UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Desenvolvimento de um Subconjunto de uma Linguagem de Alto Nível Real e um Compilador para o Processador Didático do ICMC

Desenvolvimento de um Subconjunto de uma Linguagem de Alto Nível Real e um Compilador para o Processador Didático do ICMC

Marcelo Koti Kamada

Orientador: Eduardo do Valle Simões

Monografia de conclusão de curso apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP - para obtenção do título de Bacharel em Ciências de Computação

Área de Concentração: Arquitetura de Computadores e Compiladores.

USP – São Carlos
Novembro de 2014

Resumo

Neste documento, apresenta-se o desenvolvimento de um compilador de uma linguagem de programação baseada em C para a linguagem de máquina de um processador didático desenvolvido no Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo. São apresentados também um montador e um simulador desenvolvidos em trabalhos anteriores, assim como um jogo árcade do tipo Pacman que foi desenvolvido especificamente para verificar e depurar o código gerado pelo compilador. Trata-se, portanto, de uma continuação desses trabalhos, a fim de produzir um kit didático para ensino de lógica digital, organização e arquitetura de computadores aos alunos do curso de Bacharelado em Ciências da Computação que inclui o hardware de um processador, um simulador de sua operação e as ferramentas de compilação e montagem de código especificamente desenvolvidos para o mesmo.

Palavras Chave: Compilador, Simulador, Arquitetura de Computadores.

Sumário

Lista de Figuras

Lista de Abreviaturas e Siglas

FPGA –Field Programmable Gate Array

RISC – Reduced Instruction Set Computer

ICMC – Instituto de Ciencias Matematicas e Computacao

CAD – Computer Aided Design

VHDL - Very High Speed Integrated Circuits Hardware Description Langurage

HDL - Hardware Description Langurage

CAPÍTULO 1: INTRODUÇÃO

1.1. Contextualização e Motivação

Com a evolução das arquiteturas de computadores, a partir da década de sessenta, novas linguagens de programação, como Fortran e Cobol, surgiram para prover abstrações aos programadores; além de utilizar de maneira eficiente os novos hardwares disponíveis, desenvolvidos em série e que compartilhavam a mesma linguagem de máquina. Para que isso fosse possível, os desenvolvedores de compiladores tiveram que acompanhar com novos algoritmos e estruturas de dados que representassem novas funcionalidades [Aho e Ullman 2007, Tanenbaum e Woodhull 2006].

O desenvolvimento na área de compiladores aumentou as áreas de atuação destes, que não só tratam de gerar código de máquina para linguagens de alto nível, como também atuam no desenvolvimento de novas arquiteturas de computadores como foi o caso da RISC (Reduced Instruction Set Computer) [1]; traduzem códigos binários de uma plataforma para outra, aumentando o tempo de vida de códigos legados, aumentando o número de softwares disponíveis para arquiteturas menos populares; permitem a simulação compilada como, por exemplo, softwares que fazem a tradução de máquinas de estados para HDL (Hardware Description Langurage); e otimizam códigos usando paralelismo e hierarquia de memória.

Todas estas funções estão relacionadas à capacidade que os compiladores possuem em fornecer abstrações aos programadores que não precisam lidar diretamente com instruções de baixo nível, nem ter conhecimentos aprofundados da arquitetura alvo [Aho e Ullman 2007]. Porém, com o aumento da complexidade das arquiteturas comerciais e consequentemente dos compiladores que produzem código para elas, a utilização destes para ensino fica bastante comprometida, uma vez que os princípios básicos de operação, mais interessantes didaticamente, ficam difundidos entre estruturas mais complexas.

Durante as disciplinas de Introdução a Eletrônica, Elementos de Lógica Digital, Sistemas Digitais e Organização de Computadores do Instituto de Ciências Matemáticas e

de Computação da Universidade de São Paulo ICMC-USP, os alunos desenvolvem um processador didático [Kamada e Simões 2012] com arquitetura RISC que pode ser programado usando uma linguagem montadora própria e um montador especialmente desenvolvido para ele. A execução dos programas pode ser feita diretamente sobre o hardware do processador implementado em uma placa de desenvolvimento contendo um FPGA (Field-Programmable Gate Array) [Brown 1992] da empresa Altera ou através de um simulador multiplataforma em um IBM-PC, desenvolvido em linguagem C especificamente para facilitar a depuração de código. Todos estes componentes juntos formam um kit didático para ensino de logica digital, organização e arquitetura de computadores disponível no laboratório didático SAP2 do ICMC-USP.

Este projeto continua trabalhos anteriores de Iniciação Científica, acrescentando ao kit, Figura 1, um compilador de uma linguagem de alto nível baseada em C para a linguagem montadora do processador, facilitando a transição dos alunos entre as linguagens, pois conta com uma arquitetura alvo didática e um simulador para depuração de código passo a passo. Com o compilador, será possível a elaboração de programas de maior porte e complexidade do que os atuais feitos em linguagem montadora. Futuramente será possível incrementar mais uma vez o kit com um Sistema Operacional, pois, desta forma, o kit abordará todas as etapas do ciclo de um programa.

Figura 1Kit para ensino de lógica digital, contando com processador, montador e simulador

1.2. Objetivos

Este trabalho visa desenvolver um compilador que permite a tradução de código em uma linguagem baseada em C para código em linguagem montadora que pode ser transformado em código de máquina entendido pelo processador do ICMC, através de um montador que foi resultado de trabalhos anteriores. Este compilador vem complementar um kit para ensino de lógica digital, organização e arquitetura de computadores que vem sendo utilizado nas aulas práticas para os alunos do curso de Bacharelado em Ciências de Computação do ICMC. O kit é composto pelo hardware do processador que contém módulos de vídeo e teclado, um montador de linguagem montadora para o código do processador, um simulador do processador que roda em IBM-PC e permite o desenvolvimento dos projetos dos alunos dentro e fora dos laboratórios, e agora o compilador.

Como o compilador será utilizado como uma ferramenta didática, o código gerado por ele deverá conter mensagens de debug para facilitar a leitura do código em linguagem montadora ou Assembly gerado pelos alunos. A linguagem desenvolvida é uma versão reduzida da linguagem C, utilizadas pelos alunos nas disciplinas de introdução a programação. Esta linguagem deve permitir declarações de variáveis simples, vetores e matrizes em escopo global ou local, operações aritméticas em expressões, declaração de funções, operações com ponteiros e endereços de memória, laços como for while, desvio de execução com if/else e goto, breakpoints, printf e getch. A gramática da linguagem, listando todas as regras aceitas pelo compilador está no apêndice A.

Com o compilador, completa-se todas as etapas no processo de geração de código desde uma linguagem de alto nível para código de máquina. Assim é possível fazer uma distinção didática de todo o processo de como um código de alto nível é traduzido para Assembly pelo compilador, depois para código de máquina pelo montador e finalmente executado pelo processador.

1.3. Organização da Monografia

A seção a seguir introduz conceitos, técnicas e trabalhos anteriores que foram a base para a realização deste. No capitulo 3, descreve-se as etapas, justificando cada uma, no desenvolvimento do compilador, e finalmente, no capitulo 4 mostra-se os resultados obtidos e a conclusão do trabalho.

CAPÍTULO 2: REVISÃO BIBLIOGRÁFICA

2.1. Considerações Iniciais

As seções seguintes estão organizadas da seguinte forma: primeiramente explica-se os conceitos usados neste trabalho e em seguida, os trabalhos anteriores que serviram como base para que este fosse possível.

2.2. Conceitos e Técnicas Relevantes

2.2.1 Gramaticas Livre de Contexto

Gramaticas Livre de Contexto [2] são usadas para especificar o que é permitido nas linguagens de programação através de regras, incluindo a linguagem C usada neste trabalho. Estas linguagens são formadas por quatro caraterísticas principais:

1. Um conjunto de tokens ou símbolos terminais que representam os símbolos elementares da linguagem. Exemplo: números, strings, palavras reservadas.

2. Um conjunto de não terminais, que servem para a criação das regras da linguagem;

3. Um conjunto de regras ou produções que seguem o formato A -> aB. No qual o lado esquerdo da seta aceita apenas um não terminal “A”, e o lado direito aceita uma sequencia de terminais “a” e não terminais “B” em qualquer ordem;

4. Um símbolo inicial para especificar a partir de onde começa a gramática, sendo ele um dos não terminais;

A gramatica usada no trabalho, antes das alterações necessárias para a geração de código, está no apêndice A, sendo que todos os símbolos terminais estão em letras maiúsculas e os não terminais em letras minúsculas.

2.2.2 Tabela de Símbolos

A tabela de símbolos [Aho e Ullman 2006], Figura 2, é uma estrutura de dados que armazena informações sobre os identificadores usados no código lido pelo compilador, ele usa essas informações para checar o código por erros que não seriam descobertos com apenas uma gramática livre de contexto. Gramáticas livre de contexto, como o próprio nome diz, não guardam o contexto que em estão, e por isso não é possível saber, por exemplo, se um identificador já foi declarado em um dado escopo; mas isso se torna possível quando se usa uma tabela de símbolos e pode-se checar se ela contém o identificador.

i	variavel	int	28000	0	1	int	4	null	null
char	tipo	null	-1	0	0	-1	-	null	null
int	tipo	null	-1	0	0	-1	-	null	null
Nome	id	tipo	endereço	ponteiro	nbytes	tipo_elementar	valor	parametros	função

Figura 2 Formato seguido pela Tabela de Símbolos. A tabela funciona como uma pilha, com tipos e funções built-in na base e variaveis, funções e parametros inseridos no topo

2.3. Trabalhos Relacionados

Neste projeto, o compilador gera código Assembly específico para o processador do ICMC. Esse processador foi implementado utilizando a linguagem de descrição de

hardware VHDL (Very High Speed Integrated Circuit Hardware Description Language) [Brown 2005], disponível na ferramenta de CAD Quartus II [3] da empresa Altera e uma FPGA Cyclone 2 da placa DE2-70 da empresa Terasic [4]. Ele possui uma arquitetura RISC de 16 bits, executa a uma velocidade variável de até 50MHz, conta com 8 registradores genéricos, uma ULA para operações aritméticas, memória de 32Kb compartilhada pelo programa e variáveis (arquitetura de Von-Neuman)[5], uma memória para o mapa de caracteres, e módulos de interface com monitor e teclado para interação com usuário. Em relação ao Assembly, apêndice B, a linguagem conta com 58 instruções entre operações aritméticas e lógicas, comparações entre registradores, movimentação de dados entre registradores e memória, desvio de execução, operações de push e pop com stack, chamada de rotinas, entrada de dados do teclado e saída para vídeo. A arquitetura do processador está apresentada na Figura 3.

