INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA MINAS GERAIS Campus Formiga

Graduação em Ciência da Computação

Disciplina: Arquitetura de Computadores
Professor: Walace de Almeira Rodrigues
Atividade: Trabalho Prático

INSTRUÇÕES:

1. Esta atividade deve ser resolvida individualmente. Opcionalmente pode ser resolvida em grupo de NO MÁXIMO dois alunos mas, se for esse o caso, os nomes dos alunos do grupo devem ser comunicados ao professor por escrito antecipadamente. Não será permitido alterar a formação do grupo posteriormente!

2. Se você achar que falta algum detalhe nas especificações, faça as suposições que julgar necessárias e as inclua no seu relatório. Pode acontecer da especificação desta atividade conter mais informações que o necessário para a solução, nesse caso utilize sua capacidade de julgamento para separar o supérfluo do necessário.

3. Esta atividade compreende o desenvolvimento de um montador e de um simulador para uma arquitetura inspirada num MIPS de 32 bits. Cada parte desta atividade vale 20 pontos e tem seu prazo de entrega específico:

   1a parte Desenvolvimento do montador, com entrega até 23:59:00 de 25/05/2017
   2a parte Desenvolvimento do simulador, com entrega até 23:59:00 de 29/06/2017

4. Para cada parte desta atividade serão gerados três artefatos como produtos: os códigos fontes da implementação, um arquivo MakeFile, e um relatório documentando o desenvolvimento da parte em questão.

   - Todos os arquivos fontes (*.c e *.h), mais o arquivo MakeFile, devem estar armazenados dentro de uma mesma pasta.
   - Cada arquivo fonte deve ter um cabeçalho contendo as seguintes informações: nome do autor(a) e matrícula, breve sinopse do conteúdo e data.
   - O arquivo MakeFile tem a função de compilar os fontes e gerar um executável. Pesquise na internet sobre como criar o seu MakeFile; como sugestão há uma introdução breve em `http://orion.lcg.ufrj.br/compgraf1/downloads/MakefileTut.pdf`.
   - O relatório da atividade deve ser entregue no formato PDF devidamente identificado com o nome e matrícula do(a) autor(a) do trabalho.
   - Compacte todos os artefatos gerados num único arquivo no formato ZIP (ou RAR) antes de entregar.

5. Todos os produtos serão entregues via portal acadêmico em `https://meu.ifmg.edu.br/`.

6. O envio é de total responsabilidade do aluno. Não serão aceitos trabalhos enviados fora do prazo estabelecido.

7. Trabalhos plagiados serão desconsiderados, sendo atribuída nota zero a todos os envolvidos.

8. O trabalho deve, obrigatoriamente, ser implementado na linguagem de programação C usando o compilador gcc.

1 Introdução

Este documento apresenta a especificação do trabalho prático para a disciplina de Arquitetura de Computadores. O trabalho visa a implementação de um montador e de um simulador funcional para o processador teórico MIPS-IF32, inspirado num MIPS (*Microprocessor without Interlocked Pipeline Stages*) de 32 bits.

Um montador é um programa que converte instruções simbólicas para instruções binárias. Detalhes funcionais sobre a arquitetura do MIPS-IF32, sobre a linguagem de montagem (Assembly), bem como os formatos dos arquivos fonte e alvo envolvidos no processo de tradução serão apresentados na sequência deste documento. Instruções para o desenvolvimento do montador e do simulador também serão apresentadas nas seções correspondentes.

1.1 Sobre a arquitetura MIPS-IF32

Um diagrama simplificado descrevendo a arquitetura do MIPS-IF32 é apresentado na Figura 1.

