Relatório
Decodificador de Instruções THUMB

Jonas Bezerra da C. Máximo, Sávia Rafaella L. Fernandes

Universidade Federal do Ceará – Campus Quixadá
Curso de Engenharia de Computação
Arquitetura de Computadores II
Prof. Roberto Cabral

4 de Dezembro de 2019

1 Introdução

Pode-se dizer que o tempo em que um computador ou CPU leva para executar uma tarefa atende ao ciclo de instrução. Esse ciclo consiste nos seguintes passos:

1. Cálculo do endereço de memória que contém a instrução;
2. Busca da instrução;
3. Decodificação da Instrução;
4. Busca pelo operando;
5. Execução da operação;
6. Armazenamento do dado em um local na memória.

Com isso, o presente trabalho foca na implementação de um programa que atenda a funcionalidade do item 3. A decodificação de instruções é realizada antes da instrução ser executada, sendo que cada uma pertence a um conjunto específico. O conjunto trabalhado para esse desenvolvimento foi de instruções THUMB onde os autores utilizaram a tabela B.5 do livro ARM System Developer's Guide. Essa tabela proporcionou o acesso às informações necessárias para que pudesse ser desenvolvido um programa capaz de receber um conjunto de instruções e decodificar em um valor hexadecimal.

2 Implementação

Para implementar o decodificador de instruções foi utilizada a linguagem Python e as IDE's Sublime e Visual Studio. O modelo de implementação foi dividido em módulos que são operados principalmente na main e que realizam chamadas de funções de outros arquivos. O fluxograma na Figura 1 demonstra o comportamento da decodificação de forma geral:

Para implementação desses módulos o projeto foi dividido em 5 arquivos: 1 do tipo *.txt* para leitura e que contém as instruções, 3 arquivos *.py* de funções e tratamentos, além de outro arquivo *.txt* que recebe o resultado da decodificação.

A partir da leitura do arquivo foi transferido seu conteúdo para uma lista de listas (semelhante a uma matriz) onde cada posição continha uma linha e cada linha continha um conjunto de instruções, registradores, imediatos, etc, como mostrado no exemplo abaixo:

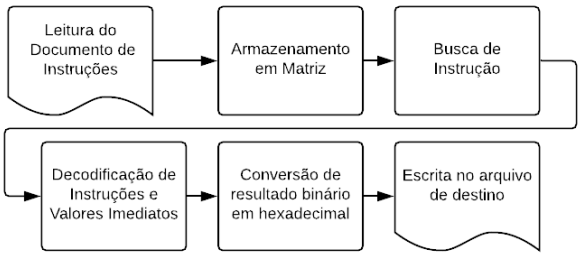

Figura 1: Módulos de Implementação

[['lsr', 'r1', 'r4', '#5'],['sub', 'r0', 'r1', '3'],
push', '{r3', 'lr}']

Após os dados estarem prontos no arquivo da *main* é identificada a primeira posição de cada linha que é referente a instrução então essa linha e passada como parâmetro para o para outro arquivo para ser tratada o arquivo de instrução onde e analisada os diferentes tipos de instruções e onde cada um se encaixa, no arquivo de instrução está lincado com o arquivo de função onde possui algumas funções para auxiliar na decodificação.

Cada instrução possui um conjunto de bits que as representam e tornam possível o seu funcionamento. A montagem se deu com base na tabela do conjunto de instruções THUMB e o desenvolvimento de um código capaz de montar os valores de cada uma.

A sequência de execução para obtenção do resultado requerido pode ser explicada através do significado de cada módulo e conteúdos implementados em cada arquivo.

3 comandos.txt

O arquivo *comandos.txt* contem o código THUMB que será decodificadas. O conteúdo desse arquivo possui o

formato padrão de código em *assembly*.