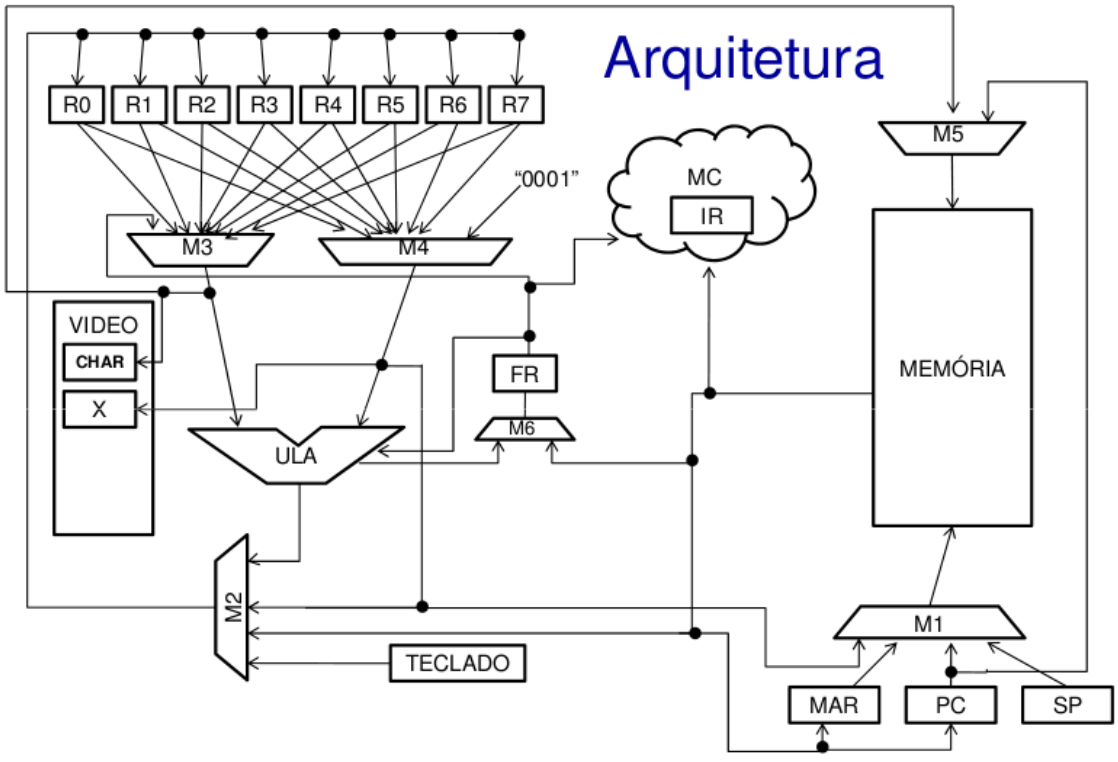


Figura 3 Arquitetura do Processador do ICMC com seus 8 registradores, memória de vídeo e de programa e módulos de teclado e vídeo.

Para executar o código Assembly no processador, é necessário converter ele para o código binário entendido pelo processador através de um montador [Moreira, Martins e Simões]. Este código é enviado para a placa de FPGA onde o processador está implementado através de um cabo USB. Um exemplo da execução de um jogo Pacman pelo processador está na Figura 4.

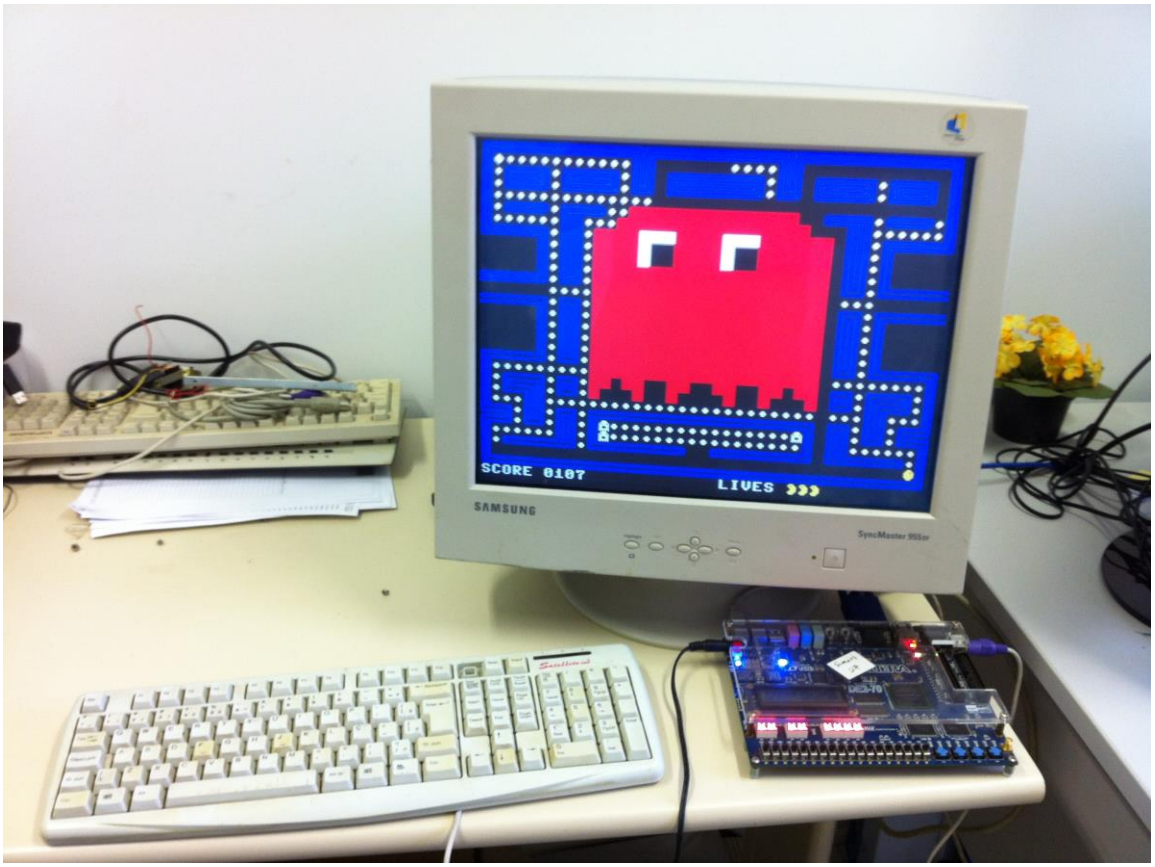


Figura 4 Processador do ICMC na placa de FPGA DE2-70 da Altera rodando o jogo Pacman.

Uma alternativa para a execução do código é o simulador [Kamada e Simões 2012] mostrado na Figura 5. Esse simulador multiplataforma (Linux, Windows e Mac) permite o teste exaustivo dos códigos gerados pelo compilador, que podem ser executados em três velocidades ou passo a passo para depuração, também permite visualizar o conteúdo dos registradores, o código Assembly sendo executado e inserção de breakpoints no código.

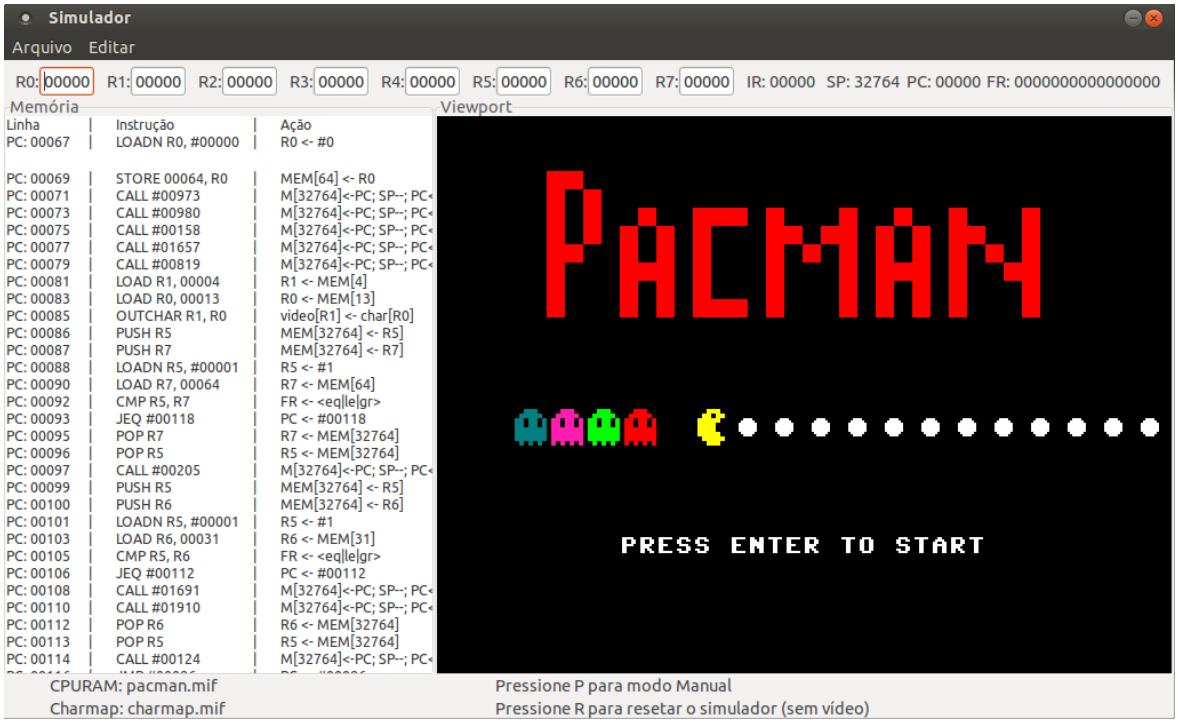


Figura 5 Interface do simulador rodando o mesmo jogo Pacman da Figura 4. O simulador permite visualizar o código Assembly sendo executado, o conteúdo dos registradores e labels com os comandos aceitos.

2.4. Considerações Finais

Neste capitulo introduziu-se os conceitos de tabela de símbolos e gramáticas livres de contexto que servem para o entendimento do próximo capitulo. Também se introduziu o processador, o montador e o simulador que são resultados de trabalhos anteriores e serviram de base para realização deste. No próximo capitulo, descrevem-se as etapas de projeto, os problemas encontrados e as soluções que foram encontradas durante o desenvolvimento do compilador.

CAPÍTULO 3: DESENVOLVIMENTO DO TRABALHO

3.1. Considerações Iniciais

Uma descrição do problema é apresentada em 3.2, apresentando a metodologia escolhida para o trabalho. Nos itens seguintes, descreve-se cada passo da metodologia.

3.2. Descrição do Problema

O desenvolvimento de compiladores é divido em módulos como em um pipeline, cada módulo possui uma tarefa específica e gera dados para o próximo módulo. A divisão adotada no trabalho está representada na Figura 6.

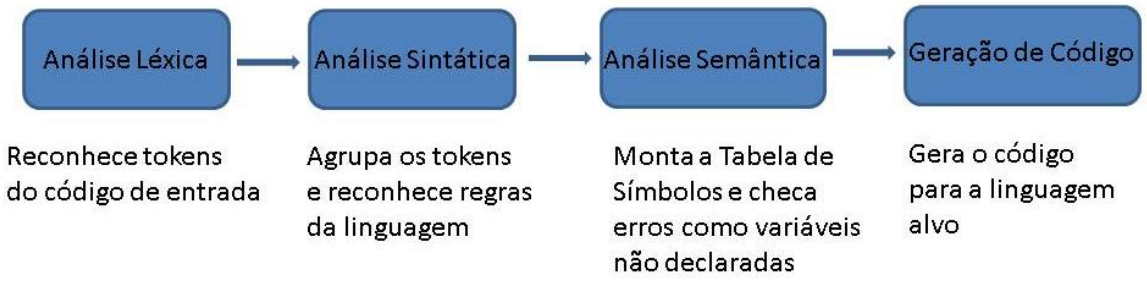


Figura 6 Diagrama de divisão do compilador em seus módulos como um pipeline. O código fonte é recebido pela analise léxica e o código Assembly é produzido pela geração de código.

O trabalho foi dividido no desenvolvimento dos módulos: análise léxica, análise sintática, análise semântica e geração de código, essa divisão é comum e é descrita na literatura [Aho e Ullman 2007]. A checagem de tipo não foi implementada porque o processador só suporta inteiros, por isso os tipos de dados ficaram restritos a char e inteiros de 16 bits sem sinal, sendo que operações entre os dois tipos não são checados já que eles possuem a mesma representação no hardware.

