Processo de Tradução de Instruções Assembly do MIPS[1]

Vitor Bueno de Camargo[1]

Universidade Tecnológica Federal do Paraná – UTFPR

COCIC – Coordenação do Curso de Bacharelado em Ciência da Computação

Campo Mourão, Paraná, Brasil

[1]vitorcamargo@alunos.utfpr.edu.br

Resumo

O seguinte trabalho consiste no desenvolvimento de um programa que decodifica instruções binárias para instruções do Assembly do MIPS, assim como o seu caminho reverso, codificando Assembly do MIPS para instruções binárias, promovendo um entendimento sobre conceitos de compilação de algoritmos em linguagens de alto nível como código de máquina. O trabalho também abordará a transformação de um código na linguagem c em instruções Assembly do MIPS.

1. Introdução

A arquitetura de um computador é uma descrição lógica de seus componentes e suas operações básicas. Na linguagem de montagem pura, uma instrução de linguagem *assembly* corresponde a uma operação básica do processador. Quando um programador escreve em linguagem *assembly*, o programador está solicitando as operações básicas do processador. A arquitetura do processador é visível em todas as instruções do programa.

Linguagem de montagem pura é rara. A maioria dos programas aplicativos é escrita em uma linguagem de alto nível. Mesmo quando a linguagem *assembly* é usada, ela geralmente foi aprimorada. Recursos são adicionados a ele para torná-lo mais amigável ao programador. Ou seja, dentro de linguagens de alto nível, como C ou Java, existe uma alta independência sobre o processador, resultando numa facilidade de entendimento do código pelos programadores, mas baixo entendimento do próprio computador.

Para isso é necessário um sistema que transforma códigos em alto nível em linguagem de máquina, a qual é entendida pelo processador.

No artigo será apresentado as etapas de transformação de um código, juntamente com a explicação sobre a implementação de um decodificador, codificador e a transformação de um código em C para MIPS.

2. Processo de Transformação

Existem 3 tipos de tradutores de programas: compilador, intérprete e montador.

Um Compilador responsável por traduzir o código escrito em uma linguagem de alto nível para uma linguagem de nível inferior, código de máquina. O motivo mais comum para traduzir o código-fonte é criar um programa executável (convertendo de uma linguagem de alto nível em linguagem de máquina).

Um montador traduz linguagem de montagem em código de máquina. A linguagem de montagem consiste em mnemônicos para *opcodes* de máquina, de forma que os montadores executam uma conversão de 1:1 de mnemônicos para uma instrução direta.

Um programa de intérprete executa outros programas diretamente, executando o código do programa e executando-o linha por linha. Como ele analisa cada linha, um interpretador é mais lento do que executar o código compilado, mas pode levar menos tempo para interpretar o código do programa do que para compilá-lo e executá-lo, isso é muito útil ao criar protótipos e testar códigos. Os intérpretes são escritos para múltiplas plataformas, isto significa que o código escrito uma vez pode ser executado imediatamente em sistemas diferentes sem ter que recompilar para cada um.

Dentro da aplicação do trabalho, é possível ver que o decodificador assume o papel de um montador ao contrário, ou seja, ele traduz o código de máquina em instruções em linguagem de montagem, enquanto o codificador assume o papel central do montador. Além deles, ao transformar um código de uma linguagem C para MIPS estamos fazendo na mão o que um compilador faz.

[1]Trabalho desenvolvido para a disciplina de BCC33B – Arquitetura e Organização de Computadores

3. Decodificador

A implementação inicia com a declarações de máscaras e registradores usados no sistema, assim como mostra a Figura 3.1.

```
// Campos do formato de instrução.
// opcode  := ir(31 downto 26)
// rs      := ir(25 downto 21);
// rt      := ir(20 downto 16);
// rd      := ir(15 downto 11);
// shamt   := ir(10 downto  6);
// imm     := ir(15 downto  0);
// address := ir(25 downto  0);

/* Definição das Máscaras. */
int mascaraOpCode     = 0xFC000000;
int mascaraRs         = 0x03E00000;
int mascaraRt         = 0x001F0000;
int mascaraRd         = 0x0000F800;
int mascaraShamt      = 0x000007C0;
int mascaraFunct      = 0x0000003F;
int mascaraImmediate  = 0x0000FFFF;
int mascaraAddress    = 0x03FFFFFF;

char* registerName[32] = {"$zero", "$at", "$v0", "$v1", "$a0", "$a1", "$a2",
                          "$a3", "$t0", "$t1", "$t2", "$t3", "$t4", "$t5",
                          "$t6", "$t7", "$s0", "$s1", "$s2", "$s3", "$s4",
                          "$s5", "$s6", "$s7", "$t8", "$t9", "$k0", "$k1",
                          "$gp", "$sp", "$fp", "$ra"};
```

