МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«Дальневосточный федеральный университет»

ШКОЛА ЕСТЕСТВЕННЫХ НАУК

Кафедра информационных систем управления

Бажан Всеволод Евгеньевич

КОМПИЛЯТОР С ЯЗЫКА ВЫСОКОГО УРОВНЯ НА ЯЗЫК АССЕМБЛЕРА

КУРСОВАЯ РАБОТА

Студент гр. Б8216 _____
(подпись)

Ассистент
_____ Д.А.Бушко

Регистрационный № _____

_____ _____
подпись И.О.Фамилия
« _____» _____ 2018 г.

Оценка _____

_____ _____
подпись И.О.Фамилия

«_____» _____ 2019 г.

г. Владивосток
2019

Оглавление

Введение

Данный курсовой проект направлен на применение знаний, полученных в процессе изучения курса «Организация ЭВМ и периферийные устройства» и на получение практических навыков создания компилятора с языка *C* на язык ассемблера.

Целью курсовой работы является изучение структуры, основных принципов построения и функционирования трансляторов, и практическое освоение способов создания компилятора.

1 Постановка задачи

Разработать компилятор с подмножество языка *C*, включающего в себя: определение переменных, определение и вызов функций, условный оператор (*if*), оператор цикла (*while*), бинарные арифметические операции (+, –, *, /), бинарные логические выражения (<, >, ==, <=, >=), и унарную операцию побитовой инверсии (^), на язык ассемблера для процессора lilpM32. Этот компилятор носит название lilpcc (lilp C Compiler).

2 Язык ассемблера для процессора lilpM32

Ассемблер — компьютерная программа, преобразующая исходный текст программы, написанной на языке ассемблера, в программу на машинном языке (то есть в последовательность инструкций процессора). *Язык ассемблера* — язык программирования низкого уровня, мнемонические команды которого соответствуют инструкциям процессора. *Инструкция процессора* — элементарная операция, которую может выполнить процессор. Различные процессоры имеют различные *наборы инструкций*, поэтому язык ассемблера не является универсальным, он зависит от конкретного процессора.

Процессор lilpM32 разрабатывался для образовательных целей. Его набор инструкций и внутреннее устройство представляют собой очень сильно упрощённый вариант реализации архитектуры MIPS I. *MIPS* — семейство процессоров, разработанных компанией MIPS Technologies, и являющихся реализацией архитектуры RISC. *RISC* — архитектура процессора, в которой быстродействие увеличивается за счёт упрощения инструкций. В настоящее время различные реализации MIPS используются в основном во встроенных системах.

lilpM32 — 32-битный процессор. Это означает, что длина инструкции, а также разрядность обрабатываемых им данных — 32 бита. В lilpM32 нельзя получить доступ к отдельному байту памяти (как это обычно принято) — только к 32-разрядному слову. Значение, поступающее на

адресный вход ОЗУ, задаёт номер 32-разрядного слова (а не байта) для чтения или записи. Таким образом, в нашем распоряжении примерно 16 миллионов 4-байтовых ячеек (то есть 64 мегабайта памяти). Для инструкций и для данных используется общая память, то есть lilpM32 имеет архитектуру фон Неймана.

Для работы с целыми числами и адресами lilpM32 имеет 32 32-битных регистра. Они объединены в регистровый файл. Номера и имена регистров принято начинать со знака доллара («$»); для ассемблера это обязательное условие. Регистр с номером 0 ($0) всегда содержит значение «0» (запись в него не имеет смысла). В регистр $31 сохраняется адрес возврата из подпрограммы (функции). Эти два условия гарантируются аппаратной реализацией процессора. Кроме того, все остальные регистры имеют свои имена и назначение, однако это только соглашение, и сам процессор никак не контролирует выполнение этого соглашения — это задача программиста. В таблице 2.1 представлены названия регистров и их назначение.

Таблица 2.1 — Назначение и названия регистров

Название	Номер	Назначение	Сохраняется вызываемой функцией?
$zero	$0	Константа 0	Нет
$at	$1	Временный для ассемблера	Нет
$v0–$v1	$2–$3	Значения, возвращаемые функциями	Нет
$a0–$a3	$4–$7	Аргументы функций	Нет
$t0–$t7	$8–$15	Временные	Нет
$s0–$s7	$16–$23	Сохраняемые временные	Да
$t8–$t9	$24–$25	Временные	Нет
$k0-$k1	$26–$27	Зарезервированы для ядра ОС	Нет

Окончание таблицы 2.1

$gp	$28	Глобальный указатель на данные	Да
$sp	$29	Указатель стека	Да
$fp	$30	Указатель окна	Да
$ra	$31	Адрес возврата из функции	Нет

lilpM32 имеет два регистра для сохранения двух частей 64-битного результата умножения и целочисленного деления: регистры LO и HI. Для ввода и вывода данных у процессора предусмотрено по четыре 32-битных буфера (регистра) ввода/вывода. Для доступа к этим буферам процессор имеет по четыре 32-битных входа и выхода для данных, тактовых входа, тактовых выхода, сигнализирующих о чтении данных процессором, и тактовых выхода сигнализирующих о записи данных процессором. При записи нового значения (после выполнения соответствующей инструкции) в буфер, процессор подаёт тактовый импульс на соответствующий тактовый выход. Чтобы записать новое значение в буфер ввода, нужно подать это значение на соответствующий вход для данных, а на тактовый вход подать тактовый импульс.
После этого можно обращаться к записанному значению посредством соответствующей инструкции. Каждый раз, когда процессор считывает значение из буфера, он подаёт тактовый импульс на соответствующий тактовый выход. Буферы ввода/вывода можно использовать для подачи в процессор символов с клавиатуры, или для вывода информации на терминал или на светодиодную матрицу (непосредственно или используя отдельную видеопамять). Заметьте, что наличие буферов ввода/вывода и инструкций для работы с ними — отличительная особенность lilpM32; это сделано для упрощения. Как правило, ввод и вывод MIPS процессоры осуществляют несколько другими способами.

Кроме того, у процессора предусмотрен тактовый выход, который позволяет подавать во внешнюю схему импульсы непосредственно с тактового генератора, расположенного внутри схемы процессора.

Все инструкции процессора lilpM32 имеют длину 32 бита. В зависимости от типа инструкции (существуют *R*, *I* и *J* инструкции), эти 32 бита разбиваются на поля, каждое из которых несёт определённую информацию — код операции (*opcode*), номера регистров *($s, $t, $d*), величину сдвига (*shamt*), код функции (*funct*), непосредственное значение (*imm*) и адрес перехода (*addr*). Для инструкций R-типа значение кода операции всегда равно нулю, а инструкция определяется по коду функции. Для инструкций I-типа сама инструкция содержит непосредственное 16-битное целочисленное значение, которое может быть интерпретировано (и расширено до 32-битного) как беззнаковое, или как знаковое, записанное в дополнительном коде (в зависимости от конкретной инструкции). Форматы инструкций представлены в таблице 2.2.

