

A decorative graphic featuring stylized green leaves and swirling vines, positioned horizontally across the middle of the slide, framing the title.

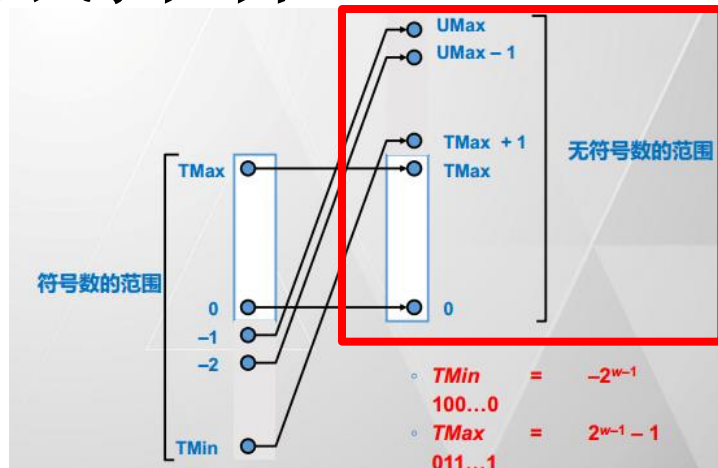
汇编作业讲解

费翔 feix16@mails.tsinghua.edu.cn
2021.8



无符号整数的加减操作

- x 是任意整数, $x > 0$, y 是正数, $y > 0$
- $x + y \rightarrow x$ 往上移动 y 个位子
- $x - y \rightarrow x$ 往下移动 y 个位子
- 类似于数轴的概念
- 如何能统一往上/往下两个方向的移动?
- 范围有限并且循环 (也就是往上 R 个位子, 溢出后回到原地, R 是总位子数)



- 往下移动 k 个位子 \leftrightarrow 往上移动 $R - k$ 个位子 (很关键)
- x 减 $y \rightarrow x$ 往下移动 y 个位子 $\rightarrow x$ 往上移动 $R - y$ 个位子 $\rightarrow x$ 加 $(R - y)$ 这是一个整体并且是正数
- w 位无符号整数中, $R = 2^w$
- $-y = R - y = (\sim y) + 1$, 证明: $R - 1 = y + (\sim y) =$ 全1的二进制串, 移项后可得左侧





无符号整数的加减操作

- x 是任意整数, $x > 0$, y 是正数, $y > 0$

- $x+y \rightarrow x$ 往上移动 y 个位子

- $x-y \rightarrow x$ 往下移动 y 个位子

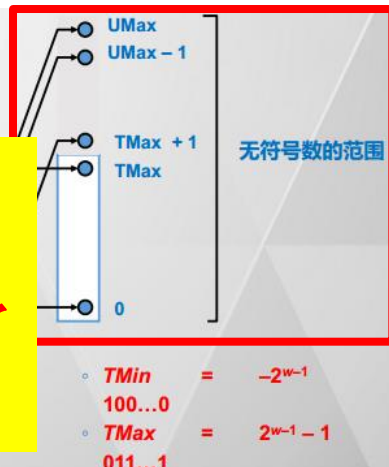
- 类似于数轴的概念

- 如何能统一往上/往下两个

$$x-y = x + (\sim y) + 1$$

只有加法和位运算

用数学的恒等变换简化了
减法的电路实现



- 范围有限并且循环（也就是往上 R 个位子，溢出后回到原地， R 是总位子数）

- 往下移动 k 个位子 \leftrightarrow 往上移动 $R-k$ 个位子

这是一个整体并且是正数

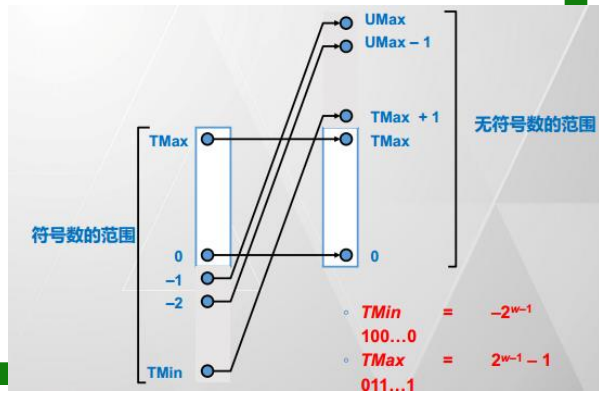
- x 减 $y \rightarrow x$ 往下移动 y 个位子 $\rightarrow x$ 往上移动 $R-y$ 个位子 $\rightarrow x$ 加 $(R-y)$

- w 位无符号整数中, $R=2^w$

- $-y = R-y = (\sim y) + 1$ ，证明： $R-1 = y + (\sim y) =$ 全1的二进制串，移项后可得左侧



- 电路/硬件层面，针对无符号整数，可以不专门设计减法电路，只设计加法和位运算电路（上页slice）
- 汇编代码层面，不区分整数的符号，统一按无符号整数对待，只在输入输出时，针对有/无符号整数，翻译成不同的数值
- 为此，人们设计了一套整合有/无符号的整数的加/减规则（也就是补码系统）
- 补码系统的核心：把负整数映射到高范围（如图），也就是在计算机中用无符号整数 $\sim y + 1$ 代替有符号整数 $-y$ 保存，其中 $y > 0$
- $TMax = -128 \rightarrow (\sim 128) + 1 = (\sim 1000\ 0000) + 1 = 0111\ 1111 + 1 = 1000\ 0000 = 128$
- $-1 \rightarrow (\sim 1) + 1 = (\sim 00\dots 001) + 1 = 11\dots 110 + 1 = 11\dots 111 = \text{全}1$
- 思考负数的补码：符号位不变，其他取反，再+1

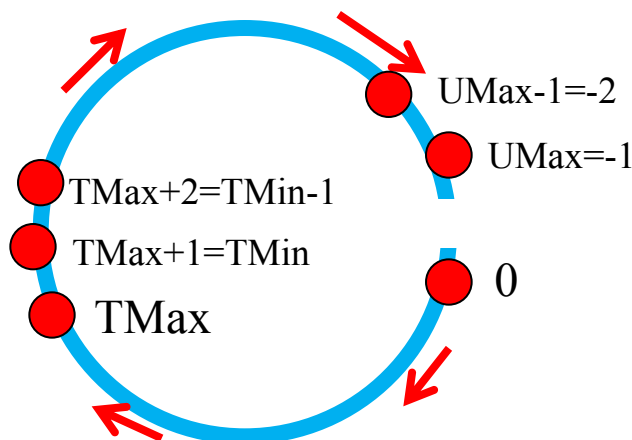
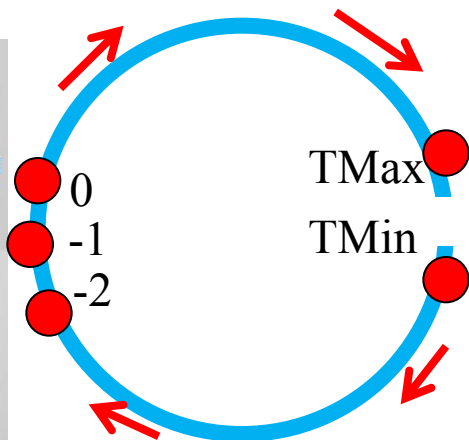
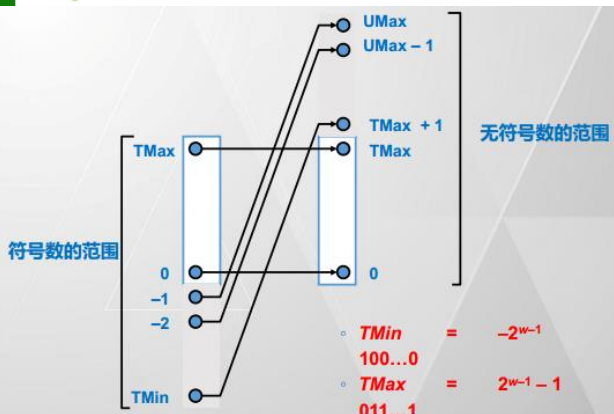




- 无符号整数的公式 $x-y=x+(\sim y)+1$ ，对有符号整数还成立吗
- 回忆无符号整数的推导的逻辑
- x 减 $y \rightarrow x$ 往下移动 y 个位子 $\rightarrow x$ 往上移动 $R-y$ 个位子 $\rightarrow x$ 加 $(R-y)$ (结论1)
- $-y = R-y = (\sim y)+1$ (结论2)
- $\implies x-y=x+(\sim y)+1$ (结论3)
- 对有符号整数，结论2是已知条件，则结论3可以直接推导得到
- 问题：对有符号整数，结论1能推导得到吗



- 对符号整数，往下移动 k 个位子 \leftrightarrow 往上移动 $R-k$ 个位子（还成立吗）



- 两个圆都是循环的，有什么区别，有没有本质区别
- 把负整数映射到高范围的本质：只改变圆的起始位置，不改变圆上的数值的相对位置。所以对符号整数，红色命题仍旧成立，结论1也成立
- x 减 $y \rightarrow x$ 往下移动 y 个位子 $\rightarrow x$ 往上移动 $R-y$ 个位子 $\rightarrow x$ 加 $(R-y)$ （结论1）




- 
- 能否说明已经把有符号整数的加减操作转换成无符号整数的加减操作
 - 无符号整数，假设 $x > 0, y > 0$

- $x+y$ =正常计算
- $x-y=x+(\sim y)+1$

- 有符号整数，假设 $x > 0, y > 0$

- $x+y$ =正常计算
- $x-y=x+(\sim y)+1$
- $x+(-y)=x+(\sim y)+1$ $-y$ 在内存中按 $(\sim y)+1$ 的值保存，电路按第1种情况计算
- $x-(-y)=x+(\sim(-y))+1 = x+(\sim((\sim y)+1))+1 = x+y$ ，电路按第2种情况计算

y 求两次补码

- 请注意区分每个等号是因为什么原因而成立
 - 什么时候考虑溢出？我们认定整数数值范围是循环的，就是在模拟/考虑溢出的情况
- 

1. 已知某 32 位整数 X, 其值为 -101 (十进制), 则其以 16 进制表示的补码为 _____, 另一 32 位整数 Y 的补码为 0xFFFFF6A, 则 X+Y 的 16 进制补码(32 位)为 _____, X-Y 的 16 进制补码为 _____。

