Machine Level Programming Note & BOMB LAB

Instructions

movq: q 是指 quad word 四字，其中1 word = 16bit 是按16位计算机算的，那么quad word即64bit

`movzbl`

- is an x86 assembly language instruction that is used to move a value from a memory location to a register **while also performing a zero-extension operation.** 高位0扩展

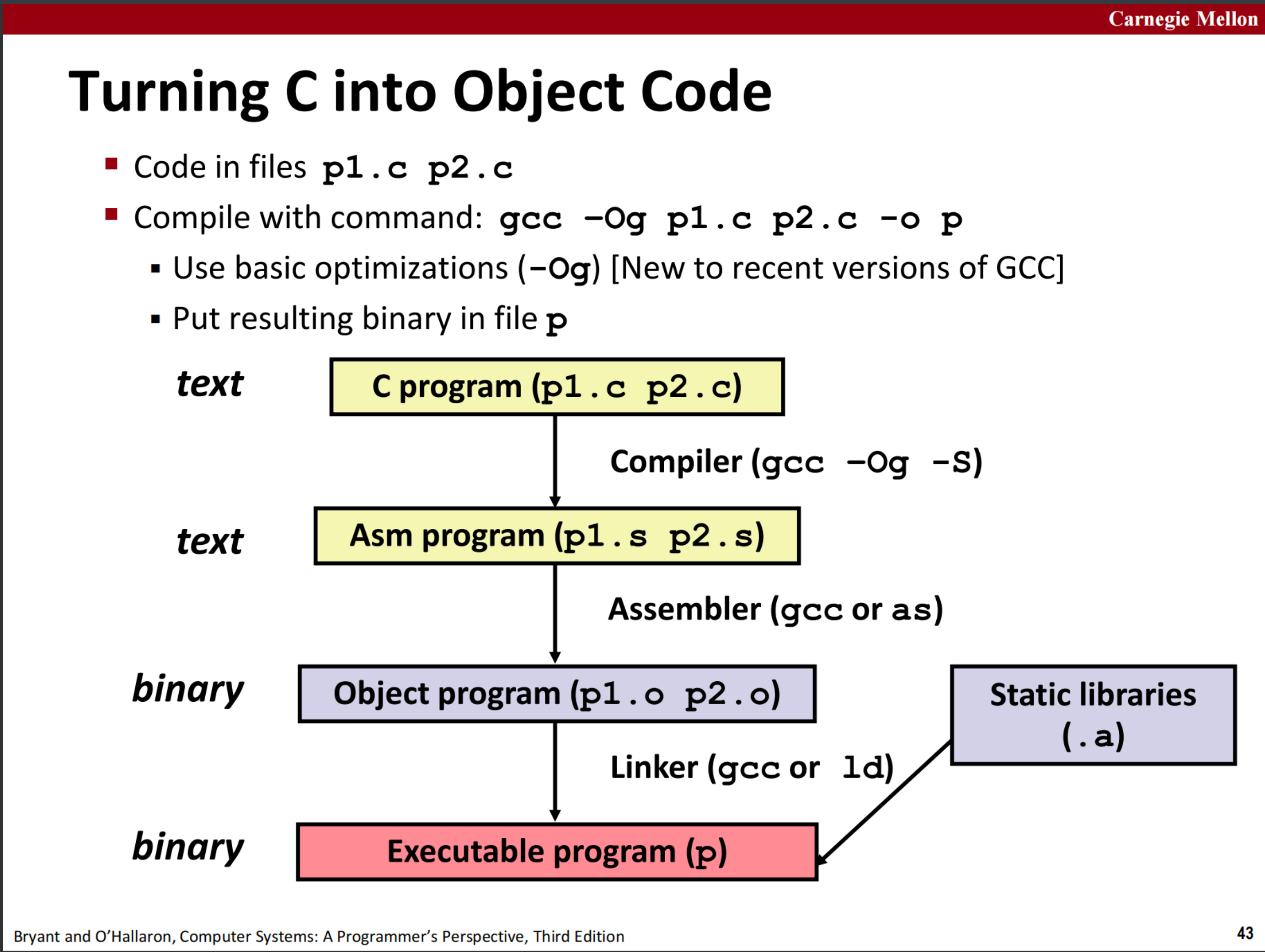
Registers

Register	Callee Save	Description
%rax		result register; also used in idiv and imul instructions.
%rbx	yes	miscellaneous register
%rcx		fourth argument register
%rdx		third argument register; also used in idiv and imul instructions.
%rsp		stack pointer
%rbp	yes	frame pointer
%rsi		second argument register
%rdi		first argument register
%r8		fifth argument register
%r9		sixth argument register
%r10		miscellaneous register
%r11		miscellaneous register
%r12-%r15	yes	miscellaneous registers

