Universidade Federal de Alfenas - UNIFAL-MG
Bacharelado em Ciência da Computação
Prof. Luiz Eduardo da Silva

Trabalho de Compiladores
Geração de código para uma máquina MIPS

Objetivo

O objetivo deste projeto do compilador é gerar código para uma máquina real (MIPS), além de permitir a escrita de valores literais.

Problema

A arquitetura **MIPS** (Microprocessor without Interlocked Pipeline Stages) é uma arquitetura RISC (Reduced Instruction Set Computing), conhecida por seu design simples e eficiente, amplamente utilizada em sistemas embarcados. Desenvolvida pela MIPS Computer Systems nos anos 1980, a arquitetura visa maximizar o desempenho minimizando a complexidade dos circuitos.

Características principais:

- **Conjunto de Instruções Simples**: As instruções MIPS são de tamanho fixo (32 bits) e possuem um número reduzido de tipos, como instruções de carga/armazenamento, aritméticas, lógicas e de controle de fluxo. Isso simplifica a decodificação e otimiza a execução das instruções.
- **Pipeline**: A arquitetura MIPS implementa um pipeline de 5 estágios (IF, ID, EX, MEM, WB), permitindo a execução simultânea de múltiplas instruções. Cada estágio realiza uma parte da execução, aumentando a eficiência e o desempenho.
- **Registradores**: Possui 32 registradores de uso geral (de 32 bits), onde as operações aritméticas e lógicas são realizadas. O uso intensivo de registradores minimiza acessos à memória, que são mais lentos.
- **Load/Store**: Seguindo o modelo RISC, MIPS utiliza a abordagem load/store, onde apenas instruções específicas acessam a memória. Operações aritméticas são feitas exclusivamente entre registradores, simplificando o controle e reduzindo latências.
- **Modularidade e Escalabilidade**: O design modular permite a implementação em diferentes configurações e extensões, como MIPS16e (para instruções compactas) e MIPS64 (para suporte a 64 bits), tornando a arquitetura flexível para diversas aplicações.

Registradores MIPS

Os registradores do MIPS são divididos em várias categorias, cada uma com uma função específica. Aqui estão os principais registradores:

Registradores de Uso Geral:

- **$zero**: Sempre contém o valor 0.
- **$at** (Assembler Temporary): Usado pelo montador para armazenar valores temporários.
- **$v0-$v1**: Usados para armazenar valores de retorno de funções (valores de 0 a 1).
- **$a0-$a3**: Usados para passar os primeiros quatro argumentos para funções.
- **$t0-$t9**: Registradores temporários (não preservados entre chamadas de função).
- **$s0-$s7**: Registradores salvos (preservados entre chamadas de função).
- **$t8-$t9**: Registradores temporários (similares a $t0-$t7, mas preservados em algumas convenções).
- **$k0-$k1**: Usados pelo sistema operacional para armazenar dados temporários.
- **$gp** (Global Pointer): Aponta para a região de memória global.
- **$sp** (Stack Pointer): Aponta para o topo da pilha.
- **$fp** (Frame Pointer): Usado para acessar variáveis locais em funções.
- **$ra** (Return Address): Armazena o endereço de retorno de uma função.

Registradores Especiais:

- **$hi**: Parte alta de resultados de multiplicação e divisão.
- **$lo**: Parte baixa de resultados de multiplicação e divisão.

Registradores de Propósito Geral:

- **$pc**: Contador de programa, mantém o endereço da próxima instrução a ser executada. Não é acessível diretamente em MIPS.

Esses registradores são usados para executar operações aritméticas, armazenar endereços, passar argumentos entre funções, e realizar operações de controle de fluxo, entre outras tarefas.

