DOCUMENTAÇÃO DO SIMULADOR (BÁSICO) DE ARQUITETURA MIPS

MATHEUS VIEIRA DE SOUZA
DAVID CORINO

PLANALTINA – GO
2012

Sobre o simulador MIPS

O simulador (básico) de arquitetura mips, foi desenvolvido na linguagem de programação C++, utilizando os compiladores Dev-C++ e Microsoft Visual Studio 2010. O simulador de arquitetura MIPS é um software desenvolvido para realizar operações básicas de aritmética através de intruções que contenham 32 bits.

PLANALTINA – GO
2012

DOCUMENTAÇÃO TÉCNICA

Método principal (MAIN)

A função “MAIN” está encarregada de iniciar a execução do programa, chamando primeiramente o método “LOAD” dando sequência a execução do programa.

```
int main()
{
    load();
    int z = 0;
    do
    {
        run();
        z++;
    }
    while(z <= objLINHAS.linhas);
    dump();
    system("pause");
    return 0;
}
```

Classe Memory

Esta classe define quantos espaços de memória serão disponibilizados para armazenas as instruções de 32 bits.

```
#ifndef __MEMORY_H__
#define __MEMORY_H__
#define MEM_SIZE 1024 //Tamanho reservado na memória.

class Memory
{
    public:
            int read(int address);
            void write(int address, int data);
    private:
             int m_array[MEM_SIZE];
};
#endif
```

Classe MIPS

Na classe mips estão localizados os métodos a serem executados, LOAD, FETCH, DECODE, EXECUTE, STEP, RUN e tambem estão localizados os registradores OPCODE, RS, RT, RD, SHIFT AMOUNT e FUNCTION.

```
#include <iostream>
#include <stdlib.h>
#include <string>


using namespace std;

class Mips
{
    public:
    //Métodos principais.
            //Busca a instrução e incrementa o contador.
            void fecth();
            //Extrai os campos da instrução, OP, RS, RD, RT, SHAMT e FUNCT.
            void decode();
            //Executa as intruções.
            void execute();
            //Executa uma-a-uma instrução.
            void step();
            //Executa o código até o final.
            void run();

    //Métodos auxiliares.
            //Mostra o conteúdo dos registradores.
            void dump_regs();
            //Mostra o conteúdo da memória incluindo os endereços.
            void dump_mem(int start, int end, char format);
            //Carrega as instruções de um arquivo de texto para a memória.
            void load();

    //Registradores.
            /*Contador de Programa, Opcode, Registrador fonte 1,
            Registrador fonte 2, Registrador destino, Shift Amont, Função. */
            int pc;
            char ir[32], op[5], rs[4], rt[4], rd[15], shamt[4], funct[5];
            //Registrador de instrução.

    //Variável auxilar para contagem de quantas instruções tem o arquivo.
            int linhas;
};
```

Método LOAD

O método LOAD é responsável por localizar um arquivo de texto com as instruções em binário, ler este arquivo e armazenar em memória (utilizando da classe Memory) cada instrução na sua respectiva posição.

```
void load()
{
    setlocale(LC_ALL, "");
    char caminho[300], inst[32];
    unsigned long int instrucao;

    cout << line << "Digite o endereço (caminho) onde está seu documento "
    << "com as instruções."
    << "\nExemplo: \"C:\\Arquivos de programas\\MIPS\\mips.txt\" \n" << endl;
    cin >> caminho;

    ifstream arquivo;
    arquivo.open(caminho);
    if(arquivo.fail())
    {
        cout << "\n\nErro na abertura do arquivo!!! \n\n" << "Verifique se o "
        << "caminho digitado está correto. \n"
        << "Verifique se o nome do arquivo está correto. \nVerifique se a "
        << "extensão do arquivo "
        << "esta correta." << "\n\n\tPressione qualquer tecla para sair. ";
        getch();
        exit(0);
    }

    int i = 0; //a variável "i" fará a função do PC nesse método.
    while(arquivo.getline(inst, 33))
    {
        instrucao = strtoul(inst, NULL, 2);
        objMemory.write(i, instrucao);
        objLINHAS.linhas = i;
        i+=4;
    }
    arquivo.close();
}
```

Método FETCH

O método FETCH busca em determinada posição de memória o conteúdo da mesma.

```
void fetch()
{
    unsigned long int instrucao1;
    instrucao1 = objMemory.read(pc);
    itoa(instrucao1, objIR.ir, 2);
    pc++;
}
```

Método DECODE

O método DECODE separa a instrução carregada da memória e dependendo do tipo de instrução, e separa a instrução armazenando cada parte nos seus registradores.

```
void decode()
{
    int quantidade;
    quantidade = strlen(objIR.ir);
    if(quantidade < 32);
        complemento(quantidade);

    string s = objIR.ir;
    unsigned long int aux;

    s.copy(objOP.op, 6, 0);
    aux = strtoul(objOP.op, NULL, 2);

    switch(aux)
    {
        case LW:
             s.copy(objRS.rs, 5, 6);
             s.copy(objRT.rt, 5, 11);
             s.copy(objRD.rd, 16, 16);
             break;
        case SW:
             s.copy(objRS.rs, 5, 6);
             s.copy(objRT.rt, 5, 11);
             s.copy(objRD.rd, 16, 16);
             break;
        default:
                s.copy(objRS.rs, 5, 6);
                s.copy(objRT.rt, 5, 11);
                s.copy(objRD.rd, 5, 16);
                s.copy(objSHAMT.shamt, 5, 21);
                s.copy(objFUNCT.funct, 6, 26);
    }
}
```

