80X86汇编语言与C语言-2

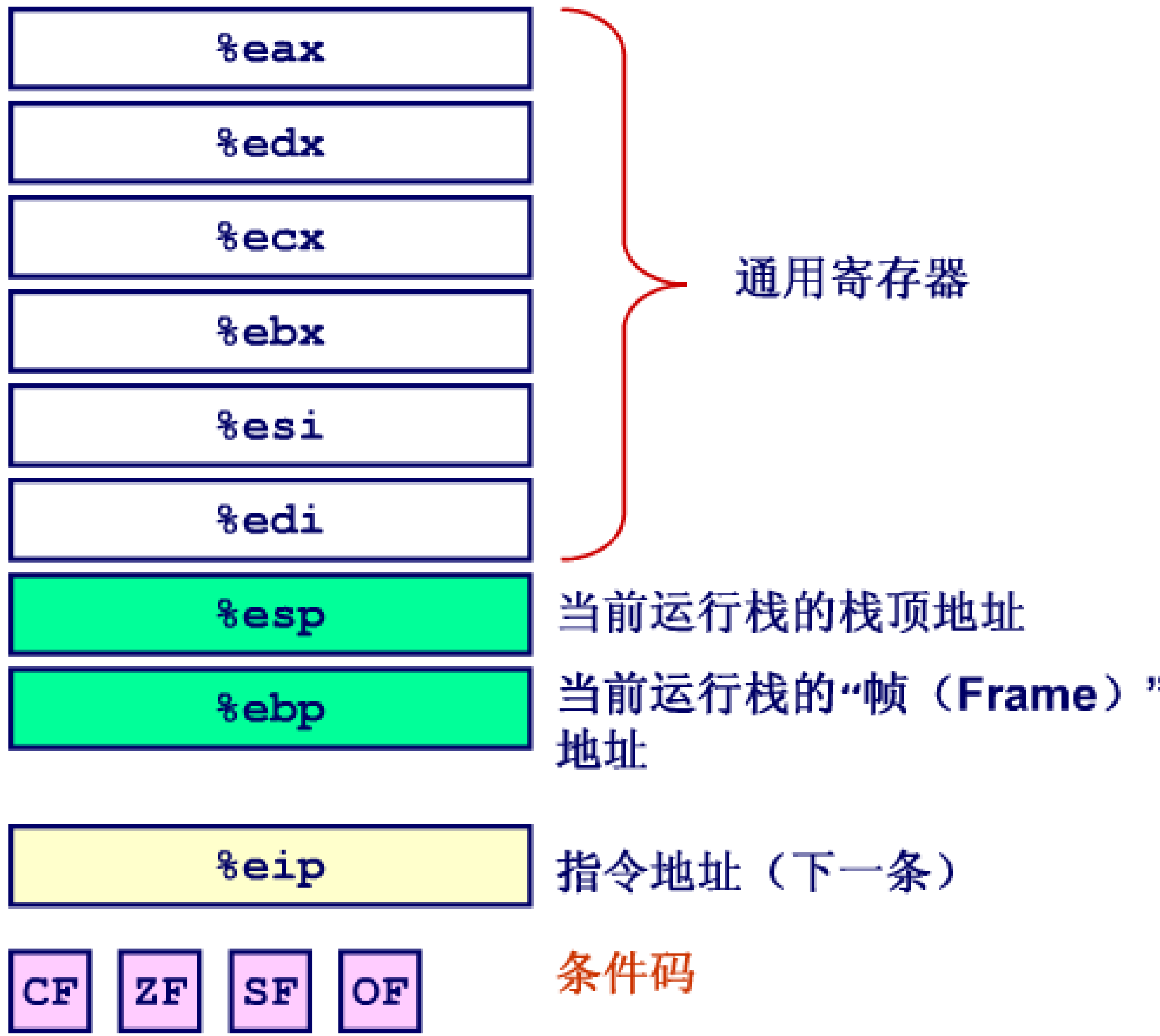
控制流

- 条件码
  - 设置
  - 读取
- 程序控制流
  - If-then-else
  - 循环结构
  - switch语句
- x86-64 模式
  - 条件传送指令
  - 循环结构的不同实现

汇编程序员眼中的系统结构（部分）

当前执行程序的信息

- 数据
- 指令地址
- 运行栈地址
- 条件码



条件码

CF 进位标志（Carry）　　SF 符号位（Sign）

ZF Zero Flag　　OF 溢出标志（Overflow ）

这些条件码由算术指令隐含设置

`addl` *Src,Dest*　　`addq` *Src,Dest*

类似的C语言表达式：`t = a + b`　（`a` = Src, `b` = Dest)