Cada módulo recebe como entrada a saída do anterior, por exemplo: o analisador léxico faz a leitura do código fonte, elimina os comentários e gera uma sequência de tokens que provêm do reconhecimento de expressões regulares. Os tokens reconhecidos são enviados para o parser que tem o trabalho de "encaixar" os tokens em uma das regras da gramática da linguagem ou alertar para um erro sintático.

A partir do reconhecimento dos tokens nas regras da gramática, o módulo de análise semântica monta uma tabela de símbolos que possui identificadores, palavras reservadas, constantes e funções *built-in*. A tabela de símbolos por sua vez, fornece à análise semântica a capacidade de gerar erros caso um identificador tenha sido declarado mais de uma vez ou se o programador está usando uma variável que não foi declarada.

Por fim, o módulo de geração de código utiliza a tabela de símbolos da analise semântica e o reconhecimento das regras da linguagem para fazer o seu papel de gerar código para a linguagem alvo.

3.3. Descrição das Atividades Realizadas

3.3.1. Definição da Linguagem e Gramática

O primeiro passo do trabalho foi definir a gramatica da linguagem, para isso, começou-se com a gramática da linguagem C, disponível em [6]. Como o processador possui limitações que impedem a implementação de todas as funcionalidades da linguagem, alterou-se a gramatica, retirando dela o que não era suportado. Como, por exemplo, tipos de dados como float, double e boolean que não são suportados pois o processador trata apenas de inteiros de 16 bits sem sinal, por isso a gramatica aceita apenas os tipos int e char, sendo que operações com char são tratadas da mesma forma que int. Outra modificação foi a remoção da "linkagem" de múltiplos arquivos fonte através da palavra reservada include, essa decisão foi tomada porque a memória atual se limita a 32Kb.

A linguagem usada no compilador suporta declaração de variáveis globais e locais, desvio de execução com if/else, lacos com while e for, operações aritméticas e a resolução de expressões com estas, declaração de funções. O compilador não suporta linkagem com outros arquivos via include, uso de macros via define, estruturas como switch/case, do while, structs, union, enum, tipos de dados como float, double, limita as matrizes para no máximo duas dimensões, e não permite a declaração de função seja separada da implementação

3.3.2 Analise Léxica

Com a gramatica definida, o passo seguinte foi a elaboração do analisador léxico com ajuda da ferramenta Flex [7] [Levine 2009]. O Flex (Fast Lexical Analyzer Generator) é um software livre escrito por Vern Paxson que, assim como o seu antecessor o Lex, gera scanners que leem o código, separando este em “tokens”.

Como o Flex espera um formato especifico para gerar o scanner, que também pode ser chamado de analisar léxico, neste passo teve-se apenas de definir expressões regulares que reconhecessem os tokens e ignorassem os comentários da gramatica do passo anterior.

O arquivo de entrada do Flex é um arquivo texto com a extensão .l, sendo um exemplo mostrado na Figura 7, esse arquivo é dividido em três partes: includes de bibliotecas C, expressões regulares e por fim uma parte para código, sendo essa opcional. A partir deste arquivo de entrada, a ferramenta gera um analisador léxico em código C, assim pode-se desenvolver os outros módulos linkando este arquivo com o resto do projeto.

```
%{
    /* includes de bibliotecas */
    #include <stdio.h>
%}

%%

    /* expressao para reconhecer a palavra reservada int */
"int"   {
            /* Codigo entre as chaves e' executado quando "int" for reconhecido */
            printf("achou int");
        }

%%

/* Codigo C opcional */
```

Figura 7 Formato do arquivo para o Flex. O arquivo consiste de três partes separadas por "%%". A primeira parte contém os includes, variáveis globais e declaração de funções; a segunda parte contém as expressões regulares para identificação dos tokens e a terceira parte contém código C para implementação das funções declaradas na primeira seção.

O formato de cada token que deverá ser reconhecidos pelo analisador é definido usando expressões regulares na segunda seção do arquivo do Flex. Essas expressões seguem o formato da Figura 8 e elas aceitam um código C que é executado toda vez que a expressão é encontrada no arquivo analisado, com isso consegue-se salvar o conteúdo dos tokens usando os macros abaixo [8].

```
%{
#include <string>
#include <stdio.h>

#define TOKEN(t) (yylval.token = t)
#define SAVE_TOKEN yylval.string = new std::string(yytext, yyleng)
%}

D  [0-9]

%%

    /* expressoes */
"int"   {
            SAVE_TOKEN; return TOKEN(INT);
        }
```

Figura 8 Macros e código C para recuperar o valor dos tokens na analise léxica, estes serão enviados posteriormente para análise sintática.

Para tratar caracteres inválidos, usou-se uma regra que contém apenas o caractere ponto final. O ponto nas expressões regulares representa qualquer caractere, por isso se nenhuma das expressões regulares cobrir um caractere, por exemplo, o @, ele será encaixado nessa e uma mensagem de erro será emitida. Outros erros que a analise léxica trata são fim de arquivo não esperado, constantes mal formadas e início de comentários de bloco sem final, sendo que, uma maneira de tratar estes erros está exemplificado na Figura 9.

Para eliminar os comentários, o Flex permite que diferentes modos de execução sejam definidos na primeira parte do arquivo, como é feito para o modo COMMENT. Quando o caractere * é encontrado, o Flex entra no modo COMMENT, e apenas as expressões regulares iniciadas por <COMMENT> são checadas, evitando conflito com regras de outros modos. A regra <COMMENT>. Faz tudo que estiver entre comentários ser ignorado e caso o arquivo termine antes de terminar o comentário com *, um erro é exibido na tela. Se um caractere desconhecido for encontrado fora de um comentário, uma mensagem de erro também é exibida.

```
%{
#define TOKEN(t) (yylval.token = t)
#define SAVE_TOKEN yylval.string = new std::string(yytext, yyleng)
%}

%x COMMENT

%%

    /* comments */
"/*"                    { BEGIN(COMMENT); }
"//".*

<COMMENT>"*/"           { BEGIN(INITIAL); }
<COMMENT><<EOF>>        { printf("ERRO: comentario nao foi fechado"); return 1; }
<COMMENT>{eol}          { }
<COMMENT>.
<<EOF>>                 { yyterminate(); }
.                       { printf("ERRO: caracter nao reconhecido"); }

%%
```

Figura 9 Exemplo de como tratar comentários e caracteres inválidos.

3.3.3 Analise Sintática

Neste passo, utilizou-se a ferramenta Bison [9] [Levine 2009] para gerar o parser da gramática. O Bison é um software escrito por Charles Donnely e Richard Stallman que converte uma descrição de gramática LALR(1) em um parser C para esta gramática. Gramáticas LALR (Look Ahead LR) são uma versão simplificada de uma LR, e assim como estas, elas encaixam os tokens nas regras da esquerda para a direita (L Left) e usam a derivação mais a direita (R – Right). Esse tipo de execução permite que regras da gramatica que apresentem recursão a esquerda sejam reconhecidas pelo parser e não precisam ser alteradas, diferentemente dos parsers top down como o Javacc [10] que não permitem este tipo de recursão. Um exemplo de recursão a esquerda é mostrado na Figura 10.

```
regra1:
        regra1 TOKEN1
    |   TOKEN1
    ;
```

Figura 10 Recursão à esquerda. A regra acima é equivalente a um ou mais TOKEN1 em sequência

A gramatica da linguagem foi convertida para um arquivo de extensão .y, Figura 11, que segue o mesmo padrão do arquivo .l do Flex, sendo dividido em três partes, uma para includes, a segunda para as regras da gramatica e a terceira para código em C. Este arquivo foi processado pelo Bison para gerar um parser em código C.

```
%{
    /* includes */
    #include <stdio.h>
%}

    /*  */
%union {
    int token;
    std::string *string;
}

%token <token> INT

%start regra1

%%

regra1:
        INT {   /* Codigo C */
                printf("achou int");
            }
    ;

%%

int main(int argc, char *argv[]) {
    return 0;
}
```

Figura 11 Formato de arquivo para o Bison. O arquivo é separado em três partes com "%%". A primeira parte contém código C, declarações dos tokens reconhecidos pela analise léxica e a regra inicial da gramática; a segunda parte contém as regras da gramatica e o código C que será executado quando a regra for reconhecida. A ultima seção contém o main e a implementação de funções.

Quando uma sequência de tokens não é reconhecida em nenhuma das regras da gramatica, a função yyerror, se especificada, é chamada, caso contrário uma mensagem de erro padrão é impressa no terminal, um exemplo de como especificar a função yyerror está na Figura 13. Erros da analise sintática incluem abertura de parentes ou colchetes sem fechar com o seu equivalente, sequencias de tokens que não encaixam em nenhuma das regras da gramática, etc.

3.3.4 Analise Semântica

O módulo de análise semântica possui a função de reconhecer erros como: identificadores declarados múltiplas vezes no mesmo escopo, uso de identificadores não declarados. Para que isso seja possível, uma tabela de símbolos é usada, pois ela guarda todas as informações de identificadores, tipos básicos, funções *built-in* da linguagem e assim pode-se fazer tais checagens. Ela foi implementada como um TAD (Tipo Abstrato de Dados) [Sedgewick e Wayne 2011] usando a estrutura de dados vector de C++, com a lista de operações da Figura 12. A estrutura de dados usada na tabela de símbolos, geralmente, é um hashtable, porém optou-se pelo vector para simplificação da implementação já que a eficiência de execução não será prejudicada dado que o processador possui uma memória limitada a 32Kb para dados e instruções, limitando também o número de entradas na tabela de símbolos.

```
// ----- ESCOPO ------
void insereEscopo();

void removeEscopo();

// ----- FUNCOES DE BUSCA -----
Entrada* declarado(string &nome);

Entrada* busca(string nome, bool *declarado);

vector<Entrada> variaveis_escopo();

// ----- INSERCOES NA TABELA -----
Entrada* insere(string id, bool *declarado2, Entrada *entrada);

Entrada* insereFuncao(string id, Entrada *tipo, int nparametros);

Entrada* insereVariavel(string id, Entrada *tipo);

Entrada* insereConstante(string id, Entrada *tipo, string valor);

Entrada* insereParametro(string id, Entrada *tipo);

Entrada* insereTipo(string id, Entrada *te);

// ---------- LABELS ----------
Entrada* insereLabel(string id, bool *declarado, Entrada *f);

bool buscaLabel(string nome, Entrada *funcao);

void labelLog();

// ----- FUNCOES DE DEBUG -----
void log();
```

Figura 12 Lista de operações do Tipo Abstrato de Dados. O TAD prove uma interface para realizar inserções, remoções e buscas na tabela de símbolos para a análise semântica.

Para preencher a tabela de símbolos, adicionou-se código C nas regras da analise sintática, como mostrado na Figura 13. Esses códigos serão executados assim que a regra for reconhecida.

```
%{
    /* includes */
    #include <stdio.h>

    // funcao chamada quando nenhuma das regras puder ser aplicada
    // nos tokens recebidos
    int yyerror(const char *errmsg);

    TAD *t;
%}

%union {
    int token;
    std::string *string;
}

%token <token> INT
%token <string> IDENTIFICADOR

%%

declaracao:
        INT IDENTIFICADOR ';' {
            t->insereVariavel(*$2, int);
        }
    ;

uso:
        IDENTIFICADOR {
            bool declarado = true;
            t->busca(*$1, &declarado);
            if(!declarado) {
                printf("ERRO: identificador %s nao declarado", $1);
            }
        }
    ;

%%

int yyerror(const char *errmsg) {
    // logica para imprimir a mensagem de erro
}

int main(int argc, char *argv[]) {
    ...
}
```

Figura 13 Exemplo de como especificar a função yyerror para customizar as mensagens de erro, e como utilizar as funções da tabela de símbolos para fazer a análise semântica.

