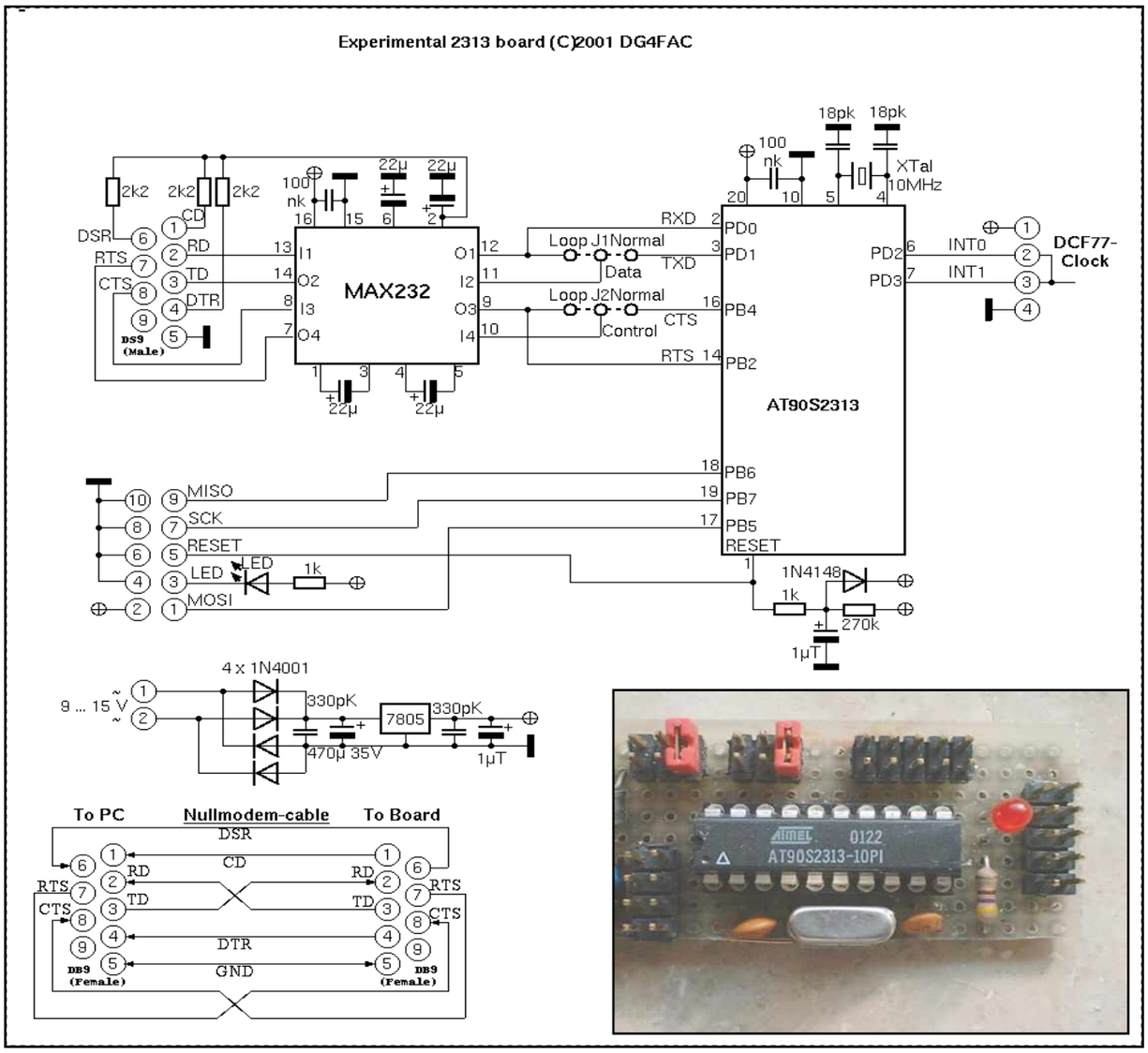
برای مقاصد آزمایشی از یک AT90S2313 روی بورد آزمایش استفاده می‌کنیم. شماتیک مدار دارای قسمت‌های زیر است:

- یک منبع تغذیه کوچک برای اتصال به ترانسفورماتور AC و رگولاتور ولتاژ 5V/1A،
- یک مولد پالس ساعت XTAL (در اینجا ۱۰ Mcs/s (ده مگاسیکل بر ثانیه)، تمام فرکانس‌های پایین‌تر از فرکانس حداکثرِ تراشهٔ 2313 نیز به طور صحیح کار می‌کنند)،
- قطعات لازم برای ریست مطمئن هنگام قطع و وصل ولتاژ ورودی،
- رابط برنامه‌ریز ISP (در اینجا ISP10PIN).

۱- مترجم: اگر یک پین ورودی از نوع MOS را در حالت Open رها کنید، امپدانس ورودی آن در حدود چند ترا اهم است. سیگنال‌های عبوری از پین‌های مجاور، که قطع و وصل می‌شوند، به دلیل وجود یک ظرفیت خازنی بسیار کم بین دو پین مجاور (مثلاً ۰/۵ pF)، به داخل این پین کوپلاژ شده و درایور در حالت نامنظمی قطع و وصل می‌شود.

تمام آنچه که نیاز دارید همین‌ها هستند. قطعات جانبی اضافی را می‌توانید به پایه‌های ورودی- خروجی استفاده نشدهٔ 2313 وصل کنید.

ساده‌ترین وسیله خروجی می‌تواند یک دیود نورانی باشد که از طریق یک مقاومت به ولتاژ ورودی مثبت متصل شده است. به این طریق می‌توانید نوشتن اولین برنامه اسمبلی خود را که دیود نورانی را روشن و خاموش می‌کند آغاز کنید.



بوردهای تجاری آماده برای برنامه‌ریزی ریزپردازنده‌های خانواده AVR

اگر علاقه ای به سخت‌افزار دست ساز ندارید، و مقداری پول اضافی دارید که نمی‌دانید با آن چه کار کنید، می‌توانید یک بورد برنامه‌ریز تجاری خریداری کنید. STK500 (به طور مثال از شرکت ATMEL) به راحتی قابل تهیه است. این وسیله دارای مشخصات سخت‌افزاری زیر می‌باشد:

- سوکت‌هایی برای برنامه‌ریزی اکثر مدل‌های AVR،
- برنامه‌ریز سریال و موازی،
- رابط‌های ISP6PIN و ISP10PIN برای انجام برنامه‌ریزی-درمدار سیستم (In-System-Programming) در خارج از بورد،
- اسیلاتور قابل برنامه‌ریزی و منابع ولتاژ،
- سوییچ‌های ورودی(plug-in switches) و دیودهای نورانی،
- رابط RS232 برای اتصال به کامپیوتر شخصی (UART)،

- حافظه Flash-EEPROM سریال،
- دسترسی به تمام پورت‌ها توسط یک رابط ۱۰ پایه.

آزمایشات را می‌توانید با استفاده از AT90S8515 که قبلاً تهیه کرده‌اید انجام دهید. بورد از طریق پورت سریال (COMx) به کامپیوتر شخصی متصل شده و توسط جدیدترین نسخه‌های AVR Studio که در صفحه وب ATMEL موجودند کنترل می‌شود. این وسیله تمام نیازهای سخت‌افزاری را که یک مبتدی باید داشته باشد فراهم می‌کند.

ابزارهای برنامه‌نویسی اسمبلی AVR

این بخش اطلاعاتی درباره ابزارهای لازم برای برنامه‌ریزی ریزپردازنده‌های AVR با بورد STK200 ارائه می‌دهد. برنامه‌ریزی با STK500 بسیار متفاوت بوده و جزئیات بیشتر آن در بخش برنامه AVR Studio نشان داده شده است. توجه داشته باشید که نرم‌افزار قدیمی مورد استفاده برای STK200 دیگر پشتیبانی نمی‌شود.

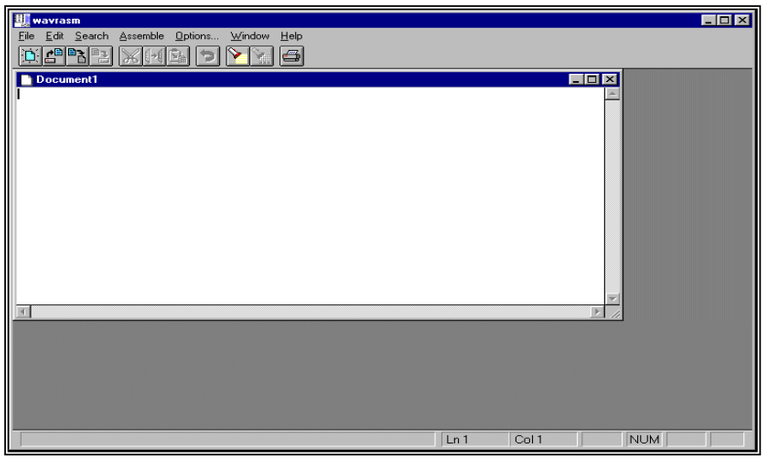
چهار برنامه اصلی برای برنامه‌نویسی به زبان اسمبلی ضروری هستند. این ابزارها عبارتند از:

- ویرایشگر،
- برنامه اسمبلر،
- رابط برنامه‌ریزی تراشه، و
- شبیه‌ساز.

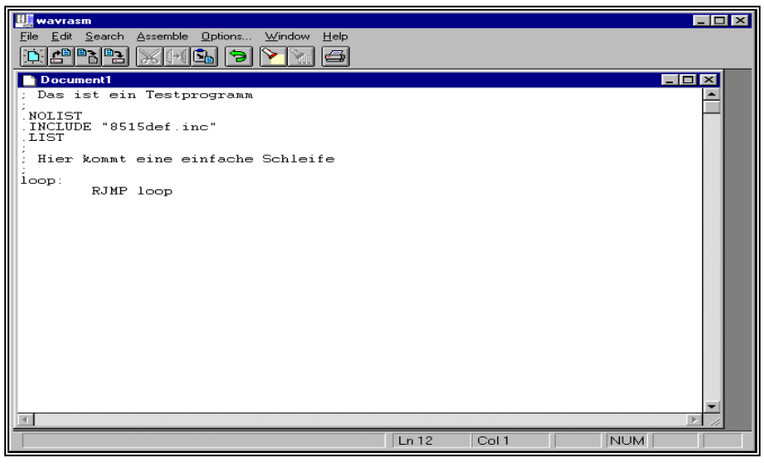
نرم‌افزارهای مورد نیاز تحت کپی رایت ATMEL بوده (©ATMEL) و در صفحه وب ATMEL قابل دانلود هستند. تصاویر این بخش تحت کپی رایت ATMEL می‌باشند. لازم به ذکر است که نسخه‌های متفاوتی از این نرم‌افزارها وجود دارند و بنابراین برخی از تصاویر بسته به نسخه مورد استفاده ممکن است متفاوت باشد. ظاهر بعضی پنجره‌ها یا منوها در نسخه‌های مختلف، متفاوت است. اما توابع اصلی اساساً بدون تغییر باقی می‌مانند. بهتر است به راهنمای برنامه‌نویسان نرم‌افزار مربوطه مراجعه نمایید؛ این بخش تنها یک دید کلی برای مبتدیان فراهم می‌کند و برای استفادهٔ برنامه‌نویسان پیشرفته اسمبلی نوشته نشده است.

ویرایشگر

برنامه‌های اسمبلی به کمک یک ویرایشگر نوشته می‌شوند. ویرایشگر تنها باید قادر به ایجاد و ویرایش فایل‌های متنی ASCII باشد. بنابراین، اصولاً هر ویرایشگر ساده‌ای این کار را انجام می‌دهد. من استفاده از یک ویرایشگر پیشرفته مانند WAVRASM از شرکت ATMEL (©) یا ویرایشگر نوشته شده توسط Tan Silliksaar را توصیه می‌کنم (تصاویر زیر را ببینید).



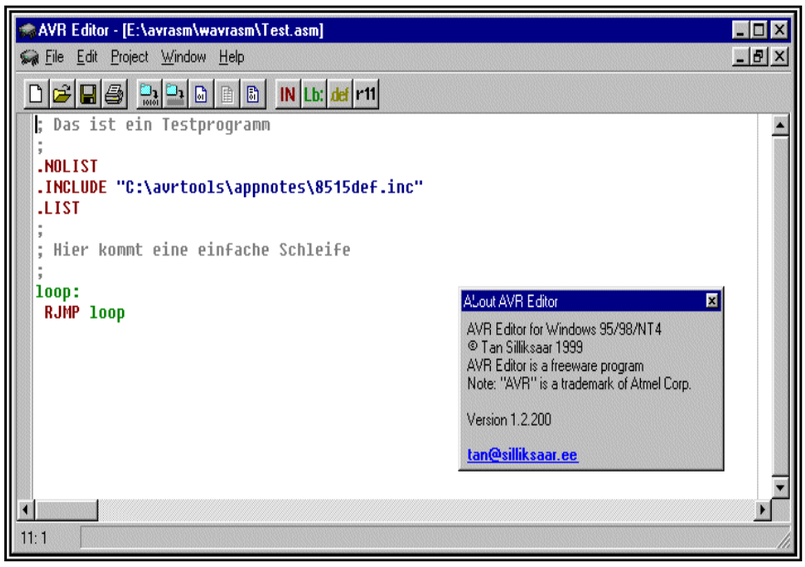
یک برنامه اسمبلی نوشته شده در WAVRASM© به این شکل است. تنها کافی است WAVRASM© را نصب کرده و برنامه را اجرا کنید:



اکنون راهنماها و دستورات اسمبلی خود را همراه با توضیحاتی (که با ; شروع می‌شوند) در پنجره ویرایش WAVRASM می‌نویسیم. نتیجه به این شکل خواهد بود:

حالا با استفاده از منوی File، متن برنامه را با نام something.asm در پوشه موردنظر ذخیره می‌کنیم. اکنون برنامه اسمبلی کامل است.

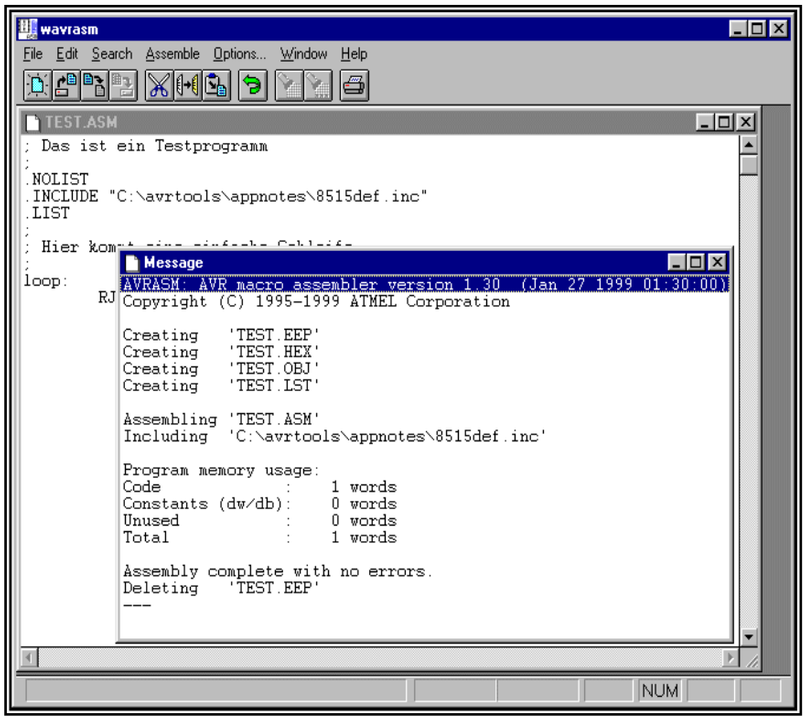
اگر می‌خواهید برنامه‌هایتان را به صورت حرفه‌ای ویرایش نمایید می‌توانید از ویرایشگر بسیار خوبی که توسط Tan Silliksaar نوشته شده است استفاده کنید. ابزارهای این ویرایشگر مخصوص برنامه‌نویسی AVR طراحی شده و به طور رایگان در صفحه وب Tan موجود است. در این ویرایشگر برنامه ما به شکل زیر می‌باشد:



ویرایشگر دستورات اسمبلی را به طور خودکار تشخیص داده و از رنگ‌های متفاوت (syntax highlighting – متمایز کردن بخش‌های مختلف یک دستور) برای نمایش ثابت‌ها و خطاهای تایپی در دستورات (که به رنگ سیاه هستند) استفاده می‌کند. ذخیره این کد در یک فایل asm. تقریباً همان فایل متنی قبلی را نتیجه می‌دهد.

اسمبلر

اکنون باید کد نوشته شده را به فرم زبان ماشینی که برای تراشه AVR قابل فهم است ترجمه کنیم. این عمل، اسمبل‌کردن نامیده شده و به معنی کنار هم قرار دادن کلمات کدِ مربوط به دستورات است. اگر از WAVRASM© استفاده می‌کنید فقط روی منوی Assemble کلیک کنید. نتیجه در شکل زیر نشان داده شده است:



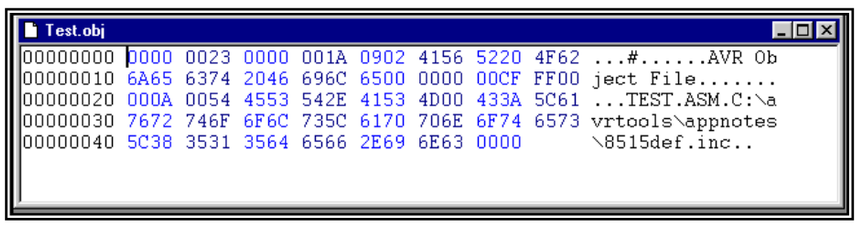
اسمبلر اتمام عملیات ترجمه را بدون خطا گزارش می‌دهد. اگر خطایی موجود باشد اطلاع داده می‌شود. حاصل ترجمه، یک کلمه کد بود که نتیجهٔ استفاده از فرمان به کار برده شده (RJMP) است. اسمبل‌کردن این فایلِ متنی اسمبلی، چهار فایل مختلف تولید کرده است (که همه آنها در اینجا به کار نمی‌آیند).

اولین فایل از این چهار فایل جدید،TEST.EEP، محتوی بخشی است که باید به حافظه EEPROM داخل AVR نوشته شود. در اینجا این فایل چندان مورد توجه نیست زیرا ما هیچ برنامه‌ای برای محتویات EEPROM ننوشته‌ایم. بنابراین اسمبلر این فایل را پس از اتمام عمل اسمبل‌کردن حذف کرده است.

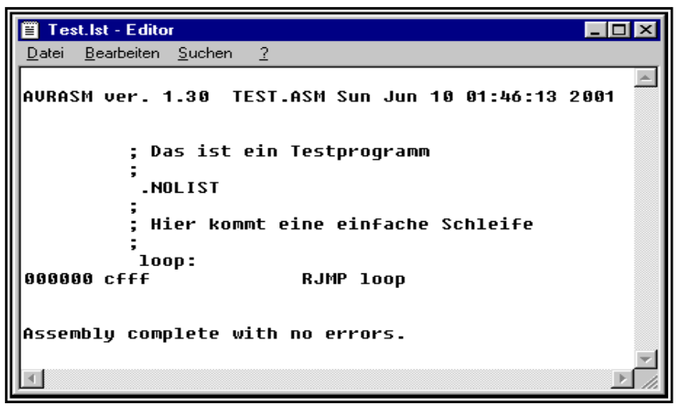
دومین فایل، TEST.HEX، اهمیت بیشتری دارد زیرا حاوی دستوراتی است که به داخل تراشه AVR نوشته خواهند شد. این فایل به شکل روبرو است:

اعداد مبنای شانزده در قالب ویژه‌ای به فرم ASCII، همراه با اطلاعات مربوط به آدرس و یک Checksum (مجموع مقابله ای برای بررسی خطا) برای هر خط نوشته شده است. این قالب، قالب مبنای شانزده اینتل(Intel-Hex-Format) نامیده می‌شود که بسیار قدیمی‌است. این قالب فایل بخوبی توسط نرم‌افزار برنامه‌ریز قابل فهم است.

سومین فایل، TEST.OBJ، بعداً معرفی خواهد شد. این فایل برای عملیات شبیه‌سازی AVR مورد نیاز است. قالب آن به صورت مبنای شانزده بوده و توسط ATMEL تعریف شده است. محتویات این فایل با استفاده از یک ویرایشگر اعداد مبنای شانزده، مشابه شکل روبرو است. توجه: قالب این فایل با نرم‌افزار برنامه‌ریز سازگاری ندارد و نباید از آن برای برنامه‌ریزی AVR استفاده کنید (یک اشتباه بسیار معمول هنگام شروع).



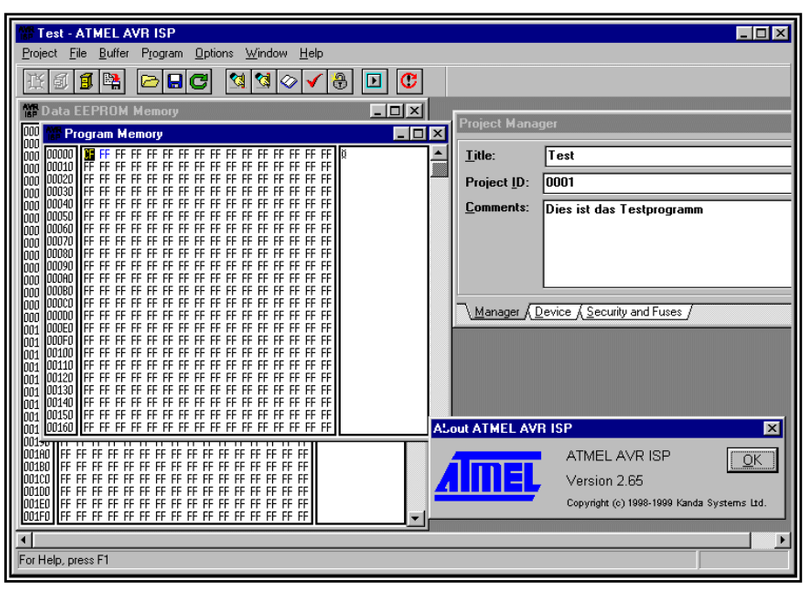
چهارمین فایل، TEST.LST، یک فایل متنی است. محتویات آن را بوسیله یک ویرایشگر ساده مشاهده کنید. نتیجه به صورت مقابل است.



تمام آدرس‌ها، دستورات و پیغام‌های خطای مربوط به برنامه به شکلی قابل فهم نمایش داده شده است. در برخی موارد برای اشکالزدایی برنامه به این فایل نیاز خواهید داشت.

برنامه‌ریزی تراشه‌ها

برای برنامه‌ریزی کردن کد مبنای شانزده خود به داخل AVR، شرکت ATMEL بسته نرم‌افزاری ISP را تولید کرده است.(لازم به ذکر است که این نرم‌افزار دیگر پشتیبانی و توزیع نمی‌شود.) نرم‌افزار ISP را اجرا کرده و فایل HEX. خود را که به تازگی ایجاد کرده‌ایم مطابق شکل زیر به داخل آن لود می‌کنیم(از طریق گزینه LOAD PROGRAM در منوها).

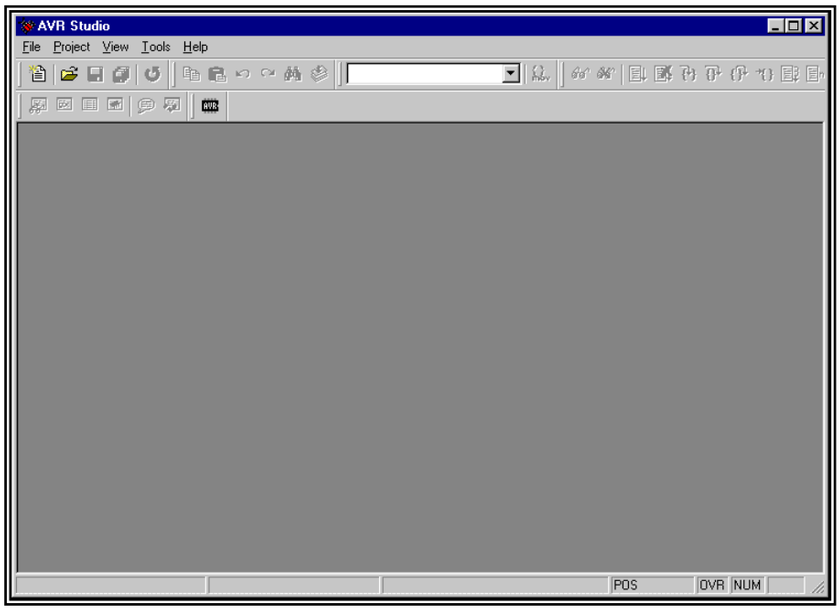


انتخاب گزینه PROGRAM از منو باعث نوشته شدن کدها به داخل حافظهٔ برنامه تراشه می‌شود. برای انجام موفقیت‌آمیز این مرحله باید شرایط لازم فراهم شده باشد(باید پورت موازی درست انتخاب شده باشد، برنامه‌ریز باید متصل باشد، تراشه باید به طور مناسبی روی بورد قرار گرفته باشد، منبع تغذیه باید روشن باشد و ...).

علاوه بر ATMEL-ISP و بوردهای برنامه‌ریز آن، بوردهای برنامه‌ریز دیگر نیز می‌توانند همراه با نرم‌افزار برنامه‌ریز مربوطه مورد استفاده قرار گیرند که برخی از آنها بر روی اینترنت موجودند.

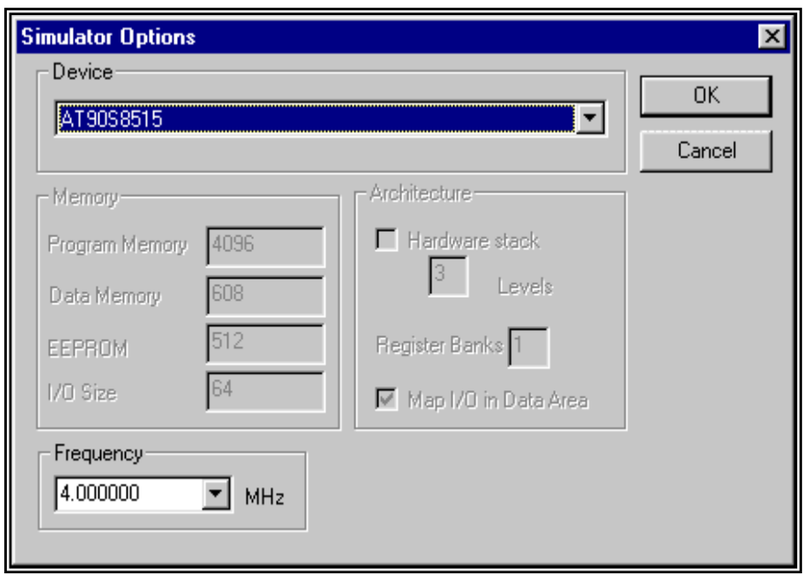
شبیه‌سازی در AVR Studio

در بعضی مواقع، برنامهٔ اسمبلی نوشته شده، حتی وقتی بدون خطا اسمبل شده باشد، پس از برنامه‌ریزی بر روی تراشه، به درستی عمل مورد انتظار را انجام نمی‌دهد. آزمایش نرم‌افزار بر روی تراشه می‌تواند کاری مشکل و پیچیده باشد، به خصوص اگر حداقل سخت‌افزار مورد نیاز در اختیارتان باشد و نیز امکان نمایش نتایج موقتی یا علائم اشکالزدایی برایتان میسر نباشد. در اینگونه موارد برنامه AVR Studio از شرکت ATMEL امکانات خوبی را برای اشکالزدایی فراهم می‌کند. می‌توان کل برنامه یا بخشی از آن را آزمایش کرده و نتایج را مرحله به مرحله مشاهده نمود.



برنامه Studio اجرا شده که به صورت شکل مقابل است:

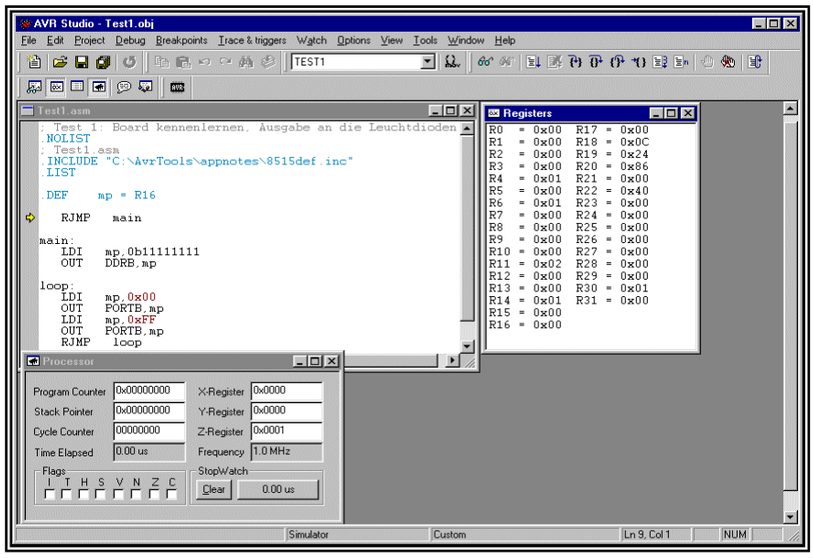
ابتدا فایلی را باز می‌کنیم(منوی FILE گزینه OPEN). ما این بخش را با استفاده از فایل test.asm مربوط به این خودآموز شرح می‌دهیم، زیرا در این مثال دستورات و اعمال بیشتری نسبت به برنامه تک دستورهٔ قبلی ما وجود دارد.