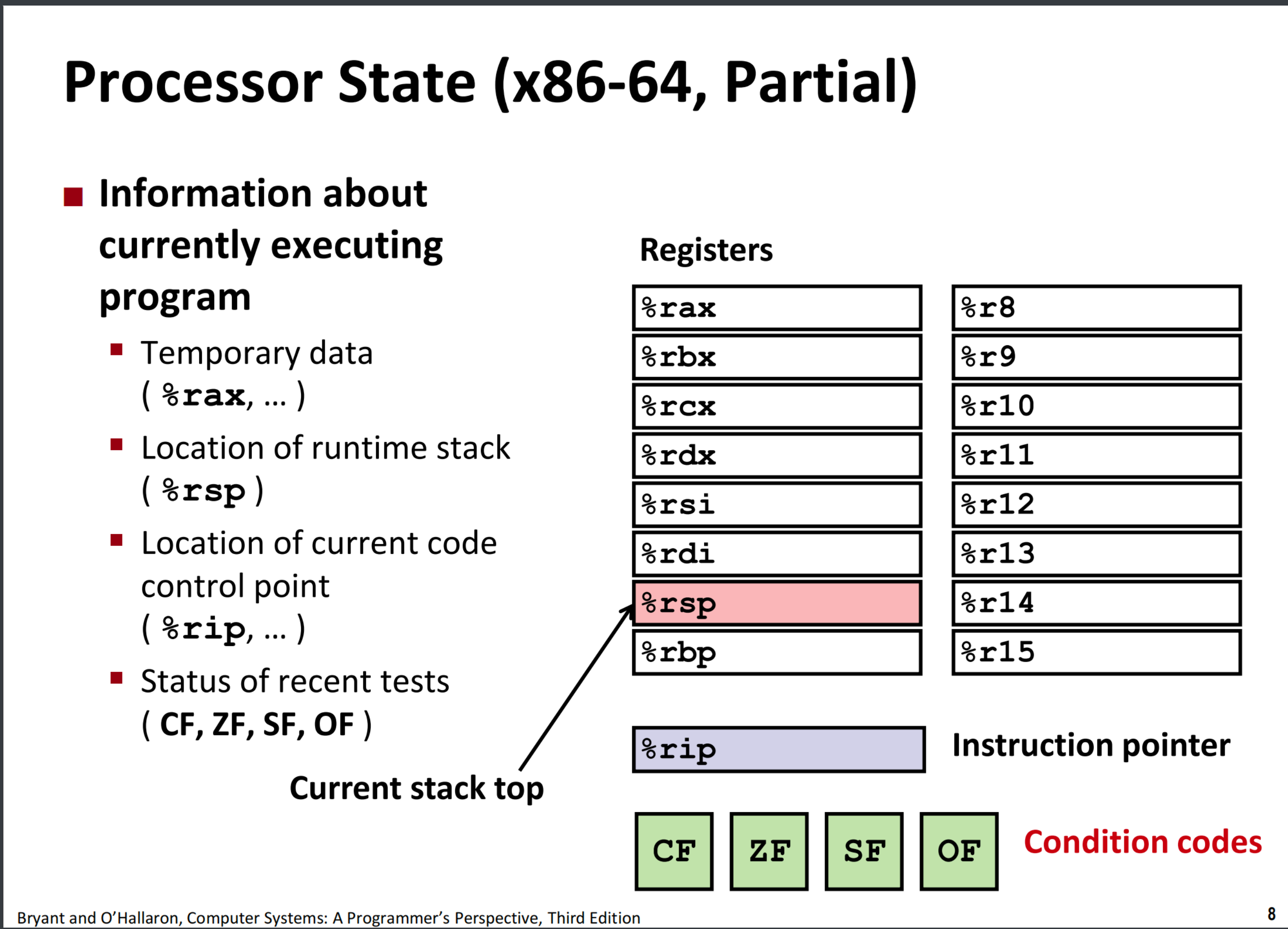
63	31	15	7 0	
%rax	%eax	%ax	%al	返回值
%rbx	%ebx	%bx	%bl	被调用者保存
%rcx	%ecx	%cx	%cl	第4个参数
%rdx	%edx	%dx	%dl	第3个参数
%rsi	%esi	%si	%sil	第2个参数
%rdi	%edi	%di	%dil	第1个参数
%rbp	%ebp	%bp	%bpl	被调用者保存
%rsp	%esp	%sp	%spl	栈指针
%r8	%r8d	%r8w	%r8b	第5个参数
%r9	%r9d	%r9w	%r9b	第6个参数
%r10	%r10d	%r10w	%r10b	调用者保存
%r11	%r11d	%r11w	%r11b	调用者保存
%r12	%r12d	%r12w	%r12b	被调用者保存
%r13	%r13d	%r13w	%r13b	被调用者保存
%r14	%r14d	%r14w	%r14b	被调用者保存
%r15	%r15d	%r15w	%r15b	被调用者保存

