

CAPÍTULO 1

Começando

O Raspberry Pi é um computador do tamanho de um cartão de crédito que custa apenas US\$ 35. Foi originalmente desenvolvido para fornecer computadores de baixo custo para escolas e crianças, que não podiam pagar por PCs ou Macs comuns. Desde o seu lançamento, o Raspberry Pi tem sido incrivelmente bem-sucedido - até o momento, vendendo mais de 25 milhões de unidades. O Raspberry Pi tornou-se a base de todo um movimento DIY com diversas aplicações, incluindo sistemas de controle de automação residencial, atuando como o cérebro de robôs ou conectados para construir um supercomputador pessoal. O Pi também é uma ótima ferramenta educacional.

Este livro aproveitará o Raspberry Pi para ajudá-lo a aprender a linguagem Assembly. Programar em linguagem Assembly é programar seu computador no nível mais baixo de bits e bytes. As pessoas geralmente programam computadores em linguagens de programação de alto nível, como Python, C, Java, C# ou JavaScript. As ferramentas que acompanham essas linguagens convertem seu programa em linguagem Assembly, sejam elas feitas de uma só vez ou à medida que são executadas.

A linguagem assembly é específica para o processador do computador usado. Como estamos aprendendo para o Raspberry Pi, aprenderemos a linguagem Assembly para o processador Advanced RISC Machine (ARM). Usaremos o sistema operacional Raspbian, um sistema operacional de 32 bits baseado no Debian Linux, então aprenderemos Assembly de 32 bits no processador ARM do Raspberry Pi.

O Raspberry Pi 3 possui um processador ARM que pode operar no modo de 64 bits, mas o Raspbian não faz isso. Destacaremos algumas diferenças importantes entre o Assembly de 32 bits e o de 64 bits, mas todos os nossos programas de amostra estarão no ARM Assembler de 32 bits e serão compilados para serem executados no Raspbian.

Capítulo 1 Introdução

Sobre o processador ARM

O processador ARM foi originalmente desenvolvido por um grupo na Grã-Bretanha, que queria construir um sucessor do microcomputador BBC usado para fins educacionais. O microcomputador da BBC usava o processador 6502, que era um processador simples com um conjunto de instruções simples. O problema era que não havia sucessor para o 6502. Eles não estavam satisfeitos com os microprocessadores que existiam na época, pois eram muito mais complicados que o 6502 e não queriam fazer outro clone do IBM PC. Eles tomaram a atitude ousada de criar os seus próprios. Eles desenvolveram o computador Acorn que o utilizou e tentaram posicionará-lo como o sucessor do BBC Microcomputer. A ideia era usar a tecnologia do computador com conjunto de instruções reduzidas (RISC) em oposição ao computador com conjunto de instruções complexas (CISC), defendido pela Intel e pela Motorola. Falaremos longamente sobre o que esses termos realmente significam mais tarde.

O desenvolvimento de chips de silício é uma proposta cara e, a menos que você consiga um bom volume, a fabricação é cara. O processador ARM provavelmente não teria ido a lugar nenhum, exceto que a Apple ligou procurando um processador para um novo dispositivo que eles tinham em desenvolvimento - o iPod. O principal ponto de venda para a Apple era que, como o processador ARM era RISC, ele usava menos silício do que os processadores CISC e, como resultado, consumia muito menos energia. Isso significava que era possível construir um dispositivo que funcionasse por muito tempo com uma única carga de bateria.

Ao contrário da Intel, a ARM não fabrica chips; apenas licencia os projetos para que outros otimizem e fabriquem. Com a Apple a bordo, de repente houve muito interesse no ARM, e vários grandes fabricantes começaram a produzir chips. Com o advento dos smartphones, o chip ARM realmente decolou e agora é usado em praticamente todos os telefones e tablets.

Os processadores ARM alimentam até mesmo alguns Chromebooks. O processador ARM é o processador número um no mercado de computadores.

O que você vai aprender

Você aprenderá programação em linguagem Assembly para o processador ARM no Raspberry Pi, mas tudo o que aprender é diretamente aplicável a todos esses outros dispositivos. Aprender a linguagem Assembly para um processador fornece as ferramentas para aprendê-la para outro processador, talvez o futuro RISC-V.

O chip que é o cérebro do Raspberry Pi não é apenas um processador, é também um sistema em um chip. Isso significa que a maior parte do computador está em um único chip. Este chip contém um processador ARM quad-core, o que significa que pode processar instruções para quatro programas em execução ao mesmo tempo. Ele também contém vários coprocessadores para coisas como cálculos de ponto flutuante, uma unidade de processamento gráfico (GPU) e suporte multimídia especializado.

A ARM faz um bom trabalho no suporte a coprocessadores e permite que os fabricantes construam seus chips de maneira modular, incorporando os elementos de que precisam. Todos os Raspberry Pi incluem um coprocessador de ponto flutuante (**FPU**). O Raspberry Pi mais recente possui recursos avançados, como processadores paralelos NEON. A Tabela 1-1 fornece uma visão geral das unidades que programaremos e quais Raspberry Pi as suportam. Na Tabela 1-1, o **SoC** é um sistema em um chip e contém o número de peça da Broadcom para a unidade incorporada.

Capítulo 1 Introdução

Tabela 1-1. Modelos Raspberry Pi comuns e seus recursos relevantes para este livro

Modelo SoC		Instrução de divisão de memória		FPU NEON coprocessador		64 bits apoiar
Pi A+	BCM2835	256 MB		v2		
Pi B	BCM2835	512 MB		v2		
Pi Zero	BCM2835	512 MB		v2		
Pi 2	BCM2836	1 GB	Sim	v3	Sim	Sim
Pi 3	BCM2837	1 GB	Sim	v4	Sim	Sim
Pi 3+	BCM2837B0	1 GB	Sim	v4	Sim	Sim
Pi 4	BCM2711	1, 2 ou 4 GB	Sim	v4	Sim	Sim

Por que usar a montagem

A maioria dos programadores hoje escreve em uma linguagem de programação de alto nível como Python, C#, Java, JavaScript, Go, Julia, Scratch, Ruby, Swift ou C. Essas são linguagens altamente produtivas usadas para escrever os principais programas do sistema operacional Linux para sites como o Facebook para software de produtividade como o LibreOffice. Se você aprender a ser um bom programador em alguns deles, poderá encontrar um emprego interessante e bem remunerado e escrever ótimos programas. Se você criar um programa em uma dessas linguagens, poderá fazê-lo funcionar facilmente em vários sistemas operacionais em várias arquiteturas de hardware. Você nunca precisa aprender os detalhes de todos os bits e bytes, e eles podem permanecer ocultos com segurança.

Quando você programa em linguagem Assembly, você está fortemente acoplado a uma determinada CPU, e mover seu programa para outra requer uma reescrita completa de seu programa. Cada instrução da linguagem Assembly faz apenas uma fração da quantidade de trabalho, portanto, para fazer qualquer coisa, é preciso muito Assembly

Capítulo 1 Introdução

declarações. Portanto, para fazer o mesmo trabalho que, digamos, um programa Python, exige uma quantidade de esforço muito maior para o programador.

Escrever em Assembly é mais difícil, pois você deve resolver problemas com endereçamento de memória e registradores de CPU que são tratados de forma transparente por linguagens de alto nível. Então, por que você iria querer aprender programação em linguagem Assembly? Aqui estão dez razões pelas quais as pessoas aprendem e usam a linguagem Assembly:

1. Mesmo que você não escreva código em linguagem Assembly, saber como o computador funciona internamente permite que você escreva um código mais eficiente. Você pode tornar suas estruturas de dados mais fáceis de acessar e escrever código em um estilo que permite ao compilador gerar código mais eficiente. Você pode fazer melhor uso dos recursos do computador, como coprocessadores, e usar o computador fornecido em todo o seu potencial.
2. Para escrever seu próprio sistema operacional. O próprio núcleo do sistema operacional que inicializa a CPU lida com a segurança do hardware e o multithreading/multitarefa requer o código Assembly.
3. Para criar uma nova linguagem de programação. Se for uma linguagem compilada, você precisará gerar o código Assembly para executar. A qualidade e a velocidade da sua linguagem dependem em grande parte da qualidade e da velocidade do código da linguagem Assembly que ela gera.
4. Você deseja tornar o Raspberry Pi mais rápido. A melhor maneira de tornar o Raspbian mais rápido é melhorar o compilador GNU C. Se você melhorar o código Assembly ARM de 32 bits produzido pelo GNU C, todos os programas Linux compilados para os benefícios do Pi.

Capítulo 1 Introdução

5. Você pode estar conectando seu Pi a um hardware

dispositivo, por meio de portas USB ou GPIO, e a velocidade de transferência de dados é altamente sensível à rapidez com que seu programa pode processar os dados. Talvez haja muitas manipulações em nível de bit que são mais fáceis de programar em Assembly.

6. Para fazer aprendizado de máquina ou gráficos 3D mais rápidos

programação. Ambos os aplicativos contam com matemática de matriz rápida. Se você puder tornar isso mais rápido com o Assembly e/ou usando os coprocessadores, poderá tornar seu robô ou videogame baseado em IA muito melhor.

7. A maioria dos programas grandes tem componentes escritos em

diferentes linguagens. Se o seu programa for 99% C++, o outro 1% pode ser Assembly, talvez dando ao seu programa um aumento de desempenho ou alguma outra vantagem competitiva.

8. Talvez você trabalhe para uma empresa de hardware que

faz um concorrente de computador de placa única para o Raspberry Pi. Essas placas possuem algum código em linguagem Assembly para gerenciar os periféricos incluídos na placa. Esse código geralmente é chamado de BIOS (sistema básico de entrada/saída).

9. Procurar vulnerabilidades de segurança em um programa ou peça de

hardware. Geralmente, você precisa consultar o código Assembly para fazer isso; caso contrário, você pode não saber o que realmente está acontecendo e, portanto, onde podem existir buracos.

Capítulo 1 Introdução

10. Procurar ovos de Páscoa em programas. São mensagens ocultas, imagens ou piadas internas que os programadores ocultam em seus programas. Eles geralmente são ativados encontrando uma combinação de teclado secreta para exibilos. Encontrá-los requer engenharia reversa do programa e leitura da linguagem Assembly.

Ferramentas que você precisa

Este livro foi projetado para que tudo o que você precisa seja um Raspberry Pi que execute o sistema operacional Raspbian. O Raspbian é baseado no Debian Linux, então qualquer coisa que você saiba sobre o Linux é diretamente útil. Existem outros sistemas operacionais para o Pi, mas abordaremos apenas o Raspbian neste livro.

Um Raspberry Pi 3, modelo B ou B+, é o ideal. A maior parte do que está neste livro também funciona em modelos mais antigos, já que as diferenças estão principalmente nas unidades do coprocessador e na quantidade de memória. Falaremos sobre como desenvolver programas para rodar nos modelos compactos A e Raspberry Pi Zero, mas você não gostaria de desenvolver seus programas diretamente neles.

Uma das grandes vantagens do sistema operacional Raspbian é que ele se destina a ensinar programação e, como resultado, possui muitas ferramentas de programação pré-instaladas, incluindo

- GNC Compiler Collection (GCC) que usaremos para construir nossos programas em linguagem Assembly. Usaremos o GCC para compilar programas C em capítulos posteriores.
- GNU Make para construir nossos programas.
- GNU Debugger (GDB) para encontrar e resolver problemas em nossos programas.

Capítulo 1 Introdução

Você precisará de um editor de texto para criar os arquivos de programa de origem. Qualquer editor de texto pode ser usado. O Raspbian inclui vários por padrão, tanto na linha de comando quanto por meio da GUI. Normalmente, você aprende a linguagem Assembly depois de já dominar uma linguagem de alto nível como C ou Java. Portanto, é provável que você já tenha um editor favorito e possa continuar a usá-lo.

Mencionaremos outros programas úteis ao longo do livro que você pode usar opcionalmente, mas não são obrigatórios, por exemplo:

- Uma calculadora de programador melhor
- Uma ferramenta de análise de código melhor

Todos eles são de código aberto e você pode instalá-los gratuitamente.

Agora vamos mudar de assunto para como os computadores representam números. Sempre ouvimos que os computadores lidam apenas com zeros e uns, agora veremos como eles os juntam para representar números maiores.

Computadores e números

Normalmente representamos números usando a base 10. A teoria comum é que fazemos isso porque temos 10 dedos para contar. Isso significa que um número como 387 é realmente uma representação para

$$\begin{aligned} 387 &= 3 \cdot 10^2 + 8 \cdot 10^1 + 7 \cdot 10^0 \\ &= 3 \cdot 100 + 8 \cdot 10 + 7 \\ &= 300 + 80 + 7 \end{aligned}$$

Não há nada de especial em usar o 10 como base e um exercício divertido na aula de matemática é fazer aritmética usando outras bases. Na verdade, a cultura maia usava a base 20, talvez porque temos 20 dígitos: 10 dedos das mãos e 10 dedos dos pés.

Os computadores não têm dedos das mãos e dos pés e, no mundo deles, tudo é um interruptor que está ligado ou desligado. Como resultado, é natural que os computadores

Capítulo 1 Introdução

para usar aritmética de base 2. Assim, para um computador, um número como 1011 é representado por

$$\begin{aligned}
 1011 &= 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
 &= 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \\
 &= 8 + 0 + 2 + 1 \\
 &= 11 \text{ (decimais)}
 \end{aligned}$$

Isso é ótimo para computadores, mas estamos usando 4 dígitos para o número decimal 11 em vez de 2 dígitos. A grande desvantagem para os humanos é que escrever números binários é cansativo, porque ocupam muitos dígitos.

Os computadores são incrivelmente estruturados, então todos os seus números são os mesmos tamanho. Ao projetar computadores, não faz sentido ter todos os tipos de números de tamanhos diferentes, então alguns tamanhos comuns se estabeleceram e tornar-se padrão.

Um byte é 8 bits binários ou dígitos. Em nosso exemplo anterior com 4 bits, existem 16 combinações possíveis de 0s e 1s. Isso significa que 4 bits podem representar os números de 0 a 15. Isso significa que pode ser representado por um dígito de base 16. Os dígitos da base 16 são representados pelos números de 0 a 9 e depois pelas letras A–F para 10–15. Podemos então representar um byte (8 bits) como dois dígitos de base 16. Referimo-nos aos números de base 16 como hexadecimais (Figura 1-1).

Decimal	0 - 9	10	11	12	13	14	15
Hex Digit	0 – 9	A	B	C	D	E	F

Figura 1-1. Representando dígitos hexadecimais

Como um byte contém 8 bits, ele pode representar 28 (256) números. Assim, o byte e6 representa

$$\begin{aligned}
 e6 &= e \cdot 16^1 + 6 \cdot 16^0 \\
 &= 14 \cdot 16 + 6 \\
 &= 230 \text{ (decimal)} = \\
 &1110\ 0110 \text{ (binário).}
 \end{aligned}$$

Capítulo 1 Introdução

Estamos executando o processador ARM no modo de 32 bits; chamamos uma quantidade de 32 bits de palavra e ela é representada por 4 bytes. Você pode ver uma string como B6 A4 44 04 como uma representação de 32 bits de memória, ou uma palavra de memória, ou talvez o conteúdo de um registrador.

Se isso for confuso ou assustador, não se preocupe. As ferramentas farão todas as conversões para você. É apenas uma questão de entender o que é apresentado a você na tela. Além disso, se você precisar especificar um número binário exato, geralmente o faz em hexadecimal, embora todas as ferramentas aceitem todos os formatos.

Uma ferramenta útil é a calculadora Linux Gnome (Figura 1-2). A calculadora incluída no Raspbian pode executar matemática em diferentes bases em seu modo científico, mas a calculadora Gnome possui um modo de programação mais agradável, que mostra uma representação de números em várias bases ao mesmo tempo. Para instalá-lo, use a linha de comando

```
sudo apt-get install gnome-calculator
```

Execute-o no menu Acessórios (provavelmente a segunda calculadora lá). Se você colocá-lo no “Modo de Programação”, poderá fazer as conversões e mostrará números em vários formatos ao mesmo tempo.

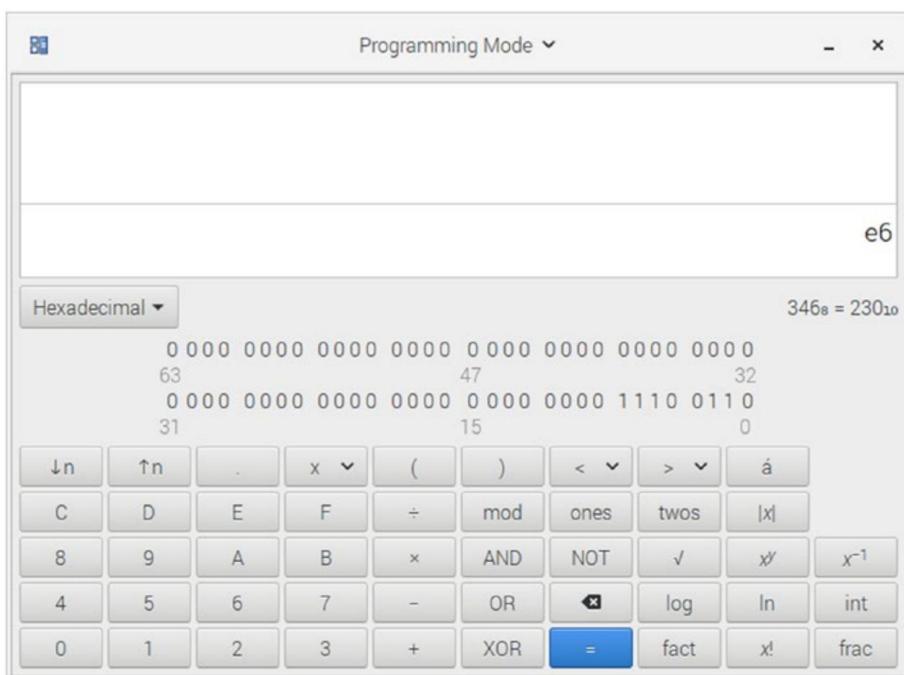


Figura 1-2. A calculadora Gnome

É assim que representamos a memória do computador. Há um pouco mais de complexidade em como os números inteiros com sinal são representados e como a aritmética funciona. Abordaremos isso um pouco mais tarde, quando formos fazer alguma aritmética.

No Assembler representamos números hexadecimais (hex para abreviar) com um 0x na frente. Portanto, 0x1B é como especificaríamos o número hexadecimal 1B.

Instruções de Montagem do ARM

Nesta seção, apresentamos alguns elementos arquitetônicos básicos do processador ARM e começamos a examinar a forma de suas instruções de código de máquina.

O ARM é o que chamamos de Computador com Conjunto de Instruções Reduzido (RISC), que teoricamente facilitará o aprendizado de Assembly. Há menos instruções e cada instrução é mais simples, então o processador pode executar

Capítulo 1 Introdução

cada instrução muito mais rápido. Embora isso seja verdade, o sistema ARM em um chip usado no Raspberry Pi é um computador altamente sofisticado. Os processadores ARM principais lidam com vários conjuntos de instruções e, em seguida, há os conjuntos de instruções para todos os coprocessadores.

Nossa abordagem para isso é dividir e conquistar. Nos primeiros capítulos deste livro, abordaremos apenas as instruções de montagem ARM padrão de 32 bits. Isso significa que os seguintes tópicos são adiados para capítulos posteriores, onde podem ser abordados em detalhes sem causar muita confusão:

- Instruções para o processador de ponto flutuante
- Instruções para o processador NEON
- Instruções para 64 bits
- Instruções do modo Thumb (compacto especial de 16 bits modo)

Desta forma, só precisamos atacar um tópico de cada vez. Cada conjunto de instruções é consistente e fácil de entender.

Em tópicos técnicos de informática, muitas vezes há problemas de galinha e ovo na apresentação do material. O objetivo desta seção é apresentar todos os termos e ideias que usaremos posteriormente. Esperançosamente, isso apresenta todos os termos, para que sejam familiares quando os cobrirmos com todos os detalhes.

Registros da CPU

Em todos os computadores, os dados não são operados na memória do computador; em vez disso, ele é carregado em um registrador da CPU, então o processamento de dados ou operação aritmética é executado nos registradores. Os registradores fazem parte do circuito da CPU permitindo acesso instantâneo, enquanto a memória é um componente separado e há um tempo de transferência para a CPU acessá-la.

Capítulo 1 Introdução

Se você deseja adicionar dois números, pode carregar um em um registrador, o outro em outro registrador, execute a operação de adição colocando o resultado em um terceiro registrador, então copie a resposta do registrador de resultado na memória. Como você pode ver, são necessárias algumas instruções para realizar operações simples.

Um programa em um processador ARM no modo de usuário tem acesso a 16 registradores:

- **R0 a R12:** Esses 13 são de propósito geral que você pode usar para o que quiser.
- **R13:** O ponteiro da pilha.
- **R14:** O registro do link. **R13 e R14** são usados no contexto de chamada de funções, e vamos explicá-los com mais detalhes quando cobrirmos as sub-rotinas.
- **R15:** O contador do programa. O endereço de memória do instrução atualmente em execução.
- Registro de Status do Programa Atual (**CPSR**): Este 17º registro contém bits de informação sobre a última instrução executada. Mais sobre o **CPSR** quando instruções de desvio de cobertura (se declarações).

Formato de Instrução ARM

Cada instrução binária ARM tem 32 bits de comprimento. Ajustar todas as informações para uma instrução em 32 bits é uma grande conquista que requer o uso de cada bit para dizer ao processador o que fazer. Existem alguns formatos de instrução, e vamos explicá-los quando cobrirmos essa instrução. Para lhe dar uma ideia das instruções de processamento de dados, vamos considerar o formato de uma classe comum de instruções com as quais lidaremos no início. A Figura 1-3 mostra o formato da instrução e o que os bits especificam.

Capítulo 1 Introdução

31 – 28	27 – 25	24 – 21	20	19 – 16	15 – 12	11 – 0
Condition	Operand type	OpCode	Set Condition Codes	Operand Register	Destination Register	Immediate Operand

Figura 1-3. Formato de instrução para instruções de processamento de dados

Vejamos cada um desses campos:

- **Condição:** permite que a instrução seja executada dependendo dos bits no CPSR. Examinaremos isso em detalhes quando chegarmos às instruções de ramificação.
- **Tipo de operando:** Especifica quais são os operandos nos bits 19–0. Poderíamos ter especificado alguns desses bits, pois usamos dois registradores e um operando imediato neste exemplo.
- **Opcode:** Qual instrução estamos executando, como ADICIONE ou MUL.
- **Definir código de condição:** Este é um único bit que indica se esta instrução deve atualizar o CPSR. Se não quisermos que o resultado desta instrução afete as instruções de desvio seguintes, devemos defini-la como 0.
- **Registrador de operando:** Um registrador para usar como entrada.
- **Registro de destino:** Onde colocar o resultado do tudo o que esta instrução faz.
- **Operando imediato:** geralmente é um pouco de dados que você pode especificar diretamente na instrução. Então, se você quiser adicionar 1 a um registro, você pode ter isso como 1, em vez de colocar 1 em outro registro e adicionar os dois registros. O formato deste campo é bastante complicado e requer uma seção maior para explicar todos os detalhes, mas esta é a ideia básica.

Capítulo 1 Introdução

Quando as coisas estão funcionando bem, cada instrução é executada em um ciclo de clock. Uma instrução isolada leva três ciclos de clock, ou seja, um para carregar a instrução da memória, um para decodificar a instrução e então um para executar a instrução. O ARM é inteligente e funciona em três instruções por vez, cada uma em uma etapa diferente do processo, chamada pipeline de instruções. Se você tiver um bloco linear de instruções, todas elas serão executadas em média levando um ciclo de clock.

Memória Raspberry Pi

A Tabela 1-1 mostra a quantidade de memória que cada Raspberry Pi contém.

Os programas são carregados do cartão SD do Pi na memória e executados. A memória contém o programa, juntamente com quaisquer dados ou variáveis associadas a ele. Essa memória não é tão rápida quanto a CPU registra, mas é muito mais rápida do que acessar dados armazenados no cartão SD ou em um dispositivo conectado a uma porta USB.

Já falamos muito sobre o modo de 32 bits, mas o que é? O que o modo de 32 bits realmente significa é que os endereços de memória são especificados usando 32 bits e os registradores da CPU têm cada um 32 bits de largura.

As instruções também têm tamanho de 32 bits quando executadas no modo de 64 bits; o A diferença é que 64 bits são usados para especificar um endereço de memória e os registradores têm 64 bits de largura.

Se quisermos carregar um registrador de um endereço de memória conhecido de 32 bits, por exemplo, uma variável na qual queremos realizar aritmética. Como vamos fazer isso? A instrução tem apenas 32 bits de tamanho e já usamos 4 bits para o opcode, 4 bits para uma instrução condicional, 3 bits para o tipo de operando e 1 bit para dizer se afetamos o CPSR. Precisamos de 4 bits para especificar um registrador, então deixamos 16 bits para o endereço de memória (12 bits se precisarmos listar dois registradores).

Esse é um problema ao qual voltaremos várias vezes, pois existem várias maneiras de resolvê-lo. Em um computador CISC, isso não é um problema, pois as instruções geralmente são muito grandes e de tamanho variável.

Capítulo 1 Introdução

Você pode carregar da memória usando um registrador para especificar o endereço a ser carregado. Isso é chamado de acesso indireto à memória. Mas tudo o que fizemos foi mover o problema, já que não temos como colocar o valor naquele registrador (em uma única instrução).

Você poderia carregar dois registradores, cada um com metade do endereço, depois deslocar a parte superior e, em seguida, adicionar os dois. Quatro instruções para carregar um endereço, o que parece bastante ineficiente.

A maneira rápida de carregar a memória que não está muito longe do programa registrador do contador (PC) é usar a instrução load via PC, já que permite um deslocamento de 12 bits do registrador. Parece que você pode acessar a memória com eficiência dentro de 4096 palavras do PC, mas é mais porque alguns dos bits especificam uma mudança para fornecer um intervalo maior. Eca, como você escreveria esse código? É aqui que entra o GNU Assembler. Ele permite que você especifique o local simbolicamente e descobrirá o deslocamento deslocamento para você.

No Capítulo 2, “Carregando e Adicionando”, veremos o operando imediato com mais detalhes. Abordaremos muitas outras maneiras de especificar endereços de memória em capítulos futuros, como pedir ao Linux para nos fornecer um bloco de memória, retornando o endereço em um registrador para nós. Por enquanto, usar o PC com offset atende às nossas necessidades.

Sobre o GCC Assembler

Escrever código Assembler em binário como instruções de 32 bits seria extremamente tedioso. Entre no Assembler do GNU, que lhe dá o poder de especificar tudo o que o ARM pode fazer, mas cuida de colocar todos os bits no lugar certo para você. A maneira geral de especificar as instruções de montagem é

rótulo: Códigos de operação operandos

O rótulo: é opcional e só é necessário se você quiser que a instrução ser o alvo de uma instrução de desvio.

Capítulo 1 Introdução

Existem alguns opcodes, cada um é um pequeno mnemônico que é legível por humanos e fácil para o Assembler processar. Eles incluem

- **ADD** para adição
- **LDR** para carregar um registrador
- **B** para ramificação

Existem alguns formatos diferentes para os operandos, e vamos cobrir aqueles como nós cobrimos as instruções que os usam.

Olá Mundo

Em quase todos os livros de programação, o primeiro programa é um programa simples para exibir a string "Hello World". Faremos o mesmo com Assembly para demonstrar alguns dos conceitos sobre os quais falamos.

Em nosso editor de texto favorito, vamos criar um arquivo "HelloWorld.s" contendo isso na Listagem 1-1.

Listagem 1-1. O programa Alô Mundo

```
@ @ Programa Assembler para imprimir "Hello World!" @ para
stdout. @ @ R0-R2 - parâmetros para serviços de função linux
@ R7 - número de função linux @
```

```
.global _start @           @ Fornecer início do programa
endereço para vinculador
```

```
@ Configure os parâmetros para imprimir hello world @ e chame
o Linux para fazer isso.
```

Capítulo 1 Introdução

```
_start: mov R0, #1          @ 1 = StdOut
        ldr R1, =helloworld @ string para imprimir mov R2, #13 mov
        R7, #4              @ comprimento da nossa string
        @ linux write system call
        svc 0               @ Chame o linux para imprimir
```

@ Defina os parâmetros para sair do programa @ e depois chame
o Linux para fazer isso.

```
movimento      R0, #0 @ Use 0 código de retorno
movimento      R7, #1 @ Código de comando de serviço 1 @
                finaliza este programa
svc            0       @ Chame o linux para encerrar
```

.dados

Olá Mundo: .ascii "Olá Mundo\n"

Esta é nossa primeira olhada em um programa completo em linguagem Assembly,
portanto, há algumas coisas sobre as quais falar. Mas primeiro vamos compilar e executar isso
programa.

Em nosso editor de texto, crie um arquivo chamado “build” que contém

as -o HelloWorld.o HelloWorld.s

ld -o HelloWorld HelloWorld.o

Estes são os comandos para compilar nosso programa. Primeiro, temos que tornar este
arquivo executável usando o comando do terminal

chmod +x compilação

Agora, podemos executá-lo digitando **./build**. Se os arquivos estiverem corretos,
podemos executar nosso programa digitando **./HelloWorld**. Na Figura 1-4, usei **bash -x** (modo
de depuração), para que você possa ver os comandos sendo executados.

A screenshot of a terminal window titled "pi@stevepi: ~/asm/HelloWorld". The window has a blue header bar with the title and standard window controls. Below the title, there is a menu bar with "File", "Edit", "Tabs", and "Help". The main area of the terminal shows the following command-line session:

```
pi@stevepi:~/asm/HelloWorld $ bash -x build
+ as -o HelloWorld.o HelloWorld.s
+ ld -o HelloWorld HelloWorld.o
pi@stevepi:~/asm/HelloWorld $ ./HelloWorld
Hello World!
pi@stevepi:~/asm/HelloWorld $
```

Figura 1-4. Construindo e executando o *HelloWorld*

Se executarmos “ls -l”, a saída será

```
-rwxr-xr-x 1 pi pi 62 Jun 6 19:25 build -rwxr-xr-x 1 pi pi 884 Jun
6 19:25 HelloWorld -rw-r--r-- 1 pi pi 728 Jun 6 19:25 HelloWorld.o -rw-
r--r-- 1 pi pi 803 Jun 6 19:23 HelloWorld.s
```

Observe como esses arquivos são pequenos. O executável tem apenas 884 bytes, nem mesmo 1 KB. Isso ocorre porque não há tempo de execução ou qualquer outra biblioteca necessária para executar este programa; é inteiramente completo em si mesmo. Se você deseja criar executáveis muito pequenos, a programação em linguagem Assembly é o caminho a percorrer.

O formato deste programa é uma convenção comum para Assembly programas de linguagem em que cada linha está nestas quatro colunas:

- Rótulo de declaração opcional

- Código de operação

Capítulo 1 Introdução

- Operandos
- Comente

Todos eles são separados por guias, então eles se alinharam bem.

Yay, nosso primeiro programa em linguagem Assembly funcional. Agora, vamos falar sobre todas as partes.

Sobre o comentário inicial

Iniciamos o programa com um comentário que indica o que ele faz. Também documentamos os registradores utilizados. Manter o controle de quais registradores estão fazendo o que se torna importante à medida que nossos programas aumentam:

- Sempre que você vir um caractere "@" em uma linha, então tudo depois do "@" é um comentário. Isso significa que está lá para documentação e é descartado pelo GNU Assembler quando processa o arquivo.
- A linguagem assembly é enigmática, por isso é importante documentar o que você está fazendo. Caso contrário, você retornará ao programa depois de algumas semanas e não terá ideia do que o programa faz.
- Cada seção do programa tem um comentário informando o que faz e cada linha do programa tem um comentário no final informando o que faz. Tudo entre /ÿ e ÿ/ também é um comentário e será ignorado.

Onde começar

Em seguida, especificamos o ponto de partida do nosso programa:

Capítulo 1 Introdução

- Precisamos defini-lo como um símbolo global, para que o vinculador (o comando `ld` em nosso arquivo de compilação) tenha acesso a ele. O Assembler marca a instrução contendo `_start` como o ponto de entrada do programa, então o vinculador pode encontrá-lo porque foi definido como uma variável global.
Todos os nossos programas conterão isso em algum lugar.
- Nosso programa pode consistir em vários arquivos `.s`, mas apenas um pode conter `_start`.

Instruções de montagem

Usamos apenas três instruções diferentes da linguagem Assembly neste exemplo:

1. **MOV** que move os dados para um registrador. Neste caso, utilizamos um operando imediato, que inicia com o sinal `#`. Portanto, `"MOV R1, #4"` significa mover o número 4 para **R1**. Nesse caso, o 4 faz parte da instrução e não está armazenado em outro lugar na memória. No arquivo de origem, os operandos podem ser maiúsculos ou minúsculos; Costumo preferir letras minúsculas em minhas listas de programas.
2. Instrução `"LDR R1, =helloworld"` que carrega registrar 1 com o endereço da string que queremos imprimir.
3. Comando **SVC 0** que executa a interrupção do software número 0. Isso envia o controle para o manipulador de interrupção no kernel do Linux, que interpreta os parâmetros que definimos em vários registros e faz o trabalho real.

Capítulo 1 Introdução

Dados

Em seguida, temos .data que indica que as seguintes instruções estão na seção de dados do programa:

- Nisto, temos um rótulo “helloworld” seguido por um **.ascii** e depois a string que queremos imprimir.
- A instrução **.ascii** diz ao Assembler apenas para colocar nossa string na seção de dados e então podemos acessá-la através do rótulo como fazemos na instrução LDR. Conversaremos mais tarde sobre como o texto é representado como números, o esquema de codificação aqui sendo chamado de ASCII.
- O último caractere “\n” é como representamos uma nova linha. Se não incluímos isso, você deve pressionar retornar para ver o texto na janela do terminal.

Chamando o Linux

Este programa faz duas chamadas de sistema Linux para fazer seu trabalho. O primeiro é o comando Linux write to file (#4). Normalmente, teríamos que abrir um arquivo antes de usar este comando, mas quando o Linux executa um programa, ele abre três arquivos para ele:

1. stdin (entrada do teclado)
2. stdout (saída para a tela)
3. stderr (também enviado para a tela)

O shell do Linux irá redirecioná-los quando você pedir para usar >, < e | em seus comandos. Para qualquer chamada do sistema Linux, você coloca os parâmetros nos registradores R0–R4 , dependendo de quantos parâmetros são necessários. Então uma

Capítulo 1 Introdução

o código de retorno é retornado em **R0** (que estamos errados e não verificamos). Cada chamada do sistema é especificada colocando seu número de função em **R7**.

A razão pela qual fazemos uma interrupção de software em vez de uma ramificação ou chamada de sub-rotina é para que possamos chamar o Linux sem precisar saber onde essa rotina está na memória. Isso é bastante inteligente e significa que não precisamos alterar nenhum endereço em nosso programa, pois o Linux é atualizado e suas rotinas se movem na memória. A interrupção de software tem outro benefício de fornecer um mecanismo padrão para alternar os níveis de privilégio. Discutiremos as chamadas do sistema Linux posteriormente no Capítulo 7, “Serviços do sistema operacional Linux”.

Engenharia reversa do nosso programa

Falamos sobre como cada instrução Assembly é compilada em uma palavra de 32 bits. O Assembler fez isso por nós, mas podemos ver o que ele fez? Uma maneira é usar o programa de linha de comando objdump

```
objdump -s -d HelloWorld.o
```

que produz a Listagem 1-2.

Listagem 1-2. Desmontagem do Hello World

OláMundo.o: formato de arquivo elf32-littlearm

Conteúdo da seção .text:

```
0000 0100a0e3 14109fe5 0d20a0e3 0470a0e3 ..... p.. 0010 000000ef  
0000a0e3 0170a0e3 000000ef ..... p.....  
0020 00000000 .....  
....
```

Conteúdo da seção .data:

```
0000 48656c6c 6f20576f 726c6421 0a Olá Mundo!.
```

Conteúdo da seção .ARM.attributes:

```
0000 41130000 00616561 62690001 09000000 A....aeabi.....  
0010 06010801 .....  
....
```

Capítulo 1 Introdução

Desmontagem da seção .text:

00000000 <_start>: 0:

```
e3a00001 mov r0, #1 4: e59f1014
ldr r1, [pc, #20] 8: e3a0200d mov r2, #13 c: ; 20 <_start+0x20>
e3a07004 mov r7, #4 10: ef000000 svc
0x00000000
```

14: e3a00000 mov r0, #0 18:

```
e3a07001 mov r7, #1 1c: ef000000
```

```
svc 0x00000000
```

20: 00000000 .word 0x00000000

A parte superior da saída mostra os dados brutos no arquivo, incluindo nossas oito instruções e, em seguida, nossa string para imprimir na seção .data. A segunda parte é uma desmontagem da seção .text executável.

Vejamos a primeira instrução **MOV** compilada para 0xe3a00001 (Figura 1-5):

Hex Digit	e	3	a	0	0	0	0	1
Binary	1110	0011	1100	0000	0000	0000	0000	1

Figura 1-5. Representação binária da primeira instrução MOV

- Cada instrução começa com o dígito hexadecimal “e” (14 decimal ou 1110 binário). Este é o código de condição, que nos permite executar condicionalmente uma instrução, e agora sabemos que “e” significa executar a instrução incondicionalmente.
- Os próximos 3 bits especificam 001 que indica o tipo de operando, que neste caso é um registrador e um operando imediato.

Capítulo 1 Introdução

- Os próximos 4 bits são 1110, que é o opcode para o instrução MOV.
- O próximo bit é 0 que indica o tipo de parâmetro de modo imediato, que neste caso simples não matéria.
- Os próximos 4 bits são o número do registrador que é 0.
- Se você observar as outras instruções **do MOV**, poderá ver o número do registro neste local.
- Os bits restantes compõem nosso número de modo imediato, que é 1.

Veja a instrução LDR, ela mudou de

`ldr R1, =olá mundo`

para

`ldr r1, [peça, nº 20] ; 20 <_start+0x20>`

Este é o Assembler ajudando você com o obscuro mecanismo do processador ARM de endereçar a memória. Ele permite que você especifique um endereço simbólico, ou seja, “helloworld”, e traduza isso em um deslocamento do contador do programa. Certamente estou feliz por ter uma ferramenta que faz esse tipo de maldade para mim.

Você pode notar que as instruções brutas na parte superior da saída têm seus bytes invertidos, em comparação com aqueles listados na listagem de desmontagem. Isso ocorre porque estamos usando uma codificação little-endian, que abordaremos no próximo capítulo.

Sinta-se à vontade para brincar com o programa, por exemplo:

- Altere a string, mas lembre-se de alterar o comprimento carregado em **R2**.
- Altere o código de retorno carregado em **R0** antes da segunda chamada **SVC** e veja o que acontece.

Capítulo 1 Introdução

Dica Você só aprende programação experimentando e escrevendo seu próprio código.

À medida que avançamos em cada capítulo, você será capaz de fazer mais e mais.

Resumo

Neste capítulo, apresentamos o processador ARM e a programação em linguagem Assembly junto com o motivo pelo qual queremos usar o Assembly. Cobrimos as ferramentas que usaremos. Também vimos como os computadores representam números inteiros positivos.

Em seguida, examinamos com mais detalhes como a CPU ARM representa as instruções de montagem junto com os registradores que ela contém para processamento de dados. Apresentamos a memória do Raspberry Pi e o GNU Assembler que nos ajudará a escrever nossos programas em linguagem Assembly.

Por fim, criamos um programa simples e completo para imprimir “Hello World!” em nossa janela de terminal.

No Capítulo 2, “Carregando e Adicionando”, veremos como carregar dados nos registradores da CPU e realizar adições básicas. Veremos como os números negativos são representados e aprenderemos novas técnicas para manipular bits binários.

CAPÍTULO 2

Carregando e adicionando

Neste capítulo, examinaremos lentamente as instruções **MOV** e **ADD** para estabelecer as bases de como elas funcionam, especialmente na maneira como lidam com parâmetros (operandos). Assim, nos capítulos seguintes, podemos prosseguir em um ritmo mais rápido, pois encontramos o restante do conjunto de instruções ARM.

Antes de entrar nas instruções **MOV** e **ADD**, discutiremos a representação de **números negativos** e os conceitos de **deslocamento** e **rotação** de bits.

Números Negativos

No capítulo anterior, discutimos como os computadores representam inteiros positivos como números binários, chamados de inteiros sem sinal, mas e os números negativos? Nosso primeiro pensamento pode ser fazer com que 1 bit represente se o número é positivo ou negativo. Isso é simples, mas requer lógica extra para ser implementado, já que agora a CPU deve olhar para os bits de sinal, então decidir se adiciona ou subtrai e em qual ordem.

Sobre o Complemento de Dois

O grande matemático John von Neumann, do Projeto Manhattan, teve a ideia da representação **em complemento de dois** para números negativos, em 1945, quando trabalhava no Electronic Discrete Variable Automatic Computer (EDVAC) — um dos primeiros computadores eletrônicos.

Capítulo 2 Carregando e Adicionando

Considere um número hexadecimal de 1 byte como 01. Se somarmos

$$0x01 + 0xFF = 0x100$$

(todos os binários), obtemos 0x100.

No entanto, se estivermos limitados a números de 1 byte, o 1 será perdido e ficam com 00:

$$0x01 + 0xFF = 0x00$$

A definição matemática do negativo de um número é um número que, quando adicionado a ele, resulta em zero; portanto, matematicamente, FF é -1. Você pode obter a forma de complemento de dois para qualquer número tomando

$$2N - \text{número}$$

Em nosso exemplo, o complemento de dois de 1 é

$$28 - 1 = 256 - 1 = 255 = 0xFF$$

É por isso que é chamado de complemento de dois. Uma maneira mais fácil de calcular o complemento de dois é mudar todos os 1s para 0s e todos os 0s para 1s e então adicionar 1. Se fizermos isso para 1, obtemos

$$0xFE + 1 = 0xFF$$

O complemento de dois é uma esquisitice matemática interessante para números inteiros que são limitados a ter um valor máximo de um menor que uma potência de dois (que são todas as representações computacionais de números inteiros).

Por que queremos representar números inteiros negativos dessa maneira em computadores? Acontece que a adição é simples para o computador executar. Não há casos especiais; se você descartar o estouro, tudo dá certo. Isso significa que menos circuitos são necessários para realizar a adição e, como resultado, ela pode ser executada mais rapidamente. Além de manipular os sinais corretamente, isso também faz com que a CPU use a mesma lógica de adição para aritmética com e sem sinal, outra medida de economia de circuito. Considerar

$$5 + -3$$

3 em 1 byte é 0x03 ou 0000 0011.

A inversão dos bits é

1111 1100

Adicione 1 para obter

1111 1101 = 0xDF

Agora adicione

5 + 0xDF = 0x102 = 2

Uma vez que estamos limitados a 1 byte ou 8 bits.

Sobre a Calculadora do Programador Gnome

Felizmente, temos computadores para fazer as conversões e aritmética para nós, mas quando vemos números assinados na memória, precisamos reconhecê-los. A **calculadora do programador Gnome** pode calcular o complemento de dois para você. A Figura 2-1 mostra a calculadora Gnome representando -3.

Nota A calculadora do programador Gnome usa representações de 64 bits.

Capítulo 2 Carregando e Adicionando

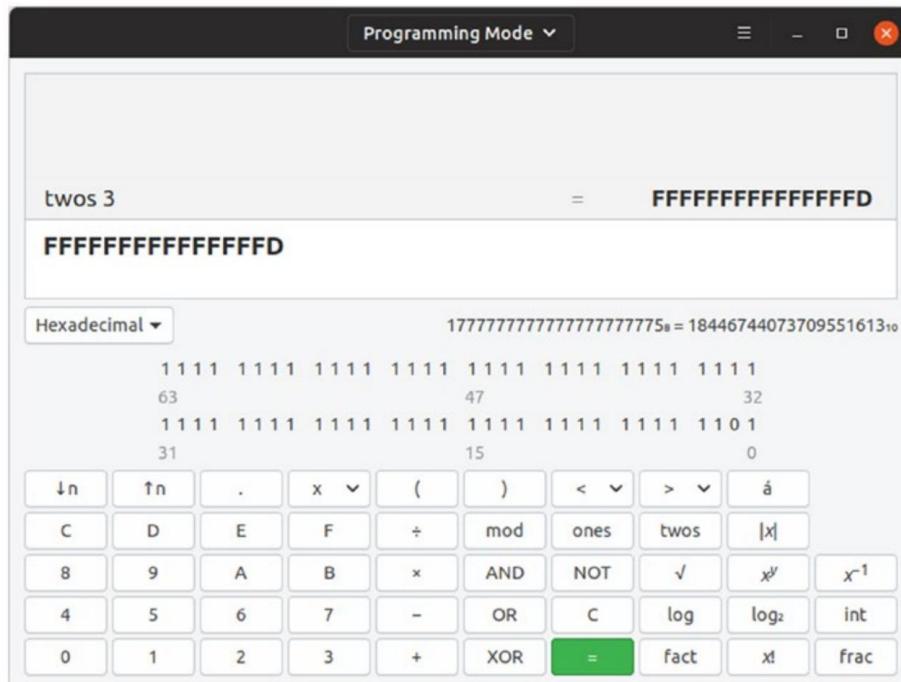


Figura 2-1. A calculadora Gnome calculando o complemento de dois de 3

Sobre o complemento de alguém

Se não adicionarmos 1 e apenas mudarmos todos os 1s para 0s e vice-versa, isso é chamado de **complemento de um**. Existem usos para a forma de complemento de um, e vamos encontrá-los em como algumas instruções processam seus operandos.

Big vs. Little-endian

No final do Capítulo 1, “Iniciando”, vimos que as palavras de nosso programa compilado tinham seus bytes armazenados na ordem inversa à que poderíamos esperar que fossem armazenados. De fato, se olharmos para uma representação de 1 de 32 bits armazenada na memória, é

01 00 00 00

em vez de

00 00 00 01

A maioria dos processadores escolhe um formato ou outro para armazenar números.

Os mainframes Motorola e IBM usam o que é chamado de big-endian, onde os números são armazenados na ordem do dígito mais significativo para o dígito menos significativo, neste caso

00 00 00 01

Os processadores Intel usam o formato little-endian e armazenam os números em ordem inversa com o dígito menos significativo primeiro, ou seja:

01 00 00 00

A Figura 2-2 mostra como os bytes em números inteiros são copiados para a memória nos formatos little e big-endian. Observe como os bytes terminam na ordem inversa entre si.

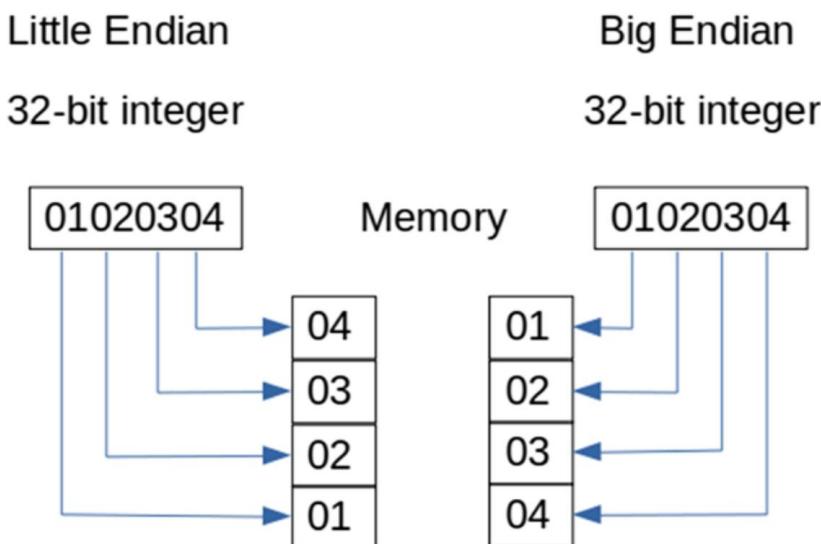


Figura 2-2. Como os números inteiros são armazenados na memória no formato Little vs. big endian

Capítulo 2 Carregando e Adicionando

Sobre o Bi-endian

A CPU ARM é chamada de **bi-endiana**, porque pode fazer qualquer um dos dois.

Há um sinalizador de status do programa no **CPSR** que diz qual endianidade usar.

Veremos todos os bits no **CPSR** um pouco mais tarde. Por padrão, o Raspbian e seus programas usam processadores Intel little-endian. Você pode mudar isso se quiser.

Veremos uma aplicação para alterar esse sinalizador em um capítulo posterior.

Prós de Little-endian

A vantagem do formato little-endian é que facilita a alteração do tamanho dos números inteiros, sem exigir nenhuma aritmética de endereço. Se você quiser converter um inteiro de 4 bytes em um inteiro de 1 byte, pegue o primeiro byte. Supondo que o número inteiro esteja no intervalo de 0 a 255 e os outros 3 bytes sejam zero.

Por exemplo, se a memória contém a representação de 4 bytes ou palavras para 1, em little-endian, a memória contém

01 00 00 00

Se quisermos a representação de 1 byte desse número, pegamos o primeiro byte; para a representação de 16 bits, tomamos os primeiros 2 bytes. O ponto chave é que o endereço de memória que usamos é o mesmo nos casos de chamada, economizando um ciclo de instrução ajustando-o.

Quando estivermos no depurador, veremos mais representações, e estas serão apontadas novamente à medida que as encontrarmos.

Observação Embora o Raspbian use little-endian, muitos protocolos como TCP/IP usados na Internet usam big-endian e, portanto, requerem uma transformação ao mover dados do Raspberry Pi para o mundo externo.

Deslocamento e Rotação

Temos 16 registradores de 32 bits, e grande parte da programação consiste em manipular os bits desses registradores. Duas manipulações de bit extremamente úteis são deslocamento e rotação. Deslocar matematicamente todos os bits à esquerda de um ponto é o mesmo que multiplicar por 2, e geralmente deslocar n bits é equivalente a multiplicar por 2^n . Por outro lado, deslocar bits para a direita em n bits é equivalente a dividir por 2^n .

Por exemplo, considere deslocar o número 3 à esquerda em 4 bits:

0000 0011 (a representação binária do número 3)

Desloque os bits restantes por 4 bits e obtemos

0011 0000

qual é

$$0x30 = 3 \times 16 = 3 \times 2^4$$

Agora, se deslocarmos 0x30 para a direita em 4 bits, desfazemos o que acabamos de fazer e vemos como é equivalente a dividir por 24.