Таблица 2.2 — Форматы инструкций

Тип	Формат (биты)					
R	opcode (6)	$s (5)	$t (5)	$d(5)	shamt (5)	funct (6)
I	opcode (6)	$s (5)	$t (5)	imm (16)		
J	opcode (6)	addr (26)				

Далее приводится список всех доступных инструкций lilpM32 с подробными описаниями (таблица 2.3). В конце таблицы располагаются *псевдоинструкции* (они отмечены прочерком в столбце «Формат»). Псевдоинструкции не являются инструкциями процессора; при компиляции ассемблер заменяет их определённой комбинацией настоящих инструкций. Использование псевдоинструкций облегчает написание и чтение кода.

Таблица 2.3 — Набор инструкций процессора lilpM32

Синтаксис	Название	Действие	Формат / opcode / funct		
addu $d,$s,$t	Add unoverflow (Сложить без переполнения)	$d = $s + $t	R	0x00	0x21
Складывает два регистра, игнорируя переполнение.					
subu $d,$s,$t	Subtract unoverflow (Вычесть без переполнения)	$d = $s – $t	R	0x00	0x23
Вычитает два регистра, игнорируя переполнение.					
addiu $t,$s,imm	Add immediate unoverflow (Сложить непосредственное без переполнения)	$t = $s + imm	I	0x09	–
Складывает регистр с константой, игнорируя переполнение. `imm` — знаковое.					

На каждой строке может быть не больше одной инструкции. Регистры можно указывать по номерам (*$0–$31*) или по именам (*$zero*, *$at*, *$t0-$t9* и другие). Непосредственные значения (*imm* и *shamt*) можно задавать как в шестнадцатеричном виде (с префиксом «0x»), так и в десятичном. Если такое значение выходит за допустимый диапазон, то ассемблер выдаст сообщение об ошибке. Параметры *imm* и *addr* для инструкций *beq*, *bne*, *j* и *jal* могут быть численными значениями или метками, причём если для инструкций *beq* и *bne* адрес перехода задан меткой, то переход осуществляется непосредственно на эту метку, то есть дополнительная единица к значению программного счётчика не прибавляется. Параметр *label* для псевдоинструкций *la*, *bgt*, *blt*, *bge* и *ble* может быть только меткой. Строка кода может начинаться с объявления метки. Оно состоит из идентификатора метки (который может состоять из латинских букв, цифр, и символов подчёркивания, и не может начинаться с цифры), за которым следует двоеточие. При использовании в инструкциях метка заменяется ассемблером на фактический адрес в памяти инструкции или области данных, следующей за объявлением метки.

Продолжение таблицы 2.3

mult $s,$t	Multiply (Умножить)	LO = (($s * $t) << 32) >> 32; HI = ($s * $t) >> 32;	R	0x00	0x18
Перемножает два регистра и помещает 64-битный результат в два специальных 32-битных регистра — LO и HI.					
divu $s,$t	Divide unsigned (Делить беззнаковое)	LO = $s / $t; HI = $s % $t;	R	0x00	0x1b
Выполняет целочисленное деление над двумя регистрами и помещает частное в регистр LO, а остаток — в регистр HI.					
lw $t,imm($s)	Load word (Загрузить слово)	$t = Memory[$s + imm]	I	0x23	–
Загружает в регистр слово, находящееся в ОЗУ по адресу $s + imm. imm — знаковое.					
sw $t,imm($s)	Store word (Сохранить слово)	emory[$s + imm]= $	I	0x2b	–
Сохраняет слово из регистра в ОЗУ по адресу $s + imm. imm — знаковое.					
lui $t,imm	Load upper immediate (Загрузить в страшную половину непосредственное)	$t = imm << 16	I	0x0f	–
Загружает непосредственный 16-битный операнд в старшие 16 бит регистра. imm — беззнаковое.					
mfhi $d	Move from HI (Копировать из HI)	$d = HI	R	0x00	0x10
Копирует значение из регистра HI в указанный регистр.					
mflo $d	Move from LO (Копировать из LO)	$d = LO	R	0x00	0x12
Копирует значение из регистра LO в указанный регистр.					
lfb $t,imm($s)	Load from buffer (Сохранить в буфер)	$t = inbuf[$s + imm]	I	0x2d	–
Загружает в регистр значение из буфера, номер которого (от 0 до 3) задаётся значением $s + imm (используются только 2 младших бита этого значения). imm — знаковое. Чтение буфера происходит на стадии конвейера «Доступ к памяти».					

Продолжение таблицы 2.3

stb $t,imm($s)	Store to buffer (Сохранить в буфер)	outbuf[$s + imm]= $t	I	0x2e	–
Записывает значение регистра в буфер, номер которого (от 0 до 3) задаётся значением $s + imm (используются только 2 младших бита этого значения). imm — знаковое. Запись в буфер происходит на стадии конвейера «Доступ к памяти».					
and $d,$s,$t	And (И)	$d = $s & $t	R	0x00	0x24
Выполняет побитовое "И" над двумя регистрами и записывает результат в третий.					
andi $t,$s,imm	And immediate (И с непосредственным)	$t = $s & imm	I	0x0c	–
Выполняет побитовое "И" над регистром и константой и записывает результат в регистр. imm — беззнаковое.					
or $d,$s,$t	Or (ИЛИ)	$d = $s | $t	R	0x00	0x25
Выполняет побитовое "ИЛИ" над двумя регистрами и записывает результат в третий.					
ori $t,$s,imm	Or immediate (ИЛИ с непосредственным)	$t = $s | imm	I	0x0d	–
Выполняет побитовое "ИЛИ" над регистром и константой и записывает результат в регистр. imm — беззнаковое.					
xor $d,$s,$t	Exclusive or (Исключающее ИЛИ)	$d = $s ^ $t	R	0x00	0x26
Выполняет побитовое "Исключающее ИЛИ" над двумя регистрами и записывает результат в третий					
nor $d,$s,$t	Nor (ИЛИ-НЕ)	$d = ! ($s | $t)	R	0x00	0x27
Выполняет побитовое ИЛИ-НЕ над двумя регистрами и записывает результат в третий.					
slt $d,$s,$t	Set on less than (Установить, если меньше)	$d = ($s < $t)	R	0x00	0x2a
Сравнивает значения двух регистров как знаковые (в дополнительном коде) и записывает результат в третий («1» — если значение первого меньше, «0» — в противном случае).					

