UNIVERSIDADE
FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM **ENGENHARIA DE COMPUTAÇÃO**
QUARTO SEMESTRE

THUMBLATOR, UM TRADUTOR DE OPCODES THUMB

PEDRO HENRIQUE **MAGALHÃES BOTELHO**

PAULA ARAUJO **FEITOSA**

FRANCISCO ÍTALO **DE ANDRADE MORAES**

QUIXADÁ - CE
2021

471047 — PEDRO HENRIQUE **MAGALHÃES BOTELHO**

475547 — PAULA ARAUJO **FEITOSA**

472012 — FRANCISCO ÍTALO **DE ANDRADE MORAES**

THUMBLATOR, UM TRADUTOR DE OPCODES THUMB

Orientador: Prof. Msc. Roberto Cabral Rabêlo Filho

Trabalho escrito apresentado no curso de graduação em Engenharia de Computação, pela Universidade Federal do Ceará, campus em Quixadá, para a disciplina de Arquitetura e Organização de Computadores II.

QUIXADÁ - CE
2021

Conteúdo

1 Introdução

Este é o relatório do primeiro trabalho da disciplina de **Arquitetura e Organização de Computadores II**, do quarto semestre do curso de graduação em Engenharia de Computação, na Universidade Federal do Ceará, campus de Quixadá.

Nas seções seguintes serão tratados tópicos essenciais na concepção e implementação do tema proposto pelo orientador, a fim de relatar os acontecimentos no decorrer do trabalho, assim como os resultados finais obtidos.

1.1 Proposta

O trabalho proposto consiste na implementação de um programa em alguma linguagem de programação, seja ela qual for, que decodifique um mapa de memória em seu respectivo código mnemônico *Thumb*.

O decodificador deverá receber como entrada uma mapa de memória, e retornar um arquivo de texto contendo o código devidamente decodificado, em *Thumb*, conforme a tabela B-5 do livro **ARM System Developer's Guide**[2]. O programa deverá ler a entrada de um arquivo com *.in*, e salvar o resultado em um arquivo *.out*.

1.2 Projeto

O projeto foi devidamente testado e implementado para atender aos requisitos e a proposta do orientador. O nome escolhido, **Thumblator**, deriva de duas palavras:

- *Thumb*, do inglês, "polegar". No escopo do estudo das arquiteturas de computadores, denota o conjunto de instruções adicional *Thumb*, da arquitetura ARM, objeto de estudo da disciplina e foco deste trabalho.
- *Translator*, do inglês, "tradutor". Denota o caráter de tradução do projeto, traduzindo códigos de operação em linguagem mnemônica.

Logo, **Thumblator** significa **Tradutor *Thumb***, o que denota com exatidão o objetivo do projeto. O projeto está disponível em repositório do GitHub[1][1], e é dividido em:

- Um programa desenvolvido pela equipe, bem como seu código-fonte e documentação.
- Este relatório, com toda a documentação exigida.
- Apresentações de Slide, desde a concepção até a finalização.

[1]Disponível em https://github.com/pedrobotelho15/thumblator

2 Desenvolvimento

Nessa seção serão apresentados assuntos relativos à concepção e desenvolvimento do projeto. Cada subseção, e seus tópicos, abordarão um tema diferente, levando sempre em conta todos os aspectos possíveis do desenrolar da construção do projeto, podendo assim esclarecer o funcionamento do programa, bem como a implementação e o planejamento empregado, que foi seguido estritamente.

2.1 Planejamento

O planejamento é parte essencial de um projeto, mesmo que seja individual. A presença ativa de companheiros requer um planejamento mais rígido, que funcione não somente para um membro, mas para todos. Reuniões, padronização, ferramentas, tudo irá interferir em um planejamento, mesmo que de forma positiva.

Os tópicos a seguir irão dissertar em torno da questão:

"Como se deu o planejamento do projeto, tendo em vista as dificuldades, sejam elas de comunicação ou técnicas?"