Sobre Carregar Bandeira

No **CPSR**, há um bit para **carregar**. Isso normalmente é usado para realizar adição em números maiores. Se você adicionar dois números de 32 bits e o resultado for maior que 32 bits, o sinalizador de transporte será definido. Veremos como usar isso quando examinarmos a adição em detalhes mais adiante neste capítulo. Quando mudamos e giramos, acaba sendo útil incluir a bandeira de transporte. Isso significa que podemos fazer uma lógica condicional com base no último bit deslocado do registrador.

Capítulo 2 Carregando e Adicionando

Sobre o Barril Shifter

O processador ARM tem circuitos para deslocamento, chamados de **barril shifter**, mas não há instruções nativas para deslocamento ou rotação de bits; em vez disso, é feito como um efeito colateral de outras instruções, como a instrução **MOV** que estamos prestes a cobrir. A razão para isso é que o câmbio barril está fora da **Unidade Lógica Aritmética (ALU)** e, em vez disso, faz parte do circuito que carrega o segundo operando para uma instrução. Veremos isso em ação quando cobrirmos **Operand2** para a instrução **MOV**. A Figura 2-3 mostra a localização do trocador de tambor em relação à ALU.

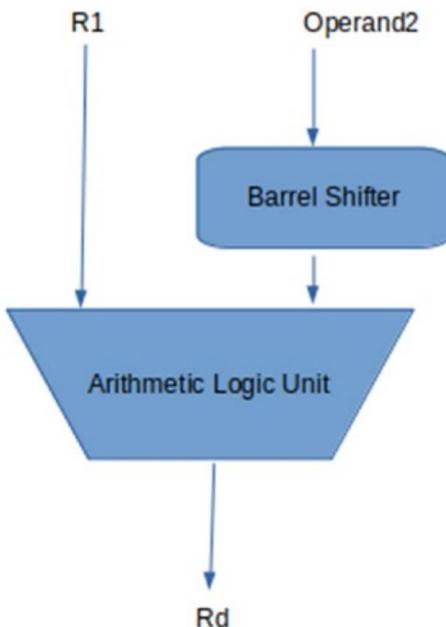


Figura 2-3. A localização do barril shifter para realizar turnos como parte do carregamento Operand2

Noções básicas de deslocamento e rotação

Temos cinco casos para cobrir, como segue:

1. Deslocamento lógico à esquerda
2. Deslocamento lógico para a direita
3. Deslocamento aritmético para a direita
4. Gire para a direita
5. Gire a extensão direita

Deslocamento Lógico à Esquerda

Isso é bastante direto, pois deslocamos os bits deixados pelo número indicado de casas e os zeros vêm da direita. O último bit deslocado termina na bandeira de transporte.

Deslocamento Lógico para a Direita

Igualmente fácil, aqui deslocamos os bits para a direita, os zeros vêm da esquerda e o último bit deslocado termina no sinalizador de transporte.

Deslocamento Aritmético para a Direita

O problema com o deslocamento lógico para a direita é que, se for um número negativo, ter um zero vindo da esquerda repentinamente torna o número positivo. Se quisermos preservar o bit de sinal, use o deslocamento aritmético para a direita. Aqui um 1 vem da esquerda, se o número for negativo, e um 0 se for positivo. Esta é a forma correta se você estiver deslocando números inteiros com sinal.

Vire à direita

Girar é como mudar, exceto que os bits não saem do final; em vez disso, eles se enrolam e reaparecem do outro lado. Assim, girar para a direita desloca para a direita, mas os bits que saem pela direita, reaparecem à esquerda.

Capítulo 2 Carregando e Adicionando

Girar para a direita Estender

Girar para a direita se comporta como girar para a direita, exceto que trata o registro como um registro de 33 bits, onde o sinalizador de transporte é o 33º bit e está à direita do bit 0. Esse tipo de rotação é limitado a mover 1 bit por vez ; portanto, o número de bits não é especificado na instrução.

MOV/MVN

Nesta seção, veremos várias formas da instrução MOV:

1. MOV RD, #imm16
2. MOVT RD, #imm16
3. MOV RD, RS
4. MOV RD, operando2
5. MVN RD, operando2

Vimos exemplos do primeiro caso, colocando um pequeno número em um registrador. Aqui, o valor imediato pode ser qualquer quantidade de 16 bits e será colocado nos 16 bits inferiores do registrador especificado. Esta forma da instrução **MOV** é tão simples quanto possível; portanto, vamos usá-lo com freqüência.

Sobre o MOVT

A segunda forma responde à nossa questão de como carregar os 32 bits completos de um registrador. **MOVT**, a instrução move top, carrega o operando imediato de 16 bits nos 16 bits superiores do registrador sem perturbar o

16 bits inferiores. Suponha que queremos carregar o registrador R2 com o valor hexadecimal 0x4F5D6E3A. nós poderíamos usar

```
MOV R2, #0x6E3A  
MOVT R2, #0x4F5D
```

Apenas duas instruções, portanto não muito dolorosas, mas um pouco irritantes.

Registre-se para registrar o MOV

No próximo caso 3, temos uma versão que move um registrador para outro isso soa útil.

O Temido Operando Flexível2

Todas as instruções de processamento de dados do ARM têm a opção de tomar um Operando2 flexível como um de seus parâmetros. Neste ponto, não ficará claro por que você deseja algumas dessas funcionalidades, mas à medida que encontrarmos mais instruções e começarmos a criar pequenos programas, veremos como elas nos ajudam. No nível de bit, há muita complexidade aqui, mas as pessoas que projetaram o Assembler fizeram um bom trabalho ao fornecer sintaxe para ocultar muito disso de nós. Ainda assim, ao fazer programação Assembly, é bom sempre saber o que está acontecendo nos bastidores.

Existem dois formatos para Operand2:

1. Um registro e uma mudança
2. Um pequeno número e uma rotação

Operand2 é processado através do barril shifter, é apenas uma questão do que é mudou e por quanto.

Capítulo 2 Carregando e Adicionando

Registre-se e mude

Primeiro, você pode especificar um registrador e um turno. Para isso, você especifica um registro que leva 4 bits e, em seguida, um deslocamento de 5 bits (para um total de um deslocamento completo de 32 bits). Por exemplo:

MOV R1, R2, LSL #1	@ Deslocamento lógico à esquerda
--------------------	----------------------------------

é como especificamos para pegar **R2**, deslocá-lo logicamente para a esquerda em 1 bit e colocar o resultado em **R1**. Podemos então lidar com os outros cenários de mudança e rotação que mencionado anteriormente com

MOV R1, R2, LSR #1	@ Deslocamento lógico para a direita
MOV R1, R2, ASR #1	@ Arithmetic deslocar para a direita
MOV R1, R2, ROR #1	@ Vire à direita
MOV R1, R2, RRX	@ Girar estendido para a direita

Como deslocamento e rotação são bastante comuns, o Assembler fornece mnemônicos para eles, para que você possa especificar

LSL R1, R2, #1	@ Deslocamento lógico à esquerda
LSR R1, R2, #1	@ Deslocamento lógico para a direita
ASR R1, R2, #1	@ Arithmetic deslocar para a direita
ROR R1, R2, #1	@ Vire à direita
RRX R1, R2	@ Girar estendido para a direita

Estes montam para o mesmo código de byte. A intenção é tornar o código um pouco mais legível, pois fica claro que você está fazendo uma operação de deslocamento ou rotação e não apenas carregando um registrador.

Número pequeno e rotação

Em segundo lugar, a outra forma do operando2 consiste em um pequeno número, ou seja, uma quantidade de 8 bits (1 byte) que pode ser rotacionada por um número par de posições, como RORs de 0, 2, 4, 8 , . . . , 30. Isso usa os 12 bits que temos

para operando2, 8 para o número e 4 para a rotação. Os valores que obtemos são assim:

- 0 - 255 [0 - 0xff]
- 256.260.264,...,1020 [0x100-0x3fc, etapa 4, 0x40-0xff
ror 30]
- 1024,1040,1056,...,4080 [0x400-0xffff, passo 16, 0x40-0xff
ror 28]
- 4096,4160, 4224,...,16320 [0x1000-0x3fc0, passo 64, 0x40-0xffff ror
26]

Este é um esquema bastante inteligente, pois permite representar qualquer potência de 2 de 0 a 31, para que você possa definir qualquer bit individual em um registrador. Ele também permite definir qualquer byte individual em um registro. Esses são cenários bastante frequentes e você pode especificá-los como parte da maioria das instruções de processamento de dados.

Felizmente, não precisamos descobrir tudo isso. Nós apenas especificamos um número e o Assembler descobre como representá-lo. Como existem apenas 12 bits, nem todos os números de 32 bits podem ser representados; portanto, se você especificar algo que não pode ser tratado, o Assembler fornecerá uma mensagem de erro. Você então precisa usar um par **MOV/MOV_T** conforme descrito anteriormente.

O **MOV** tem a vantagem de poder receber um operando **#imm16**, o que geralmente pode nos livrar de problemas. No entanto, outras instruções que devem especificar um terceiro registrador, como a instrução **ADD**, não têm esse luxo.

Freqüentemente, os programadores lidam com pequenos inteiros como índices de loop, digamos para fazer um loop de 1 a 10. Esses casos simples são tratados facilmente e não precisamos nos preocupar.

@ Grande demais para #imm16

MOV R1, #0xAB000000

@ Muito grande para #imm16 e não pode ser representado.

MOV R1, #0xABCD_EEF11

Capítulo 2 Carregando e Adicionando

A segunda instrução dá o erro

Erro: constante inválida (abcdef11) após correção

quando você executa seu programa através do Assembler. Isso significa que o Assembler tentou todos os seus truques e não conseguiu representar o número. Para carregá-lo, você precisa usar um par **MOV/MOVT**.

MVN

Esta é a instrução **Move Not**. Funciona exatamente como o **MOV**, exceto que inverte todos os 1s e 0s enquanto carrega o registrador. Isso significa que ele carrega o registro com a forma de complemento de um do que você especificou. Outra maneira de dizer isso é que ele aplica uma operação **lógica NOT** a cada bit na palavra que você está carregando no registrador.

MVT é um opcode distinto e não um alias para outra instrução com parâmetros enigmáticos. O conjunto de instruções ARM32 possui apenas 16 opcodes, portanto, esta é uma instrução importante com três usos principais:

1. Para calcular o complemento de algo para você. Isso tem seus usos, mas garante seu próprio opcode?

2. Multiplique por -1. Vimos que com as operações de turno podemos multiplicar ou dividir por potências de 2. Esta instrução nos leva a meio caminho da multiplicação por -1. Lembre-se de que o negativo de um número é o complemento de dois do número ou o complemento de um mais um. Isso significa que podemos multiplicar por -1 fazendo esta instrução e, em seguida, adicionar um. Por que faríamos isso em vez de usar a instrução **Multiply (MUL)**? O mesmo para deslocamento, por que fazer isso em vez de usar **MUL**? A resposta é que a instrução **MUL** é bastante lenta e pode demorar

alguns ciclos de clock para fazer seu trabalho. A mudança leva apenas um ciclo e, usando **MVN** e **ADD**, podemos multiplicar por -1 em apenas dois ciclos de clock. Multiplicar por -1 é muito comum, e agora podemos fazer isso rapidamente.

3. Você obtém o dobro do número de valores devido ao bit extra - 13×12 . Acontece que todos os números obtidos usando um valor de byte e deslocamento uniforme são diferentes para **MVN** e **MOV**. Isso significa que se o O Assembler vê que o número que você especificou não pode ser representado em uma instrução **MOV**, então ele tenta mudá-lo para uma instrução **MVN** e vice-versa. Portanto, você realmente tem 13 bits de dados imediatos, em vez de 12. OBSERVAÇÃO: ainda pode não ser capaz de representar seu número e você ainda pode precisar usar um par **MOV/MOVT**.

Exemplos de MOV

Nesta seção, escreveremos um pequeno programa para exercitar todas as instruções **MOV**. Crie um arquivo chamado

movexamps.s

contendo a Listagem 2-1.

Listagem 2-1. exemplos de MOV

@ @ Exemplos da instrução MOV. @ .global_start

 @ Forneça o endereço inicial do programa

 @ Carregue R2 com 0x4F5D6E3A primeiro usando MOV e MOVT

Capítulo 2 Carregando e Adicionando

```
_começar:      MOV R2, #0x6E3A  
                  MOVT R2, #0x4F5D
```

@ Apenas move R2 para R1

MOV R1, R2

@ Agora vamos ver todas as versões de turno do MOV

MOV R1, R2, LSL #1 @ Deslocamento lógico à esquerda

MOV R1, R2, LSR #1 @ Deslocamento lógico para a direita

MOV R1, R2, ASR #1 @Deslocamento aritmético para a direita

MOV R1, R2, ROR #1 @ Girar para a direita

MOV R1, R2, RRX @ Girar estendido para a direita

@ Repita os turnos acima usando @ os

mnemônicos Assembler.

LSL R1, R2, #1 @ Deslocamento lógico à esquerda

LSR R1, R2, #1 @ Deslocamento lógico para a direita

ASR R1, R2, #1 @ Arithmetic deslocar para a direita

ROR R1, R2, #1 @ Vire à direita

RRX R1, R2 @ Girar estendido para a direita

@ Exemplo que funciona com imediato e deslocamento de 8 bits

MOV R1, #0xAB000000 @ Grande demais para #imm16

@ Exemplo que não pode ser representado e @

resulta em um erro

@ Descomente a instrução se quiser @ ver o erro @

MOV R1, #0ABCDEF11 @ Grande demais para #imm16

@ Exemplo de MVN

MVN R1, n° 45

@ Exemplo de um MOV que o Assembler vai @ mudar para MVN

MOV R1, #0xFFFFFFFF @ (-2)

@ Defina os parâmetros para sair do programa @ e depois chame o Linux para fazer isso.

```
Movimento R0, #0      @ Use 0 código de retorno  
movimento R7, #1      @ Código de comando de serviço 1  
svc      0            @ Chamar          Linux para encerrar
```

Você pode compilar este programa com o arquivo de compilação

```
as -o movexamps.o movexamps.s ld -o  
movexamps movexamps.o
```

Você pode executar o programa depois de construí-lo.

Nota Este programa não faz nada além de mover vários números para registradores.

Veremos como ver o que está acontecendo no Capítulo 3, “Estruturando”, quando cobrirmos o GNU Debugger (GDB).

Se desmontarmos o programa usando

```
objdump -s -d movexamps.o
```

obtemos a Listagem 2-2.

Listagem 2-2. Desmontagem dos exemplos de MOV

Desmontagem da seção .text:

00000000 <_start>: 0:

```
e3062e3a movw r2, #28218 ; 0x6e3a 4: e3442f5d movt r2,  
#20317 ; 0x4f5d
```

Capítulo 2 Carregando e Adicionando

```
8: e1a01002 mov r1, r2 c:  
e1a01082 lsl r1, r2, #1 10: e1a010a2  
lsr r1, r2, #1 14: e1a010c2 asr r1, r2,  
#1 18: e1a010e2 ror r1, r2, #1 1c6:  
e1a0100 rrx r1, r2 20: e1a01082 lsl r1,  
r2, #1 24: e1a010a2 lsr r1, r2, #1 28:  
e1a010c2 asr r1, r2, #1 2c: e1a010e2  
ror r1, r2, #1 30: e1a01062,rrx rrx r2  
34: e3a014ab mov r1, #-1426063360 ;  
0xab000000 38: e3e0102d mvn r1,  
#45 ; 0x2d 3c: e3e01001 mvn r1, #1  
40: e3a00000 mov r0, #0 44: e3a07001 mov r7, #1 48:  
ef000000 svc 0x00000000
```

Todas as instruções começam com

0xe

isso significa sempre executar a instrução.

A maioria das instruções restantes tem

0x1a

como seus próximos dígitos. Os primeiros 3 bits são para o formato de instrução e são 0 significando

- Registro
- Registro
- Imediato

Temos então os 4 bits para o opcode. Todas as instruções **MOV** variantes têm isso como

1101

o opcode para **MOV**. Vemos que todas as operações de deslocamento são realmente instruções **MOV**, e o computador está tentando ser útil, informando-nos o que a instrução faz.

A instrução **MVN** tem um opcode de

1111

Isso inclui o **MVN** que colocamos em nosso arquivo de origem e a instrução **MOV** que o Assembler alterou para **MVN** para que pudesse carregar -2.

As duas primeiras instruções que carregam operandos de 16 bits são diferentes. Observe que o Assembler mudou nosso primeiro **MOV** para uma instrução **Move Wide (MOVW)**. Eles não fazem parte das instruções de processamento de dados que estamos vendo agora e são casos especiais, mas são úteis.

ADICIONAR/ADC

Agora podemos colocar qualquer valor que quisermos em um registrador, então vamos começar a fazer alguns cálculos. Vamos começar com a adição. As instruções que abordaremos são

1. ADD{S} Rd, Rs, Operando2
2. ADD{S} Rd, Rs, #imm12
3. ADD{S} Rd, Rs1, Rs2
4. ADC{S} Rd, Rs, Operando2
5. ADC{S} Rd, Rs1, Rs2

Capítulo 2 Carregando e Adicionando

Todas essas instruções adicionam seu segundo e terceiro parâmetros e colocam o resultado em seu primeiro parâmetro, **Register Destination (Rd)**. Já sabemos o seguinte:

- Registros
- Operando2
- #imm12

Empurrar essas coisas com as instruções **do MOV** foi difícil, mas está feito. O caso com três registradores é um caso especial de Operando2, apenas com deslocamento de 0 aplicado. Os registradores Rd e **Source Register (Rs)** podem ser iguais. Se você deseja apenas adicionar 1 a R1, pode especificar

ADICIONE R1, nº 1

O Assembler compila isso como

ADICIONE R1, R1, #1

Isso economiza um pouco de digitação e é um pouco mais claro. Este é um cenário comum para incrementar contadores de loop.

Ainda não desenvolvemos o código para imprimir um número, pois devemos primeiro converter o número em uma string ASCII. Chegaremos a isso depois de cobrirmos **loops** e **declarações condicionais**. Enquanto isso, podemos obter um número de nosso programa por meio do código de retorno do programa. Este é um inteiro sem sinal de 1 byte. Vejamos um exemplo de multiplicação de um número por -1 e vejamos o resultado. A Listagem 2-3 é o código para fazer isso.

Listagem 2-3. Um exemplo de MVN e ADD

```
@ @ Exemplo das instruções ADD/ADC.
@ .global_start
@ Forneça o endereço inicial do programa

@ Multiplique 2 por -1 usando MVN e adicionando 1
```

```
_início: MVN          R0, #2
                    ADICIONE R0, nº 1
```

- @ Defina os parâmetros para sair do programa @ e depois chame o Linux para fazer isso.
@ R0 é o código de retorno e será o que @ calculamos acima.

movimento	R7, #1	@ Código de comando de serviço 1
SVC	0	@ Chamar Linux para encerrar

Aqui usamos a instrução **MVN** para calcular o complemento de um do nosso número, neste caso 2, então adicionamos 1 para obter a forma de complemento de dois. Usamos **R0**, pois este será o código de retorno retornado pelo Linux terminar comando. Para ver o código de retorno, digite

```
echo $?
```

depois de executar o programa, ele imprime 254. Se você examinar os bits, verá que esta é a forma de complemento de dois para -2 em 1 byte.

Adicionar com transporte

Os novos conceitos nesta seção são o que o **{S}** após a instrução significa junto com o motivo pelo qual temos **ADD** e **ADC**. Este será nosso primeiro uso do **CPSR**.

Pense em como aprendemos a somar números:

$$\begin{array}{r} 17 \\ +78 \\ \hline 95 \end{array}$$

Capítulo 2 Carregando e Adicionando

1. Primeiro adicionamos $7 + 8$ e obtemos 15.
2. Colocamos 5 em nossa soma e elevamos o 1 às dezenas coluna.
3. Agora somamos $1 + 7 +$ o carry da coluna das unidades, então somamos $1+7+1$ e obtemos 9 para a coluna das dezenas.

Esta é a ideia por trás da bandeira de transporte. Quando uma adição estoura, ela define o sinalizador de carry, para que possamos incluí-lo na soma da próxima parte.

NOTA: Um carry é sempre 0 ou 1, então só precisamos de um sinalizador de 1 bit para isso.

O processador ARM adiciona 32 bits por vez, então só precisamos do carry sinalizador se estivermos lidando com números maiores que cabem em 32 bits. Isso significa que, embora estejamos no modo de 32 bits, podemos facilmente adicionar números inteiros de 64 bits ou ainda maiores.

No Capítulo 1, "Introdução", mencionamos rapidamente que o bit 20 no formato da instrução especifica se uma instrução altera o **CPSR**. Até agora, não definimos esse bit, portanto nenhuma das instruções que escrevemos até agora alterará o **CPSR**. Se quisermos que uma instrução altere o **CPSR**, colocamos um "**S**" no final do opcode e o Assembler definirá o bit 20 quando criar a versão binária da instrução. Isso se aplica a todas as instruções, incluindo as instruções **MOV** que acabamos de ver.

ADIÇÕES R0, #1

é como

ADICIONE R0, nº 1

exceto que define vários bits no **CPSR**. Abordaremos todas as partes quando abordarmos **as instruções condicionais**. Por enquanto, estamos interessados no sinalizador de transporte designado C. Se o resultado de uma adição for muito grande, o sinalizador C é definido como 1; caso contrário, é definido como 0.

Para somar dois inteiros de 64 bits, use dois registradores para armazenar cada número.

Em nosso exemplo, usaremos os registradores **R2** e **R3** para o primeiro número, **R4** e **R5** para o segundo e depois **R0** e **R1** para o resultado. O código seria então

```
ADICIONA R1, R3, R5 @ Palavra de ordem inferior
ADC R0, R2, R4 @ Palavra de ordem superior
```

O primeiro **ADDS** adiciona os 32 bits de ordem inferior e define o sinalizador de transporte, se necessário. Ele pode definir outros sinalizadores no CPSR, mas nos preocuparemos com eles mais tarde. A segunda instrução, **ADDC**, adiciona as palavras de ordem superior, mais o sinalizador de transporte.

O bom aqui é que, embora estejamos no modo de 32 bits, ainda podemos fazer uma adição de 64 bits em apenas dois ciclos de clock. Vejamos um exemplo completo simples na Listagem 2-4.

Listagem 2-4. Exemplo de adição de 64 bits com ADD e ADC

```
@ @ Exemplo de adição de 64 bits com @
@ .global _start
```

@ Forneça o endereço inicial do programa

```
@ Carregue os registradores com alguns dados
@ O primeiro número de 64 bits é 0x00000003FFFFFFF _start:
```

```
MOV R2, #0x00000003
```

```
MOV R3, #0xFFFFFFFF @ as
```

vai mudar para MVN

```
@ O segundo número de 64 bits é 0x0000000500000001
```

```
MOV R4, #0x00000005
```

```
MOV R5, #0x00000001
```

```
ADIÇÕES R1, R3, R5
```

@ Palavra de ordem inferior

```
ADC R0, R2, R4
```

@ Palavra de ordem superior

Capítulo 2 Carregando e Adicionando

@ Defina os parâmetros para sair do programa @ e depois chame o Linux para fazer isso.

@ R0 é o código de retorno e será o que @ calculamos acima.

movimento	R7, #1	@ Código de comando de serviço 1	
svc	0	@ Chamar	Linux para encerrar

Aqui estamos adicionando

00000003 FFFFFFFF

00000005 00000001

00000009 00000000

Nós manipulamos este exemplo para demonstrar a bandeira de transporte e para produzir uma resposta que podemos ver no código de retorno. O maior inteiro sem sinal é

0xFFFFFFFF

e somar 1 resulta em

0x100000000

que não cabe em 32 bits, então obtemos

0x00000000

com um porte. As palavras de ordem superior adicionam 3 + 5 + carregam para resultar em 9. A palavra de ordem superior está em R0, portanto, é o código de retorno quando o programa é encerrado. Se digitarmos

echo \$?

obtemos 9 como esperado.

Aprender sobre **o MOV** foi difícil, porque era a primeira vez; encontramos shifting e Operand2. Com isso atrás de nós, aprender sobre **ADD** foi muito mais fácil. Ainda temos alguns tópicos complicados para cobrir, mas à medida que nos tornamos mais experientes em como manipular bits e bytes, o aprendizado deve se tornar mais fácil.

Resumo

Neste capítulo, aprendemos como inteiros negativos são representados em um computador. Passamos a discutir a ordenação de bytes big vs. little-endian. Em seguida, examinamos o conceito de deslocamento e rotação dos bits em um registrador.

Em seguida, examinamos em detalhes a instrução **MOV** que nos permite mover dados pelos registradores da CPU ou carregar constantes da instrução **MOV** em um registrador. Descobrimos os truques do operando² sobre como o ARM representa um grande intervalo de valores, dado o número limitado de bits que tem à sua disposição.

Por fim, cobrimos as instruções **ADD** e **ADC** e discutimos como para adicionar números de 32 e 64 bits.

No Capítulo 3, “Aprimoramento de ferramentas”, veremos maneiras melhores de construir nossos programas e começar a depurar nossos programas com o GNU Debugger (**gdb**).

CAPÍTULO 3

Preparando-se

Neste capítulo, aprenderemos uma maneira melhor de construir nossos programas usando o **GNU Make**. Com o GNU Debugger (GDB), iremos depurar nossos programas. E apresentaremos rapidamente o sistema de controle de origem **Git** e o servidor de compilação **Jenkins**.

GNU Make

Construímos nossos programas usando um script de shell simples para executar o **GNU Assembler** e, em seguida, o **vinculador/carregador do Linux**. À medida que avançamos, queremos uma ferramenta mais sofisticada para construir nossos programas. O **GNU Make** é o utilitário padrão do Linux para fazer isso e vem pré-instalado com o Raspbian. Em **GNU Make**

1. Especifique as regras de como construir algo a partir de outro.
2. O **GNU Make** examina as datas/horas do arquivo para determinar o que precisa ser construído.
3. **GNU Make** emite os comandos para construir o componentes.

Vamos ver como construir nosso programa `HelloWorld` do Capítulo 1, "Iniciando," usando **make**. Primeiro, crie um arquivo de texto chamado **makefile** contendo o código da Listagem 3-1.

Capítulo 3 Preparando-se

Listagem 3-1. Makefile simples para HelloWorld

```
HelloWorld: HelloWorld.o  
    ld -o HelloWorld HelloWorld.o  
  
HelloWorld.o: HelloWorld.s  
    as -o HelloWorld.o HelloWorld.s
```

Observação O comando make é específico e as linhas recuadas devem começar com uma tabulação, não espaços, ou você obterá um erro.

Para construir nosso arquivo, digite

fazer

Reconstruindo um arquivo

Se já construímos o programa, isso não fará nada, pois make vê que o executável é mais antigo que o arquivo .o e que o arquivo .o é mais antigo

do que o arquivo .s . Podemos forçar uma reconstrução digitando

fazer -B

Em vez de especificar cada arquivo separadamente junto com o comando para criá-lo, podemos definir uma regra de compilação para, digamos, criar um arquivo .o a partir de um arquivo .s .

Uma regra para criar arquivos .s

A Listagem 3-2 mostra uma versão mais avançada, na qual definimos uma regra para construir um arquivo .o a partir de um arquivo .s . Ainda precisamos especificar a dependência, mas não precisamos mais da regra de compilação. À medida que nos tornamos mais sofisticados e adicionamos parâmetros de linha de comando ao comando as, agora centralizamos o local para fazer isso.

Listagem 3-2. Hello World makefile com uma regra

```
% .o: %.s  
    como $< -o $@  
  
HelloWorld: HelloWorld.o  
    ld -o HelloWorld HelloWorld.o
```

Agora make sabe como criar um arquivo .o a partir de um arquivo .s . Nós dissemos para fazer para construir **HelloWorld** de **HelloWorld.o** e fazer pode olhar para sua lista de regras para descobrir como construir **HelloWorld.o**. Existem alguns símbolos estranhos neste arquivo, e seu significado é

- **%.s** é como um curinga, significando qualquer arquivo .s .
- **\$<** é um símbolo para o arquivo fonte.
- **\$@** é um símbolo para o arquivo de saída.

Há muita documentação boa sobre **o make**, então não vamos em muitos detalhes aqui.

Definindo Variáveis

A Listagem 3-3 mostra como definir variáveis. Aqui faremos isso para centralizar a lista de arquivos que queremos montar.

Listagem 3-3. Adicionando uma variável ao makefile Hello World

```
OBJS = HelloWorld.o  
  
% .o: %.s  
    como $< -o $@  
  
HelloWorld: $(OBJS) ld -o  
    HelloWorld $(OBJS)
```

Capítulo 3 Preparando-se

Com este código, à medida que adicionamos arquivos de origem, apenas adicionamos o novo arquivo ao **OBJS=** line e **make** cuida do resto.

Esta é apenas uma introdução ao GNU Make—há muito mais sobre isso ferramenta poderosa. À medida que avançamos no livro, introduziremos novos elementos em nossos **makefiles** conforme necessário.

GDBName

A maioria das linguagens de alto nível vem com ferramentas para enviar facilmente quaisquer strings ou números para o console, uma janela ou uma página da web. Freqüentemente, ao usar essas linguagens, os programadores não se preocupam em usar o depurador; em vez disso, eles dependem de bibliotecas que fazem parte da linguagem.

Mais adiante, veremos como aproveitar as bibliotecas que fazem parte de outras linguagens, mas chamá-las dá um pouco de trabalho. Também desenvolveremos uma biblioteca útil para converter números em strings, para que possamos usar as técnicas usadas no capítulo 1 “HelloWorld” para imprimir nosso trabalho.

Ao fazer programação em linguagem Assembly, ser proficiente com o depurador é fundamental para o sucesso. Isso não apenas ajudará com sua programação em linguagem Assembly, mas também será uma ótima ferramenta para você usar com sua programação em linguagem de alto nível.

Preparando para depurar

GDB pode depurar seu programa como está, mas esta não é a maneira mais conveniente de ir. Por exemplo, em nosso programa HelloWorld temos a string **helloworld**. Se depurarmos o programa como está, o depurador não saberá nada sobre esse rótulo, pois o Assembler o transformou em um endereço em uma seção .data. Existe uma opção de linha de comando para o Assembler que inclui uma tabela de todos os nossos rótulos e símbolos de código-fonte, para que possamos usá-los no depurador. Isso torna nosso programa executável um pouco maior.

Freqüentemente, definimos um sinalizador de depuração enquanto desenvolvemos o programa e, em seguida, remova o sinalizador de depuração antes de liberar o programa. Ao contrário de algumas linguagens de programação de alto nível, o sinalizador de depuração não afeta o código de máquina gerado, portanto, o programa se comporta exatamente da mesma forma nos modos de depuração e não depuração.

Não queremos deixar as informações de depuração em nosso programa para lançamento, porque além de tornar o programa executável maior, é uma riqueza de informações para hackers para ajudá-los a fazer engenharia reversa de seu programa. Houve vários casos em que os hackers causaram danos porque o programa ainda tinha informações de depuração presentes.

Para adicionar informações de depuração ao nosso programa, devemos montá-lo com o sinalizador **-g**. Na Listagem 3-4, adicionamos um sinalizador de depuração ao nosso **makefile**. Para o primeiro programa que iremos depurar, vamos usar nossos exemplos das instruções **MOV**, já que não vimos as operações funcionando nos vários registradores.

Listagem 3-4. Makefile com um sinalizador de depuração

```
OBJS = movexamps.o ifdef  
DEBUG  
DEBUGFLGS = -g  
senão  
DEBUGFLGS =  
fim se  
  
%.o: %.s  
    as $(DEBUGFLGS) $< -o $@  
  
movexamps: $(OBJS) ld -o  
    movexamps $(OBJS)
```

Este **makefile** define o sinalizador de depuração se a variável **DEBUG** for definida. Podemos defini-lo na linha de comando para **make** com

fazer DEBUG = 1

Capítulo 3 Preparando-se

Ou, na linha de comando, defina uma variável de ambiente com
exportar DEBUG=1

Para limpar a variável de ambiente, digite
exportar DEBUG=

Ao alternar entre **DEBUG** e não **DEBUG**, execute make com
a opção -B para compilar tudo.

Dica Freqüentemente, eu crio para shell scripts **buildd** e **buildr** para
chamar make com e sem **DEBUG** definido.

Começando GDB

Para começar a depurar nosso programa **movexamps** , digite o comando

gdb movexamps

Isso produz a saída abreviada

GNU gdb (Raspbian 7.12-6) 7.12.0.20161007-git Copyright (C)
2016 Free Software Foundation, Inc.

...

Leitura de símbolos de movexamps...concluído. (gdb)

- **Gdb** é um programa de linha de comando.
- **(gdb)** é o prompt de comando onde você digita os
comandos.
- **(aba hit)** para conclusão do comando. Digite a primeira letra ou duas
de um comando como um atalho.

Capítulo 3 Preparando-se

Para executar o programa, digite

correr

(ou r).

O programa é executado até a conclusão, como se fosse executado normalmente a partir do linha de comando.

Para listar nosso programa, digite

lista

(ou eu).

Isso lista dez linhas. Tipo

eu

para as próximas dez linhas. Tipo

lista 1.1000

para listar todo o nosso programa.

Observe que a lista nos fornece o código-fonte do nosso programa, incluindo comentários.

Esta é uma maneira prática de encontrar números de linha para outros comandos.

Se quisermos ver o código bruto da máquina, podemos fazer com que o gdb desmonte nosso programa com

desmontar _start

Isso mostra o código real produzido pelo Assembler sem comentários. Podemos ver se MOV ou MVN foi usado entre outros comandos dessa maneira.

Para parar o programa, definimos um ponto de interrupção. Nesse caso, queremos parar o programa no início para um único passo, examinando os registradores à medida que avançamos. Para definir um ponto de interrupção, use o comando **breakpoint** (ou b):

b_start

Capítulo 3 Preparando-se

Podemos especificar um número de linha ou um símbolo para nosso ponto de interrupção. Como neste exemplo, agora se rodarmos o programa, ele para no breakpoint:

```
(gdb) b _start
Breakpoint 1 em 0x10054: arquivo movexamps.s, linha 8. (gdb) r
Iniciando o programa: /home/pi/asm/Chapter 2/movexamps
```

Ponto de interrupção 1, _start () em movexamps.s:8
`_start: MOV R2, #0x6E3A (gdb)`

Agora podemos percorrer o programa com o comando **step** (ou **s**). À medida que avançamos, queremos ver os valores dos registradores. Nós os obtemos com **registradores de informação** (ou **i r**):

```
(gdb) s
9          MOVT R2, #0x4F5D
(gdb) ir
r0          0x0      0
r1          0x0      0
r2          0x6e3a 28218
r3          0x0      0
r4          0x0      0
r5          0x0      0
r6          0x0      0
r7          0x0      0
r8          0x0      0
r9          0x0      0
r10         0x0      0
r11         0x0      0
r12         0x0      0
sp          0x7efff040 0x7efff040
lr          0x0      0
```

```
pc           0x10058      0x10058 <_start+4>
cpsr        0x10 16
(gdb)
```

Vemos **0x6E3A** colocado em **R2** como esperado.

Podemos continuar avançando ou digitar **continue** (ou c) para continuar para o próximo **ponto de interrupção** ou para o final do programa. Podemos definir quantos pontos de interrupção quisermos. Podemos vê-los todos com o comando **info breakpoints** (ou i b). Podemos excluir um ponto de interrupção com o comando **delete**, especificando o número do ponto de interrupção a ser excluído.

```
(gdb) ib
Num Tipo Disp Enb Ponto de interrupção de endereço      O que
1          manter y 0x00010054 ponto de interrupção já atingido 1 vez (gdb)
          excluir 1 (gdb) ib Sem pontos de interrupção ou pontos de
controle. (gdb)
```

Não lidamos muito com a memória, mas o **gdb** possui bons mecanismos para exibir a memória em diferentes formatos. O comando principal sendo x. Tem o formato

x /Nfu addr

onde

- **N** é o número de objetos a serem exibidos
- **f** é o formato de exibição onde alguns comuns são
 - t para binário
 - x para hexadecimal
 - d para decimal

Capítulo 3 Preparando-se

- **u** para instrução
- **s** para string
- **u** é o tamanho da unidade e é qualquer um dos
 - **b** para bytes
 - **h** para meias palavras (16 bits)
 - **w** para palavras (32 bits)
 - **g** para palavras gigantes (64 bits)

Alguns exemplos usando nosso código armazenado na localização da memória `_start`, ou `0x10054`:

```
(gdb) x /4ubft _start 0x10054
<_start>: 00111010 00101110 00000110
               11100011

(gdb) x /4ubfi _start => 0x10054
<_start>: movw r2, #28218 ; 0x6e3a movt r2, #20317 ; 0x4f5d 0x10058
0x10060 <_start+12>;           <_start+4>: 0x1005c <_start+8>:
                      mov r1, r2 lsl r1,
                      r2, #1

(gdb) x /4ubfx _start 0x10054
<_start>: (gdb) x /4ubfd _start    0x3a 0x2e 0x06 0xe3
0x10054 <_start>:
            58      46      6       -29
```

Para sair do **gdb**, digite **q** (para sair ou digite **control-d**).

A Tabela 3-1 fornece uma referência rápida aos comandos GDB que apresentamos neste capítulo. À medida que aprendemos coisas novas, precisamos aumentar nosso conhecimento de **gdb**. É uma ferramenta poderosa para nos ajudar a desenvolver nossos programas. Os programas em linguagem assembly são complexos e sutis, e o **gdb** é ótimo para nos mostrar o que está acontecendo com todos os bits e bytes.

Tabela 3-1. Resumo dos comandos úteis do GDB

Comando (forma abreviada)	Descrição
quebrar (b) linha	Definir ponto de interrupção na linha
correr (r)	Execute o programa
passos)	Programa de passo único
continuar (c)	Continue executando o programa
quit (q ou control-d)	Sair do gdb
controle-c	Interromper o programa em execução
registradores de informação (ir)	Imprima os registros
quebra de informação	Imprima os pontos de interrupção
excluir n	Excluir ponto de interrupção n
x /Nuf expressão	Mostrar conteúdo da memória

Vale a pena percorrer nossos três programas de exemplo e examinar os registradores em cada etapa para garantir que você entenda o que cada instrução está fazendo.

Mesmo que você não saiba de um bug, muitos programadores gostam de através de seu código para procurar problemas e se convencer de que seu código é bom. Frequentemente, dois programadores fazem isso juntos como parte da metodologia ágil de programação em pares.

Controle de Origem e Servidores de Construção

git

À medida que seu programa fica maior, considere usar um sistema de controle de origem para gerenciar arquivos de origem. Os sistemas de controle de origem mantêm todas as versões do seu programa. Com o controle de origem, é fácil recuperar os arquivos que compõem

Capítulo 3 Preparando-se

versão 1.15 do seu programa; você pode ter várias ramificações, então você pode trabalhar na versão 1.16 enquanto também trabalha na versão 2.1 e manter tudo em ordem.

Depois de ter uma equipe de programadores trabalhando em seu projeto, você precisamos regular quem está editando o quê, para que as pessoas não substituam o trabalho umas das outras. **O Git** leva isso a um novo nível, onde duas pessoas podem editar o mesmo arquivo, então o Git pode mesclar as alterações para manter o trabalho de ambas as pessoas. Git é um ótimo programa para fazer isso. O Git foi desenvolvido por Linus Torvalds como o sistema de controle de origem para todo o desenvolvimento do Linux. Existem versões em nuvem, como o GitHub, que mantêm seus arquivos na nuvem e, portanto, você não precisa se preocupar em fazer backup deles.

Observação Os cartões SD que o Raspberry Pi usa em vez de discos rígidos ou SSDs não são tão confiáveis. Eles podem falhar, então você sempre deve ter um backup do seu trabalho. Se você não fizer backup na nuvem com um serviço como o Github, faça o backup com um dos seguintes:

- Copie seus arquivos para o Google Drive.
- Envie seus arquivos por e-mail para você mesmo.
- Copie-os para um disco rígido USB.

Não confie no cartão SD, pois ele falhará em algum momento.

Git é um sistema sofisticado além do escopo deste livro, mas vale a pena conferir.

Jenkins

Depois de usar o GNU Make e o Git, considere verificar **o Jenkins**. Jenkins é um servidor de compilação que monitora o Git e toda vez que você faz o check-in de uma nova versão de um arquivo de programa, ele inicia uma compilação. Isso faz parte de um sistema de desenvolvimento contínuo que pode até mesmo implantar seu programa.

Isso é especialmente útil se você tiver uma equipe de programadores, onde a compilação leva muito tempo ou você precisa que o resultado seja implantado automaticamente, digamos em um servidor da web.

Se você tiver um conjunto de testes automatizados, eles serão executados após cada compilação. Ter os testes automatizados executados com frequência ajuda a detectar quando seu programa está quebrado. O custo de correção de um bug tende a ser proporcional ao tempo que o bug existe no código, portanto, encontrar e corrigir bugs rapidamente é um grande ganho de produtividade.

Resumo

Neste capítulo, apresentamos o programa GNU Make que usaremos para construir nossos programas. Essa é uma ferramenta poderosa usada para lidar com todas as regras dos vários compiladores e vinculadores de que precisamos.

Em seguida, apresentamos o GNU Debugger que nos permitirá solucionar problemas de nossos programas. Infelizmente, os programas têm bugs e precisamos de uma maneira de passar por eles e examinar todos os registros e a memória enquanto o fazemos. O GDB é uma ferramenta técnica, mas indispensável para descobrir o que nossos programas estão fazendo.

Por fim, mencionamos o sistema de controle de origem Git e o servidor de compilação Jenkins. Não os usaremos neste livro, mas à medida que suas necessidades forem ficando mais sofisticadas, você deve dar uma olhada neles.

No Capítulo 4, “Controlando o Fluxo do Programa”, veremos execução de código, ramificação e loop - os principais blocos de construção da lógica de programação.

CAPÍTULO 4

programa de controle Fluxo

Agora conhecemos um punhado de instruções em linguagem Assembly e podemos executá-las linearmente uma após a outra. Aprendemos como iniciar e encerrar um programa. Construímos programas e os depuramos.

Neste capítulo, tornaremos nossos programas mais interessantes usando lógica condicional — instruções if/then/else — em linguagem de alto nível. Também apresentaremos loops — instruções for e while — em linguagens de alto nível. Com essas instruções em mãos, teremos todo o básico para codificar a lógica do programa.

ramificação incondicional

A instrução de desvio mais simples é
rótulo B

que é uma ramificação incondicional para um rótulo. O rótulo é interpretado como um deslocamento do registro atual do PC e possui 24 bits na instrução, permitindo um intervalo de 8 megapalavras em qualquer direção ou um salto de até 32 MB em qualquer direção. Esta instrução é como uma instrução goto em algumas linguagens de alto nível.

Capítulo 4 Controlando o Fluxo do Programa

Se codificarmos a Listagem 4-1, o programa estará em um loop fechado e travará nossa janela de terminal até que pressionemos **Control + C**.

Listagem 4-1. Uma instrução de desvio de loop fechado

```
_começar:      MOV R1, nº 1
                B _start
```

Sobre o CPSR

Mencionamos várias vezes o Current Program Status Register (CPSR) sem realmente examinar o que ele contém. Falamos sobre a bandeira de transporte quando examinamos as instruções ADDS/ADC. Nesta seção, veremos mais alguns sinalizadores no CPSR.

Começaremos listando todos os sinalizadores que ele contém, embora muitos deles não sejam discutidos até os capítulos posteriores. Neste capítulo, estamos interessados no grupo de bits de código de condição que nos dizem coisas sobre o que acontece quando uma instrução é executada (Figura 4-1).

31	30	29	28	27	-	24	-	19 – 16	-	9	8	7	6	5	4 – 0
N	Z	C	V	Q		J		GE		E	A	I	F	T	M

Figura 4-1. Os bits no CPSR

Os sinalizadores de condição são

- Negativo: N é 1 se o valor com sinal for negativo e apagado se o resultado for positivo ou 0.
- Zero: É definido se o resultado for 0; isso geralmente denota um resultado igual de uma comparação. Se o resultado for diferente de zero, este sinalizador é apagado.

Capítulo 4 Controlando o Fluxo do Programa

- **Carry:** Para operações do tipo adição, este sinalizador é definido se o resultado produzir um estouro. Para operação do tipo subtração, este sinalizador é definido se o resultado exigir um empréstimo.

Além disso, é usado no deslocamento para manter o último bit deslocado.

- **Overflow:** Para adição e subtração, este sinalizador é definido se ocorreu um estouro assinado. NOTA: Algumas instruções podem definir especificamente o oVerflow para sinalizar uma condição de erro.

Os sinalizadores de interrupção são

- I: Quando definido, desativa as interrupções de IRQ
- F: Quando definido, desabilita as interrupções do FIQ
- A: Quando definido, desabilita abortos imprecisos

Os sinalizadores do conjunto de instruções são

- **Thumb:** instruções compactas de 16 bits
- **Jazelle:** Modo obsoleto para execução direta de bytecodes Java

Os outros bits são

- Q: Este sinalizador é definido para indicar subfluxo e/ou saturação.
- **GE:** Esses sinalizadores controlam o comportamento Maior ou Igual nas instruções SIMD.
- E: É um sinalizador que controla o “endianness” para tratamento de dados.

M é o modo do processador, como usuário ou supervisor.

Capítulo 4 Controlando o Fluxo do Programa

Filial na condição

A instrução de desvio, no início deste capítulo, pode receber um modificador que o instrui a desviar apenas se um determinado sinalizador de condição no CPSR for ativado ou desativado.

A forma geral das instruções de desvio é

B{condição} rótulo

onde {condição} é retirada da Tabela 4-1.

Tabela 4-1. Códigos de condição para a instrução de desvio

{doença}	Bandeiras	Significado
equifator	conjunto Z	Igual
NE	Z claro	Não igual
CS ou HS	conjunto C	Superior ou igual (sem sinal \geq)
CC ou LO	C claro	Inferior (sem sinal $<$)
MI	conjunto N	Negativo
PL	N claro	Positivo ou zero
VS	conjunto V	Transbordar
VC	V claro	Sem estouro
OI	C definido e Z claro	Superior (sem sinal $>$)
LS	C claro e Z definido	Inferior ou igual (sem sinal \leq)
GE	N e V iguais	Assinado \geq
LT	N e V diferem	Assinado $<$
GT	Z claro, N e V iguais	Assinado $>$
LE	Conjunto Z, N e V diferem	Assinado \leq
AL	Qualquer	Sempre (o mesmo que sem sufixo)

Por exemplo:

BEQ_start

irá ramificar para _start se o sinalizador Z estiver definido. Isso parece um pouco estranho, por que a instrução BZ para desvio não está em zero? O que é igual aqui? Para responder a essas perguntas, precisamos examinar a instrução CMP.

Sobre a Instrução CMP

O formato da instrução CMP é

CMP Rn, Operando2

Esta instrução compara o conteúdo do registrador **Rn** com **Operand2** subtraindo Operand2 de Rn e atualizando os sinalizadores de status de acordo. Ele se comporta exatamente como a instrução **SUBS** (que é apenas como a instrução **ADDS**, faz subtração em vez de adição), exceto que apenas atualiza os sinalizadores de status e descarta o resultado. Por exemplo, para fazer uma ramificação somente se o registrador **R4** for 45, podemos codificar

CMP R4, nº 45

BEQ_start

Nesse contexto, vemos como o mnemônico BEQ faz sentido; como CMP subtrai 45 de **R4**, o resultado é zero se forem iguais e o sinalizador **Z** será definido. Se você voltar à Tabela 4-1 e considerar os códigos de condição nesse contexto, eles farão sentido.

rotações

Com as instruções de ramificação e comparação em mãos, vamos ver como construir alguns loops modelados com base no que encontramos em linguagens de programação de alto nível.

Capítulo 4 Controlando o Fluxo do Programa

FOR Loops

Suponha que queremos fazer o loop for básico

PARA I = 1 a 10

...algumas declarações...

PROXIMO EU

Podemos implementar isso como mostrado na Listagem 4-2.

Listagem 4-2. Básico para loop

MOV R2, #1 @ R2 mantém I loop:

@ corpo do loop vai aqui.

@ A maior parte da lógica está no final

ADICIONE R2, nº 1 @ eu = eu + 1

CMP R2, nº 10

Círculo BLE @ IF I <= 10 goto loop

Se fizéssemos isso contando

PARA I = 10 PARA 1 PASSO -1

...algumas declarações...

PROXIMO EU

Podemos implementar isso como mostrado na Listagem 4-3.

Listagem 4-3. Inverter para loop

MOV R2, #10 @R2 detém I loop: @

corpo do loop vai aqui.

@ O CMP é redundante, já que @ estamos
fazendo SUBS.

SUBS R2, #1 @ I = I - 1 BNE loop @ ramo

até I = 0

Aqui salvamos uma instrução, pois com a instrução SUBS não precisa da instrução CMP.

Enquanto Loops

Vamos codificar:

```
ENQUANTO X < 5
... outras declarações ....
FINALIZAR ENQUANTO
```

Observação Inicializar e alterar as variáveis não faz parte da instrução **while**. Essas são instruções separadas que aparecem antes e no corpo do loop. Em Assembly, podemos codificar conforme mostrado na Listagem 4-4.

Listagem 4-4. loop while

```
@ R4 é X e foi inicializado loop: CMP R4, #5 BGE
loopdone ... outras instruções no corpo do loop ...
loop loopdone: @program continue
```

B

Observação Um loop while só é executado se a instrução for inicialmente verdadeira, portanto, não há garantia de que o corpo do loop será executado.

Capítulo 4 Controlando o Fluxo do Programa

Se/Então/Senão

Nesta seção, veremos a codificação

SE <expressão> ENTÃO ...

declarações ...

OUTRO

... declarações ...

FIM SE

Em Assembly, precisamos avaliar <expressão> e obter o resultado terminam em um registrador que podemos comparar. Por enquanto, assumiremos que <expressão> é simplesmente da forma

registrar comparação imediata-constante

Desta forma, podemos avaliá-lo com uma única instrução CMP. Para exemplo, suponha que queremos codificar

SE R5 < 10 ENTÃO

.... se declarações...

OUTRO

... outras declarações ...

FIM SE

Podemos codificar isso como Listagem 4-5.

Listagem 4-5. Declaração If/Then/Else

CMP R5, nº 10

BGE outra cláusula

... se declarações...

B endif
outra cláusula:
... outras declarações ...

endif: @ continua após o /then/else ...

Isso é bastante simples, mas ainda vale a pena colocar comentários para ficar claro quais instruções fazem parte do if/then/else e quais instruções estão no corpo dos blocos if ou else.

Dica Adicionar uma linha em branco pode tornar o código muito mais legível.

