UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de São José do Rio Preto

Processadores ARM: visão geral e aplicações

Cássio Henrique Volpatto Forte

Grupo de Sistemas Paralelos e Distribuídos(GSPD)
Departamento de Ciência da Computação e Estatística
Universidade Estadual Paulista "Júlio de Mesquita Filho"(UNESP)
São José do Rio Preto - SP - Brasil

Abstract

ARM processors are found in many everyday devices. The success of the architecture is due to several design decisions that led her to adress the needs of a wide range of embedded applications, dedicated or not. This article aims to present an overview of the ARM architecture in order to enable basic understanding of it, and of the reasons that lead to it's success. Some recent applications that use ARM processors are also seen in this article.

Resumo

Processadores ARM estão presentes em diversos dispositivos do cotidiano. O sucesso da arquitetura se deve a diversas decisões de projeto que levaram-na a atender aos critérios impostos por amplo leque de aplicações embarcadas, dedicadas ou não. Este artigo tem por finalidade apresentar uma visão geral da arquitetura ARM, a fim de permitir o entendimento básico da mesma, e dos motivos que levam ao seu sucesso. São vistas também algumas aplicações recentes que utilizam processadores ARM.

São José do Rio Preto
2015

Conteúdo


1 Introdução

Segundo [16], processadores e microcontroladores ARM são baseados em RISC, têm seu projeto focado em baixo consumo de energia e apresentam tamanho reduzido do chip. Essas características tornam a arquitetura ideal para aplicações que sofrem restrições de consumo de energia, e de tamanho dos dispositivos. De fato, processadores ARM podem ser encontrados em grande variedade de aparelhos, e, entre eles, estão muitos que já fazem parte do cotidiano de inúmeros usuários, como *Smartphones*, *Tablets*, televisores, *eReaders*, etc. A evolução da arquitetura desde o seu advento trouxe diversas capacidades ao processador ARM, tornando-o ainda mais utilizado, e fazendo dele objeto de interesse em projetos acadêmicos cada vez mais elaborados. Dessa forma, é de grande interesse o entendimento de suas características básicas.

A história da arquitetura ARM começou em 1980, quando a *Acorn Computer* assinou um contrato com a emissora britânica BBC, para projetar a arquitetura do processador que seria usado no projeto *Computer Literacy Project*. O sucesso do projeto e os lucros obtidos permitiram o projeto do primeiro processador ARM, ARM1, que foi usado posteriormente como coprocessador no computador da BBC e foi sucedido por versões gradativamente melhores. Na época, ARM era a sigla para *Acorn RISC Machine*.

No início, a *Acorn Computer* encomendava a fabricação dos processadores à *VLSI Technology Inc.*, mas o sucesso da arquitetura levou a grandes demandas, que a fabricante não podia atender. Isso levou à necessidade de outro fabricante ou à criação de uma nova empresa. Decidiu-se por uma nova empresa, a ARM Ltd., fundada por *Acorn Computer*, *VLSI Technology* e *Apple Computer*. Foi mantida a opção de não fabricar os processadores, mas licenciar as arquiteturas projetadas para fabricantes de semicondutores. A sigla ARM passou a significar *Advanced RISC Machine*, mas no fim da década de 90, a arquitetura passou a se chamar apenas ARM. Neste artigo, serão vistas as características gerais da arquitetura ARM.

2 Características Gerais da Arquitetura ARM

Apesar das mudanças que ocorrem de um processador para outro, a arquitetura ARM apresenta alguns aspectos gerais, explicados com base em [16] e [13]:

- Processadores ARM possuem 37 registradores, dos quais 30 são de propósito geral. Cada modo de operação do processador utiliza diferentes registradores, fazendo com que apenas 15, dos 30 registradores gerais, estejam visíveis em cada modo. Além disso, dois deles são reservados para ponteiro de pilha e *link register*, que armazena endereços de retorno para quando ocorrem chamadas de função. Os 7 registradores de uso específico incluem contador de programa e registradores de estado, que são especialmente importantes na arquitetura ARM, devido à execução condicional de instruções;

- A maioria das instruções apresenta um campo de 4 bits, em que é definida uma condição para que a operação seja executada, ou seja substituída por uma instrução NOP (*No Operation*), caso a condição não seja atendida. Há 4 *flags* de estado, N,Z,C e V (Negativo, Zero, *Carry* e *Overflow*), que são armazenadas no registrador de estado. Os valores assumidos por essas *flags* definem se as condições das operações foram atendidas. Há, nas instruções de processamento de dados, um bit, chamado bit S, cujo valor define se o resultado da operação altera os valores das *flags*. Essa abordagem influencia no pipeline, como será visto, mas apresenta a vantagem de diminuir a quantidade instruções explícitas de desvio condicional, e contribui

para desempenho e eficiência energética, uma vez que a instrução de desvio condicional não faz processamento útil de fato, apenas altera o fluxo do programa;

- Os modos de execução do processador são 7. O primeiro deles é o modo usuário, que não oferece nenhum tipo de privilégio ao programa em execução. Os outros modos oferecem diferentes opções de privilégios e são usados pelo Sistema Operacional. Oferecer tantos modos de operação permite ao Sistema Operacional fácil adaptação a diferentes cenários e tipos de aplicações, algo fundamental para uma arquitetura que abrange muitos dispositivos. Alguns registradores são de uso exclusivo de modos de execução, o que promove trocas de contexto mais rápidas, uma vez que os valores desses registradores não tem que ser salvos e recuperados para programas que executam em outros modos;

