Montador e Simulador RISC
IFMG - Campus Formiga

Guilherme Cardoso Silva

[1] Arquitetura e Organização de Computadores – Instituto Federal de Minas Gerais (IFMG)
Caixa Postal 35570-000 – Formiga – MG – Brasil

godgcs@gmail.com.br

1. Resumo Maquina Simulada

O processador IFMG-RISC implementado possui as seguintes características:

- 32 Registradores de uso geral;
- 6 Registradores reservados para verificações do processador.
- 33 Instruções disponiveis;
- Instruções de até 3 operandos, sendo elas para numeros inteiros, como soma, subtração, multiplicação, etc;
- Memória endereçada com quatro bytes. No total, o processador possui 64k(2^{16}) endereços. Com um total de 256 KBytes.

2. Resumo Montador Implementado

O montador converte instruções simbólicas construidas em assembly e as converte em instruções binárias que podem ser utilizadas para simular o processador IFMG-RISC implementado.

O montador realiza a leitura de um arquivo contendo instruções em assembly do processador IFMG-RISC e produz um arquivo de saída em formato compatível com o simulador implementado. Os nomes dos arquivos de entrada e de saída devem ser passados ao programa montador como parâmetro na linha de comando para que possa ser gerado os binarios.

3. Decisões de Implementação

3.1. Montador

Primeiramente é feita uma leitura no arquivo, armazenando todas as linhas do arquivo de instruções em assembly para um vetor de strings. Neste processo apenas as instruções são copiadas, os labels são copiados para um outro vetor que armazena o seu nome junto da posição de memoria a qual ele se inicia.

Após todo o arquivo ser carregado e os labels identificados e filtrados, é feita uma leitura no vetor de instruções, onde cada instrução é identificada e convertida para binario, de acordo com seus valores. Sempre que a instrução se refere a um label, é buscado no vetor de labels qual o endereço foi definido a ele e então substituido na inscrução em binario.

Cada uma das instruções em binario geradas são escritas em um arquivo de saida que serão utilizadas para a entrada no simulador.

3.2. Simulador

Ao executar o simulador, primeiramente é carregado todas as instruções em binario para a memória. Sendo que é verificado se existe um ADDRESS para que seja contabilizado em onde deve se iniciar o carregamento.

Cada instrução lida do arquivo em binário é convertido para um numero inteiro, e então armazenado na memoria, a cada instrução o contador de memoria é incrementado de 1, para que a proxima instrução possa ser carregada.

Quando todas as instruções são carregadas para a memoria se inicia o processo de simulação, que são divididos em 4 ciclos:

- **Primeiro ciclo: IF** Aqui as instruções são buscadas na memoria na posição definida em PC, a cada busca o valor do registrador PC é incrementado;
- **Segundo ciclo: ID** Aqui é identificado qual a instrução vai ser executada. Para isto é utilizado um campo de bits referente a instrução, o campo de bit tem o papel de quebrar em bits para cada valor necessario. Assim é possivel verificar qual o opcode seguido do registrador ra, rb e rc;
- **Terceiro e Quarto ciclo: EX/MEM e WB** Como se trata de um simulador, estes ciclos se localizam dentro do segundo ciclo, pois a identificação é implementada em um CASE. Quando ela começa a ser executada os valores já são gravados no registrador referente a operação.

Este processo de 4 ciclos é repetido até que seja encontrado uma instrução halt que finaliza a execução da aplicação.

4. Instruções Implementadas

4.1. MULT - Multiplicação

Endereçacamento binario: opcode(8) ra(8) rb(8) rc(8)

Formato: mult rc,ra,rb

Exemplo: mult r3,r2,r7

Operação: rc = ra*rb

Descrição: Multiplica o conteúdo de ra e de rb e coloca o resultado em rc.

Flags afetados: neg, zero, carry = 0 e overflow.

Motivo Implementação: Para que não seja necessario realizar sucessivas somas codificadas em inumeras instruções, esta função pode resumi-las em uma unica.

4.2. DIV - Divisão

Endereçacamento binario: opcode(8) ra(8) rb(8) rc(8)

Formato: div rc,ra,rb

Exemplo: div r3,r2,r7

Operação: rc = ra/rb

Descrição: Divide o conteúdo de ra e de rb e coloca o resultado em rc.

Flags afetados: neg, zero, carry = 0 e overflow. **Motivo Implementação:** Para que não seja necessario realizar sucessivas subtrações codificadas em inumeras instruções, esta função pode resumi-las em uma unica.

4.3. LOADI - LOAD Imediato

Endereçacamento binario: opcode(8) constan(16) rc(8)

Formato: loadi rc,constan

Exemplo: loadi r3,constan

Operação: rc = constan

Descrição: Carrega no registrador rc o conteúdo do constan.

