2-й семестр. Домашние задания.

Бабичев С. Л.

25 февраля 2022 г.

Содержание

1 Архитектура компьютера MIPT

MIPT — машина с архитектурой Фон-Неймана с адресным пространством в 2^{20} слов, каждое из которых занимает 32 бита.

Каждая команда занимает ровно одно слово, 8 старших бит которого — код операции, а использование остальных 24 битов зависит от операции.

Компьютер оснащён шестнадцатью однословными (по 32 бита) регистрами r0-r15, их назначение приведено в таблице 1.

Таблица 1: Распределение регистров процессора MIPT

r0-r12	свободно используются
r13	указатель фрейма вызова
r14	указатель стека
r15	счётчик команд
flags	результат операции сравнения

В зависимости от кода операции каждая команда может быть одного из следующих форматов:

- RM — 8 старших бит код команды, 4 следующих бита — код регистра (приёмника или источника), 20 младших бит — адрес в памяти в виде беззнакового числа от 0 до $2^{20} - 1$. Пример такой команды:

  ```
  load r0, 12323
  ```

- RR — 8 бит код команды, 4 бит код регистра-приёмника, 4 бит код регистра-источника, 16 бит модификатор источника, число со знаком от -2^{15} до $2^{15} - 1$. Примеры такой команды:

  ```
  mov r1, r2, -123
  ```

  ```
  add r1, r2, 0
  ```

  В командах такого типа непосредственный операнд всегда прибавляется к значению регистра-источника. Если этого делать не требуется, в поле непосредственного операнда заносится 0.

- RI — 8 бит код команды, 4 бит код регистра-приёмника, 20 бит непосредственный операнд, число со знаком от -2^{19} до 2^{19}. Пример такой команды:

  ```
  ori r2, 64
  ```

- J — 8 бит код команды, 4 бита игнорируются, 20 младших бит — адрес в памяти в виде беззнакового числа от 0 до $2^{20} - 1$. Пример такой команды:

  ```
  calli 3121
  ```

Для простоты написания программы-ассемблера все аргументы в команде обязательны.

```
; Программа, которая вводит число с клавиатуры, прибавляет к нему 1 и
;  выводит результат.
   syscall r0, 100 ; запрос числа в регистр r0
   addi r0, 1      ; r0++
   syscall r0, 102 ; вывод содержимого r0
   lc r1, 10       ; загрузка кода '\n'
   syscall r1, 105 ; вывод символа
   syscall r1, 0   ; выход
```

2 Описание процессора MIPT

2.1 Система команд процессора MIPT

Таблица 2: Описание машинных команд процессора M

Код	Имя	Формат	Описание
0	halt	RI	Останов процессора. `halt r1 0`
1	syscall	RI	Вызов операционной системы. `syscall r0, 100`
2	add	RR	Сложение регистров. К R_1 (регистру-первому аргументу) прибавляется содержимое R_2 (регистра второго аргумента), модифицированное непосредственным операндом. `add r1, r2, 3` К регистру `r1` прибавляется `r2+3`.
3	addi	RI	Сложение регистра с непосредственным операндом. К содержимому R_1 прибавляется значение Imm (непосредственного операнда). `addi r4, 10`
4	sub	RR	Вычитание регистров. Из R_1 вычитается содержимое R_2. `sub r3, r5, 5` Из регистра `r3` вычитается `r5+5`.
5	subi	RI	Вычитание из регистра непосредственного операнда. `subi r4 1`
6	mul	RR	Умножение регистров. Содержимого R_1 умножается на R_2. Результат помещается в пару регистров, начинающуюся с R_1. `mul r3, r10, 0` Результат произведения `r3` и `r10` будет помещён в `r3` и `r4`.