É preciso notar que se utilizou $1 para recuperar o valor do identificador. Para que o Bison saiba o tipo deste elemento, deve-se mapear cada um na primeira seção do código do Bison, como mostrado na Figura 13 para o token int e identficador. Desta forma, recupera-se o valor salvo dos tokens, que foram salvos na analise léxica. Quanto às checagens da analise semântica, são adicionados códigos C em algumas das regras da gramatica, exemplificado pelas regras de declaração e uso da Figura 13, estas inserem identificadores na tabela de símbolos e depois checam se eles estão na tabela.

A seguir, apresenta-se um exemplo de tabela de símbolos como está implementada no trabalho. Inicialmente ela contém apenas os tipos de dados e funções *built-in*, Figura 14; ela é preenchida com variáveis globais e funções em uma primeira leitura do código, Figura 15.

Código:

```
int a = 3;

int funcao1(int x, int y) {
   return x + y;
}

int b;

void main() {
   int x = 3;
   int y = 4;
   a = funcao1(x, y);
}
```

Escopo	Nome	id	tipo	endereço	ponteiro	nbytes	tipo_elementar	valor	parametros	função
0	char	tipo	null	-1	0	0	-1	-	null	null
0	int	tipo	null	-1	0	0	-1	-	null	null

Figura 14 Tabela de símbolos antes da leitura do código pelo compilador

Código:

```
int a = 3;

int funcao1(int x, int y) {
   return x + y;
}

int b;

void main() {
   int x = 3;
   int y = 4;
   a = funcao1(x, y);
}
```

Escopo	Nome	id	tipo	endereço	ponteiro	nbytes	tipo_elementar	valor	parametros	função
0	b	variavel	int	27999	0	1	int	-	null	null
0	a	variavel	int	28000	0	1	int	-	null	null
0	main	funcao	int	-1	0	0	-1	-	null	null
0	funcao1	funcao	int	-1	0	0	-1	-	int, int	null
0	char	tipo	null	-1	0	0	-1	-	null	null
0	int	tipo	null	-1	0	0	-1	-	null	null

Figura 15 Tabela de símbolos após a primeira leitura

Na primeira leitura checa-se se todas as funções utilizadas no código foram declaradas previamente, pois na segunda leitura isso já não é possível. A segunda leitura do código é feita para a geração de código e continuação da análise semântica, pois agora a tabela de símbolos já alocou espaço na memória para as variáveis globais e não há perigo de conflito na alocação para variáveis locais. Os espaços para as variáveis globais são preenchidos a medida elas são declaradas no código Figura 16 e 19.

Código:

```
int a = 3;

int funcao1(int x, int y) {
   return x + y;
}

int b;

void main() {
   int x = 3;
   int y = 4;
   a = funcao1(x, y);
}
```

Escopo	Nome	id	tipo	endereço	ponteiro	nbytes	tipo_elementar	valor	parametros	função
0				27999		1				
0	a	variavel	int	28000	0	1	int	-	null	null
0	main	funcao	int	-1	0	0	-1	-	null	null
0	funcao1	funcao	int	-1	0	0	-1	-	int, int	null
0	char	tipo	null	-1	0	0	-1	-	null	null
0	int	tipo	null	-1	0	0	-1	-	null	null

Figura 16 Uma segunda leitura do código é feita pelo compilador.

A partir deste ponto, toda vez que o compilador entra em uma função, if, while ou for seu escopo é incrementado e todas as variáveis declaradas são inseridas nele. Geralmente a delimitação do escopo é marcada pelos caracteres {, }; exceção a esta regra é o escopo da função que começa a partir do caractere (. Figura 17 e 20.

Código:

```
int a = 3;

int funcao1(int x, int y) {
   return x + y;
}

int b;

void main() {
   int x = 3;
   int y = 4;
   a = funcao1(x, y);
}
```

Escopo	Nome	id	tipo	endereço	ponteiro	nbytes	tipo_elementar	valor	parametros	função
1	y	variavel	int	27997	0	1	int	-	null	null
1	x	variavel	int	27998	0	1	int	-	null	null
0				27999		1				
0	a	variavel	int	28000	0	1	int	-	null	null
0	main	funcao	int	-1	0	0	-1	-	null	null
0	funcao1	funcao	int	-1	0	0	-1	-	int(27998), int(27997)	null
0	char	tipo	null	-1	0	0	-1	-	null	null
0	int	tipo	null	-1	0	0	-1	-	null	null

Figura 17 O escopo de uma função começa a partir dos seus parâmetros

Este modo de operação permite que variáveis com o mesmo nome sejam declaradas em escopos diferentes e não haja conflito entre elas quando são referenciadas. Na Figura 18, a variável ‘a’ foi declarada no escopo zero, por isso não há problemas em declarar outra variável ‘a’ dentro de funcao1.

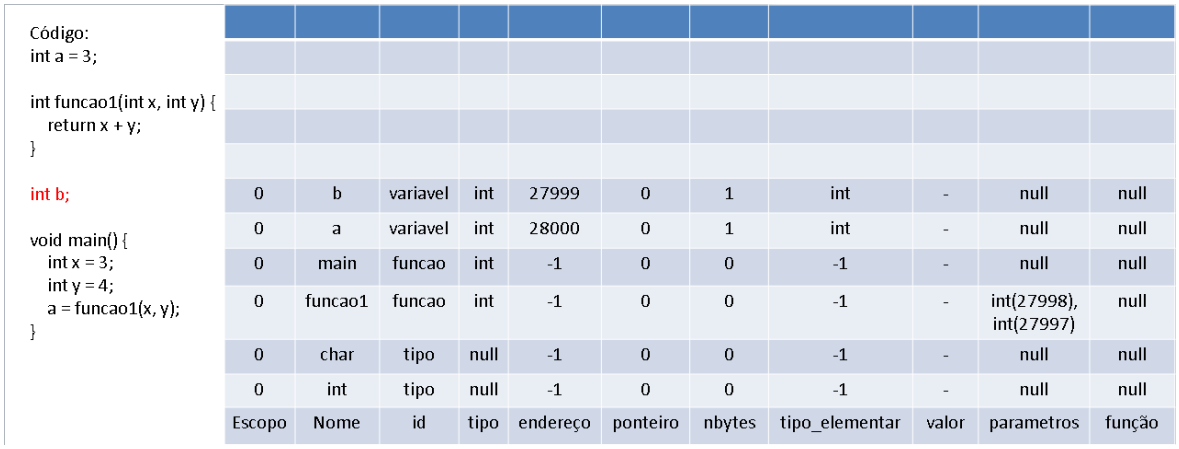
Código:

```
int a = 3;

int funcao1(int x, int y) {
   return x + y;
}

int b;

void main() {
   int x = 3;
   int y = 4;
   a = funcao1(x, y);
}
```

Escopo	Nome	id	tipo	endereço	ponteiro	nbytes	tipo_elementar	valor	parametros	função
1	y	variavel	int	27997	0	1	int	-	null	null
1	x	variavel	int	27998	0	1	int	-	null	null
0				27999		1				
0	a	variavel	int	28000	0	1	int	-	null	null
0	main	funcao	int	-1	0	0	-1	-	null	null
0	funcao1	funcao	int	-1	0	0	-1	-	int(27998), int(27997)	null
0	char	tipo	null	-1	0	0	-1	-	null	null
0	int	tipo	null	-1	0	0	-1	-	null	null

Figura 18 Referências as variáveis não altera a tabela de símbolos

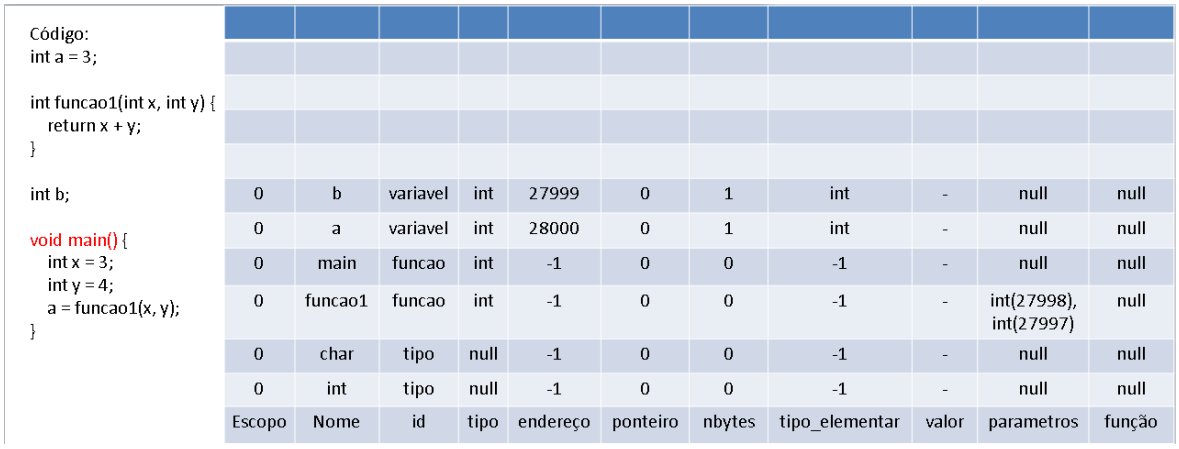
Quando o compilador sai de um escopo, todas as variáveis que estavam nele são removidas da tabela de símbolos, Figuras 19 e 22, evitando conflitos com variáveis declaradas no mesmo nível de escopo. Nas Figuras 18 e 21, por exemplo, as variáveis x e y possuem o mesmo nome e escopo, porém não há conflito, pois o escopo da funcao1 foi removido antes de começar o escopo do main; o possível conflito de endereços 27998 e 27997 para os dois x e dois y é resolvido durante a geração de código.



```
Código:
int a = 3;

int funcao1(int x, int y) {
   return x + y;
}

int b;

void main() {
   int x = 3;
   int y = 4;
   a = funcao1(x, y);
}
```

Escopo	Nome	id	tipo	endereço	ponteiro	nbytes	tipo_elementar	valor	parametros	função
0	b	variavel	int	27999	0	1	int	-	null	null
0	a	variavel	int	28000	0	1	int	-	null	null
0	main	funcao	int	-1	0	0	-1	-	null	null
0	funcao1	funcao	int	-1	0	0	-1	-	int(27998), int(27997)	null
0	char	tipo	null	-1	0	0	-1	-	null	null
0	int	tipo	null	-1	0	0	-1	-	null	null

Figura 19 O espaço na tabela para o 'b' finalmente é preenchido

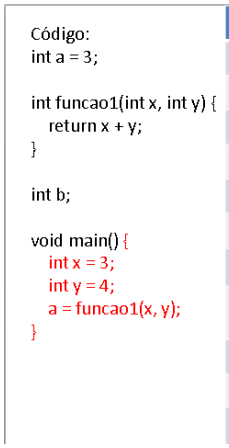


```
Código:
int a = 3;

int funcao1(int x, int y) {
   return x + y;
}

int b;

void main() {
   int x = 3;
   int y = 4;
   a = funcao1(x, y);
}
```

Escopo	Nome	id	tipo	endereço	ponteiro	nbytes	tipo_elementar	valor	parametros	função
0	b	variavel	int	27999	0	1	int	-	null	null
0	a	variavel	int	28000	0	1	int	-	null	null
0	main	funcao	int	-1	0	0	-1	-	null	null
0	funcao1	funcao	int	-1	0	0	-1	-	int(27998), int(27997)	null
0	char	tipo	null	-1	0	0	-1	-	null	null
0	int	tipo	null	-1	0	0	-1	-	null	null

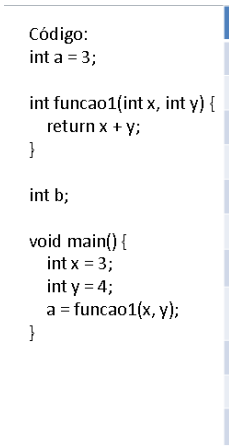
Figura 20 main não possui parâmetros, então nada é acrescentado a tabela de símbolos

Em um dado escopo é possível usar todas as variáveis declaradas em escopos menores do que ele. Na Figura 21, a variável global ‘a’ pode ser referenciada, pois está em um escopo menor do que o escopo da função main.



