
# Introdução para o iniciante à Linguagem Assembly dos Microprocessadores ATMEL-AVR

***por***

***Gerhard Schmidt***

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

***Dezembro de 2003***

Versão corrigida em Julho de 2006

Correções adicionais e atualizações em Janeiro de 2008

Traduzido por Guilherme Groke – *ggroke@gmail.com*

# Conteúdo

# Porque aprender Assembler?

Assembler ou outras linguagens, esta é a questão. Porque eu deveria aprender mais uma linguagem, se eu já conheço outras linguagens de programação? O melhor argumento: enquanto você viver na França, poderá sobreviver falando Inglês, mas você nunca se sentirá em casa, e a vida fica difícil. Você pode apenas continuar desta forma, mas isto não é apropriado. Se as coisas ficarem complicadas, você deverá usar a linguagem corrente do país

Muitas pessoas já experientes em programação de AVRs e que usam linguagens de alto nível em seu trabalho normal, recomendam que os iniciantes comecem aprendendo linguagem assembly. A razão para isto é que, algumas vezes, podem ocorrer certas situações, como:

- se bugs têm que ser analizados,
- se o programa faz coisas diferentes das que foram escritas e são esperadas,
- se linguagens de alto nível não suportam o uso de certas características do hardware,
- se rotinas em que a temporização é crítica requerem porções de linguagem assembly,

é necessário entender a linguagem assembly, e.g., entender o que os compiladores de linguagens de alto nível produzem. Sem compreender a linguagem assembly, você não terá chance de ir adiante nestes casos.

## Curta e fácil

Os comandos assembler são traduzidos um a um para serem comandos executados pela máquina. O processador necessita apenas executar o que você quer fazer e o necessário para executar a tarefa. Nenhum loop extra ou características desnecessárias poluem o código. Se o espaço para o seu programa é curto e limitado e você terá que otimizar seu programa para caber na memória, assembler é a escolha número um. Programas mais curtos são mais fáceis de depurar (“debugar”), cada passo faz sentido.

## Veloz

Como apenas os passos necessários são executados, os programas em assembly são tão rápidos quanto possível. Aplicações onde o tempo é crítico, como medições de tempo que devam ter boa performance, sem que haja um hardware de temporização, devem ser escritas em assembler. Se você tiver mais tempo e não se importar que seu chip permaneça 99% em um estado de espera (wait state) de operação, você pode escolher a linguagem que desejar.

## Assembler é de fácil aprendizado

Não é verdade que a linguagem assembly é mais complicada ou não é tão fácil de compreender quanto outras linguagens. Aprender linguagem assembly para qualquer tipo de hardware facilita a compreensão de conceitos de qualquer outro dialeto da linguagem assembly. Aprender outros dialetos depois é mais fácil. Algumas características são dependentes do hardware, e isto requer alguma familiaridade com os conceitos de hardware e seus dialetos. O que faz o assembler parecer complicado algumas vezes é que ele requer uma compreensão das funções do controlador do hardware. Linguagens de alto nível não permitem a utilização de características especiais do hardware, e escondem estas funções.

O primeiro código assembly não parece muito interessante, mas depois de 100 linhas adicionais programadas, parecerá melhor. Programas perfeitos requerem apenas alguns milhares de linhas de código de exercício, e otimização requer bastante trabalho. Os primeiros passos são difíceis em qualquer linguagem. Após algumas semanas programando, você dará risada se analisar seu primeiro código. Alguns comandos em assembler requerem meses de experiência.

## AVRs são ideais para se aprender assembler

Programas em assembler são um pouco tolos: o chip executa tudo que você disser a ele para fazer, e não pergunta se você tem certeza se quer sobrescrever isso ou aquilo. Todas as características de proteção devem ser programadas por você, o chip faz exatamente aquilo que lhe é comandado, mesmo que não faça sentido algum. Nenhuma janela o alertará, a menos que você a tenha programado anteriormente.

Para corrigir erros de digitação é tão fácil ou complicado como qualquer outra linguagem. Existem erros básicos ou mais complicados. Porém: testar os programas nos chips ATMEL é muito fácil. Se o chip não faz o que você espera, você pode facilmente adicionar algumas linhas de diagnóstico ao código, reprogramá-lo e testá-lo. Adeus, programadores de EPROM, lâmpadas UV usadas para apagar o programa, pinos que não se encaixam mais no soquete após tê-los removido uma dúzia de vezes.

As mudanças agora são programadas rapidamente, compiladas imediatamente, ou mesmo simuladas no studio ou checadas no próprio circuito. Nenhum pino tem que ser removido, e nenhuma lâmpada de UV te

deixará na mão justamente no momento em que você teve uma excelente idéia sobre aquele bug.

# Teste!

Seja paciente nos seus primeiros passos! Se você tem familiaridade com outra linguagem (alto nível): esqueça-a por enquanto. A maioria das características especiais de outras linguagens de computação não fazem nenhum sentido em assembler.

As primeiras cinco instruções não são fáceis de aprender, depois disso sua velocidade de aprendizado aumentará rapidamente. Depois que você escreveu as primeiras linhas: pegue o conjunto de instruções e leia-o deitado em sua banheira, imaginando para que servem todas as outras instruções.

Aviso sério: Não tente criar uma mega-máquina logo de início. Isto não faz sentido algum em nenhuma linguagem de programação, e apenas produz frustração. Comece com pequenos exemplos tipo “Olá Mundo”, e.g., ligando e desligando LEDs por algum tempo, e só então comece a explorar as características do hardware mais profundamente.

Recomendação: Comente suas subrotinas e armazene-as em um diretório especial, se debugadas: você precisará delas em breve.

Sucesso!

# Hardware para programação em Assembler AVR

Aprender assembler requer um equipamento simples para testar seus programas e ver se eles funcionam na prática.

Está página mostra dois esquemas fáceis que permitem a você construir em casa o hardware necessário. Este hardware é realmente fácil de construir. Desconheço qualquer esquema mais fácil do que este para o seus primeiros passo em software. Se você quiser fazer mais experimentos, reserve espaço para expansões futuras em sua placa de experimentos.

Se você não gosta do cheiro de solda, você pode comprar uma placa pronta para uso. As placas disponíveis estão listadas na seção abaixo.

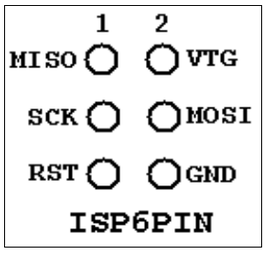
## A interface ISP da família de processadores AVR

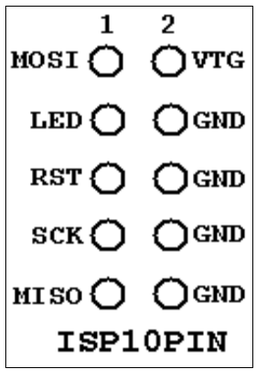
Antes de entrarmos na prática, temos que aprender alguns conceitos essenciais no modo de programação serial da família AVR. Não, você não precisa três diferentes voltagens para programar e ler a memória flash do AVR. Não, você não precisa de outro microprocessador para programar os AVRs. Não você não precisa de 10 linhas de E/S para dizer ao chip o que você gostaria que ele fizesse. E você não tem sequer que remover o AVR da sua placa de experimentos, antes de programá-lo. É fácil assim.

Tudo isto é feito pela interface embutida em cada chip AVR, que permite a você escrever e ler o conteúdo da memória flash e EEPROM embutidas. Esta interface trabalha serialmente e precisa de três linhas de sinal:

- SCK: Um sinal de clock que move os bits a serem escritos na memória para um shift register interno, que move os bits para serem lidos por outro shift register.
- MOSI: Um sinal de dados que envia os bits para serem escritos no AVR,
- MISO: Um sinal de dados que recebe os bits lidos do AVR.

Estes três pinos de sinal são internamente conectados ao sistema de programação apenas se você deixar o sinal RESET (algumas vezes chamado de RST ou restart) em nível zero. Caso contrário, durante a operação normal do AVR, estes pinos são portas de E/S programáveis como todas as outras. Se você quiser usar estes pinos para outros propósitos durante a operação normal, e para programação ISP, você terá que fazê-lo de forma que estes dois propósitos não conflitem. Normalmente você deve desacoplar os sinais por um resistor ou utilizar um multiplexador. O necessário para o seu caso, depende do seu uso dos pinos no modo de operação normal. Se você tiver sorte, pode deixar estes pinos exclusivamente paa a programação ISP. Não necessário, porém recomendado para a programação ISP, é que você tenha uma tensão de alimentação para o hardware no próprio circuito de programação. Isto torna a programação mais fácil, e requer apenas duas linhas adicionais entre o programador e a placa AVR. GND é o terra comum, VTG (target voltage) é a tensão de alimentação (normalmente 5.0 Volts). Isto totaliza 6 linhas entre o programador e a placa AVR. A conexão ISP6 é, conforme definida pela ATMEL, mostrada à esquerda.



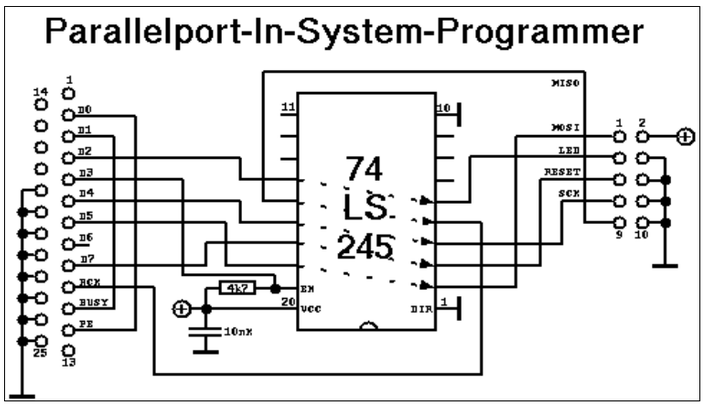

Padrões sempre têm padrões alternativos. Esta é a base técnica que constitui a empresa de adaptadores. No nosso caso, o padrão alternativo foi projetado utilizando um ISP10 e foi utilizado na placa STK200. É um padrão bastante disseminado, e mesmo o STK500 está equipado com este padrão. O padrão ISO10 tem um sinal adicional que aciona um LED vermelho. Este LED sinaliza que o programador está fazendo o seu trabalho. Uma boa idéia. Conecte o LED a um resistor e à fonte de tensão.

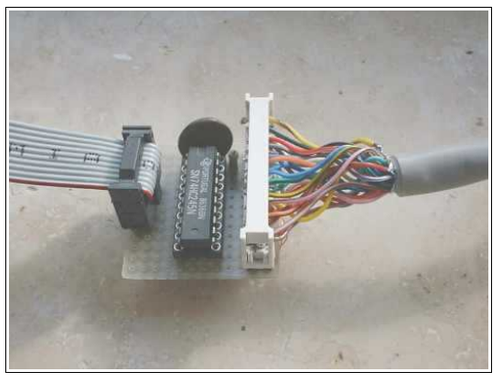
## Programador para a porta paralela do PC

Agora, aqueça o seu ferro de soldar e construiremos seu programador. É um esquema bastante fácil e funciona com peças facilmente encontráveis na sua caixa de experimentos.

Sim, é tudo que você precisa para programar um AVR. O plugue de 25 pinos se liga à porta paralela do seu PC, e o conector ISP de 10 pinos se conecta à placa de experimentos AVR. Se você não tiver um 72LS245, você pode utilizar também o 74HC245 (sem mudanças no circuito) ou um 74LS244/74HC244

(com pequenas mudanças no circuito). Se você utilizar HC, não se esqueça de conectar os pinos não utilizados ao terra ou à tensão de alimentação, caso contrário os buffers podem produzir ruído capacitivo pelo chaveamento.



Todo o algoritmo de programação necessário é feito pelo software ISP. Esteja ciente que esta interface paralela não é mais suportada pelo software Atmel Studio. Portanto, se você desejar programar seu AVR diretamente do studio, utilize programadores diferentes. A internet fornece diversas soluções.

Se você já tiver uma placa de programação, você não precisará construir este programador, porque você encontrará a interface ISP em alguns pinos. Consulte o manual para localizá-los.

# Placas experimentais

Você provavelmente quer fazer seus primeiros programas com uma placa AVR feita em casa. Aqui apresentamos duas versões:

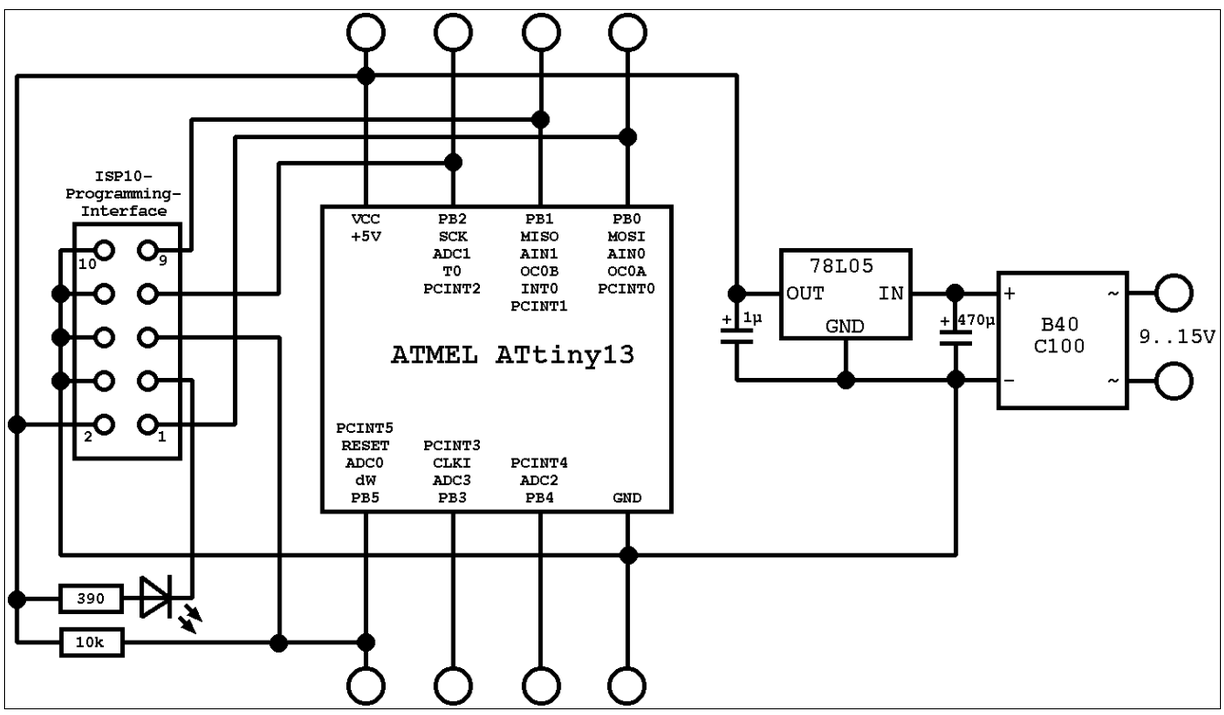
- Uma bem pequena com um ATtiny13, ou
- uma mais complicada com um AT90S2313 ou ATmega2313, incluindo uma interface serial RS232.

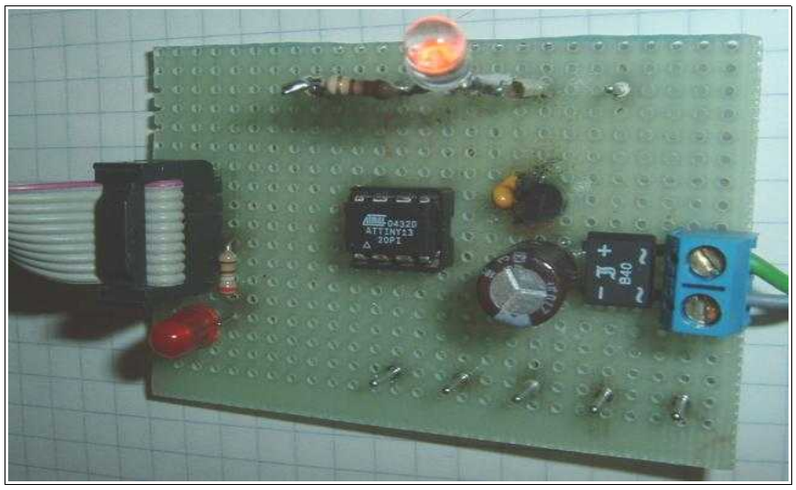
## Placa experimental com ATtiny13

Esta é uma placa bem pequena que permite experimentos com o hardware interno do Attiny13. A figura mostra

- A interface de programação ISP10 à esquerda, com o LED de programação ligado a um resistor de 390 ohms,
- o ATtiny13 com um resistor pull-up de10k no pino RESET (pino 1),

1. a fonte de alimentação com um retificador em ponte, que aplica de 9 a 15V a partir de uma fonte AC ou DC, e um pequeno regulador de 5V.





O ATtiny13 não requer cristal externo ou gerador de clock, pois trabalha com o seu gerador RC interno de 9,6 Mcs/s e, por padrão, com um divisor de clock em 8 (freqüência de clock de 1,2 Mcs/s).

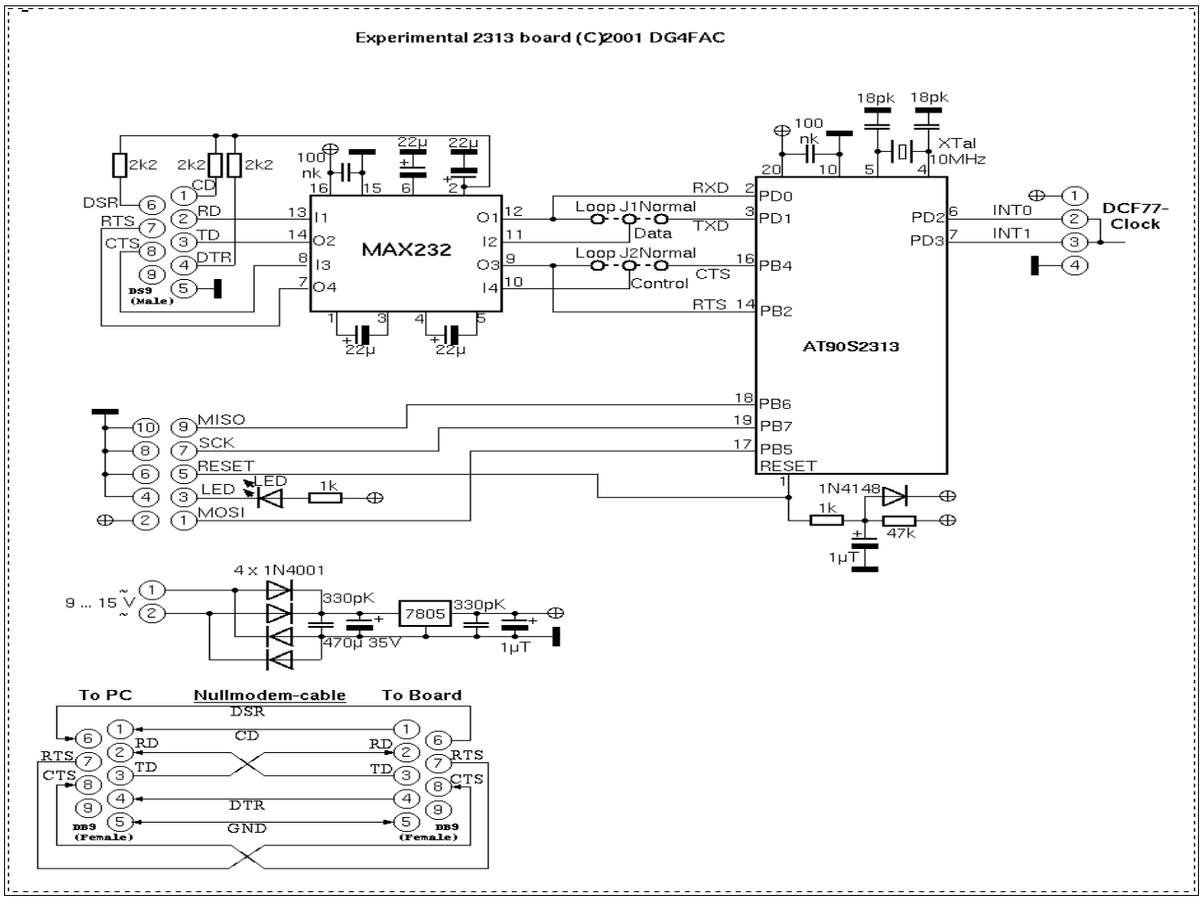
O hardware pode ser construido em uma pequena placa como a mostrada na figura. Todos os pinos do tiny13 são acessíveis, e componentes externos ao hardware, como o LED mostrado, são facilmente conectados.

Esta placa permite o uso dos componentes de hardware do Attiny13, como portas E/S, timers, conversores AD, etc.

## Placa experimental com um AT90S2313/ATmega2313

Para fins de teste, ou se mais pinos de E/S ou comunicação serial forem necessários, podemos utilizar um AT90S2313 ou Atmega2313 em uma placa experimental. O esquema mostra

- uma pequena fonte de alimenação para conecção a um transformador AC e um regulador de tensão 5V/1A,
- um gerador de clock a cristal (aqui com um cristal de 10 MHz, qualquer freqüência abaixo do máximo especificado para o AT90S2313 funcionarão), os componentes necessários para um reset seguro durante o momento da partida da alimentação,
- a interface de programação ISP (com o conector ISP10).




É isto que você precisa para iniciar. Conecte os outros periféricos e adicionais aos numerosos pinos livres de E/S ao 2313.

O dispositivo de saída mais fácil pode ser um LED, conectado a um resistor à tensão de alimentação. Com isso, você pode escrever o seu primeiro programa em assembler para acender e apagar o LED.

Se você

- não precisar da interface de comunicação serial, simplesmente ignore o hardware conectado aos pinos 2/3 e 14/16,

- não precisar dos sinais de handshaking, ignore os pinos 14/16 e conecte RTS ao conector de 9 pinos via um resistor de 2,2k a +9V.

Se você usar um ATmega2313 no lugar do AT90S2313, serão feitas as seguintes mundanças:

- o cristal externo não é necessário, pois o ATmega possui um gerador de clock RC interno, basta ignorar as conexões aos pinos 4 e 5,
- se você quiser usar um cristal externo no lugar do circuito RC interno, você terá que programar os fusíveis do Atmega de acordo.

# Placas de programação comerciais para a família AVR

Se você não gosta de hardware feito em casa, e tem algum dinheiro extra e não sabe o que fazer com ele, você pode comprar uma placa de programação comercial pronta. Dependendo da quantidade de dinheiro extra que você gostaria de gastar, você pode escolher entre versões mais baratas ou mais caras. Para o amador, a critério de seleção deve ser baseado em:

- preço,
- interface com o PC (de preferência USB; menos conveniente ou durável: RS232; requer software adicional para programação: porta paralela do PC),
- suporte a novos dispositivos (atualizações são necessárias de tempos em tempos, caso contrário você montará em um cavalo morto),
- características do hardware (depende de suas necessidades para os próximos cinco anos).

A seção a seguir descreve três placas padrão da ATMEL, a STK200, a STK500 e a Dragon. A seleção é baseada em minhas próprias experiências, e não é uma recomendação.

## STK200

A STK200 da ATMEL é uma placa histórica. Se você conseguir uma usada, você terá

- uma placa com alguns soquetes (para dispositivos de 8, 20, 28 e 40 pinos),
- oito chaves e LEDs, conectados permanentemente às portas D e B,
- um LCD com interface padrão de 14 pinos,
- uma opção para conectar uma SRAM de 28 pinos,
- uma interface RS232 para comunicação,
- um cabo de interface com a porta paralela do PC em um lado e um ISP de 10 pinos do outro.

Programação em HV (alta voltagem) não é suportada.

A placa não pode ser programada do Studio, o software de programação não é mais suportado, e você deve usar programas externos capazes de programar via porta paralela.

Se alguém lhe oferecer esta placa, pegue-a apenas de graça, e se você estiver acostumado a utilizar os softwares necessários.

## STK500

É fácil obter a STK500 (e.g. direto da ATMEL). Esta placa tem o seguinte hardware:

- Soquetes para programar a maioria dos tipos de AVR (e.g. Dispositivos de 14 pinos ou encapsulamentos TQFP, exigindo algum hardware adicional),
- programação serial e paralela em modo normal ou alta voltagem (programação HV traz de volta à vida dispositivos mesmo se o pino RESET foi programado para ser uma porta de entrada),
- Conexões ISP6 e ISP10 para programação ISP,
- oscilador com freqüência e tensões de alimentação programáveis,
- chaves e LEDs configuráveis,
- um conector RS232C (UART),
- uma EEPROM-Flash serial (somente as placas mais antigas),
- acesso a todas as portas via conectores de 10 pinos.

A principal desvantagem desta placa é que, após programar um dispositivo, várias conexões têm que ser feitas manualmente com os cabos que a acompanham.

A placa é conectada ao PC usando uma porta serial (COMx). Se o seu laptop não possui uma interface serial, você pode utilizar um conversor USB-Serial comum com driver. Neste caso, o driver deve ser ajustada para usar entre COM1 e COM2 e uma velocidade de 115k para ser detectada automaticamente pelo software Studio.

A programação é realizada e controlada pelas versões recentes do AVR studio, que está disponível gratuitamente para download pela página da ATMEL após o registro. Atualizações na lista de dispositivos e algoritmos de programação são fornecidos com as versões do Studio, então o suporte para novos dispositivos é mais provável do que com outras placas e softwares de programação.

Os experimentos podem começar com o AVR fornecido (versões antigas: AT90S8515, novas placas incluem tipos diferentes). Este kit cobre todas as necessidades que o iniciante pode ter.

## AVR Dragon

O AVR Dragon é uma pequena placa. Ela possui uma interface USB, que alimenta a placa e a interface ISP de 6 pinos. A interface ISP de 6 pinos é acompanhada pela interface de programação HV de 20 pinos. A placa é preparada para receber alguns soquetes, mas não possui soquetes para dispositivos para gravação ou outro hardware.

O dragon é suportado pelo software Studio e é atualizado automaticamente.

Seu preço e design o torna um excelente presente para o amador de AVR.

# Ferramentas para programação assembly AVR

Quatro ferramentas básicas são necessários para programação em assembly. Estas ferramentas são:

- o editor,
- o compilador,
- a interface para programação do chip, e
- o simulador.

Para realizar estas tarefas, existem dois caminhos:

1. todos as ferramentas em um só programa,
2. cada tarefa é realizada por um programa específico, e os resultados são armazenados em arquivos específicos.

