Практическое занятие по СПО №4

Густов Владимир Владимирович
gutstuf@gmail.com

Цитата дня: Вся история разработки программного обеспечения - это история повышения уровня абстракции. (с) Гради Буч

Repeat it

1
```
if i < 5 then
begin
  i := i + 1;
  j := 2 – 1 div 3;
end;
```

2
```
for i := 0 + 3 * 2 to 2 div 2 * 10 do
    a := i + (3 – 2) div 2;
```

3
```
while i * 2 < 5 do
    i := i + 1;
```

4
```
repeat
  i := i + 2;
    while i < 5 do
        i := i + 1;
until i >= 10;
```

Repeat it

1
```
case a div a + a of
  0: a := 0;
  1: a := 1;
  2: a := 2;
  3: for a := 0 to a do
       b := a + 2;
end;
```

2
```
begin
  for i := 3 xor 5 to 5 do
  begin
    a := 3 and 5 + 2;
    break;
  end;
end.
```

3
```
goto g;
if a > 5 then
  a := 5
else
  a := 2;
g:
a := 0;
```

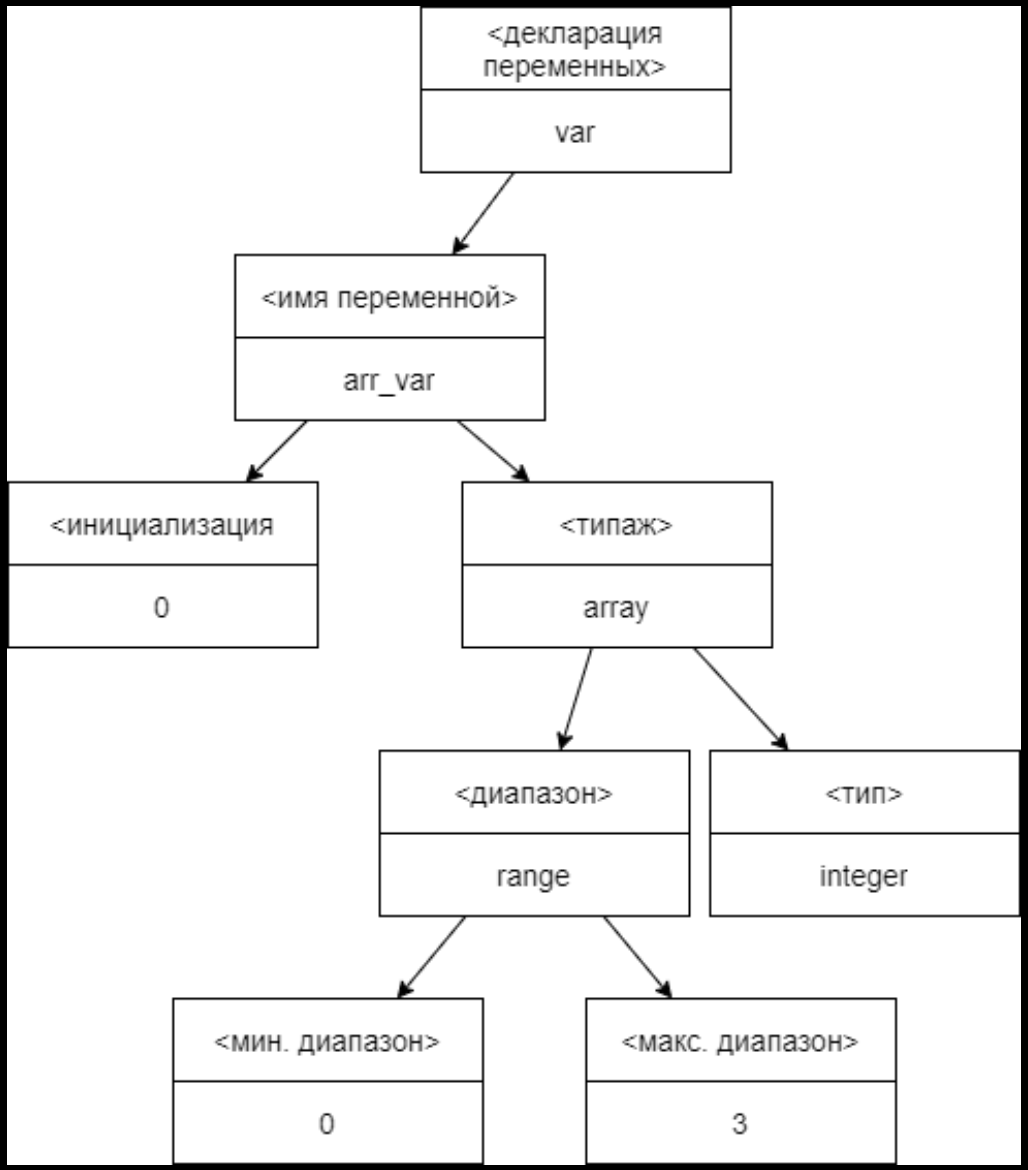
Массивы

```
<program> ::= program <identifier> ; <block> .
<block> ::= <variable declaration part>

<variable declaration part> ::= <empty>
   | var <variable declaration> {; <variable declaration>} ;
<variable declaration> ::=
          <identifier> {,<identifier>} : <type>

<type>             ::= <simple type> | <array type>
<array type>       ::= array [ <index range> ] of <simple type>
<index range>      ::= <integer constant> .. <integer constant>

<simple type>      ::= <type identifier>
<type identifier>  ::= <identifier>
```