2.1.1 Ferramentas

Tendo em vista que os integrantes da equipe moram em cidades diferentes, foram utilizadas diversas ferramentas para trabalho compartilhado de forma online, no intuito de auxiliar no desenvolvimento do projeto. A padronização de ferramentas teve papel fundamental na concepção do projeto.

- Editor de Código: **Visual Studio Code**
- Compilador: **G++**
- Linguagem de Programação: **C++**
- Depurador: **GDB Debugger**
- Automação de Compilação: **Makefile**
- Editor de Documentos: **LaTeX**
- Controle de Versões e Repositório: **GitHub**
- Comunicação Síncrona: **WhatsApp**
- Comunicação Assíncrona: **Dontpad**
- Videoconferência: **Meet**

2.1.2 Equipe

A fim de realizar com êxito o trabalho proposto pelo professor, foi decidido, a princípio, que seria realizada uma primeira reunião para o entendimento da lógica do trabalho e o que seria feito, além de dividir as tarefas entre os membros da equipe. Foi criado um grupo no *WhatsApp* para facilitar a comunicação entre os membros, trocar ideias e sanar dúvidas, além de marcar semanalmente as reuniões para acompanhar o andamento do projeto. Ficou acertado que, primeiramente, seria feita a implementação do programa para depois fazer o relatório.

2.1.3 Dificuldades

No decorrer do trabalho, algumas dificuldades foram travadas pela equipe. No que diz respeito a fatores externos, provavelmente uma dificuldade a ser relatada foi na organização de horários específicos para a implementação do código em equipe, já que nas poucas vezes que isso ocorreu foi nos encontros semanais, ou seja, a maior parte do trabalho foi realizado individualmente.

A linguagem escolhida para a realização do trabalho foi a linguagem C++ que, de um modo geral, foi tranquila de trabalhar mas que, em alguns momentos, foi necessário pesquisar alguns comandos que melhor se encaixavam de acordo com o objetivo em questão, como, por exemplo, o uso do operador ternário para simplificar o uso dos *if* e *else*.

Com relação ao trabalho em si, uma das dificuldade encontradas por um dos membros da equipe foi para compilar o código inicialmente. Todavia, a mesma foi solucionada realizando a troca do arquivo *header* de *hpp* para *h* e incluindo a *lib* <*string*>. Também foi preciso o estudo em equipe para aprender o conjunto *Thumb*.

A modalidade de ensino remoto também foi um desafio para a realização deste trabalho pela equipe, visto que a interação presencial tem muita importância no aprendizado e desempenho em equipe. Impossibilitados desse recurso, tanto por causa do contexto de pandemia como pelo fato dos três membros da equipe morarem em cidades diferentes, foi preciso optar por encontros síncronos via *Google Meet* para compensar tal necessidade.

2.2 Funcionamento

O programa foi projetado em C++, logo pode ser compilado e executado em qualquer sistema operacional com suporte ao compilador G++. Nessa subseção serão tratados alguns tópicos referentes ao funcionamento do programa após sua compilação ser efetuada, ou seja, o programa executável em si.

Para que o programa possa ser executado ele deve ser devidamente compilado para o sistema operacional desejado. Um arquivo **Makefile** facilita esse processo, automatizando a compilação. Tudo a se fazer é ir na pasta raiz do projeto, e pelo terminal escrever *make all*, comando esse que ordenará que as operações descritas no Makefile sejam executadas. O Makefile auxilia principalmente na organização de arquivos. O repositório do projeto tem seis pastas, onde cada uma guarda arquivos importantes:

- Arquivos binários, os executáveis, ficam em bin/
- Arquivos objetos ficam em build/
- Documentações, como este relatório e slides, ficam em doc/
- Arquivos de cabeçalho(*.hpp) ficam em inc/
- Arquivos de entrada(*.in) e saida(*.out) ficam em io_files/
- Código-fonte a ser compilado fica em src/

O Makefile, então, irá compilar o código, e separar os componentes em pastas. Irá pegar os arquivos de código-fonte em src/ e o arquivo de cabeçalho em inc/, e ao compila-los irá salvar os arquivos objeto em build/ e o binário final em bin/.