- O mapeamento de endereços da memória *cache* é feito de forma associativa em conjuntos. Até a família de processadores ARM10, a memória *cache* usava endereços lógicos (*cache* lógica), mas hoje usa-se *cache* física. Um aspecto particular da organização de *cache* dos processadores ARM é a existência de um *buffer* de escrita entre a *cache* e a memória principal, ilustrado na figura 1. Esse *buffer* armazena dados que devem ser escritos na memória, enquanto as escritas ocorrem, de forma que, se ele estiver cheio, instruções *Store*, que não sejam de blocos, são suspensas até que haja espaço. O tamanho do *buffer* é reduzido a fim de evitar ou reduzir o prejuízo imposto pela não disponibilidade dos dados que aguardam escrita. A menos que haja muitas escritas, o uso do *buffer* promove uma melhora de desempenho;

- A memória principal é organizada em superseções de 16MB, seções de 1MB, páginas grandes de 64KB e páginas pequenas de 4KB. A fim de acelerar buscas na memória, são mantidas, na memória principal, duas tabelas, L1 e L2, para superseções/seções e páginas grandes/pequenas, respectivamente. O processo de mapeamento para endereço de páginas pequenas está ilustrado na figura 2. Além das divisões em seção e página, existe a divisão em domínios de memória, que consistem em coleções de páginas e seções. Para acessar um domínio, é necessária permissão, e os processos podem ser clientes ou gerentes de um domínio. Ativar ou desativar o acesso a um domínio requer apenas a alteração de dois bits nos registradores de controle de acesso a domínio. A ideia de cliente e gerente de domínio permite o compartilhamento de domínios entre processos sem comprometer totalmente as restrições, que podem ser definidas pelo processo gerente;

- São suportados 7 tipos de exceção para tratar interrupções. Cada tipo de exceção tem uma rotina específica, cujo endereço é fixo. O conjunto de endereços dessas rotina é chamado vetor de exceções. Sempre que ocorre uma interrupção, o contador de programa recebe o endereço da rotina correspondente.

Nas subseções a seguir, são apresentados os aspectos gerais do conjunto de instruções.

2.1 Tipos de instruções

Os tipos de instruções oferecidos são:

- *Load*/*Store*: instruções de carregamento e escrita de dados na memória. Podem ler/escrever palavras inteiras(32 bits), meias-palavras(16 bits), bytes sem sinal, e podem ler e estender o sinal de meias-palavras;

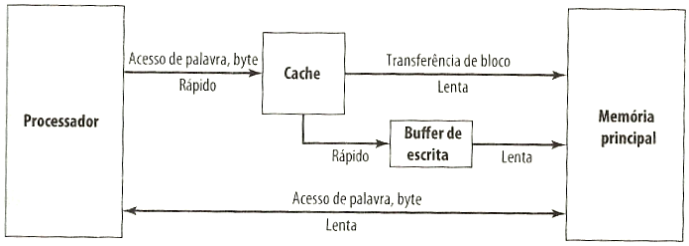

Figura 1: Buffer de escrita[16]

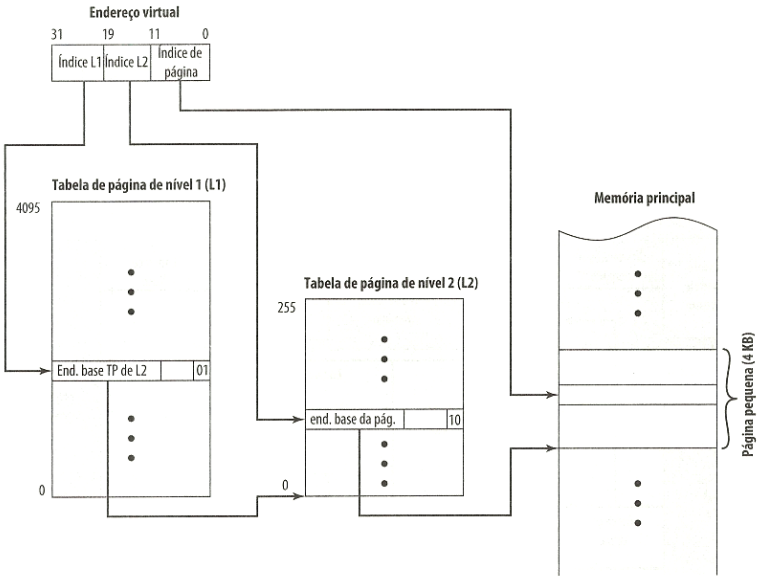

Figura 2: Mapeamento de endereço para páginas pequenas[16]

- *Load/Store* em blocos: são instruções que permitem definir a leitura/escrita de blocos de até 16 registradores em uma única instrução. Isso é feito com uma máscara de 16 bits na instrução que define quais registradores serão usados, e permite atingir uma banda de comunicação quatro vezes maior do que uma instrução que usa um único registrador. Essas instruções podem ser usadas para início e retorno de chamadas, assim como para ler blocos de memória.
- Desvio: permitem desvio condicional de, no máximo 32MB, para frente ou para trás. A condição é dada por um campo de 4 bits;
- Processamento de Dados: são operações de adição e subtração, operações lógicas AND, OR e XOR, e instruções de comparação e teste;
- Multiplicação de inteiros: as multiplicações de inteiros podem ser feitas com operandos de 32 ou 16 bits, e o resultado pode ocupar 32 ou 64 bits;

