Dotando ArchC com infraestrutura para geração de montadores e simuladores ARM

Rafael Auler, Paulo Cesar Centoducatte
Universidade Estadual de Campinas
Instituto de Computação
Campinas, SP, Brasil
rafaelauler@gmail.com ducatte@ic.unicamp.br

Resumo

O processador ARM constitui um importante ponto de referência para o estudo de arquiteturas de computadores, graças à sua recorrente utilização na indústria de sistemas embarcados. Neste artigo discutimos o desenvolvimento de um modelo da arquitetura ARM utilizando a ADL ArchC e modificações necessárias no projeto ArchC para viabilizar o modelo deste processador. Dedicamos boa parte do estudo à sintaxe da linguagem de montagem ARM e sua descrição integrada ao modelo, de modo a modificar o gerador de montadores do projeto ArchC. Os resultados são um gerador de montador para programas ARM-ELF e um simulador comportamental do ARM capaz de executar esses programas, de forma a disponibilizar um conjunto completo de ferramentas para estudo do processador ARM.

1. Introdução

O processador ARM[1] e sua arquitetura, na época de suas concepções, foram motivados pelas ideias promissoras do modelo RISC (Reduced Instruction Set Computer), a exemplo do processador MIPS[6]. A empresa ARM Ltd. teve apreciável aceitação no mercado de processadores embarcados de baixo consumo de potência, o que contribuiu para a disseminação desta arquitetura. Entre suas características comuns, podemos citar o tamanho fixo de instruções (32 bits), o padrão *load/store*, e um grande e uniforme banco de registradores. O que diferencia a arquitetura ARM das demais são particularidades como a execução condicional de todas as instruções e o uso frequente de deslocamentos binários de operandos.

O projeto ArchC[3] engloba diferentes ferramentas que, de posse das informações contidas em um modelo do processador, produzem simuladores (com ou sem precisão de ciclo), montadores, ligadores e depuradores. Os simuladores são produzidos com uma descrição SystemC do processador, o que significa que o módulo do processador pode ser integrado como parte de uma plataforma maior. A extensão do ArchC para geração automática de montadores, denominada *acasm*[4], enriquece a capacidade de descrição de um modelo arquitetural, com a possibilidade da descrição das sintaxes da linguagem de montagem específica do conjunto de instruções do modelo. Deste modo, uma ferramenta é capaz de extrair informações suficientes para produzir automaticamente um montador para a arquitetura alvo. As ferramentas binárias são produzidas com o redirecionamento do pacote GNU binutils[7], o que significa que a mesma interface usada pela suíte de programas binutils está disponível para o projetista gerar executáveis para a sua arquitetura.

Entretanto, não era possível o desenvolvimento completo do modelo ARM para ArchC. A sintaxe da linguagem de montagem ARM mostrou-se ligeiramente mais complexa do que o `acasm` original era capaz de processar. O subconjunto da linguagem que era possível de se expressar no modelo que `acasm` esperava possuía construções desnecessariamente longas e tediosas (devido ao compromisso de `acasm` trabalhar com sintaxe simples e trivial, sem sofisticações). Ainda, como o projeto ArchC foi desenvolvido majoritariamente com arquiteturas big-endian, o suporte a little-endian estava confuso e incipiente.

O projeto de software livre SimIt ARM[8] realiza a simulação de programas que utilizam o conjunto de instruções da arquitetura ARMv5. Contudo, o usuário deve alterar o código-fonte original se quiser incluir novas instruções para fins de pesquisa. Para a geração de montadores, pode-se utilizar o próprio pacote binutils[7], mas incrementar o montador depende de familiaridade com o código-fonte. Nossa proposta é dotar o projeto ArchC da capacidade de compreender o modelo de um processador ARM, para que possa gerar as ferramentas simulador e montador automaticamente para a arquitetura ARM, assim como já faz para outras arquiteturas. Desta forma, o simulador e montador podem ser facilmente estendidos com modificações no modelo ARM para ArchC.