- -101 的补码 = $(\sim 101) + 1 = 1111\ 1111\ 1001\ 1011 = \text{FFFF FF9B}$
- $X+Y$ = 两个补码直接做加法, 取后 32 位作为答案 FFFF FF05
- $X-Y = X + \sim Y + 1 = 31$ (这是 16 进制)



2. 请证明补码加法公式： $[x]_{\text{补}} + [y]_{\text{补}} \equiv [x + y]_{\text{补}} \pmod{2^w}$ 。 $[*]_{\text{补}}$ 表示整型数据*的补码表示，机器字长为w。

证明：

(1) $x \geq 0$ 且 $y \geq 0$ ：由于非负数的补码=原码，且 $x + y \geq 0$ ，所以 $[x]_{\text{补}} + [y]_{\text{补}} = x + y = [x + y]_{\text{补}}$ ；

(2) $x \geq 0$ 且 $y < 0$ ：有 $[x]_{\text{补}} = x$ ， $[y]_{\text{补}} = 2^w + y$ ，

a) $x + y \geq 0$ ：所以 $[x]_{\text{补}} + [y]_{\text{补}} = x + y + 2^w \equiv x + y \pmod{2^w} = [x + y]_{\text{补}} \pmod{2^w}$ ；

b) $x + y < 0$ ：所以 $[x]_{\text{补}} + [y]_{\text{补}} = x + y + 2^w = [x + y]_{\text{补}}$ ；

(3) $x < 0$ 且 $y < 0$ ：有 $[x]_{\text{补}} = 2^w + x$ ， $[y]_{\text{补}} = 2^w + y$ ， $[x + y]_{\text{补}} = 2^w + x + y$ ，所以 $[x]_{\text{补}} + [y]_{\text{补}} = x + y + 2^w + 2^w \equiv x + y + 2^w \pmod{2^w} = [x + y]_{\text{补}} \pmod{2^w}$ ；

综上，可以证明 $[x]_{\text{补}} + [y]_{\text{补}} \equiv [x + y]_{\text{补}} \pmod{2^w}$ 。



3. 将 8 位无符号数 129 转换为 8 位浮点数 (exp 域宽度为 4 bits, frac 域宽度为 3bits)

Exp = ?

Frac = ?

- 首先, $129 = 1 * 2^7 + 1 * 2^1$, 因此10进制的129的8位二进制表示为10000001, 若要转为题目中的浮点数, 则二进制表示可以写作 $1.0000001 * 2^7$, 所以其浮点数表示中尾数M为000, 同时其浮点数表示中无偏置的阶码 $E = 7 + 2^3 - 1 = 14$ 。综上所述, 其浮点数表示为01110000。Exp=1110, Frac=000





3. 将 8 位无符号数 129 转换为 8

Exp = ?

Frac = ?

- 首先, $129 = 1 * 2^7$
十进制表示为 1000
, 则二进制表示
点数表示中尾数
偏置的阶码 $E = 7 +$
示为 01110000。

规格化浮点数 (Normalized)

▶ 满足条件: $\text{exp} \neq 000\dots 0$ 且 $\text{exp} \neq 111\dots 1$

▶ 真实的阶码值需要减去一个偏置 (biased) 量

$$E = \text{Exp} - \text{Bias}$$

- Exp : exp域所表示的无符号数值
- Bias的取值

- 单精度数: 127 (Exp: 1...254, E: -126...127)

- 双精度数: 1023 (Exp: 1...2046, E: -1022...1023)

- $\text{Bias} = 2^{e-1} - 1$, $e = \text{exp域的位数}$

▶ frac域的第一位隐含为1

$$M = 1.\text{xxx}\dots\text{x}_2$$

- 因此, 第一位的 "1" 可以省去, xxx...x: bits of frac
- Minimum when 000...0 ($M = 1.0$)
- Maximum when 111...1 ($M = 2.0 - \epsilon$)



```
/* Create some arbitrary values */  
int x = random();  
int y = random();  
int z = random();  
/* Convert to other forms */  
unsigned ux = (unsigned) x;  
unsigned uy = (unsigned) y;  
double dx = (double) x;  
double dy = (double) y;  
double dz = (double) z;
```



$(x < y) == (-x > -y)$ 并非恒成立。反例: $x=0, y=0x80000000 = -2147483648$ 。在计算机中 $-y == y$, 因此不成立

$((x+y) << 4) + y - x == 17*y + 15*x$ 恒成立, $\sim x = 2^w - x - 1$, $-x = \sim x + 1 = 2^w - x$, 左边 $= ((x+y) << 4) + y + (2^w - x)$ 左边 $= 15x + 17y$ 。即使考虑溢出, 则左右同时溢出。

$\sim x + \sim y + 1 == \sim(x+y)$ 恒成立, $-x = \sim x + 1$ 所以左边 $= \sim x + \sim y + 1 + 1 - 1 = (\sim x + 1) + (\sim y + 1) - 1 = -x - y - 1$ 右边 $= \sim(x+y) + 1 - 1 = -(x+y) - 1$

$ux - uy == -(y - x)$ 恒成立, 把 $-x = 2^w - x$ 代入右边, $-(y - x) = -(y + 2^w - x) = 2^w - (y + 2^w - x) = x - y$

$(x >= 0) || (x < ux)$ 并非恒成立。反例: $x=0x80000000 = -2147483648$ 。 $(x < ux)$ 按无符号比较, 注意 x 和 ux 的二进制串相同, 所以不成立

$((x >> 1) << 1) <= x$ 恒成立。按 $x >= 0$ 和 $x < 0$ 分类讨论即可

$(double)(float)x == (double)x$ 并非恒成立。反例: $x=0x7fffffff$

$dx + dy == (double)(y + x)$ 并非恒成立。反例: $x+y$ 溢出的情况

$dx + dy + dz = dz + dy + dx$ 恒成立。因为3个数字都是int 转double, double的精度足够表示int

1. One's complement of a

2. a.

3. a & b.

4. a * 7.

5. a / 4 .

6. (a < 0) ? 1 : -1 .

a. $\sim(\sim a \mid (b \wedge (\text{MIN_INT} + \text{MAX_INT})))$

b. $((a \wedge b) \& \sim b) \mid (\sim(a \wedge b) \& b)$

c. $1 + (a \ll 3) + \sim a$

d. $(a \ll 4) + (a \ll 2) + (a \ll 1)$

e. $((a < 0) ? (a + 3) : a) \gg 2$

f. $a \wedge (\text{MIN_INT} + \text{MAX_INT})$

g. $\sim((a \mid (\sim a + 1)) \gg W) \& 1$

h. $\sim((a \gg W) \ll 1)$

i. $a \gg 2$

1-f, 2-b, 3-a, 4-c, 5-e, 6-h

提示

f: $\text{MIN_INT} + \text{MAX_INT} = -1 = 0xffffffff$

c: $\sim a + 1 = -a$

//1's Complement: 反码, 即按位取反↵

//2's Complement: 补码↵



1. One's complement of a

a. $\sim(\sim a \mid (b \wedge (\text{MIN_INT} + \text{MAX_INT})))$

2. a.

b. $((a \wedge b) \& \sim b) \mid (\sim(a \wedge b) \& b)$

c. $1 + (a \ll 3) + \sim a$

3. a & b.

d. $(a \ll 4) + (a \ll 2) + (a \ll 1)$

e. $((a < 0) ? (a + 3) : a) \gg 2$

4. a * 7.

f. $a \wedge (\text{MIN_INT} + \text{MAX_INT})$

5. a / 4 .

g. $\sim((a \mid (\sim a + 1)) \gg W) \& 1$

h. $\sim((a \gg W) \ll 1)$

6. $(a < 0) ? 1 : -1$.

i. $a \gg 2$

1-f, 2-b, 3-a, 4-c, 5-e, 6-h

提示

f: $\text{MIN_INT} + \text{MAX_INT} = -1 = 0xffffffff$

c: $\sim a + 1 = -a$

$a^b \leftrightarrow (a \& \sim b) \mid (b \& \sim a)$

异或：不进位加法

题目中 a^b 看做一个整体

$(a^b)^b \rightarrow a$



1. One's complement of a

2. a.

3. a & b.

4. a * 7.

5. a / 4 .

6. (a < 0) ? 1 : -1 .

a. $\sim(\sim a \mid (b \wedge (\text{MIN_INT} + \text{MAX_INT})))$

b. $((a \wedge b) \& \sim b) \mid (\sim(a \wedge b) \& b)$

c. $1 + (a \ll 3) + \sim a$

d. $(a \ll 4) + (a \ll 2) + (a \ll 1)$

e. $((a < 0) ? (a + 3) : a) \gg 2$

f. $a \wedge (\text{MIN_INT} + \text{MAX_INT})$

g. $\sim((a \mid (\sim a + 1)) \gg W) \& 1$

h. $\sim((a \gg W) \ll 1)$

i. $a \gg 2$

1-f, 2-b, 3-a, 4-c, 5-e, 6-h

提示

f: $\text{MIN_INT} + \text{MAX_INT} = -1 = 0xffffffff$

c: $\sim a + 1 = -a$

$\sim(\sim a \mid \sim b)$



1. One's complement of a

2. a.

3. a & b.

4. a * 7.

5. a / 4 .

6. (a < 0) ? 1 : -1 .

a. $\sim(\sim a \mid (b \wedge (\text{MIN_INT} + \text{MAX_INT})))$

b. $((a \wedge b) \& \sim b) \mid (\sim(a \wedge b) \& b)$

c. $1 + (a \ll 3) + \sim a$

d. $(a \ll 4) + (a \ll 2) + (a \ll 1)$

e. $((a < 0) ? (a + 3) : a) \gg 2$

f. $a \wedge (\text{MIN_INT} + \text{MAX_INT})$

g. $\sim((a \mid (\sim a + 1)) \gg W) \& 1$

h. $\sim((a \gg W) \ll 1)$

i. $a \gg 2$

1-f, 2-b, 3-a, 4-c, 5-e, 6-h

提示

f: $\text{MIN_INT} + \text{MAX_INT} = -1 = 0xffffffff$

c: $\sim a + 1 = -a$



1. One's complement of a

a. $\sim(\sim a \mid (b \wedge (\text{MIN_INT} + \text{MAX_INT})))$

b. $((a \wedge b) \& \sim b) \mid (\sim(a \wedge b) \& b)$

c. $1 + (a \ll 3) + \sim a$

d. $(a \ll 4) + (a \ll 2) + (a \ll 1)$

e. $((a < 0) ? (a + 3) : a) \gg 2$

f. $a \wedge (\text{MIN_INT} + \text{MAX_INT})$

g. $\sim((a \mid (\sim a + 1)) \gg W) \& 1$

h. $\sim((a \gg W) \ll 1)$

i. $a \gg 2$

2. a.