Продолжение на следующей странице

7	`muli`	`RI`	Умножение регистра R_1 на Imm. Результат помещается в пару регистров, начинающуюся с R_1. `muli r5, 100` После исполнения: в регистре `r5` младшие 32 разряда результата, в регистре `r6` — старшие 32 разряда.
8	`div`	`RR`	Деление регистров. Младшие 32 бита первого операнда находятся в регистре R_1, старшие — в регистре, номер которого на 1 больше R_1. Второй операнд находится в регистре. Частное помещается в регистр R_1, остаток — в следующий регистр. `div r3, r10, 0` `r3` будет содержать частное от деления пары `(r3,r4)` на `r10`, `r4` — остаток от этого деления.
9	`divi`	`RI`	Деление регистра на непосредственный операнд. Аналогично `div`. `div r3 10` `r3` будет содержать частное от деления пары `(r3,r4)` на `10`, `r4` — остаток от этого деления.
12	`lc`	`RI`	Загрузка константы Imm в регистр R_1. Для загрузки констант, бóльших 2^{20} требуются дополнительные команды сдвига и логического сложения. `lc r7, 123` `r7` будет содержать `123`.
13	`shl`	`RR`	Сдвинуть биты в регистре R_1 влево на значение регистра R_2. `shl r1, r2, 0`
14	`shli`	`RI`	Сдвинуть биты в регистре R_1 влево на Imm. `shli r1, 2`
15	`shr`	`RR`	Сдвинуть биты в регистре R_1 вправо на значение регистра R_2. Сдвижка на 32 или более разрядов обнуляет регистр R_1. `shr r1, r2, 0`
16	`shri`	`RI`	Сдвинуть биты в регистре R_1 впрево на Imm. `shri r1, 2`

Продолжение на следующей странице

17	`and`	`RR`	Логическое И регистров R_1 и R_2. Результат — в регистре R_1. `and r1 r2 0`
18	`andi`	`RI`	Логическое И над регистром R_1 и Imm. `andi r1, 255`
19	`or`	`RR`	Логическое ИЛИ регистров R_1 и R_2. Результат — в регистре R_1. `or r1, r2, 0`
20	`ori`	`RI`	Логическое ИЛИ над регистром R_1 и Imm. `ori r1, 255`
21	`xor`	`RR`	Логическое исключающее ИЛИ регистров R_1 и R_2. Результат — в регистре R_1. `xor r1 r2 0`
22	`xori`	`RI`	Логическое исключающее ИЛИ над регистром R_1 и Imm. `xori r1 255`
23	`not`	`RI`	Поразрядное НЕ над всеми битами R_1. Непосредственный операнд игнорируется, но он присутствует в команде для простоты компилятора. `not r1, 0`
24	`mov`	`RR`	Пересылка из регистра R_2 в регистр R_1. `mov r0, r3, 22` В регистр `r0` помещается значение `r3+22`.
32	`addd`	`RR`	Вещественное сложение регистров R_0 и R_1. Вещественные числа занимают пару регистров, младшие из которых фигурируют в коде операции. `addd r2, r5, 0` К вещественному числу, занимающему пару регистров `(r2,r3)` прибавляется значение вещественного числа из пары регистров `(r5,r6)`.
33	`subd`	`RR`	Вещественное вычитание регистров R_0 и R_1. `subd r2, r5, 0` Из вещественного числа, занимающему пару регистров `(r2,r3)` вычитается значение вещественного числа из пары регистров `(r5,r6)`.

Продолжение на следующей странице

34	`muld`	`RR`	Вещественное умножение регистров R_0 и R_1. `muld r2, r5, 0` Вещественное число, занимающее пару регистров (`r2,r3`) умножается на значение вещественного числа из пары регистров (`r5,r6`).
35	`divd`	`RR`	Вещественное деление регистров R_0 и R_1. `subd r2, r5, 0` Вещественное число, занимающее пару регистров (`r2,r3`) делится на значение вещественного числа из пары регистров (`r5,r6`).
36	`itod`	`RR`	Преобразование целого числа R_2 в вещественное R_1 `itod r2, r5, 0` Значение, находящееся в регистре `r5` преобразуется в вещественное и помещается в пару регистров (`r2,r3`).
37	`dtoi`	`RR`	Преобразование вещественного числа R_2 в целое R_1 `itod r2, r5, 0` Вещественное число, занимающее пару регистров (`r5,r6`) преобразуется в целое отбрасыванием дробной части. Если число не помещается в регистр, возникает ошибка.
38	`push`	`RI`	Отправить значение, находящееся в регистре R_1 с добавлением к нему Imm в стек. `push r0, 255` Поместить в стек число, равное `r0+255`. Указатель стека (`r14`) уменьшится на `1`.

39	`pop`	`RI`	Извлечь из стека находящееся там число, поместить его в регистр R_1 и затем прибавить к регистру R_1 значение Imm.
			`push r2, 0`
			`pop r3, 1`
			После исполнения в `r3` будет содержаться `r2+1`.