- Adição/Subtração paralelas: essas instruções permitem aplicar a operação em partes dos operandos, de forma paralela. Um exemplo é a instrução ADD16, que adiciona as primeiras meias-palavras dos operandos para formar a meia palavra superior do resultado, enquanto faz o mesmo com as meias-palavras inferiores para formar a parte inferior do resultado. Essas instruções são úteis para processamento de imagens;
- Extensão: são instruções para reagrupar dados;
- Acesso a registrador de estado: é permitido ler e escrever em partes do registrador de estados;
- Interface para coprocessadores: há um grupo de instruções para trocas de dados com coprocessadores, assim como para controle destes. Todo o tratamento de ponto flutuante é feito por um coprocessador cuja presença nos chips é opcional;
- Intercâmbio com *Thumb*: como será visto adiante, há um subconjunto de instruções de 16 bits chamado *Thumb*. Há instruções para indicar se serão executadas instruções de 32 ou 16 bits.

2.2 Formato das Instruções

Apesar das diferenças entre as instruções, os projetistas mantiveram certa regularidade no formato delas, facilitando a decodificação. O formato geral é, do bit menos significativo ao mais significativo, o seguinte:

- 4 bits para condição de execução;
- 3 bits para o tipo de instrução;
- 5 bits para o *opcode*;
- 20 bits, cujo uso e repartição varia para acomodar endereços, referências a registradores, deslocamentos e rotações.

Maiores detalhes sobre o formato das instruções podem ser vistos na figura 3.

2.3 *Thumb*

Há sistemas embarcados com barramentos de apenas 16 bits, assim como há aqueles em que as restrições de memória exigem a maior densidade de código possível. Para atender a esses tipos de aplicações, parte das instruções ARM foi adaptada para ocupar apenas 16 bits, dando origem ao conjunto *Thumb* de instruções. Uma consequência dessa redução é a possível necessidade de usar duas instruções para realizar um desvio para um endereço de 32 bits. A redução no tamanho das instruções se deve a três alterações:

1. Apenas algumas instruções podem ser executadas condicionalmente, e sua condição está implícita no *opcode*, o que permite a remoção do campo de 4 bits de condição. Sem as condições, o bit S das instruções das instruções de processamento de dados não é necessário;
2. A quantidade reduzida de instruções que foram adaptadas permite a redução do *opcode* em 1 bit, além da redução do campo de tipo em 1 bit também;

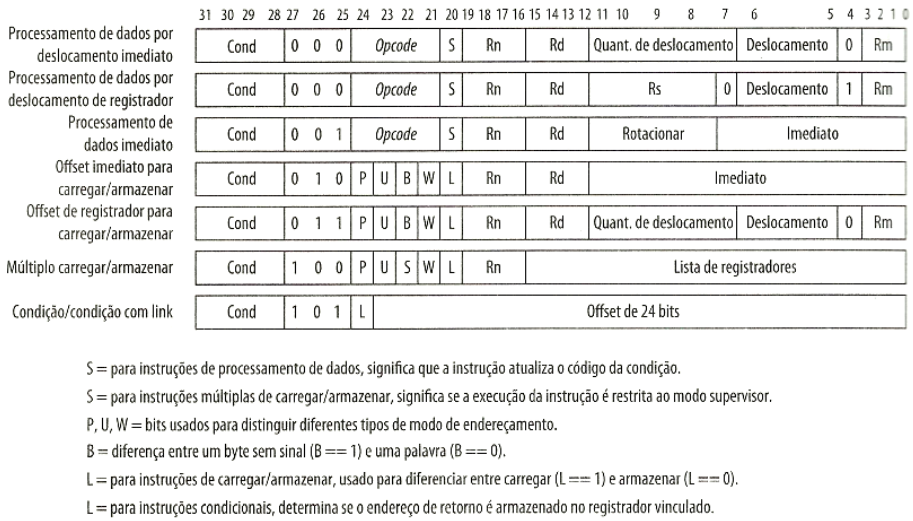

Figura 3: Formato das instruções ARM[16]

3. Apenas 8 registradores podem ser usados por instruções *Thumb*, o que reduz para 3 o número de bits para operandos. O grupo de registradores utilizáveis formam o grupo chamado Lo, enquanto os outros estão no grupo Hi. Dados que estão em registradores Hi devem ser transmitidos para Lo, alterados e devolvidos.

A segunda versão da *Thumb*, chamada *Thumb-2* adiciona instruções 32 bits ao conjunto de instruções, além de outras modificações. Hoje, *Thumb* e *Thumb-2* são referenciadas como T32.

2.4 Modos de Endereçamento

São oferecidos 6 modos de endereçamento. As instruções *Load* e *Store* utilizam o valor armazenado em um registrador base, e um valor *offset* para obter os endereços de memória. Esse procedimento dispõe de três formas de indexação:

- *Offset*: o endereço de memória é formado pela adição do registrador base com o *offset*;
- Pré-indexação: é feito o mesmo procedimento citado acima, mas, o endereço obtido é armazenado no registrador base, resultando em incremento ou decremento do registrador base;
- Pós-indexação: utiliza-se como endereço o valor do registrador base, e, depois, o *offset* e adicionado ao registrador base ou subtraído dele.