Normalmente a primeira opção é escolhida. Porém, como isto é um tutorial, e você deve compreencer o mecanismo primeiro, começaremos descrevendo a segunda opção.

# De um arquivo texto a palavras de instrução para a memória flash

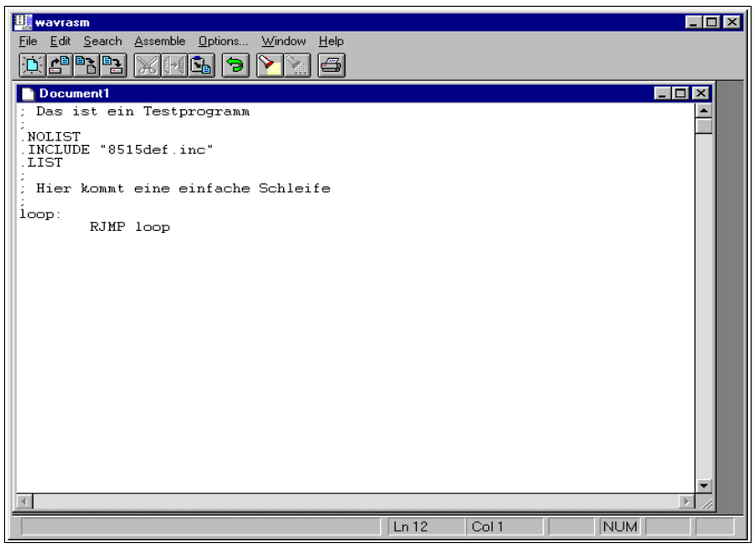
## O editor

Os programas assembler são escritos com um editor. O editor simplesmente tem que criar e editar texto ASCII. Então, basicamente, qualquer editor simples serve.

Algumas características do editor podem ter efeitos positivos:

- Erros, que o assembler detecta depois, são reportados juntamente com o número da linha do arquivo texto. Números de linha são também uma poderosa invenção da era do computador com respeito a discussões no seu código com alguém. Então o seu editor deve ser capaz de mostrar o número da linha. Infelizmente, quase todos os editores que uma poderosa empresa de software fornece como parte dos seus sistemas operacionais, não possuem esta característica. Provavelmente o Windows 2019 reinvente esta característica, e venda melhor entre os malucos por assembler.
- Erros de digitação são muito reduzidos, se estes erros forem marcados com cores. É uma ótima caracteristica de um editor, destacar os componentes de uma linha em diferentes cores. Reconhecimento inteligente dos erros facilita a escrita. Mas é uma característica que eu não sinto falta.
- Se seu editor permite a seleção de fonte, escolha uma fonte com espaço fixo, como Courier. Os cabeçalhos ficam melhor assim.
- Seu editor deve ser capaz de reconhecer fim de linhas com qualquer combinação de caracteres (carriage return, line feed, ambos) sem produzir telas inaceitáveis. Outro item na lista de desejos para o Windows 2013.

Se você prefere matar moscas com canhão, você pode usar um software processador de texto poderodo para escrever os seus programas em assembler. Pode parecer mais bonito, com cabeçalhos em fonte grande e negritos, comentários em cinza, avisos em vermelho, mudanças destacadas, e lembretes e todos em campos extra destacados. Algumas desvantagens, contudo: você terá que converter o seu texto em texto simples no final, perdendo todo o seu trabalho de design, e o seu arquivo texto resultante não poderá ter um único byte de controle faltando. Caso contrário, este único byte gerará uma mensagem de erro, quando você tentar processar o texto. E lembre-se: Números de linha estarão apenas corretos na página do seu código-fonte.

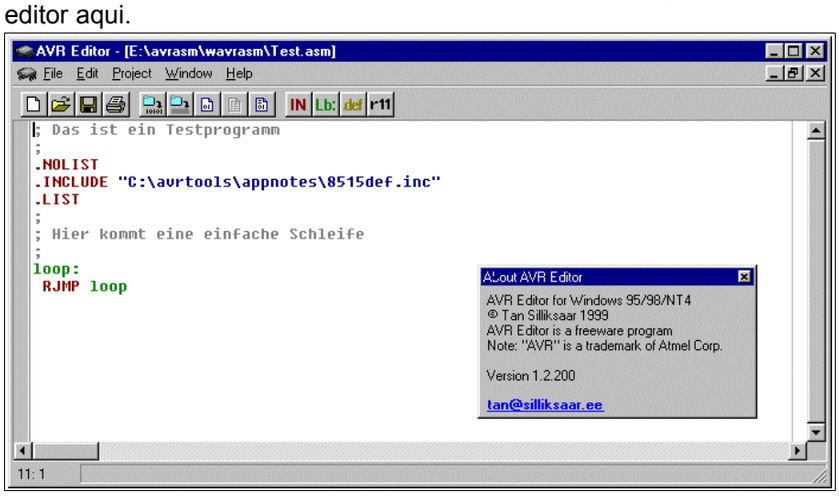



Assim, qualquer programa de texto que você quiser, é escolha sua. Os exemplos a seguir foram escritos no wavrasm, um editor fornecido pela ATMEL há algum tempo.

No campo do editor simples, escrevemos as nossas diretivas e comandos de assembler. É altamente recomendado que estas linhas sejam acompanhadas por alguns comentados (começando por ;). Entender o que você havia planejado aqui são importantes na depuração posterior.

Agora armazene o arquivo texto, chamado TEST.ASM, em um diretório dedicado, usando o menu File. O programa assembly está completo agora

Se você gostaria de ver o que destaque pela sintaxe significa, coloquei a captura de uma tela do AVR editor aqui.




O editor reconhece os comandos automaticamente e usa diferentes cores (destaque da sintaxe) para sinalizar ao usuário constantes e erros de digitação nestes comandos (em preto). O código armazenado no arquivo .asm é basicamente o mesmo, pois as cores não são armazenadas no arquivo.

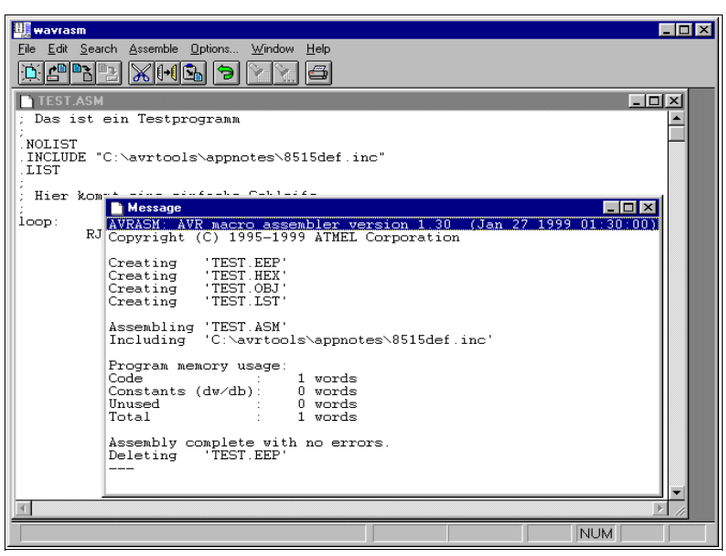
Não tente encontrar este editor ou seu autor; o editor virou história e não é mais mantido.

## O assembler

Agora temos um arquivo texto, com caracteres ASCII em branco. O próximo passo é traduzir este código para forma orientada para máquina, que será compreendida pelo chip AVR. Isto é chamado de “assemblar”, compilar, ou seja “montar as palavras de comando corretas”. Este programa lê o arquivo-texto e produz um tipo de saída chamada Assembler. O formato mais simples deste programa é em linha de comando, e quando chamado com o endereço do arquivo texto e alguns parâmetros adicionais, inicia a tradução dos comandos para um segundo arquivo-texto.

Se o seu editor permite a chamada de programas externos, é uma tarefa fácil. Se não (outro item para a lista de desejos para o editor do Windows 2010), é mais conveniente escrever um arquivo batch (usando outro editor). Este arquivo batch deve ter uma linha assim:

Path do Assembler\Assembler.exe -opções Path do Arquivo-Texto\Arquivotexto.asm



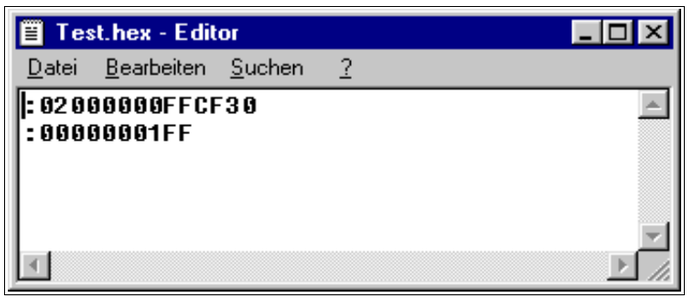

Ao clicar no botão para executar programa externo, ou no arquivo batch, inicia-se o compilador. Este software reporta a tradução completa (na janela menor), neste caso sem erros.

Se erros ocorrerem, eles são notificados, juntamente com o seu tipo e o número da linha onde ele ocorreu.

A compilação resultou em uma palavra de código que resultou da instrução RJMP que utilizamos. Durante o processo de assembler, nosso arquivo de texto produziu outros quatro arquivos (nem todos serão tratados aqui).

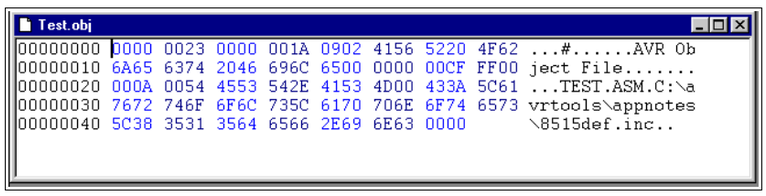
O primeiro destes novos quatro arquivos, TEST.EEP, contém o que efetivamente deverá ser gravado na EEPROM do AVR. Isto não é muito interessante no nosso

caso, porque não programamos nada para a EEPROM. O compilador, conseqüentemente deletou este arquivo quando terminou a montagem, pois era vazio.

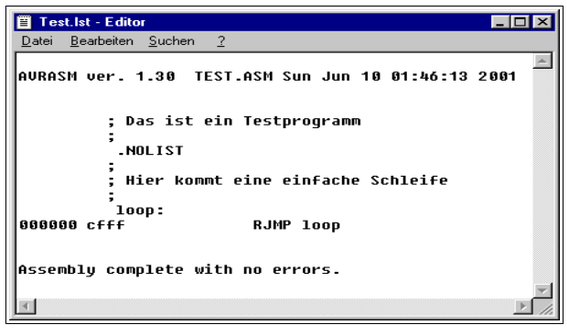



O segundo arquivo, TEST.HEX, é mais relevante pois ele guarda os comandos que serão posteriormente gravados no chip AVR. Este arquivo se apresenta assim.

Os números hexadecimais estão escritos em um formato ASCII especial, juntamente com as informações de endereço e checksum para cada linha. Este formato é chamado Intel-hex, é muito antigo e remonta para os primórdios da computação. Este formato é bem compreendido pelo software de programação.




O terceiro arquivo, TEST.OBJ, será explicado depois, este arquivo é necessário para simular um AVR. Seu formato é hexadecimal e definido pela ATMEL. Usando um editor hexa, seu conteúdo é assim. Atenção: Este formato de arquivo não é compatível com o software de programação, não use este arquivo para programar um AVR (um erro muito comum no início). Arquivos OBJ são produzidos apenas por alguns assemblers ATMEL, não os espere de outros assemblers.



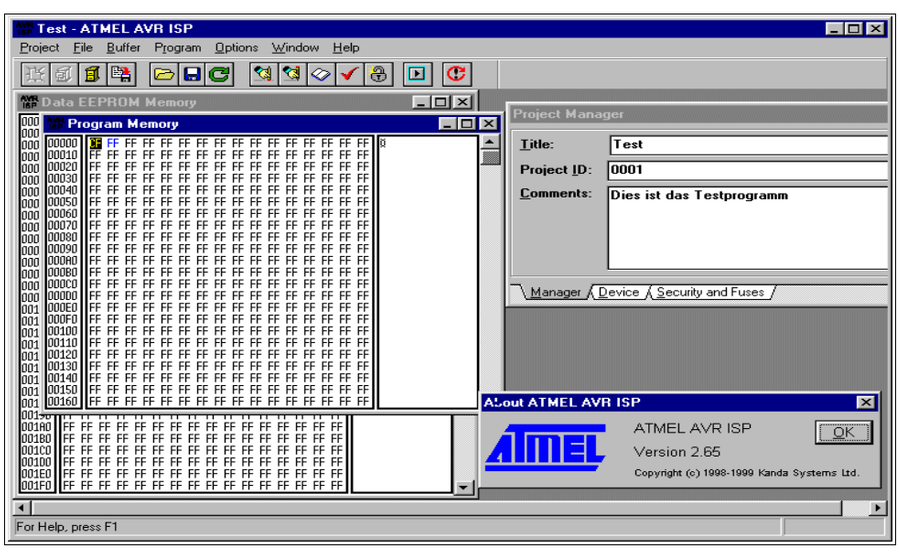

O quarto arquivo, TEST.LST, é um arquivo texto. Abra-o com um editor simples. Será visto uma tela como a mostrada aqui.

O programa com todos os seus endereços, comandos e mensagens de erro são mostrados em uma forma legível. Você precisará deste arquivo em alguns casos para depurar erros.

Arquivos de lista são gerados apenas se a opção é especificada na linha de comando e se a diretiva .NOLIST não suprimir a listagem.

# Programando os chips

Para programar nosso código hexa, como codificado no arquivo .HEX, no AVR, um software programador é necessário. Este software lê o arquivo .HEX e transfere o seu conteúdo, seja bit a bit (programação serial), seja byte a byte (programação paralela) para a memória flash do AVR. Iniciamos o software de programação e carregamos o arquivos hex que acabamos de gerar.



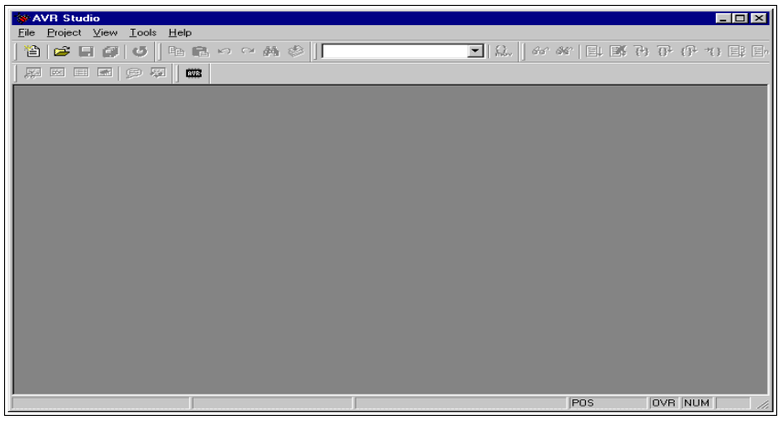

Em nosso exemplo, a tela seria semelhante a esta. Note: a janela apresentada mostra a tela do ISP.exe, um programa histórico não mais distribuído pela ATMEL. Outros softwares programadores são semelhantes.

Este software gravará nosso código no local designado no chip. Há algumas precondições necessárias para fazer isto, e várias razões possíveis para uma falta. Consulte a ajuda do software programador, se ocorrerem problemas.

Hardware de programação e alternativas de software apropriadas para diferentes sistemas operacionais estão disponíveis na Internet. Como exemplo de um programador pela porta paralela ou serial, menciono o PonyProg2000.

# Simulação no studio

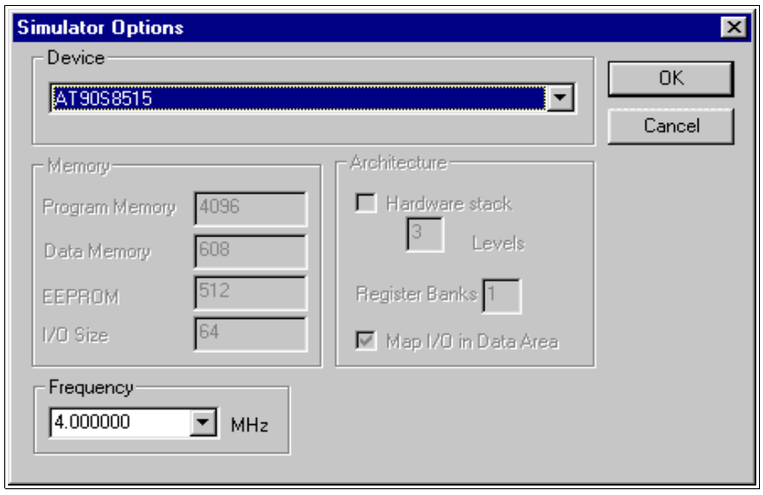
Às vezes, o código escrito em assembly, mesmo que seja compilado sem erros, não faz exatamente o que deveria depois de gravado no chip. Testar o software no chip pode ser complicado, especialmente se você tem um hardware mínimo e não tem oportunidade de visualizar resultados intermediários ou sinais de debug. Nestes casos, o pacote de software Studio da ATMEL fornece oportunidades perfeitas para depuração. Testar o software ou somente partes é possível, e o código do programa pode ser testado passo a passo mostrando os resultados.




As imagens mostradas aqui são tiradas da Versão 3 do Studio. A Versão 4 atual é visualmente diferente, mas faz praticamente a mesma coisa.

Primeiramente abriremos um arquivo (menu FILE – OPEN). Demonstraremos utilizando o arquivo tutorial test1.asm, pois nele há mais alguns comandos e ações do que em nosso programa de um único comando acima.

Abra o arquivo TEST1.OBJ, que resultou da compilação do TEST1.asm. Lhe é perguntado que opções você gostaria de usar (se não você pode alterar através do menu SIMULATOR – OPTIONS). Selecione as seguintes opções:

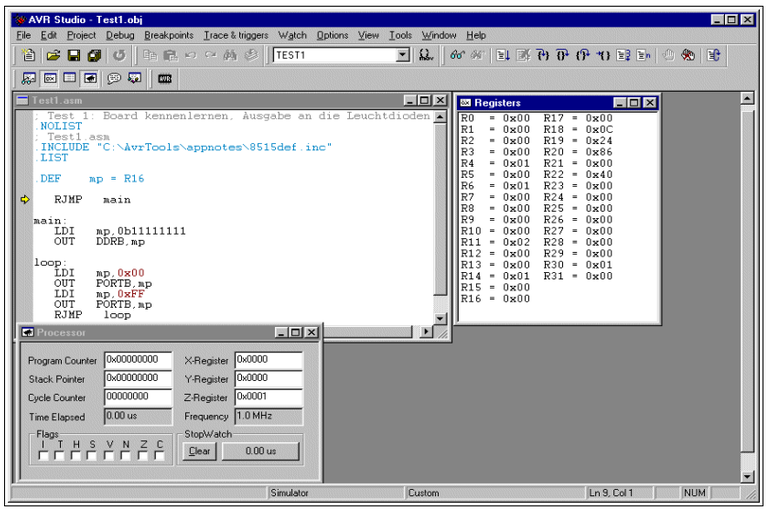



- Dispositivo é um AT90S8515,
- Freqüência do clock é 4 MHz.

Na seção de seleção de dispositivos, selecionamos o tipo de chip desejado. A frequência correta deve ser selecionada, se você deseja simular temporização correta.

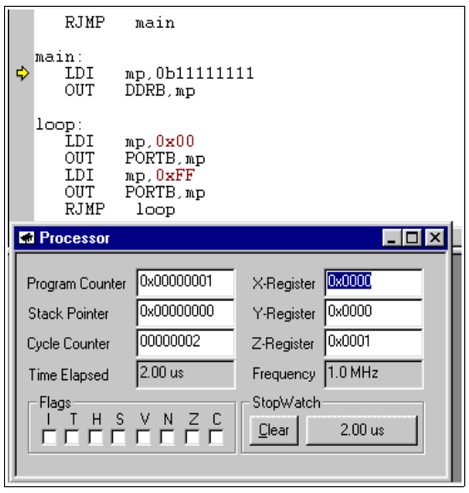
Para podermos ver o conteúdo de alguns registradores e o que o status atual do processador, selecionamos VIEW PROCESSOR e REGISTERS.

Agora a imagem deve ser a seguinte.




A janela do processador mostra vários valores, como onde o contador de comando está, os registradores de flag de status, e a informação sobre temporização (aqui a um clock de 1 MHz). O cronômetro por ser usado para fazer uma medida exata do tempo necessário para executar rotinas etc.

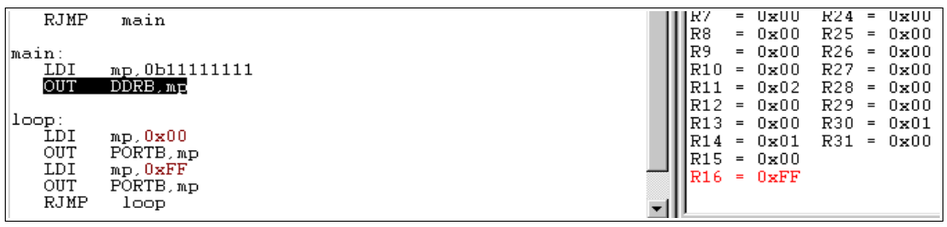
Agoa iniciaremos a execução do programa. Usaremos o modo de passo a passo (TRACE INTO ou F11). Usando GO resultaria em uma execução contínua e nada seria visto devido à alta velocidade da simulação.




Após o primeiro passo ser executado, a janela do processador ficaria como ao lado.

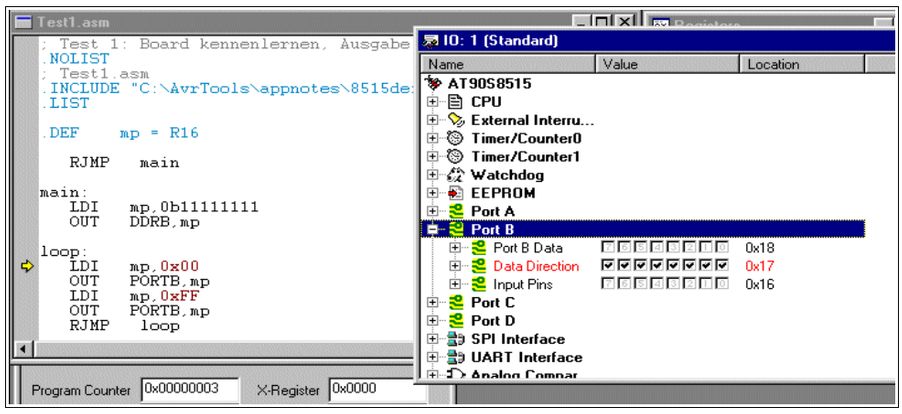
O contador de programa (Program Counter), mostrado na janela do processador, está no passo 1, o contador de ciclo em 2 (RJMP precisa de dois ciclos para executar). Com clock de 1 MHz, dois microssegundos foram gastos, os flags e registradores não foram alterados. A janela com a arquivo texto fonte mostra um ponteiro no próximo comando a ser executado.

Pressionar F11 novamente executa o próximo comando, o registrador mp (=R16) será setado para 0xFF. Agora a janela de registradores deve destacar esta mudança.



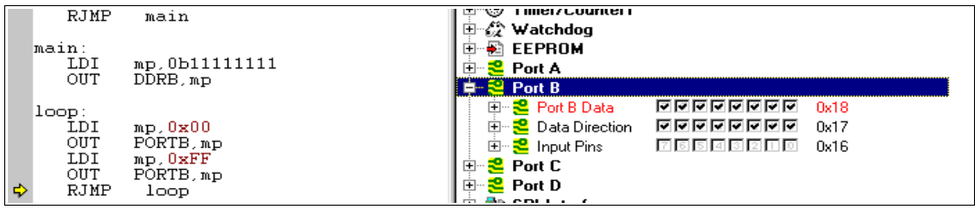

O novo valor do registrador R16 é mostrado em cor vermelha. Podemos alterar o valor de um registrador a qualquer momento para checar o que acontece.

Agora o passo 3 foi executado, a saída é direcionada ao registrador da porta B. Para mostrar, abrimos uma nova janela de visualização de E/S e selecionamos porta B. A imagem deve ser como a seguir.




O registrador de direção de dados na janela de visualização de E/S na porta B agora mostra o novo valor. Os valores podem ser alterados manualmente, se desejado, pino a pino.

Os próximos dois passos são simulados usando F11. Eles não serão mostrados aqui. Setando as portas de saída para nível um com a instrução LDI mp, 0xFF e OUT PORTB, mp resulta na seguinte figura na visualização de E/S. Agora todos os bits de saída da porta estão em nível um, a visualização de E/S mostra isso.




Este é o nosso pequeno passeio através do mundo do software de simulação. O simulador é capaz de muito mais, devendo portanto ser utilizado extensivamente nos casos de erro de projeto. Visite os diferentes itens de menu, há muito mais do que o mostrado aqui.

# Registrador

## O que é um registrador?

Registradores são locais especiais de armazenagem com capacidade de 8 bits e são da seguinte forma:

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|

Note a numeração destes bits: o bit menos significativo inicia com zero ($2^0 = 1$).

Um registrador pode tanto armazenar números de 0 a 255 (somente números positivos), ou números de -128 a +127 (número inteiro com o bit 7 indicando o sinal), ou o valor que representa um caractere ASCII (e.g. 'A'), ou apenas oito bits que não têm nada a ver um com outro (e.g., para oita flags usados para sinalizar oito decisões sim/não diferentes).

A característica especial dos registradores, comparada a outros locais de armazenagem, é que

- eles podem ser usados diretamente por comandos assembler,
- operações com o seu conteúdo requerem somente uma palavra de comando,
- eles são conectados diretamente à unidade central de processamento chamada acumulador,
- eles são origem e destino para cálculos.

Há 32 registradores em um AVR. Eles são originalmente chamados R0 a R31, mas você pode escolher nomeá-los com nomes que façam sentido em uma diretiva assembler. Um exemplo:

```
.DEF MeuRegistradorPreferido = R16
```

As diretivas de assembler sempre iniciam com um ponto na coluna 1 do texto. Instruções NUNCA iniciam na coluna 1, eles sempre serão precedidos por um caractere vazio ou de tabulação!

Note que as diretivas de assembler como esta significam alguma coisa apenas para o compilador assembler mas não produzem nenhum código executável no chip destino AVR. No lugar de utilizar o registrador com nome R16, podemos utilizar nosso próprio nome MeuRegistradorPreferido, quando quisermos usar R16 com algum comando. Então escrevemos um pouco mais cada vez que queremos utilizar este registrador, mas temos uma associação ao que pode ser o conteúdo deste registrador.

