Decodificador MIPS

Henrique Souza Marcuzzo[1], Matheus Henrique Batistela[2]

Universidade Tecnológica Federal do Paraná – UTFPR

COCIC – Coordenação do Curso de Bacharelado em Ciência da Computação
Campo Mourão, Paraná, Brasil

[1]henriquemarcuzzo@alunos.utfpr.edu.br
[2]matheushenriquebatistela@gmail.com

Resumo

O presente projeto tem como objetivo aprimorar o conhecimento sobre a linguagem de máquina, utilizando uma linguagem estruturada de alto nível, C. O desenvonvimento do trabalho é baseado no processo de tradução de uma sequência binária (Linguagem de máquina) para uma instrução em linguagem de montagem (Assembly).

1. Introdução

As linguagens utilizadas para se programar são classificadas por nível de abstração, no caso de C, Python e C++ são linguagens de alto nível, o nível baixo de abstração é representado pelas linguagens de montagem (Assembly), que se aproximam do nível mais baixo, que são as instruções em binário.

Tanto as linguagens de alto nível quanto as linguagens de montagem, para serem executadas, são traduzidas para instruções em binário para que então sejam processadas pelo computador.

Este relatório tem como intuito apresentar um decodificador implementado na linguagem C, no qual tradu-zirá um código em binário gerado no simulador MARS para que a partir dos bits possa se chegar nas instruções na linguagem do MIPS .

2. Contexto do Algoritmo

Inicialmente o decodificador abre o arquivo passado por parâmetro durante a execução do programa, e armazena a primeira linha numa variável inteira sem sinal. Logo após, chama a função “decodificar”, a qual utiliza as funções “getOpCode”, “getRs”, “getRt”, “getRd”, “getShamt”, “getFunct”, “getImmediate” e “getAddress” para traduzir o código em binário para as instrução em assembly, seguindo sua sintaxe.

3. Processo de Tradução

Na linguagem MIPS, existem 3 tipos básicos de intruções, as do tip R, I e J.

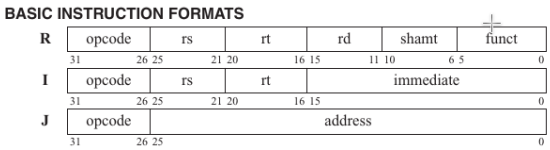

Figura 1 – Tipos de instruções MIPS.

Como podemos observar, cada instrução possui uma forma difirente de distribuição dos bits, e para identificar qual a função e o seu tipo, utilizamos o opcode. No programa, temos a função “getOpCode”, representada na Figura 2, a qual a partir de uma máscara de bits pré-definida, retorna o opcode da instrução passada por parâmetro com seu valor no sistema de numeração decimal.

```
63  /* Função que recupera o campo OpCode. */
64  unsigned int getOpCode(unsigned int ir) {
65      PRINT_FUNC_NAME;
66      unsigned int opcode = ((ir & mascaraOpCode) >> 26);
67      TRACE("Opcode: %u\n ", opcode);
68      return opcode;
69  }
```

Figura 2 – Código da função “getOpCode” em C.

As demais funções com o prefixo “get”, funcionam de forma similar à esta citada acima, porém retornando diferentes resultados associados a diferentes máscaras de bits.

```c
/* Função que recupera o campo registrador Rs. */
unsigned int getRs(unsigned int ir) {
    PRINT_FUNC_NAME;
    unsigned int rs = (ir & mascaraRs) >> 21;
    TRACE("Rs: %u\n ", rs);
    return rs;
}
```

Figura 3 – Código da função “getRs” em C.

```c
/* Função que recupera o campo registrador Rt. */
unsigned int getRt(unsigned int ir) {
    PRINT_FUNC_NAME;
    unsigned int rt = (ir & mascaraRt) >> 16;
    TRACE("Rt: %u\n ", rt);
    return rt;
}
```

