# 机器级编程 1
# Machine-Level Programming I

课 程 名 ：计算机系统

第 3 讲 （2025 年 4 月 25 日）

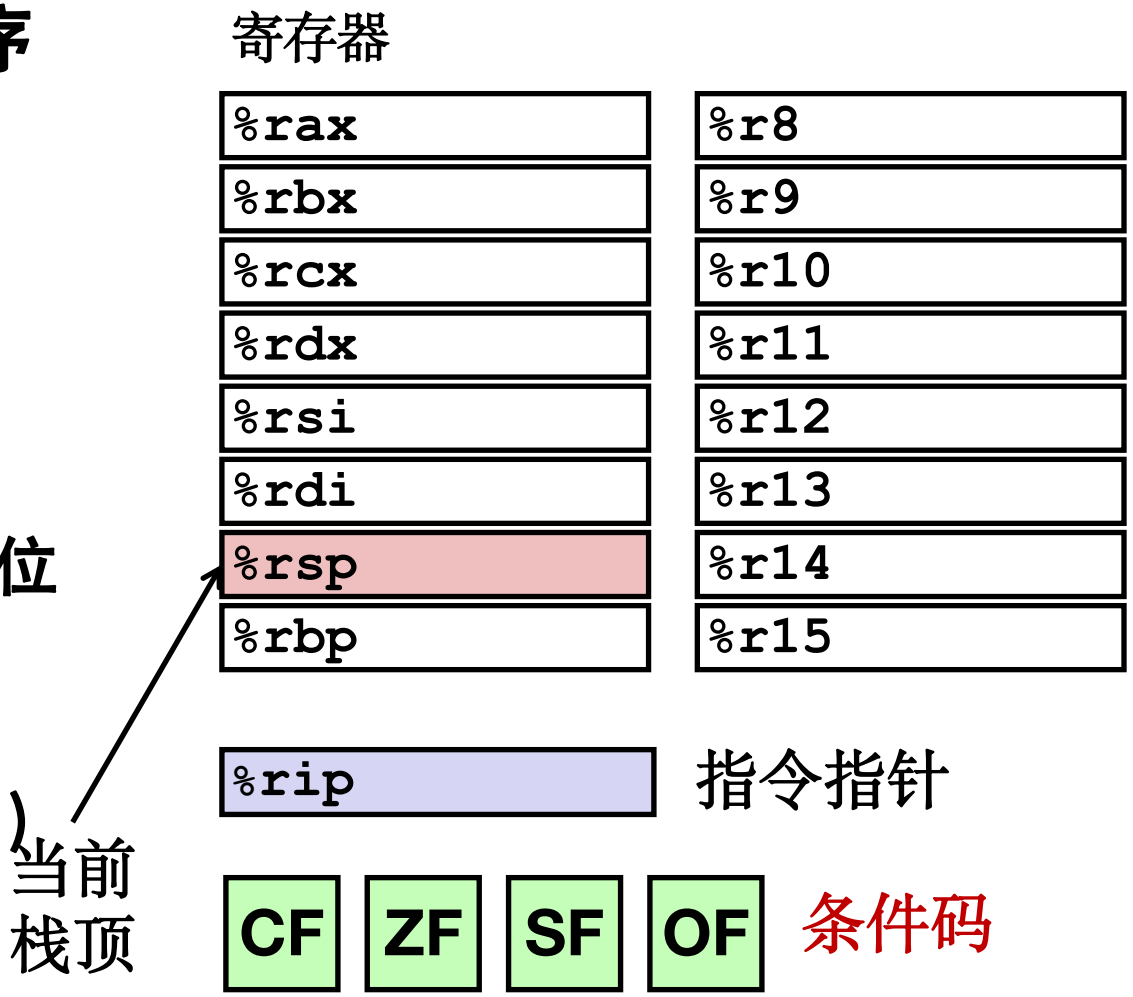
主 讲 人 ：杜海文

# 本课内容

- **控制：条件码**
- **条件转移**
- **循环**

# x86-64 处理器状态（部分）

- 关于当前运行程序的实时信息
  - 临时数据
    ( %rax, … )
  - 运行时栈的位置
    ( %rsp )
  - 当前的代码控制点位
    ( %rip, … )
  - 近期的测试状态
    ( CF, ZF, SF, OF )



# 条件码（隐式设置）

- 1 位的寄存器（触发器）
  - CF 进位标志（针对 unsigned）　　SF 符号标志（针对 signed）
  - ZF 零标志　　OF 溢出标志（针对 signed）
- 由算术操作指令隐式设置（类似于副作用）
  - 举例：addq Src,Dst ↔ t = a+b

CF 置 1 当最高位（MSB）产生进位时（无符号数溢出）

ZF 置 1 当 t == 0 时

SF 置 1 当 t < 0 (对于带符号数)