Figura 3.1: Declaração de máscaras e registradores

Essas máscaras serão usadas para receber o valor de uma determinada parte da instrução, assim como o vetor possui em suas posições um respectivo registrador que será utilizado dependendo da instrução.

```
/* Função que recupera o campo OpCode. */
unsigned int getOpCode(unsigned int ir) {
    unsigned int opcode = ((ir & mascaraOpCode) >> 26);
    return opcode;
}

/* Função que recupera o campo registrador Rs. */
unsigned int getRs(unsigned int ir) {
    unsigned int rs = (ir & mascaraRs) >> 21;
    return rs;
}

/* Função que recupera o campo registrador Rt. */
unsigned int getRt(unsigned int ir) {
    unsigned int rt = (ir & mascaraRt) >> 16;
    return rt;
}

/* Função que recupera o campo registrador Rd. */
unsigned int getRd(unsigned int ir) {
    unsigned int rd = (ir & mascaraRd) >> 11;
    return rd;
}
```

Figura 3.2: Funções básicas do decodificador

Na Figura 3.2 é possível ver o uso do vetor de registradores e as máscaras dentro do sistema. Por exemplo, na função getOpCode, é devolvido de forma decimal o valor da instrução junto com a máscara do *opcode*, nesse caso a máscara serve para zerar todos os números da instrução exceto os 6 primeiros dígitos.

Na função getRs, assim como no *opcode*, é devolvido um valor decimal da posição 7 a 11 (valor do rs na instrução binária). Com esse valor determinado, podemos colocar no vetor dos registradores e descobrir qual é o nome do registrador.

Ao todo, foram catalogadas 104 instruções diferentes para o decodificador, cada caso possui seu tipo (*R-type, I-type, J-type*) e sua especificação própria, ilustrada na Figura 3.3:

```
/* Decodificação das instruções. */
void decodificar(unsigned int ir) {
    switch (getOpCode(ir)) {
        case 0: { // 000000, Aritmética
            switch (getFunct(ir)) {
                case 0: { // nop
                    if(getRd(ir) == 0 && getRs(ir) == 0 && getRt(ir) == 0) {…
                    } else { // 000000 -> sll, R-Type. sll rd, rt, shamt…
                    }
                    break;
                }
                case 2: { // 000010 -> srl, R-Type. srl rd, rt, shamt
                    printf("srl ");
                    printf("%s, ", registerName[getRd(ir)]);
                    printf("%s, ", registerName[getRt(ir)]);
                    printf("%d\n", getShamt(ir));
                    break;
                }
                case 3: { // 000011 -> sra, R-Type. sra rd, rt, shamt…
                }
                case 4: { // 000100 -> sllv, R-Type. sllv rd, rt, rs…
                }
                case 6: { // 000110 -> srlv, R-Type. srlv rd, rt, rs…
                }
                case 7: { // 000110 -> srav, R-Type. srav rd, rt, rs…
                }
                case 8: { // 001000 -> jr, R-Type. jr rs…
                }
                case 9: { // 001001 -> jalr, R-Type. jalr rd, rs…
                }
                case 10: { // 001010 -> movz, R-Type. movz rd, rs, rt…
                }
```

Figura 3.3: Função de decodificação

O programa funciona a partir da função principal (decodificar) e devolve na tela ou em um arquivo o resultado, o qual é recebido por um arquivo. Ele pega linha por linha do arquivo e devolve a instrução em MIPS correspondente. Por exemplo, em uma instrução binária qualquer, o programa pegará o *opcode* do mesmo, caso seja 0, o programa sabe que se trata de uma instrução *R-Type* e analisa em seguida a sua *funct*, supondo a partir da figura acima que exista uma função srl (*opcode*: 0 e *funct*: 2), o software devolve o nome da função (“srl”), o registrador em rd, registrador em rt e o valor que está em *shamt*. Formando a estrutura da função: srl rd, rt, *shamt*.

4. Codificador

Ao contrário do decodificador, o codificador deve ver o nome da função e sua especificação e devolver o valor binário da mesma. A lógica utilizada é basicamente a mesma.