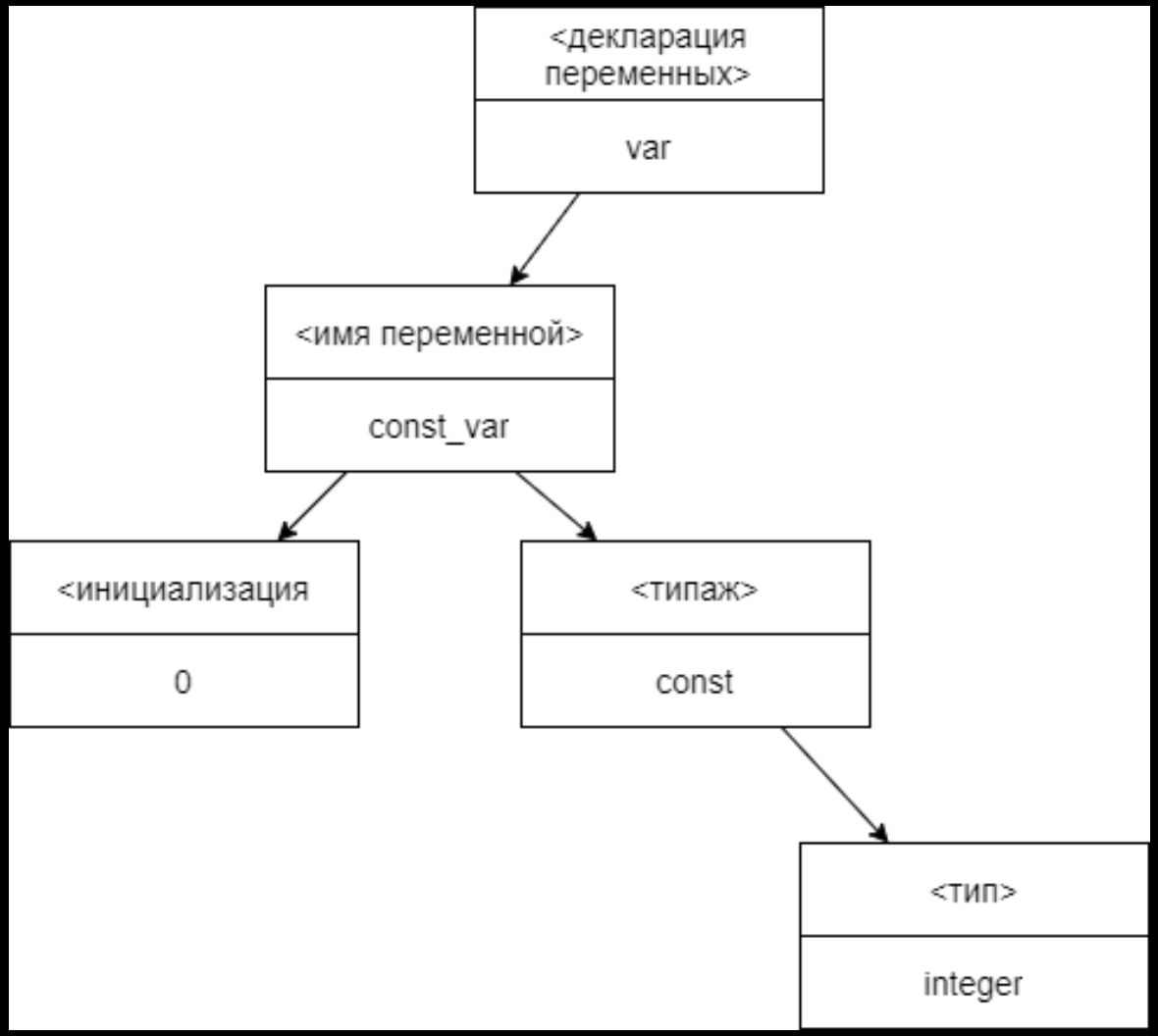



```
program arr;
var arr_var : array [0..3] of integer = (0);
```