Escopo	Nome	id	tipo	endereço	ponteiro	nbytes	tipo_elementar	valor	parametros	função
1	y	variavel	int	27997	0	1	int	-	null	null
1	x	variavel	int	27998	0	1	int	-	null	null
0	b	variavel	int	27999	0	1	int	-	null	null
0	a	variavel	int	28000	0	1	int	-	null	null
0	main	funcao	int	-1	0	0	-1	-	null	null
0	funcao1	funcao	int	-1	0	0	-1	-	int(27998), int(27997)	null
0	char	tipo	null	-1	0	0	-1	-	null	null
0	int	tipo	null	-1	0	0	-1	-	null	null

Figura 21 Referências a variável 'a' e a função 'funcao1' são permitidas, pois foram declaradas previamente (escopo menor do que o atual)



Escopo	Nome	id	tipo	endereço	ponteiro	nbytes	tipo_elementar	valor	parametros	função
0	b	variavel	int	27999	0	1	int	-	null	null
0	a	variavel	int	28000	0	1	int	-	null	null
0	main	funcao	int	-1	0	0	-1	-	null	null
0	funcao1	funcao	int	-1	0	0	-1	-	int(27998), int(27997)	null
0	char	tipo	null	-1	0	0	-1	-	null	null
0	int	tipo	null	-1	0	0	-1	-	null	null

Figura 22 Tabela de símbolos após a leitura

3.3.5 Geração de Código

O ultimo passo do trabalho foi a geração de código para a linguagem Assembly do processador. A geração é feita através da tradução das regras da gramatica para o equivalente em Assembly, modificando o código C da análise semântica para que ele também gere código, como mostrado na Figura 23. A referência de todas as instruções aceitas pelo processador do ICMC está no Apêndice B.

```
%%

id:
        IDENTIFICADOR {
            // checagem da analise semantica
            printf("load r0, %d", endereco_do_identificador);
            printf("store %d, r0", stack);
            stack--;
        }
    ;

adicao:
        id '+' id {
            printf("load r0, %d\n", stack+2);
            printf("load r1, %d\n", stack+1);
            printf("add r2, r0, r1\n");
            printf("store %d, r2", stack+2);
            stack++;
        }
    ;

%%
```

Figura 23 Geração de Código através das regras da gramática. O endereço do identificador é obtido da tabela de símbolos.

A memória do processador foi mapeada como uma pilha para resolver expressões aritméticas compostas de vários operandos e operadores diferentes. Deste modo, quando o parser recebe um operando do analisador léxico, ele já o insere na pilha e quando uma regra de operação aritmética é completada, os dois últimos elementos do topo da pilha são removidos e o resultado da operação é colocado de volta no topo da pilha; esse ciclo continua até que a expressão inteira seja resolvida. Exemplo abaixo, na Figura 24:

Expressão avaliada:
X = 3;
Y = 7;
...
Z = 1 + (Y – X) - 2;

Endereço	Conteúdo
27994	
27995	
27996	
27997	
27998	
27999	
28000	

Expressão avaliada:
X = 3;
Y = 7;
...
Z = 1 + (Y – X) - 2;

Endereço	Conteúdo
27994	
27995	
27996	
27997	
27998	
27999	
28000	1

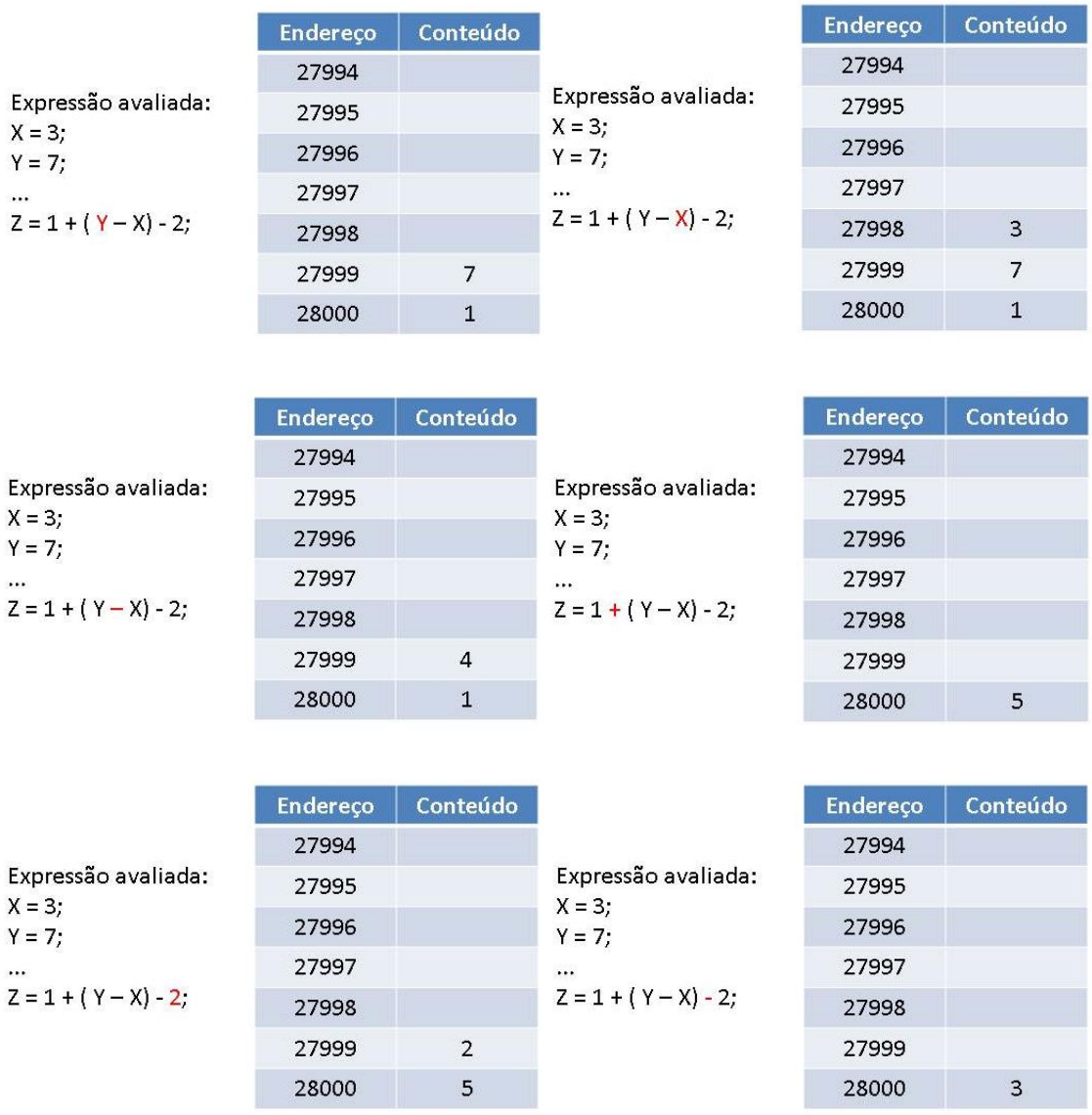


Figura 24 Operação da Pilha (*Stack*) durante avaliação de expressões

Para resolver os conflitos de endereço de memoria da tabela de símbolos apresentados na seção da analise semântica e também suportar recursão, utilizou-se o esquema das Figuras 25 a 28. Continuando o exemplo da tabela de símbolos, a Figura 25 mostra o estado da memória antes que funcao1 seja executada.

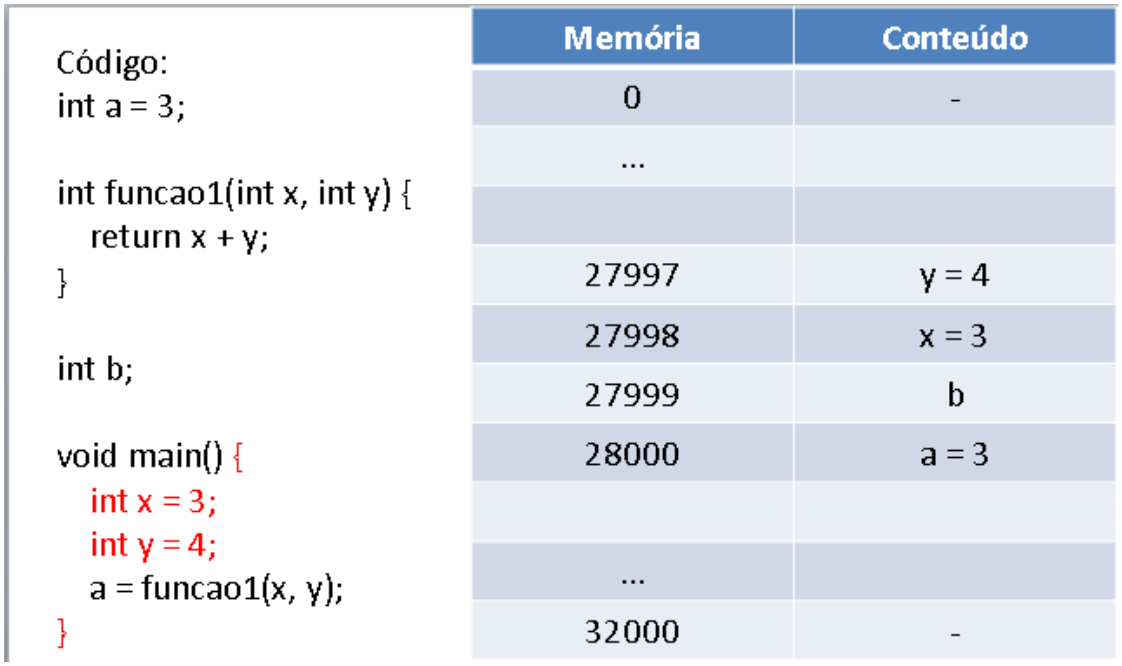


Figura 25 Memória antes da chama da funcao1

Quando funcao1 vai ser chamada no main, todas as variáveis locais do main são empilhadas usando a instrução “push” do processador, e em seguida os parâmetros da função também são empilhados da mesma forma, Figura 26; os conteúdos de memória de 28000 a 27997 não são alterados.