Figura 4 – Código da função “getRt” em C.

```c
/* Função que recupera o campo registrador Rd. */
unsigned int getRd(unsigned int ir) {
    PRINT_FUNC_NAME;
    unsigned int rd = (ir & mascaraRd) >> 11;
    TRACE("Rd: %u\n ", rd);
    return rd;
}
```

Figura 5 – Código da função “getRd” em C.

```c
/* Função que recupera o campo imm. */
int getImmediate(unsigned int ir) {
    PRINT_FUNC_NAME;
    unsigned int imm = (ir & mascaraImmediate);
    TRACE("imm: %u\n ", imm);
    return imm;
}
```

Figura 6 – Código da função “getImmediate” em C.

```c
/* Função que recupera o campo Funct. */
unsigned int getFunct(unsigned int ir) {
    PRINT_FUNC_NAME;
    unsigned int funct = (ir & mascaraFunct);
    TRACE("funct: %u\n ", funct);
    return funct;
}
```

Figura 6 – Código da função “getFunct” em C.

```c
/* Função que recupera o campo Address. */
unsigned int getAddress(unsigned int ir) {
    PRINT_FUNC_NAME;
    unsigned int address = (ir & mascaraAddress);
    TRACE("address: %u\n ", address);
    return address;
}
```

Figura 7 – Código da função “getAddress” em C.

```c
/* Função que recupera o campo Shamt (deslocamento). */
int getShamt(unsigned int ir) {
    PRINT_FUNC_NAME;
    unsigned int shamt = (ir & mascaraShamt) >> 6;
    TRACE("Shamt: %u\n ", shamt);
    return shamt;
}
```

Figura 8 – Código da função “getShamt” em C.

Por fim, a função “decodificar” é composta por um switch case que tem como parâmetro o opcode retornado pela função da Figura 2, assim, cada caso foi implementado com a sintaxe correta para “printar” na tela a instrução referente.

```c
case 33: {  // ADDU -- Add unsigned, Description: Adds two
            // Syntax: addu $d, $s, $t
            // 0000 00ss ssst tttt dddd d000 0010 0001
    fprintf(stdout, "addu ");
    fprintf(stdout, "%s, ", registerName[getRd(ir)]);
    fprintf(stdout, "%s, ", registerName[getRs(ir)]);
    fprintf(stdout, "%s\n", registerName[getRt(ir)]);
    break;
}
```

Figura 9 – Exemplo da instrução ADDU do tipo R.

```c
case 35: { // 100011 (lw), I-Type. lw rt, imm(rs)
    fprintf(stdout, "lw ");
    fprintf(stdout, "%s, ", registerName[getRt(ir)]);
    fprintf(stdout, "%d", getImmediate(ir));
    fprintf(stdout, "(%s)\n", registerName[getRs(ir)]);
    break;
}
```

Figura 10 – Exemplo da instrução LW do tipo I.

```c
case 3: { // 000011 -> jal , J-Type
    fprintf(stdout, "jal ");
    fprintf(stdout, "0x%0.8X\n", getAddress(ir));
    break;
}
```

Figura 11 – Exemplo da instrução JAL do tipo J.

4. Conclusões

Ao final deste trabalho foi possível compreender o funcionamento de tradução de linguagens de alto nível para baixo nível, no caso de C a binário, passando pelo Assembly. Foi possível analisar o comportamento das intruções, seus formatos, tamanhos, registradores e outras estruturas internas de cada uma, como campos de endereço e valores imediatos. Além de perceber como os bits são interpretados pelo computador, desde da comunicação da memória com processador e como são realizadas operações. Conclui-se que o contato com o baixo nível de interpretação é indispensável para se ter ciência de como os códigos de alto nível funcionam e devem ser desempenhados.

5. Referências

[1] MIPS Reference Data Card (“Green Card”),Copyright 2009 by Elsevier, Inc., All rights reserved. From Patterson and Hennessy, Computer Organization and Design, 4th ed.

[2] M. S. University. MARS Simulator. hhttp://courses.
missouristate.edu/kenvollmar/mars/i.