Константы

```
<block> ::= <const def part> <var decl part>

<const def part> ::= <empty>
                   | const <const def> { ; <const def>} ;
<const def> ::= <identifier> = <constant>
<constant> ::= <number> | <sign> <number>
                        | <const id> | <sign> <const id>
<const id> ::= <identifier>
```




```
program arr;
const const_var = 0;
```

```
program ag41n;
    var
        a : array [0 .. 2] of integer;
    const b = 23;
          c = b + 2;
    begin
        a[0] := 2 + c div b;
        a[1] := a[0] * b;
    end.
```

Сериализация

Процесс перевода структуры данных в формат пригодный для хранения/передачи и реконструированию (десериализации) позже.

Наиболее известные форматы:

- JSON — JavaScript Object Notation;
- XML — Extensible Markup Language;
- CSV — Comma-Separated Values;

И другие варианты (google it):

- YAML, NMEA, Pickle, OpenDDL, INI, Protobuf.

Для проверки корректности, используйте валидаторы (google JSON/XML/CSV validator)

JSON

1

```
{
  "program": {
    "type": "prog_tk",
    "line": 1
   },
  "exmpl2": {
    "type": "id_tk",
    "line": 1
  }
}
```

2

```
 {
   "name": ["program", "exmpl3", "var", "a"],
   "type": ["program_tk", "id_tk", "var_tk", "id_tk"],
   "line": [1, 1, 2, 2]
 }
```

Tools in C++: Boost, Nlohmann JSON, RapidJSON, etc

XML

1

```
<lex_table>
  <lex1>
    <name> program </name>
    <type> program_tk </type>
    <line> 1 </line>
  </lex1>
  <lex2>
    <name> exmpl1 </name>
    <type> id_tk </type>
    <line> 1 </line>
  </lex2>
</lex_table>
```

2

```
<lex_table>
  <name> program
    <type> program_tk </type>
    <line> 1 </line>
  </name>
  <name> exmpl1
    <type> id_tk </type>
    <line> 1 </line>
  </name>
</lex_table>
```

Tools in C++: Boost, gSOAP, TinyXML-2, etc

CSV

1

```
id,name,type,line
1,program,program_tk,1
2,exmpl,id_tk,2
```

2

```
line,name,type
1,program,program_tk
1,exmpl,id_tk
```

Tools in C++: Boost, CSVparser, fast-cpp-csv-parser, etc

a := 2 + 1 - b * 2 - 1;