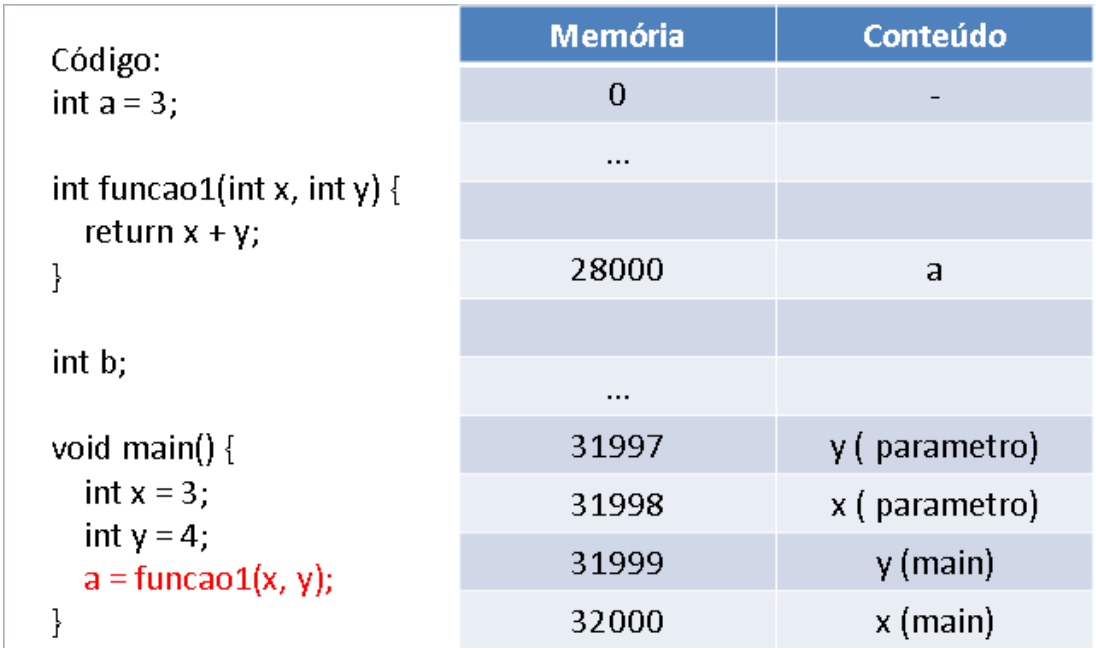
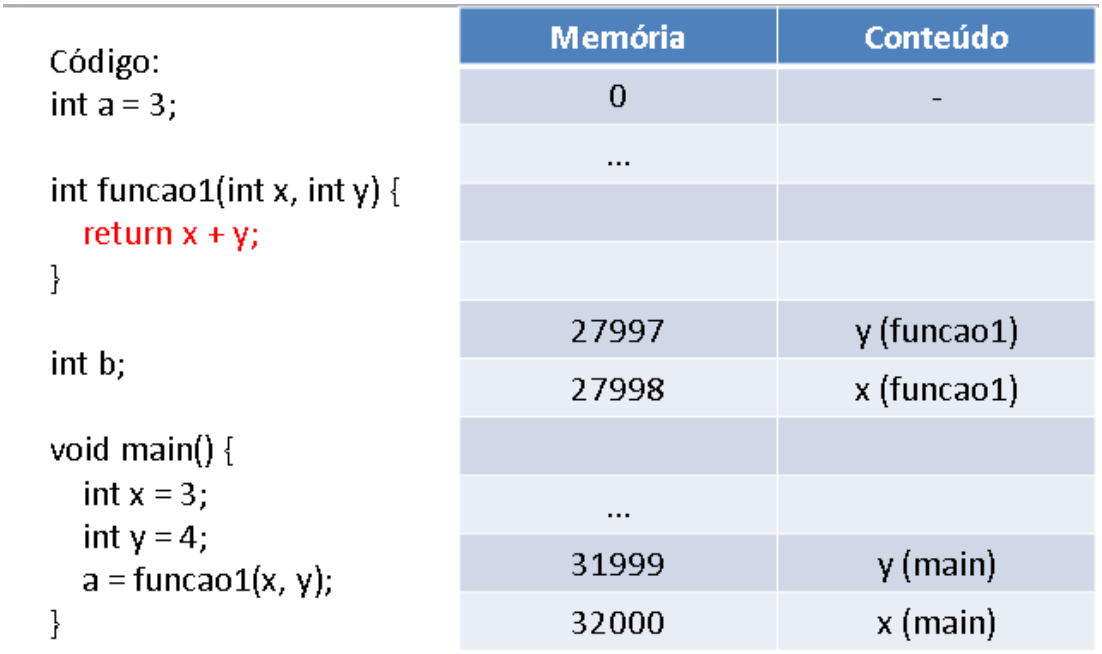


Figura 26 Memória na chamada da funcao1

Antes de executar o código da funcao1, os parâmetros da função são recuperados com a sequência de instrução "pop", "store" do processador, sobrescrevendo os conteúdos de 27998 e 27997, Figura 27. O compilador sabe quantos parâmetros e os endereços destes através da lista de parâmetros que a tabela de símbolos tem para cada função.

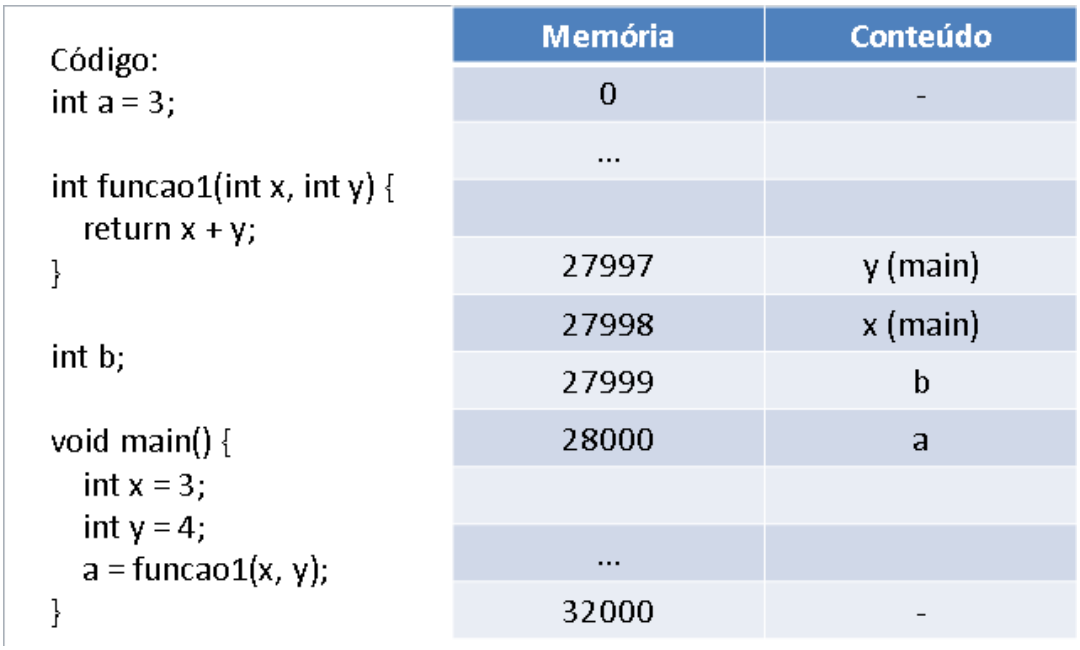


```
Código:
int a = 3;

int funcao1(int x, int y) {
   return x + y;
}

int b;

void main() {
   int x = 3;
   int y = 4;
   a = funcao1(x, y);
}
```

Memória	Conteúdo
0	-
...	
27997	y (funcao1)
27998	x (funcao1)
...	
31999	y (main)
32000	x (main)

Figura 27 Memória dentro da funcao1

Depois de executada a funcao1, o main recupera as suas variáveis locais com a sequência de instruções “pop”, “store”, retornando ao estado que estava antes da funca1. O compilador sabe o numero de variáveis locais e seus endereços através do escopo da tabela de símbolos. Figura 28.



```
Código:
int a = 3;

int funcao1(int x, int y) {
   return x + y;
}

int b;

void main() {
   int x = 3;
   int y = 4;
   a = funcao1(x, y);
}
```

Memória	Conteúdo
0	-
...	
27997	y (main)
27998	x (main)
27999	b
28000	a
...	
32000	-

Figura 28 Memória retornando para o main

As mudanças mais significativas em relação ao parser da análise sintática com o parser da geração de código são: as alterações nas regras da gramática como a regra para o if, for, while. Estas regras requerem a inserção de novas regras na gramática para que o código C nas regras dos terminais saibam se eles estão dentro de um if ou de um for, pois a gramatica livre de contexto não suporta este conhecimento. Na Figura 29 foi adicionada uma nova regra chamada begin_if para inserir o código de comparação do jump, pois a regra statement não consegue fazer este papel já que ele não sabe que está dentro de um if.

```
%%

expression:
    ... {
        // resolve a expressao e deixa o resultado no topo
        // da pilha na memoria
    }
    ;

statement:
    ... {
        // gera o Assembly do conteudo do if
    }
    ;

selection_statement
    : IF ABRE_PARENTESES expression begin_if statement {
        printf("__exit_if_%d__:\n", if_stack.top());
        if_stack.pop();
    }
    ;

begin_if:
        FECHA_PARENTESES {
        ifseq++;
        if_stack.push(ifseq);

        // carrega o resultado de expression em r0
        printf("load r0, %d", stack);
        // r1 recebe 0
        printf("loadn r1, #0");
        // compara r0 com 0 (falso)
        printf("cmp r0, r1");
        // se for falso pula o conteudo do if
        printf("jeq __exit_if_%d__\n", if_stack.top());
    }
    ;

%%
```

Figura 29 Mudança na gramática para o if

O funcionamento do parser também foi alterado para que ele leia o código duas vezes, essa forma de operação permite que funções, variáveis globais e labels sejam identificadas e alocadas na memória antes do resto do código, sem isso não seria possível usar “goto” para labels que apareçam após ele.

3.4. Resultados Obtidos

Um jogo árcade do tipo Pacman, Figura 30, foi desenvolvido para testar o código gerado pelo compilador. Além disso, o compilador juntamente com o simulador e o montador foi integrado aos editores de texto Gedit e Notepad++, permitindo que os usuários pressionem a tecla F6 para compilar os seus códigos C, e rodar o código de máquina gerado no Simulador para depuração. O Apêndice C explica como configurar esses editores de texto para esta funcionalidade.