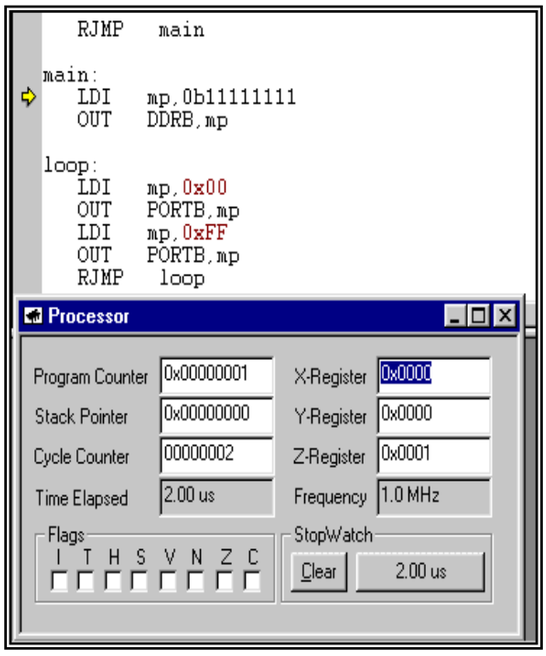
فایل TEST1.OBJ را که نتیجه اسمبل‌کردن TEST1.asm است باز کنید. پس از بازکردن فایل، برنامه از شما می‌خواهد که تنظیمات مورد نظر خود را انتخاب کنید (اگر اینگونه نشد می‌توانید این تنظیمات را از طریق گزینه SIMULATOR OPTIONS در منوها تغییر دهید). تنظیمات زیر را انتخاب می‌کنیم:

در قسمت انتخاب وسیله(Device) نوع تراشه مورد نظر خود را انتخاب می‌کنیم. اگر می‌خواهید هنگام شبیه‌سازی زمان‌بندی‌های صحیحی داشته باشید باید فرکانس مناسب را نیز انتخاب کنید.

برای مشاهده محتویات ثبات‌ها و آگاهی از چگونگی وضعیت پردازنده گزینه‌های PROCESSOR و REGISTERS را از منوی VIEW انتخاب می‌کنیم. اکنون برنامه باید مشابه شکل زیر باشد:



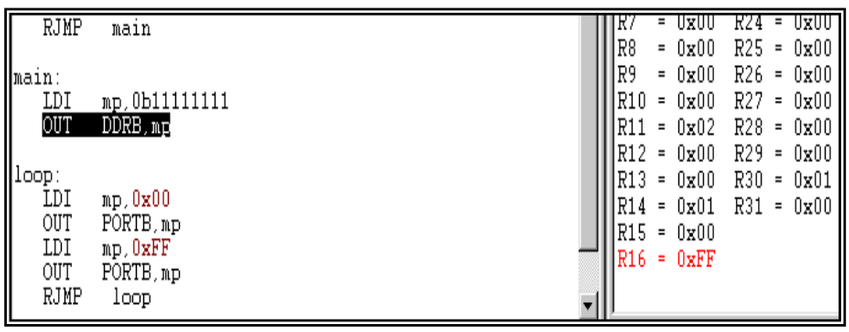
پنجرهٔ Processor تمام مقادیر مربوط به پردازنده مانند شمارنده دستور، پرچم‌ها و اطلاعات زمان‌بندی(در اینجا ۱ MHz) را نمایش می‌دهد. از StopWatch می‌توان برای اندازه‌گیری زمان مورد نیاز برای اجرای زیروال‌ها و غیره استفاده کرد.



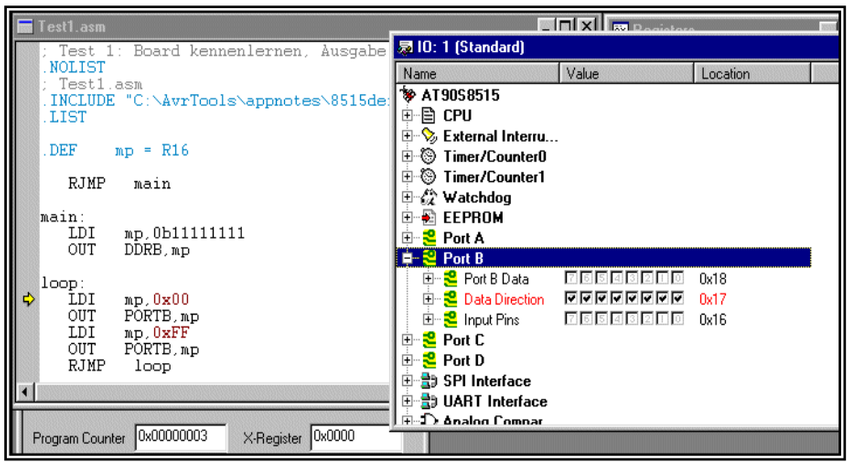
اکنون اجرای برنامه را آغاز می‌کنیم. ما از قابلیت اجرای تک مرحله‌ای(TRACE INTO یا F11) استفاده می‌کنیم. استفاده از گزینه GO باعث اجرای پیوسته و بدون وقفه برنامه می‌شود و به دلیل سرعت بالای شبیه‌سازی چیز زیادی قابل مشاهده نخواهد بود. پس از اجرای نخستین گام، پنجره Processor باید مانند شکل روبرو باشد:

شمارنده برنامه در گام ۱ و شمارنده سیکل در مقدار ۲ (دستور RJMP برای اجرا نیاز به دو سیکل ساعت دارد) قرار دارد. در فرکانس ساعت ۱ MHz دو میکروثانیه سپری شده، پرچم‌ها و ثبات‌های اشاره‌گر تغییری نکرده‌اند. پنجره متن برنامه اشاره‌گری را روی دستور بعدی که باید اجرا شود قرار می‌دهد.

فشردن دوباره کلید F11 دستور بعدی را اجرا کرده، ثبات mp (=R16) برابر 0xFF می‌شود. اکنون باید پنجره Registers این تغییر را به صورت برجسته نشان دهد.

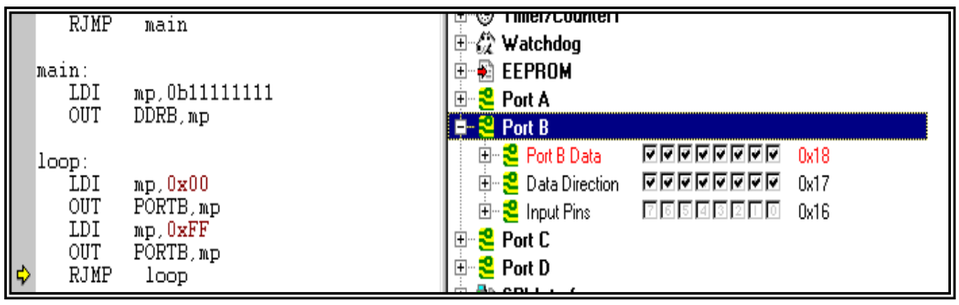


مقدار جدید ثبات R16 با حروف قرمز نشان داده شده است. می‌توانیم مقدار هر ثبات را به طور دلخواه تغییر داده و تأثیر آن را در اجرای برنامه مشاهده کنیم.



اکنون گام ۳ اجرا شده است: خروجی به ثبات تعیین‌کننده جهت پورت B (Data Direction Register). برای مشاهده تغییرات، پنجره I/O جدیدی باز کرده و پورت B را انتخاب می‌کنیم. نتیجه باید به صورت شکل روبرو باشد:

اکنون پنجره نمایش I/O مقدار جدید ثبات تعیین‌کننده جهت پورت B را نشان می‌دهد. در صورت تمایل می‌توان این مقادیر را به طور دستی و پایه به پایه (پین به پین) تغییر داد.



دو گام بعدی با استفاده از F11 شبیه‌سازی شده‌اند. نتایج آنها در اینجا نشان داده نشده است. با تنظیم مقدار پورت‌های خروجی به ۱ با دستورات LDI mp,0xFF و OUT PORTB,mp نتیجهٔ نشان داده شده در شکل بدست می‌آید.

تا اینجا نگاهی اجمالی بر نرم‌افزار شبیه‌ساز داشته‌ایم. شبیه‌ساز دارای قابلیت‌های بسیار بیشتری بوده و باید هنگام وقوع خطاها به طور گسترده‌ای برای اشکالزدایی به کار رود. به گزینه‌های منوهای مختلف نگاهی بیاندازید؛ مطالب بسیار بیشتری از آنچه در اینجا نشان داده شده وجود دارد.

ثبات

ثبات چیست؟

ثبات‌ها حافظه‌های ویژه‌ای با ظرفیت ۸ بیت بوده و به شکل زیر می‌باشند:

بیت ۰	بیت ۱	بیت ۲	بیت ۳	بیت ٤	بیت ٥	بیت ٦	بیت ۷

به نحوه شمارش این بیت‌ها توجه کنید: بیت با کمترین ارزش از صفر($۲^{۰}$-۱) شروع می‌شود.
یک ثبات می‌تواند اعداد ۰ تا ۲۵۵(اعداد مثبت و نامنفی)، یا اعداد از ۱۲۸- تا ۱۲۷+ (اعداد کامل با یک بیت علامت در بیت ۷)، یا یک مقدار نشان‌دهنده یک کاراکتر ASCII (مثلاً 'A')، و یا تنها هشت بیت را که هیچ ارتباطی به هم ندارند(به طور مثال هشت پرچم که به عنوان علامتی برای هشت انتخاب بله/خیر استفاده شده‌اند) در خود ذخیره نماید.

در مقایسه با دیگر محل‌های ذخیره‌سازی، ثبات‌ها ویژگی‌های خاصی دارند:

- آنها را می‌توان به طور مستقیم در دستورات اسمبلر به کار برد،
- انجام عملیات بر روی محتوای آنها تنها نیاز به یک دستور تک کلمه ای دارد،
- آنها به طور مستقیم به واحد پردازش مرکزی که Accumulator نامیده می‌شود متصل هستند،
- آنها ورودی(منبع) و خروجی(مقصد) اعمال محاسباتی هستند.

۳۲ ثبات در یک AVR وجود دارند. آنها در اصل R0 تا R31 نامگذاری شده‌اند، اما می‌توانید با استفاده از راهنمای اسمبلر نام‌های بامعناتری برای آنها انتخاب کنید. مثال:

```
.DEF MyPreferredRegister = R16
```

راهنماهای اسمبلر همواره با یک نقطه در ستون اول خط آغاز می‌شوند، اما دستورات هرگز از ستون ۱ شروع نمی‌شوند، و همواره قبل از آنها یک کاراکتر جای خالی یا Tab باید آورده شود.
توجه داشته باشید که راهنماهای اسمبلر تنها برای اسمبلر دارای معنی هستند و هیچ کد قابل اجرایی برای تراشه AVR تولید نمی‌کنند. اکنون اگر بخواهیم در یک دستور از ثبات R16 استفاده کنیم، می‌توانیم به جای استفاده از نام ثبات R16، نام خودمان یعنی MyPreferredRegister را به کار ببریم. بنابراین در هر بار استفاده از این ثبات باید تعداد کاراکترهای بیشتری بنویسیم، اما در عوض نام این ثبات برای ما نشان‌دهنده نوع محتویات آن خواهد بود.
استفاده از دستور زیر:

```
LDI MyPreferredRegister, 150
```

به این معنی است که عدد ۱۵۰ مستقیماً در داخل ثبات R16 قرار داده شود، LoaD Immediate. این دستور یک مقدار ثابت را در ثبات مورد نظر قرار می‌دهد. با اسمبل‌کردن این کد برنامه، محتویات حافظه برنامهٔ نوشته شده در تراشه AVR به این شکل خواهد بود:

```
000000 E906
```

کد دستور لود همراه با ثبات مقصد(R16) و مقدار ثابت(۱۵۰) بخشی از عدد مبنای شانزده E906 هستند، هرچند نمی‌توانید آن‌ها را به طور مستقیم ببینید. نگران نباشید: مجبور نیستید این روش کدکردن را حفظ باشید زیرا خود اسمبلر می‌داند چگونه این دستورات را ترجمه کرده و E906 را تولید نماید.
می‌توان در داخل یک دستور، دو ثبات مختلف به کار برد. ساده‌ترین دستور از این نوع، دستور کپی‌کردن(MOV) است. این دستور محتویات یک ثبات را در ثبات دیگری کپی می‌کند. به این صورت:

```
.DEF MyPreferredRegister = R16
.DEF AnotherRegister = R15
        LDI MyPreferredRegister, 150
        MOV AnotherRegister, MyPreferredRegister
```

دو خط اول این برنامه راهنماهایی هستند که نام‌های جدید ثبات‌های R16 و R15 را برای اسمبلر تعریف می‌کنند. باز هم یادآوری می‌کنیم که این دو خط هیچ کدی برای AVR تولید نمی‌کنند. خطوط دستوری شامل LDI و MOV سبب تولید کد می‌شوند:

```
000000 E906
000001 2F01
```

این دستورات مقدار ۱۵۰ را در داخل ثبات R16 قرار داده و محتویات آن را به ثبات مقصد R15 کپی می‌کنند. نکته مهم:

ثبات مقصد که نتیجه در آن نوشته می‌شود، همواره اولین ثبات است!

(متأسفانه این مطلب متفاوت از آن چیزی است که شخص انتظار دارد یا آنگونه که ما به طور عادی صحبت می‌کنیم. این قاعدهٔ ساده‌ای است که به این شکل تعریف شده تا مبتدیان یادگیری زبان اسمبلی را گمراه کند! به این دلیل اسمبلی این‌قدر پیچیده است.)

ثبات‌های متفاوت

یک مبتدی ممکن است بخواهد دستورات بالا را به صورت زیر بنویسد:

```
.DEF AnotherRegister = R15
        LDI AnotherRegister, 150
```

و شما اشتباه کرده‌اید. تنها ثبات‌های R16 تا R31 با دستور LDI یک مقدار ثابت را لود می‌کنند، R0 تا R15 قادر به انجام این کار نیستند. این محدودیت چندان خوب نیست، اما هنگام ساخت مجموعه دستورات AVR اجتناب‌ناپذیر بوده است.

در این قاعده یک استثنا وجود دارد: صفر کردن مقدار یک ثبات. دستورِ

```
CLR MyPreferredRegister
```

برای تمام ثبات‌ها صحیح است.

علاوه بر دستور LDI، این نوع محدودیت استفاده از ثبات در دستورات زیر نیز وجود دارد:

- ANDI Rx,K ؛ AND بیتی ثبات Rx با مقدار ثابت K،
- CBR Rx,M ؛ پاک کردن تمام بیت‌هایی از ثبات Rx که بیت‌های متناظرشان در مقدار ثابت M (به عنوان مقدار ماسک) برابر یک است،
- CPI Rx,K ؛ مقایسه محتویات ثبات Rx با مقدار ثابت K،
- SBCI Rx,K ؛ کم کردن مقدار ثابت K و مقدار جاری پرچم نقلی (Carry) از محتویات ثبات Rx و قرار دادن نتیجه در ثبات Rx،
- SBR Rx,M ؛ یک کردن تمام بیت‌هایی از ثبات Rx که بیت‌های متناظرشان در مقدار ثابت M (به عنوان مقدار ماسک) برابر یک است،
- SER Rx ؛ یک کردن تمام بیت‌های ثبات Rx (معادل با LDI Rx,255)،
- SUBI Rx,K ؛ کم کردن مقدار ثابت K از محتویات ثبات Rx و قرار دادن نتیجه در ثبات Rx.

در تمام این دستورات ثبات Rx می‌بایست بین R16 و R31 باشد! اگر بخواهید از این دستورات استفاده کنید باید یکی از این ثبات‌ها را برای انجام عمل مورد نظر انتخاب کنید. استفاده از این دستورات، برنامه‌نویسی را آسان‌تر می‌کند. این دلیل دیگری است بر اینکه چرا باید از راهنماهای اسمبلر برای تعیین نام ثبات‌ها استفاده کنید، زیرا با این کار می‌توانید به راحتی موقعیت ثبات‌ها را تغییر دهید.

ثبات اشاره‌گر

عملکرد ویژه‌ای برای زوج ثبات‌های R27:R26، R29:R28 و R31:R30 تعریف شده است. این عملکرد به قدری مهم است که در اسمبلر AVR اسامی اظافی برای این زوج ثبات‌ها در نظر گرفته شده است: X، Y و Z. این زوج ثبات‌ها، ثبات‌های ۱۶ بیتی هستند که قادر به اشاره به محل‌های حافظه با آدرس حداکثر ۱۶ بیتی در داخل SRAM (X، Y و Z) یا در داخل حافظه برنامه (Z) هستند.

بایت پایین آدرس ۱۶ بیتی در ثباتِ با شمارهٔ پایین‌تر و بایت بالا در ثباتِ با شمارهٔ بالاتر قرار می‌گیرد. هر دو بخش نام‌های مخصوص به خودشان را دارند. به طور مثال بایت بالای Z را ZH (= R31) و بایت پایین آن را ZL (= R30) می‌نامند. این نام‌ها در سرفایل‌های استاندارد(Standard Header File) مربوط به تراشه‌ها تعریف شده است. تقسیم‌بندی این نام‌های اشاره‌گرِ ۱۶ بیتی به دو بایت مختلف به صورت زیر انجام می‌شود:

```
.EQU Address = RAMEND ; RAMEND is the highest 16-bit address in SRAM
    LDI YH,HIGH(Address) ; Set the MSB
    LDI YL,LOW(Address) ; Set the LSB
```

دستورات خاصی برای دسترسی به حافظه با استفاده از اشاره‌گرها در نظر گرفته شده است. دستور خواندن از حافظه، LD (LoaD) و دستور نوشتن در حافظه، ST (STore) نامگذاری شده است. به طور مثال برای اشاره‌گر X:

اشاره‌گر	عملکرد	مثال‌ها
X	خواندن از/نوشتن به آدرس X، بدون تغییر مقدار اشاره‌گر	LD R1,X یا ST X,R1
X+	خواندن از/ نوشتن به آدرس X و سپس افزودن یک واحد به مقدار اشاره‌گر	LD R1,X+ یا ST X+,R1
-X	کاهش یک واحد از مقدار اشاره‌گر و سپس خواندن از/ نوشتن به آدرس جدید	LD R1,-X یا ST -X,R1

به طور مشابه می‌توانید Y یا Z را برای عمل مورد نظر به کار ببرید.

برای خواندن از حافظهٔ برنامه تنها یک دستور وجود دارد. این دستور فقط برای اشاره‌گر Z تعریف شده و بنام LPM (Load from Program Memory) نامگذاری شده است. این دستور، بایت موجود در آدرس Z حافظهٔ برنامه را در ثبات R0 کپی می‌کند. از آنجا که ساختار حافظهٔ برنامه به صورت کلمه به کلمه[1] است (هر دستور موجود در یک آدرس خاص، از ۱۶ بیت یا دو بایت یا یک کلمه تشکیل شده است)، انتخاب بایت بالا یا پایین توسط بیت با کمترین ارزش انجام می‌شود(۰= بایت پایین، ۱= بایت بالا). به همین دلیل باید آدرس اصلی در ۲ ضرب شود و بنابراین حوزهٔ دسترسی محدود به ۱۵ بیت یا ۳۲ کیلوبایت حافظه برنامه است. به این صورت:

```
LDI ZH,HIGH(2*Address)
LDI ZL,LOW(2*Address)
LPM
```

پس از این دستور، آدرس باید برای اشاره به بایت بعدی در حافظه برنامه افزایش یابد. از آنجا که این کار اکثر اوقات لازم می‌شود دستور ویژه‌ای برای افزایش مقدار اشاره‌گر تعریف شده است:

```
ADIW ZL,1
LPM
```

ADIW به معنی افزودن بی‌واسطه یک کلمه(Add Immediate Word) بوده و حداکثر مقداری که می‌توان به این روش اظافه کرد ۶۳ می‌باشد. توجه داشته باشید که اسمبلر بخش پایینی ثبات اشاره‌گر ZL را به عنوان اولین پارامتر می‌پذیرد. این روش قدری نامأنوس است زیرا عمل جمع به صورت ۱۶ بیتی انجام می‌شود.

مکمل این دستور، تفریق یک مقدار ثابت بین ۰ تا ۶۳ از یک ثبات اشاره‌گر ۱۶ بیتی، بنام SBIW، تفریق بی‌واسطهٔ یک کلمه (SuBtract Immediate Word)، نامگذاری شده است. امکان استفاده از ADIW وSBIW برای زوج ثبات‌های

1- Word-Wise Organized

اشاره‌گر X، Y و Z، و نیز زوج ثبات R25:R24 که نام خاصی نداشته و اجازه آدرس‌دهی در SRAM و حافظه برنامه را نمی‌دهد، وجود دارد. هنگام استفاده از مقادیر ۱۶ بیتی، R25:R24 می‌تواند یک انتخاب بسیار مناسب باشد.

چگونه جدولی از مقادیر ثابت را در حافظه برنامه جای دهیم؟ این کار با استفاده از راهنماهای اسمبلر DB. وDW. انجام می‌شود. با استفاده از آنها می‌توانید لیست‌های مقادیر را به صورت بایت به بایت یا کلمه به کلمه در حافظهٔ برنامه قرار دهید. لیست‌های به صورت بایت به بایت به این شکل هستند:

```
.DB 123,45,67,89 ; a list of four bytes
.DB "This is a text. " ; a list of byte characters
```

تعداد بایت‌های قرار گرفته در یک خط باید همیشه زوج باشد؛ در غیراینصورت اسمبلر یک بایت صفر، که ممکن است هیچ استفاده‌ای نداشته باشد، به انتهای خط اظافه می‌کند.

به طور مشابه، لیستی از مقادیر کلمه ای به شکل زیر است:

```
.DW 12345,6789 ; a list of two words
```

در لیست‌ها به جای استفاده از مقادیر ثابت می‌توانید از برچسب‌ها(آدرس‌های پرش) نیز به شکل زیر استفاده کنید:

```
[ ... here are some commands ... ]
Label2:
[ ... here are some more commands ... ]
Table:
.DW Label1,Label2 ; a wordwise list of labels
```

برچسب‌ها همواره از ستون ۱ شروع می‌شوند! توجه کنید که خواندن برچسب‌ها با دستور LPM ابتدا بایت پایین کلمه را بدست می‌دهد.

یک کاربرد بسیار خاص ثبات‌های اشاره‌گر، استفاده از آنها برای دسترسی به خود ثبات‌ها است. ثبات‌ها در اولین ۳۲ بایت فضای آدرس تراشه(در آدرس 0x0000 تا 0x001F) قرار دارند. این روش دسترسی به ثبات‌ها تنها زمانی مفید و بامعنی است که نیاز به کپی‌کردن محتوای یک ثبات به داخل SRAM یا EEPROM و یا خواندن مقدار آن از آنجا به داخل ثبات داشته باشید. متداول‌ترین کاربرد اشاره‌گرها دسترسی به جداول حاوی مقادیر ثابت در فضای حافظه برنامه است. به عنوان مثال، در اینجا جدولی با ۱۰ مقدار ۱۶ بیتی متفاوت داریم که پنجمین مقدار جدول به داخل R25:R24 خوانده شده است:

```
MyTable:
.DW 0x1234,0x2345,0x3456,0x4568,0x5678 ; The table values, wordwise
.DW 0x6789,0x789A,0x89AB,0x9ABC,0xABCD ; organised
Read5: LDI ZH,HIGH(MyTable*2) ; Address of table to pointer Z
       LDI ZL,LOW(MyTable*2) ; multiplied by 2 for bytewise access
       ADIW ZL,10 ; Point to fifth value in table
       LPM ; Read least significant byte from program memory
       MOV R24,R0 ; Copy LSB to 16-bit register
       ADIW ZL,1 ; Point to MSB in program memory
       LPM ; Read MSB of table value
       MOV R25,R0 ; Copy MSB to 16-bit register
```

این فقط یک مثال است. شما می‌توانید براساس برخی مقادیر ورودی، آدرس مورد نظر را در اشاره‌گر Z محاسبه کرده و به مقادیر مطلوب در جدول برسید. جداول می‌توانند دارای ساختار بایتی یا کاراکتری نیز باشند.

توصیه‌هایی برای استفاده از ثبات‌ها

- با راهنمای DEF. برای ثبات‌ها نام تعریف کنید و هرگز آنها را با نام مستقیمشان یعنی Rx به کار نبرید.
- اگر نیاز به دسترسی از طریق اشاره‌گر دارید ثبات‌های R26 تا R31 را به این کار اختصاص دهید.
- برای شمارندهٔ ۱۶ بیتی، بهترین انتخاب R25:R24 است.
- اگر نیاز به خواندن از حافظهٔ برنامه دارید، مثلاً برای جداول ثابت، Z و R0 را به این کار اختصاص دهید.

- **اگر می‌خواهید به بیت‌های برخی ثبات‌ها به طور مستقیم دسترسی داشته باشید(مثلاً برای آزمایش پرچم‌ها)، ثبات‌های R16 تا R23 را برای این منظور به کار ببرید.**

پورت‌ها

پورت چیست؟

در AVR پورت‌ها گذرگاه‌هایی از واحد پردازش مرکزی به اجـزای سـخت‌افـزاری و نـرم‌افـزاری داخلـی و خـارجی هستند.CPU با این اجزا، مثلاً تایمرها یا پورت‌های موازی، ارتباط برقرار کرده و داده‌ها را از آنهـا خوانـده یـا بـه آنهـا می‌نویسد. پرکاربردترین پورت، ثبات پرچم (Flag Register) است که نتایج عملیات‌های پیشین در آن نوشـته شـده و شرط‌های انشعاب (پرش) از آن خوانده می‌شود.

مجموعاً ٦٤ پورت مختلف وجود دارند که به طور فیزیکی در تمام مدل‌های AVR موجود نیستند. بسته به میزان فـضای ذخیره‌سازی و دیگر سخت‌افزارهای داخلی، پورت‌های مختلفی ممکن اسـت موجـود و قابـل دسترسـی باشـند. لیـست پورت‌های قابل استفاده برای هر نوع پردازنده در برگه‌های اطلاعاتی (Data Sheet) مربوط به آن آورده شده است.

پورت‌ها آدرس‌های ثابتی دارند کهCPU از طریق آن با آنها ارتباط برقرار می‌کند. این آدرس‌هـا مـستقل از نـوع AVR هستند. پس به طور مثال آدرس پورت B همواره 0x18 (پیشوند 0x علامت مبنای شانزده بودن عـدد اسـت) مـی‌باشـد. مجبور به حفظ کردن آدرس پورت‌ها نیستید، برای آنها نام‌های مستعار مناسبی در نظر گرفته شده است. این نـام‌هـا بـرای انواع مختلف AVR در فایل‌های ضمیمه (فایل‌های سرآیند، Header File) تعریف شده و توسط تولیـد کننـده عرضـه شده‌اند. در فایل‌های ضمیمه، خطی به شکل زیر آدرس پورت B را تعریف می‌کند:

```
.EQU PORTB, 0x18
```

بنابراین ما فقط باید نام پورت B را به خاطر داشته باشیم، نه محل آن در فضای I/O تراشه. فایل ضـمیمهٔ 8515def.inc با استفاده از راهنمای اسمبلر ضمیمه شده است:

```
.INCLUDE "C:\Somewhere\8515def.inc"
```

و بنابراین تمام ثبات‌های 8515 تعریف شده و به آسانی قابل دسترسی می‌باشند.

پورت‌ها معمولاً دارای ساختار ۸ بیتی هستند، اما می‌توانند ۸ تک بیت جدا از هم را نیز که هیچ ارتباطی به یکدیگر ندارند در خود نگه دارند. اگر این تک بیت‌ها معنای خاصی داشته باشند، در فایل ضمیمه اسامی‌خاصی برای آنها درنظـر گرفتـه شده تا مثلاً امکان دستکاری یک تک بیت فراهم شود. به واسطهٔ این قرارداد نامگذاری، شما مجبور بـه حفـظ محـل ایـن بیت‌ها نیستید. این اسامی‌در برگه‌های اطلاعاتی(Data Sheet) تعریف شده و در فایل ضمیمه نیز وجود دارنـد. در اینجـا این اسامی‌در جداول پورت آورده شده است.