Este trabalho se propõe a descrever as etapas envolvidas no processo de desenvolvimento e adaptação dos módulos necessários para a descrição de arquiteturas ARM, no con-

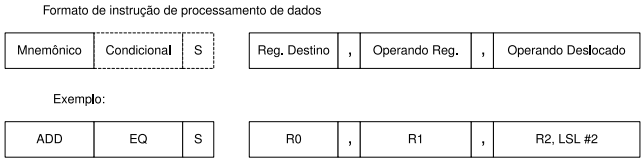

Figura 1. Sintaxe de instruções de processamento de dados do ARM e um exemplo.

texto da ADL ArchC. A descrição será, em grande parte, focada no gerador de montadores, onde estão as maiores dificuldades encontradas. Pode-se ainda generalizar e aplicar as alterações aqui discutidas para outras arquiteturas que não possuam um conjunto de instruções homogêneo e de fácil descrição, como a arquitetura do processador AVR32[2], que é semelhante à arquitetura do ARM.

O restante deste artigo é organizado como a seguir. A seção 2 apresenta o conjunto de instruções da arquitetura ARM. A seção 3 apresenta modificações necessárias em `acasm` para representar este conjunto de instruções. A seção 4 discute aspectos do simulador relevantes para o modelo ARM. A seção 5 apresenta os resultados, e a seção 6 as conclusões.

2. Modelagem da Arquitetura ARM

O conjunto de instruções do processador ARM pode ser dividido em instruções de processamento de dados, instruções de salto, instruções *load/store* ou transferência de dados, instruções de troca e multiplicação, instruções que envolvem coprocessador e instruções especiais.

Estas categorias foram elaboradas de acordo com a estrutura de codificação do conjunto de instruções ARM. É importante salientar que o conjunto de instruções do ARM está em sua quinta versão[1], já tendo sofrido algumas ampliações. Portanto, algumas instruções com semântica parecida possuem codificação diferente. Por esse motivo há a necessidade de segregar a categoria "Multiplicação" de "Instruções de processamento de dados". A versão da arquitetura ARM descrita no modelo é a ARMv5, desconsiderando instruções especiais DSP (*Digital Signal Processing*) e todo o conjunto de instruções Thumb[1], que estão fora do escopo deste modelo. A figura 1 mostra a sintaxe em linguagem de montagem ARM da primeira categoria da lista (processamento de dados). Cada retângulo indica um símbolo que deve ser reconhecido pelo parser do montador. Retângulos tracejados indicam símbolo anulável, ou que pode ser omitido.

Ainda que as instruções ARM possam ser organizadas de forma coerente nas categorias listadas, o modelo ARM para ArchC não conta com apenas estes formatos de codificação. Algumas categorias precisam ser expressas em mais de um formato, como a categoria de instruções de processamento de dados. Isso se deve ao fato de que o modo de endereçamento tem um papel crítico na codificação da instrução, e muitas vezes é necessária a segmentação de um modelo de codificação em diversos outros semelhantes em opcode, mas que diferem no modo como representam os operandos.

3. Modificações em Acasm

Para expressar a linguagem de montagem ARM em sua totalidade, é necessário prover maior flexibilidade na interpretação de sintaxes pelo `acasm`. A figura 2 mostra um diagrama onde a atuação de diversas ferramentas do projeto ArchC, com destaque para o gerador de montadores `acasm`, é ilustrada, com o intuito de contextualizar a ferramenta `acasm` no âmbito do projeto ArchC. Neste trabalho, fazemos uso especificamente das ferramentas `acasm` e gerador de simulador comportamental `acsim`[3].