Продолжение на следующей странице

40	`call`	`RR`	Вызвать функцию, адрес которой расположен в $R_2 + Imm$. `call r0, r5, 0` Вызвать функцию, адрес которой извлекается из регистра `r5`. При вызове функции в стек помещается адрес команды, следующей за текущей и управление передаётся по указанному адресу. Этот же адрес помещается и в регистр `r0`.
41	`calli`	`J`	Вызвать функцию, адрес которой расположен в $imm20$. `call 13323` Вызвать функцию, начинающуюся с оператора по адресу `13323`. При вызове функции в стек помещается адрес команды, следующей за текущей и управление передаётся по указанному адресу.
42	`ret`	`RI`	Возврат из функции. `ret r0, 0` Вернуться из вызванной функции в вызвавшую. Из стека извлекается адрес, по которому передаётся управление и который будет адресом следующего исполняемого слова. Непосредственный аргумент показывает количество дополнительных слов, на которое требуется продвинуть указатель стека (оно должно соответствовать количеству аргументов при вызове функции).
43	`cmp`	`RR`	Сравнение. `cmp r0, r1, 0` Сравнить `r0` с `r1`. Результат записывается в регистр флагов, который используется в командах перехода.
44	`cmpi`	`RI`	Сравнение с константой. `cmpi r0, 0` Сравнить `r0` с нулём. Установить соответствующий флаг.

Продолжение на следующей странице

45	`cmpd`	`RR`	Сравнение вещественных чисел. `cmpd r1, r4, 0` Сравнить пару `(r1,r2)` с `(r4,r5)`. Установить соответствующий флаг.
46	`jmp`	`J`	Безусловный переход. `jmp 2212` Следующая исполняемая команда будет находиться по адресу 2212.
47	`jne`	`J`	Переход при наличии условия `!=`. `jne 2212` Если регистр флагов содержит условие `!=`, то следующая исполняемая команда будет находиться по адресу 2212 иначе ход исполнения программы не нарушается.
48	`jeq`	`J`	Переход при наличии условия `==`. `jne 2212` Если регистр флагов содержит условие `==`, то следующая исполняемая команда будет находиться по адресу 2212 иначе ход исполнения программы не нарушается.
49	`jle`	`J`	Переход при наличии условия `<=`. `jle 2212` Если регистр флагов содержит условие `<=`, то следующая исполняемая команда будет находиться по адресу 2212 иначе ход исполнения программы не нарушается.
50	`jl`	`J`	Переход при наличии условия `<`. `jl 2212` Если регистр флагов содержит условие `<`, то следующая исполняемая команда будет находиться по адресу 2212 иначе ход исполнения программы не нарушается.

Продолжение на следующей странице

51	jge	J	Переход при наличии условия >=. `jne 2212` Если регистр флагов содержит условие >=, то следующая исполняемая команда будет находиться по адресу 2212 иначе ход исполнения программы не нарушается.
52	jg	J	Переход при наличии условия >. `jg 2212` Если регистр флагов содержит условие >, то следующая исполняемая команда будет находиться по адресу 2212 иначе ход исполнения программы не нарушается.
64	load	RM	Загрузка из памяти в регистр `load r0, 12345` Содержимое из ячейки памяти по адресу 12345 копируется в регистр r0.
65	store	RM	Выгрузка из регистра в память `store r0, 12344` Содержимое из регистра r0 копируется в ячейку памяти по адресу 12344.
66	load2	RM	Загрузка из памяти в пару регистров `load r0, 12345` Содержимое из ячеек памяти по адресу 12345 и 12346 копируется в регистры r0 и r1.
67	store2	RM	Выгрузка из пары регистров в память `load r0, 12345` Содержимое из регистра r0 копируется по адресу 12345, а из ячейки r1 — по адресу 12346.
68	loadr	RR	Загрузка из памяти в регистр `load r0, r1, 15` Содержимое из ячейки памяти по адресу (r1+15) копируется в регистр r0.
69	loadr2	RR	Загрузка из памяти в пару регистров `load r0, r13, 7` Содержимое из ячеек памяти по адресу r13+7 и r13+8 копируется в регистры r0 и r1.

Продолжение на следующей странице

70	`storer`	`RR`	Выгрузка из регистра в память `store r0, r13, 3` Содержимое из регистра `r0` копируется в ячейку памяти по адресу `r13+3`.
71	`storer2`	`RR`	Выгрузка из пары регистров в память `load r0, r13, 11` Содержимое из регистра `r0` копируется по адресу `r13+11`, а из ячейки `r1` — по адресу `r13+12`.

