|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | |  |
| МИНОБРНАУКИ РОССИИ | | | | | | |  |
| Федеральное государственное бюджетное образовательное учреждение  высшего профессионального образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | | | | | |  |
|  | Институт информационных технологий (ИТ) | | | | | | |
|  | Кафедра инструментального и прикладного программного обеспечения (ИППО) | | | | | | |
| **Отчет по лабораторной работе №3**  По дисциплине «Разработка ПАОИ и АС»  Тема: «Разработка ассемблера для модели процессора» | | | | |
|  | | | | |
|  | | | | |
| Выполнил студент группы ИКМО-01-19 | | | Косиков М.И. | |
| Преподаватель | | | Тарасов И.Е. | |
|  | |  | |  | |
|  | |  | |

Москва 2019

**Цель работы:** разработка ассемблера для модели процессора на языке программирования высокого уровня.

**Введение**

Инструментальное программное обеспечение является важнейшей составной частью проекта процессорной системы. Ручное формирование машинных кодов крайне непродуктивно и его невозможно использовать даже для отладки сколько-нибудь объемных программ. Поэтому для разработки процессора необходимо иметь хотя бы базовые инструменты для создания кода, пусть даже и с ограниченными возможностями.

Согласно определению, в ГОСТ 19781-90, «кросс-система программирования – система программирования, программные компоненты которой порождают программы на машинном языке, отличном от того, в среде которого они работают».

Под кросс-компиляцией (cross-compilation) понимается процесс перевода программы в машинный код, который должен исполняться на процессоре с архитектурой, отличной от той, на которой запускается сам компилятор. Иными словами, речь идет о том, чтобы на ПК с процессором x86 была запущена программа, преобразующая некий исходный текст в машинные коды для процессора с другой архитектурой. Созданный машинный код может быть использован как для загрузки в макет процессора в ПЛИС, так и для проведения с его помощью моделирования.

Хорошо известным трудом в области разработки компиляторов является т.н. «Книга дракона» («Dragon book») [1]. Ее настоящее название «Компиляторы: принципы, технологии и инструментарий», а «книга дракона» вошла в обиход из-за оформления обложки, на которой процесс разработки компилятора представлен в виде борьбы рыцаря-программиста с драконом-компилятором. Книга при ее большом объеме описывает различные аспекты разработки компиляторов, не все из которых обязательны для построения практического продукта. В ней дается общий маршрут работы компилятора, который представлен на рис. 1.

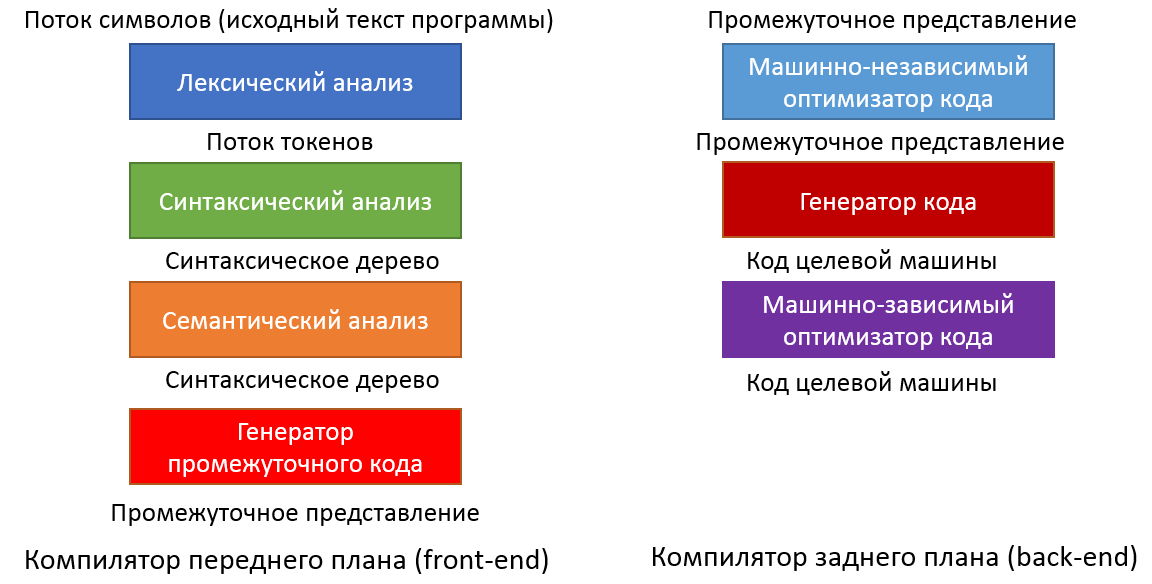


Рис. 1. Стадии компиляции.