به عنوان یک مثال،MCU ثبات کنترل عمومی(General Control Register)، که MCUCR نامیده می‌شـود، شـامل تعدادی بیت کنترلی است که ویژگی‌های عمومی‌تراشه را کنترل می‌کنند(جزئیات MCUCR را ببینید). این پورت شـامل ۸ بیت کنترلی است که دارای نام‌های مخصوص به خودشان (ISC00 , ISC01 , ...) هستند. کسی که می‌خواهد تراشهٔ AVR خود را به یک خواب عمیق فرو ببرد(یعنی مد Sleep)، باید از برگه‌های اطلاعـاتی چگـونگی تنظـیم بیـت‌هـای مربوطه را بداند. این کار به این صورت انجام می‌شود:

```
.DEF MyPreferredRegister = R16
        LDI MyPreferredRegister, 0b00100000
        OUT MCUCR, MyPreferredRegister
        SLEEP
```

دستور OUT محتویات ثبات مورد نظر را، که بیت فعال‌سازی Sleep (Sleep-Enable-Bit) بنام SE در آن یک شـده است، به پورت MCUCR انتقال داده و با اجرای دستورالعمل SLEEP، بلافاصله AVR را به حالت Sleep می‌برد. از آنجا که بیت‌های دیگر MCUCR نیز توسط دستورات بالا تغییر داده شده و بیت Sleep Mode که SM نامیده می‌شود

صفر شده است، AVR با دستور SLEEP به حالتی بنام Half-Sleep وارد خواهد شد: اجرای دستورات متوقف شده، اما هنوز تراشه به وقفه‌های تایمر و دیگر سخت‌افزارها واکنش نشان می‌دهد. این رویدادهای خارجی هنگامی‌که نیاز به فعال کردن CPU باشد، CPU را از حالت Sleep خارج می‌کنند.

خواندن مقدار یک پورت در اکثر موارد با استفاده از دستور IN امکان‌پذیر است. دستورات زیر

```
.DEF MyPreferredRegister = R16
        IN MyPreferredRegister, MCUCR
```

بیت‌های پورت MCUCR را به داخل ثبات می‌خواند. از آنجا که اکثر پورت‌ها بیت‌های تعریف نشده و بلااستفاده‌ای دارند، اینگونه بیت‌ها هنگام خواندن همواره برابر صفر هستند.

در اکثر اوقات به جای خواندن همهٔ ۸ بیت یک پورت، لازم است براساس حالت معینی از پورت عمل کنیم. در اینگونه موارد نیازی به خواندن کل پورت نداشته و بیت مورد نظر را از بقیه بیت‌ها جدا می‌کنیم. برخی دستورات امکان اجرای دستورات را بسته به مقدار یک بیت خاص فراهم می‌کنند(بخش مربوط به انشعاب و تغییر مسیر برنامه (JUMP) را ببینید). تنظیم و پاک کردن بیت‌های معینی از یک پورت بدون خواندن و نوشتن بیت‌های دیگر پورت نیز امکان‌پذیر است. دستورات مربوطه SBI (Set Bit I/o) و CBI (Clear Bit I/o) هستند. نحوه اجرا به صورت زیر است:

```
.EQU ActiveBit=0 ; The bit that is to be changed
        SBI PortB, ActiveBit ; The bit will be set to one
        CBI PortB, Activebit ; The bit will be cleared to zero
```

این دو دستورالعمل یک محدودیت دارند: تنها پورت‌های با آدرس کوچک‌تر از 0x20 قابل استفاده‌اند؛ با این روش نمی‌توان به پورت‌های بالاتر دسترسی داشت.

قابل توجه برنامه‌نویسان نامتعارف: می‌توان با استفاده از دستورات دستیابی به حافظهٔ SRAM، مانند ST و LD، به پورت‌ها دسترسی داشت. تنها کافی است مقدار 0x20 را به آدرس پورت اضافه کرده (۳۲ آدرس اول متعلق به ثبات‌ها است!) و به این روش به پورت دسترسی داشت. مثالی در اینجا آورده شده است:

```
.DEF MyPreferredRegister = R16
        LDI ZH,HIGH(PORTB+32)
        LDI ZL,LOW(PORTB+32)
        LD MyPreferredRegister,Z
```

استفاده از این روش تنها در برخی موارد مفید و معنی‌دار است، اما به هر حال امکان‌پذیر است. علت اینکه چرا همیشه اولین آدرس SRAM برابر 0x60 می‌باشد همین است.

جزئیات پورت‌های پرکاربرد در AVR

اسامی‌پورت‌های پرکاربرد در جدول زیر آورده شده است. همه پورت‌ها در اینجا لیست نشده‌اند؛ از برخی مدل‌های MEGA و AT90S4434/8535 صرف‌نظر شده است. اگر در صحت این جدول تردید دارید به منابع اصلی مراجعه کنید.

Component	*Portname*	*Port-Register*
Accumulator	SREG	Status Register
Stack	SPL/SPH	Stackpointer
External SRAM/External Interrupt	MCUCR	MCU General Control Register
External Interrupt	GIMSK	Interrupt Mask Register
	GIFR	Interrupt Flag Register
Timer Interrupt	TIMSK	Timer Interrupt Mask Register
	TIFR	Timer Interrupt Flag Register
Timer 0	TCCR0	Timer/Counter 0 Control Register
	TCNT0	Timer/Counter 0

Component	*Portname*	*Port-Register*
Timer 1	TCCR1A	Timer/Counter Control Register 1 A
	TCCR1B	Timer/Counter Control Register 1 B
	TCNT1	Timer/Counter 1
	OCR1A	Output Compare Register 1 A
	OCR1B	Output Compare Register 1 B
	ICR1L/H	Input Capture Register
Watchdog Timer	WDTCR	Watchdog Timer Control Register
EEPROM	EEAR	EEPROM Address Register
	EEDR	EEPROM Data Register
	EECR	EEPROM Control Register
SPI	SPCR	Serial Peripheral Control Register
	SPSR	Serial Peripheral Status Register
	SPDR	Serial Peripheral Data Register
UART	UDR	UART Data Register
	USR	UART Status Register
	UCR	UART Control Register
	UBRR	UART Baud Rate Register
Analog Comparator	ACSR	Analog Comparator Control and Status Register
I/O-Ports	PORTx	Port Output Register
	DDRx	Port Direction Register
	PINx	Port Input Register

ثبات وضعیت به عنوان پرکاربردترین پورت

ثبات وضعیت با ۸ بیت آن به مراتب پرکاربردترین پورت می‌باشد. معمولاً دسترسی به این پورت تنها به صورت تنظیم و پاک‌کردن خودکار بیت‌ها توسط CPU یا Accumulator انجام می‌شود. گاهی هم دسترسی‌ها به صورت انشعاب مسیر برنامه براساس مقدار بیت‌های معینی از این پورت بوده، و در موارد اندکی امکان دستکاری مستقیم این بیت‌ها (با استفاده از دستورات اسمبلر SEx یا CLx، که x حرف اختصاری بیت مورد نظر است) میسر است. اکثر این بیت‌ها حین عملیات‌های آزمایش بیتی، مقایسه‌ای یا محاسباتی توسط Accumulator صفر یا یک می‌شوند. در لیست زیر تمام دستوراتی از اسمبلر که بیت‌های وضعیت را براساس نتیجه اجرای دستور، صفر یا یک می‌کنند آورده شده است.

بیت	دستورات محاسباتی	دستورات منطقی	دستورات مقایسه‌ای	دستورات بیتی	دستورات انتقالی	دستورات دیگر
Z	ADD, ADC, ADIW, DEC, INC, SUB, SUBI, SBC, SBCI, SBIW	AND, ANDI, OR, ORI, EOR, COM, NEG, SBR, CBR	CP, CPC, CPI	BCLR Z, BSET Z, CLZ, SEZ, TST	ASR, LSL, LSR, ROL, ROR	CLR
C	ADD, ADC, ADIW, SUB, SUBI, SBC, SBCI, SBIW	COM, NEG	CP, CPC, CPI	BCLR C, BSET C, CLC, SEC	ASR, LSL, LSR, ROL, ROR	-
N	ADD, ADC, ADIW, DEC, INC, SUB, SUBI, SBC, SBCI, SBIW	AND, ANDI, OR, ORI, EOR, COM, NEG, SBR, CBR	CP, CPC, CPI	BCLR N, BSET N, CLN, SEN, TST	ASR, LSL, LSR, ROL, ROR	CLR
V	ADD, ADC, ADIW, DEC, INC, SUB, SUBI, SBC, SBCI, SBIW	AND, ANDI, OR, ORI, EOR, COM, NEG, SBR, CBR	CP, CPC, CPI	BCLR V, BSET V, CLV, SEV, TST	ASR, LSL, LSR, ROL, ROR	CLR
S	SBIW	-	-	BCLR S, BSET S, CLS, SES	-	-

بیت	دستورات محاسباتی	دستورات منطقی	دستورات مقایسه‌ای	دستورات بیتی	دستورات انتقالی	دستورات دیگر
H	ADD, ADC, SUB, SUBI, SBC, SBCI	NEG	CP, CPC, CPI	BCLR H, BSET H, CLH, SEH	-	-
T	-	-	-	BCLR T, BSET T, BST, CLT, SET	-	-
I	-	-	-	BCLR I, BSET I, CLI, SEI	-	RETI

جزئیات پورت‌ها

جزئیات مربوط به معمول‌ترین پورت‌ها در جدول دیگری آورده شده است (ضمایم را ببینید).

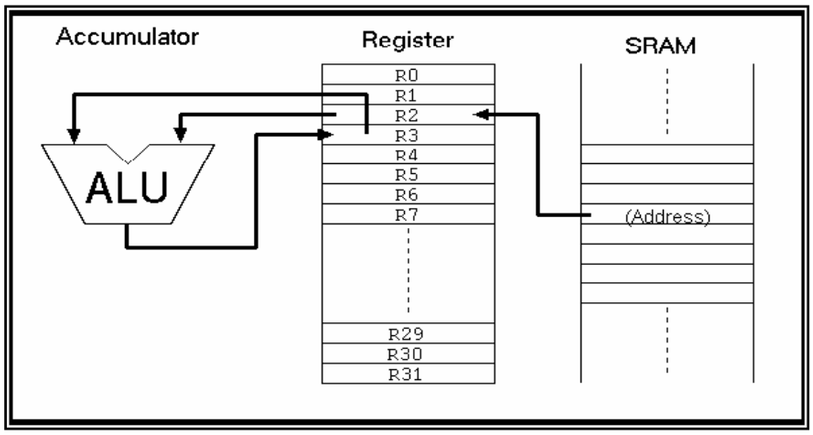
SRAM

استفاده از SRAM در زبان اسمبلی AVR

تقریباً واحد کنترل اصلی(MCU) تمام مدل‌های AT90S خانواده AVR دارای Static RAM (SRAM) داخلی هستند(بعضی مدل‌ها ندارند). تنها برنامه‌های بسیار ساده اسمبلر می‌توانند با قرار دادن تمام اطلاعات خود در داخل ثبات‌ها از استفاده از این فضای حافظه اجتناب کنند. در صورت کمبود ثبات‌ها باید قادر به برنامه‌نویسی SRAM باشید تا از فضای حافظهٔ بیشتری استفاده کنید.

SRAM چیست؟

SRAM متشکل از خانه‌های حافظه‌ای است که بر خلاف ثبات‌ها به طور مستقیم توسط واحد پردازش مرکزی(واحد محاسبه و منطق ALU، که گاهی نیز Accumulator نامیده می‌شود) قابل دسترسی نیستند. هنگام دسترسی به این محل‌های حافظه معمولاً از یک ثبات به عنوان حافظه میانی استفاده می‌شود. در مثال زیر یک مقدار از داخل SRAM به ثبات R2 کپی می‌شود(اولین دستور)، سپس با استفاده از مقدار R3 محاسباتی انجام شده و نتیجه در ثبات R3 نوشته می‌شود(دستور دوم). سپس این مقدار دوباره در SRAM ذخیره می‌شود(دستور سوم، که در اینجا نشان داده نشده است).



بنابراین کاملاً واضح است که انجام عملیات بر روی مقادیر ذخیره‌شده در SRAM کندتر از انجام آنها با استفاده از ثبات‌ها است. از طرف دیگر،کوچک‌ترین نوع AVR، ۱۲۸ بایت حافظهٔ SRAM دارد که بسیار بیشتر از مقداری است که ۳۲ ثبات می‌توانند در خود نگهدارند.

در مدل‌های AT90S8515 و بالاتر امکان اتصال RAM اضافی برای افزایش حافظه داخلی ۵۱۲ بایتی وجود دارد. از دید اسمبلر، دسترسی به SRAM خارجی مشابه دسترسی به SRAM داخلی است. هیچ دستور اضافی برای استفاده از SRAM خارجی لازم نیست.

برای چه مقاصدی می‌توان از SRAM استفاده کرد؟

علاوه بر ذخیره مقادیر، SRAM امکانات اضافی دیگری برای استفاده از آن می‌دهد. دسترسی نه تنها با آدرس‌های ثابت، بلکه از طریق اشاره‌گرها نیز امکان‌پذیر بوده و می‌توان دستیابی شناور و نسبی به محل‌های متوالی حافظه را برنامه‌نویسی کرد. به این وسیله می‌توانید بافرهای حلقوی برای ذخیره موقتی مقادیر یا جداول محاسباتی ایجاد کنید. این کار غالباً بدون استفاده از ثبات‌ها انجام می‌شود، زیرا تعداد آنها بسیار کم بوده و دسترسی به آنها به صورت ایستا (ثابت) است.

از آن جالبتر اینکه می‌توان با استفاده از یک آفست نسبت به آدرس آغازینِ موجود در یکی از ثبات‌های اشاره‌گر، به محل‌های حافظه دسترسی داشت. در این حالت آدرس ثابتی در یک ثبات اشاره‌گر ذخیره می‌شود، مقدار ثابتی به این آدرس افزوده شده و دسترسی خواندن/نوشتن به این آدرس(که براساس آفست محاسبه شده) صورت می‌گیرد. با این روشِ دستیابی به حافظه، استفاده از جداول ساده‌تر می‌شود .

بهترین و مهمترین کاربرد SRAM، اصطلاحاً پشته نامیده می‌شود. شما می‌توانید مقادیر مختلفی را در پشته قرار دهید. این مقادیر می‌تواند محتوای یک ثبات، آدرس برگشت قبل از فراخوانی یک زیرروال، و یا آدرس برگشت قبل از فعال‌شدن یک وقفه سخت‌افزاری باشد.

نحوه استفاده از SRAM

برای کپی‌کردن یک مقدار به محلی در حافظه SRAM باید آدرس آن محل از حافظه را تعیین کنید. آدرس‌های SRAM قابل استفاده از 0x0060 (در مبنای شانزده) تا انتهای SRAM فیزیکی موجود بر روی تراشه است (در AT90S8515 بالاترین محل حافظه قابل دسترسیِ SRAM، 0x025F است). با دستورِ

```
STS 0x0060, R1
```

محتویات ثبات R1 در اولین محل حافظه SRAM کپی می‌شود. با

```
LDS R1, 0x0060
```

محتویات حافظه SRAM موجود در آدرس 0x0060 به داخل این ثبات کپی می‌شود. این روش، دسترسی مستقیم نامیده می‌شود که آدرس آن باید توسط برنامه‌نویس تعیین شده باشد.

برای پرهیز از بکارگیری آدرس‌های ثابت در برنامه، که در صورتی که بخواهید ساختار داده‌های خود را درSRAM تغییر دهید نیاز به کار بسیار زیادی دارند، می‌توان از نام‌های نمادین استفاده کرد. استفاده از این نام‌ها راحت‌تر از به کار بردن اعداد مبنای شانزده است. بنابراین برای این آدرس به صورت زیر نامی تعریف کنید:

```
.EQU MyPreferredStorageCell = 0x0060
        STS MyPreferredStorageCell, R1
```

درست است که این روش، کوتاه‌تر و مختصرتر نیست اما به خاطر سپردن آن ساده‌تر است. هر نامی را که به نظرتان مناسب است استفاده کنید.

روش دیگر دستیابی به SRAM، استفاده از اشاره‌گرهاست. برای این کار نیاز به دو ثبات دارید که آدرس ۱۶ بیتی محل را در خود نگه می‌دارند. همانگونه که در قسمت تقسیم بندی ثبات‌های اشاره‌گر یاد گرفتیم، ثبات‌های اشاره‌گر به صورت زوج ثبات‌های X(XH:XL ،R27:R26)، Y(YH:YL ،R29:R28) و Z(ZH:ZL ،R31:R30) هستند. این ثبات‌ها اجازه دسترسی به محلی از حافظه را که به صورت مستقیم به آن اشاره می‌کنند(مانند ST X,R1) می‌دهند؛ همچنین امکان کاهش یک واحدی آدرس قبل از دسترسی به حافظه(مانند ST –X,R1) و یا افزایش یک واحدی آدرس پس از دسترسی به حافظه (مانند ST X+,R1) را می‌دهند. دسترسی به سه خانهٔ موجود در یک ردیف به صورت زیر است:

```
.EQU MyPreferredStorageCell = 0x0060
.DEF MyPreferredRegister = R1
.DEF AnotherRegister = R2
.DEF AndAnotherRegister = R3
        LDI XH, HIGH(MyPreferredStorageCell)
        LDI XL, LOW(MyPreferredStorageCell)
        LD MyPreferredRegister, X+
        LD AnotherRegister, X+
        LD AndAnotherRegister, X
```

به کار بردن این اشاره‌گرها آسان است. حتی به آسانی زبان‌های غیر از اسمبلی که ادعا می‌کنند یادگیری آنها آسان است.

سومین ترکیب قدری مبهم و نامتعارف بوده و تنها برنامه‌نویسان باتجربه آن را به کار می‌برند. فرض کنیم در برنامه‌مان بارها نیاز به دسترسی به سه محل حافظه در SRAM داریم. همچنین فرض می‌کنیم که یک زوج ثبات اشاره‌گر بلااستفاده داریم و می‌توانیم آن را انحصاراً برای این منظور به کار بگیریم. اگر از دستورات ST/LD استفاده کنیم همیشه مجبور به تغییر اشاره‌گر برای دسترسی به این محل‌های حافظه هستیم. این روش جندان راحت و مناسب نیست.

برای پرهیز از این کار و گمراه کردن(!) فرد مبتدی، دسترسی با استفاده از آفست ابداع شد. در این روشِ دسترسی، مقدار ثبات اشاره‌گر تغییر نکرده و با افزودن موقتی آفست، آدرس محاسبه می‌شود. در مثال بالا دسترسی به محل حافظه

0x0062 می‌توانست با استفاده از این روش به صورت زیر انجام شود. ابتدا آدرس محل مرکزی، یعنی 0x0060، در ثبات اشاره‌گر قرار می‌گیرد:

```
.EQU MyPreferredStorageCell = 0x0060
.DEF MyPreferredRegister = R1
        LDI YH, HIGH(MyPreferredStorageCell)
        LDI YL, LOW(MyPreferredStorageCell)
```

جایی در این برنامه می‌خواهیم به خانه حافظه 0x0062 دسترسی داشته باشیم:

```
        STD Y+2, MyPreferredRegister
```

توجه کنید که مقدار ۲ واقعاً به Y اضافه نشده است و فقط یک مقدار موقتی است. برای گمراه‌کردن بیشتر شما، این کار تنها با زوج ثبات‌های Y و Z قابل انجام است نه با اشاره‌گر X !
برای خواندن از حافظه SRAM نیز دستور مشابهی وجود دارد:

```
        LDD MyPreferredRegister, Y+2
```

این تمام مطالب مربوط به SRAM است. اما صبر کنید: هنوز مهمترین کاربرد آن به عنوان پشته باقی مانده است که باید یاد بگیرید.

استفاده از SRAM به عنوان پشته (Stack)

رایج‌ترین کاربرد SRAM استفاده از آن به عنوان پشته است. پشته ستونی از بلاک‌هاست. هر بلاک جدید در بالای ستون قرار می‌گیرد، و هر بازخوانی بلاک، بالاترین بلاک موجود بر روی ستون را از روی آن حذف می‌کند. این نوع ساختار را آخرین ورودی- اولین خروجی(Last-In-First-Out یا LIFO) می‌نامند، یا به زبان ساده‌تر: آخرین قلم داده‌ای که در بالا قرار می‌گیرد اولین قلم داده‌ای خواهد بود که پایین می‌آید.

تعریف SRAM به عنوان پشته

برای استفاده از SRAM به عنوان پشته، ابتدا باید اشاره‌گر پشته تنظیم شود. اشاره‌گر پشته یک اشاره‌گر ١٦ بیتی بوده و مشابه یک پورت دسترسی به آن انجام می‌شود. نام این ثبات دو قسمتی SPH:SPL است. SPH بایت با ارزش بیشتر و SPL بایت با ارزش کمتر آدرس را در خود نگه می‌دارد. این مطلب تنها زمانی صحیح است که نوع AVR مورد استفاده، بیش از ٢٥٦ بایت SRAM داشته باشد. در غیراینصورت SPH تعریف نشده بوده و نباید و نمی‌توان از آن استفاده کرد. در مثال‌های زیر فرض می‌کنیم که بیش از ٢٥٦ بایت حافظه در اختیار داریم.
برای ایجاد پشته، اشاره‌گر پشته را به بالاترین آدرس موجود در SRAM تنظیم می‌کنیم (در مثال ما پشته از بالا به سمت پایین پیش می‌رود، به سمت آدرس‌های پایین‌تر!).

```
.DEF MyPreferredRegister = R16
        LDI MyPreferredRegister, HIGH(RAMEND) ; Upper byte
        OUT SPH,MyPreferredRegister ; to stack pointer
        LDI MyPreferredRegister, LOW(RAMEND) ; Lower byte
        OUT SPL,MyPreferredRegister ; to stack pointer
```

بدیهی است که مقدار RAMEND وابسته به نوع پردازنده است. این مقدار در فایل ضمیمه(INCLUDE) مربوط به هر نوع پردازنده تعریف شده است. فایل 8515def.inc شامل خط زیر است:

```
.equ RAMEND =$25F ; Last On-Chip SRAM Location
```

فایل 8515def.inc با استفاده از راهنمای اسمبلر

```
.INCLUDE "C:\somewhere\8515def.inc"
```

در ابتدای کد برنامهٔ اسمبلر ضمیمه شده است.

پس اکنون پشته را تعریف کرده‌ایم و دیگر لازم نیست هیچگونه نگرانی درباره اشاره‌گر پشته داشته باشیم، زیرا ایجاد تغییرات و دستکاری در این اشاره‌گر به طور خودکار صورت می‌گیرد.

استفاده از پشته

استفاده از پشته ساده است. محتویات ثبات‌ها به صورت زیر بر روی پشته قرار می‌گیرد:

```
PUSH MyPreferredRegister ; Throw that value
```

اینکه این مقدار به کجا می‌رود اصلاً مهم نیست. لازم نیست نگران این باشیم که مقدار اشاره‌گر پشته پس از عمل PUSH کاهش یافته و تغییر کرده است. اگر دوباره به مقدار PUSH شده نیاز داشته باشیم، تنها کافی است دستورالعمل زیر را اضافه کنیم:

```
POP MyPreferredRegister ; Read back the value
```

با دستور POP فقط آخرین مقداری را که بر روی پشته قرار داده شده بدست می‌آوریم. PUSH کردن و POP کردن ثبات‌ها زمانی مفید و بامعنی است که:

- در چند خط بعدی برنامه دوباره به آن مقدار نیاز داشته باشیم،
- تمام ثبات‌ها در حال استفاده باشند، و
- ذخیره این مقدار در هیچ محل دیگری امکان‌پذیر نباشد.

در صورت عدم وجود این شرایط، استفاده از پشته برای صرفه‌جویی در استفاده از ثبات‌ها کاری بیهوده بوده و فقط وقت پردازنده را تلف می‌کند.

استفاده از پشته در زیرروال‌ها، که باید به محلی از برنامه که زیرروال را فراخوانی کرده است برگردید، مفهوم بیشتری پیدا می‌کند. در این حالت کد برنامهٔ فراخواننده، آدرس برگشت (مقدار جاری شمارنده برنامه) را بر روی پشته قرار داده و به زیرروال مورد نظر پرش می‌کند. پس از اجرا، زیرروال آدرس برگشت را از روی پشته برداشته و در شمارنده برنامه قرار می‌دهد. اجرای برنامه دقیقاً از اولین دستور بعد از دستورالعمل فراخوانی ادامه می‌یابد:

```
RCALL Somewhat ; Jump to the label somewhat
[...] here we continue with the program.
```

در اینجا عمل پرش به برچسب Somewhat که جایی در برنامه قرار دارد انجام شده است،

```
Somewhat: ; this is the jump address
[...] Here we do something
[...] and we are finished and want to jump back to the calling location:
RET
```

با اجرای دستورالعمل RCALL، شمارنده برنامه که قبلاً افزایش یافته است، به عنوان یک آدرس ۱۶ بیتی با دو بار عمل PUSH بر روی پشته قرار می‌گیرد. با رسیدن به دستورالعمل RET، محتویات پیشین شمارنده برنامه با استفاده از دو بار عمل POP بازگردانده شده و اجرا از محل قبلی ادامه می‌یابد.

لازم نیست نگران آدرسی از پشته که شمارنده در آنجا قرار داده شده است باشید. این آدرس به طور خودکار تولید می‌شود. حتی اگر در داخل آن زیرروال، زیرروال دیگری را فراخوانی کنید، پشته به طور صحیح عمل می‌کند. این کار دو آدرس برگشت بر روی پشته قرار می‌دهد؛ اولین آدرس را زیرروال داخلی و آدرس باقیماندهٔ بعدی را زیرروال فراخواننده از پشته خارج می‌کند. تا زمانی که SRAM به اندازه کافی موجود باشد، همه چیز درست کار می‌کند.

بکارگیری وقفه‌های سخت‌افزاری بدون استفاده از پشته غیرممکن است. وقفه‌ها اجرای عادی برنامه را صرف‌نظر از محل جاری اجرای برنامه متوقف می‌کنند. بعد از اجرای روالِ مربوط به سرویس معینی در پاسخ به آن وقفه، اجرای برنامه باید به محل پیشین قبل از وقوع وقفه بازگردد. اگر پشته قادر به ذخیره آدرس‌های برگشت نبود این کار غیرممکن می‌شد.

مزایای زیاد وجود پشته برای وقفه‌ها دلیلی است بر اینکه چرا حتی کوچک‌ترین AVRهای فاقد SRAM حداقل دارای یک پشته سخت‌افزاری بسیار کوچک هستند.

خطاهای کار با پشته

برای یک مبتدی که برای اولین بار از پشته استفاده می‌کند خطاهای زیادی ممکن است پیش بیاید:
یک خطای بسیار نادر، استفاده از پشته بدون تنظیم اشاره‌گر پشته است. چون در شروع اجرای برنامه این اشاره‌گر برابر صفر تنظیم می‌شود، اشاره‌گر به ثبات R0 اشاره می‌کند. PUSH کردن یک بایت باعث نوشته شدن بر آن ثبات شده و محتویات قبلی آن را از بین می‌برد. یک PUSH دیگر بر روی پشته، در آدرس 0xFFFF که یک محل نامعین است، می‌نویسد(البته اگر در آن آدرس SRAM خارجی نداشته باشید). اجرای RCALL و RET به آدرس نامشخصی در حافظه برنامه بازگشت می‌کند. مطمئن باشید که هیچ پیغام خطایی، مانند یک پنجره پیغام که چیزی شبیه "دسترسی غیرمجاز به محل حافظه xxxx" بگوید، وجود نخواهد داشت که به شما این خطا را هشدار دهد.
یکی دیگر از موارد وقوع خطا این است که POP کردن یک مقدار از قبل PUSH شده را فراموش کنید، و یا بدون PUSH کردن یک مقدار، عمل POP انجام دهید.
در موارد بسیار معدودی، پشته به اولین محل پایینی SRAM سرریز می‌شود. این حالت به دلیل وجود فراخوانی‌های بازگشتی بی‌پایان پیش می‌آید. پس از رسیدن به پایین‌ترین محل در حافظهٔ SRAM، PUSHهای بعدی بر روی پورت‌ها(0x005F تا 0x0020)، و سپس بر روی ثبات‌ها(0x001F تا 0x0000) می‌نویسد. اگر این حالت ادامه یابد اتفاقات جالب و غیرقابل پیش‌بینی در سخت‌افزار تراشه اتفاق می‌افتد. از این خطا اجتناب کنید، زیرا حتی می‌تواند سخت‌افزار شما را از بین ببرد!

انشعاب و تغییر مسیر برنامه

در این بخش تمام دستورالعمل‌هایی را که مسیر اجرای برنامه را کنترل می‌کنند مورد بررسی قرار خواهیم داد. بحث ما از لحظه روشن شدن پردازنده آغاز شده و به پرش‌ها، وقفه‌ها و غیره می‌پردازد.