As operações de processamento de dados utilizam o modo de endereçamento direto, referenciando diretamente os registradores, e podem combinar o conteúdo dos registradores com valores imediatos. As instruções de desvio contém um campo de 24 bits para o endereço de destino, o que requer o deslocamento para que seja obtido o endereço de 32 bits, e explica a distância máxima de 32MB para os desvios. Por fim, as instruções *Load/Store* em blocos, utilizam um registrador e fazem incremento/decremento antes ou depois, para cobrir a faixa de endereços de memória que será lida ou escrita.


2.5 Suporte a 64 bits - *ARMv8-A*

Segundo [2], a primeira versão da arquitetura ARM a dar suporte a instruções de 64 bits é a *ARMv8-A*, utilizada nos processadores *Cortex-A72*, *Cortex-A57* e *Cortex-A53*. Essa versão adiciona registradores de propósito geral, de 64 bits, assim como utiliza endereçamento virtual estendido e processamento de dados de 64 bits. A compatibilidade com instruções de 32 e 16 bits é mantida, e o processador pode trabalhar em duas possíveis configurações, chamadas *AArch64* e *AArch32*, que incluem, para 64 e 32 bits respectivamente, modos de execução, modos de exceção, e suporte para o conjunto de instruções correspondente. Os conjuntos de instruções suportados são: A32(instruções de 32 bits), T32(*Thumb* e *Thumb-2*) e A64, que inclui as instruções de 64 bits mantendo as características gerais vistas.

3 Famílias de Processadores ARM

Segundo [8] e [4], há quatro grupos de processadores. O primeiro deles contém os processadores clássicos, pertencentes às famílias ARM11, ARM9, e ARM7. Os outros grupos são baseados nos tipos de aplicação em que estão focados seus integrantes. O primeiro tipo de aplicação consiste em dar suporte a sistemas operacionais complexos, como os que ocorrem em *Smartphones* e *Smartphones*, e até mesmo o Linux. Dessa forma, grande variedade de aplicações podem ser executadas, o que define os processadores desse tipo como processadores de aplicação. Processadores *Cortex-A* constituem esse grupo e, para tanto apresentam algumas características:

- Suporte para instruções de 16, 32 e 64 bits(apenas alguns processadores), e apresentam suporte a instruções de criptografia;
- Pipelines de 8 a mais de 15 estágios, com funções de predição de desvios e capacidade de execução fora de ordem, a depender do modelo do processador;
- Suporte para as extensões de aceleração, criptografia, virtualização e coprocessador de ponto flutuante;
- Possibilidade de 1 a 4 *cores* em um único *cluster*, com coerência de *cache* L1 integrada.

Processadores da família *Cortex-R*, são projetados para atender a aplicações de tempo real como sistemas de freio ABS e equipamentos de rede. Processadores *Cortex-M* são voltados para aplicações embarcadas de baixo custo, principalmente microcontroladores, e, para isso, apresentam reduzida quantidade de transistores, e consumo de energia ainda mais reduzido do que os outros processadores ARM. O último grupo de processadores contém os processadores especialistas, que atendem a aplicações específicas como FPGA e aplicações que requerem alto nível de segurança.

4 Pipeline - *ARM Cortex-A8*

Um ponto central para processadores RISC é o pipeline, as instruções de tamanho fixo, a simplicidade das instruções, e o uso de apenas instruções *Load/Store* para lidar com a memória são algumas das características que contribuem para o desempenho e facilidade de construção do pipeline. A arquitetura do pipeline muda de um processador ARM para outro, o que força a escolha

de uma implementação particular para explorar neste artigo. Segundo [16] e [13], o pipeline do *Cortex-A8* é um bom exemplo de pipeline superescalar RISC.

Nas figuras 4, 5 e 6 encontram-se os esquemas do pipeline em foco, e as subseções a seguir tratam das partes do mesmo.

4.1 Busca de instruções

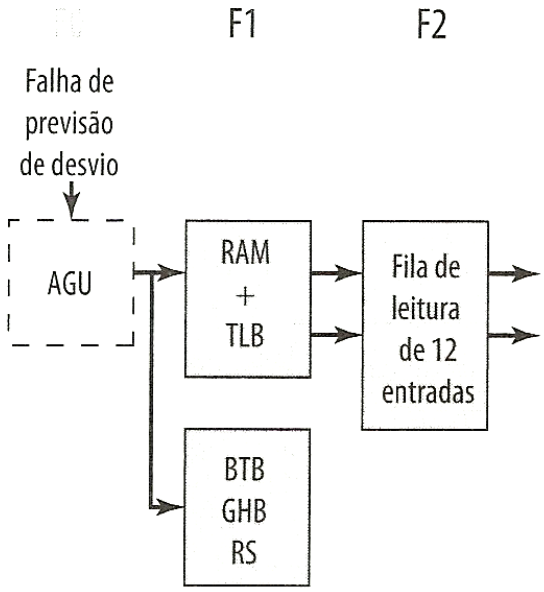

Figura 4: Estágios de busca do pipeline do *ARM Cortex-A8* [16]

A unidade de busca pode ser vista na figura 4. Nos estágios F0, F1 e F2, ocorre a busca de instruções, começando pela previsão de desvio. Em F0, calcula-se o endereço da instrução a ser buscada no nível 1 de memória *cache*. Em paralelo ao cálculo de endereço, verifica-se se, na busca da próxima instrução, a previsão de desvio deve ser utilizada. Em F2, a instrução obtida é colocada em uma fila, que pode ter até 12 instruções, e que envia 2 instruções por vez para a unidade de decodificação. É importante notar que o *buffer* de instruções contem instruções cuja execução não é garantida, dada a possibilidade de desvios, e subsequente esvaziamento do pipeline. A previsão de desvios é feita com duas estruturas, um buffer de alvos de desvio e um buffer de histórico.