Algumas instruções MIPS

1. A instrução `lw` (Load Word) em MIPS carrega um valor de 32 bits da memória para um registrador. Sua sintaxe é `lw rt, offset(rs)`, onde o valor armazenado no endereço calculado como `rs + offset` é copiado para o registrador `rt`. É usada para acessar dados na memória durante operações de leitura.
2. A instrução `sw` (Store Word) em MIPS armazena um valor de 32 bits de um registrador em um endereço da memória. Sua sintaxe é `sw rt, offset(rs)`, onde o conteúdo do registrador `rt` é escrito no endereço calculado como `rs + offset`. É usada para salvar dados da CPU na memória.

3. A instrução `addiu` (Add Immediate Unsigned) em MIPS realiza uma soma entre um registrador e um valor imediato de 16 bits, armazenando o resultado em outro registrador. Apesar do nome "Unsigned", ela opera como uma soma inteira comum, ignorando *overflow*. A sintaxe é `addiu rd, rs, immediate`, onde `rd` recebe o valor de `rs + immediate`. É amplamente usada para cálculos com constantes pequenas e manipulação de endereços.

4. A instrução `li` (Load Immediate) em MIPS carrega um valor imediato diretamente em um registrador. Sua sintaxe é `li rd, immediate`, onde o valor constante `immediate` é armazenado no registrador `rd`. É usada para inicializar registradores com valores específicos.

5. A instrução `add` em MIPS realiza a soma de dois valores de registradores, armazenando o resultado em outro registrador. Sua sintaxe é `add rd, rs, rt`, onde `rd` recebe o valor de `rs + rt`. Diferente de `addu`, verifica *overflow* durante a operação.

6. A instrução `sub` em MIPS realiza a subtração de dois valores de registradores, armazenando o resultado em outro registrador. Sua sintaxe é `sub rd, rs, rt`, onde `rd` recebe o valor de `rs - rt`. Verifica *overflow* durante a operação, diferentemente de `subu`.

7. A instrução `mult` em MIPS realiza a multiplicação de dois valores inteiros com sinal provenientes de registradores. Sua sintaxe é `mult rs, rt`, onde os valores de `rs` e `rt` são multiplicados, e o resultado de 64 bits é armazenado nos registradores especiais `HI` (parte alta) e `LO` (parte baixa). Para acessar o resultado, são usadas instruções como `mflo` (para a parte baixa) e `mfhi` (para a parte alta).

8. A instrução `div` em MIPS realiza a divisão inteira de dois valores com sinal provenientes de registradores. Sua sintaxe é `div rs, rt`, onde o valor de `rs` é dividido por `rt`. O quociente da divisão é armazenado no registrador especial `LO`, e o resto no registrador `HI`. Para acessar esses valores, utiliza-se `mflo` (quociente) e `mfhi` (resto). Se `rt` for zero, o comportamento é indefinido.

9. A instrução `slt` (Set on Less Than) em MIPS compara dois valores de registradores e define um terceiro como 1 se a condição for verdadeira, ou 0 caso contrário. Sua sintaxe é `slt rd, rs, rt`, onde `rd` recebe 1 se `rs < rt` ou 0 caso contrário. É usada para implementações de comparações condicionais.

10. A instrução `beq` (Branch on Equal) em MIPS realiza um desvio condicional com base na comparação de dois registradores. Sua sintaxe é `beq rs, rt, offset`, onde o programa salta para o endereço calculado como `PC + offset` se os valores de `rs` e `rt` forem iguais. Caso contrário, a execução continua normalmente. É comumente usada para estruturas de controle, como loops e condições.

11. A instrução `beqz` (Branch if Equal to Zero) em MIPS é uma forma simplificada de `beq` e realiza um desvio condicional quando o valor de um registrador é igual a zero. Sua sintaxe é `beqz rs, offset`, onde o programa salta para o endereço `PC + offset` se o valor de `rs` for igual a zero. Caso contrário, a execução continua normalmente. Essa instrução é frequentemente usada para verificar se um registrador é zero antes de executar uma ação.