کنترل اجرای ترتیبی برنامه

در طول عمل ریست (Reset) چه اتفاقی می‌افتد؟

هنگامی‌که منبع تغذیه ورودی یک AVR وصل شده و پردازنده کارش را آغاز می‌کند، عمل ریست توسط سخت‌افزار فعال می‌شود. در طول عملیات ریست، مقدار شمارنده گام‌های برنامه صفر خواهد شد. اجرای دستورالعمل‌ها همواره از این آدرس آغاز می‌شود و بنابراین اولین کلمه کد اجرایی باید در این محل قرار گرفته باشد. البته این آدرس نه فقط هنگام روشن شدن پردازنده، بلکه در حالت‌های دیگری نیز فعال می‌شود:

- با اعمال ریست خارجی روی پایه ریست، عملیات Restart اجرا می‌شود.
- اگر شمارنده Watchdog به مقدار حداکثر خود برسد، عملیات ریست را فعال می‌کند. شمارنده Watchdog یک ساعت داخلی است که باید هر از چند گاهی توسط برنامه ریست شود، در غیر اینصورت پردازنده را Restart می‌کند.
- شما می‌توانید عمل ریست را با پرش مستقیم به آن آدرس فراخوانی کنید (بخش مربوط به پرش را در ادامه ببینید).

سومین حالت، یک ریست واقعی نیست زیرا عمل خودکار مقداردهی اولیه به ثبات‌ها و پورت‌ها انجام نمی‌شود. بنابراین در حال حاضر به آن فکر نکنید.

دومین حالت، یعنی ریست توسط Watchdog، نخست باید توسط برنامه فعال شود. این گزینه به طور پیش فرض غیرفعال است. فعال کردن Watchdog از طریق نوشتن بر پورت Watchdog صورت می‌گیرد. برای صفرکردن دوبارهٔ شمارنده Watchdog باید دستور زیر را اجرا کرد:

```
WDR
```

با اجرای عمل ریست، که طی آن ثبات‌ها و پورت‌ها مقداردهی اولیه می‌شوند، کد موجود در آدرس 0000 به صورت یک کلمه کامل به بخش اجرایی پردازنده خوانده شده و اجرا می‌شود. در حین اجرای این دستور، شمارندهٔ برنامه از قبل یک واحد افزایش یافته و کلمه بعدی کد به داخل بافر کد خوانده شده است (عمل واکشی دستورات هنگام اجرا: Fetch During Execution). اگر فرمان اجرا شده نیاز به پرش به محل دیگری از برنامه نداشته باشد، فرمان بعدی بلافاصله اجرا می‌شود. به این دلیل است که AVRها دستورات را بسیار سریع اجرا می‌کنند؛ هر سیکل ساعت یک دستور را اجرا می‌کند (در صورتی که پرش اتفاق نیافتد).

اولین دستورِ هر برنامهٔ اجرایی همواره در آدرس 0000 قرار می‌گیرد. برای اینکه به کامپایلر (برنامه اسمبلر) بگوییم که کد برنامه ما از حالا و از این محل شروع می‌شود، راهنمای اسمبلر ویژه‌ای را می‌توان در ابتدای کد برنامه، قبل از محل نوشتن اولین دستور، قرار داد:

```
.CSEG
.ORG 0000
```

اولین راهنما به کامپایلر می‌گوید که وارد بخش کد شده است. تمام دستورات بعدی به عنوان کد برنامه ترجمه شده و به بخش حافظه برنامهٔ پردازنده نوشته می‌شوند. سگمنت قابل استفاده دیگر، بخش EEPROM تراشه است، جایی که می‌توانید بایت‌ها یا کلمات را در آن بنویسید.

.ESEG

سومین سگمنت، بخش SRAM تراشه است.

.DSEG

برخلاف محتویات EEPROM، که هنگام برنامه‌ریزی تراشه واقعاً به داخل EEPROM نوشته می‌شود، محتویات سگمنت DSEG به داخل تراشه نوشته نمی‌شود. از این سگمنت فقط هنگام اسمبل‌کردن برای محاسبه صحیح آدرس برچسب‌ها استفاده می‌شود.

راهنمای ORG به کار رفته در بالا مخفف کلمه Origin (محل) بوده و برای دستکاری آدرس‌های داخل سگمنت کد، یعنی محل قرارگیری کلمات ترجمه شده به کار می‌رود. از آنجا که برنامه‌های ما همواره از آدرس 0x0000 آغاز می‌شوند، راهنماهای CSEG/ORG غیرضروری هستند و می‌توانید بدون اینکه خطایی در برنامه پیش آید از آنها صرف‌نظر کنید. می‌توانستیم اجرا را از آدرس 0x0100 شروع کنیم اما از آنجا که پردازنده اجرای دستورات را از آدرس 0x0000 شروع می‌کند، این کار ما معقول و منطقی نخواهد بود. برای قراردادن یک جدول در محل دقیق و مشخصی از سگمنت کد، می‌توانید از ORG استفاده کنید. اگر در برنامه خود می‌خواهید بعد از تعریف‌های اولیه با راهنماهای DEF. و EQU.، از علامت مشخصی به عنوان محل شروع کدهای برنامه استفاده کنید، ترکیب CSEG/ORG را به کار ببرید، حتی اگر استفاده از آن در آنجا ضروری نباشد.

از آنجا که اولین کلمهٔ کد همواره در آدرس صفر قرار می‌گیرد، این محل را بردار ریست نیز می‌نامند. مکان‌های بعد از بردار ریست در فضای حافظهٔ برنامه، یعنی آدرس‌های 0x0001، 0x0002 و غیره، بردارهای وقفه هستند. آن‌ها محل‌هایی هستند که در صورت فعال بودن یک وقفه داخلی یا خارجی و رخ دادن آن، کنترل اجرا به آنجا منتقل می‌شود. این محل‌ها که بردار نامیده می‌شوند، برای هر نوع پردازنده متفاوت بوده و به سخت‌افزار داخلی آن بستگی دارد (بخش‌های بعدی را ببینید). دستوراتی که به چنین وقفه‌هایی واکنش نشان می‌دهند باید در محل صحیح بردار مربوطه قرار داده شوند. اگر از وقفه‌ها استفاده کنید اولین کلمه کد که در محل بردار ریست قرار می‌گیرد می‌بایست یک دستور پرش باشد تا از روی بردارهای دیگر پرش کند. هر بردار وقفه باید حاوی یک دستور پرش به روال سرویس وقفه (Interrupt Service Routine) باشد. به عنوان یک مثال، آغاز یک برنامه می‌تواند به شکل زیر باشد:

```
.CSEG
.ORG 0000
        RJMP Start
        RJMP IntServRout1
[...] here we place the other interrupt vector commands
[...] and here is a good place for the interrupt service routines themselves
Start: ; This here is the program start
[...] Here we place our main program
```

فرمان RJMP سبب پرش به برچسب :Start که چند خط بعد از آن قرار دارد می‌شود. یادآوری می‌کنیم که برچسب‌ها همواره از ستون اول هر خط برنامه شروع شده و به یک : ختم می‌شوند. برچسب‌هایی که این شرایط را برآورده نکنند در اکثر کامپایلرها به عنوان خطای جدی محسوب نمی‌شوند. برچسب‌های تعریف نشده پیغام خطای ("Undefined label" – "برچسب تعریف نشده") را تولید کرده و عمل کامپایل متوقف می‌شود.

اجرای خطی برنامه و انشعاب

اگر چیزی اجرای ترتیبی برنامه را تغییر ندهد، اجرای برنامه همواره به صورت خطی پیش خواهد رفت. منظور از این تغییرات، اجرای یک وقفه یا دستورات پرشی است.

عمل پرش در اکثر مواقع براساس شرایط خاصی صورت می‌گیرد که آن را پرش شرطی می‌نامند. به عنوان یک مثال، فرض کنید می‌خواهیم با استفاده از ثبات‌های R1 تا R4 یک شمارنده ۳۲ بیتی بسازیم. بایت با کمترین ارزش در R1 یک

واحد افزایش داده می‌شود. اگر با انجام این کار، ثبات R1 سرریز شود (0 = 1 + 255)، باید ثبات R2 را به همین نحو افزایش دهیم. اگر R2 سرریز شود باید R3 را افزایش دهیم، و به همین ترتیب.

افزودن یک واحد به ثبات، توسط دستورالعمل INC انجام می‌شود. اگر با اجرای INC R1 سرریز اتفاق بیافتد بیت صفر (Zero Bit) در ثبات وضعیت یک می‌شود (به این معنا که نتیجه عملیات صفر بوده است). بیت رقم نقلی (Carry) در ثبات وضعیت که معمولاً با وقوع سرریز یک می‌شود، با اجرای INC بدون تغییر باقی می‌ماند. این کار برای به اشتباه انداختن افراد تازه‌کار نیست بلکه در عوض از رقم نقلی برای مقاصد دیگری استفاده می‌شود. در این حالت، بیت Zero یا پرچم Zero برای تشخیص سرریز کافی است. اگر سرریز اتفاق نیافتد عمل شمارش را بدون تغییر ادامه می‌دهیم.

اگر بیت Zero، یک شود باید ثبات‌های دیگر را افزایش دهیم. برخلاف آنچه که انتظار دارید نام دستور پرشی که باید استفاده کنیم، BRNZ نیست بلکه BRNE (BRanch if Not Equal) است. این هم نوعی سلیقه در نامگذاری...

بنابراین، کل توالی دستورات شمارش برای این شمارندهٔ ۳۲ بیتی باید به صورت زیر باشد:

```
    INC R1
    BRNE GoOn32
    INC R2
    BRNE GoOn32
    INC R3
    BRNE GoOn32
    INC R4
GoOn32:
```

کل کار همین است. یک چیز بسیار ساده. شرط متضاد BRNE، BREQ یا Branch if Equal می‌باشد.

لیست بیت‌های وضعیت (پرچم‌های پردازنده) که حین اجرای یک دستور تغییر می‌کنند، در جداول دستورالعمل‌ها آورده شده است. فهرست دستورات را ببینید. از دیگر بیت‌های وضعیت نیز می‌توانید مشابه بیت Zero، برای واکنش به شرایط مختلف، به شکل زیر استفاده کنید:

```
BRCC label/BRCS label; Carry-flag 0 or 1
BRSH label; Equal or greater
BRLO label; Smaller
BRMI label; Minus
BRPL label; Plus
BRGE label; Greater or equal (with sign bit)
BRLT label; Smaller (with sign bit)
BRHC label/BRHS label; Half overflow flag 0 or 1
BRTC label/BRTS label; T-Bit 0 or 1
BRVC label/BRVS label; Two's complement flag 0 or 1
BRIE label/BRID label; Interrupt enabled or disabled
```

اگر شرط مورد نظر محقق شود، عمل پرش انجام می‌شود. نگران نباشید، اکثر این دستورات به ندرت استفاده می‌شوند. برای یک تازه‌کار، فقط بیت‌های صفر (Zero) و رقم نقلی (Carry) مهم هستند.

زمان‌بندی حین اجرای برنامه

همانگونه که در بالا به آن اشاره شد، مدت زمان مورد نیاز برای اجرای یک دستورالعمل برابر با یک سیکل ساعت پردازنده است. اگر پردازنده در فرکانس ساعت ٤ MHz کار کند، در اینصورت اجرای هر دستور نیاز به ١ / ٤ µs یا ۲۵۰ns، و در فرکانس ساعت ١٠ MHz، تنها به ١٠٠ ns نیاز خواهد داشت. مدت زمان مورد نیاز دقیقاً برابر با ساعت xtal است. اگر احتیاج به زمان‌بندی دقیق داشته باشید، AVR بهترین راه حل برای مشکل شماست. توجه داشته باشید که تعداد اندکی از دستورات وجود دارند که برای اجرا به دو یا بیشتر سیکل ساعت نیاز دارند، مانند دستورات انشعاب (اگر انشعاب رخ بدهد) یا عمل خواندن/نوشتن SRAM. برای جزئیات بیشتر جدول دستورات را ببینید.

برای ایجاد زمان‌بندی دقیق باید قابلیتی وجود داشته باشد که کاری جز ایجاد تأخیر در اجرای برنامه انجام ندهد. شما ممکن است از دستورات دیگری که عمل خاصی انجام نمی‌دهند استفاده کنید، اما رایج‌ترین روش، استفاده از دستور NOP یا No Operation است. این دستور، بی اثرترین دستور موجود است:

```
NOP
```

این دستور هیچ عملی انجام نمی‌دهد اما به میزان یک سیکل، وقت پردازنده را تلف می‌کند. در فرکانس ساعت ٤ MHz برای اتلاف زمان ۱ µs تنها نیاز به چهارتا از این دستور داریم. هیچ مطلب ناگفته دیگری در مورد دستور NOP وجود ندارد. برای ساخت یک مولد سیگنال (Signal Generator) با فرکانس ۱ KHz، لازم نیست چهار هزارتا از این دستورات را به کد برنامه اضافه کنیم، بلکه از یک شمارنده نرم‌افزاری و برخی دستورات انشعابی استفاده می‌کنیم. شمارنده می‌تواند یک ثبات ۸ بیتی باشد که با دستورالعمل DEC مقدار آن کاهش پیدا می‌کند، مثلاً به صورت زیر:

```
        CLR R1
Count:
        DEC R1
        BRNE Count
```

از شمارنده ١٦ بیتی نیز می‌توان برای ایجاد تأخیر به میزان دقیق، به شکل زیر استفاده کرد:

```
        LDI ZH,HIGH(65535)
        LDI ZL,LOW(65535)
Count:
        SBIW ZL,1
        BRNE Count
```

اگر از ثبات‌های بیشتری برای ساخت شمارنده‌های تودرتو استفاده کنید می‌توانید به هر میزان تأخیر دلخواهی دست پیدا کنید. تأخیر ایجاد شده به این طریق کاملاً دقیق است، حتی بدون استفاده از تایمر سخت‌افزاری.

ماکروها و اجرای برنامه

اغلب اوقات مجبور می‌شوید توالی دستورات یکسان یا مشابهی را در محل‌های مختلفی از کد منبع برنامه‌تان بنویسید. اگر نمی‌خواهید آن‌ها را یک بار نوشته و از طریق فراخوانی زیرروال به آن پرش کنید، می‌توانید از ماکرو برای اجتناب از زحمت نوشتن چندین بارهٔ توالی دستورات مشابه استفاده کنید. ماکروها توالی دستوراتی هستند که برای یک بار ساخته و آزمایش شده، و از طریق نام ماکرو به داخل کد برنامه افزوده می‌شوند. به عنوان یک مثال، فرض کنید چندین بار نیاز به ایجاد تأخیر در اجرای برنامه به میزان ۱ µs در فرکانس ساعت ٤ MHz داریم. بنابراین جایی در کد برنامه یک ماکرو تعریف می‌کنیم:

```
.MACRO Delay1
        NOP
        NOP
        NOP
        NOP
.ENDMACRO
```

تعریف ماکرو به این شکل هیچ کدی تولید نمی‌کند و به اصطلاح، خاموش (ساکت) است. با فراخوانی ماکرو از طریق نام آن، کد تولید می‌شود:

```
[...] somewhere in the source code
        Delay1
[...] code goes on here
```

این کار سبب قرارگرفتن چهار دستورالعمل NOP در داخل کد برنامه در آن محل می‌شود. یک Delay1 دیگر، چهار دستور NOP دیگر اضافه می‌کند.

هنگام فراخوانی یک ماکرو از طریق نام آن می‌توانید پارامترهایی را برای ایجاد تغییرات در کد تولید شده اضافه کنید. اما این روش فراتر از سطحی است که یک مبتدی باید درباره ماکروها بداند.

زیرروال‌ها (Subroutine)

برخلاف ماکروها، یک زیرروال باعث صرفه‌جویی در استفاده از فضای حافظه می‌شود. توالی کد مربوطه تنها یک بار در داخل کد برنامه ذخیره شده و از هر جایی در داخل کد برنامه فراخوانی می‌شود. برای اطمینان از ادامه اجرای برنامه پس

از فراخوانی زیرروال، باید به فراخوانندهٔ زیرروال بازگشت کنید. برای ایجاد تأخیری به اندازه ۱۰ سیکل، به زیرروال زیر نیاز دارید:

```
Delay10:
        NOP
        NOP
        NOP
        RET
```

زیرروال‌ها همواره با یک برچسب شروع می‌شوند، در اینجا با برچسب :Delay10، در غیراینصورت نمی‌توانید به آنها پرش کنید. سه NOP و یک دستور RET به دنبال هم قرار گرفته‌اند. اگر تعداد سیکل‌های لازم برای اجرای این زیرروال را بشمارید فقط ۷ سیکل را تشخیص می‌دهید (۳ سیکل برای NOPها، ٤ سیکل برای RET). ۳ سیکل باقیمانده مربوط به فراخوانی زیرروال است:

```
[...] somewhere in the source code:
        RCALL Delay10
[...] further on with the source code
```

RCALL یک فراخوانی نسبی است. این فراخوانی به صورت پرش نسبی در نظرگرفته می‌شود. فاصله نسبی از روال فراخوانی کننده تا زیرروال، توسط کامپایلر محاسبه می‌شود. دستورالعمل RET از طریق یک پرش، کنترل اجرا را به روال فراخوانی کننده بازمی‌گرداند. توجه داشته باشید که قبل از استفاده از فراخوانی‌های زیرروال، باید اشاره‌گر پشته را تنظیم کرده باشید (مبحث پشته (Stack) را ملاحظه کنید)، زیرا آدرس برگشت توسط دستور RCALL بر روی پشته قرار می‌گیرد.

اگر بخواهید مستقیماً به محل دیگری در داخل کد برنامه پرش کنید باید دستورالعمل پرش را به کار ببرید:

```
[...] somewhere in the source code
        RJMP Delay10
Return:
[...] further on with source code
```

در این حالت، روالی که به آن پرش کرده‌اید نمی‌تواند از دستور RET استفاده کند. جهت بازگشت به محلی از برنامه که فراخوانی از آنجا صورت گرفته، لازم است برچسب دیگری را اضافه کرده و روال فراخوانی شده عمل پرش به این برچسب را انجام دهد. پرش به این شکل شباهتی به فراخوانی یک زیرروال ندارد زیرا نمی‌توانید این روال را از محل‌های مختلف کد برنامه فراخوانی کنید.

RCALL و RJMP دستورات انشعاب غیرشرطی هستند. برای پرش به محلی از برنامه براساس شرایطی خاص، باید این شرایط را با دستورات انشعابی ترکیب کنید. فراخوانی شرطی یک زیرروال با دستورات زیر به خوبی قابل انجام است. اگر می‌خواهید زیرروالی را براساس مقدار جاری بیت مشخصی از یک ثبات فراخوانی کنید، توالی دستورات زیر را به کار ببرید:

```
        SBRC R1,7 ; Skip the next instruction if bit 7 is 0
        RCALL UpLabel ; Call that subroutine
```

دستورالعمل SBRC به این صورت تعبیر می‌شود: «از دستور بعدی صرف‌نظر کن اگر بیت ۷ در ثبات R1 صفر است». تنها در صورتی دستورالعمل RCALL برچسب :UpLabel را فراخوانی می‌کند که بیت ۷ ثبات R1 برابر یک باشد، زیرا اگر صفر باشد از دستورالعمل بعدی صرف‌نظر می‌شود. اگر بخواهید این زیرروال را در حالتی که مقدار این بیت صفر است فراخوانی کنید، از دستورالعمل مشابه، یعنی SBRS استفاده کنید. دستوری که به دنبال دستور SBRS/SBRC می‌آید می‌تواند یک دستور تک کلمه‌ای یا دو کلمه‌ای باشد. خود پردازنده تشخیص می‌دهد که چند کلمه را باید رد کند. توجه کنید که در این صورت زمان‌های اجرا متفاوت خواهد بود. نمی‌توان این دستورات را برای پرش از روی بیش از یک دستور متوالی به کار برد.

اگر بخواهید در صورت یکسان بودن مقادیر دو ثبات، از دستور بعدی صرف‌نظر شود، می‌توانید از دستور غیرعادی زیر استفاده کنید:

```
CPSE R1,R2 ; Compare R1 and R2, skip if equal
RCALL SomeSubroutine ; Call SomeSubroutine
```

از این دستور به ندرت استفاده می‌شود. اگرتازه‌کار هستید فعلاً آن را فراموش کنید. اگر بخواهید براساس مقدار بیت خاصی از یک پورت، از دستورالعمل بعدی صرف‌نظر شود از دستورات زیر، SBIC و SBIS، استفاده کنید که به صورت «صرف‌نظر کن در صورتی که بیت مورد نظر در فضای I/O صفر (یا یک) است» خوانده شده و به شکل زیر به کار می‌رود:

```
SBIC PINB,0 ; Skip if Bit 0 on port B is 0
RJMP ATarget ; Jump to the label ATarget
```

دستورالعمل RJMP تنها در صورتی اجرا می‌شود که بیت صفرِ پورت B به ولتاژ High (یک منطقی) متصل باشد. در این رابطه مطلبی است که ممکن است باعث سردرگمی افراد مبتدی شود. دسترسی به بیت‌های پورت‌ها تنها به نیمه پایینی پورت‌ها محدود بوده و نمی‌توان از ۳۲ پورت بالایی در اینجا استفاده کرد.
اکنون به کاربرد جالبی از دستورات انشعاب اشاره می‌کنیم که مخصوص حرفه‌ای‌ها است. اگر یک تازه‌کار هستید از این قسمت صرف‌نظر کنید. یک انتخاب کنندهٔ ٤ بیتی (٤ کلیدی) را در نظر بگیرید که به پورت B متصل است. براساس حالات مختلفی که این ٤ بیت می‌توانند داشته باشند می‌خواهیم به ١٦ محل مختلف در برنامه پرش کنیم. یک راه این است که مقدار پورت را بخوانیم و با استفاده از چندین دستورالعمل انشعابی، به محل مربوطه پرش کنیم. به عنوان روشی دیگر، می‌توانید جدولی از آدرس‌های ١٦ بیتی به شکل زیر تشکیل دهید:

```
MyTab:
    RJMP Routine1
    RJMP Routine2
    [...]
    RJMP Routine16
```

سپس در برنامه، آدرس جدول را در داخل ثبات اشاره‌گر Z قرار می‌دهیم:

```
LDI ZH,HIGH(MyTab)
LDI ZL,LOW(MyTab)
```

و حالت جاری پورت B را (که در ثبات R16 قرار داده شده است) به این آدرس اضافه می‌کنیم:

```
    ADD ZL,R16
    BRCC NoOverflow
    INC ZH
NoOverflow:
```

اکنون می‌توانیم به این محل از جدول پرش کنیم؛ یا به صورت فراخوانی یک زیرروال:

```
ICALL
```

و یا به صورت انشعاب بدون بازگشت:

```
IJMP
```

پردازنده محتویات زوج ثبات Z را به داخل شمارنده برنامه کپی کرده و ادامه عملیات را از آنجا از سر می‌گیرد. آیا این کار بهتر و زیرکانه‌تر از چندین مرتبه انشعاب نیست؟

وقفه‌ها و اجرای برنامه

اغلب اوقات لازم است که به شرایط سخت‌افزاری یا دیگر رویدادها، مثلاً تغییر در مقدار یک پین ورودی، واکنش نشان دهیم. چنین عکس‌العملی را می‌توانید با نوشتن یک حلقه و بررسی وقوع تغییر در پین، برنامه‌ریزی کنید. این روش، Polling (نمونه‌برداری) نامیده شده و به مثابه یک زنبور عسل است که در حال گردش بر روی یک دایره و جستجوی گل‌های جدید است. اگر هیچ عمل دیگری برای انجام دادن وجود نداشته و زمان عکس‌العمل نیز مهم نباشد، می‌توانید از این روش در پردازنده استفاده کنید. این روش را نمی‌توان برای تشخیص پالس‌های کوتاهِ با دوام کمتر از یک میکروثانیه به کار برد. در اینگونه موارد نیاز به برنامه‌ریزی وقفه دارید.

وقفه توسط برخی شرایط سخت‌افزاری رخ می‌دهد. شرط مورد نظر نخست باید فعال شود زیرا تمام وقفه‌های سخت‌افزاری هنگام ریست به طور پیش فرض غیرفعال هستند. در ابتدا بیت‌های پورتِ مربوط به فعال‌سازی قابلیت وقفهٔ بخش مورد نظر تنظیم می‌شود. در ثبات وضعیت پردازنده بیتی بنام پرچم فعال‌سازی وقفه (Interrupt Enable Flag) وجود دارد که پردازنده را قادر به پاسخگویی به وقفه‌ها می‌کند. فعال‌کردن قابلیت پاسخگویی عمومی به وقفه‌ها، با دستور زیر صورت می‌گیرد:

```
SEI ; Set Int Enable Bit
```

اگر شرط وقوع وقفه رخ بدهد، مثلاً وقوع تغییر در بیت یک پورت، پردازنده مقدار واقعی شمارنده برنامه را در پشته (که باید از قبل تنظیم شده باشد! مبحث آماده‌سازی اشاره‌گر پشته را در بخش پشته در قسمت توضیحات SRAM ببینید.) قرار می‌دهد. بدون این کار، پردازنده قادر به بازگشت به محل قبل از وقوع وقفه (که می‌تواند هر زمان و هر محلی از اجرای برنامه باشد) نیست. پس از آن، پردازنده به محل از پیش تعریف شده، یعنی بردار وقفه، پرش کرده و دستورات موجود در آنجا را اجرا می‌کند. به طور معمول دستور موجود در آنجا یک دستورالعمل پرش به روال سرویس وقفه، که جایی در داخل کد برنامه قرار دارد، است. محل بردارهای وقفه مختص نوع پردازنده بوده و بسته به اجزای سخت‌افزاری و شرایطی که باعث وقوع وقفه می‌شود متفاوت است. هر چه اجزای سخت‌افزاری و شرایط وقوع وقفه بیشتری وجود داشته باشد، تعداد بردارهای بیشتری موجود خواهد بود. بردارهای مختلفِ مربوط به برخی مدل‌های AVR در جدول زیر آورده شده است. (اولین بردار موجود در جدول، وقفه نیست بلکه بردار ریست است که برای آن هیچ عملی در رابطه با پشته انجام نمی‌شود!)

نام	آدرس بردار وقفه			عامل وقوع وقفه
	2313	2323	8515	
RESET	0000	0000	0000	ریست سخت‌افزاری، ریست آغاز به کار (Power-On)، ریست Watchdog
INT0	0001	0001	0001	تغییر لبه در پین INT0 خارجی
INT1	0002	-	0002	تغییر لبه در پین INT1 خارجی
TIMER1 CAPT	0003	-	0003	رویداد CAPTURE برای تایمر/شمارندهٔ ۱
TIMER1 COMPA	-	-	0004	Timer/Counter 1 = Compare value A
TIMER1 COMPB	-	-	0005	Timer/Counter 1 = Compare value B
TIMER1 COMP1	0004	-	-	Timer/Counter 1 = Compare value 1
TIMER1 OVF	0005	-	0006	سرریز تایمر/شمارندهٔ ۱
TIMER0 OVF	0006	0002	0007	سرریز تایمر/شمارندهٔ ۰
SPI STC	-	-	0008	اتمام ارسال سریال (وقفه SPI)
UART TX	0007	-	0009	وجود کاراکتر در بافر ورودی UART
UART UDRE	0008	-	000A	خالی بودن بافر خروجی UART
UART TX	0009	-	000B	اتمام ارسال داده توسط UART
ANA_COMP	-	-	000C	مقایسه کنندهٔ آنالوگ

توجه کنید که قابلیت واکنش به رویدادها برای مدل‌های مختلف، بسیار متفاوت است. آدرس‌ها متوالی هستند، اما برای مدل‌های مختلف یکسان نیستند. برای هر مدل خاص AVR از برگه‌های اطلاعاتی (Data Sheet) مربوط به آن کمک بگیرید.

بالاتر بودن یک بردار در این جدول نشانه بالاتر بودن اولویت (Priority) آن است. اگر وقفه‌های مربوط به دو یا چند جزء، به طور همزمان رخ دهد، بالاترین بردار که دارای آدرس بردار کوچک‌تری است بر بقیه غالب می‌شود. وقفهٔ پایین‌تر باید منتظر بماند تا وقفه بالاتر به طور کامل اجرا شود. به منظور جلوگیری از اجرای وقفه‌های دیگر در حین اجرای روال سرویس مربوط به یک وقفه، اولین وقفهٔ ایجاد شده، پرچم I (I-Flag) پردازنده را غیرفعال می‌کند. روال سرویس وقفه باید این پرچم را پس از اتمام کار دوباره فعال کند.

برای یک کردن دوبارهٔ بیت وضعیت I دو راه وجود دارد. روال سرویس می‌تواند با دستور زیر پایان یابد:

```
RETI
```

اینگونه بازگشت از روال وقفه، بیت I را پس از قرار دادن آدرس برگشت در شمارنده برنامه به حالت اول بازمی‌گرداند.