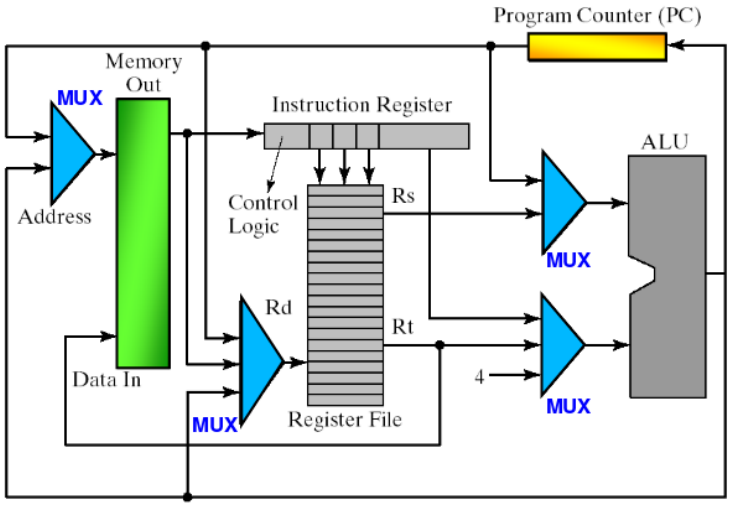

Figura 1: Diagrama simplificado do MIPS-IF32

O MIPS-IF32 é um caso de arquitetura RISC. Ele foi projetado para executar um conjunto reduzido de instruções simples que usam apenas registradores para realizar as operações aritméticas e lógicas. Foi uma exigência do projeto que todas as instruções fossem finalizadas em um único ciclo, então para cumprir esse objetivo quaisquer instruções que iriam requerer mais ciclos (tais como as instruções de multiplicação e de divisão) foram eliminadas do conjunto de instruções suportadas pelo processador. Assim sendo, se um programador quiser realizar uma operação de multiplicação, por exemplo, ele deverá escrever um algoritmo que usa as instruções mais simples suportadas pelo processador. Doravante utilizaremos a sigla ISA (*Instruction Set Architecture*) para referirmos ao conjunto de instruções suportadas pelo processador.

O processador MIPS-IF32 possui 32 registradores de uso geral e implementa um ISA de 40 instruções. Todas as instruções foram clonadas do MIPS tradicional de 32 bits e apresentam o mesmo comportamento funcional apresentado naquela arquitetura. Uma vantagem disso é que todas as instruções do MIPS tradicional encontram-se muito bem documentadas em livros (consulte as referências) e em apostilas que podem ser encontradas facilmente na internet.

DICA: se você quiser praticar, programar e testar pequenos programas escritos em Assembly do MIPS, existem vários simuladores gratuítos disponíveis. Como sugestão, há um simulador para um MIPS tradicional de 32 bits em `http://courses.missouristate.edu/kenvollmar/mars/download.htm`, juntamente com um programa exemplo escrito em Assembly que imprime os doze primeiros números da série de Fibonnaci. O simulador sugerido é o MARS, escrito em java, que tem a vantagem de ser portável e não necessitar instalação.

Informações sobre o projeto da memória

Para a arquitetura MIPS-IF32, toda a memória é organizada como uma sequência de bytes consecutivos endereçados por byte a partir do endereço zero. Todavia, para aumentar a performance, qualquer acesso à memória nessa arquitetura, tanto para leitura quanto para escrita, é sempre efetuado sobre uma word (palavra formada por 4 bytes). Nesse esquema, o alinhamento da memória é uma exigência que impõe que o endereço de cada word seja sempre múltiplo de 4: por exemplo, uma word pode começar nos bytes 0, 4, 8, mas não nos bytes 1, 2, 3, etc (ver Figura 2).

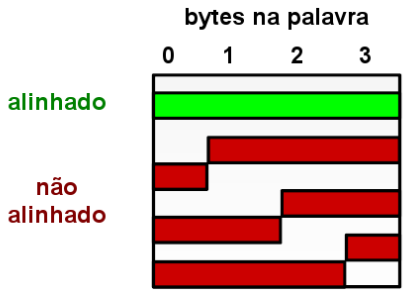

Figura 2: Alinhamento da memória no MIPS-IF32

O projeto da memória na arquitetura MIPS-IF32 segue o modelo de Harvard. Esse processador utiliza 32 bits para acessar uma memória endereçável por byte, onde entretanto qualquer dado é sempre referenciado como uma word, então foi preciso estabelecer uma convenção sobre como os quatro bytes ficam armazenados dentro da word. A convenção adotada nessa questão foi a *big-endian*: os bits mais significativos ficam armazenados nos menores endereços. A Figura 3 ilustra o armazenamento do inteiro 0x03020100 na word de endereço 12.