3. a & b.

4. a * 7.

5. a / 4 .

6. $(a < 0) ? 1 : -1$.

1-f, 2-b, 3-a, 4-c, 5-e, 6-h

提示

f: $\text{MIN_INT} + \text{MAX_INT} = -1 = 0xffffffff$

c: $\sim a + 1 = -a$

带符号整数除以2的幂

◦ $x \gg k$ gives $\lfloor x / 2^k \rfloor$

◦ 采用算术右移

◦ 但是 $x < 0$ 时, 舍入错误

◦ Want $\lceil x / 2^k \rceil$ (需要向0舍入, 而不是向下舍入)

◦ Compute as $\lfloor (x + 2^k - 1) / 2^k \rfloor$

• In C: $(x + (1 \ll k) - 1) \gg k$

• Biases dividend toward 0



1. One's complement of a

2. a.

3. a & b.

4. a * 7.

5. a / 4 .

6. (a < 0) ? 1 : -1 .

a. $\sim(\sim a \mid (b \wedge (\text{MIN_INT} + \text{MAX_INT})))$

b. $((a \wedge b) \& \sim b) \mid (\sim(a \wedge b) \& b)$

c. $1 + (a \ll 3) + \sim a$

d. $(a \ll 4) + (a \ll 2) + (a \ll 1)$

e. $((a < 0) ? (a + 3) : a) \gg 2$

f. $a \wedge (\text{MIN_INT} + \text{MAX_INT})$

g. $\sim((a \mid (\sim a + 1)) \gg W) \& 1$

h. $\sim((a \gg W) \ll 1)$

i. $a \gg 2$

1-f, 2-b, 3-a, 4-c, 5-e, 6-h

提示

f: $\text{MIN_INT} + \text{MAX_INT} = -1 = 0xffffffff$

c: $\sim a + 1 = -a$

W is one less than the word length (e.g., W = 31 for 32-bit integers)

a>>w 带符号右移。让符号位填充所有位
(a>>w)<<1 最右边一位是0

如果a是负数, 0xffffffffe
如果a是非负数, 0x00000000
最后取反





6. 有如下的 C 代码，在 linux X86-64 系统下，生成的汇编代码如有下图，请填上缺失部分。

```
long arith2  
(long x, long y, long z)  
{  
    long t1 = x+z+y;  
    long t3 = x+4;  
    long t4 = y * 48;  
    long t5 = t3 + t4;  
    long rval = t1 * t5;  
    return rval;  
}
```

arith2:

```
    leaq    (%rsi,%rsi,2), %rcx  
    addq    %rdi, %rdx  
    addq    %rdx, %rsi  
    salq    $4, %rcx  
    leaq    4(%rdi,%rcx), %rax  
    imulq   %rsi, %rax  
    ret
```

x -> rdi

y -> rsi

z -> rdx

rcx=rsi*2+rsi=3y

rdx+=rdi =x+z

rsi+=rdx =y+x+z=t1

rcx*=16 = 48y

rax=4+rdi+rcx=4+x+48y=t5

rax*=rsi=t1*t5

返回值保存在rax中



有如下的 C 语言代码，以及编译生成的对应汇编代码，其中注释掉 if (likely (a == 2))这行生成汇编代码段-1，注释掉 if (unlikely (a == 2)) 这行生成汇编代码段-2。

问题：请简要分析编译指示（directives）

```
"#define likely(x)    __builtin_expect(!!(x), 1)
```

```
#define unlikely(x)  __builtin_expect(!!(x), 0)"
```

的作用——为何生成的指令序列的顺序不同，与处理器流水线的运行过程与优化有何关系？

```
#include<stdlib.h>
#define likely(x)    __builtin_expect(!!(x), 1)
#define unlikely(x)  __builtin_expect(!!(x), 0)
int main(char *argv[], int argc)
{
    int a,b;
    /* Get the value from somewhere GCC can't optimize */
    a = atoi (argv[1]);
    b = a*a;
    if (unlikely (a == 2))
    // if (likely (a == 2))
    {
        a++; b++;
    }
    else
    {
        a--; b--;
    }
    return a+b;
}
```



```

a = atoi (argv[1]);
b = a*a;
if (unlikely (a == 2))
// if (likely (a == 2))
{
    a++; b++;
}
else
{
    a--; b--;
}
return a+b;

```

代码段-1

main:

```

subq    $8, %rsp      栈顶减小 (栈扩大)
movq    8(%rdi), %rdi 将%rdi赋值为argv[1]?
xorl    %esi, %esi    将%esi赋值为0
movl    $10, %edx     将%edx赋值为10
call    strtol        # atoi 调用, 返回值在 eax 中
movl    %eax, %esi    将a的值移至%esi (a)
movl    $3, %ecx      %ecx赋值为3
imull    %eax, %esi    %esi乘上%eax (b=a*a)
cmpl    $2, %eax      计算a-2的值
leal    1(%rsi), %edx  %edx赋值为b+1
je      .L3           如果a==2转.L3
leal    -1(%rax), %ecx %ecx赋值为a-1
leal    -1(%rsi), %edx %edx赋值为b-1

```

.L3:

```

leal    (%rcx,%rdx), %eax (1) a==2, %eax赋值为b+4, 也即b+1+a+1
addq    $8, %rsp          (2) a!=2, %eax赋值为(a-1)+(b-1)
ret     返回

```

将栈恢复原位

```
a = atoi (argv[1]);
b = a*a;
```

代码段-2

```
    if (likely (a == 2))
    {
        a++; b++;
    }
    else
    {
        a--; b--;
    }
    return a+b;
```

main:

```
subq    $8, %rsp
movq    8(%rdi), %rdi
xorl    %esi, %esi
movl    $10, %edx
call    strtol
movl    %eax, %ecx 将%ecx赋值为%eax (也就是a)
imull   %eax, %ecx %ecx的值变为a*a (也就是b)
cmpl    $2, %eax 计算a-2的值
jne     .L2 如果a!=2则跳转至. L2
leal    1(%rcx), %eax %eax赋值为b+1
movl    $3, %edx %edx赋值为3
```

.L3:

```
addl    %edx, %eax (1) a==2: %eax为3+b+1=b+1+a+1
addq    $8, %rsp (2) a!=2: %eax为a-1+b-1
ret
```

.L2:

```
leal    -1(%rax), %edx %edx赋值为a-1
leal    -1(%rcx), %eax %eax赋值为b-1
jmp     .L3
```



- 1, 查阅相关资料, 说明__builtin_expect的作用。
- 2, 结合代码段1, 2说明likely和unlikely的区别, 并说明重要的几句汇编代码的含义(a, b自增、自减以及跳转语句)。
- 3, 结合流水线说明两段代码的作用。

其中, 许多同学的回答不够准确:

因为代码段1中a不等于2的概率大, 所以将else分支放到顺序执行的位置, ...
因为代码段2中a等于2的概率大, 所以将if分支放到顺序执行的位置, ...

以下表述也不够准确:

__builtin_expect(!!(x), 1) / likely(x) 表示x的值为真的可能性更大;
__builtin_expect(!!(x), 0) / unlikely(x) 表示x的值为假的可能性更大。

正确的理解应该是, 如果经过大量的真实数据结果表明、或是程序员认为, 在这里x取真的可能性很大, 那么通过使用likely(x), 可以通过__builtin_expect来告诉编译器, 在生成汇编代码时尽可能把x取真值的分支放到顺序执行的位置, 减少不必要的跳转, 增加流水线的执行效率。如果程序员认为执行到这里, x大部分时候取假, 那么使用unlikely的效率则会更高。“程序员认为x取真/假可能性高”作为前提, 是需要强调的。



问题：请简要分析编译指示

```
#define likely(x)    __builtin_expect((x),1)
#define unlikely(x)  __builtin_expect((x),0)
```

的作用——为何生成的指令序列的顺序不同，与处理器流水线的运行过程与优化有何关系？

答：__builtin_expect (long exp, long c)函数：

__builtin_expect(exp, c)接受两个 long 型的参数，用来告诉 gcc：exp==c 的可能性比较大。例如，__builtin_expect(exp, 1) 表示程序执行过程中，exp 取到 1 的可能性比较大。该函数的返回值为 exp 自身。

由此可知内核中 likely(x) 和 unlikely(x) 宏的作用：

likely(x) 等价于 x，即 if(likely(x)) 等价于 if(x)，但是它告诉 gcc，x 取 1 的可能性比较大。

unlikely(x) 等价于 x，即 if(unlikely(x)) 等价于 if(x)，但是它告诉 gcc，x 取 0 的可能性比较大。

使用 __builtin_expect (long exp, long c) 函数可以帮助 gcc 优化程序编译后的指令序列，使汇编指令尽可能的顺序执行，从而提高 CPU 预取指令的正确率和执行效率。

该函数用来引导 gcc 进行条件分支预测。在一条指令执行时，由于流水线的作用，CPU 可以同时完成下一条指令的读取，这样可以提高 CPU 的利用率。在执行条件分支指令时，CPU 也会预取下一条执行，但是如果条件分支的结果为跳转到了其他指令，那 CPU 预取的下一条指令就没用了，这样就降低了流水线的效率。另外，跳转指令相对于顺序执行的指令会多消耗 CPU 时间，如果可以尽可能不执行跳转，也可以提高 CPU 性能。

3个问题，需要分别回答

参考答案1



编译指示的作用:

`__builtin_expect((x), 1)` 表示 x 为真的可能性更大; `__builtin_expect((x), 0)` 表示 x 为假的可能性更大。

因此代码中若使用 `if(unlikely(a == 2))`, 则 `else` 分支更有可能被执行; 若使用 `if(likely(a == 2))`, 则 `if` 后面的分支更有可能被执行。

通过这样的编译指示可以显示地告知编译器如何优化。

生成的指令顺序不同的原因:

编译器根据编译指示减少代码段执行条件跳转指令的可能性。

如本题中汇编代码段-1 (`unlikely a == 2`) 中条件跳转指令为 `je .L3` 表示如果 $a == 0$ 则跳转; 而汇编代码段-2 (`likely a == 2`) 中条件跳转指令为 `jne .L2` 表示如果 $a != 0$ 则跳转。

编译器还可能进行分支预测, 即在判断前就计算某一分支的内容, 储存在某一寄存器中。进行判断后, 若与预测结果一致, 则把先前计算内容移动到 `%rax` (或 `%eax`) 中; 否则重新计算正确分支的结果, 先前的计算结果被舍弃。

如本题中汇编代码段-1 在判断前, 进行 `movl $3, %ecx` 和 `leal 1(%rsi), %edx` 指令, 表示在 `%ecx` 中储存了 3 ($a=2$ 时的 $a+1$), `%edx` 中储存了 $b+1$ 的值, 表示分支预测时预测了 $a == 2$ 的分支。

与处理器流水线运行过程与优化的关系:

由于处理器是流水线运行的, 系统可以提前取多条指令进行并行处理, 所以在判断得到结果前, 处理器可能已经提前取出后面的多条指令进行处理。

如果判断结果表明需要执行跳转指令, 则先前取出的指令无效, 而需要重新取指令, 这样效率会降低。

如果使用 `__builtin_expect((x), 1)` 和 `__builtin_expect((x), 0)` 可以增加分支预测的准确性, 提高 CPU 效率

参考答案2





- Thank You !
- Any Questions ?