دومین روش فعال کردن بیت I، استفاده از دستورالعمل زیر است:

```
SEI ; Set Interrupt Enabled
RET ; Return
```

این روش با حالت استفاده از دستور RETI یکسان نیست، زیرا قبل از قرار دادن آدرس برگشت در شمارنده برنامه، وقفه‌های بعدی فعال شده‌اند. اگر وقفه دیگری منتظر اجرا باشد، اجرای آن قبل از برداشتن آدرس برگشت از روی پشته آغاز می‌گردد. دو یا چند آدرس در نتیجهٔ فراخوانی‌های تودرتو بر روی پشته قرار می‌گیرند. انتظار نمی‌رود خطایی اتفاق بیافتد اما انجام این کار یک ریسک غیرضروری است. بنابراین فقط از دستور RETI استفاده کنید تا از پرشدن غیرضروری پشته به این شکل جلوگیری شود.

بردار وقفه تنها می‌تواند حاوی یک دستور پرش نسبی به روال سرویس باشد. اگر وقفهٔ خاصی استفاده نشده و یا تعریف‌نشده باشد، می‌توانیم فقط یک دستور RETI در محل آن قرار دهیم که در این حالت یک وقفهٔ کاذب (شبه وقفه) اتفاق می‌افتد. در موارد معدودی واکنش به این وقفه‌ها کاملاً ضروری است. این وضعیت برای حالتی است که روال سرویس مربوطه، پرچم شرایط وقفهٔ وسیله جانبی را به طور خودکار ریست نمی‌کند. در اینجا یک دستور RETI می‌تواند وقفه‌های بی پایان را ریست کند. این حالت در برخی وقفه‌های مربوط به UART رخ می‌دهد.

از آنجا که پس از آغاز اجرای یک سرویس وقفه، از اجرای وقفه‌های با اولویت پایین‌تر جلوگیری می‌شود، تمام روال‌های سرویس وقفه باید تا حد امکان کوتاه باشند. اگر لازم است که از یک روال طولانی برای سرویس وقفه استفاده کنید، یکی از روش‌های زیر را به کار بگیرید. اولین روش این است که پس از انجام کارهای بسیار ضروری، وقفه‌ها را با دستور SEI در داخل روال سرویس وقفه فعال کنید. این روش چندان مناسب و معمول نیست. روش مناسب‌تر این است که پس از انجام اَعمال ضروری، برای واکنش‌های آهسته‌تر و غیرضروری، پرچمی را در یک ثبات، یک کنید و بلافاصله از روال وقفه خارج شوید.

روال‌های سرویس وقفه یک وظیفه بسیار جدی بر عهده دارند: اولین دستور همواره باید ثبات وضعیت را قبل از بکارگیری دستوراتی که ممکن است پرچم‌های ثبات وضعیت را تغییر دهند، در پشته ذخیره نماید. ممکن است برنامه اصلی که اجرای آن قطع شده است، درست در وضعیتی بوده باشد که می‌خواهد از یکی از پرچم‌های ثبات وضعیت برای یک تصمیم انشعاب استفاده کند، و سرویس وقفه، آن پرچم را به حالت دیگری تغییر بدهد. چیزهای جالبی اتفاق خواهد افتاد. بنابراین آخرین دستور قبل از RETI باید محتویات اصلی ثبات وضعیت را از پشته برداشته و در ثبات وضعیت قرار دهد. به دلیل مشابهی، تمام ثبات‌های به کار رفته در یک روال سرویس وقفه یا باید انحصاراً به آن کار اختصاص داده شده باشند و یا بر روی پشته ذخیره شده و در انتهای روال سرویس بازیابی شوند.هرگز در داخل روال سرویسِ وقفه، محتویات ثباتی را که جایی در برنامه اصلی به کار رفته است بدون بازیابی محتویات آن تغییر ندهید.

به دلیل وجود این قواعد پایه‌ای و اصلی، مثال کامل‌تر و ساخت‌یافته‌تری برای یک روال سرویس وقفه در اینجا آورده شده است.

```
.CSEG ; Code-Segment starts here
.ORG 0000 ; Address is zero
```

```
        RJMP Start ; The reset-vector on Address 0000
        RJMP IService ; 0001: first Int-Vector, INT0 service routine
[...] here other vectors

Start: ; Here the main program starts
[...] here is enough space for defining the stack and other things

IService: ; Here we start with the Interrupt-Service-Routine
        PUSH R16 ; save a register to stack
        IN R16,SREG ; read status register
        PUSH R16 ; and put on stack
[...] Here the Int-Service-Routine does something and uses R16
        POP R16 ; get previous flag register from stack
        OUT SREG,R16 ; restore old status
        POP R16 ; get previous content of R16 from the stack
        RETI ; and return from int
```

کمی پیچیده به نظر می‌رسد، ولی این کار برای استفادهٔ صحیح و بدون خطا از وقفه‌ها لازم است. اگر بتوانید ثبات R16 را انحصاراً برای استفاده در این روال سرویس وقفه اختصاص دهید، PUSH R16 و POP R16 را حذف کنید. از آنجا که امکان وقوع وقفه وجود ندارد (مگر اینکه شما در داخل روال، وقفه‌ها را فعال بکنید)، همه روال‌های سرویس وقفه می‌توانند از همین ثبات به طور مشترک استفاده نمایند.

این تمام مطالبی است که یک مبتدی باید بداند. مطالب دیگری در رابطه با وقفه‌ها وجود دارد، اما تا همین اندازه برای شروع کافی است.

محاسبات

در این بخش تمام دستورات لازم برای انجام محاسبات در زبان اسمبلی AVR را مورد بحث قرار می‌دهیم. این مطالب شامل سیستم‌های اعداد، تنظیم و پاک کردن بیت‌ها، شیفت و چرخش بیتی، جمع/تفریق/مقایسه و تبدیل قالب اعداد است.

سیستم‌های اعداد در اسمبلر

استفاده از قالب‌های عددی زیر در اسمبلر معمول است:

- اعداد کامل مثبت (بایت‌ها، کلمه‌ها و غیره)
- اعداد کامل علامت‌دار (اعداد صحیح)
- ارقام کد شده به صورت دودویی،BCD
- BCD فشرده
- اعداد در قالب ASCII

اعداد کامل مثبت (بایت‌ها، کلمه‌ها و غیره)

کوچک‌ترین عدد کاملی که در اسمبلر قابل استفاده می‌باشد، یک بایت یا هشت بیت است. این مقدار فضا می‌تواند اعداد بین ۰ تا ۲۵۵ را در خود نگه دارد. اندازه این بایت‌ها دقیقاً به اندازه یک ثبات MCU است. اعداد بزرگ‌تر باید براساس این قالب پایه، با بکارگیری بیش از یک ثبات، ساخته شوند. دو بایت، تشکیل یک کلمه (با گسترهٔ از ۰ تا ٦٥،٥٣٥)، سه بایت تشکیل یک کلمهٔ بلند (با گسترهٔ از ۰ تا ١٦،٧٧٧،٢١٥) و چهار بایت تشکیل یک کلمه مضاعف (با گسترهٔ از ۰ تا ٤،٢٩٤،٩٦٧،٢٩٥) را می‌دهند.

هر یک از بایت‌های یک کلمه یا یک کلمهٔ مضاعف را می‌توان در هر ثبات دلخواهی ذخیره کرد. اَعمال انجام شده بر روی این بایت‌ها در برنامه، به صورت بایت به بایت صورت می‌گیرد، و بنابراین نیازی به قرار دادن این بایت‌ها در یک ردیف نیست. برای تشکیل یک ردیف برای یک کلمه مضاعف، می‌توانیم آن را به این شکل ذخیره کنیم:

```
.DEF r16 = dw0
.DEF r17 = dw1
.DEF r18 = dw2
.DEF r19 = dw3
```

dw0 تا dw3 در ردیفی از ثبات‌ها قرار دارند. اگر بخواهیم به این کلمه مضاعف در ابتدای یک برنامه کاربردی، مقدار اولیه ای (مثلاً ٤،٠٠٠،٠٠٠) بدهیم، روش کار باید به صورت زیر باشد:

```
.EQU dwi = 4000000 ; define the constant
        LDI dw0,LOW(dwi) ; The lowest 8 bits to R16
        LDI dw1,BYTE2(dwi) ; bits 8 .. 15 to R17
        LDI dw2,BYTE3(dwi) ; bits 16 .. 23 to R18
        LDI dw3,BYTE4(dwi) ; bits 24 .. 31 to R19
```

با این عمل، عدد دهدهی dwi را به بخش‌های دودویی آن تجزیه کرده و در داخل چهار بسته یک بایتی قرار داده‌ایم. اکنون می‌توانیم این کلمه مضاعف را در محاسبات به کار ببریم.

اعداد علامت‌دار (اعداد صحیح)

گاهی اوقات، اما در موارد اندکی، نیاز به محاسبات با اعداد منفی دارید. یک عدد منفی با در نظر گرفتن باارزش‌ترین بیت یک بایت به عنوان بیت علامت، تعریف می‌شود. اگر این بیت، صفر باشد، عدد مثبت، و اگر ۱ باشد، عدد منفی است. اگر عدد مورد نظر منفی باشد، معمولاً بقیه عدد را به صورتی که است ذخیره نمی‌کنیم بلکه از مقدار مکمل آن استفاده می‌کنیم. منظور از مقدار مکمل این است که عدد ۱- به عنوان یک عدد صحیح تک بایتی به شکل 1000.0001 نوشته

نمی‌شود بلکه آن را به فرم 1111.1111 می نویسند. یعنی ۱ را از صفر کم کرده و سرریز به وجود آمده را در نظر نگیریم. اولین بیت، همان بیت علامت است که نشان‌دهنده منفی بودن عدد است. دلیل به کار بردن این شکل متفاوت (یعنی تفریق عدد منفی از صفر) ساده‌تر بودن فهم آن است: حاصل جمع ۱- (1111.1111) و ۱+ (0000.0001)، در صورت در نظر نگرفتن بیت سرریز ایجاد شده در حین عمل جمع (بیت نهم)، دقیقاً برابر صفر است.

بزرگ‌ترین عدد صحیح قابل استفاده در یک بایت، ۱۲۷+ (دودویی: 0,1111111) و کوچک‌ترین عدد، ۱۲۸- (دودویی: 1,0000000) می‌باشد. در زبان‌های برنامه‌نویسی دیگر، به این قالب عددی، عدد صحیح کوتاه (Short Integer) گفته می‌شود. اگر نیاز به گسترهٔ بزرگ‌تری از مقادیر داشته باشید می‌توانید بایت دیگری را اضافه کرده و یک مقدار صحیح معمولی (با گسترهٔ از ۳۲،۷۶۸- تا ۳۲،۷۶۷+) را تشکیل دهید. استفاده از چهار بایت،گسترهٔ مقادیر از ۲،۱۴۷،۴۸۳،۶۴۸- تا ۲،۱۴۷،۴۸۳،۶۴۷+ را در اختیار می‌گذارد که معمولاً آن را LongInt یا DoubleInt می‌نامند.

ارقام کد شده به صورت دودویی، BCD

قالب‌های اعداد کامل مثبت و علامت‌دار بحث شده در بالا، از فضای حافظهٔ اختصاص یافته به خود به طور مؤثری استفاده می‌کنند. قالب عددی دیگر، ذخیره اعداد دهدهی به صورت یک بایت برای هر رقم است. این قالب عددی دارای چگالی کمتری بوده ولی استفاده از آن راحت‌تر است. در این قالب عددی، هر رقم دهدهی به شکل دودویی آن در داخل یک بایت ذخیره می‌شود. هر یک از ارقام ۰ تا ۹ نیاز به چهار بیت (0000 تا 1001) دارند. چهار بیت بالایی بایت با صفر پر می‌شود که این کار فضای حافظه زیادی از بایت را به هدر می‌دهد. به طور مثال، برای نمایش عدد ۲۵۰ حداقل به سه بایت نیاز خواهیم داشت:

ارزش مکانی بیت	1	2	4	8	16	32	64	128
R16، رقم اول = 2	0	1	0	0	0	0	0	0
R17، رقم دوم = 5	1	0	1	0	0	0	0	0
R18، رقم سوم = 0	0	0	0	0	0	0	0	0

```
;Instructions to use:
LDI R16,2
LDI R17,5
LDI R18,0
```

می‌توانید محاسبات خود را با این اعداد انجام دهید، اما در اسمبلر این کار کمی پیچیده‌تر از محاسبه با مقادیر دودویی است. مزیت استفاده از این قالب عددی این است که به شما اجازهٔ استفاده از اعداد بزرگ با اندازه دلخواه را می‌دهد، البته تا زمانی که حافظه کافی برای ذخیره آنها موجود باشد. دقت محاسبات انجام شده با این قالب عددی کاملاً اهداف شما را ارضا خواهد کرد (اگر از AVR برای برنامه‌های کاربردی مالی و بانکی استفاده می‌کنید)، و به آسانی می‌توانید این اعداد را به رشته‌های کاراکتری تبدیل کنید.

BCD فشرده

اگر دو رقم دهدهی را در داخل یک بایت جای دهید نسبت به حالت قبل فضای حافظه کمتری را از دست می‌دهید. این روش، ارقام کد شده به صورت دودوییِ فشرده شده نامیده می‌شود. دو بخش یک بایت را نیبل‌های بالایی و پایینی[1] (نیم‌بایت‌های بالایی و پایینی) می‌نامند. معمولاً رقم با ارزش بالاتر در نیبل بالایی قرار می‌گیرد، زیرا این طرز نمایش مزایایی در انجام محاسبات به همراه دارد (دستورالعمل‌های مخصوص در زبان اسمبلی AVR). عدد دهدهی ۲۵۰ پس از تبدیل به قالب BCD فشرده، به شکل زیر خواهد بود:

1- Upper Nibble / Lower Nibble

بایت	ارقام	مقدار	1	2	4	8	1	2	4	8
۲	۳ و ٤	02	0	1	0	0	0	0	0	0
۱	۱ و ۲	50	0	0	0	0	1	0	1	0

```
; Instructions for setting:
LDI R17,0x02 ; Upper byte
LDI R16,0x50 ; Lower byte
```

برای تنظیم صحیح موقعیت ارقام در بایت‌ها، می‌توانید از طرز نمایش دودویی (0b...) یا طرز نمایش مبنای شانزده (0x...) برای تنظیم درست موقعیت بیت‌ها در نیبل‌های بالا و پایین استفاده کنید.

انجام محاسبات با اعداد BCD فشرده در مقایسه با حالت دودویی قدری پیچیده‌تر است. تبدیل به رشته‌های کاراکتری به همان سادگی تبدیل با اعداد BCD است. طول اعداد و دقت محاسبات، تنها محدود به فضای حافظه موجود است.

اعداد در قالب ASCII

ذخیره اعداد در قالب ASCII بسیار مشابه قالب BCD غیرفشرده است. ارقام ۰ تا ۹ با استفاده از معادل ASCII خود (ASCII = American Standard Code for Information Interchange[1]) ذخیره می‌شوند. ASCII قالبی است بسیار قدیمی‌که به منظور استفاده در تله‌تایپ‌ها (ماشین‌های تحریر راه دور) ایجاد و بهینه‌سازی شد، به ناچار برای استفاده در کامپیوترها بسیار پیچیده شد (آیا می‌دانید کاراکتری به نام EOT (End Of Transmission)[2] هنگامی‌که ابداع شد به چه معنی بود؟)، دارای گسترهٔ بسیار محدودی برای زبان‌های غیر از آمریکایی (US) بوده (فقط ۷ بیت به ازای هر کاراکتر)، و امروزه به دلیل همت کم برخی برنامه‌نویسان سیستم‌های عامل در استفاده از سیستم‌های کاراکتری کاراتر، هنوز در مخابرات به کار می‌رود. این سیستمِ سنتی تنها به واسطهٔ مجموعه کاراکترهای تله‌تایپ ۵ بیتی اروپایی به نام مجموعهٔ Baudot، و یا به خاطر استفاده از آن در کد مورس پابرجا مانده است.

در سیستم کد ASCII، رقم دهدهی ۰ با عدد ٤۸ (مبنای شانزده:0x30، دودویی:0b0011.0000) و رقم ۹ با عدد دهدهی ۵۷ (مبنای شانزده: 0x39، دودویی: 0b0011.1001) متناظر است. ASCII به گونه ای طراحی نشده است که اعداد در ابتدای مجموعهٔ کدها قرار گیرند، و پیش از آنها کاراکترهای کنترلی تله‌تایپ نظیر EOT که در بالا به آن اشاره شد، قرار گرفته‌اند. بنابراین برای تبدیل یک عدد BCD به ASCII باید عدد ٤۸ را به BCD اضافه کنیم (یا بیت‌های ٤ و ۵ آن را یک کنیم). اعداد در قالب ASCII و اعداد BCD برای ذخیره‌سازی به اندازهٔ حافظه یکسانی نیاز دارند. ذخیره عدد ۲۵۰ در یک مجموعه ثبات به صورت زیر خواهد بود:

```
LDI R18,'2'
LDI R17,'5'
LDI R16,'0'
```

معادل‌های ASCII این کاراکترها در ثبات‌ها نوشته می‌شود.

دستکاری بیتی

برای تبدیل یک رقم BCD به معادل ASCII آن باید بیت‌های ٤ و ۵ آن را یک کنیم. به عبارت دیگر باید این رقم BCD را با مقدار ثابت 0x30 در مبنای شانزده، OR کنیم. این کار در اسمبلر به شکل زیر انجام می‌شود:

```
ORI R16,0x30
```

اگر از قبل ثباتی داریم که حاوی مقدار مبنای شانزده 0x30 است، می‌توانیم برای تبدیل BCD، عمل OR را با این ثبات انجام دهیم:

```
OR R1,R2
```

عکس عمل تبدیل، یعنی تبدیل یک کاراکتر ASCI به معادل BCD نیز به همان سادگی است. دستورالعمل

```
ANDI R16,0x0F
```

۱- کد استاندارد آمریکایی برای تبادل اطلاعات ۲- پایان ارسال

چهار بیت پایینی (نیبل پایینی) را جدا می‌کند. توجه کنید که ORI و ANDI تنها با ثبات‌های بالاتر از R15 قابل استفاده هستند. بنابراین برای انجام این کار یکی از ثبات‌های R16 تا R31 را به کار ببرید!

اگر عدد مبنای شانزده 0x0F از قبل در ثبات R2 قرار داشته باشد، می‌توانید کاراکتر ASCII را با این ثبات AND کنید:

```
AND R1,R2
```

دستورات دیگرِ مربوط به دستکاری بیت‌های یک ثبات نیز محدود به ثبات‌های بالاتر از R15 هستند. آنها را می‌توان به شکل زیر به کار برد:

```
SBR R16,0b00110000 ; Set bits 4 and 5 to one
CBR R16,0b00110000 ; Clear bits 4 and 5 to zero
```

اگر لازم است یک یا چند بیت خاص از یک بایت، معکوس شود، می‌توانید از دستورات زیر استفاده کنید (که قابل استفاده با مقادیر ثابت نیستند):

```
LDI R16,0b10101010 ; Invert all even bits
EOR R1,R16 ; in register R1 and store result in R1
```

برای معکوس کردن تمام بیت‌های یک بایت که به آن مکمل یک (One's Complement) می‌گویند دستورالعمل زیر را به کار ببرید:

```
COM R1
```

این دستور محتویات ثبات R1 را معکوس کرده، صفرها را با یک و یک‌ها را با صفر جایگزین می‌کند. تفاوت آن با مکمل دو (Two's Complement) در این است که مکمل دو، یک عدد علامت‌دار مثبت را به مکمل منفی آن تبدیل می‌کند (با تفریق آن عدد از صفر). این کار با دستورالعمل زیر انجام می‌شود:

```
NEG R1
```

بنابراین نتیجهٔ حاصل از ۱+ (دهدهی: ۱) برابر ۱- (دودویی: 1.1111111) و نتیجهٔ حاصل از ۲+ برابر با ۲- (دودویی: 1.1111110) خواهد بود و به همین ترتیب.

علاوه بر دستکاری بیت‌های یک ثبات، امکان کپی کردن یک تک بیت با استفاده از بیت T ثبات وضعیت وجود دارد. با دستور

```
BLD R1,0
```

بیت T برابر با مقدار جاری بیت صفرِ ثبات R1 می‌شود. بیت T را می‌توان یک یا صفر کرده، و محتویات آن را به هر بیت موجود در هر ثباتی کپی کرد:

```
CLT ; clear T-bit, or
SET ; set T-bit, or
BST R2,2 ; copy T-bit to register R2, bit 2
```

شیفت و چرخش بیتی

شیفت و چرخش بیتی اعداد دودویی به معنای ضرب و تقسیم آنها در/ بر ۲ است. عمل شیفت شامل چندین زیردستورالعمل است.

ضرب در ۲ با انتقال کلیهٔ بیت‌های یک بایت به اندازه یک رقم دودویی به سمت چپ و نوشتن یک صفر در بیت با کمترین ارزش، به سادگی قابل انجام است. این عمل، شیفت منطقی به چپ نامیده می‌شود. مقدار پیشین بیت ۷ به داخل بیت Carry (رقم نقلی) در ثبات وضعیت انتقال می‌یابد.

```
LSL R1
```

عکس این عمل، یعنی تقسیم بر ۲، شیفت منطقی به راست نامیده می‌شود.

```
LSR R1
```

مقدار قبلی بیت ۷ که اکنون به محل بیت ٦ انتقال یافته است، با صفر پر می‌شود، در حالی که مقدار پیشین بیت صفر به داخل بیت Carry در ثبات وضعیت منتقل شده است. از بیت Carry می‌توان برای گردکردن عدد به سمت بالا و پایین استفاده کرد (اگر بیت Carry یک شد، یک واحد به نتیجه اضافه کنید). مثال زیر یک عدد را بر چهار تقسیم کرده و نتیجه را به سمت بالا گرد می‌کند:

```
        LSR R1 ; division by 2
        BRCC Div2 ; Jump if no round up
        INC R1 ; round up
Div2:
        LSR R1 ; Once again division by 2
        BRCC DivE ; Jump if no round up
        INC R1 ; Round Up
DivE:
```

بنابراین تقسیم بر مضارب ۲، با اعداد دودویی ساده است.

اگر اعداد صحیح علامت‌دار استفاده شوند، شیفت منطقی به راست ممکن است بیت علامت موجود در بیت شماره ۷ را تغییر دهد. دستورالعمل "شیفت ریاضی به راست[1]" یا ASR، بیت هفتم را بدون تغییر باقی گذاشته و ۷ بیت پایینی را با اضافه کردن یک صفر در محل بیت ٦ شیفت می‌دهد.

```
        ASR R1
```

در اینجا نیز مشابه شیفت منطقی، مقدار پیشین بیت صفر به داخل بیت Carry در ثبات وضعیت منتقل می‌شود.

ضرب یک کلمهٔ ١٦ بیتی در ۲ چگونه انجام می‌شود؟ باارزش‌ترین بیت بایتِ پایینی باید به داخل پایین‌ترین بیت بایتِ بالایی انتقال داده شود. در این مرحله، عمل شیفت، پایین‌ترین بیت را صفر خواهد کرد، اما لازم است که بیت Carry حاصل از شیفت بایت پایین را به داخل بیت صفر انتقال دهیم. این عمل، چرخش نامیده می‌شود. طی عمل چرخش، بیت Carry موجود در ثبات وضعیت به داخل بیت صفر منتقل شده، مقدار پیشین بیت ۷ به داخل Carry انتقال می‌یابد.

```
        LSL R1 ; Logical Shift Left of the lower byte
        ROL R2 ; ROtate Left of the upper byte
```

شیفت منطقی به چپ در اولین دستور، بیت ۷ را به داخل Carry منتقل کرده، دستورالعمل ROL آن را به داخل بیت صفرِ بایت بالایی می‌چرخاند. پس از اجرای دومین دستور، بیت Carry حاوی مقدار پیشین بیت ۷ خواهد بود. بیت Carry می‌تواند برای تشخیص وقوع سرریز (اگر محاسبه به صورت ١٦ بیتی انجام می‌شود) و یا چرخش آن به داخل بایت‌های بالاتر (اگر محاسبه با بیش از ١٦ بیت انجام می‌شود) به کار رود.

چرخش به راست نیز امکان‌پذیر بوده، طی آن عمل تقسیم بر ۲ انجام شده و بیت Carry به داخل بیت هفتم مقدار حاصل منتقل می‌شود:

```
        LSR R2 ; Logical Shift Right, bit 0 to carry
        ROR R1 ; ROtate Right and shift carry in bit 7
```

تقسیم اعداد بزرگ‌تر نیز به همین سادگی است. چنان که می بینید فراگیری اسمبلی چندان هم سخت و پیچیده نیست.

آخرین دستورالعمل، که قادر به انتقال چهار بیت به طور همزمان است، اغلب هنگام کار با اعداد BCD فشرده به کار می‌رود. این دستور نیبل‌های بالایی و پایینی عدد BCD را جابه‌جا می‌کند. در این مثال می‌خواهیم نیبل بالایی را به موقعیت نیبل پایینی عدد منتقل کنیم. به جای استفاده از دستورات

```
        ROR R1
        ROR R1
        ROR R1
        ROR R1
```

می‌توانیم این کار را تنها با تک دستورالعمل زیر انجام دهیم:

```
        SWAP R1
```

این دستورالعمل، نیبل‌های بالایی و پایینی را با هم تعویض می‌کند. توجه کنید که محتویات نیبل بالایی پس از اِعمال هر یک از این دو روش، متفاوت خواهد بود.

1- Arithmetic Shift Right

جمع، تفریق و مقایسه

اَعمال محاسباتی زیر برای مبتدی‌ها بسیار پیچیده بوده و نشان می‌دهند که اسمبلی فقط مخصوص متخصصین بسیار پیشرفته است! این بخش را با مسئولیت خودتان بخوانید!

برای شروع، دو عدد ١٦ بیتی موجود در R1:R2 و R3:R4 را با هم جمع می‌کنیم (منظور از این طرز نمایش این است که اولین ثبات، بایت با بیشترین ارزش، و دومین ثبات بایت با کمترین ارزش است).

```
ADD R2,R4 ; first add the two low-bytes
ADC R1,R3 ; then the two high-bytes
```

در دومین دستورالعمل، به جای ADD از ADC استفاده کرده‌ایم. این دستور به معنای جمع با رقم نقلی (Carry) است که مقدار آن با اجرای اولین دستور، بسته به مقدار نتیجه، صفر یا یک شده است. آیا به اندازه کافی از این عملیات پیچیده ریاضی ترسیده‌اید؟ اگر نه، بعدی را ببینید!

R3:R4 را از R1:R2 کم می‌کنیم.

```
SUB R2,R4 ; first the low-byte
SBC R1,R3 ; then the high-byte
```

باز هم همان شگرد قبلی: اگر اولین دستورالعمل ایجاد سرریز کرده باشد، هنگام اجرای دستورالعمل دوم، از مقدار نتیجه یک واحد کم می‌کنیم. هنوز نفس می‌کشید؟ اگر این طور است قسمت بعدی را ببینید!

اکنون کلمهٔ ١٦ بیتی موجود در R1:R2 را با کلمهٔ موجود در R3:R4 مقایسه می‌کنیم تا ببینیم کدامیک از دیگری بزرگ‌تر است. به جای استفاده از SUB، دستورالعمل مقایسه‌ای CP و به جای استفاده از SBC، دستورالعمل CPC را به کار می‌بریم:

```
CP R2,R4 ; compare lower bytes
CPC R1,R3 ; compare upper bytes
```

حالا اگر رقم نقلی یک شده باشد، R1:R2 بزرگ‌تر از R3:R4 است.

اکنون مسائل پیچیده‌تری را مطرح می‌کنیم. محتویات ثبات R16 را با مقدار ثابت 0b10101010 مقایسه می‌کنیم:

```
CPI R16,0xAA
```

اگر پس از اجرای این دستور، بیت Zero در ثبات وضعیت یک شده باشد می فهمیم که R16 برابر 0xAA است. اگر بیت رقم نقلی یک شده باشد، می فهمیم که از 0xAA کوچک‌تر است. اگر رقم نقلی یک نشده و بیت Zero هم یک نشده باشد، می فهمیم که از 0xAA بزرگ‌تر است.

و اکنون پیچیده‌ترین مسئله: صفر یا منفی بودن R1 را تعیین می‌کنیم:

```
TST R1
```

اگر بیت Z یک شده باشد، مقدار ثبات R1 برابر صفر است و در ادامه می‌توانیم با استفاده از دستورات BREQ، BRNE، BRMI، BRPL، BRLO، BRSH، BRGE، BRLT، BRVC یا BRVS، براساس مقدار یک بیت، پرش کنیم.

هنوز با ما همراه هستید؟ اگر این طور است، اکنون محاسباتی را با اعداد BCD فشرده انجام می‌دهیم. جمع دو عدد BCD فشرده می‌تواند باعث ایجاد دو سرریز متفاوت شود. رقم نقلی عادی (Carry) نشان‌دهنده وقوع سرریز در نیبل بالایی به مقداری بیشتر از عدد دهدهی ١٥ است. سرریز دیگر، از نیبل پایینی به نیبل بالایی، در صورتی رخ می‌دهد که حاصل جمع دو نیبل پایینیِ دو عدد، بیشتر از عدد دهدهی ١٥ گردد.