Usando a linha de comando

```
    LDI     MeuRegistradorPreferido, 150
```

significa: carregue o número 150 imediatamente para o registrador R16, *LoaD Immediate,* carregar imediato. Isto carrega um valor fixo ou uma constante neste registrador. Se observarmos a tradução deste código no programa escrito no AVR, veremos o seguinte:

```
000000 E906
```

O comando load, bem como o registrador destino (R16), assim como o valor da constante (150) é parte do valor hexa E906, mesmo que você não os veja diretamente. Não tenha medo: você não tem que lembrar esta codificação porque o compilador sabe traduzir tudo isto para o esquisito E906.

Dentro de um único comando, dois diferentes registradores podem desempenhar um papel. O comando mis fácil teste tipo é o comando de cópia MOV. Ele copia o conteúdo de um registrador em outro. Assim:

```
.DEF MeuRegistradorPreferido = R16
.DEF OutroRegistrador = R15
    LDI MeuRegistradorPreferido, 150
    MOV OutroRegistrador, MeuRegistradorPreferido
```

As duas primeiras linhas deste programa monstruoso são diretivas que definem os novos nomes dos registradores R16 e R15 para o compilador assembler. Novamente, estas linhas não produzem nenhum código para o AVR. As linhas de comando com LDI e MOV produzem o código:

```
000000 E906
000001 2F01
```

Os comandos escrevem 150 no registrador R16 e copiam o seu conteúdo para o registrador destino R15. NOTA IMPORTANTE:

**O primeiro registrador é sempre o registrador destino onde o resultado será escrito!**

(Isto infelizmente é diferente da forma normalmente esperada ou de como falamos. É uma convenção simples que foi definida certa vez para confundir os iniciantes aprendendo assembler. É por isso que assembler é tão complicado.)

# Registradores diferentes

O iniciante pode querer escrever os comandos acima assim:

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

E: você falha. Apenas os registradores de R16 a R31 carregam uma constante imediamente com o comando LDI, R0 a R15 não fazem isso. Esta restrição não é muito interessante, mas não pôde ser evitada durante a construção do conjunto de comandos dos AVRs.

Há uma exceção a esta regra: escrever Zero em um registrador. O comando

```
CLR MeuRegistradorPreferido
```

é válido para todos os registradores.

Além do comando LDI, você descobrirá que esta restrição de classes de registradores ocorre também com os seguintes comandos:

- ANDI Rx,K ; Operação And dos bits do registrador Rx com os bits do valor constante K,
- CBR Rx,M ; zera todos os bits no registrador Rx que estão em um dentro do valor de máscara constante M.
- CPI Rx,K ; Compara o conteúdo do registrador Rx com o valor constante K,
- SBCI Rx,K ; Subtrai a constante K e o valor atual da flag de carry (transporte) do registrador Rx e armazena o resultado no registrador Rx,
- SBR Rx,M ; Seta todos os bits no registrador Rx para um, que são um na máscara constante M,
- SER Rx ; Seta todos os bits no registrador Rx para um (igual a LDI Rx,255),
- SUBI Rx,K ; Subtrai a constante K do conteúdo do registrador Rx e armazena o resultado no registrador Rx.

Em todos estes comandos o registrador deve ser entre R16 e R31! Se você planeja utilizar estes comandos, você deve selecionar um destes registradores para a operação. É mais fácil de programar. Há uma razão adicional pela qual você deve definir o nome dos registradores, é porque você pode facilmente alterar a localização dos registradores posteriormente.

# Registradores ponteiros

Um papel muito especial é desempenhado pelos pares de registradores R27:R26, R29:R28 e R31:R32. Este papel é tão importante que estes pares têm nomes extra curtos em AVR assembler: X, Y e Z. Estes nomes curtos são compreendidos pelo compilador. Estes pares são registradores ponteiros de 16 bits, capazes de apontar endereços da SRAM com até 16 bits (X, Y ou Z) ou localizações na memória do programa (Z).

O byte menor do endereço de 16 bits está localizado no registrador inferior, o byte superior na registrador superior. Ambas partes têm seu próprio nome, e.g., o byte mais alto de Z é chamado de ZH (=R31), e o byte mais baixo é ZL (=R30). Estes nomes são definidos no arquivo de cabeçalho padrão para os chips. Dividir estes dois ponteiros de 16 bits em dois bytes diferentes é feito da seguinte forma:

```
.EQU endereco = RAMEND ; RAMEND é o endereço de 16 bits mais alto da SRAM (fim da memória)
        LDI YH,HIGH(endereco) ; Seta o MSB
        LDI YL,LOW(endereco) ; Set the LSB
```

Acessos via ponteiros são feitos com comandos especialmente designados. A leitura é feita pelo comando chamado LD (*LoaD*, carregar), e a escrita pelo comando chamado ST (*STore*, armazenar), e.g. Com o ponteiro X:

| ***Ponteiro*** | ***Seqüência*** | ***Exemplos*** |
|---|---|---|
| X | Lê/Escreve do endereço X, não altera o ponteiro | LD R1,X ou ST X,R1 |
| X+ | Lê/Escreve de/para endereço X e incrementa o ponteiro por um | LD R1,X+ ou ST X+,R1 |
| -X | Decrementa o ponteiro em um e lê/escreve de/para o novo endereço | LD R1,-X ou ST -X,R1 |

De forma similar, você pode usar Y e Z para este propósito.

Há somente um comando para acesso à leitura dos dados do programa. Ele é definido pelo par de ponteiros Z e é chamado LPM (*Load from Program Memory,* carregar da memória do programa). O comando copia o byte no endereço Z da memória do programa para o registrador R0. Como a memória do programa é organizada na forma de palavras (um comando em um endereço consiste de 16 bits ou dois bytes ou uma palavra), o bit menos significativo seleciona o byte inferior ou superior (0=inferior, 1=superior). Por causa disto, o endereço original deve ser multiplicado por 2 e o acesso é limitado a 15 bits ou 32kb da memória do programa. Desta forma:

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

Seguindo este comando, o endereço deve ser incrementado para apontar o próximo byte na memória do programa. Como isto é usado freqüentemente, um comando de incremento de pointeiro foi definido para fazer isto:

```
ADIW ZL,1
LPM
```

ADIW significa *ADd Immediate Word,* adicione palavra imediato, e um máximo de 63 pode ser adicionado desta forma. Note que o compilador espera o mais baixo do par de registradores ZL como primeiro parâmetro. Isto pode causar confusão, pois a adição é realizada como uma operação de 16 bits.

O comando complementar, subtrair um valor constante entre 0 e 63 de um ponteiro de 16 bits é chamado de SBIW (*SuBtract Immediate Word*, subtrair palavra imediato). ADIW e SBIW são possíveis para os pares de registradores X, Y e Z e para o par de registradores R25:R24, que não têm um nome extra e não permitem acesso à SRAM ou a localizações da memória do programa. R25:R24 são ideais para manipular valores de 16 bits.

Como inserir uma tabela de valores na memória do programa? Isto é feito através das diretivas do assembler .DB e .DW. Com elas você pode inserir uma lista de bytes ou uma lista de palavras. Listas de bytes organizados ficam assim:

```
.DB 123,45,67,89 ; uma lista de quatro bytes, escritos em formato decimal
.DB "Isto e um texto. " ; uma lista de caracteres byte, escritos como texto
```

Você deve sempre colocar um número par de bytes em cada linha. Caso contrário o compilador adicionará um byte zero ao final, que pode ser indesejado.

Uma lista similar de valores ficaria assim:

```
.DW 12345,6789 ; uma lista de duas palavras
```

No lugar de constantes, você pode também colocar labels (destinos para salto) nesta lista, como:

```
Label1:
[ ... aqui vão alguns comandos ... ]
Label2:
[ ... aqui vão mais alguns comandos ... ]
Tabela:
.DW Label1,Label2 ; uma lista de labels
```

Labels SEMPRE iniciam na coluna 1! Note que ler os labels com LPM primeiro traz o byte inferior da palavra.

Uma aplicação muito especial para os ponteiros é acessar os próprios registradores. Os registradores estão localizados nos primeiros 32 bytes do espaço do chip (do endereço 0x0000 a 0x001F). Este acesso é útil se você tiver que copiar o conteúdo dos registradores para a SRAM ou EEPROM ou ler estes valores destes locais de volta para os registradores. O uso mais comum dos ponteiros é para acessar tabelas com valores fixos na memória do programa. Aqui está um exemplo, uma tabela com 10 diferentes valores de 16 bits, onde o quinto valor da tabela é lido para R25:R24:

```
MinhaTabela:
.DW 0x1234,0x2345,0x3456,0x4568,0x5678 ; Os valores da tabela
.DW 0x6789,0x789A,0x89AB,0x9ABC,0xABCD ; organizados em palavras
Read5: LDI ZH,HIGH(MinhaTabela*2) ; endereço do ponteiro Z
       LDI ZL,LOW(MinhaTabela*2) ; multiplicado por 2 para acesso à tabela
       ADIW ZL,10 ; Aponta para o quinto valor da tabela
       LPM ; Lê o byte menos significativo da memória do programa
       MOV R24,R0 ; Copia o LSB para o registrador de 16 bits
       ADIW ZL,1 ; Aponta para o MSB na memória do programa
       LPM ; Lê o MSB da tabela de valores
       MOV R25,R0 ; Copia MSB para o registrador de 16 bits.
```

Este é apenas um exemplo, Você pode calcular o endereço da tabela em Z a partir de algum valor de entrada, levando os respectivos valores na tabela. As tabelas podem ser organizadas por byte ou caracteres também.

# Recomendação para uso dos registradores

- Defina nomes para os registradores com a diretiva .DEF, nunca os use diretamente com o seus nomes diretos Rx.
- Se você precisar acesso aos ponteiros, reserve R26 a R31 para este propósito.
- A melhor localização para contadores de 16 bits é R25:R24.
- Se precisar ler da memória do programa, e.g. Tabelas fixas, reserve Z (R31;R30) e R0 para isso.
- Se você pretende acessar bits dentro de certos registradores (flags), use R16 a R23 para isso.

# Portas

## O que é uma Porta?

As portas do AVR são caminhos entre a unidade central de processamento e os componentes externos de hardware e software. A CPU se comunica com estes componentes, lendo deles ou escrevendo neles, e.g. timers ou portas paralelas. A porta mais utilizada é o registrador de flag, onde os resultados das operações anteriores são escritos e de onde as árvores de condições (decisões) são lidas.

Há 64 portas diferenentes, que não estão disponíveis em todos tipos de AVR. Dependendo do espaço de armazenamento e outros hardwares internos, as portas podem estar disponíveis e acessíveis ou não. A lista de portas que podem ser usadas está listada nos data sheets do processador.

As portas têm endereço fixo, pelo qual a CPU se comunica. O endereço é independente do tipo de AVR. Assim, por exemplo, o endereço da porta B é sempre 0x18 (0x indica notação hexadecimal). Você não tem que lembrar destes endereços das portas, elas têm apelidos convenientes. Estes nomes são definidos nos arquivos include (cabeçalhos) para os diferentes tipos de AVR, fornecidos pelo fabricante. Os arquivos include têm uma linha definindo o endereço da porta B como a seguir:

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

Então temos que lembrar apenas o nome da porta B, não sua localização no espaço de E/S do chip. O arquivo de inclde 8515def.inc está envolvido através da diretiva do assembler

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

e todos os registradores de 8515 estarão definidos e facilmente acessíveis.

As portas normalmente são organizadas como números de 8 bits, mas podem manipular 8 bits individuais que não têm a ver uns com os outros. Se estes bits têm um significado, eles podem ter seu próprio nome associado no arquivo include, para permitir a manipulação deste bit. Graças à convenção dos nomes, você não tem que lembrar a posição destes bits. Estes nomes são definidos nos data sheets e estão incluídos nos arquivos include, também. Eles são fornecidos aqui nas tabelas das portas.

Como exemplo, o registrador de controle geral MCU, chamado MCUCR, consiste de um número de bits de controle individuais que controlam algumas características gerais do chip (veja a descrição em MCUCR para maiores detalhes). É uma porta, com 8 bits de controle e seus próprios nomes (ISC00, ISC01, ...). Quem quiser mandar o seu AVR para um sono mortal precisa saber pelo data sheet como setar os bits respectivos. Assim:

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

O comando Out envia o conteúdo do meu registrador preferido, um bit Sleep-Enable chamado SE, para a porta MCUCR e seta o AVR imediatamente para entrar em repouso, se houver uma instrução SLEEP sendo executada. Como todos os outros bits do MCUCR foram setados pelas instruções acima, e o bit SM (Sleep Mode) ficou zero, ocorrerá um modo chamado meio-repouso: não haverá execução de nenhum outro comando, mas o chip ainda reage a timers e outras interrupções de hardware. Estes eventos externos interromperão o sono da CPU se eles acharem que devem notificá-la.

Ler o conteúdo de uma porta é possível na maioria dos casos utilizando o comando IN. A sequência

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

lê os dados da porta MCUCR para o registrador. Como muitas portas são usadas parcialmente ou não utilizadas, normalmente os dados lidos são zeros.

Mais freqüentemente do que ler todos os 8 bits de uma porta, deve-se reagir a certos status de uma porta. Neste caso, não precisamos ler a porta toda e isolar o bit de interesse. Certos comandos permitem executar comandos dependendo do estado de certo bit (veja a seção JUMP). Setar ou zerar certos bits também é possível sem ter que ler e escrever os outros bits da porta. Os dois comandos são SBI (Set Bit I/o, setar bit E/S) e CBI (Clear Bit I/o, zerar bit E/S). A execução é assim:

```
.EQU BitAtivo=0 ; O bit que será mudado
        SBI PortB, BitAtivo ; O bit será setado para um
        CBI PortB, BitAtivo ; O bit será setado para zero
```

Estas duas instruções têm uma limitação: somente portas com endereço inferior a 0x20 podem ser manipuladas desta forma.

Para programadores mais exóticos: as portas podem ser acessadas utilizando comandos de acesso à SRAM, como ST e LD. Apenas some 0x20 ao endereço da porta (os primeiros 32 endereços são os registradores) e acesse a porta desta forma. Vamos demonstrar aqui:

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

*LD MeuRegistradorPreferido,Z*

Isto faz sentido apenas em certos casos, mas é possível. É por esta razão que o primeiro endereço da SRAM é sempre 0x60.

# Deatlhes de portas relevantes do AVR

A tabela a seguir mostra as portas mais usadas. Nem todas portas estão listadas aqui, algumas do MEGA e AT90S4434/8535 foram omitidas. Se houver dúvida, verifique as referências originais.

| ***Componente*** | ***Nome da porta*** | ***Registrador da Porta*** |
|---|---|---|
| Acumulador | SREG | Registrador de Status |
| Pilha (*stack)* | SPL/SPH | Stackpointer (apontador de pilha) |
| SRAM Externa/Interrupção Externa | MCUCR | Registrador de Controle Geral MCU |
| Interrupção Externa | GIMSK | Registrador de Interrupção Mascarado |
| | GIFR | Registrador de Flag de Interrupção |
| Interrupção de Timer | TIMSK | Registrador de Interrupção de Timer Mascarada |
| | TIFR | Registrador de Interrupção de Flag de Timer |
| Timer 0 | TCCR0 | Registrador controle de de Timer/Contador 0 |
| | TCNT0 | Timer/Contador 0 |
| Timer 1 | TCCR1A | Registrador controle de de Timer/Contador 1 A |
| | TCCR1B | Registrador controle de de Timer/Contador 1 B |
| | TCNT1 | Timer/Contador 1 |
| | OCR1A | Registrador Comparador de Saída 1 A |
| | OCR1B | Registrador Comparador de Saída 1 B |
| | ICR1L/H | Registrador de Captura de Entrada |
| Timer Watchdog | WDTCR | Registrador de Controle do Timer Watchdog |
| EEPROM | EEAR | Registrador de endereço da EEPROM |
| | EEDR | Registrador de dados da EEPROM |
| | EECR | Registrador de controle da EEPROM |
| SPI | SPCR | Registrador de controle de periféricos Seriais |
| | SPSR | Registrador de status de periféricos Seriais |
| | SPDR | Registrador de dados de periféricos Seriais |
| UART | UDR | Registrador de dados da UART |
| | USR | Registrador de status da UART |
| | UCR | Registrador de controle da UART |
| | UBRR | Registrador de velocidade (baud rate) da UART |
| Comparador analógico | ACSR | Registrador de status e controle do comparador analógico |
| Portas de E/S | PORTx | Registrador de porta de saída |
| | DDRx | Registrador de direção da porta |
| | PINx | Registrador de porta de entrada |

# O registrador de status como a porta mais utilizada

De longe, a porta mais freqüentemente usada é o registrador de status com seus 8 bits. Normalmente o acesso a esta porta é feito somente por alteração dos bits pela CPU ou acumulador, alguns acessos são por leitura ou alteração dos bits na porta, e em raros casos é possível manipular estes bits diretamente (utilizando os comandos assembler SEx ou CLx, onde x é a abreviação do bit). A maioria destes bits é alterada por operações de teste de bit, comparação ou cálculos. A seguinte lista tem todos os comandos em assembler que podem alterar os bits de status, dependendo do resultado da execução.

| *Bit* | *Cálculo* | *Lógico* | *Comparação* | *Bits* | *Rolagem* | *Outro* |
|---|---|---|---|---|---|---|
| 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 | - | - |
| 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 |

# Detalhes das portas

Detalhes das portas mais comuns são mostrados na tabela extra (veja anexo).

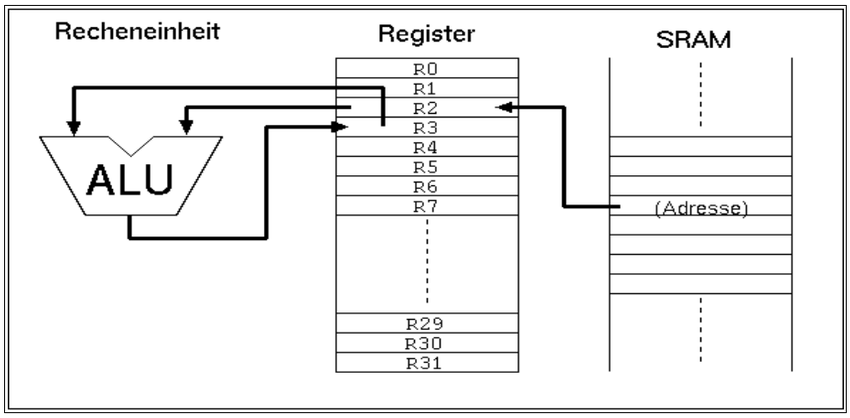
# SRAM

## Usando SRAM em linguagem assembler AVR

Praticamente todos os MCUs AVR tipo AT90S têm RAM estática (SRAM, *Static RAM*) (alguns não). Somente compiladores muito simples evitam a utilização deste espaço da memória, colocando tudo nos registradores. Se os seus registradores acabarem, você deverá ser capaz de programar a SRAM para obter mais espaço.

## O que é SRAM?

SRAM são memórias que não são diretamente acessíveis da unidade de processamento central (Unidade Aritmética e Lógica ALU *(Arithmetic and Logical Unit*), algumas vezes chamada de acumulador), como os registradores o são. Para acessar estas localidades de memória, você normalmente usa um registrador como armazém intermediário. No exemplo a seguir, um valor na SRAM será copiado para o registrador R2 (primeiro comando), um cálculo utilizando o valor de R3 é feito e o resultado é escrito em R3 (comando2). Depois disso, o valor é escrito de volta na SRAM (comando 3, não mostrado aqui).




É evidente que operações com valores armazenados na SRAM são mais lentas do que aquelas utilizando apenas os registradores. Por outro lado: o menor AVR tem 128 bytes de SRAM disponível, muito mais do que os 32 registradores podem guardar.

Os tipos mais recentes do que AT90S8515 oferecem a possibilidade de conectar RAM externa, expandindo a interna de 512 bytes. Do ponto de vista do assembler, a SRAM externa é acessada da mesma forma que a interna. Não existe nenhum comando extra para SRAM externa.

## Para que propósitos posso usar a SRAM?

Além de simples armazenagem de valores, a SRAM oferece outras possibilidades de uso. Não apenas o acesso com endereços fixos é pssível, como também o uso de ponteiros, de forma que o acesso flutuante a localizações subsequentes pode ser programado. Desta forma você pode construir buffers em anel para armazenagem temporária de valores ou tabelas calculadas. Isto não é feito frequentemente com registradores, pois eles são muito poucos e acesso fixo é preferido.

Outra forma de acesso é utilizando um offset para um endereço de início fixo em um dos ponteiros. Neste caso, o endereço fixo é armazenado no ponteiro, e um valor constante é adicionado a este endereço e os acesso de leitura e escrita são feitos a esse endereço com um offset. Com este tipo de acesso, as tabelas são melhores utilizadas.

O uso mais interessande para a SRAM é a chamada pilha (*stack*). Você coloca (“empurra”, *push*) valores para a pilha, seja um registrador, um endereço de retorno após o fim de uma sub-rotina, ou o endereço de retorno após uma interrupção disparada por hardware.

## Como usar a SRAM?

Para copiar um valor para uma localização da memória SRAM, você tem que definir o endereço. O endereçamento da SRAM começa em 0x0060 (notação hexa) até o fim da SRAM física no chip (no AT90S8515, o endereço mais alto da SRAM é 0x025F). Com o comando

```
STS 0x0060, R1
```

o conteúdo do registrador R1 é copiadao para o primeiro endereço da SRAM. Com

```
LDS R1, 0x0060
```

o conteúdo da SRAM no endereço 0x0060 é copiado para o registrador. Este é o acesso direto através do endereço definido pelo programador.

Nomes simbólicos devem ser evitados para evitar manipular endereços fixos, pois requerem muito trabalho se você depois quiser alterar a estrutura da SRAM. Estes nomes são mais fáceis de lembrar do que os números hexa, então você pode nomear um endereço da seguinte forma:

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

Sim, fica mais longo, mas mais fácil de lembrar. Use o nome que você achar conveniente.

Outro tipo de acesso à SRAM é com o uso de ponteiros. Você precisa de dois registradores para este propósito, para acomodar o endereço em 16 bits. Como aprendemos na divisão dos ponteiros, os registradores ponteiros são os pares X (XH:XL, R27:R26), Y (YH:YL, R29:R28) e Z (ZH:ZL, R31:R30). Eles permite acesso à localização que apontam diretamente (e.g., com ST X, R1), antes da decrementação do endereço por um (e.g., ST -X, R1) ou antes do incremento deste endereço (e.g., ST X+, R1). Um acesso completo a três endereços em seguida ficaria então da seguinte forma:

```
.EQU MeuEnderecoPreferido= 0x0060
.DEF MeuRegistradorPreferido = R1
.DEF OutroRegistrador= R2
.DEF MaisUmRegistrador = R3
        LDI XH, HIGH(MeuEnderecoPreferido)
        LDI XL, LOW(MeuEnderecoPreferido)
        LD MeuRegistradorPreferido, X+
        LD OutroRegistrador, X+
        LD MaisUmRegistrador, X
```

Fáceis de operar, estes ponteiros. E tão fácil quanto em outras linguagens diferentes do assembler, que dizem ser fácil de aprender.

A terceira construção é um pouco mais exótica e somente utilizada por programadores experientes. Vamos assumir que precisamos acessar três localizações da SRAM com freqüência. Vamos também assumir que temos um par de registradores ponteiros sobrando, então podemos utilizá-lo exclusivamente para nosso propósito. Se usássemos as instruções ST/LD, teríamos que alterar o ponteiro a cada vez que precisássemos acessar outra localização. Não muito conveniente.

Para evitar isso, e confundir o iniciante, o acesso com offset foi inventado. Durante este acesso, o valor do registrador não é alterado. O endereço é calculado adicionando temporariamente um offset fixo. No exemplo acima, o acesso à localização 0x0062 ficaria assim. Primeiro, o ponteiro é setado para nossa localização central 0x0060:

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

Em algum lugar depois no programa, para acessar o endereço 0x0062:

```
        STD Y+2, MeuRegistradorPreferido
```

Note que 2 não é realmente adicionado a Y, apenas temporariamente. Para confundi-lo mais, isto pode ser feito apenas com os pares de registradores Y e Z, não com o ponteiro X!

A instrução correspondente para ler a SRAM com um offset

```
        LDD MeuRegistradorPreferido, Y+2
```

também é possível.

Esta é a SRAM, mas espere: o uso mais interessante como pilha ainda deve ser aprendido.

# O uso da SRAM como pilha

O uso mais comum da SRAM é como pilha. A pilha é uma torre de blocos de madeira. Cada bloco adicional vai no topo da torre, e cada chamada remove o bloco superior da torre. Esta estrutura é chamada de Last-In-First-Out (o primeiro a entrar é o último a sair – LIFO), ou mais fácil: o último a ir no topo, será o primeiro a sair.

## Definindo SRAM como pilha

Para usar a SRAM como pilha, devemos ajustar o ponteiro de pilha (stack pointer) primeiro. O ponteiro de pilha é um ponteiro de 16 bits, acessível como uma porta. O registrador duplo é chamado de SPH:SPL. SPH armazena o byte mais significativo, e o SPL, o menos significativo. Isto é verdadeiro apenas se o AVR tiver mais de 256 bytes de SRAM. Caso contrário, SPH é indefinido, não pode e não deve ser usado. Vamos assumir que temos mais de 256 bytes nos exemplos a seguir.

Para contruir a pilha, o ponteiro de pilha é carregado com o endereço mais alto da SRAM. (No nosso caso, a torre cresce para baixo, em direção aos endereços menores!)

```
.DEF MeuRegistradorPreferido = R16
        LDI MeuRegistradorPreferido, HIGH(RAMEND) ; Byte Superior
        OUT SPH,MeuRegistradorPreferido ; para o ponteiro de pilha
        LDI MeuRegistradorPreferido, LOW(RAMEND) ; Byte inferior
        OUT SPL,MeuRegistradorPreferido ; para o ponteiro de pilha
```

O valor RAMEND (que indica o endereço final da memória) é, obviamente, específico para o tipo de

progrssador. Ele é definido no arquivo INCLUDE para o tipo de processador.O arquivo 8515def.inc tem a linha:

*.equ RAMEND =$25F ; Last On-Chip SRAM Location (n.t.: última localização da SRAM no chip)*

O arquivo 8515def.inc é incluído através da diretiva de assembler

.INCLUDE "C:\algumlugar\8515def.inc"

no início do nosso código fonte em assembler.

Então definimos a pilha agora, e não temos mais que nos preocupar sobre o ponteiro da pilha, porque as manipulações do ponteiro são automáticas.