Таблица 3: Коды системных вызовов операционной системы FUMPOS

Код	Аргументы	Описание
0	EXIT	код возврата.
100	SCANINT	
101	SCANDOUBLE	
102	PRINTINT	
103	PRINTDOUBLE	
104	GETCHAR	
105	PUTCHAR	C

Для удобства все коды команд собраны в перечислимый тип code:

```
enum code {
    HALT = 0,
    SYSCALL = 1,
    ADD = 2,
    ADDI = 3,
    SUB = 4,
    SUBI = 5,
    MUL = 6,
    MULI = 7,
    DIV = 8,
    DIVI = 9,
    LC = 12,
    SHL = 13,
    SHLI = 14,
    SHR = 15,
    SHRI = 16,
    AND = 17,
    ANDI = 18,
    OR = 19,
    ORI = 20,
    XOR = 21,
    XORI = 22,
    NOT = 23,
    MOV = 24,
    ADDD = 32,
    SUBD = 33,
    MULD = 34,
    DIVD = 35,
    ITOD = 36,
    DTOI = 37,
```

```
    PUSH = 38,
    POP = 39,
    CALL = 40,
    CALLI = 41,
    RET = 42,
    CMP = 43,
    CMPI = 44,
    CMPD = 45,
    JMP = 46,
    JNE = 47,
    JEQ = 48,
    JLE = 49,
    JL = 50,
    JGE = 51,
    JG = 52,
    LOAD = 64,
    STORE = 65,
    LOAD2 = 66,
    STORE2 = 67,
    LOADR = 68,
    LOADR2 = 69,
    STORER = 70,
    STORER2 = 71
};
```

2.2 Особенности некоторых команд

2.2.1 Арифметические и логические команды

Арифметические и логические команды имеют два варианта. Первый вариант имеет тип `RR` (регистр-регистр). Регистром-приёмником результата всегда является первый регистр в команде, часто он же есть и один из участвующих в операции регистров. Например в команде

```
add r4, r5, 111
```

регистр `r4` есть один из операндов операции сложения, а полностью результатом операции будет изменение значения регистра `r4`, который станет равен старому значению `r4` плюс сумма значения регистра `r5` и непосредственного, то есть содержащегося в коде операции операнда `111`.

```
r4 = r4 + r5 + 111;
```

Аналогично и в других операциях:

```
and r7, r9, 13
```

означает

```
r7 = r7 & (r9 + 13)
```

Длина непосредственного операнда — 16 битов, могут присутствовать и положительные и отрицательные значения ($-32768 \ldots 32767$).

В другом варианте команды, формата `RI` (регистр-непосредственное значение), вместо второго регистра используется 20-битный непосредственный операнд *со знаком*.

```
addi r1, 1
```

есть увеличение значения регистра `r1` на `1`, а

```
ori r5, 8
```

есть установка бита `4` в регистре `r5`.

2.2.2 Команды умножения и деления

В команде умножения формата `RR` имеется два операнда, указанные в команде (второй операнд, как и в простых командах, модифицируется непосредственным операндом). Результатом операции умножения является пара соседних регистров, первый из которых есть регистр-операнд 1. Например:

```
mul r4, r7, 1
```

Умножаются два 32-битных значения — `r4` и `r7+1`. Получившийся 64-битный результат укладывается в 2 регистра: в `r4` будут содержаться младшие 32 бита, в `r5` — старшие 32 бита.

Такое расположение регистров — исходное для команды деления. В ней всегда пара регистров, образующая 64-битное число, делится на второй регистр (или непосредственный операнд). Результат деления тоже размещается в двух регистрах — в первом регистры пары оказывается частное, во втором — остаток.

Пример. Пусть в регистре `r4` содержится 123, в регистре `r5` — 0, а в регистре `r7` — 100. Тогда после

```
div r4, r7, 0
```

регистры будут содержать следующее: `r4=1`, `r5=23`.

Если при делении результат не помещается в 32 бита, возникает исключительная ситуация деления на 0 и исполнение программы прекращается.

2.2.3 Работа с памятью

Наш компьютер — типичный представитель RISC-архитектуры, то есть он содержит сокращённый набор команд и у него нет команд, в которых производятся операции над ячейками памяти. Всегда требуется сначала загрузить из памяти нечто в процессор (его регистры) и в конце концов отправить назад в память, если это будет нужно. Память компьютера состоит из 2^{20} 32-битных *слов*. Для того, чтобы изменить какую-либо ячейку памяти, требуется *загрузить* её в регистр процессора, произвести регистровую операцию и затем *выгрузить* регистр в память. Например, для увеличения содержимого ячейки памяти с адресом 1234 на 1 можно исполнить следующие команды:

```
load r1, 1234
addi r1, 1
store r1, 1234
```

Имеются двухсловные варианты этих команд.