Продолжение таблицы 2.3

slti $t,$s,imm	Set on less than immediate (Установить, если меньше, чем непосредственное)	$t = ($s < imm)	I	0x0a	–
Сравнивает значения регистра и константы как знаковые и записывает результат в третий («1» — если значение регистра меньше, «0» — в противном случае).					
sll $t,$s,shamt	Shift left logical (Левый логический сдвиг)	$t = $s << shamt	R	0x00	0x00
Сдвигает значение регистра на shamt битов влево и записывает результат в другой регистр. Фактически, умножает значение регистра на 2^{shamt}. shamt — беззнаковое.					
srl $t,$s,sham	Shift right logical (Правый логический сдвиг)	t = $s >> shamt	R	0x00	0x02
Сдвигает значение регистра на shamt битов вправо, заполняя освободившееся место нулями, и записывает результат в другой регистр. Фактически, делит значение регистра на 2^{shamt}, если оно рассматривается как беззнаковое, или положительное знаковое. shamt — беззнаковое.					
sra $t,$s,shamt	Shift right arithmetic (Правый арифметический сдвиг)	$t = ($s >> shamt) + sign_extension	R	0x00	0x03
Сдвигает значение регистра на shamt битов вправо, заполняя освободившееся место битами знака, и записывает результат в другой регистр. Фактически, делит значение регистра на 2^{shamt}, если оно рассматривается как знаковое. shamt — беззнаковое.					
beq $s,$t,imm	Branch on equal (Ветвление при равенстве)	if ($s == $t) PC = PC + 1 + imm	I	0x04	–
Переходит на выполнение инструкции по адресу PC + 1 + imm (где PC — текущее значение программного счётчика), если два регистра равны. imm — знаковое целое или метка.					
bne $s,$t,imm	Branch on not equal (Ветвление при неравенстве)	if ($s != $t) PC = PC + 1 + imm	I	0x05	–
Переходит на выполнение инструкции по адресу PC + 1 + imm (где PC — текущее значение программного счётчика), если два регистра не равны. imm — знаковое целое или метка.					

Продолжение таблицы 2.3

j addr	Jump (Прыжок)	PC = addr	J	0x02	–
Переходит на выполнение инструкции по адресу addr. addr — беззнаковое 26-битное значение или метка; поскольку разрядность адреса ОЗУ — 24 бита, используются только 24 младших бита addr.					
jr $s	Jump register (Прыжок к значению регистра)	PC = $s	R	0x00	0x08
Переходит на выполнение инструкции по адресу, хранящемуся в регистре. Используются только 24 младших бита этого значения.					
jal addr	Jump and link (Прыжок и связывание)	$31 = PC + 4; PC = addr;	J	0x03	–
Переходит на выполнение инструкции по адресу addr и записывает в регистр $31 ($ra) адрес возврата. Используется для вызова подпрограмм. Возврат из подпрограммы осуществляется вызовом инструкции jr $ra. addr — беззнаковое целое или метка, используются только 24 его младших бита.					
nop	No operation (Нет операции)	–	–	–	–
Не выполняет никакой операции. Часто используется в качестве «заглушки». Заменяется ассемблером на: sll $0,$0,0					
move $t,$s	Move (Копировать)	$t = $s	–	–	–
Копирует содержимое одного регистра в другой. Заменяется ассемблером на: addiu $t,$s,0					
la $at,label	Load address (Загрузить адрес)	$at = label	–	–	–
Загружает в регистр фактический адрес метки. Заменяется ассемблером на: lui $at,label[31:16]; nop; nop; nop; ori $at,$at,label[15:0]					
li $at,imm	Load immediate (Загрузить непосредственное)	$at = imm	–	–	–
Загружает в регистр непосредственное 32-битное знаковое значение. Заменяется ассемблером на:lui $at,imm[31:16]; nop; nop; nop; ori $at,$at,imm[15:0]					
lli $t,imm	Load lower immediate (Загрузить в младшую половину непосредственное)	$t = imm	–	–	–
Загружает в регистр непосредственное 16-битное знаковое значение. Заменяется ассемблером на: addiu $t,$0,imm					

Продолжение таблицы 2.3

bgt $s,$t,label	Branch if greater than (Ветвление, если больше)	bgt $s,$t,label	–	–	–
Переходит на выполнение инструкции по адресу label, если значение одного регистра больше, чем значение другого. Заменяется ассемблером на: slt $at,$t,$s; nop; nop; nop; bne $at,$zero,label					
blt $s,$t,label	Branch if less than (Ветвление, если меньше)	blt $s,$t,label	–	–	–
Переходит на выполнение инструкции по адресу label, если значение одного регистра меньше, чем значение другого. Заменяется ассемблером на: slt $at,$s,$t; nop; nop; nop; bne $at,$zero,label					
bge $s,$t,label	Branch if greater than or equal (Ветвление, если больше или равно)	bge $s,$t,label	–	–	–
Переходит на выполнение инструкции по адресу label, если значение одного регистра больше или равно значению другого. Заменяется ассемблером на: slt $at,$s,$t; nop; nop; nop; beq $at,$zero,label					
ble $s,$t,label	Branch if less than or equal (Ветвление, если меньше или равно)	if ($s <= $t) PC = label	–	–	–
Переходит на выполнение инструкции по адресу label, если значение одного регистра меньше или равно значению другого. Заменяется ассемблером на:slt $at,$t,$s; nop; nop; nop; beq $at,$zero,label					
mul $d,$s,$t	Multiply and return low (Умножить и вернуть младшее)	$d = (($s * $t) << 32) >> 32	–	–	–
Перемножает значения двух регистров и записывает младшие 32 бита результата в третий. Заменяется ассемблером на:mult $s,$t; nop; nop; nop; mflo $d					

Окончание таблицы 2.3

div $d,$s,$t	Divide and return quotient (Делить и вернуть частное)	$d = $s / $t	–	–	–
Выполняет целочисленное деление над двумя регистрами и помещает частное в регистр. Заменяется ассемблером на: divu $s,$t; nop; nop; nop; mflo $d					
rem $d,$s,$t	Divide and return reminder (Делить и вернуть остаток)	$d = $s % $t	–	–	–
Выполняет целочисленное деление над двумя регистрами и помещает остаток в регистр. Заменяется ассемблером на: divu $s,$t; nop; nop; nop; mfhi $d					