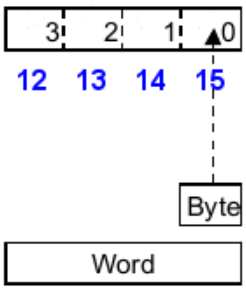

Figura 3: Convenção *big-endian* para posicionamento dos bytes numa word

1.2 Sobre a linguagem de montagem

O processador MIPS-IF32 implementa um ISA de 40 instruções, apresentadas na Figura 4.

Instruction	Usage
Load upper immediate	`lui rt,imm`
Add	`add rd,rs,rt`
Subtract	`sub rd,rs,rt`
Set less than	`slt rd,rs,rt`
Add immediate	`addi rt,rs,imm`
Set less than immediate	`slti rd,rs,imm`
AND	`and rd,rs,rt`
OR	`or rd,rs,rt`
XOR	`xor rd,rs,rt`
NOR	`nor rd,rs,rt`
AND immediate	`andi rt,rs,imm`
OR immediate	`ori rt,rs,imm`
XOR immediate	`xori rt,rs,imm`
Load word	`lw rt,imm(rs)`
Store word	`sw rt,imm(rs)`
Jump	`j L`
Jump register	`jr rs`
Branch less than 0	`bltz rs,L`
Branch equal	`beq rs,rt,L`
Branch not equal	`bne rs,rt,L`

Instruction	Usage
Move from Hi	`mfhi rd`
Move from Lo	`mflo rd`
Add unsigned	`addu rd,rs,rt`
Subtract unsigned	`subu rd,rs,rt`
Multiply	`mult rs,rt`
Multiply unsigned	`multu rs,rt`
Divide	`div rs,rt`
Divide unsigned	`divu rs,rt`
Add immediate unsigned	`addiu rs,rt,imm`
Shift left logical	`sll rd,rt,sh`
Shift right logical	`srl rd,rt,sh`
Shift right arithmetic	`sra rd,rt,sh`
Shift left logical variable	`sllv rd,rt,rs`
Shift right logical variable	`srlv rd,rt,rs`
Shift right arith variable	`srav rd,rt,rs`
Load byte	`lb rt,imm(rs)`
Load byte unsigned	`lbu rt,imm(rs)`
Store byte	`sb rt,imm(rs)`
Jump and link	`jal L`
System call	`syscall`

Figura 4: Conjunto de instruções do MIPS-IF32

Mesmo para a programação de baixo nível, não é costume programar diretamente em linguagem máquina, mas sim com recurso a linguagem de montagem, ou Assembly. O Assembly fornece certas facilidades como: labels simbólicos, alocações de dados globais e pseudo-instruções.

Programa Hello World

Segue o código em linguagem de montagem exemplificando como programar o MIPS-IF32:

```
1  #-------------------------------------------------
2  #        Programa Hello world
3  #-------------------------------------------------
4          .data
5  hello:  .asciiz "hello world\n"
6
7          .text
8  main:   li $v0, 4          # print_string
9          la $a0, hello      # endereço da string
10         syscall            # chamada ao sistema
11         li $v0, 10         # fim do programa
12         syscall            # chamada ao sistema
```

Esse código merece algumas observações por ter sido apresentado como exemplo para ilustrar detalhes da programação em Assembly.

- A diretiva ".data" marca o início da área dos dados. Após ela seguem as definições dos dados, nesse caso uma área de memória etiquetada como "hello" que armazena uma string.

- A diretiva ".asciiz" é usada para alocar memória para string. Existem outros tipos de diretivas usadas para alocar memória para outros tipos de dados.
- A diretiva ".text" marca o início da área do código, após ela seguem as instruções do programa. Normalmente, uma instrução por linha de programa, de acordo com a sintaxe:

  [label:] op-code [operando], [operando], [operando] [# comentário]

  O label é opcional. Labels são etiquetas simbólicas usadas pelos programadores para referenciar endereços específicos. No exemplo, "hello" e "main" são labels. O label "main" indica a primeira instrução a executar.
- Talvez você tenha notado que a Figura 4 não enumera as instruções "li" e "la". De fato, "li" e "la" são pseudo-instruções que terão que ser traduzidas pelo montador em termos de sequências de instruções contidas no ISA. Mais detalhes em `https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/pseudo.html`.
- A instrução "syscall" chama o sistema. No exemplo foram feitas duas chamadas, uma imprimir a string e outra para terminar a execução.
- Os operandos $v0, $a0 e $ra denotam registradores. O processador MIPS-IF32 possui 32 registradores de uso geral, ilustrados na Figura 5.