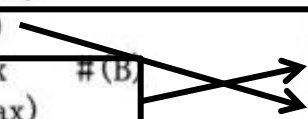
A decorative graphic featuring stylized green leaves and swirling vines, positioned horizontally across the upper middle of the slide.

汇编作业讲解2

费翔 feix16@mails.tsinghua.edu.cn
2021.8

- 过程调用以及返回的顺序在一般情况下都是“过程返回的顺序恰好与调用顺序相反”，但是我们可以利用汇编以及对运行栈的理解来编写汇编过程打破这一惯例。有如下汇编代码（x86-32 架构），其中 **GET 过程唯一的输入参数是一个用于存储当前处理器以及栈信息的内存块地址**（假设该内存块的空间足够大），而 SET 过程则用于恢复被 GET 过程所保存的处理器及栈信息，**其唯一的输入参数也是该内存块地址**。在理解代码的基础上，回答下列问题

GET:	SET:
<code>movl 4(%esp), %eax # (A)</code>	<code>movl 4(%esp), %eax</code>
<code>...</code>	<code>...</code>
<code>movl %edi, 20(%eax)</code>	<code>movl 20(%eax), %edi</code>
<code>movl %esi, 24(%eax)</code>	<code>movl 24(%eax), %esi</code>
<code>movl %ebp, 28(%eax)</code>	<code>movl 28(%eax), %ebp</code>
<code>movl %ebx, 36(%eax)</code>	<code>movl 36(%eax), %ebx</code>
<code>movl %edx, 40(%eax)</code>	<code>movl 40(%eax), %edx</code>
<code>movl %ecx, 44(%eax)</code>	<code>movl 44(%eax), %ecx</code>
<code>movl \$1, 48(%eax)</code>	<code>movl ____(%eax), %esp # (D)</code>
<code>movl (%esp), %ecx # (B)</code>	<code>pushl 60(%eax) # (E)</code>
<code>movl %ecx, 60(%eax)</code>	<code>movl 48(%eax), %eax</code>
<code>leal 4(%esp), %ecx # (C)</code>	<code>ret</code>
<code>movl %ecx, 72(%eax)</code>	
<code>movl 44(%eax), %ecx</code>	
<code>movl \$0, %eax</code>	
<code>ret</code>	



GET:	SET:
movl 4(%esp), %eax #(A)	movl 4(%esp), %eax
...	...
movl %edi, 20(%eax)	movl 20(%eax), %edi
movl %esi, 24(%eax)	movl 24(%eax), %esi
movl %ebp, 28(%eax)	movl 28(%eax), %ebp
movl %ebx, 36(%eax)	movl 36(%eax), %ebx
movl %edx, 40(%eax)	movl 40(%eax), %edx
movl %ecx, 44(%eax)	movl 44(%eax), %ecx
movl \$1, 48(%eax)	movl (%eax), %esp #(D)
movl (%esp), %ecx #(B)	pushl 60(%eax) #(E)
movl %ecx, 60(%eax)	movl 48(%eax), %eax
leal 4(%esp), %ecx #(C)	ret
movl %ecx, 72(%eax)	
movl 44(%eax), %ecx	
movl \$0, %eax	
ret	

- SET 过程的返回地址是什么？
【rsp指向的内容就是返回地址（谁改变了rsp）。就是get的返回地址，本来保存在(%esp)中】
- 其返回值是多少？【1，保存在%eax中】



GET:	SET:
movl 4(%esp), %eax # (A)	movl 4(%esp), %eax
...	...
movl %edi, 20(%eax)	movl 20(%eax), %edi
movl %esi, 24(%eax)	movl 24(%eax), %esi
movl %ebp, 28(%eax)	movl 28(%eax), %ebp
movl %ebx, 36(%eax)	movl 36(%eax), %ebx
movl %edx, 40(%eax)	movl 40(%eax), %edx
movl %ecx, 44(%eax)	movl 44(%eax), %ecx
movl \$1, 48(%eax)	movl ____(%eax), %esp # (D)
movl (%esp), %ecx # (B)	pushl 60(%eax) # (E)
movl %ecx, 60(%eax)	movl 48(%eax), %eax
leal 4(%esp), %ecx # (C)	ret
movl %ecx, 72(%eax)	
movl 44(%eax), %ecx	
movl \$0, %eax	
ret	

movl 4(%esp), %eax # (A)
 eax=mem[esp+4]
 leal 4(%esp), %ecx # (C)
 ecx=esp+4

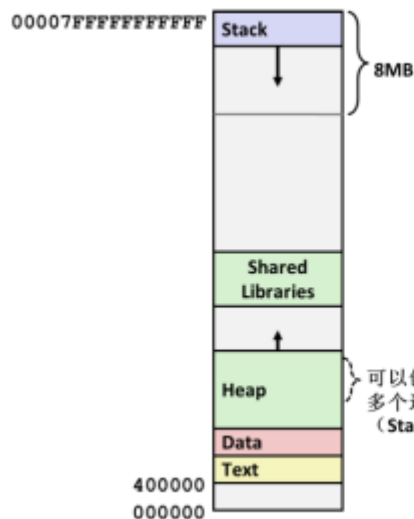
- 代码段中的 (A) 指令执行后，eax 中存放的是什么？【get 的唯一参数，内存块的首地址】
- (B) 指令执行后，ecx 中存放的是什么？【get 的返回地址】
- (C) 指令的作用是什么？【恢复 esp，或者说，保存调用 get 前的栈顶指针】（为什么 +4？后面有 push，隐含 -4）
- (E) 指令的作用是什么？【把 get 的返回地址压入栈，当做 set 的返回地址】
- 并将 (D) 指令补充完整【72】





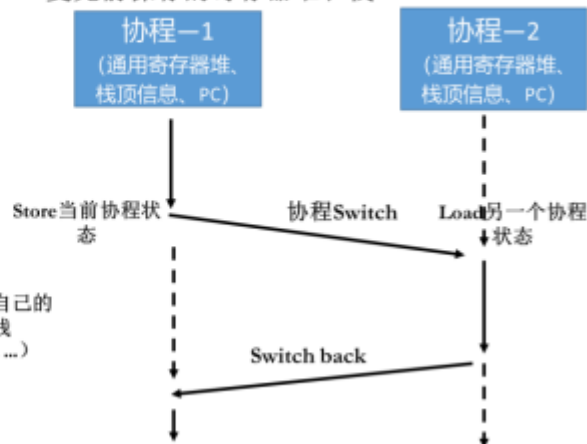
2. 上述的思路也可用于“协程”任务切换（上课讲过，PPT 加下图），

Linux进程的内存布局 (x86-64)



协程：轻量级任务（OS不可见）

- 拥有**自己的寄存器堆和栈**，协程调度切换时，将寄存器堆和栈保存起来，在切换回来时恢复先前保存的寄存器堆和栈



切换协程用的过程的代码如下：

```
ribs_swapcurcontext:
```

```
    movq    current_ctx, %rsi
```

```
    /* Save the preserved registers. */
```

```
    movq    %rsp, 0(%rsi)
```

```
    movq    %rbx, 8(%rsi)
```

```
    movq    %rbp, 16(%rsi)
```

```
    movq    %r12, 24(%rsi)
```

```
    movq    %r13, 32(%rsi)
```

```
    movq    %r14, 40(%rsi)
```

```
    movq    %r15, 48(%rsi)
```

```
    movq    %rdi, current_ctx
```

```
    /* Load the new stack pointer and the preserved registers.*/
```

```
    movq    0(%rdi), %rsp
```

```
    movq    8(%rdi), %rbx
```

```
    movq    16(%rdi), %rbp
```

```
    movq    24(%rdi), %r12
```

```
    movq    32(%rdi), %r13
```

```
    movq    40(%rdi), %r14
```

```
    movq    48(%rdi), %r15
```

```
    ret
```

将寄存器堆和栈保存起来

恢复先前保存的
寄存器堆和栈

- 2.1 请简要介绍其工作原理；
- rsi是当前协程（旧协程）对应的内存地址；rdi是切换的目标协程（新协程）对应的内存地址。首先保存7个寄存器的值，然后把current_ctx变成切换的目标协程，并从rdi中恢复目标协程的寄存器值，然后正常返回。
- 旧协程的寄存器数据被转移到内存中保存，新协程的数据从内存中取出保存到了寄存器上，完成协程的切换。



切换协程用的过程的代码如下：

```
ribs_swapcurcontext:
```

```
    movq    current_ctx, %rsi
```

```
    /* Save the preserved registers. */
```

```
    movq    %rsp, 0(%rsi)
```

```
    movq    %rbx, 8(%rsi)
```

```
    movq    %rbp, 16(%rsi)
```

```
    movq    %r12, 24(%rsi)
```

```
    movq    %r13, 32(%rsi)
```

```
    movq    %r14, 40(%rsi)
```

```
    movq    %r15, 48(%rsi)
```

```
    movq    %rdi, current_ctx
```

```
    /* Load the new stack pointer and the preserved registers. */
```

```
    movq    0(%rdi), %rsp
```

```
    movq    8(%rdi), %rbx
```

```
    movq    16(%rdi), %rbp
```

```
    movq    24(%rdi), %r12
```

```
    movq    32(%rdi), %r13
```

```
    movq    40(%rdi), %r14
```

```
    movq    48(%rdi), %r15
```

```
    ret
```

- 2.2 为何 save/load 的通用寄存器个数这么少（x86-64 有 16 个通用寄存器）？

如何使用寄存器作为程序的临时存储？

• 使用惯例——通用寄存器分为两类

- “调用者负责保存”