На каждой строке может быть не больше одной инструкции. Регистры можно указывать по номерам (*$0–$31*) или по именам (*$zero*, *$at*, *$t0-$t9* и другие). Непосредственные значения (*imm* и *shamt*) можно задавать как в шестнадцатеричном виде (с префиксом «0x»), так и в десятичном. Если такое значение выходит за допустимый диапазон, то ассемблер выдаст сообщение об ошибке. Параметры *imm* и *addr* для инструкций *beq*, *bne*, *j* и *jal* могут быть численными значениями или метками, причём если для инструкций *beq* и *bne* адрес перехода задан меткой, то переход осуществляется непосредственно на эту метку, то есть дополнительная единица к значению программного счётчика не прибавляется. Параметр *label* для псевдоинструкций *la*, *bgt*, *blt*, *bge* и *ble* может быть только меткой. Строка кода может начинаться с объявления метки. Оно состоит из идентификатора метки (который может состоять из латинских букв, цифр, и символов подчёркивания, и не может начинаться с цифры), за которым следует двоеточие. При использовании в инструкциях метка заменяется ассемблером на фактический адрес в памяти инструкции или области данных, следующей за объявлением метки.

3 Лексический анализ

Лексический анализ — первый этап компиляции программы. Часть исходного кода компилятора, отвечающего за лексический анализ, называют *лексером*, *сканером* или *токенайзером*. *Токен* или *лексема* — наименьшая единица программы. К токенам относятся: идентификаторы, числа, зарезервированные слова, разделители, знаки математических и логических операций. Особым случаем являются пробельные символы — пробел, табуляция, конец строки — в языках семейства *C* эти символы игнорируются лексером, но в некоторых других языках, например в языке *Python*, пробелы являются значимыми символами. Таким образом с позиции лексера, программа — это набор входных символов, которые необходимо превратить в токены.

При реализации лексера токенам сопоставляется числовое значение, называемое *код токена*. Код токена однозначно определяет его вид. Однако для токенов «число» и «идентификатор» установления вида недостаточно. На этапе генерации кода также требуется информация о том, какое именно число представляет токен и какое имя имеет идентификатор. Из чего следует, что этим токенам помимо вида, необходимо присвоить и значение.

Закодированная последовательность токенов, получающаяся после «прохода» лексера по тексту исходной программы, называются *лексической сверткой*. С помощью лексической свертки гарантируется независимость следующих этапов компиляции от представления исходной программы и используемого набора символов.

3.1 Лексический анализатор в компиляторе lilpcc

В данной реализации для создания лексического анализатора используется генератор лексических анализаторов *Flex*. Генератор получает на вход текст исходной программы и формирует код анализатора на языке *C*.

Рисунок 3.1 — Положение лексического анализа в компиляторе lilpcc

Правила определения токенов являются отдельными символами или регулярными выражениями.

Все пробельные символы игнорируются с помощью правила:

```
[ \t\n]+ { /* Ignore whitespaces. */ }
```

Символы пунктуации считываются следующими правилами:

```
"{" { return OP_BRACE; }
"}" { return CL_BRACE; }
"(" { return OP_PAREN; }
")" { return CL_PAREN; }
";" { return SEMICOLON; }
"," { return COMMA; }
```

Арифметические операторы и операторы сравнения также, как и символы пунктуации, являются явно указанными символами и описываются нижеперечисленными правилами:

```
"="  { return ASSIGN; }
"+"  { return PLUS; }
"-"  { return MINUS; }
"*"  { return MULT; }
"/"  { return DIV; }
"~"  { return XOR; }
"<"  { return LESS; }
">"  { return GREATER; }
"<=" { return LESS_OR_EQUAL; }
">=" { return GREATER_OR_EQUAL; }
"==" { return EQUAL; }
```

Ключевые слова представляют собой конкретные последовательности символов, а числа и идентификаторы задаются регулярными выражениями.

```
"if"     { return IF; }
"while"  { return WHILE; }
"return" { return RETURN; }
"int"    { return INT; }
[0-9]+   { SAVE_TOKEN; return NUMBER; }
[a-z_][a-zA-Z0-9_]* { SAVE_TOKEN; return ID; }
```

4 Синтаксический анализ

Синтаксический анализ или *парсинг* — второй этап компиляции программы. Задача *парсера* заключается в преобразовании лексической свертки в *абстрактное синтаксическое дерево*. Абстрактное синтаксическое дерево (АСД) — один из способов представления структуры программы. В языке *C* сложные конструкции, такие как условия или определения функций, образованы из более простых структур, например, переменных и констант. АСД фиксирует взаимосвязь между этими конструкциями. Корнем АСД всегда является вся программа, отдельные конструкций представляются листами дерева, а сложные структуры поддеревьями. Рассмотрим следующий пример:

```
int main() {
   int a = 5;
   if (a < 10) {
      a = a + 10;
      return a;
   }
}
```

В этом примере есть определение функции (main), определение константы (int a), условное выражение (a < 10), бинарная операция сложения с последующим присвоением (a + 10) и оператор возврата. На рисунке ?.1 изображено АСД для данного примера.