## Uso da pilha

Usar a pilha é fácil. O conteúdo dos registradores são empurrados na pilha assim:

```
PUSH MeuRegistradorPreferido ; Coloca aquele valor no topo da pilha
```

Para onde o valor vai, não nos interessa. O ponteiro de pilha foi decrementado após colocarmos este valor, e não temos que nos preocupar. Se precisarmos deste conteúdo de volta, simplesmente usamos a seguinte instrução:

```
POP MeuRegistradorPreferido ; Lê o valor de volta
```

Com POP simplesmente pegamos o último valor colocado no topo da pilha. Empurrar e pegar registradores faz sentido, se

- o conteúdo será necessário novamente algumas linhas de código depois,
- todos os registradores estão em uso, e se
- não há como armazenar aquele valor em nenhum outro lugar.

Se nenhuma destas condições se verificarem, o uso da pilha para guardar registradores é inútil e apenas desperdiça tempo do processador.

Faz mais sentido usar a pilha em subrotinas, onde você tem que retornar à localização do programa que chamou a rotina. Neste caso, o código que chama a subrotina deve guardar o endereço de retorno (o valor atual do contador de programa, PC, *program counter*) na pilha e salta para a subrotina. Após a sua execução, a subrotina pega o endereço de retorno da pilha e carrega no contador de programa. A execução do programa então continua exatamente uma instrução após a instrução de chamada:

```
RCALL AlgumLugar ; Salta para o label “AlgumLugar:”
[...] aqui continuamos o programa
```

Aqui a execução para para o label AlgumLugar em algum lugar do código,

AlgumLugar: ; este é o endereço de salto

```
[...] Aqui fazemos alguma coisa
[...] e aqui terminamos o que queríamos fazer e saltamos de volta para o local onde essa rotina foi chamada:
RET
```

Durante a execução da instrução RCALL, o contador de programa (PC) já incrementado, com 16 bits, é colocado na pilha usando dois pushes. Quando a instrução RET é executada, o conteúdo anterior do contador de programa é recarregado usando dois pops e a execução continua.

Você não precisa se preocupar sobre o endereço da pilha, ou onde o contador está gravado. O endereço é gerado automaticamente. Mesmo se você chamar uma subrotina dentro de uma subrotina, a função da pilha é perfeita. Os dois endereços são colocados no topo da pilha, a subrotina dentro da primeira remove o primeiro endereço, e a subrotina de chamada remove o segundo. Enquanto haja SRAM suficiente, tudo funciona.

A utilização de interrupções de hardware não é possível sem a pilha. A interrupção pára a execução normal do programa, onde quer que ele esteja. Após a execução do serviço como reação àquela interrupção, a execução deve retornar à localização anterior, antes da interrupção. Isto não seria possível se a pilha não fosse capaz de armazenar o endereço de retorno.

As enormes vantagens de se ter uma pilha para interrupções são as razões de mesmo o menor dos AVRs que não tenham SRAM, tenham ao menos uma pequena pilha de hardware.

## Bugs com a operação de pilhas

Para o iniciante, há uma grande possibilidade de bugs, após aprender a usar a pilha.

Muito inteligente é o uso da pilha sem setar primeiro o ponteiro da pilha. Como o valor deste ponteiro é zero no início do programa, o ponteiro aponta para o registrador R0. Colocar algo na pilha resulta em escrever neste registrador, sobrescrevendo qualquer conteúdo que ali estivesse. Um outro push na pilha, e a pilha vai gravar em 0xFFFF, uma posição indefinida (se você não tiver SRAM externa). RCALL e RET retornarão um endereço estranho na memória. E tenha certeza: não haverá nenhum aviso, nenhuma janela indicando “Acesso ilegal à memória localização xxxx”.

Outra possibilidade de ter bug é esquecer de fazer o pop em um valor armazenado anteriormente, ou fazer pop sem fazer push antes.

Em alguns poucos causos, a pilha estoura para abaixo do primeiro endereço da SRAM. Isto ocorre no caso de uma chamada recursiva infinita. Após alcançar o endereço mais baixo da SRAM, os próximos pushes escrevem nas portas (0x005F até 0x0020), e então os registradores (0x001F a 0x0000). Coisas engraçadas e imprevisíveis podem ocorrer com o chip, se isto ocorrer. Evite este bug, ele pode destruir o seu hardware!

# Saltos e Desvios

Agora discutiremos todos os comandos que controlam a execução seqüencial de um programa, iniciando com a seqüência inicial de partida do processador, continuando com saltos, interrupções, etc.

# Controlando a execução seqüencial de um programa

## O que acontece durante um reset?

Quando a tensão de trabalho de um AVR sobe e o processador inicia o seu trabalho, o hardware dispara uma seqüência de reset. O contador de passos do programa é zerado. Neste endereço, a execução sempre inicia. Aqui temos que ter nossa primeira palavra de código. Mas este endereço pode ser ativado também de outras formas:

- Durante um reset externo via pino de reset, o processador é reiniciado.
- Se o contador Watchdog alcança a sua contagem máxima, um reset é iniciado. O timer witchdog é um temporizador interno que deve ser resetado de tempos em tempos pelo programa, caso contrário ele reinicia o processador.
- Você pode chamar o reset fazendo um salto direto a este endereço (veja a seção de saltos abaixo).

O terceiro caso não é um reset real, pois o reinício de registradores e valores de porta a valores conhecidos, não é executado. Portanto, esqueça-o por enquanto.

A segunda opção, o reset watchdog, deve ser habilitado pelo programa. Se estiver desabilitado por padrão, a habilitação requer escrever comandos na porta do watchdog. Zerar o contador watchdog requer a execução do comando

```
WDR
```

para evitar o reset.

Após a execução de um reset, e restabelecimento dos registradores e portas para seus valores padrão, o código no endereço 0000 é lido pela parte de execução do processador, e executado. Durante esta execução o contador de programa (PC, *program counter*) já é incrementado em um e a próxima palavra do código já é lida para o buffer de busca de código (Busca durante Execução). Se o comando executado não requer um salto para outra localidade no programa, o próximo comando é executado imediatamente. É por isso que os AVRs executam extremamente rápido, cada ciclo de clock executa um comando (se não ocorrerem saltos).

O primeiro comando de um executável está sempre localizado no endereço 0000. Para dizer ao compilador que nosso código fonte inicia aqui e agora, uma diretiva especial pode ser colocada no início, antes do primeiro código:

```
.CSEG
.ORG 0000
```

A primeira diretiva diz ao compilador que a saída deve ser para a seção de códigos. Tudo que se segue será traduzido como código, e será escrito à memória de programa do processador. Também é possível direcionar a saída para a seção EEPROM do chip, onde você pode escrever bytes e palavras.

```
.ESEG
```

O terceiro segmento é a porção SRAM do chip.

```
.DSEG
```

Diferente do conteúdo para EEPROM, que realmente é gravado no EEPROM durante a programação, o segmento DSEG não é programado no chip. É utilizado somente para cálculo dos labels durante o processo de compilação.

A diretiva ORG indica a origem e manipula os endereços dentro do segmento de código, onde as palavras montadas são gravadas. Como nosso programa sempre inicia a 0x0000, as diretivas CSEG/ORG são triviais, você poderá ignorá-las sem obter um erro. Poderíamos começar, por exemplo, a 0x0100, mas não faz sentido, visto que o processador inicia a execução em 0000. Se você quiser colocar uma tabela exatamente em uma certa localização no segmento de código, você pode usar ORG. Se você quiser demarcar onde realmente começa o seu programa, após definir outras coisas com diretivas .DEF e .EQU, use a seqüência CSEG/ORG para sua orientação, mesmo sabendo que não é necessário utilizá-las.

Como a primeira palavra de código sempre inicia no endereço zero, esta localização é também chamada de vetor de reset. Procedendo os próximos endereços, 0x0001, 0x0002 etc., temos os vetores de interrupção. Estas são as posições para onde a execução salta se uma interrupção externa ou interna foi habilitada e ocorre. Estas posições chamadas vetores são específicas para cada tipo de processador, e dependem do hardware interno disponível (veja abaixo). Os comandos para reagir às interrupções devem ser colocadas nas localizações de vetor apropriados. Se você usar interrupções, o primeiro código, no vetor de reset, deve ser um comando de salto (jump), para saltar por cima dos outros vetores. Cada vetor de interrupção deve possuir um comando de salto para a respectiva rotina para tratar a interrupção. A típica seqüência do programa no início é como se segue:

```
.CSEG
.ORG 0000
        RJMP Inicio
        RJMP TrataInterrupcao1

[...] aqui colocamos os outros comandos no vetor de interrupção

[...] e aqui é um bom lugar para colocar as próprias rotinas de tratamento da interrupção

Inicio: ; É aqui que o programa inicia
[...] Aqui colocamos o programa principal
```

O comando RJMP resulta em um salto para o label “Inicio:”, localizado algumas linhas abaixo. Lembre-se, labels sempre iniciam na coluna 1 do código fonte e terminam com um “:”. Labels que não atendam esta condições não são interpretadas corretamente pelo compilador. Labels faltantes resultam em uma mensagem de erro (“Label indefinido”, *Undefined Label*), e a compilação é interrompida.

# Execução linear do programa e desvios

A execução de um programa é sempre linear, se nada mudar a execução sequencial. Estas mudanças são a execução de uma interrupção, ou instruções de desvio.

Os desvios normalmente são dependentes de alguma condição, chamados portanto desvios condicionais. Como exemplo, vamos imaginar que construímos um contador de 32 bits utilizando os registradores R1 a R4. O byte menos significativo R1 é incrementado. Se o registrador estoura (*overflow*) durante esta operação (255 + 1 = 0), temos que incrementar R2 similarmente. Se R2 estoura, incrementamos R3, e assim por diante.

O incremento por um é feito pela instrução INC. Se um estouro (overflow) ocorre durante esta execução de INC R1, o bit zero do registrador de status é setado para um (o resultado da operação é zero). O carry do registrador de status, usualmente setado em overflows, não é alterado durante um INC. Isto não é para confundir o iniciante, mas o carry é usado para outros propósitos. A flag de Zero ou bit de zero neste caso é suficiente para detectar um estouro. Se não ocorrer nenhum estouro, mantemos a seqüência de contagem.

Se o bit-Zero for setado, devemos executar um incremento adicional nos outros registradores. Para confundir o iniciante, o comando de desvio que temos que usar não é chamado BRNZ mas BRNE (*Branch if Not Equal*, Desviar se não for igual). Uma questão de gosto...

A seqüência de contagem inteira do contador de 32 bits então deve ficar assim:

```
        INC R1
        BRNE Continue32
        INC R2
        BRNE Continue32
        INC R3
        BRNE Continue32
        INC R4
Continue32:
```

É só isso. A condição oposta a BRNE é BREQ ou *Branch EQual*, desviar se for igual.

Para saber quais os bits de status, também chamados de flags do processador são alterados durante a execução de um comando, olhe a Lista de Instruções. Da mesma forma que o bit Zero, você pode usar outros bits como:

```
BRCC  label/BRCS label; Carry-flag 0 (BRCC) ou 1 (BRCS)
BRSH  label; Igual ou maior
BRLO label; Menor
BRMI label; Menos
BRPL label; Mais
BRGE label; Maior ou igual (com bit de sinal)
BRLT label; Smaller (with sign bit)
BRHC label/BRHS label; Flag de meio-overflow 0 ou 1
BRTC label/BRTS label; T-Bit 0 ou 1
BRVC label/BRVS label; Flag de complemento de dois 0 ou 1
BRIE label/BRID label; Interrupções habilitadas ou desabilitadas
```

para reagir a diferentes condições. Desvios sempre ocorrem quando as condições são cumpridas. Não tenha medo, a maioria destes comandos é raramente usada. Para o iniciante, apenas Zero e Carry são importantes.

# Temporização durante a execução do programa

Como mencionado anteriormente, o tempo necessário para executar uma instrução é igual ao tempo de ciclo do clock do processador. Se o processador roda a 4 MHz. Então uma instrução requer 1/4 µs ou 250 ns, e a 10 MHz, apenas 100 ns. O cálculo do tempo é tão preciso quanto o cristal do clock. Se você precisa temporização exata, um AVR é a melhor solução. Note que há poucos comandos que necessitam de dois ou mais ciclos, e.g. As instruções de desvio (se este ocorre) ou seqüências de leitura/escrita na

SRAM. Veja a tabela de instruções para detalhes.

Para definir o tempo exato, deve haver uma oportunidade para não fazer nada e retardar a execução do programa. Você pode utilizar outras instruções para fazer nada, mas mais inteligente é a utilização do comando NOP (*NO oPeration,* sem operação). Esta é a instrução mais inútil:

```
        NOP
```

Esta instrução não faz nada além de desperdiçar tempo do processador. A 4 MHz, precisamos de 4 instruções desta para desperdiçar 1 µs. Não há outros significados ocultos na instrução NOP. Para um gerador de sinal de 1 kHz não precisamos adicionar 4000 instruções destas em nosso código fonte, mas utilizamos um contador por software e algumas instruções de desvio. Com isso, construímos um elo que é executado por um certo número de vezes, provocando retardo. Um contador por ser um registrador de 8 bits que é decrementado pela instrução DEC, e.g. Desta forma:

```
        CLR R1
Conta:
        DEC R1
        BRNE Conta
```

Contagem em 16 bit também pode ser usada para um retardo exato, assim:

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

Se você usar mais registradores para construir contadores compostos, você pode conseguir qualquer retardo. E este é absolutamente exato, mesmo sem um hardware de temporização.

# Macros e execução do programa

Freqüentemente você tem que escrever seqüências de código idênticas ou similares em diferentes ocasiões no seu código fonte. Se você não quiser escrevê-la uma vez e saltar para ela via chamada de subrotina, você pode usar um macro para evitar escrever a mesma seqüência diversas vezes. Macros são sequências de codigos, escritas e testadas uma vez, e inseridas no código por seu nome macro. Como exemplo, vamos imaginar que precisamos retardar a execução do programa várias vezes de 1 µS com clock de 4 MHz. Então definimos uma macro em algum lugar do código:

```
.MACRO Retardo1
        NOP
        NOP
        NOP
        NOP
.ENDMACRO
```

A definição da macro não produz ainda nenhum código, é silenciosa. O código é produzido quando você chama a macro pelo seu nome:

```
[...] em algum lugar no código fonte
        Retardo1
[...] o código vai aqui
```

Isto resulta em quatro instruções NOP inseridas no código nesta localização. Um Retardo1 adicional insere mais 4 instruções NOP.

Ao chamar um macro por seu nome você pode adicionar alguns parâmetros para manipular o código produzido. Mais isto é mais que um iniciante precisa saber sobre macros.

Se seu macro tem longas sequências de código, ou se você está com pouco espaço de armazenamento, deve evitar o uso de macros e usar subrotinas no lugar.

# Subrotinas

Ao contrário das macros, uma subrotina economiza espaço de armazenamento. A seqüência respectiva é apenas armazenada uma vez no código e é chamada de qualquer parte. Para assegurar a execução continuada da sequencia, você deve retornar ao código que chamou a subrotina. Para um retardo de 10 ciclos você precisa escrever esta subrotina:

```
Retardo10:
        NOP
        NOP
        NOP
        RET
```

Subrotinas sempre iniciam com um label, caso contrário você não seria capaz de saltar para elas, no nosso caso, “Retardo10:”. Três NOPs seguem a instrução RET. Se você contar os ciclos necessários, você descobrirá que são 7 (3 para os NOPs, 4 para o RET). Os 3 que faltam estão contidos nesta rotina:

```
[...] algum lugar do código fonte:
        RCALL Retardo10
```

```
[...] continuando com o código fonte
```

RCALL é uma chamada relativa. A chamada é codificada como um salto relativo, a distância relativa da rotina de chamada para a subrotina é calculada pelo compilador. A instrução RET volta à rotina de chamada. Note que antes de você usar chamadas de subrotina, você deve setar o ponteiro de pilha (veja Pilha), porque o endereço para retorno deve ser colocado na pilha pela instrução RCALL.

Se você quiser saltar diretamente a algum outro lugar no código, você tem que usar a instrução jump:

```
[...] em algum lugar do código fonte
        RJMP Retardo10
Retorno:
[...] continuando com o código fonte
```

A rotina para onde você pulou não pode usar o comando RET neste caso. Para voltar à localização de onde você chamou, requer outro label e que a rotina que você chamou, salte de volta para este label. Este tipo de salto não é chamado de subrotina, porque você não pode chamá-la de diferentes localidades do código.

RCALL e RJMP são desvios incondicionais. Para pular para uma localidade, dependendo de alguma condição, você terá que combinar com instruções de desvio. A chamada condicional de uma subrotina pode ser feita com os seguintes comandos. Se você quiser chamar uma subrotina dependendo de um certo bit em um registrador, use a sequência seguinte:

```
SBRC R1,7 ; Pule a próxima instrução se o bit 7 for 0
RCALL OutroLabel ; Chame a subrotina
```

SBRC quer dizer “Pule a próxima instrução se o Bit 7 no Registrador R1 for zero, *skip next instruction if Bit 7 in Register R1 is Clear*). A instrução RCALL para o OutroLabel: somente será executada se o bit 7 no registrador R1 for 1, pois a próxima instrução será pulada se for 0. Se você desejar chamar a subrotina caso o bit seja 0, você pode utilizar a instrução correspondente SBRS. A instrução que se segue a SBRS/ SBRC pode ser de um ou dois bytes, o processador sabe quanto ele tem que pular. Note que os tempos de execução são diferentes neste caso. Para pular sobre mais de uma instrução, estes comandos não podem ser usados.

Se você tiver que pular uma instrução se dois registradores tiverem o mesmo valor, você pode usar a instrução exótica

```
CPSE R1,R2 ; Compara R1 e R2, pula se forem iguais
RCALL AlgumaSubrotina ; Chama alguma subrotina
```

É um comando raramente utilizado, esqueça-o no começo. Se você quiser pular a instrução seguinte dependendo de um certo bit em uma porta, use as instruções SBIC e SBIS. Elas querem dizer “Pule se o Bit no espaço E/S estiver zero (ou Setado), *Skip if the Bit in I/o space is Clear (ou Set)”*, assim:

```
SBIC PINB,0 ; Pule se o Bit 0 na porta B for 0
RJMP DestinoA ; Pule para o label DestinoA
```

A instrução RJMP será executada apenas se o bit 0 na porta B for 1. É de alguma forma confuso para o iniciante. O acesso aos bits da porta é limitado à metade inferior, as 32 portas superiores não podem ser usadas aqui.

Agora, outra aplicação exótica para o expert. Pule esta parte se você for iniciante. Imagine que temos 4 chaves, uma para cada bit, conectados à porta B. Dependendo do estado destes 4 vits, gostaríamos de pular para 16 diferentes localizações no código. Agora podemos ler a porta e usar diversas instruções de desvio para descobrir para onde teremos que pular hoje. Como alternativa, você pode escrever uma tabela contendo os 16 endereços, assim:

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

Em nosso código, copiamos o endereço da tabela para o ponteiro Z:

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

e adicionar o estado atual da porta B (em R16) a este endereço.

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

Agora podemos saltar às localidades na tabela, da mesma forma que chamamos uma subrotina:

```
ICALL
```

ou pular, sem possibilidade de retorno:

```
IJMP
```

O processador carrega o conteúdo do par de registradores Z no contador de programa e continua a operação a partir daí. Mais inteligente que fazer desvios e mais desvios?

# Interrupções e a execução do programa

Frequentemente temos que reagir a condições do hardware e outos eventos. Uma exemplo, é uma alteração em um pino de entrada. Você pode programar uma reação como um loop, sempre verificando se houve alteração em um pino. Este método é chamado de busca, como uma abelha voando em círculos procurando por novas flores. Se você tem que detectar pulsos curtos com duração inferior a microssegundos, este método é inútil. Neste caso, você terá que programar uma interrupção.

Uma interrupção é disparada por alguma condição de hardware. Esta condição tem que ser habilitada primeiro, todas as interrupções de hardware são disabilitadas por padrão, no momento do reset. Os bits respectivos da porta que habilitam a resposta à interrupção devem ser habilitados primeiro. O processador tem um bit no seu registrador de status que o habilita a responder à interrupção de qualquer componente, a flag de habilitação de interrupção (*Interrupt Enable Flag*). Habilitar a resposta às interrupções requer o seguinte comando:

```
SEI ; Seta bit de habilitação de Interrupção
```

Se a condição de interrupção ocorre, e.g. uma mudança no bit da porta, o processador coloca o contador de programa atual na pilha (que deve ser habilitada primeiro! Veja a inicialização do ponteiro de pilha na seção Pilha da descrição da SRAM). Sem isso, o processador não seria capaz de retornar ao ponto onde estava quando ocorreu a interrupção (o que pode ocorrer a qualquer momento em qualquer lugar da execução do programa). Depois, o processamento salta para a localização predefinida, o vetor de interrupção, e aí executa as instruções.

Normalmente a instrução é um JUMP para a rotina de tratamento da interrupção, localizado em algum lugar do código. A localização do vetor de interrupção é específico do processador, e dependente dos componentes de hardware e da condição que levou à interrupção. Quanto mais componentes de hardware e mais condições, mais vetores. Os diferentes vetores para alguns tipos de AVR estão listados na tabela a seguir (o primeiro vetor não é uma interrupção, mas sim o vetor de reset, que não faz operação com a pilha!)

| ***Nome*** | ***Endereço do vetor de interrupção*** | | | ***Disparado por*** |
|---|---|---|---|---|
| | ***2313*** | ***2323*** | ***8515*** | |
| RESET | 0000 | 0000 | 0000 | Hardware Reset, Power-On-Reset, Watchdog Reset |
| INT0 | 0001 | 0001 | 0001 | Mudança de nível no pino externo INT0 |
| INT1 | 0002 | - | 0002 | Mudança de nível no pino externo INT1 |
| TIMER1CAPT | 0003 | - | 0003 | Evento de captura no Timer/Contador 1 |
| TIMER1COMPA | - | - | 0004 | Timer/Contador 1 = Valor de comparação A |
| TIMER1 COMPB | - | - | 0005 | Timer/Contador 1 = Valor de comparação B |
| TIMER1 COMP1 | 0004 | - | - | Timer/Contador 1 = Valor de comparação 1 |
| TIMER1 OVF | 0005 | - | 0006 | Estouro (overflow) de Timer/Contador 1 |
| TIMER0 OVF | 0006 | 0002 | 0007 | Estouro (overflow) de Timer/Contador 0 |
| SPI STC | - | - | 0008 | Transmissão Serial Completa |
| UART TX | 0007 | - | 0009 | Caractere UART disponível no buffer de recepção |
| UART UDRE | 0008 | - | 000A | Transmissão UART esgotada |
| UART TX | 0009 | - | 000B | UART totalmente enviada |
| ANA_COMP | - | - | 000C | Comparador Analógico |

Note que a capacidade de reagir a eventos é muito diferente para os diferentes tipos. Os endereços são sequenciais, mas não idênticos para diferentes tipo. Consulte o datasheet para cada tipo de AVR.

Quanto mais alto um vetor na lista, maior é sua prioridade. Se dois ou mais componentes têm uma condição de interrupção ao mesmo tempo, o vetor com o menor endereço vence. A interrupção inferior tem que esperar até que a primeira tenha sido tratada. Para evitar que as interrupções inferiores interrompam a execução do tratamento de uma interrupção superior, a primeira interrupção desabilita a flag I (interrupção) do processador. A rotina deve reabilitar esta flag após o término do trabalho.

Para reabilitar o bit de status I há duas formas. O tratamento da interrupção para terminar com o comando

```
RETI
```

Este comando retorna o bit I depois do contador de programa ter sido restabelecido com o valor do endereço de retorno.

A segunda forma é habilitar o bit I pela instrução

```
SEI ; Habilita bit de Interrupção
RET ; Retorna
```

Isto não faz a mesma coisa do RETI, pois interrupções subsequentes estarão habilitadas antes do contador de programa ser recarregado com o endereço de retorno. Se outra interrupção estiver pendente, sua execuçao já inicia antes que o endereço de retorno seja removido da pilha. Dois ou mais endereços de retorno permanecem na pilha. Isto não causa bug imediato, mas é um risco desnecessário. Portanto, use a instrução RETI para evitar fluxo desnecessário na pilha.

Um vetor de interrupção tem capacidade para apenas uma instrução de salto para a rotina de serviço. Se certa interrupção não é usada ou indefinida, colocamos simplesmente uma instrução RETI ali, no caso de uma falsa interrupção ocorrer. É o caso da execução do serviço respectivo não resetar automaticamente a condição de interrupção do periférico. Neste caso, um simples RETI causaria um loop contínuo de resets. Este é o caso com algumas interrupções da UART.

Quando uma interrupção está em tratamento, a execução de outras interrupções de menor prioridade é bloqueada. Por este motivo, todas as rotinas de tratamento de interrupção devem ser tão curtas quanto possível. Se você precisar de uma longa rotina de tratamento da interrupção, use um dos seguintes métodos. O primeiro é permitir interrupções com SEI dentro da rotina de tratamento, enquanto você executa as tarefas mais urgentes. Não é muito inteligente. Mais conveniente seria executar as tarefas urgentes, setar uma flag em algum lugar em um registrador para as reações mais lentas, e retornar da interrupção imediatamente.

Uma regra muito importante para o tratamento de interrupções é: A primeira instrução sempre tem que guardar o registrador de status na pilha, antes de você utilizar instruções que possam alterar as flags do registrador de status. O programa principal, que foi interrompido, pode precisar utilizar uma flag que estava setada, para tomar alguma decisão, e a interrupção alterou justamente esta flag. Coisas engraçadas podem acontecer de tempos em tempos. A última instrução antes de RETI deverá ser recuperar o conteúdo do registrador de status da pilha e restaurar seu conteúdo original.

Pela mesma razão, todos os registradores usados no tratamento da interrupção devem ser reservados exclusivamente para aquele propósito, ou salvos na pilha e restaurados ao fim da rotina. Nunca altere o conteúdo de um registrador dentro de uma tratamento de interrupção que for usado em outras partes do programa, sem restaurá-lo antes.

