MAC-221 — Construção de Montadores
MAC-222 — Sistemas de Programação

1998 — Especificação da terceira fase do Montador p/ MIPS

Idéias gerais

Na terceira fase do montador para o processador RISC MIPS R2000 deve-se completar o montador. A novidade em relação à segunda fase é a montagem dos "opcodes" proprimante ditos.

Nesta fase deverá ser utilizado o analisador léxico gerado na primeira fase e o analisador sintático gerado na segunda fase. A maior parte das alterações no código será feita nas ações associadas às regras gramaticais presentes no fonte do bison[1].

O analisador léxico

Nesta fase deverá ser utilizado o analisador léxico gerado na primeira fase. Poucas alterações serão necessárias. A principal diferença é que os valores semânticos deverão ser tratados agora, inclusive o valor semântico do token `STR`.

Os módulos já definidos, incluindo-se aquele gerado pelo flex e que implementa a função `yylex()` e o módulo necessário à manipulação da tabela de hashing serão usados como antes. O módulo `testlex` será omitido mas pode ser utilizado nalgum outro executável de forma a se obter uma listagem dos ítens léxicos que estão sendo sucessivamente remetidos ao analisador sintático.

O analisador sintático

Todas as *instruções*, *pseudoinstruções* e *diretivas* deverão ser reconhecidas corretamente.

No caso das instruções, o reconhecimento sintático deve ser *estrito*, ou seja, todas as possíveis combinações de operandos devem ser reconhecíveis e nenhuma a mais. Todas as diretivas `.align`, `.text`, `.data`, `.sdata`, `.rdata`, `.comm`, `.lcomm`, `.word`, `.byte`, `.half`, `.double`, `.ascii` e `.asciiz` também devem ter reconhecimento sintático *estrito*, bem como as pseudoinstruções `la` (load address), `li` (load immediate), `move` e `nop` (no operation).

Todas as demais pseudoinstruções e diretivas podem ter um reconhecimento sintático *relaxado*, ou seja, todas as combinações corretas de operandos devem ser reconhecíveis mas pode-se reconhecer como sintaticamente corretos operandos incorretos. Isto não trará problemas para estas pseudoinstruções e diretivas pois não serão implementadas a princípio.

---

[1] Quanto à depuração do analisador sintático, além da opção `-v` dada ao bison, que gera o arquivo `.output`, pode-se inserir a linha `#define YYDEBUG 1` à seção de cabeçalhos do arquivo fonte do bison e uma atribuição não nula à variável `yydebug` faz um "tracing" no empilhamento dos tokens e das reduções realizadas.

Várias instruções dos tipos `R3` e `J` podem ter uma segunda sintaxe, e devem ser implementadas, como já comentado na segunda fase. Mais detalhes também podem ser encontrados no ítem 3 do arquivo `Observacoes.txt`.

Recuperação e relatório de erros

O programa deve ser capaz de se recuperar de um erro sintático, prosseguindo a análise sintática nas linhas subsequentes. Quanto à função `yyerror()` deve dar informações sobre onde ocorreu o erro sintático. Ao menos o número da linha em que foi detectado o erro (basta que `yylex()` conte os caracteres de mudança de linha) e o valor de `yytext` do último token lido deve ser impresso.

De fato, o arquivo fornecido `testsint.c` já possui uma implementação de uma tal função `yyerror()` e também tem uma função `main()` que inicializa a tabela de símbolos, lê os parâmetros da linha de comando e chama `yyparse()`. Este arquivo, como todos os demais que foram fornecidos, pode ser alterado. Em particular, a função `main()` nele presente não cuida de chamar a função `savecoff()` ou tratar adequadamente o caso de um montador de um ou de dois passos. Esta função `main()` certamente *deverá* ser alterada.

Será recusado um projeto que simplemente dê uma mensagem do tipo '`parse error`' sem dar as informações necessárias à localização da linha e do token em que ocorreu o erro.