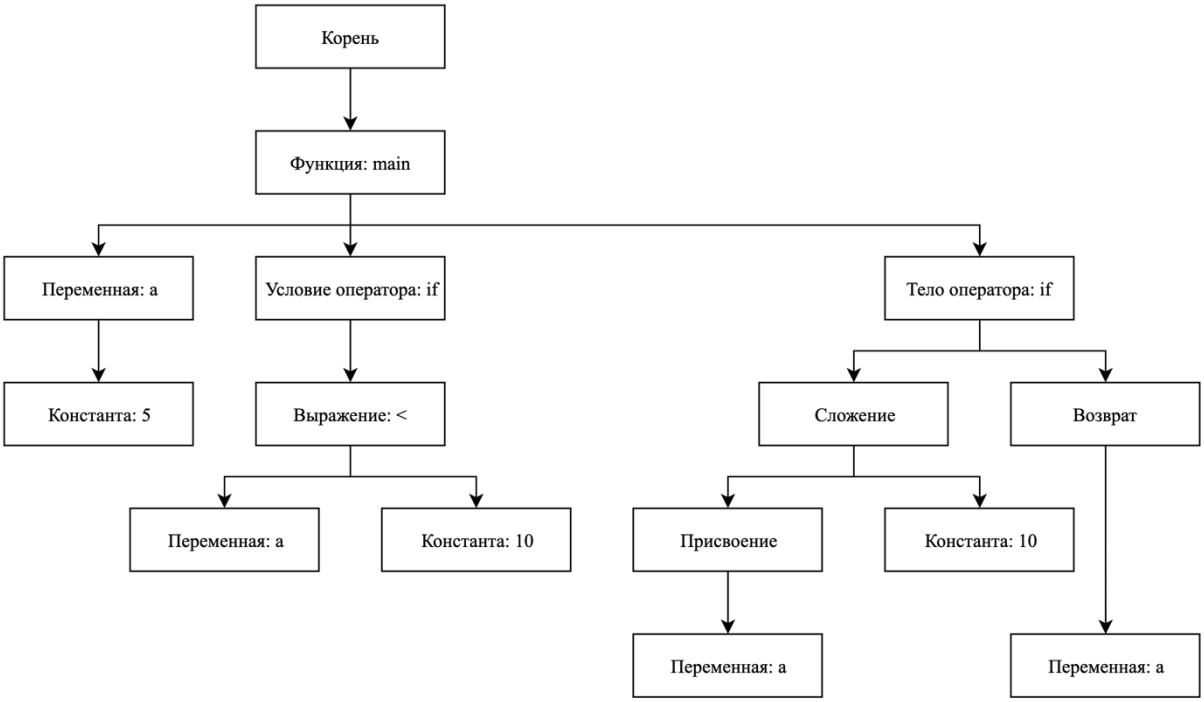

Рисунок 4.1 — Пример абстрактного синтаксического дерева

Парсер формирует АСД согласно набору правил, называемых *грамматикой* языка. Одним из классов грамматик является *контекстно-свободная грамматика (КС)*. КС-грамматика имеет четыре элемента:

1. *Терминальные* символы. Также именуемых *терминалами* или *токенами*. Именно эти токены были описаны в разделе, посвященному лексическому анализу.
2. *Нетерминалы*. Нетерминал является конечной последовательностью токенов.
3. *Продукции*. Каждая продукция состоит из *левой части*, являющейся единственным нетерминалом, символа стрелки (или другого специального символа) и последовательности терминалов и нетерминалов, называемых *правой частью* продукции.
4. *Стартовый* нетерминал. Стартовый нетерминал является корнем синтаксического дерева, это начальный элемент.

КС-грамматика не имеет ограничений на вид правых частей продукций, однако левая часть всегда должна быть единственным терминалом.

4.1 Синтаксический анализатор в компиляторе lilpcc

В *lilpcc* для создания парсера используется генератор синтаксических анализаторов *GNU Bison* (далее — *Bison*). *Bison* работает в паре с *Flex*, который используется для создания лексера. *Flex* применяется для определения токенов, которые играют роль терминальных символов в описании грамматики, на основе которой *Bison* автоматически формирует синтаксический анализатор.

Рисунок 4.2 — Положение синтаксического анализа в компиляторе lilpcc

Далее опишем *продукции*, используемые в парсере *lilpcc*, начиная с описания *стартового* нетерминала. Прочесть такую конструкцию можно как: “Программа — это функция, за которой следует программа, или пустой нетерминал.” Нетрудно заметить, что данное определение является рекурсивным — «программа» определена через «программу». Это одна из особенностей КС-грамматики.

```
program :
  function program
  |
  ;
```

В определении ниже и далее в качестве терминальных символов используются токены, описанные в прошлом разделе.

```
function :
  INT ID OP_PAREN parameter_list CL_PAREN OP_BRACE block CL_BRACE
  ;
```

Распространенной практикой при описании грамматики является использование перехода от общего к частному и обратно.

Так, определив правило для списка параметров, мы можем определить правило для не пустого списка параметра, обратившись в нем к общему правилу для списка параметров.

```
parameter_list :
   non_empty_parameter_list
   |
   ;
non_empty_parameter_list :
   INT ID COMMA parameter_list
   | INT ID
   ;
```

Блоком (*block*) называется набор команд языка, заключенного в фигурные скобки ({}). Это правило демонстрирует пользу использования рекурсии в описании продукций, поскольку с помощью рекурсии легко реализуются парсинг вложенных блоков.

```
block :
   statement block
   |
   ;
```

Списки аргументов организованы аналогично спискам параметров.

```
argument_list :
   non_empty_argument_list
   |
   ;
non_empty_argument_list :
   expression COMMA non_empty_argument_list
   | expression
   ;
```

Продукция высказывания (*statement*) является важной составляющей парсера, так как она определяет сложные конструкции, которые всегда будут иметь наследников в абстрактном синтаксическом дереве.

```
statement :
   RETURN expression SEMICOLON
   | IF OP_PAREN expression CL_PAREN OP_BRACE block CL_BRACE
   | WHILE OP_PAREN expression CL_PAREN OP_BRACE block CL_BRACE
   | INT ID ASSIGN expression SEMICOLON
   | expression SEMICOLON {}
   ;
```

Наиболее объемной продукция является продукция выражения (*expression*), которая описывает правила для чисел и идентификаторов, оператора присваивания, операторов сравнения и арифметических операторов, а также вызова функции.

```
expression :
   NUMBER
   | ID
   | ID ASSIGN expression
   | XOR expression
   | expression PLUS expression
   | expression MINUS expression
   | expression MULT expression
   | expression DIV expression
   | expression LESS expression
   | expression LESS_OR_EQUAL expression
   | expression GREATER expression
   | expression GREATER_OR_EQUAL expression
   | ID OP_PAREN argument_list CL_PAREN
   ;
```

5 Генерация кода

Генерация кода — последний этап компиляции программы. Хорошей практикой считается разделение модулей, которые отвечают за анализ от модулей, отвечающих за генерацию кода. Поэтому порождение (*emitting*) машинных команд, сосредотачивают в отдельной части компилятора, называемой *генератором кода*. Генератор кода получает на вход промежуточное представление исходной программы

(например, абстрактное синтаксическое дерево) и выводит набор машинных команд, логически эквивалентный коду исходной программы.

5.1 Генерация кода в компиляторе lilpcc

Генератор кода является технически наиболее сложной частью данного компилятора и состоит из нескольких частей.

Рисунок 5.1 — Положение генерации кода в компиляторе lilpcc

5.1.1 Представление синтаксических конструкций

Первой частью является модуль, отвечающий за представление синтаксических конструкций языка в оперативной памяти во время работы компилятора. Этот модуль определен в файле *syntax.h* и реализован в файле *syntax.c*.