Método COMPLEMENTO

O método COMPLEMENTO verifica quantos bits há naquela instrução, se houver menos que 32 bits, o método complementa a instrução com a quantidade que falta na instrução com zeros a esquerda, fazendo com que a instrução fique exatamente com 32 bits.

```
void complemento(int quantidade01)
{
     int k = 0;
     char zero[100];
     while(quantidade01 < 32)
     {
          zero[k] = '0';
          quantidade01++;
          k++;
     }
     zero[k] = '\0';
     strcat(zero, objIR.ir);
     strcpy(objIR.ir, zero);
}
```

Método EXECUTE

O método EXECUTE verifica o registrador OP e o registrador FUNCT para saber qual operação deverá executar. Neste método existem cinco tipos de operações, sendo elas: SW (store word), ADD (adição), SUB (subtração), MUL (multiplicação), e DIV (divisão).

```
void execute()
{
    unsigned long int funcao, opcode=0, s0=0, s1=0, s2=0, t0=0, t1=0, t2=0;
    unsigned long int i, HI, LO, C;

    opcode = strtoul(objOP.op, NULL, 2);
    funcao = strtoul(objFUNCT.funct, NULL, 2);

    switch(opcode)
    {
        case SW:
            t0 = strtoul(objRS.rs, NULL, 2);
            C = strtoul(objRD.rd, NULL, 2);
            objMemory.write(C, t0);
            itoa(t0, objRD.rd, 2);
            break;

        case 0:
            switch(funcao)
            {
                case ADD: //Função de adição.
                    s2 = strtoul(objRS.rs,NULL, 2);
                    s1 = strtoul(objRT.rt, NULL, 2);
                    s0 = s2 + s1;
                    itoa(s0, objRD.rd, 2);
                    break;

                case SUB: //Função de subtração.
                    s2 = strtoul(objRS.rs,NULL, 2);
                    s1 = strtoul(objRT.rt, NULL, 2);
                    s0 = s2 - s1;
                    itoa(s0, objRD.rd, 2);
                    break;
                case MUL: //Função de multiplicação.
                    s2 = strtoul(objRS.rs,NULL, 2);
                    s1 = strtoul(objRT.rt, NULL, 2);
                    i = 1;
                    t0 = s2;
                    while(i < s1)
                    {
                        t0 = t0 + s2;
                        i++;
                    }
                    itoa(t0, objRD.rd, 2);
                    break;
```

```
                case DIV: //Função de divisão.
                    s2 = strtoul(objRS.rs,NULL, 2);
                    s1 = strtoul(objRT.rt, NULL, 2);
                    t0 = s2;
                    HI=1;
                    while(1)
                    {
                        t0-=s1;
                        if(t0 <= 1)
                            break;
                        HI++;
                    }
                    s0 = HI;
                    itoa(s0, objRD.rd, 2);
                    break;
            }
    }
}
```

Método RUN e STEP

Os métodos RUN e STEP são parte fundamental para o funcionamento do programa. O método RUN chama o método STEP que por sua vez chama o FETCH, DECODE e EXECUTE até que o método RUN diga que não há mais instruções a serem executadas.

```
// Métodos para realizar busca, decidificação e execução das instruções.

void step()
{
    fetch();
    decode();
    execute();
}

/* Método para realizar os passos do 'step' até que não tenha mais intruções
em memória. */

void run()
{
    step();
}
```

DUMP REGS e DUMP MEM

Estes métodos são responsáveis por mostrar na tela do computador o valor final dos registradores e mostrar a posição e o conteúdo da memória definidos pelo usuário.

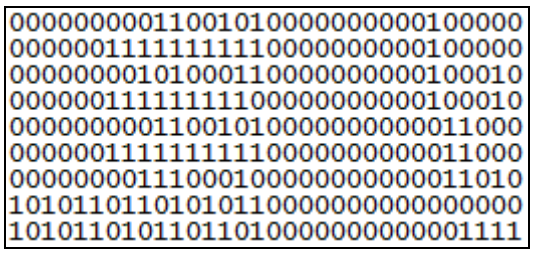
```
void dump_regs()
{
    cout << "Registrador RS: " << objRS.rs << "\nRegistrador RT: "
    << objRT.rt << "\nRegistrador RD: " << objRD.rd
    << "\nRegistrador funct: " << objFUNCT.funct << "\n" << endl;
}

// Método para mostrar o contúdo da memória.

void dump_mem(int start, int end, char format)
{
    int x;
    unsigned long int aux1;
    x = start;
    cout << "\n" << "Posição   Conteúdo \n\n";
    while(x <= end)
    {
        switch(format)
        {
            case 'd':
            case 'D':
                aux1 = objMemory.read(start);
                printf("   %.3ld -- %ld \n", start, aux1);
                break;
            case 'h':
            case 'H':
                aux1 = objMemory.read(start);
                printf("   %.3ld -- %X \n", start, aux1);
                break;
            default:
                cout << line
                << "ATENÇÃO!!! \nVocê digitou uma opção não compatível. "
                << "O programa sera encerrado! \n" << endl;
                getch();
                exit(0);
        }
        start++;
        x++;
    }
    cout << "\n";
}
```