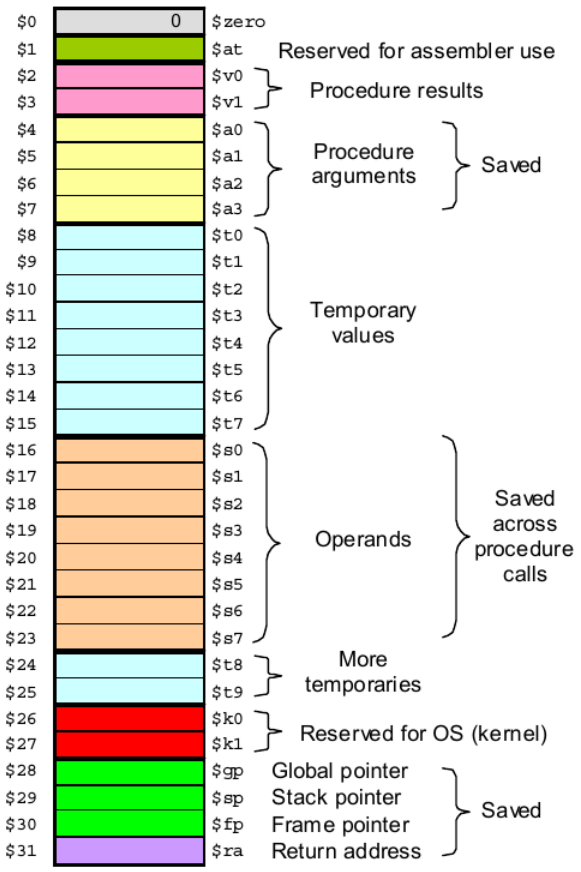

Figura 5: Registradores do MIPS-IF32

Para aprender como funciona cada instrução do Assembly, consulte as referências apresentadas. Todas as instruções do MIPS-IF32 foram clonadas do MIPS tradicional. Elas se encontram muito bem documentadas em livros e apostilas que podem ser encontradas facilmente na biblioteca ou na internet.

2 Projeto do Montador

Um montador é um programa quer converte instruções simbólicas em instruções binárias. O trabalho de implementação do montador é mais fácil do que parece inicialmente. Basicamente, o que você precisará fazer é ler, uma a uma, as instruções Assembly que estão escritas em um arquivo texto chamado de arquivo-fonte, traduzi-las para binário, e então salvar o valor hexadecimal dessa tradução (em formato texto) em outro arquivo texto chamado de arquivo-alvo. Nessa etapa não é necessário o entendimento do programa que você vai traduzir, apenas saber como traduzir isoladamente cada instrução, o que não é algo muito complicado.

Como exemplo, considere a instrução "add $t8, $s2, $s1" lida de uma linha do arquivo-fonte. Sua tradução seria feita da forma indicada na Figura 6, e o texto a ser salvo em uma linha do arquivo-alvo seria "0251C020", correspondente ao valor hexadecimal da tradução.

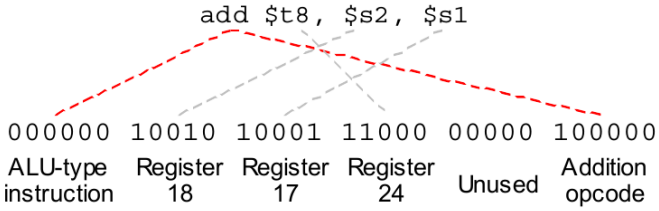

Figura 6: Conjunto de instruções do MIPS-IF32

Existem basicamente duas maneiras de se implementar um montador, a saber: montador de um passo e montador de dois passos. Nesse trabalho usaremos a segunda opção, conforme sugerido na seção 7.3 da referência TANENBAUM. A tradução em dois passos oferece uma estratégia simples para determinar antecipadamente os valores dos endereços dos labels, tão necessários para a tradução. A estratégia é a seguinte:

PASSO UM: o montador abre o arquivo-fonte para leitura e vai lendo instrução por instrução até o fim. Lembre-se que no MIPS, cada instrução ocupa uma word. Desse modo, sempre que um label aparece, o montador anota seu endereço e armazena essa informação numa lista encadeada para uso no passo dois. Terminado esse passo, o montador fecha o arquivo-fonte.