- Caller 在调用子过程之前将这些寄存器内容存储在它的栈帧内

- “被调用者负责保存”

- Callee 在使用这些寄存器之前将其原有内容存储在它的栈帧内
- 退出前恢复

■ x86-64 寄存器使用惯例


%rax	Return Value	%r8	Argument #5
%rbx	Callee Saved	%r9	Argument #6
%rcx	Argument #4	%r10	Caller Saved
%rdx	Argument #3	%r11	Caller Saved
%rsi	Argument #2	%r12	Callee Saved
%rdi	Argument #1	%r13	Callee Saved
%rsp	Stack Pointer	%r14	Callee Saved
%rbp	Callee Saved	%r15	Callee Saved

栈帧指针(可选)

3. 请对照下列的 C 代码与对应的汇编代码, 解释下 C 函数返回 struct 类型是如何实现的? 可以通过画出 call return_struct 时栈的 layout 以及传参情况, 并辅以说明来解释。

```
typedef struct{
int age; int bye; int coo; int ddd; int eee;
} TEST_Struct;
int i = 2;
TEST_Struct __cdecl return_struct(int n)
{
    TEST_Struct local_struct;
    local_struct.age = n;
    local_struct.bye = n;
    local_struct.coo = 2*n;
    local_struct.ddd = n;
    local_struct.eee = n;
    i = local_struct.eee + local_struct.age *2 ;
    return local_struct;
}
int function1()
{
    TEST_Struct main_struct = return_struct(i);
    return 0;
}
```

```
return_struct:
    movq    %rdi, %rax
    movl    %esi, (%rdi)
    movl    %esi, 4(%rdi)
    leal    (%rsi,%rsi), %edx
    movl    %edx, 8(%rdi)
    movl    %esi, 12(%rdi)
    movl    %esi, 16(%rdi)
    addl    %edx, %esi
    movl    %esi, i(%rip)
    ret
function1:
    subq    $32, %rsp
    movl    i(%rip), %esi
    movq    %rsp, %rdi
    call    return_struct
    movl    $0, %eax
    addq    $32, %rsp
    ret
```



return_struct:

```
    movq    %rdi, %rax
    movl    %esi, (%rdi)
    movl    %esi, 4(%rdi)
    leal    (%rsi,%rsi), %edx
    movl    %edx, 8(%rdi)
    movl    %esi, 12(%rdi)
    movl    %esi, 16(%rdi)
    addl    %edx, %esi
    movl    %esi, i(%rip)
    ret
```

function1:

```
    subq    $32, %rsp
    movl    i(%rip), %esi
    movq    %rsp, %rdi
    call    return_struct
    movl    $0, %eax
    addq    $32, %rsp
    ret
```


Func1的返回地址

高地址

rsp

低地址





return_struct:

```
movq    %rdi, %rax
movl    %esi, (%rdi)
movl    %esi, 4(%rdi)
leal    (%rsi,%rsi), %edx
movl    %edx, 8(%rdi)
movl    %esi, 12(%rdi)
movl    %esi, 16(%rdi)
addl    %edx, %esi
movl    %esi, i(%rip)
ret
```

function1:

```
subq    $32, %rsp
movl    i(%rip), %esi
movq    %rsp, %rdi
call    return_struct
movl    $0, %eax
addq    $32, %rsp
ret
```

Func1的返回地址

高地址

rsp

低地址



return_struct:

```
movq    %rdi, %rax
movl    %esi, (%rdi)
movl    %esi, 4(%rdi)
leal    (%rsi,%rsi), %edx
movl    %edx, 8(%rdi)
movl    %esi, 12(%rdi)
movl    %esi, 16(%rdi)
addl    %edx, %esi
movl    %esi, i(%rip)
ret
```

function1:

```
subq    $32, %rsp
movl    i(%rip), %esi
movq    %rsp, %rdi
call    return_struct
movl    $0, %eax
addq    $32, %rsp
ret
```

把i的值放入esi

Func1的返回地址

高地址

rsp

低地址



return_struct:

```
movq    %rdi, %rax
movl    %esi, (%rdi)
movl    %esi, 4(%rdi)
leal    (%rsi,%rsi), %edx
movl    %edx, 8(%rdi)
movl    %esi, 12(%rdi)
movl    %esi, 16(%rdi)
addl    %edx, %esi
movl    %esi, i(%rip)
ret
```

function1:

```
subq    $32, %rsp
movl    i(%rip), %esi
movq    %rsp, %rdi
call    return_struct
movl    $0, %eax
addq    $32, %rsp
ret
```

把栈顶指针rsp
放入rdi, 当做
ret函数的参数

Func1的返回地址

高地址

rsp rdi
低地址

return_struct:

```
movq    %rdi, %rax
movl    %esi, (%rdi)
movl    %esi, 4(%rdi)
leal    (%rsi,%rsi), %edx
movl    %edx, 8(%rdi)
movl    %esi, 12(%rdi)
movl    %esi, 16(%rdi)
addl    %edx, %esi
movl    %esi, i(%rip)
ret
```

function1:

```
subq    $32, %rsp
movl    i(%rip), %esi
movq    %rsp, %rdi
call    return_struct
movl    $0, %eax
addq    $32, %rsp
ret
```

把下一条指令的地址压入栈

Func1的返回地址

ret_str函数的返回地址

高地址

rdi

rsp

低地址

return_struct:

```
    movq    %rdi, %rax
    movl    %esi, (%rdi)
    movl    %esi, 4(%rdi)
    leal    (%rsi,%rsi), %edx
    movl    %edx, 8(%rdi)
    movl    %esi, 12(%rdi)
    movl    %esi, 16(%rdi)
    addl    %edx, %esi
    movl    %esi, i(%rip)
    ret

function1:
    subq    $32, %rsp
    movl    i(%rip), %esi
    movq    %rsp, %rdi
    call    return_struct
    movl    $0, %eax
    addq    $32, %rsp
    ret
```

返回值是结构的首地址，用rax保存。虽然后面没用到

→ 结构体赋值

Func1的返回地址

eee

ddd

coo

bye

age

ret_str函数的返回地址

高地址

```
local_struct.age = n;
local_struct.bye = n;
local_struct.coo = 2*n;
local_struct.ddd = n;
local_struct.eee = n;
i = local_struct.eee
  + local_struct.age *2 ;
```

rdi

rsp

低地址

return_struct:

```
movq    %rdi, %rax
movl    %esi, (%rdi)
movl    %esi, 4(%rdi)
leal    (%rsi,%rsi), %edx
movl    %edx, 8(%rdi)
movl    %esi, 12(%rdi)
movl    %esi, 16(%rdi)
addl    %edx, %esi
movl    %esi, i(%rip)
ret
```

function1:

```
subq    $32, %rsp
movl    i(%rip), %esi
movq    %rsp, %rdi
call    return_struct
movl    $0, %eax
addq    $32, %rsp
ret
```

→ 计算i并保存到全局数据空间

Func1的返回地址

eee

ddd

coo

bye

age

ret_str函数的返回地址

高地址

```
local_struct.age = n;
local_struct.bye = n;
local_struct.coo = 2*n;
local_struct.ddd = n;
local_struct.eee = n;
i = local_struct.eee
  + local_struct.age *2 ;
```

rdi

rsp

低地址

return_struct:

```
movq    %rdi, %rax
movl    %esi, (%rdi)
movl    %esi, 4(%rdi)
leal    (%rsi,%rsi), %edx
movl    %edx, 8(%rdi)
movl    %esi, 12(%rdi)
movl    %esi, 16(%rdi)
addl    %edx, %esi
movl    %esi, i(%rip)
```

ret

function1:

```
subq    $32, %rsp
movl    i(%rip), %esi
movq    %rsp, %rdi
call    return_struct
movl    $0, %eax
addq    $32, %rsp
ret
```

Func1的返回地址

eee

ddd

coo

bye

age

ret_str函数的返回地址

高地址

rdi

rsp

低地址

return_struct:

```
movq    %rdi, %rax
movl    %esi, (%rdi)
movl    %esi, 4(%rdi)
leal    (%rsi,%rsi), %edx
movl    %edx, 8(%rdi)
movl    %esi, 12(%rdi)
movl    %esi, 16(%rdi)
addl    %edx, %esi
movl    %esi, i(%rip)
ret
```

function1:

```
subq    $32, %rsp
movl    i(%rip), %esi
movq    %rsp, %rdi
call    return_struct
movl    $0, %eax
addq    $32, %rsp
ret
```

Func1的返回地址

eee

ddd

coo

bye

age

ret_str函数的返回地址

高地址

rsp rdi

低地址

return_struct:

```
movq    %rdi, %rax
movl    %esi, (%rdi)
movl    %esi, 4(%rdi)
leal    (%rsi,%rsi), %edx
movl    %edx, 8(%rdi)
movl    %esi, 12(%rdi)
movl    %esi, 16(%rdi)
addl    %edx, %esi
movl    %esi, i(%rip)
ret
```

function1:

```
subq    $32, %rsp
movl    i(%rip), %esi
movq    %rsp, %rdi
call    return_struct
movl    $0, %eax
addq    $32, %rsp
ret
```

后来func1函数
没有用到结构
体，自顾自返
回了

Func1的返回地址

eee

ddd

coo

bye


age

ret_str函数的返回地址

高地址

rsp rdi

低地址



3. 请对照下列的 C 代码与对应的汇编代码, 解释下 C 函数返回 struct 类型是如何实现的? 可以通过画出 `call return_struct` 时栈的 layout 以及传参情况, 并辅以说明来解释。

- 父函数在自己的栈中开辟用于保存结构体的空间, 把结构体的首地址传给子函数, 子函数把返回值保存在该地址中



4. 请分别对照下列的 C 代码与对应的汇编代码, 解释下 C 函数是如何传入 struct 类型参数的? 可以通过画出 call input_struct 时栈的 layout, 并辅以说明来解释。

