SIMULADOR
RELATÓRIO

Álissom Vieira da Cunha

00216123

Luiz Eduardo Pereira

0021619

Instituto Federal de Minas Gerais, Formiga, MG.

INTRODUÇÃO

Este trabalho tem como objetivo a implementação de um simulador para o processador teórico MIPS-IF32, inspirado em um MIPS de 32 bits.

IMPLEMENTAÇÃO

Este trabalho foi desenvolvido na linguagem C e foi dividido nas estruturas: Arquivo.c, Simulador.c, Registrador.c, Busca.c, Decodifica.c, Executa.c.

Arquivo.c

A parte de Arquivo faz simplesmente as operações básicas com arquivo texto, abertura, fechamento e leitura.

Simulador.c

O simulador mantém a memória de dados e memória de instrução, a qual chamamos de “struct informacao”. Essa struct possui dois vetores de inteiros, cada posição representa uma instrução. Dois inteiros informam o tamanho da memória de dados e o tamanho da memória de instruções.

O Program Counter também fica na TAD simulador. Ele guarda a linha de execução do programa. O Program Counter controla a execução, enquanto ele não chegar ao fim, o programa continua em execução. O simulador chama todas as outras TAD’s necessarias para a execução.

Registrador.c

A TAD registrador manipula o vetor de inteiros registrador. A declaração do registrador é feita no simulador, mas sua manipulação é feita totalmente nesta TAD. Inicialmente o simulador inicia todos os registradores com 0.

Existe uma função, “verifica_registrador”, que, dado como entrada uma instrução e o registrador requisitado (1 - $d / 2 - $s, / 3 - $t / 4 - shamt), devolve o número do registrador. Também foi implementado duas funções para retornar o immediate e o address da instrução.

Busca.c

A TAD busca (fetch) faz a leitura do arquivo de entrada, e separa o .data para um vetor de dados e o .text para um vetor de instrução. A única particularidade do busca está no busca.h, que é a struct que guarda a memória de dados e instruções.

A struct informacao guarda os dados que serão usados pelo simulador. Seus campos são:

- unsigned int *dado;
- int tamanho_dado;
- unsigned int *instrucao;
- int tamanho_instrucao;

Decodifica.c

A TAD Decodifica (decode) é bem simples e possui apenas duas funções:

decodifica_tipo:

Dado uma instrução, é possível saber seu tipo olhando seu opcode. Se for 0, é do tipo R, se for 2 ou 3, tipo J, e o restante, tipo I. Sendo assim, foi dado um shift rigth, ou seja, o deslocamento de 26 bits para a direita, fazendo com que sobre apenas os 6 bits de opcode, e assim o tipo da instrução é definido.

decodifica_instrucao:

Para definir qual instrução é qual, devemos saber seu tipo, o que foi feito na função acima. Assim, para instruções do tipo R, precisamos descobrir o seu funct, ou seja, seus 6 bits menos significativos. Para isso, foi utilizado um “and” (&) lógico entre a instrução e o número 63 (111111). Feito isso, o resultado é os 6 bits do funct. Agora, com o tipo da instrução e o funct, podemos dizer qual instrução do tipo R será executada.

Já para as instruções do tipo J e I, o que define qual instrução é, é o seu opcode. Para pegar o opcode fazemos o mesmo passo de shift rigth usado na função acima. Para instruções desse tipo, o opcode já é suficiente para identificar a instrução requerida.

Executa.c

A TAD Executa (execute) recebe como parâmetro o vetor de registradores, a instrução, o tipo da instrução, e o identificador da instrução.

Com esses dados, o programa é capaz de encontrar a instrução requerida. Quando o programa entra no “IF” de cada tipo, são inicializados os números dos registradores.

Tipo R:

Iniciliza: $s, $t, $d, shamt;

Tipo J:

Inicializa: address;

Tipo I:

Inicializa: $s, $t, immediate.

Em seguida a instrução requerida é executada. A instrução “syscall”, apesar de não ser do tipo R, está junto com esse grupo, pois sua identificação é muito semelhante.

VALIDAÇÃO

Para fazer os testes, comparamos resultados de determinados arquivos com outros colegas de turma.

PENDÊNCIAS

Não foram implementadas as chamadas do sistema (syscall) 4 e 8.

CONCLUSÃO

Com este trabalho foi possível ter um melhor entendimento da matéria, já que nos mostra como um computador funciona a um nível mais baixo do que estamos acostumados. Também podemos ver mais como o jeito de programar em alto nível pode afetar no desempenho, já que uma simples instrução pode se tornar várias instruções em assembly.