Текущий адрес	Значение	Адрес левой ветви	Адрес правой ветви
0x16a03f0	:=	0x16a0440	0x16a0490
0x16a0440	a	0	0
0x16a0490	+	0x16a04e0	0x16a0530
0x16a04e0	2	0	0
0x16a0530	-	0x16a0580	0x16a06c0
0x16a0580	1	0	0
0x16a06c0	-	0x16a05d0	0x16a0710
0x16a05d0	*	0x16a0620	0x16a0670
0x16a0620	b	0	0
0x16a0670	2	0	0
0x16a0710	1	0	0

```
program exmpl1;
var a, b, c, efi : integer;
    raze : integer = 0;
const ezar = 25;
label gogo;
begin
   a := raze * 12 - 15 div 5;
   if a < ezar then
       b := 2
   else if a = ezar then
   begin
       c := ezar;
       b := 2 * 3;
   end else
   begin
       gogo:
       efi := a;
       b := efi + raze - a;
   end;
   case b of
    0: c := b;
    1: goto gogo;
    2: a := b + ezar * raze;
    else a := 1;
   end;
end.
```

Вопросы & ответы

Генератор кода

Asm/OS	Windows	DOS	Linux	MacOS
FASM	V	V	V	
GAS	V	V	V	V
GoAsm	V			
HLA	V		V	
MASM	V	V		
NASM	V	V	V	V
RosAsm	V			
TASM	V	V		

Tools

MinGW — *Minimalist GNU for Windows* - набор GNU приложений (bash, gcc/g++, ar, as, ld, nm, etc) для Windows.

build: i686-7.1.0-posix-dwarf-rt_v5-rev2

```
export PATH=$PATH:/c/<путь до mingw32/bin/>
alias make=mingw32-make.exe
<диск>/.../home/<user>/.bashrc
touch <file> # создаёт файл с именем <file>
```

GDB

`gdb ./your_exmpl.exe`

`b main` — устанавливаем breakpoint на метку main

`layout reg` — меняем отображение на «более» наглядное

`run` — загружаем программу

`step` — пошагово проходим отладчиком

`print $eax` — вывод содержимого регистра eax

`p/x (<тип>)<метка>` - вывод содержимого метки

`x/s <адрес памяти>` - вывод содержимого ячейки памяти

Генератор кода

Отличие AT&T ассемблера (GAS) от Intel-ассемблера (F/N/M/TASM):

- Комментарии начинаются с «#», а не «;». «;» - разделяет команды.
- Имена регистров начинаются с «%», т. е. %eax, %bx, вместо eax, bx.
- Отсутствие префикса операнда указывает на адрес в памяти:
- movl $foo, %eax — помещает адрес переменной foo в регистр %eax;
- movl foo, %eax — помещает содержимое переменной foo в регистр %eax;
- Суффикс инструкции определяет размер операнда:
- *mov***b** — (byte) — операнды размером в 1 байт;
- *mov***w** — (word) — операнды размеров в 2 байта (1 слово);
- *mov**l*** — (long) — операнды размером в 4 байта;
- ..etc..
- Порядок операндов: вначале источник, затем приёмник:
- `mov eax, ebx    ; Intel`
- `movl %ebx, %eax # AT&T`
- Для записи числовых констант используется символ «$»:
- `mov ebx, 10h     ; Intel`
- `movl $0x10, %ebx # AT&T`
- Для разыменования значения по указанному адресу используются (), вместо []

GNU Assembler (GAS)

```
# директивы начинаются с «.»
.section .data   # .section — необязательная директива
                 # указывающая на начало секции
.text
  .global main   # точка входа в программу
                 # (для Linux; _main - для Windows)
main:            # метка main обязательна
  movl $0, %eax  # регистр eax = 0
  ret            # аналог return N в C/C++, где N  -
                 # младший байт регистра ax (al)
```

Для компиляции:

gcc –o <исполняемый файл> <ассемблер>.s

Из C в asm:

gcc -o - -S -fno-asynchronous-unwind-tables <C код>.c

Пример (при запуске через *msys*):

gcc -o test test.s # компиляция (ассемблирование и линковка)
./test.exe # запуск получившейся программы test
echo $? # проверка результата возвращаемого программой test

РОН и СР

Регистры общего назначения (GPR):

%eax: *Accumulator register* — аккумулятор, применяется для хранения результатов промежуточных вычислений.

%ebx: *Base register* — базовый регистр, применяется для хранения адреса (указателя) на некоторый объект в памяти.

%ecx: *Counter register* — счетчик, его неявно используют некоторые команды для организации циклов.