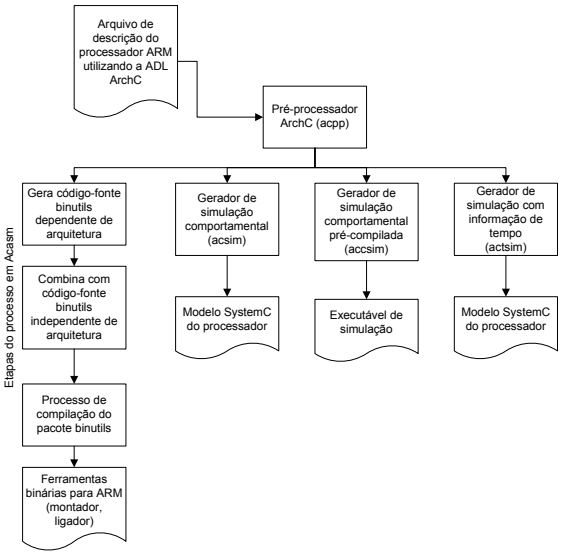

Figura 2. Diagrama que descreve a estrutura geral do ArchC, com destaque para o gerador de montadores.

3.1 Mapas de Símbolos

Como a descrição de sintaxe de linguagem de montagem em um modelo ArchC foi mantido o mais simples possível, o único modo de expressar uma linguagem mais abrangente é recorrendo ao recurso de sobrecarga de sintaxe. Desse modo, para uma única instrução, podemos definir mais de uma descrição de sintaxe, cada uma delas levando a uma codificação diferente para a mesma instrução. Para abordar

```
 1. ac_asm_map cond {
 2.  "eq" = 0;    "ne" = 1;
 3.  "cs", "hs" = 2;
 4.  "cc", "lo" = 3;
 5.  "mi" = 4;    "pl" = 5;
 6.  "vs" = 6;    "vc" = 7;
 7.  "hi" = 8;    "ls" = 9;
 8.  "ge" = 10;   "lt" = 11;
 9.  "gt" = 12;   "le" = 13;
10.  ""   = 14;
11. }
12.
13. ac_asm_map sf {
14.   "" = 0;    "s" = 1;
15. }
16.
17. add.set_asm("add%cond%sf %reg, %reg, %reg",
18.   cond, s, rd, rn, rm, shiftamount=0, shift=0);
```

Figura 3. Exemplo de especificação da sintaxe da instrução ADD

o problema do excesso de sobrecarga, um mecanismo para definição de símbolos não terminais pelo usuário foi melhor elaborado, de forma a suportar diversos símbolos não terminais unidos ao mnemônico, viabilizando a descrição de um mnemônico com diversas variações em apenas uma regra de formação.

Por exemplo, a sintaxe da figura 1 precisaria de 32 descrições para expressar o mnemônico "add" com todas as combinações possíveis de sufixos de condição e sufixo "s" (atualização de registrador de estado). Com o uso de símbolos definidos pelo usuário, entretanto, podemos usar apenas uma descrição, indicando o mnemônico seguido por dois símbolos anuláveis. Esta descrição aparece na figura 3, onde dois mapas, `cond` e `sf`, são definidos pelas *tokens* que podem assumir. Ao fim, a sintaxe da instrução `add` é definida com a diretiva `set_asm`, fazendo também referência ao símbolo `reg`, cuja definição não aparece explicitamente no exemplo. No caso em que a sobrecarga é usada para substituir os mapas, teríamos uma construção `add.set_asm` para cada mnemônico `add` com um sufixo válido.

Os símbolos definidos pelo usuário são organizados em mapas através da diretiva `ac_asm_map`, onde qualquer quantidade de mapas pode ser criada, de forma a atender as necessidades de símbolos não terminais. Entretanto, `acasm` trabalhava com uma restrição em que elementos de mesmo nome em mapas diferentes não poderiam existir. Esta restrição foi removida, de forma que diferentes mapas possam conter o elemento `""` (vazio). Com o elemento vazio, é possível criar símbolos não terminais anuláveis, um recurso valioso na definição da sintaxe das instruções do ARM. Outra modificação importante para descrever formatos complexos é a elaboração de um mecanismo para concatenar 3 ou mais campos do formato de instrução e atribuí-los a um único operando da sintaxe da instrução.