图 3-2 整数寄存器。所有 16 个寄存器的低位部分都可以作为字节、字(16 位)、双字(32 位)和四字(64 位)数字来访问

Compile Process



Processor State



- RIP register ——namely PC

Condition Codes–Testq–Cmpq

- CF: Carry Falg
- ZF: Zero Flag
- SF: Sign Flag
- OF: Overflow Flag

Leaq 指令不改变condition codes的值

Condition Codes (Implicit Setting)

- **Single bit registers**
  - **CF** Carry Flag (for unsigned) **SF** Sign Flag (for signed)
  - **ZF** Zero Flag **OF** Overflow Flag (for signed)

- **Implicitly set (as side effect) of arithmetic operations**

  Example: **`addq`** *Src,Dest* ↔ **`t = a+b`**

  CF set if carry out from most significant bit (unsigned overflow)

  ZF set if `t == 0`

  SF set if `t < 0` (as signed)

  OF set if two's-complement (signed) overflow
  `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

- **Not set by `leaq` instruction**

cmpq

> Cmpq namely
> cmpq b, a ------ computing a-b

Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**
  - **`cmpq`** *Src2*, *Src1*
  - **`cmpq b,a`** like computing **`a-b`** without setting destination

  - **CF set** if carry out from most significant bit (used for unsigned comparisons)
  - **ZF set** if **`a == b`**
  - **SF set** if **`(a-b) < 0`** (as signed)
  - **OF set** if two's-complement (signed) overflow
    `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