Devido a todas estas necessidades, um exemplo mais sofisticado de tratamento de interrupção, é mostrado aqui.

```
.CSEG ; Segmento de código começa aqui
.ORG 0000 ; Endereço é zero
        RJMP Inicia; O vetor de reset no endereço 0
        RJMP ServicoInt ; 0001: Primeiro vetor de interrupção, tratamento de INT0
[...] aqui outros vetores

Inicia: ; Aqui o programa principal inicia
[...] aqui há espaço suficiente para definir pilha e outras coisas

ServicoInt: ; Aqui iniciamos a rotina de tratamento da interrupção
        PUSH R16 ; salvamos um registrador na pilha
        IN R16,SREG ; lemos o registrador de status (n.t. e colocamos em R16)
        PUSH R16 ; e o colocamos na pilha
[...] Aqui a rotina de tratamento da interrupção faz alguma coisa e usa R16
        POP R16 ; obtemos o registrador de flags de volta da pilha (n.t. Para R16)
        OUT SREG,R16 ; e o restauramos na flag de status
        POP R16 ; obtemos o conteúdo anterior de R16 na pilha
        RETI ; e retornamos da interrupção
```

Parece um pouco complicado, mas é um pré-requisito para usar interrupções sem produzir sérios bugs. Ignore PUSH R16 e POP R16 se você pode reservar este registrador exclusivamente para uso no tratamento das interrupções. Como o tratamento da interrupção não pode ser interrompido (a não ser que você permita, no código), todas as rotinas de tratamento de interrupção podem usar o mesmo registrador.

É isso para o iniciante. Há mais algumas coisas com interrupções, mas é o suficiente para começar e não o confundir.

# Cálculos

Agora discutiremos os comandos necessários para calcular em linguagem assembler AVR. Isto inclui sistemas numéricos, setar e zerar bits, deslocar e rotacionar, e adicionar/subtrair/comparar e conversão de formatos numéricos.

# Sistemas numéricos em assembler

Os seguintes formatos numéricos são comuns em assembler:

- Números positivos inteiros (Bytes, Palavras, etc.),
- Números com sinal (inteiros),
- Dígitos codificados em binário (BCD, *Binary Coded Digits*)
- BCDs compactados,
- Números formatados em ASCII.

## Números positivos inteiros (bytes, palavras, etc.)

O menor número inteiro a ser manipulado em assembler é um byte com oito bits. Ele codifica números entre 0 e 255. Tais bits encaixam perfeitamente em um registrador da MCU. Todos os números maiores devem ser baseados neste formato básico, usando mais do que um registrador. Dois bytes formam uma palavra, *word*, (que vai de 0 a 65.535), três bytes formam uma palavra maior (vai de 0 a 16.777.215) e quatro bytes formam uma palavra dupla, *double word* (de 0 a 4.294.967.295).

Os bytes de uma palavra ou palavra dupla podem ser armazenados no registrador que você quiser. Operações com estes bytes são programados byte por byte, então você não tem que se preocupar em colocá-los em ordem. Para formar um conjunto de uma palavra dupla, podemos armazenar assim:

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

dw0 a dw3 estão em ordem nos registradores. Se quisermos iniciar esta palavra dupla no início de uma aplicação, devemos fazer da seguinte forma:

```
.EQU dwi = 4000000 ; definimos a constante
        LDI dw0,LOW(dwi) ; Os 8 bits mais baixos em R16
        LDI dw1,BYTE2(dwi) ; bits 8 .. 15 em R17
        LDI dw2,BYTE3(dwi) ; bits 16 .. 23 em R18
        LDI dw3,BYTE4(dwi) ; bits 24 .. 31 em R19
```

Então, convertemos este número decimal, chamado dwi, em suas porções binárias e as quebramos em quatro pacotes de bytes. Agora você pode fazer cálculos com esta palavra dupla.

## Números com sinal (inteiros)

Algumas vezes, mas em raros casos, você precisa calcular com números negativos. Um número negativo é definido interpretando o bit mais significativo de um byte como um bit de sinal. Se for 0, o número é positivo. Se for 1, o número é negativo. Se for negativo, usualmente não armazenamos o número em sua forma normal, mas usamos seu valor invertido. Invertido significa que -1 como um byte inteiro não é escrito 1000.0001 mas 1111.1111. Isto significa: subtraia 1 de 0 e esqueça o estouro (overflow). O primeiro bit é o bit de sinal, sinalizando que este é um número negativo. O porquê deste formato diferente (subtrair o número negativo de 0) ser usado é fácil de entender. Somar -1 (1111.1111) e +1 (0000.0001) dá exatamente zero, se você esquecer o estouro que ocorre durante esta operação (o nono bit).

Em um byte, o maior número inteiro é +127 (binário 0,111111), o menor é -128 (binário 1,0000000). Em outras linguagens de computador, este formato numérico é chamado de inteiro curto *(short integer).* Se você precisar de uma faixa de valores maior, você pode adicionar outro byte para formar um número inteiro, indo de +32.767 até -32.768, enquanto que quatro bytes fornecem uma faixa de +2.147.483.647 até -2.147.483.648, usualmente chamado de Inteiro longo (LongInt) ou Inteiro Duplo (DoubleInt).

## Dígitos Codificados em Binário, BCD *(Binary Coded Digits)*

Números inteiros positivos ou com sinal nos formatos assima utilizam o espaço disponível da forma mais eficiente. Outra formato numérico, menos denso, mas fácil de trabalhar, é armazenar os números decimais em um byte para cada dígito. O dígito decimal é armazenado em sua forma binário em um byte. Cada dígito de 0 a 9 precisa de quatro bits (0000 a 1001), os quatro bits superiores são zeros, desperdiçando muito espaço em cada byte). Por exemplo, para mostrarmos o número 250, precisamos de no mínimo três bytes:

| *Valor do bit* | *128* | *64* | *32* | *16* | *8* | *4* | *2* | *1* |
|---|---|---|---|---|---|---|---|---|
| R16, Dígito 1 =2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| R17, Dígito 2 = 5 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| R18, Dígito 3 = 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

```
;Instruções para codificar:
LDI R16,2
LDI R17,5
LDI R18,0
```

Você pode calcular com estes números, mas é um pouco mais complicado em assembler do que calcular com valores binários. A vantagem dste formato é que você pode manipular números tão grandes quanto você quiser, desde que haja espaço suficiente. Os cálculos são tão precisos quanto você quiser (se você programa AVRs para aplicações de banco), e você pode convertê-los facilmente para cordões (*strings*) de caracteres.

## BCDs compactados

Se você comprimir dois dígitos decimais dentro de um byte, você não perderá tanto espaço de armazenagem. Este método é chamado de dígitos codificados em binário compactados (*packed binary coded digits*). As duas partes de um byte são chamados de nibble superior e inferior. O nibble superior normalmente guardo o dígito mais significativo, o que tem vantagens no cálculo (instruções especiais na linguagem assembler do AVR). O número decimal 250 então ficaria assim formatado em BCD compactado:

| *Byte* | *Dígitos* | *Valor* | *8* | *4* | *2* | *1* | *8* | *4* | *2* | *1* |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 & 3 | 02 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 2 & 1 | 50 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |

```
; Instruções para codificar
LDI R17,0x02 ; Byte superior
LDI R16,0x50 ; Byte inferior
```

Para fazer isto corretamente, você pode usar a notação binária (0b...) ou a notação hexadecimal (0x...) para colocar os bits na posição correta dentro dos nibbles.

O cálculo com BCDs compactados é um pouco mais complicado comparado com a forma binária. Para converter em cordões de caracteres, é tão fácil quanto com BCDs. O comprimento dos números e a precisão dos cálculos é limitado somente pelo espaço de armazenagem.

## Números em formato ASCII

Muito similar ao formato BCD é a armazenagem de números em formato ASCII. Os dígitos 0 a 9 são armazenados utilizando sua representação ASCII (ASCII = *American Standard Code for Information Interchange*, Padrão de Códigos Americanos para Intercâmbio de Informação). O ASCII é um formato muito antigo, desenvolvido e otimizado para escritores de teletipo, desnecessariamente complicado para utilização em computadores (você sabe o que um caractere chamado *End Of Transmission* (fim da transmissão) EOT significava quando foi inventado?), muito limitado para outras línguas que não o inglês (somente 7 bits por caractere), ainda usado em comunicações hoje devido aos esforços limitados de programadores de sistemas operacionais no sentido de mudar para sistemas de caracteres mais efetivos. Este sistema antigo só é ultrapassado pelo conjunto de caracteres de 5 bit Europeu chamado conjunto Baudot ou o código Morse, ainda em uso por algumas pessoas com dedos rápidos.

Dentro do sistema de codificação ASCII, o dígito decimal 0 é representado pelo número 48 (hexa 0x30, binário 0b0011.0000), dígito 9 é 57 (hexa 0x39, binário 0b0011.1001). O ASCII não foi projetado para ter estes números no início do conjunto de código, pois já há alguns caracteres de comando como o EOT mencionado para teletipo. Então temos que adicionar 48 a um BCD (ou setar os bit 4 e 5 para 1) para converter BCD para ASCII. Os números formatados em ASCII precisam do mesmo espaço de armazenagem que o BCD. Carregar 250 para um conjunto de registradores representando este número ficaria desta forma:

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

A representação ASCII destes caracteres são escrito nos registradores.

# Manipulação de bits

Para converter um dígito codificado em BCD para sua representação em ASCII, temos que setar os bits 4 e 5 para um. Em outras palavras, precisamos fazer uma operação OR no BCD com um valor hexa de 0x30. Em assembler isto é feito da seguinte forma:

```
ORI R16,0x30
```

Se temos um registrador que já tem o valor hexa 0x30, podemos fazer OR com este registrador para convertê-lo para BCD:

```
OR R1,R2
```

Converter um ASCII de volta para BCD é fácil. A instrução

```
ANDI R16,0x0F
```

isola os quatro bits mais baixos (= o nibble inferior). Note que ORI e ANDI são possíveis apenas com registradores acima de R15. Se precisar utilizá-los, use um dos registradores entre R16 e R31!

Se o valor hexa 0x0F já estiver no registrador R2, você pode fazer AND o caractere ASCII com este registrador:

```
AND R1,R2
```

As outras instruções para manipular bits em um registrador são tambem limitados para os registradores acima de R15. Eles podem ser formulados assim:

```
SBR R16,0b00110000 ; Seta bits 4 e 5 para um
CBR R16,0b00110000 ; zera bits 4 e 5 para zero
```

Se um ou mais bits de um byte têm que ser invertidos, você pode usar a seguinte instrução (que não é possível usar com uma constante):

```
LDI R16,0b10101010 ; Inverte todos os bits pares
EOR R1,R16 ; do registrador R1 e armazena o resultado em R1
```

Inverter todos os bits de um byte é chamado de Complemento de Um.

```
COM R1
```

inverte o conteúdo do registrador R1 e substitui zeros por uns e vice versa. Isto é diferente do Complemento de Dois, que converte um número com sinal positivo pelo seu complemento negativo (subtraindo de zero). Isto é feito com a instrução

```
NEG R1
```

Então +1 (decimal: 1) vira -1 (binário 1.1111111), +2 vira -2 (binário 1.1111110), e assim por diante.

Além da manipulação de bits em um registrador, a cópia de apenas um bit é possível usando a bit T do registrador de status. Com

```
BLD R1,0
```

O bit T é carregado com uma cópia do bit 0 do registrador R1. O bit T pode ser setado ou zerado, e seu conteúdo pode ser copiado para qualquer bit em qualquer registrador:

```
CLT ; zera bit T, ou
SET ; seta bit T, ou
BST R2,2 ; copia bit T para o bit 2 do registrador R2
```

# Desloca e rotaciona

Deslocamento e rotação de números binários significa multiplicação ou divisão por 2. O deslocamento tem várias sub-instruções.

A multiplicação por 2 é facilmente feita deslocando todos os bits de um byte um dígito binário à esquerda e escrevendo um zero no bit menos significativo. Isto é chamado desvio à esquerda lógico (*Logic Shift Left*). O antigo bit 7 do byte será deslocado para o carry no registrador de status.

```
LSL R1
```

O inverso, divisão por 2 é a instrução chamada desvio à direita lógico (*Logic Shift Right*)

```
LSR R1
```

O antigo bit 7, agora deslocado para o bit 6, é preenchido com 0, enquanto o antigo bit 0 é deslocado para o carry do registrador de status. Este carry poderia ser usado para arredondar para cima e para baixo (se estiver setado, adicionar um ao resultado). Exemplo, divisão por quatro com arredondamento:

```
        LSR R1 ; divisão por 2
        BRCC Div2 ; Pula se não arredondar
        INC R1 ; arredonda para cima
Div2:
        LSR R1 ; Mais uma divisão por 2
        BRCC DivE ; Pula se não arredondar
        INC R1 ; Arredonda
DivE:
```

Assim, dividir é fácil com binários, contanto que você divida por múltiplos de 2.

Se inteiros com sinal forem usados, o desvio à direito sobrescreveria o bit de sinal 7. A instrução desvio à direita aritmético (*arithmetic shift right*) ASR deixa o bit 7 intocado e desloca os 7 bits inferiores, inserindo um bit zero na localização 6.

```
ASR R1
```

Da mesma forma que o desvio lógico, o antigo bit 0 vai para o carry no registrador de status.

Que tal multiplicarmos uma palavra de 16 bits por 2? O bit mais significativo do byte inferior tem que ser deslocado para se tornar o bit inferior do byte superior. Neste passo, um desvio setaria o bit mais baixo para zero, mas precisamos deslocar o carry do deslocamento anterior do byte mais baixo para o bit 0. Isto é chamado rotação. Durante a rotação, o carry no registrador de status é alterado para bit 0, e o antigo bit 7 é deslocado para o carry durante a rotação.

```
LSL R1 ; Desvio à Esquerda Lógico do byte mais baixo
ROL R2 ; Rotaciona para esquerda (ROtate Left) do byte superior
```

O desvio à esquerda lógico da primeira instrução desloca o bit 7 para o carry, a instrução ROL rola o carry para o bit 0 do byte superior. Após a segunda instrução, o carry tem o antigo bit 7. O carry pode ser usado tanto para indicar um estouro (se for feito um cálculo de 16 bits) ou para ser rolado para bytes superiores (se está sendo feito cálculo com mais de 16 bits).

A rolagem para a direita é também possível, dividindo por 2 e deslocando o carry para o bit 7 do resultado:

```
LSR R2 ; Desvio lógico à direita, bit 0 vai para o carry
ROR R1 ; Rotaciona para direita (ROtate Right) e desvia o carry para o bit 7
```

É fácil dividir números grandes. Você vê que aprender assembler não é TÃO complicado.

A última instrução que desloca quatro bits em um único passo é normalmente usado em BCDs compactados. Esta instrução desloca um nibble inteiro da posição superior para a inferior e vice-versa. Em nosso exemplo, precisamos deslocar o nibble superior para o nibble inferior. No lugar de usarmos

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

podemos fazer isso com um simples

```
SWAP R1
```

Esta instrução troca os nibbles superior e o inferior entre si. Note que o conteúdo do nibble superior será diferente após aplicar estes dois métodos.

# Somando, subtraindo e comparando

As seguintes operações de cálculo são complicadas demais para os iniciantes e demonstram que assembler é apenas para experts extremos. Leia por seu próprio risco!

Para começar a complicar, somaremos dois números de 16 bits, em R1:R2 e R3:R4. (Nesta notação, queremos dizer que o primeiro registrador é o byte mais significativo, o segundo o menos significativo).

```
ADD R2,R4 ; primeiro soma os dois bytes inferiores
ADC R1,R3 ; depois os dois bytes superiores
```

No lugar do segundo ADD, usamos ADC na segunda instrução. Isso significa somar com o carry (*Add with Carry*), que é setado ou zerado durante a primeira instrução, dependendo do resultado. Já está apavorado o suficiente por essa matemática complicada? Se não: tome isso!

Subtrairemos R3:R4 de R1:R2.

```
SUB R2,R4 ; primeiro o byte inferior
SBC R1,R3 ; depois o byte superior
```

Novamente o mesmo truque: durante a segunda instrução subtraímos outro 1 do resultado caso o resultado da primeira instrução teve um estoruo. Ainda respirando? Se sim, siga adiante!

Agora comparemos uma palavra de 16 bits em R1:R2 com outro em R2:R4 para avaliar se é maior do que o segundo. No lugar de SUB, usamos uma instrução de comparação CP, e no lugar de SBC usamos CPC:

```
CP R2,R4 ; compara bytes inferiores
CPC R1,R3 ; compara bytes superiores
```

Se a flag de carry estiver setada agora, R1:R2 é maior que R3:R4

Agora adicionamos coisas mais complicadas. Comparemos o conteúdo de R16 com a constante 0b10101010.

```
CPI R16,0xAA
```

Se o bit de Zero do registrador de status for 1 depois disso, sabemos que R16 é 0xAA. Se o carry for 1, sabemos que é menor. Se nem o carry nem o Zero forem 1, sabemos que é maior.

E agora o teste mais complicado. Avaliamos se R1 é zero ou negativo:

```
TST R1
```

Se o bit Z estiver setado, o registrador R1 é zero e podemos utilizar uma das seguintes instruções: BREQ, BRNE, BRMI, BRPL, BRLO, BRSH, BRGE, BRLT, BRVC or BRVS para desviarmos um pouco.

Ainda conosco? Se sim, há alguns cálculos para BCD compactados. Somar dois BCDs compactados pode resultar em dois estouros diferentes. O carry usual mostra um overflow, se o nibble superior resultar em mais do que 15 decimal. Outro estouro ocorre do nibble inferior para o superior, se a soma dos dois nibbles der mais do que 15 decimal.

Para servir de exemplo, vamos somar os BCDs compactados 49 (=hexa 49) e 99 (=hexa 99) para chegar a 148 (=hexa 0x0148). Somá-los usando matemática binária resultaria em um byte com o hexa 0xE2, e não ocorreria estouro. O mais baixo dos nibbles deveria ter dado um overflow, pois 9+9=18 (mais que 9) e o nibble inferior pode somente manipular números até 15. O estouro foi somado ao bit 4, o bit menos significativo do nibble superior. O que está correto! Mas o nibble inferior deveria ser 8 e é somente 2 (18 = 0b0001.0010). Deveríamos adicionar 6 a este nibble para chegarmos ao resultado correto. O que é lógico, porque sempre que o nibble inferior passar de 9, temos que adicionar 6 para corrigi-lo.

O nibble superior está totalmente incorreto, porque está 0xE e deveria ser 3 (com um 1 estourando para o próximo dígito do BCD compactado). Se adicionarmos 6 a este 0xE, conseguiremos 0x4 e o carry é setado (=0x14). Então o truque é primeiro somarmos estes dois números e depois adicionarmos 0x66 para corrigir os dois dígitos do BCD compactado. Mas espere: e se a soma do primeiro e do segundo dígitos não resultarem em estouro para o próximo nibble e não resultar em um número maior do que 9 no nibble inferior? Somar 0x66 então resultaria em um resultado totalmente incorreto. O 6 inferior deve ser adicionado se o nibble inferior estoura para o nibble superior, ou resulta em um dígito maior que 9. O mesmo com o nibble superior.

Como sabemos, se ocorre um estouro do nibble inferior para o superior? A MCU seta o bit H no registrador de status, o bit meio-carry (*half-carry*). A seguir mostramos o algoritmo para diferentes casos que são possíveis após somar dois nibbles e somar o hexa 0x6 a eles.

1. Some os nibbles. Se ocorrer estouro (C para os nibbles superiores, ou H para os inferiores), adicione 6 para corrigir. Se não, passe para o passo 2.
2. Some 6 ao nibble. Se ocorrer estouro (C ou H), está terminado. Se não, subtraia 6.

Para programar um exemplo, vamos assimir que dois BCDs compactados estão em R2 e R3, R1 ficará com o estouro, e R16 e R17 estão disponíveis para cálculos. R16 é o registrador de soma para adicionar 0x66 (o registrador R2 não pode receber um valor constante), R17 é usado para corrigir o resultado dependendo das diferentes flags. Somar R2 com R3 fica assim:

```
        LDI R16,0x66 ; para somar 0x66 ao resultado
        LDI R17,0x66 ; para depois subtrai-lo do resultado
        ADD R2,R3 ; soma os dois BCDs de dois dígitos
        BRCC NoCy1 ; pula se não ocorrer estouro
        INC R1 ; incrementa o próximo byte superior
        ANDI R17,0x0F ; não subtraia 6 do nibble inferior
NoCy1:
        BRHC NoHc1 ; pula se não ocorrer meio-carry
        ANDI R17,0xF0 ; não subtraia 6 do nibble inferior
NoHc1:
        ADD R2,R16 ; some 0x66 ao resultado
        BRCC NoCy2 ; pule se não ocorreu carry
        INC R1 ; incremente o próximo byte superior
        ANDI R17,0x0F ; não subtraia 6 do nibble superior
NoCy2:
        BRHC NoHc2 ; pule se não ocorreu meio-carry
        ANDI R17,0xF0 ; não subtraia 6 do nibble inferior
NoHc2:
        SUB R2,R17 ; correção da subtração
```

Um pouco mais curto:

```
        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
```

Pergunta para pensar: porque ambos estão corretos, e a segunda versão ocupa metade do tamanho e

menos complicado, onde está o truque?

# Conversão de formatos numéricos

Todos os formatos numéricos podem ser convertidos. A conversão entre BCD e ASCII e vice-versa já foi mostrada acima (Manipulação de bits).

A conversão para BCDs compactados não é muito complicada também. Primeiro temos que copiar o número para outro registrador. Com o valor copiado, trocamos os nibbles usando a instrução SWAP (trocar) para trocar o superior com o inferior. A parte superior é zerada, e.g. Fazendo AND com 0x0F. Agora temos o BCD do nibble superior e podemos usá-lo como BCD ou setar bit 4 e 5 para convertê-lo em um caractere ASCII. Depois disso, copiamos novamente o byte e tratamos o nibble inferior sem fazer SWAP para obter o BCD inferior.

Um pouco mais complicada é a conversão de dígitos BCD para binário. Dependendo dos números a serem manipulados, primeiramente liberamos os bytes necessários que receberão o resultado da conversão. Então iniciamos com o dígito BCD mais alto. Antes de adicionar este resultado, multiplicamos o resultado por 10. (Note que no primeiro passo isto não é necessário, pois o resultado é zero).

Para fazermos esta multiplicação por 10, copiamos o resultado para algum lugar. Depois multiplicamos o resultado por 4 (duas rolagens à esquerda). Adicionar o número anteriormente copiado resulta na multiplicação por 5. Agora uma multiplicação por 2 (uma rolagem à esquerda) chega ao resultado decaduplicado. Finalmente adicionamos o BCD e repetimos este algoritmo até todos os dígitos estarem convertidos. Se, em uma destas operações, ocorrer um carry (transporte) do resultado, o BCD é grande demais para ser convertido. Este algoritmo funciona com números de qualquer comprimento, desde que haja registradores suficientes.

A conversão de binários para BCD é um pouco mais complicada. Se convertermos um binário de 16 bits podemos subtrair 10.000 (0x2710) até que ocorra um estouro, obtendo o primeiro dígito. Então repetimos com 1000 para obtermos os segundo dígito. E assim por diante com 100 (0x0064) e 10 (0x000A), e então o que remanescente é o último dígito. As constantes 10.000, 1.000, 100 e 10 podem ser colocados na memória do programa em uma tabela de palavras, organizada da seguinte forma:

```
DezTab:
.DW 10000, 1000, 100, 10
```

e podem ser lidas as palavras com uma instrução LPM da tabela.

Uma alternativa é uma tabela que contém o valor decimal de cada bit do binário de 16 bits, e.g.,

```
.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 ; e assim por diante até
.DB 0,0,0,0,0,1
```

E então você desloca os bits do binário para que este desloque dos registradores para o carry. Se for um, você adiciona o número na tabela ao resultado, lendo on números da tabela usando LPM. Isto é mais complicado para programar e um pouco mais lento do que o método acima.

O terceiro método é calcular o valor da tabela, iniciando com 000001, adicionando-o ao BCD, cada vez que você desloca um bit do seu binário para a direita e adiciona ao BCD.

Muitos métodos, muito a ser otimizado.

# Multiplicação

A multiplicação de números binários será explicada aqui.

## Multiplicação decimal

Para multiplicarmos dois números binários de 8 bits, vamos lembrar como fazemos com os números decimais:

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

Passo a passo, em decimal:

- Multiplicamos o primeiro número com o dígito menos significativo do segundo número e adicionamos ao resultado.
- Multiplicamos o primeiro número por 10 e pelo próximo dígito do segundo número e adicionamos ao resultado.
- Multiplicamos o primeiro número por 100, e ao terceiro dígito, e adicionamos ao resultado.

## Multiplicação binária

Agora em binário. A multiplicação com dígitos simples não é nececssária, pois há somente os dígitos 1 (some o número) e 0 (não some o número). A multiplicação por 10 em decimal equivale à multiplicação por 2 em modo binário. Multiplicação em 2 é feita facilmente, adicionando o número a si mesmo, ou deslocando todos os bits uma posição para a esquerda e escrevendo um 0 para a posição vazia à direita. Veja que matemátia binária é muito mais fácil que decimal. Porque a humanidade não a usou desde o começo?