DOCUMENTAÇÃO DE USO

MANUAL BÁSICO PARA O USUÁRIO

Como utilizar o simulador MIPS.

O software é dividido em duas partes: 1º o arquivo de texto com as instruções em binário e 2º o software em si. O usuário devera redigir um arquivo de texto cada instrução com 32 bits em formato binário sem espaço entre os números e com as instruções uma em baixo da outra como mostra a figura abaixo:



As instruções são divididas em partes que serão lidas pelo programa e armazenas no seus respectivos registradores, por exemplo:

INSTRUÇÕES DO TIPO R					
OPCODE	**RS**	**RT**	**RD**	**SHAMT**	**FUNCT**
000000	**00011**	**00101**	**00000**	**00000**	**100000**

	INSTRUÇÕES DO TIPO I		
OPCODE	**RS**	**RT**	**RD**
000000	**00011**	**00101**	**0000000000001111**

O programa realize operações de SW (store word), ADD (adição), SUB (subtração), MUL (multiplicação), e DIV (divisão). Portanto deve ser observar a escrita das instruções no documento de texto conforme mostra as instruções abaixo:

- **ADD** (operação de adição)

  Para essa operação aritmética **OPCODE** sempre será **000000**;
  RS contem um número positivo entre 0 e 31, exemplo: **00011** (em decimal = 3);
  RT contem um número positivo entre 0 e 31, exemplo: **01001** (em decimal = 9);
  RD é onde será armazenado o resultado, então deverá ser **00000**;
  SHAMT não será utilizado nesse tipo de operação, então deve-se deixar **00000**;
  FUNCT é o mais importante pois indica que se trata de uma operação de soma, nesse caso será **100000**.

- **SUB** (operação de subtração)

  Para essa operação aritmética **OPCODE** sempre será **000000**;
  RS contem um número positivo entre 0 e 31, exemplo: **00011** (em decimal = 3);

RT contem um número positivo entre 0 e 31, exemplo: **00001** (em decimal = 1);
RD é onde será armazenado o resultado, então deverá ser **00000**;
SHAMT não será utilizado nesse tipo de operação, então deve-se deixar **00000**;
FUNCT é o mais importante pois indica que se trata de uma operação de subtração, nesse caso será **100010**.
OBS: caso o **RS** seja menor que **RT** ocorrerá um erro e o programa será finalizado.

- **MUL** (operação de multiplicação)

  Para essa operação aritmética **OPCODE** sempre será **000000**;
  RS contem um número positivo entre 0 e 31, exemplo: **00100** (em decimal = 4);
  RT contem um número positivo entre 0 e 31, exemplo: **01001** (em decimal = 9);
  RD é onde será armazenado o resultado, então deverá ser **00000**;
  SHAMT não será utilizado nesse tipo de operação, então deve-se deixar **00000**;
  FUNCT é o mais importante pois indica que se trata de uma operação de multiplicação, nesse caso será **011000**.

- **DIV** (operação de divisão)

  Para essa operação aritmética **OPCODE** sempre será **000000**;
  RS contem um número positivo entre 0 e 31, exemplo: **00011** (em decimal = 3);
  RT contem um número positivo entre 0 e 31, exemplo: **01001** (em decimal = 9);
  RD é onde será armazenado o resultado, então deverá ser **00000**;
  SHAMT não será utilizado nesse tipo de operação, então deve-se deixar **00000**;
  FUNCT é o mais importante pois indica que se trata de uma operação de divisão, nesse caso será **011010**.
  OBS: caso **RS** seja menor que **RT** ocorrerá um erro e o programa será finalizado.

- **SW** (store word ou armazenar palavra)

  Para essa operação aritmética **OPCODE** sempre será **101011**;
  RS contem um número positivo entre 0 e 31, exemplo: **00110** (em decimal = 6);
  RD nesse caso terá o endereço de onde será armazenado o valor de **RS**;

Feito a parte de escrita das instruções em um documento de texto, deve-se executar o 2º passo que é o software em si.

Quando o software for aberto ele solicitará que o usuário digite o caminho do arquivo, como exemplo: "**D:\SIMULADOR_MIPS\mips.txt**", devendo lembrar de digitar sempre a extensão do arquivo.

Feito isso basta seguir as instruções que estão no próprio software para a completa execução do programa.

Autores: Matheus Vieira - 2º Semestre, Tecnólogo em Redes de computadores,
David Corino – 2º Semestre, Tecnólogo em Redes de computadores.