Если адрес ячейки памяти содержится в регистре, то можно использовать команды `loadr` и `storer`. Пусть регистр `r13` содержит адрес начала локальных переменных. Тогда увеличение 3-й переменной (то есть, переменной, отстоящей на 2 ячейки от начала) на 10 можно записать так:

```
loadr r0, r13, 2
addi r0, 10
storer r0, r13, 2
```

И эти команды имеют двухрегистровый вариант.

2.2.4 Вещественная арифметика

Любая соседняя пара регистров может использоваться как единичный регистр, содержащий вещественное число. Имеется 4 арифметические команды для работы с этими регистрами и одна команда сравнения. Вещественные числа представляются в следующем формате:

бит 63 — знак числа `s`.

биты 52-62 — значение степени `n`, увеличенное на 1023.

биты 51-0 — двоичное представление мантиссы с подразумеваемой точкой слева от всех разрядов и приписанной единицей (подробнее о формате вещественного числа — на лекциях и семинарах).

Для того, чтобы производить операции с вещественными числами требуется загрузить их в регистры, например, с помощью команды `load2`.

```
load2, r2, pi
mov r4, r2, 0
mov r5, r3, 0
```

```
    muld r2, r4
    store2 r2, pi2
    ...
    ...
pi: double 3.1415927
e:  double 2.718281828
    ...
```

2.2.5 Команды сравнения и переходов

Существует три команды сравнения: `cmp`, `cmpi` и `cmpd`. Эти команды производят виртуальную операцию вычитания второго операнда из первого и устанавливают регистр флагов сравнения в соответствующее состояние. Никакие другие команды на регистр флагов сравнения влияния не оказывают. Имеется 6 команд условного перехода: `jeq`, `jne`, `jge`, `jg`, `jle`, `jl`, которые в зависимости от состояния регистра флагов перехода могут произвести переход на заданное место в программе.

```
    cmpi r4, 10
    jne  skip1
    lc r5, 20
    jmp done
skip1:
    lc r5 25
done:
```

Данный фрагмент программы сравнивает значение регистра `r4` с константой 10, в случае их неравенства происходит переход на метку `skip1`. В случае равенства происходит присваивание регистру `r5` значения 20, после чего происходит безусловный переход на метку `done`.

2.2.6 Работа со стеком

Имеются две явные операции со стеком — `push` и `pop`. Первая команда сначала уменьшает регистр `r14` на единицу, а затем помещает операнд (возможно, модифицированный непосредственной частью операнда) в ячейку, адресуемую регистром `r14`.

Вторая команда сначала извлекает значение по этому адресу, прибавляет непосредственный операнд и помещает на место второго операнда и затем увеличивает регистр `r14` на 1.

2.2.7 Вызовы функций

Вызов функции происходит при команде `call`. Регистр `r14` уменьшается на единицу и по этому адресу помещается адрес следующей исполнимой инструк-

ции. Затем управление передаётся по адресу вызываемой функции, он или в регистре-аргументе, или в непосредственном операнде. В функцию могут передаваться аргументы. Как именно — определяет программист. Например, их можно передавать через стек (можно и через регистры). Если функция рекурсивная прямо или косвенно, то наилучший способ — воспользоваться стеком.

Возврат из функции происходит при исполнении команды `ret`. Непосредственный аргумент команды определяет, сколько слов требуется снять со стека перед тем, как извлечь из него адрес возврата. Детали вызова функции — в приведённом чуть ниже примере программы, вычисляющей факториал числа.

2.2.8 Системные вызовы

Программа, не содержащая системных вызовов, не способна взаимодействовать с окружающей средой. На первое время нам достаточно нескольких системных вызовов, вводящих и выводящих числа или символы.

Существует единственная команда `syscall` для выдачи системного вызова. Первый аргумент (регистр) содержит передаваемые системе данные (или принимаемые от системы данные), второй аргумент (непосредственный операнд) — код вызова. Второй и остальные аргументы системного вызова передаются в стеке.

2.2.9 Метки

Перед любой командой или в отдельной строке можно поместить **метку** — слово, состоящее из букв латинского алфавита и цифр, не начинающееся с цифры. Её можно использовать далее, как более удобную запись для того адреса, где она располагается. Использование метки, которая нигде не определена — ошибка, которая приводит к невозможности создать исполнимую программу. Обратиться к метке, находящейся далее по коду программы допустимо. Метки не должны повторяться и не должны совпадать с другими словами в программе.