O código é capaz de decodificar um grande conjunto de instruções THUMB, são elas:

LSL, LSR, ASR, ADD, SUB, MOV, CMP, AND, EOR, LSL, LSR, ADC, SBC, ROR, TST, NEG, CMP, CMN, ORR, MUL, BIC, MVN, CPY, BX, BLX, LOL, STR, STRH, STRB, LDRSB, LDRSH, STR, LDR, LDRB, SXTH, SXTB, UXTH, UXTB, REV, REV16, REVSH, PUSH, POP, SETEND LE, STEEND BE, CPSIE, SPSID, BKPT, STMIA, LDMIA e SWI.

Algumas instruções possuem ramificações de forma que podem ser operadas apenas com registrados e também com valores imediatos. Esses casos também foram tratados de acordo com o segmento ofertado pela tabela de instruções a qual a implementação foi baseada.

4 main.py

O arquivo *main.py* contém as operações principais que tornam possível a decodificação através da chamada de funções de outros arquivos. Foram utilizadas estruturas de dados para que fosse possível tratar cada linha das instruções. A *main* pode ser dividida nos módulos utilizados para o planejamento da implementação, os quais serão descritos a seguir.

4.1 Leitura do Documento de Instruções

A leitura é feita de forma que é criada uma lista que seu conteúdo são todas as linhas do arquivo *comandos.txt*. Para cada linha, existe outra lista que contém todos os elementos da linha. O programa identifica os elementos válidos através da remoção de vírgulas utilizando a função *sub* que pertence a biblioteca *re.py*. Assim, serão utilizados apenas registradores, valores o caractere $'\#'$ que pertence ao uso de valores imediatos.

4.2 Armazenamento em Matriz

Todas as linhas do arquivo lido se tornaram a lista *lista_instrucoes* que através dela todo o conteúdo pode ser acessado em forma de matriz. Como por exemplo: a posição [0] contém a primeira linha do arquivo de comandos. Já utilizando a posição [0][0] é possível acessar o primeiro elemento da primeira linha do documento lido.

Com essa possibilidade de acesso a matriz, é possível identificar qual instrução é requerida a decodificação. O programa busca o termo o nome da instrução em todas as primeiras posições de todas as linhas, já que ela é o primeiro elemento a ser escrito. Como por exemplo, uma instrução de soma é dada por "**mov r0, r1**". O mov ocupa a primeira posição da linha do documento em que está sendo acessada.

4.3 Escrita no arquivo de destino

Ainda na *main* é realizada a escrita do resultado no arquivo de destino logo após o tratamento da decodificação ser realizado. Para isso foi utilizada a função python *write*.

5 funcoes.py

O arquivo *funcoes.py* contém as funções que decodificam cada tipo de instrução, ou seja, são as funções que tratam quando a instrução é identificada no arquivo *main.py* e há a chamada de função para os tratadores.

O livro **Arm System Developer's Guide and Optimizing System** conta com as tabelas de definições de instruções THUMB. Elas foram necessárias devido a necessidade de tratar manualmente os bits. Isso se deu pois as mesmas contêm o mapeamento dos bits referentes a cada elemento que compõe uma instrução e formam a estrutura necessária para decodificação.

Para implementar os tratadores foram criadas, inicialmente, duas listas chamadas *lista_ins_reg* e *lista_dados_asm*. A primeira objetiva salvar o valor binário que corresponde a cada instrução e que pode ser observado na tabela de instruções.

Existem diferentes tipos de instruções: algumas com operandos sendo apenas registradores e outras com os operandos sendo registradores e valores imediatos. Para descobrir qual o formato da instrução a ser decodificada, foi utilizado um método em que um laço acessa sempre o primeiro caractere de cada string contida no vetor da instrução.