4.2 Decodificação de Instruções

A unidade de decodificação pode ser vista na figura 5 e contém dois canais que trabalham em paralelo. O canal 0 contém sempre a instrução anterior a aquela que esta no canal 1, e a execução é sempre feita em ordem, ou seja, a instrução decodificada no canal 1, só será repassada ao pipeline

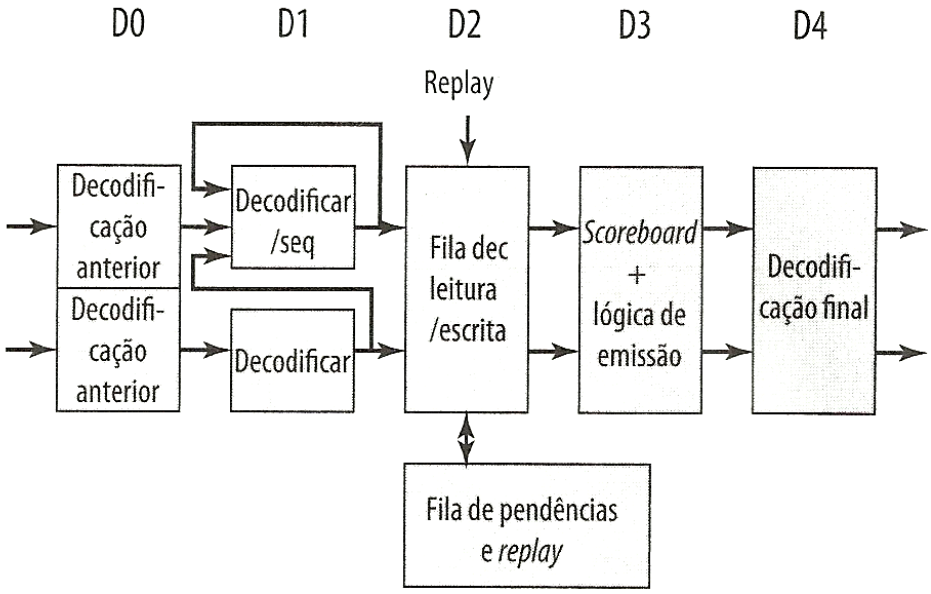

Figura 5: Estágios de decodificação do pipeline do *ARM Cortex-A8*[16]

de execução, se a instrução do canal 0 for executada. Essa abordagem evita problemas do tipo *WAR*, e facilita o tratamento de de *WAW*. Os estágios de decodificação são:

- D0: as instruções *Thumb* são convertidas em instruções de 32 bits, e uma função inicial de decodificação é aplicada;
- D1: operandos de origem e destino são calculados, assim como os requisitos de execução. Além disso, instruções que demoram mais de um ciclo para executar, são divididas em instruções menores;
- D2: instruções são armazenadas em uma fila, ou lidas desta. A fila contém instruções que ainda não puderam ser executadas, ou cuja execução não pode ser completada, devido à falta de um operando, caso em que a instrução sofre *replay*;
- D3: é prevista a disponibilidade de registradores e dos dados de entrada para as instruções decodificadas, assim como são verificados os *hazards* de pipeline;
- D4: decodificação final de sinais de controle para a execução nos canais de leitura/escrita, e execução de inteiros.

O uso da fila de instruções pendentes garante que, caso não haja falhas na previsão desvios, os estágios de execução sempre terão trabalho a fazer. Além disso, a penalidade por atrasos de leitura de operandos da memória fica limitado estágio D0, uma vez que instruções que sofrem *replay* não precisam ser lidas novamente da memória.