3.2 Listas de Símbolos

Para a descrição da sintaxe de instruções *load/store* de múltiplos registradores, usadas tanto no conjunto de instruções ARM quanto AVR32, é necessário um parser que espere por um número arbitrário de registradores. Isto ocorre porque tais instruções, utilizadas em operações de pilha, podem salvar ou carregar qualquer número de registradores. Portanto, tais instruções não podem ser descritas como mnemônico seguido de seus operandos, uma vez que não se sabe exatamente quantos operandos existirão. Para abordar este problema, a funcionalidade de listas de símbolos foi criada, de modo que a sintaxe de uma instrução pode especificar um operando representado por uma lista arbitrária de símbolos, como por exemplo, uma lista de registradores.

4. Gerador de Simuladores

Um módulo importante do simulador é o decodificador de instruções, que, de posse do mapa de formatos do conjunto de instruções, irá determinar qual é a instrução indicada pelo *program counter*. O decodificador deve buscar bytes da memória de instruções quantas vezes for necessário para determinar a instrução. A organização dos bytes da arquitetura (little-endian ou big-endian) é crucial neste momento. Algumas arquiteturas de processador são little-endian, mas o formato de suas instruções é definido, da esquerda para a direita, na ordem em que os bytes são buscados pelo decodificador, como a Intel x86. Já o ARM little-endian especifica um formato que não corresponde à organização real dos bytes das instruções na memória. Este formato deve ser invertido byte a byte, como uma palavra de 32 bits little-endian. Para o decodificador, o mais razoável é que o conjunto de instruções seja descrito na ordem em que os bytes aparecem, como acontece com a arquitetura Intel x86. Contudo, o decodificador do processador ARM espera instruções de tamanho fixo (4 bytes por vez). Dessa forma, o fato de que as instruções são invertidas como dados na memória não gera grandes desafios ao decodificador (basta buscar 4 bytes e invertê-los). O gerador de simuladores `acsim` interpreta o formato das instruções little-endian de forma a invertê-lo, para que seja facilitada a descrição de modelos como o ARM.

5. Resultados

Segundo o diagrama da figura 2, ao fornecermos um modelo do ARM completamente especificado, temos um montador e um simulador para esta arquitetura. Durante o processo de validação deste trabalho, o montador e o simulador foram avaliados separadamente.

A tabela 1 mostra os resultados da validação do montador ARM produzido. Este procedimento consistiu em utili-

Programa	# Arquivos	Tamanhos das seções ELF [bytes]			
		Total	.text	.data	.bss
basicmath_small	4	10.449	5.208	0	0
basicmath_large	4	11.768	6.300	0	0
bitcount	9	10.632	4.064	568	0
qsort_small	1	5.469	1.116	0	0
qsort_large	1	6.365	1.924	0	0
susan	1	63.200	51.520	0	0
jpeg	60	314.152	232.080	0	16
typeset	1	474.498	26.696	411.184	7.680
dijkstra_small	1	7.493	2.296	0	40.832
dijkstra_large	1	7.493	2.296	0	40.832
ispell	1	5.778	1.108	372	0
stringsearch_small	4	13.965	4.828	256	4.140
stringsearch_large	4	26.617	4.896	256	4.140
blowfish	7	25.404	15.400	4.172	0
sha	2	7.836	2.996	0	0
rawcaudio	2	7.721	2.260	420	2.504
rawdaudio	2	7.701	2.240	420	2.504
crc32	1	6.798	1.320	1.024	0
fft	3	10.709	5.192	0	0