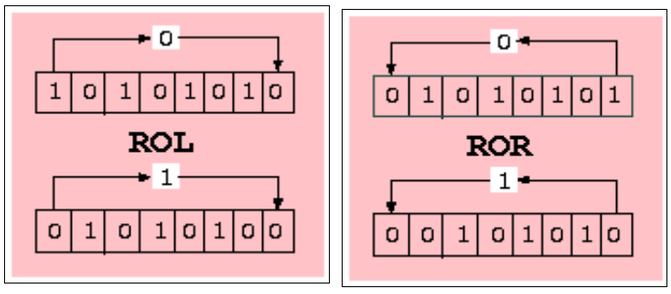
## Programa em Assembler AVR

O código fonte a seguir demonstra a realização de multiplicação em assembler.

```
; Mult8.asm multiplica dois números de 8 bit para chegar a um resultado de 16 bits.
;
.NOLIST
.INCLUDE "C:\avrtools\appnotes\8515def.inc"
.LIST
;
; Fluxo da multiplicação
;
; 1. O binário a ser multiplicado é deslocado no sentido do carry. Se for um, o número é adicionado ao resultado, se não for um, o número não é adicionado.
; 2. O número binário é multiplicado por 2, rotacionando uma posição à esquerda, e preenchendo a posição vazia com 0.
; 3. Se o binário com que será multiplicado não for zero, o ciclo de multiplicação é repetido. Se for zero, a multiplicação está terminada.
;
; Registradores usados
;
.DEF rm1 = R0 ; Número binário a ser multiplicado (8 Bit)
.DEF rmh = R1 ; Armazém intermediário
.DEF rm2 = R2 ; Número binário com que o primeiro será multiplicado (8 Bit)
.DEF rel = R3 ; Resultado, LSB (16 Bit)
.DEF reh = R4 ; Resultado, MSB
.DEF rmp = R16 ; Registrador multi propósitos para carregamento
;
.CSEG
.ORG 0000
;
      rjmp START
;
START:
      ldi rmp,0xAA ; exemplo binário 1010.1010
      mov rm1,rmp  ; para o primeiro registrador
      ldi rmp,0x55 ; examplo binário 0101.0101
      mov rm2,rmp  ; para o segundo registrador
;
; Aqui iniciamos a multiplicação de um dos binários em rm1 e rm2, o resultado vai para reh:rel (16 bits)
;
MULT8:
;
; Zera valores de início
      clr rmh ; zera armazém intermediário
      clr rel ; zera registradores de resultado
      clr reh
;
; Aqui iniciamos o ciclo de multiplicação
;
MULT8a:
;
; Passo 1: Rotaciona o bit mais baixo do número 2 para a flag de carry (divide por 2, rotaciona um zero no bit 7)
;
      clc ; zera carry
      ror rm2 ; bit 0 vai para o carry, bit 1 a 7 vão uma posição para direita, carry vai para o bit 7
;
; Passo 2: Decisão dependendo se foi 0 ou 1 que foi rotacionado para o carry
;
      brcc MULT8b ; pula por cima da adição, se o carry for 0
;
; Passo 3: Soma 16 bits em rmh:rml ao resultado, com estouro do LSB para o MSB
;
      add rel,rm1 ; soma o LSB de rm1 ao resultado
      adc reh,rmh ; soma carry e MSB de rm1
;
MULT8b:
;
```

```
; Passo 4: Multiplicar rmh:rm 1 por 2 (16 bits, desvio à esquerda)
;
	clc ; zera carry (clear carry bit)
	rol rm1 ; rotaciona o LSB à esquerda (multiplica por 2)
	rol rmh ; rotaciona o carry para o MSB e o MSB uma casa à esquerda
;
; Passo 5: Checa se há ainda números 1 no binário 2, se sim, continua multiplicando
;
	tst rm2 ; todos os bits zero?
	brne MULT8a ; se não, vá para o loop
;
; Fim da multiplicação, resultado em reh:rel
;
; Loop sem fim
;
LOOP:
	rjmp loop
```

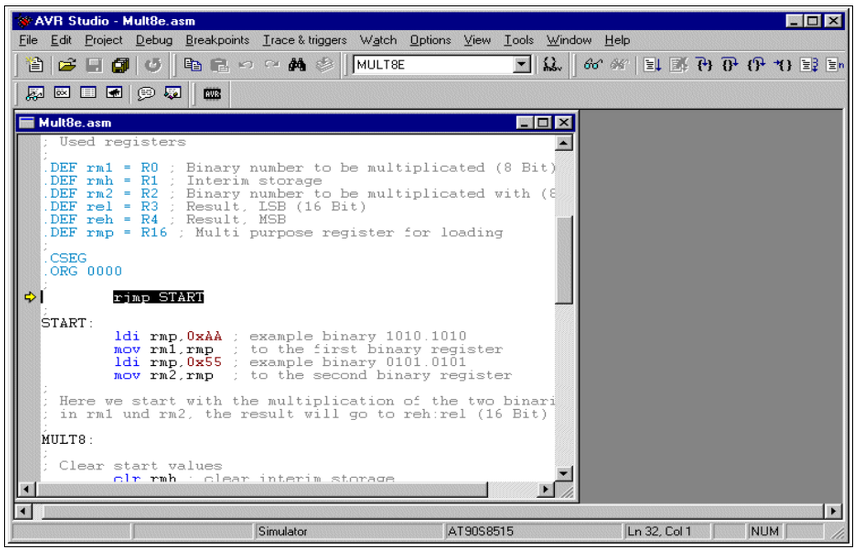
## Rotação binária



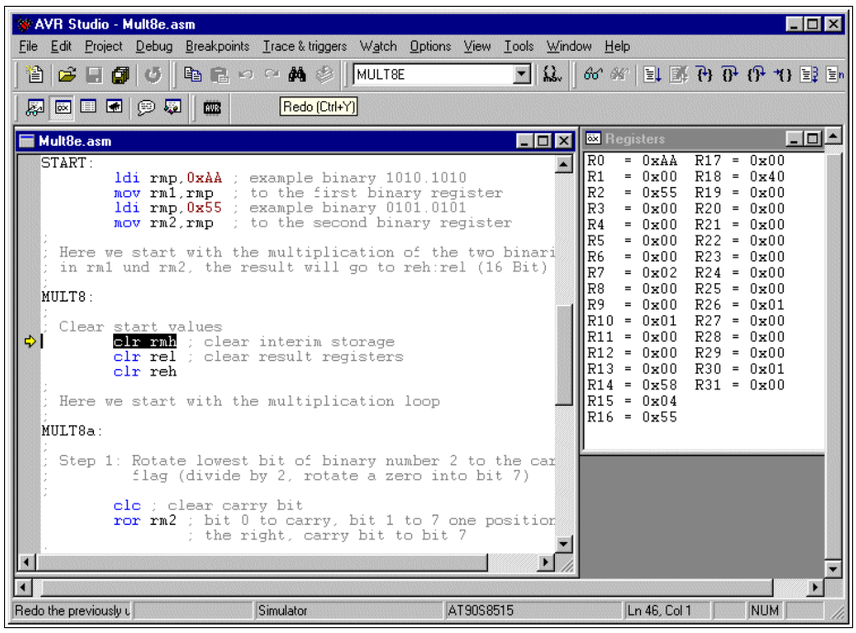

Para entender a operação de multiplicação, é necessário entender os comandos de rotação binária ROL e ROR. Estas instruções deslocam todos os bits de um registrador uma posição à esquerda (ROL) ou à direita (ROR). A posição vazia no registrador recebe o conteúdo do carry do registrador de status, e o bit que rola para fora do registrador vai para o carry. Esta operação é demonstrada usando 0xAA como exemplo de ROL e 0x55 como exemplo de ROR.

## Multiplicação no studio

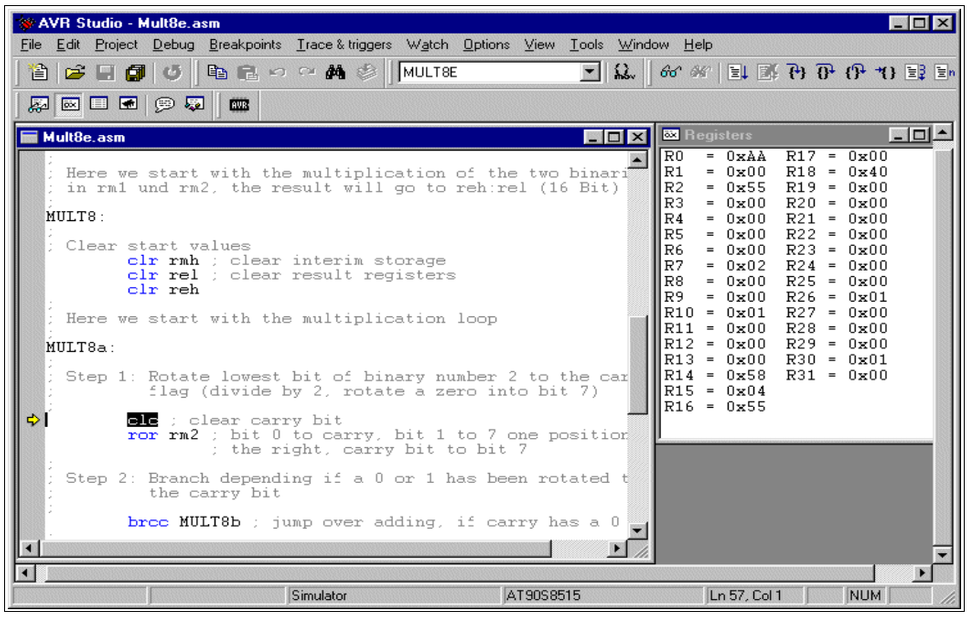
A tela abaixo mostra o programa de multiplicação no simulador.



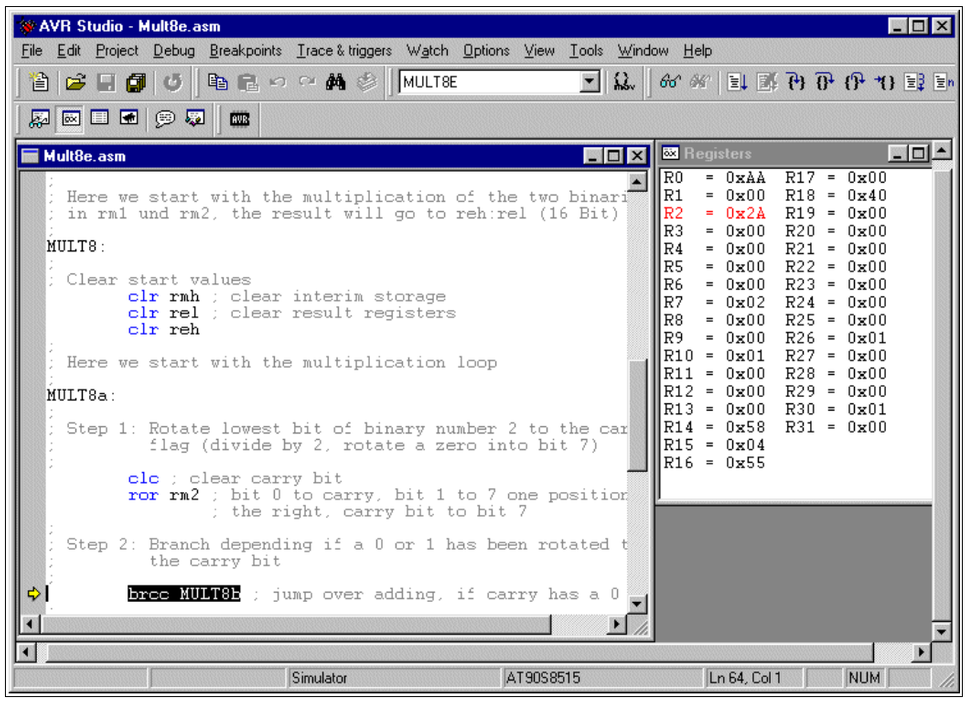

O código-objeto foi aberto, e o cursor é colocado na primeira instrução executável. F11 executa um passo.



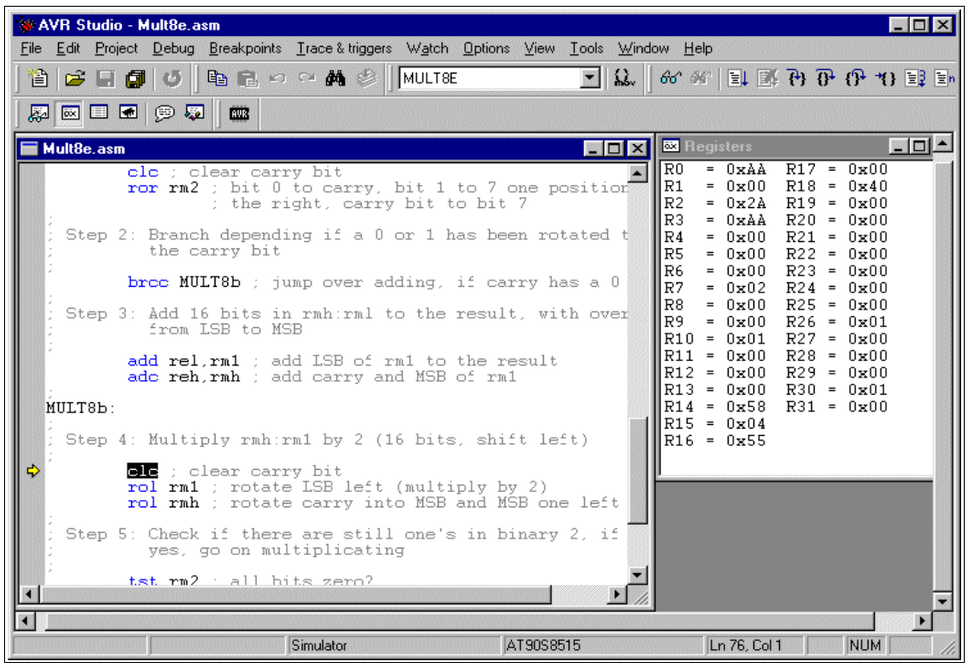

Os registradores R0 e R2 são setados para 0xAA e 0x55, nossos binários de teste, para serem multiplicados.




R2 é rotacionado à direita, para rolar o bit menos significativo para o carry, 0x55 (0101.0101) vira 0x2A (0010.1010).

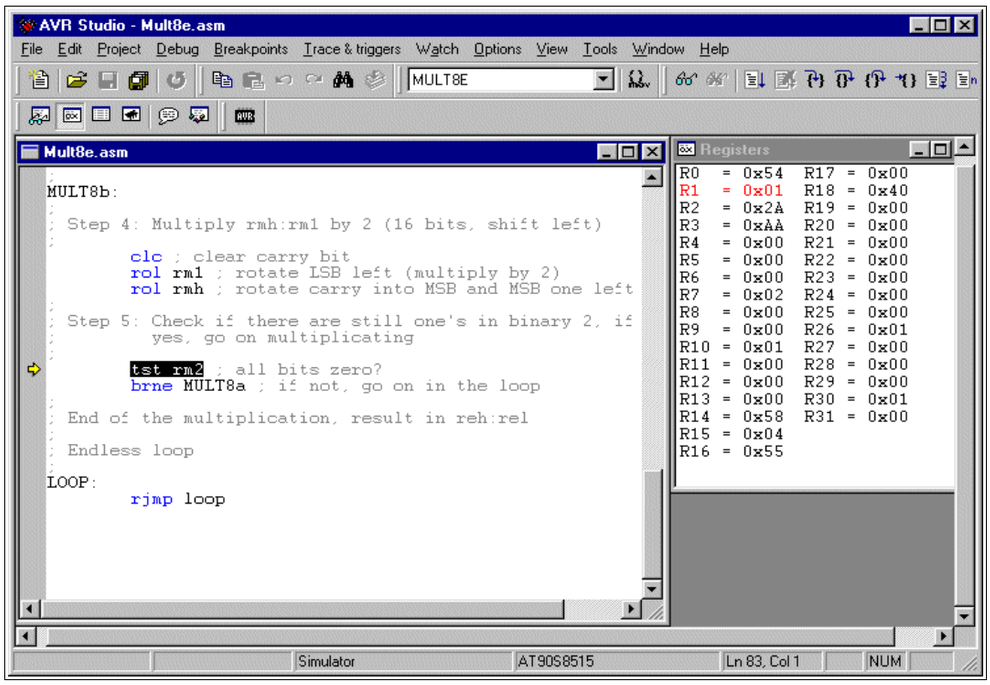



Com o carry era 1, o conteúdo dos registradores R2:R1 é somado ao par de registradores R4:R3 (vazio), resultando em 0x00AA nestes.




Agora o par de registradores R1:R0 é rotacionado uma posição à esquerda para multiplicar este binário por 2. A partir de 0x00AA, a multiplicação por 2 resulta 0x0154.

O ciclo de multiplicação é repetido enquanto haja ao menos um binário 1 no registrador R2. Os passos seguintes não são mostrados aqui.




Usando a tecla F5 no studio podemos pular estes loops até um ponto de parada no fim da rotina de multiplicação. O resultado no par de registradores R4:R3 mostra o resultado da multiplicação de 0xAA por 0x55: 0x3872.

Não foi tão complicado, basta lembrar na similaridade com as operações decimais. A multiplicação em binário é muito mais fácil do que decimal.

# Divisão

## Divisão decimal

Novamente começamos com divisão decimal, para compreender melhor a divisão binária. Vamos assumir uma divisão de 5678 por 12. Isto é feito assim:

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

==========

## Divisão binária

Em binário, a multiplicação do segundo número (4 * 1200, etc) não é necessária, devido ao fato que temos apenas 0 e 1 como dígitos. Infelizmente, números binários têm muito mais dígitos do que seu equivalente decimal, então transferir a divisão decimal em binária é um pouco inconveniente. Assim, o programa trabalha de forma um pouco diferente.

A divisão de um número binário de 16 bits por um número de 8 bits em assembler AVR está listado na seção seguinte.

```
; Div8 divide um número de 16 bits por um número de 8 bits (Testee: número de 16 bits: 0xAAAA, número de 8 bits: 0x55)
.NOLIST
.INCLUDE "C:\avrtools\appnotes\8515def.inc"
.LIST
; Registradores
.DEF rd1l = R0 ; LSB do número de 16 bits a ser dividido (dividendo)
.DEF rd1h = R1 ; MSB do número de 16 bits a ser dividido (dividendo)
.DEF rd1u = R2 ; registrador intermediário
.DEF rd2  = R3 ; número de 8 bits divisor
.DEF rel  = R4 ; LSB do resultado
.DEF reh  = R5 ; MSB do resultado
.DEF rmp  = R16; registrador multi-propósito para carregamento
;
.CSEG
.ORG 0
      rjmp start
start:
; Carregamos os números do teste para os registradores apropriados
      ldi rmp,0xAA ; 0xAAAA é o dividendo
      mov rd1h,rmp
      mov rd1l,rmp
      ldi rmp,0x55 ; 0x55 é o divisor
      mov rd2,rmp
; Divide rd1h:rd1l por rd2
div8:
      clr rd1u ; zera registrador intermediário
      clr reh  ; zera resultado (os registradores de resultado)
      clr rel  ; também usados para contar até 16 para
      inc rel  ; os passos da divisão, é setado em 1 no início)
; Aqui o loop de divisão inicia
div8a:
      clc      ; zera carry
      rol rd1l ; rotaciona o próximo bit superior do número
      rol rd1h ; para o registrador intermediário (multiplica por 2)
      rol rd1u
      brcs div8b ; um foi rolado para a esquerda, portanto subtraia
      cp rd1u,rd2 ; Resultado da divisão 1 ou 0?
      brcs div8c  ; pule sobre a subtração, se for menor
div8b:
      sub rd1u,rd2; subtraia os números para dividir
      sec      ; set carry, o resultado é 1
      rjmp div8d  ; pule para o desvio do bit de resultado
div8c:
      clc      ; zere o carry, o bit resultado é 0
div8d:
      rol rel  ; rotacione o carry para dentro dos registradores de resultado
      rol reh
      brcc div8a  ; enquanto zero rotacionar para fora dos registradores de resultado: vá para o loop de divisão
; Fim da divisão
stop:
      rjmp stop   ; loop sem fim
```

## Passos do programa durante a divisão

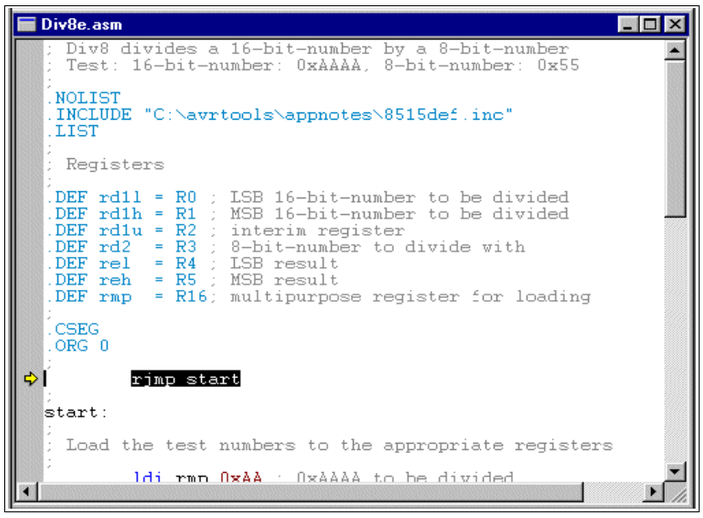
Durante a execução do programa os seguintes passos são seguidos

- Definição e pré-ajuste dos registradores com os binários de teste,
- pré-ajuste do registrador intermediário e o par de registradores de resultado (os registradores de resultado são pré-ajustados para 0x0001! Após 16 rotações, a rolagem do 1 para fora interrompe as divisões.),
- o binário de 16 bits em rd1h:rd1l é rotacionado para o registrador intermediário rd1u (multiplicação por 2), se um 1 for rotacionado para fora de rd1u, o programa desvia para o passo de subtração no passo 4 imediatamente.
- O conteúdo do registrador intermediário é comparado com o binário de 8 bits em rd2. Se rd2 for menor, ele é subtraído do registrador intermediário e o carry é setado 1, se rd2 for maior, a subtração é pulada e um zero é setado na flag de carry.
- O conteúdo da flag de carry é rotacionada no registrador de resultado reh:rel da direita.

- Se um zero foi rotacionado para fora do registrador de resulltado, temos que repetir o loop de divisão. Se foi um 1, a divisão está completa.

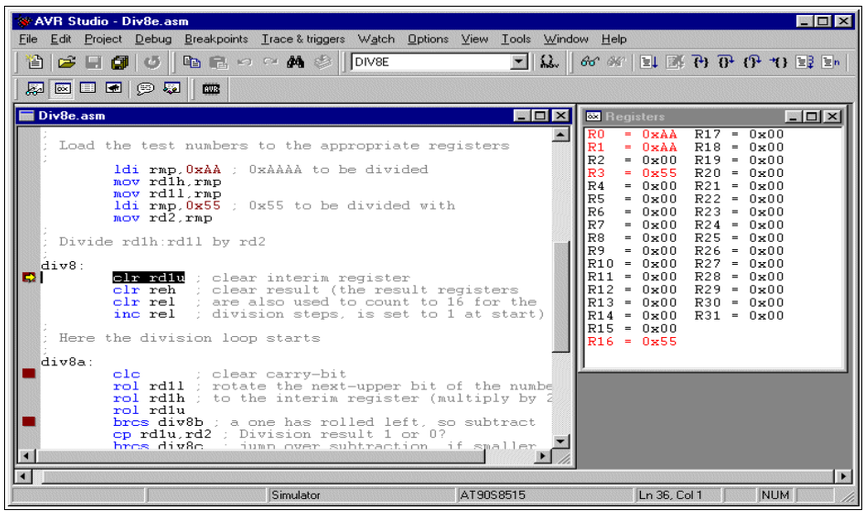
Se você não entende rotação ainda, você encontrará esta operação discutida na seção de multiplicação.

## Divisão no simulador

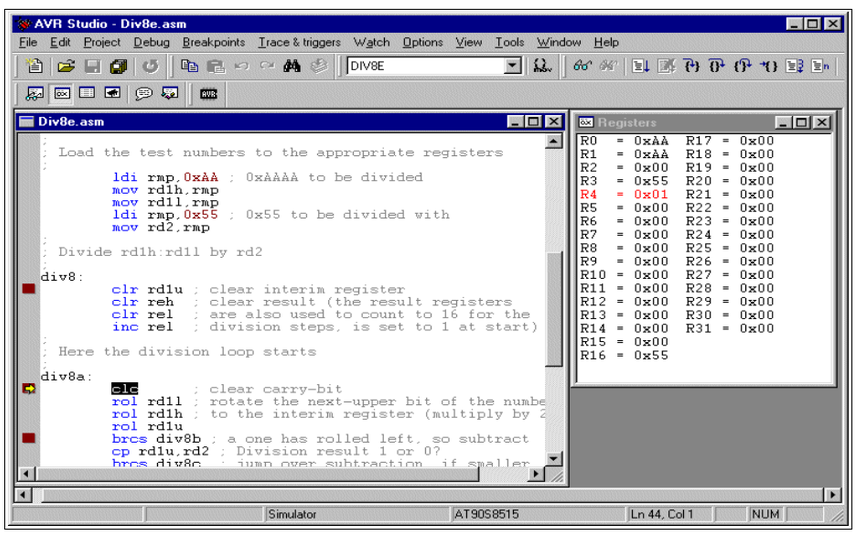



A tela a seguir demonstra os passos do programa no studio. Para fazer isto, você tem que compilar o código fonte e abrir o arquivo objeto resultante no studio.

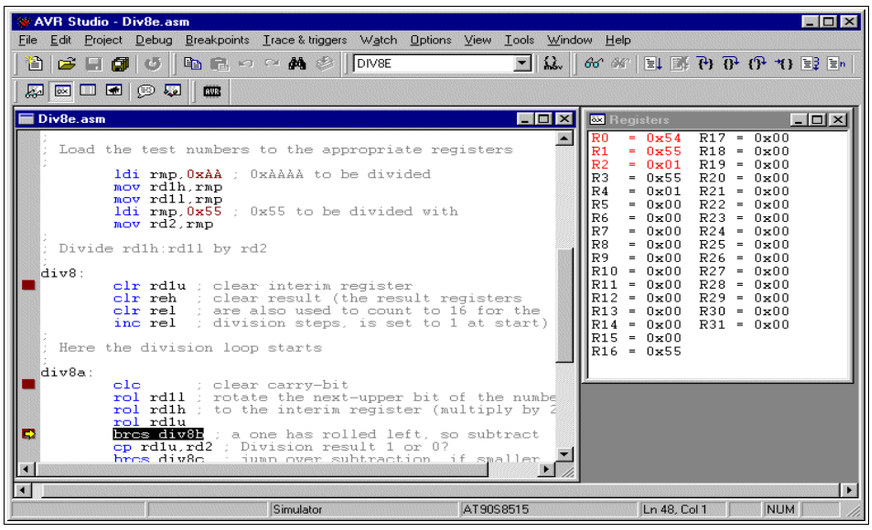
O código objeto foi iniciado, o cursor está na primeira instrução executável. F11 executa passo a passo.



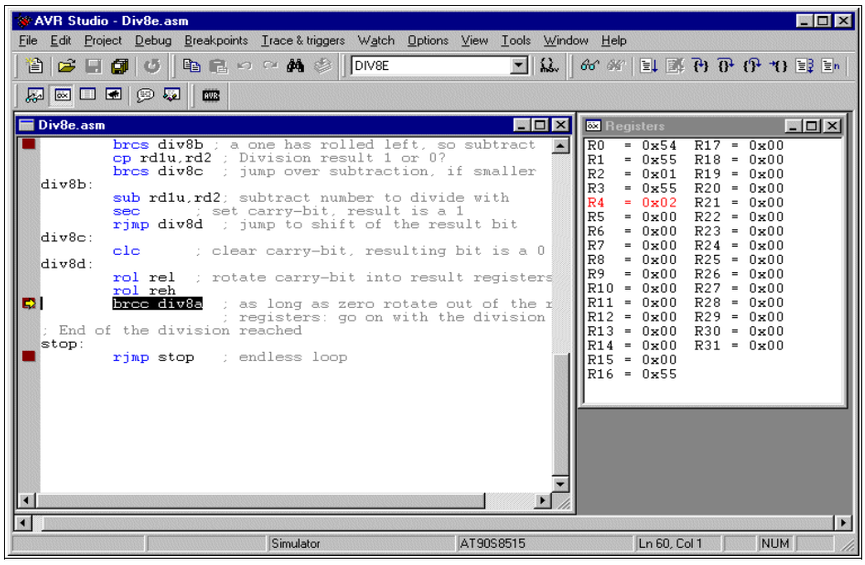

O teste, binários 0XAAAA e 0x55, serão divididos, e são escritos nos registradores R1:R0 e R3.