به عنوان یک مثال، دو عدد BCD فشردهٔ ٤٩ (0x49 در مبنای شانزده) و ٩٩ (0x99 در مبنای شانزده) را با هم جمع می‌کنیم تا عدد ١٤٨ (0x0148 در مبنای شانزده) حاصل شود. جمع این دو عدد در سیستم دودویی، مقدار مبنای شانزده 0xE2 را بدون وقوع سرریز در بایت، نتیجه می‌دهد. جمع دو نیبل پایینی باید ایجاد سرریز کرده باشد، زیرا ١٨ = ٩ + ٩

(بیشتر از ۹)، در حالی که نیبل پایین تنها قادر به نگهداری اعداد با مقدار حداکثر ۱۵ است. این سرریز به بیت چهارم، بیت با کمترین ارزش نیبل بالایی، اضافه شده است. این عمل، کاملاً صحیح است! اما نیبل پایین باید ۸ باشد در حالی که ۲ است (0b0001.0010 = 18). می‌بایست عدد ٦ را به این نیبل اضافه کنیم تا نتیجهٔ صحیح به دست آید. این کار، کاملاً منطقی است زیرا هر گاه مقدار نیبل پایین از ۹ بیشتر شود، برای تصحیح آن باید عدد ٦ را به آن بیافزاییم.

مقدار نیبل بالایی کاملاً غلط است زیرا برابر با 0xE است، در حالی که باید 4 (همراه با سرریزی به اندازهٔ ۱ به رقم بعدی BCD فشرده) باشد. اگر ٦ را با 0xE جمع کنیم مقدار 0x4 به دست آمده و رقم نقلی نیز یک می‌شود (= 0x14). بنابراین روش کار به این ترتیب است که ابتدا این دو عدد را با هم جمع کرده و سپس 0x66 را برای تصحیح دو رقم BCD فشرده، به آن اضافه می‌کنیم. اما دست نگه دارید: اگر حاصل جمع اولین و دومین عدد باعث ایجاد سرریز به نیبل بعدی نشود چه اتفاقی می‌افتد؟ و اگر رقم حاصل در نیبل پایینی بزرگ‌تر از ۹ نشود چه اتفاقی می‌افتد؟ در این حالت افزودن عدد 0x66، به نتیجهٔ کاملاً نادرستی منتهی می‌شود. عدد ٦ تنها در صورتی باید به نیبل پایینی اضافه شود که یا نیبل پایینی به نیبل بالایی سرریز شود، و یا رقم حاصل بزرگ‌تر از ۹ گردد. در مورد نیبل بالایی هم قاعده به همین شکل است.

چگونه از وقوع سرریز از نیبل پایین به نیبل بالا مطلع شویم؟ با وقوع این سرریز، MCU بیت H در ثبات وضعیت را که بیت Half-Carry خوانده می‌شود، یک می‌کند. مراحل زیر، الگوریتم لازم برای حالات مختلفی که ممکن است بعد از جمع دو نیبل و به دنبال آن افزودن عدد مبنای شانزده 0x6 رخ دهد را بیان می‌کند.

۱. نیبل‌ها را با هم جمع کنید. در صورت وقوع سرریز (C برای نیبل‌های بالا، یا H برای نیبل‌های پایین)، عدد ٦ را برای تصحیح نتیجه اضافه کنید. در غیراینصورت مرحلهٔ ۲ را انجام دهید.
۲. عدد ٦ را با نیبل جمع کنید. اگر سرریز رخ دهد (به ترتیب C و H)، کار تمام است. در غیراینصورت عدد ٦ را از آن کم کنید.

به عنوان یک مثالِ برنامه‌نویسی، فرض می‌کنیم که دو عدد BCD فشرده در ثبات‌های R2 و R3 قرار داشته و سرریز حاصل از محاسبه در R1 قرار خواهد گرفت. همچنین فرض می‌کنیم که R16 و R17 برای انجام محاسبات در دسترس هستند. از R16 برای اضافه کردن 0x66 به R2 (نمی‌توان یک مقدار ثابت را به ثبات R2 اضافه کرد) و از R17 برای تصحیح نتیجه براساس پرچم‌های مختلف استفاده شده است. جمع R2 و R3 به شکل زیر است:

```
        LDI R16,0x66 ; for adding 0x66 to the result
        LDI R17,0x66 ; for later subtracting from the result
        ADD R2,R3 ; add the two two-digit-BCDs
        BRCC NoCy1 ; jump if no byte overflow occurs
        INC R1 ; increment the next higher byte
        ANDI R17,0x0F ; don't subtract 6 from the higher nibble
NoCy1:
        BRHC NoHc1 ; jump if no half-carry occurred
        ANDI R17,0xF0 ; don't subtract 6 from lower nibble
NoHc1:
        ADD R2,R16 ; add 0x66 to result
        BRCC NoCy2 ; jump if no carry occurred
        INC R1 ; increment the next higher byte
        ANDI R17,0x0F ; don't subtract 6 from higher nibble
NoCy2:
        BRHC NoHc2 ; jump if no half-carry occurred
        ANDI R17,0xF0 ; don't subtract 6 from lower nibble
NoHc2:
        SUB R2,R17 ; subtract correction
```

روش زیر کمی مختصرتر از روش قبلی است:

```
        LDI R16,0x66
        ADD R2,R16
        ADD R2,R3
        BRCC NoCy
        INC R1
        ANDI R16,0x0F
NoCy:
        BRHC NoHc
        ANDI R16,0xF0
```

```
NoHc:
    SUB R2,R16
```

به این سؤال فکرکنید: چرا این روش درست کار می‌کند، در حالی که نصف طول و پیچیدگی روش قبلی است و چه نکته ای در آن به کار رفته است؟

تبدیل قالب اعداد

تمام قالب‌های اعداد، قابل تبدیل به هر قالب دیگری هستند. روش تبدیل از قالب BCD به ASCII و بالعکس در بخش‌های پیش بیان شده است (بخش دستکاری بیتی).

تبدیل اعداد BCD فشرده هم خیلی پیچیده نیست. ابتدا باید عدد مورد نظر را به ثبات دیگری کپی کنیم. با استفاده از دستور SWAP، محل نیبل‌های پایین و بالای عدد را در مقدار کپی شده تعویض می‌کنیم. نیم‌بایت بالا را، مثلاً از طریق AND کردن با 0x0F، پاک می‌کنیم. اکنون فرم BCDیِ نیبل بالا را در اختیار داریم و می‌توانیم از آن به عنوان یک عدد BCD استفاده کرده و یا بیت‌های ٤ و ٥ را برای تبدیل آن به کاراکترهای ASCII یک کنیم. پس از آن، دوباره بایت اولیه را کپی کرده و همین کار را با نیبل پایین (بدون SWAP کردن) انجام می‌دهیم تا BCD پایینی به دست آید.

تبدیل ارقام BCD به دودویی کمی پیچیده‌تر است. نخست، بسته به اندازهٔ اعدادی که باید تبدیل شوند بایت‌های لازم برای نگهداری نتیجهٔ عملیات تبدیل را پاک می‌کنیم. سپس کار را با بالاترین رقم BCD آغاز می‌کنیم. قبل از افزودن آن به نتیجه، باید نتیجه را در ۱۰ ضرب کنیم (دقت کنید که در اولین مرحله نیازی به این کار نیست، زیرا نتیجه صفر است). برای ضرب کردن در ۱۰، نتیجه را در جای دیگری کپی کرده، سپس آن را در چهار ضرب می‌کنیم (دو بار شیفت به چپ و چرخش). افزودن عدد کپی شدهٔ قبلی به این عدد، ضرب در ۵ را نتیجه می‌دهد. اکنون با ضرب آن در ۲ (شیفت و چرخش به چپ)، ۱۰ برابر نتیجه به دست می‌آید. در خاتمه، عدد BCD را با نتیجه جمع کرده و این الگوریتم را تا تبدیل تمام ارقام دهدهی تکرار می‌کنیم. اگر حین اجرای هر یک از این اَعمال، رقم نقلی در مقدار نتیجه ایجاد شود، عدد BCD برای تبدیل، بیش از اندازه بزرگ است. این الگوریتم، قابلِ اعمال به اعداد با هر طولی است، البته به شرطی که ثبات‌های لازم برای نگهداری نتیجه موجود باشد.

تبدیل یک عدد دودویی به BCD از آن هم پیچیده‌تر است. اگر بخواهیم یک عدد دودویی ۱٦ بیتی را تبدیل کنیم، می‌توانیم عدد ۱۰،۰۰۰ (0x2710) را از آن کم کنیم تا زمانی که سرریز رخ دهد. با این کار اولین رقم به دست می‌آید. سپس این کار را با عدد ۱،۰۰۰ (0x03E8) تکرار می‌کنیم تا رقم دوم حاصل شود. این کار را با اعداد ۱۰۰ (0x0064) و ۱۰ (0x000A) ادامه می‌دهیم. در نهایت، مقدار باقیمانده، آخرین رقم خواهد بود. مقادیر ثابتِ ۱۰،۰۰۰، ۱،۰۰۰، ۱۰۰ و ۱۰ را می‌توان در حافظهٔ ذخیرهٔ برنامه در قالب یک جدول به صورت کلمه به کلمه، به شکل زیر، قرار داد:

```
[1]DezTab:
.DW 10000, 1000, 100, 10
```

این مقادیر را می‌توان با استفاده از دستور LPM، به صورت کلمه به کلمه، از جدول خواند.

روش دیگر، استفاده از جدولی مانند جدول زیر است که ارزش مکانی هر بیتِ عدد دودوییِ ۱٦ بیتی به صورت دهدهی در آن قرار دارد:

```
.DB 0,3,2,7,6,8
.DB 0,1,6,3,8,4
.DB 0,0,8,1,9,2
.DB 0,0,4,0,9,6
.DB 0,0,2,0,4,8 ; and so on until
.DB 0,0,0,0,0,1
```

در این روش تک تک بیت‌های عدد دودویی را به سمت چپ شیفت دهید تا از ثبات خارج شده و به داخل رقم نقلی منتقل شود. اگر این بیت یک بود، عدد موجود در جدول را (که به ارزش مکانی آن بیت مربوط می‌شود) به نتیجه بیافزایید. باز هم از دستور LPM برای خواندن اعداد از جدول استفاده کنید. نوشتن برنامه برای این روش پیچیده‌تر بوده

۱- مترجم: Dez مخفف Dezimal است که Dezimal، معادل آلمانی Decimal می‌باشد.

و سرعت اجرای آن کمتر از روش قبلی است.

روش سوم، محاسبهٔ مقدار جدول، با شروع از 000001، از طریق جمع این عدد BCD با خودش، پس از هر بار شیفت یک بیت از عدد ورودی به راست و افزودن عدد BCD است.

روش‌های بسیاری برای این کار وجود دارند؛ روشی را انتخاب کنید که تطابق بیشتری با نیازهای شما دارد.

ضرب

در این بخش ضرب اعداد دودویی مورد بحث قرار می‌گیرد.

ضرب در مبنای ده

برای ضرب دو عدد ۸ بیتی، ابتدا نحوه انجام این کار با اعداد دهدهی را یادآوری می‌کنیم:

```
    1234 * 567 = ?
------------------------
    1234 *   7 =   8638
+   1234 *  60 =  74040
+   1234 * 500 = 617000
------------------------
    1234 * 567 = 699678
========================
```

مراحل انجام ضرب دهدهی:

- اولین عدد را در کم ارزش‌ترین رقم دومین عدد ضرب کرده و حاصل را به جواب اضافه می‌کنیم.
- اولین عدد را در ۱۰ و سپس در رقم بعدیِ دومین عدد ضرب کرده و حاصل را به جواب اضافه می‌کنیم.
- اولین عدد را در ۱۰۰، و سپس در سومین رقم ضرب کرده، و حاصل را به جواب اضافه می‌کنیم.

ضرب دودویی

اکنون به ضرب در حالت دودویی می‌پردازیم. در این حالت نیازی به ضرب در ارقام عدد نیست، زیرا فقط ارقام 1 (اضافه کردن عدد) و 0 (اضافه نکردن عدد) وجود دارند. ضرب در ۱۰ در حالت دهدهی، به ضرب در ۲ در حالت دودویی تبدیل می‌شود. ضرب کردن در ۲ یا از طریق جمع عدد با خودش، و یا از طریق انتقال تمام بیت‌های آن به اندازهٔ یک واحد به چپ و افزودن صفر از طرف راست، به آسانی قابل انجام است. همانطور که ملاحظه می‌کنید ریاضیات دودویی بسیار ساده‌تر از حالت دهدهی است. پس چرا انسان از همان آغاز، این فرم اعداد را به کار نبرده است؟

برنامهٔ نمونهٔ اسمبلر AVR

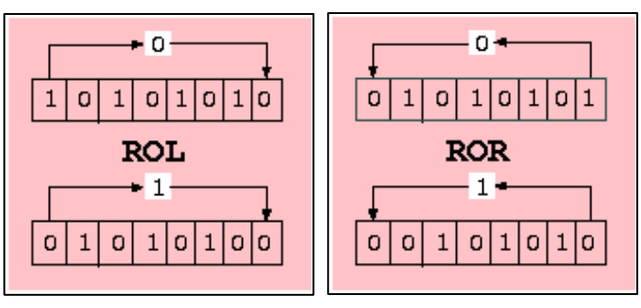
کد منبع زیر نحوه انجام ضرب در زبان اسمبلی را نشان می‌دهد.

```
; Mult8.asm multiplies two 8-bit-numbers to yield a 16-bit-result
;
.NOLIST
.INCLUDE "C:\avrtools\appnotes\8515def.inc"
.LIST
;
; Flow of multiplication
;
; 1.The binary to be multiplicated with, is shifted bitwise into the carry bit. If it is a one, the binary number is added to the
;    result, if it is not a one that was shifted out, the number is not added.
; 2.The binary number is multiplied by 2 by rotating it one position left, shifting a 0 into the void position.
; 3.If the binary to be multiplied with, is not zero, the multiplication loop is repeated. If it is zero, the multiplication is done.
;
; Used registers
;
.DEF rm1 = R0 ; Binary number to be multiplicated (8 Bit)
.DEF rmh = R1 ; Interim storage
.DEF rm2 = R2 ; Binary number to be multiplicated with (8 Bit)
```

```
.DEF rel = R3 ; Result, LSB (16 Bit)
.DEF reh = R4 ; Result, MSB
.DEF rmp = R16 ; Multi purpose register for loading
;
.CSEG
.ORG 0000
;
        rjmp START
;
START:
        ldi rmp,0xAA ; example binary 1010.1010
        mov rm1,rmp ; to the first binary register
        ldi rmp,0x55 ; example binary 0101.0101
        mov rm2,rmp ; to the second binary register
;
; Here we start with the multiplication of the two binaries in rm1 and rm2, the result will go to reh:rel (16 Bit)
;
MULT8:
;
; Clear start values
        clr rmh ; clear interim storage
        clr rel ; clear result registers
        clr reh
;
; Here we start with the multiplication loop
;
MULT8a:
;
; Step 1: Rotate lowest bit of binary number 2 to the carry flag (divide by 2, rotate a zero into bit 7)
;
        clc ; clear carry bit
        ror rm2 ; bit 0 to carry, bit 1 to 7 one position to the right, carry bit to bit 7
;
; Step 2: Branch depending if a 0 or 1 has been rotated to the carry bit
;
        brcc MULT8b ; jump over adding, if carry has a 0
;
; Step 3: Add 16 bits in rmh:rml to the result, with overflow from LSB to MSB
;
        add rel,rm1 ; add LSB of rm1 to the result
        adc reh,rmh ; add carry and MSB of rm1
;
MULT8b:
;
; Step 4: Multiply rmh:rm1 by 2 (16 bits, shift left)
;
        clc ; clear carry bit
        rol rm1 ; rotate LSB left (multiply by 2)
        rol rmh ; rotate carry into MSB and MSB one left
;
; Step 5: Check if there are still one's in binary 2, if yes, go on multiplicating
;
        tst rm2 ; all bits zero?
        brne MULT8a ; if not, go on in the loop
;
; End of the multiplication, result in reh:rel
;
; Endless loop
;
LOOP:
        rjmp loop
```

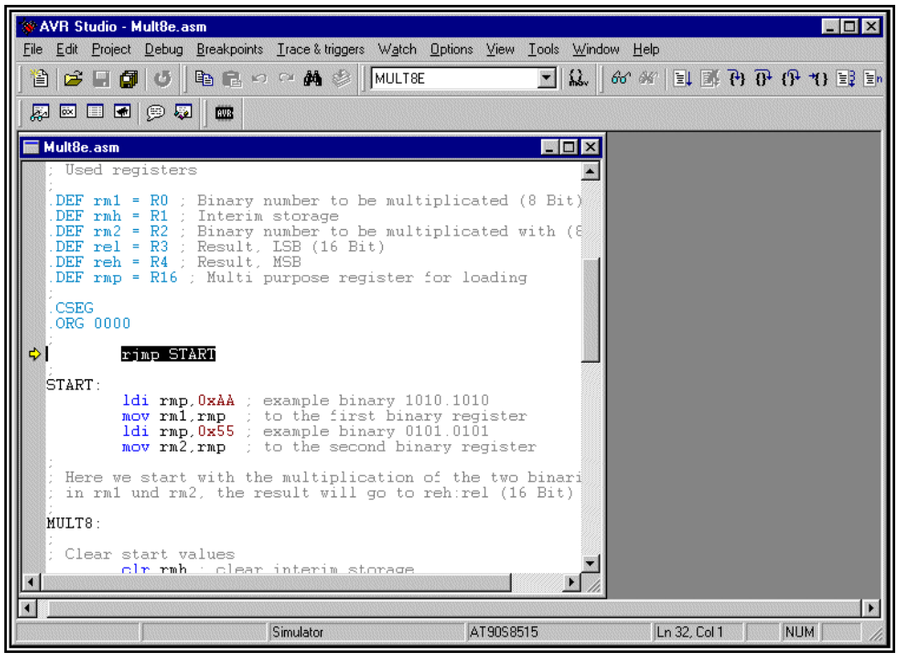
چرخش دودویی

برای درک نحوه عملکرد عمل ضرب، داشتن درکی از دستورات چرخش دودویی، یعنی ROL و ROR، ضروری است. این دستورات تمام بیت‌های یک ثبات را به اندازهٔ یک واحد به چپ (ROL) یا به راست (ROR) انتقال می‌دهند. محل خالی شده در ثبات با مقدار جاری بیت Carry موجود در ثبات وضعیت پر شده، و بیتِ خارج شده از ثبات به داخل بیت Carry منتقل می‌شود. این عملیات با استفاده از عدد 0xAA به عنوان مثالی برای ROL، و عدد 0x55 به عنوان مثالی برای ROR، نشان داده شده است.

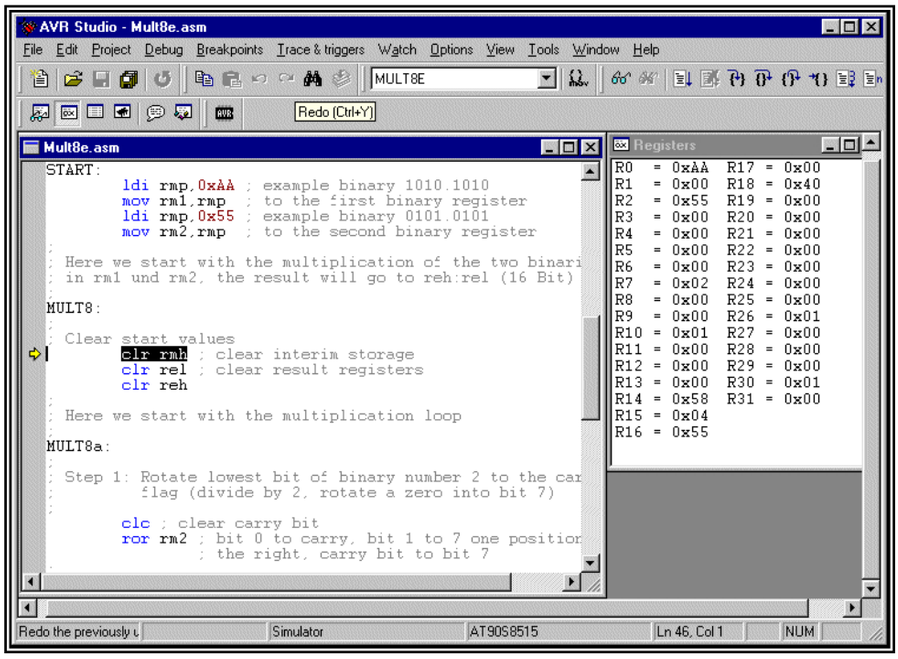


برنامه ضرب در محیط AVR Studio

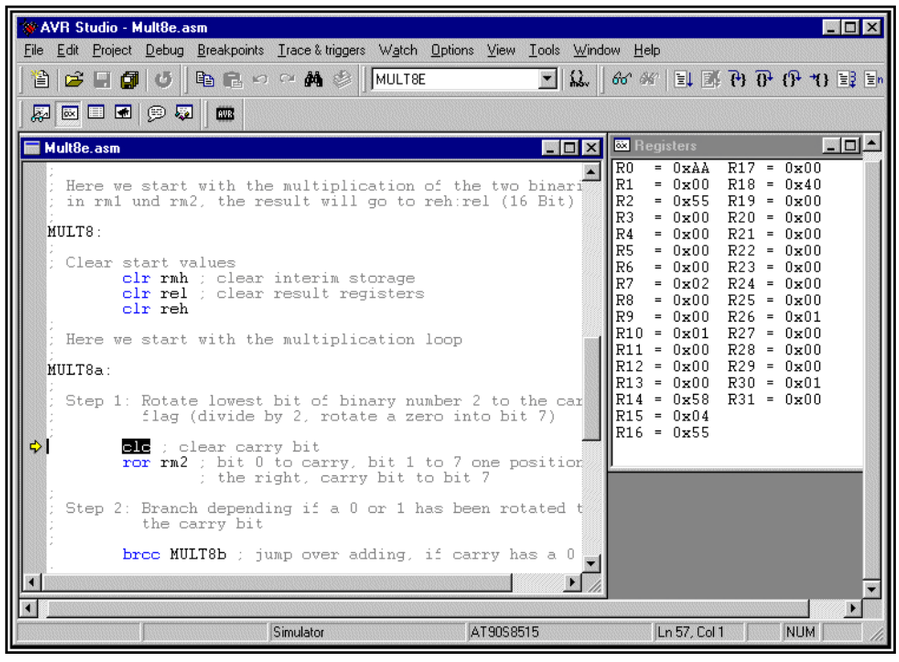
تصاویر زیر برنامه ضرب را در محیط شبیه‌ساز نشان می‌دهند.



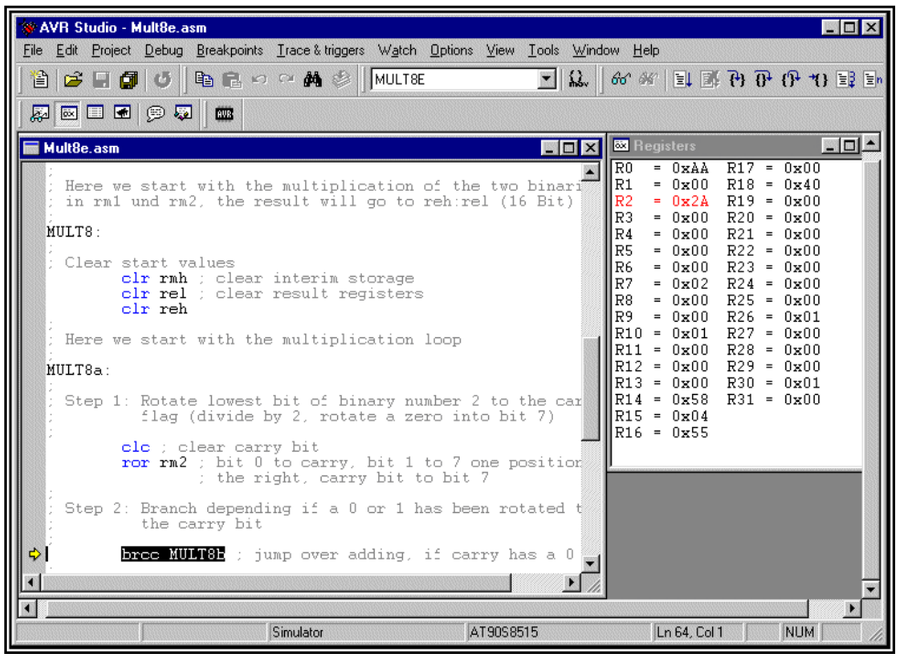
کد مقصد باز شده است و مکان‌نما بر روی اولین دستور قابل اجرا قرار دارد. کلید F11 دستورات را تک به تک اجرا می‌کند.



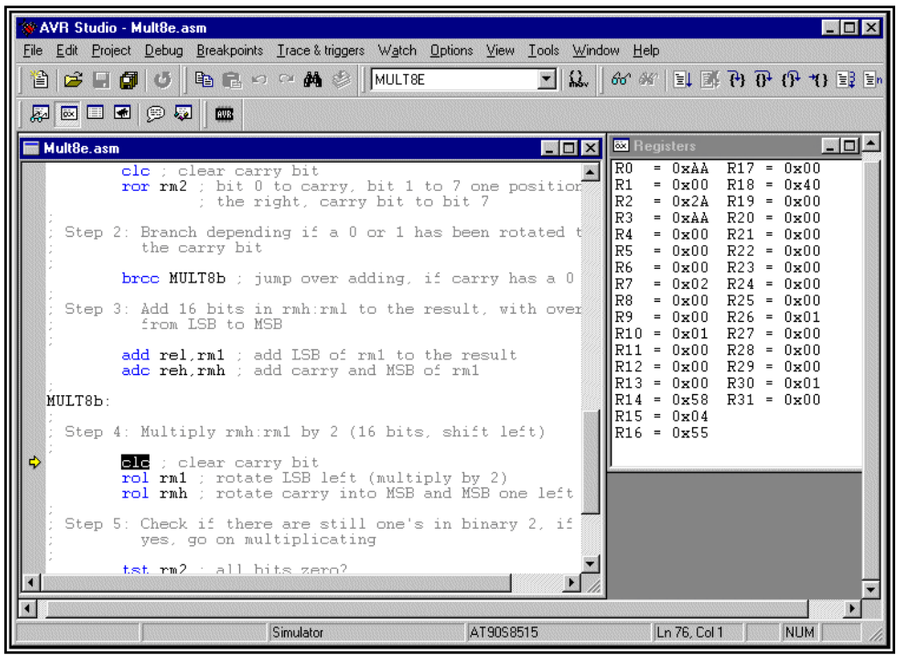
اعداد دودویی مورد آزمایش ما، یعنی 0xAA و 0x55، به ترتیب در ثبات‌های R0 و R2 قرار گرفته‌اند تا در هم ضرب شوند.



R2 به راست چرخش داده شده تا کم ارزش‌ترین بیت به داخل بیت Carry منتقل شود. نتیجهٔ حاصل از چرخش 0x55 (0101.0101) به راست، 0x2A (0010.1010) است.

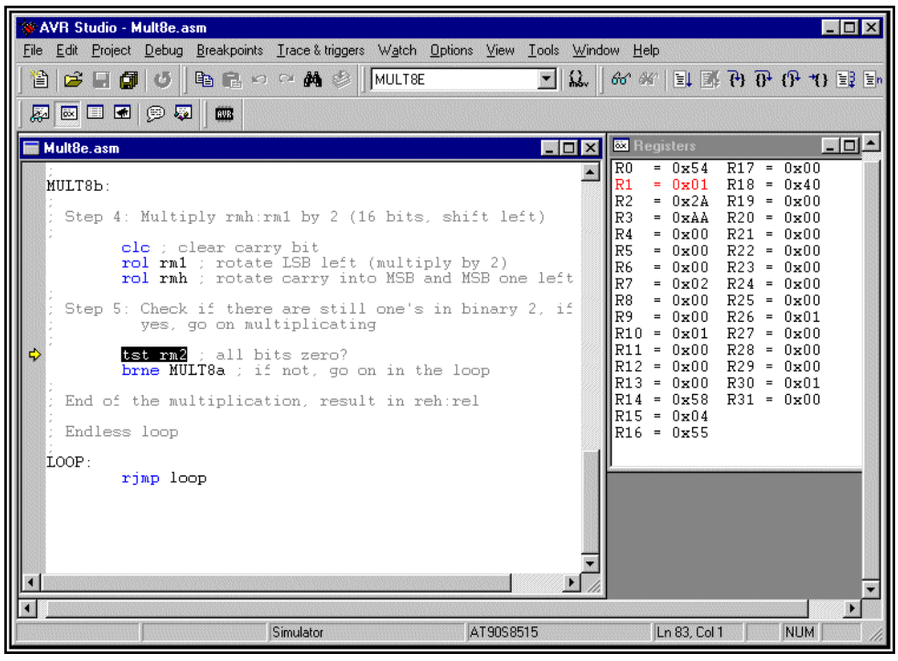


چون بیت Carry یک شده است، محتویات ثبات های R1:R0 به زوج ثبات (خالی) R4:R3 افزوده شده، و مقدار 0x00AA به عنوان نتیجه در آنها قرار می‌گیرد.