Основной структурой в этом модуле является структура *Syntax*, содержащая тип и указатель на синтаксическую конструкцию (далее — конструкцию).

```
struct Syntax
{
  SyntaxType type;
  union {
    Immediate *immediate;
    Variable *variable;
    UnaryExpression *unary_expression;
    BinaryExpression *binary_expression;
    Assignment *assignment;
    ReturnStatement *return_statement;
    FunctionArguments *function_arguments;
```

```
        FunctionCall *function_call;
        IfStatement *if_statement;
        DefineVarStatement *define_var_statement;
        WhileStatement *while_statement;
        Block *block;
        Function *function;
        TopLevel *top_level;
    };
};
```

Структура *SyntaxType* является перечислением всех поддерживаемых компилятором конструкций.

```
typedef enum
{
    IMMEDIATE,
    VARIABLE,
    UNARY_OPERATOR,
    BINARY_OPERATOR,
    BLOCK,
    IF_STATEMENT,
    RETURN_STATEMENT,
    DEFINE_VAR,
    FUNCTION,
    FUNCTION_CALL,
    FUNCTION_ARGUMENTS,
    ASSIGNMENT,
    WHILE_SYNTAX,
    TOP_LEVEL
} SyntaxType;
```

Типы для унарного и бинарного выражения также являются перечислением всех поддерживаемых арифметических и логических операторов.

```
typedef enum
{
   BITWISE_NEGATION,
   LOGICAL_NEGATION
} UnaryExpressionType;
typedef enum
{
   ADDITION,
   SUBTRACTION,
   MULTIPLICATION,
   DIVISION,
   GREATER_THAN,
   GREATER_THAN_OR_EQUAL,
   LESS_THAN,
   LESS_THAN_OR_EQUAL,
} BinaryExpressionType;
```

Структуры, отвечающие за представление отдельных конструкций, являются элементами абстрактного синтаксического дерева, описанного в разделе, посвященному синтаксическому анализу.

Структуры, представляющие константы и переменные являются листами дерева, поскольку не имеют потомков и содержат только значения.

```
typedef struct Immediate
{
   int value;
} Immediate;
typedef struct Variable
{
   char *var_name;
} Variable;
```

Унарные и бинарные выражения имеют потомков. Этими потомками являются операнды выражения. Остальные конструкции также имеют потомков.

```
typedef struct UnaryExpression
{
   UnaryExpressionType unary_type;
   Syntax *expression;
} UnaryExpression;
typedef struct BinaryExpression
{
   BinaryExpressionType binary_type;
   Syntax *left;
   Syntax *right;
} BinaryExpression;
```

Параметры и аргументы функций — это связанный список конструкций.

```
typedef struct FunctionArguments
{
   List *arguments;
} FunctionArguments;
typedef struct FunctionCall
{
   char *function_name;
   Syntax *function_arguments;
} FunctionCall;
```

Оператор присваивания и определение переменной построены одинково. Они хранят название переменой и инициализирующее выражение.

```
typedef struct Assignment
{
   char *var_name;
   Syntax *expression;
} Assignment;
```

```
typedef struct DefineVarStatement
{
   char *var_name;
   Syntax *expression;
} DefineVarStatement;
```

Оператор ветвления и оператор цикла также построены по единому принципу. Они имеют условие и набор конструкций, являющихся «телом» конструкции.

```
typedef struct WhileStatement
{
   Syntax *condition;
   Syntax *body;
} WhileStatement;
typedef struct ReturnStatement
{
   Syntax *expression;
} ReturnStatement;
```

Блоком называется список высказываний.

```
typedef struct Block
{
   List *statements;
} Block;
typedef struct TopLevel
{
   List *declarations;
} TopLevel;
```

Структура функции — это её название, список параметров и блок.

```
typedef struct Function
{
   char *name;
   List *parameters;
   Syntax *root_block;
} Function;
```

5.1.2 Таблица символов

Второй частью генератора кода является модуль, задача которого заключается в сопоставлении имен идентификаторов на *смещения* в *текущем стековом фрейме*. Стековый фрейм всегда будет соответствовать некоторой *области видимости*.

Областью видимости будем называть часть программы на языке высокого уровня (в данном случае *C*), в рамках которой идентификатор, являющийся именем переменной, остается связанным с этой переменной. На языке *C* область видимости начинается с токена открывающейся фигурной скобки ({) и заканчивается токенов закрывающейся фигурной скобки (}).

Стеком называется область памяти процессора для динамически выделяемых данных. Таким образом, стек — это область памяти для хранения объектов, появляющихся во время выполнения программы.

Стековым фреймом называется область стека, относящиеся к текущей области видимости.

Смещением является «отступ» в байтах от вершины стека. То есть, зная смещение, можно перейти в ту ячейку памяти, в которой хранится переменная.

Структура, которая хранит идентификаторы и соответствующие ему смешения называется *таблицей символов*. Программный код, реализующий эту структуру определен в файле *symbol_table.h* и реализован в файле *symbol_ table.c*.

Основной структурой данных в этих файлах является структура *SymbolTable*, хранящая пары переменная–смещение и их количество.

```
typedef struct SymbolTable
{
   size_t size;
   VarWithOffset *items;
} SymbolTable;
```

Структура, представляющая пару переменная–смещение называется *VarWithOffset*.

```
typedef struct VarWithOffset
{
   char *var_name;
   int offset;
} VarWithOffset;
```

В файлах *context.*» и *context.*» определена структура, представляющая область видимости. Это структура хранит смещение последней переменной, структуру *SymbolTable*, количество *ассемблерных меток*.

```
typedef struct Context
{
   int stack_offset;
   SymbolTable *s_table;
   int label_count;
} Context;
```

Ассемблерной меткой является идентификатор, представляющий собой строку символов и обозначающее ячейку памяти, с которой начинается некоторый набор команд. Хранение количества таких меток позволяет сгенерировать новое уникальное имя метки, если таковое надобится.

5.1.3 Порождение команд

Последней частью генератора кода является модуль, который порождает (от англ. «*emit*») команды ассемблера, логически эквивалентные конструкциям языка *C*. Этот модуль определен в файле *assembly.h* и реализован в файле *assembly.c*.

Основной функцией в этом модуле является функция *write_assembly*, которая принимает на вход абстрактное синтаксическое дерево в виде структуры *Syntax*, описанной в разделе 5.1.1 и выводит в файл *out.asm* команды ассемблера. Именно эта функция используется в главной функции компилятора (файл *lilpcc.c*).

```
void write_assembly(Syntax *syntax)
{
   FILE *out = fopen("out.asm", "wb");

   Context *ctx = new_context();
   emit_instr(out, current_indent, "j", "main\n");
   write_syntax(out, syntax, ctx);
   context_free(ctx);

   fclose(out);
}
```

Функция, разбирающая абстрактное синтаксическое дерево и порождающая команды ассемблера, называется *write_syntax*. Эта функция слишком обширна для того, чтобы приводить её описание целиком, поэтому далее будут приведены части, которые заслуживают наибольшего внимания.