OF 置 1 当发生补码（带符号数）溢出

```
(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
```

- leaq 指令不影响上述标志位

# 条件码（显式设置：compare 指令族）

- 由 compare 指令族进行显式设置
  - `cmpq Src2, Src1`
    - `cmpq b,a` 计算 `a-b` 但不改变目的操作数的值
  - CF 置 1 当最高位产生进位（用于无符号数比较）
  - ZF 置 1 当 `a == b`
  - SF 置 1 当 `(a-b) < 0` (针对带符号数)
  - OF 置 1 当产生补码（带符号数）溢出

```
(a>0 && b<0 && (a-b)<0) ||
      (a<0 && b>0 && (a-b)>0)
```

# 条件码（显式设置：test 指令族）

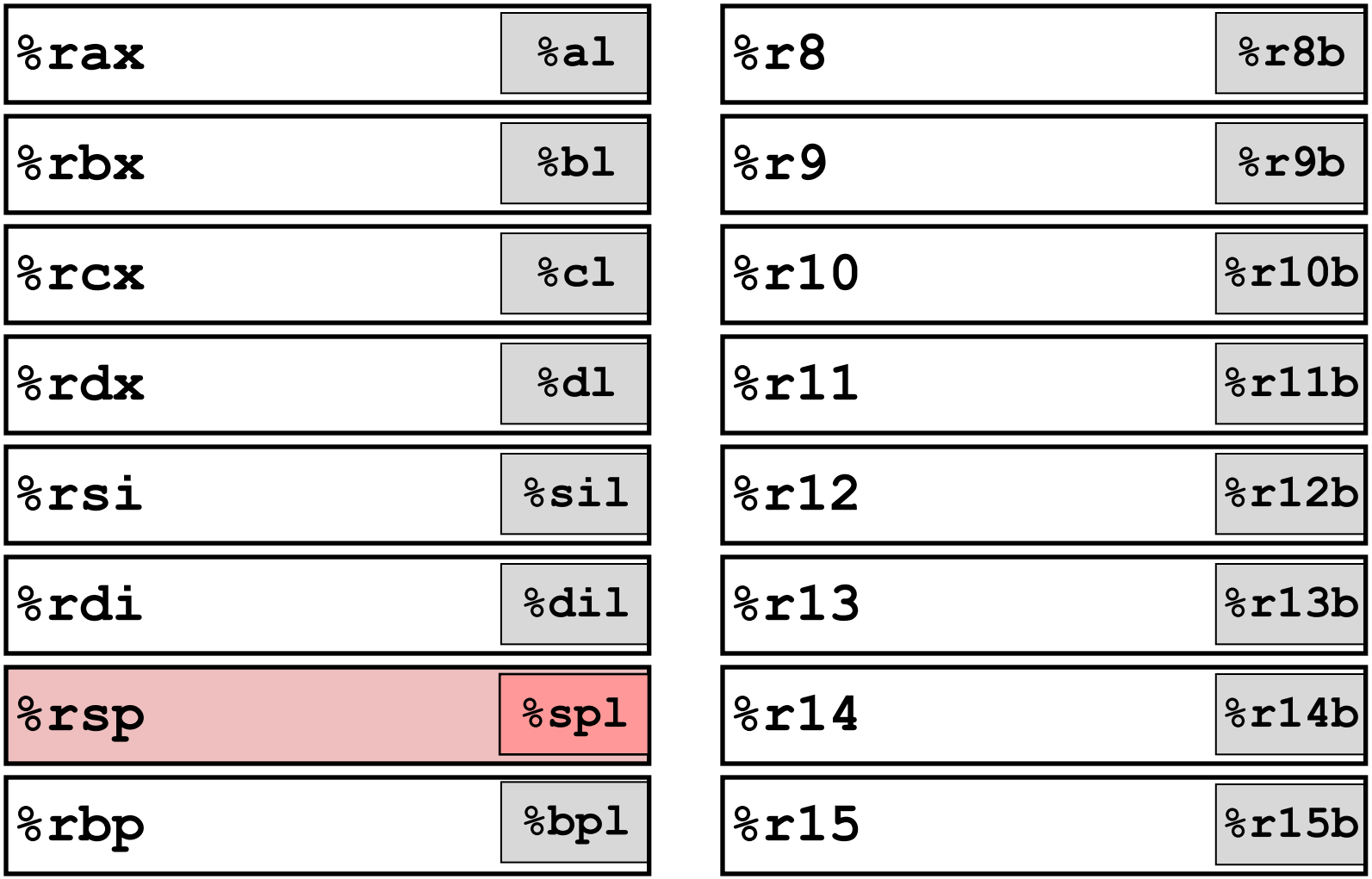
- 由 test 指令族进行显式设置
  - `testq` Src2,Src1
    - `testq b,a` 计算 `a&b` 但不改变目的操作数的值
  - 根据 *Src1* `&` *Src2* 的值设置条件码
  - 经常令其中一个操作数为掩码mask
  - ZF 置 1 当 `a & b == 0`
  - SF 置 1 当 `a & b < 0`