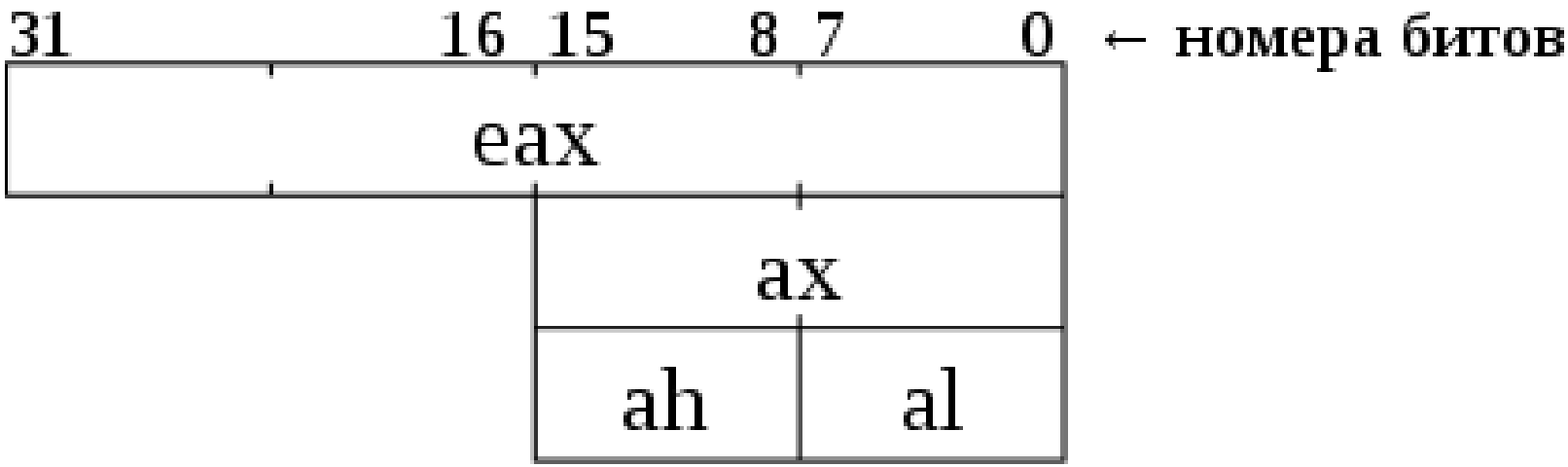
%edx: *Data register* — регистр данных, используется для хранения результатов промежуточных вычислений и ввода-вывода.

%esp: *Stack pointer register* — указатель стека. Содержит адрес вершины стека.

%ebp: *Base pointer register* — указатель базы кадра стека (англ. *stack frame*). Предназначен для организации произвольного доступа к данным внутри стека.

%esi: *Source index register* — индекс источника, указатель на данные в сегменте, указанном регистром **%ds**; указатель источника для строковых операций.

%edi: *Destination index register* — индекс приёмника, указатель на данные (или пункт назначения) в сегменте, указанном регистром **%es**; указатель назначения для строковых операций.


31
16
15
8
7
0
← номера битов
eax
ax
ah
al

Сегментные регистры:

%cs: *Code segment* — описывает текущий сегмент кода.

%ds: *Data segment* — описывает текущий сегмент данных.

%ss: *Stack segment* — описывает текущий сегмент стека.