12. A instrução `bnez` (Branch if Not Equal to Zero) em MIPS realiza um desvio condicional quando o valor de um registrador é diferente de zero. Sua sintaxe é `bnez rs, offset`, onde o programa salta para o endereço `PC + offset` se o valor de `rs` for diferente de zero. Caso contrário, a execução continua normalmente. Essa instrução é comumente usada

para verificar se um registrador contém um valor diferente de zero antes de prosseguir com uma ação.

13. A instrução `j` (Jump) em MIPS realiza um desvio incondicional para um endereço de destino especificado. Sua sintaxe é `j target`, onde o programa salta para o endereço indicado por `target`, que é dado por um deslocamento de 26 bits que é concatenado com os 4 bits mais significativos do contador de programa (PC). A execução do programa continua a partir desse novo endereço. A instrução `j` é utilizada para realizar saltos longos, como em funções e saltos para rotinas.

14. A instrução `la` (Load Address) em MIPS não é uma instrução nativa da arquitetura, mas sim uma pseudoinstrução que carrega o endereço de uma variável ou rótulo (label) em um registrador. Sua sintaxe é `la rd, label`, onde o `rd` recebe o endereço de memória do `label`. A pseudoinstrução `la` é comumente usada para obter o endereço de variáveis globais ou locais em programas MIPS, facilitando o acesso à memória.

15. A instrução `syscall` em MIPS é usada para invocar serviços do sistema operacional, como entrada/saída (I/O), alocação de memória, ou controle do processo. Quando executada, o código no registrador `$v0` determina o tipo de serviço solicitado, e os outros registradores (como `$a0`, `$a1`, etc.) fornecem os argumentos necessários. A sintaxe é simplesmente `syscall`. Por exemplo, para imprimir um número, `$v0` é configurado com o código do serviço de impressão (1 para imprimir inteiro), e o valor a ser impresso é colocado em `$a0`.

É possível simular uma máquina de pilha, numa arquitetura real como MIPS.

Descrição

1. O objetivo do trabalho é modificar o compilador desenvolvido para gerar o código correspondente para MIPS do programa fonte.

2. As tabelas seguintes apresentam as correspondências das instruções MVS para a(s) instrução(ções) MIPS correspondente(s).

Operação	**MIPS**	**microcódigo**
Empilha	sw $a0 0($sp) addiu $sp $sp -4	pilha[topo] ← acc topo ← topo + 1 (4 bytes)
Desempilha	lw $t1 4($sp) addiu $sp $sp 4	t1 ← pilha[topo] topo ← topo - 1 (4 bytes)

Tabela 1: Tabela de equivalência MVS para MIPS

Instrução MVS	**Instrução MIPS**	**Microcódigo**
CRVG x	lw $a0 x	acc ← x
CRCT 10	li $a0 10	acc ← 10
ARZG x	sw $a0, x	x ← acc
SOMA	add $a0 $t1 $a0	acc ← t1 + acc
SUBT	sub $a0, $t1, $a0	acc ← t1 - acc