A ideia é ter funções bases que pegam valores dentro da instrução, como, o nome da função e as informações que vem depois dela (quase sempre sendo registradores) e usar o strcmp para comparar esses valores com cada uma das 104 instruções catalogadas e devolver o valor de cada determinada instrução.

Figura 4.1: Declaração base de registradores

Assim como ilustrada acima, vemos a declaração de um vetor de registradores, mas ao invés de ter seu nome, possui o valor em binário de cada registrador.

```c
/* Função que devolve o nome da função. */
unsigned char* getFunction(char* linha) {
    char *linhaCpy = malloc(strlen(linha));
    strcpy(linhaCpy, linha);

    strtok(linhaCpy, " ");

    return linhaCpy;
}

/* Função que devolve o registrador em uma posição da linha. */
unsigned int getReg(char* linha, int pos) {
    char *pc, *linhaCpy = malloc(strlen(linha));
    strcpy(linhaCpy, linha);
    int i;

    pc = strtok(linhaCpy, ", $()");
    for(i = 0; i < pos; i++)
        pc = strtok(NULL, ", $()");

    for(i = 0; i < 32; i++)
        if(strcmp(registerName[i], pc) == 0)
            return i;
}
```

Figura 4.2: Funções bases do codificador

Na Figura 4.2 podemos ver como é o comportamento básico das funções do programa, no caso de getFunction, o sistema recebe a linha a ser analisada e coma ajuda da função *strtok* devolve a palavra até o primeiro espaço, ou seja, o nome da função.

Enquanto na função getReg, é recebida não só a linha, mas também qual é a posição do registrador, ou seja, se ele está logo após do nome (pos: 1), no meio (pos: 2) ou no final da linha (pos: 3). Depois de pegar o nome do registrador em questão, é feita uma comparação com cada um dos registradores dentro do segundo vetor da Figura 4.1, quando encontra o registrador igual, a função devolve a posição do mesmo, que pode ser usado no outro vetor, imprimindo, no fim, o valor binário do registrador.

A partir da Figura 4.3, vemos o funcionamento da função principal do programa (codifica), a qual inicialmente pega o nome da instrução e encontra por meio de if's a especificação correspondente. Após encontrar sua correspondência e entrar no if, o software exibe de forma direta ou indireta o resultado em binário.

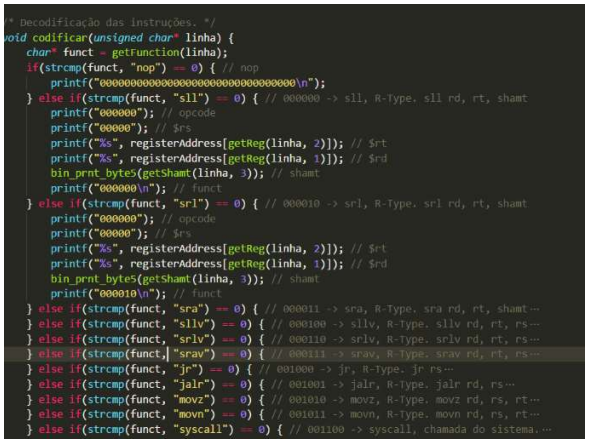

Figura 4.3: Função de codificação

Em funções como *nop*, *syscall* e *break*, o software devolve diretamente o valor completo da função, isso acontece porque não existe mudança dentro da instrução, como o registrador, endereço ou *shamt*.

Usando o mesmo exemplo presente na seção 3, imaginamos que exista uma linha com o nome de "slt", neste caso, o sistema encontra o if com a condição que validará por meio da subfunção *strcmp* e entrará no 3º if. Nele, será exibido o valor do *opcode*, sabendo que a função é um *R-Type* (*opcode*: 0), devolve também rs como 0 já que é usado como padrão para essa instrução e busca qual é o código binário correspondente para os registradores rt e rd, assim como o valor de shamt e por fim o valor binário para a funct.

5. Tradutor C - MIPS

A última etapa do projeto é a tradução de um código na linguagem C para *Assembly* do MIPS, no qual consiste em aplicar conhecimentos adquiridos em sala para criar uma aplicação na IDE Mars que corresponda com o código em C.

A escolha do programa em C deve se basear em um código que contenha estruturas de repetição (for, while), estruturas de condição (if else, switch case) e estruturas complexas (chamada de função e recursividade). Um bom exemplo a ser escolhido é algum algoritmo de ordenação.