Operadores lógicos

Para nosso próximo programa de amostra, precisamos começar a manipular os bits nos registradores. Os operadores lógicos do ARM fornecem várias ferramentas para fazermos isso, como segue:

E{S}	Rd, Rs, Operando2
EOR{S}	Rd, Rs, Operando2
OR{S}	Rd, Rs, Operando2
BIC{S}	Rd, Rs, Operando2

Eles operam em cada bit dos registradores separadamente.

E

AND realiza uma operação lógica bit a bit entre cada bit em **Rs** e **Operand2**, colocando o resultado em **Rd**. Lembre-se que o AND lógico é verdadeiro (1) se ambos os argumentos forem verdadeiros (1) e falsos (0) caso contrário, por exemplo.

Vamos usar AND para mascarar um byte de informação. Suponha que queremos apenas o byte de ordem superior de um registrador (Listagem 4-6).

Capítulo 4 Controlando o Fluxo do Programa

Listagem 4-6. Usando AND para mascarar um byte de informação

@ mascara o byte de ordem superior

E R6, #0xFF000000

@ deslocar o byte para baixo para a

@ posição de baixa ordem.

LSR R6, n° 24

EOR

EOR executa uma operação ou exclusiva bit a bit entre cada bit em **Rs** e **Operando2**, colocando o resultado em **Rd**. Lembre-se de que OR exclusivo é verdadeiro (1) se exatamente um argumento for verdadeiro (1) e falso (0) caso contrário.

ORR

ORR executa uma lógica ou operação bit a bit entre cada bit em **Rs** e **Operando2**, colocando o resultado em **Rd**. Lembre-se que o OU lógico é verdadeiro (1) se um ou ambos os argumentos forem verdadeiros (1) e falso (0) se ambos os argumentos forem falsos (0), por exemplo:

ORR R6, #0xFF

Isso define o byte de ordem inferior de **R6** para todos os bits 1 (0xFF), deixando os outros três bytes inalterados.

BIC

BIC (Bit Clear) executa **Rs** AND NOT **Operando2**. O motivo é que, se o bit no **Operando2** for 1, o bit resultante será 0. Se o bit no **Operando2** for 0, o bit correspondente em **Rs** será colocado no resultado **Rd**.

Algumas vezes o Assembler substitui esta instrução para codificar um Operand2 que não funciona com AND, semelhante a MOV e MVN, por exemplo:

BIC R6, #0xFF

Isso limpa o byte de ordem inferior de R6, deixando os outros 3 bytes não afetado (Figura 4-2).

X	Y	X AND Y	X EOR Y	X ORR Y	X BIC Y
0	0	0	0	0	0
0	1	0	1	1	0
1	0	0	1	1	1
1	1	1	0	1	0

Figura 4-2. O que cada operador lógico faz com cada par de bits

Padrões de design

Ao escrever o código da linguagem Assembly, há uma grande tentação de ser criativo. Por exemplo, poderíamos fazer um loop dez vezes definindo o décimo bit em um registrador e, em seguida, deslocando-o para a direita até que o registrador seja zero. Isso funciona, mas dificulta a leitura do seu programa. Se você deixar seu programa e vier para ele no próximo mês, estará coçando a cabeça para saber o que o programa faz.

Padrões de projeto são soluções típicas para padrões de programação comuns. Se você adotar alguns padrões de projeto padrão sobre como executar loops e outras construções de programação, isso tornará a leitura de seus programas muito mais fácil.

Padrões de projeto tornam sua programação mais produtiva, já que você pode apenas usar um exemplo de uma coleção de padrões testados e comprovados para a maioria das situações.

Dica Na montagem, certifique-se de documentar qual padrão de projeto você está usando, juntamente com a documentação dos registros usados.

Capítulo 4 Controlando o Fluxo do Programa

Portanto, implementamos loops e if/then/else no padrão de uma linguagem de alto nível. Se fizermos isso, nossos programas serão mais confiáveis e mais rápidos de escrever. Posteriormente, veremos como usar o recurso de macro no Assembler para ajudar nisso.

Convertendo Inteiros para ASCII

Como primeiro exemplo de loop, vamos converter um registrador de 32 bits para ASCII, para que possamos exibir o conteúdo no console. Em nosso programa HelloWorld no Capítulo 1, “Iniciando”, usamos a chamada do sistema Linux número 4 para gerar nossa mensagem “Hello World!” corda. Neste programa, converteremos os dígitos hexadecimais do registrador em caracteres ASCII dígito por dígito. ASCII é uma maneira que os computadores representam todas as letras, números e símbolos que lemos, como números que um computador pode processar. Por exemplo:

- A é representado por 65.
- B é representado por 66.
- 0 é representado por 48.
- 1 é representado por 49 e assim por diante.

O ponto chave é que as letras de A a Z são contíguas, assim como os números de 0 a 9. Consulte o Apêndice E para todos os 255 caracteres.

Observação Para um único caractere ASCII que caiba em 1 byte, coloque-o entre aspas simples, por exemplo, 'A'. Se os caracteres ASCII forem formar uma string, use aspas duplas, por exemplo, “Hello World!”.

Aqui está um pseudocódigo de linguagem de alto nível para o que implementaremos na linguagem Assembly (Listagem 4-7).

Listagem 4-7. Pseudo-código para imprimir um registro

outstr = memória onde queremos a string + 9 @ (string é a forma 0x12345678 e queremos @ o último caractere)

PARA R5 = 8 PARA 1 PASSO -1

 dígito = R4 E 0xf

 SE dígito < 10 ENTÃO

 asciichar = dígito + '0'

 OUTRO

 asciichar = dígito + 'A' - 10

 FIM SE

 *outstr = asciichar

 outstr = outstr - 1

PRÓXIMA R5

A Listagem 4-8 é o programa em linguagem Assembly para implementar isso. ele usa o que aprendemos sobre loops, if/else e instruções lógicas.

Listagem 4-8. Imprimindo um registrador em ASCII

@ @ Programa Assembler para imprimir um registrador em hex
@ para stdout. @ @ R0-R2 - parâmetros para serviços de função
linux @ R1 - também é o endereço do byte que estamos
escrevendo @ R4 - registrar para imprimir @ R5 - índice de loop
@ R6 - caractere atual @ R7 - número da função linux @

Capítulo 4 Controlando o Fluxo do Programa

```

.global _start @ Forneça o endereço inicial do programa ao vinculador

_start: MOV R4, #0x12AB @ número para imprimir
        MOVT R4, #0xDE65 @ bits altos do número a imprimir
        LDR R1, =hexstr @ início da string
        ADD R1, #9 @ iniciar pelo menos o dígito sig

@ O loop é FOR r5 = 8 TO 1 STEP -1
        MOV R5, loop          @ 8 dígitos para imprimir

#8: AND R6, r4, #0xf @ máscara do menor dígito sig
@ Se R6 >= 10 então vá para a letra
        CMP R6, #10 @ é 0-9 ou AF
        carta BGE

@ Caso contrário, é um número, então converta para um dígito ASCII
        ADICIONE R6, #'0'
        B continua          @ ir para terminar se

letra: @ lida com os dígitos de A a F
        ADD R6, #('A'-10) cont: @
end if

        STRB R6, [R1]          @ armazena o dígito ascii
        SUB R1, nº 1            @ diminui o endereço para o próximo dígito @
        LSR R4, nº 4            desativa o dígito que acabamos de processar

        @ próximo R5
        SUBS R5, nº 1           @ passo R5 por -2 @
        Circuito BNE           outro loop for se não for feito

@ Configure os parâmetros para imprimir nosso número hexadecimal
@ e depois chame o Linux para fazer isso. mov R0, #1 @ 1 = StdOut
ldr R1, =hexstr @ string para imprimir @ R0, #1 @ R1, =printox
        write system call @ Chamar linux para enviar a string

svc 0

```

Capítulo 4 Controlando o Fluxo do Programa

@ Defina os parâmetros para sair do programa @ e depois chame o Linux para fazer isso.

```

movimento      R0, #0 @ Use 0 código de retorno
movimento      R7, #1 @ Código de comando de serviço 1 encerra este
programa
svc      0          @ Chame o linux para encerrar o programa

.dados
hexstr:        .ascii "0x12345678\n"

```

Se compilarmos e executarmos o programa, veremos

```

pi@stevepi:~/asm/Chapter 4 $ make as
printword.s -o printword.o ld -o printword
printword.o pi@stevepi:~/asm/Chapter 4 $ ../
printword 0xDE6512AB pi@stevepi:~/asm /Capítulo 4 $

```

como seria de esperar. A melhor maneira de entender este programa é passar por ele no **gdb** e observar como ele está usando os registros e atualizando memória.

Certifique-se de entender por que

```
E      R6, r4, #0xf
```

mascara o dígito de ordem inferior; caso contrário, revise a seção “E” sobre operadores lógicos.

Como AND exige que ambos os operandos sejam 1 para resultar em 1, and'ing algo com 1s (como 0xf) mantém o outro operador como está, enquanto and'ing algo com 0s sempre torna o resultado 0.

Em nosso loop, deslocamos R4, 4 bits para a direita com

```
LSR      R4, no 4
```

Capítulo 4 Controlando o Fluxo do Programa

Isso muda o próximo dígito para a posição de processamento no próximo iteração.

Observação Isso é destrutivo para R4 e você perderá seu número original durante esse algoritmo.

Já discutimos a maioria dos elementos presentes neste programa, mas há alguns novos elementos; eles são demonstrados a seguir.

Usando Expressões em Constantes Imediatas

ADICIONAR R6, #('A'-10)

Isso demonstra alguns novos truques do GNU Assembler:

1. Podemos incluir caracteres ASCII em imediato
operandos colocando-os entre aspas simples.
2. Podemos colocar expressões simples nos operandos
immediatos. O GNU Assembler traduz 'A' para 65, subtrai
10 para obter 55 e usa isso como Operand2.

Isso torna o programa mais legível, pois podemos ver nossa intenção, em vez de apenas codificarmos 55 aqui. Não há penalidade para o programa ao fazer isso, pois o trabalho é feito quando montamos o programa, não quando o executamos.

Armazenando um registrador na memória

STRB R6, [R1]

A instrução **Store Byte (STRB)** salva o byte de ordem inferior do primeiro registrar no local de memória contido em R1. A sintaxe [R1] é para deixar claro que estamos usando indireção de memória e não apenas colocando o

byte no registrador R1. Isso é para tornar o programa mais legível, para não confundirmos esta operação com uma instrução MOV correspondente.

Acessar dados na memória é o tópico do Capítulo 5, “Obrigado pelas memórias”, onde entraremos em mais detalhes. A maneira como estamos armazenando o byte pode ser mais eficiente, e veremos isso então.

Por que não imprimir em decimal?

Neste programa de exemplo, convertemos facilmente em uma string hexadecimal porque usar AND 0xf é equivalente a obter o resto ao dividir por 16.

Da mesma forma, deslocar o registro para a direita 4 bits é equivalente a dividir por 16. Se quiséssemos converter para uma string decimal, base 10, precisaríamos ser capazes de obter o resto da divisão por 10 e depois dividir por 10.

Até agora, não vimos uma instrução de divisão. Isso coloca a conversão para decimal além do escopo deste capítulo. Poderíamos escrever um loop para implementar o algoritmo de divisão longa que aprendemos no ensino fundamental, mas, em vez disso, vamos adiar a divisão até o [Capítulo 10](#), “Multiplicar, Dividir e Acumular”.

Desempenho das Instruções da Filial

No Capítulo 1, “Iniciando”, mencionamos que o conjunto de instruções ARM32 é executado em um pipeline de instruções. Individualmente, uma instrução requer três ciclos de clock para ser executada, uma para cada uma das

1. Carregue a instrução da memória para a CPU.
2. Decodifique a instrução.
3. Execute a instrução.

No entanto, a CPU trabalha com três instruções ao mesmo tempo, cada uma em um passo diferente, portanto, em média, executamos uma instrução a cada ciclo de clock. Mas o que acontece quando ramificamos?

Capítulo 4 Controlando o Fluxo do Programa

Quando executamos o desvio, já decodificamos a próxima instrução e carregamos a instrução 2 à frente. Quando ramificamos, jogamos esse trabalho fora e começamos de novo. Isso significa que a instrução após a ramificação levará três ciclos de clock para ser executada.

Se você colocar muitas ramificações em seu código, sofrerá uma penalidade de desempenho, talvez reduzindo a velocidade do seu programa em um fator de 3. Outro problema é que, se você programar com muitas ramificações, isso levará a um código espaguete - ou seja, todas **as** linhas de código estão emaranhados como um pote de espaguete, comprehensivelmente muito difícil de manter.

Quando aprendi a programar no ensino médio e em meus anos de graduação, antes que a programação estruturada estivesse disponível, usei as linguagens de programação BASIC e Fortran para escrever códigos complexos. Sei de primeira mão que decifrar programas cheios de ramificações é um desafio.

As primeiras linguagens de programação de alto nível dependiam da instrução **goto** que levava a um código difícil de entender; isso levou à programação estruturada que vemos em linguagens modernas de alto nível que não precisam de uma instrução goto. Não podemos eliminar completamente as ramificações, já que o ARM32 não possui construções de programação estruturadas, mas precisamos estruturar nosso código de acordo com essas linhas para torná-lo mais eficiente e fácil de ler - outro ótimo uso para alguns bons padrões de design .

O conjunto de instruções ARM32 possui um mecanismo para lidar com isso, utilizando o código de condição em cada instrução. Veremos isso no Capítulo 13, “Instruções condicionais e otimização de código”.

Mais instruções de comparação

Vimos a instrução CMP, que é a principal instrução de comparação; no entanto, existem mais três:

- CMNRn, Operando2
- TEQ Rn, Operando2
- TST Rn, Operando2

Lembre-se que a instrução CMP subtraiu Operand2 de Rn e defina os sinalizadores de condição no CPSR de acordo. O resultado da subtração é descartado. Essas três instruções funcionam da mesma maneira, exceto que usam uma operação diferente da subtração.

O Assembler tem a capacidade de alternar entre as quatro comparações instruções para refinar alguns valores extras para Operand2, que de outra forma seriam impossíveis. Neste livro, usaremos apenas CMP, mas você pode usá-los se encontrar uma aplicação, e vale a pena ficar atento caso o Assembler faça uma substituição. Os outros três são

- **CMP:** usa adição em vez de subtração. Então indica que é o negativo (oposto) de CMP.
- **TEQ:** Executa um OU exclusivo bit a bit entre Rn e Operando2. Ele atualiza o CPSR com base no resultado.
- **TST:** executa uma operação AND bit a bit entre Rn e Operando2. Ele atualiza o CPSR com base no resultado.

Resumo

Neste capítulo, estudamos as principais instruções para executar a lógica do programa com loops e instruções if. Estes incluíram as instruções para comparações e ramificação condicional. Discutimos vários padrões de projeto para codificar as construções comuns de linguagens de programação de alto nível em Assembly. Examinamos as instruções para trabalhar logicamente com os bits em um registrador. Examinamos como poderíamos enviar o conteúdo de um registrador em formato hexadecimal.

CAPÍTULO 5

Obrigado pelo Recordações

Neste capítulo, discutimos a memória do Raspberry Pi. Até agora, usamos a memória para armazenar nossas instruções de montagem; agora veremos em detalhes como definir os dados na memória, como carregar a memória nos registradores para processamento e como gravar os resultados de volta na memória.

O ARM32 usa o que é chamado de **arquitetura load-store**. Isso significa que o conjunto de instruções é dividido em duas categorias: uma para carregar e armazenar valores de e para a memória e outra para realizar operações aritméticas e lógicas entre os registradores. Passamos a maior parte do tempo olhando para as operações aritméticas e lógicas. Agora vamos olhar para a outra categoria.

Endereços de memória são 32 bits e instruções são 32 bits, então temos os mesmos problemas que experimentamos no Capítulo 2, “Carregando e Adicionando”, onde usamos todos os tipos de truques para carregar 32 bits em um registrador. Neste capítulo, usaremos esses mesmos truques para carregar endereços, juntamente com alguns novos. O objetivo é carregar um endereço de 32 bits em uma instrução da forma tantos casos quanto pudermos.

O conjunto de instruções ARM32 possui algumas instruções poderosas para acessar a memória, incluindo várias técnicas para acessar matrizes de estruturas de dados e incrementar ponteiros em loops durante o carregamento ou armazenamento de dados.

Capítulo 5 Obrigado pelas memórias

Definindo o conteúdo da memória

Antes de carregar e armazenar a memória, primeiro precisamos definir alguma memória para operar. O GNU Assembler contém várias diretivas para ajudá-lo a definir a memória a ser usada em seu programa. Eles aparecem em uma seção .data do seu programa. Veremos alguns exemplos e resumiremos na Tabela 5-1. A Listagem 5-1 começa mostrando como definir bytes, palavras e strings ASCII.

Listagem 5-1. Alguns exemplos de diretivas de memória

```
label: .byte 74, 0112, 0b00101010, 0x4A, 0X4a, 'J', 'H' + 2 .word 0x1234ABCD,  
-1434 .ascii "Hello World\n"
```

A primeira linha define 7 bytes, todos com o mesmo valor. Podemos definir nossos bytes em decimal, octal (base 8), binário, hexadecimal ou ASCII. Em qualquer lugar onde definimos números, podemos usar expressões que o Assembler avaliará quando compilar nosso programa.

Iniciamos a maioria das diretivas de memória com um rótulo, para que possamos acessá-lo a partir do código. A única exceção é se estivermos definindo uma matriz maior de números que se estende por várias linhas.

A instrução .byte define 1 ou mais bytes de memória. A Listagem 5-1 mostra os vários formatos que podemos usar para o conteúdo de cada byte, como segue:

- Um inteiro decimal começa com um dígito diferente de zero e contém dígitos decimais de 0 a 9.
- Um inteiro octal começa com zero e contém dígitos octais de 0 a 7.

Capítulo 5 Obrigado pelas memórias

- Um inteiro binário começa com 0b ou 0B e contém dígitos binários 0–1.
 - Um inteiro hexadecimal começa com 0x ou 0X e contém o dígito hexadecimal 0–F.
 - Um número de ponto flutuante começa com 0f ou 0e, seguido por um número de ponto flutuante.
-

Observação Tenha cuidado para não iniciar números decimais com zero (0), pois isso indica que a constante é um número octal (base 8).

O exemplo então mostra como definir uma palavra e uma string ASCII, como vimos em nosso programa HelloWorld no Capítulo 1, “Primeiros passos”. Existem dois operadores de prefixo que podemos colocar na frente de um número inteiro:

- Negativo (-) levará o complemento de dois do inteiro.
- Complemento (~) tomará o complemento de um do inteiro.

Por exemplo:

```
.byte -0x45, -33, ~0b00111001
```

A Tabela 5-1 lista os vários tipos de dados que podemos definir dessa maneira.

Capítulo 5 Obrigado pelas memórias

Tabela 5-1. A lista de diretivas Assembler de definição de memória

Diretiva	Descrição
.ascii	Uma string contida entre aspas duplas
.asciz	Uma string ascii terminada em zero byte
.byte	inteiros de 1 byte
.dobro	Valores de ponto flutuante de precisão dupla
.flutuador	Valores de ponto flutuante
.octa	inteiros de 16 bytes
.quad	inteiros de 8 bytes
.curto	inteiros de 2 bytes
.palavra	inteiros de 4 bytes

Se quisermos definir um conjunto maior de memória, existem alguns mecanismos para fazer isso sem precisar listar e contar todos, como:

.fill repeat, tamanho, valor

Isso repete um valor de um determinado tamanho, repita vezes, por exemplo:

zeros: .preencher 10, 4, 0

cria um bloco de memória com dez palavras de 4 bytes, todas com valor zero.

O seguinte código

contagem de .rept

...

.endr

repete as declarações entre .rept e .endr, conta vezes. Isso pode cercar qualquer código em seu Assembly, por exemplo, você pode fazer um loop repetindo sua contagem de códigos vezes, por exemplo:

Capítulo 5 Obrigado pelas memórias

```
rpn: .rept 3 .byte 0,
      1, 2 .endr
```

é traduzido para

```
.byte 0, 1, 2 .byte
0, 1, 2 .byte 0, 1, 2
```

Em strings ASCII, vimos o caractere especial “\n” para nova linha.

Existem mais alguns para caracteres não imprimíveis comuns, bem como para nos dar a capacidade de colocar aspas duplas em nossas strings. O “\” é chamado de caractere de escape, que é um metacaractere para definir casos especiais. A Tabela 5-2 lista as sequências de caracteres de escape suportadas pelo GNU Assembler.

Tabela 5-2. Códigos de sequência de caracteres de escape ASCII

Sequência de caracteres de escape Descrição

\b	Backspace (código ASCII 8)
\f	Formfeed (código ASCII 12)
\n	Nova linha (código ASCII 10)
\r	Retorno (código ASCII 13)
\t	Guia (código ASCII 9)
\ddd	Um código ASCII octal (ex \123)
\xdd	Um código ASCII hexadecimal (ex \x4F)
\\"	O personagem
\"	O caractere de aspas duplas
\algo mais	algo mais

Capítulo 5 Obrigado pelas memórias

Carregando um Registro

Nesta seção, veremos a instrução **LDR** e suas variações.

Usamos o LDR para carregar um endereço em um registrador e para carregar os dados apontados por esse endereço. Existem métodos para indexar através da memória, bem como suporte para todos os truques para obter o máximo possível de nossas instruções de 32 bits. Analisaremos os casos um por um, incluindo

- Endereçamento relativo do PC
- Carregando da memória
- Indexação através da memória

Endereçamento relativo do PC

No Capítulo 1, “Iniciando”, apresentamos a instrução **LDR** para carregar o endereço do nosso “Hello World!” corda. Precisávamos fazer isso para passar o endereço do que imprimir para o comando **de gravação** do Linux . Este é um exemplo simples de endereçamento relativo de PC. É conveniente, pois não envolve nenhum outro registrador. Contanto que você mantenha seus dados próximos ao seu código, é indolor. Lembre-se que quando olhamos para a desmontagem da instrução LDR

LDR R1, =alômundo

era

LDR r1, [peça, nº 20]

Aqui estamos escrevendo uma instrução para carregar o endereço de nossa string helloworld em **R1**. O Assembler conhece o valor do contador do programa neste ponto, então ele pode fornecer um deslocamento para o endereço de memória correto. Portanto, é chamado de endereçamento relativo do PC. Há um pouco mais de complexidade nisso; que vamos chegar em um minuto.

O deslocamento acima leva 12 bits na instrução, o que dá um intervalo de 0–4095. Há outro bit na instrução para dizer qual direção compensar, então obtemos um intervalo de ± 4095 . Neste caso, estamos carregando uma palavra, então o intervalo de endereços é de ± 4095 palavras.

A forma geral desta instrução é

LDR{tipo} R_t, = etiqueta

onde type é um dos tipos listados na Tabela 5-3.

Tabela 5-3. Os tipos de dados para as instruções load/store

Tipo	Significado
B	Byte não assinado
SB	byte assinado
H	Meia palavra sem sinal (16 bits)
SH	Meia palavra assinada (16 bits)
—	palavra omitida

Neste caso simples, onde estamos apenas carregando o endereço, a única coisa aproveitada do tipo é o tamanho dos dados. Se carregarmos um byte, o deslocamento será em bytes. Esta parte assinada é importante quando carregamos e salvamos dados, como veremos em breve.

Observação O deslocamento é de ± 4095 nas unidades dos dados que estamos carregando.

O endereçamento relativo do PC tem mais um truque na manga; nos dá uma maneira de carregar qualquer quantidade de 32 bits em um registrador em apenas uma instrução, por exemplo, considere

LDR R1, =0x1234ABCDF

Capítulo 5 Obrigado pelas memórias

Isso se reúne em

```
ldr      r1, [pc, #8] .word
0x1234abcd
```

O GNU Assembler está nos ajudando colocando a constante que queremos na memória e, em seguida, criando uma instrução relativa ao PC para carregá-lo.

No Capítulo 2, “Carregar e adicionar”, realizamos isso com um par MOV/MOVT. Aqui estamos fazendo a mesma coisa em uma instrução. Ambos usam a mesma memória, duas instruções de 32 bits ou uma instrução de 32 bits e um local de memória de 32 bits.

Na verdade, é assim que o Assembler lida com todos os rótulos de dados. Quando nós Especificadas

```
LDR      R1, =alômundo
```

o Montador fez a mesma coisa; ele criou o endereço da hellostring na memória e, em seguida, carregou o conteúdo desse local de memória, não a string helloworld. Examinaremos cuidadosamente esse processo quando discutirmos nosso programa para converter strings em letras maiúsculas mais adiante neste capítulo.

Essas constantes que o Assembler cria são colocadas no final da seção .text , que é onde vão as instruções do Assembly. Não na seção .data . Isso os torna somente leitura em circunstâncias normais, portanto, não podem ser modificados. Todos os dados que você deseja modificar devem ir para uma seção .data .

Por que o Assembler faria isso? Por que não apenas apontar o índice relativo do PC diretamente para os dados? Existem várias razões para isso, nem todas específicas para o conjunto de instruções ARM32:

1. Um deslocamento de 4096 não é muito grande, especialmente se você tiver várias strings grandes. Dessa forma, podemos acessar 4.096 objetos em vez de 4.096 palavras. Isso ajuda a manter nosso programa igualmente eficiente à medida que cresce.

Capítulo 5 Obrigado pelas memórias

2. Todos os rótulos que definimos vão para a tabela de símbolos do arquivo objeto, tornando esse array de endereços essencialmente nossa tabela de símbolos. Dessa forma, é fácil para o vinculador/carregador e o sistema operacional alterar os endereços de memória sem a necessidade de recompilar o programa.
3. Se você precisar que qualquer uma dessas variáveis seja global, basta torná-las globais (acessíveis a outros arquivos) sem alterar seu programa. Se não tivéssemos esse nível de indireção, tornar uma variável global exigiria ajustes nas instruções que a carregam e salvam.

Este é outro exemplo das ferramentas que nos ajudam, embora a princípio possa não parecer. Em nossos exemplos simples de uma linha, parece adicionar uma camada de complexidade, mas em um programa real, esse é o padrão de projeto que funciona.

Carregando da memória

Em nosso programa HelloWorld, precisávamos apenas que o endereço passasse para o Linux, que então o usaria para imprimir nossa string. Geralmente, gostamos de usar esses endereços para carregar dados em um registrador, conforme demonstrado na Listagem 5-2.

Listagem 5-2. Carregando um endereço e depois o valor

```
@ carregar o endereço de mynumber em R1
LDR      R1, =meunúmero

@ carrega a palavra armazenada em mynumber em R2
LDR      R2, [R1]           .dados

meu numero: .WORD 0x1234ABCD
```

Se você passar por isso no depurador, poderá vê-lo carregar 0x1234ABCD em **R2**.

Capítulo 5 Obrigado pelas memórias

Observação A sintaxe do colchete representa o acesso indireto à memória. Isso significa carregar os dados armazenados no endereço apontado por **R1**, não mover o conteúdo de **R1** para **R2**.

Quando encontramos “LDR r1, [pc, #20]”, parecia que estávamos apenas carregando o endereço de pc+20, mas agora sabemos que na verdade estamos carregando o endereço armazenado em pc+20, e é por isso que os colchetes são usados.

Nota Se você deseja carregar um byte deste local de memória, você precisa adicionar o tipo a ambas as instruções, ou haverá uma incompatibilidade de comprimento e não carregará o byte em que você está pensando.

Isso funciona, mas você pode estar insatisfeito com o fato de termos levado duas instruções para carregar R2 com nosso valor da memória: uma para carregar o endereço e outra para carregar os dados. Esta é a vida programando um processador RISC; cada instrução é executada muito rapidamente, mas realiza um pequeno pedaço de trabalho.

À medida que desenvolvemos algoritmos, veremos que normalmente carregamos um endereço uma vez e depois o usamos um pouco, então a maioria dos acessos leva uma instrução assim que vamos.

Indexação através da memória

Todas as linguagens de programação de alto nível possuem uma construção de matriz. Eles podem definir uma matriz de objetos e acessar os elementos individuais por índice. A linguagem de alto nível definirá o array com algo como

DIM A[10] COMO PALAVRA

em seguida, accesse os elementos individuais com instruções como as da Listagem 5-3.

Listagem 5-3. Pseudocódigo para percorrer um array

```

// Define o 5º elemento do array com o valor 6
A[5] = 6

// Define a variável X igual ao 3º elemento do array
X = A[3]

// Percorre todos os 10 elementos
PARA I = 1 A 10
    // Define o elemento I como I ao cubo
    A[I] = I ** 3
    PROXIMO EU

```

O conjunto de instruções ARM32 nos dá suporte para fazer esses tipos de operações.

Suponha que temos um array de dez palavras (4 bytes cada) definido por

```
arr1: .FILL 10, 4, 0
```

Vamos carregar o endereço do array em **R1**:

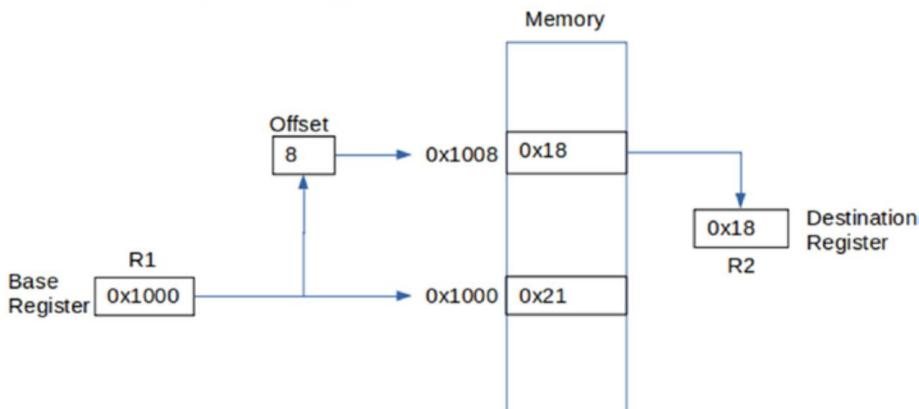
```
LDR      R1, =arr1
```

Agora podemos acessar os elementos usando **LDR** conforme demonstrado na Listagem 5-4 e representado graficamente na Figura 5-1.

Listagem 5-4. Indexação em um array

- @ Carregar o primeiro elemento
- LDR R2, [R1]
- @ Carregar elemento
- 3 @ Os elementos contam a partir de 0, então 2 é
@ o terceiro. Cada palavra tem 4 bytes, @ então
precisamos multiplicar por 4 LDR R2, [R1, #(2 * 4)]

Capítulo 5 Obrigado pelas memórias

LDR R2, [R1 + #(2 * 4)]**Figura 5-1.** Visualização gráfica do uso de R1 e um índice para carregar R2

Isso é bom para acessar elementos codificados, mas e por meio de um variável? Podemos usar um registrador conforme demonstrado na Listagem 5-5.

Listagem 5-5. Usando um registrador como offset

@ O terceiro elemento ainda é o número 2 MOV R3,

#(2 * 4)

@ Adicione o deslocamento em R3 a R1 para obter nosso elemento.

LDR R2, [R1, R3]

Podemos fazer essas mudanças ao contrário. Se **R2** aponta para o final da matriz, nós podemos fazer

LDR R2, [R1, #-(2 * 4)]

LDR R2, [R1, -R3]

Com o registrador como offset, é o mesmo que um registrador e tipo shift Operand2 que estudamos no Capítulo 2, “Carregando e Adicionando”. Para as constantes anteriores, poderíamos fazer um * 4 na instrução imediata, mas se estiver em um registrador, precisaríamos fazer uma operação de deslocamento adicional e colocar o resultado em outro registrador. Com o formato register/shift, podemos lidar com alguns casos facilmente. O cálculo do endereço de uma matriz de palavras é demonstrado na Listagem 5-6.

Listagem 5-6. Multiplicando um deslocamento por 4 usando uma operação de deslocamento

@ Suponha que nosso array seja de WORDs, mas @ queremos apenas o byte de ordem inferior.
MOV R3, #2 @
Shift R3 à esquerda por 2 posições para multiplicar @ por
4 para obter o endereço correto.
LDR R2, [R1, R3, LSL #2]

Escreva de volta

Quando o endereço é calculado pelas adições e deslocamentos, o resultado é descartado após carregarmos o registrador. Ao executar um loop, é útil manter o endereço calculado. Isso nos poupa de fazer um ADD separado em nosso registrador de índice.

A sintaxe para isso é colocar um ponto de exclamação (!) após a instrução, então o Assembler irá setar o bit na instrução gerada pedindo para a CPU salvar o endereço calculado, assim

LDR R2, [R1, R3, LSL #2]!

atualiza **R1** com o valor calculado. Nos exemplos que estudamos, isso não é tão útil, mas se tornará muito mais útil na próxima seção.

Capítulo 5 Obrigado pelas memórias

Endereçamento pós-indexado

A seção anterior cobre o que é chamado de **endereçamento pré-indexado**. Isso ocorre porque o endereço é calculado e, em seguida, os dados são recuperados usando o endereço calculado. No **endereçamento pós-indexado**, os dados são recuperados primeiro usando o registrador base, então qualquer deslocamento e adição de deslocamento é feito. No contexto de uma instrução, isso parece estranho, mas quando escrevemos loops, veremos que é isso que queremos. O endereço calculado é escrito de volta para o registrador de endereço base, caso contrário não há sentido em usar este recurso, então não precisamos do !.

Indicamos que queremos endereçamento pós-indexado colocando os itens a serem adicionados fora dos colchetes. Nos exemplos a seguir, o LDR carregará **R1** com o conteúdo da memória apontada por **R2** e atualizará **R2** usando o método indicado em cada instrução. A Listagem 5-7 fornece alguns exemplos de endereçamento pós-indexado.

Listagem 5-7. Exemplos de endereçamento pós-indexado

```
@ Carregar R1 com a memória apontada por R2  
@ Então faça R2 = R2 + R3  
LDR R1, [R2], R3  
  
@ Carregar R1 com a memória apontada por R2  
@ Então faça R2 = R2 + 2  
LDR R1, [R2], nº 2  
  
@ Carregar R1 com a memória apontada por R2  
@ Então faça R2 = R2 + (R3 deslocado 2 à esquerda)  
LDR R1, [R2], R3, LSL #2
```

Convertendo para Maiúsculas

Como um exemplo de como o endereçamento pós-indexado nos ajuda a escrever loops, vamos considerar o loop através de uma string de bytes ASCII. Suponha que queremos converter qualquer caractere minúsculo em maiúsculo. A Listagem 5-8 fornece um pseudocódigo para fazer isso.

Listagem 5-8. Pseudo-código para converter uma string em letras maiúsculas

```

eu = 0

FAZER
    char = instr[i]
    SE char >= 'a' AND char <= 'z' ENTÃO
        char = char - ('a' - 'A')
    FIM SE
    outstr[i] = char i = i + 1

```

ATÉ char == 0

IMPRIMIR outstr

Neste exemplo, vamos usar strings terminadas em NULL. Estes são muito comuns na programação C. Aqui, em vez de uma string ser um comprimento e uma sequência de caracteres, a string é a sequência de caracteres, seguida por um caractere NULL (código ASCII 0 ou \0). Para processar a string, simplesmente fazemos um loop até atingirmos o caractere NULL. Isso é bem diferente da string de comprimento fixo com a qual lidamos ao imprimir dígitos hexadecimais no Capítulo 4, “Controlando o fluxo do programa”.

Já cobrimos os loops for e while. A terceira estrutura comum loop de programação é o loop **DO/UNTIL** que coloca a condição no final do loop. Nesta construção, o loop é sempre executado uma vez. No nosso caso, queremos isso, pois se a string estiver vazia, ainda queremos copiar o caractere NULL, então a string de saída também estará vazia.

Outra diferença é que não estamos alterando a string de entrada. Em vez disso, deixamos a string de entrada sozinha e produzimos uma nova string de saída com a versão maiúscula da string de entrada.

Como é comum no processamento da linguagem Assembly, invertemos a lógica para pular o código no bloco IF. A Listagem 5-9 mostra o pseudocódigo atualizado.

Capítulo 5 Obrigado pelas memórias

Listagem 5-9. Pseudo-código sobre como implementaremos a instrução IF

```
SE char < 'a' GOTO continuar
SE char > 'z' GOTO continuar
char = char - ('a' - 'A') continue: //  
o restante do programa
```

Não temos as construções de programação estruturada de um programador de alto nível linguagem para nos ajudar, e isso acaba sendo bastante eficiente em linguagem Assembly.

A Listagem 5-10 é o código Assembly para converter uma string em letras maiúsculas.

Listagem 5-10. Programa para converter uma string para letras maiúsculas

```
@ @ Programa Assembler para converter uma string
em @ todas as letras maiúsculas. @ @ R0-R2 -
parâmetros para serviços de função Linux @ R3 -
endereço da string de saída @ R4 - endereço da string de
entrada @ R5 - caractere atual sendo processado @ R7 -
número da função Linux @
```

```
.global _start @ Forneça o endereço inicial do programa
_start: LDR R4, =instr @ início da string de entrada
        LDR R3, =outstr @ endereço da string de saída
@ O loop é até que o byte apontado por R1 seja diferente de zero
@ Carregue o caractere e incremente o loop do
ponteiro: LDRB R5, [R4], #1
@ Se R5 > 'z' então vá para cont
        CMP      R5, #'z'           @ é a letra > 'z'?
        BGT      cont.
```

@ Else if R5 < 'a' then goto end if R5, #'a'

B1T cont @ goto to end if @ se chegamos
aqui então a letra é minúscula, @ então converta.

SUB R5, #('a'-'A') cont: @
end if STRB R5, [R3], #1 @ armazenar
caractere para superar CMP R5, #0 BNE loop
@ pare ao atingir um caractere nulo @
loop se o caractere não for nulo

@ Configure os parâmetros para imprimir nosso número hexadecimal
@ e depois chame o Linux para fazer isso.

MOV R0, #1 @ 1 = StdOut
LDR R1, =outstr @ string para imprimir @ obtenha
o comprimento subtraindo os ponteiros
SUB R2, R3, R1
MOV R7, nº 4 @ chamada de sistema de gravação linux
SVC 0 @ Chame o linux para enviar a string

@ Defina os parâmetros para sair do programa @ e depois
chame o Linux para fazer isso.

MOV	R0, #0	@ Use 0 código de retorno
MOV	R7, #1	@ Código de comando de serviço 1
SVC	0	@ Chame o linux para encerrar

.dados

instr: .asciz "Esta é nossa string de teste que iremos converter.\n"

outstr: .preencher 255, 1, 0

Capítulo 5 Obrigado pelas memórias

Se compilarmos e executarmos o programa, obtemos a saída desejada:

```
pi@raspberrypi:~/asm/Capítulo 5 $ ./upper
ESTA É A NOSSA CORDA DE TESTE QUE CONVERTEREMOS.
pi@raspberrypi:~/asm/Capítulo 5 $
```

Este programa é bastante curto. Além de todos os comentários e o código para imprimir a string e sair, existem apenas 11 instruções Assembly para inicializar e executar o loop:

- **Duas instruções:** inicializar nossos ponteiros para instr e ultrapassar
- **Cinco instruções:** crie a instrução if
- **Quatro instruções:** Para o loop, incluindo carregar um caractere, salvar um caractere, atualizar ambos os ponteiros, verificar se há um caractere nulo e ramificar se não for nulo

Seria bom se o STRB também configurasse os sinalizadores de condição, mas não há uma versão do STRBS. LDR e STR apenas carregam e salvam; eles não têm funcionalidade para examinar o que estão carregando e salvando, portanto, não podem definir o CPSR. Daí a necessidade da instrução CMP na parte UNTIL do loop para testar NULL.

Neste exemplo, utilizamos as instruções LDRB e STRB, pois estamos processamento byte a byte. A instrução STRB é o inverso da instrução LDRB. Ele salva seu primeiro argumento no endereço criado a partir de todos os outros parâmetros. Ao cobrir o LDR com tantos detalhes, também cobrimos o STR, que é a imagem espelhada.

Para converter a letra para maiúscula, usamos

```
SUB R5, #('a'-'A')
```

Os caracteres minúsculos têm valores mais altos que os maiúsculos caracteres, então usamos apenas uma expressão que o Assembler irá avaliar para obter o número correto para subtrair.

Quando vamos imprimir a string, não sabemos seu comprimento e o Linux requer o comprimento. Usamos a instrução

```
SUB R2, R3, R1
```

Aqui acabamos de carregar R1 com o endereço de outstr. R3 manteve o endereço de outstr em nosso loop, mas como usamos endereçamento pós-indexado, ele foi incrementado a cada iteração do loop. Como resultado, agora está apontando 1 após o final da string. Em seguida, calculamos o comprimento subtraindo o endereço do início da string do endereço do final da string. Poderíamos ter mantido um contador para isso em nosso loop, mas em Assembly estamos tentando ser eficientes, então queremos o mínimo possível de instruções em nossos loops.

Vejamos a Listagem 5-11, uma desmontagem do nosso programa.

Listagem 5-11. Desmontagem do programa em maiúsculas

Conteúdo da seção .text:

00010074 <_start>:

10074: e59f4044 ldr r4, [pc, #68]	10078: e59f3044	; 100c0 <cont+0x2c> ;
ldr r3, [pc, #68]		100c4 <cont+0x30>

0001007c <loop>:

1007c: e4d45001	ldrb r5, [r4], #1	r5, #122 ;
10080: e355007a	cmp	0x7a 10094 <cont> r5,
10084: ca000002	bgt	#97 ; 0x61 10094 <cont>
10088: e3550061	cmp	r5, r5, #32
1008c: ba000000	blt	
10090: e2455020	sub	

00010094 <cont>:

10094: e4c35001	strb r5, [r3], #1	r5, #0
10098: e3550000	cmp	1007c <loop>
1009c: 1affffff	bne	

Capítulo 5 Obrigado pelas memórias

100a0: e3a00001 100a4:	movimento	r0, #1
e59f1018 100a8:	ldr	r1, [pc, #24] ; 100c4 <cont+0x30> r2, r3, r1 r7,
e0432001	sub	#4
100ac: e3a07004 100b0:	movimento	
ef000000	svc	0x00000000
100b4: e3a00000 100b8:	movimento	r0, #0
e3a07001 100bc:	movimento	r7, #1
ef000000	svc	0x00000000
100c0: 000200c8	.word	0x000200c8
100c4: 000200f7	.word	0x000200f7

Conteúdo da seção .data:

200c8 54686973 20697320 6f757220 54657374 Este é o nosso teste 200d8 20537472
 696e6720 74686174 20776520 String que nós 200e8 77696c6c 20636f6e 766570270
 20e will convert.
 200f8 00000000 00000000 00000000 00000000

A instrução

LDR R4, =instr

foi convertido para

ldr r4, [peça, nº 68] ; 100c0

O comentário nos diz que pc+68 é o endereço 0x100c0. Podemos calcular isso nós mesmos se pegarmos o endereço da instrução 2 após esta (aquele que está sendo carregada enquanto esta é executada), que está em 0x1007c, e adicionando 68 na calculadora Gnome para obter o mesmo 0x100c0.

Isso mostra como o Assembler adicionou o literal para o endereço do string instr no final da seção de código. Quando fazemos o LDR, ele acessa esse literal e carrega na memória; isso nos dá o endereço que precisamos na memória. O outro literal adicionado à seção de código é o endereço de outstr.

Capítulo 5 Obrigado pelas memórias

Para ver este programa em ação, vale a pena dar uma olhada nele no **gdb**. Você pode monitorar os registradores com o comando “ir” (info registradores).

Para visualizar instr e oustr conforme o processamento ocorre, existem algumas maneiras de fazê-lo. Pela desmontagem sabemos que o endereço de instr é 0x200c8, para que possamos entrar

```
(gdb) x /2s 0x200c8
0x200c8:      "Esta é nossa string de teste que iremos converter.\n"
0x200f7:      "ISS"
(gdb)
```

Isso é conveniente, pois o comando x sabe como formatar strings, mas não sabe sobre rótulos. Também podemos entrar

```
(gdb) p (char[10]) outstr
$8 = "TH\000\000\000\000\000\000\000" (gdb)
```

O comando print (p) conhece nossos rótulos, mas não conhece nossos tipos de dados, e devemos converter o rótulo para informar como formatar a saída.

Gdb lida melhor com linguagens de alto nível porque conhece os tipos de dados das variáveis. Em Assembly, estamos mais próximos do metal.

Armazenando um Registro

A instrução Store Register **STR** é um espelho da instrução **LDR**. Todos os modos de endereçamento sobre os quais falamos para LDR funcionam para STR. Isso é necessário, pois em uma arquitetura de armazenamento de carga, precisamos armazenar tudo o que carregamos depois de processado na CPU. Já vimos a instrução STR algumas vezes em nossos exemplos.

Se estivermos usando os mesmos registradores para carregar e armazenar os dados em um loop, normalmente a primeira chamada LDR usará endereçamento pré-indexado sem write-back e, em seguida, a instrução STR usará endereçamento pós-indexado com write-back para avançar para o próximo item para a próxima iteração do loop.

Capítulo 5 Obrigado pelas memórias

Registros duplos

Existem versões de duas palavras de todas as instruções LDR e STR que visto. A instrução LDRD leva dois registradores para carregar como parâmetros e, em seguida, carrega 64 bits de memória neles. Da mesma forma, para a instrução STRD.

Por exemplo, a Listagem 5-12 carrega o endereço de um dword (ainda com 32 bits) e então carrega o dword em R2 e R3. Em seguida, armazenamos R2 e R3 de volta no mydword.

Listagem 5-12. Exemplo de carregamento e armazenamento de uma palavra dupla

```
LDR R1, =minhapalavra  
LDRD R2, R3, [R1]  
STRD R2, R3, [R1]  
.dados  
minha palavra: .DWORD 0x1234567887654321
```

Isso será útil quando olharmos para a multiplicação.

Resumo

Com este capítulo, agora podemos carregar dados da memória, operá-los nos registradores e então salvar o resultado de volta na memória. Examinamos como as instruções de carregamento e armazenamento de dados nos ajudam com arrays de dados e como elas nos ajudam a indexar dados em loops.

No próximo capítulo, veremos como tornar nosso código reutilizável; afinal, nosso programa em letras maiúsculas não seria útil se pudéssemos chamá-lo sempre que quisermos?

CAPÍTULO 6

Funções e o Pilha

Neste capítulo, examinaremos como organizar nosso código em pequenas unidades independentes chamadas **funções**. Isso nos permite construir componentes reutilizáveis que podemos chamar facilmente de qualquer lugar que desejarmos.

Normalmente, no desenvolvimento de software, começamos com componentes de baixo nível, em seguida, desenvolva-os para criar aplicativos de nível cada vez mais alto. Até agora, sabemos como fazer um loop, realizar lógica condicional e realizar alguma aritmética. Agora, examinamos como compartmentalizar nosso código em blocos de construção.

Apresentamos a **pilha**; esta é uma estrutura de dados de ciência da computação para armazenar dados. Se vamos construir funções reutilizáveis úteis, precisaremos de uma boa maneira de gerenciar o uso de registradores, para que todas essas funções não se sobreponham. No Capítulo 5, “Obrigado pelas memórias”, estudamos como armazenar dados em um segmento de dados na memória principal. O problema com isso é que essa memória existe enquanto nosso programa é executado.

Com funções pequenas, como nosso programa de conversão para letras maiúsculas, elas são executadas rapidamente e podem precisar de alguns locais de memória enquanto são executadas, mas quando terminam, não precisam mais dessa memória. As pilhas nos fornecem uma ferramenta para gerenciar o uso de registro em chamadas de função e uma ferramenta para fornecer memória para funções durante a invocação.

Primeiro, introduzimos vários conceitos de baixo nível e, em seguida, os reunimos para criar e usar funções com eficácia.

Capítulo 6 Funções e a pilha

Pilhas em Raspbian

Em ciência da computação, uma pilha é uma área da memória onde ocorrem duas operações:

- **push:** Adiciona um elemento à área
- **pop:** Retorna e remove o elemento que foi adicionado mais recentemente

Esse comportamento também é chamado de fila **LIFO** (último a entrar, primeiro a sair).

Quando o Raspbian executa um programa, ele fornece uma pilha de 8 MB.

No Capítulo 1, “Iniciando”, mencionamos que o registrador **R13** tinha um propósito especial como Stack Pointer (**SP**). Você deve ter notado que **R13** é nomeado **SP** em **gdb**, e você deve ter notado que quando você depurou programas, ele tinha um valor grande, algo como 0x7efff380. Este é um ponteiro para o local da pilha atual.

O conjunto de instruções ARM32 possui duas instruções para manipular pilhas, Carregue Múltiplos (**LDM**) e Armazene Múltiplos (**STM**). Essas duas instruções têm algumas opções. Eles servem para dar suporte a coisas como se a pilha cresce aumentando os endereços ou diminuindo os endereços - se o **SP** aponta para o final da pilha ou para o próximo local livre na pilha. Essas opções podem ser úteis se você estiver criando sua própria pilha ou para corresponder ao requisito de um sistema operacional diferente. Mas tudo o que queremos é trabalhar com a pilha que o Raspbian nos fornece.

Felizmente, o GNU Assembler oferece pseudo-instruções mais simples que são mapeados de volta para as formas corretas de **LDM** e **STM**. Estes são

```
PUSH {regista}
POP    {regista}
```

O parâmetro **{reglist}** é uma lista de registros, contendo uma vírgula lista separada de registradores e faixas de registradores. Um intervalo de registradores é algo como **R2-R4**, que significa **R2**, **R3** e **R4**, por exemplo:

```
EMPURRE {r0, r5-r12}
POP {r0-r4, r6, r9-r12}
```

Os registradores são armazenados na pilha em ordem numérica, com o registrador mais baixo no endereço mais baixo. Você não deve incluir **PC** ou **SP** nesta lista. A Figura 6-1 mostra o processo de colocar um registrador na pilha e, em seguida, a Figura 6-2 mostra a operação inversa de retirar esse valor da pilha.