- CF 进位标志
  - 可用于检测无符号整数运算的溢出
- 如果 `t == 0`，那么`ZF=1`;否则`ZF=0`
- 如果`t < 0`，那么`SF=1`；否则`SF=0`
- 如果补码运算溢出，那么`OF=1`（即带符号整数运算）

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

比较（Compare）指令

`cmpl` *Src2,Src1*　　　`cmpq` *Src2,Src1*

- `cmpl b,a` 类似于计算a-b（但是不改变目的操作数）
- 如果向最高位有借位，那么CF=1;否则CF=0
  - 可用于无符号数的比较
- 如果a == b，那么ZF=1；否则ZF=0
- 如果(a - b)<0，那么SF=1；否则SF=0
  - 即运算后若结果最高位为1，那么SF=1；否则为0
- 如果补码运算溢出，那么OF=1
  - (a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)

测试（Test）指令

`testl` *Src2,Src1*

`testq` *Src2,Src1*

- 计算*Src1* & *Src2*并设置相应的条件码，但是不改变目的操作数

- 如果a&b == 0，那么ZF=1；否则为0
- 如果a&b < 0 ，那么SF=1；否则为0
  - 即运算后结果最高位为1，那么SF=1；否则为0

读取条件码

SetX 指令

- 读取当前的条件码（或者某些条件码的组合），并存入目的字节寄存器

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

SetX 指令

- 读取当前的条件码（或者某些条件码的组合），并存入目的“字节”寄存器
  - 余下的三个字节不会被修改
  - 通常使用“ movzbl”指令对目的寄存器进行“0”扩展


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

Body

```
movl 12(%ebp),%eax    # eax = y
cmpl %eax,8(%ebp)     # Compare x : y
setg %al              # al = x > y
movzbl %al,%eax       # Zero rest of %eax
(movsbl)
```

x86-64下读取条件码

SetX 指令

- 读取当前的条件码（或者某些条件码的组合），并存入目的“字节”寄存器
  - 余下的七个字节不会被修改

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

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

- x86-64 下的函数参数
  - x in %rdi
  - y in %rsi

“64-bit operands generate a 64-bit result in the destination general-purpose register.
32-bit operands generate a 32-bit result, zero-extended to a 64-bit result in the destination general-purpose register.“

摘自“Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture“

这两个过程的汇编代码主体是一样的！

```
xorl %eax, %eax       # eax = 0
cmpq %rsi, %rdi       # Compare x : y
setg %al              # al = x > y
```

Δ微体系结构背景*

"　32-bit operands generate a 32-bit result, zero-extended to a 64-bit result in the destination general-purpose register."

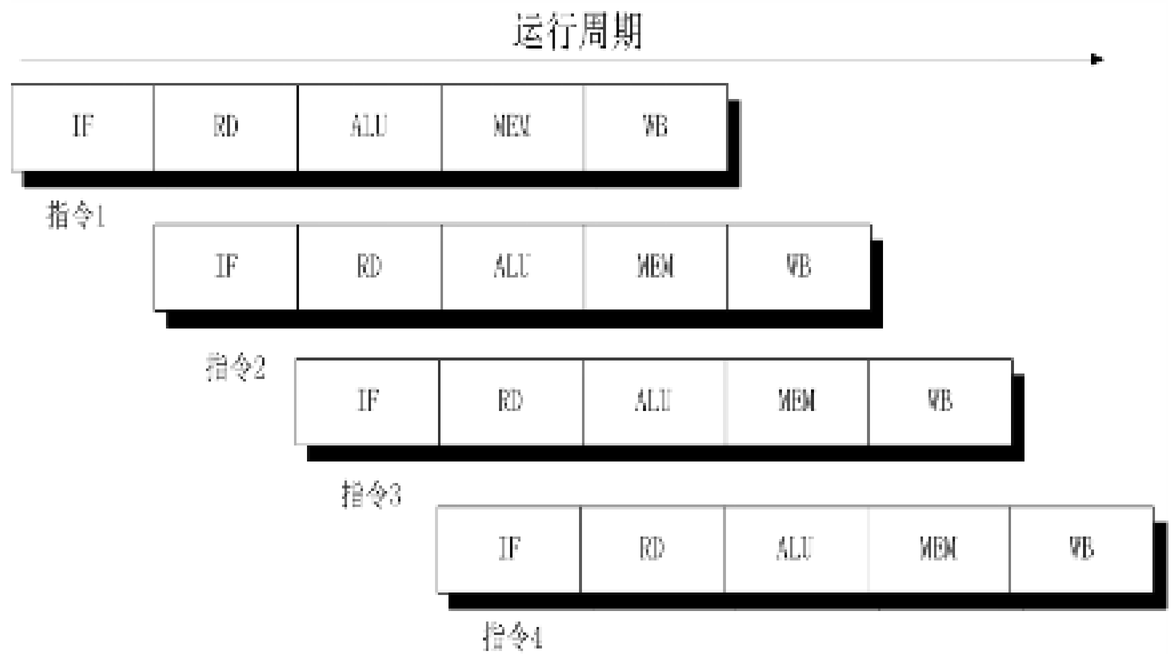
为什么?