4.1 gcc -Og ...

```
typedef struct{
int age; int bye; int coo; int ddd; int eee;
} TEST_Struct;
int i = 2;
int input_struct(TEST_Struct in_struct)
{
    return in_struct.eee + in_struct.age*2 ;
}
int function2()
{
    TEST_Struct main_struct;
    main_struct.age = i;
    main_struct.bye = i;
    main_struct.coo = 2*i;
    main_struct.ddd = i;
    main_struct.eee = i;
    return input_struct(main_struct);
}
```

input_struct:

```
movl    8(%rsp), %eax    #age
addl    %eax, %eax
addl    24(%rsp), %eax   #eee
ret
```

function2:

```
subq    $56, %rsp
movl    i(%rip), %eax
movl    %eax, 24(%rsp)   #age
movl    %eax, 28(%rsp)   #bye
leal    (%rax,%rax), %edx
movl    %edx, 32(%rsp)   #coo
movl    %eax, 36(%rsp)   #ddd
movq    24(%rsp), %rdx
movq    %rdx, (%rsp)     #age/bye
movq    32(%rsp), %rdx
movq    %rdx, 8(%rsp)    #coo/ddd
movl    %eax, 16(%rsp)   #eee
call    input_struct
addq    $56, %rsp
ret
```



input_struct:

```
movl    8(%rsp), %eax    #age
addl    %eax, %eax
addl    24(%rsp), %eax    #eee
ret
```

function2:

```
subq    $56, %rsp
movl    i(%rip), %eax

movl    %eax, 24(%rsp)    #age
movl    %eax, 28(%rsp)    #bye
leal    (%rax,%rax), %edx
movl    %edx, 32(%rsp)    #coo
movl    %eax, 36(%rsp)    #ddd

movq    24(%rsp), %rdx
movq    %rdx, (%rsp)      #age/bye

movq    32(%rsp), %rdx
movq    %rdx, 8(%rsp)     #coo/ddd

movl    %eax, 16(%rsp)    #eee
call    input_struct
addq    $56, %rsp
ret
```

func自己的

func复制后给
子函数的

Func1的返回地址

~~eee~~

ddd

coo

bye

age

...

eee

coo/ddd

age/bye

高地址

rsp+24

rsp+16

rsp+8

rsp

低地址



input_struct:

```
movl    8(%rsp), %eax    #age
addl    %eax, %eax
addl    24(%rsp), %eax    #eee
ret
```

function2:

```
subq    $56, %rsp
movl    i(%rip), %eax
movl    %eax, 24(%rsp)    #age
movl    %eax, 28(%rsp)    #bye
leal    (%rax,%rax), %edx
movl    %edx, 32(%rsp)    #coo
movl    %eax, 36(%rsp)    #ddd
movq    24(%rsp), %rdx
movq    %rdx, (%rsp)      #age/bye
movq    32(%rsp), %rdx
movq    %rdx, 8(%rsp)     #coo/ddd
movl    %eax, 16(%rsp)    #eee
call    input_struct
addq    $56, %rsp
ret
```

func自己的

func复制后给
子函数的

Func1的返回地址

~~eee~~

ddd

coo

bye

age

...

eee

coo/ddd

age/bye

高地址

rsp+24

rsp+16

rsp+8

rsp



input_struct:

```
movl    8(%rsp), %eax    #age
addl    %eax, %eax
addl    24(%rsp), %eax    #eee
ret
```

```
return in_struct.eee + in_struct.age*2 ;
```

function2:

```
subq    $56, %rsp
movl    i(%rip), %eax
movl    %eax, 24(%rsp)    #age
movl    %eax, 28(%rsp)    #bye
leal    (%rax,%rax), %edx
movl    %edx, 32(%rsp)    #coo
movl    %eax, 36(%rsp)    #ddd
movq    24(%rsp), %rdx
movq    %rdx, (%rsp)      #age/bye
movq    32(%rsp), %rdx
movq    %rdx, 8(%rsp)     #coo/ddd
movl    %eax, 16(%rsp)    #eee
call    input_struct
addq    $56, %rsp
ret
```

func自己的

func复制后给
予函数的

Func1的返回地址

~~eee~~

ddd

coo

bye

age

...

eee

coo/ddd

age/bye

input函数的返回地址

高地址

新值 旧值

rsp+32 rsp+24

rsp+24 rsp+16

rsp+16 rsp+8

rsp+8 rsp




4. 请分别对照下列的 C 代码与对应的汇编代码，解释下 C 函数是如何传入 struct 类型参数的？可以通过画出 call input_struct 时栈的 layout，并辅以说明来解释。

4.1 gcc -Og ...

- 父函数在自己的栈中复制了一份结构体，保存在 `rsp` 堆栈的栈顶，子函数从 `rsp` 堆栈中读取数据





4.2 gcc -O1/2 ...

C 代码不变。汇编如下：

```
input_struct:
    movl    24(%rsp), %eax
    movl    8(%rsp), %edx
    leal    (%rax,%rdx,2), %eax
    ret
function2:
    movl    i(%rip), %eax
    leal    (%rax,%rax,2), %eax
    ret
```

请分析这段代码，编译器做了什么优化工作。

- 父函数不调用input子函数，编译器理解子函数语义后在父函数本地实现了子函数的功能，类似于inline内联函数的实现



4.3 如果在上面的C代码的 `int input_struct (...)` 声明前加上 `static`, `gcc`

-O1/2 ... 编译后的代码如下:

```
function2:
    movl    i(%rip), %eax
    leal    (%rax,%rax,2), %eax
    ret
```

- 请分析这段代码，编译器做了什么优化工作？
- 在函数 `input_struct` 声明前加上 `static`，即此函数只在该源码文件中可见，不会有其他地方调用它。编译器调用 `gcc -O1/2` 时 `input_struct` 相当于内联函数，其逻辑直接在 `function2` 中实现，所以子函数没有出现在汇编代码中。



5. 请分别对照下列的 C 代码与对应的汇编代码(编译开关: -S -O2 -fno-stack-protector), 解释下 C99 标准中引入的 variable-length array (简称 VLA, 即允许使用变量定义数组各维) 在这一块代码中是如何实现的? 可以画出函数运行时的栈 layout, 并辅以说明来解释。

read_and_process:

```
pushq    %rbp
movslq   %edi, %rax
leaq     22(,%rax,8), %rax
movq     %rsp, %rbp
pushq    %r14
pushq    %r13
pushq    %r12
pushq    %rbx
andq     $-16, %rax
leal     -1(%rdi), %r13d
subq     %rax, %rsp
testl    %edi, %edi
movq     %rsp, %r14
jle      .L4
leal     -1(%rdi), %eax
movq     %rsp, %rbx
leaq     8(%rsp,%rax,8), %r12
movq     %rax, %r13
```

.L3:

```
xorl     %eax, %eax
addq     $8, %rbx
call     read_val
cltq     # sign-extend eax to all of rax
movq     %rax, -8(%rbx)
cmpq     %r12, %rbx
jne      .L3
```

.L4:

```
movslq   %r13d, %r13
movq     (%r14,%r13,8), %rax
leaq     -32(%rbp), %rsp
popq     %rbx
popq     %r12
popq     %r13
popq     %r14
popq     %rbp
ret
```

```
long read_and_process(int n)
{
    long vals[n];

    for (int i = 0; i < n; i++)
        vals[i] = read_val();
    return vals[n-1];
}
```

rsp



高地址

低地址



5. 请分别对照下列的C代码与对应的汇编代码(编译开关: -S -O2 -fno-stack-protector), 解释下 C99 标准中引入的 variable-length array (简称 VLA, 即允许使用变量定义数组各维) 在这一块代码中是如何实现的? 可以画出函数运行时的栈 layout, 并辅以说明来解释。

read_and_process:

```
pushq    %rbp
movslq   %edi, %rax
leaq     22(%rax,8), %rax
movq     %rsp, %rbp
pushq    %r14
pushq    %r13
pushq    %r12
pushq    %rbx
andq     $-16, %rax
leal     -1(%rdi), %r13d
subq     %rax, %rsp
testl    %edi, %edi
movq     %rsp, %r14
jle      .L4
leal     -1(%rdi), %eax
movq     %rsp, %rbx
leaq     8(%rsp,%rax,8), %r12
movq     %rax, %r13
```

edi是输入参数n
8n是数组vals的字节长度
多加22个可能用于其他用途

直接跳过整个循环

进入循环前的准备

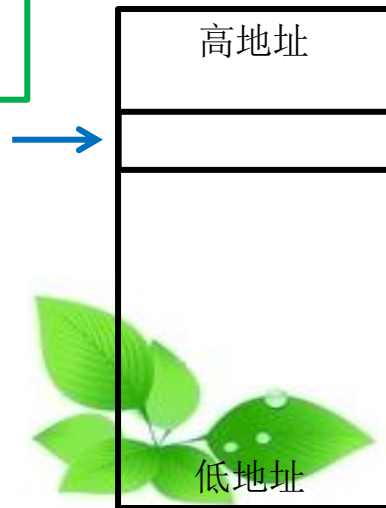
```
.L3:
xorl     %eax, %eax
addq     $8, %rbx
call     read_val
cltq     # sign-extend eax to all of rax
movq     %rax, -8(%rbx)
cmpq     %r12, %rbx
jne      .L3

.L4:
movslq   %r13d, %r13
movq     (%r14,%r13,8), %rax
leaq     -32(%rbp), %rsp
popq     %rbx
popq     %r12
popq     %r13
popq     %r14
popq     %rbp
ret
```

```
long read_and_process(int n)
{
    long vals[n];

    for (int i = 0; i < n; i++)
        vals[i] = read_val();
    return vals[n-1];
}
```

明显是循环体



5. 请分别对照下列的C代码与对应的汇编代码(编译开关: -S -O2 -fno-stack-protector), 解释下 C99 标准中引入的 variable-length array (简称 VLA, 即允许使用变量定义数组各维) 在这一块代码中是如何实现的? 可以画出函数运行时的栈 layout, 并辅以说明来解释。

read_and_process:

```
pushq    %rbp                                .L3:
movslq   %edi, %rax                          edi是输入参数n
leaq     22(,%rax,8), %rax                   8n是数组vals的字节长度
                                                多加22个可能用于其他用途
movq     %rsp, %rbp
pushq    %r14
pushq    %r13
pushq    %r12
pushq    %rbx
andq     $-16, %rax
leal     -1(%rdi), %r13d
→ subq    %rax, %rsp                        在栈上开辟一块8n的空间
testl    %edi, %edi
movq     %rsp, %r14
jle      .L4
leal     -1(%rdi), %eax
movq     %rsp, %rbx                          把vals的起始地址给rbx
leaq     8(%rsp,%rax,8), %r12
movq     %rax, %r13
```

```
xorl     %eax, %eax
addq     $8, %rbx
call     read_val
cltq     # sign-extend
movq     %rax, -8(%rbx)
cmpq     %r12, %rbx                          退出for循环的条件
jne      .L3                                是当前计算的地址到达vals[n], 而非
                                                i<n, 两者等价
```

```
movslq   %r13d, %r13
movq     (%r14,%r13,8), %rax
leaq     -32(%rbp), %rsp
popq     %rbx
popq     %r12
popq     %r13
popq     %r14
popq     %rbp
ret
```