Flags afetados: nenhum

Motivo Implementação: Para que possa carregar um valor de até 16 bits diretamente para o registrador.

4.4. STOREI - STORE Imediato

Endereçacamento binario: opcode(8) constan(16) rc(8)

Formato: storei rc,constan

Exemplo: storei r3,constan

Operação: memoria[constan] = rc

Descrição: Carrega na memoria na posição constan o conteudo do registrador rc.

Flags afetados: nenhum

Motivo Implementação: Para que possa ser salvo na memoria o valor de um registrador em uma posição que é passada diretamente para a função.

4.5. DEC - Decrementa

Endereçacamento binario: opcode(8) ra(8) rb(8) rc(8)

Formato: dec rc

Exemplo: dec r3

Operação: rc- -

Descrição: Decrementa 1 do conteúdo do registrador rc.

Flags afetados: neg, zero, carry = 0 e overflow.

Motivo Implementação: Para que se torna mais rapido e eficiente a forma como se subtrai 1 de um valor em um registrador, não sendo necessario carregar o valor 1 para um registrador para que possa se realizar uma subtração.

4.6. INC - Incrementa

Endereçacamento binario: opcode(8) ra(8) rb(8) rc(8)

Formato: inc rc

Exemplo: inc r3

Operação: rc++

Descrição: Incrementa 1 do conteúdo do registrador rc.

Flags afetados: neg, zero, carry e overflow.

Motivo Implementação: Para que se torna mais rapido e eficiente a forma como se soma 1 de um valor em um registrador, não sendo necessario carregar o valor 1 para um registrador para que possa se realizar uma soma.

4.7. SQRT - Raiz Quadrada

Endereçacamento binario: opcode(8) constan(8) rb(8) rc(8)

Formato: sqrt rc,constan

Exemplo: sqrt r3,4

Operação: rc = sqrt(constan)

Descrição: Realiza a raiz quadrada do valor imediato em constan e coloca o resultado em rc.

Flags afetados: neg, zero, carry = 0 e overflow.

Motivo Implementação: Para que se possa ser facilmente calculada a raiz quadrada de um numero imediato.

4.8. POW - Potência

Endereçacamento binario: opcode(8) constan1(8) constan2(8) rc(8)

Formato: pow rc,constan1,constan2

Exemplo: pow rc,2,3

Operação: rc = pow(constan1,constan2)

Descrição: Realiza a potencia do conteúdo do valor em constan1 elevado pelo valor em constan2 e coloca o resultado em rc.

Flags afetados: neg, zero, carry = 0 e overflow.

Motivo Implementação: Para que se possa ser facilmente calculada a potência entre uma base e um expoente, ambos carregados em imediato.

4.9. ADDI - Adição Imediato

Endereçacamento binario: opcode(8) constan1(8) constan2(8) rc(8)

Formato: addi rc,constan1,constan2

Exemplo: addi r3,2,7

Operação: rc = constan1 + constan2

Descrição: Soma (aritmética) o conteúdo de constan1 e de constan2 e coloca o resultado em rc.

Flags afetados: neg, zero, carry e overflow.

Motivo Implementação: Para que se possa diminuir a quantidade de operações para realizar uma soma, em vez de ter que carregar o operador 1 e o operador 2, eles já são carregados de imediato, reduziando o numero de loads.

4.10. SUBI - Subtração Imediato

Endereçacamento binario: opcode(8) constan1(8) constan2(8) rc(8)

Formato: subi rc,constan1,constan2

Exemplo: subi r3,2,7

Operação: rc = constan1 - constan2

Descrição: Subtrai (aritmética) o conteúdo de constan2 de constan1 e coloca o resultado em rc.

Flags afetados: neg, zero, carry = 0 e overflow.

Motivo Implementação: Para que se possa diminuir a quantidade de operações para realizar uma subtração, em vez de ter que carregar o operador 1 e o operador 2, eles já são carregados de imediato, reduziando o numero de loads.

5. Tutorial de como utilizar o Montador e Simulador

5.1. Montador

Para utilizar o montador é necessario primeiramente gerar um arquivo em assembly de acordo com os padrões adotados pelas inscrições da especificação do trabalho, junto com as instruções criadas.

Em seguida para compilar o montador basta utilizar o comando *gcc *.c -o exe*.

Após a compilação para executar é necessario utilizar o comando *./exe "arquivoEntrada" "arquivoSaida"*, que ao ser executado irá gerar um arquivo de saida para o codigo em assemble convertido para binario, para que possa ser executado pelo simulador.

5.2. Simulador

Para utilizar o simulador é necessario compilar utilizando o comando *gcc *.c -lm -o exe*.

Após a compilação pra executar uma simulação é necessario utilizar um arquivo de instruções em binario gerado na saida do montador, como entrada para o simulador. Para isto basta executar o comando *./exe "arquivoEntrada"*.