Arquivos fornecidos

O diretórios (que são idênticos)

```
<http://www.ime.usp.br/~alair/mac221-98/TerceiraFase/>
<http://www.ime.usp.br/~alair/mac222-98/TerceiraFase/>
```

contêm a especificação da terceira fase e arquivos auxiliares à confecção do montador. Um arquivo com todo o conteúdo do diretório pode ser encontrado em

```
<http://www.ime.usp.br/~alair/mac221-98/TerceiraFase.tar.gz>
<http://www.ime.usp.br/~alair/mac222-98/TerceiraFase.tar.gz>
```

Para ser extraído, basta usar o comando

```
tar xvzf TerceiraFase.tar.gz
```

no linux[2].

Nesses diretórios são fornecidos os arquivos:

- `Tokens-2.6` com a especificação dos tokens;
- `Observacoes.txt` com várias observações sobre o projeto.

Estes dois arquivos *fazem parte da especificação do projeto* e poderão ser alterados de forma a dar mais esclarecimentos porventura necessários.

---

[2]Quem usa o `DJGPP` pode fazer o mesmo já que o `DJGPP` possui o `tar` e o `gzip`.

São fornecidos ainda vários arquivos que *podem* ser usados na implementação do projeto:

- `symrec.h` com a especificação de uma tabela de símbolos;
- `symrechash.c` com uma implementação da tabela de símbolos através de uma tabela de hashing (com uma função de hashing simples);
- `initsymrec.c` com uma função que inicializa a tabela de hashing;
- `token2hash` um script AWK que processa o arquivo `Tokens-2.6` e gera um vetor de inicialização com as palavras reservadas a serem inseridas inicialmente na tabela de Hashing;
- `testsint.c` com a função `yyerror()` e a função `main()`;
- `token2bison` um script AWK que processa o arquivo `Tokens-2.6` e gera definições como `%token` e `%union` para o bison;
- `Makefile` um arquivo descritor das dependências entre os diferentes arquivos do projeto;
- `coff.h` um arquivo com o a descrição dos cabeçalhos e seções presentes no executável a ser gerado;
- `savecoff.h` um arquivo com o protótipo da função `savecoff()`, necessária à geração do arquivo executável a ser simulado pelo `SPIM`;
- `savecoff.c` um arquivo que implementa a função `savecoff()`;
- outros arquivos/scripts AWK relacionados às fases anteriores do projeto;
- `exemplos` um diretório que será comentado mais tarde;
- `<ftp://ftp.linux.ime.usp.br/pub/alair/>`, um diretório com programas auxiliares, que também será comentado mais tarde.

Usando os arquivos fornecidos

Com os arquivos fornecidos, basta escrever os arquivos `analex.l`, `anasintdef.y` e `anasintregras.y` compatíveis e alterar propriamente o arquivo `testasint.c`. Rodando `make`, o `Makefile` fornecido é analisado e é gerado o arquivo executável `./monta`. Se só estes arquivos forem implementados, basta entregar um arquivo-pacote `.tar`[3] (ou um `.zip`) dos quatro arquivos. Se os arquivos fornecidos forem alterados ou outros arquivos forem acrescentados, todos estes arquivos devem ser entregues neste pacote. Com o pacote entregue e extraído sobre um diretório contendo previamente os mesmos arquivos entregues na especificação da TerceiraFase, deve-se ser capaz de gerar o executável, de nome `monta` simplesmente rodando `make`.

Documentação fornecida

Está à disposição no Xerox o apêndice A da primeira edição do excelente livro "Computer Organization & Design: The Hardware / Software Interface" de autoria de Patterson & Hennessy. Este apêndice contém os 'opcodes' das instruções do MIPS 2000 bem como sua sintaxe e *DEVE* ser consultado. Há alguns erros no apêndice e o arquivo `Observacoes.txt` e `Tokens-2.6` têm in-

---

3

O comando `tar cvzf meutar.tgz analex.l anasintdef.y anasintregras.y testasint.c` monta o pacote.

formações mais confiáveis quando houver discrepância. Uma terceira fonte de informação é o manual do SPIM que também está no xerox. Já no apêndice A há alguma informação referente ao SPIM.

Diretório ***exemplos***

Este diretório possui vários arquivos assembly (`.s`) para o MIPS. Possui também alguns arquivos C (`.c`) que podem ser cross-compilados de forma que se gerem os correspondentes arquivos `.s`. O arquivo `Makefile` presente neste diretório oferece regras de como fazer esta cross-compilação.

