Montador e Simulador IFMG-RISC

Arthur Alexsander Martins Teodoro[1]

[1]Instituto Federal de Minas Gerais - Campus Formiga

arthurmteodoro@gmail.com

Introdução

Este trabalho tinha como objetivo a implementação de um montador e um simulador funcional para o processador RISC de 32 bits IFMG-RISC. O processador é o dispositivo principal de um computador. Nele e realizada as operações lógicas e aritméticas. Para isso, o processador lê os dados e instruções da memória, analisa e executa tais instruções.

O montador é o *software* que converte instruções simbólicas entendíveis para programadores em instruções binárias, estas lidas pelo processador. Para o programação de um programa em linguagem de montagem, deve conhecer as instruções que o processador possui, e gerar um programa usando somente tais.

Considerações de implementação geral

Para a implementação, foi usado um computador com processador *i5 64bits*. Como processador implementado é de 32 bits, no computador usado para a implementação o tio inteiro possui 32 bits. Para tentar manter esse número de bits, foi alocada a memória e o banco de registradores com cada palavra com 4 *bytes*, porém, para a manipulação, teve que ser feita uma conversão de ponteiros. Porém, isso apenas reforça que o conhecimento da arquitetura é necessária, uma vez que a quantidade de *bytes* mapeada em um computador pode ser diferente que para outros.

Máquina Simulada

A máquina simulada é um processador de 32 bits, com 33 registradores, memória de 256 *KBytes* e possui 32 instruções. Além dos 33 registradores de propósito geral, existe os registradores IR e PC, estes armazenando a instrução e o ponteiro para a próxima instrução respectivamente. Também possui o registrador temporário onde o resultado da operação fica armazenado antes de ser atualizado no registrador destino.

A máquina possui 4 estágios, sendo estes IF(*instruction fetch*), ID(*instruction decode*), EX/MEM(*execute and memory*) e WB(*write back*).

Considerações de Implementação

Para a facilitação e melhor uso dos registradores de propósito geral, o processador possui 33 registradores, uma vez que na linguagem de programação usada, caso fosse criada 32 registradores, o registrador r1 teria endereço 0. Com a criação de 33 registradores, isso não acontecia, além de que o registrador r0 possui o valor 0 como constante.

Os registradores não endereçados por 8 bits, uma vez que existindo operações de 3 operandos, a quantidade de bits da instrução é 32.

Montador Implementado

O montador implementado foi um montador do tipo de dois passos, que, no primeiro passo busca por rótulos e endereços abstratos. Para o mapeamento destas estruturas, foi usada uma tabela *hash*.

Considerações de Implementação

O arquivo de entrada do montador pode possuir além das instruções, comentários, que são definidos pelo caractere *#*. Além disso, os rótulos possuem dois pontos(*:*) na definição e os endereços de memória abstrato definidos por colchetes(*[]*) e como primeiro caractere, uma letra. Ex: *[Num1]*. Além disso, o montador não é *case sensitive*.

Compilação e Execução

Como foi pedido para a geração de um arquivo *Makefile* para a compilação tanto do simulador e montador. Para executar o montador, deve ter como entrada dois argumentos, o primeiro o nome do arquivo de entrada(linguagem simbólica) e o segundo o nome do arquivo de saída(código binário). Já o simulador, para ser executado, deve ter como entrada o arquivo com código binário.

Instruções Projetadas

Foi implementadas nove instruções, estas instruções foi escolhidas para facilitar a criação de programas e economizar registradores de propósito geral. Elas são:

Multiplicação

Opcode 8bits	Ra 8bits	Rb 8bits	Rc 8ibts

Formato: mult rc, ra, rb

Exemplo: mult r3, r1, r2

Operação: rc = ra * rb

Descrição: Multiplica os valores de *ra* e *rb* e coloca o resultado em *rc*

Flags Afetadas: neg, zero, carry, overflow

Tal instrução foi criada para facilitar a programação, uma fez que a multiplicação pode ser feita por soma sucessiva, porém isso iria necessitar que o programar fizesse a multiplicação.

Divisão

Opcode 8bits	Ra 8bits	Rb 8bits	Rc 8ibts

Formato: div rc, ra, rb

Exemplo: div r3, r1, r2

Operação: rc = ra / rb

Descrição: Divide os valores de *ra* por *rb* e coloca o resultado em *rc*

Flags Afetadas: neg, zero, carry, overflow

Tal instrução foi criada para facilitar a programação, uma fez que a divisão pode ser feita por subtração sucessiva, porém isso iria necessitar que o programar fizesse a divisão.