2.2.10 Остальные директивы

Директива `word` служит для резервирования ячейки памяти с заданным значением.

Пример:

```
one: word 1
```

Директива `double` служит для резервирования ячейки памяти с заданным значением.

Пример:

```
pi: double 3.141592653589793
```

Директива `end` должна быть последней строкой программы. Её аргумент — адрес первой исполняемой ячейки программы.

Пример:

```
    ...
    ...
main:
    ...
    end main
```

3 Формат исполнимого файла MIPTOS

Все программы для процессора MIPT исполняются под управлением операционной системы MIPTOS. Для запуска программы требуется сформировать исполнимый файл, в котором присутствует метаинформация (описание секций) и сами секции. Программа хранится во внешней памяти (скажем, на жёстком диске или SDD) в виде набора байтов. Слово состоит из 4-х байтов в следующем порядке: 3210, то есть, в формате LSB. Заголовок исполнимого файла содержит ровно 512 байт, содержимое которых описано ниже. начиная с 512 байта размещаются последовательно секции кода, констант и данных, они загружаются в виртуальную память начиная с адреса 0. Исполнение программы начинается с адреса, записанного в заголовке исполнимого файла, а начальное значение стека определяется соответствующим полем в заголовке.

Формат исполнимого файла приведён в таблице 3.

Таблица 4: формат исполнимого файла MIPTOS

Байты	Содержимое
0..15	Строка ASCII "ThisIsMIPT2Exec"
16..19	Размер кода программы
20..23	Размер констант программы
24..27	Размер данных программы
28..31	Начальный адрес исполнения программы
32..35	Начальный адрес стека
36..39	Идентификатор целевого процессора
512..	Сегмент кода
	Сегмент констант
	Сегмент данных

Для операций с двоичными файлами используйте функции fopen, fread, fwrite, fseek, fclose.

4 Примеры программ с комментариями

4.1 Простая программа возведения числа в квадрат

```
main:
    syscall r0, 100  ;ввод числа со стандартного ввода в регистр r0
    mov r2, r0, 0     ;копирование регистра r0 в регистр r2
    mul r0, r2, 0     ;пара регистров (r0,r1) содержит произведение
    syscall r0, 102  ;вывод содержимого регистра r0 (младшая часть)
    lc r0, 10        ;загрузка константы 10 ('\n') в регистр r0
    syscall r0, 105  ;вывод '\n'
    lc r0, 0         ;
    syscall r0, 0    ;выход из программы с кодом 0
    end main        ;начать исполнение с main
```

4.2 Программа возведения числа в квадрат, использующая функции

```
sqr:                ;функция sqr с одним аргументом в стеке
    loadr r0, r14, 1  ;загрузка r0 ячейки с первым аргументом
    mov r2, r0, 0     ;копируем r0 в r2
    mul r0, r2, 0     ;(r0,r1) = r0*r2
    ret 1           ;возвращаемся из функции, убрав аргумент из стека
intout:             ;функция, распечатывающая аргумент + '\n'
    load r0, r14, 1   ;загрузка первого аргумента в r0
    syscall r0, 102  ;вывод r0 на экран
    lc r0, 10         ;загрузка '\n'
    syscall r0, 105  ;вывод '\n'
    ret 1           ;возврат
main:
    syscall r0, 100  ;считывание в r0
    push r0, 0        ;помещение r0+0 в стек
    calli sqr       ;вызов функции sqr. В r0 функция оставит результат.
    push r0, 0        ;передаём результат в  фунцкию intout
    calli intout    ;и вызываем её
    lc r0, 0
    syscall r0, 0    ;exit(0)
    end main
```

4.3 Программа, использующая рекурсивную функцию факториала

```
fact:
    loadr r0, r14, 1
    cmpi r0, 1
    jg skip0
    lc r0, 1
    ret 1
skip0:
    push r0, 0
    subi r0, 1
    push r0, 0
    calli fact
    pop r2, 0
    mul r0, r2, 0
    ret 1
main:
    syscall r0, 100
    push r0, 0
    calli fact
    syscall r0, 102
    lc r0, 10
    syscall r0, 105
    lc r0, 0
    syscall r0, 0
    end main
```