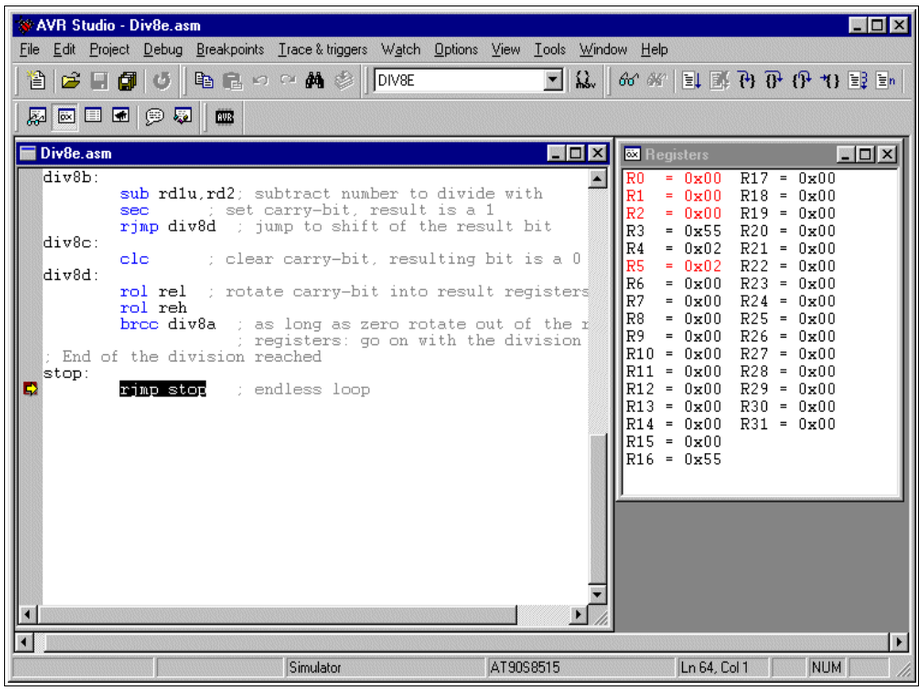

O registrador intermediário R2 e o par de registradores do resultado são setados para seus valores predefinidos.



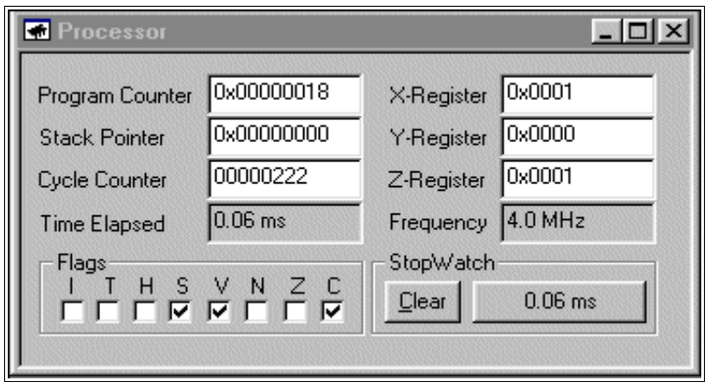

R1:R0 foi rotacionado à esquerda para R2, a partir do valor 0xAAAA chegamos ao valor dobrado 0x015554.




Não houve estouro da rotação para o carry e 0x01 em R2 era menor que 0x55 em R3, então a subtração foi pulada. Um zero no carry foi rotacionado para o registrador de resultado R5:R4. O conteúdo anterior destes registradores, um bit na posição 0 foi rotacionado para a posição 1 (conteúdo agora: 0x0002). Como um zero foi rotacionado para fora do registrador de resultados, o próximo passo a ser executado é um desvio para o início do loop de divisão e o loop é repetido.




Após executar o loop 16 vezes, chegamos ao ponto de parada no final da rotina de divisão. O registrador de resultado R5:R4 tem 0x0202, o resultado da divisão. Os registradores R2:R1:R0 estão vazios, portanto não temos resto. Se houvesse resto, poderíamos usá-lo para decidir um incremento do resultado, arredondando-o para cima. Este passo não foi codificado aqui.




Toda a divisão precisou de 60 microssegundos do tempo do processador (abra o processor view do menu studio). Um tempo bastante grande para uma divisão.

# Conversão numérica

Rotinas de conversão numéricas não são incluídas aqui. Por favor visite o website, se precisar do código fonte ou uma melhor compreensão.

# Frações decimais

Primeiro: Não use pontos flutuantes, a menos que você realmente precise deles. Pontos flutuantes são vorazes consumidores de recursos em um AVR, e precisam de muito tempo de execução. Você estará neste dilema, se você pensa que assembler é muito complicado, e preferir Basic ou outras linguagens como C e Pascal.

Mas não, se você usa assembler. Lhe será demonstrado aqui, como você pode realizar uma multiplicação de um número real em menos de 60 microssegundos, em casos especiais até com 18 microssegundos, usando uma freqüência de clock de 4 MHz. Sem quaisquer extensões para ponto flutuante ou outros truques caros para pessoas preguiçosas demais para usar o seu cérebro.

Como fazer isto? De volta à raízes de matemática! A maioria das tarefas com reais usando ponto flutuante pode ser feita usando números inteiros. Inteiros são fáceis de programar em assembler, e são velozes. O ponto decimal está somente no cérebro do programador, e é adicionado depois no fluxo digital. Ninguém se dá conta disso, este é o truque.

## Conversões lineares

Como um exemplo, a tarefa seguinte: um conversor AD de 8 bits mede a entrada de um sinal de entrada variando de 0,00 a 2,55 Volts, e retorna como resultado um binário entre $00 e $FF. O resultado é uma voltagem, para ser mostrada em um display LCD. Exemplo tão simples quanto fácil: O binário é convertido em uma string ASCII decimal entre 000 e 255, e logo depois do primeiro dígito, o ponto decimal tem que ser inserido. Pronto!

O mundo eletrônico é mais complicado. E. g., o conversor AD retorna um hexa de 8 bits para tensões de entrada entre 0,00 e 5,00 Volt. Agora complicou e não sabemos como fazer. Para mostrar o valor correto no LCD, teríamos que multiplicar o binário por 500/255, que é 1,9608. Este é um número complicado, pois é quase 2, mas apenas quase. E não queremos este tipo de imprecisão de 2%, já que temos um conversor AD com cerca de 0,25% de precisão.

Para resolver isso, multiplicamos a entrada por 200/255*256 ou 501,96 e dividimos o resultado por 256. Porque primeiro multiplicar por 256 e depois dividir por 256? É apenas para maior precisão. Se multiplicarmos a entrada por 502 ao invés de 501,96, o erro é da ordem de 0,008%. Isto está bom para nosso conversor AD, podemos conviver com isso. E dividir por 256 é uma tarefa fácil, pois é uma potência de 2 bem conhecida. Dividir números que são potência de 2, o AVR se sente muito confortável e realiza de forma muito rápida. Para dividir por 256, o AVR é ainda mais rápido, pois basta pularmos o último byte do número binário. Não precisamos nem mesmo deslocar e rotacionar!

Na multiplicação de um binário de 8 bits com o binário de 9 bits 502 (hex 16F), podemos obter um valor maior do que 16 bits. Portanto temos que reservar 24 bits ou 3 registradores para o resultado. Durante a multiplicação, a constante 502 tem que ser deslocada à esquerda (multiplicação por 2) para somar estes números ao resultado cada vez que um 1 rola para fora do número de entrada. Como podem ser necessários 8 desvios à esquerda, precisamos de mais três bytes para esta constante. Então escolhemos a seguinte combinação dos registradores para multiplicação:

| ***Número*** | ***Valor (exemplo)*** | ***Registrador*** |
|---|---|---|
| Valor de entrada | 255 | R1 |
| Multiplicador | 502 | R4 : R3 : R2 |
| Resultado | 128,010 | R7 : R6 : R5 |

Após colocar o valor 502 (00.01.F6) em R4:R3:R2 e zerar os registradores de resultado R7:R6:R5, a multiplicação segue da seguinte forma:

1. Testar, se o número da entrada já é zero. Se for, terminamos.
2. Se não, um bit do número de entrada é deslocado para fora do registrador para a direita, no carry, enquanto um bit zero é colocado no bit 7. Esta instrução é chamada desvio à direta lógico ou LSR (*Logical Shift Right*)
3. Se o bit no carry for um, somamos o multiplicador (durante o passo 1 o valor 502, no passo 2 é 1004 e assim por diante) ao resultado. Durante a adição, tomamos o cuidado com qualquer carry (somando R2 a R5 por ADD, somando R3 a R6 e R4 a R7 com instrução ADC!). Se o bit no carry era zero, não adicionamos o multiplicador ao resultado e pulamos para o próximo passo.
4. Agora o multiplicador foi multiplicado por 2, pois o próximo bit deslocado para fora do número de entrada vale o dobro. Então deslocamos R2 para a esquerda (inserindo um zero no bit 0) usando LSL. O bit 7 é deslocado para o carry. Então rotacionamos este carry para R3, rotacionando seu conteúdo um bit à esquerda, e o bit 7 para o carry. O mesmo com R4.
5. Agora terminamos com um dígito do número de entrada, e prosseguimos no passo 1 novamente.

O resultado da multiplicação por 502 agora está nos registradores de resultado R7:R6:R5. Se simplesmente ignorarmos o registrador R5 (divisão por 256), temos o nosso resultado desejado. Para aumentar a precisão, podemos usar o bit 7 em R5 para arredondar o resultado. Agora temos que converter o resultado de sua forma binária para o ASCII decimal (veja a tabela de conversão para decimal-ASCII no website). Se colocarmos um ponto decimal no local correto na string ASCII, nossa string de tensão está pronta para ser mostrada.

O programa todo, desde o número de entrada até a ASCII resultado, requer entre 79 e 228 ciclos de relógio, dependendo do número de entrada. Aqueles que quiserem vencer com rotinas de ponto flutuante de uma linguagem de programação mais sofisticada que assembler, sinta-se à vontade para me escrever o seu tempo de conversão (e a utilização de memória flash e de programa).

## Exemplo 1: Conversor AD 8-bit para saída decimal com ponto fixo

```
; Demontra conversão de ponto flutuante em Assembler, © 2003 www.avr-asm-tutorial.net
;
; A tarefa: Você lê um resultado de 8 bits de um conversor analógico-digital, o número está na faixa de hex 00 a FF.
;   Você precisa convertê-lo em um número com ponto flutuante na faixa de 0,00 a 5,00 Volts.
; O esquema do programa:
;   1. Multiplicação por 502 (hex 01F6). Este passo multipllica por 500, 256 e divide por 255 em um passo!
;   2. Arredondar e cortar o últmo byte do resultado. Este passo divide por 256 ao ignorar o último byte do resultado.
;       Antes de fazer isso, o bit 7 é usado para arredondar o resultado.
;   3. Converter a palavra resultante para ASCII e setar o sinal decimal correto. A palavra resultante que varia de 0 a 500
;       é mostrada em caracteres ASCII como 0.00 a 5.00.
; Os registradores usados:
;   As rotinas usam os registradores R8..R1 sem salvá-los antes. Também é necessário um registrador multipropósito rmp,
;   localizado na metade superior dos registradores. Assegure-se que estes registradores não conflitem com registradores
;   em uso no restante do programa.
;   O número de 8 bits é esperado no registrador R1. A multiplicação usa R4:R3:R2 para guardar o multiplicador 502
;   (é deslocado à esquerda até oito vezes durante a multiplicação). O resultado da multiplicação é calculado nos
;   registradores R7:R6:R5. O resultado da divisão por 256 é feita ignorando R5 no resultado, usando somente R7:R6.
;   R7:R6 é arredondado, depenendo do bit mais alto de R5, e o resultado é copiado para R2:R1.
;   A conversão para ASCII-string usa a entrada em R2:R1, o par de registradores R4:R3 como divisor para convesão, e
;   coloca o resultado ASCII em R5:R6:R7:R8 (R6 é o caractere decimal).
; Outras convenções:
;   A conversão usa subtotinas e a pilha. A pilha deve estar funcionando para uso de três níveis (seis bytes de SRAM).
; Temos de conversão:
;   A rotina inteira requer 228 ciclos de relógico no máximo (convertendo $FF), e 79 no mínimo (convertendo $00).
;   A 4 Mhz os tempos são 56,75 microssegundos e 17,75 microssegundos, respectivamente.
; Definições:
;   Registradores
.DEF rmp = R16 ; usado como registrador multipropósito
;   tipo de AVR: Testado em AT90S8515, necessário para pilha, as rotinas devem funcionar com outros tipos AT90S-.
.NOLIST
.INCLUDE "8515def.inc"
.LIST
; Início do programa de teste
; Escreve um número em R1 e inicia a conversão da rotina, para propósitos de teste somente
.CSEG
.ORG $0000
        rjmp main
main:
        ldi rmp,HIGH(RAMEND) ; Inicia a pilha
        out SPH,rmp
        ldi rmp,LOW(RAMEND)
        out SPL,rmp
        ldi rmp,$FF ; Converte $FF
        mov R1,rmp
        rcall fpconv8 ; chama a rotina de conversão
no_end:   ; loop infinito, quando terminar
        rjmp no_end
; Rotina de conversão, chama diferentes passos da conversão
```

```
fpconv8:
	rcall fpconv8m ; multiplica pot 502
	rcall fpconv8r ; arredonda e divide por 256
	rcall fpconv8a ; converte para string ASCII
	ldi rmp,'.' ; seta caractere decimal
	mov R6,rmp
	ret ; tudo pronto
; Subrotina de multiplicação por 502
fpconv8m:
	clr R4 ; seta o multiplicador como 502
	ldi rmp,$01
	mov R3,rmp
	ldi rmp,$F6
	mov R2,rmp
	clr R7 ; zera o resultado
	clr R6
	clr R5
fpconv8m1:
	or R1,R1 ; checa se o número é zero
	brne fpconv8m2 ; ainda tem um, continuamos com a versão
	ret ; pronto, retorne
fpconv8m2:
	lsr R1 ; desloca número à direita (divide por 2)
	brcc fpconv8m3 ; se o número mais baixo era 0, pula adição
	add R5,R2 ; some o número em R6:R5:R4:R3 ao resultado
	adc R6,R3
	adc R7,R4
fpconv8m3:
	lsl R2 ; multiplica R4:R3:R2 por 2
	rol R3
	rol R4
	rjmp fpconv8m1 ; repete para o próximo bit
; Arredonda o valor em R7:R6 com o valor do bit 7 de R5
fpconv8r:
	clr rmp ; coloca zero em rmp
	lsl R5 ; rotaciona bit 7 para o carry
	adc R6,rmp ; soma LSB com carry
	adc R7,rmp ; soma MSB com carry
	mov R2,R7 ; copia o valor para R2:R1 (divide por 256)
	mov R1,R6
	ret
; Converte a palavra em R2:R1 para uma string ASCII em R5:R6:R7:R8
fpconv8a:
	clr R4 ; Seta o valor do divisor decimal para 100
	ldi rmp,100
	mov R3,rmp
	rcall fpconv8d ; obtém o dígito ASCII por subtrações repetidas
	mov R5,rmp ; seta caractere como sendo da centena
	ldi rmp,10 ; Seta o valor do divisor decimal para 10
	mov R3,rmp
	rcall fpconv8d ; obtém o próximo dígito ASCII
	mov R7,rmp ; seta caractere como sendo da dezena
	ldi rmp,'0' ; converte o resto para dígito ASCII
	add rmp,R1
	mov R8,rmp ; seta caractere como sendo da unidade
	ret
; Converte palavra binária em R2:R1 para dígito decimal subtraindo o valor do divisor decimal em R4:R3 (100,10)
fpconv8d:
	ldi rmp,'0' ; inicia com o valor decimal '0'
fpconv8d1:
	cp R1,R3 ; Compara palabra com o valor do divisor decimal
	cpc R2,R4
	brcc fpconv8d2 ; zera carry, subtrai valor do divisor
	ret ; subtração feita
fpconv8d2:
	sub R1,R3 ; subtrai valor do divisor
	sbc R2,R4
	inc rmp ; soma um
	rjmp fpconv8d1 ; de novo
; Fim da rotina de conversão
```

## Exemplo 2: Conversor AD de 10 bits com saída decimal fixa

Este exemplo é um pouco mais complicado. Veja o website se precisar dele.

# Anexo

## Instruções ordenadas por função

Para as abreviações usadas, veja a lista de abreviações.

| *Função* | *Sub função* | *Comando* | *Flags* | *Clk* |
|---|---|---|---|---|
| Registrador | 0 | CLR r1 | Z N V | 1 |
| | 255 | SER rh | | 1 |
| | Constante | LDI rh,c255 | | 1 |
| Cópia | Registrador => Registrador | MOV r1,r2 | | 1 |
| | SRAM => Registrador, direto | LDS r1,c65535 | | 2 |
| | SRAM => Registrador | LD r1,rp | | 2 |
| | SRAM => Registrador e INC | LD r1,rp+ | | 2 |
| | DEC, SRAM => Registrador | LD r1,-rp | | 2 |
| | SRAM, deslocado => Registrador | LDD r1,ry+k63 | | 2 |
| | Porta => Registrador | IN r1,p1 | | 1 |
| | Pilha => Registrador | POP r1 | | 2 |
| | Program storage Z => R0 | LPM | | 3 |
| | Registrador => SRAM, direto | STS c65535,r1 | | 2 |
| | Registrador => SRAM | ST rp,r1 | | 2 |
| | Registrador => SRAM e INC | ST rp+,r1 | | 2 |
| | DEC, Registrador => SRAM | ST -rp,r1 | | 2 |
| | Registrador => SRAM, deslocado | STD ry+k63,r1 | | 2 |
| | Registrador => Porta | OUT p1,r1 | | 1 |
| | Registrador => Pilha | PUSH r1 | | 2 |
| Soma | 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, constante | ADIW rd,k63 | Z C N V S | 2 |
| Subtração | 8 Bit, -1 | DEC r1 | Z N V | 1 |
| | 8 Bit | SUB r1,r2 | Z C N V H | 1 |
| | 8 Bit, constante | SUBI rh,c255 | Z C N V H | 1 |
| | 8 Bit - carry | SBC r1,r2 | Z C N V H | 1 |
| | 8 Bit - carry, constante | SBCI rh,c255 | Z C N V H | 1 |
| | 16 Bit | SBIW rd,k63 | Z C N V S | 2 |
| Desvio | Lógico, esquerda | LSL r1 | Z C N V | 1 |
| | Lógico, direita | LSR r1 | Z C N V | 1 |
| | Rotação, esquerda sobre carry | ROL r1 | Z C N V | 1 |
| | Rotação, direita sobre carry | ROR r1 | Z C N V | 1 |
| | Aritmética, direita | ASR r1 | Z C N V | 1 |
| | Troca de nibbles | SWAP r1 | | 1 |
| Binários | E (And) | AND r1,r2 | Z N V | 1 |
| | E, constante | ANDI rh,c255 | Z N V | 1 |
| | Ou (Or) | OR r1,r2 | Z N V | 1 |
| | Ou, constante | ORI rh,c255 | Z N V | 1 |
| | Ou-Exclusivo (XOR) | EOR r1,r2 | Z N V | 1 |
| | Complemento de Um | COM r1 | Z C N V | 1 |
| | Complemento de Dois | NEG r1 | Z C N V H | 1 |

| *Função* | *Sub função* | *Comando* | *Flags* | *Clk* |
|---|---|---|---|---|
| Troca de Bits | Registrador, seta | SBR rh,c255 | Z N V | 1 |
| | Registrador, zera | CBR rh,255 | Z N V | 1 |
| | Registrador, copia para Flag T | BST r1,b7 | T | 1 |
| | Registrador, copia da Flag T | BLD r1,b7 | | 1 |
| | Porta, seta | SBI pl,b7 | | 2 |
| | Porta, zera | CBI pl,b7 | | 2 |
| Seta bit de Status | Flag de Zero | SEZ | Z | 1 |
| | Flag de carry | SEC | C | 1 |
| | Flag de Negativo | SEN | N | 1 |
| | Flag de carry de Complemento de Dois | SEV | V | 1 |
| | Flag de meio carry | SEH | H | 1 |
| | Flag de Sinal | SES | S | 1 |
| | Flag de Transferência | SET | T | 1 |
| | Flag de Habilitação de Interrupção | SEI | I | 1 |
| zera bit de status | Flag de Zero | CLZ | Z | 1 |
| | Flag de carry | CLC | C | 1 |
| | Flag de Negativo | CLN | N | 1 |
| | Flag de carry de Complemento de Dois | CLV | V | 1 |
| | Flag de meio carry | CLH | H | 1 |
| | Flag de Sinal | CLS | S | 1 |
| | Flag de Transferência | CLT | T | 1 |
| | Flag de Desabilitação de Interrupção | CLI | I | 1 |
| Compara | Registrador, Registrador | CP r1,r2 | Z C N V H | 1 |
| | Registrador, Registrador + carry | CPC r1,r2 | Z C N V H | 1 |
| | Registrador, constante | CPI rh,c255 | Z C N V H | 1 |
| | Registrador, ≤0 | TST r1 | Z N V | 1 |
| Salto Imediato | Relativo | RJMP c4096 | | 2 |
| | Indireto, Endereço em Z | IJMP | | 2 |
| | Subrotina, relativo | RCALL c4096 | | 3 |
| | Subrotina, Endereço em Z | ICALL | | 3 |
| | Retorna da Subrotina | RET | | 4 |
| | Retorna da Interrupção | RETI | I | 4 |

| Função | Sub função | Comando | Flags | Clk |
|---|---|---|---|---|
| Salto Condicional | Bit de status setado | BRBS b7,c127 | | 1/2 |
| | Bit de status zera | BRBC b7,c127 | | 1/2 |
| | Salta se igual | BREQ c127 | | 1/2 |
| | Salta se diferente | BRNE c127 | | 1/2 |
| | Salta se carry setado | BRCS c127 | | 1/2 |
| | Salta se carry zero | BRCC c127 | | 1/2 |
| | Salta se igual ou maior | BRSH c127 | | 1/2 |
| | Salta se menor | BRLO c127 | | 1/2 |
| | Salta se negativo | BRMI c127 | | 1/2 |
| | Salta se positivo | BRPL c127 | | 1/2 |
| | Salta se igual ou maior (com sinal) | BRGE c127 | | 1/2 |
| | Salta se menor que zero (com sinal) | BRLT c127 | | 1/2 |
| | Salta se meio carry setado | BRHS c127 | | 1/2 |
| | Salta se meio carry zero | BRHC c127 | | 1/2 |
| | Salta se Flag T setada | BRTS c127 | | 1/2 |
| | Salta se Flag T zera | BRTC c127 | | 1/2 |
| | Salta se flag de complemento de 2 setado | BRVS c127 | | 1/2 |
| | Salta se flag de complemento de 2 zero | BRVC c127 | | 1/2 |
| | Salta se interrupção habilitada | BRIE c127 | | 1/2 |
| | Salta se interrupção desabilitada | BRID c127 | | 1/2 |
| Salto condcionado | Bit do registrador=0 | SBRC r1,b7 | | 1/2/3 |
| | Bit do registrador=1 | SBRS r1,b7 | | 1/2/3 |
| | Bit da porta=0 | SBIC pl,b7 | | 1/2/3 |
| | Bit da porta=1 | SBIS pl,b7 | | 1/2/3 |
| | Compara, salta se for igual | CPSE r1,r2 | | 1/2/3 |
| Outros | Sem Operação | NOP | | 1 |
| | Repouso | SLEEP | | 1 |
| | Watchdog Reset | WDR | | 1 |

# Lista de Diretivas e Instruções em ordem alfabética

## Diretivas de Assembler em ordem alfabética

| Diretiva | ... significa ... |
|---|---|
| .CSEG | *Compilar para o segmento de código* |
| .DB | *Inserir bytes de dados* |
| .DEF | *Definir o nome de um registrador* |
| .DW | *Insirir palavras de dados* |
| .ENDMACRO | *A macro está completa, parar a gravação* |
| .ESEG | *Compilar para o segmento da EEPROM* |
| .EQU | *Definir uma constante por nome e setar o seu valor* |
| .INCLUDE | *Inserir o conteúdo de um arquivo neste local como se fosse parte deste arquivo* |
| .MACRO | *Iniciar a gravação das seguintes instruções como definições de macro* |
| .ORG | *Setar o endereço de saída do assembler para o seguinte número* |