Рассмотрим вкратце шаги, которые могли бы помочь разработать простой компилятор, переводящий исходный текст программы в последовательность машинных кодов. Исходя из рис. 1, первым шагом компиляции является лексический анализ, переводящий исходный текст программы в последовательность *токенов*. Под токенами понимаются пары «имя – значение», которые строятся для каждого найденного элемента программы. Например, ассемблерная команда mov r0, r1 будет разобрана на элементы «mov» «r0» «r1».

Синтаксический анализ имеет целью построить *промежуточное представление* программы. Существуют различные способы представление порядка операций программы, например, в виде *синтаксического дерева*. Однако такое представление может быть несколько избыточно для простого ассемблера.

На стадии семантического анализа проверяется смысловая корректность построенного промежуточного представления. Например, строка x = y + z является синтаксически корректной, однако если x, y – это целые числа, а z – строка, то в представленном виде такой оператор будет ошибочным. Необходимо или констатировать ошибку, или, если разработанный язык это допускает, выполнить преобразование строковой переменной в число перед сложением.

В результате проверки корректности промежуточного представления образуется машинно-независимый код. С его получением завершается т.н. *компиляция переднего плана* (front-end). Данное представление все еще не является последовательностью машинным кодов, за получение которой отвечает *компилятор заднего плана* (back-end). Его центральным элементом является собственно генератор кода, однако как с промежуточным представлением, так и с итоговым кодом возможно выполнение оптимизирующих преобразований.

Сложность разработки компилятора, особенно на уровне синтаксического анализа, существенно зависит от класса реализуемого языка. Классификация грамматик была предложена Хомским и включает четыре типа: фразовую, контекстно-зависимую, контекстно-свободную и регулярную грамматики.

Представим программу на ассемблере в виде потока *лексем* («элементов языка»). Например, команда *nop* не имеет аргументов и может быть использована в исходном виде, а для команды mov требуется два операнда. Если рассматривать строки вида

*<command> <op1>, <op2>*

можно видеть, что в строке за командой следуют два операнда, разделенные запятой. Тогда алгоритм анализа строки может содержать следующие шаги:

1. Выделить команду (ограниченную пробелом) и сравнить ее со списком допустимых команд.

2. Выделить первый операнд (ограниченный пробелом или запятой).

3. Выделить второй операнд.

Поскольку данные действия происходят последовательно, а от найденной команды зависит список требуемых операндов (например, если найдена команда nop, поиск операндов необходимо остановить, а если они найдены, это следует считать ошибкой), можно реализовать синтаксический разбор с помощью конечного автомата.

Фрагмент диаграммы для простого синтаксического анализатора показан на рис. 2.

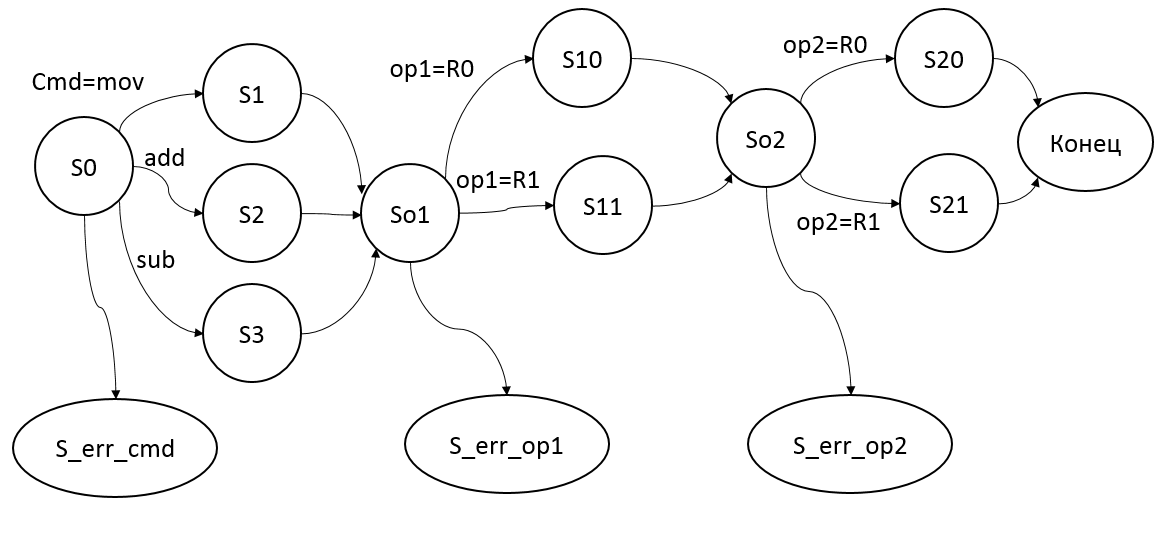


Рис. 2. Диаграмма переходов конечного автомата для простого синтактического анализатора

Пример реализации подобного синтаксического анализа показан в листинге. В примере разбираются строки вида «cmd op1, op2», причем команда может быть *mov* или *add*, а в качестве операндов принимаются только r0 и r1. Можно видеть, что в процессе анализа определяются команда и оба операнда, что позволяет получить значение для машинного кода проанализированной операции, просто подставив номера команды и операндов в соответствующие поля машинного кода.