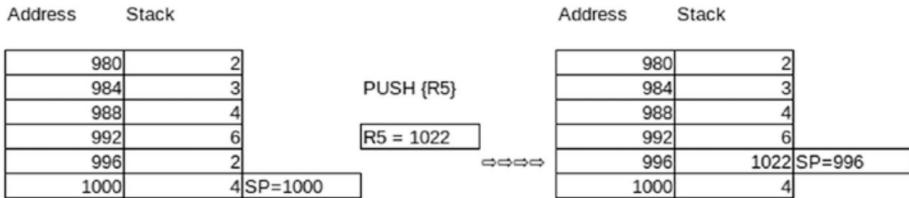


Figura 6-1. Empurrando R5 para a pilha

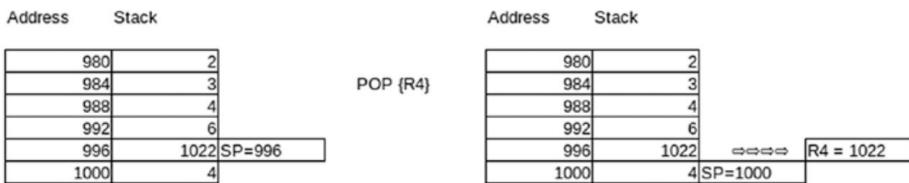


Figura 6-2. Tirando R4 da pilha

Filial com link

Para chamar uma função, precisamos configurar a capacidade da função retornar a execução após o ponto em que a chamamos. Fazemos isso com o outro registrador especial que listamos no Capítulo 1, “Iniciando”, o **Link Register (LR)** que é **R14**. Para fazer uso de **LR**, introduzimos a instrução **Branch with Link (BL)**, que é a mesma que a instrução **Branch (B)**, exceto que coloca o endereço da próxima instrução em **LR** antes de realizar o desvio, dando-nos um mecanismo para retornar da função.

Para retornar de nossa função, utilizamos a **Agência e Câmbio (BX)** instrução. Esta instrução de desvio toma um registrador como seu argumento,

Capítulo 6 Funções e a pilha

permitindo-nos desviar para o endereço armazenado em **LR** para continuar o processamento após a conclusão da função.

Na Listagem 6-1, a instrução **BL** armazena o endereço da seguinte instrução **MOV** em **LR** e, em seguida, desvia para myfunc. Myfunc faz o trabalho útil para o qual a função foi escrita, então retorna a execução para o chamador fazendo com que **BX** desvie para o local armazenado em **LR**, que é a instrução **MOV** seguindo a instrução **BL**.

Listagem 6-1. Código esqueleto para chamar uma função e retornar

```
@ ... outro código ...
```

```
BL      myfunc
```

```
MOV R1, #4 @ ...
```

```
mais código ...
```

```
minhafunção:      @ faça algum trabalho
```

```
BX LR
```

Chamadas de função de aninhamento

Chamamos e retornamos com sucesso de uma função, mas nunca usamos a pilha. Por que introduzimos a pilha primeiro e depois não a usamos? Primeiro pense no que acontece se durante o processamento myfunc chamar outra função. Esperamos que isso seja bastante comum, pois escrevemos código com base na funcionalidade que escrevemos anteriormente. Se myfunc executar uma instrução **BL**, então **BL** copiará o próximo endereço em **LR**, sobrescrevendo o endereço de retorno para myfunc e myfunc não poderá retornar. O que precisamos é de uma maneira de manter uma cadeia de endereços de retorno conforme chamamos função após função. Bem, não uma cadeia de endereços de retorno, mas uma pilha de endereços de retorno.

Capítulo 6 Funções e a pilha

Se myfunc vai chamar outras funções, então ele precisa empurrar LR para a pilha como a primeira coisa que ele faz e retira-a da pilha logo antes de retornar, por exemplo, a Listagem 6-2 mostra esse processo.

Listagem 6-2. Código esqueleto para uma função que chama outra função

```

@ ... outro código ...
BL      myfunc
MOV R1, #4
@ ... mais código ...
-----
minhafunção:    PUSH {LR}
@ fazer algum trabalho...
BL      myfunc2
@ fazer mais algum trabalho...
POP {LR}
BX LR
minhafunc2:     @ faça algum trabalho ....
BX LR

```

Neste exemplo, vemos como a pilha é conveniente para armazenar dados que só precisa existir durante uma chamada de função.

Se uma função, como myfunc, chama outras funções, ela deve salvar **LR**; se não chamar outras funções, como myfunc2, não será necessário salvar **LR**. Os programadores geralmente pressionam e removem LR independentemente, pois se a função for modificada posteriormente para adicionar uma chamada de função e o programador se esquecer de adicionar **LR** à lista de registros salvos, o programa falhará ao retornar e entrará em um loop infinito ou falhará. A desvantagem é que há apenas tanta largura de banda entre a CPU e a memória, então fazer PUSHing e POPing em mais registros requer ciclos de execução extras. A compensação entre velocidade e facilidade de manutenção é uma decisão subjetiva, dependendo das circunstâncias.

Capítulo 6 Funções e a pilha

Parâmetros de função e valores de retorno

Em linguagens de alto nível, as funções recebem parâmetros e retornam seus resultados.

A programação em linguagem assembly não é diferente. Poderíamos inventar nossos próprios mecanismos para fazer isso, mas isso é contraproducente. Eventualmente, desejaremos que nosso código interopere com o código escrito em outras linguagens de programação. Queremos chamar nossas novas funções super-rápidas do código C e podemos chamar funções que foram escritas em C.

Para facilitar isso, há um conjunto de padrões de design para funções de chamada. Se seguirmos isso, nosso código funcionará de maneira confiável, pois outros já resolveram todos os bugs, além de atingirmos o objetivo de escrever código interoperável.

O chamador passa os primeiros quatro parâmetros em **R0**, **R1**, **R2** e **R3**. Se houver parâmetros adicionais, eles serão colocados na pilha. Se tivéssemos apenas dois parâmetros, usaríamos apenas **R0** e **R1**. Isso significa que os quatro primeiros parâmetros já estão carregados nos registradores e prontos para serem processados. Parâmetros adicionais precisam ser retirados da pilha antes de serem processados.

Para retornar um valor ao chamador, coloque-o em **R0** antes de retornar. Se você precisar retornar mais dados, um dos parâmetros seria um endereço para um local de memória onde você pode colocar os dados adicionais a serem retornados. Isso é o mesmo que C, onde você retorna dados por meio de chamada por parâmetros de referência.

Gerenciando os Registros

Se você chamar uma função, é provável que ela tenha sido escrita por um programador diferente e você não saiba quais registradores ela usará. Seria muito ineficiente se você tivesse que recarregar todos os seus registradores toda vez que chamassem uma função. Como resultado, há um conjunto de regras para governar quais registradores uma função pode usar e quem é responsável por salvar cada um:

- **R0–R3:** Estes são os parâmetros da função. O função pode usá-los para qualquer outra finalidade, modificando-os livremente. Se a rotina de chamada precisar deles salvos, deve salvá-los em si.
- **R4–R12:** Podem ser usados livremente pela rotina chamada, mas se ela for responsável por salvá-los. Isso significa que a rotina de chamada pode assumir que esses registradores estão intactos.
- **SP:** Pode ser utilizado livremente pela rotina chamada. O rotina deve POP a pilha o mesmo número de vezes que ele empurre, então está intacto para a rotina de chamada.
- **LR:** A rotina chamada deve preservar isso conforme discutimos na última seção.
- **CPSR:** Nenhuma das rotinas pode fazer suposições sobre o **CPSR**. Na medida em que a rotina chamada é em causa, todas as bandeiras são desconhecidas; da mesma forma, eles são desconhecidos para o chamador quando a função retorna.

Resumo do Algoritmo de Chamada de Função

Rotina de chamada

1. Se precisarmos de qualquer um dos **R0–R4**, salve-os.
2. Mova os primeiros quatro parâmetros para os registros **R0–R4**.
3. Empurre quaisquer parâmetros adicionais para a pilha.
4. Use **BL** para chamar a função.
5. Avalie o código de retorno em **R0**.
6. Restaure qualquer um dos R0–R4 que salvamos.

Capítulo 6 Funções e a pilha

função chamada

1. EMPURRE **LR** e **R4–R12** na pilha.
 2. Faça o nosso trabalho.
 3. Coloque nosso código de retorno em **R0**.
 4. POP **LR** e **R4–R12**.
 5. Use a instrução **BX** para retornar a execução ao chamador.
-

Observação Podemos economizar algumas etapas se usarmos apenas R0–R3 para parâmetros de função e códigos de retorno e trabalho de curto prazo. Então, nunca precisamos salvá-los e restaurá-los em chamadas de função.

Eu especifiquei salvar todo **LR** e **R4–R12**, que é a prática mais segura e de melhor manutenção. No entanto, se sabemos que não usamos alguns desses registradores, podemos deixar de salvá-los e economizar algum tempo de execução na entrada e saída da função.

Estas não são todas as regras. Os coprocessadores também possuem registradores que podem precisar ser salvos. Discutiremos essas regras quando discutirmos os coprocessadores.

Maiúscula revisitada

Vamos organizar nosso exemplo de letras maiúsculas do Capítulo 5, “Thanks for the Memories,” como uma função apropriada. Moveremos a função para seu próprio arquivo e modificaremos o makefile para criar tanto o programa de chamada quanto a função em letras maiúsculas.

Primeiro crie um arquivo chamado main.s contendo a Listagem 6-3 para o aplicativo de direção.

Listagem 6-3. Programa principal para exemplo de letras maiúsculas

```

@ Programa Assembler para converter uma string em @
todas as letras maiúsculas chamando uma função. @

@ R0-R2 - parâmetros para serviços de função linux
@ R1 - endereço da string de saída
@ R0 - endereço da string de entrada
@ R5 - caractere atual sendo processado
@ R7 - número da função linux @

.global_start          @ Forneça o endereço inicial do programa

_start: LDR R0, =instr @ início da string de entrada
        LDR      R1, =outstr @ endereço da string de saída
        BL       superior

@ Configure os parâmetros para imprimir nosso número hexadecimal
@ e depois chame o Linux para fazer isso.

        MOV R2,R0 @ código de retorno é o comprimento da string
        MOV R0, #1           @ 1 = StdOut
        LDR R1, =outstr @ string to print @ linux write
        MOV R7, n° 4         system call
        SVC 0               @ Chame o linux para enviar a string

@ Defina os parâmetros para sair do programa @ e depois
chame o Linux para fazer isso.

        MOV      0 R0, #0 @ O código de saída é 0x40
        MOV      o programa   R7, #1 @ Chame o linux para encerrar
        SVC      0

```

Capítulo 6 Funções e a pilha

```
.dados
instr: .asciz "Esta é nossa string de teste que iremos converter.\n"
outstr: .preencher 255, 1, 0
```

Agora crie um arquivo chamado upper.s contendo a Listagem 6-4, a função de conversão de letras maiúsculas.

Listagem 6-4. Função para converter strings para letras maiúsculas

```
@
@ Programa Assembler para converter uma string em @
todas as letras maiúsculas. @

@ R1 - endereço da string de saída @ R0 -
endereço da string de entrada @ R4 - string
de saída original para comprimento calc.

@ R5 - caractere atual sendo processado @

.topper global          @ Permitir que outros arquivos chamem esta rotina
superior:      EMPURRAR {R4-R5} @ Salve os registradores que usamos.
               MOV R4, R1
@ O loop é até que o byte apontado por R1 seja um loop diferente de zero:
LDRB R5, [R0], #1 @ carregar caractere e incremento
                  ponteiro
@ Se R5 > 'z' então vá para cont
      CMP R5, #'z'          @ é a letra > 'z'?
      BGT cont.
@ Else if R5 < 'a' então vá para o fim se
      CMP R5, #'a'
      BLT cont @ ir para terminar se
```

Capítulo 6 Funções e a pilha

@ se chegamos aqui, então a letra é minúscula, então converta-a.

SUB R5, #'('a'-'A') cont: @ end

if STRB R5, [R1], #1 CMP R5, #0

@ armazenar caractere para saída str @ parar
ao atingir um nulo
personagem

SUB R0, R1, R4 @ obtém o caractere que está na memória para o topo da pilha

Restaura o registro que usamos.

POP {R4-R5}

BX LR @ Retornar ao chamador

Para criá-los, use o makefile na Listagem 6-5.

Listagem 6-5. Makefile para o exemplo de função maiúscula

```
UPPEROBJ = main.o upper.o
```

```
ifdef DEBUG
```

```
DEBUGFLGS = -g
```

```
senão
```

```
DEBUGFLGS =
```

```
fim se
```

```
LSTFLGS =
```

```
todos: superior
```

```
% .o: %.s
```

```
as $(DEBUGFLGS) $(LSTFLGS) $< -o $@
```

```
superior: $(UPPEROBJ) ld
```

```
-o superior $(UPPEROBJ)
```

Vamos percorrer a chamada de função e examinar o conteúdo de registradores importantes e a pilha. Definimos um ponto de interrupção em _start e

Capítulo 6 Funções e a pilha

dê um único passo pelas primeiras instruções e pare na instrução **BL**. Eu defino **R4** para 12 e **R5** para 13, para que possamos acompanhar como eles são salvos para a pilha.

r4	0xc	12
r5	0xd	13
sp	0x7efff380	0x7efff380
lr	0x0	0
pc	0x10084	0x10084 <_start+16>

Vemos que a instrução **BL** está em 0x10084. Agora vamos executar uma única etapa novamente para executar a instrução BL. Aqui estão os mesmos registros:

r4	0xc	12
r5	0xd	13
sp	0x7efff380	0x7efff380
lr	0x10088	65672
pc	0x100b0	0x100b0 <topper>

O **LR** foi definido como 0x10088, que é a instrução após o **BL** instrução (0x10084+4). O **PC** agora é 0x100b0, apontando para a primeira instrução na rotina superior. A primeira instrução em toupper é a instrução PUSH para salvar os registradores R4 e R5. Vamos percorrer essa instrução passo a passo e examinar os registradores novamente.

r4	0xc	12
r5	0xd	13
sp	0x7efff378	0x7efff378
lr	0x10088	65672
pc	0x100b4	0x100b4 <topper+4>

Vemos que o ponteiro da pilha (**SP**) foi decrementado em 8 bytes (duas palavras) para 0x7efff378. Nenhum dos outros registradores mudou. Colocar registradores na pilha não afeta seus valores; apenas os salva. Se olharmos para a localização 0x7efff378, veremos

```
(gdb) x /4xw 0x7efff378  
0x7efff378: 0x0000000c 0x0000000d 0x00000001 0x7efff504
```

Vemos cópias dos registradores **R4** e **R5** na pilha.

A partir deste pequeno exercício, podemos ver que tipo de pilha o Linux usa, ou seja, é uma pilha descendente; os endereços ficam pequenos à medida que a pilha cresce. Mais **SP** aponta para o último item salvo (e não para o próximo slot livre).

Nota A função toupper não chama nenhuma outra função, então não salvamos **LR** junto com **R4** e **R5**. Se mudarmos para fazer isso, precisaremos adicionar **LR** à lista. Esta versão do toupper pretende ser o mais rápida possível, então não adicionei nenhum código extra para futura manutenção e segurança.

A maioria dos programadores C objetará que esta função é perigosa. Se a string de entrada não for terminada em NULL, ela ultrapassará o buffer da string de saída, sobrescrevendo a memória após o final. A solução é passar um terceiro parâmetro com os comprimentos do buffer e verificar no loop que paramos no final do buffer se não houver nenhum caractere NULL.

Essa rotina processa apenas os caracteres ASCII principais. Ele não lida com os caracteres localizados como é; não será convertido para É.

Empilhar Quadros

Em nossa função maiúscula, não precisávamos de memória adicional, pois podíamos fazer todo o nosso trabalho com os registradores disponíveis. Quando codificamos funções maiores, geralmente exigimos mais memória para nossas variáveis do que cabe nos registradores. Em vez de adicionar confusão à seção .data , armazenamos essas variáveis na pilha.

Capítulo 6 Funções e a pilha

Empurrar essas variáveis na pilha não é prático, já que geralmente precisam acessá-los em uma ordem aleatória, em vez do protocolo **LIFO** estrito que o PUSH/POP impõe.

Para alocar espaço na pilha, usamos uma instrução de subtração para aumentar o empilhar pela quantidade que precisamos. Suponha que precisamos de três variáveis, cada uma com números inteiros de 32 bits, digamos a, b e c. Portanto, precisamos de 12 bytes alocados na pilha (3 variáveis x 4 bytes/palavra).

SUB SP, nº 12

Isso move o ponteiro da pilha 12 bytes para baixo, fornecendo uma região de memória na pilha para colocar nossas variáveis. Suponha que a esteja em **R0**, b em **R1** e c em **R2**, podemos armazená-los usando

STR R0, [SP]	@ Loja a
STR R1, [SP, #4]	@ Loja b
STR R2, [SP, #8]	@ Loja c

Antes do final da função, precisamos executar

ADICIONE SP, nº 12

para liberar nossas variáveis da pilha. Lembre-se, é responsabilidade de uma função restaurar **SP** ao seu estado original antes de retornar.

Esta é a maneira mais simples de alocar algumas variáveis. No entanto, se estivermos fazendo muitas outras coisas com a pilha em nossa função, pode ser difícil acompanhar esses deslocamentos. A maneira como aliviamos isso é com um quadro de pilha. Aqui alocamos uma região na pilha e mantemos um ponteiro para esta região em outro registrador que chamaremos de Frame **Pointer (FP)**. Você pode usar qualquer registrador como **FP**, mas seguiremos a convenção de programação C e usaremos **R11**.

Para usar um quadro de pilha, primeiro definimos nosso ponteiro de quadro para o próximo ponto livre na pilha (ele cresce em endereços decrescentes) e, em seguida, alocamos o espaço como antes:

SUB FP, SP, nº 4

SUB SP, nº 12

Agora abordamos nossas variáveis usando um deslocamento de **FP**.

STR R0, [FP] @ Loja a

STR R1, [FP, #-4] @ Loja b

STR R2, [FP, #-8] @ Loja c

Quando usamos **FP**, precisamos incluí-lo na lista de registradores que PUSH no início da função e, em seguida, POP no final. Desde **R11**, o **FP** é aquele que somos responsáveis por salvar.

Neste livro, tendemos a não usar **FP**. Isso economiza alguns ciclos na entrada e saída da função. Afinal, em programação em linguagem Assembly, queremos ser eficientes.

Exemplo de quadro de pilha

A Listagem 6-6 é um exemplo de esqueleto simples de uma função que cria três variáveis na pilha.

Listagem 6-6. Função esquelética simples que demonstra uma estrutura de pilha

@ Função simples que aceita 2 parâmetros @ VAR1

e VAR2. A função os soma, @ armazenando o

resultado em uma variável SUM.

@ A função retorna a soma.

@ Presume-se que esta função faça outro trabalho, @
incluindo outras funções.

@ Defina nossas variáveis

.EQU VAR1,

0 .EQU VAR2,

4 .EQU SUM, 8

Capítulo 6 Funções e a pilha

```

SUMFN:    EMPURRAR {R4-R12, LR}

        SUB      SP, nº 12          @ espaço para três valores de 32 bits @ salvar

        FOR     R0, [SP, #VAR1]      passado no parâmetro. @ salvar o

        FOR     R1, [SP, #VAR2]      segundo parâmetro.

```

@ Faça um monte de outros trabalhos, mas não mude o SP.

LDR	R4, [SP, #VAR1]
LDR	T5, [SP, #VAR2]
ADICIONAR	R6, R4, R5
FOR	R6, [SP, #SUM]

@ Fazer outro trabalho

@ Função Epílogo

LDR	R0, [SP, #SOMA]	@ carga soma para retornar
ADICIONAR	SP, nº 12	@ Liberar variáveis locais
POP	{R4-R12, PC}	@ Restaurar regs e retornar

Definindo Símbolos

Neste exemplo, apresentamos a diretiva .EQU Assembler. Esta diretiva nos permite definir os símbolos que serão substituídos pelo Assembler antes de gerar o código compilado. Dessa forma, podemos tornar o código mais legível. Neste exemplo, manter o controle de qual variável é qual na pilha torna o código difícil de ler e propenso a erros. Com a diretiva .EQU, podemos definir o deslocamento de cada variável na pilha uma vez.

Infelizmente, .EQU define apenas números, então não podemos definir todo o “[SP, #4]” tipo cadeia.

Mais uma Otimização

Você pode notar que nosso SUMFN não termina em “**BX LR**”. Esta é uma pequena otimização. A instrução **BX** basicamente move **LR** para o **PC**, então por que não apenas POP **LR** diretamente para o **PC**? Observe que é isso que a instrução POP no

final da rotina faz. Se pressionarmos **LR**, podemos salvar uma instrução dessa maneira. Isso funciona bem, desde que o chamador seja um código Assembly ARM32 regular. Existe outro tipo de código chamado Thumb code, que veremos no Capítulo 15, “Thumb Code”. O **BX** nos permite retornar a um chamador que está sendo executado no modo Thumb, onde o pop-up para o **PC** não fará com que o processador altere a forma como interpreta as instruções.

Macros

Outra maneira de transformar nosso loop de letras maiúsculas em um trecho de código reutilizável é usar macros. O GNU Assembler possui um poderoso recurso de macro; com macros ao invés de chamar uma função, o Assembler cria uma cópia do código em cada local onde é chamado, substituindo quaisquer parâmetros.

Considere esta implementação alternativa de nosso programa em letras maiúsculas; o primeiro arquivo é mainmacro.s contendo o conteúdo da Listagem 6-7.

Listagem 6-7. Programa para chamar nossa macro topver

```
@ @ Programa Assembler para converter uma string em
@ todas as letras maiúsculas chamando uma macro. @ @
R0-R2 - parâmetros para serviços de função linux @ R1 -
endereço da string de saída @ R0 - endereço da string de entrada
@ R7 - número da função linux @
```

```
.include "macro superior.s"
```

```
.global_start          @ Forneça o endereço inicial do programa
_começar:      topper tststr, buffer
```

Capítulo 6 Funções e a pilha

@ Configure os parâmetros para imprimir nosso número hexadecimal
 @ e depois chame o Linux para fazer isso.

```
MOV R2,R0 @ R0 é o comprimento da string
MOV R0, #1           @ 1 = StdOut
LDR R1, =buffer @ string para imprimir
MOV R7, #4 @ linux write system call
SVC 0           @ Chame o linux para enviar a string
```

@ Ligue uma segunda vez com nossa segunda string. topper
 tststr2, buffer

@ Configure os parâmetros para imprimir nosso número hexadecimal
 @ e depois chame o Linux para fazer isso.

```
MOV R2,R0           @ R0 é o comprimento da string
MOV R0, #1           @ 1 = StdOut
LDR R1, =buffer @ string para imprimir
MOV R7, n° 4         @ chamada de sistema de gravação linux
SVC 0           @ Chame o linux para enviar a string
```

@ Defina os parâmetros para sair do programa @ e depois
 chame o Linux para fazer isso.

```
MOV      R0, #0       @ Use 0 código de retorno
MOV      R7, #1       @ Código de comando de serviço 1 termina
                      este programa
SVC      0 @ Chame o linux para terminar
```

.dados

tststr: .asciz "Esta é nossa string de teste que iremos converter.\n"

tststr2: .asciz "Uma segunda string para letras maiúsculas!!
 amortecedor: \n" .fill 255, 1, 0

A macro para colocar a string em maiúsculas está em uppermacro.s contendo
 Listagem 6-8.

Listagem 6-8. Versão macro da nossa função toupper

```

@ Programa Assembler para converter uma string em @
todas as letras maiúsculas. @

@ R1 - endereço da string de saída @
R0 - endereço da string de entrada @
R2 - string de saída original para comprimento calc.
@ R3 - caractere atual sendo processado @

@ label 1 = loop @
label 2 = cont

.MACRO superior instr, outstr
    LDR R0, =\instr
    LDR R1, =\outstr
    MOV R2, R1

    @ O loop é até que o byte apontado por R1 seja diferente de zero @
    1:   LDRB R3, [R0], #1           caractere de carga e ponteiro de
                                incremento

    @ Se R5 > 'z' então vá para cont
    CMP R3, #'z' @ é a letra > 'z'?
    BGT 2f

    @ Else if R5 < 'a' então vá para o fim se
    CMP R3, #'a'
    BLT 2f           @ goto para terminar

    if @ se chegamos aqui então a letra é minúscula, então converta-a.
    SUB R3, #('a'-'A') @ end
    2:   if STRB R3, [R1], #1 @
          armazena o caractere para saída str @ pára ao atingir um
          caractere nulo CMP R3, #0

```

Capítulo 6 Funções e a pilha

```

    BNE 1b          @ loop se o caractere não for nulo
    SUB R0, R1, R2 @ obtém o comprimento subtraindo os ponteiros
.ENDM

```

Incluir diretiva

O arquivo uppermacro.s define nossa macro para converter uma string para letras maiúsculas. A macro não gera nenhum código; apenas define a macro para o Assembler inserir de onde for chamado. Este arquivo não gera um arquivo objeto (.o); em vez disso, ele é incluído por qualquer arquivo que precise usá-lo.

A diretiva .include

```
.include "macro superior.s"
```

pega o conteúdo deste arquivo e o insere neste ponto, para que nosso arquivo de origem fique maior. Isso é feito antes de qualquer outro processamento. Isso é semelhante à diretiva de pré-processador C **#include** .

Definição de macro

Uma macro é definida com a diretiva .MACRO . Isso fornece o nome da macro e lista seus parâmetros. A macro termina na seguinte diretiva .ENDM . A forma da diretiva é

```
.MACRO      nome da macro      parâmetro1, parâmetro2, ...
```

Dentro da macro, você especifica os parâmetros precedendo seus nomes com uma barra invertida, por exemplo, \parâmetro1 para colocar o valor de parâmetro1. Nossa macro superior define dois parâmetros instr e outstr:

```
.MACRO      superior      instr, outstr
```

Capítulo 6 Funções e a pilha

Você pode ver como os parâmetros são usados no código com `\instr` e `\oustr`. Essas são substituições de texto e precisam resultar na sintaxe correta do Assembly ou você receberá um erro.

Etiquetas

Nossos rótulos “loop” e “cont” são substituídos pelos rótulos “1” e “2”.

Isso tira a legibilidade do programa. A razão pela qual fazemos isso é que, se não o fizermos, receberemos um erro informando que um rótulo foi definido mais de uma vez, se usarmos a macro mais de uma vez. O truque aqui é que o Assembler permite que você defina rótulos numéricos quantas vezes quiser. Então, para referenciá-los em nosso código, usamos

`BGT 2f`

`BNE 1b @ loop se o caractere não for nulo`

O `f` após o `2` significa o próximo rótulo `2` na direção direta. o `1b` significa a próxima etiqueta `1` na direção inversa.

Para provar que isso funciona, chamamos `toupper` duas vezes no arquivo `mainmacro.s` para mostrar que tudo funciona e que podemos reutilizar essa macro quantas vezes quisermos.

Por que macros?

As macros substituem uma cópia do código em cada ponto em que são usadas. Isso tornará seu arquivo executável maior. Se você

`objdump -d mainmacro`

você verá as duas cópias do código inseridas. Com funções, não há código extra gerado a cada vez. É por isso que as funções são bastante atraentes, mesmo com o trabalho extra de lidar com a pilha.

Capítulo 6 Funções e a pilha

A razão pela qual as macros são usadas é o desempenho. A maioria dos modelos de Raspberry Pi tem um gigabyte ou mais de memória que é espaço para muitas cópias de código.

Lembre-se que sempre que ramificamos, temos que reiniciar o pipeline de execução, tornando a ramificação uma instrução cara. Com macros, eliminamos o ramal **BL** para chamar a função e o ramal **BX** para retornar.

Também eliminamos as instruções **PUSH** e **POP** para salvar e restaurar quaisquer registradores que usamos. Se uma macro é pequena e a usamos muito, pode haver uma economia considerável de tempo de execução.

Nota Observe na implementação da macro de toupper que usei apenas os registradores **R0–R3**. Isso é para tentar evitar o uso de quaisquer registradores importantes para o chamador. Não existe um padrão de como regular o uso de cadastros com macros, como ocorre com as funções, então cabe a você, programador, evitar conflitos e bugs estranhos.

Resumo

Neste capítulo, cobrimos a pilha ARM e como ela é usada para ajudar a implementar funções. Cobrimos como escrever e chamar funções como uma primeira etapa para criar bibliotecas de código reutilizável. Aprendemos como gerenciar o uso de registradores, para que não haja conflitos entre nossos programas de chamada e nossas funções. Aprendemos o protocolo de chamada de função que nos permitirá interoperar com outras linguagens de programação. Vimos a definição de armazenamento baseado em pilha para variáveis locais e como usar isso memória.

Por fim, cobrimos a capacidade de macro do GNU Assembler como uma alternativa para funções em determinados aplicativos de desempenho crítico.

CAPÍTULO 7

Operacional Linux Serviços do sistema

No Capítulo 1, “Iniciando”, precisávamos da capacidade de sair do programa e exibir uma string. Usamos o Raspbian Linux para fazer isso, invocando diretamente os serviços do sistema operacional. Em todas as linguagens de programação de alto nível, existe uma biblioteca de tempo de execução que inclui wrappers para chamar o sistema operacional. Isso faz parecer que esses serviços fazem parte da linguagem de alto nível. Neste capítulo, veremos o que essas bibliotecas de tempo de execução fazem nos bastidores para chamar o Linux e quais serviços estão disponíveis para nós.

Vamos revisar a sintaxe para chamar o sistema operacional e o erro códigos devolvidos para nós. Há uma lista completa de todos os serviços e códigos de erro no Apêndice B, “Linux System Calls”.

tantos serviços

Se você observar o Apêndice B, “Linux System Calls”, parece que existem cerca de 400 serviços do sistema Linux. Por que tantos? O Linux completou 25 anos em 2019. Isso é bastante antigo para um programa de computador. Esses serviços foram adicionados peça por peça ao longo de todos esses anos. O problema desse desenvolvimento de retalhos surge na compatibilidade de software. Se uma chamada de serviço exigir uma alteração de parâmetro, o serviço atual não poderá ser alterado sem interromper vários programas.

Capítulo 7 Serviços do sistema operacional Linux

A solução para a incompatibilidade de software geralmente é apenas adicionar uma nova função. A função antiga se torna um wrapper fino que traduz os parâmetros para o que a nova função requer. Exemplos disso são quaisquer rotinas de acesso a arquivos que usam um deslocamento em um arquivo ou um parâmetro de tamanho. Originalmente, o Linux de 32 bits suportava apenas arquivos de 32 bits (4 GB). Isso se tornou muito pequeno e todo um novo conjunto de rotinas de E/S de arquivo foram adicionados que usam um parâmetro de 64 bits para deslocamentos e tamanhos de arquivo. Todas essas funções são como as versões de 32 bits, mas com 64 anexado aos seus nomes.

Felizmente, a documentação do Linux para todos esses serviços é bastante bom. Ele é totalmente orientado para programadores C, portanto, qualquer pessoa que o use deve conhecer C o suficiente para converter o significado para o que é apropriado para a linguagem que está usando.

O Linux é um sistema operacional poderoso - como programador de aplicativos ou sistemas, certamente o ajudará a aprender a programar o sistema Linux. Existem muitos serviços para ajudá-lo. Você não quer reinventar tudo isso sozinho, a menos que esteja criando um novo sistema operacional.

Convenção de chamada

Usamos duas chamadas de sistema: uma para gravar dados ASCII no console e a segunda para sair de nosso programa. A convenção de chamada para chamadas de sistema é diferente daquela para função. Ele usa uma interrupção de software para alternar o contexto do nosso programa em nível de usuário para o contexto do kernel do Linux.

A convenção de chamada é

1. **r0–r6:** Parâmetros de entrada, até sete parâmetros para a chamada do sistema.
2. **r7:** O número de chamada do sistema Linux (consulte o Apêndice B, “Chamadas do sistema Linux”).

3. Chame a interrupção de software 0 com “**SVC 0**”.
4. **R0:** O código de retorno da chamada (consulte o Apêndice B, “Chamadas do sistema Linux”).

A interrupção de software é uma maneira inteligente de chamarmos rotinas no Linux kernel sem saber onde estão armazenados na memória. Ele também fornece um mecanismo para executar em um nível de segurança mais alto enquanto a chamada é executada. O Linux verificará se você tem os direitos de acesso corretos para executar a operação solicitada e retornará um código de erro como EACCES (13) se você for negado.

Embora não siga a convenção de chamada de função do Capítulo 6, “Funções e a pilha”, o mecanismo de chamada do sistema Linux preservará todos os registradores não usados como parâmetros ou código de retorno. Quando as chamadas do sistema exigem um grande bloco de parâmetros, elas tendem a usar um ponteiro para um bloco de memória como um parâmetro, que contém todos os dados de que precisam. Portanto, a maioria das chamadas do sistema não usa tantos parâmetros.

O código de retorno para essas funções geralmente é zero ou um número positivo para sucesso e um número negativo para falha. O número negativo é o negativo dos códigos de erro no Apêndice B, “Linux System Calls”. Por exemplo, a chamada open para abrir um arquivo retorna um descritor de arquivo se for bem-sucedida. Um descritor de arquivo é um pequeno número positivo, então um número negativo se falhar, onde é o negativo de uma das constantes no Apêndice B, “Linux System Calls”.

Estruturas

Muitos serviços do Linux usam ponteiros para blocos de memória como seus parâmetros. O conteúdo desses blocos de memória é documentado com estruturas C, portanto, como programadores Assembly, temos que fazer engenharia reversa do C e duplicar a estrutura de memória. Por exemplo, o serviço nanosleep permite que seu programa durma por alguns nanosegundos; é definido como

```
int nanosleep(const struct timespec *req, struct timespec *rem);
```

Capítulo 7 Serviços do sistema operacional Linux

e então o struct timespec é definido como

```
struct timespec { time_t
    tv_sec; longo          /* segundos */ /*
    tv_nsec;               nanossegundos */
};
```

Em seguida, devemos descobrir que esses são dois inteiros de 32 bits e, em seguida, definir em Assembly

```
timespecsec: .word 0
timespecnano: .word 100000000
```

Para usá-los, carregamos seus endereços nos registradores para os dois primeiros Parâmetros:

```
ldr      r0, =timespecsec r1,
ldr      =timespecsec
```

Estaremos usando a função nanosleep no Capítulo 8, “Programação de pinos GPIO”, mas isso é típico do que é necessário para chamar diretamente alguns serviços do Linux.

Invólucros

Ao invés de descobrir todos os registradores cada vez que queremos chamar um serviço Linux, vamos desenvolver uma biblioteca de rotinas ou macros para facilitar nosso trabalho. A linguagem de programação C inclui wrappers de chamada de função para todos os serviços do Linux; veremos como usá-los no Capítulo 9, “Interagindo com C e Python”.

Em vez de duplicar o trabalho da biblioteca de tempo de execução C, desenvolveremos uma biblioteca de chamadas do sistema Linux usando a funcionalidade de macro do GNU Assembler. Não desenvolveremos isso para todas as funções, apenas para as funções de que precisamos. A maioria dos programadores faz isso e, com o tempo, suas bibliotecas se tornam bastante extensas.

Um problema com macros é que muitas vezes você precisa de várias variantes com diferentes tipos de parâmetros. Por exemplo, às vezes você pode querer chamar a macro com um registrador como parâmetro e outras vezes com um valor imediato.

Convertendo um arquivo para letras maiúsculas

Neste capítulo, apresentamos um programa completo para converter o conteúdo de um arquivo de texto para letras maiúsculas. Usaremos nossa função toupper do Capítulo 6, “Funções e a pilha”, e praticaremos loops de codificação e instruções if.

Para começar, precisamos de uma biblioteca de rotinas de E/S de arquivo para ler em nosso arquivo de entrada e, em seguida, grave a versão em maiúsculas em outro arquivo. Se você já fez alguma programação C, isso deve parecer familiar, já que o tempo de execução C fornece uma camada fina sobre esses serviços. Criamos um arquivo fileio.s contendo a Listagem 7-1 para fazer isso.

Listagem 7-1. Macros para nos ajudar a ler e escrever arquivos

@ Várias macros para executar E/S de arquivo

- @ O parâmetro fd precisa ser um registrador.
- @ Usa R0, R1, R7.
- @ O código de retorno está em R0.

.include "unistd.s"

```
.equ O_RDONLY, 0 .equ
O_WRONLY, 1 .equ
O_CREAT, 0100 .equ
S_RDWR, 0666

.macro openFile ldr      fileName, flags r0,
                      =\fileName r1, #\flags
```

movimento

Capítulo 7 Serviços do sistema operacional Linux

```

movimento      r2, #S_RDWR @ direitos de acesso RW r7,
movimento      #sys_open
SVC            0
.endm

.macro readFile fd, buffer, length r0, \fd @ file
    movimento      descriptor r1, =\buffer#\length r7, #sys_read
    ldr
    movimento
    movimento
    SVC            0
.endm

.macro writeFile fd, buffer, comprimento @ descriptor
    movimento      r0, \fd r1,           de arquivo
    ldr            =\buffer r2,
    movimento      \length r7,
    movimento      #sys_write
    SVC            0
.endm

.macro flushClose fd
    @fsync syscall
        movimento      r0, \fd r7,
        movimento      #sys_fsync
        SVC            0
    @close syscall
        movimento      r0, \fd r7,
        movimento      #sys_close
        SVC            0
.endm

```

Agora precisamos de um programa principal para orquestrar o processo. Chamaremos isso de main.s contendo o conteúdo da Listagem 7-2.

Listagem 7-2. Programa principal para o nosso programa de conversão de caixas

```
@  

@ Programa Assembler para converter uma string em @ todas as  

letras maiúsculas chamando uma função. @  

@ R0-R2, R7 - usado por macros para chamar linux  

@ R8 - descritor de arquivo de entrada  

@ R9 - descritor de arquivo de saída  

@ R10 - número de caracteres lidos @
```

```
.include "fileio.s"  

.equ BUFFERLEN, 250
```

.global_start	@ Forneça o endereço inicial do programa	
_começar:	openFile inFile, O_RDONLY @ salvar	
MOVIMENTOS	R8, R0	descritor de arquivo
BPL	nxtfil	@ arquivo de número pos aberto ok
MOV	R1, #1	@ stdout
LDR	R2, =inpErrsz	@ Mensagem de erro
LDR	R2, [R2]	
escreverArquivo	R1, inpErr,	R2 @ imprime a saída de erro
B		

nxtfil: openFile outFile, O_CREAT+O_WRONLY		
MOVIMENTOS	R9,	@ salvar descritor de arquivo @
BPL	circuito R0	arquivo de número pos aberto ok
MOV	R1, nº 1	
LDR	R2, =outErrsz	
LDR	R2, [R2]	
escreverArquivo	R1, outErr,	R2 exit
B		

Capítulo 7 Serviços do sistema operacional Linux

@ percorre o arquivo até terminar.

readFile MOV R10, R0 R1, #0 R8, buffer, loop BUFFERLEN:

MOV		@ Manter o comprimento lido @ Terminador nulo para string
STRB		@ configura a chamada para toupper e chama a função R0, @ coloca null no final do buffer @.primeiro parâmetro para toupper LDR R1, [R0, R10]
LDR	R1, =outBuf	
BL	topper	
escreverArquivo	R9, outBuf, R10	
CMP	R10, loop #BUFFERLEN	
BEQ		
flushClose	R8 flushClose	
	R9	

@ Defina os parâmetros para sair do programa @ e depois chame o Linux para fazer isso.

saída: MOV R0, #0 @ Use 0 código de retorno R7, #1

MOV		@ Termos do código de comando 1
SVC	0	@ Chame o linux para encerrar

.dados

inFile: .asciz "main.s"

outArquivo: .asciz "upper.txt" .fill

amortecedor: BUFFERLEN + 1, 1, 0 .fill BUFFERLEN

outBuf: + 1, 1, 0 inpErr: .asciz "Falha ao abrir

arquivo de entrada.\n" inpErrsz: Falha ao abrir o arquivo de saída.\n"

outErrsz: .word .-outErr

O makefile está contido na Listagem 7-3.

Listagem 7-3. Makefile para nosso programa de conversão de arquivos

```
UPPEROBJ = main.o upper.o

ifdef DEBUG
DEBUGFLGS = -g
senão
DEBUGFLGS =
fim se
LSTFLGS =

todos: superior

%.o: %.s
    as $(DEBUGFLGS) $(LSTFLGS) $< -o $@

superior: $(UPPEROBJ)
    ld -o superior $(UPPEROBJ)
```

Esse programa usa o arquivo upper.s do Capítulo 6, “Funções e a pilha”, que contém a versão da função de nossa lógica de letras maiúsculas. O programa também usa os unistd.s do Apêndice B, “Linux System Calls”, que fornece definições significativas dos números das funções de serviço do Linux.

Se você criar este programa, observe que ele tem apenas 13 KB de tamanho. Esse é um dos apelos da programação em linguagem Assembly pura. Não há nada extra adicionado ao programa - nós controlamos cada byte - nenhuma biblioteca misteriosa ou tempos de execução adicionados.

Observação Os arquivos nos quais este programa opera são codificados na seção .data . Sinta-se à vontade para alterá-los, brincar com eles, gerar alguns erros para ver o que acontece. Siga uma única etapa do programa em **gdb** para garantir que você entenda como ele funciona.

Capítulo 7 Serviços do sistema operacional Linux

Abrindo um arquivo

O serviço aberto do Linux é típico de um serviço do sistema Linux. Leva três parâmetros:

1. **Nome do arquivo:** o arquivo a ser aberto como terminado em NULL

corda.

2. **Flags:** Para especificar se estamos abrindo para

leitura ou gravação ou se deseja criar o arquivo. Incluímos algumas diretivas .EQU com os valores de que precisamos (usando os mesmos nomes do tempo de execução C).

3. **Modo:** O modo de acesso do arquivo ao criá-lo

o arquivo. Incluímos alguns define, mas em octal são os mesmos parâmetros para o **chmod**
Comando Linux.

O código de retorno é um descritor de arquivo ou um código de erro. Como muitos Serviços Linux, a chamada se encaixa em um único código de retorno tornando os erros negativos e os resultados bem-sucedidos positivos.

Verificação de erros

Os livros tendem a não promover boas práticas de programação para verificação de erros. Os programas de exemplo são mantidos o menor possível, de modo que as ideias principais explicadas não se percam em um mar de detalhes. Este é o primeiro programa onde testamos quaisquer códigos de retorno. Em parte, tivemos que desenvolver código suficiente para poder fazer isso, e o segundo código de verificação de erros tende a não revelar nenhum novo conceito.

Chamadas de abertura de arquivo são propensas a falhar. O arquivo pode não existir, talvez porque estamos na pasta errada ou podemos não ter direitos de acesso suficientes ao arquivo. Geralmente, verifique o código de retorno para cada chamada do sistema ou função que você chama, mas praticamente os programadores são preguiçosos e tendem a verificar apenas aqueles que provavelmente falharão. Neste programa, verificamos as duas chamadas abertas de arquivo.

Capítulo 7 Serviços do sistema operacional Linux

Primeiramente, temos que copiar o descritor do arquivo para um registrador que não será substituído, então o movemos para R8. Fazemos isso com uma instrução MOVS, então o CPSR será definido.

MOVIMENTOS	R8, R0	@ salvar descritor de arquivo
------------	--------	-------------------------------

Isso significa que podemos testar se é positivo e, em caso afirmativo, passar para a próxima parte do código.

BPL	nxtfil @ arquivo de número pos aberto ok
-----	--

Se a ramificação não for tomada, openFile retornará um número negativo.

Aqui, usamos nossa rotina writeFile para gravar uma mensagem de erro em stdout e, em seguida, ramificamos para o final do programa para sair.

MOV	R1, #1 @ stdout
LDR	R2, =inpErrsz @ Mensagem de erro sz
LDR	R2, [R2]
writeFile R1, inpErr, R2 @ imprimir o erro	
B	saída

Em nossa seção .data, definimos as mensagens de erro da seguinte forma:

```
inpErr: .asciz "Falha ao abrir arquivo de entrada.\n" inpErrsz: .word .-inpErr
```

Vimos .asciz e isso é padrão. Para writeFile, precisamos do comprimento da string para gravar no console. No Capítulo 1, “Primeiros passos”, contamos os caracteres em nossa string e colocamos o número embutido em nosso código. Poderíamos fazer isso aqui também, mas as mensagens de erro começam a ficar longas e contar os caracteres parece algo que o computador deveria fazer. Poderíamos escrever uma rotina como a função **strlen()** da biblioteca C para calcular o comprimento de uma string terminada em NULL. Em vez disso, usamos um pequeno truque do GNU Assembler. Adicionamos uma diretiva .word logo após a string e a inicializamos com “.-inpErr”. O que contém o endereço atual em que o Assembler está enquanto trabalha. Portanto, o endereço atual logo “.” é uma variável Assembler especial

Capítulo 7 Serviços do sistema operacional Linux

a string é o comprimento. Agora as pessoas podem revisar o texto da mensagem de erro para o conteúdo de seus corações, sem precisar contar os caracteres cada vez.

A maioria dos aplicativos contém um módulo de erro, portanto, se uma função falhar, o módulo de erro será chamado. Em seguida, o módulo de erro é responsável por relatar e registrar o erro. Dessa forma, o relatório de erros pode ser bastante sofisticado sem sobrecarregar o restante do código com códigos de tratamento de erros. Outro problema com o código de tratamento de erros é que ele tende a não ser testado. Muitas vezes, coisas ruins podem acontecer quando um erro finalmente acontece e problemas com o manifesto de código não testado anteriormente.

Looping

Em nosso loop, nós

1. Leia um bloco de 250 caracteres do arquivo de entrada.
2. Acrescente um terminador NULL.
3. Chame o topo.
4. Grave os caracteres convertidos no arquivo de saída.
5. Se ainda não terminarmos, vá para o topo do loop.

Verificamos se terminamos

```
CMP      R10, loop #BUFFERLEN
BEQ
```

R10 contém o número de caracteres retornados do serviço de leitura chamar. Se for igual ao número de caracteres solicitados, desviamos para o loop. Se não for exatamente igual, ou chegamos ao fim do arquivo, então o número de caracteres retornados é menor (e possivelmente 0), ou ocorreu um erro, caso em que o número é negativo. De qualquer maneira, terminamos e caímos na saída do programa.

Resumo

Neste capítulo, fornecemos uma visão geral de como chamar os vários serviços do sistema Linux. Cobrimos a convenção de chamada e como interpretar os códigos de retorno. Não cobrimos o propósito de cada chamada e, em vez disso, encaminhamos o usuário para a documentação do Linux.

Apresentamos um programa para ler um arquivo, convertê-lo em letras maiúsculas e gravá-lo em outro arquivo. Esta é nossa primeira chance de reunir o que aprendemos nos Capítulos 1–6 para construir um aplicativo completo, com loops, instruções if, mensagens de erro e E/S de arquivo.

No próximo capítulo, usaremos chamadas de serviço Linux para manipular os pinos GPIO na placa Raspberry Pi.

CAPÍTULO 8

Programação GPIO

alfinetes

O Raspberry Pi possui um conjunto de pinos General Purpose I/O (**GPIO**) que você pode usar para controlar projetos eletrônicos caseiros. A maioria dos kits iniciais do Raspberry Pi inclui uma placa de ensaio e alguns componentes eletrônicos para brincar. Neste capítulo, veremos a programação de pinos GPIO a partir da linguagem Assembly.

Vamos experimentar uma breadboard contendo vários LEDs e resistores, para que possamos escrever algum código real. Vamos programar os pinos GPIO de duas maneiras, primeiro usando o driver de dispositivo Linux incluído e, em segundo lugar, acessando os registradores do controlador GPIO diretamente.

Visão geral do GPIO

O Raspberry Pi 1 original possui 26 pinos GPIO; o Raspberry Pi mais recente expandiu isso para 40 pinos. Nesta seção, limitaremos nossa discussão aos 26 pinos originais. Eles fornecem energia ou geralmente são programáveis:

- **Pinos 1 e 17:** fornecem alimentação de +3,3 V CC
- **Pinos 2 e 4:** fornecem alimentação +5V DC

Capítulo 8 Programação de pinos GPIO

- **Pinos 6, 9, 14, 20 e 25:** fornecem aterramento elétrico
- **Pinos 3, 5, 7–8, 10–13, 15, 16, 18, 19, 21–24 e 26:** são propósito geral programável

Para os pinos programáveis, podemos usá-los para saída, onde controlamos se eles geram potência ou não (são binários 1 ou 0). Podemos lê-los para ver se há energia, por exemplo, se está conectado a um interruptor.

No entanto, isso não é tudo para o GPIO; além das funções sobre as quais falamos até agora, vários pinos têm funções alternativas que você pode selecionar programaticamente. Por exemplo, os pinos 3 e 5 podem suportar o padrão I2C que permite que dois microchips se comuniquem.

Existem pinos que podem suportar duas portas seriais que são úteis para conectar rádios ou impressoras. Existem pinos que suportam modulação por largura de pulso (PWM) e modulação por posição de pulso (PPM) que convertem digital em analógico e são úteis para controlar motores elétricos.

No Linux, tudo é um arquivo

O modelo para controlar dispositivos no Linux é mapear cada dispositivo para um arquivo. O arquivo aparece em /dev ou /sys e pode ser manipulado com as mesmas chamadas de serviço do Linux que operam em arquivos comuns. Os pinos GPIO não são diferentes. Há um driver de dispositivo Linux para eles que controla o operações de pinos por meio de programas de aplicativos que abrem arquivos e, em seguida, leem e escrevem dados neles.

Os arquivos para controlar o pino GPIO aparecem em /sys/class/pasta gpio. Ao escrever strings de texto curtas nos arquivos aqui, controlamos a operação dos pinos.

Suponha que queremos controlar programaticamente o pino 17; a primeira coisa que fazemos é dizer ao motorista que queremos fazer isso. Escrevemos a string “17” em /sys/class/gpio/export. Se isso for bem-sucedido, agora controlaremos o pino. O driver cria os seguintes arquivos em uma pasta gpio17:

- `/sys/class/gpio/gpio17/direction`: Usado para especificar se o pino é para entrada ou saída
- `/sys/class/gpio/gpio17/value`: Usado para definir ou ler o valor do pino
- `/sys/class/gpio/gpio17/edge`: Usado para definir uma interrupção para detectar alterações de valor
- `/sys/class/gpio/gpio17/active_low`: usado para inverter o significado de 0 e 1

A próxima coisa que fazemos é definir a direção do pino, use-o para entrada ou para saída. Nós escrevemos “in” ou “out” no arquivo de direção para fazer isso.

Agora podemos escrever no arquivo de valor para um pino de saída ou ler o arquivo de valor para um pino de entrada. Para ligar um pino, escrevemos “1” no valor e, para desligá-lo, escrevemos “0”. Quando ativado, o pino GPIO fornece +3,3V.

Quando terminarmos com um pino, devemos escrever seu número de pino em `/sys/class/gpio/unexport`. No entanto, isso será feito automaticamente quando nosso programa terminar.

Podemos fazer tudo isso com as macros que criamos no Capítulo 7, “Linux Operating System Services,” em `fileio.s`. Na verdade, ao fornecer essa interface, você pode controlar os pinos GPIO por meio de qualquer linguagem de programação capaz de ler e gravar arquivos, ou seja, praticamente todas. O Raspbian inclui algumas bibliotecas especiais para controlar os pinos GPIO para Python e Scratch para facilitar, mas nos bastidores eles estão apenas fazendo as chamadas de E/S de arquivo que estamos descrevendo.