اکنون زوج ثبات R1:R0 به اندازه یک واحد به چپ چرخش داده می‌شود تا این عدد دودویی در ۲ ضرب شود. حاصل ضرب 0x00AA در ۲ برابر با 0x0154 خواهد بود.

حلقهٔ ضرب تا زمانی تکرار می‌شود که حداقل یک رقم دودویی 1 در ثبات R2 موجود باشد. حلقه های بعدی در اینجا نشان داده نشده‌است.



با استفاده از کلید F5 در محیط AVR Studio، به صورت یک جا از این حلقه‌ها می گذریم تا به یک نقطهٔ شکست در انتهای روال ضرب برسیم. حاصل ضرب 0xAA در0x55، یعنی مقدار0x3872، در زوج ثبات نتیجهٔ R4:R3 قرار خواهد گرفت.

همانطور که دیدید این کار آنقدرها هم پیچیده نبود. تنها کافی است عملیات مشابه در حالت دهدهی را به خاطر بیاورید. ضرب دودویی بسیار ساده‌تر از ضرب دهدهی است.

تقسیم

تقسیم در مبنای ده

باز هم ما با تقسیم در مبنای ده آغاز می‌کنیم تا تقسیم در حالت دودویی را بهتر درک کنیم. تقسیم عدد ٥٦٧٨ بر ١٢ را در نظر می‌گیریم. این عمل به شکل زیر انجام می‌شود:

```
              5678 : 12 = ?
-------------------------
- 4 * 1200 = 4800
             ----
              878
- 7 *  120 =  840
              ---
               38
- 3 *   12 =   36
               --
                2
Result: 5678 : 12 = 473 Remainder 2
==================================
```

تقسیم دودویی

با توجه به این حقیقت که در حالت دودویی تنها ارقام 0 و 1 را در اختیار داریم، نیازی به ضرب در عدد دوم(1200 * 4، و غیره) نیست. متأسفانه اعداد دودویی نسبت به معادل دهدهی‌شان تعداد ارقام بسیار بیشتری دارند و بنابراین استفاده از روش تقسیم دهدهی در حالت دودویی کمی خسته کننده خواهد بود. بنابراین روش به کار گرفته شده در این برنامه کمی با آن روش تفاوت دارد.

تقسیم یک عدد دودویی ١٦ بیتی بر یک عدد دودویی ٨ بیتی در زبان اسمبلی AVR، در زیر آورده شده است.

```
; Div8 divides a 16-bit-number by a 8-bit-number (Test: 16-bit-number: 0xAAAA, 8-bit-number: 0x55)
.NOLIST
.INCLUDE "C:\avrtools\appnotes\8515def.inc"
.LIST
; Registers
.DEF rd1l = R0 ; LSB 16-bit-number to be divided
.DEF rd1h = R1 ; MSB 16-bit-number to be divided
.DEF rd1u = R2 ; interim register
.DEF rd2 = R3 ; 8-bit-number to divide with
.DEF rel = R4 ; LSB result
.DEF reh = R5 ; MSB result
.DEF rmp = R16; multipurpose register for loading
;
.CSEG
.ORG 0
        rjmp start
start:
; Load the test numbers to the appropriate registers
        ldi rmp,0xAA ; 0xAAAA to be divided
        mov rd1h,rmp
        mov rd1l,rmp
        ldi rmp,0x55 ; 0x55 to be divided with
        mov rd2,rmp
; Divide rd1h:rd1l by rd2
div8:
        clr rd1u ; clear interim register
        clr reh ; clear result (the result registers
        clr rel ; are also used to count to 16 for the
        inc rel ; division steps, is set to 1 at start)
; Here the division loop starts
div8a:
        clc ; clear carry-bit
        rol rd1l ; rotate the next-upper bit of the number
        rol rd1h ; to the interim register (multiply by 2)
        rol rd1u
        brcs div8b ; a one has rolled left, so subtract
        cp rd1u,rd2 ; Division result 1 or 0?
```

```
        brcs div8c ; jump over subtraction, if smaller
div8b:
        sub rd1u,rd2; subtract number to divide with
        sec ; set carry-bit, result is a 1
        rjmp div8d ; jump to shift of the result bit
div8c:
        clc ; clear carry-bit, resulting bit is a 0
div8d:
        rol rel ; rotate carry-bit into result registers
        rol reh
        brcc div8a ; as long as zero rotate out of the result registers: go on with the division loop
; End of the division reached
stop:
        rjmp stop ; endless loop
```

مراحل برنامه در طول عملیات تقسیم

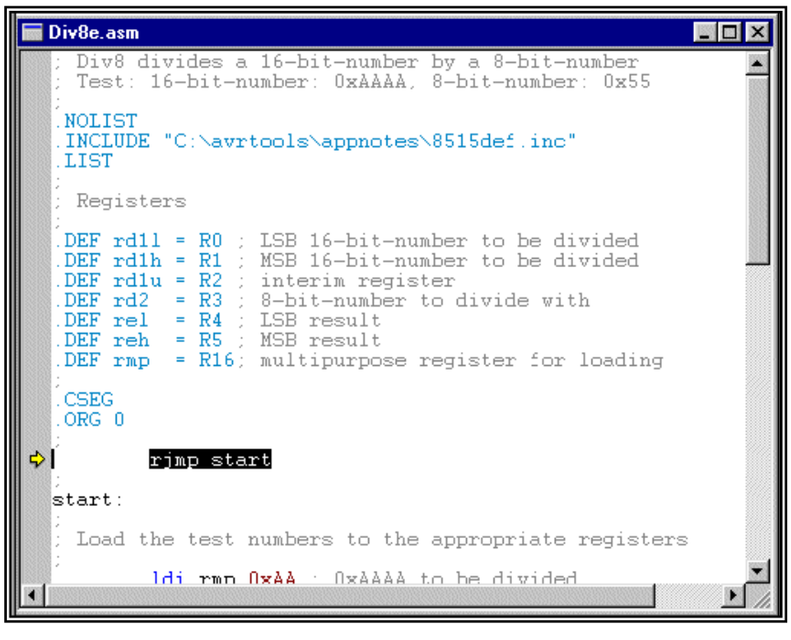
در طول اجرای برنامه مراحل زیر انجام می‌شود:

- تعریف و تنظیم مقادیر اولیهٔ ثبات‌ها با مقادیر دودویی آزمایشی،
- تنظیم مقدار اولیهٔ ثبات میانی(موقتی) و زوج ثبات نتیجه(ثبات‌های نتیجه برابر 0x0001 مقداردهی می‌شوند! پس از ١٦ بار چرخش، خارج شدن عددِ یک، از ادامه مراحل تقسیم جلوگیری می‌کند.)،
- عدد دودویی ١٦ بیتیِ موجود در rd1h:rd1l، بیت به بیت به داخل ثبات میانی rd1u چرخش داده می‌شود (ضرب کردن در ٢)، اگر با عمل چرخش، رقم 1 از rd1u خارج شود، اجرای برنامه بلافاصله به مرحلهٔ تفریق در مرحله ٤ منشعب می‌شود،
- محتویات ثبات میانی با عدد دودویی ٨ بیتیِ موجود در rd2 مقایسه می‌شود. اگر rd2 کوچک‌تر باشد، rd2 از محتویات ثبات میانی کم شده و بیت Carry یک می‌شود، اگر rd2 بزرگ‌تر باشد از عمل تفریق صرف‌نظر شده و پرچم Carry صفر می‌شود،
- محتویات پرچم Carry از طرف راست به داخل زوج ثبات نتیجه (reh:rel) چرخش داده می‌شود،
- اگر با چرخش ثبات نتیجه، صفر از آن خارج شود، باید حلقهٔ تقسیم را تکرار کنیم، اما اگر یک خارج شود، عملیات تقسیم به اتمام می‌رسد.

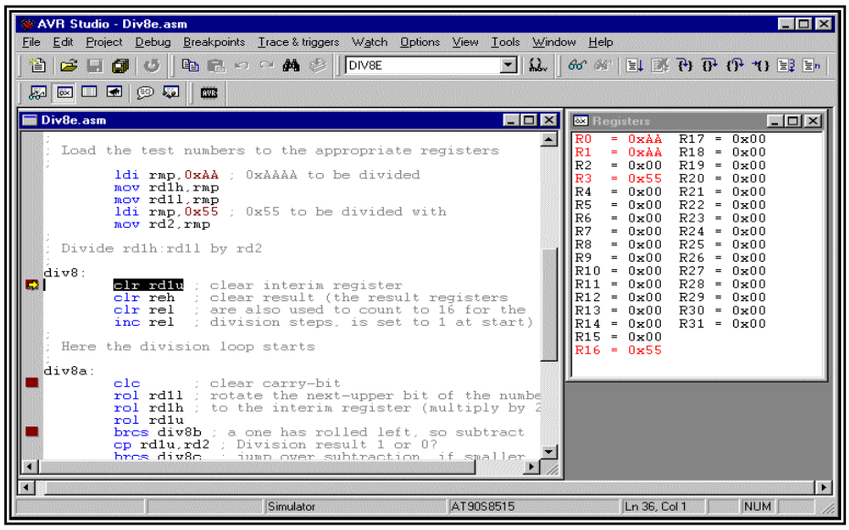
اگر هنوز نحوه عملکرد عمل چرخش را نمی‌فهمید، این عملیات در بخش ضرب مورد بحث قرار گرفته است.

برنامه تقسیم در شبیه‌ساز

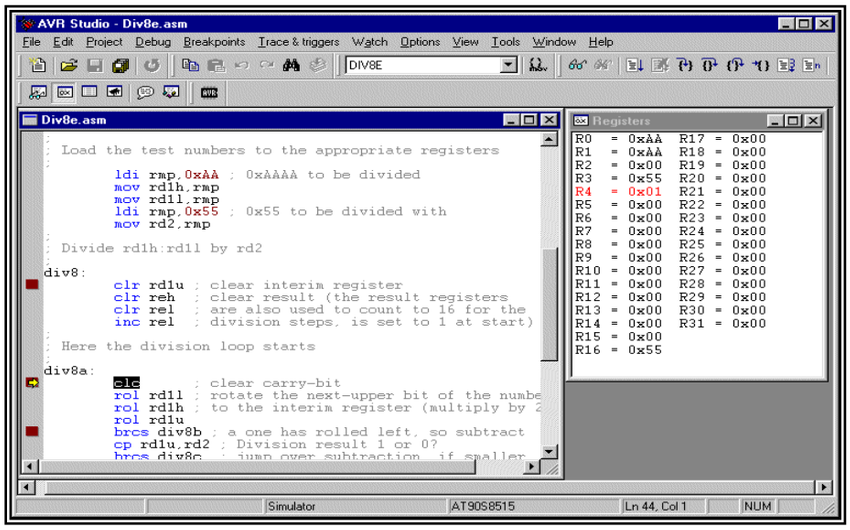
تصاویر زیر مراحل اجرای برنامه را در محیط AVR Studio نشان می‌دهند. برای انجام این کار، باید کد منبع برنامه را اسمبل کرده و فایل مقصد خروجی را در برنامه Studio باز کنید.



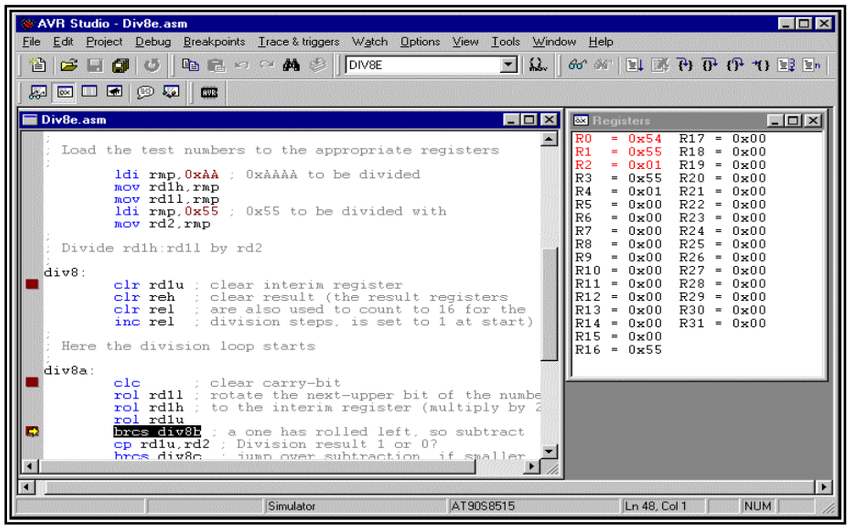
اجرای کد مقصد آغاز شده است و مکان‌نما بر روی اولین دستور قابل اجرا قرار دارد. کلید F11 باعث اجرای مرحله به مرحلهٔ برنامه می‌شود.



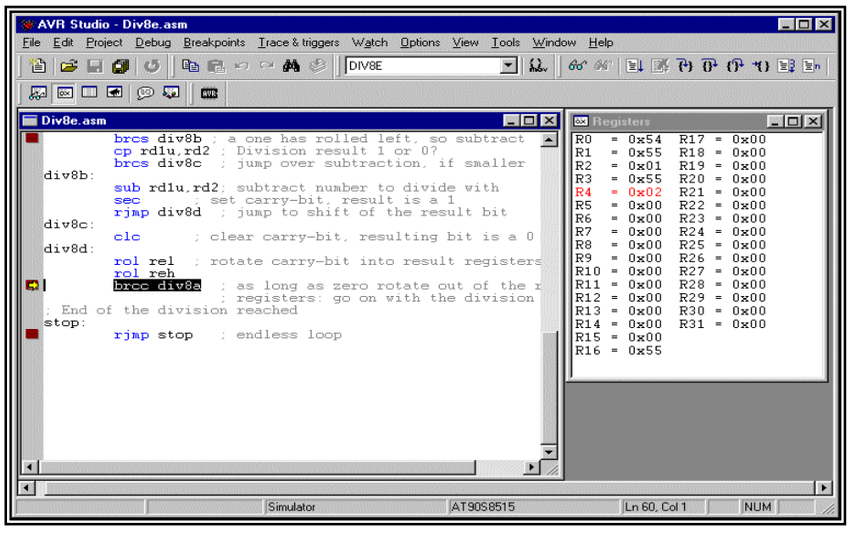
اعـــداد دودویـــی آزمایـــشی، 0xAAAA و 0x55، برای انجـام عمل تقسیم در ثبات‌هـای R1:R0 و R3 نوشته شده‌اند.



ثبات میـانی (مـوقتی) R2 و زوج ثبات نتیجـه، بـه مقـادیر از پـیش تعریف‌شده‌شان تنظیم شده‌اند.

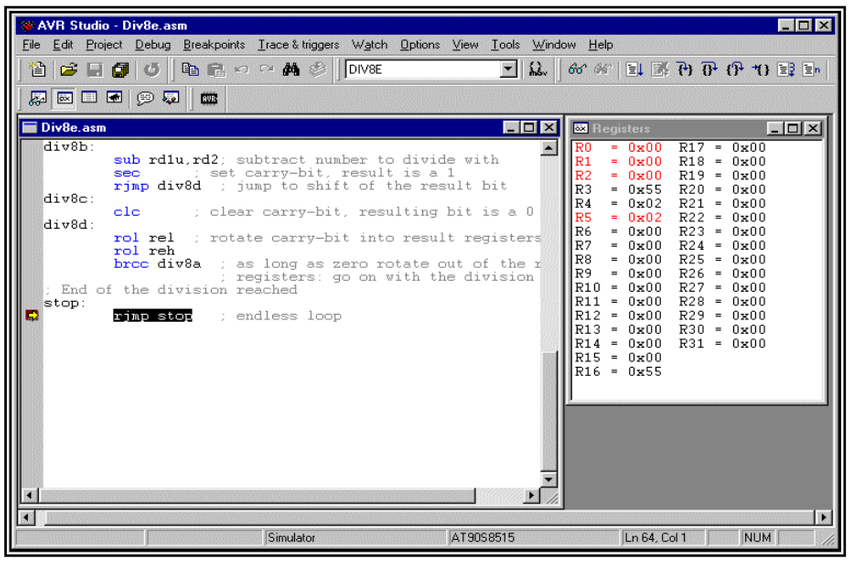


R1:R0 از سمت چپ بـه داخـل R2 چرخش داده شـده و دو برابـرِ عدد 0xAAAA یعنی 0x015554 حاصل شده است.

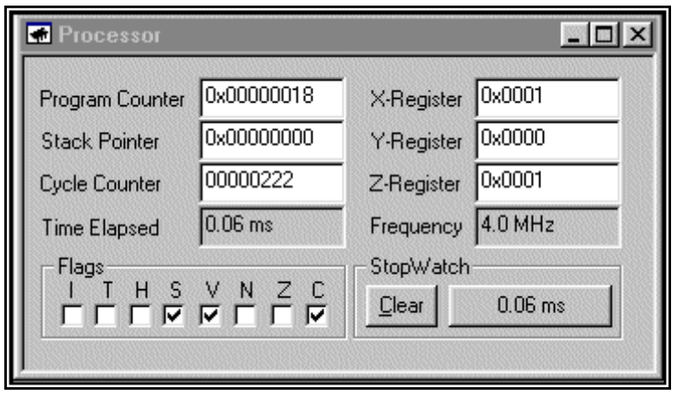


با انجام عمل چرخش به داخل بیت Carry، هیچ سرریزی رخ نـداده و مقـدار 0x01 موجـود در R2 نیـز کوچک‌تر از مقدار 0x55 موجـود در R3 بـوده اسـت، و بنـابراین از تفریق صرف‌نظر شده اسـت. صـفر موجود در Carry به داخـل ثبـات نتیجـهٔ R5:R4 چرخانـده شـده و محتوای قبلی ثبات نتیجه، که یـک

تک بیتِ موجود در بیت صفر بوده است، به محل بیت ۱ منتقل شده است (محتوای کنونی: 0x0002). از آنجا که با عمل چرخش، صفر از زوج ثبات نتیجه خارج شده است، گام بعدی برای اجرا، پرش به ابتدای حلقه تقسیم می‌باشد و حلقه دوباره تکرار می‌شود.



پس از ۱۶ بار اجرای حلقه، به نقطهٔ شکست که در انتهای روال تقسیم قرار دارد رسیده‌ایم. ثبات نتیجهٔ R5:R4 حاوی حاصل تقسیم، یعنی مقدار 0x0202، می‌باشد. ثبات‌های R2:R1:R0 خالی هستند، و بنابراین مقدار باقیمانده‌ای نداریم. اگر باقیمانده داشتیم می‌توانستیم از آن برای تعیین اینکه آیا برای گردکردن نتیجه به سمت بالا نیازی به افزایش مقدار نتیجه است یا نه، استفاده کنیم. این کار در اینجا برنامه‌نویسی نشده است.



اجرای کامل عملیات تقسیم، ۶۰ میکروثانیه از وقت پردازنده را می‌گیرد (پنجرهٔ Processor را از منوی برنامهٔ AVR Studio باز کنید)؛ زمانی نسبتاً طولانی برای یک عمل تقسیم.

تبدیل اعداد

این خودآموز، روال‌های مربوط به تبدیل اعداد را شامل نمی‌شود. اگر نیاز به کدهای منبع و یا فهم بهتری از این مبحث دارید، لطفاً به وب سایت مراجعه کنید.

اعداد اعشاری (Decimal Fractions)

نخست به این نکته توجه داشته باشید که نباید از اعداد اعشاری ممیز شناور استفاده کنید مگر اینکه واقعاً به آنها احتیاج داشته باشید. اعداد اعشاری ممیز شناور در AVR، منابع موجود را به شدت مصرف کرده، چیزهای مفیدی نبوده و برای اجرا به زمان زیادی نیاز دارند. اگر با این مقوله برخورد کردید و به این نتیجه رسیدید که اسمبلی بیش از حد پیچیده است، ترجیح خواهید داد که از Basic یا دیگر زبان‌ها از قبیل C یا Pascal استفاده کنید.

بنابراین اگر از اسمبلی استفاده کنید، چنین تفکری ندارید. در اینجا به شما نشان داده خواهد شد که چگونه می‌توانید ضرب اعداد حقیقی ممیز ثابت را در کمتر از ۶۰ میکروثانیه، و در موارد خاصی حتی در عرض ۱۸ میکروثانیه در فرکانس ساعت ۴ Mcs/s، انجام دهید؛ بدون اینکه به امکانات اضافی ممیز شناور در پردازنده، و یا دیگر روش‌های گران قیمت، که برای افراد تنبلی که از فکرشان استفاده نمی‌کنند ساخته شده است، نیازی داشته باشید.

چگونه می‌توان این کار را انجام داد؟ به ریشه‌های ریاضیات برگردید! بیشتر اَعمال انجام شده با اعداد حقیقی ممیز شناور با استفاده از اعداد صحیح قابل انجام هستند. اعداد صحیح در زبان اسمبلی به راحتی قابل برنامه‌نویسی بوده و سرعت

اجرای بالایی دارند. ممیز اعشاری تنها در ذهن برنامه‌نویس وجود دارد، و جایی میان دسته رقم‌های دهدهی قرار داده می‌شود. هیچ کس به وجود این ترفند پی نمی‌برد.

تبدیل‌های خطی

به عنوان یک مثال، این عمل را در نظر بگیرید: یک مبدل A/D هشت بیتی، سیگنال ورودی را که در گسترهٔ 0.00 تا 2.55 ولت تغییر می‌کند را اندازه‌گیری کرده و یک عدد دودویی در گسترهٔ بین $00 تا $FF را به عنوان خروجی برمی‌گرداند. خروجی، یعنی ولتاژ، باید بر روی یک نمایشگر LCD نمایش داده شود. یک مثال پیش پا افتاده که بسیار ساده است: عدد دودویی به یک رشتهٔ دهدهی ASCII مابین 000 و 255 تبدیل می‌شود و ممیز باید درست بعد از اولین رقم قرار داده شود. کل کار همین است!

دنیای الکترونیک گاهی اوقات پیچیده‌تر است. به طور مثال، مبدل A/D یک عدد مبنای شانزده ۸ بیتی را برای ولتاژهای ورودیِ بین 0.00 و 5.00 ولت برمی‌گرداند. اکنون به مشکل برخورده‌ایم و نمی‌دانیم چگونه کار را پیش ببریم. برای نمایش صحیح خروجی بر روی LCD می‌بایست عدد دودویی را در عدد ۲۵۵ / ۵۰۰، که برابر با ۱/۹۶۰۸ است، ضرب کنیم. این عدد تقریباً ۲ است، اما فقط تقریباً. و اصلاً دوست نداریم در شرایطی که صحت مبدل A/Dای که در اختیار داریم حدود ٪۰/۲۵ است، ٪۲ خطای تقریب داشته باشیم.

برای اینکه از عهده این کار برآییم، ورودی را در عدد ۲۵۶ × ۲۵۵ / ۵۰۰ یا ۵۰۱/۹۶ ضرب کرده و نتیجه را بر ۲۵۶ تقسیم می‌کنیم. چرا نخست در ۲۵۶ ضرب کرده و سپس بر ۲۵۶ تقسیم می‌کنیم؟ این کار تنها برای داشتن دقت زیاد است. اگر ورودی را به جای ضرب در ۵۰۱/۹۶ در ۵۰۲ ضرب کنیم، فقط خطایی حدود ٪۰/۰۰۸ خواهیم داشت. این مقدار خطا برای مبدل A/D ما به اندازهٔ کافی مناسب و برای ما نیز قابل تحمل است. از طرفی تقسیم بر ۲۵۶ عملی بسیار ساده است، زیرا این عدد توان شناخته‌شده‌ای از ۲ است. در AVR تقسیم بر اعدادی که توان‌هایی از ۲ هستند، به راحتی قابل انجام بوده و سرعت اجرای آن بسیار زیاد است. در تقسیم بر عدد ۲۵۶، AVR حتی سریع‌تر هم عمل می‌کند زیرا فقط باید آخرین بایت عدد دودویی را کنار بگذاریم؛ حتی به شیفت و چرخش هم نیازی نیست!

ضرب یک عدد دودویی ۸ بیتی در عدد دودویی ۹ بیتیِ ۵۰۲ (1F6 در مبنای شانزده) می‌تواند منجر به نتیجه‌ای بزرگ‌تر از ۱۶ بیت گردد. بنابراین باید ۲۴ بیت یا ۳ ثبات را به نتیجه اختصاص دهیم. در حین عمل ضرب، باید عدد ثابت ۵۰۲ به چپ شیفت داده شود (در ۲ ضرب شود) تا هرگاه رقم "1" از عدد ورودی به بیرون چرخانده شد، اعداد به دست آمده به نتیجه اضافه شوند. از آنجا که این کار ممکن است نیاز به هشت بار شیفت به چپ داشته باشد، سه بایت دیگر هم برای این عدد ثابت نیاز داریم. بنابراین ما برای عمل ضرب ترکیبی از ثبات‌ها را به شکل زیر انتخاب کرده‌ایم:

Number	*مقدار (نمونه)*	*ثبات*	*عدد*
Input value	۲۵۵	R1	مقدار ورودی
Multiplicator	۵۰۲	R4:R3:R2	مضروب فیه
Result	۱۲۸،۰۱۰	R7:R6:R5	نتیجه

پس از قرار دادن مقدار ۵۰۲ (00.01.F6) در R4:R3:R2 و پاک کردن ثبات‌های نتیجهٔ R7:R6:R5، عمل ضرب به این صورت انجام می‌شود:

۱. آزمایش اینکه آیا عدد ورودی صفر شده است. اگر اینگونه باشد، عمل ضرب به اتمام می‌رسد.
۲. اگر اینگونه نیست، یک بیت از عدد ورودی از سمت راست به داخل Carry انتقال داده شده و بیت ۷ با صفر پر می‌شود. این دستورالعمل، شیفت منطقی به راست یا LSR نامیده می‌شود.
۳. اگر بیت موجود در Carry یک باشد، مضروب فیه را (که در گام اول برابر ۵۰۲، در گام دوم برابر ۱۰۰۴ و به همین ترتیب است) به نتیجه اضافه می‌کنیم. در حین عمل جمع، رقم نقلی را نیز در نظر می‌گیریم (با دستور

ADD، R2 را به R5 اضافه می‌کنیم و با دستور ADC، R3 را به R6 و R4 را به R7 اضافه می‌کنیم!). اگر بیت موجود در Carry صفر باشد، مضروب فیه را به نتیجه اضافه نمی‌کنیم و به مرحله بعد می‌رویم.

٤. اکنون مضروب فیه را در ۲ ضرب می‌کنیم، زیرا ارزش مکانی بیت بعدی که به خارج از عدد ورودی شیفت داده می‌شود دو برابر است. بنابراین R2 را با استفاده از دستور LSL یک واحد به سمت چپ شیفت می‌دهیم (با درج صفر از سمت راست). بیت ۷ به داخل بیت Carry منتقل می‌شود. سپس با چرخاندن محتویات R3 به اندازهٔ یک بیت به سمت چپ، Carry را به داخل R3 می‌چرخانیم و بیت ۷ را به داخل Carry منتقل می‌کنیم. همین کار را با R4 نیز انجام می‌دهیم.

۵. تا اینجا پردازش یک رقم از عدد ورودی انجام شده و دوباره کار را از مرحلهٔ اول پیش می‌گیریم.

اکنون حاصل ضرب در عدد ۵۰۲، در ثبات‌های نتیجهٔ R7:R6:R5 قرار دارد. حالا اگر ثبات R5 را در نظر نگیریم (تقسیم بر ۲۵٦)، به نتیجه مطلوب رسیده‌ایم. به منظور افزایش دقت، می‌توانیم از بیت هفتمِ ثبات R5 برای گردکردن نتیجه استفاده کنیم. حالا تنها کار باقیمانده، تبدیل نتیجه از شکل دودویی به معادل دهدهی ASCII (بحث تبدیل اعداد دودویی به دهدهی ASCII را بر روی وب سایت ببینید) و قراردادن یک ممیز در محل صحیح است. اکنون رشتهٔ متنی ولتاژ، آماده نمایش می‌باشد.

اجرای کل برنامه، از عدد ورودی تا تولید رشته ASCII خروجی، بسته به عدد ورودی به تعداد سیکل ساعتی بین ۷۹ تا ۲۲۸ نیاز دارد. کسانی که علاقه دارند تا این روال را با روال ممیز شناور دیگری در زبان پیشرفته‌تری غیر از اسمبلی به مقایسه بگذارند، در صورت تمایل زمان تبدیل به دست آمده (همراه با فایل فلش برنامه و میزان حافظه مصرفی) را برای من پست کنند.