- 部分原因来自于微体系结构（即处理器）内部实现的效率方面的考虑，目的是为了消除"部分数据依赖"

处理器流水线的概念

五级流水线（仅为示例，非x86处理器流水线）

- Instruction Fetch (IF)
- Read Registers (RD)
- Arithmetic Operation (ALU)
- Memory Access (MEM)
- Write Back (WB)



当前面指令产生的结果作为后续指令的操作数时，会引起"数据相关"(Data dependency)。

现代的通用处理器 支持深度流水线以及多发射结构，如 Pentium 4： >= 20 stages, up to 126 instructions on-fly

Superscalar execution

Clock cycle →	1	2	3	4	5	6	7	8	9	10	11
Instr. i	FI	DI	CO	FO	EI	WO					
Instr. i+1	FI	DI	CO	FO	EI	WO					
Instr. i+2		FI	DI	CO	FO	EI	WO				
Instr. i+3		FI	DI	CO	FO	EI	WO				
Instr. i+4			FI	DI	CO	FO	EI	WO			
Instr. i+5			FI	DI	CO	FO	EI	WO			

数据相关会导致指令执行效率降低（因为必须等待前面的结果出来），且流水线越深，影响越大。

•例子(**Partial Register Stall** is a problem that occurs when you write to part of a 32 bit register and later read from the whole register or a bigger part of it.)

```
addw %bx, %ax
movl %eax, %ecx
```

跳转指令

jX 指令

- 依赖当前的条件码选择下一条执行语句（是否顺序执行）

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)

条件跳转指令实例

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

```
absdiff:
   pushl   %ebp                  Set
   movl    %esp, %ebp            Up
   movl    8(%ebp), %edx
   movl    12(%ebp), %eax
   cmpl    %eax, %edx
   jle     .L7                   Body1
   subl    %eax, %edx
   movl    %edx, %eax
.L8:
   leave                         Finish
   ret
.L7:
   subl    %edx, %eax            Body2
   jmp     .L8
```

```c
int goto_ad(int x, int y)
{
  int result;
  if (x<=y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

- 将原始的C代码变形为“goto”模式，使之接近编译出来的机器语言风格

Body1

```
   # x in %edx, y in %eax
   cmpl    %eax, %edx     # Compare x:y
   jle     .L7            # <= Goto Else
   subl    %eax, %edx     # x-= y
   movl    %edx, %eax     # result = x
.L8:  # Exit:
```

Body2

```
.L7:  # Else:
   subl    %edx, %eax     # result = y-x
   jmp     .L8            # Goto Exit
```

C语言:条件表达式

```
val = Test ? Then-Expr : Else-Expr;
```

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

Goto Version

```
  nt = !Test;
  if (nt) goto Else;
  val = Then-Expr;
Done:
  . . .
Else:
  val = Else-Expr;
  goto Done;
```

条件表达式的执行顺序：

- 先求解表达式Test，若为非0（真）则求解表达式*Then-Expr*，此时表达式2的值就作为整个表达式的值。
- 若Test的值为0（假），则求解*Else-Expr*，其值就是整个条件表达式的值。

x86-64下…

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

```
absdiff: # x in %edi, y in %esi
  movl   %edi, %eax  # v  = x
  movl   %esi, %edx  # ve = y
  subl   %esi, %eax  # v  -= y
  subl   %edi, %edx  # ve -= x
  cmpl   %esi, %edi  # x:y
  cmovle %edx, %eax  # v=ve if <=
  ret
```

较新的32位Gcc也可以编译出类似代码
(-march=i686)

- 条件传送指令
  - cmov*C* src, dest
  - 如果条件*C*成立，将数据从src 传送至dest
  - 从执行角度来看，比一般的条件跳转指令的效率高
    - » 因为其控制流可预测（即条件*C是已知的*）