O "cross-compiler", disponível apenas em linux, pode ser encontrado em

`<ftp://ftp.linux.ime.usp.br/pub/alair/>`

juntamente com o simulador `spim`. (Já estão instalados na rede linux do IME.) Há também uma versão do simulador `spim` que roda em DOS.

O diretório `<ftp://ftp.linux.ime.usp.br/pub/alair/cross-compiler/>` contém um cross-compiler e cross-assembler que gera código para uma máquina MIPS little endian: uma decstation rodando o sistema operacional ultrix. O arquivo `<ftp://ftp.linux.ime.usp.br/pub/alair/spim.tar.gz>` é uma versão do `SPIM` (simulador MIPS) que simula a execução de um binário gerado para esta decstation, com algumas restrições que serão vistas abaixo. Eventuais versões do `SPIM` encontradas na internet não são suficientes.

Cross-compilação e simulação do MIPS

Juntamente com os executáveis para linux do SPIM (e XSPIM), que permitem a simulação da execução de um programa gerado para a plataforma decstation/ultrix, foram fornecidos vários exemplos e arquivos necessários à confecção destes exemplos. Estes exemplos podem ser executados dentro do SPIM como veremos abaixo e encontram-se no subdiretório `exemplos`.

O SPIM não emula as chamadas de sistema necessárias à implementação de toda uma biblioteca de funções como a `libc`. Isto em particular implica que os arquivos `.c` cross-compilados não podem fazer referência a funções como `printf`, `scanf` ou `fopen`. O SPIM fornece dez chamadas de sistema enumeradas em `spim-syscall.h`. Uma interface para reconhecimento destas chamadas a partir de programas `.c` está descrita no arquivo `spimio.h` e a implementação desta interface está presente no arquivo `start.asm`, um arquivo assembly que deve ser preprocessado primeiramente, gerando o arquivo `start.s`. O arquivo `start.o` deve ser linkado com cada arquivo objeto cross-compilado que se deseje simular no SPIM. Além da implementação da interface antes descrita, o arquivo faz as inicializações e finalizações necessárias para executar a função `main`. Existe um `Makefile` que descreve como devem ser gerados os executáveis para serem simulados no SPIM.

Tomemos como exemplo o arquivo `hello.c`. O arquivo `hello.c`, é compilado para um arquivo assembly `hello.s` (a opção `-S` é usada para a geração do assembly). O arquivo `hello.s`, é então montado para um arquivo objeto `hello.o`. Uma listagem dos bits montados pode ser obtida com a opção `-a -EB` fornecida ao montador e pode-se fazer uma separação primária dos campos de bits fornecendo esta saída como entrada para o script AWK `separabits`. O arquivo `hello.o` é então linkado com `start.o`. É fornecida ao ligador `ld` a opção `-T` `script-spim` que diz em que endereços devem ser alocados quais segmentos de forma que o SPIM possa simular o programa. O SPIM supõem que hajam somente dois segmentos: `.text` que começa em $0x04000000$, e `.data` que começa em $0x10000000$. Para a execução do arquivo executável, já linkado, executamos o spim como da forma:

```
./spim -execute hello
```

ou ainda trocando `./spim` por `./xspim -fn fixed`.

Geração de código

O montador deverá gerar código suficiente para que os arquivos `test0.s`, `test1.s` e `test2.s` sejam montados e executados pelo SPIM. Estes arquivos `test*.s` têm versões editadas dos arquivos assembly gerados pela cross-compilação dos exemplos acima mencionados. Esta edição é necessária para incluir o conteúdo do arquivo `start.asm`, já que o arquivo gerado pelo montador não será linkado. (Isto requeria a geração de um arquivo objeto com as informações necessárias a uma realocação e solução de referências cruzadas.)

Como já especificamos na segunda fase, cada diretiva, pseudoinstrução e instrução deverá ser reconhecida sintaticamente. Quando nada além da análise sintática for feito, diremos que a linha está sendo *ignorada*. Caso contrário diremos que está sendo *implementada*. A implementação de uma instrução ou pseudoinstrução implica a geração de código e a montagem de alguns bytes. A implementação de uma diretiva tipicamente altera a configuração corrente do montador e pode também implicar a cópia de alguns bytes nalgum segmento. De fato, cada diretiva é um caso. Todas as diretivas implementadas deverão ter um reconhecimento sintático estrito.