4.3 Execução de instruções

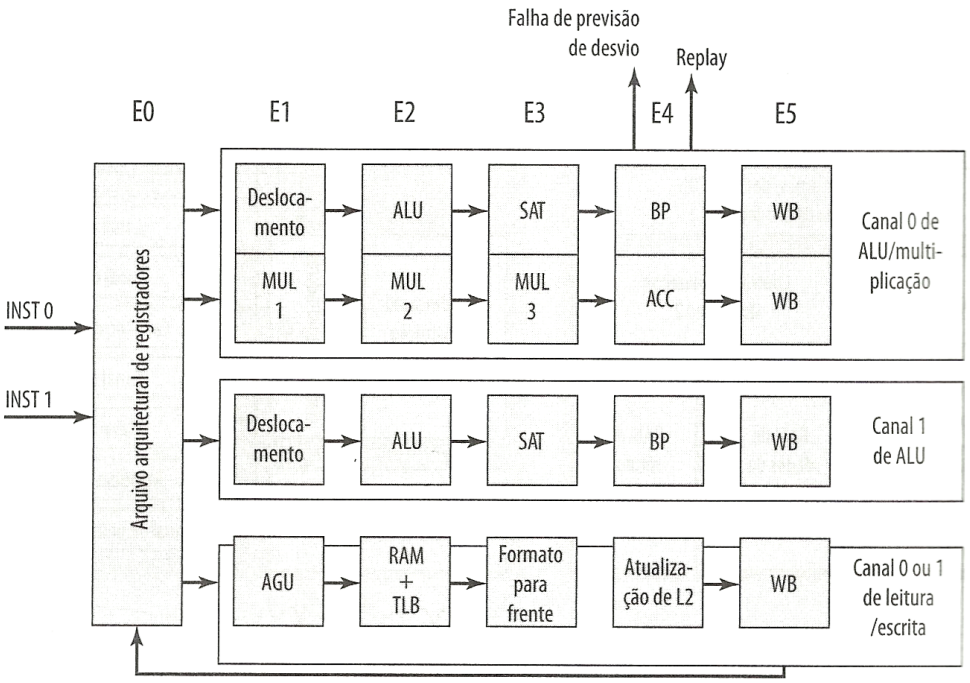

Figura 6: Estágios de execução do pipeline do *ARM Cortex-A8*[16]

A unidade de execução pode ser vista na figura 6 e apresenta dois canais de para execução de operações com inteiros, um canal de instruções *Load/Store*, e um canal de instruções de multiplicação. Os estágios desta etapa são:

- E0: leitura/escrita no conjunto de registradores, permitindo até seis leituras para duas instruções;
- E1: nos pipeline ALU e *Load/Store*, ocorrem operações de deslocamento de dados e geração de endereços de memória;
- E2: unidades ALU executam as instruções de inteiros e lógicas, o primeiro estágio do primeiro pipeline de multiplicação é executado, e a tradução de endereços ocorre no pipeline de leitura/escrita;
- E3: unidades ALU executam as instruções de saturação aritmética (valores teto e piso), o segundo estágio do primeiro pipeline de multiplicação é executado. No pipeline de leitura/escrita, se houver uma escrita, o dado é formatado e enviado à unidade no pipeline em que será usado. Em caso de escrita, os dados são preparados para escrita em *cache*;

- E4: canais ALU verificam tratam a ocorrência de alterações no fluxo de execução, o pipeline de multiplicação efetua etapas de adição, e a atualização da *cache* L2, caso necessária, ocorre no pipeline de leitura/escrita;
- E5: resultados das operações são colocados nos registradores.

As instruções de ponto flutuante e SIMD são tratadas previamente pelos canais de do pipeline de inteiros, e então enviadas para o coprocessador VFP, no caso de ponto flutuante e para a unidade NEON no caso de SIMD. A unidade NEON, e o coprocessador VFP serão vistos mais adiante, e o esquema de uso dessas unidades pode ser visto na figura 7.

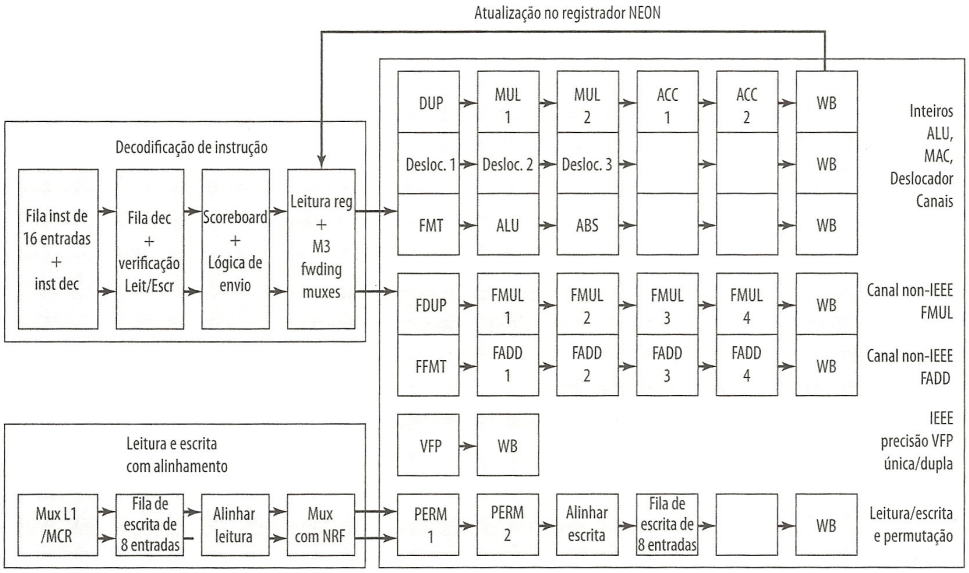

Figura 7: Estágios de SIMD e ponto flutuante do pipeline do *ARM Cortex-A8*[16]

5 Extensões ARM

Conforme o uso processador ARM ganhou abrangência, foram desenvolvidas extensões para a arquitetura, a fim de atender às aplicações citadas na seção anterior. Algumas extensões são citadas a seguir:

- NEON[7]: trata-se de uma unidade SIMD projetada para a série *Cortex-A* e apresenta 32 registradores de 64 bits, que podem ser usados como 16 registradores de 128 bits, organizados em vetores e acessados aos pares simultaneamente, procedimento ilustrado na figura 8. O NEON permite executar instruções SIMD de 64 e 128 bits, e acelera operações de manipulação de áudio e vídeo;
- VFP[1]: unidade de processamento de ponto flutuante, seguindo o padrão IEEE 754. É utilizada em controle automotivo, gráficos 3D, controle industrial e tratamento de imagens;

- DSP[5]: adiciona instruções para processamento de sinais digitais, mas projetado para impactar o mínimo possível em consumo de energia e acelerar aplicações que usem sinais;
- Jazelle[6]: implementação em *hardware* da máquina virtual Java, permitindo a execução direta de *bytecodes* (DBX (*Direct Bytecode eXecution*)). Além da implementação em *hardware*, há um conjunto de *softwares* que dá suporte para o uso eficiente da extensão;
- *big.Little*[3]: combinação de *hardware* e *software* para processamento mais eficiente em processadores *multicore* heterogêneos. Essa tecnologia permite distribuir as aplicações entre *cores* de diferentes velocidades, e consumos de energia, para que os processos sejam executados rapidamente com o menor consumo possível de energia. *GPUs* também podem ser alocadas;
- *TrustZone*[9]: conjunto de *hardware* e *software* para proteger aplicações de ataques, colocando-as em ambientes de execução isolados por meio de modos de exceção apropriados . Na figura 9 há um esquema dos modos de execução usados.

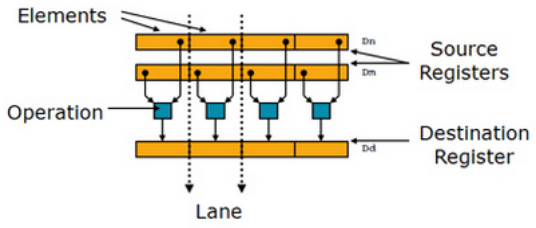

Figura 8: Operação com dados de 128 bits no NEON

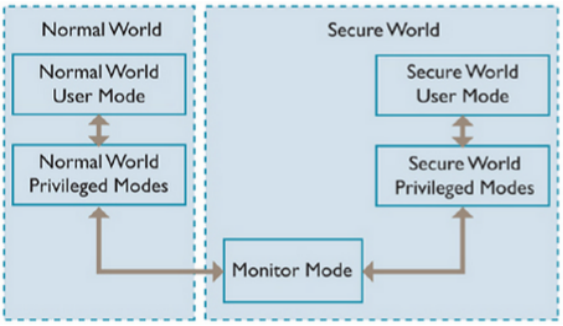

Figura 9: Intercâmbio de modos de operação para segurança

6 Aplicações Recentes

A arquitetura ARM têm servido a muitos projetos acadêmicos recentes em diversas áreas. Nesta seção são vistos alguns projetos.

6.1 Ensino

Um dos desafios do ensino de arquitetura e organização de computadores é motivar os alunos, dada a complexidade e quantidade de detalhes necessários ao entendimento. Os processadores ARM fazem parte do cotidiano dos estudantes, estão presentes em *Smartphones*, *Tablets*, Automóveis, Aparelhos de TV, etc. Além disso, as raízes RISC desses processadores fazem com que eles apresentem as características gerais dos processadores modernos e reduzida necessidade de compatibilidade com aplicações mais antigas. Isso facilita a adaptação das disciplinas de arquitetura a mudanças [10].

O tamanho reduzido dos processadores ARM, seu baixo consumo de energia e tecnologias de suporte a SoC's (*System on a Chip*), permitem o desenvolvimento de sistemas baixíssimo custo como o *Raspberry Pi*. Trata-se de um pequeno computador de custo máximo de 35 dólares, e que executa diversas distribuições do sistema operacional Linux. A combinação de baixo custo e a possibilidade de entrever o comportamento do sistema via soluções de baixo nível do Linux têm feito do pequeno computador um elemento presente em muitos cursos de arquitetura [18].

6.2 Instrumentação e Otimização de Código

Ferramentas de instrumentação não são novidade para programar aplicações de propósito geral, mas não há muitas opções para sistemas embarcados. Identificar gargalos de desempenho, problemas com uso de memória, trechos otimizáveis, e entrever o consumo de energia são questões fundamentais para todo tipo de aplicação, mas são ainda mais importantes para aplicações embarcadas, cujas restrições podem ser muito severas. Em [12], os autores desenvolveram, a partir de uma ferramenta de instrumentação da Intel, uma versão para processadores ARM. Em [15],os autores desenvolvem técnicas para diminuir o *overhead* imposto pelos mecanismos necessários à instrumentação.

Hoje, o compilador executa diversas formas de otimização de código. O GCC, por exemplo, oferece diferentes níveis de otimização na plataforma x86 e também para ARM. Entretanto, dado o uso geral que o GCC se propõe a atender, não há muitas otimizações específicas para plataformas ARM. A fim de resolver esse problema, os autores de [14] , propõem e implementam medidas para melhorar a otimização de programas ARM pelo GCC.