6. Estrutura de Dados Utilizada

6.1. Montador

Para que as instruções em assembly fossem armazenas e convertidas para binario foi utilizado a seguinte estrutura:

- **char Memoria[TamMemoria][TamInstrucoes]:** Matriz de char responsável por armazenar as instruções em assembly lidas do arquivo de entrada.

- **Labels VetLabels[TamMemoria]:** Vetor do Tipo Labels que armazena o nome de um label e qual o valor do ContMemoria no momento em que foi encontrado. Quando alguma instrução redireciona o fluxo de execução, é realizada uma busca neste vetor afim de encontrar a posição em que se encontra na memoria. Caso ele não seja encontrado a aplicação é finalizada.
  A estrutura para o tipo Label é a seguinte:
  typedef struct labels{
  char Nome[TamInstrucoes];
  int posiMemoria;
  }Labels;

- **int ContMemoria=0, ContLabel=0:** Ambas as variaveis são utilizadas para varrer os vetores, sendo que:
  - O ContMemoria é utilizado na leitura dos dados, onde os dados são carregados do arquivo para o vetor Memória.
  - O ContLabel identifica quandos labels já foram encontrados, para que possam ser buscados na codificação dos binarios.

6.2. Simulador

Para realizar a simulação foram utilizadas as senguintes estruturas:

- **int Memoria[65536];** Vetor de inteiros que simula uma memoria de 256Kbytes utilizado para armazenar as instruções e dados das execuções.
- **int Registradores[32];** Vetor de inteiros que simula os 32 registradores gerais para uso no processador.
- **int IR, neg, zero, carry, overflow, ADDRESS = 0;** inteiros que simulam os registradores reservados pelo processador para verificações, sendo eles:
  - IR: Registrador que recebe a instrução a ser decodificada na função IF;
  - neg: Registrador que verifica se uma operação resultou em um numero negativo;
  - zero: Registrador que verifica se uma operação resultou em um numero zero;
  - carry: Registrador que verifica se uma operação resultou em um carry;
  - overflow: Registrador que verifica se uma operação resultou em um overflow.
  - ADDRESS: Inteiro que realiza um deslocamento no carregamento de instruções na memória.
- **int ContMemoria=0, PC=0;** Ambas as variaveis são utilizadas para varrer o vetor Memoria, sendo que:
  - O ContMemoria é utilizado na leitura dos instruções binarios, onde os dados são carregados do arquivo para o vetor Memória;
  - O PC é o Registrador responsável por definir as posições de memória que serão as proximas a serem buscadas para execução.

7. Códigos Fontes

7.1. Montador

As principais funções do Montador foram:

- **void itob(int Num, int Bits, char Binario[]):** Função de conversão de um numero inteiro para um numero binario em uma string que é passada por parametro;
- **void LeInstrucoesArquivo(char ArquivoEntrada[]):** Função que faz a leitura do arquivo de entrada de instruções em assembly e as armazena na matriz de memória;
- **void CodificaInstrucoes(char ArquivoSaida[]):** Função que faz a codificação de todas as instruções gravadas na memoria para binario, para isto é verificado cada opcode e cada parametro lido, a partir dai os valores são convertidos para binario e concatenados, gerando um binario de 32 bits.

7.2. Simulador

As principais funções do Simulador foram:

- **void LeInstrucoesArquivo(char ArquivoEntrada[]):** Função que carrega todos os binarios do arquivo de entrada para a Memória. Todos os Binarios são convertidos para inteiro para serem armazenados;
- **void ExecutaInstrucoes(void):** Função que realiza sucessivas instruções a partir do endereço de PC até que uma instrução halt seja encontrada;
  Para executar as instruções o simulador é divido em 4 ciclos para a execução das instruções conforme já detalhado.
- **int InstructionFetch(void):** Primeiro ciclo.Função que busca na memória qual a instrução esta armazenada na posição de memoria contabilizada por PC. Após pegar a instrução é incrementado o valor do registrador PC;
- **void InstructionDecode(long int Instrucao):** Segundo, Terceiro e Quarto ciclo. Função responsavel por decodificar executar cada instrução buscada e escrever seus resultados nos registradores.
  Através de um case com todos os opcodes é definido qual instrução deve ser executada, e apartir dai são executadas e gravadas nos registradores ou memória.

8. Testes Realizados

Como testes, foram implementadas diversas funções em assembly para simular todo o processador implementado:

- Teste 1: Fatorial
- Teste 2: For10to0 Versão 1
- Teste 3: For10to0 Versão 2
- Teste 4: For0to10 Versão 1
- Teste 5: For0to10 Versão 2
- Teste 6: Pow
- Teste 7: Sqrt
- Teste 8: Soma Versão 1
- Teste 9: Soma Versão 2