```
pushl  %ebp
movl   %esp,  %ebp
pushl  %ebx

movl 8(%ebp),  %ecx
movl 12(%ebp),  %edx
movl %ecx,  %ebx
subl  %edx,  %ebx
movl %edx,  %eax
subl  %ecx,  %eax
cmpl %edx,  %ecx
cmovg  %ebx,  %eax

popl  %ebx
popl  %ebp
ret
```

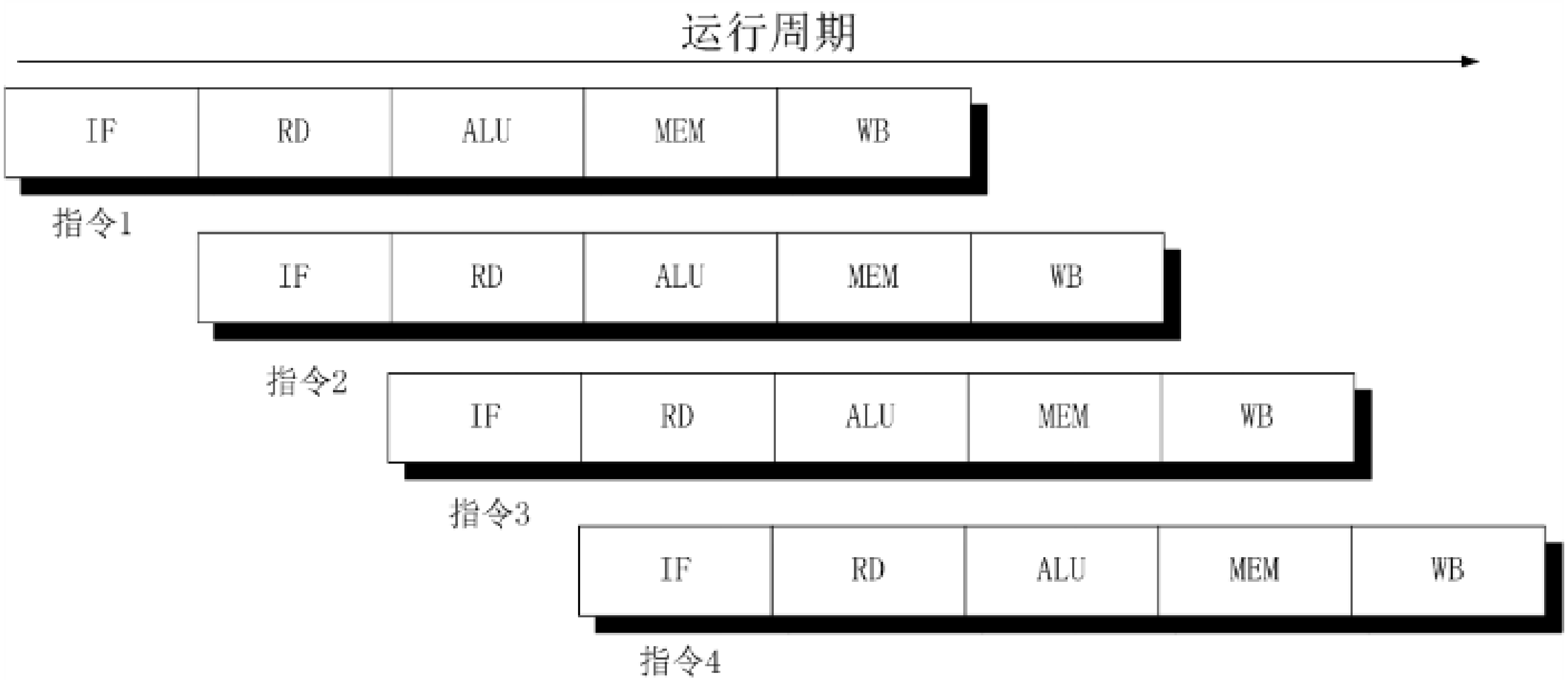
```c
int absdiff(
    int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