4.4 Программа, вводящая два вещественных числа и печатающая их сумму, разность, произведение и частное

```
fout:
    syscall r0, 103
    lc r0, 10
    syscall r0, 105
    ret 0
main:
    syscall r2, 101
    syscall r4, 101

    mov r0, r2, 0
    mov r1, r3, 0
    addd r0, r4, 0
    calli fout
    mov r0, r2, 0
    mov r1, r3, 0
    subd r0, r4, 0
    calli fout
    mov r0, r2, 0
    mov r1, r3, 0
    muld r0, r4, 0
    calli fout
    mov r0, r2, 0
    mov r1, r3, 0
    divd r0, r4, 0
    calli fout
    lc r0, 100
    itod r0, r0, 0
    calli fout
    lc r0, 0
    syscall r0, 0
end main
```

4.5 Программа, вычисляющая перебором число счастливых билетов

```
start: lc r6, 0     ;counter
    lc r0, 0     ;first digit i1
l0: cmpi r0, 10  ; for (i1 = 0; i1 < 10; i++) {
    jge e0
    lc r1, 0
l1: cmpi r1, 10
    jge e1
    lc r2, 0
l2: cmpi r2, 10
    jge e2
    lc r3, 0
l3: cmpi r3, 10
    jge e3
    lc r4, 0
l4: cmpi r4, 10
    jge e4
    lc r5, 0
l5: cmpi r5, 10
    jge e5
    mov r7, r0, 0   ;r7 = r0+r1+r2
    add r7, r1, 0
    add r7, r2, 0
    mov r8, r3, 0   ;r8 = r3+r4+r5
    add r8, r4, 0
    add r8, r5, 0
    cmp r7, r8, 0
    jne skip
    addi r6, 1     ;counter++
skip: addi r5, 1
    jmp l5
e5: addi r4, 1
    jmp l4
e4: addi r3, 1
    jmp l3
e3: addi r2, 1
    jmp l2
e2: addi r1, 1
    jmp l1
e1: addi r0, 1
    jmp l0
```

```
e0: syscall r6, 102 ; print counter, '\n'
    lc r0, 10
    syscall r0, 105
    lc r0, 0
    syscall r0, 0   ;exit(0)
    end start
```

5 Собственно домашнее задание

Задание состоит из следующих задач:

- Написать программу, переводящую текст на языке ассемблера `MIPT` в исполнимый на эмуляторе машины `MIPT` файл.
- Написать программу-эмулятор машины `MIPT` в операционной системе `MIPTOS`, считывающую исполнимый файл и исполняющую машинные команды процессора `MIPT`, содержащиеся в этом файле.
- Написать программу, считывающую исполнимый файл формата `MIPTOS` и выводящую на экран ассемблерный текст, соответствующий этому файлу.
- Написать программы на ассемблере `MIPT` для алгоритмов, которые будут индивидуально выдаваться каждому.

Описание машины `MIPT` будет уточняться для того, чтобы избежать неопределённостей и двусмысленностей. Примеры программ на языке ассемблера и исполнимые файлы находятся на сайте.

Ваши программы — ассемблер, дизассемблер и эмулятор, сдаются в контесте. Последний срок сдачи всех задач — 12 часа 19 минут **10 мая**. Начинайте решение как можно раньше, не бейтесь за баллы, чтобы мы могли обсудить правильность избранного направления.

У сдавшего с первой попытки задание принято не будет.

После **10 мая** задание приниматься тоже не будет.

За коллективное творчество баллы за задачу будут поровну делиться между всеми участниками коллектива.