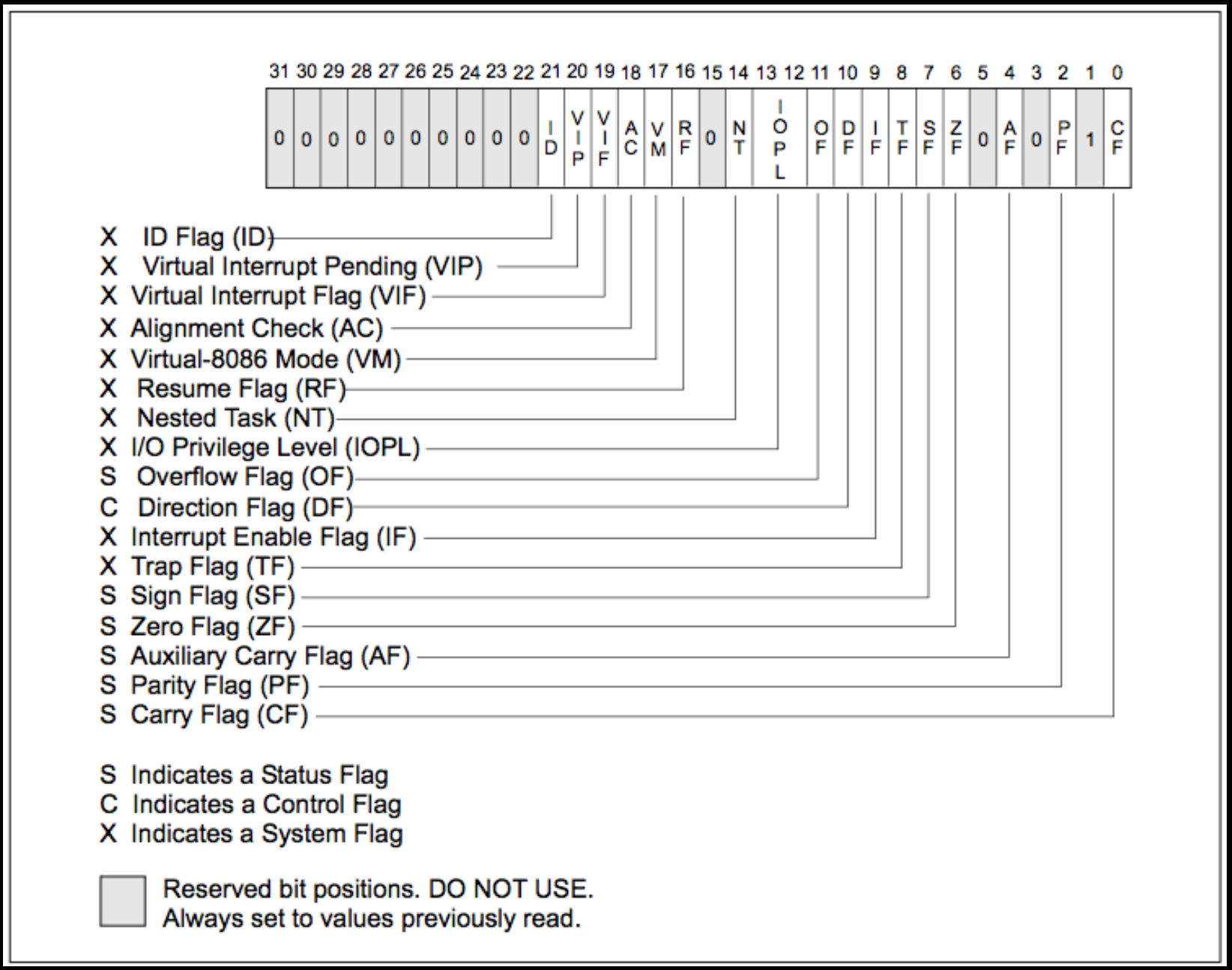
%es: *Extra segment* — дополнительный сегмент, используется неявно в строковых командах как сегмент-получатель.

%fs: *F segment* — дополнительный сегментный регистр без специального назначения.

%gs: *G segment* — дополнительный сегментный регистр без специального назначения.

eip — *Instruction pointer* — содержит указатель на следующую команду.

Регистр флагов **eflag**



Директивы (секции)

.data – сегмент данных, которые инициализируются значением
.bss — *block storage segment* – сегмент резервирования памяти (без инициализации)
.text – сегмент кода
.global (.globl) – объявление глобальной метки (функции)

Типы данных

Для .data:
.byte <value> - 1 байт (аналог Byte)
.short <value> - 2 байта (аналог Smallint)
.long <value> - 4 байта (аналог integer)

Для .bss:
.space <N> - занимает N байт

Адресация

Смещение = +<размер переменной>
$ - непосредственная
→ - абсолютная
() – косвенная (со смещением)
% - регистровая

Адресация

1) $ - непосредственная – значение берётся «буквально» (напрямую)

2) _ - абсолютная – значение считается как адрес, и производится попытка считывания этого адреса (за исключением обращения к регистрам)

3) ($) – косвенная – работает как и абсолютная; при обращении к регистру, пытается считать значение как адрес

Указатели

смещение(база, индекс, множитель)

*Вычисленный адрес = база + индекс * множитель + смещение*

Множитель может принимать значения 1, 2, 4 или 8 (кратные 2).