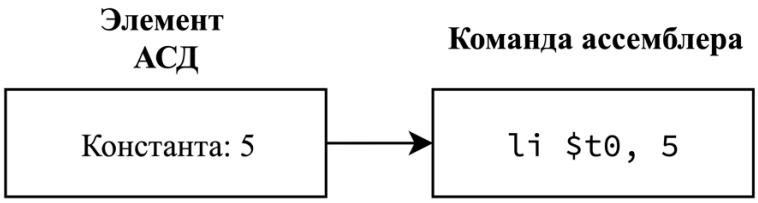

Рисунок 5.4 — Загрузка константы

Загрузка константы происходит одной командой ассемблера. Константа загружается во временный регистр *$t0*.

```
emit_instr_format(out, current_indent, "li", "$t0, %d", syntax->immediate->value);
```

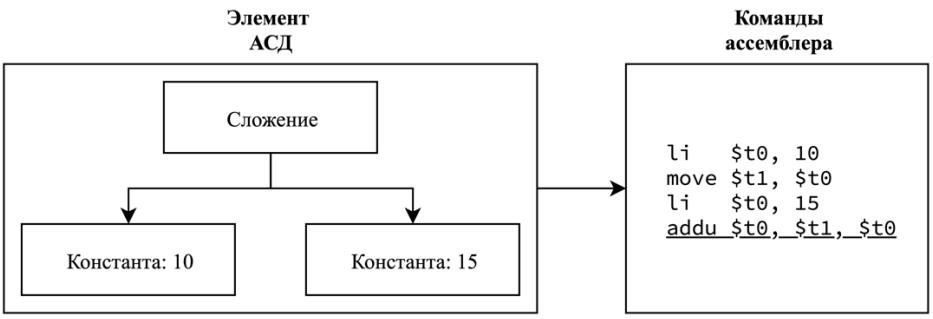

Рисунок 5.5 — Сложение двух констант

Сложение двух констант происходит в четыре команды: сначала загружается константа слева от оператора сложения, затем значение этой константы перемещается во временный регистр *$t1*, далее загружается значение константы справа от оператора, вслед за этим выполняется команда сложения и результат записывается в регистр *$t0*.

```
BinaryExpression *binary_syntax = syntax->binary_expression;
ctx->stack_offset += WORD_SIZE;
write_syntax(out, binary_syntax->left, ctx);
emit_instr(out, current_indent, "move", "$t1, $t0");
write_syntax(out, binary_syntax->right, ctx);
emit_instr_format(out, current_indent, "addu", "$t0, $t1, $t0");
```

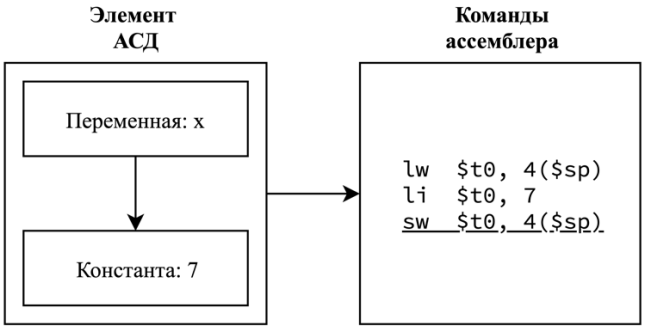

Рисунок 5.6 — Определение переменной

Для определения переменной необходимо получить свободное смещение от вершины текущего стекового фрейма. Это смещение хранится в поле *stack_offset* структуры *SymbolTable*, описанной в разделе 5.1.2. После того, как смещение получено, происходит порождение команд: сначала значение по адресу смещения загружается в регистр *$t0*, затем в этот регистр загружается значение константы, после чего значение регистра сохраняется обратно по адресу смещения.

```
DefineVarStatement *define_var_statement = syntax->define_var_statement;
int stack_offset = ctx->stack_offset;
symbol_table_set_offset(ctx->s_table, define_var_statement->var_name, stack_offset);
emit_instr_format(out, current_indent, "lw", "$t0, %d($sp)", stack_offset);
ctx->stack_offset += WORD_SIZE;
```

write_syntax(out, define_var_statement->init_value, ctx);
emit_instr_format(out, current_indent, "sw", "$t0, %d($sp)", stack_offset);

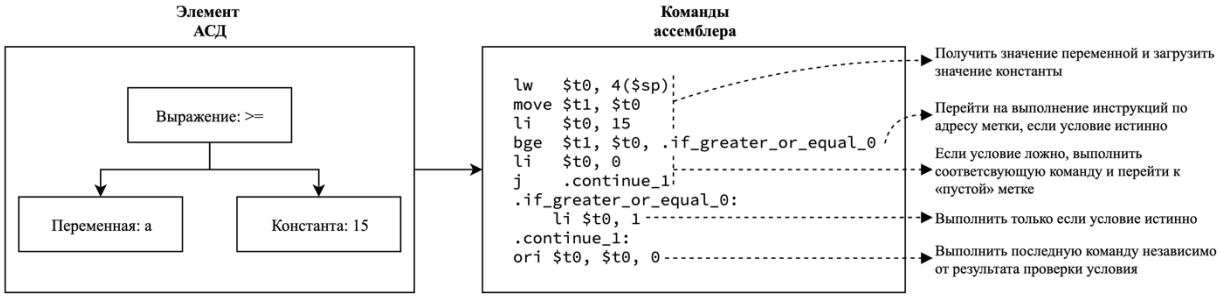

Рисунок 5.7 — Бинарное выражение «больше или равно»

В бинарном выражении «больше или равно» как и при сложении, первым этапом является получение значений переменной и загрузка значения константы. Затем выполняется команда, переходящая на выполнение инструкций по адресу «истинной» метки, если условие истинно, и в регистр *$t0* загружается значение *1*, означающее истинность условия. Если же условие ложно, в регистр *$t0* загружается значение *0* и выполняется команда, переходящая на «пустую» метку. «Пустой» меткой является метка, не имеющих никаких команд по своему адресу. Функция такой метки — «перепрыгнуть» команды, исполняемые при истинности условия.

```
char *true_label = fresh_local_label("if_greater_or_equal", ctx);
char *false_label = fresh_local_label("continue", ctx);
emit_instr_format(out, current_indent, "bge", "$t1, $t0, %s", true_label);
emit_instr_format(out, current_indent, "li", "$t0, 0");
emit_instr_format(out, current_indent, "j", "%s",false_label);
emit_label(out, current_indent, true_label);
emit_instr_format(out, current_indent, "li", "$t0, 1");
decrease_indent();
emit_label(out, current_indent, false_label);
decrease_indent();
emit_instr_format(out, current_indent, "ori", "$t0, $t0, 0");
```