Serão implementadas todas as instruções. Instruções que possuirem uma segunda sintaxe também devem ter esta segunda sintaxe implementada. As pseudoinstruções `la` (load address), `li` (load immediate), `move` e `nop` deverão ser implementadas.

Uma pseudoinstrução `la $v0,x` pode ser implementada pelos opcodes correspondentes às instruções:

```
lui     $v0,xh
addiu   $v0,$v0,xl
```

onde `xh` corresponde aos 16 bits mais significativos do endereço associado ao rótulo `x` e `xl` corresponde aos 16 bits menos significativos. Caso o endereço `x` possa ser alcançado por `$gp` mais um deslocamento de 16 bits (o que significa que o endereço está entre $0x10000000$ e $0x1000FFFF$) a pseudoinstrução também pode ser implementada pelo opcode correspondente à instrução

```
addiu   $v0,$gp,x-0x10008000
```

já que o valor de `$gp` é $0x10008000$. Esta hipótese pode ser pressuposta pelo montador e isto leva a um montador que restringe ao máximo de $64Kbytes$ no segmento de dados. Uma pseudoinstrução `li $v1,num` pode ser implementada de duas formas: pelo opcode correspondente à instrução

```
addiu   $v1,$zero,num
```

caso `num` seja um inteiro de 16 bits; ou ainda pelos opcodes correspondente às instruções

```
lui     $v1,numh
ori     $v1,$v1,numl
```

onde `numh` corresponde aos 16 bits mais significativos e `numl` corresponde aos 16 bits menos significativos de `num`. Quanto à pseudoinstrução `move $t1,$t2`, ela pode ser implementada pelo mesmo opcode da instrução `addu $t1,$t2,$zero`. A pseudointrução `nop` corresponde ao opcode 0 (zero). Pode-se fazer testes com o cross-assembler caso se queira verificar o código que está sendo gerado pelo `GNU as` para estas pseudoinstruções. O arquivo `pseudos.s` no diretório `exemplos` já possui estas pseudoinstruções.

As seguintes diretivas deverão ser *implementadas*: `.align`, `.text`, `.data`, `.ascii`, `.asciiz`. As diretivas `.sdata` e `.rdata` serão "implementadas" como sinônimos de `.data`. Os possiveis parâmetros numéricos que seguem às diretivas `.text`, `.data`, `.sdata` e `.rdata` são ignorados. As diretivas `.word`, `.byte`, `.half`, `.double`, `.ascii` e `.asciiz` também devem ter reconhecimento sintático *estrito*, bem como as pseudoinstruções `la` (load address), `li` (load immediate), `move` e `nop`.

As diretivas `.comm` e `.lcomm` serão implementadas da seguinte forma: `.comm primo,2044` terá o efeito de definir o rótulo `primo` no corrente endereço do segmento `.data` e reservar-lhe um total de 2044 bytes que serão inicializados com zeros. Estas diretivas deveriam ser tratadas pelo linker, usualmente. As diretivas `.word`, `.byte`, `.half` e `.double` também devem ser implementadas.

Outras diretivas e/ou pseudoinstruções também poderão ser implementadas.

Para cada um dos segmentos `.text` e `.data` deveremos ter: um vetor, que armazena os bytes que são montados nos respectivos segmentos; e um contador de endereços, que conta os endereços correntemente associados à próxima instrução/palavra a ser montada no segmento. Uma vez terminada a montagem do programa, deve-se gerar um arquivo executável no formato COFF littleMIPS que possa ser carregado no SPIM. Para tanto, faça uso da função `savecoff()`, cujo protótipo está fornecido no arquivo `savecoff.h`. O contador de endereços do segmento `.text` deve ser inicializado com $0x400000$ e o do segmento `.data` deve ser inicializado com $0x10000000$. Isto significa que se um rótulo é associado ao primeiro byte do segmento `.text` ele estará associado ao endereço $0x400000$.

Entrega

A entrega deste pacote deve ser feita de forma eletrônica em

`<http://panda.ime.usp.br>`

até o dia 7 de dezembro. Este prazo não pode ser prorrogado devido à data limite de entrega de notas.

Quanto à divisão de grupos, será divulgado o processo de formação dos grupos nas páginas do curso.