Capítulo 8 Programação de pinos GPIO

LEDs piscando

Para demonstrar a programação do GPIO, vamos conectar alguns LEDs a uma breadboard e depois fazê-los piscar em sequência.

Conectaremos cada um dos três LEDs a um pino GPIO (neste caso, 17, 27 e 22) e, em seguida, ao aterramento por meio de um resistor. Precisamos do resistor porque o GPIO é especificado para manter a corrente abaixo de 16mA, ou você pode danificar os circuitos. A maioria dos kits vem com vários resistores de 220 Ohm. Pela lei de Ohm, $I = V / R$, isso faria com que a corrente fosse $3,3V/220\Omega = 15mA$, então está certo. Você precisa ter um resistor em série com o LED, pois a resistência do LED é bastante baixa (normalmente em torno de 13 Ohms e variável).

AVISO: Os LEDs têm um lado positivo e um lado negativo. O lado positivo precisa se conectar ao pino GPIO; invertê-lo pode danificar o LED.

A Figura 8-1 mostra como os LEDs e resistores são conectados em um breadboard.

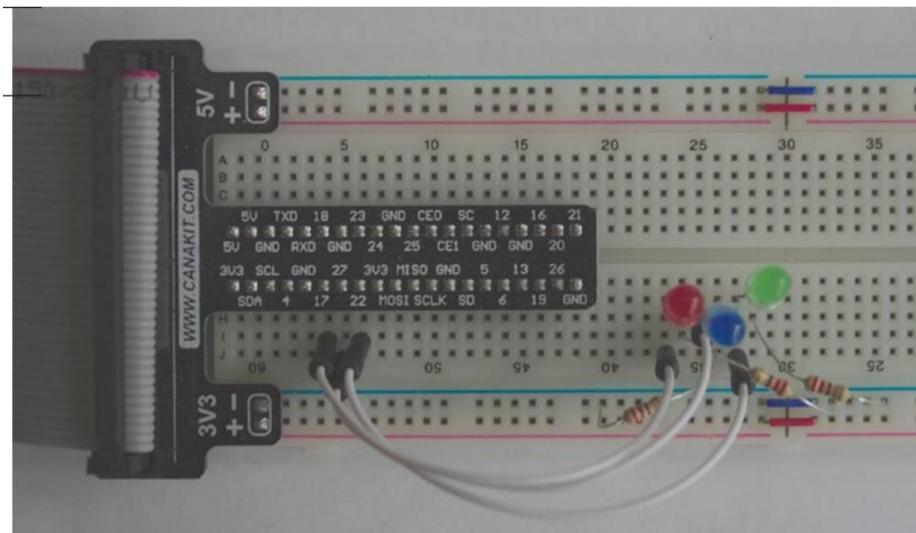


Figura 8-1. Placa de ensaio com LEDs e resistores instalados

Inicialmente, definiremos um conjunto de macros em gpiomacros.s contendo a Listagem 8-1 que usam as macros em fileio.s para executar as várias funções GPIO.

Listagem 8-1. Macros para controlar os pinos GPIO

@ Várias macros para acessar os pinos GPIO @ no
Raspberry Pi. @ @ R8 - descriptor de arquivo. @

.include "fileio.s"

@ Macro nanoSleep para dormir 0,1 segundo @

Chama o serviço nanosleep do Linux que é a função 162.

@ Passe uma referência para uma especificação de tempo em r0 e r1 @

Primeiro é o tempo de entrada para dormir em segundos e nanossegundo

@ Segundo é o tempo restante para dormir se interrompido .macro

`nanoSleep ldr`

r0, =timespecsec r1,

```

ldr      =timespecsec r7,
        movimento #sys_nanosleep
        svc      0
.endm
```

endm

```
macro GPIOExport pin openFile
```

gpioexp, O_WRONLY r8,
r0 @ save o arquivo r8. \pin. #2

flushClose r8

.endm

.macro GPIODirectionOut pino @ copiar pino

no padrão de nome de arquivo ldr r1, =\pin

Capítulo 8 Programação de pinos GPIO

```

ldr          r2, =gpiopinfile r2, #20
adicionar    r3, [r1], #1 @ carregar
lrb          pino e postar incr r3, [r2], #1 @ armazenar no nome
strb         do arquivo e postar incr r3, [r1] r3, [r2] gpiopinfile, O_WRONLY
lrb          r8, r0 writeFile r8, outstr, #3 flushClose r8
strb
abrir arquivo
movimento   @ salve o descriptor de arquivo

```

.endm

.macro GPIOWrite pin, value @ copy pin

```

into filename pattern ldr r1, =\pin r2,
=gpiovaluefile r2, #20 r3, [r1], #1 r3, [r2], #1
ldr
adicionar
lrb          @ carregar o pino e postar o incremento
strb         @ armazenar no nome do arquivo e
            postar o incremento
lrb          r3, [r1] r3,
strb         [r2]
abrir arquivo gpiovaluefile, O_WRONLY r8, r0
movimento   @ salve o descriptor writeToFile value, #1 flushClose r8

```

.endm

.dados

timespecsec: .word 0

timespecnano: .word 100000000 gpioexp:

gpiopinfile: .asciz "sys/class/gpio/gpio<export"

direction"

```
gpiovaluefile: .asciz "/sys/class/gpio/gpioxx/value" .asciz "out" outstr:
```

```
.align 2 @ salvar os usuários deste arquivo tendo  
para fazer isso.
```

Agora precisamos de um programa de controle, main.s contendo a *Listagem 8-2*, para orquestrar o processo.

Listagem 8-2. Programa principal para piscar os LEDs

```
@  
@ Programa Assembler para piscar três LEDs conectados à @ porta  
Raspberry Pi GPIO. @  
  
@ r6 - variável de loop para piscar as luzes 10 vezes @  
  
.include "gpiomacros.s"  
  
.global_start @ Forneça o endereço inicial do  
programa ao vinculador  
  
_start: GPIOExport pin17  
        GPIOExport pin27  
        GPIOExport pin22  
        nanoSleep  
  

```

Capítulo 8 Programação de pinos GPIO

```

GPIOWrite pin27, high
nanoSleep GPIOWrite pin27,
low GPIOWrite pin22, high
nanoSleep GPIOWrite pin22,
low @decrement loop counter
and see if we loop @ Subtract
1 from loop register @ setting status register. subs r6, #1

```

@ Se não tivermos contado até 0, faça um loop

```
bne    laço
```

_fim: movimento	R0, #0 @ Use 0 código de retorno
movimento	R7, #1 @ Código de comando 1 termina
svc	0 @ Linux comando para encerrar

pino17:	.asciz "17" .asciz
pino27:	"27" .asciz "22"
pino22:	
baixo:	.asciz "0"
alto:	.asciz "1"

Este programa é uma aplicação direta das chamadas de serviço do sistema Linux que aprendemos no Capítulo 7, “Serviços do sistema operacional Linux”.

Aproximando-se do Metal

Para programadores de linguagem Assembly, o exemplo anterior não é satisfatório. Quando programamos em Assembly, geralmente manipulamos dispositivos diretamente por motivos de desempenho ou para executar operações que simplesmente não podem ser feitas em linguagens de programação de alto nível. Nesta seção, vamos interagir diretamente com o controlador GPIO.

AVISO: Certifique-se de fazer backup do seu trabalho antes de executar o programa, pois pode ser necessário desligar e ligar novamente. O controlador GPIO controla 54 pinos; o Raspberry Pi expõe apenas 26 ou 40 deles, dependendo do modelo do Pi, para uso externo; muitos dos outros são usados pelo Raspberry Pi para outras tarefas importantes. Na seção anterior, o driver do dispositivo forneceu um nível de proteção, portanto não poderíamos causar nenhum dano facilmente. Agora que estamos escrevendo diretamente para o controlador GPIO, não temos essa proteção; se cometermos um erro e manipularmos os pinos errados, podemos interferir na operação do Raspberry Pi e fazer com que ele trave ou trave.

Memória virtual

Vimos como acessar a memória no Capítulo 5, “Obrigado pelas memórias”, e vimos os endereços de memória em que nossas instruções estão armazenadas no **gdb**. Esses endereços de memória não são endereços de memória física; em vez disso, são endereços de memória virtual. Como um processo Linux, nosso programa recebe um espaço de endereço virtual de 4 GB. 3 GB disso é para nós e 1 GB é para coisas do sistema. Dentro desse espaço de endereço, parte dele é mapeado para a memória física para armazenar nossas instruções de montagem, nossas seções .data e nossa pilha de 8 MB. Além disso, o Linux pode trocar parte dessa memória para armazenamento secundário, como o cartão SD, pois precisa de mais memória física para outros processos. Há muita complexidade no processo de gerenciamento de memória para permitir que dezenas de processos sejam executados independentemente uns dos outros, cada um pensando que tem todo o sistema para si.

Na próxima seção, queremos acesso a endereços de memória física específicos, mas quando solicitamos esse acesso, o Linux retorna um ponteiro de memória virtual diferente do endereço físico que solicitamos.

Tudo bem, pois sabemos que nos bastidores o hardware de gerenciamento de memória no Raspberry Pi fará as traduções de memória entre a memória virtual e a física para nós.

Sobre Raspberry Pi 4 RAM

Você pode se perguntar por que o Raspberry Pi 4 vem com até 4 GB de RAM, mas nosso processo só pode acessar 3 GB dele? No entanto, toda essa RAM será utilizada, pois cada processo e o kernel podem ter até 3 GB de RAM. Na verdade, o controlador de memória no Raspberry Pi possui 40 pinos de endereço, portanto, pode endereçar mais de 4 GB de memória física.

No futuro, se houver uma versão de 16 GB do Pi, essa memória poderá ser usada, mesmo que o Raspbian ainda seja de 32 bits. Cada processo de 32 bits pode mapear diferentes seções de memória, portanto, mesmo que um processo de 32 bits possa acessar apenas 3 GB de memória por vez, ele pode usar mais trocando partes de seu espaço de endereço virtual para diferentes regiões físicas.

Em dispositivos, tudo é memória

O controlador GPIO possui 41 registradores. Não podemos lê-los ou escrevê-los como os registradores da CPU ARM. O conjunto de instruções ARM32 não sabe nada sobre o controlador GPIO e não há instruções especiais para suportá-lo. A maneira como acessamos esses registradores é lendo e escrevendo em locais de memória específicos. Há um circuito no sistema do Raspberry Pi em um chip (SoC) que verá essas leituras e gravações de memória e as redirecionará para os registros do GPIO. É assim que a maioria dos hardwares se comunica. Este é o trabalho dos drivers de dispositivo Linux, para traduzir esses acessos de registro de memória em um conjunto padrão de chamadas de E/S de arquivo.

O endereço de memória para os registros GPIO no Raspberry Pi 2, 3 e 4 é 0x3F200000 (para o Raspberry Pi 0 e 1, é 0x20200000). Parece fácil - sabemos como carregar endereços em registradores e então referenciar a memória armazenada lá. Não tão rápido, se tentássemos isso, nosso programa travaria com um erro de acesso à memória. Isso ocorre porque esses endereços de memória estão fora daqueles atribuídos ao nosso programa e não temos permissão para usá-los. Nossa primeira tarefa, então, é obter acesso.

Isso nos leva de volta a tudo sendo um arquivo no Linux. Existe um arquivo que nos dará um ponteiro que podemos usar para acessar esses locais de memória, da seguinte forma:

1. Abra o arquivo /dev/mem.
2. Em seguida, pedimos /dev/mem para mapear os registros para

GPIO em nosso espaço de memória. Fazemos isso com o serviço Linux mmap2. Mmap2 leva os seguintes parâmetros:

- **R0:** Dica para o endereço virtual que desejamos. Nós realmente não me importo e usarei NULL, o que dá ao Linux total liberdade de escolha.
- **R1:** Comprimento da região. Deve ser um múltiplo de 4096, o tamanho da página de memória.
- **R2:** proteção de memória necessária.
- **R3:** Descritor de arquivo para abrir /dev/mem.
- **R4:** Offset na memória física em páginas de 4096 bytes (usaremos 0x3f200000/4096).

Essa chamada retornará um endereço virtual em **R0** que mapeia para o endereço físico que solicitamos. O mmap original levou um deslocamento em bytes para o endereço físico; isso restringiu a chamada ao mapeamento dos primeiros 4 GB de memória. A chamada mmap2 mais recente leva o endereço em páginas, permitindo uma maior variedade de endereços físicos sem a necessidade de ir para 64 bits completos. Esta função retorna um pequeno número negativo se falhar.

Registros em Bits

Abordaremos apenas os registradores de que precisamos para configurar nossos pinos para saída e, em seguida, definir os bits para piscar os LEDs. Se você estiver interessado na funcionalidade completa, verifique a folha de dados da Broadcom para o controlador GPIO.

Capítulo 8 Programação de pinos GPIO

Embora tenhamos mapeado esses registros para locais de memória, eles nem sempre agem como memória. Alguns dos registradores são somente para gravação e, se os leremos, não travaremos, apenas leremos alguns bits aleatórios. A Broadcom define o protocolo de interação com os registradores; é uma boa ideia seguir exatamente a documentação deles. Eles não são como registradores de CPU ou memória real. O circuito está interceptando nossas leituras e gravações de memória nesses locais, mas apenas agindo nas coisas que ele entende. Nas seções anteriores, o driver de dispositivo Linux para o GPIO ocultou todos esses detalhes de nós.

Registros de Seleção de Função GPIO

A primeira coisa que precisamos fazer é configurar os pinos que estamos usando para saída. Há um banco de seis registradores para configurar todos os pinos GPIO para entrada ou saída. Esses registros de seleção de função GPIO são denominados GPSEL0–GPSEL5. Cada pino recebe 3 bits em um desses registradores para configurá-lo. Estes são registradores de leitura e escrita. Como cada registrador tem 32 bits, cada um pode controlar dez pinos, com 2 bits deixados sem uso (o GPSEL5 controla apenas quatro pinos). A Tabela 8-1 mostra os detalhes de cada registro selecionado.

Tabela 8-1. Registros de Seleção de Função GPIO

Não.	Endereço	Nome	alfinetes
0	0x3f200000	GPSEL0	0–9
1	0x3f200004	GPSEL1	10–19
2	0x3f200008	GPSEL2	20–29
3	0x3f20000c	GPSEL3	30–39
4	0x3f200010	GPSEL4	40–49
5	0x3f200014	GPSEL5	50–53

Para usar esses registradores, o protocolo é

1. Leia o registro.
 2. Defina os bits do nosso registrador.
 3. Escreva o valor de volta.
-

Nota Devemos ter cuidado para não afetar outros bits no registrador.

A Tabela 8-2 mostra os bits correspondentes a cada pino no registrador GPSEL1.

Tabela 8-2. Número do pino e bits correspondentes para o registro GPSEL1

Número do pino	GPSEL1 bits
10	0–2
11	3–5
12	6–8
13	9–11
14	12–14
15	15–17
16	18–20
17	21–23
18	24–26
19	27–29

Armazenamos 000 nos 3 bits se quisermos inserir a partir do pino e armazenamos 001 nos bits se quisermos escrever no pino.

Capítulo 8 Programação de pinos GPIO

GPIO Output Set e Clear Registers

Existem dois registros para definir pinos e dois registros para limpá-los. O primeiro registrador controla os primeiros 32 pinos e o segundo controla os 22 pinos restantes que não estão acessíveis para nós. A Tabela 8-3 mostra os detalhes desses registros.

Tabela 8-3. O conjunto GP e os registros de pinos limpos

Nº	Endereço	Nome	alfinetes
0	0x3f20001c	GPSET0	0–31
1	0x3f200020	GPSET1	32–53
2	0x3f200028	GPCLR0	0–31
3	0x3f20002c	GPCLR1	32–53

Esses registradores são somente para gravação. Você deve definir o bit para o registro você deseja (com todos os outros bits 0) e escreva esse bit. A leitura desses registradores não tem sentido.

A planilha de dados da Broadcom afirma isso como um recurso, pois eles economizam a leitura do registro primeiro, então é mais fácil definir apenas um único bit do que editar um bit em uma sequência de bits. No entanto, também pode ser que isso tenha economizado alguns circuitos e reduzido o custo do chip controlador.

Mais LEDs piscando

Vamos agora repetir nosso programa de LEDs piscando, mas desta vez usaremos a memória mapeada e acessaremos os registradores do GPIO diretamente. Em primeiro lugar, as macros que fazem o trabalho básico da Listagem 8-3 vão em gpiomem.s.

Listagem 8-3. GPIO suporta macros usando memória mapeada

@ Várias macros para acessar os pinos GPIO @ no Raspberry

Pi. @

@ R8 - endereço do mapa de memória.

@

.include "fileio.s"

.equ pagelen, 4096 .equ

setregoffset, 28 .equ clrregoffset,

40 .equ PROT_READ, 1 .equ

PROT_WRITE, 2 .equ

MAP_SHARED, 1

@ Macro para mapear memória para GPIO Registers .macro

mapMem openFile

devmem, S_RDWR @ open /dev/mem @ fd for

movimentos há erro e imprima mensagem de erro se necessário

1f @ número pos arquivo aberto ok

BPL

MOV R1, #1 @ stdout

LDR R2, =memOpnsz @ Mensagem de erro

LDR R2, [R2]

writeFile R1, memOpnErr, R2 @ imprimir o erro _end

B

@ Setup pode chamar o serviço mmap2 Linux r5, =gpioaddr @

1: ldr endereço que queremos / 4096 @ carregar o endereço r1,

ldr #pagelen @ tamanho da mem que queremos

movimento

Capítulo 8 Programação de pinos GPIO

@ opções de proteção de memória r2,

```

movimento      #(PROT_READ + PROT_WRITE) r3,
movimento      #MAP_SHARED @ opções de compartilhamento de memória
movimento      r0, #0 @ deixe o linux escolher um
                           endereço virtual
movimento      r7, #sys_mmap2          @ serviço mmap2 num @
SVC            0           serviço de chamada
movimentos     r8, r0 @ mantém o endereço virt retornado

@ verifique se há erros e imprima a mensagem de erro @ se
necessário. 2f @ pos number arquivo aberto ok BPL R1, #1
memMapErr, R2 @ iOpndubR2=BeemM
MOV
LDR           @ Mensagem de erro
LDR

```

2:

.endm

@ Macro nanoSleep para dormir 0,1 segundo @ Chama o

ponto de entrada nanosleep do Linux que é a função 162.

@ Passe uma referência para uma especificação de tempo em r0 e r1 @ Primeiro
é o tempo de entrada para dormir em segundos e nanosegundos.

@ Segundo é o tempo restante para dormir se for interrompido (o que ignoramos) .macro nanoSleep ldr

```

r0, =timespecsec r1,
ldr           =timespecsec r7,
movimento     #sys_nanosleep
svc            0

```

.endm

.macro pino GPIODirectionOut

```

ldr r2, =\pin @ deslocamento do registro selecionado ldr r2, [r2] @ carrega
o valor

```

ldr r1, [r8, r2] @ endereço do registrador ldr r3, =\pin
 @ endereço da tabela de pinos add r3, #4 @
 quantidade de carga para deslocar da tabela ldr r3, [r3] @
 valor de carga do deslocamento para o bit para r0, r3 @
 deslocar para a posição lsl r0, r3 @ definir o bit para deslocar para r0, r3 @
 valor da tabela lsl r0, r3 @ definir o bit para deslocar para r0, r3 @
 r2] @ salve-o em reg para fazer o trabalho

.endm

.macro GPIOTurnOn pin, value mov r2,
 r8 @ address of gpio regs add r0, regoffset @ off to
 set reg mov r0, #1 @ 1 bit to shift into pos ldr r3,
 =\pin @ base of pin info table add r3, #8 @ add offset para
 deslocamento amt ldr r3, [r3] @ carregar deslocamento
 da tabela lsl r0, r3 str r0, [r2]

@ faz o turno @
 escreve no registrador

.endm

.macro GPIOTurnOff pin, value mov r2,
 r8 @ address of gpio regs add r0, regoffset @ off set of
 clr reg mov r0, #1 @ 1 bit to shift into pos ldr r3, =\pin @
 base of pin info table add r3, #8 @ add offset para
 deslocamento amt ldr r3, [r3] @ carregar deslocamento da tabela lsl
 r0, r3 str r0, [r2]

@ faz o turno @
 escreve no registrador

.endm

Capítulo 8 Programação de pinos GPIO

```
.dados
```

```
timespecsec: .word 0
```

```
timespecnano: .word 100000000 .asciz "/dev/
```

```
"Falha ao abrir /dev/mem" mem: .asciz
```

```
memOpnErr: memOpnErr .asciz "Falha ao mapear
```

```
memOpnsz: memória\n" .word .-memMapErr .align 4 @ realign
```

```
memMapErr: after strings @ endereço mem do registro gpio /
```

```
memMapsz: 4096 gpioaddr: .word 0x3F200 pin17: .word 4 @  
deslocamento para selecionar o registro
```

```
.word 21 @ deslocamento de bit no registro selecionado .word
```

```
17 @ deslocamento de bit no registro set & clr
```

```
pin22: .word 8 @ offset para selecionar o registro
```

```
.word 6 @ bit offset no registro selecionado .word 22 @ bit  
offset no registro set & clr
```

```
pin27: .word 8 @ offset para selecionar o registro
```

```
.word 21 @ bit offset no registro de seleção .word 27 @ bit  
offset no registro set & clr
```

```
.texto
```

Agora, o programa de condução mainmem.s contém a Listagem 8-4 que é bastante semelhante ao último. As principais diferenças estão nas macros.

Listagem 8-4. Programa principal para as luzes piscantes mapeadas na memória

```
@
```

```
@ Programa Assembler para piscar três LEDs conectados à porta @ Raspberry  
Pi GPIO usando acesso direto à memória. @
```

```
@ r6 - variável de loop para piscar as luzes 10 vezes
```

```

@ .include "gpiomem.s"

.global _start
                @ Forneça o endereço inicial do
                programa ao vinculador

_start: mapMem
        nanoSleep

        GPIODirectionOut pin17
        GPIODirectionOut pin27
        GPIODirectionOut pin22 @
                configura um contador de loop para 10 iterações
        movimento          r6, #10

loop: GPIOTurnOn pin17 nanoSleep

        GPIODesligar pin17
        GPIOTurnOn pin27
        nanoSono
        GPIODesligar pin27
        GPIOTurnOn pin22
        nanoSono

brk1:
        GPIOTurnOff pin22
        @decrement contador de loop e veja se fazemos um loop subs
        r6, #1

@ Se não tivermos contado até 0, faça o loop bne
        laço

_fim: movimento      R0, #0           @ Use 0 código de retorno
        movimento      R7, #1           @ Termos do código de comando 1
        SVC            0               @ Linux comando para encerrar

```

Capítulo 8 Programação de pinos GPIO

O programa principal é o mesmo do primeiro exemplo, exceto que inclui um conjunto diferente de macros.

A primeira coisa que precisamos fazer é chamar a macro mapMem. Isso abre /dev/mem e configura e chama o serviço mmap2 conforme descrito na seção “Em dispositivos, tudo é memória”. Armazenamos o endereço retornado em **R8**, para que seja facilmente acessível a partir do restante das macros. Há verificação de erro no arquivo aberto e nas chamadas mmap2, pois elas podem falhar.

Acesso raiz

Para acessar /dev/mem, você precisa de acesso root, então execute este programa com root acesso via

```
sudo ./flashmem
```

Caso contrário, o arquivo aberto falhará. Nós não tivemos que fazer isso com o último programa, porque o driver do dispositivo GPIO mantém tudo seguro.

Acessar /dev/mem é muito poderoso e dá acesso a toda a memória e todos os dispositivos de hardware.

Esta é uma operação restrita, então precisamos ser root. Os programas que acessam diretamente a memória geralmente são implementados como drivers de dispositivo Linux ou módulos carregáveis do kernel, mas a instalação deles também requer acesso root. Um vírus ou outro malware adoraria ter acesso a todos os arquivos físicos memória.

Acionado por tabela

Não abordaremos multiplicação ou divisão até o Capítulo 10, “Multiplicar, Dividir e Acumular”; sem eles, é difícil calcular os deslocamentos dos pinos dentro desses registradores. A divisão é uma operação lenta e os programadores da linguagem Assembly tendem a evitá-la. A solução comum é usar uma tabela de valores pré-computados, em vez de calcular os valores conforme necessário. Uma pesquisa de tabela é muito rápida e examinamos todos os recursos

no conjunto de instruções ARM para nos ajudar a fazer isso no Capítulo 5, "Thanks for the Memories".

Para cada pino, fornecemos três valores em nossa seção .data :

1. O deslocamento para o registro selecionado (do endereço de memória base)
2. O bit offset no registro selecionado para este pino
3. O deslocamento de bits no registro set e clr

Com isso em mãos, acessar e manipular os registradores de controle GPIO é um piscar de olhos.

Observação Apenas preenchemos essas tabelas para os três pinos que usamos.

Definindo a direção do pino

Comece carregando o deslocamento do registro de seleção para o nosso pino - para o pino 17, é 4.

```
ldr r2, =\pin ldr r2, [r2]           @ deslocamento do registro selecionado
                                         @ carrega o valor
```

Agora use o endereçamento pré-indexado para carregar o conteúdo atual do registrador de seleção. **r8** é o endereço, mais o deslocamento que acabamos de carregar em **r2**.

```
ldr r1, [r8, r2] @ endereço do registrador
```

Agora carregamos o segundo item na tabela, o shift no controle registre-se para nossos 3 bits.

```
ldr r3, =\pin add r3,             @ endereço da tabela de pinos
#4 @ quantidade de carga a deslocar da tabela ldr r3, [r3]
                                         @ valor de carga do turno amt
```

Capítulo 8 Programação de pinos GPIO

Limpe os 3 bits com uma máscara de binário 111 que mudamos para a posição, em seguida, chame bit clear (**bic**) para limpar.

```
mov r0, #0b111 lsl r0,          @ máscara para limpar 3 bits
r3 bic r1, r0                  @ deslocar para a posição @
                                limpar os três bits
```

Movemos um para a posição, para que possamos definir o menor dos 3 bits para 1 usando uma lógica ou instrução (**orr**).

```
mov r0, #1 lsl r0,          @ 1 bit para mudar para pos @
r3                         mudar por valor da tabela @ definir o bit
ou r1, r0
```

Finalmente, agora que definimos nossos 3 bits, escrevemos o valor de volta no Registro de controle GPIO para executar nosso comando.

```
str r1, [r8, r2] @ salve-o em reg para fazer o trabalho
```

Definindo e removendo pinos

Definir e limpar os pinos é mais fácil, pois não precisamos ler o registro primeiro. Só precisamos construir o valor para escrever nele e executá-lo.

Como todos os nossos pinos são controlados por um registrador, temos apenas seu deslocamento definido em uma diretiva .EQU . Pegamos o endereço virtual base e adicionamos isso desvio.

```
r8 add r2, #setregoffset @ @address gpio regs mov r2,
```

Em seguida, queremos ter um registrador com apenas 1 na posição correta. Começamos com 1 e o mudamos para a posição. Procuramos esse valor de deslocamento como o terceiro item em nossa tabela de pesquisa de pinos.

```
r0, #1 ldr r3, =\pin @ base @table for determining position of pin
```

```
add r3, #8 ldr          @ adicionar deslocamento para turno amt  
r3, [r3] @ deslocamento de carga da tabela ls1 r0, r3  
          @ fazer o turno
```

Agora temos **r0** contendo um 1 no bit correto; nós escrevemos de volta para o O GPIO define o registro para ligar o LED.

```
r0, [r2]          @ escreve no registrador
```

Limpar o pino é o mesmo, exceto que usamos o registrador clear ao invés do registrador set.

Resumo

Neste capítulo, desenvolvemos tudo o que aprendemos até agora para escrever um programa para piscar uma série de LEDs conectados às portas GPIO em nosso Raspberry Pi. Fizemos isso de duas maneiras:

1. Usando o driver de dispositivo GPIO acessando os arquivos em /sys/class/gpio
2. Usando o acesso direto à memória, solicitando ao driver do dispositivo /dev/mem que nos forneça um bloco virtual de memória correspondente aos registros de controle do GPIO

Dispositivos de controle são um caso de uso chave para programação em linguagem Assembly. Esperançosamente, este capítulo deu a você uma amostra do que está envolvido.

No Capítulo 9, “Interagindo com C e Python”, aprenderemos como interagir com linguagens de programação de alto nível como C e Python.

CAPÍTULO 9

Interagindo com C e Python

Nos primórdios dos microcomputadores, como o Apple II, as pessoas escreviam aplicativos completos em linguagem Assembly, como o primeiro programa de planilhas VisiCalc. Muitos videogames foram escritos em Assembly para extrair todo o desempenho possível do hardware. Hoje em dia, compiladores modernos como o compilador GNU C geram um código bastante bom e os microprocessadores são muito mais rápidos; como resultado, a maioria dos aplicativos é escrita em uma coleção de linguagens de programação, onde cada uma se destaca em uma função específica. Se você está escrevendo um videogame hoje, provavelmente escreveria a maioria em C, C++ ou mesmo C# e, em seguida, usará Assembly para desempenho ou para acessar partes do hardware de vídeo não expostas por meio da biblioteca de gráficos que está usando.

Neste capítulo, veremos como usar componentes escritos em outras linguagens do nosso código de linguagem Assembly e veja como outras linguagens podem fazer uso do código rápido e eficiente que estamos escrevendo em Assembly.

Chamando Rotinas C

Se quisermos chamar funções C, devemos reestruturar nosso programa. O tempo de execução C possui um rótulo `_start`; ele espera ser chamado primeiro e se inicializar antes de chamar nosso programa, como faz ao chamar uma função principal. Se deixarmos nosso rótulo `_start`, obteremos um erro informando que `_start` é mais definido

Capítulo 9 Interagindo com C e Python

que uma vez. Da mesma forma, não chamaremos mais o serviço de programa de encerramento do Linux; em vez disso, retornaremos de main e deixaremos o tempo de execução C fazer isso junto com qualquer outra limpeza que ele executar.

Para incluir o tempo de execução C, podemos adicioná-lo aos argumentos da linha de comando no comando **ld** em nosso makefile. No entanto, é mais fácil compilar nosso programa com o compilador GNU C (que inclui o GNU Assembler), então ele será vinculado ao tempo de execução C automaticamente. Para compilar nosso programa, usaremos

```
gcc -o meuprograma meuprograma.s
```

Isso chamará **como** em myprogram.s e, em seguida, executará o comando **ld**, incluindo o tempo de execução C.

O tempo de execução C nos oferece muitos recursos, incluindo wrappers para a maioria dos serviços do sistema Linux. Há uma extensa biblioteca para manipulação de strings terminadas em NULL, rotinas para gerenciamento de memória e rotinas para conversão entre todos os tipos de dados.

Imprimindo informações de depuração

Um uso útil do tempo de execução C é imprimir dados para rastrear o que nosso programa está fazendo. Escrevemos uma rotina para produzir o conteúdo de um registrador em hexadecimal, e poderíamos escrever mais código Assembly para estender isso ou poderíamos apenas obter o tempo de execução C para fazer isso. Afinal, se estamos imprimindo informações de rastreamento ou depuração, elas não precisam ser eficientes, apenas fáceis de adicionar ao nosso código.

Para este exemplo, usaremos a função `printf` do tempo de execução C para imprimir o conteúdo de um registrador em formato decimal e hexadecimal.

Empacotaremos essa rotina como uma macro e preservaremos todos os registradores com instruções push e pop. Desta forma, podemos chamar a macro sem nos preocuparmos com conflitos de registradores. A exceção é o **CPSR**, que não pode ser preservado, portanto, não coloque essas macros entre as instruções que definem o **CPSR** e, em seguida, teste o **CPSR**. Também fornecemos uma macro para imprimir uma string para fins de registro ou formatação.

A função C printf é poderosa; leva um número variável de argumentos dependendo do conteúdo de uma string de formato. Existe uma extensa documentação on-line sobre printf; então, para uma compreensão mais completa, por favor, dê uma olhada. Chamaremos nossa coleção de macros debug.s, e ela contém o código da Listagem 9-1.

Listagem 9-1. Depurar macros que usam a função printf do tempo de execução C

@ Várias macros para ajudar na depuração

@ Essas macros preservam todos os registros.

@ Cuidado, eles vão mudar cpsr.

```
.macro printReg reg {r0-r4, lr}
    @ para %d r@ R@reg@egapushr21,R@reg
    movimento      r1, #'0' r0, =ptfStr @ printf format
    movimento      str printf @ call printf {r0-r4, lr} @
    movimento      restaurar regs
    adicionar      @ para %c
    ldr
    bl
    pop
.endm

.macro      printStr {r0-          str
                    r4, lr} @ salvar regs r0, =1f @
empurre ldr  carregar print str@printf {r0-r4,
bl        lr} @ restaurar regs 2f @ ramificar em
pop       torno de str
b
1:       .asciz           "\str\n"
.alinhar      4
2:
.endm
```

Capítulo 9 Interagindo com C e Python

```
.dados
```

```
ptfStr: .asciz "R%c = %16d, 0x%08x\n" .align 4 .text
```

Preservando o Estado

Primeiro, empurramos os registradores **R0–R4** e **LR**; nós usamos esses registradores ou printf pode alterá-los. Eles não são salvos como parte do protocolo de chamada de função. No final, nós os restauramos. Isso torna a chamada de nossas macros o mais minimamente possível para o código de chamada.

Chamando printf

Chamamos a função C com estes argumentos:

```
printf("R%c = %16d, 0x%08x\n", reg, Rreg, Rreg);
```

Como existem quatro parâmetros, nós os configuramos em **R0–R3**. Em printf cada string que começa com um sinal de porcentagem ("%"), pega o próximo parâmetro e o formata de acordo com a próxima letra:

- **c** para caractere.
- **d** para decimal.
- **x** para hex.
- **0** significa 0 pad.
- Um número especifica o comprimento do campo a ser impresso.

Observação É importante mover o valor do registro para **R2** e **R3** primeiro, pois preencher os outros registros pode apagar o valor passado se estivermos imprimindo **R0** ou **R1**. Se nosso registrador for **R2** ou **R3**, uma das instruções **MOV** não fará nada. Felizmente, não recebemos um erro ou aviso, então não precisamos de um caso especial.

Passando uma String

Na macro printStr, passamos uma string para imprimir. O assembly não lida com strings, então incorporamos a string no código com uma diretiva `.asciz` e, em seguida, ramificamos em torno dela.

Há uma diretiva `.align` logo após a string, pois as instruções de montagem devem ser alinhadas por palavra. É uma boa prática adicionar uma diretiva `.align` após as strings, pois outros tipos de dados serão carregados mais rapidamente se forem alinhados por palavra.

Geralmente, não gosto de adicionar dados à seção de código, mas para nossa macro, essa é a maneira mais fácil. A suposição é que as chamadas de depuração serão removidas do código final. Se adicionarmos muitas strings, podemos tornar os deslocamentos relativos de PC muito grandes para serem resolvidos. Se isso acontecer, podemos precisar encurtar as cordas ou remover algumas.

Adicionando com Carry revisitado

No Capítulo 2, “Carregando e Adicionando”, fornecemos um exemplo de código para adicionar dois números de 64 bits usando as instruções **ADDS** e **ADC**. O que faltava neste exemplo era alguma maneira de ver a saída. Agora pegaremos addexamp2.s e adicionaremos algumas chamadas às nossas macros de depuração, na Listagem 9-2, para mostrá-lo em ação.

Listagem 9-2. addexamp2.s atualizado para imprimir as entradas e saídas

```
@ @ Exemplo de adição de 64 bits com as instruções ADD/ADC  
@. @
```

```
.include "debug.s"  
.global main @ Fornecer início do programa
```

Capítulo 9 Interagindo com C e Python

@ rotina principal a ser chamada pelo C runtime main:

```
empurre {R4-R12, LR}
```

@ Carregue os registradores com alguns dados

@ O primeiro número de 64 bits é 0x00000003FFFFFF

```
MOV R2, #0x00000003
```

```
MOV R3, #0xFFFFFFFF @Assembler mudará para MVN
```

@ O segundo número de 64 bits é 0x0000000500000001

```
MOV R4, #0x00000005
```

```
MOV R5, #0x00000001
```

```
printStr "Entradas:"
```

```
printReg 2 printReg 3
```

```
printReg 4 printReg 5
```

ADICIONA R1, R3, R5 @ Palavra de ordem inferior

ADC R0, R2, R4 @ Palavra de ordem superior

```
printStr "Saídas:" printReg
```

```
1 printReg 0
```

```
mov r0, #0 @ @ Código de retorno
```

restaura registros e retorna popping para PC pop {R4-R12, PC}

O makefile, na Listagem 9-3, para isso é bem simples.

Listagem 9-3. Makefile para addexam2.s atualizado

```
addexam2: addexam2.s debug.s
```

```
gcc -o addexam2 addexam2.s
```

Capítulo 9 Interagindo com C e Python

Se compilarmos e executarmos o programa, veremos:

```
pi@raspberrypi:~/asm/Capítulo 9 $ make gcc -o  
addexamp2 addexamp2.s pi@raspberrypi:~/asm/  
Capítulo 9 $ ./addexamp2 Entradas:
```

R2 = 3, 0x00000003 -1,
R3 = 0xffffffff
R4 = 5, 0x00000005
R5 = 1, 0x00000001

Saídas:

R1 = 0, 0x00000000
R0 = 9, 0x00000009

```
pi@raspberrypi:~/asm/Capítulo 9 $
```

Além de adicionar as instruções de depuração, observe como o programa é reestruturado como uma função. O ponto de entrada é principal, e segue o protocolo de função de salvar todos os registradores. Como esta é a rotina principal e chamada apenas uma vez, salvamos todos os registradores em vez de tentar rastrear os registradores que realmente estamos usando. Este é o mais seguro, desde então não precisamos nos preocupar com isso enquanto trabalhamos em nosso programa.

Apenas adicionando o tempo de execução C, trazemos uma poderosa caixa de ferramentas para economizar tempo enquanto desenvolvemos nosso aplicativo Assembly completo. No lado negativo, observe que nosso executável cresceu para mais de 8 KB.

Chamando rotinas de montagem de C

Um cenário típico é escrever a maior parte de nosso aplicativo em C e, em seguida, chamar rotinas de linguagem Assembly em casos de uso específicos. Se seguirmos o protocolo de chamada de função do Capítulo 6, “Funções e a pilha”, C não será capaz de diferenciar nossas funções de quaisquer outras funções escritas em C.

Capítulo 9 Interagindo com C e Python

Como exemplo, vamos chamar nossa função toupper do Capítulo 6, “Funções e a pilha”, e chamá-la de C. A Listagem 9-4 contém o código C para uppertst.c para chamar nossa função Assembly.

Listagem 9-4. Programa principal para mostrar chamando nossa função toupper de C

```
//  
// Programa em C para chamar nossa  
rotina Assembly // superior. //  
  
#include <stdio.h>  
  
extern int mytoupper( char *, char * );  
  
#define MAX_BUFFSIZE 255  
int main() {  
  
    char *str = "Isto é um teste."; char  
    outBuf[MAX_BUFFSIZE], int len;  
  
    len = mytoupper(str, outBuf);  
    printf("Antes de str: %s\n", str);  
    printf("Depois de str: %s\n", outBuf);  
    printf("Str len = %d\n", len); retornar(0);  
}
```

O makefile está na Listagem 9-5.

Listagem 9-5. Makefile para C e nossa função superior

```
uppertst: uppertst.c upper.s gcc -o  
          uppertst uppertst.c upper.s
```

Tivemos que mudar o nome da nossa função toupper para mytoupper, pois já existe uma função toupper no tempo de execução C, e isso levou a um erro de definição múltipla. Isso tinha que ser feito tanto no código C quanto no código Assembly. Caso contrário, a função é a mesma do Capítulo 6, “Funções e a pilha”.

Devemos definir os parâmetros e o código de retorno de nossa função para o compilador C. Nós fazemos isso com

```
extern int mytopper( char *, char * );
```

Isso deve ser familiar para todos os programadores C, pois você também deve fazer isso para funções C. Normalmente, você reuniria todas essas definições e as colocaria em um arquivo de cabeçalho (.h) .

No que diz respeito ao código C, não há diferença em usar esta função Assembly do que se a escrevêssemos em C. Quando compilamos e executamos o programa, obtemos

```
pi@raspberrypi:~/asm/Chapter 9 $ make gcc -o  
uppertst uppertst.c upper.s pi@raspberrypi:~/asm/  
Chapter 9 $ ./uppertst Antes de str: Isto é um teste.
```

Após str: ISSO É UM TESTE.

Str len = 16

```
pi@raspberrypi:~/asm/Capítulo 9 $
```

A string está em letras maiúsculas como seria de esperar, mas o comprimento da string parece um maior do que poderíamos esperar. Isso ocorre porque o comprimento inclui o caractere NULL que não é o padrão C. Se realmente quisermos usar muito isso com C, devemos subtrair 1, para que nosso comprimento seja consistente com outras rotinas de tempo de execução C.

Empacotando Nosso Código

Poderíamos deixar nosso código Assembly em arquivos de objetos individuais (`.o`) , mas é mais conveniente para os programadores que usam nossa biblioteca empacotá-los juntos em uma biblioteca. Dessa forma, o usuário de nossas rotinas Assembly só precisa adicionar uma biblioteca para obter todo o nosso código, em vez de possivelmente dezenas de arquivos `.o` . No Linux, há duas maneiras de fazer isso; a primeira maneira é empacotar nosso código em uma biblioteca estática vinculada ao programa. O segundo método é empacotar nosso código como uma biblioteca compartilhada que fica fora do programa de chamada e pode ser compartilhada por vários aplicativos.

Biblioteca Estática

Para empacotar nosso código como uma biblioteca estática, usamos o comando Linux `ar` . Este comando pegará vários arquivos `.o` e os combinará em um único arquivo pela convenção `lib<nosso nome>.a`, que pode então ser incluído em um comando `gcc` ou `ld` . Para fazer isso, modificamos nosso makefile para construir dessa maneira, conforme demonstrado na Listagem 9-6.

Listagem 9-6. Makefile para construir `upper.s` em uma biblioteca vinculada estaticamente

```
LIBOBJJS = superior.o

todos: uppertst2

%.o: %.s
    as $(DEBUGFLGS) $(LSTFLGS) $<-o $@

libupper.a: $(LIBOBJJS) ar -cvq
    libupper.a upper.o

uppertst2: uppertst.c libupper.a gcc -o
    uppertst2 uppertst.c libupper.a
```

Capítulo 9 Interagindo com C e Python

Se construirmos e executarmos este programa, obteremos

```
pi@raspberrypi:~/asm/Capítulo 9 $ make  
as upper.s -o upper.o ar -cvq  
libupper.a upper.o  
a - upper.o gcc  
-o uppertst2 uppertst.c libupper.a pi@raspberrypi:~/  
asm/Capítulo 9 $ ./uppertst2 Antes de str: Este é um teste.
```

Após str: ISSO É UM TESTE.

Str len = 16

```
pi@raspberrypi:~/asm/Capítulo 9 $
```

A única diferença para o último exemplo é que primeiro usamos **as** para compilar upper.s em upper.o e então usamos **ar** para construir uma biblioteca contendo nossa rotina. Se quisermos distribuir nossa biblioteca, incluímos libupper.a, um arquivo de cabeçalho com as definições de função C e alguma documentação. Mesmo se você não estiver vendendo ou distribuindo seu código, construir bibliotecas internamente pode ajudar organizacionalmente a compartilhar código entre programadores e reduzir o trabalho duplicado.

Biblioteca Compartilhada

As bibliotecas compartilhadas são muito mais técnicas do que as bibliotecas vinculadas estaticamente.

Eles colocam o código em um arquivo separado do executável e são carregados dinamicamente pelo sistema conforme necessário. Existem vários problemas, mas vamos apenas abordá-los, como o controle de versão e o posicionamento da biblioteca no sistema de arquivos. Se você decidir empacotar seu código como uma biblioteca compartilhada, esta seção fornece um ponto de partida e demonstra que ele se aplica tanto ao código Assembly quanto ao código C.

A biblioteca compartilhada é criada com o comando **gcc**, fornecendo o parâmetro de linha de comando **-shared** para indicar que queremos criar uma biblioteca compartilhada e, em seguida, o parâmetro **-soname** para nomeá-la.

Capítulo 9 Interagindo com C e Python

Para usar uma biblioteca compartilhada, ela deve estar em um local específico no sistema de arquivos. Podemos adicionar novos locais, mas vamos usar um local criado pelo tempo de execução C, ou seja, /usr/local/lib. Depois de construirmos nossa biblioteca, nós a copiamos aqui e criamos alguns links para ela. Todas essas etapas são necessárias como parte do sistema de controle de versão da biblioteca compartilhada.

Então, para usar nossa biblioteca compartilhada libup.so.1, incluímos -lup no comando gcc para compilar uppertst3. O makefile é apresentado na Listagem 9-7.

Listagem 9-7. Makefile para construir e usar uma biblioteca compartilhada

```
LIBOJBS = superior.o

todos: uppertst3

%.o: %.s
    as $(DEBUGFLGS) $(LSTFLGS) $< -o $@

libup.so.1.0: $(LIBOJBS) gcc
    -shared -Wl,-soname,libup.so.1 -o libup.so.1.0 $(LIBOJBS) mv libup.so.1.0 /usr/local/
    lib ln -sf /usr/local/lib/libup.so.1.0 /usr/local/lib/libup.so.1 ln -sf /usr/local/lib/libup.so.1.0 /
    usr/local/lib/libup.so

uppertst3: libup.so.1.0 gcc -o
    uppertst3 -lup uppertst.c
```

Se executarmos isso, vários comandos falharão. Para copiar os arquivos para /usr/local/lib, precisamos de acesso root, então use o comando sudo. A seguir está a sequência de comandos para construir e executar o programa

```
pi@raspberrypi:~/asm/Chapter 9 $ sudo make -B as upper.s -o
upper.o gcc -shared -Wl,-soname,libup.so.1 -o libup.so.1.0 upper.o
```

Capítulo 9 Interagindo com C e Python

```
mv libup.so.1.0 /usr/local/lib ln -sf /usr/local/
lib/libup.so.1.0 /usr/local/lib/libup.so.1 ln -sf /usr/local/lib/libup .so.1.0 /usr/local/lib/
libup.so gcc -o uppertst3 -lup uppertst.c pi@raspberrypi:~/asm/Chapter 9 $ sudo
ldconfig pi@raspberrypi:~/asm/Chapter 9 $ ./uppertst3 Antes de str: Este é um teste.
```

Após str: ISSO É UM TESTE.

Str len = 16

```
pi@raspberrypi:~/asm/Capítulo 9 $
```

Observe que há uma chamada para o seguinte comando:

```
sudo ldconfig
```

antes de executarmos o programa. Isso faz com que o Linux pesquise todas as pastas que contêm bibliotecas compartilhadas e atualize sua lista principal. Temos que executar isso uma vez depois de compilar nossa biblioteca com sucesso, ou o Linux não saberá que ela existe.

Se você usar objdump para procurar dentro de uppertst3, não encontrará o código para a rotina mytopper; em vez disso, em nosso código principal, você encontrará

```
104c0: ebffffb4 bl 10398 <mytoupper@plt>
```

que chama

```
00010398 <mytoupper@plt>:
```

```
10398: e28fc600 add 1039c: ip, pc, #0, 12 ip, ip,
e28cca10 add 103a0: e5bcfc78 #16, 20 ; 0x10000 pc, [ip, #3192]!
ldr 0xc78
```

O Gcc inseriu essa indireção em nosso código, para que o carregador possa corrigir o endereço quando carregar dinamicamente a biblioteca compartilhada.

Incorporando Código Assembly Dentro do Código C

O compilador GNU C permite que o código Assembly seja incorporado bem no meio do código C. Ele contém recursos para interagir com variáveis C e rótulos e cooperar com o compilador C e otimizador para uso de registro.

A Listagem 9-8 é um exemplo simples, onde incorporamos o algoritmo principal para a função toupper dentro do programa principal C.

Listagem 9-8. Incorporando nossa rotina Assembly diretamente no código C

```
//  
// Programa C para incorporar nossa rotina  
Assembly // superior inline. //
```

```
#include <stdio.h>  
  
extern int mytopper( char *, char * );  
  
#define MAX_BUFFSIZE 255  
int main() {  
  
    char *str = "Isto é um teste."; char  
    outBuf[MAX_BUFFSIZE]; int len;  
  
    asm  
    (  
        "MOV R4, %2\n"  
        "loop:      LDRB R5, [%1], #1\n"  
        "CMP R5, #'z'\n"  
        "BGT cont\n"  
        "CMP R5, #'a'\n"  
        "BLT cont\n"  
        "SUB R5, #('a'-'A')\n"  
    );  
}
```

```

    "cont.:           STRB R5, [%2], #1\n"
    "CMP R5, #0\n"
    "Laço BNE\n"
    "SUB %0, %2, R4\n" :
    "=r" (len): "r" (str),
    "r" (outBuf): "r4", "r5"

);

printf("Antes de str: %s\n", str);
printf("Depois de str: %s\n", outBuf);
printf("Str len = %d\n", len); retornar(0);

}

```

A instrução **asm** nos permite incorporar o código Assembly diretamente em nosso código C. Fazendo isso, poderíamos escrever uma mistura arbitrária de C e Assembly. Retirei os comentários do código Assembly, para que a estrutura do C e do Assembly seja um pouco mais fácil de ler. A forma geral da declaração `asm` é

```

asm asm-qualifiers ( AssemblerTemplate
                      : OutputOperands
                      [ : Operandos de entrada]
                      [ : Clobbers ] ]
                      [: GotoLabels])

```

Os parâmetros são

- `AssemblerTemplate`: string AC contendo o Código de montagem. Existem substituições de macro que começam com `%` para permitir que o compilador C insira as entradas e saídas.