## Instruções em ordem alfabética

| Instrução | ... faz ... |
|---|---|
| ADC r1,r2 | *Soma r2 com carry com r1, e armazena o resultado em r1* |
| ADD r1,r2 | *Soma r2 com r1 e armazena o resultado em r1* |
| ADIW rd,k63 | *Soma a palavra constante imediata k63 ao registrador duplo rd+1:rd (rd = R24, R26, R28, R30)* |

| | |
|---|---|
| AND r1,r2 | Faz operação and de r1 com o valor em r2 e armazena o resultado em r1 |
| ANDI rh,c255 | Faz operação and com o registrador superior rh com a constante c255 e armazena o resultado em rh |
| ASR r1 | Desvio à direita aritmético do registrador r1 |
| BLD r1,b7 | Copia a flag T do registrador de status para o bit b7 do registrador r1 |
| BRCC c127 | Desvia ou volta para instruções c127 se a flag de carry for 0 |
| BRCS c127 | Desvia ou volta para instruções c127 se a flag de carry for 1 |
| BREQ c127 | Desvia ou volta para instruções c127 se a flag de zero for 1 |
| BRGE c127 | Desvia ou volta para instruções c127 se a flag de carry for 0 |
| BRHC c127 | Desvia ou volta para instruções c127 se a flag de meio-carry for 0 |
| BRHS c127 | Desvia ou volta para instruções c127 se a flag de meio-carry for 1 |
| BRID c127 | Desvia ou volta para instruções c127 se a flag de interrupção for 0 |
| BRIE c127 | Desvia ou volta para instruções c127 se a flag de interrupção for 1 |
| BRLO c127 | Desvia ou volta para instruções c127 se a flag de carry for 1 |
| BRLT c127 | Desvia ou volta para instruções c127 se as flags de negativo e estouro forem 1 |
| BRMI c127 | Desvia ou volta para instruções c127 se a flag de negativo for 1 |
| BRNE c127 | Desvia ou volta para instruções c127 se a flag de zero for 1 |
| BRPL c127 | Desvia ou volta para instruções c127 se a flag de negativo for 0 |
| BRSH c127 | Desvia ou volta para instruções c127 se a flag de carry for 0 |
| BRTC c127 | Desvia ou volta para instruções c127 se a flag de transferência for 0 |
| BRTS c127 | Desvia ou volta para instruções c127 se a flag de transferência for 1 |
| BRVC c127 | Desvia ou volta para instruções c127 se a flag de estouro for 0 |
| BRVS c127 | Desvia ou volta para instruções c127 se a flag de estouro for 1 |
| BST r1,b7 | Copia o bit 7 no registrador r1 para a flag de transferência no registrador de status |
| CBI pl,b7 | Zera bit b7 na porta mais baixa pl |
| CBR rh,k255 | Zera todos os bits no registrador superior rh, que estão setados na constante k255 (máscara) |
| CLC | Zera o bit de carry |
| CLH | Zera o bit de meio-carry |
| CLI | Zera o bit de interrupção, desabilita a execução de interrupções |
| CLN | Zera o bit de negativo no registrador de status |
| CLR r1 | Zera o registrador r1 |
| CLS | Zera a flag de sinal no registrador de status |
| CLT | Zera a flag de transferência no registrador de status |
| CLV | Zera a flag de overflow do registrador de status |
| CLZ | Zera a flag de zero do registrador de status |
| COM r1 | Complementa registrador r1 (complemento de um) |
| CP r1,r2 | Compara registrador r1 com registrador r2 |
| CPC r1,r2 | Compara registrador r1 com registrador r2 e flag de carry |
| CPI rh,c255 | Compara o registrador superior rh com a constante imediata c255 |
| CPSE r1,r2 | Compara r1 com r2 e pula sobre a próxima instrução se for igual |
| DEC r1 | Decrementa o registrador r1 por 1 |
| EOR r1,r2 | Ou-exclusive dos bits do registrador r1 com registrador r2 e armazena resultado em r1 |
| ICALL | Chama a subrotina no endereço no par de registradores Z (ZH:ZL, R31:R30) |
| IJMP IN r1,p1 | Salta para o endereço no par de registradores Z (ZH:ZL, R31:R30) |
| INC r1 | Incrementa o registrador r1 por 1 |
| LD r1,(rp,rp+,-rp) | Carrega o registrador r1 com o conteúdo localizado no endereço apontado pelo par de registradores rp (X, Y ou Z) (rp+ incrementa o par de registradores depois da carga, -rp decrementa o par antes da carga) |
| LDD r1,ry+k63 | Carrega o registrador r1 com o conteúdo localizado no endereço apontado pelo par de registradores ry (Y ou Z), deslocado pela constante k63 |
| LDI rh,c255 | Carrega o registrador superior rh com a constante c255 |
| LDS r1,c65535 | Carrega o registrador r1 com o conteúdo do endereço c65535 |
| LPM<br>LPM r1<br>LPM r1,Z+<br>LPM r1,-Z | Carrega o registrador R0 com o conteúdo da memory flash localizado no endereço apontado pelo par de registradores Z (ZH:ZL, R31:R30), dividido por 2, bit 0 em Z aponta para o byte inferior (0) ou superior (1) no flash (Carrega registrador r1, Z+ incrementa Z depois da carga, -Z decrementa Z antes da carga). |
| LSL r1 | Desvio à esquerda lógico do registrador r1 |
| LSR r1 | Desvio à direita lógico do registrador r1 |

| | |
|---|---|
| MOV r1,r2 | Move registrador r2 para o registrador r1 |
| NEG r1 | Subtrai registrador r1 de Zero |
| NOP | Sem operação |
| OR r1,r2 | Faz operação OR do registrador r1 com r2 e armazena o resultado em r1 |
| ORI rh,c255 | Faz operação OR do registrador superior r1 com a constante c255 |
| OUT p1,r1 | Copia registrador r1 para porta de E/S p1 |
| POP r1 | Incrementa o ponteiro de pilha e retira o último byte da pilha e o coloca no registrador r1 |
| PUSH r1 | Coloca o conteúdo do registrador r1 na pilha e decrementa o ponteiro de pilha |
| RCALL c4096 | Coloca o contador de programa na pilha e adiciona a constante com sinal c4096 ao contador de programa (chamada relativa) |
| RET | Recupera o contador de programa da pilha (retorna ao endereço de chamada) |
| RETI | Habilita interrupções e recupera o contador de programa da pilha (retorna de interrupção) |
| RJMP c4096 | Salto relativo, soma constante com sinal c4096 ao endereço do programa |
| ROL r1 | Rotaciona registrador r1 à esquerda, copia a flag de carry para o bit 0 |
| ROR r1 | Rotaciona o registrador r1 à direita, copia a flag de carry para o bit 7 |
| SBC r1,r2 | Subtrai r2 e a flag de carry do registrador r1 e armazena o resultado em r1 |
| SBCI rh,c255 | Subtrai constante c255 e carry do registrador superior rh e armazena o resultado em rh |
| SBI pl,b7 | Seta o bit b7 na porta baixa pl |
| SBIC pl,b7 | Se o bit b7 na porta baixa pl for zero, pula a próxima instrução |
| SBIS pl,b7 | Se o bit b7 na porta baixa pl for um, pula a próxima instrução |
| SBIW rd,k63 | Subtrai a constante k63 do par de registradores rd (rd+1:rd, rd = R24, R26, R28, R30) |
| SBR rh,c255 | Seta os bits no registrador superior rh, que são 1 da constante c255 |
| SBRC r1,b7 | Se o bit b7 do registrador r1 for zero, pula a próxima instrução |
| SBRS r1,b7 | Se o bit b7 no registrador r1 for um, pula a próxima instrução |
| SEC | Seta flag de carry no registrador de status |
| SEH | Seta flag de meio carry no registrador de status |
| SEI | Seta flag de interrupção no registrador de status, habilita a execução de interrupção |
| SEN | Seta flag de negativo no registrador de status |
| SER rh | Seta todos os bits no registrador superior rh |
| SES | Seta flag de sinal no registrador de status |
| SET | Seta flag de transferência no registrador de status |
| SEV | Seta flag de estouro (overflow) no registrador de status |
| SEZ | Seta flag de zero no registrador de status |
| SLEEP | Coloca o controlador no modo de repouso selecionado |
| ST (rp/rp+/-rp),r1 | Armazena o conteúdo do registrador r1 na localização da memória indicada pelo par de registradores rp (rp = X, Y, Z; rp+: incrementa o par de registradores depois do armazenamento; -rp: decrementa o par de registradores antes do armazenamento) |
| STD ry+k63,r1 | Armazena o conteúdo do registrador r1 na localização apontada pelo par de registradores ry (Y ou z), deslocado pela constante k63. |
| STS c65535,r1 | Armazena o conteúdo do registrador r1 na localização c65535 |
| SUB r1,r2 | Subtrai o registrador r2 do registrador r1 e escreve o resultado em r1 |
| SUBI rh,c255 | Subtrai a constante c255 do registrador superior rh |
| SWAP r1 | Troca os nibbles superior com o inferior no registrador r1 |
| TST r1 | Compara o registrador r1 com Zero |
| WDR | Watchdog reset |

# Detalhes das Portas

Esta tabela contém as portas importantes nos AVR tipo AT90S2313, 2323 e 8515. Portas ou pares de registradores acessíveis não são mostradas em detalhes. Não há garantias da exatidão destes dados, confira os data sheets originais!

## Registrador de Status, Flags do Acumulador

| Porta | Função | Endereço da porta | Endereço da RAM |
|---|---|---|---|
| SREG | Acumulador do registrador de status | 0x3F | 0x5F |

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

| Bit | Nome | Significado | Indica | Comando |
|---|---|---|---|---|
| 7 | I | Flag de Interrupção Global | 0: Interrupções desabilitadas | CLI |
| | | | 1: Interrupções habilitadas | SEI |
| 6 | T | Armazém de bits | 0: O bit armazenado é 0 | CLT |
| | | | 1: O bit armazenado é 1 | SET |
| 5 | H | Flag de Meio-Carry | 0: Não ocorreu meio-carry | CLH |
| | | | 1: Ocorreu meio-carry | SEH |
| 4 | S | Flag de Sinal | 0: Sinal positivo | CLS |
| | | | 1: Sinal negativo | SES |
| 3 | V | Flag de complemento de dois | 0: Não ocorreu transporte (carry) | CLV |
| | | | 1: Ocorreu transporte (carry) | SEV |
| 2 | N | Flag de Negativo | 0: O resultado não era negativo/menor | CLN |
| | | | 1: O resultado era negativo/menor | SEN |
| 1 | Z | Flag de Zero | 0: Resultado não era zero/diferente | CLZ |
| | | | 1: Resultado era zero/igual | SEZ |
| 0 | C | Flag de Carry | 0: Não ocorreu transporte (carry) | CLC |
| | | | 1: Ocorreu transporte (carry) | SEC |

## Ponteiro de pilha

| Porta | Função | Endereço da porta | Endereço da RAM |
|---|---|---|---|
| SPL/SPH | Ponteiro de pilha | 003D/0x3E | 0x5D/0x5E |

| Nome | Significado | Disponibilidade |
|---|---|---|
| SPL | Byte baixo do ponteiro de pilha | Do AT90S2313 para frente, não em 1200 |
| SPH | Byte alto do ponteiro de pilha | Do AT90S8515 para frente, apenas para dispositivos com mais de 256 bytes de SRAM interna. |

## SRAM e controle externo de interrupções

| Porta | Função | Endereço da porta | Endereço da RAM |
|---|---|---|---|
| MCUCR | Registrador de Controle Geral MCU | 0x35 | 0x55 |

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

| Bit | Nome | Significado | Indica |
|---|---|---|---|
| 7 | SRE | Habilita SRAM Externa | 0=Não há SRAM externa conectada |
| | | | 1=Há SRAM externa conectada |

| Bit | Nome | Significado | Indica |
|---|---|---|---|
| 6 | SRW | Estados de espera (wait states) da SRAM Ext. | 0=Sem wait state extra na SRAM externa |
| | | | 1=Wait state adicional na SRAM externa |
| 5 | SE | Habilita repouso (sleep) | 0=Ignora comando SLEEP |
| | | | 1=Aceita comando SLEEP |
| 4 | SM | Modo de respouso | 0=Inatividade (Meio-repouso) |
| | | | 1=Desligamento (Repouso total) |
| 3 | ISC11 | Pino de controle de interrupção INT1 (conectado a GIMSK) | 00: Nível baixo inicia interrupção |
| 2 | ISC10 | | 01: Indefinido |
| | | | 10: Borda de descida dispara interrupção |
| | | | 11: Borda de subida dispara interrupção |
| 1 | ISC01 | Pino de controle de interrupção INT0 (conectado a GIMSK) | 00: Nível baixo inicia interrupção |
| 0 | ISC00 | | 01: Indefinido |
| | | | 10: Borda de descida dispara interrupção |
| | | | 11: Borda de subida dispara interrupção |

## Controle de Interrupção Externa

| Porta | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| GIMSK | Registrador mascarado de Interrupção Geral | 0x3B | 0x5B |

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

| Bit | Nome | Significado | Indica |
|---|---|---|---|
| 7 | INT1 | Interrupção por pino externo INT1 (conectado a MCUCR) | 0: INT1 Externa desabilitada |
| | | | 1: INT1 Externa habilitada |
| 6 | INT0 | Interrupção por pino externo INT0 (conectado a MCUCR) | 0: INT0 Externa desabilitada |
| | | | 1: INT0 Externa habilitada |
| 0...5 | (Não usados) | | |

| Porta | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| GIFR | Registrador geral de Interrupções | 0x3A | 0x5A |

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

| Bit | Name | Significado | Indica |
|---|---|---|---|
| 7 | INTF1 | Ocorreu interrupção externa pelo pino INT1 | Zera o bit pela execução da rotina de tratamento ou por comando |
| 6 | INTF0 | Ocorreu interrupção externa pelo pino INT0 | |
| 0...5 | (Não usados) | | |

## Controle do Timer de Interrupção

| *Port* | *Função* | *Endereço da Porta* | *Endereço da RAM* |
|---|---|---|---|
| TIMSK | Registrador mascarado do timer de interrupção | 0x39 | 0x59 |

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

| Bit | Nome | Significado | Indica |
|---|---|---|---|
| 7 | TOIE1 | Estouro de Timer/Contador 1-Interrupção | 0: Sem int no estouro |
| | | | 1: Int no estouro |
| 6 | OCIE1A | Interrup. do Timer/Contador 1 Comparador A | 0: Sem int igual a A |
| | | | 1: Int igual a A |
| 5 | OCIE1B | Interrup. do Timer/Contador 1 Comparador B | 0: Sem int em B |
| | | | 1: Int em B |
| 4 | (Não usado) | | |
| 3 | TICIE1 | Captura do Timer/Contador 1 Interrupção | 0: Sem int na captura |
| | | | 1: Int na captura |
| 2 | (Não usado) | | |
| 1 | TOIE0 | Estouro de Timer/Contador 0-Interrupção | 0: Sem int no estouro |
| | | | 1: Int no estouro |
| 0 | (Não usado) | | |

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| TIFR | Registrador de timer de interrupção | 0x38 | 0x58 |

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

| Bit | Nome | Significado | Indica |
|---|---|---|---|
| 7 | TOV1 | Timer/Contador 1 estourou | Modo-Interrupção:<br>Zero pela execução da rotina da interrupção<br><br>OU<br><br>Zero por comando |
| 6 | OCF1A | Timer/Contador 1 chegou a Comparaç A | |
| 5 | OCF1B | Timer/Contador 1 chegou a Comparaç B | |
| 4 | (Não usado) | | |
| 3 | ICF1 | Evento de captura de Timer/Contador 1 | |
| 2 | (Não usado) | | |
| 1 | TOV0 | Timer/Contador 0 estourou | |
| 0 | (Não usado) | | |

## Timer/Contador 0

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| TCCR0 | Registrador de controle de Timer/ Contador 0 | 0x33 | 0x53 |

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

| Bit | Nome | Significado | Indica |
|---|---|---|---|
| 2..0 | CS02..CS00 | Timer Clock | 000: Pára timer |
| | | | 001: Clock = clock do chip |
| | | | 010: Clock = clock do chip / 8 |
| | | | 011: Clock = clock do chip / 64 |
| | | | 100: Clock = clock do chip / 256 |
| | | | 101: Clock = clock do chip / 1024 |
| | | | 110: Clock = borda de descida do pino T0 |
| | | | 111: Clock = borda de subida do pino T0 |
| 3..7 | (Não usado) | | |

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| TCNT0 | Registrador de Controle de Timer/Contador 0 | 0x32 | 0x52 |

## Timer/Contador 1

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| TCCR1A | Registrador de Controle A de Timer/ Contador 1 | 0x2F | 0x4F |

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

| Bit | Nome | Significado | Indica |
|---|---|---|---|
| 7 | COM1A1 | Saída Comparadora A | 00: OC1A/B não conectado<br>01: OC1A/B troca polaridade<br>10: OC1A/B para zero<br>11: OC1A/B para um |
| 6 | COM1A0 | | |
| 5 | COM1B1 | Saída Comparadora B | |
| 4 | COM1B0 | | |
| 3 | (Não usado) | | |
| 2 | | | |
| 1..0 | PWM11<br>PWM10 | Modulador por largura de pulso (PWM) | 00: PWM desligado<br>01: 8-Bit PWM<br>10: 9-Bit PWM<br>11: 10-Bit PWM |

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| TCCR1B | Registrador de Controle B de Timer/ Contador 1 | 0x2E | 0x4E |

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

| Bit | Nome | Significado | Indica |
|---|---|---|---|
| 7 | ICNC1 | Cancelador de ruído no pino ICP | 0: desabilitado, primeira borda inicia a amostragem |
| | | | 1: habilitado, pelo menos quatro ciclos |
| 6 | ICES1 | Seleção de bordo na captura | 0: borda de descida dispara captura |
| | | | 1: borda de subida dispara captura |
| 5..4 | (Não usado) | | |
| 3 | CTC1 | Limpa quando a comparação for igual a A | 1: Contador zera se for igual |
| 2..0 | CS12..CS10 | Seleção do clock | 000: Contador parado<br>001: Clock<br>010: Clock / 8<br>011: Clock / 64<br>100: Clock / 256<br>101: Clock / 1024<br>110: borda de descida em pino T1<br>111: borda de subida em pino T1 |

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| TCNT1L/H | Registrador Timer/Contador 1 | 0x2C/0x2D | 0x4C/0x4D |

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| OCR1AL/H | Timer/Contador 1 Output Compare register A | 0x2A/0x2B | 0x4A/0x4B hex |

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| OCR1BL/H | Registrador de Saída do Comparador B Timer/Contador 1 | 0x28/0x29 | 0x48/0x49 |

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| ICR1L/H | Registrador de Entrada de Captura do Timer/Contador 1 | 0x24/0x25 | 0x44/0x45 |

## Watchdog-Timer

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| WDTCR | Registrador de Controle do Watchdog Timer | 0x21 | 0x41 |

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

| Bit | Nome | Significado | Ciclo-WDT a 5,0 Volts |
|---|---|---|---|
| 7..5 | (Não usado) | | |
| 4 | WDTOE | Habilita ciclo de Watchdog | Requerido setar antes de desabilitar o WDE |
| 3 | WDE | Habilita Watchdog | 1: Watchdog ativo |
| 2..0 | WDP2..WDP0 | Tempo do Watchdog | 000: 15 ms<br>001: 30 ms<br>010: 60 ms<br>011: 120 ms<br>100: 240 ms<br>101: 490 ms<br>110: 970 ms<br>111: 1,9 s |

## EEPROM

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| EEARL/H | Registrador de Endereço da EEPROM | 0x1E/0x1F | 0x3E/0x3F |

EEARH apenas nos tipo com EEPROM com mais de 256 Bytes (do AT90S8515 para frente)

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| EEDR | Registrador de Dados da EEPROM | 0x1D | 0x3D |

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| EECR | Registrador de Controle da EEPROM | 0x1C | 0x3C |

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

| Bit | Nome | Significado | Função |
|---|---|---|---|
| 7..3 | (Não usado) | | |
| 2 | EEMWE | Habilita Escrita Master EEPROM | Habilita ciclo de leitura |
| 1 | EEWE | Habilita Escrita EEPROM | Setar para iniciar gravação |
| 0 | EERE | Habilita Leitura EEPROM | Setar para iniciar leitura |

## Interface de Periféricos Seriais (SPI)

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| SPCR | Registrador de Controle SPI | 0x0D | 0x2D |

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

| Bit | Nome | Significado | Função |
|---|---|---|---|
| 7 | SPIE | Interrupções SPI | 0: Interrupções desabilitadas |
| | | | 1: Interrupções habilitadas |
| 6 | SPE | Habilita SPI | 0: SPI desabilitado |
| | | | 1: SPI habilitado |
| 5 | DORD | Ordem de Dados | 0: MSB primeiro |
| | | | 1: LSB primeiro |
| 4 | MSTR | Seleção Master/Slave | 0: Slave |
| | | | 1: Master |
| 3 | CPOL | Polaridade do Clock | 0: Fase positiva do Clock |
| | | | 1: Fase negativa do Clock |
| 2 | CPHA | Fase do Clock | 0: Amostragem no início da fase do clock |
| | | | 1: Amostragem no fim da fase do clock |
| 1 | SPR1 | Freqüência do clock SCK | 00: Clock / 4 |
| 0 | SPR0 | | 01: Clock / 16 |
| | | | 10: Clock / 64 |
| | | | 11: Clock / 128 |

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| SPSR | Registrador de Status SPI | 0x0E | 0x2E |

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

| Bit | Nome | Significado | Indica |
|---|---|---|---|
| 7 | SPIF | Flag de Interrupção SPI | Requisita Interrupção |
| 6 | WCOL | Flag de Colisão de Escrita | Ocorreu colisão na escrita |
| 5..0 | (Não usado) | | |

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| SPDR | Registrador de Dados da SPI | 0x0F | 0x2F |

# UART

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| UDR | Registrador de Dados da E/S UART | 0x0C | 0x2C |

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| USR | Registrador de Status da UART | 0x0B | 0x2B |

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

| Bit | Nome | Significado | Função |
|---|---|---|---|
| 7 | RXC | Recepção UART Completa | 1: Caractere recebido |
| 6 | TXC | Transmissão UART Completa | 1: Shift register vazio |
| 5 | UDRE | Registrador de Dados da UART Vazio | 1: Registrador de transmissão disponível |
| 4 | FE | Erro de enquadramento (framing) | 1: Bit de Parada Ilegal |
| 3 | OR | Perda (Overrun) | 1: Caractere perdido |
| 2..0 | (Não usado) | | |

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| UCR | Registrador de Controle da UART | 0x0A | 0x2A |

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

| Bit | Nome | Significado | Função |
|---|---|---|---|
| 7 | RXCIE | Habilita Interrupção quando RX terminada | 1: Interrupção quando recebe caractere |
| 6 | TXCIE | Habilita Interrupção quando TX completo | 1: Interrupção quando transmissão completada |
| 5 | UDRIE | Habilita Interrupção quando Registrador de Dados vazio | 1: Interrupção quando buffer de dados vazio |
| 4 | RXEN | Recepção Habilitada | 1: Receptor habilitado |
| 3 | TXEN | Transmissão Habilitada | 1: Transmissor habilitado |
| 2 | CHR9 | Caracteres de 9 bits | 1: Tamanho do caractere 9 bits |
| 1 | RXB8 | Recebe Bit de Dado 8 | (segura nono bit na recepção) |
| 0 | TXB8 | Transmite Bit de Dado 8 | (escreve nono bit para transmissão) |

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| UBRR | Registrador de velocidade da UART (Baud Rate) | 0x09 | 0x29 |

## Analog Comparator

| Port | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|
| ACSR | Registrador de Status e de Controle do Comparador Analógico | 0x08 | 0x28 |

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

| Bit | Nome | Significado | Função |
|---|---|---|---|
| 7 | ACD | Desabilita | Desabilita Comparadores |
| 6 | (Não usado) | | |
| 5 | ACO | Saída do Comparador | Saída dos Comparadores |
| 4 | ACI | Flag de Interrupção | 1: Interrupção requisitada |
| 3 | ACIE | Habilita Interrupção | 1: Interrupção habilitada |
| 2 | ACIC | Habilita Captura de entrada | 1: Conecta a Captura do Timer1 |
| 1 | ACIS1 | Habilita Captura de Entrada | 00: Interrupção na mudança de nível |
| 0 | ACIS0 | | 01: (Não usado) |
| | | | 10: Interrupção em borda de descida |
| | | | 11: Interrupção em borda de subida |

## Portas E/S

| Port | Register | Função | Endereço da Porta | Endereço da RAM |
|---|---|---|---|---|
| A | PORTA | Registrador de Dados | 0x1B | 0x3B |
| | DDRA | Registrador da Direção dos Dados | 0x1A | 0x3A |
| | PINA | Endereço dos Pinos de Entrada | 0x19 | 0x39 |
| B | PORTB | Registrador de Dados | 0x18 | 0x38 |
| | DDRB | Registrador da Direção dos Dados | 0x17 | 0x37 |
| | PINB | Endereço dos Pinos de Entrada | 0x16 | 0x36 |
| C | PORTC | Registrador de Dados | 0x15 | 0x35 |
| | DDRC | Registrador da Direção dos Dados | 0x14 | 0x34 |
| | PINC | Endereço dos Pinos de Entrada | 0x13 | 0x33 |
| D | PORTD | Registrador de Dados | 0x12 | 0x32 |
| | DDRD | Registrador da Direção dos Dados | 0x11 | 0x31 |
| | PIND | Endereço dos Pinos de Entrada | 0x10 | 0x30 |

# Portas, ordem alfabética

*ACSR, Registrador de Status e Controle do Comparador Analógico*
*DDRx, Registrador da Direção dos Dados da Porta x*
*EEAR, Registrador de endereço da EEPROM*
*EECR, Registrador de controle da EEPROM*
*EEDR, Registrador de Dados da EEPROM*
*GIFR, Registrador da Flag de Interrupção Geral*
*GIMSK, Registrador de Interrupção Mascarada Geral*
*ICR1L/H, Registrador de Captura de Entrada 1*
*MCUCR, Registrador de Controle Geral da MCU*
*OCR1A, Registrador de Saída do Comparador 1 A*
*OCR1B, Registrador de Saída do Comparador 1 B*
*PINx, Acesso à Porta de Entrada*
*PORTx, Port x Output Register*
*SPL/SPH, Ponteiro de Pilha*
*SPCR, Registrador de Controle de Periféricos Seriais*
*SPDR, Registrador de Dados de Periféricos Seriais*
*SPSR, Registrador de Status de Periféricos Seriais*
*SREG, Registrador de Status*
*TCCR0, Registrador de Controle do Timer/Contador Timer0*
*TCCR1A, Registrador de Controle do Timer/Contador 1 A*
*TCCR1B, Registrador de Controle do Timer/Contador 1 B*
*TCNT0, Registrador do Timer/Contador, Contador 0*
*TCNT1, Registrador do Timer/Contador, Contador 1*
*TIFR, Flag de Interrupção do Timer*
*TIMSK, Registrador Mascarado de Interrupção do Timer*
*UBRR, Registrador de velocidade da UART (Baud Rate)*
*UCR, Registrador de Controle da UART*
*UDR, Registrador de Dados da UART*
*WDTCR, Registrador de Controle do Timer Watchdog*

# Lista de abreviações

As abreviações usadas foram escolhidas para incluírem a faixa de valores. Pares de registradores são nomeadas pelo menor dos dois registradores. Constantes em comandos de salto são automaticamente calculadas das respectivas labels durante a compilação.

| ***Categ.*** | ***Abrev.*** | ***Significa ...*** | ***Faixa de Valores*** |
|---|---|---|---|
| Register | r1 | Registradores origem e destino comuns | R0..R31 |
| | r2 | Registrador fonte comum | |
| | rh | Registrador de página superior | R16..R31 |
| | rd | Registrador gêmeo | R24(R25), R26(R27), R28(R29), R30(R31) |
| | rp | Registrador de ponteiro | X=R26(R27), Y=R28(R29), Z=R30(R31) |
| | ry | Registrador de ponteiro com deslocamento | Y=R28(R29), Z=R30(R31) |
| Constant | k63 | Constante ponteiro | 0..63 |
| | c127 | Distância de salto condicional | -64..+63 |
| | c255 | Constante de 8 bits | 0..255 |
| | c4096 | Distância de salto relativo | -2048..+2047 |
| | c65535 | Endereço de 16 bits | 0..65535 |
| Bit | b7 | Posição do bit | 0..7 |
| Port | p1 | Porta comum | 0..63 |
| | pl | Porta de página inferior | 0..31 |