(%ecx) - адрес операнда находится в регистре %ecx.

4(%ecx) - адрес операнда равен %ecx + 4. Например, в %ecx адрес некоторой структуры, второй элемент которой находится «на расстоянии» 4 байта от её начала (говорят «по смещению 4 байта»);

-4(%ecx) - адрес операнда равен %ecx − 4;

foo(,%ecx,4) - адрес операнда равен foo + %ecx × 4, где foo — некоторый адрес. Если foo — указатель на массив, элементы которого имеют размер 4 байта, то мы можем заносить в %ecx номер элемента и таким образом обращаться к самому элементу.

Указатели

```
.data
arr:    # 0  1  2  3
   .byte 1, 4, 8, 7

.text
.globl _main

_main:
  movl $2, %ecx           # используем, как номер эл-та, к-му хотим
                          # обратиться (в данном случае, ко 2 эл-ту)
  movb arr(,%ecx, 1), %al # в регистре al будет хранится число 8,
                          # т.к. arr + 2 * 1 == адрес указывающий на 2 эл.

  movl $arr, %ebx         # сохраняем адрес массива в регистре ebx
  movb 3(%ebx), %al       # в регистре al будет хранится число 7, т.к.
                          # %ebx + 3 == arr + 3 = адрес указ-ий на 3 эл.

  movb arr + 1, %al       # в регистре al будет хранится число 4, т.к.
                          # arr + 1 = адрес указывающему на первый элемент
  ret
```

```
.data
some_var:
        .long 0x00000072
other_var:
        .long 0x00000001, 0x00000002, 0x00000003


.text
.globl main
main:
  movl  $0x48, %eax # поместить число 0x00000048 в %eax
  movl  $some_var, %eax
  /* поместить в %eax значение метки some_var, то есть
адрес числа в памяти; например, содержимое %eax равно
0x08049589 */

  movl  some_var, %eax /* обратиться к содержимому
                       переменной;
                       в %eax теперь 0x00000072 */

  movl  other_var + 4, %eax /* other_var указывает на
                            0x00000001
                            размер одного значения
                            типа long — 4 байта;
                            значит, other_var + 4
                            указывает на 0x00000002;
                            в %eax теперь 0x00000002 */

  movl  $1, %ecx        /* поместить число 1 в %ecx */
```

```
movl  other_var(,%ecx,4), %eax  /* поместить
в %eax первый элемент массива other_var,
пользуясь %ecx как индексным регистром */

movl  $other_var, %ebx  /* поместить в %ebx
адрес массива other_var  */


movl  4(%ebx), %eax     /* обратиться по
адресу %ebx + 4; в %eax снова 0x00000002 */

movl  $other_var + 4, %eax  /* поместить в
%eax адрес, по которому расположен
0x00000002 (адрес массива плюс 4 байта --
 пропустить нулевой элемент)*/


movl  $0x15, (%eax)     /* записать по
адресу "то, что записано в %eax" число
0x00000015 */
```

Compile time

```
program varies;
var a : integer = 0;
    d : integer;
const b = 5;
    j : integer = 2;
const i : integer = 2;
begin
    for d := 0 to b do
      writeln ('Hello World');
    i := 3;
    writeln(i);
end.
```

<program> ::= **program** <identifier> ; <block> .

<block> ::= <variable declaration part> <statement part>

<statement part> ::= <compound statement>
<compound statement> ::= **begin** <statement> { ; <statement> } **end**
<statement> ::= <simple statement> | <structured statement>
<structured statement> ::= <compound statement> | <if state> | <repet state>
<if state> ::= if <exp> then <statement> | .. else <statement>
<repet state> ::= <while state> | <repeat state> | <for state>
<while state> ::= while <exp> do <statement>
<repeat state> ::= repeat <state> {; <state>} until <exp>
<for state> ::= for <id> := <for list> do <state>