PASSO DOIS:: o montador abre novamente o arquivo-fonte para leitura e também abre o arquivo-alvo para escrita. Feito isso, até esgotar o arquivo-fonte, executa o laço: (1) lê instrução do arquivo-fonte; (2) traduz a instrução lida; (3) salva o resultado da tradução no arquivo-alvo. Por fim, fecha os dois arquivos.

Note que o montador deve produzir um arquivo-alvo em formato texto compatível com o formato esperado pelo simulador que será desenvolvido na segunda etapa deste trabalho. O formato do arquivo-alvo é bem simples e podemos defini-lo através das seguintes características:

- Todo o conteúdo do arquivo estará em formato texto. Primeiro, seguirá a tradução dos dados. Depois, seguirá a tradução do código.
- A parte contendo a tradução dos dados trará esse conteúdo:
  (a) Uma linha informando o tamanho da memória dos dados: **.data <inteiro>**. O número inteiro indica o tamanho em words da memória de dados que será solicitada ao simulador quando esse for executar o programa traduzido. O tamanho solicitado deve ser suficiente para armazenar todos os dados globais alocados, mais a memória de pilha utilizada para controlar as chamadas de procedimentos no momento da execução.

(b) Linhas contendo informações para debug: uma linha por label criado, no formato = <**label**> <**endereço**>. Essa informação será útil para ajudar o programador a checar a corretude da tradução, mas não será necessariamente utilizada pelo simulador.
(c) Linhas com a tradução dos dados globais, uma linha por word (dado), codificada em hexadecimal de 8 dígitos (32 bits).
(d) Finalizando a parte dos dados, uma linha contendo a palavra ".enddata"

- A parte contendo a tradução do código trará esse conteúdo:
  (a) Uma linha informando o tamanho da memória do código: **.text** <**inteiro**>. O número inteiro indica o tamanho em words da memória do código que será solicitada ao simulador quando esse for executar o programa traduzido. O tamanho solicitado deve ser suficiente para armazenar todo o código traduzido.
  (b) Linhas contendo informações para debug: uma linha por label criado, no mesmo formato da parte dos dados.
  (c) Linhas com a tradução do código, uma linha por word (instrução), codificada em hexadecimal de 8 dígitos (32 bits).
  (d) Finalizando a parte do código, uma linha contendo a palavra ".endtext"

Requisitos para o montador

1. Os nomes dos arquivos fonte e alvo serão passados ao programa montador como parâmetro na linha de comando;
2. desenvolva um TAD para tratar a lista encadeada utilizada para armazenar as informações sobre os labels coletadas no passo um;
3. desenvolva um TAD para tratar as informações sobre cada instrução, informação que serão utilizadas no momento de fazer a tradução;
4. desenvolva um TAD para tratar os arquivos fonte e alvo;
5. desenvolva os passos um e dois do montador como procedimentos distintos. Não codifique tudo em um único bloco gigantesco de código, aprenda a dividir para conquistar.
6. trate as diretivas: .data, .text, .byte, .word, .asciiz, .space (Ver `http://students.cs.tamu.edu/tanzir/csce350/reference/assembler_dir.html`).
7. trate as pseudo-instruções: mov, li, la, lw, sw, bge, bgt, ble, blt (Ver `https://en.wikipedia.org/wiki/MIPS_instruction_set#Pseudo-instructions`).

3 Projeto do Simulador

O simulador vai abrir o arquivo-alvo para leitura, ler as linhas do arquivo, alocar dinamicamente os blocos de memória para código e dados, segundo os tamanhos solicitados no arquivo-alvo. Em seguida, o simulador vai carregar as informações dos arquivos dentro dos blocos alocados, inicializar o contador de instruções e então iniciar a execução do programa simulado.