Capítulo 9 Interagindo com C e Python

- OutputOperands: Uma lista de variáveis ou registradores
retornado do código. Isso é necessário, pois é esperado que
a rotina faça algo. No nosso caso, é “=r” (len) onde =r significa
um registrador de saída e queremos que ele vá para a variável C
len.
 - InputOperands: Uma lista de variáveis de entrada ou registros
usado por nossa rotina, neste caso “r” (str), “r” (outBuf)
significando que queremos dois registradores, um segurando str
e outro segurando outBuf. É uma sorte que as variáveis string
C contenham o endereço da string, que é o que queremos no
registrar.
 - Clobbers: Uma lista de registros que usamos e serão
derrotados quando nosso código é executado, neste caso “r4” e “r5”.
 - GotoLabels: Uma lista de rótulos de programas C para os quais
nossa código pode querer pular. Normalmente, esta é uma
saída de erro. Se você pular para um rótulo C, deverá avisar o
compilador com um qualificador goto asm.
- Você pode rotular os operandos de entrada e saída, não o fizemos, e isso
significa que o compilador atribuirá a eles nomes %0, %1, ... como você pode ver usado
no código Assembly.
- Como este é um único arquivo C, é fácil compilar com
- ```
gcc -o uppertst4 uppertst4.c
```
- A execução do programa produz a mesma saída da última seção.  
Se você desmontar o programa, descobrirá que o compilador C evita totalmente  
o uso dos registradores R4 e R5 , deixando-os conosco. Você verá que ele carrega nossos  
registros de entrada das variáveis na pilha, antes de nosso código ser executado e, em  
seguida, copia nosso valor de retorno do registro atribuído para a variável len na pilha. Ele  
não fornece os mesmos registradores que usamos originalmente, mas isso não é um  
problema.

Essa rotina é simples e não tem efeitos colaterais. Se o seu código Assembly está modificando coisas nos bastidores, você precisa adicionar uma palavra-chave volátil à instrução asm para fazer com que a compilação C seja mais conservadora em quaisquer suposições feitas sobre o seu código.

## Chamando Assembly do Python

Se escrevermos nossas funções seguindo o protocolo de chamada de função Raspbian do Capítulo 6, “Funções e a pilha”, podemos seguir a documentação sobre como chamar funções C para qualquer linguagem de programação. Python tem uma boa capacidade de chamar funções C em seu módulo ctypes. Este módulo requer que empacotemos nossas rotinas em uma biblioteca compartilhada. Como Python é uma linguagem interpretada, não podemos vincular bibliotecas estáticas a ela, mas podemos carregar e chamar bibliotecas compartilhadas dinamicamente. As técnicas que abordamos aqui para Python têm componentes correspondentes em muitas outras linguagens interpretadas.

A parte difícil já está feita, construímos a versão da biblioteca compartilhada de nossa função maiúscula; tudo o que devemos fazer é chamá-lo de Python. A Listagem 9-9 é o código Python para uppertst5.py.

### Listagem 9-9. Código Python para chamar mytoupper

```
da importação de ctypes *
```

```
libupper = CDLL("libup.so")
```

```
libupper.mytoupper.argtypes = [c_char_p, c_char_p]
```

```
libupper.mytoupper.restype = c_int
```

```
inStr = create_string_buffer(b"Isto é um teste!") outStr =
```

```
create_string_buffer(250)
```

```
len = libupper.mytoupper(inStr, outStr)
```

## Capítulo 9 Interagindo com C e Python

```
print(inStr.value)
print(outStr.value) print(len)
```

O código é bastante simples; primeiro importamos o módulo `ctypes` para que possamos usá-lo. Em seguida, carregamos nossa biblioteca compartilhada com a função `CDLL`. Este é um nome infeliz, pois se refere a DLLs do Windows, em vez de algo mais neutro do sistema operacional. Como instalamos nossa biblioteca compartilhada em `/usr/local/lib` e a adicionamos ao cache da biblioteca compartilhada do Linux, o Python não tem problemas para encontrá-la e carregá-la.

As próximas duas linhas são opcionais, mas uma boa prática. Eles definem os parâmetros da função e o tipo de retorno para o Python, para que ele possa fazer uma verificação extra de erros.

Em Python, as strings são imutáveis, o que significa que você não pode alterá-las, e elas estão em Unicode, o que significa que cada caractere ocupa mais de 1 byte. Precisamos fornecer as strings em buffers regulares que podemos alterar e precisamos das strings em ASCII em vez de Unicode. Podemos criar uma string ASCII em Python colocando um “`b`” na frente da string; isso significa torná-lo um array de bytes usando caracteres ASCII. A função `create_string_buffer` no módulo `ctypes` cria um buffer de string que é compatível com C (e, portanto, Assembly) para usarmos.

Em seguida, chamamos nossa função e imprimimos as entradas e saídas. O Raspbian vem com o Thonny Python IDE pré-instalado, conforme mostrado na Figura 9-1, para que possamos usá-lo para testar o programa.

## Capítulo 9 Interagindo com C e Python

The screenshot shows the Thonny IDE interface. The top bar displays "Thonny - /home/pi/asm/Chapter 9/uppertst5.py @ 9:34". Below the bar are standard file operations (New, Load, Save, Run, Debug, Over, Into, Out, Resume, Stop) and a "Switch to regular mode" button. The main area has two tabs: "uppertst5.py" (selected) and "Shell". The code in "uppertst5.py" is:

```

1 from ctypes import *
2
3 libupper = CDLL("libup.so")
4
5 libupper.mytoupper.argtypes = [c_char_p, c_char_p]
6 libupper.mytoupper.restype = c_int
7
8 inStr = create_string_buffer(b"This is a test!")
9 outStr = create_string_buffer(250)
10
11 len = libupper.mytoupper(inStr, outStr)
12
13 print(inStr.value)
14 print(outStr.value)
15 print(len)

```

The "Shell" tab shows the output of running the script:

```

>>> %Run uppertst5.py
b'This is a test!'
b'THIS IS A TEST!'
16
>>> |

```

A "Variables" sidebar on the right lists various Python types and their corresponding C structures.

| Name                  | Value                                   |
|-----------------------|-----------------------------------------|
| ARRAY                 | <function ARRAY>                        |
| ArgumentError         | <class 'ctypes.ArgumentError'>          |
| Array                 | <class '_ctypes.Array'>                 |
| BigEndianStructure    | <class '_ctypes.BigEndianStructure'>    |
| CDLL                  | <class 'ctypes.CDLL'>                   |
| CFUNCTYPE             | <function CFUNCTYPE>                    |
| DEFAULT_MODE          | 0                                       |
| LibraryLoader         | <class 'ctypes.LibraryLoader'>          |
| LittleEndianStructure | <class '_ctypes.LittleEndianStructure'> |
| POINTER               | <built-in function POINTER>             |
| PYFUNCTYPE            | <function PYFUNCTYPE>                   |
| PyDLL                 | <class 'ctypes.PyDLL'>                  |
| RTLD_GLOBAL           | 256                                     |
| RTLD_LOCAL            | 0                                       |
| SetPointerType        | <function SetPointerType>               |
| Structure             | <class '_ctypes.Structure'>             |
| Union                 | <class '_ctypes.Union'>                 |
| addressof             | <built-in function addressof>           |
| alignment             | <built-in function alignment>           |
| byref                 | <built-in function byref>               |
| c_bool                | <class 'ctypes.c_bool'>                 |
| c_buffer              | <function c_buffer>                     |

**Figura 9-1.** Nossa programa Python rodando no IDE Thonny

## Resumo

Neste capítulo, examinamos a chamada de funções C a partir de nosso código Assembly. Usamos o tempo de execução C padrão para desenvolver algumas funções auxiliares de depuração para tornar o desenvolvimento de nosso código Assembly um pouco mais fácil. Em seguida, fizemos o inverso e chamamos nossa função Assembly em letras maiúsculas de um programa principal em C.

Aprendemos como empacotar nosso código como bibliotecas estáticas e compartilhadas. Discutimos como empacotar nosso código para consumo. Vimos como chamar nossa função maiúscula do Python, que é típica de linguagens de alto nível com a capacidade de chamar bibliotecas compartilhadas.

No próximo capítulo, Capítulo 10, “Multiplicar, Dividir e Acumular”, voltaremos à matemática. Abordaremos multiplicação, divisão e multiplicação com acúmulo.

## CAPÍTULO 10

# Multiplicar, Dividir e Acumular

Neste capítulo, voltamos à matemática. Cobrimos adição, subtração e uma coleção de operações de bit em nossos registradores de 32 bits.

Agora vamos cobrir a multiplicação e a divisão. O processador ARM tem um excesso de instruções de multiplicação e uma escassez de operações de divisão.

Abordaremos a multiplicação com as instruções de acumulação. Iremos fornecer algumas informações sobre por que o processador ARM tem tantos circuitos dedicados a realizar esta operação. Isso nos levará à mecânica da multiplicação de vetores e matrizes.

## Multiplicação

No Capítulo 7, “Serviços do sistema operacional Linux”, discutimos por que existem tantas chamadas de serviço do Linux e como parte do motivo foi a compatibilidade quando eles precisavam de novas funcionalidades; eles adicionaram uma nova chamada, então a chamada antiga é preservada. As instruções de multiplicação ARM têm uma história semelhante. O Multiply está na arquitetura ARM há muito tempo, mas as instruções originais eram inadequadas e novas instruções foram adicionadas, mantendo as instruções antigas para compatibilidade de software.

A instrução original de 32 bits é

`MUL{S} Rd, Rn, Rm`

## Capítulo 10 Multiplicar, Dividir e Acumular

Esta instrução calcula  $Rd = Rn * Rm$ . Parece bom, mas as pessoas familiarizadas com a multiplicação podem perguntar imediatamente “Todos esses são registradores de 32 bits, então quando você multiplica dois números de 32 bits, você não obtém um produto de 64 bits?” Isso é verdade, e essa é a limitação mais óbvia desta instrução. Aqui estão algumas notas sobre esta instrução:

- **Rd** são os 32 bits mais baixos do produto. Os 32 bits superiores são descartados.
- A versão **MULS** da instrução define apenas o **N** e bandeiras **Z**; ele não define os sinalizadores **C** ou **V**, então você não sabe se estourou.
- Não há versões assinadas e não assinadas separadas; a multiplicação não é como a adição onde o complemento de dois torna as operações iguais.
- Todos os operandos são registradores; operandos imediatos são não permitido.
- **Rd** não pode ser igual a **Rn**.

Para superar algumas dessas limitações, versões posteriores do ARM processador adicionou uma abundância de instruções de multiplicação:

- **SMULL{S}** RdLo, RdHi, Rn, Rm
- **UMULL{S}** RdLo, RdHi, Rn, Rm
- **SMMUL{R}** {Rd}, Rn, Rm
- **SMUL<x><y>** {Rd}, Rn, Rm
- **SMULW<y>** {Rd}, Rn, Rm

A primeira instrução **SMULL** executará a multiplicação de 32 bits com sinal, colocando o resultado de 64 bits em dois registradores. A segunda instrução **UMULL** é a versão não assinada desta. **SMMUL** complementa a instrução **MUL** original fornecendo os 32 bits superiores do produto e descartando os 32 bits inferiores.

## Capítulo 10 Multiplicar, Dividir e Acumular

A multiplicação é uma operação cara, então há algum mérito em multiplicar números pequenos rapidamente. A **SMUL** fornece isso; ele multiplica duas quantidades de 16 bits para fornecer uma quantidade de 32 bits. Os modificadores `<x>` e `<y>` especificam quais 16 bits dos registradores de operandos são usados:

- `<x>` é B ou T. B significa usar a metade inferior (bits [15:0]) de Rn; T significa usar a metade superior (bits [31:16]) de Rn.
- `<y>` é B ou T. B significa usar a metade inferior (bits [15:0]) de Rm; T significa usar a metade superior (bits [31:16]) de Rm.

**SMULW** é uma versão intermediária que multiplica um valor de 32 bits por um valor de 16 bits e mantém apenas os 32 bits superiores do produto de 48 bits. O modificador `<y>` é o mesmo que para **SMUL**. Quando vejo essa instrução sendo usada, um dos operandos geralmente é deslocado para que o produto termine nos 32 bits superiores.

Todas essas instruções têm o mesmo desempenho. A capacidade de detectar quando uma multiplicação é feita (os dígitos restantes são 0) foi adicionado ao processador ARM há algum tempo, então a necessidade de versões mais curtas de multiplicação, na minha opinião, não existe mais. Eu recomendaria sempre usar **SMULL** e **UMULL**, pois há menos coisas para dar errado se seus números mudarem com o tempo.

## Exemplos

A Listagem 10-1 é um código para demonstrar todas as várias instruções de multiplicação. Usamos nosso arquivo debug.s do Capítulo 9, “Interagindo com C e Python”, o que significa que nosso programa deve ser organizado tendo em mente o tempo de execução C.

## Capítulo 10 Multiplicar, Dividir e Acumular

**Listagem 10-1.** Exemplos das várias instruções de multiplicação

@

@ Exemplo de multiplicação de 16 e 32 bits @

.include "debug.s"

.global main @ Forneça o endereço inicial do programa ao vinculador

@ Carregue os registradores com alguns dados

@ Use pequenos números positivos que funcionem para todas as @  
instruções de multiplicação. principal:

empurre {R4-R12, LR}

MOV R2, nº 25

MOV R3, nº 4

printStr "Entradas:"

printReg 2 printReg 3

MUL R4, R2, R3

printStr "MUL R4=R2\*R3:"

printReg 4

SMULL R4, R5, R2, R3

printStr "SMULL R5, R4=R2\*R3:"

printReg 4 printReg 5

UMULL R4, R5, R2, R3

printStr "UMULL R5, R4=R2\*R3:"

printReg 4 printReg 5

## Capítulo 10 Multiplicar, Dividir e Acumular

```

SMMUL R4, R2, R3
printStr "SMMUL R4 = 32 bits principais de R2*R3:"
printReg 4

SMULBB R4, R2, R3
printStr "SMULBB R4 = R2*R3:"
printReg 4

SMULWB R4, R2, R3
printStr "SMULWB R4 = 32 bits superiores de R2*R3:"
printReg 4 mov r0, #0 pop {R4-R12, PC}
@ Código de retorno

```

O makefile é como seria de esperar. A saída é

```

pi@raspberrypi:~/asm/Capítulo 10 $ make gcc -o
mulexamp mulexamp.s pi@raspberrypi:~/asm/
Capítulo 10 $./mulexamp Entradas: R2 =

```

```

25, 0x00000019
R3 = 4, 0x00000004
MUL R4=R2*R3:
R4 = 100, 0x00000064
SMULL R5, R4=R2*R3:
R4 = 100, 0x00000064
R5 = 0, 0x00000000
UMULL R5, R4=R2*R3:
R4 = 100, 0x00000064
R5 = 0, 0x00000000
SMMUL R4 = top 32 bits de R2*R3:
R4 = 0, 0x00000000
SMULBB R4 = R2*R3:
R4 = 100, 0x00000064

```

## Capítulo 10 Multiplicar, Dividir e Acumular

**SMULWB** R4 = 32 bits superiores de R2\*R3:

R4 = 0, 0x00000000

pi@raspberrypi:~/asm/Capítulo 10 \$

Multiplicar é direto, especialmente usando **SMULL** e **UMULL**.

## Divisão

A divisão inteira é uma adição muito mais recente ao processador ARM. Na verdade, o Raspberry Pi 1 e o Zero não possuem instrução de divisão inteira. A segunda geração do Raspberry Pi 2 usa processadores ARM Cortex-A53, que introduzem a divisão inteira no mundo Pi. O Raspberry Pi 4 inclui processadores Cortex-A72 mais recentes.

Se você estiver visando o Raspberry Pi Zero ou 1, precisará implementar seu próprio algoritmo de divisão no código, chamar algum código C ou usar o coprocessador de ponto flutuante. Abordaremos o coprocessador de ponto flutuante no Capítulo 11, “Operações de ponto flutuante”.

As instruções de divisão do Raspberry Pi 2, 3 e 4 são

- SDIV {Rd}, Rn, Rm
- UDIV {Rd}, Rn, Rm

onde

- **Rd** é o registrador de destino.
- **Rn** é o registrador que contém o numerador.
- **Rm** é um registrador contendo o denominador.

Existem alguns problemas ou notas técnicas nestas instruções:

- Não há opção “S” desta instrução, pois não definir **CPSR** em tudo.
- A divisão por 0 deve lançar uma exceção; com estas instruções, ele retorna 0, o que pode ser muito enganoso.

## Capítulo 10 Multiplicar, Dividir e Acumular

- Estas instruções não são o inverso de **SMULL** e

**UMULL.** Pois este Rn precisa ser um par de registradores, então o valor a ser dividido pode ser de 64 bits. Para dividir um arquivo de 64 bits valor, precisamos ir para o processador de ponto flutuante ou rolar nosso próprio código.

- A instrução retorna apenas o quociente, não o restante. Muitos algoritmos requerem o resto, e você deve calculá-lo como resto = numerador - (quociente \* denominador).

## Exemplo

O código para executar as instruções de divisão é simples; A Listagem 10-2 é um exemplo como fizemos para a multiplicação.

### Listagem 10-2. Exemplos das instruções SDIV e UDIV

```
@
@ Exemplos de divisão inteira de 32 bits @

.include "debug.s"

.global principal @ Forneça o endereço inicial do programa para
 vinculador

@ Carregue os registradores com alguns dados
@ Execute várias instruções de divisão
principal:
 empurre {R4-R12, LR}
 MOV R2, nº 100
 MOV R3, nº 4
```

## Capítulo 10 Multiplicar, Dividir e Acumular

```
printStr "Entradas:"
printReg 2 printReg 3
```

```
SDIV R4, R2, R3
printStr "Saídas:" printReg
4
```

```
UDIV R4, R2, R3
printStr "Saídas:" printReg
4
```

```
mov r0, #0 pop @ Código de retorno
{R4-R12, PC}
```

Se tentarmos construir isso da mesma forma que fizemos para a multiplicação exemplo, obteremos o erro

```
pi@raspberrypi:~/asm/Chapter 10 $ make -B gcc -o
divexamp divexamp.s divexamp.s: Mensagens do
Assembler: divexamp.s:21: Erro: processador selecionado
não suporta `sdiv R4,R2,R3' em Modo ARM make: *** [makefile:15: divexamp] Erro 1
pi@raspberrypi:~/asm/Chapter 10 $
```

Isso é executado em um Raspberry Pi 4. Não dissemos que ele suporta a instrução SDIV? A razão é que a fundação Raspberry Pi se esforça ao máximo para garantir que todo o seu software seja executado em todos os Raspberry Pi, independentemente da idade. A configuração padrão da GNU Compiler Collection no Raspbian é atingir o menor denominador comum. Se mudarmos o makefile para o seguinte

```
divexamp: divexamp.s debug.s gcc
-march="armv8-a" -o divexamp divexamp.s
```

## Capítulo 10 Multiplicar, Dividir e Acumular

então o programa irá compilar e rodar. O parâmetro `-march` é para a arquitetura da máquina e “`arm8-a`” é o correto para o Raspberry Pi 4. Poderíamos ter usado um para corresponder a um Raspberry Pi 3, mas queremos explorar alguns novos recursos no Pi 4 depois.

Com isso instalado, o programa é executado e obtemos os resultados esperados:

```
pi@raspberrypi:~/asm/Chapter 10 $ make gcc
-march="armv8-a" -o divexamp divexamp.s pi@raspberrypi:~
asm/Chapter 10 $./divexamp Entradas:
```

R2 = 100, 0x00000064

R3 = 4, 0x00000004

Saídas:

R4 = 25, 0x00000019

Saídas:

R4 = 25, 0x00000019

```
pi@raspberrypi:~/asm/Capítulo 10 $
```

# Multiplicar e Acumular

A operação de multiplicar e acumular multiplica dois números e os adiciona a um terceiro. À medida que avançamos nos próximos capítulos, veremos essa operação reaparecer repetidas vezes. O processador ARM é RISC, se o conjunto de instruções é reduzido, então por que encontramos tantas instruções e, portanto, tantos circuitos, dedicados a realizar multiplicação e acumulação. A resposta remonta ao nosso curso de matemática favorito do primeiro ano da universidade sobre álgebra linear. A maioria dos estudantes de ciências é forçada a fazer este curso, aprender a trabalhar com vetores e matrizes e esperar nunca mais ver esses conceitos. Infelizmente, eles formam a base para gráficos e aprendizado de máquina. Antes de nos aprofundarmos nas instruções ARM para multiplicar e acumular, vamos rever um pouco de álgebra linear.

## Capítulo 10 Multiplicar, Dividir e Acumular

# Vetores e Matrizes

Um vetor é uma lista ordenada de números. Por exemplo, em gráficos 3D, pode representar sua localização no espaço 3D onde [x, y, z] são suas coordenadas. Os vetores têm uma dimensão que é o número de elementos

Eles contêm. Acontece que uma computação útil com vetores é algo chamado produto escalar. Se  $A = [a_1, a_2, \dots, a_n]$  é um vetor e  $B = [b_1, b_2, \dots, b_n]$  é outro vetor, então o produto escalar é definido como

$$A \cdot B = a_1 * b_1 + a_2 * b_2 + \dots + a_n * b_n$$

Se quisermos calcular esse produto escalar, um loop executando instruções de multiplicação e acumulação deve ser bastante eficiente. Uma matriz é uma tabela 2D de números, como

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}$$

A multiplicação de matrizes é um processo complicado que enlouquece os alunos de álgebra linear do primeiro ano. Quando você multiplica a matriz A vezes a matriz B, cada elemento da matriz resultante é o produto escalar de uma linha da matriz A com uma coluna da matriz B.

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \begin{vmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{vmatrix} = \begin{vmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{vmatrix}$$

Se fossem matrizes 3x3, haveria nove produtos escalares cada com nove termos. Também podemos multiplicar uma matriz por um vetor da mesma maneira.

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \begin{vmatrix} b_1 \\ b_2 \end{vmatrix} = \begin{vmatrix} a_{11}b_1 + a_{12}b_2 \\ a_{21}b_1 + a_{22}b_2 \end{vmatrix}$$

## Capítulo 10 Multiplicar, Dividir e Acumular

Em gráficos 3D, se representarmos um ponto como um vetor 4D [x, y, z, 1], então as transformações afins de escala, rotação, cisalhamento e reflexão podem ser representadas como matrizes 4x4. Qualquer número dessas transformações pode ser combinado em uma única matriz. Assim, transformar um objeto em cena requer uma multiplicação de matrizes aplicada a cada um dos vértices do objeto. Quanto mais rápido pudermos fazer isso, mais rápido poderemos renderizar um quadro em um videogame.

Nas redes neurais, o cálculo para cada camada de neurônios é calculado por uma multiplicação da matriz, seguida da aplicação de uma função não linear. A maior parte do trabalho é a multiplicação da matriz. A maioria das redes neurais tem muitas camadas de neurônios, cada uma exigindo uma multiplicação de matriz. O tamanho da matriz corresponde ao número de variáveis e ao número de neurônios; portanto, as dimensões da matriz geralmente estão na casa dos milhares. A rapidez com que realizamos o reconhecimento de objetos ou a tradução da fala depende da rapidez com que podemos multiplicar as matrizes, ou seja, da rapidez com que podemos multiplicar com acumular.

Essas aplicações importantes são o motivo pelo qual o processador ARM dedica tanto silício para multiplicar e acumular. Voltaremos a como acelerar esse processo enquanto exploramos os coprocessadores FPU e NEON do Raspberry Pi nos capítulos seguintes.

## Acumular Instruções

Como vimos com a multiplicação, houve uma grande proliferação de instruções de multiplicação com acumulação. Felizmente, cobrimos a maioria dos detalhes na seção “Multiplicação”. Aqui estão eles:

- MLA{S} Rd, Rn, Rm, Ra
- SMLAL{S} RdLo, RdHi, Rn, Rm
- SMLA<x><y> Rd, Rn, Rm, Ra
- SMLAD{X} Rd, Rn, Rm, Ra

## Capítulo 10 Multiplicar, Dividir e Acumular

- PEQUENO{X} RdLo, RdHi, Rn, Rm
- SMLAL<x><y> RdLo, RdHi, Rn, Rm
- SMLAW<y> Rd, Rn, Rm, Ra
- SMLSD{X} Rd, Rn, Rm, Ra
- SMLSD{X} RdLo, RdHi, Rn, Rm
- SMMLA{R} Rd, Rn, Rm, Ra
- SMMLS{R} Rd, Rn, Rm, Ra
- SMUAD{X} {Rd}, Rn, Rm
- UMAAL RdLo, RdHi, Rn, Rm
- UMLAL{S} RdLo, RdHi, Rn, Rm

São muitas instruções, portanto não abordaremos cada uma delas em detalhes, mas podemos reconhecer que há uma multiplicação com acumulação para cada instrução de multiplicação regular. Vejamos o que leva a uma maior proliferação de instruções.

Se houver um operando Ra, então o cálculo é

$$Rd = Rn * Rm + Ra$$


---

**Nota** **Rd** pode ser o mesmo que **Ra** para calcular uma soma corrente.

---

Se não houver um operando Ra, então o cálculo é

$$Rd = Rd + Rn * Rm$$

Esta segunda forma tende a ser para instruções com resultados de 64 bits, então o soma precisa ser de 64 bits, portanto, não pode ser um único registrador.

## Capítulo 10 Multiplicar, Dividir e Acumular

**Multiplicação dupla com acumulação**

As instruções que terminam em **D** são duais. Eles fazem dois multiplicam e acumulam em uma única etapa. Eles multiplicam os 16 bits superiores de Rn e Rm e multiplicam os 16 bits inferiores de Rn e Rm e, em seguida, adicionam os dois produtos ao acumulador.

Se houver um **S** na instrução em vez de um **A**, isso significa que subtrai os dois valores antes de adicionar o resultado ao acumulador.

$$Rd = Ra + (Rn \text{ inferior} * Rm \text{ inferior} - Rn \text{ superior} * Rm \text{ superior})$$

Se a precisão funcionar para você e você puder codificar todos os dados dessa maneira, poderá dobrar sua taxa de transferência usando essas instruções. Veremos isso no Exemplo 2.

**Exemplo 1**

Já falamos sobre como multiplicar e acumular é ideal para multiplicar matrizes, então, por exemplo, vamos multiplicar duas matrizes 3x3.

O algoritmo que estamos implementando é mostrado na Listagem 10-3.

**Listagem 10-3.** Pseudo-código para nosso programa de multiplicação de matrizes

PARA linha = 1 a 3

    PARA col = 1 a 3

        acum = 0

        FOR i = 1 a 3 acum

            = acum + A[linha, i]\*B[i, col]

        PROXIMO EU

        C[linha, coluna] = acum

    PRÓXIMA coluna

PRÓXIMA linha

## Capítulo 10 Multiplicar, Dividir e Acumular

Basicamente, os loops de linha e coluna passam por cada célula da matriz de saída e calculam o produto escalar correto para aquela célula no loop mais interno.

A Listagem 10-4 mostra nossa implementação em Assembly.

### Listagem 10-4. Multiplicação de matrizes 3x3 em Assembly

```

@

@ Multiplique 2 matrizes inteiros 3x3 @

@ Registra: @ @ @

@ @ @ @ R4 @Índice de linha

 R2 - Índice de coluna

 R4 - Endereço da linha

 R5 - Endereço da coluna

 R7 - soma acumulada de 64 bits

 R8 - soma acumulada de 64 bits

 R9 - Célula de A

 R10 - Célula de B

 R11 - Posição em C

 R12 - linha em dotloop

 R6 - col in dotloop

.global main @ Forneça o endereço inicial do programa

 .equ N, 3 @ Dimensões da matriz .equ

 WDSIZE, 4 @ Tamanho do elemento

principal:

 empurre {R4-R12, LR} @ Salve os regs necessários

 MOV R1, #N @ Índice de linha

 LDR R4, =A @ Endereço da linha atual

 LDR R11, =C @ Endereço da matriz de resultados

```

## Capítulo 10 Multiplicar, Dividir e Acumular

loop de linha:

|            |                                       |
|------------|---------------------------------------|
| LDR R5, =B | @ primeira coluna em B                |
| MOV R2, #N | @ Colindex (fará contagem regressiva) |

colloop:

|                                    |                                      |
|------------------------------------|--------------------------------------|
| @ Registradores de acumulador zero |                                      |
| MOV R7, nº 0                       |                                      |
| MOV R8, nº 0                       |                                      |
| MOV R0, #N                         | @ contador de loop de produto        |
| MOV R12, R4                        | escalar @ linha para produto escalar |
| MOV R6, R5                         | @ coluna para produto escalar        |

dotloop:

|                                                              |                                   |
|--------------------------------------------------------------|-----------------------------------|
| @ Faça o produto escalar de uma linha de A com a coluna de B |                                   |
| LDR R9, [R12], #WDSIZE                                       | @ load A[row, i] e incr           |
| LDR R10, [R6], #(N*WDSIZE)                                   | @ load B[i, col]                  |
| SMLAL R7, R8, R9, R10                                        | @ Multiplique e acumule           |
| SUBS R0, #1                                                  | @ Contador de loop Dec            |
| BNE dotloop                                                  | @ Se não for loop zero            |
| STR R7, [R11], #4                                            | @ C[linha, coluna] = pontoprod    |
| ADICIONE R5, #WDSIZE                                         | @ Incrementar col atual           |
| SUBS R2, nº 1                                                | @ Contador de loop de dezembro    |
| colloop BNE                                                  | @ Se não for loop zero            |
| ADD R4, #(N*WDSIZE)                                          | @ Incremento para a próxima linha |
| SUBS R1, nº 1                                                | @ Dec contador de loop de linha   |
| Circuito BNE                                                 | @ Se não for loop zero            |

@ Imprima a matriz C @

Percorra 3 linhas imprimindo 3 colunas de cada vez.

MOV R5, #3 @ Imprimir 3 linhas LDR R11, =C @

Addr of results matrix printloop:

## Capítulo 10 Multiplicar, Dividir e Acumular

```

LDR R0, =prtstr @ string de formato printf
LDR R1, [R11], #WDSIZE @ primeiro elemento na linha atual
LDR R2, [R11], #WDSIZE @ segundo elemento na linha atual
LDR R3, [R11], #WDSIZE @ terceiro elemento na linha atual
BL printf @ Chama printf
SUBS R5, nº 1 @ Contador de loop Dec
BNE printloop @ Se não for zero loop

 mov r0, #0 pop @ Código de retorno
 {R4-R12, PC} @ Restaurar regs e retornar

.dados
@ Primeira matriz
A: .palavra 1, 2,
 3 .palavra 4, 5,
 6 .palavra 7, 8, 9
@ Segunda matriz
B: .palavra 9, 8,
 7 .palavra 6, 5,
 4 .palavra 3, 2, 1
@ Matriz de resultados
C: .preencher 9, 4, 0

prtstr: .asciz "%3d %3d %3d\n"

```

Compilando e executando este programa, obtemos

```

pi@raspberrypi:~/asm/Capítulo 10 $ make gcc -o
matrixmult matrixmult.s pi@raspberrypi:~/asm/Capítulo
10 $./matrixmult
30 24 18 84 69
54 138 114 90
pi@raspberrypi:~/asm/Capítulo 10 $

```

## Capítulo 10 Multiplicar, Dividir e Acumular

## Acessando Elementos da Matriz

Armazenamos as três matrizes na memória, em ordem de linha. Eles estão organizados nas diretivas .word para que você possa ver a estrutura da matriz. No pseudocódigo, nos referimos aos elementos da matriz usando matrizes 2D. Não há instruções ou formatos de operando para especificar o acesso ao array 2D, então devemos fazer isso sozinhos. Para Assembly, cada array é apenas uma sequência de nove palavras da memória. Agora que sabemos como multiplicar, podemos fazer algo como

$$A[i, j] = A[i*N + j]$$

onde N é a dimensão da matriz. Nós não fazemos isso; em Assembly, vale a pena observar que acessamos os elementos do array em ordem e podemos ir de um elemento em linha para o próximo adicionando o tamanho de um elemento - o tamanho de uma palavra ou quatro. Podemos passar de um elemento em uma coluna para o próximo adicionando o tamanho de uma linha. Portanto, usamos a constante  $N * WDSIZE$  com tanta frequência no código. Dessa forma, percorremos o array de forma incremental e nunca precisamos multiplicar os índices do array. Geralmente, a multiplicação e a divisão são operações caras e devemos tentar evitá-las o máximo possível.

Podemos usar técnicas de pós-indexação para acessar ponteiros de incremento de elementos para o próximo elemento. Usamos a pós-indexação para armazenar o resultado de cada cálculo na matriz C. Vemos isso em

```
STR R7, [R11], #4
```

que armazena nosso produto escalar calculado em C, então incrementa o ponteiro em C em 4 bytes. Vemos isso novamente quando imprimimos a matriz C no final.

## Capítulo 10 Multiplicar, Dividir e Acumular

### Multiplicar com Acumular

O núcleo do algoritmo depende da instrução **SMLAL** para multiplicar um elemento de A por um elemento de B e adicioná-lo à soma corrente do produto escalar.

**SMLAL R7, R8, R9, R10**

Esta instrução acumula uma soma de 64 bits, mas tomamos apenas o menor 32 bits em **R7**. Não verificamos estouro; se no final **R8** não for 0, daremos um resultado incorreto.

### Registrar Uso

Usamos quase todos os registradores; temos sorte de poder acompanhar todos os nossos índices de loop e ponteiros em registradores e não precisar movê-los para dentro e para fora da memória. Se precisássemos fazer isso, teríamos alocado espaço na pilha para armazenar quaisquer variáveis necessárias.

### Exemplo 2

Quando discutimos as instruções de multiplicação com acumulação, mencionamos as instruções duplas que executarão duas etapas em uma instrução.

O principal problema é empacotar dois números que precisam ser processados em cada registrador de 32 bits. Podemos criar inteiros de 16 bits facilmente usando a diretiva `.short` Assembler. Processar as linhas é fácil, pois as células estão próximas umas das outras, mas para as colunas, cada elemento está a uma linha de distância. Como podemos carregar facilmente dois elementos de coluna em um registrador de 32 bits?

O que podemos fazer é pegar a transposta da segunda matriz. Isso significa fazer as linhas colunas e as colunas linhas, basicamente trocando  $B[i, j]$  por  $B[j, i]$ . Se fizermos isso, os elementos da coluna estarão próximos uns dos outros e fáceis de carregar em um único registrador de 32 bits.

A Listagem 10-5 é o código para fazer isso.

## Capítulo 10 Multiplicar, Dividir e Acumular

**Listagem 10-5.** Multiplicação de matrizes 3x3 usando uma dupla multiplicação/acumulação

@

@ Multiplique 2 matrizes inteiras 3x3 @ Usa  
uma instrução dual de multiplicação/acumulação @ para  
processar dois elementos no produto escalar @ por loop. @

@ Registra: @ @ @

@ @ @ @ @R4 @Índice de linha

R2 - Índice de coluna

R4 - Endereço da linha

R5 - Endereço da coluna

R7 - soma acumulada de 64 bits

R8 - soma acumulada de 64 bits

R9 - Célula de A

R10 - Célula de B

R11 - Posição em C

R12 - linha em dotloop

R6 - col in dotloop

.global main @ Fornece o endereço inicial do programa ao vinculador .equ N, 3  
@ Dimensões da matriz .equ ELSIZE, 2 @ Tamanho do elemento

principal:

push {R4-R12, LR} @ Salve os regs necessários

MOV R1, #N @ Índice de linha

LDR R4, =A @ Endereço da linha atual

LDR R11, =C loop @ Endereço da matriz de resultados

de linha:

LDR R5, =B @ primeira coluna em B

MOV R2, #N @ Índice da coluna (contará até 0)

## Capítulo 10 Multiplicar, Dividir e Acumular

coloop:

```

@ Registradores de acumulador zero
MOV R7, nº 0
MOV R8, nº 0

MOV R0, #((N+1)/2) @ contador de loop de produto escalar
MOV R12, R4 @ linha para produto escalar
MOV R6, R5 @ coluna para produto escalar

```

dotloop:

```

@ Faça o produto escalar de uma linha de A com a coluna de B
LDR R9, [R12], #(ELSIZE*2) @ load A[row, i] and incr
LDR R10, [R6], #(ELSIZE*2) @ load B[i, col]
SMLAD R7, R9, R10, R7 @ Faça dupla multiplicação e acumulação
SUBS R0, #1 @ Contador de loop Dec
BNE dotloop @ Se não for loop zero

STR R7, [R11], #4 @ C[linha, coluna] = pontoprod
ADD R5, #((N+1)*ELSIZE) @ Incrementar col atual
SUBS R2, nº 1 @ Contador de loop de dezembro
coloop BNE @ Se não for loop zero

ADD R4, #((N+1)*ELSIZE) @ Incrementa para a próxima linha
SUBS R1, nº 1 @ Dec contador de loop de linha
Circuito BNE @ Se não for loop zero

```

@ Imprima a matriz C @

Percorra 3 linhas imprimindo 3 colunas de cada vez.

MOV R5, #3 @ Imprimir 3 linhas LDR R11, =C @

Addr of results matrix printloop:

```

LDR R0, =prtstr @ string de formato printf
LDR R1, [R11], #4 @ primeiro elemento na linha atual
LDR R2, [R11], #4 @ segundo elemento na linha atual

```

## Capítulo 10 Multiplicar, Dividir e Acumular

```

LDR R3, [R11], #4 @ terceiro elemento na linha atual
BL printf @ Chama printf
SUBS R5, no 1 @ Contador de loop Dec
BNE printloop @ Se não for zero loop mov r0,
#0 @ código de retorno pop {R4-R12, PC}
 @ Restaurar regs e retornar

```

.dados

@ Primeira matriz

|    |        |               |
|----|--------|---------------|
| A: | .curto | 1, 2, 3, 0 4, |
|    | .curto | 5, 6, 0 7, 8, |
|    | .curto | 9, 0          |

@ Segunda matriz

|    |        |               |
|----|--------|---------------|
| B: | .curto | 9, 6, 3, 0 8, |
|    | .curto | 5, 2, 0 7, 4, |
|    | .curto | 1, 0          |

@ Matriz de resultados

|    |            |         |
|----|------------|---------|
| C: | .preencher | 9, 4, 0 |
|----|------------|---------|

prtstr: .asciz "%3d %3d %3d\n"

A economia nas instruções está na redução do loop interno que calcula o produto escalar.

MOV R0, #((N+1)/2) @ contador de loop de produto escalar

Se nossa matriz tivesse uma dimensão uniforme, teríamos economizado mais. Para nosso exemplo 3x3, o loop de produto escalar ainda tem dois elementos. Mas então, se estivéssemos fazendo duas matrizes 4x4, também seriam duas vezes neste loop. Observe que tivemos que adicionar um 0 ao final de cada linha de ambas as matrizes, pois a instrução dual processará um número par de entradas.

O verdadeiro burro de carga deste programa é

SMLAD R7, R9, R10, R7

## Capítulo 10 Multiplicar, Dividir e Acumular

que multiplica a parte alta de **R9** pela parte alta de **R10** e ao mesmo tempo a parte baixa de **R9** pela parte baixa de **R10**, então adiciona ambos a **R7** e coloca a nova soma em **R7**. Observe que não há problema em ter **Rd=Ra**, que é o que você mais deseja.

Ainda usamos **o LDR** para carregar os registradores das matrizes. Isso carregará 32 bits; como especificamos cada elemento para ocupar 16 bits, ele carregará dois de cada vez, melhorando nosso desempenho.

# Resumo

Cobrimos as várias formas da instrução de multiplicação suportadas no Conjunto de instruções ARM de 32 bits. Cobrimos as instruções de divisão incluídas em versões mais recentes dos processadores ARM, como os do Raspberry Pi 3 e 4. Para processadores mais antigos, podemos usar o FPU, escrever nossa própria rotina ou chamar algum código C.

Em seguida, abordamos o conceito de multiplicar e acumular e por que essas instruções são tão importantes para aplicativos modernos em gráficos e aprendizado de máquina. Revisamos as muitas variações dessas instruções e, em seguida, apresentamos duas versões de multiplicação de matrizes para mostrá-las em ação.

No Capítulo 11, “Operações de ponto flutuante”, veremos mais matemática, mas desta vez em notação científica, permitindo frações e expoentes, indo além dos números inteiros pela primeira vez.

## CAPÍTULO 11

# Ponto flutuante Operações

O Raspberry Pi é baseado em um sistema em um chip. Este chip contém a CPU ARM quad-core que estudamos junto com alguns coprocessadores. Neste capítulo, veremos o que a unidade de ponto flutuante (FPU) faz. Algumas documentações do ARM referem-se a isso como Vector Floating-Point (VFP) para promover o fato de que ele pode fazer algum processamento vetorial limitado. Qualquer processamento vetorial na FPU agora é substituído pelo processamento paralelo muito melhor fornecido pelo coprocessador NEON, que estudamos no Capítulo 12, “Coprocessador NEON”. Independentemente disso, o FPU fornece várias instruções úteis para realizar matemática de ponto flutuante.

Revisaremos o que são números de ponto flutuante, como eles são representados na memória e como inseri-los em nossos programas Assembly. Veremos como transferir dados entre a FPU e os registradores regulares e a memória do ARM. Veremos como realizar operações aritméticas básicas, comparações e conversões.

## Capítulo 11 Operações de ponto flutuante

## Sobre números de ponto flutuante

Números de ponto flutuante são uma forma de representar números em notação científica no computador. A notação científica representa números mais ou menos assim:

1,456354 x 10<sup>16</sup>

Existe uma parte fracionária e um expoente que permite mover o decimal coloque à esquerda se for positivo e à direita se for negativo. O Raspberry Pi lida com números de ponto flutuante de precisão única com tamanho de 32 bits e números de ponto flutuante de precisão dupla com tamanho de 64 bits.

O Raspberry Pi usa o padrão IEEE 754 para números de ponto flutuante. Cada número contém um bit de sinal para indicar se é positivo ou negativo, um campo de bits para o expoente e uma sequência de dígitos para a parte fracionária. A Tabela 11-1 lista o número de bits para as partes de cada formato.

**Tabela 11-1.** Bits de um número de ponto flutuante

| Precisão do nome | Sinal   | Dígitos decimais do expoente fracionário |    |    |    |
|------------------|---------|------------------------------------------|----|----|----|
| Solteiro         | 32 bits | 1                                        | 24 | 8  | 7  |
| Duplo            | 64 bits | 1                                        | 53 | 11 | 16 |

A coluna de dígitos decimais da Tabela 11-1 é o número aproximado de dígitos decimais que o formato pode representar ou a precisão decimal.

## Normalização e NaNs

Nos números inteiros que vimos até agora, todas as combinações dos bits fornecem um número único válido. Dois padrões diferentes de bits não produzem o mesmo número; no entanto, este não é o caso em ponto flutuante. Em primeiro lugar, temos o conceito de não ser um número ou **NaN**. NaNs são produzidos a partir de

## Capítulo 11 Operações de ponto flutuante

operações como dividir por zero ou tirar a raiz quadrada de um número negativo. Isso permite que o erro se propague silenciosamente através do cálculo sem travar um programa. Na especificação IEEE 754, um NaN é representado por um expoente de todos os bits 1.

Um número de ponto flutuante normalizado significa que o primeiro dígito na parte fracionária é diferente de zero. Um problema com números de ponto flutuante é que os números geralmente podem ser representados de várias maneiras. Por exemplo, uma parte fracionária de 0 com bit de sinal ou qualquer expoente é zero. Considere uma representação de 1:

$$1E0 = 0,1E1 = 0,01E2 = 0,001E3$$

Todos eles representam 1, mas chamamos o primeiro sem zeros à esquerda de forma normalizada. O ARM FPU tenta manter os números de ponto flutuante na forma normal, mas vai quebrar essa regra para números pequenos, onde o expoente já é o mais negativo possível, então para tentar evitar erros de underflow, o FPU desistirá da normalização para representar números um pouco menores do que poderia.

## Erros de arredondamento

Se pegarmos um número como  $\frac{1}{3} = 0.33333\dots$  e representá-lo em ponto flutuante, então mantemos apenas 7 ou mais dígitos para precisão única. Isso introduz erros de arredondamento. Se isso for um problema, geralmente a precisão dupla resolve os problemas, mas alguns cálculos são propensos a aumentar os erros de arredondamento, como a subtração de dois números que têm uma diferença mínima.

---

**Nota** Números de ponto flutuante são representados na base 2, portanto, as expansões decimais que levam a padrões repetidos de dígitos são diferentes daquelas da base 10. Muitas pessoas se surpreendem ao saber que 0,1 é uma fração binária repetida: 0,00011001100110011..., significando que adicionar dólares e centavos em ponto flutuante introduzirá erros de arredondamento em cálculos suficientes.

---

## Capítulo 11 Operações de ponto flutuante

Para cálculos financeiros, a maioria dos aplicativos usa aritmética de ponto fixo que é construída sobre aritmética inteira para evitar erros de arredondamento na adição e subtração.

## Definindo Números de Ponto Flutuante

O GNU Assembler tem diretivas para definir armazenamento para números de ponto flutuante de precisão simples e dupla. Estes são .single e .double, por exemplo:

```
.único 1.343, 4.343e20, -0.4343, -0.4444e-10 .duplo
-4.24322322332e-10, 3.141592653589793
```

Essas diretivas sempre usam números de base 10.

## Registros FPU

O ARM FPU tem seu próprio conjunto de registradores. Existem 32 registradores de ponto flutuante de precisão única que são referidos como S0, S1, ..., S31. Esses registradores também podem ser referidos como 16 registradores de dupla precisão D0, ..., D15. A Figura 11-1 mostra essa configuração de registradores.

S0-S31      D0-D15

|     |     |
|-----|-----|
| S0  | D0  |
| S1  |     |
| S2  | D1  |
| S3  |     |
| S4  | D2  |
| S5  |     |
| S6  | D3  |
| S7  |     |
| ... | ... |
| S28 | D14 |
| S29 |     |
| S30 | D15 |
| S31 |     |

**Figura 11-1.** Os registros FPU do ARM (os registros de precisão simples à esquerda se sobrepõem aos registros de precisão dupla à direita)

Nota Os registradores **S0** e **S1** ocupam o mesmo espaço que **D0**. Os registradores **S2** e **S3** ocupam o mesmo espaço que **D1** e assim por diante. A FPU fornece apenas uma sintaxe mais fácil para fazer operações de precisão simples ou precisão dupla. Cabe a nós manter as coisas em ordem e não corromper nossos registros acessando o mesmo espaço incorretamente.

Os Raspberry Pi 2, 3 e 4 possuem 16 registradores adicionais de precisão dupla D16–D31 que não possuem contrapartes de precisão simples.

Esses são um subconjunto dos registradores disponíveis para o processador NEON, que abordaremos no próximo capítulo. Por enquanto, apenas um aviso de que pode haver um conflito com o processador NEON se o estivermos usando também.

## Capítulo 11 Operações de ponto flutuante

## Protocolo de chamada de função

No Capítulo 6, “Funções e a pilha”, fornecemos o protocolo para quem salva quais registradores ao chamar funções. Com esses registradores de ponto flutuante, temos que adicioná-los ao nosso protocolo:

- **Chamado salvo:** A função é responsável por salvar os registradores S16–S31 (**D8–D15**) necessários para serem salvos por uma função se a função os utilizar.
  - **Chamador salvo:** Todos os outros registros não precisam ser salvos por uma função, portanto, devem ser salvos pelo chamador caso seja necessário mantê-los. Isso inclui **S0–S15** (D0–D7) e D16–D31 se estiverem presentes. Isso também se aplica a quaisquer registros adicionais para o NEON coprocessador.
- 

Nota O double também é nosso primeiro tipo de dados de 64 bits. Existe uma regra adicional sobre como colocá-los em registradores, ou seja, ao passar um item de 64 bits, ele pode ir nos registradores **R0** e **R1** ou **R2** e **R3**. Não pode ser colocado em **R1** e **R2**. E não pode estar metade em **R3** e metade na pilha. Veremos isso mais tarde chamando printf com um double como parâmetro.

---

Aqui estão as instruções do nosso primeiro coprocessador:

- **VPUSH** {lista de registros}
- **VPOP** {reglist}

Por exemplo:

```
VPUSH {S16-S31}
VPOP {S16-S31}
```

Você só tem permissão para uma lista nestas instruções que você pode criar com registradores **S** ou **D**.

---

Nota A lista não pode ter mais de 16 registros **D**.

---

## Sobre a construção

Todos os exemplos neste capítulo usam o tempo de execução C e são construídos usando **gcc**. Isso funciona bem da mesma maneira que os capítulos anteriores. Se quisermos usar o GNU Assembler diretamente por meio do comando **as** , precisamos modificar nosso makefile com

`%o: %.s`

```
as -mfpu=vfp $(DEBUGFLGS) $(LSTFLGS) $< -o $@
```

Aqui especificamos que temos uma FPU. Isso nos dará o vfpv2, que funciona para todos os Raspberry Pi. Poderíamos usar vfpv3 ou vfpv4 para Pi mais recente se precisarmos de um recurso mais recente. Todos os exemplos de ponto flutuante neste livro funcionam para qualquer Pi e podem usar apenas a versão genérica da linha de comando parâmetro.

## Carregando e salvando registros FPU

No Capítulo 5, “Obrigado pelas memórias”, abordamos as instruções **LDR** e **STR** para carregar registradores da memória e depois armazená-los de volta na memória. O coprocessador de ponto flutuante possui instruções semelhantes para seus registradores:

- VLDR                       $F_d, [R_n\{, \#offset\}]$
- VSTR                       $F_d, [R_n\{, \#offset\}]$

## Capítulo 11 Operações de ponto flutuante

Vemos que ambas as instruções suportam deslocamentos de endereçamento pré-indexados.

O registrador **Fd** pode ser um registrador **S** ou **D**. Por exemplo:

```
LDR R1, =fp1
VLDR S4, [R1]
VLDR S4, [R1, nº 4]
VSTR S4, [R1]
VSTR S4, [R1, nº 4]
...
```

.dados

|              |         |
|--------------|---------|
| fp1: .single | 3.14159 |
| fp2: .single | 4.3341  |
| fp3: .single | 0.0     |

Há também uma instrução load multiple e store multiple—essas são

- VLDM Rn{!}, Registradores
- VSTM Rn{!}, Registradores

Os registradores são uma variedade de registradores, como para as instruções **VPUSH** e **VPOP**. Apenas uma faixa é permitida, podendo ter no máximo 16 registradores duplos. Estes carregarão o número do endereço apontado por Rn, e o número de registradores e se são simples ou duplos determinará quantos dados serão carregados. O opcional! atualizará o ponteiro em **Rn** após a operação, se presente.

## Aritmética Básica

O processador de ponto flutuante inclui as quatro operações aritméticas básicas, junto com algumas extensões como nossa multiplicação e acumulação favoritas. Existem algumas funções especializadas, como raiz quadrada, e algumas variações que afetam o sinal — versões negativas de funções.