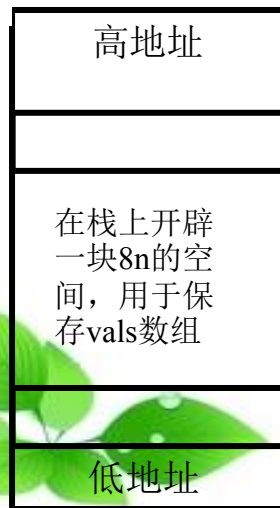
```
long read_and_process(int n)
{
    long vals[n];

    for (int i = 0; i < n; i++)
        vals[i] = read_val();
    return vals[n-1];
}
```

退出for循环的条件
是当前计算的地址到
达vals[n], 而非
i<n, 两者等价

old rsp →

new rsp=rsp-rax →



5. 请分别对照下列的C代码与对应的汇编代码(编译开关: -S -O2 -fno-stack-protector), 解释下 C99 标准中引入的 variable-length array (简称 VLA, 即允许使用变量定义数组各维) 在这一块代码中是如何实现的? 可以画出函数运行时的栈 layout, 并辅以说明来解释。

read_and_process:

```
pushq    %rbp
movslq   %edi, %rax
leaq     22(,%rax,8), %rax
movq     %rsp, %rbp
pushq    %r14
pushq    %r13
pushq    %r12
pushq    %rbx
andq     $-16, %rax
leal     -1(%rdi), %r13d
subq     %rax, %rsp
testl    %edi, %edi
movq     %rsp, %r14
jle      .L4
leal     -1(%rdi), %eax
movq     %rsp, %rbx
leaq     8(%rsp,%rax,8), %r12
movq     %rax, %r13
```

edi是输入参数n
8n是数组vals的字节长度
多加22个可能用于其他用途

在栈上开辟一块8n的空间

.L3:

```
xorl     %eax, %eax
addq     $8, %rbx
call     read_val
cltq     # sign-extend eax to all of rax
movq     %rax, -8(%rbx)
cmpq     %r12, %rbx
jne      .L3
```

.L4:

```
movslq   %r13d, %r13
movq     (%r14,%r13,8), %rax
leaq     -32(%rbp), %rsp
popq     %rbx
popq     %r12
popq     %r13
popq     %r14
popq     %rbp
ret
```

为什么会有-32的
偏移量, 因为后面
有4个pop

```
long read_and_process(int n)
{
    long vals[n];

    for (int i = 0; i < n; i++)
        vals[i] = read_val();
    return vals[n-1];
}
```

old rsp

rsp=rsp-rax

高地址

在栈上开辟
一块8n的空间, 用于保
存vals数组

低地址

5. 请分别对照下列的C代码与对应的汇编代码(编译开关: -S -O2 -fno-stack-protector), 解释下 C99 标准中引入的 variable-length array (简称 VLA, 即允许使用变量定义数组各维) 在这一块代码中是如何实现的? 可以画出函数运行时的栈 layout, 并辅以说明来解释。

read_and_process:

```
pushq    %rbp
movslq   %edi, %rax
leaq     22(,%rax,8), %rax
movq     %rsp, %rbp
```

edi是输入参数n
8n是数组vals的字节长度
多加22个可能用于其他用途

```
pushq    %r14
pushq    %r13
pushq    %r12
pushq    %rbx
andq     $-16, %rax
leal     -1(%rdi), %r13d
subq     %rax, %rsp
```

在栈上开辟一块8n的空间

```
testl    %edi, %edi
movq     %rsp, %r14
jle      .L4
leal     -1(%rdi), %eax
movq     %rsp, %rbx
leaq     8(%rsp,%rax,8), %r12
movq     %rax, %r13
```

代码放这里, 则可以删除-32的偏移量, 和左侧的操作顺序相反

```
leaq     (%rbp), %rsp
```

.L3:

```
xorl     %eax, %eax
addq     $8, %rbx
call     read_val
cltq     # sign-extend eax to all of rax
movq     %rax, -8(%rbx)
cmpq     %r12, %rbx
jne      .L3
```

.L4:

```
movslq   %r13d, %r13
movq     (%r14,%r13,8), %rax
```

```
popq     %rbx
popq     %r12
popq     %r13
popq     %r14
popq     %rbp
ret
```

```
long read_and_process(int n)
{
    long vals[n];

    for (int i = 0; i < n; i++)
        vals[i] = read_val();
    return vals[n-1];
}
```

old rsp →

rsp=rsp-rax →

高地址

在栈上开辟一块8n的空间, 用于保存vals数组

低地址

5. 请分别对照下列的C代码与对应的汇编代码(编译开关: -S -O2 -fno-stack-protector), 解释下 C99 标准中引入的 variable-length array (简称 VLA, 即允许使用变量定义数组各维) 在这一块代码中是如何实现的? 可以画出函数运行时的栈 layout, 并辅以说明来解释。

read_and_process:

```
pushq    %rbp
movslq   %edi, %rax
leaq     22(,%rax,8), %rax
movq     %rsp, %rbp
pushq    %r14
pushq    %r13
pushq    %r12
pushq    %rbx
andq     $-16, %rax
leal     -1(%rdi), %r13d
subq     %rax, %rsp
testl    %edi, %edi
movq     %rsp, %r14
jle      .L4
leal     -1(%rdi), %eax
movq     %rsp, %rbx
leaq     8(%rsp,%rax,8), %r12
movq     %rax, %r13
```

edi是输入参数n
8n是数组vals的字节长度
多加22个可能用于其他用途

在栈上开辟一块8n的空间

代码放这里, 则可以删除-32的偏移量, 和左侧的操作顺序相反

```
leaq     (%rbp), %rsp
```

.L3:

.L4:

```
movslq   %r13d, %r13
movq     (%r14,%r13,8), %rax
popq     %rbx
popq     %r12
popq     %r13
popq     %r14
popq     %rbp
ret
```

程序在自己的栈中开辟8n长度的内存空间, 用于保存vals数组。当退出该程序时, 需要恢复栈顶指针rsp, 相当于释放vals数组对应的内存空间

问题: 开辟8n长度的内存空间时, 一定需要修改rsp吗?

```
long read_and_process(int n)
{
    long vals[n];

    for (int i = 0; i < n; i++)
        vals[i] = read_val();
    return vals[n-1];
}
```

old rsp →

rsp=rsp-rax →

高地址

在栈上开辟一块8n的空间, 用于保存vals数组

低地址

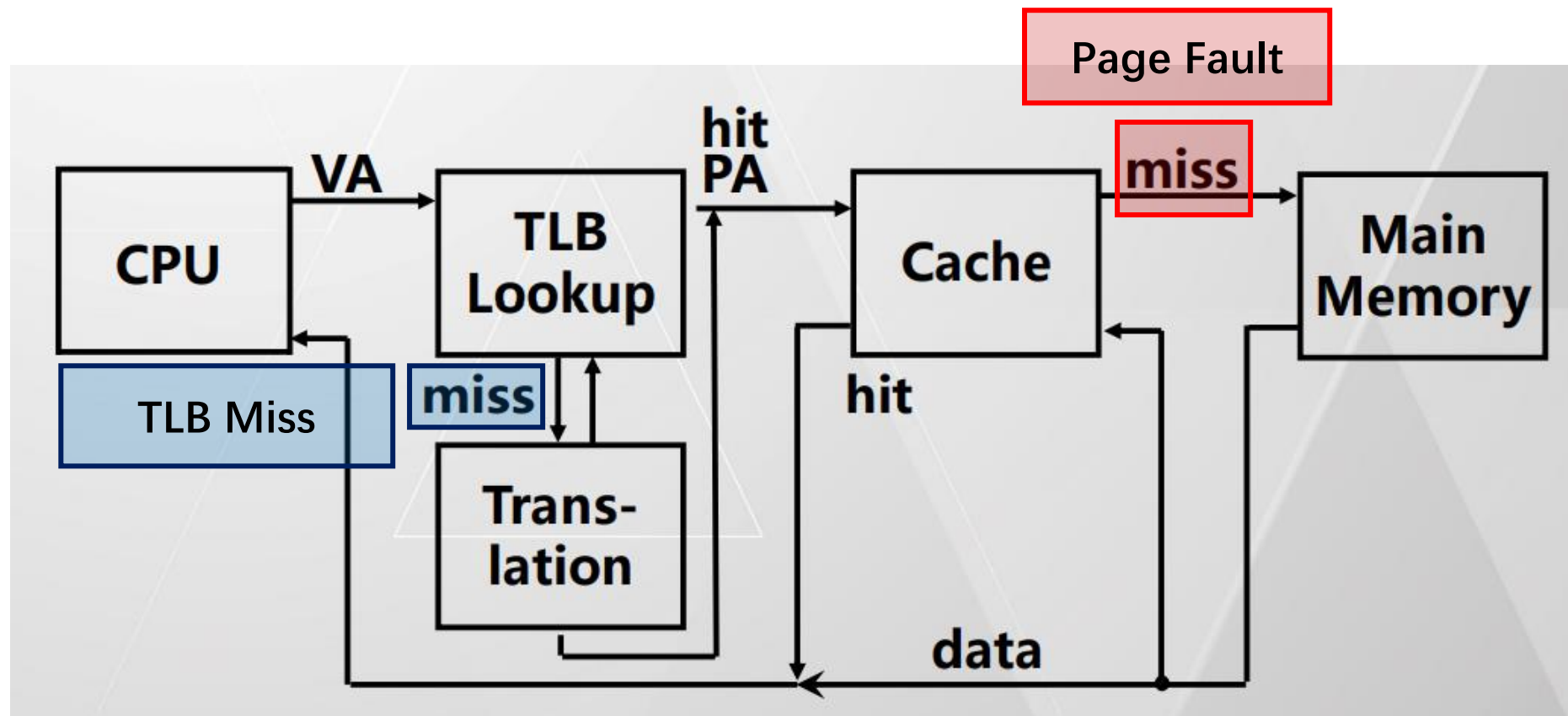
汇编作业讲解3

杨乐 2021.9

1 TLB工作原理

- MMU使用**页表**将**虚拟地址**转换为**物理地址**。
- TLB则通过**缓存**的方式来加速这一转换的过程。
- Page Fault异常：该页不在内存当中
- 如何解决：将该页存放到内存中
- TLB Miss异常：某虚拟地址在TLB中不存在匹配项
- 如何解决：通过页表查询，随后将该项填入TLB