Instrução MVS	**Instrução MIPS**	**Microcódigo**
MULT	mult $t1, $a0 mflo $a0	acc(LO) ← t1 * acc
DIVI	div $t1, $a0 mflo $a0	acc(LO) ← t1 / acc
CMMA	slt $a0, $a0, $t1	acc ← (acc < t1)? 1 : 0
CMME	slt $a0, $t1, $a0	acc ← (t1 < acc)? 1 : 0
CMIG	beq $a0, $t1, Lx li $a0, 0 j Ly Lx: li $a0, 1 Ly: nop	se acc = t1 entao va para Lx acc ← 0 acc ← 1
NEGA	beqz $a0, Lx li $a0, 0 j Ly Lx: li $a0, 1 Ly: nop	se acc = 0 entao va para Lx acc ← 0 acc ← 1
CONJ	beqz $a0, Lx beqz $t1, Lx li $a0, 1 j Ly Lx: li $a0, 0 Ly: nop	se acc = 0 entao va para Lx se t1 = 0 entao va para Lx acc ← 1 acc ← 0
DISJ	bnez $a0, Lx bnez $t1, Lx li $a0, 0 j Ly Lx: li $a0, 1 Ly: nop	se acc != 0 entao va para Lx se t1 != 0 entao va para Lx acc ← 0 acc ← 1
DSVF L1	beqz $a0, L1	se acc = 0 entao va para L1
DSVS L1	j L1	
ESCR (string)	la $a0, string li $v0, 4 syscall	Em $a0 o endereço do valor para escrever $v0 = 4, imprime string syscall(4) = imprime string
ESCR (inteiro)	la $a0, inteiro li $v0, 1 syscall	Em $a0 o endereço do valor para escrever $v0 = 1, imprime inteiro syscall(1) = imprime inteiro
LEIA x (inteiro)	li $v0, 5 syscall sw $v0, x	$v0 = 4, lê inteiro depois da chamada, em $v0 está o valor lido x ← inteiro
FIMP	li $v0, 10 li $a0, 0 syscall	retorna 0 para o SO

3. Para simplificar essa tradução, ao invés de alocar registradores para as variáveis nas expressões, é feita a simulação de uma máquina de pilha em MIPS, usando a pilha (através do registrador $sp, stack pointer) e os registrados $a0 (acumulador) e $t1 (registrador temporário 1). Verifique nos exemplos de tradução e nos trechos de códigos do Programa

1 e Programa 2, que as operações MIPS são seguidas pelas operações com pilha para garantir que os registradores $a0 e $t1 contenham os valores que são operados (somados, subtraídos, testados, etc) e que o resultado da operação, ao final, esteja em $a0;

Programa 1: Algoritmo geral de tradução das operações com pilha

```
geracod ("e1 OP e2"):
    geracod (e1)
    print "sw $a0 0($sp)"
    print "addiu $sp $sp -4"
    geracod (e2)
    print "lw $t1 4(&sp)"
    print "addiu $sp $sp 4"
    traducao da operacao (OP): $a0 <- $t1 OP $a0
```

Programa 2: Exemplo de ações semântica para tradução de expressões com soma

```

expressao:
      expressao T_MAIS
        {
           //--- Empilha o resultado que esta em $a0
           printf ("\tsw $a0 0($sp)\n");
           printf ("\taddiu $sp $sp -4\n");
        }
      expressao
        {
           //--- Desempilha em $t1
           printf ("\tlw $t1 4($sp)\n");
           printf ("\taddiu $sp $sp 4\n");
           //--- SOMA ----
           printf ("\tadd $a0, $t1, $a0\n");
        }
```

Para o programa dobro.simples:

Programa 3: Programa para calcular o dobro

```
programa dobro
  inteiro n
inicio
  leia n
  escreva 2 * n
fimprograma
```

A tradução em MIPS deve ser:

```
.text
        .globl main
main:   nop
        li $v0, 5
        syscall
        sw $v0, n
        li $a0 2
        sw $a0 0($sp)
        addiu $sp $sp -4
        lw $a0 n
        lw $t1 4($sp)
        addiu $sp $sp 4
        mult $t1, $a0
        mflo $a0
```

```
15          li $v0, 1
16          syscall
17          la $a0 _ent
18          li $v0, 4
19          syscall
20  fim:    nop
21          li $v0, 10
22          li $a0, 0
23          syscall
24  .data
25          n: .word 1
26          _esp: .asciiz " "
27          _ent: .asciiz "\n"
```

4. O compilador deverá ainda verificar a compatibilidade de tipos nos comandos de atribuição, seleção e repetição.

5. O compilador deverá também possibilitar a escrita de literais, como no exemplo seguinte:

   Para o programa fatorial.simples:

Programa 4: Programa para calcular o fatorial

```
1  programa fatorial
2    inteiro n fat
3  inicio
4    escreva "Digite o valor de n: "
5    leia n
6    fat <- 1
7    enquanto n > 0 faca
8       fat <- fat * n
9       n <- n - 1
10   fimenquanto
11   escreva "fatorial = "
12   escreva fat
13 fimprograma
```