Q&A

Q. Может ли функция вызывать функции, описанные после неё?

A. Да. Поэтому при ассемблировании требуется два прохода.

Q. Является ли `end main` концом программы?

A. Да. После `end` стоит адрес, с которого должно начаться исполнение.

Q. Должны ли мы строго следить за синтаксическими ошибками и выводить гневные слова, если они совершены?

A. Почему нет? Для себя, конечно. В тестах их нет.

Q. Должны ли мы делать проверку деления на 0?

A. Да, программа должна аварийно завершиться. В тестах такого нет, так как такое поведение не определено.

Q. Можем ли мы при написании кода пользоваться C++ деревьями и векторами?

A. Конечно, это очень даже стоит делать. `map/string/vector`

Q. Считать ли всю память зануленной перед выполнением программы, или же там может лежать мусор?

A. Считать занулённой.

Q. Сама программа (машинный код) лежит внутри адресного пространства?

A. Так ведь архитектура фон Неймана же.

Q. Можно ли считать, что заголовок файла находится в нулевой ячейке.

A. Нет, заголовок не входит в адресное пространство.

Q. Переход по меткам осуществляется внутри относительной адресации 32-битных слов файла?

A. После загрузки исполнимого файла в память — по абсолютным адресам в памяти.

Q. Программа может в процессе исполнения изменить свой код?

A. Да, у нас запись во все слова возможна, в том числе в слова с кодом.

Q. Стек находится в адресуемой памяти, или это отдельное адресное пространство? Если да, то какой у него размер?

A. Архитектура всё ещё фон Неймана. В общей памяти. Неограниченный.

Q. Стек заполняется сверху вниз, навстречу кодам программы?

A. Да.

Q. `r13` — указатель фрейма вызова. Что это такое, и как мы его используем в машине?

A. Это требуется для вызова функций. Будут примеры на семинарах.

Q. `r15` — счётчик команд. В чем его роль?

A. В нём содержится адрес исполняющейся команды.

Q. Как моделировать вычислительную систему?

A. Программой-эмулятором. Моделируется сразу и процессор и операционная система. Каждой ячейке памяти модели должна соответствовать ячейка памяти эмулятора. Таким образом, в эмуляторе появляется

большой массив `mem[1024*1024]`. В модели — 16 регистров. В эмуляторе появляется маленький массив `regs[16]`. Программа хранится в памяти `mem` — она туда попадает либо после загрузки с диска, либо после ассемблирования исходного кода.

Q. Как ассемблировать?

A. Требуется два прохода по исходному файлу. В первом проходе вы назначаете значения всем меткам, во втором — генерируете код и

1. Устанавливаете счётчик команд `pc` в 0.
2. Считываете строку и разбиваете её на *лексемы*. То, что кончается двоеточием, `start:` — метка, на неё может быть ссылка (либо она уже была). В первом проходе заводите в таблице меток запись, ключом в которой — имя метки, а значением — текущее значение счётчика команд. Так проделываете с каждой меткой в строке и убираете их.
3. Если в строке ничего больше нет — переходите к пункту 2.
4. В строке что-то есть.
   - Если это директива `end`, тогда вторая лексема — метка, определяющая начало исполнения программы. Запоминаете значение метки (из таблицы меток) в `regs[15]` и прекращаете проход ассемблирования. Если это директивы `word` или `double`, заполняете `mem[pc]` соответствующими значениями и продвигаете `pc` на следующее место в памяти.
   - Если это исполнимая команда то на первом проходе ничего не делаете, а на втором формируете слово в памяти, которое её кодирует. Для этого находите в таблице соответствий именам машинных команд их кодов, заполняете нужные биты в кодовом слове. Устанавливаете тип команды — `RR`, `RI`, `J`, `RM`. По типу команды устанавливаете её операнды. Если в команде есть метка, находите её числовое значение. Заполняете остальные биты в кодовом слове. Записываете кодовое слово в `mem[pc]`, продвигаете `pc` на следующее слово. Отправляемся к 2.

Q. Как эмулировать?

A. Если программу надо загрузить с диска, загружаете её в массивы `mem` и `regs`. Если вы её только что ассемблировали, эти массивы уже заполнены нужным образом. Теперь эмулируем:

1. `regs[15]` всегда должен содержать адрес эмулируемой команды.
2. Считываем команду. Извлекаем биты с её кодом и присваиваем переменной `code`. По таблице соответствий кодов команды и её типу вычисляем тип — `RR`, `RI`, `RM` или `J`.

3. Если это вызов операционной системы `syscall`, то или вводите со стандартного ввода в указанный в команде регистр `regs[r]` в нужном формате, либо выводите на стандартный вывод его же.
4. Если это команда типа `J`, устанавливаем, требуется ли переход. Если он требуется, присваиваем `regs[15]` нужное значение.
5. Для команд остального типа исполняем, модифицируем нужные регистры, память и продвигаем счётчик команд (`regs[15]` на 1.
6. Возвращаемся к пункту 2.

Q. Как дизассемблировать?

A. Элементарно, Watson! Вместо эмуляции команды просто вывести её расшифровку и никуда более не переходить!

Q. Какие структуры данных понадобятся для компиляции?

A. Массив, в который будет складываться код программы. Таблица меток. Таблица перевода кодов операций и регистров в число. Таблица типов операций.

Q. Какие структуры данных понадобятся для эмуляции?

A.. Массив, в котором хранится образ памяти модели и массив с образом регистров. Таблица типов операций.