Tabela 1. Resultados experimentais do montador ARM gerado

Programa	Entradas pequenas		Entradas grandes	
	# instruções	MIPS	# instruções	MIPS
Quick Sort	15.041.031	3,16	200.072.535	3,02
Susan (Corners)	3.987.253	3,02	46.836.594	3,13
Susan (Edges)	7.547.235	3,07	183.427.123	3,13
Susan (Smoothing)	28.180.598	3,26	280.612.545	3,42
Basic Math	1.619.944.365	3,13	25.340.627.730	3,11
Bit Count	42.885.165	3,29	643.742.593	3,29
CRC32	22.025.235	3,14	428.096.758	3,15
ADPCM Coder	25.747.893	2,76	512.422.308	2,8
ADPCM Decoder	22.275.313	2,92	437.317.734	2,94
FFT	919.254.894	3,09	18.794.836.954	3,12
FFT Inv	2.236.580.567	3,08	18.264.730.916	3,1
GSM Coder	26.394.969	3,34	1.418.931.661	3,36
GSM Decoder	10.179.406	3,1	554.091.317	3,14
Dijkstra	54.240.220	3,2	251.831.995	3,22
Patricia	307.306.769	3,16	1.949.502.029	3,12
Rijndael Encode	28.813.981	3,25	300.027.673	3,28
Rijndael Decode	29.008.859	3,18	302.057.381	3,22
SHA	14.032.166	3,49	146.074.724	3,48
JPEG Encoder	24.490.995	3,21	90.559.844	3,21
JPEG Decoder	6.973.038	3,24	23.461.972	3,29
LAME MP3 Encoder	9.980.953.916	3,11	116.772.981.373	3,09

Tabela 2. Resultados experimentais do simulador ARM gerado.

zar um compilador ARM cross-gcc versão 3.4 para produzir código fonte em linguagem de montagem ARM de diversos programas do benchmark Mibench[5]. Em seguida, um montador ARM original (do pacote binutils versão 2.15) e o montador ARM sintetizado pelo ArchC (utilizando o mesmo pacote do binutils) foram usados para produzir dois executáveis ELF para cada programa do benchmark. As seções de código .text, dados inicializados .data e não inicializados .bss foram comparadas uma a uma entre os dois executáveis. Para todos os programas da tabela, os executáveis apresentaram conteúdo idêntico seção por seção, indicando o sucesso na geração do montador ARM. A tabela indica, ainda, o tamanho das seções produzidas, em bytes, bem como o tamanho total do executável ELF. O número de arquivos indica quantos arquivos foram ligados (utilizando o ligador gerado) para produzir o executável final.

Na tabela 2 podemos encontrar o número de instruções ARM simuladas para cada programa do benchmark Mibench, bem como a velocidade da simulação em milhões de instruções por segundo (MIPS). A validação do simulador consistiu em comparar a saída produzida pelo simulador a cada programa com a saída produzida pelos mesmos programas quando compilados e executados em outra arquitetura (por exemplo Intel x86) como referência de corretude (*golden model*).

6. Conclusões

Este trabalho apresentou o processo de geração de um montador e simulador para a arquitetura ARM através da descrição de um processador ARM utilizando a ADL ArchC. Algumas modificações no projeto ArchC foram realizadas e o objetivo de suportar a arquitetura ARM foi alcançado. Ainda, os resultados apresentados mostram fortes evidências da corretude das ferramentas produzidas com ArchC. O modelo construído para a arquitetura ARM e as ferramentas elaboradas são de domínio público, disponibilizados na página[1] do projeto ArchC. Vale destacar como trabalho futuro o suporte, no ArchC, à descrição do conjunto de instruções 16 bits Thumb da arquitetura ARM.

Referências

[1] ARM Limited. *ARM Architecture Reference Manual*, 2000.

[2] Atmel Corporation. *AVR32 Architecture Document*, 2007.

[3] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros. The ArchC architecture description language and tools. *International Journal of Parallel Programming*, 33(5):453–484, October 2005.

[4] A. Baldassin, P. C. Centoducatte, and S. Rigo. Extending the ArchC language for automatic generation of assemblers. In *Proceedings of the 17th Symposium on Computer Architecture and High Performance Computing*, pages 60–68, October 2005.

[5] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th Annual IEEEWorkshop on Workload Characterization*, pages 3–14, December 2001.

[6] G. Kane. *MIPS RISC architecture*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[7] R. H. Pesch and J. M. Osier. *The GNU binary utilities*. Free Software Foundation, Inc., May 1993. version 2.15.

[8] W. Qin. `http://www.simit-arm.sourcefourge.net`. Acessado em setembro de 2009.

[1] http://www.archc.org