Módulo

Opcode 8bits	Ra 8bits	Rb 8bits	Rc 8ibts

Formato: mod rc, ra, rb

Exemplo: mod r3, r1, r2

Operação: rc = ra % rb

Descrição: Divide os valores de *ra* por *rb* e coloca o resto em *rc*

Flags Afetadas: neg, zero, carry, overflow

Tal instrução foi criada para facilitar a programação, uma vez que como só existe a divisão inteira, o resto pode ser útil.

Adição com um valor imediato

Opcode 8bits	Ra 8bits	Const8 8bits	Rc 8ibts

Formato: addi rc, ra, Const8

Exemplo: addi r3, r1, 1

Operação: rc = ra + Const8

Descrição: Soma os valores de *ra* e a contante de 8 *bits* e coloca o resultado em *rc*

Flags Afetadas: neg, zero, carry, overflow

Tal instrução foi criada para facilitar a programação e economizar registradores, uma vez que uma soma que sempre irá ocorrer com um valor pode ser feita com esta instrução, não sendo necessário armazenar este valor em um registrador.

Subtração com um valor imediato

Opcode 8bits	Ra 8bits	Const8 8bits	Rc 8ibts

Formato: subi rc, ra, Const8

Exemplo: subi r3, r1, 1

Operação: rc = ra - Const8

Descrição: Subtrai o valor da Contante de 8 *bits* do valor de *ra* e coloca o resultado em *rc*

Flags Afetadas: neg, zero, carry, overflow

Tal instrução foi criada para facilitar a programação e economizar registradores, uma vez que uma subtração que sempre irá ocorrer com um valor pode ser feita com esta instrução, não sendo necessário armazenar este valor em um registrador.

Multiplicação com um valor imediato

Opcode 8bits	Ra 8bits	Const8 8bits	Rc 8ibts

Formato: multi rc, ra, Const8

Exemplo: multi r3, r1, 1

Operação: rc = ra * Const8

Descrição: Multiplica o valor de *ra* com o valor da contante de 8 *bits* e coloca o valor em *rc*

Flags Afetadas: neg, zero, carry, overflow

Tal instrução foi criada para facilitar a programação e economizar registradores, uma vez que uma multiplicação que sempre irá ocorrer com um valor pode ser feita com esta instrução, não sendo necessário armazenar este valor em um registrador.

Divisão com um valor imediato

Opcode 8bits	Ra 8bits	Const8 8bits	Rc 8ibts

Formato: divi rc, ra, Const8

Exemplo: divi r3, r1, 1

Operação: rc = ra / Const8

Descrição: Divide o valor de *ra* pela constante de 8 *bits* e guarda o valor em *rc*

Flags Afetadas: neg, zero, carry, overflow

Tal instrução foi criada para facilitar a programação e economizar registradores, uma vez que uma divisão que sempre irá ocorrer com um valor pode ser feita com esta instrução, não sendo necessário armazenar este valor em um registrador.

Load direto

Opcode 8bits	Endereco 16bits	Rc 8ibts

Formato: loadd rc, 16Bits

Exemplo: loadd r1, [A]

Operação: rc = memoria[A]

Descrição: Carrega os valores que estão na posição A da memória e carrega para *rc*

Flags Afetadas: Nenhuma flag afetada

Tal instrução foi criada para facilitar a programação uma vez que o programador pode abstrair os endereços de memória e simplesmente carregar tal valor.

Store direto

Opcode 8bits	Endereco 16bits	Rc 8ibts

Formato: stored rc, 16Bits

Exemplo: stored r1, [A]

Operação: memoria[A] = rc

Descrição: Carrega os valores de *rc* e salva na posição A da memória

Flags Afetadas: Nenhuma flag afetada

Tal instrução foi criada para facilitar a programação uma vez que o programador pode abstrair os endereços de memória e simplesmente carregar tal valor.

Testes

Para a validação do montador e simulador, foi feita uma bateria de testes. Para a validação do montador, foi conferido manualmente o código binário gerado por ele, conferindo se confere com a entrada esperada do simulador.

No caso do simulador, foi testado se o programa estava funcionando corretamente, gerando o resultado esperado. Para isso, foi criado testes reais, de programas reais e alguns com a finalidade de testar partes críticas e de difícil implementação, como os rótulos e endereços abstratos. Tais testes podem ser encontrados na pasta teste do projeto enviado.

Conclusão

Com este trabalho, foi possível aprender muito sobre a arquitetura e organização de um computador, uma fez que questões de implementações geradas na criação de um processador também ocorreram neste trabalho. Sem mencionar um grande aprendizado no entendimento de um processador, como um código para uma arquitetura não se torna portável facilmente e todas questões que são abstraídas do programador pelo compilador.