Dados esses acessos, se for detectado que cada inicial de cada string seguir a sequência apenas com o caractere "r", significa que as instruções operam apenas com registradores. Esse padrão com um total de 3 registradores se aplica às instruções **add, sub, str, strh, strb, ldrh e ldrb**. Como por exemplo:

add r0, r0, r1

Já com um total de 2 registradores pode se aplicar às instruções **asr, adc, sbc, tst, ror, neg, cmp, orr, bic, cpy, lsl, lsr, and e eor**, como por exemplo:

asr r2, r4

Por outro lado, também é possível instruções operarem com registradores e valores imediatos. Esse tipo de operação será detectada quando durante a busca no vetor da instrução forem encontradas iniciais tanto com o caractere "["quanto como o caractere "#", sendo este referente ao símbolo que procede o valor imediato. Como por exemplo:

str r0, [r1, #8]

Esse tipo de operação seguindo a ordem 'valor registrador/registrador/imediato' pode se aplicar a instruções de **add, sub, str, strh, strb, ldr, ldrh, ldrb**, como temos:

ldrh #2, r1, r2

falar sobre os que foram colocados flags

Para cada modelo citado acima foi desenvolvida uma função que consiga obter exatamente os bits referentes a cada operando. Essas funções serão citadas no tópico "X- Tratamento de modelo"

Logo após descobrir a qual modelo a instrução de operação se aplica, se inicia a decodificação. Nesse desenvolvimento foi utilizado o vetor *lista_ins_reg* para guardar os bits encontrados. Para cada instrução foi criada uma função em que ocorre esse processo que pode ser visualizado através de três etapas:

1. São adicionados os primeiros bits das instruções no vetor;
2. São chamadas as funções de tratamento de registradores ou de valores imediatos;
3. A sequência de bits é adicionada no vetor *lista_dados_asm*.

A **etapa 1** ocorre de acordo com a tabela de instruções contida na bibliografia utilizada. Os primeiros bits são únicos e individuais, portanto a sua inclusão no vetor foi feita de forma manual.

A **etapa 2** ocorre de acordo com a implementação de funções que tratem tanto registradores quanto valores imediatos, tais que o processo de implementação será explicado no tópico do arquivo *tratamentos.py*

O primeiro foi desenvolvimento da seguinte forma: criou-se uma função em que cada registrador, quando utilizado, retorne um valor de 3 bits. Ou seja, cada valor foi atribuído de forma a corresponder com o número do registrador em operação.

O segundo tratamento foi por meio da criação de diferentes funções no arquivo *funcoes.py*, cada uma de acordo com o valor correspondente ao imediato, sendo eles 3, 5, 7 ou 8.

A **etapa 3** insere os valores retornados em *lista_dados_asm* para formar o valor final. Logo após, o valor é transformado em hexadecimal. Essa função de transformação é chamada *transforma_hexa* e foi implementada no arquivo *tratamentos.py*.

Com cada instrução na instrução na tabelas contendo 16 bits, o resultado é na verdade de tamanho 32, pois a cada duas instruções decodificadas decorreu o resultado final.

5.1 Conversão do resultado binário para hexadecimal

Como o programa adquire caracteres que formam um número binário, é realizada a conversão para base 16 a qual foi utilizada uma função de conversão implementada manualmente. Esse valor é o resultado o qual será escrito no arquivo de destino.

6 tratamentos.py

Esse arquivo contém os tratamentos necessários para as funções na main e no arquivo de instruções. Foram implementadas as seguintes funções:

1. *transforma_hexa*;
2. *busca_registrador*;
3. *registrador_grandes*;
4. *trata_imediato_8*;
5. *trata_imediato_7*;
6. *trata_imediato_5*;
7. *trata_imediato_3*;
8. *retira_lixo*.
9. *register_range*.
10. *register_list*.

A **função 1** *(transforma_hexa)* é capaz de transformar valores para a base hexadecimal. Ela recebe *lista_dados_asm* como parâmetro pois é esse vetor que contém os bits da decodificação e será transformada para fornecer o resultado final na base 16.