## Capítulo 11 Operações de ponto flutuante

Cada uma dessas funções vem em duas versões, uma versão de 32 bits que você coloca .F32 depois e uma versão de 64 bits que você coloca .F64 depois. Seria bom se o Assembler apenas fizesse isso para você com base nos registradores que você forneceu, mas se você deixar de fora a parte do tamanho, a mensagem de erro é enganosa. Aqui está uma seleção das instruções:

- VADD.F32 {Sd}, Sn, Sm
- VADD.F64 {Dd}, Dn, Dm
- VSUB.F32 {Sd}, Sn, Sm
- VSUB.F64 {Dd}, Dn, Dm
- VMUL.F32 {Sd}, Sn, Sm
- VMUL.F64 {Dd}, Dn, Dm
- VDIV.F32 {Sd}, Sn, Sm
- VDIV.F64 {Dd}, Dn, Dm
- VMLA.F32 Sd, Sn, Sm
- VMLA.F64 Dd, Dn, Dm
- VSQRT.F32 Sd, Sm
- VSQRT.F64 Dd, Dm

Se o registrador de destino estiver entre colchetes {}, ele é opcional, portanto, podemos omiti-lo. Isso significa que aplicamos o segundo operando ao primeiro, portanto, para adicionar **S1** a **S4**, simplesmente escrevemos

VADD.F32 S4, S1

Essas funções são bastante simples, então vamos passar para um exemplo.

## Capítulo 11 Operações de ponto flutuante

# Distância Entre Pontos

Se tivermos dois pontos  $(x_1, y_1)$  e  $(x_2, y_2)$ , então a distância entre eles é dada pela fórmula

$$d = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$$

Vamos escrever uma função para calcular isso para quaisquer dois pares de coordenadas de ponto flutuante de precisão simples. Usaremos a função printf do tempo de execução C para imprimir nossos resultados. Primeiro, a função de distância da Listagem 11-1, no arquivo distance.s.

## Listagem 11-1. Função para calcular a distância entre dois pontos

```
@ @ Exemplo de função para calcular a distância @ entre dois pontos
em precisão simples @ ponto flutuante. @ @ Entradas: R0 - ponteiro
para os 4 números FP @ eles são x1, y1, x2, y2 @ @ Saídas: R0 - o
comprimento (como FP de precisão simples) @
```

```
.distância global @ Permitir que a função seja chamada por outros
```

```
@ distância:
```

```
@ empurre todos os registradores para ser seguro, nós @
realmente não precisamos empurrar tantos. push {R4-R12, LR}
vpush {S16-S31}
```

```
@ carregar todos os 4 números de uma vez
vldm R0, {S0-S3}
```

```

@ calc s4 = x2 - x1
vsub.f32 S4, S2, S0 @ calc
s5 = y2 - y1 vsub.f32 S5, S3,
S1 @ calc s4 = S4 - S4 (x2-
X1)^2 vmul.f32 S4, S4 @ calc s5 = s5 - Y
s5 (Y2-Y1)^2 vmul.f32 S5, S5 @ calc
S4 = S4 + S5 vadd.f32 S4, S5 @ calc
sqrt(S4) vsqrt.f32 S4, S4 @ move o
resultado para R0 a ser devolvido vmov
R0, S4

```

```

@ restaurar o que preservamos. vpop
{S16-S31} pop {R4-R12, PC}

```

Agora colocamos o código da Listagem 11-2 em main.s, que chama a distância três vezes com três pontos diferentes e imprime a distância para cada um.

### Listagem 11-2. Programa principal para chamar a função de distância três vezes

```

@
@ Programa principal para testar nossa função de distância
@
@ r7 - contador de loop
@ r8 - endereço para o conjunto atual de pontos
.global main @ Fornece o ponto de entrada do programa

```

## Capítulo 11 Operações de ponto flutuante

@

.equ N, 3 @ Número de pontos.

principal:

empurre {R4-R12, LR}

ldr r8, =points @ ponteiro para os pontos atuais

mov r7, #N @ número de iterações do loop

loop: mov r0, r8 @ move o ponteiro para o parâmetro 1

distância bl @ função de distância de chamada

@ precisa pegar o valor de retorno de precisão simples @ e convertê-lo em duplo, porque a função C printf @ só pode imprimir duplos.

vcvt.f64.f32 d0, s2 @ converte@single para duplo para conversão d0@ e imprumar duplo para r2, r3 ldr r0, =prtstr @ carregar string de impressão bl printf @ imprimir a distancia

add r8, #(4\*4) subs @ 4 pontos cada 4 bytes @  
r7, #1 loop bne contador de loop de decremento  
@ loop se mais pontosmov r0, #0 pop @ Código de retorno  
{R4-R12, PC}

.dados

Pontos: .single 0.0, 0.0, 3.0, 4.0 .single 1.3, 5.4, 3.1,  
-1.5 .single 1.3231e-10, -1.2e-4, 34.55, 5454.234

.asciz "Distância = %f\n"

O makefile está na Listagem 11-3.

### Listagem 11-3. Makefile para o programa a distância

distância: distance.s main.s

```
gcc -o distância distance.s main.s
```

Se construirmos e executarmos o programa, obtemos

```
pi@raspberrypi:~/asm/Chapter 11 $ make gcc -g -o
distance distance.s main.s pi@raspberrypi:~/asm/Chapter
11 $./distance Distance = 5.000000 Distance = 7.130919 Distance
= 13230000128.000000 pi@raspberrypi :~/asm/Capítulo 11 $
```

Construímos os dados, então o primeiro conjunto de pontos compreende um triângulo 3-4-5, e é por isso que obtemos a resposta exata de 5 para a primeira distância.

A função de distância é direta. Ele carrega todos os quatro números em uma instrução **VLDM** e então chama as várias funções aritméticas de ponto flutuante para realizar o cálculo. Realmente não precisamos salvar nenhum registrador, mas inclui as instruções **VPUSH** e **VPOP** como exemplo.

A parte da rotina principal que faz um loop e chama a rotina de distância é direta. A parte que chama printf tem algumas novas complexidades.

O problema é que a rotina C printf só tem suporte para imprimir duplas.

Em C, isso não é um grande problema, pois você pode simplesmente lançar o argumento para forçar uma conversão. Em Assembly, precisamos converter nossa soma de precisão simples em um número de precisão dupla, para que possamos imprimi-lo.

Para fazer a conversão, nós **VMOV** a soma de volta para o FPU. **VMOV** é uma instrução útil para mover valores entre registradores FPU e entre registradores FPU e CPU. Usamos a estranha instrução **VCVT.F64.F32** para converter de precisão simples para dupla. Esta função é o tema de

## Capítulo 11 Operações de ponto flutuante

a próxima seção. Em seguida, **VMOV** o duplo recém-construído de volta aos registradores **R2** e **R3**.

Quando chamamos `printf`, o primeiro parâmetro vai em **R0**. Atingimos então a regra sobre ter que colocar o próximo parâmetro de 64 bits em **R2** e **R3**.

**Nota** Se você estiver depurando o programa com **gdb** e quiser ver o conteúdo dos registradores FPU em qualquer ponto, use o comando “**info all-registers**” que listará exaustivamente todos os registradores do coprocessador. Não veremos alguns deles até o próximo capítulo, quando cobrirmos o coprocessador NEON.

## Conversões de Ponto Flutuante

No último exemplo, vimos pela primeira vez a instrução de conversão **VCVT**. A FPU suporta uma variedade de versões desta função; ele não apenas oferece suporte a conversões entre números de ponto flutuante de precisão simples e dupla, mas também oferece suporte a conversões de e para números inteiros. Ele também oferece suporte à conversão para números decimais de ponto fixo (inteiros com um decimal implícito). Ele também suporta vários métodos de arredondamento. As versões mais usadas desta função são

- **VCVT.F64.F32** Dd, Sm
- **VCVT.F32.F64** Sd, Dm

Eles convertem precisão simples em dupla e precisão dupla em simples.

Para converter de um número inteiro para um número de ponto flutuante, temos

- **VCVT.F64.S32** Dd, Sm
- **VCVT.F32.S32** Sd, Sm
- **VCVT.F64.U32** Dd, Sm
- **VCVT.F32.U32** Sd, Sm

onde a fonte pode ser um inteiro assinado ou não assinado.

Para converter de ponto flutuante para inteiro, temos

- Modo VCVT.S32.F64 Sd, Dm • Modo

## VCVT.S32.F32 Sd, Sm • Modo

## VCVT.U32.F64Sd, Dm • Modo

## VCVT.U32.F32Sd, Sm

Nesta direção, temos o arredondamento, então especificamos o método de arredondamento que queremos com mode. Modo deve ser um dos

- A: Arredondar para o mais próximo, empatar a partir de zero
  - N: Arredondar para o mais próximo, empatar para igualar
  - P: Arredondar para mais infinito
  - M: Arredondar para menos infinito

Existem versões semelhantes para ponto fixo, como

- VCVT.S32.F64 Dd. Dd. #fbits

onde #fbits são o número de bits na parte fracionária do número de ponto fixo.

Nota Este formulário não é útil para cálculos de dinheiro, para aqueles que você deve multiplicar por 100, para duas casas decimais e converter.

## Comparação de ponto flutuante

As instruções de ponto flutuante não afetam o CPSR. Há um Floating Point Status

Control Register (**FPSCR**) para operações de ponto flutuante.

Ele contém sinalizadores N, Z, C e V como o **CPSR**. O significado destes é basicamente o mesmo. Não há versões S das instruções de ponto flutuante.

## Capítulo 11 Operações de ponto flutuante

existe apenas uma instrução que atualiza esses flags, a saber, o **VCMP** instrução. Aqui estão algumas de suas formas:

- VCMP.F32 Sd, Sm
- VCMP.F32 Sd, #0
- VCMP.F64 Dd, Dm
- VCMP.F64 Dd, #0

Ele pode comparar dois registradores de precisão simples ou dois registradores de precisão dupla. registros. Ele permite um valor imediato, ou seja, zero, para que possa comparar um registro de precisão simples ou dupla com zero.

A instrução **VCMP** atualiza o **FPSCR**, mas todas as nossas instruções de condição de ramificação se ramificam com base em sinalizadores no **CPSR**. Isso força uma etapa extra para copiar os sinalizadores do **FPSCR** para o **CPSR** antes de usar uma de nossas instruções regulares para atuar nos resultados da comparação. Existe uma instrução específica para esse fim:

- |        |                  |
|--------|------------------|
| • VMRS | APSR_nzcv, FPSCR |
|--------|------------------|

O VMRS copia apenas os sinalizadores N, Z, C e V do **FPCR** para o **CPSR**. Após a cópia, podemos usar qualquer instrução que leia esses sinalizadores.

O teste de igualdade de números de ponto flutuante é problemático devido ao erro de arredondamento, os números geralmente são próximos, mas não exatamente iguais. A solução é decidir sobre uma tolerância  $e$ , em seguida, considerar os números iguais se estiverem dentro da tolerância  $um$  do outro. Por exemplo, podemos definir  $e = 0,000001$  e então considerar dois registradores iguais se

$$\text{abs}(S1 - S2) < e$$

onde  $\text{abs}()$  é uma função para calcular o valor absoluto.

## Exemplo

Vamos criar um roteamento para testar se dois números de ponto flutuante são iguais usando esta técnica. Primeiro adicionaremos 100 centavos e, em seguida, testaremos se eles são exatamente iguais a \$ 1,00 (alerta de spoiler, eles não serão). Em seguida, compararemos a soma usando nossa rotina fpcomp que os testa dentro de uma tolerância fornecida (geralmente chamada de epsilon).

Começamos com nossa rotina de comparação de ponto flutuante, colocando o conteúdo da Listagem 11-4 em fpcomp.s.

**Listagem 11-4.** Rotina para comparar dois números de ponto flutuante dentro de uma tolerância

```
@ @ Função para comparar com números de ponto flutuante @ os
parâmetros são um ponteiro para os dois números @ e um epsilon de erro.
@ @ Entradas: @ R0 - ponteiro para os 3 números FP @ são x1, x2, e @
Saídas: @
```

```
R0 - 1 se forem iguais, senão 0
```

```
.global fpcomp @ Permitir que a função seja chamada por outros
```

```
@fpcomp:
```

```
 @empurre todos os registradores para ser seguro, nós realmente não
 @precisamos empurrar tantos. push {R4-R12, LR} vpush {S16-S31}
```

## Capítulo 11 Operações de ponto flutuante

@ carregar todos os 3 números de uma vez vldm R0, {S0-S2}

@ calc s3 = x2 - x1 vsub.f32

vabs.f32 vcmp.f32 S3, S1, S0

S3, S3

S3, S2

vmrs APSR\_nzcv, FPSCR

BLE notequal

MOV R0, #1

B feito

notequal:MOV R0, #0

@ restaurar o que preservamos. feito:

vpop {S16-S31} pop {R4-R12, PC}

Agora, o programa principal maincomp.s contém a Listagem 11-5.

**Listagem 11-5.** Programa principal para somar 100 centavos e comparar com \$ 1,00

@

@ Programa principal para testar nossa função de distância @

@ r7 - contador de loop @

r8 - endereço para o conjunto atual de pontos

.global main @ Forneça o endereço inicial do programa ao vinculador

.equa N, 100 @ Número de adições.

principal:

empurre {R4-R12, LR}

@ Some cem centavos e teste @ se eles forem iguais a \$ 1,00

mov r7, #N @ número de iterações do loop

@ carregar centavos, soma corrente e soma real para FPU

```
ldr r0, =cent vldm
r0, {S0-S2}
```

laço:

```
@ adicionar centavos à soma
corrente vadd.f32 s1, s0 subs r7,
#1 @ decrementar o loop se houver mais
pontos bne loop
```

```
@ compara a soma acumulada com a soma
real vcmp.f32 s1, s2 @ copia FPSCR para
CPSR APSR_nzcv, FPSCR @ imprime se
vmrs os números são iguais ou
não são iguais ldr r0, =notequalstr bl printf b next
```

igual: ldr r0, =igualstr bl printf

próximo:

@ carregar ponteiro para soma corrente, soma real e epsilon

```
ldr r0, =runsum vldm
r0, {S0-S2} @ função
de comparação de chamada
```

## Capítulo 11 Operações de ponto flutuante

bl fpcmp @ chama a função de comparação @  
 compara o código de retorno com 1 e imprime se os números @ são  
 iguais ou não (dentro de epsilon). cmp r0, #1 beq equal2 ldr r0,  
 =notequalstr printf bl feito

b

equal2: ldr r0, =equalstr bl printf

feito: mov r0, #0 @ Código de retorno

pop {R4-R12, PC}

.dados

cent: .single 0,01

runsum: .single 0,0

sum: .single 1,00

epsilon:.single 0,00001

equalstr: .asciz "igual\n"

notequalstr: .asciz "diferente\n"

O makefile, na Listagem 11-6, é o esperado.

**Listagem 11-6.** O makefile para o exemplo de comparação de ponto flutuante

```
fpcmp: fpcmp.s maincomp.s gcc
-o fpcmp fpcmp.s maincomp.s
```

Se construirmos e executarmos o programa, obtemos

```
pi@raspberrypi:~/asm/Capítulo 11 $ make gcc -g
-o fpcmp fpcmp.s maincomp.s pi@raspberrypi:~
asm/Capítulo 11 $./fpcmp
```

```
não igual
igual
pi@raspberrypi:~/asm/Capítulo 11 $
```

O programa demonstra como comparar números de ponto flutuante e como copiar os resultados para o **CPSR**, para que possamos ramificar com base no resultado.

Se executarmos o programa em **gdb**, podemos examinar a soma de 100 centavos. Nós vemos

```
S1 = 0x3f7ffff5
S2 = 0x3f80
```

Não falamos sobre o formato de bit dos números de ponto flutuante, mas o primeiro bit é zero indicando positivo. Os próximos 8 bits são o expoente, que é 7F; o expoente não usa complemento de dois; em vez disso, seu valor é o que há menos 127. Nesse caso, o expoente é 0. Como S2 não tem mais bits, mas na forma normalizada, há um 1 implícito após o expoente, então isso dá o valor de 1. Então S1 tem um valor de 0,99999934, mostrando o erro de arredondamento rastejando, mesmo no pequeno número de adições que realizamos.

Então chamamos nossa rotina **fpcmp** que determina se os números são dentro da tolerância prevista e que os considere iguais.

Não foram necessárias muitas adições para começar a introduzir erros de arredondamento em nossas somas. Você deve ter cuidado ao usar ponto flutuante para isso razão.

## Resumo

Neste capítulo, abordamos o que são números de ponto flutuante e como eles são representados. Cobrimos normalização, NaNs e erro de arredondamento. Mostramos como criar números de ponto flutuante em nossa seção **.data** e

## Capítulo 11 Operações de ponto flutuante

discutiu o banco de registradores de ponto flutuante de precisão simples e dupla e como eles se sobrepõem. Cobrimos como carregá-los nos registradores de ponto flutuante, realizar operações matemáticas e salvá-los de volta no memória.

Vimos como converter entre diferentes tipos de ponto flutuante, como comparar números de ponto flutuante e como copiar o resultado de volta para a CPU ARM. Examinamos o efeito do erro de arredondamento nessas comparações.

No Capítulo 12, “Coprocessador NEON”, veremos como executar múltiplas operações de ponto flutuante em paralelo.

## CAPÍTULO 12

# Coprocessador NEON

Neste capítulo, começamos a executar a verdadeira computação paralela. O coprocessador NEON compartilha muitas funcionalidades com a FPU do Capítulo 11, “Operações de ponto flutuante”, mas pode realizar várias operações ao mesmo tempo. Por exemplo, você pode executar quatro operações de ponto flutuante de 32 bits com uma instrução e essas quatro operações são executadas ao mesmo tempo. O tipo de processamento paralelo realizado pelo coprocessador NEON é Single Instruction Multiple Data (**SIMD**). No processamento SIMD, cada instrução única que você emite é executada em paralelo em vários itens de dados múltiplos.

---

**Observação** O coprocessador NEON foi introduzido com o Raspberry Pi 2.

Não está disponível no Raspberry Pi 1 ou no Raspberry Pi Zero. Os programas neste capítulo são executados apenas em um Raspberry Pi 2 ou posterior.

---

O coprocessador NEON compartilha o mesmo arquivo de registradores que examinamos no Capítulo 11, “Operações de ponto flutuante”, exceto que ele vê um conjunto maior de registradores. Todas as instruções que aprendemos para carregar e armazenar os registradores FPU são as mesmas aqui, incluindo **VMOV**, **VLDR**, **VSTR**, **VLDM**, **VSTM**,

## Capítulo 12 Coprocessador NEON

e **VPOP**. Examinaremos como os registradores NEON estendem o conjunto de registradores FPU e como eles devem ser usados com o NEON.

Examinaremos como organizar os dados para que possamos operá-los em paralelo e estudaremos as instruções que o fazem. Em seguida, atualizaremos nossos programas de distância vetorial e multiplicação de matrizes 3x3 para usar o processador NEON para ver quanto do trabalho podemos fazer em paralelo.

## Os registros NEON

O coprocessador NEON pode operar nos registradores de 64 bits, que estudamos no capítulo anterior, e em um conjunto de 16 registradores de 128 bits, que são novos neste capítulo.

---

**Observação** Todos esses registros se sobrepõem, portanto, tome cuidado ao usar uma combinação. Veja a Figura 12-1 para as sobreposições.

---

O coprocessador NEON não pode referenciar os registradores **S** de 32 bits que o FPU comumente usa. Qualquer processador ARM com um coprocessador NEON terá todos os registradores 32 **D**.

| S0-S31   | D0-D15      | Q0-Q15    |
|----------|-------------|-----------|
| FPU Only | FPU or Neon | Neon Only |
| S0       | D0          | Q0        |
| S1       |             |           |
| S2       | D1          |           |
| S3       |             |           |
| S4       | D2          | Q1        |
| S5       |             |           |
| S6       | D3          |           |
| S7       |             |           |
| ...      | ...         |           |
| S28      | D14         | Q7        |
| S29      |             |           |
| S30      | D15         |           |
| S31      |             |           |
|          | D16         | Q8        |
|          | D17         |           |
|          | ...         | ...       |
|          | D30         | Q15       |
|          | D31         |           |

**Figura 12-1.** O conjunto completo de registradores de coprocessador para os coprocessadores FPU e NEON

Ter registradores de 128 bits não significa que o processador NEON executa aritmética de 128 bits. Na verdade, o processador NEON não pode nem executar aritmética de ponto flutuante de 64 bits. Se você se lembra do Capítulo 10, “Multiplicar, Dividir e Acumular”, no segundo exemplo de multiplicação de matrizes, otimizamos o programa usando uma versão da instrução de multiplicação inteira que fazia a multiplicação como duas operações independentes de 16 bits

## Capítulo 12 Coprocessador NEON

ao mesmo tempo. Este foi nosso primeiro encontro com a programação SIMD.

O coprocessador NEON leva essa ideia a um novo nível, porque em um registrador de 128 bits, podemos ajustar quatro números de ponto flutuante de precisão simples de 32 bits.

Se multiplicarmos dois desses registradores, todos os quatro números de 32 bits serão multiplicados ao mesmo tempo.

O coprocessador NEON pode operar em inteiros e flutuantes números de ponto. No entanto, os tamanhos são limitados a 8, 16 e 32 bits para números inteiros e 16 e 32 bits para ponto flutuante para executar o máximo de operações possível de uma só vez. O maior paralelismo é obtido usando números inteiros de 8 bits, onde 16 operações podem acontecer ao mesmo tempo.

O coprocessador NEON pode operar em registradores **D** de 64 bits ou **Q** de 128 bits ; é claro, se você usar registradores **D** de 64 bits , terá apenas metade da quantidade de paralelismo.

A Tabela 12-1 mostra o número de elementos que cabem em cada tipo de registrador.

A seguir, veremos como podemos realizar aritmética sobre esses elementos.

**Tabela 12-1. Número de elementos em cada tipo de registro por tamanho**

|                        | elementos de 8 bits | Elementos de 16 bits | Elementos de 32 bits |
|------------------------|---------------------|----------------------|----------------------|
| registrar D de 64 bits | 8                   | 4                    | 2                    |
| Registro Q de 128 bits | 16                  | 8                    | 4                    |

## Fique na sua pista

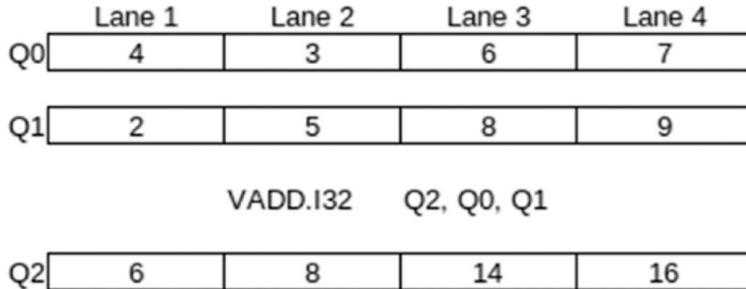
O coprocessador NEON usa o conceito de pistas para todos os seus cálculos.

Quando você escolhe seu tipo de dado, o processador considera o registrador dividido em número de raias – uma raia para cada elemento de dados. Se nós

trabalham com inteiros de 32 bits e usam um registrador **Q** de 128 bits , então o registrador é considerado dividido em quatro raias, uma para cada inteiro.

A Figura 12-2 mostra como os registradores **Q** são divididos em quatro raias, uma para cada número de 32 bits, como a operação aritmética é aplicada a cada faixa independentemente. Desta forma, realizamos quatro adições em uma

instrução, e o coprocessador NEON as executa todas ao mesmo tempo - em paralelo.



**Figura 12-2.** Exemplo das quatro pistas envolvidas na adição de 32 bits nos registradores Q

## Operações aritméticas

A Figura 12-2 mostra nosso primeiro exemplo de uma instrução de coprocessador NEON. As duas formas da instrução VADD para NEON são

- VADD.datatype {Qd}, Qn, Qm
- VADD.datatype {Dd}, Dn, Dm

O tipo de dados deve ser um de **I8**, **I16**, **I32**, **I64** ou **F32**.

**Observação** Isso é muito semelhante à instrução VADD que vimos para o FPU. O Assembler sabe que a instrução é para o FPU, se você usar registradores **S** - o NEON não suporta esses ou se você usar o tipo **F64** - que o NEON não suporta.

O truque para usar o NEON é organizar seu código para que você mantenha todos os pistas fazendo um trabalho útil.

## Capítulo 12 Coprocessador NEON

O coprocessador NEON tem muitas instruções aritméticas e há muitas semelhanças com os FPUs. No entanto, existem algumas diferenças, como NEON não suporta divisão, mas suporta recíproco, então você pode fazer a divisão pegando o recíproco e multiplicando.

Estranhamente, NEON não suporta raiz quadrada, mas suporta raiz quadrada recíproca.

Como o processador NEON suporta operações inteiras, ele suporta todas as operações lógicas como **and**, **bic** e **orr**. Há também mais operações de comparação do que o FPU suporta.

Se você olhar para a lista de instruções NEON, há muitas instruções especiais fornecidas para ajudar com algoritmos específicos. Por exemplo, há suporte direto para polinômios sobre o anel binário para suportar certas classes de algoritmos criptográficos.

Mostraremos como usar algumas das instruções em exemplos de trabalho. Isso lhe dará conhecimento suficiente para aplicar os princípios gerais de operações para o coprocessador NEON, então você pode examinar todas as instruções no “Guia de referência do conjunto de instruções ARM”.

## Distância vetorial 4D

Para nosso primeiro exemplo, vamos expandir o exemplo de cálculo de distância do Capítulo 11, “Operações de ponto flutuante,” para calcular a distância entre dois vetores 4D. A fórmula se generaliza para qualquer número de dimensões apenas adicionando os quadrados extras das diferenças para as dimensões adicionais sob a raiz quadrada.

First distance.s, mostrado na Listagem 12-1, usando o coprocessador NEON.

### Listagem 12-1. Rotina para calcular a distância entre dois vetores 4D usando o coprocessador NEON

@

@ Função de exemplo para calcular a distância @ entre dois pontos 4D em precisão simples @ ponto flutuante usando o processador NEON @

@ Entradas:

são (x1, R0, x2, y1, y2, y3, y4) números FP @ eles

@

@

@ Saídas:

R0 - o comprimento (como FP de precisão simples) @

.distância global @ Permitir que a função seja chamada por outros

@ distância:

@ empurre todos os registradores para ser seguro, nós  
 @ realmente não precisamos empurrar tantos. push {R4-R12, LR} vpush {S16-S31}

@ carregar todos os 4 números de  
 uma vez vldm R0, {Q2-Q3}

@ calc q1 = q2 - q3  
 vsub.f32 Q1, Q2, Q3 @ calc  
 $Q1 = Q1 * Q1 (xi - yi)^2$  vmul.f32 Q1, Q1,  
 Q1 @ calc S0 = S0 + S1 + S2 + S3  
 vpadd. f32 D0, D2, D3 vadd.f32 S0, S1

## Capítulo 12 Coprocessador NEON

```

@ calc sqrt(S4)
vsqrt.f32 S4, S0 @
move o resultado para R0 para ser retornado
vmov R0, S4

@ restaurar o que preservamos. vpop
{S16-S31} pop {R4-R12, PC}

```

Agora main.s, mostrado na Listagem 12-2, para testar a rotina.

**Listagem 12-2.** O programa principal para testar a função de distância 4D

```

@
@ Programa principal para testar nossa função de distância
@
@ r7 - contador de loop
@ r8 - endereço para o conjunto atual de pontos
.global main @ Fornecer início do programa

.equ N, 3 @ Número de pontos.

principal:
 empurre {R4-R12, LR}

 ldr r8, =points @ ponteiro para os pontos atuais

 mov r7, #N @ número de iterações do loop

loop: mov r0, r8 @ mover o ponteiro para o parâmetro 1 (r0)

 bl distância @ função de distância de chamada

 @ precisa pegar o valor de retorno de precisão simples @ e
 convertê-lo em duplo, porque a função C printf @ só pode imprimir
 duplos.

```

```
r0 vcvt.f64.f32 d0, s2 @ convertir para duplo versão 2, r0, d2,
@ retornar duplo para r2, r3 ldr r0, =prtstr @ carregar string de
impressão bl printf @ imprimir a distancia
```

```
add r8, #(8*4) @ 8 elementos cada 4 bytes subs r7,
#1 @ contador de loop se
```

```
mov r0, #0 @ Código de retorno
pop {R4-R12, PC}
```

.dados

```
pontos: .single .single .single 0,0, 0,0, 0,0, 0,0, 17,0, 4,0, 2,0, 1,0
1.323e10, 1,3, 5,4, 3,1, -1,5, -2,4, 0,323, 3,4, -0,232
-1.2e-4, 34.55, 5454.234, 10.9, -3.6, 4.2, 1.3 .asciz "Distância = %f\n"
prtstr:
```

O makefile está na Listagem 12-3.

### Listagem 12-3. O makefile para o programa à distância

distância: distance.s main.s

```
gcc -mfpu=neon-vfpv4 -o distance distance.s main.s
```

Se construirmos e executarmos o programa, veremos

```
pi@raspberrypi:~/asm/Chapter 12 $ make gcc
-mfpu=neon-vfpv4 -g -o distance distance.s main.s pi@raspberrypi:~/
asm/Chapter 12 $./distance Distance = 17.606817 Distance =
6.415898 Distância = 13230000128.000000 pi@raspberrypi:~/asm/
Capítulo 12 $
```

## Capítulo 12 Coprocessador NEON

Carregamos um vetor em **Q2** e o outro em **Q3**. Cada vetor consiste de quatro números de ponto flutuante de 32 bits, para que cada um possa ser colocado em um registrador **Q** de 128 bits e tratado como quatro faixas. Em seguida, subtraímos todos os quatro componentes de uma vez usando uma única instrução **VSUB.F32**. Calculamos os quadrados todos de uma vez usando uma instrução **VMUL.F32**. Ambas as instruções operam em todas as quatro pistas em paralelo.

Queremos somar todas as somas que estão todas em **Q1**. Isso significa que todos os números estão em faixas diferentes e não podemos adicioná-los em paralelo. Esta é uma situação comum para entrar; felizmente, o conjunto de instruções NEON nos dá alguma ajuda. Ele não somará todas as faixas em um registro, mas fará adições aos pares em paralelo. A instrução

```
vpadd.f32 D0, D2, D3
```

adicionará os dois números de 32 bits em **D2**, colocará a soma na metade de **D0** e, da mesma forma, adicionará as duas metades de **D3**, colocando a soma na outra metade de **D0**. A instrução pairwise opera apenas nos registradores **D** e faz apenas duas adições de 32 bits por vez. Os números a somar estão todos em **Q1**, por isso selecionamos **D2** e **D3** para a instrução, pois são os registradores que se sobrepõem a **Q1**; consulte a Figura 12-1.

Isso realiza duas das adições de que precisamos; então executamos o terceiro usando a instrução FPU **VADD.F32** regular, observando que **S0** e **S1** se sobrepõem a **D0**.

Depois que os números são adicionados, usamos a instrução de raiz quadrada da FPU para calcular a distância final.

A Figura 12-3 mostra como essas operações fluem pelas pistas em nossos registradores.

| Lane 1                                            | Lane 2              | Lane 3    | Lane 4    |
|---------------------------------------------------|---------------------|-----------|-----------|
| Q2[x1]                                            | x2                  | x3        | x4        |
| Q3[y1]                                            | y2                  | y3        | y4        |
| vsub.f32 Q1, Q2, Q3                               |                     |           |           |
| Q1[x1-y1]                                         | x2-y2               | x3-y3     | x4-y4     |
| vmul.f32 Q1, Q1, Q1                               |                     |           |           |
| Q1=D2, D3[(x1-y1)^2]                              | (x2-y2)^2           | (x3-y3)^2 | (x4-y4)^2 |
| vpadd.f32 D0, D2, D3                              |                     |           |           |
| D0=S0, S1[(x1-y1)^2+(x2-y2)^2]                    | (x3-y3)^2+(x4-y4)^2 |           |           |
| vadd.f32 S0, S1                                   |                     |           |           |
| S0[(x1-y1)^2+(x2-y2)^2+(x3-y3)^2+(x4-y4)^2]       |                     |           |           |
| vsqrt.f32 S4, S0                                  |                     |           |           |
| S4[SQRT((x1-y1)^2+(x2-y2)^2+(x3-y3)^2+(x4-y4)^2)] |                     |           |           |

**Figura 12-3.** Fluxo dos cálculos pelos registros mostrando as pistas

Isso mostra um bom recurso de ter os registradores de compartilhamento NEON e FPU, pois nos permite misturar as instruções FPU e NEON sem a necessidade de mover os dados.

A única alteração no programa principal é tornar os vetores 4D e ajustar o loop para usar o novo tamanho de vetor.

Observe que o makefile inclui a opção gcc:

```
-mfpu=neon-vfpv4
```

Isso informa ao compilador GNU C ou GNU Assembler que você possui um coprocessador NEON e deseja gerar código para ele. Se não incluirmos isso, receberemos muitos erros sobre instruções que não são suportadas em nosso processador. Isso ocorre porque, por padrão, as ferramentas visam todos os modelos Raspberry Pi, e o que estamos fazendo não funcionará no Pi 1 ou Zero.

## Multiplicação de Matrizes 3x3

Vamos pegar o programa de exemplo de multiplicação de matrizes 3x3 do Capítulo 10, “Multiplicar, Dividir e Acumular”, e otimizá-lo usando as habilidades de processamento paralelo do coprocessador NEON.

## Capítulo 12 Coprocessador NEON

O coprocessador NEON tem uma função de produto escalar **VSDOT**, mas, infelizmente, só opera em números inteiros de 8 bits. Isso não é adequado para a maioria das matrizes, portanto não o usaremos. Como vimos no último exemplo, adicionar dentro de um registro é um problema e, da mesma forma, há problemas em apenas multiplicar com acumulações. A solução recomendada é inverter dois dos nossos loops do programa anterior. Dessa forma, fazemos a multiplicação com acumulações como instruções separadas, mas fazemos isso em três vetores por vez. O resultado é que eliminamos um de nossos loops do programa anterior e alcançamos algum nível de operação paralela.

O truque é perceber que uma multiplicação de matriz 3x3 é realmente três matrizes por cálculos vetoriais, ou seja

- $C_{col1} = A * B_{col1}$
- $C_{col2} = A * B_{col2}$
- $C_{col3} = A * B_{col3}$

Se olharmos para uma dessas matrizes vezes um vetor, por exemplo

$$\left| \begin{array}{ccc|c} a & b & c & x \\ d & e & f & y \\ g & h & i & z \end{array} \right|$$

vemos que o cálculo é

$$\left| \begin{array}{c} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{array} \right|$$

Se colocarmos a, d e g em um registro em faixas separadas e b, e e h em outro registro e c, f e i em um terceiro registro nas faixas correspondentes, podemos calcular uma coluna nos resultados matriz, conforme mostrado na Figura 12-4.

| Lane 1                                 | Lane 2 | Lane 3 |                                      |
|----------------------------------------|--------|--------|--------------------------------------|
| D1<br>a                                | d      | g      | Mult D1 by x and place result in D4  |
| D2<br>b                                | e      | f      | Mult D2 by y and add to result in D4 |
| D3<br>c                                | f      | i      | Mult D3 by z and add to result in D4 |
| D4<br>ax+by+cz    dx+ey+fz    gx+hy+iz |        |        |                                      |

**Figura 12-4.** Mostrando como os cálculos fluem pelas pistas

Este é o algoritmo recomendado para multiplicação de matrizes no coprocessador NEON. Usaremos inteiros curtos como fizemos antes, para que possamos encaixar uma coluna de qualquer uma de nossas matrizes em um registrador **D**.

O que fizemos anteriormente foi para uma coluna da matriz de resultados; então precisamos fazer isso para todas as colunas. Colocaremos essa lógica em uma macro, para que possamos repetir o cálculo três vezes. Como o objetivo é a multiplicação de matrizes o mais rápido possível, vale a pena remover os loops, pois economiza lógica extra. Isso faz com que o programa pareça muito mais simples.

A Listagem 12-4 é o código para nossa multiplicação de matriz habilitada para NEON.

#### Listagem 12-4. Exemplo de multiplicação de matriz 3x3 habilitado para NEON

```
@ @ Multiplica 2 matrizes inteiras 3x3 @ Usa o
coprocessador NEON para fazer @ algumas
operações em paralelo. @ @ Registradores: D0
- primeira coluna da matriz A @ @ @ @ @ @
@ @ @
```

```
D1 - segunda coluna da matriz A
D2 - terceira coluna da matriz A
D3 - primeira coluna da matriz B
D4 - segunda coluna da matriz B
D5 - terceira coluna da matriz B
D6 - primeira coluna da matriz C
D7 - segunda coluna da matriz C
D8 - terceira coluna da matriz C
```

## Capítulo 12 Coprocessador NEON

.global main @ Forneça o endereço inicial do programa

principal:

push {R4-R12, LR} @ Salve os regs necessários

@ carregar a matriz A nos registros NEON D0, D1, D2

|        |             |                                  |
|--------|-------------|----------------------------------|
| LDR    | R0, =A      | @ Endereço de A @                |
| VLDL M | R0, {D0-D2} | carrega em massa as três colunas |

@ carrega a matriz B nos registros NEON D3, D4, D5

|        |             |                                  |
|--------|-------------|----------------------------------|
| LDR    | R0, =B      | @ Endereço de B @                |
| VLDL M | R0, {D3-D5} | carrega em massa as três colunas |

.MACRO mulcol ccol bcol

VMUL.I16 \ccol, D0, \bcol[0]

VMLA.I16 \ccol, D1, \bcol[1]

VMLA.I16 \ccol, D2, \bcol[2]

.ENDM

|        |        |                               |
|--------|--------|-------------------------------|
| mulcol | D6, D3 | @ processa a primeira coluna  |
| mulcol | D7, D4 | @ processa a segunda coluna @ |
| mulcol | D8, D5 | processa a terceira coluna    |

LDR R1, =C @ Endereço de C @

VSTM R1, {D6-D8} armazena as três colunas

@ Imprima a matriz C @

Percorra 3 linhas imprimindo 3 colunas de cada vez.

|                    |                                    |
|--------------------|------------------------------------|
| MOV R5, #3 LDR     | @ Imprimir 3 linhas                |
| R11, =C printloop: | @ Endereço da matriz de resultados |

LDR R0, =prtstr @ printf string de formato @ imprime

transposição de modo que a matriz esteja @ na ordem usual de linha e coluna. @ primeiros pós-índices ldrh por 2 para a próxima linha

@ então o segundo ldrh adiciona 6, então está à frente

@ em  $2+6=8$ =tamanho da linha @ da mesma forma

para o terceiro ldr à frente @ em  $2+14=16 = 2$  x

tamanho da linha

|                           |                                            |
|---------------------------|--------------------------------------------|
| LDRH R1, [R11], nº 2      | @ primeiro elemento da linha atual @       |
| LDRH R2, [R11, #6]        | segundo elemento da linha atual @ terceiro |
| LDRH R3, [R11,#14] printf | elemento da linha atual                    |
| BL                        | @ Chama printf                             |
| SUBS R5, nº 1             | @ Contador de loop Dec                     |
| Círculo de impressão BNE  | @ Se não for loop zero                     |
| movimento r0, #0          | @ Código de retorno                        |
| pop {R4-R12, PC}          | @ Restaurar regs e retornar                |

.dados

@ Primeira matriz na ordem principal da coluna

A: .short 1, 4, 7, 0 2, 5,

.curto 8, 0 3, 6, 9, 0

.curto

@ Segunda matriz na ordem principal da coluna

B: 9, 6, 3, 0 8, 9, 2, 0 7, 4, 1, 0

.curto

.curto

@ Matriz de resultados na ordem principal da coluna

C: .preencher 12, 2, 0

prtstr: .asciz "%3d %3d %3d\n"

Armazenamos ambas as matrizes na ordem principal da coluna, e a matriz C é produzido na ordem principal da coluna. Isso é para facilitar a configuração dos cálculos, já que tudo está alinhado corretamente para carregar em massa em nossos registradores NEON. Alteramos o loop de impressão, para que ele imprima a matriz de resultados em nosso formulário de ordem de linha usual, basicamente fazendo uma transposição de matriz à medida que percorre a matriz C.

## Capítulo 12 Coprocessador NEON

Na macro, fazemos a multiplicação escalar

```
VMUL.I16 \ccol, D0, \bcol[0]
```

que se traduz em algo como

```
VMUL.I16 D6, D0, D3[0]
```

Não temos acesso aos registradores S para acessar um único ponto flutuante número, mas em muitas instruções, podemos nos referir a uma determinada pista. Aqui, a sintaxe **D3[0]** é como um índice de matriz em **D3** ou pode ser considerada como **D3** pista 0—contando pistas a partir de zero. Quando multiplicamos um registrador com raias por uma única raia, o **VMUL** realizará uma multiplicação escalar do número único por cada raia no primeiro operando - D3 neste caso.

## Resumo

Este capítulo foi uma rápida visão geral de como o coprocessador NEON funciona e como escrever programas para ele. Cobrimos como o NEON usa pistas para realizar cálculos paralelos e uma seleção das instruções disponíveis para cálculos. Demos dois exemplos, um para calcular a distância entre dois vetores 4D e outro para realizar a multiplicação de matrizes 3x3 para demonstrar como você pode aproveitar facilmente o poder do NEON coprocessador.

No Capítulo 13, “Instruções condicionais e código de otimização”, veremos o que os quatro bits do código de condição em cada instrução fazem e como aproveitá-los.

## CAPÍTULO 13

# Condisional Instruções e Código de Otimização

No Capítulo 4, “Controlando o fluxo do programa”, aprendemos como ramificar o código condicionalmente com base em sinalizadores no **CPSR**. Neste capítulo, veremos como isso pode ser generalizado para todas as instruções.

No Capítulo 1, “Introdução”, analisamos o formato de instrução ARM e observamos que quase todas as instruções contêm um código de condição de 4 bits. Até agora, ignoramos o propósito desses bits; agora veremos como usá-los e por que queremos executar instruções condicionalmente para reduzir o número de instruções de desvio. Queremos minimizar o número de instruções de desvio, porque são caras de executar, pois interrompem o pipeline de execução.

Para demonstrar, aplicaremos instruções condicionais ao nosso programa em letras maiúsculas, sobre o qual falamos no Capítulo 4, “Controlando o fluxo do programa” e no Capítulo 6, “Funções e a pilha”. Vamos otimizar sua comparação de intervalo para usar uma única instrução condicional e, em seguida, examinar algumas outras otimizações de código que podemos aplicar.

## Capítulo 13 Instruções Condicionais e Código de Otimização

## Razões para não usar instruções condicionais

Antes de entrarmos em como e por que adicionar códigos de condição a todas as nossas instruções, quero observar alguns motivos pelos quais eles estão se tornando obsoletos e não são necessários hoje tanto quanto eram nos primeiros dias do processador ARM, como quando portar seu código Assembly para 64 bits e o pipeline aprimorado.

### Nenhuma instrução condicional em 64 bits

Quando os engenheiros da ARM projetaram o conjunto de instruções de 64 bits, eles mantiveram o comprimento da instrução em 32 bits. No entanto, eles queriam dobrar o número de registradores e aumentar o número de opcodes. Para fazer isso, eles pegaram os 4 bits dedicados às instruções condicionais e os distribuíram para esses outras finalidades.

Portanto, no modo de 64 bits do ARM, há apenas um punhado de instruções do tipo ramificação que podem ser executadas condicionalmente. Se você planeja portar seu código Assembly para 64 bits um dia, será muito menos trabalhoso se não estiver cheio de instruções condicionais.

### Pipeline Melhorado

Neste capítulo, promovemos instruções condicionais como uma solução para desvios que fazem com que o pipeline de instruções pare e prejudique o desempenho. Isso não fará com que o código de 64 bits seja muito mais lento do que o código de 32 bits? Os engenheiros da ARM atenuaram esse problema tornando o pipeline de instruções mais sofisticado. Eles criaram uma tabela que mantém um registro de onde vão as instruções de desvio recentes, com a maioria dos loops retornando ao loop 90% do tempo e avançando 10%. Sabendo disso, o pipeline de instruções pode fazer uma suposição informada e continuar trabalhando como se a ramificação fosse ocorrer. Dessa forma, a ramificação só interromperá o pipeline

## Capítulo 13 Instruções Condicionais e Código de Otimização

quando segue sua rota menos percorrida. Isso é chamado de **previsão de ramificação**. Existem outras melhorias no cache e no pipeline para acelerar as coisas em geral.

A principal conclusão é que as instruções condicionais apresentadas neste capítulo ajudarão o código executado em Raspberry Pis mais antigo do que novos.

## Sobre o Código Condisional

Podemos adicionar qualquer código de condição da Tabela 4-1 do Capítulo 4 a praticamente qualquer instrução Assembly. As únicas exceções incluem definir um ponto de interrupção, interromper o processador e algumas outras instruções. Quando a condição não é atendida, a instrução executa uma operação no e executa a próxima instrução, por exemplo:

ADDEQ                  R2, R3, R4

só executa a adição se o sinalizador **Z** estiver definido. Se o sinalizador **Z** não estiver definido, essa instrução será ignorada, mas ainda levará um ciclo de instrução.

Podemos combinar isso com o modificador **S**

ADICIONAIS                  R2, R3, R4

nesse caso, o **CPSR** é atualizado se a instrução **ADD** for executada. O **S** deve vir por último, ou você confundirá o Montador.

## Otimizando a rotina de letras maiúsculas

Nossa rotina original de letras maiúsculas implementa o pseudocódigo

```
SE (R5 >= 'a') E (R5 <= 'z') ENTÃO
 R5 = R5 - ('a'-'A')
FIM SE
```

## Capítulo 13 Instruções Condicionais e Código de Otimização

com o seguinte código Assembly:

@ Se R5 > 'z' então vá para cont

```
CMP R5, #'z' @ é a letra > 'z'?
BGT cont.
```

@ Else if R5 < 'a' então vá para o fim se

```
CMP R5, #'a'
BLT cont. @ goto para terminar
```

if @ se chegamos aqui então a letra é minúscula, então converta-a.

```
SUB R5, #('a'-'A') cont: @
```

end if

Este código implementa a lógica reversa de desvio em torno da instrução **SUB** se **R5 < 'a'** ou **R5 > 'z'**. Isso foi bom para um capítulo ensinando instruções de ramificação, uma vez que demonstrou duas delas. Neste capítulo, examinamos a eliminação completa das ramificações, então vamos ver como podemos melhorar esse código passo a passo.

## Simplificando a comparação de intervalo

Uma maneira comum de simplificar as comparações de intervalo é deslocar o intervalo, portanto, não precisamos de uma comparação inferior. Se subtraímos 'a' de tudo, nosso pseudo-código se torna

```
R5 = R5 - 'a'
SE (R5 >= 0) E R5 <= ('z'-'a') ENTÃO
 R5 = R5 + 'A'
FIM SE
```

Se tratarmos **R5** como um inteiro sem sinal, então a primeira comparação não nada, pois todos os inteiros sem sinal são maiores que 0. Nesse caso, simplificamos nosso intervalo de duas comparações para uma comparação que é **R5 <= ('z'-'a')**.

## Capítulo 13 Instruções Condicionais e Código de Otimização

Isso nos leva à primeira versão melhorada de nosso arquivo `upper.s`. Esta nova `upper.s` é mostrado na Listagem 13-1.

**Listagem 13-1.** Rotina de letras maiúsculas com comparação de intervalo simplificada

```
@ @ Programa Assembler para converter uma string em
@ todas as letras maiúsculas. @ @ R1 - endereço da
string de saída @ R0 - endereço da string de entrada @
R4 - string de saída original para comprimento calc.
```

```
@ R5 - caractere atual sendo processado @ R6 -
menos 'a' para comparar < 26. @
```

```
.global toupper @ Permitir que outros arquivos chamem esta rotina
```

```
superior: MOV R4, R0 @ Salve os registradores que usamos.
```

```
@ O loop é até que o byte apontado por R1 seja um loop diferente de
zero: LDRB R5, [R0], #1 @ load character and incr
```

```
@ Quer saber se 'a' <= R5 <= 'z'
```

```
@ Primeiro subtraia 'a'
```

```
 SUB R6, R5, #'a'
```

```
@ Agora quero saber se R6 <= 25
```

```
 CMP R6, n° 25 @ caracteres são 0-25 após o turno
 BHI cont
```

```
@ se chegamos aqui, então a letra é minúscula, então converta-a.
```

```
 SUB R5, #('a'-'A') cont: @
```

```
end if
```

```
 STRB R5, [R1], #1 @ armazena caractere para saída str
```

```
 CMP R5, n° 0 @ pare ao atingir um nulo
```

## Capítulo 13 Instruções Condicionais e Código de Otimização

```
SUB R0, R1, R4 @ obtém@ temporárias para armazenar os ponteiros (R0 e R6)
R6} @ Restaura o registrador que usamos.
```

POP

BX LR @ Retornar ao chamador

Todos os exemplos neste capítulo usam os mesmos main.s da Listagem 6-3, exceto o terceiro que ignora a necessidade de um main.s. A Listagem 13-2 é um makefile para todo o código neste capítulo. Comente todos os programas que você ainda não acessou, ou você receberá um erro de compilação.

### Listagem 13-2. Makefile para a versão de rotina em maiúsculas neste capítulo

```
UPPEROBJJS = main.o upper.o
```

```
UPPER2OBJJS = main.o upper2.o
```

```
UPPER3OBJJS = upper3.o UPPER4OBJJS
```

```
= main.o upper4.o
```

```
todos: superior superior2 superior3 superior4
```

```
% .o: %.s
```

```
 as -mfpu=neon-vfpv4 $(DEBUGFLGS) $(LSTFLGS) $< -o $@
```

```
superior: $(UPPEROBJJS) ld
```

```
 -o superior $(UPPEROBJJS)
```

```
upper2: $(UPPER2OBJJS) ld -o
```

```
 upper2 $(UPPER2OBJJS)
```

```
upper3: $(UPPER3OBJJS) ld -o
```

```
 upper3 $(UPPER3OBJJS)
```

```
upper4: $(UPPER4OBJJS) ld -o
```

```
 upper4 $(UPPER4OBJJS)
```

## Capítulo 13 Instruções Condicionais e Código de Otimização

Esta é uma melhoria e uma ótima otimização para usar quando você precisa de comparações de intervalo. Vamos usar uma instrução condicional para remover outro ramo.

## Usando uma instrução condicional

A instrução óbvia para tornar condicional é a subtração que faz o conversão para maiúsculas. A Listagem 13-3 é a versão upper2.s da rotina.