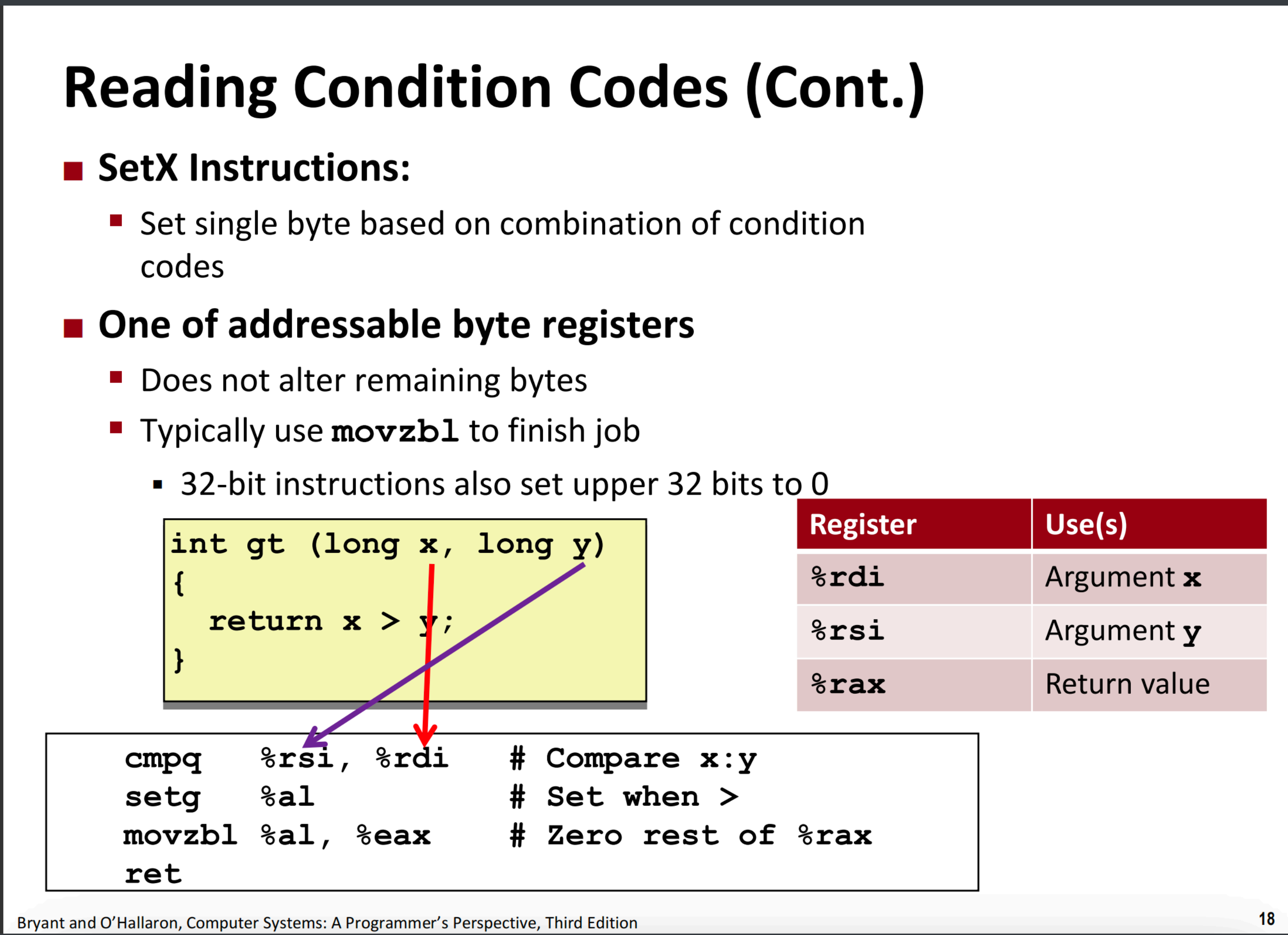
testq

Condition Codes (Explicit Setting: Test)

- **Explicit Setting by Test instruction**
  - **`testq`** *Src2*, *Src1*
    - **`testq b,a`** like computing **`a&b`** without setting destination
  - Sets condition codes based on value of *Src1* & *Src2*
  - Useful to have one of the operands be a mask
  - **ZF set** when **`a&b == 0`**
  - **SF set** when **`a&b < 0`**

Very often:

```
testq  %rax,%rax
```



movzbl：

move zero-extended byte to long. 将一字节 移动到 destination（必须是32bit reg），同时一字节以外的bit用0填充

The "destination" operand must be a 32-bit register that will receive the zero-extended value.

Conditional

Jumping

- **jX Instructions**
  - Jump to different part of code depending on condition codes

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

note that `SF ^ OF` **represent less**

Since : SF or OF is set , if SF set , means that the comparasion 'minus ' result is less than 0

if OF set, means comparasion 'minus ' result is a negtive minus another positive

if **SF and OF are set simultaneously** , it may be two positive addition , since SF^OF is false

Conditional Move

其实就是把两个分支都计算一遍，然后根据条件取其中一个结果

但有的时候不能使用条件移动

Conditional Move Example

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Register	Use(s)
`%rdi`	Argument `x`
`%rsi`	Argument `y`
`%rax`	Return value

```
absdiff:
   movq    %rdi, %rax  # x
   subq    %rsi, %rax  # result = x-y
   movq    %rsi, %rdx
   subq    %rdi, %rdx  # eval = y-x
   cmpq    %rsi, %rdi  # x:y
   cmovle  %rdx, %rax  # if <=, result = eval
   ret
```

When is this bad?

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

Bad Performance

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Unsafe

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

Illegal

LOOP

General "Do-While" Translation

C Code

```
do
  Body
  while (Test);
```

Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```

- **Body:**

```
{
  Statement1;
  Statement2;
    …
  Statementn;
}
```

General “While” Translation #1

- “Jump-to-middle” translation
- Used with –Og

While version

```
while (Test)
  Body
```

Goto Version

```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```


General “While” Translation #2

While version