A estrutura do Makefile é simples, e faz apenas o necessário. Ao escrever *make clean* os arquivos resultantes da compilação serão apagados. Salientando que, pela utilização de -g na compilação, o arquivo binário pode ser depurado. O código referente a esse arquivo Makefile consta a seguir.

```
1  all: app
2
3  app: main.o thumblator.o
4          g++ -g -O0 -Wall build/main.o build/thumblator.o -o bin/thumblator
5
6  main.o: src/main.cpp
7          g++ -g -O0 -Wall -c src/main.cpp -Iinc -o build/main.o
8
9  thumblator.o: src/thumblator.cpp
10         g++ -g -O0 -Wall -c src/thumblator.cpp -Iinc -o build/thumblator.o
11
12 clean:
13         rm build/*.o bin/thumblator
```

O programa irá realizar traduções de códigos de operação *Thumb* em seus devidos códigos mnemônicos *Assembly*. Esse códigos de operação que serão traduzidos devem estar em um arquivo com extensão *.in. Logo, para que haja sucesso na tradução, o arquivo de entrada deve seguir três regras de escrita.

- Cada linha deve ter 4 caracteres, referentes ao código de operação, e uma quebra de linha.
- A primeira e última linha não podem estar vazias.
- No decorrer to código não poderá ter espaços entre linhas.

Essas praticas permitem que o código seja traduzido de maneira correta. Erros na tradução, como por exemplo extensão inválida ou instrução inválida, são reportados pelo programa ao usuário. Isso será tratado na seção 2.3.2. Um exemplo de código de entrada:

```
0A5B
18AD
1A5B
26DF
2BF9
```

Para que o programa possa traduzir o código de entrada é necessário especificar o arquivo de entrada antes da execução do programa. Esse arquivo de entrada obrigatoriamente deverá ter extensão *.in. Caso nenhum arquivo de saída seja especificado, o arquivo de saída gerado terá o mesmo nome do arquivo de entrada, com a diferença de ter uma extensão *.out. A especificação de um arquivo de saída é opcional, mas caso especificado, é desimportante a utilização de alguma extensão como *.out.

Logo, generalizando, temos: ./thumblator <entrada> <saida[opcional]>. Utilizando o sistema de organização do repositório como exemplo: após a compilação com *make all*, é possível iniciar o programa como:

./bin/thumblator io_files/main.in ou *./bin/thumblator io_files/main.in io_files/main.out*, já terão as mesmas saídas.

./bin/thumblator io_files/main.in io_files/main

./bin/thumblator io_files/main.in main

No último exemplo o arquivo de saída será gerado no atual diretório de trabalho, onde foi executado o programa, mesmo que não seja no mesmo diretório em que está o programa. Abaixo há um exemplo de arquivo de saída:

```
0A5B:   LSR r3, r3, #9
18AD:   ADD r5, r5, r2
1A5B:   SUB r3, r3, r1
26DF:   MOV r6, #223
2BF9:   CMP r3, #249
```

2.3 Implementação

Como já dito antes, o código foi totalmente escrito na linguagem C++. A grande quantidade de classes previamente implementadas na linguagem facilitam bastante o desenvolvimento de uma aplicação, e permitem que haja uma padronização condizente.

Nessa subseção será tratado a implementação do algoritmo de decodificação, bem como a utilização do C++ como facilitador e padronizações adotadas, com finalidade d organizar o código e trazer algum ganho de desempenho, tanto em produtividade pessoal quando em tempo de execução.