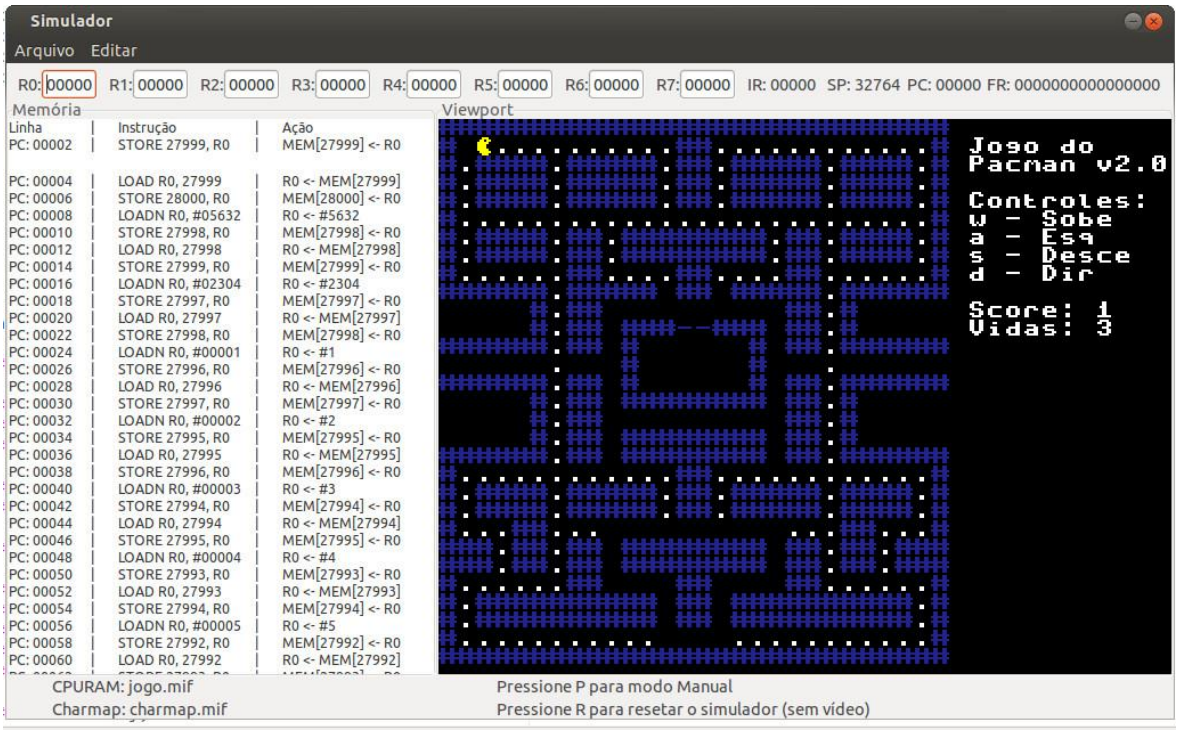


Figura 30 Simulador rodando o jogo Pacman compilado para Assembly pelo compilador.

O Assembly gerado também possui comentários para ajudar o usuário saber o que cada instrução representa.

3.5. Dificuldades, Limitações e Trabalhos Futuros

O desenvolvimento do front end de compiladores para as fases de análise léxica, gramatical e semântica já é uma questão tratada exaustivamente na literatura. Porém, a geração de código intermediário e a otimização deste para código de maquina ainda são passos que são feitos a partir da experiência dos desenvolvedores, pois não existe um compilador que sempre gera o melhor código para qualquer entrada [Aho e Ullman 2007]. Por isso, essas etapas foram as mais difíceis durante este trabalho.

A linguagem desenvolvida foi baseada na linguagem C, mas ela não oferece suporte a todas as funcionalidades desta, devido ao tempo do projeto e também as limitações do processador. Em relação às limitações da linguagem, pode-se citar linkagem com outros arquivos via include, uso de macros via define, limitação das matrizes para no máximo duas dimensões, não suporte a estruturas como switch/case, do while, structs, union, enum, e tipos de dados como float, double.

Para trabalhos futuros, pode-se sugerir o acréscimo de novas funcionalidades na linguagem e no processador, como a adição de includes e aumento da memória disponível, adição de uma unidade aritmética com suporte a ponto flutuante e tipo de dados float e double, afim de superar as limitações descritas acima.

Agora que temos o compilador, é possível também o desenvolvimento de um sistema operacional - SO, desta forma o kit didático abordaria deste os princípios de funcionamento de um processador, as etapas que o código em linguagem de alto nível percorre até ser executado pela CPU e, por fim, o funcionamento de um SO. Este SO poderia oferecer acesso a periféricos como dispositivos de áudio, comunicação serial com um PC, acesso a memórias externas como pendrive, SD cards, etc.

3.6. Considerações Finais

Neste capitulo, abordou-se as etapas das analises léxica, sintática e semântica e a geração de código, mostrando exemplos de como cada uma foi implementada e como cada

uma se relaciona com as outras. Também se mostrou os resultados obtidos com este trabalho utilizando o jogo Pacman e as dificuldades encontradas durante o desenvolvimento do compilador.

CAPÍTULO 4: CONCLUSÃO

Este trabalho exigiu conhecimentos específicos sobre linguagens de programação, desde as formas de representação das estruturas da linguagem na memória, de como fazer o reconhecimento e checagem do código através de máquinas de estados, e das etapas e estruturas de dados necessárias de todo o processo de compilação. Esses conhecimentos foram adquiridos durante as disciplinas de Linguagens Formais e Compiladores no ICMC e Programming Languagens durante intercâmbio na universidade Sogang University, Coréia do Sul.

Além disso, também foram necessários conhecimentos de organização e arquiteturas de computadores para a geração de código, já que sem esses conhecimentos não é possível utilizar todos os recursos disponíveis no hardware, otimizar o código, e checar a qualidade do código Assembly gerado. Neste caso, as disciplinas de Lógica Digital 1 e 2, no ICMC, e Embedded Computer Architecture, durante estágio no exterior, foram importantes durante o aprendizado.

O ICMC oferece disciplinas relacionadas com o projeto, tais como Arquitetura de Computadores e Organização de Computadores Digitais 1 e 2, porém estas disciplinas foram puramente teóricas e os conteúdos ensinados não puderam ser aproveitados. As disciplinas realizadas no exterior foram citadas, pois forneceram conhecimentos que serviram de base durante o projeto, e apesar de serem teóricas, tiveram seus conteúdos fortemente relacionados com projetos reais. Espera-se ter contribuído com este trabalho para que as aulas práticas de laboratório de organização de computadores, agora disponíveis para os novos alunos do ICMC possam trazer oportunidades de vivenciar soluções reais implementadas pelos próprios alunos.

A habilidade mais importante durante o projeto foi a capacidade de implementar em software conceitos e ideias apresentados em sala de aula. Como a maior parte dos conceitos foram vistos apenas na teoria, a realização deste projeto proporcionou a experiência de como extrair a essência destes conceitos e implementá-los na prática.

4.1. Contribuições

Este trabalho resultou em um compilador que complementará um kit didático que pode ser usado nas disciplinas de lógica digital, organização de computadores, compiladores e seus laboratórios. Este kit aborda desde a implementação de um processador, programação em baixo nível e, agora, programação em alto nível, relacionado esta com os itens anteriores.

O compilador é de código fonte aberto, disponível em [11], utilizando uma estruturação em módulos ensinada nos cursos de compiladores e livros da área. Estes recursos juntamente com este documento que descreve todos os passos e as decisões tomadas durante a implementação permitem que o compilador seja incrementado de novas funcionalidades pelos próprios alunos que utilizarem este software.

Como profissional este projeto, assim como os anteriores, permitiu trabalhar em softwares que serão usados por outros alunos, melhorando a formação destes. Também melhorou a minha formação, colocando em pratica os conceitos aprendidos em sala de aula e incrementando meu currículo profissional.

4.2. Considerações sobre o Curso de Graduação

Os pontos fortes do curso de Bacharelado em Ciências da Computação foram as aulas de Lógica Digital com os trabalhos práticos em FPGA, compiladores com o trabalho prático utilizando linguagens reais, Programação Orientada a Objetos juntamente com Padrões de Projeto Web que ensinaram as linguagens e como utilizar o poder delas com padrões, e Algoritmos, também com a implementação do conteúdo da aula teórica.

Os pontos fracos foram as aulas essenciais para um bacharel em Ciências da Computação que não tiveram seus conteúdos passados de maneira satisfatória. Exemplos são Redes de Computadores, Redes de Alto Desempenho, Sistemas Operacionais 1 e 2, Organização de Computadores; estas disciplinas em geral são aulas teóricas longas, a duração da aula só prejudica pois não se dá ênfase em conceitos importantes e portanto fica

difícil o aprendizado destes pois requer atenção durante toda a aula e ainda sorte em conseguir classificar a importância de cada assunto já que se soubéssemos não estaríamos na aula.

Sugestões para resolver os problemas acima são: no caso de disciplinas de redes, ter um laboratório onde a teoria pode ser aplicada, o Senac da cursos de 60 horas da Cisco e durante este curso aprendi mais do que Redes e Redes de Alto Despenho juntas pois como o tempo é curto e curso visa formar profissionais, apenas o conteúdo importante é dado com prova ou prática no raque na aula seguinte; vale a pena conversar com o professor do curso, Edelber, pois ele trabalha aqui na USP. No exterior as aulas também tinham duração menor e por isso os professores colocavam mais ênfase no conteúdo importante já que não dava tempo para dar toda a matéria.

REFERÊNCIAS

[Aho e Ullman 2007] AHO, A. V. ; LAM, M. S. ; SETHI, R. ULMAN, J. D. Compilers: Principles, Techniques and Tools

[Brown 1992] BROWN, S. D.; Field-Programmable Gate Arrays; Kluwer Academic Publishers, 1992.

[Brown 2005] BROWN, S.; VRANESIC, Z. Fundamentals of Digital Logic with VHDL Design, McGraw Hill, 2005

[Levine 2009] LEVINE, J. Flex & Bison: Text Processing Tools O`Reilly Media, 2009

[Kamada e Simões 2012] KAMADA M. K. ; SIMOES E. V. Um processador RISC para ensino de arquitetura de computadores, SIICUSP, 2012

[Kamada e Simões 2012] KAMADA M. K. ; SIMOES E. V. Um simulador didático de um processador RISC para ensino de logica digital, WICT, 2012

[Moreira, Martins e Simões] MOREIRA, R. P. G. ; MARTINS G. A. A. ; SIMOES E. V. Um Montador Assembly para Processador Didático, SIICUSP

[Sedgewick e Wayne 2011] SEDGEWICK, R. ; WAYNE, K. Algorithms Addison-Wesley Professional, 2011

[Tanenbaum e Woodhull 2006] Tanenbaum, A. S. ; Woodhull A. S. Operating Systems Design and Implementation Prentice Hall, 2006

[1] http://www.hardware.com.br/artigos/risc-cisc/, acessado em 10/11/2014

[2] http://en.wikipedia.org/wiki/Context-free_grammar, acessado em 10/11/2014

[3] http://www.altera.com/products/software/quartus-ii/web-edition/qts-we-index.html, acessado em 10/11/2014

[4] http://www.terasic.com.tw/cgi-bin/page/archive.pl?No=226, acessado em 10/11/2014

[5] http://en.wikipedia.org/wiki/Von_Neumann_architecture, acessado em 10/11/2014

[6] http://www.quut.com/c/ANSI-C-grammar-y.html, acessado em 10/11/2014

[7] http://flex.sourceforge.net/, acessado em 10/11/2014

[8] http://gnuu.org/2009/09/18/writing-your-own-toy-compiler/, acessado em 10/11/2014

[9] http://www.gnu.org/software/bison/, acessado em 10/11/2014

[10] https://javacc.java.net/, acessado em 10/11/2014

[11] https://github.com/marcelokk/compilador, acessado em 10/11/2014

APÊNDICE A – Gramática Utilizada

Palavras em letras maiúsculas são tokens reconhecidos pelo analisador léxico, e as palavras em letras minúsculas são regras da gramática.