Para este projeto foi escolhido uma variação do algoritmo de heapSort. O código é chamado isMaxHeap e possui a seguinte proposta: "Escreva uma função recursiva que recebe 1 vetor e seu tamanho, e um índice. O retorno deve ser 1 se esse vetor representa um max Heap, ou 0, caso contrário

a. Se o índice for de um nó folha, é maxHeap (percorreu todos os nós não folha e todos eram)

b. Se o índice não é nó folha, verifica se o elemento é maior que os 2 filhos, e chama recursivamente a função para os filhos.”

Baseado no exercício, foi montado o seguinte código:

```c
#include <stdio.h>

int v[7];

int isMaxHeap(int* v, int n, int i) {
    if(i >= n/2) return 1;
    else if(v[i] > v[2*i+1] && v[i] > v[2*i+2]) isMaxHeap(v, n, i+1);
    else return 0;
}

void main() {
    int i, j;
    for(i = 0; i < 7; i++)
        scanf("%d", &v[i]);

    printf("Digite a posição a ser verificada: ");
    scanf("%d", &j);

    printf("%d", isMaxHeap(v, 7, j));
}
```

Figura 5.1: Código base escolhido

A partir do algoritmo criado podemos visualizar sua versão em *Assembly* do MIPS ilustrada na Figura 5.2.

```
.data
        v: .space 0
        msg: .asciiz "Digite a posição a ser verificada: "
.text
                j main

isMaxHeap:      nop
                addi $sp, $sp, -32
                sw $ra, 28($sp)
                sw $fp, 24($sp)
                move $fp, $sp

                sw $a1, 36($fp)         # Armazena n na memória
                sw $a2, 40($fp)         # Armazena i na memória

                lw $t0, 36($fp)         # Armazena em $t0 o valor n
                srl $t0, $t0, 1         # Faz n/2
                lw $t1, 40($fp)         # Armazena em $t1 o valor de i

                bgt $t0, $t1, else_if   # Se (n/2) > i -> Vai para else_if
                addi $v0, $zero, 1      # Armazena resultado em $v0
                j desaloca

else_if:        lw $t1, 40($fp)         # Armazena em $t1 o valor de i
                sll $t1, $t1, 2         # Caarrega em $t1 a posição i correspondente na memória
                lw $t1, v($t1)          # Carrega em $t1 o valor de v[i]

                lw $t2, 40($fp)         # Armazena em $t2 o valor de i
                sll $t2, $t2, 1         # i*2
                addi $t2, $t2, 1        # (i*2)+1
                sll $t2, $t2, 2         # Caarrega em $t2 a posição i correspondente na memória
                lw $t2, v($t2)          # Carrega em $t2 o valor de v[(i*2)+1]
```

Figura 5.2: Parte do algoritmo em MIPS

Na Figura 5.2, é possível analisar somente um pedaço do arquivo completo.

6. Resultados Obtidos

Para o decodificador foi utilizado um arquivo teste com um conjunto de instruções, o ideal é que o resultado obtido seja usado como base para o codificador, obtendo, por fim, o mesmo conjunto de instruções binárias utilizadas no início.

Já para a tradução C-MIPS é necessário utilizar o mesmo vetor no algoritmo em C e em MIPS, afim de encontrar a mesma resposta nos dois códigos.

```
10101111101111100000000000000000
00000000000100010100000000100000
00000000000000000000000000100000
00000000000000000000000000000010
00000010010100111000100000100010
10001101001010000000000110010000
00000010011100110100000000100010
10101101001010000000000110010000
10001100000100110000000000000100
10101100000011000000000000001100
00001000000100000000000011111111
vitor@VITOR-PC: /mnt/c/Users/vitor/Docs/UTFPR/3rd Period/Arquitetura e Organização de Computadores/APS A/BCC_BCC
vitor@VITOR-PC: /mnt/c/Users/vitor/Docs/UTFPR/3rd Period/Arquitetura e Organização de Computadores/APS A/BCC_BCC/vitor_1921059/Implementacao/decod-mips$ ./decodificador exemplo.b
sw $ra, 0($sp)
add $t0, $zero, $s1
add $zero, $zero, $zero
srl $zero, $zero, 0
sub $s1, $s2, $s3
lw $t0, 400($t1)
add $t0, $s2, $t0
sub $t0, $s7, $s3
sw $t0, 400($t1)
lw $s3, 4($zero)
sw $t4, 12($zero)
j 0x001000FF
vitor@VITOR-PC: /mnt/c/Users/vitor/Docs/UTFPR/3rd Period/Arquitetura e Organização de Computadores/APS A/BCC_BCC/vitor_1921059/Implementacao/decod-mips$
```