2.3.1 Algoritmo de Decodificação

A decodificação é feita com base na tabela B-5, do apêndice do livro citado anteriormente, e é particionada em várias etapas, que dependendo da instrução pode levar mais etapas do que outras. O algoritmo se baseia em verificar os bits de cada código de operação, e ao passo que os bits são verificados a instrução vai tomando forma. Várias instruções são semelhantes no código de operação, mas diferentes na instrução mnemônica, e em outras vezes o contrário também ocorre. Isso dificulta uma padronização, tornando o processo de decodificação mais complexo.

Para se verificar bits são usadas operações bit-a-bit. A operação **and**, denotada em C++ como &, e deslocamentos, denotados como « e », são utilizados primariamente para verificação de bits. Deve-se, a princípio, indexar as instruções, diminuindo o escopo da decodificação gradativamente. Então, é possível se utilizar de bits de índice, na sua maioria marcado como cinza na tabela. Ao indexar as instruções pelo índice é possível diminuir o escopo para poucas instruções, facilitando bastante na decodificação. Após varrer todos os bits de índice, deve-se verificar o bit *Op*, que mostra de qual instrução específica se trata aquele código de operação. Ao descobrir qual a instrução, deve-se então descobrir seus operandos, que podem ser registradores e listas de registradores, imediatos e endereços de memória.

Como exemplo de decodificação tem-se a instrução 0x9BFF. Para fazer a decodificação são utilizadas operações bit-a-bit, então é interessante passar de hexadecimal para binário para a decodificação manual. Com 0b1001_1011_1111_1111, deve-se indexar os 4 primeiros bits. Logo, 0b1001_1011_1111_1111 AND 0b1111_0000_0000_0000 irá retornar o valor do índice da instrução. Como as operações são não destrutivas, realizar deslocamentos ajuda na leitura do código, então (0b1001_1011_1111_1111 » 12) AND 0b1111. Logo, pode ser tanto a instrução STR quanto LDR. Após descobrir os índices, é necessário descobrir se é a instrução mais a esquerda ou a instrução mais a direita(veja as imagens da tabela) utilizando o bit *Op*: se for totalmente ativado, então é a instrução mais a direita. Com o bit 11 referente a *Op* ativado, tem-se que a instrução é referente a LDR. Para descobrir os operandos é o mesmo processo: comparar bits, e comparar com a tabela. Um operando se trata de um imediato, que deve ser multiplicado por quatro, e é denotado por uma cerquilha, e o outro de um registrador, de r0 a r7. O produto final dessa decodificação é **LDR r3, [sp, #1020]**.

Instruction classes (indexed by *op*)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	*op*	*immed5*					*Lm*			*Ld*			`LSL | LSR`
0	0	0	1	0	*immed5*					*Lm*			*Ld*			`ASR`
0	0	0	1	1	0	*op*	*Lm*			*Ln*			*Ld*			`ADD | SUB`
0	0	0	1	1	1	*op*	*immed3*			*Ln*			*Ld*			`ADD | SUB`
0	0	1	0	*op*	*Ld/Ln*			*immed8*								`MOV | CMP`
0	0	1	1	*op*	*Ld*			*immed8*								`ADD | SUB`
0	1	0	0	0	0	0	0	*op*		*Lm/Ls*			*Ld*			`AND | EOR | LSL | LSR`
0	1	0	0	0	0	0	1	*op*		*Lm/Ls*			*Ld*			`ASR | ADC | SBC | ROR`
0	1	0	0	0	0	1	0	*op*		*Lm*			*Ld/Ln*			`TST | NEG | CMP | CMN`
0	1	0	0	0	0	1	1	*op*		*Lm*			*Ld*			`ORR | MUL | BIC | MVN`
0	1	0	0	0	1	1	0	0	0	*Lm*			*Ld*			`CPY Ld, Lm`
0	1	0	0	0	1	*op*	0	0	1	*Hm* & 7			*Ld*			`ADD | MOV Ld, Hm`
0	1	0	0	0	1	*op*	0	1	0	*Lm*			*Hd* & 7			`ADD | MOV Hd, Lm`
0	1	0	0	0	1	*op*	0	1	1	*Hm* & 7			*Hd* & 7			`ADD | MOV Hd, Hm`
0	1	0	0	0	1	0	1	0	1	*Hm* & 7			*Ln*			`CMP`
0	1	0	0	0	1	0	1	1	0	*Lm*			*Hn* & 7			`CMP`
0	1	0	0	0	1	0	1	1	1	*Hm* & 7			*Hn* & 7			`CMP`
0	1	0	0	0	1	1	1	*op*	*Rm*				0	0	0	`BX | BLX`
0	1	0	0	1	*Ld*			*immed8*								`LDR Ld, [pc, #immed*4]`
0	1	0	1	0	*op*		*Lm*			*Ln*			*Ld*			`STR | STRH | STRB | LDRSB` *pre*
0	1	0	1	1	*op*		*Lm*			*Ln*			*Ld*			`LDR | LDRH | LDRB | LDRSH` *pre*
0	1	1	0	*op*	*immed5*					*Ln*			*Ld*			`STR | LDR Ld, [Ln, #immed*4]`
0	1	1	1	*op*	*immed5*					*Ln*			*Ld*			`STRB | LDRB Ld, [Ln, #immed]`
1	0	0	0	*op*	*immed5*					*Ln*			*Ld*			`STRH | LDRH Ld, [Ln, #immed*2]`