Cada instrução executada no MIPS-IF32 demora um único ciclo para completar e é processada em quatro etapas a saber:

1. IF (*instruction fetch*): a instrução a ser executada é buscada na memória e armazenada no registrador de instruções (IR), ao final desta etapa o contador de programa (PC) é incrementado;

2. ID (*instruction decode*): a instrução sendo executada é decodificada e os operandos são buscados no banco de registradores;
3. EX/MEM (*execute and memory*): a instrução é executada. As operações de leitura e escrita na memória (load, store), bem como o cálculo do endereço efetivo para as operações de desvio (jump, branch) são feitas também nessa etapa;
4. WB (*write back*): os resultados são escritos no banco de registradores.

Requisitos para o simulador

1. Espelhe o esquema adotado na arquitetura e desenvolva um TAD para tratar o *fetch*, outro TAD para tratar o *decode* e outro TAD para tratar o *execute* das instruções;

2. desenvolva um TAD para representar os registradores (e possivelmente alguma outra informação necessária, tal como o valor do PC);

3. restrinja a comunicação entre os TADs à troca de informação via acesso aos registradores.

4. implemente os serviços (chamada ao sistema via syscall): 1, 4, 5, 8, 10, 11, 12, 34. (Ver `http://courses.missouristate.edu/KenVollmar/mars/Help/SyscallHelp.html`.)

4 Critérios de Correção

Serão adotados os seguintes critérios de correção para o trabalho:

1. **correção:** somente serão corrigidos códigos portáveis e sem erros de compilação;
2. **precisão:** execução correta dos testes práticos; (60%)
3. **modularização:** uso adequado das TADs e das estruturas de dados; (40%)
4. **qualidade do código fonte:** legibilidade, indentação, comentários; (requisito)
5. **documentação:** relatório, conforme instruções apresentadas a seguir. (requisito)

O critério "correção" é obrigatório e condiciona a avaliação toda, o seu código deve compilar senão seu trabalho não será aceito e vai receber nota zero. Alguns critérios não receberão pontuação específica, mas funcionarão como "requisitos" que poderão afetar a pontuação se não forem atendidos adequadamente, portanto leve-os em conta durante o desenvolvimento do trabalho.

Haverá uma apresentação oral e individual do trabalho, no valor de 1 ponto (intervalo [0-1]). A nota da valiação será individual, mesmo se você desenvolveu o trabalho em grupo, e condicionada pela nota da apresentação. Na ausência de plágio, as notas dos trabalhos corretos serão computadas individualmente da seguinte forma: $\text{nota} = \text{nota\_apresentacao} * \text{nota\_trabalho}$, ou seja, a nota final é ponderada pela nota da apresentação.

Documentação

Esta seção descreve o formato e o conteúdo do relatório que deve ser gerado como produto final do trabalho. Seja sucinto: em geral é possível escrever um bom relatório entre 2 e 4 páginas. O relatório documentando seu sistema deve conter as seguintes informações:

1. **introdução:** descrever o problema resolvido e apresentar uma visão geral do sistema implementado;
2. **implementação:** descrever as decisões de projeto e implementação do programa. Essa parte da documentação deve mostrar como as estruturas de dados foram planejadas e

implementadas. Sugestao: mostre uma figura ilustrativa da TAD, os tipos definidos e os cabeçalhos das principais funções implentadas, os detalhes de implementação e especificação que porventura estavam omissos no enunciado, etc.

3. **validação:** descrição dos testes que o grupo fez para validar o trabalho, se seguiu alguma metodologia etc.
4. **pendências:** relate possíveis falhas que não foram resolvidas no tempo disponível para entrega do trabalho;
5. **conclusão:** avaliação do grupo sobre o trabalho considerando a experiência adquirida, a contribuição para o aprendizado da disciplina, as principais dificuldades encontradas ao implementá-lo e como tais dificuldades foram superadas;
6. **bibliografia:** cite as fontes consultadas na resolução do trabalho, se houver.

Referências:

1. PATTERSON, David A. e HENNESSY, John L. Organização e Projeto de Computadores: a interface hardware/software. 3ª edição. São Paulo: Campus, 2008.
2. PARHAMI, Behrooz. Arquitetura de Computadores: de microprocessadores a supercomputadores. 1ª edição. McGraw-Hill, 2008.
3. TANENBAUM, Andrew S. Organização Estruturada de Computadores. 5ª edição. São Paulo: Pearson Prentice Hall, 2007.