```
while (Test)
  Body
```

- “Do-while” conversion
- Used with –O1

Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

Bomb Lab

> GDB command
>
> `disas` function
>
> - 反汇编该函数的代码

> `display/3i $pc`
>
> - 从当前指令展示三条instruction

> `p *int(*) ($rsp + 8)`
>
> - 打印栈中第三个元素 用int形式打印
>
> **`p (char*) 0x402400`**
>
> **`x/s 0x402400`**
>
> - 打印字符串：$1 = 0x402400 "Border relations with Canada have never been better."
> - 打印字符串： 0x402400: "Border relations with Canada have never been better."
>
> (gdb) **`x/s $esi`**
>
> - 0x40245e: "flyers"
>
> `print /x ($rsp+8)`

> - Print (contents of %rsp) + 8 in hex

So, `*(int *)` as a whole means to dereference the pointer and treat the data stored at the memory address as an integer value.

> `break phase_4`
>
> - Phase_4函数开始处打断点
>
> `break *0x80483c3`
>
> - Set breakpoint at address 0x80483c3
>
> b 84
>
> - c代码84行打断点

> `layout asm`
>
> - 展示出动态的汇编代码框，可以在里面si

layout src 显示源码窗口 layout asm 显示汇编窗口 layout split 显示源码 & 汇编窗口 layout regs 显示汇编 & 寄存器窗口 layout next 下一个layout layout prev 上一个layout C-x 1 单窗口模式 C-x 2 双窗口模式 C-x a 回到传统模式

> `step i`
>
> - instruction 单步进入（会进入函数
>
> `next i`
>
> - instruction下一步

phase_3

Phase_3 本身就是一个switch 语句，要善用调试

Phase_4

递归，但是可以直接看代码，跑一遍 跳过递归

Phase_5

- %cl

> Cl register —— counter of LOOP / low 8 bit of $rcx
>
> %rcx is a 64-bit register that is used for storing information related to the control of certain instructions, just like %cx is a 16-bit register used for the same purpose. The lower 8 bits of the %rcx register correspond to the %cl register, the middle 16 bits correspond to the %cx register, and the full 64 bits correspond to the %rcx register.
>
> In particular, %cl is the low 8 bits of the %cx register and %rcx register, and it can be used to specify the number of iterations in certain loop instructions such as LOOP and LOOPZ/LOOPE.
>
> The %cl register is a 8-bit register in the x86 architecture used for **storing information related to the control of certain instructions.**
>
> In particular, it is commonly used as the counter register in loop instructions such as LOOP and LOOPZ/LOOPE. These instructions repeat a block of code a specified number of times, where the number of repetitions is determined by the

> value stored in the %cx register.
>
> **%cl, as a part of %cx, can be used to specify the number of iterations in the range of 0 to 255, and it is decremented by 1 each time the loop instruction is executed until it reaches zero.**

```
0x40107a <phase_5+24>    callq  0x40131b <string_length>
0x40107f <phase_5+29>    cmp    $0x6,%eax
0x401082 <phase_5+32>    je     0x4010d2 <phase_5+112>
0x401084 <phase_5+34>    callq  0x40143a <explode_bomb>
0x401089 <phase_5+39>    jmp    0x4010d2 <phase_5+112>
0x40108b <phase_5+41>    movzbl (%rbx,%rax,1),%ecx
0x40108f <phase_5+45>    mov    %cl,(%rsp)
0x401092 <phase_5+48>    mov    (%rsp),%rdx      LOOP
0x401096 <phase_5+52>    and    $0xf,%edx
0x401099 <phase_5+55>    movzbl 0x4024b0(%rdx),%edx
0x4010a0 <phase_5+62>    mov    %dl,0x10(%rsp,%rax,1)
0x4010a4 <phase_5+66>    add    $0x1,%rax
0x4010a8 <phase_5+70>    cmp    $0x6,%rax
0x4010ac <phase_5+74>    jne    0x40108b <phase_5+41>
0x4010ae <phase_5+76>    movb   $0x0,0x16(%rsp)
0x4010b3 <phase_5+81>    mov    $0x40245e,%esi
0x4010b8 <phase_5+86>    lea    0x10(%rsp),%rdi
0x4010bd <phase_5+91>    callq  0x401338 <strings_not_equal>
```

This loop is for read the 6 inputs and find the concrete letter from 0x4024b0

(gdb) x/s 0x4024b0
0x4024b0 <array.3449>: "**maduiersnfotvbyl**So you think you can stop the bomb with ctrl-c, do you?"
(gdb)

根据ascii码表，其中<phase_5+52> 只保留了最后4bit，即我们要看ascii码表16进制的末尾，即是偏移

比如我的样例'test12' 这个循环处理过后变成

(gdb) x/s ($rsp+0x10)
0x7fffffffe480: **"ieuiad"**

t ascii 0x74, maduiersnfotvbyl 的第4偏移就是i

e ascii 0x0x65 maduiersnfotvbyl 第5偏移为u

所以想要得到<phase_5_81>该地址下的字符串 flyers

(gdb) x/s $esi
0x40245e: "flyers"

flyers 在**0m 1a 2d 3u 4i 5e 6r 7s 8n 9f Ao Bt Cv Db Ey Fl**中的偏移分别为 9 , F, E, 5,6, 7

在ascii码表中寻找第四位为上述偏移的即可

Eg: yonefg