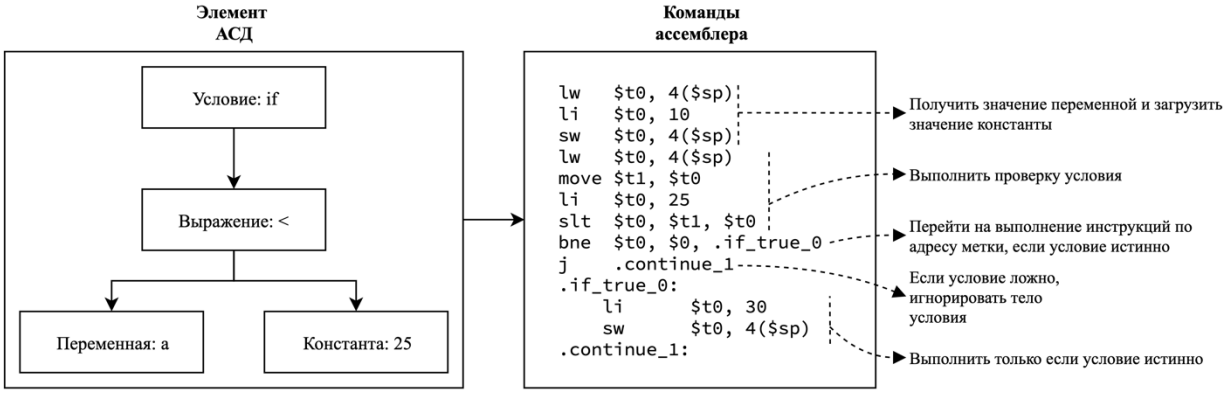

Рисунок 5.8 — Условный оператор

Условный оператор устроен подобно бинарным выражениям. В первую очередь инициализируются переменные и константы, затем выполняется проверка условия. Результат этой проверки записывается в регистр *$t0*. Если значение в этом регистре не равно нулю, то выполняются инструкции по адресу «истинной» метки. Эти инструкции являются «телом» условного оператора. Если значение в регистре равно нулю, то есть условие ложно, выполняется переход по адресу «пустой» метки, находящейся ниже «тела» оператора.

```
char *true_label = fresh_local_label("if_true", ctx);
char *false_label = fresh_local_label("continue", ctx);
IfStatement *if_statement = syntax->if_statement;
write_syntax(out, if_statement->condition, ctx);
emit_instr_format(out, current_indent, "bne", "$t0, $0, %s", true_label);
emit_instr_format(out, current_indent, "j", "%s", false_label);
emit_label(out, current_indent, true_label);
write_syntax(out, if_statement->then, ctx);
emit_label(out, current_indent, false_label);
decrease_indent();
```

6 Примеры реализации

Таблица 6.1 — Примеры реализации

Исходный код на языке *C*	Код на языке ассемблера

int main()
{
  int a = 5;
  if (a < 10)
  {
  a = a + 1;
  }
  return a;
}
	j main
	
	main:
	  lw $t0, 4($sp)
	  li $t0, 5
	  sw $t0, 4($sp)
	  lw $t0, 4($sp)
	  move $t1, $t0
	  li $t0, 10
	  slt $t0, $t1, $t0
	  bne $t0, $0, .if_true_0
	  j .continue_1
	  .if_true_0:
	  lw $t0, 4($sp)
	  sw $t0, 4($sp)
	  move $t1, $t0
	  li $t0, 1
	  addu $t0, $t1, $t0
	  .continue_1:
	  lw $t0, 4($sp)
	  jr $ra

Продолжение таблицы 6.1

int main()
{
  int a = 30;
  int b = 25;
  while (a >= b)
  {
  a = a - 1;
  }
  return a;
}
	j main
	
	main:
	  lw $t0, 4($sp)
	  li $t0, 30
	  sw $t0, 4($sp)
	  lw $t0, 8($sp)
	  li $t0, 25
	  sw $t0, 8($sp)
	  lw $t0, 4($sp)
	  move $t1, $t0
	  lw $t0, 8($sp)
	  bge $t1, $t0, .if_greater_or_equal_2
	  li $t0, 0
	  j .continue_3
	  .if_greater_or_equal_2:
	  li $t0, 1
	  .continue_3:
	  ori $t0, $t0, 0
	  bne $t0, $0, .while_true_0
	  j .continue_1
	  .while_true_0:
	  lw $t0, 4($sp)
	  sw $t0, 4($sp)
	  move $t1, $t0
	  li $t0, 1
	  subu $t0, $t1, $t0
	  lw $t0, 4($sp)
	  move $t1, $t0
	  lw $t0, 8($sp)
	  bge $t1, $t0, .if_greater_or_equal_4
	  li $t0, 0
	  j .continue_5
	  .if_greater_or_equal_4:
	  li $t0, 1
	  .continue_5:
	  ori $t0, $t0, 0
	  bne $t0, $0, .while_true_0
	  .continue_1:
	  lw $t0, 4($sp)
	  jr $ra

Окончание таблицы 6.1

int test()
{
  return 5;
}
 	
int main()
{
  int a = 65;
  test();
  return a;
}
	j main
	
	test:
	  li $t0, 5
	  jr $ra
	
	main:
	  lw $t0, 4($sp)
	  li $t0, 65
	  sw $t0, 4($sp)
	  jal test
	  lw $t0, 4($sp)
	  jr $ra

Заключение

В результате курсовой работы были изучены средства и инструменты лексического и синтаксического анализа, а также методы генерации ассемблерного кода. Реализован компилятор с подмножества языка *C* на язык ассемблера для процессора lilpM32.

Дальнейшая эволюция компилятора связана с расширением функциональности.

Список литературы

1. Ахо А., Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструменты: Пер. с англ. — М.: Издательский дом «Вильямс», 2003. — 768 с.

2. Лилов И.П. «Организация ЭВМ. Лабораторные работы в программе Logisim» — Москва, 2012.

3. С. Свердлов. Конструирование компиляторов. Учебное пособие. - LAP LAMBERT Academing Publishing, 2015 – 571 с.

4. Фельдман Ф.К. Системное программирование на персональном компьютере. – 2004. – 512 с.: ил.

5. Кормен, Томас Х. Алгоритмы: вводный курс.: Пер. с англ. – М.: ООО «И.Д. Вильямс», 2017. – 208 с.: ил.