Símbolo inicial da gramática = translation_unit

```
primary_expression
   : IDENTIFIER
   | getch
   | CONSTANT
   | STRING_LITERAL
   | ( expression )
   | IDENTIFIER ( )
   | IDENTIFIER ( argument_expression_list )
   | IDENTIFIER [ expression ]
   | IDENTIFIER [ expression ] [ expression ]
   ;

postfix_expression
   : primary_expression
   | postfix_expression PONTO IDENTIFIER
   | postfix_expression ++
   | postfix_expression --
   | ( type_name ) { initializer_list }
   ;

argument_expression_list
   : assignment_expression
   | argument_expression_list , assignment_expression
   ;

unary_expression
   : postfix_expression
   | ++ unary_expression
   | -- unary_expression
   | unary_operator cast_expression
   ;

unary_operator
   : &
   | *
   | +
   | -
```

```
    | ~
    | !
    ;

cast_expression
    : unary_expression
    | ( type_name ) cast_expression
    ;

multiplicative_expression
    : cast_expression
    | multiplicative_expression * cast_expression
    | multiplicative_expression / cast_expression
    | multiplicative_expression % cast_expression
    ;

additive_expression
    : multiplicative_expression
    | additive_expression + multiplicative_expression
    | additive_expression - multiplicative_expression
    ;

shift_expression
    : additive_expression
    | shift_expression << additive_expression
    | shift_expression >> additive_expression
    ;

relational_expression
    : shift_expression
    | relational_expression MENOR shift_expression
    | relational_expression MAIOR shift_expression
    | relational_expression <= shift_expression
    | relational_expression >= shift_expression
    ;

equality_expression
    : relational_expression
    | equality_expression == relational_expression
    | equality_expression != relational_expression
    ;

and_expression
    : equality_expression
    | and_expression & equality_expression
    ;
```

```
exclusive_or_expression
    : and_expression
    | exclusive_or_expression ^ and_expression
    ;

inclusive_or_expression
    : exclusive_or_expression
    | inclusive_or_expression | exclusive_or_expression
    ;

logical_and_expression
    : inclusive_or_expression
    | logical_and_expression && inclusive_or_expression
    ;

logical_or_expression
    : logical_and_expression
    | logical_or_expression || logical_and_expression
    ;

conditional_expression
    : logical_or_expression
    | logical_or_expression ? expression : conditional_expression
    ;

assignment_expression
    : conditional_expression
    | unary_expression assignment_operator assignment_expression
    ;

assignment_operator
    : =
    | *=
    | /=
    | %=
    | +=
    | -=
    | <<=
    | >>=
    | &=
    | ^=
    | |=
    ;

expression
    : assignment_expression
    | expression , assignment_expression
```

```
    ;

constant_expression
    : conditional_expression
    ;

declaration
    : declaration_specifiers ;
    | declaration_specifiers init_declarator_list ;
    ;

declaration_specifiers
    :  type_specifier
    | type_specifier declaration_specifiers
    | function_specifier
    | function_specifier declaration_specifiers
    ;

init_declarator_list
    : init_declarator
    | init_declarator_list , init_declarator
    ;

init_declarator
    : declarator
    | declarator = initializer
    ;

type_specifier
    : VOID
    | CHAR
    | INT
    ;

specifier_qualifier_list
    : type_specifier specifier_qualifier_list
    | type_specifier
    ;

declarator
    : pointer direct_declarator
    | direct_declarator
    ;

direct_declarator
    : IDENTIFIER
    | IDENTIFIER [ CONSTANT ]
```

```
    | IDENTIFIER [ CONSTANT ] [ CONSTANT ]
    ;

pointer
    : *
    | * pointer
    ;


type_name
    : specifier_qualifier_list
    | specifier_qualifier_list abstract_declarator
    ;


abstract_declarator
    : pointer
    | direct_abstract_declarator
    | pointer direct_abstract_declarator
    ;


direct_abstract_declarator
    : ( abstract_declarator )
    | [ ]
    | [ assignment_expression ]
    | direct_abstract_declarator [ ]
    | direct_abstract_declarator [ assignment_expression ]
    | [ * ]
    | direct_abstract_declarator [ * ]
    | ( )
    ;


initializer
    : assignment_expression
    | { initializer_list }
    ;
```

```
initializer_list
    : initializer
    | designation initializer
    | initializer_list , initializer
    | initializer_list , designation initializer
    ;

designation
    : designator_list =
    ;

designator_list
    : designator
    | designator_list designator
    ;

designator
    : [ constant_expression ]
    | PONTO IDENTIFIER
    ;

statement:
       labeled_statement
    | compound_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    | breakp_statement
    | printf_statement
    ;
```

```
breakp_statement:
        BREAKP ;
    ;

getch:
        GETCH ( )

opt_args:
    |   , assignment_expression
    |   opt_args , assignment_expression
    ;

printf_statement:
        PRINTF ( assignment_expression , assignment_expression , STRING_LITERAL
opt_args ) ;
    ;

labeled_statement
    : IDENTIFIER : statement
    | CASE constant_expression : statement
    | DEFAULT : statement
    ;

compound_statement
    : { }
    | { block_item_list }
    ;

block_item_list
    : block_item
    | block_item_list block_item
    ;
```

```
block_item
    : declaration
    | statement
    ;


expression_statement
    : ;
    | expression ;
    ;

selection_statement
    : IF ( expression ) statement
    | IF ( expression ) statement ELSE statement
    | SWITCH ( expression ) statement
    ;

iteration_statement
    : WHILE ( expression ) statement
    | DO statement WHILE ( expression ) ;
    | FOR ( expression_statement expression_statement ) statement
    | FOR ( expression_statement expression_statement expression ) statement
    ;

jump_statement
    : GOTO IDENTIFIER ;
    | CONTINUE ;
    | BREAK ;
    | RETURN ;
    | RETURN expression ;
    ;

translation_unit
    : external_declaration
    | translation_unit external_declaration
    ;

external_declaration
    : function_definition
    | declaration
    ;

function_definition
    :   declaration_specifiers IDENTIFIER ( opt_parametros ) compound_statement
    ;
```

```
opt_parametros:
    | parameter_list
    ;

parameter_list
    : parameter_declaration
    | parameter_list , parameter_declaration
    ;

parameter_declaration
    : type_specifier pointer direct_declarator
    | type_specifier direct_declarator
    ;
```

APÊNDICE B – Referência da Linguagem Montadora do Processador do ICMC

Instruções de manipulação de dados

Estas instruções movem dados entre memória e registradores ou entre os registradores. As instruções load e store são limitadas a endereços até 32000 que é o tamanho da memória do processador.

Nome	Instrução	Descrição
Load	load rx, endereço	Rx <- mem[endereço]
Loadn	loadn rx, valor	Rx <- valor
Loadi	loadi rx, ry	Rx <- mem[ry]
Store	store endereço, rx	Mem[endereço] <- rx
Storei	storei rx, ry	Mem[rx] <- ry
Move	mov rx, ry	Rx <- ry
Push	push rx	Stack <-rx
Pop	pop rx	Rx <- stack

Instruções de Entrada e Saída

A instrução outchar escreve o conteúdo de rx na posição ry da tela; a instrução inchar recebe um tecla do teclado e guarda o valor em rx.

Nome	Instrução	Descrição
Outchar	outchar rx, ry	Vídeo[ry] <- rx
Inchar	inchar rx	Rx <- teclado

Instruções Aritméticas

Estas instruções fazem operações aritméticas entre os registradores, para salvar o resultado, utilize a instrução store da tabela de manipulação de dados.

Nome	Instrução	Descrição
Adição	add rx, ry, rz	Rx <- ry + rz
Subtração	sub rx, ry, rz	Rx <- ry - rz
Multiplicação	mul rx, ry, rz	Rx <- ry * rz
Divisão	div rx, ry, rz	Rx <- ry / rz
Incrementar	inc rx	Rx++
Decrementar	dec ry	Rx--
Módulo	mod rx, ry, rz	Rx <- ry % rz
Shift	shift	

Instruções Lógicas

Instruções para comparação e operações lógicas entre os registradores.

Nome	Instrução	Descrição
And	and rx, ry, rz	Rx <- ry and rz
Or	or rx, ry, rz	Rx <- ry or rz
Xor	xor rx, ry, rz	Rx <- ry xor rz
Negação	sot rx	Rx <- not rx
comparação	cmp rx, ry	Compara rx e ry, setando as flags de maior, menor, igual

Instruções de desvio de execução

Desviam a execução do programa. As instruções de jmp e call possuem variações da tabela variações que podem ser usadas em conjunto com a instrução cmp da tabela de instruções lógicas.

Nome	Instrução	Descrição
Jump	jmp endereço	Pula para endereço
Call	call endereço	Pula para endereço e seta rts
Return	rts	Retorna para o endereço do call

Variações do jump e call

Jump	Call	Descrição

jgr	cgr	Se maior
jeg	ceg	Se maior ou igual
jle	cle	Se menor
jel	cel	Se menor ou igual
jeq	ceq	Se igual
jne	cne	Se diferente
jz	cz	Se zero
jnz	cnz	Se diferente de zero
jc	cc	Se carry
jnc	cnc	Se não carry
jo	co	Se deu overflow
jno	cno	Se não deu overflow
jdz	cdz	Se divisão por zero
jn	cn	Se resultado negativo

Instruções de controle

As funções breakp e halt alteram o modo de execução do processador/simulador, já o nop e set possuem funções especificas.

Nome	**Instrução**	**Descrição**

No Operation	nop	Não faz nada
Halt	halt	Para a execução
Setc	setc	Seta o bit de carry do processador
Break Point	breakp	Muda a execução para passo a passo

APÊNDICE C – Setup dos editores Gedit e Notepad++ para usar o compilador

1 Instale o simulador, o manual de instalação está disponível em https://github.com/marcelokk/compilador

2. Baixe e instale os programas Flex e Bison disponíveis em http://sourceforge.net/projects/flex/files/ e http://www.gnu.org/software/bison/ , colocando-os no PATH do seu sistema operacional

3 Baixe o código fonte do compilador que também está em https://github.com/marcelokk/compilador

4 Navegue com o terminal até a pasta do compilador e utilize o comando "make" para gerar o executável do compilador.

5. Coloque o executável gerado e o simulador na pasta do seu projeto

6 Siga as instruções para configurar o seu editor de texto

Configurando o notepad++ como uma IDE

1- Faça o download do notepad++ em http://notepad-plus-plus.org/ e em seguida instale-o

2- No notepad++, instale o plugin NppExec:

2.1- Na barra de ferramentas: Plugins->Plugin Manager->Show Plugin Manager

2.2- Na lista que apareceu, procure por NppExec.

2.3- Marque-o para instalação, clicando no seu quadrado a esquerda.

2.4- Clique em Install e aguarde a instalação.

2.5- Se tudo der certo uma aba do NppExec deve aparecer em Plugins.

3- Nas opções do NppExec selecione Executar e copie e cole o seguinte script:

cmd /c cd $(CURRENT DIRECTORY) && cmd /c parser $(NAME PART).c > $(NAME PART).asm && cmd /c Montador $(NAME PART).asm $(NAME PART).mif && simulador $(NAME PART).mif charmap.mif

4- Salve com o nome que desejar, denindo a hotkey para executar o script

Configurando o gedit como uma IDE

1- Caso você não possua o gedit faça o seu download e instale-o. http://projects.gnome.org/gedit/

2- Em Editar -> preferências -> Plugins, habilite o seguinte plugin: External Tools (vem por padrão)

3- Em Ferramentas -> Gerenciar Ferramentas Externas, adicione um novo comando clicando no botão de '+', no canto inferior da janela

4- Com o novo comando selecionado, nomeie-o como desejar e copie e cole o seguinte script:

```
#!/bin/bash

cur_dir=$GEDIT_CURRENT_DOCUMENT_DIR

arquivo=$(echo $GEDIT_CURRENT_DOCUMENT_NAME | sed 's/\(.*\)\..*/\1/')

echo "Compilando: ${GEDIT_CURRENT_DOCUMENT_NAME}"

$cur_dir/parser $GEDIT_CURRENT_DOCUMENT_NAME > ${arquivo}.asm

$cur_dir/montador ${arquivo}.asm ${arquivo}.mif && $cur_dir/sim ${arquivo}.mif charmap.mif
```