x86-32下条件传送指令的实例

Δ微体系结构背景

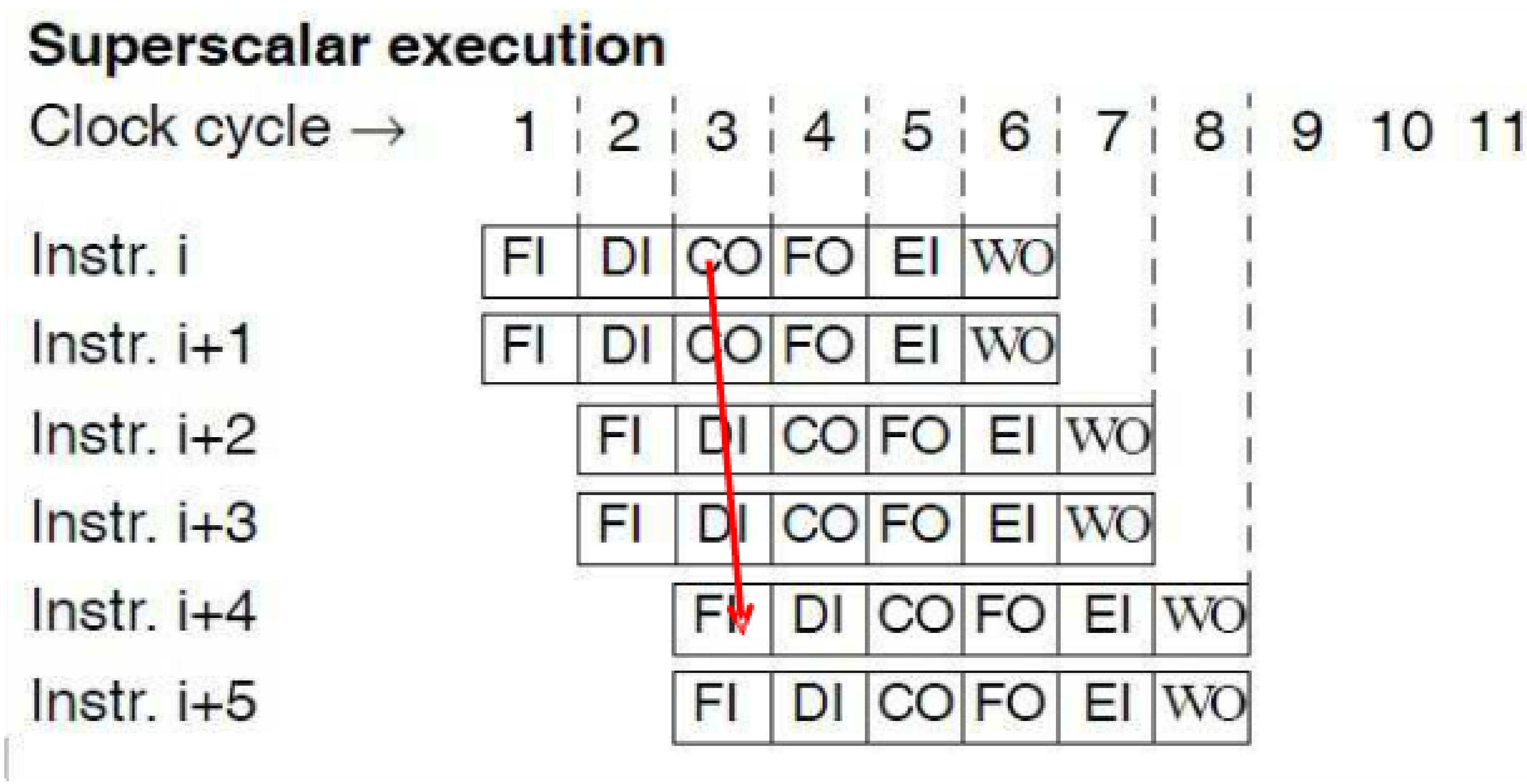
处理器流水线（五级流水示例）

- Instruction Fetch (IF)
- Read Registers (RD)
- Arithmetic Operation (ALU)
- Memory Access (MEM)
- Write Back (WB)



Δ微体系结构背景*

现代的通用处理器 支持深度流水线以及多发射结构，如
Pentium 4： >= 20 stages, up to 126 instructions on-fly



条件跳转指令往往会引起一定的性能损失，因此需要尽量消除

条件转移指令的局限性

```
val  = Then-Expr;
vale = Else-Expr;
val  = vale if !Test;
```

```
int xgty  = 0, xltey = 0;

int absdiff_se(
    int x, int y)
{
    int result;
    if (x > y) {
        xgty++;  result = x-y;
    } else {
        xltey++; result = y-x;
    }
    return result;
}
```