# 读取条件码

- **setX 指令族**
  - **根据条件码（的组合），将目的操作数的最低字节LSB置为 0 或 1**
  - **不会改变 64 位**

| setX | 条件码 | 说明 |
|---|---|---|
| sete | ZF | Equal / Zero |
| setne | ~ZF | Not Equal / Not Zero |
| sets | SF | Negative |
| setns | ~SF | Nonnegative |
| setb | CF | Below (unsigned) |
| seta | ~CF&~ZF | Above (unsigned) |
| setl | (SF^OF) | Less (Signed) |
| setge | ~(SF^OF) | Greater or Equal (Signed) |
| setle | (SF^OF)\|ZF | Less or Equal (Signed) |
| setg | ~(SF^OF)&~ZF | Greater (Signed) |

# x86-64 整数寄存器




- 可单独访问最低字节

# 读取条件码（续）

- setX 指令族
  - 根据条件码（的组合），将单个字节（LSB）置为 0 或 1
- 某个可单独访问的字节寄存器
  - 不改变相应 64 位寄存器的其它字节
  - 常使用 `movzbl` 指令完成剩余工作
    - 生成 32 位结果的指令会将高 32 位清零

```
int gt(long x, long y)
{
    return x > y;
}
```

| 寄存器 | 取值 |
|---|---|
| %rdi | 参数 x |
| %rsi | 参数 y |
| %rax | 返回值 |

```
    cmpq    %rsi,%rdi       # Compare x:y
    setg    %al             # Set when >
    movzbl  %al,%eax        # Zero rest of %rax
    ret
```

# 本课内容

- 控制：条件码
- **条件转移**
- 循环

# 跳转

- jX 指令族
  - 根据条件码的设置情况，跳转到代码的不同位置

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

# 条件转移举例（Old Style）

见 GCC 选项索引

- 编译命令

```
openEuler> gcc –Og –S –fno–if–conversion control.c
```

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

```
absdiff:
    cmpq    %rsi, %rdi   # x:y
    jle     .L2
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L2:                     # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

| 寄存器 | 取值 |
|---|---|
| %rdi | 参数 x |
| %rsi | 参数 y |
| %rax | 返回值 |

# Goto 式的写法

- C 提供了 goto 语句
- 无条件转移至标号label所示的位置

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

```
long absdiff_j
    (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
Else:
    result = y-x;
Done:
    return result;
}
```

# 一般条件表达式的翻译（使用条件转移）

C 代码

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

Goto 版本

```
    ntest = !Test;
    if (ntest) goto Else;
    val = Then_Expr;
    goto Done;
Else:
    val = Else_Expr;
Done:
    . . .
```

- 为 then 和 else 两个表达式各自书写代码
- 根据具体情况执行相应的部分

# 使用条件传送指令

- 条件传送指令族
  - 该指令族实现如下功能：

    `if (Test) Dst = Src`
  - 1995 年之后的 x86 处理器均支持
  - GCC 尽量采用此类指令
    - 但是会在确保安全的前提下
- 原因
  - 分支转移对于流水线中指令的顺畅流动极为不利
  - 条件转移指令不需要产生控制转移

C 代码

```
val = Test
    ? Then_Expr
    : Else_Expr;
```

Cmov 版本（原课件勘误）

```
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```

# 条件传送指令举例

| 寄存器 | 取值 |
|---|---|
| %rdi | 参数 x |
| %rsi | 参数 y |
| %rax | 返回值 |

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

- 编译命令

```
openEuler> gcc –Og –S \
> –fif–conversion control.c
```

```
absdiff:
        movq    %rdi, %rdx      # x
        subq    %rsi, %rdx      # x-y → %rdx
        movq    %rsi, %rax      # y
        subq    %rdi, %rax      # result = y-x
        cmpq    %rsi, %rdi      # x:y
        cmovg   %rdx, %rax      # if > result = x-y
        ret
```

# 不适合使用条件传送的情况

## 计算量庞大

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

- 两个值都要算
- 只有当计算量较小时才有意义

## 计算有风险

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

- 两个值都要算
- 可能因此产生不良后果

## 计算有副作用

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

- 两个值都要算
- 必须在没有副作用的前提下使用

# 本课内容

- 控制：条件码
- 条件转移
- **循环**

# Do-While 循环举例

## Do–While 版本

```
long pcount_do
   (unsigned long x)
{
   long result = 0;
   do {
      result += x & 0x1;
      x >>= 1;
   } while (x);
   return result;
}
```