**Listagem 13-3.** Rotina de letras maiúsculas usando uma instrução **SUBL.S** condicional

```
@ @ Programa Assembler para converter uma string em
@ todas as letras maiúsculas. @ @ R1 - endereço da
string de saída @ R0 - endereço da string de entrada @
R4 - string de saída original para comprimento calc.

@ R5 - caractere atual sendo processado @ R6 -
menos 'a' para comparar < 26. @
```

|                                                                     |                                                   |                                    |
|---------------------------------------------------------------------|---------------------------------------------------|------------------------------------|
| .topper global                                                      | @ Permitir que outros arquivos chamem esta rotina |                                    |
| topper:                                                             | PUSH {R4-R6}                                      | @ Salve os registradores           |
|                                                                     | MOV R4, R1                                        |                                    |
| @ O loop é até que o byte apontado por R1 seja um loop diferente de |                                                   |                                    |
| zero: LDRB R5, [R0], #1 @ load character and incr                   |                                                   |                                    |
| @ Quer saber se 'a' <= R5 <= 'z'                                    |                                                   |                                    |
| @ Primeiro subtraia 'a'                                             |                                                   |                                    |
|                                                                     | SUB R6, R5, #'a'                                  |                                    |
| @ Agora quero saber se R6 <= 25                                     |                                                   |                                    |
|                                                                     | CMP R6, n° 25                                     | @ caracteres são 0-25 após o turno |

## Capítulo 13 Instruções Condicionais e Código de Otimização

@ se chegamos aqui, então a letra é minúscula, então converta-a.

```
SUBL S R5, #'a'-'A'
STRB R5, [R1], #1 @ armazenar o caractere CMP
R5, #0 @ parar ao atingir seu caractere menor ou igual BNE loop
subtraindo os ponteiros SUBS R5, R4 @ subtraímos o comprimento
```

POP {R4-R6}

BX LR @ Retornar ao chamador

Usamos a instrução **SUBL S** aqui. **LS** é para menor ou mesmo que é o sufixo para unsigned <=. A instrução **SUBL S** só fará a subtração se **R5** for menor ou igual a 25, que é onde deslocamos 'z'.

Na Listagem 13-3, a única instrução de desvio é para o loop.

**Nota** Se a instrução **SUBL S** não fizer nada, ela ainda levará um ciclo para ser executada. Isso significa que só faz sentido usar isso em vez de um desvio, se colocarmos a condição em até três instruções. Caso contrário, a ramificação do código é mais rápida.

## Restringindo o Domínio do Problema

Ao otimizar o código, as melhores otimizações surgem da restrição do domínio do problema. Se estivermos lidando apenas com caracteres alfabéticos, podemos eliminar totalmente a comparação de intervalo. Se examinarmos o Apêndice E, "Conjunto de caracteres ASCII", notamos que a única diferença entre letras maiúsculas e minúsculas é que as letras minúsculas têm o conjunto de bits 0x20, enquanto as maiúsculas não. Isso significa que podemos converter uma letra minúscula em maiúscula executando uma operação Bit Clear (**BIC**) nesse bit. Se fizermos isso com caracteres especiais, isso irá atrapalhar alguns deles.

## Capítulo 13 Instruções Condicionais e Código de Otimização

Freqüentemente, na computação, queremos que nosso código não diferencie maiúsculas de minúsculas, o que significa que você pode inserir qualquer combinação de maiúsculas e minúsculas. O Assembler faz isso, então não importa se entramos em MOV ou mov. Da mesma forma, muitas linguagens de computador não diferenciam maiúsculas de minúsculas, portanto, você pode inserir nomes de variáveis em qualquer combinação de maiúsculas e minúsculas e isso significa a mesma coisa. Os algoritmos de IA que processam texto sempre os convertem em um formato padrão, geralmente jogando fora toda a pontuação e convertendo-os em um único caso. Forçar essa padronização economiza muito processamento extra posteriormente.

Vejamos uma implementação disso para nosso código — a Listagem 13-4 vai para upper3.s.

**Listagem 13-4.** Rotina de letras maiúsculas como uma macro, usando BIC apenas para caracteres alfabéticos

```
@ @ Programa Assembler para converter uma string
em @ todas as letras maiúsculas. Assume apenas
caracteres alfabéticos @. Usa bit clear cegamente sem @
verificar se o caractere é alfabético ou não. @ @ R0 -
endereço da string de entrada @ R1 - endereço da string
de saída @ R2 - string de saída original para comprimento
calc.
```

```
@ R3 - caractere atual sendo processado @
```

```
.global_start @ Forneça o endereço inicial do programa

.MACRO toupper inputstr, outputstr LDR R0,
 =\inputstr LDR R1, =\outputstr MOV@ início da string de entrada
 R2, R1 @ endereço da string de saída
```

## Capítulo 13 Instruções Condicionais e Código de Otimização

@ O loop é até que o byte apontado por R1 seja um loop diferente de zero:

LDRB R3, [R0], #1 @ load character and incr

BIC R3, #0x20 @ elimina o bit de letras minúsculas

STRB R3, [R1], #1 @ armazenar personagem

CMP R3, #0 @ pare ao atingir um @ loop nulo se o caractere não

Círculo BNE for nulo

SUB R0, R1, R2 @ obtém o comprimento subtraindo

.ENDM

\_início:

superior instr, outstr

@ Configure os parâmetros para imprimir nosso número hexadecimal

@ e depois chame o Linux para fazer isso.

MOV R2,R0 @ código de retorno é a string len

MOV R0, #1 @ 1 = StdOut

LDR R1, =outstr @ string para imprimir

MOV R7, #4 @ linux write system call

SVC 0 @ Chame o linux para imprimir a string

@ Defina os parâmetros para sair do programa @ e depois

chame o Linux para fazer isso.

MOV R0, #0 @ Use 0 código de retorno R7, #1 @

MOV código de comando de saída permanente para chame o

SVC 0

.dados

instr: .asciz "ThisIsRatherALargeVariableNameAaZz@[\n" .align 4 outstr: .fill 255, 1,

0

## Capítulo 13 Instruções Condicionais e Código de Otimização

Esse arquivo contém o ponto de entrada \_start e as chamadas de impressão do Linux, portanto, nenhum main.s é necessário. Aqui está o resultado da construção e execução desta versão:

```
pi@raspberrypi:~/asm/Chapter 13 $ make as -mfpu=neon-
vfpv4 upper3.s -o upper3.o ld -o upper3 upper3.o pi@raspberrypi:~/
asm/Chapter 13 $./upper3
THISISRATHERALARGEVARIABLENAMEAAZZ@
[@[pi@raspberrypi:~/asm/Capítulo 13 $
```

Existem alguns caracteres especiais no final da string para mostrar como alguns são convertidos corretamente e outros não.

Além de usar esta instrução **BIC** para eliminar todo o processamento condicional, implementamos a rotina superior como uma macro para eliminar a sobrecarga de chamar uma função. Mudamos o uso do registrador, então usamos apenas os quatro primeiros registradores na macro, então não precisamos salvar nenhum registrador durante a chamada.

Esta é uma rotina de conversão rápida e suja, mostrando como podemos salvar instruções se restringirmos nosso domínio de problema, neste caso, trabalhando apenas em caracteres alfabéticos em vez de todos os caracteres ASCII.

## Usando Paralelismo com SIMD

No Capítulo 12, “Coprocessador NEON”, examinamos a execução de operações em paralelo e mencionamos que esse coprocessador pode processar caracteres, bem como números inteiros e flutuantes. Vamos ver se podemos usar as instruções NEON para processar 16 caracteres por vez - 16 caracteres cabem em um registrador **Q**.

Primeiro vamos ver o código em upper4.s mostrado na Listagem 13-5.

Nota Este código não será executado até que façamos um ajuste em main.s no final desta seção na Listagem 13-6.

## Capítulo 13 Instruções Condicionais e Código de Otimização

**Listagem 13-5.** Rotina de letras maiúsculas usando o coprocessador NEON

```

@ Programa Assembler para converter uma string em @
todas as letras maiúsculas. @

@ R0 - endereço da string de entrada
@ R1 - endereço da string de saída
@ Q0 - 8 caracteres a serem processados
@ Q1 - contém todos os a's para comparação
@ Q2 - resultado da comparação com 'a's
@ Q3 - todos os 25 para comp
@ Q8 - espaços para operação bic

.topper global @ Permitir que outros arquivos chamem
.EQU N, 4

superior:
 LDR R3, =aaas
 VLDM R3, {Q1} @ Carregar Q1 com todos os a's
 LDR R3, =endch
 VLDM R3, {Q3} @ Carregar Q3 com todos os 25's
 LDR R3, =espaços
 VLDM R3, {Q8} @ Carregue Q8 com todos os espaços
 MOV R3, #N

@ O loop é até que o byte apontado por R1 seja diferente de zero: VLDM R0!, {Q0} @ carrega 16 caracteres e armazena @ Q0 para Q1 @ Subtração 2Q2, Q2,
 VSUB.U8 Q2, Q8 @ e resultado com espaços Q0, Q0, Q2 @
 VCLE.U8 eliminam o bit que o torna mais baixo
 VAND.U8
 VBIC.U8

caso
 VSTM R1!, {Q0} @ armazena caractere para saída str

```

## Capítulo 13 Instruções Condicionais e Código de Otimização

```

SUBS R3, #1 @ @ decrementar contador de loop e definir flags
Círculo BNE loop se o caractere não for nulo
MOV R0, #(Nj16) @ Defina o comprimento
BX LR @ Retornar ao chamador

.dados
aaas: .fill 16, 1, 'a' endch: .fill 16, 1, @16as
25 @ após o turno os caracteres são 0-25 espaços: .fill 16, 1, 0x20 @ espaços
para bic .align 4

```

Essa rotina usa os registradores **Q** para processar 16 caracteres por vez.

Há mais instruções do que algumas de nossas rotinas anteriores, mas o paralelismo faz valer a pena. Começamos carregando nossas constantes nos registradores. Você não pode usar constantes imediatas com instruções NEON, então elas devem estar em registradores. Além disso, eles precisam ser duplicados 16 vezes, portanto, há um para cada uma de nossas 16 pistas.

Em seguida, carregamos 16 caracteres para processar em **Q0** com um **VLDM** instrução.

Observe o ! executa um write-back para mover o ponteiro para o próximo conjunto de caracteres para quando fizermos um loop.

A Figura 13-1 mostra o processamento por meio do coprocessador NEON para as quatro primeiras pistas. Usamos o VBIC, mas poderíamos ter usado o **VSUB** com a mesma facilidade para fazer a conversão. Testamos se o caractere é minúsculo alfabético antes de fazer isso, então é correto para todos os caracteres ASCII.

## Capítulo 13 Instruções Condicionais e Código de Otimização

| Lane                    | 1    | 2    | 3    | 4         |
|-------------------------|------|------|------|-----------|
| Q0                      | T    | h    | i    | s         |
| Q1                      | a    | a    | a    | a         |
| VSUB.U8      Q2, Q0, Q1 |      |      |      |           |
| Q2                      | T-a  | h-a  | i-a  | s-a       |
| Q3                      | 25   | 25   | 25   | 25        |
| VCLE.U8      Q2, Q2, Q3 |      |      |      | 'z' - 'a' |
| Q2                      | 0    | 0xff | 0xff | 0xff      |
| Q8                      | 0x20 | 0x20 | 0x20 | 0x20      |
| VAND.U8      Q2, Q2, Q8 |      |      |      | Spaces    |
| Q2                      | 0    | 0x20 | 0x20 | 0x20      |
| VBIC.U8      Q0, Q0, Q2 |      |      |      |           |
| Q0                      | T    | H    | I    | S         |

**Figura 13-1.** As etapas de processamento paralelo para converter em letras maiúsculas

O **VCLE** é nosso primeiro encontro com uma instrução de comparação NEON.

Ele compara todas as 16 pistas de uma só vez. Ele coloca todos os 1s na faixa de destino se a comparação for verdadeira, caso contrário, 0. Todos os 1s são 0xFF hex. Isso é conveniente, pois podemos **VAND** com um registro cheio de 0x20s. Quaisquer pistas que não tenham um caractere alfabético minúsculo resultarão em zero.

Isso significa pistas com 0, não há bits para **VBIC** limpar. Então o as pistas que ainda têm 0x20 limparão esse 1 bit fazendo a conversão.

Para que essa rotina funcione, precisamos fazer uma alteração em main.s. Nós precisamos para adicionar um **.align 4** entre as duas strings. Isso ocorre porque só podemos carregar ou armazenar dados NEON em locais de memória alinhados por palavras. Se não fizermos isso, obteremos um "Bus Error" quando o programa for executado. O código atualizado é mostrado na Listagem 13-6.

**Listagem 13-6.** Alterações necessárias em main.s

```
instr: .asciz "Esta é nossa string de teste que iremos converter.
```

```
AaZz@["{\n"
```

```
.align 4 .fill
```

```
outstr: 255, 1, 0
```

## Capítulo 13 Instruções Condicionais e Código de Otimização

Eu também adicionei caracteres de maiúsculas e minúsculas ao final da string; isso garante não temos nenhum erro off-by-one em nosso código.

Esse código funciona bem, mas isso se deve em parte à forma como nossa seção .data está configurada. Observe que não há teste para o terminador NULL da string. Essa rotina apenas converte strings de comprimento fixo e definimos o comprimento fixo em 4×16 fazendo com que o loop execute quatro iterações. O processador NEON não tem uma maneira fácil de detectar um terminador NULL. Se percorrermos os caracteres fora do processador NEON para procurar o NULL, faremos quase tanto trabalho quanto nossa última rotina superior. Se formos fazer o processamento de strings no coprocessador NEON, seguem algumas observações:

- Não use strings terminadas em NULL. Use um campo de comprimento seguido pela string. Ou use strings de comprimento fixo, por exemplo, cada string tem apenas 256 caracteres e contém espaços além do último caractere.
- Preencha todas as strings para que usem o armazenamento de dados em múltiplos de 16. Dessa forma, você nunca terá que se preocupar com o processamento do NEON após o fim do seu buffer.
- Certifique-se de que todas as strings estejam alinhadas com as palavras.

## Resumo

Neste capítulo, vimos como adicionar códigos de condição a qualquer instrução e por que e quando devemos fazer isso. Observamos que isso não é suportado no conjunto de instruções ARM de 64 bits e que os pipelines de processador ARM mais recentes tornam essa técnica menos útil do que era nos primeiros dias do ARM.

Em seguida, realizamos várias otimizações em nossa função toupper. Nós procurou simplificar as comparações de intervalo, usando instruções condicionais, manipulações de bits e, finalmente, o coprocessador NEON.

No Capítulo 14, “Lendo e compreendendo o código”, examinaremos como o compilador C gera código e falar sobre a compreensão de programas compilados.

## CAPÍTULO 14

# Lendo e Compreendendo o código

Agora aprendemos um pouco da linguagem Assembly ARM de 32 bits; uma das coisas que podemos fazer é ler o código de outro programador. Ler o código de outro programador é uma ótima maneira de adicionar dicas e truques ao nosso kit de ferramentas e melhorar nossa própria codificação. Analisaremos alguns lugares onde você pode encontrar o código-fonte Assembly para o ARM32. Em seguida, veremos como o compilador GNU C escreve o código Assembly e como podemos analisá-lo. Veremos a ferramenta de hacking Ghidra da NSA, que pode converter o código Assembly de volta para o código C — pelo menos aproximadamente.

Usaremos nosso programa em letras maiúsculas para ver como o compilador C escreve o código Assembly e, em seguida, examinaremos como Ghidra pode pegar esse código e reconstituir o código C. Também veremos como o compilador C lida com a falta de uma instrução de divisão inteira em processadores ARM mais antigos.

## Raspbian e GCC

Uma das muitas coisas boas sobre como trabalhar com Raspberry Pi e GNU Compiler Collection é que eles são de código aberto. Isso significa que você pode navegar pelo código-fonte e examinar as peças de montagem contidas nele.

## Capítulo 14 Lendo e entendendo o código

Eles estão disponíveis nos seguintes repositórios do Github:

- **Kernel Linux Raspbian:** <https://github.com/raspberrypi/linux>
- **Código-fonte do GCC:** <https://github.com/gcc-mirror/gcc>

Clicando no botão “Clone or download” e escolhendo “Download ZIP” é a maneira mais fácil de obtê-lo. Dentro de todo esse código-fonte, algumas boas pastas para examinar o código-fonte do ARM 32-bit Assembly são

- Núcleo Linux Raspbian:

- arco/braço/comum
- arco/braço/núcleo
- arco/braço/cripto

- GCC:

- libgcc/config/arm

---

**Nota** O arch/arm/crypto possui várias rotinas criptográficas implementadas no coprocessador NEON.

O código-fonte do Assembly para eles está em arquivos .S (observe o S maiúsculo). O Raspbian é baseado no Debian Linux. Tanto o Debian Linux quanto o GCC suportam dezenas de arquiteturas de processador, portanto, ao procurar pelo código-fonte do Assembly, certifique-se de procurar os arquivos .S em uma pasta arm. Se estiver interessado, você pode comparar os arquivos Assembly ARM de 32 bits com os arquivos de outros processadores.

O código-fonte deles usa as diretivas do GNU Assembler, como .MACRO , e as diretivas do pré-processador C, como #define e #ifdef.

Se você for ler este código-fonte, será útil revisar o C pré-processador.

O compilador GNU suporta processadores ARM mais antigos do que os contidos em qualquer Raspberry Pi, bem como configurações do processador ARM que a fundação Raspberry nunca usou. Por exemplo, existe uma biblioteca para implementar ponto flutuante IEEE 754 para processadores ARM sem FPU. No entanto, todos os Raspberry Pis têm um FPU, então isso não é usado.

---

## Divisão Revisitada

No Capítulo 10, “Multiplicar, Dividir e Acumular”, presumimos que tínhamos um Raspberry Pi mais recente e usamos as instruções **SDIV** ou **UDIV** do processador ARM mais recente. Acabamos de deixar um comentário que, se você quiser dividir no Pi mais antigo, use o FPU ou role o seu próprio. Nós nunca cobrimos como rolar o nosso próprio. Outra abordagem é ver o que o compilador C faz. Considere a Listagem 14-1, o programa C simples.

**Listagem 14-1.** Programa C simples que divide dois números

```
#include <stdio.h>

int principal()
{
 int x = 100; int
 y = 25; int z;

 z = x/y;
 printf("%d / %d = %d\n", x, y, z);

 retornar(0);
}
```

Podemos compilar isso com

```
gcc -o div div.c
```

Capítulo 14 Lendo e entendendo o código

---

**Observação** Não podemos usar nenhuma das opções de sinalizador -O, porque qualquer otimização removerá a expressão e o compilador apenas conectará 4 para z.

---

Podemos olhar para o código Assembly gerado com

```
objdump -d div
```

Como não compilamos com a opção -O, há muito código, mas no meio da rotina principal, vemos

```
10454: e51b100c ldr r1, [fp, #-12] 10458: e51b0008
ldr r0, [fp, #-8] 1045c: eb00000b bl 10490 <__divsi3>
10460: e1a03000 mov r3, r0
```

que configura e chama uma rotina de divisão chamada `_divsi3`. O Assembly para a rotina `_divsi3` também está presente na saída do objdump. É muito longo e contém código como

```
104e0: e1530f81 cmp r3, r1, lsl #31 104e4: e0a00000
adc r0, r0, r0 104e8: 20433f81 subcs r3, r3, r1, lsl #31
```

e repetido 32 vezes. O que está acontecendo aqui? Como podemos baixar o código-fonte do `gcc` e todas as suas bibliotecas, podemos ver o código-fonte. Se procurarmos a definição de `_divsi3`, a encontraremos em `libgcc/config/arm/lib1funcs.S`. Este código-fonte é confuso, pois contém versões de suas rotinas para diferentes gerações de ARM, além de possuir versões que utilizam o código thumb. Abordaremos o código thumb no Capítulo 15, “Código Thumb”, mas até então podemos ignorar essas partes.

A Listagem 14-2 é a parte principal da rotina de divisão.

**Listagem 14-2.** Parte principal da rotina de divisão gcclib

```

ARM_FUNC_START divsi3
ARM_FUNC_ALIAS aeabi_idiv divsi3

 cmp r1, #0 beq
 LSYM(Ldiv0)
LSYM(divsi3_skip_div0_test): eor
 ip, r0, r1 @ salva o sinal do resultado. do_it mi rsbmi r1, r1, #0
 @ loops abaixo usam unsigned. subs r2, r1, #1 @ divisão por
 1 ou -1 ? beq 10f movs r3, r0 do_it mi rsbmi r3, r0, #0 cmp r3,
 r1 bls 11f

 @ valor de dividendo positivo

```

```

tst r1, r2 beq @ divisor é potência de 2?
12f

ARM_DIV_BODY r3, r1, r0, r2

 cmp ip, #0 do_it
 mi rsbmi r0, r0,
 #0
 RET

```

A rotina começa verificando a divisão por 0, o que é um erro. Em seguida, ele procura os casos fáceis de divisão por 1 ou –1, depois os outros casos de divisão por uma potência de 2. Ele também salva os bits de sinal para que a resposta possa ser definida corretamente no final.

Há muitas macros usadas neste código. A Listagem 14-3 é aquela que gera a divisão real é ARM\_DIV\_BODY.

## Capítulo 14 Lendo e entendendo o código

**Listagem 14-3.** Corpo principal da rotina de divisão

```
.macro ARM_DIV_BODY dividendo, divisor, resultado, limite
 clz \curbit, \dividendo clz \resultado,
 \divisor sub \curbit, \resultado,
 \curbit rsbs \curbit, \curbit, #31 addne
 \curbit, \curbit, \curbit, lsl #1 mov \resultado,
 #0 addne pc, pc, \curbit, lsl #2

 nop .set shift,
 32 .rept 32 .set
 shift, shift - 1 cmp \dividendo,
 \divisor, lsl #shift adc \resultado, \resultado, \resultado
 subcs \dividendo, \dividendo, \divisor, lsl #shift .endr

.endm
```

Dentro desta macro está

```
.set shift, 32 .rept
 32 .set
 shift, shift - 1 cmp \dividendo,
 \divisor, lsl #shift adc \resultado, \resultado, \resultado
 subcs \dividendo, \dividendo, \divisor, lsl #shift .endr
```

que gera o código repetitivo que vemos. Essa é uma forma de otimização chamada **desenrolamento de loop**, em que, se um loop for executado um número fixo de vezes, apenas duplicaremos o código tantas vezes. Isso nos economiza uma instrução de desvio cara, bem como a aritmética calculando o índice de loop.

## Capítulo 14 Lendo e entendendo o código

A divisão será usada com frequência suficiente para que desejemos o código o mais rápido possível e podemos poupar o espaço de código extra para conseguir isso.

O algoritmo para esta divisão é basicamente o mesmo algoritmo de divisão longa que você aprendeu na escola primária. É um pouco mais simples em binário, pois só pode haver duas respostas em cada etapa, colocar 1 no resultado ou não.

---

**Observação** Se incluíssemos a opção de compilador `-march=“armv8-a”`, o compilador usaria uma instrução **SDIV** em vez dessa chamada de função. O GCC usará recursos ARM avançados se souber que estão disponíveis.

---

Infelizmente, o código-fonte do Assembly contido no **gcc** e no Linux nem sempre está tão bem documentado quanto gostaríamos, mas nos dá bastante código-fonte para refletir e aprender.

Você pode querer olhar para `ieee754-sf.S` e `ieee754-df.S` na mesma pasta que `lib1funcs.S`, `gcc/libgcc/config/arm`. Estas são as implementações de ponto flutuante em precisão simples e dupla para processadores ARM que não possuem FPU. É interessante ver todo o trabalho que a FPU faz por nós.

## Código Criado por GCC

Na última seção, vimos alguns códigos gerados pelo **gcc** para ver como ele lida com a falta de uma instrução **SDIV**. Vamos ver como o **gcc** escreveria nosso código. Codificaremos nossa rotina de letras maiúsculas em C e compararemos o código gerado com o que escrevemos. Para este exemplo, queremos que o **gcc** faça o melhor trabalho possível, então usaremos a opção `-O3` para obter otimização máxima.

Criamos `upper.c` da Listagem 14-4.

## Capítulo 14 Lendo e entendendo o código

**Listagem 14-4.** Implementação C da nossa rotina mytoupper

```
#include <stdio.h>

int mytoupper(char *instr, char *outstr) {

 char cur;
 char *orig_outstr = outstr;

 fazer
 {
 cur = *instr; if
 ((cur >= 'a') && (cur <='z')) {

 cur = cur - ('a'-'A');

 } *outstr++ = cur;
 instr++; } while (cur !
 = '\0'); return(outstr -
 orig_outstr);
 }

#define BUFFERSIZE 250

char *tstStr = "Isto é um teste!"; char
outStr[BUFFERSIZE];

int principal()
{
 mytoupper(tstStr, outStr);
 printf("Entrada: %s\nSaída: %s\n", tstStr, outStr);

 retornar(0);
}
```

## Capítulo 14 Lendo e entendendo o código

Podemos compilar isso com

```
gcc -O3 -o superior superior.c
```

então execute objdump para ver o código gerado

```
objdump -d upper >od.txt
```

Obtemos a Listagem 14-5.

**Listagem 14-5.** Código assembly gerado pelo compilador C para nossa função maiúscula

00010318 <main>:

```
[pc, #72] ; 1036r2<[pair#72]54103681<main+0x58>: d103620e59204816rp3sh
e1a02001 mov r2, r1 1032c: e4e2440011d[12]4,0824#45923300e28330028:
add r3, r3, #1 10334: e2440061 sub r0, r4, #97 10338: e3500019 cmp r0,
95430001 strbts r0, [r3, #-4] 1034259a0f8816: e24400320<main+0x14>#43203880:
e3540000 cmp r4, #0 1034c: e5103201strbts[12,>#2][1035061aff056one
<main+0x54> 10354: e59f0210 ldr r0, [pc, #16] ; 10370 <main+0x58>
10358: e59f0010 ldr 1035c: ebfffffe10,102f8,puint0@>e80360101@00004
; 0x61
```

## Capítulo 14 Lendo e entendendo o código

```
10368: 00021028 .word 0x00021028 1036c:
00021030 .word 0x00021030 10370:
0001050c .word 0x0001050c
```

Algumas coisas a serem observadas sobre esta listagem são as seguintes:

- O compilador incorporou automaticamente a função mytopper como nossa versão macro.
- O compilador sabe sobre a otimização do intervalo e mudou o intervalo, então ele só faz uma comparação.
- O compilador fez bom uso dos registradores e não criou um quadro de pilha. Ele usa apenas cinco registradores, então só precisa apertar/pop R4.
- O compilador sabe como usar condicional instruções.
- O compilador adotou uma abordagem ligeiramente diferente para adicionando o condicional, colocando-o em uma instrução de armazenamento, para que o caractere convertido seja armazenado apenas se o caractere for minúsculo. Em seguida, ele pula para o loop, pois sabe que se estiver em minúsculas, não pode ser NULL. Caso contrário, ele falha, armazena o caractere não convertido, verifica NULL e faz um loop se não for.

No geral, o compilador fez um bom trabalho ao compilar nosso código, apenas obtendo algumas instruções extras sobre o que escrevemos no último capítulo. O GCC oferece suporte ao processador ARM há 20 anos. ARM Holdings fez grandes contribuições ao GCC para melhorar o suporte ARM. Todo o trabalho ao longo desse tempo resultou em um sistema robusto e de alto desempenho, e a melhor parte é que ele é totalmente de código aberto.

## Capítulo 14 Lendo e entendendo o código

É por isso que muitos programadores da linguagem Assembly começam com o código C e só recodificam em Assembly se o código C não for eficiente. Isso geralmente acontece quando a complexidade é maior e a necessidade de velocidade é maior, como o código no **gcclib** para aritmética e divisão de ponto flutuante, onde a velocidade é crucial e o Assembler puro é melhor em manipulações de nível de bit do que C.

No Capítulo 8, “Programação de pinos GPIO”, examinamos a programação os pinos GPIO usando os registros de memória do controlador GPIO. Esse tipo de código irá confundir o otimizador. Muitas vezes ele precisa ser desligado, ou otimiza o código que acessa esses locais. Isso ocorre porque escrevemos em locais de memória e nunca os lemos e lemos a memória que nunca configuramos. Existem palavras-chave para ajudar o otimizador, mas no final, o Assembler pode resultar em um código um pouco melhor, porque você está trabalhando contra o otimizador C, que não sabe o que o controlador GPIO está fazendo com esta memória.

## Engenharia reversa e Ghidra

No mundo Raspbian, a maioria dos programas que você encontra são de código aberto que você pode facilmente baixar o código-fonte e estudá-lo. Há documentação sobre como funciona e você é ativamente encorajado a contribuir com o programa, talvez corrigir bugs ou adicionar um novo recurso.

Suponha que encontramos um programa para o qual não temos o código-fonte e queremos saber como ele funciona. Talvez queiramos estudá-lo para ver se contém malware. Pode ser que estejamos preocupados com questões de privacidade e queiramos saber quais informações o programa envia na Internet. Talvez seja um jogo e queremos saber se existe um código secreto que podemos inserir para entrar no modo Deus. Qual é a melhor maneira de fazer isso?

Podemos examinar o código Assembly de qualquer executável do Linux usando **objdump** ou **gdb**. Sabemos o suficiente sobre Assembly para entender as instruções que encontramos. No entanto, isso não nos ajuda a formar uma visão geral de como o programa está estruturado e é demorado.

## Capítulo 14 Lendo e entendendo o código

Existem ferramentas para ajudar nisso. Até recentemente, havia apenas produtos comerciais caros disponíveis; no entanto, a NSA, sim, aquela NSA, lançou uma versão da ferramenta que seus hackers usam para analisar o código. Chama-se Ghidra, em homenagem ao monstro de três cabeças que Godzilla luta. Essa ferramenta permite analisar programas compilados e inclui a capacidade de descompilar um programa de volta ao código C. Ele inclui ferramentas para mostrar os gráficos das chamadas de função e a capacidade de fazer anotações à medida que você aprende as coisas.

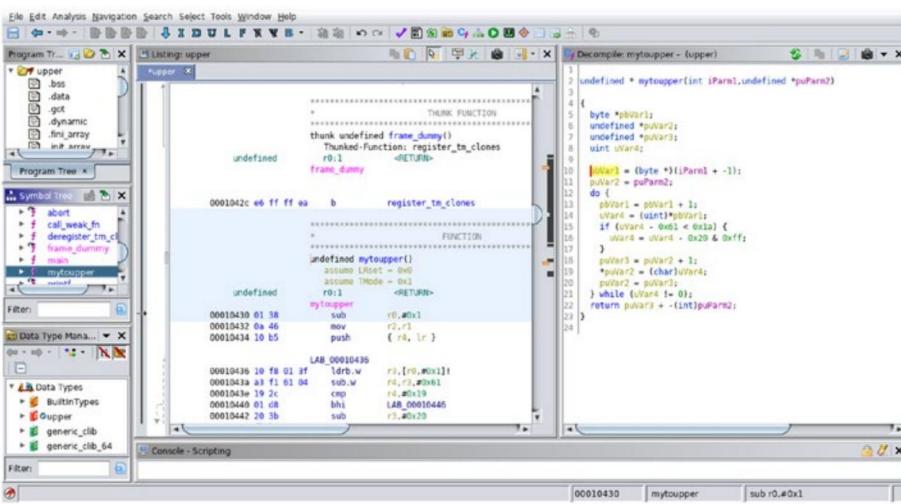
Infelizmente, o Ghidra não funciona mais corretamente no Raspberry Pi, embora seja escrito em Java. A NSA afirma que o Ghidra não será mais compatível com sistemas operacionais de 32 bits. No entanto, o Ghidra ainda suporta a análise de programas de 32 bits. Ele também tem suporte total para o processador ARM. Isso significa que precisamos transferir nosso arquivo executável para um computador com sistema operacional de 64 bits, seja Linux, macOS ou Windows.

Você pode baixar o Ghidra em <https://ghidra-sre.org/>. Para instalá-lo, descompacte-o e execute o script **ghidraRun** se estiver no Linux. Ghidra requer o Java runtime; se você ainda não o tiver instalado, precisará instalá-lo para o seu sistema operacional.

Descompilar um programa C otimizado é difícil. Como vimos na última seção, o otimizador GCC faz algumas reescritas importantes de nosso código original como parte da conversão para linguagem Assembly. Vamos pegar o programa superior que compilamos de C na última seção, entregá-lo a Ghidra para descompilar e ver se o resultado é como nosso código-fonte inicial.

Se criarmos um projeto no Ghidra, importemos nosso programa superior e execute No navegador de código, obtemos a janela mostrada na Figura 14-1.

## Capítulo 14 Lendo e entendendo o código



**Figura 14-1.** Ghidra analisando nosso programa superior

A Listagem 14-6 é o código C gerado por Ghidra. Eu adicionei as linhas acima a definição da rotina principal, para que o programa seja compilado e executado.

**Listagem 14-6.** Código C criado por Ghidra para nosso programa C superior

```

#include <stdio.h>

#define BUFFERSIZE 250

char *tstStr = "Isto é um teste!"; char
outStr[BUFFERSIZE];

typedef unsigned int uint; typedef
byte de caractere não assinado;
typedef void indefinido;

#define verdadeiro 1

```

## Capítulo 14 Lendo e entendendo o código

```

uint principal(void) {

 byte bVar1;
 indefinido *puVar2; byte
 *pbVar3; byte *pbVar4;

 puVar2 = tstStr;
 pbVar3 = tstStr;
 pbVar4 = outStr; faç
 { enquanto
 (verdadeiro) {
 bVar1 = *pbVar3; se
 (0x19 < (uint)bVar1 - 0x61) quebrar; *pbVar4 =
 bVar1 - 0x20; pbVar3 = pbVar3 + 1; pbVar4 =
 pbVar4 + 1;

 } *pbVar4 = bVar1;
 pbVar3 = pbVar3 + 1;
 pbVar4 = pbVar4 + 1; }

 while (bVar1 != 0);
 printf("Entrada: %s\nSaída: %s\n",puVar2,outStr); return
 (uint)bVar1; }

```

Se executarmos o programa, obteremos a saída esperada:

```

pi@raspberrypi:~/asm/Capítulo 14 $ make gcc -O3
-o upperghidra upperghidra.c pi@raspberrypi:~/asm/
Capítulo 14 $./upperghidra Entrada: Isto é um teste!

```

Saída: ISSO É UM TESTE!

```

pi@raspberrypi:~/asm/Capítulo 14 $

```

## Capítulo 14 Lendo e entendendo o código

O código produzido não é bonito. Os nomes das variáveis são gerados. Isto conhece `tstStr` e `outStr`, porque são variáveis globais. A lógica está em etapas menores, geralmente cada instrução C sendo equivalente a uma única instrução Assembly. Ao tentar descobrir um programa para o qual você não possui o código-fonte, ter alguns pontos de vista diferentes é uma grande ajuda.

---

**Observação** Esta técnica funciona apenas para linguagens compiladas verdadeiras, como C, Fortran ou C++. Não funciona para linguagens interpretadas como Python ou JavaScript; também não funciona para linguagens parcialmente compiladas que usam uma arquitetura de máquina virtual como Java ou C#. Existem outras ferramentas para isso e geralmente são muito mais eficazes, já que a etapa de compilação não faz tanto.

---

## Resumo

Neste capítulo, revisamos onde podemos encontrar alguns exemplos de código-fonte Assembly no kernel Raspbian Linux e na biblioteca de tempo de execução GCC. Vimos como o GCC compila o operador de divisão de C e o que acontece quando o processador ARM não suporta uma instrução de divisão. Escrevemos uma versão C do nosso programa em letras maiúsculas, para podermos comparar o código Assembly que o compilador C produz e compará-lo com o que escrevemos.

Em seguida, examinamos o sofisticado programa Ghidra para descompilar programas para reverter o processo e ver o que ele produz. Embora produza código C funcional a partir do código Assembly, não é tão fácil de ler.

No Capítulo 15, “Código Thumb”, veremos o código Thumb, onde reduzir o tamanho da instrução Assembly de 32 bits para 16 bits.

## CAPÍTULO 15

# Código Thumb

O código Assembly que desenvolvemos produz um código compacto em relação às linguagens de alto nível por não precisar de tempo de execução e cada instrução ocupar apenas 32 bits. No entanto, nos primeiros dias do processador ARM, houve muitas reclamações de que ele era muito grande. As pessoas usavam ARMs em pequenos dispositivos incorporáveis com RAM muito limitada e precisavam de programas mais compactos. Outros criaram sistemas com um barramento de memória de 16 bits que permitia 64K de memória - minúsculo para os padrões atuais e levava dois ciclos de memória para carregar cada instrução de 32 bits, tornando o processo mais lento.

processador.

A ARM levou essas preocupações e aplicações a sério e desenvolveu uma versão de 16 bits do conjunto de instruções, chamada de **código polegar**. O código polegar original foi expandido e veremos o código Thumb-2 um pouco mais recente disponível no Raspberry Pis. O menor Raspberry Pi tem 512 MB de memória e um barramento de 32 bits. No entanto, há muitos códigos de polegar por aí; é suportado pelo **GCC** e fornece programas menores.

O código Thumb é implementado no processador ARM como parte da carga de instruções e decodifica parte do pipeline. O decodificador de instrução ARM converte cada instrução de 16 bits em uma contraparte de 32 bits na CPU, para que a unidade de execução não saiba a diferença.

Neste capítulo, veremos os fundamentos do código Thumb-2, como obtemos instruções úteis de 16 bits e como podemos interoperar entre o Thumb e o código normal.

Capítulo 15 Código Thumb

---

**Observação** No mundo da instrução de 64 bits, não existe um conceito semelhante.

Não há modo Thumb de 32 bits. No mundo da instrução de 64 bits, todas as instruções têm 32 bits, sem exceção.

---

## Formato de instrução de 16 bits

Lutamos com a forma como o ARM empacota as informações em 32 bits, causando problemas ao carregar registradores com valores imediatos; geralmente precisamos de duas instruções para carregar um valor de 32 bits. Isso não vai piorar nas instruções de 16 bits? As grandes economias para reduzir o número de bits de instrução são

- Eliminar instruções condicionais; isso economiza 4 bits.

Existe uma maneira de fazer instruções condicionais em alguns casos usando a instrução **IT**.

- Acesso somente aos oito registros inferiores. Isso reduz cada codificação de registro de 4 para 3 bits.
- Reduzir o número de registradores em uma instrução.
- Reduzir o tamanho das constantes imediatas, geralmente para o que sobrar; pode ser tão pequeno quanto 3 bits.
- Elimine todos os modos de endereçamento pré e pós-indexação.

Você deve fazer isso em instruções separadas.

- O sufixo **S** para dizer se uma instrução atualiza o CPSR é fixo ligado ou desligado.

Vejamos três formas da instrução **ADD** de 16 bits :

- ADICIONA Rd, Rn, #imm  
    @ imm pode ser 0–7
- ADIÇÕES Rd, #imm  
    @ imm pode ser 0–255
- ADICIONA Rd, Rn, Rm

No primeiro exemplo, se adicionarmos um imediato a um registrador e o colocarmos em um registrador de destino separado, restam apenas 3 bits para o código imediato, portanto, ele deve estar no intervalo de 0 a 7.

O segundo exemplo é adicionar um imediato a um registrador; como há um registrador a menos, há mais bits disponíveis para o operando imediato, permitindo que ele fique na faixa de 0 a 255.

Os registradores em todos esses três exemplos devem estar no intervalo **R0–R7**, embora existam formas da instrução **ADD** para adicionar a **SP** e adicionar uma constante imediata a **PC**.

---

Observação Todos os três exemplos têm o sinalizador **S** definido; não é opcional.

---

## Chamando Thumb Code

No Capítulo 4, “Controlando o fluxo do programa”, notamos que o **CPSR** continha um bit que indica se o processador está sendo executado no modo **Thumb**. O processador ARM suporta a execução de alguns códigos no modo Thumb e alguns como as instruções ARM normais de 32 bits que estudamos até agora.

No Capítulo 6, “Funções e pilha”, mencionamos que a instrução **BX** pode alternar entre os estados do processador quando é executada. Se quisermos retornar de uma função escrita com a instrução Thumb para uma que não é, então devemos usar a instrução **BX**; não podemos simplesmente **POP** o endereço de retorno no PC - se o fizermos, obteremos uma exceção de “instrução ilegal”.

Há uma instrução **BLX** correspondente para chamar entre ARM32 e o código Thumb. Ambas as instruções podem ir de qualquer maneira entre as instruções Thumb e ARM32.

Como as instruções **BLX** e **BX** sabem se estão ramificando para o código Thumb ou ARM32? O processador ARM usa um truque. Todas as instruções ARM32 devem ser alinhadas por palavra e todas as instruções Thumb devem ser alinhadas a um limite de 16 bits. Isso significa que qualquer endereço que aponte para uma instrução deve ser par, o que significa que o bit de ordem inferior não é usado.

## Capítulo 15 Código Thumb

O processador ARM usa o bit de ordem inferior de um endereço de instrução para indicar se o ponteiro é para uma instrução ARM32 ou Thumb.

Isso significa que, se você for chamar **o BLX** para chamar o código Thumb, precisará adicionar um ao endereço. Ao fazer isso, **o LR** será configurado com o endereço correto para que **o BX** faça a coisa certa quando retornar. Isso é meio que um hack, mas o processador ARM trabalha duro para obter a funcionalidade de cada bit.

Isso vale se você passar essas instruções como um registro. Se você usar a forma de **BLX** onde passa uma etiqueta, então **o BLX** sempre mudará de modo, seja de Thumb para ARM32 ou vice-versa. Isso ocorre em parte porque o rótulo é representado por um deslocamento do **PC** em palavras, portanto, o truque par/ímpar não funcionará.

Para ver como o Assembler nos ajuda, considere o seguinte código:

```
@ ARM
Code _start:
I1: LDR R0, =myfunc
 BLX R0
...
.thumb_func
minhafunção:
L2: ADICIONA R2, R1, #2
...
```

O código ARM será compilado como

```
00010054 <_start>:
10054: e59f001c ldr r0, [pc, #28] ; 10078 <L4+0x6> blx r0
10058: e12fff30
...
00010068
<minhafunção>: 10068: 1c8a adciona r2, r1, #2
...
10078: 00010069 .word 0x00010069
```

Vemos que a instrução **LDR** carrega 0x00010069 do local pc+28 (0x10078) que é o endereço de myfunc (00010068) mais 1.

## Thumb-2 tem mais de 16 bits

O conjunto de instruções Thumb original era limitado a instruções de 16 bits, exceto por algumas exceções. A variante Thumb-2 mais recente permite muitas instruções de 32 bits, então você pode fazer muito mais no modo Thumb. Ele também adiciona uma nova instrução de TI que fornece execução condicional limitada.

Dentro do código Thumb, se quisermos forçar uma instrução a ter 32 bits, podemos adicionar um sufixo **.W** , para largura, ou se quisermos forçar a instrução a ter 16 bits, podemos adicionar um sufixo **.N** , para estreita. Ainda existem limitações nessas instruções **.W** em comparação com o que fizemos, como nenhuma instrução condicional sem uma instrução **IT** .

Para habilitar essa sintaxe, iniciamos nosso arquivo de origem com um  
.sintaxe unificada

Diretiva do montador.

Isso informa ao Assembler que este arquivo está usando todos os recursos do Thumb-2. Se quiséssemos apenas as antigas instruções do Thumb-1, iniciariímos o arquivo com uma diretiva **.Thumb** .

## Blocos de TI

O código Thumb não oferece suporte à execução condicional; no entanto, com Thumb-2 foi considerado importante o suficiente para adicionar uma nova instrução If-Then (**IT**) para tornar a seguinte instrução condicional, por exemplo:

```
ISTO
 .equilíbrio
ADDEQ R2, R1
```

## Capítulo 15 Código Thumb

As instruções em Thumb-2 só são códigos de condição permitidos ao seguir uma instrução **IT**, e as condições nas duas instruções deve ser o mesmo.

---

Observação Originalmente, a TI suportava IF-THEN-ELSE e permitia até quatro instruções a seguir. Esta funcionalidade está obsoleta, o que significa que pode não ser suportada nas gerações futuras do processador ARM, portanto não a mencionaremos.

A versão de 16 bits da instrução **ADD** é **ADDS** ou ADD<código de condição>. Outras versões irão gerar uma instrução de 32 bits.

---

## Maiúsculas em Thumb-2

Como tudo isso funciona ficará mais claro com um exemplo. Vamos converter nosso arquivo upper2.s do Capítulo 13, “Instruções condicionais e código de otimização”, para o código Thumb. A maneira como fazemos isso é adicionar as diretivas Assembly ao topo do arquivo. Adicionamos “.syntax unified” e depois “.thumb\_func” após a diretiva .global. A diretiva “.thumb\_func” diz ao Assembler que a seguinte função está no código Thumb, então monte-a de acordo. Ele também lida com os detalhes de alternar entre os modos Thumb-2 e ARM32, para que não precisemos.

Se fizermos isso no upper2.s original e compilarmos, obteremos o erro mensagem

```
pi@raspberrypi:~/asm/Chapter 15 $ make as
-march="armv8-a" -mfpu=neon-vfpv4 upper2.s -o upper2.o upper2.s: Assembler
messages: upper2.s:27: Error: instrução condicional thumb deve estar no bloco IT --
`subls R5,#(97-65)'
```

```
make: *** [makefile:14: upper2.o] Erro 1 pi@raspberrypi:~/asm/Chapter 15 $
```

Isso é esperado, pois sabemos que o código Thumb não suporta execução condicional. Se adicionarmos

TI LS

antes da instrução SUBLS, então ele irá compilar. A Listagem 15-1 é nossa primeira tentativa de código Thumb.

### Listagem 15-1. Nossa primeira tentativa de converter upper2.s em código Thumb

```
@
@ Programa Assembler para converter uma string em @
todas as letras maiúsculas. @

@ R1 - endereço da string de saída @ R0 -
endereço da string de entrada @ R4 - string
de saída original para comprimento calc.
@ R5 - caractere atual sendo processado @ R6 - menos
'a' para comparar < 26. @
```

.sintaxe unificada

.topper global @ Permitir que outros arquivos chamem isso

.thumb\_func

MOV R4, R1 PUSH {R4-R6} superior: @ Salve os registradores

@ O loop é até que o byte apontado por R1 seja um loop diferente de zero:  
LDRB R5, [R0], #1 @ carregar personagem  
@ Quer saber se 'a' <= R5 <= 'z'

## Capítulo 15 Código Thumb

@ Primeiro subtraia 'a'

SUB R6, R5, #'a'

@ Agora quero saber se R6 <= 25 CMP

R6, #25 @ se chegamos @quártieres são 0-25 após o turno  
então a letra é @ minúscula, então converta.

TI LS

SUBL S R5, #('a'-'A')

STRB R5, [R1], #1 @ armazenar personagem

CMP R5, #0 @ pare ao atingir um @ loop nulo se o caractere

Círculo BNE não for nulo

SUB R0, R1, R4 @ obter o comprimento

POP {R4-R6} @ Restaura os registradores

BX LR @ Retornar ao chamador

Temos que fazer uma modificação em main.s; temos que mudar

Topo BL

para

Topper BLX

Como colocamos ".thumb\_func" na frente da chamada de definição, ela será tratada corretamente pelo Assembler.

Agora podemos compilar e executar o programa, então obter o esperado saída

```
pi@raspberrypi:~/asm/Chapter 15 $ make as
-march="armv8-a" -mfpu=neon-vfpv4 upper2.s -o upper2.o ld -o upper2 main.o
upper2.o pi@raspberrypi:~/asm/Chapter 15 $./upper2 ESSA É A NOSSA STRING
DE TESTE QUE CONVERTEREMOS. AAZZ@[`{ pi@raspberrypi:~/asm/Capítulo
15 $
```

Isso foi muito fácil. A Listagem 15-2 é o código Assembly gerado usando objdump.

**Listagem 15-2.** Objdump saída do nosso programa em letras maiúsculas

Desmontagem da seção .text:

00010074 <\_start>:

```
10074: e59f002c ldr r0, [pc, #44] ; 100a8 <_start+0x34> ldr r1, [pc,
10078: e59f102c #44] ; 100ac <_start+0x38> mov r4, #12 mov r5, #13
1007c: e3a0400c blx 100b0 <toupper>
10080: e3a0500d
10084: fa000009
10088: e1a02000 mov r2, r0 1008c:
e3a00001 mov r0, #1 ldr r10 [pac, #20] ;
mov r7, #4 10098: ef000000st10090x38ad10090ne59f001#4 100947,s107004
```

svc 0x00000000

100a0: e3a07001

100a4: ef000000 svc 0x00000000

100a8: 000200e0 .word 0x000200e0

100ac: 00020120 .word 0x00020120

000100b0 <topper>:

100b0: b470 100b2: empurre {r4, r5, r6}

460c mov r4, r1

000100b4 <loop>:

100b8: f1a5 0665, f101, #16,05b4#9310560100bcw

2e19 100be: bf98 100c0: 3d20 subls r5, #32 100c2:f801

5b01 strb.w cmp r6, #25 é ls

r5, [r1], #1

## Capítulo 15 Código Thumb

```

100c6: 2d00 cmp r5, #0
100c8: d1f4 bne.n 100b4 <loop>
100ca: eba1 0004 sub.w r0, r1, r4 100ce: bc70
 pop {r4, r5, r6} bx
100d0: 4770 lr

```

Vemos que o programa principal em `_start` contém código normal de 32 bits.

A única mudança da versão do Capítulo 13 é chamar **BLX** em vez de **BL**. A chamada para **BLX** mudará o processador do modo ARM32 para o modo Thumb.

Se olharmos para a parte superior do programa, veremos que nove instruções têm 16 bits, mas quatro instruções têm 32 bits. Como resultado, economizamos 18 bytes na versão do Capítulo 13, mas parece que podemos fazer melhor.

Existem duas instruções **SUB** de 32 bits; eles parecem simples o suficiente, mas por que eles são 32 bits? A razão é que as instruções **ADD** e **SUB** podem ter o sufixo **S** ou fazer parte de um bloco **IT**. Se adicionarmos o **S** a estas instruções, elas se tornarão 16 bits e não afetarão o funcionamento desta rotina.

As instruções **LDRB** e **STRB** são amplas porque o modo Thumb não oferece suporte a atualizações pós-índice. Temos que movê-los para instruções **ADDS** separadas. O resultado são duas instruções de 16 bits em vez de uma Instrução de 32 bits, então vamos de uma instrução para duas instruções, mas usamos o mesmo espaço. Faremos essa mudança para mostrar que podemos tornar todos os 16 bits superiores. Quando vamos à força

```
SUB R6, R5, #'a'
```

para ser de 16 bits, encontramos o problema de que a constante imediata é limitada a 3 bits, então 'a' não se encaixa. Para contornar isso, adicionamos

```
MOVS R7, #'a'
```

perto do topo e, em vez disso, subtraia **R7**. Como tivemos que dividir esta instrução em duas, não economizamos espaço aqui. O **S** é necessário para manter esta instrução **MOV** de 16 bits.