Figura 6.1: Arquivo de teste e resultado do decodificador

Assim, como mostrado acima (Figura 6.1), obtemos um resultado para o arquivo teste e usaremos o mesmo resultado como arquivo teste para o codificador.

```
sw $ra, 0($sp)
add $t0, $zero, $s1
add $zero, $zero, $zero
srl $zero, $zero, 0
sub $s1, $s2, $s3
lw $t0, 400($t1)
add $t0, $s2, $t0
sub $t0, $s7, $s3
sw $t0, 400($t1)
lw $s3, 4($zero)
sw $t4, 12($zero)
j 0x01000FF
vitor@VITOR-PC: /mnt/c/Users/vitor/Docs/UTFPR/3rd Period/Arquitetura e Organização de Computadores/APS A/BCC_BCC/vitor_1921059/Implementacao/decod-mips$ ./codificador exemplo.txt
10101111101111100000000000000000
00000000000100010100000000100000
00000000000000000000000000100000
00000000000000000000000000000010
00000010010100111000100000100010
10001101001010000000000110010000
00000010011100110100000000100010
10101101001010000000000110010000
10001100000100110000000000000100
10101100000011000000000000001100
00001000000100000000000011111111
vitor@VITOR-PC: /mnt/c/Users/vitor/Docs/UTFPR/3rd Period/Arquitetura e Organização de Computadores/APS A/BCC_BCC/vitor_1921059/Implementacao/decod-mips$
```

Figura 6.2: Arquivo de teste e resultado do codificador

Por fim, a partir do resultado mostrado na Figura 6.2, encontramos o mesmo conjunto binário de instruções usadas no início dos testes, atestando a veracidade de ambos programas.

Agora, para testar a tradução do algoritmo isMaxHeap, usaremos o seguinte vetor: [1,2,3,4,5,6,7], e verificaremos na posição 0.

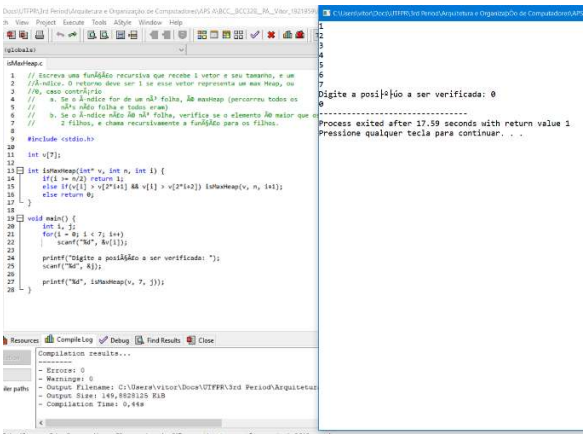

Figura 6.3: Resultado obtido do código em C

Assim como esperado (mostrado na Figura 6.3), o programa resultou em 0, ou seja, o vetor não está ordenado na forma de um Heap.

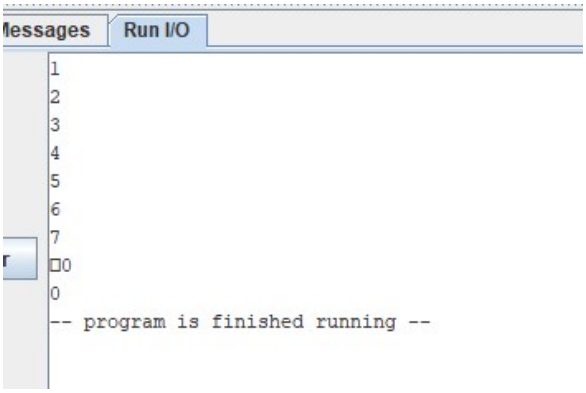

Figura 6.4: Resultado obtido do código traduzido

Logo, a partir da Figura 6.4, podemos concluir que o código traduzido traz a mesma resposta para o mesmo parâmetro apresentado no código original em C.

Nota-se também a presença de um *bug* no código em MIPS, não é possível ver o sistema exibindo o texto “Digite a posição a ser verificada:”, ao invés disso, aparece para o usuário um quadrado, contudo, a falta dessa exibição não atrapalha no projeto como um todo.

7. Referências

https://pt.wikibooks.org/wiki/Introdu %C3%A7%C3%A3o_ %C3%A0_Arquitetura_de_Computadores/Instru %C3%A7%C3%B5es_do_MIPS
http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIP Sir.html
https://www.eg.bucknell.edu/~csci320/mips_web/
http://courses.missouristate.edu/KenVollmar/mars/
http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.htm l