**void** **syntax**(**char** \* str)

{

**int** op1, op2;

**char** \* token = **strtok**(str, " ");

**if** (**strcmp**(token, "mov") == 0)

{

token = **strtok**(NULL, " ,");

op1 = -1;

**if** (**strcmp**(token, "r0") == 0)

{

op1 = 0;

}

**if** (**strcmp**(token, "r1") == 0)

{

op1 = 1;

}

token = **strtok**(NULL, " ");

op2 = -1;

**if** (**strcmp**(token, "r0") == 0)

{

op2 = 0;

}

**if** (**strcmp**(token, "r1") == 0)

{

op2 = 1;

}

**printf**("mov: op1 = %d op2 = %d \n\r", op1, op2);

}

**if** (**strcmp**(token, "add") == 0)

{

token = **strtok**(NULL, " ,");

op1 = -1;

**if** (**strcmp**(token, "r0") == 0)

{

op1 = 0;

}

**if** (**strcmp**(token, "r1") == 0)

{

op1 = 1;

}

token = **strtok**(NULL, " ");

op2 = -1;

**if** (**strcmp**(token, "r0") == 0)

{

op2 = 0;

}

**if** (**strcmp**(token, "r1") == 0)

{

op2 = 1;

}

**printf**("add: op1 = %d op2 = %d \n\r", op1, op2);

}

}

**int** **main**(**void**) {

**char** s1[] = "mov r0, r1";

**char** s2[] = "add r1, r0";

syntax(s1);

syntax(s2);

**return** 0;

}

// Результаты работы:

mov: op1 = 0 op2 = 1

add: op1 = 1 op2 = 0

Листинг. Простейший синтаксический анализ

Явно видимым недостатком представленного подхода является большая трудоемкость описания всех возможных комбинаций команд и их операндов. Поэтому при проектировании процессора следует придерживаться *ортогональной* архитектуры команд. Это свойство подразумевает возможность использования единообразного формата команд, что упрощает кодирование и позволяет в частности, после определения индексов регистров формировать машинный код без дополнительных проверок на допустимость сочетания именно этой пары операндов.

Организация переходов между командами в простейшем случае может быть реализована непосредственно в процессе ассемблирования. Для этого необходимо отметить команду, на которую требуется организовать переход. Например, ключевое слово label не генерирует команду процессора, но отмечает текущую команду для последующей ссылки на нее. После ключевого слова должен следовать номер или обозначение метки.

Например,

Label 1

Отмечаемые метки записываются в специальную структуру ассемблера – таблицу меток. Впоследствии при упоминании метки ее адрес будет уже известен из таблицы.

Однако таким образом невозможно описать переход вперед, поскольку такая метка еще не объявлена. Для этого места в программе, упоминающие переход к еще не объявленной метке, также запоминаются в таблице в качестве *неразрешенных* (unresolved, более точное понятие – «пока не решенных», «пока не выясненных»). При объявлении метки необходимо проверить, нет ли в этой таблице отмеченных ранее переходов на такую метку и вписать в соответствующие команды программы выясненный адрес.

**Исходные данные**

|  |  |
| --- | --- |
| **Команда** | **Действие** |
| lite | В регистр записывает литерал |
| plus | Сложение двух регистров и запись в третий |
| lp | Сложение регистра с литералом |
| dmem | Получение значения из памяти и запись в регистр |
| end | Выход из файла в консольном режиме |

Команды записаны в специальный файл, из которого они подаются на вход:

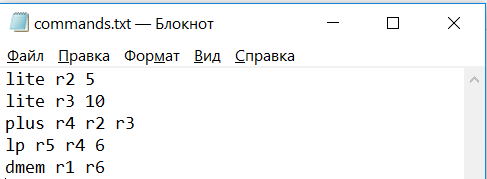


Рис. 3. Скриншот файла с командами

**Результат**

Результатом является распознавание и отображение порядка выполнения команд посредством вывода в консоль:

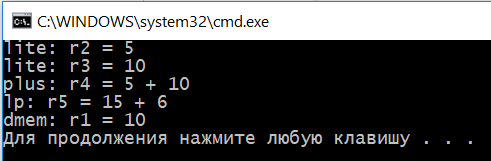


Рис. 4. Скриншот вывода в консоль результата команд.