مثال ۱: مبدل A/D هشت بیتی با خروجی اعشاری ممیز ثابت

```
; Demonstrates floating point conversion in Assembler, (C)2003 www.avr-asm-tutorial.net
;
; The task: You read in an 8-bit result of an analogue-digital-converter, number is in the range from hex 00 to FF.
;    You need to convert this into a floating point number in the range from 0.00 to 5.00 Volt
; The program scheme:
;    1. Multiplication by 502 (hex 01F6).That step multiplies by 500, 256 and divides by 255 in one step!
;    2. Round the result and cut the last byte of the result. This step divides by 256 by ignoring the last byte of the result.
;       Before doing that, bit 7 is used to round the result.
;    3. Convert the resulting word to ASCII and set the correct decimal sign. The resulting word in the range from 0 to 500
;       is displayed in ASCII-characters as 0.00 to 5.00.
; The registers used:
;    The routines use the registers R8..R1 without saving these before. Also required is a multipurpose register called rmp,
;    located in the upper half of the registers. Please take care that these registers don't conflict with the register use in the
;    rest of your program.
;    When entering the routine the 8-bit number is expected in the register R1. The multiplication uses R4:R3:R2 to hold
;    the multiplicator 502 (is shifted left max. eight times during multiplication). The result of the multiplication is calculated
;    in the registers R7:R6:R5. The result of the so called division by 256 by just ignoring R5 in the result, is in R7:R6. R7:R6
;    is rounded, depending on the highest bit of R5, and the result is copied to R2:R1.
; Conversion to an ASCII-string uses the input in R2:R1, the register pair R4:R3 as a divisor for conversion, and places the
; ASCII result string to R5:R6:R7:R8 (R6 is the decimal char).
; Other conventions:
;    The conversion uses subroutines and the stack.The stack must work fine for the use of three levels (six bytes SRAM).
; Conversion times:
;    The whole routine requires 228 clock cycles maximum (converting $FF), and 79 clock cycles minimum (converting $00).
;    At 4 MHz the times are 56.75 microseconds resp. 17.75 microseconds.
; Definitions:
;    Registers
.DEF rmp = R16 ; used as multi-purpose register
;    AVR type: Tested for type AT90S8515, only required for stack setting, routines work fine with other AT90S-types also
.NOLIST
.INCLUDE "8515def.inc"
.LIST
; Start of test program
; Just writes a number to R1 and starts the conversion routine, for test purposes only
.CSEG
.ORG $0000
        rjmp main
main:
        ldi rmp,HIGH(RAMEND) ; Set the stack
        out SPH,rmp
        ldi rmp,LOW(RAMEND)
```

```
        out SPL,rmp
        ldi rmp,$FF ; Convert $FF
        mov R1,rmp
        rcall fpconv8 ; call the conversion routine
no_end:  ; unlimited loop, when done
        rjmp no_end
; Conversion routine wrapper, calls the different conversion steps
fpconv8:
        rcall fpconv8m ; multiplicate by 502
        rcall fpconv8r ; round and divide by 256
        rcall fpconv8a ; convert to ASCII string
        ldi rmp,'.' ; set decimal char
        mov R6,rmp
        ret ; all done
; Subroutine multiplication by 502
fpconv8m:
        clr R4 ; set the multiplicand to 502
        ldi rmp,$01
        mov R3,rmp
        ldi rmp,$F6
        mov R2,rmp
        clr R7 ; clear the result
        clr R6
        clr R5
fpconv8m1:
        or R1,R1 ; check if the number is all zeros
        brne fpconv8m2 ; still one's, go on convert
        ret ; ready, return back
fpconv8m2:
        lsr R1 ; shift number to the right (div by 2)
        brcc fpconv8m3 ; if the lowest bit was 0, then skip adding
        add R5,R2 ; add the number in R6:R5:R4:R3 to the result
        adc R6,R3
        adc R7,R4
fpconv8m3:
        lsl R2 ; multiply R4:R3:R2 by 2
        rol R3
        rol R4
        rjmp fpconv8m1 ; repeat for next bit
; Round the value in R7:R6 with the value in bit 7 of R5
fpconv8r:
        clr rmp ; put zero to rmp
        lsl R5 ; rotate bit 7 to carry
        adc R6,rmp ; add LSB with carry
        adc R7,rmp ; add MSB with carry
        mov R2,R7 ; copy the value to R2:R1 (divide by 256)
        mov R1,R6
        ret
; Convert the word in R2:R1 to an ASCII string in R5:R6:R7:R8
fpconv8a:
        clr R4 ; Set the decimal divider value to 100
        ldi rmp,100
        mov R3,rmp
        rcall fpconv8d ; get ASCII digit by repeated subtraction
        mov R5,rmp ; set hundreds string char
        ldi rmp,10 ; Set the decimal divider value to 10
        mov R3,rmp
        rcall fpconv8d ; get the next ASCII digit
        mov R7,rmp ; set tens string char
        ldi rmp,'0' ; convert the rest to an ASCII char
        add rmp,R1
        mov R8,rmp ; set ones string char
        ret
; Convert binary word in R2:R1 to a decimal digit by substracting the decimal divider value in R4:R3 (100, 10)
fpconv8d:
        ldi rmp,'0' ; start with decimal value 0
fpconv8d1:
        cp R1,R3 ; Compare word with decimal divider value
        cpc R2,R4
        brcc fpconv8d2 ; Carry clear, subtract divider value
        ret ; done subtraction
fpconv8d2:
        sub R1,R3 ; subtract divider value
        sbc R2,R4
        inc rmp ; up one digit
        rjmp fpconv8d1 ; once again
; End of conversion test routine
```

مثال ۲: مبدل A/D ده بیتی با خروجی اعشاری ممیز ثابت

این مثال کمی پیچیده‌تر است. در صورت نیاز به آن، به وب سایت مراجعه کنید.

ضمایم

فهرست دستورات براساس عملکرد

لیست اختصارات بکار رفته را ببینید.

Function	*Subfunction*	*Command*	*Flags*	*Clk*
Register set	0	CLR r1	Z N V	1
	255	SER rh		1
	Constant	LDI rh,c255		1
Copy	Register => Register	MOV r1,r2		1
	SRAM => Register, direct	LDS r1,c65535		2
	SRAM => Register	LD r1,rp		2
	SRAM => Register and INC	LD r1,rp+		2
	DEC, SRAM => Register	LD r1,-rp		2
	SRAM, displaced => Register	LDD r1,ry+k63		2
	Port => Register	IN r1,p1		1
	Stack => Register	POP r1		2
	Program storage Z => R0	LPM		3
	Register => SRAM, direct	STS c65535,r1		2
	Register => SRAM	ST rp,r1		2
	Register => SRAM and INC	ST rp+,r1		2
	DEC, Register => SRAM	ST -rp,r1		2
	Register => SRAM, displaced	STD ry+k63,r1		2
	Register => Port	OUT p1,r1		1
	Register => Stack	PUSH r1		2
Add	8 Bit, +1	INC r1	Z N V	1
	8 Bit	ADD r1,r2	Z C N V H	1
	8 Bit + Carry	ADC r1,r2	Z C N V H	1
	16 Bit, constant	ADIW rd,k63	Z C N V S	2
Subtract	8 Bit, -1	DEC r1	Z N V	1
	8 Bit	SUB r1,r2	Z C N V H	1
	8 Bit, constant	SUBI rh,c255	Z C N V H	1
	8 Bit - Carry	SBC r1,r2	Z C N V H	1
	8 Bit - Carry, constant	SBCI rh,c255	Z C N V H	1
	16 Bit	SBIW rd,k63	Z C N V S	2
Shift	logic, left	LSL r1	Z C N V	1
	logic, right	LSR r1	Z C N V	1
	Rotate, left over Carry	ROL r1	Z C N V	1
	Rotate, right over Carry	ROR r1	Z C N V	1
	Arithmetic, right	ASR r1	Z C N V	1
	Nibble exchange	SWAP r1		1
Binary	And	AND r1,r2	Z N V	1
	And, constant	ANDI rh,c255	Z N V	1
	Or	OR r1,r2	Z N V	1
	Or, constant	ORI rh,c255	Z N V	1
	Exclusive-Or	EOR r1,r2	Z N V	1
	Ones-complement	COM r1	Z C N V	1
	Twos-complement	NEG r1	Z C N V H	1

Function	*Subfunction*	*Command*	*Flags*	*Clk*
Bits change	Register, set	SBR rh,c255	Z N V	1
	Register, clear	CBR rh,255	Z N V	1
	Register, copy to T-Flag	BST r1,b7	T	1
	Register, copy from T-Flag	BLD r1,b7		1
	Port, set	SBI pl,b7		2
	Port, clear	CBI pl,b7		2
Statusbit set	Zero-Flag	SEZ	Z	1
	Carry Flag	SEC	C	1
	Negative Flag	SEN	N	1
	Twos complement carry Flag	SEV	V	1
	Half carry Flag	SEH	H	1
	Signed Flag	SES	S	1
	Transfer Flag	SET	T	1
	Interrupt Enable Flag	SEI	I	1
Statusbit clear	Zero-Flag	CLZ	Z	1
	Carry Flag	CLC	C	1
	Negative Flag	CLN	N	1
	Twos complement carry Flag	CLV	V	1
	Half carry Flag	CLH	H	1
	Signed Flag	CLS	S	1
	Transfer Flag	CLT	T	1
	Interrupt Enable Flag	CLI	I	1
Compare	Register, Register	CP r1,r2	Z C N V H	1
	Register, Register + Carry	CPC r1,r2	Z C N V H	1
	Register, constant	CPI rh,c255	Z C N V H	1
	Register, ≤0	TST r1	Z N V	1
Immediate Jump	Relative	RJMP c4096		2
	Indirect, Address in Z	IJMP		2
	Subroutine, relative	RCALL c4096		3
	Subroutine, Address in Z	ICALL		3
	Return from Subroutine	RET		4
	Return from Interrupt	RETI	I	4

Function	Subfunction	Command	Flags	Clk
Conditioned Jump	Statusbit set	BRBS b7,c127		1/2
	Statusbit clear	BRBC b7,c127		1/2
	Jump if equal	BREQ c127		1/2
	Jump if not equal	BRNE c127		1/2
	Jump if carry	BRCS c127		1/2
	Jump if carry clear	BRCC c127		1/2
	Jump if equal or greater	BRSH c127		1/2
	Jump if lower	BRLO c127		1/2
	Jump if negative	BRMI c127		1/2
	Jump if positive	BRPL c127		1/2
	Jump if greater or equal (Signed)	BRGE c127		1/2
	Jump if lower than zero (Signed)	BRLT c127		1/2
	Jump on half carry set	BRHS c127		1/2
	Jump if half carry clear	BRHC c127		1/2
	Jump if T-Flag set	BRTS c127		1/2
	Jump if T-Flag clear	BRTC c127		1/2
	Jump if Twos complement carry set	BRVS c127		1/2
	Jump if Twos complement carry clear	BRVC c127		1/2
	Jump if Interrupts enabled	BRIE c127		1/2
	Jump if Interrupts disabled	BRID c127		1/2
Conditioned Jumps	Registerbit=0	SBRC r1,b7		1/2/3
	Registerbit=1	SBRS r1,b7		1/2/3
	Portbit=0	SBIC pl,b7		1/2/3
	Portbit=1	SBIS pl,b7		1/2/3
	Compare, jump if equal	CPSE r1,r2		1/2/3
Others	No Operation	NOP		1
	Sleep	SLEEP		1
	Watchdog Reset	WDR		1

فهرست الفبایی دستورات

راهنماهای اسمبلر

دستورات

جزئیات پورت‌ها

جـدول پورت‌های ATMEL AVR مربوط به مـدل‌های AT90S2313، 2323 و 8515 در اینجا آورده شده است. جزئیات کامل پورت‌ها یا جفت ثبات‌های قابل دسترسی، به صورت بیت به بیت نشان داده نشده است. تضمینی در صحت این اطلاعات نیست، برگه‌های اطلاعاتی اصلی را ببینید!

ثبات وضعیت (Status Register) و پرچم‌های انباره (Accumulator Flags)

Port	*Function*	*Port-Address*	*RAM-Address*
SREG	Status Register Accumulator	0x3F	0x5F

7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
I	T	H	S	V	N	Z	C

Bit	*Name*	*Meaning*	*Opportunities*	*Command*
7	I	Global Interrupt Flag	0: Interrupts disabled	CLI
			1: Interrupts enabled	SEI
6	T	Bit storage	0: Stored bit is 0	CLT
			1: Stored bit is 1	SET
5	H	Halfcarry-Flag	0: No halfcarry occurred	CLH
			1: Halfcarry occurred	SEH
4	S	Sign-Flag	0: Sign positive	CLS
			1: Sign negative	SES
3	V	Two's complement-Flag	0: No carry occurred	CLV
			1: Carry occurred	SEV
2	N	Negative-Flag	0: Result was not negative/smaller	CLN
			1: Result was negative/smaller	SEN
1	Z	Zero-Flag	0: Result was not zero/unequal	CLZ
			1: Result was zero/equal	SEZ
0	C	Carry-Flag	0: No carry occurred	CLC
			1: Carry occurred	SEC

اشاره‌گر پشته (Stack Pointer)

Port	*Function*	*Port-Address*	*RAM-Address*
SPL/SPH	Stackpointer	003D/0x3E	0x5D/0x5E

Name	*Meaning*	*Availability*
SPL	Low-Byte of Stackpointer	From AT90S2313 upwards, not in 1200
SPH	High-Byte of Stackpointer	From AT90S8515 upwards, only in devices with >256 bytes internal SRAM

SRAM و کنترل وقفه‌های خارجی

Port	*Function*	*Port-Address*	*RAM-Address*
MCUCR	MCU General Control Register	0x35	0x55

7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
SRE	SRW	SE	SM	ISC11	ISC10	ISC01	ISC00

Bit	Name	Meaning	Opportunities
7	SRE	Ext.SRAM Enable	0=No external SRAM connected
			1=External SRAM connected
6	SRW	Ext.SRAM Wait States	0=No extra wait state on external SRAM
			1=Additional wait state on external SRAM
5	SE	Sleep Enable	0=Ignore SLEEP commands
			1=SLEEP on command
4	SM	Sleep Mode	0=Idle Mode (Half sleep)
			1=Power Down Mode (Full sleep)
3	ISC11	Interrupt control Pin INT1 (connected to GIMSK)	00: Low-level initiates Interrupt
2	ISC10		01: Undefined
			10: Falling edge triggers interrupt
			11: Rising edge triggers interrupt
1	ISC01	Interrupt control Pin INT0 (connected to GIMSK)	00: Low-level initiates interrupt
0	ISC00		01: Undefined
			10: Falling edge triggers interrupt
			11: Rising edge triggers interrupt

کنترل وقفه‌های خارجی

Port	Function	Port-Address	RAM-Address
GIMSK	General Interrupt Maskregister	0x3B	0x5B

7	6	5	4	3	2	1	0
INT1	INT0	-	-	-	-	-	-

Bit	Name	Meaning	Opportunities
7	INT1	Interrupt by external pin INT1 (connected to mode in MCUCR)	0: External INT1 disabled
			1: External INT1 enabled
6	INT0	Interrupt by external Pin INT0 (connected to mode in MCUCR)	0: External INT0 disabled
			1: External INT0 enabled
0...5	(Not used)		

Port	Function	Port-Address	RAM-Address
GIFR	General Interrupt Flag Register	0x3A	0x5A

7	6	5	4	3	2	1	0
INTF1	INTF0	-	-	-	-	-	-

Bit	Name	Meaning	Opportunities
7	INTF1	Interrupt by external pin INT1 occurred	Automatic clear by execution of the Int-Routine or Clear by command
6	INTF0	Interrupt by external pin INT0 occurred	
0...5	(Not used)		

کنترل وقفهٔ تایمر

Port	Function	Port-Address	RAM-Address
TIMSK	Timer Interrupt Maskregister	0x39	0x59

7	6	5	4	3	2	1	0
TOIE1	OCIE1A	OCIE1B	-	TICIE1	-	TOIE0	-

Bit	Name	Meaning	Opportunities
7	TOIE1	Timer/Counter 1 Overflow-Interrupt	0: No Int at overflow
			1: Int at overflow
6	OCIE1A	Timer/Counter 1 Compare A Interrupt	0: No Int at equal A
			1: Int at equal A
5	OCIE1B	Timer/Counter 1 Compare B Interrupt	0: No Int at B
			1: Int at equal B
4	(Not used)		
3	TICIE1	Timer/Counter 1 Capture Interrupt	0: No Int at Capture
			1: Int at Capture
2	(Not used)		
1	TOIE0	Timer/Counter 0 Overflow-Interrupt	0: No Int at overflow
			1: Int at overflow
0	(Not used)		

Port	Function	Port-Address	RAM-Address
TIFR	Timer Interrupt Flag Register	0x38	0x58

7	6	5	4	3	2	1	0
TOV1	OCF1A	OCF1B	-	ICF1	-	TOV0	-

Bit	Name	Meaning	Opportunities
7	TOV1	Timer/Counter 1 Overflow reached	Interrupt-Mode: Automatic Clear by execution of the Int-Routine OR Polling-Mode: Clear by command
6	OCF1A	Timer/Counter 1 Compare A reached	
5	OCF1B	Timer/Counter 1 Compare B reached	
4	(Not used)		
3	ICF1	Timer/Counter 1 Capture-Event occurred	
2	(not used)		
1	TOV0	Timer/Counter 0 Overflow occurred	
0	(not used)		

تایمر/شمارندهٔ ۰

Port	Function	Port-Address	RAM-Address
TCCR0	Timer/Counter 0 Control Register	0x33	0x53

7	6	5	4	3	2	1	0
-	-	-	-	-	CS02	CS01	CS00

Bit	Name	Meaning	Opportunities
2..0	CS02..CS00	Timer Clock	000: Stop Timer
			001: Clock = Chip clock
			010: Clock = Chip clock / 8
			011: Clock = Chip clock / 64
			100: Clock = Chip clock / 256
			101: Clock = Chip clock / 1024
			110: Clock = falling edge of external Pin T0
			111: Clock = rising edge of external Pin T0
3..7	(not used)		

Port	Function	Port-Address	RAM-Address
TCNT0	Timer/Counter 0 count register	0x32	0x52

تایمر/شمارندهٔ ۱

Port	Function	Port-Address	RAM-Address
TCCR1A	Timer/Counter 1 Control Register A	0x2F	0x4F

7	6	5	4	3	2	1	0
COM1A1	COM1A0	COM1B1	COM1B0	-	-	PWM11	PWM10

Bit	Name	Meaning	Opportunities
7	COM1A1	Compare Output A	00: OC1A/B not connected 01: OC1A/B changes polarity 10: OC1A/B to zero 11: OC1A/B to one
6	COM1A0		
5	COM1B1	Compare Output B	
4	COM1B0		
3	(not used)		
2			
1..0	PWM11 PWM10	Pulse width modulator	00: PWM off 01: 8-Bit PWM 10: 9-Bit PWM 11: 10-Bit PWM

Port	Function	Port-Address	RAM-Address
TCCR1B	Timer/Counter 1 Control Register B	0x2E	0x4E

7	6	5	4	3	2	1	0
ICNC1	ICES1	-	-	CTC1	CS12	CS11	CS10

Bit	Name	Meaning	Opportunities
7	ICNC1	Noise Canceler on ICP-Pin	0: disabled, first edge starts sampling
			1: enabled, min four clock cycles
6	ICES1	Edge selection on Capture	0: falling edge triggers Capture
			1: rising edge triggers Capture
5..4	(not used)		
3	CTC1	Clear at Compare Match A	1: Counter set to zero if equal
2..0	CS12..CS10	Clock select	000: Counter stopped 001: Clock 010: Clock / 8 011: Clock / 64 100: Clock / 256 101: Clock / 1024 110: falling edge external Pin T1 111: rising edge external Pin T1

Port	Function	Port-Address	RAM-Address
TCNT1L/H	Timer/Counter 1 count register	0x2C/0x2D	0x4C/0x4D

Port	Function	Port-Address	RAM-Address
OCR1AL/H	Timer/Counter 1 Output Compare register A	0x2A/0x2B	0x4A/0x4B hex

Port	Function	Port-Address	RAM-Address
OCR1BL/H	Timer/Counter 1 Output Compare register B	0x28/0x29	0x48/0x49

Port	Function	Port-Address	RAM-Address
ICR1L/H	Timer/Counter 1 Input Capture Register	0x24/0x25	0x44/0x45

تایمر Watchdog

Port	Function	Port-Address	RAM-Address
WDTCR	Watchdog Timer Control Register	0x21	0x41

7	6	5	4	3	2	1	0
-	-	-	WDTOE	WDE	WDP2	WDP1	WDP0

Bit	Name	Meaning	WDT-cycle at 5.0 Volt
7..5	(not used)		
4	WDTOE	Watchdog Turnoff Enable	Previous set to disabling of WDE required
3	WDE	Watchdog Enable	1: Watchdog active
2..0	WDP2..WDP0	Watchdog Timer Prescaler	000: 15 ms 001: 30 ms 010: 60 ms 011: 120 ms 100: 240 ms 101: 490 ms 110: 970 ms 111: 1.9 s

EEPROM

Port	Function	Port-Address	RAM-Address
EEARL/H	EEPROM Address Register	0x1E/0x1F	0x3E/0x3F

EEARH فقط در مدل‌های با حافظهٔ EEPROM بیشتر از ٢٥٦ بایت (از AT90S8515 به بالا) موجود است.

Port	Function	Port-Address	RAM-Address
EEDR	EEPROM Data Register	0x1D	0x3D

Port	Function	Port-Address	RAM-Address
EECR	EEPROM Control Register	0x1C	0x3C

7	6	5	4	3	2	1	0
-	-	-	-	-	EEMWE	EEWE	EERE

Bit	Name	Meaning	Function
7..3	(not used)		
2	EEMWE	EEPROM Master Write Enable	Previous set enables write cycle
1	EEWE	EEPROM Write Enable	Set to initiate write
0	EERE	EEPROM Read Enable	Set initiates read

رابط سریال جانبی (SPI)

Port	Function	Port-Address	RAM-Address
SPCR	SPI Control Register	0x0D	0x2D

7	6	5	4	3	2	1	0
SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

Bit	Name	Meaning	Function
7	SPIE	SPI Interrupt Enable	0: Interrupts disabled
			1: Interrupts enabled
6	SPE	SPI Enable	0: SPI disabled
			1: SPI enabled
5	DORD	Data Order	0: MSB first
			1: LSB first
4	MSTR	Master/Slave Select	0: Slave
			1: Master
3	CPOL	Clock Polarity	0: Positive Clock Phase
			1: Negative Clock Phase
2	CPHA	Clock Phase	0: Sampling at beginning of Clock Phase
			1: Sampling at end of Clock Phase
1	SPR1	SCK clock frequency	00: Clock / 4
0	SPR0		01: Clock / 16
			10: Clock / 64
			11: Clock / 128

Port	Function	Port-Address	RAM-Address
SPSR	SPI Status Register	0x0E	0x2E

7	6	5	4	3	2	1	0
SPIF	WCOL	-	-	-	-	-	-

Bit	Name	Meaning	Function
7	SPIF	SPI Interrupt Flag	Interrupt request
6	WCOL	Write Collision Flag	Write collision occurred
5..0	(not used)		

Port	Function	Port-Address	RAM-Address
SPDR	SPI Data Register	0x0F	0x2F

UART

Port	Function	Port-Address	RAM-Address
UDR	UART I/O Data Register	0x0C	0x2C

Port	Function	Port-Address	RAM-Address
USR	UART Status Register	0x0B	0x2B

7	6	5	4	3	2	1	0
RXC	TXC	UDRE	FE	OR	-	-	-

Bit	Name	Meaning	Function
7	RXC	UART Receive Complete	1: Char received
6	TXC	UART Transmit Complete	1: Shift register empty
5	UDRE	UART Data Register Empty	1: Transmit register available
4	FE	Framing Error	1: Illegal Stop-Bit
3	OR	Overrun	1: Lost char
2..0	(not used)		

Port	Function	Port-Address	RAM-Address
UCR	UART Control Register	0x0A	0x2A

7	6	5	4	3	2	1	0
RXCIE	TXCIE	UDRIE	RXEN	TXEN	CHR9	RXB8	TXB8

Bit	Name	Meaning	Function
7	RXCIE	RX Complete Interrupt Enable	1: Interrupt on received char
6	TXCIE	TX Complete Interrupt Enable	1: Interrupt at transmit complete
5	UDRIE	Data Register Empty Interrupt Enable	1: Interrupt on transmit buffer empty
4	RXEN	Receiver Enable	1: Receiver enabled
3	TXEN	Transmitter Enable	1: Transmitter enabled
2	CHR9	9-bit Characters	1: Char length 9 Bit
1	RXB8	Receive Data Bit 8	9th Data bit on receive
0	TXB8	Transmit Data Bit 8	9th Data bit on transmit

Port	Function	Port-Address	RAM-Address
UBRR	UART Baud Rate Register	0x09	0x29

مقایسه‌کنندهٔ آنالوگ

Port	Function	Port-Address	RAM-Address
ACSR	Analog Comparator Control and Status Register	0x08	0x28

7	6	5	4	3	2	1	0
ACD	-	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0

Bit	Name	Meaning	Function
7	ACD	Disable	Disable Comparators
6	(not used)		
5	ACO	Comparator Output	Read: Output of the Comparators
4	ACI	Interrupt Flag	1: Interrupt request
3	ACIE	Interrupt Enable	1: Interrupts enabled
2	ACIC	Input Capture Enable	1: Connect to Timer 1 Capture

Bit	Name	Meaning	Function
1	ACIS1	Input Capture Enable	00: Interrupt on edge change
0	ACIS0		01: (not used)
			10: Interrupt on falling edge
			11: Interrupt on rising edge

پورت‌های ورودی- خروجی

Port	Register	Function	Port-Address	RAM-Address
A	PORTA	Data Register	0x1B	0x3B
	DDRA	Data Direction Register	0x1A	0x3A
	PINA	Input Pins Address	0x19	0x39
B	PORTB	Data Register	0x18	0x38
	DDRB	Data Direction Register	0x17	0x37
	PINB	Input Pins Address	0x16	0x36
C	PORTC	Data Register	0x15	0x35
	DDRC	Data Direction Register	0x14	0x34
	PINC	Input Pins Address	0x13	0x33
D	PORTD	Data Register	0x12	0x32
	DDRD	Data Direction Register	0x11	0x31
	PIND	Input Pins Address	0x10	0x30

فهرست الفبایی پورت‌ها

ACSR, Analog Comparator Control and Status Register
DDRx, Port x Data Direction Register
EEAR, EEPROM Address Register
EECR, EEPROM Control Register
EEDR, EEPROM Data Register
GIFR, General Interrupt Flag Register
GIMSK, General Interrupt Mask Register
ICR1L/H, Input Capture Register 1
MCUCR, MCU General Control Register
OCR1A, Output Compare Register 1 A
OCR1B, Output Compare Register 1 B
PINx, Port Input Access
PORTx, Port x Output Register
SPL/SPH, Stackpointer
SPCR, Serial Peripheral Control Register
SPDR, Serial Peripheral Data Register
SPSR, Serial Peripheral Status Register
SREG, Status Register
TCCR0, Timer/Counter Control Register, Timer 0
TCCR1A, Timer/Counter Control Register 1 A
TCCR1B, Timer/Counter Control Register 1 B
TCNT0, Timer/Counter Register, Counter 0
TCNT1, Timer/Counter Register, Counter 1
TIFR, Timer Interrupt Flag Register
TIMSK, Timer Interrupt Mask Register
UBRR, UART Baud Rate Register
UCR, UART Control Register
UDR, UART Data Register
WDTCR, Watchdog Timer Control Register

اختصارات بکار رفته

نام‌های اختصاری به گونه‌ای انتخاب شده‌اند که مشخص‌کننده یک محدوده باشند. زوج ثبات‌ها از روی بایت پایین دو ثبات نامگذاری شده‌اند. مقادیر ثابتِ مربوط به فرامین پرشی (آدرس‌ها یا آفست‌ها)، هنگام اسمبل‌کردن از روی برچسب‌های مربوطه محاسبه می‌شوند.

Category	*Abbrev.*	*Means ...*	*Value range*
Register	r1	Ordinary Source and Target register	R0..R31
	r2	Ordinary Source register	
	rh	Upper page register	R16..R31
	rd	Twin register	R24(R25), R26(R27), R28(R29), R30(R31)
	rp	Pointer register	X=R26(R27), Y=R28(R29), Z=R30(R31)
	ry	Pointer register with displacement	Y=R28(R29), Z=R30(R31)
Constant	k63	Pointer-constant	0..63
	c127	Conditioned jump distance	-64..+63
	c255	8-Bit-Constant	0..255
	c4096	Relative jump distance	-2048..+2047
	c65535	16-Bit-Address	0..65535
Bit	b7	Bit position	0..7
Port	p1	Ordinary Port	0..63
	pl	Lower page port	0..31