Cuja tradução para MIPS será:

```
1  .text
2          .globl main
3  main:   nop
4          la $a0 _const0
5          li $v0, 4
6          syscall
7          li $v0, 5
8          syscall
9          sw $v0, n
10         li $a0 1
11         sw $a0, fat
12 L1:     nop
13         lw $a0 n
14         sw $a0 0($sp)
15         addiu $sp $sp -4
16         li $a0 0
17         lw $t1 4($sp)
18         addiu $sp $sp 4
19         slt $a0, $a0, $t1
20         beqz $a0, L2
21         lw $a0 fat
22         sw $a0 0($sp)
```

```
23          addiu $sp $sp -4
24          lw $a0 n
25          lw $t1 4($sp)
26          addiu $sp $sp 4
27          mult $t1, $a0
28          mflo $a0
29          sw $a0, fat
30          lw $a0 n
31          sw $a0 0($sp)
32          addiu $sp $sp -4
33          li $a0 1
34          lw $t1 4($sp)
35          addiu $sp $sp 4
36          sub $a0, $t1, $a0
37          sw $a0, n
38          j L1
39  L2:     nop
40          la $a0 _const1
41          li $v0, 4
42          syscall
43          lw $a0 fat
44          li $v0, 1
45          syscall
46          la $a0 _ent
47          li $v0, 4
48          syscall
49  fim:    nop
50          li $v0, 10
51          li $a0, 0
52          syscall
53  .data
54          n: .word 1
55          fat: .word 1
56          _esp: .asciiz " "
57          _ent: .asciiz "\n"
58          _const0: .asciiz "Digite o valor de n: "
59          _const1: .asciiz "fatorial = "
```

6. Para testar o código compilado deverá ser usado o Simulador da Máquina MIPS denominado MARS (https://dpetersanderson.github.io/download.html). A linha de comando para testar o código gerado será:

```
1  java -jar Mars4_5.jar fatorial.asm
```

Onde 'fatorial.asm' é o arquivo que contém o código MIPS gerado pelo compilador.

Entrega

1. Incluir um comentário no cabeçalho de cada programa fonte com o seguinte formato:

```
1  /*+-------------------------------------------------------------------
2    |            UNIFAL - Universidade Federal de Alfenas.
3    |              BACHARELADO EM CIENCIA DA COMPUTACAO.
4    | Trabalho..: Geracao de codigo MIPS
5    | Disciplina: Compiladores
6    | Professor.: Luiz Eduardo da Silva
7    | Aluno.....: Fulano da Silva
8    | Data......: 99/99/9999
9    +-------------------------------------------------------------------*/
```

2. A pasta com o projeto deverá incluir o seguinte arquivo Makefile:

```
simples : lexico.l sintatico.y utils.c;\
        flex -t lexico.l > lexico.c;\
        bison -v -d sintatico.y -o sintatico.c;\
        gcc sintatico.c -o simples

limpa  : ;\
        rm -f lexico.c sintatico.c sintatico.output *~ sintatico.h simples\
```

3. Conforme definido neste script do make, o executável do compilador deverá ter o nome "simples"e ser chamado através da seguinte linha de comando:

```
./simples fonte[.simples]
```

Onde o nome do programa fonte poderá conter ou não a extensão e o arquivo gerado terá o mesmo nome do fonte e extensão '.asm'. Por exemplo, se a chamada for:

```
./simples teste1
```

deverá existir um arquivo de nome 'teste1.simples' e o compilador deverá gerar o código mnemônico MIPS no arquivo 'teste1.asm'

4. **Enviar num arquivo único (.ZIP), a pasta do projeto com somente os arquivos fontes (lexico.l, sintatico.y, utils.c e makefile), através do Envio de Arquivo do MOODLE.**