**Выводы**

В ходе данной работы разработана программа, которая распознаёт команды ассемблера для некой модели процессора.

Разработка программы-распознавателя ассемблера для модели некого процессора выполнено на языке программирования C# с помощью IDE Microsoft Visual Studio.

**Приложение**

1. Класс реализации распознавания команд ассемблера.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Assembler\_lab3

{

class Parsing

{

private int[] Reg = new int[16];

private int[] cmem = new int[1024];

private int[] dmem = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160 };

private int literal; // Числовое значение

private int dest; // Номер регистра для записи результата

private int op1; // Номер регистра первого операнда

private int op2; // Номер регистра второго операнда

private bool flag = true;

// Ввод в консоль.

public void EnterConsole()

{

while (flag)

{

Console.Write("Введите команду: ");

string consoleEnter = Console.ReadLine();

GetArrayToken(consoleEnter);

}

}

// Метод определяющий главный токен.

private void GetArrayToken(string consoleEnter) {

// Парсинг в массив токинов

string[] arrayOfStrings = consoleEnter.Split(' ');

if (arrayOfStrings.Length == 0)

{

new Exception();

}

Switcher(arrayOfStrings);

}

private void Switcher(string[] arrayOfStrings)

{

string cmdType = arrayOfStrings[0];

byte lenght = (byte)arrayOfStrings.Length;

switch (cmdType)

{

case "lite" : LiteCommand(arrayOfStrings, lenght, "lite: "); break;

case "dmem": DmemCommand(arrayOfStrings, lenght, "dmem: "); break;

case "plus": PlusCommand(arrayOfStrings, lenght, "plus: "); break;

case "lp": LpCommand(arrayOfStrings, lenght, "lp: "); break;

case "end": IfCommnad(arrayOfStrings, lenght, "end: "); break;

default:

break;

}

}

private void LiteCommand(string[] arrayOfStrings, byte lenght, string nameCommand)

{

if (lenght != 3)

{

new Exception();

}

else

{

dest = Convert.ToInt32(arrayOfStrings[1].Substring(1));

literal = Convert.ToInt32(arrayOfStrings[2]);

Reg[dest] = literal;

Console.WriteLine(nameCommand + arrayOfStrings[1]+ " = " + Reg[dest]);

}

}

private void DmemCommand(string[] arrayOfStrings, byte lenght, string nameCommand)

{

if (lenght != 3)

{

new Exception();

}

else

{

op1 = Convert.ToInt32(arrayOfStrings[1].Substring(1));

op2 = Convert.ToInt32(arrayOfStrings[2].Substring(1));

Reg[op1] = dmem[Reg[op2]];

Console.WriteLine(nameCommand + arrayOfStrings[1] + " = " + dmem[Reg[op2]]);

}

}

private void PlusCommand(string[] arrayOfStrings, byte lenght, string nameCommand)

{

if (lenght != 4)

{

new Exception();

}

else

{

dest = Convert.ToInt32(arrayOfStrings[1].Substring(1));

op1 = Convert.ToInt32(arrayOfStrings[2].Substring(1));

op2 = Convert.ToInt32(arrayOfStrings[3].Substring(1));

Reg[dest] = Reg[op1] + Reg[op2];

Console.WriteLine(nameCommand + arrayOfStrings[1] + " = " + Reg[op1] + " + " + Reg[op2]);

}

}

private void LpCommand(string[] arrayOfStrings, byte lenght, string nameCommand)

{

if (lenght != 4)

{

new Exception();

}

else

{

dest = Convert.ToInt32(arrayOfStrings[1].Substring(1));

op1 = Convert.ToInt32(arrayOfStrings[2].Substring(1));

literal = Convert.ToInt32(arrayOfStrings[3]);

Reg[dest] = Reg[op1] + literal;

Console.WriteLine(nameCommand + arrayOfStrings[1] + " = " + Reg[op1] + " + " + literal);

}

}

private void IfCommnad(string[] arrayOfStrings, byte lenght, string nameCommand)

{

if (lenght != 1)

{

new Exception();

}

else

{

flag = false;

}

}

public void ReadTxtFile(string filePath)

{

using (System.IO.StreamReader sr = new System.IO.StreamReader(filePath, System.Text.Encoding.Default))

{

string line;

while ((line = sr.ReadLine()) != null)

{

GetArrayToken(line);

}

}

}

}

}

1. Класс запуска приложения

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Assembler\_lab3

{

class Program

{

static void Main(string[] args)

{

const string fileTxt = @"C:\commands.txt";

Parsing p = new Parsing();

p.ReadTxtFile(fileTxt);

}

}

}