A **função 2** *(busca_registrador)* recebe como parâmetro o registrador que foi lido na matriz de instruções e que está em operação. A função foi implementada com condições as quais retornam valores diferentes para cada registrador. O critério que diferencia os valores se dá pelo número do registrador de operação e sempre terá o tamanho de 3 bits. Como por exemplo, $r0$ é o registrador 0, portanto se a função entrar na condição de ser ele, será retornado o valor 000. Com isso, tem-se que as atribuições para as condições foram da seguinte forma:
r0: '000', **r1:** '001', **r2:** '010', **r3:** '011', **r4:** '100', **r5:** '101', **r6:** '110', **r7:** '111'.

A **função 3** *(registrador_grandes)* é utilizada para tratar os registradores que são utilizados na decodificação de instruções *bx* e *blx*. A diferença desse tratamento de registradores para o anterior se dá pelo fato de que as instruções *bx* e *bxl* trabalham com pilha. Com isso, é possível serem utilizados registradores acima de r7. Considerando que o valor de retorno para os registradores é relacionado com qual está sendo usado, pode-se concluir que é necessário um número de maior de bits. Isso acontece pois do número 8 até o número 15 são necessários 4 bits para compor os valores binários correspondentes. Com tudo, a função se comporta de forma semelhante a *trata_registrador* porém agora para cada registrador é retornado um valor de 4 bits.

Os valores imediatos na tabela de instruções são divididos em *immed*8, *immed*7, *immed*5 ou *immed*3. Isso significa que os valores recebidos nas instruções precisam ter a mesma quantidade de bits que são dados em cada $'immed'$. Nem sempre isso acontece, como por exemplo, se a instrução utilizar *immed*5 mas receber o valor 2, será necessário preencher com 3 bits $'0'$. Isso acontece pois o valor 2 em binário corresponde a $'10'$, portanto precisa desse preenchimento para se tornar $'00010'$ e assim atender ao *immed*5. Para realizar esse preenchimento que atenda ao tamanho demandado foram implementadas as seguintes **funções 4, 5, 6 e 7** citadas anteriormente, as quais atendem valores de

tamanho 8, 7, 5 e 3.

A **função 8** *(retira_lixo)* recebe como parâmetro as linhas inteiras das instruções contidas no arquivo de leitura. O objetivo é retirar os elementos que não são interessantes para a decodificação. Esses elementos indesejados são dados, por exemplo, por caracteres que indiquem valores imediatos, vírgulas, espaços, colchetes que incluem registradores, entre outros que diferem de registradores e valores. Para realizar essa função foi utilizado a função do python *replace* que faz a troca dos caracteres indesejados por um caractere vazio. Dadas as alterações, a função retorna a nova linha inteiramente tratada e assim é possível a decodificação dos elementos.

A **função 9** foi implementada para setar um valor de registadores. Nela consistem dois parâmetros de entradas que são dois registradores e retorna uma string de 8 bits setando um intervalo.

Por fim, a **função 10** realiza a tarefa de transformar um registrador em uma string de 8 bits. O parâmetro de entrada é o binário correspondente ao registrador e retorna a string com o valor setado do registrador.

Essas funções foram implementadas para juntas formarem o processo minucioso de decodificação e retornam binários que serão concatenados para serem convertidos em hexadecimal.

7 Dificuldades

Durante o desenvolvimento do projeto nos deparamos com diversas dificuldades. Algumas delas foram a extensão do arquivo pois como tinham muitas instruções e para cada instrução vários casos acabou se tornando muito cansativo e demorado o desenvolvimento do decodificador, a dificuldade de compreensão de algumas instruções tornou inviável a implementação das mesmas que foram os branchs (b, blx, bl), também houve grande dificuldade de validar as instruções feitas, pois tiveram que ser feitas a mão transformando em binário e depois em hexadecimal e comparadas uma a uma com o resultado gerado pelo decodificador.

8 Considerações Finais

Com essa atividade conseguimos compreender melhor como funciona um compilador e a decodificação do modo thumb. Como o thumb sendo 16 bits executa em um ambiente como 32 bits, ajudou também a compreender o funcionamento da decodificação do ARM que e semelhante a thumb mudando apenas a quantidade de bits.