Instruction classes (indexed by *op*)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	1	*op*	*Ld*			*immed8*								`STR | LDR Ld, [sp, #immed*4]`
1	0	1	0	*op*	*Ld*			*immed8*								`ADD Ld, pc, #immed*4 | ADD Ld, sp, #immed*4`
1	0	1	1	0	0	0	0	*op*	*immed7*							`ADD sp, #immed*4 | SUB sp, #immed*4`
1	0	1	1	0	0	1	0	*op*		*Lm*			*Ld*			`SXTH | SXTB | UXTH | UXTB`
1	0	1	1	1	0	1	0	*op*		*Lm*			*Ld*			`REV | REV16 | | REVSH`
1	0	1	1	*op*	1	0	*R*	*register_list*								`PUSH | POP`
1	0	1	1	0	1	1	0	0	1	0	1	*op*	0	0	0	`SETEND LE | SETEND BE`
1	0	1	1	0	1	1	0	0	1	1	*op*	0	*a*	*i*	*f*	`CPSIE | CPSID`
1	0	1	1	1	1	1	0	*immed8*								`BKPT immed8`
1	1	0	0	*op*	*Ln*			*register_list*								`STMIA | LDMIA Ln!,{register-list}`
1	1	0	1	*cond* < 1110				signed 8-bit offset								`B<cond> instruction_address+4+offset*2`
1	1	0	1	1	1	1	0	*x*								Undefined and expected to remain so
1	1	0	1	1	1	1	1	*immed8*								`SWI immed8`
1	1	1	0	0	signed 11-bit *offset*											`B instruction_address+4+offset*2`
1	1	1	0	1	unsigned 10-bit *offset*										0	`BLX ((instruction+4+(poff<<12)+offset*4) &~ 3)` This must be preceded by a branch prefix instruction.
1	1	1	1	0	signed 11-bit prefix offset *poff*											This is the branch prefix instruction. It must be followed by a relative `BL` or `BLX` instruction.
1	1	1	1	1	unsigned 11-bit *offset*											`BL instruction+4+ (poff<<12)+offset*2` This must be preceded by a branch prefix instruction.

2.3.2 Padronização de Código

O código foi implementado no formato de três arquivos: um arquivo de cabeçalho, um arquivo de funções e uma função principal.