6.3 Virtualização

O crescente poder computacional dos processadores ARM, combinado com a adição de diversas extensões, vem permitindo o uso de aplicações cada vez mais elaboradas. Uma dessas aplicações é o Monitor de Máquinas Virtuais de [17]. Os autores implementaram modos virtuais de processador, conjunto virtual de registradores e níveis de proteção virtual de memória a fim de lidar com alguns aspectos da arquitetura ARM. Esses aspectos são: a disponibilidade de diferentes registradores a cada modo de execução; diferenças na execução de instruções em modo privilegiado e não privilegiado; gerenciamento de permissão de acesso a domínios de memória. Por enquanto, apenas uma máquina virtual é suportada e apenas o Linux pode ser executado como sistema convidado.

6.4 Aceleração do algoritmo *Multi-Scale Retinex*

Em [11], os autores procuram acelerar a execução do algoritmo *Multi-Scale Retinex*, que corrige as cores em imagens obtidas com iluminação deficiente. Tal algoritmo é interessante para dispositivos móveis com câmera, já que, em muitos casos, a iluminação ambiente não é suficiente


para garantir a qualidade da imagem. O processo de aceleração começa dividindo o algoritmo em três partes e fazendo algumas alterações nos procedimentos originais do algoritmo. O ganho de velocidade de execução depende das instruções SIMD, executadas pela unidade NEON, tratada anteriormente. Segundo os autores, o algoritmo otimizado com instruções SIMD, mostrou uma melhora de 74,25% no tempo de execução, em relação à implementação original.

7 Considerações Finais

Ao longo deste artigo foram vistas características gerais da arquitetura ARM. É possível perceber pelas decisões de projeto, a simplicidade inicial dos processadores, seguida da adição de funcionalidades opcionais, oferecidas pelas extensões, e pela evolução constante da arquitetura. Essas características permitem o atendimento de dispositivos e aplicações que variam bastante em aspectos fundamentais como alimentação, demanda, segurança e restrições de tamanho. Outro benefício das decisões de projeto tomadas é o custo, uma vez que os fabricantes tem liberdade para utilizar processadores de custo mínimo ao adicionar apenas as extensões necessárias, quando houver necessidade de alguma. A preocupação com a eficiência energética é evidente. As medidas para acelerar o processamento são acompanhadas por mecanismos como o uso de filas e *replay* vistos no pipeline do *ARM Cortex-A8*, e *buffer* de escrita entre a memória *cache*, contribuem para o uso eficiente da energia. Dessa forma, tem-se uma noção geral dos motivos que levam aos pontos fortes que explicam o sucesso da arquitetura ARM .

Referências

[1] ARM. Arm floating point architecture. http://www.arm.com/products/processors/technologies/vector-floating-point.php, 2015. Acessado em 25 de Maio de 2015.

[2] ARM. Armv8-a architecure. http://www.arm.com/products/processors/armv8-architecture.php, 2015. Acessado em 25 de Maio de 2015.

[3] ARM. big.little technology. http://www.arm.com/products/processors/technologies/biglittleprocessing.php, 2015. Acessado em 25 de Maio de 2015.

[4] ARM. Classic processors. http://arm.com/products/processors/classic/index.php, 2015. Acessado em 25 de Maio de 2015.

[5] ARM. Dsp & simd. http://www.arm.com/products/processors/technologies/dsp-simd.php, 2015. Acessado em 25 de Maio de 2015.

[6] ARM. Jazelle. http://www.arm.com/products/processors/technologies/jazelle.php, 2015. Acessado em 25 de Maio de 2015.

[7] ARM. Neon. http://www.arm.com/products/processors/technologies/neon.php, 2015. Acessado em 25 de Maio de 2015.

[8] ARM. Processors. http://arm.com/products/processors/index.php, 2015. Acessado em 23 de Maio de 2015.

[9] ARM. Trustzone. http://www.arm.com/products/processors/technologies/trustzone/index.php, 2015. Acessado em 25 de Maio de 2015.

[10] A. Clements. Arms for the poor: Selecting a processor for teaching computer architecture. In *Frontiers in Education Conference (FIE), 2010 IEEE*, pages T3E–1–T3E–6, Oct 2010.

[11] Ching-Tang Fan, Jian-Ru Chen, Chung-Wei Liang, and Yuan-Kai Wang. Accelerating multi-scale retinex using arm neon. In *Consumer Electronics - Taiwan (ICCE-TW), 2014 IEEE International Conference on*, pages 77–78, May 2014.

[12] Kim Hazelwood and Artur Klauser. A dynamic binary instrumentation engine for the arm architecture. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 261–270, New York, NY, USA, 2006. ACM.

[13] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[14] P. R. Mahalingam and Shimmi Asokan. A framework for optimizing gcc for arm architecture. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, ICACCI '12, pages 337–342, New York, NY, USA, 2012. ACM.

[15] Ryan W. Moore, José A. Baiocchi, Bruce R. Childers, Jack W. Davidson, and Jason D. Hiser. Addressing the challenges of dbt for the arm architecture. *SIGPLAN Not.*, 44(7):147–156, June 2009.

[16] W. Stallings. *Arquitetura e Organização de Computadores*. Pearson, 2011.

[17] A. Suzuki and S. Oikawa. Implementation of virtual machine monitor for arm architecture. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 2244–2249, June 2010.

[18] David Tarnoff. Integrating the arm-based raspberry pi into an architecture course. *J. Comput. Sci. Coll.*, 30(5):67–73, May 2015.