## Goto 版本

```
long pcount_goto
   (unsigned long x)
{
   long result = 0;
loop:
   result += x & 0x1;
   x >>= 1;
   if (x) goto loop;
   return result;
}
```

- 计算参数 x 中 1 的位数 (popcount 问题)
- 使用条件转移决定是否继续循环

# 编译 Do-While 代码

## Goto 版本

```
long pcount_goto(unsigned long x)
{
    long result = 0;
loop:
    result += x & 0x1;
    x >>= 1;
    if (x) goto loop;
    return result;
}
```

| 寄存器 | 取值 |
|---|---|
| %rdi | 参数 x |
| %rax | result |

```
    movl    $0, %eax        #  result = 0
.L2:                        # loop:
    movq    %rdi, %rdx
    andl    $1, %edx        #  t = x & 0x1
    addq    %rdx, %rax      #  result += t
    shrq    %rdi            #  x >>= 1
    jne     .L2             #  if (x) goto loop
    rep; ret
```

# 一般的 Do-While 控制结构翻译

Do–While 版本

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

**Goto 版本**

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

- ***Body* 表示循环体**

```
{
    语句statement1;
    语句statement2;
    ...
    语句statementn;
}
```

# 常见的 While 控制结构翻译 #1

- jump-to-middle 型
- 使用 −Og 编译

while 版本

```
while (Test)
    Body
```

Goto 版本

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

# While 循环示例 #1

While 版本

```
long pcount_while
     (unsigned long x)
{
     long result = 0;

     while (x) {
          result += x & 0x1;
          x >>= 1;
     }
     return result;
}
```

Goto (jump to middle) 版

```
long pcount_goto_jtm
     (unsigned long x)
{
     long result = 0;
     goto test;
loop:
     result += x & 0x1;
     x >>= 1;
test:
     if (x) goto loop;
     return result;
}
```

- 与 Do-While 版本进行比较
- 第一条 goto 语句转至 test 后开始循环
- 编译并对照阅读汇编代码：gcc –Og –S pcount_while.c

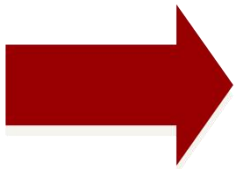
# 常见的 While 控制结构翻译 #2

While 版本

```
while (Test)
    Body
```

Do–While 版本

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



Goto 版本

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

- 向 Do-While 转化
- 使用 –O1 优化选项编译

# While 循环示例 #2

While 版本

```
long pcount_while
     (unsigned long x)
{
     long result = 0;

     while (x) {
          result += x & 0x1;
          x >>= 1;
     }
     return result;
}
```

Do-While 版本

```
long pcount_goto_dw
     (unsigned long x)
{
     long result = 0;
     if (!x) goto done;
loop:
     result += x & 0x1;
     x >>= 1;
     if (x) goto loop;
done:
     return result;
}
```

- 与 Do-While 版本进行比较
- 第一次条件判断决定循环是否准入，否则直接结束
- 编译并对照阅读汇编代码：gcc –O1 –S pcount_while.c

# For 循环结构

一般形式

```
for (Init; Test; Update)
    Body
```

```
#define LSIZE 8*sizeof(long)
long pcount_for(unsigned long x)
{
    size_t i;
    long result = 0;

    for (i = 0; i < LSIZE; i++) {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

***Init***

```
i = 0
```

***Test***

```
i < LSIZE
```

***Update***

```
i++
```

***Body***

```
{
    unsigned bit =
        (x >> i) & 0x1;
    result += bit;
}
```

# For 循环 → While 循环

**For 版本**

```
for (Init; Test; Update)
    Body
```

**While 版本**

```
Init;

while (Test) {
    Body
    Update;
}
```

# For-While 转换

***Init***

```
i = 0
```

***Test***

```
i < LSIZE
```

***Update***

```
i++
```

***Body***

```
{
    unsigned bit =
        (x >> i) & 0x1;
    result += bit;
}
```

```
long pcount_for_while
    (unsigned long x)
{
    size_t i;
    long result = 0;
    i = 0;

    while (i < LSIZE) {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
        i++;
    }
    return result;
}
```

# For - Do-While 转换

For 版本

```
long pcount_for
    (unsigned long x)
{
    size_t i;
    long result = 0;

    for (i=0; i<LSIZE; i++) {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

- 最初的测试可以被编译优化掉
- 对比 –Og 和 –O1 生成的汇编

Goto 版本

```
long pcount_for_goto_dw
    (unsigned long x)
{
    size_t i;
    long result = 0;
    i = 0;                          Init
    if (!(i < LSIZE))               !Test
        goto done;
loop:
    {
        unsigned bit =
Body        (x >> i) & 0x1;
        result += bit;
    }
    i++;                            Update
    if (i < LSIZE)                  Test
        goto loop;
done:
    return result;
}
```