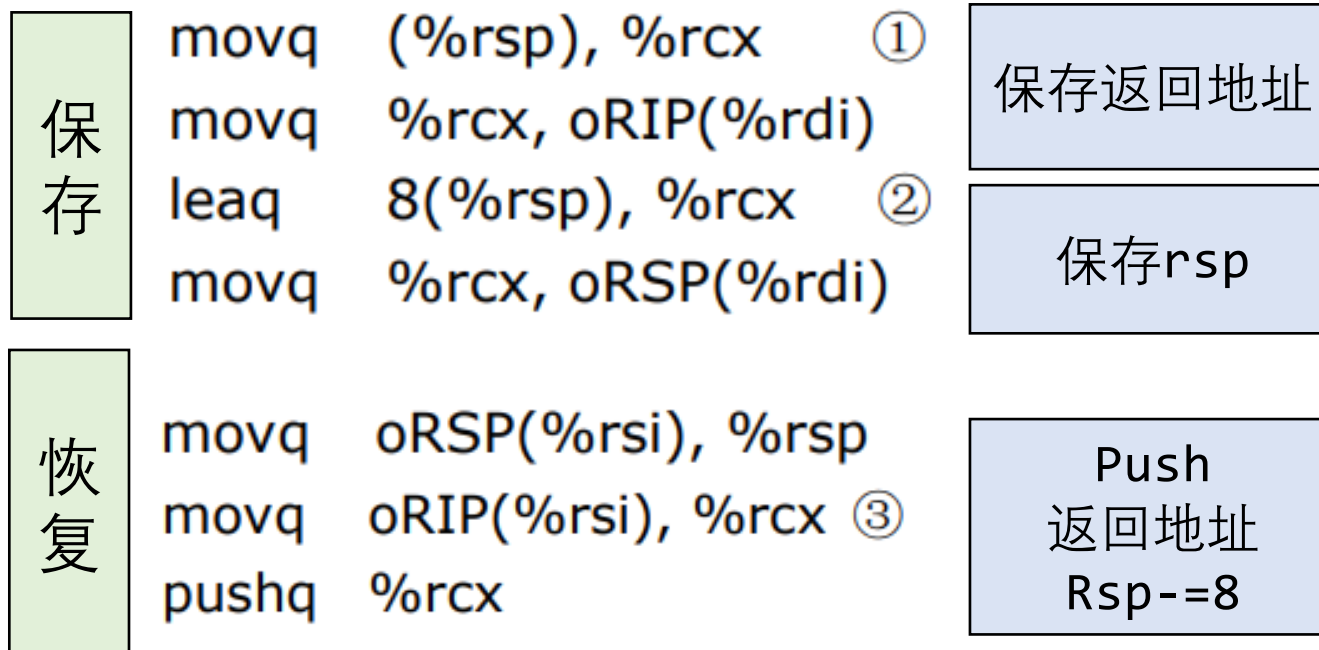
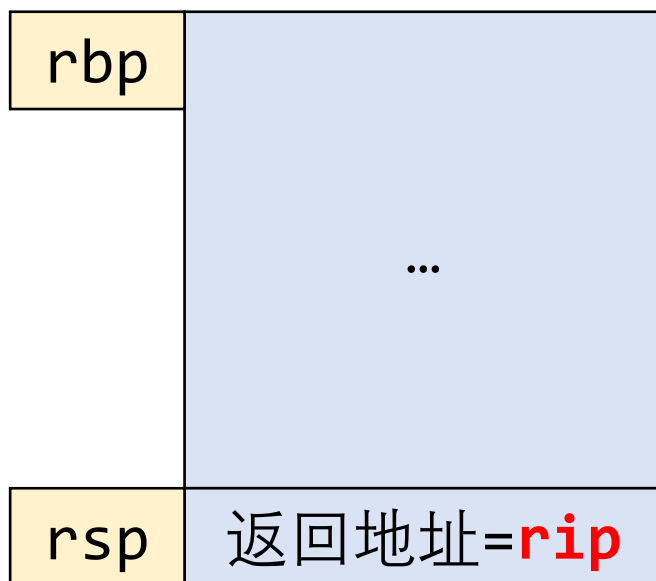
1 TLB工作原理 cont



2 SwapContext上下文切换

使用空间: $14 \times 8 = 112\text{B}$
r10/r11/rax调用者保存

- (1) 保存当前上下文: 普通寄存器, rbp, rsp, rip
- (2) 恢复至新的上下文: 普通寄存器, rbp, rsp, rip



3、有如下的 C 代码及其对应的 X86-64 汇编代码，请问

(1) 局部变量 `result` 如何存储? 存在 `rdx` 中

(2) `i` 如何存储? 存在 `edi` 中

(3) `EXPR1`、`EXPR2`、`EXPR3`、`EXPR4`、`EXPR5` 分别是?
请用常数或者 C 程序中的变量表示。

```
long int puzzle(int a, int b)
{
    int i;    i=a; i>0; i-=b
    long int result = EXPR1;    result=rdx=b
    for (i = EXPR2; i > EXPR3; i -= EXPR4)
        result *= EXPR5;    result*=i
    return result;    result=rdx
}
```

```
puzzle:
    movslq %esi,%rdx    rdx = b
    jmp .L60
.L61:
    movslq %edi,%rax    rax = a
    subl %esi, %edi    a -= b
    imulq %rax, %rdx    rdx *= rax(a)
.L60:
    testl %edi, %edi
    jg .L61            if a>0 goto L61
    movq %rdx, %rax
    ret                return rdx
```


4 MIPS指令

假设存在如下的完成计数任务的 mips32 汇编代码（左侧框图内），被两个**同时运行**的任务调用，且这两个任务代码中的地址 65540(\$4)指向同一个物理内存地址，为确保代码能够正确的实现程序语义，**需要替换原始代码中的两条指令，如何替换？**此外，汇编器将现有的左侧代码转换为了右侧框图内的**等价指令**，请填空。

(1) 替换指令

将lw替换为ll
将sw替换为sc

在多线程程序中，为了实现对共享变量的互斥访问，一般需要一个TestAndSet的**原子操作**。在MIPS中，是通过特殊的Load/Store指令：LL（Load Linked，链接加载）以及SC（Store Conditional，条件存储）这一指令对完成的。

(2) 转换指令

lw \$2, 65540(\$4)

$R[2] = R[R[4] + 65540]$

lui \$1, 1

addu \$1, \$1, \$4

lw \$2, 4(\$1)

$R[1] = 65536 * \underline{1}$

$R[1] = \underline{R[1]} + \underline{R[4]}$

$R[2] = R[R[1] + \underline{4}]$

5 union

```
union {  
    fp16 f;  
    short s;  
}
```

浮点数：求最大规格化数（符号1位，exp5位，frac10位）
整数：求对应整数

fp16 f

0

1 1 1 1 0

1 1 1 1 1 1 1 1 1 1

符号位
正

exp位：最大
且为规格化

Frac位
全部为1

$$f = 2^{(30 - 15)} * (2^1 - 2^{-10}) = 65504$$

short s

0

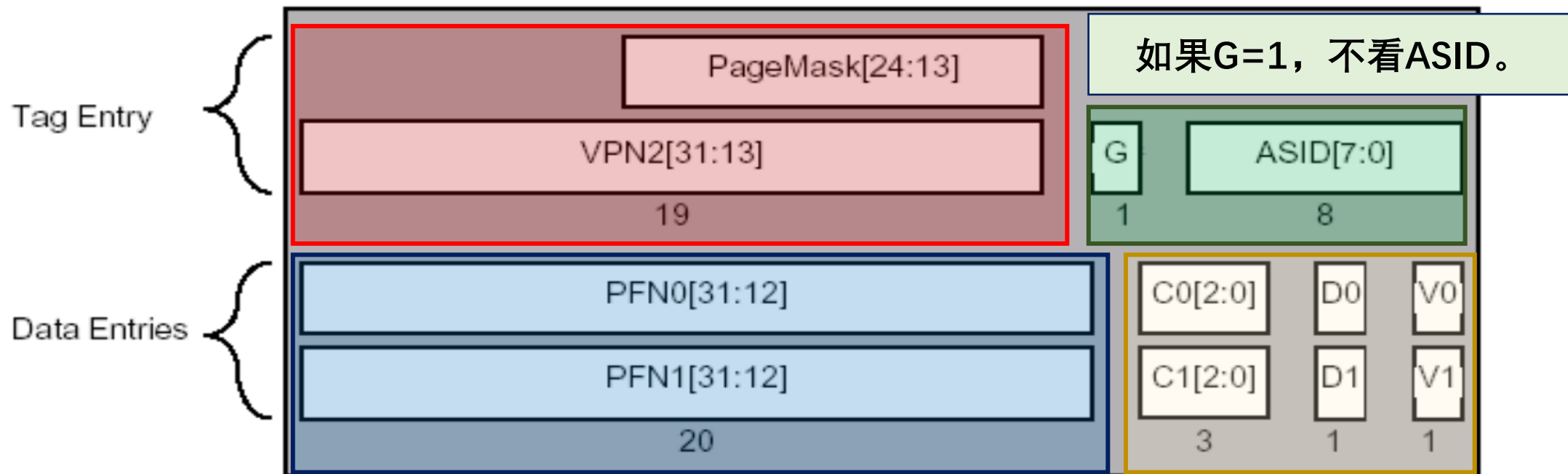
1 1 1 1 0

1 1 1 1 1 1 1 1 1 1

s = 31743

6 虚拟地址到物理地址

看page size。如果page size是4K（12位），则看31-13；
如果page size是16M（24位），则看31-25。



看tag的最低位来选择哪一路。如果page size是4K，则看第12位；
如果page size是16M，则看第24位。

标志位

6 虚拟地址到物理地址 cont

0x 8E2AE 320

0x12

写

Page size = 4K, offset=12位

0x8E2AE = 1000 1110 0010 1010 1110

VPN = 1000 1110 0010 1010 111 = 0x47157; Way = 0

	VPN2	G	ASID	PFN0	PFN1	D0/D1	V0/V1
1	0x47157 (二进制为 100 0111 0001 0101 0111)	0	0x12	0x12345	0x12340	0/0	1/1
2	0x47157 (二进制为 100 0111 0001 0101 0111)	0	0x13	0x22346	0x22340	1/1	1/1

看ASID选JTLB[1], Way=0 : PFN0 = 0x12345, D0 = 0, V0 = 1

物理地址 = 0x12345 320; 但需要看标志位

V=0直接无效; D=0写无效; 其余有效