Na função principal, onde os parâmetros são passadas antes da execução, são verificados erros no arquivo de entrada, assim como a preparação dos arquivos e das instruções a serem decodificadas. Primeiramente, é verificado o número de parâmetros: se mais de três parâmetros foram passados, ou menos de dois, então há um erro. Com a chamada do programa como primeiro parâmetro, no mínimo deve haver um segundo parâmetro para entrada e no máximo um terceiro para saída. Caso o arquivo não exista, ou a extensão seja inválida, diferente de *.in, então houve um erro. Quando há algum erro no programa, o usuário é avisado por meio do terminal:

```
(base) pedrobotelho15@botelhoDevStation:~/Projetos/Projetos em C++/thumblator$ ./bin/thumblator io_files/program.in
Thumblator: Fatal error: invalid instruction deff at line 8.
Aborting.
(base) pedrobotelho15@botelhoDevStation:~/Projetos/Projetos em C++/thumblator$ ./bin/thumblator io_files/program
Thumblator: Fatal error: file doesn't exist.
Aborting.
(base) pedrobotelho15@botelhoDevStation:~/Projetos/Projetos em C++/thumblator$ ./bin/thumblator io_files/program.out
Thumblator: Fatal error: invalid extension.
(base) pedrobotelho15@botelhoDevStation:~/Projetos/Projetos em C++/thumblator$ ./bin/thumblator
Thumblator: Fatal error: no input file.
Aborting.
(base) pedrobotelho15@botelhoDevStation:~/Projetos/Projetos em C++/thumblator$ ./bin/thumblator io_files/program.out io_files/program.out program.out
Thumblator: Fatal error: too much parameters.
Aborting.
```

O programa utiliza algumas classes para facilitar o desenvolvimento, como *ifstream* e *ofstream* para arquivos de saída, bem como *string* e *stringstream* para manejo de *strings*. Os dados do arquivo de entrada sãoi passados para uma *string* por meio de uma *stringstream*. Será nessa *string* que ocorrerá a decodificação. É feito um laço para remoção de espaços desnecessários, e então é passado cada linha da entrada pra uma *string*, para que possa ser tratada de maneira individual, aplicando o algoritmo demonstrado. Ao final, as instruções são passadas para um arquivo de saída, caso não haja nenhum erro, e seu nome é decidido baseado nos parâmetros passados pelo usuário. Cada linha, no arquivo de saída, tem o *opcode* em Hexadecimal, dois pontos e seu mnemônico correspondente.

As funções **stringHex** e **decodeInstructions** tem papel fundamental no código. A primeira diz respeito a quando ocorre um erro de instrução inexistente, retornando seu código para ser mostrado ao usuário, e a segunda realiza o procedimento de decodificação de instruções. Caso a função **decodeInstructions** encontre uma instrução inválida ou algum erro, é mostrado ao usuário a linha, a instrução inválida e o erro ocorrido.

Foram utilizadas medidas de padronização, como o operador &(AND), » e «(deslocamento) para manuseio de bits a uma maior velocidade. A utilização de *string* facilita o desenvolvimento. A utilização de switch-case diminui a quantidade de if's, que tendem a diminuir o desempenho do programa. A utilização de apenas duas funções foi para diminuir a quantidade de funções externas e de saltos.

3 Conclusão

Este trabalho foi bastante útil no que diz respeito a prática no manuseio do código *Thumb* (instruções codificadas com 16 bits), bem como a decodificação de um mapa de memória para o respectivo código. Tendo em vista que o código *Thumb* é capaz de reduzir o tamanho de um código ARM e aumentar consideravelmente o seu desempenho, é notória a importância desse trabalho para os alunos de Engenharia de Computação já que se trata de uma disciplina muito importante para o curso.

A experiência no presente trabalho foi desafiadora e ao mesmo tempo divertida, já que foi preciso estudar algumas instruções pouco conhecidas bem como aprender a decodificá-las, além do que foi possível compreender melhor o funcionamento do código *Thumb* na arquitetura ARM. Sendo assim, a realização deste trabalho foi bastante proveitosa.

Referências

[1] Feitosa P. Andrade I. Botelho, P. *Repositório Thumblator*.

[2] Sloss A. Wright C. Rayfield J. Symes, D. *ARM System Developer's Guide*, volume 1. Elsevier, 2004.