不宜使用的场合:

- Then-Expr 或Else-Expr 表达式有“副作用”
- Then-Expr 或 Else-Expr 表达式的计算量较大

练习题

使用条件移动指令来完成以下功能。

```
int cread(int *xp) {
    return (xp ? *xp : 0);
}
```

是否可以用如下汇编代码段来完成？

```
Invalid implementation of function cread
xp in register %edx
    movl     $0, %eax           Set 0 as return value
    testl    %edx, %edx         Test xp
    cmovne   (%edx), %eax       if !0, dereference xp to get return value
```

```c
int cread_alt(int *xp) {
    int t = 0;
    return *(xp ? xp : &t);
}
```

```
_cread_alt:
    ...
    movl   $0, -4(%ebp)      # t = 0
    movl   8(%ebp), %eax     #%eax = xp
    leal   -4(%ebp), %edx
    testl  %eax, %eax
    cmove  %edx, %eax
    movl   (%eax), %eax
```

如何实现循环（Loops）

- 所有的循环模式（while，do-while，for）都转换为“do-while”形式
  - 再转换为汇编形式
- 历史上gcc采用过多种转换模式，经历了“否定之否定”的过程

"Do-While" 循环实例

原始的C代码

```
int fact_do(int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);

  return result;
}
```

Goto Version

```
int fact_goto(int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

- 编译器先转换为goto模式

Goto Version

```c
int
fact_goto(int x)
{
  int result = 1;


loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;

  return result;
}
```

汇编

Registers

%edx	x
%eax	result

```asm
fact_goto:
  pushl %ebp          # Setup
  movl %esp,%ebp      # Setup
  movl $1,%eax        # eax = 1
  movl 8(%ebp),%edx   # edx = x

L11:
  imull %edx,%eax     # result *= x
  decl %edx           # x--
  cmpl $1,%edx        # Compare x : 1
  jg L11              # if > goto loop

  movl %ebp,%esp      # Finish
  popl %ebp           # Finish
  ret                 # Finish
```

"While" 循环-版本1

原始的C代码

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {


    result *= x;
    x = x-1;
  };

  return result;
}
```

Goto Version-1

```
int fact_while_goto(int x)
{
  int result = 1;
loop:
  if (!(x > 1))
    goto done;
  result *= x;
  x = x-1;
  goto loop;
done:
  return result;
}
```

- 这个实例与上一个等价吗？

“While” 循环-版本2（do-while模式）

原始的C代码

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result;
}
```

- 目前的GCC（4.0以后）使用的模式
- 内部循环与do-while 相同
- 在循环入口做额外的条件测试

Goto Version-2

```
int fact_while_goto2(int x)
{
  int result = 1;
  if (!(x > 1))
    goto done;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
done:
  return result;
}
```

“For” 循环

```
int result;
for (result = 1;
     p != 0;
     p = p>>1)
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

General Form

for (*Init*; *Test*; *Update*)

Body

Init

```
result = 1
```

Test

```
p != 0
```

Update

```
p = p >> 1
```

Body

```
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

"For"→ "While"→ "Do-While"

For Version

```
for (Init; Test; Update )

    Body
```

While Version

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

Do-While Version

```
  Init;
  if (!Test)
    goto done;
  do {
    Body
    Update ;
  } while (Test)
done:
```

Goto Version

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

补充：

历史上gcc采用过多种转换模式，经历了“否定之否定”的过程

do-while → Jump-to-middle → do-while

"While" 循环-版本3 （jump-to-middle）

原始的C代码

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result;
}
```

Goto Version

```
int fact_while_goto3(int x)
{
  int result = 1;
  goto middle;
loop:
  result *= x;
  x = x-1;
middle:
  if (x > 1)
    goto loop;
  return result;
}
```

- 由jump-to- middle开始第一轮循环

Jump-to-Middle 实例

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x--;
  };
  return result;
}
```

- gcc 3.4.4
  - -O2

```
# x in %edx, result in %eax
  jmp   L34           #   goto Middle
L35:                  # Loop:
  imull %edx, %eax    #   result *= x
  decl  %edx          #   x--
L34:                  # Middle:
  cmpl  $1, %edx      #   x:1
  jg    L35           #   if >, goto Loop
```

"For"→ "While" (Jump-to-Middle)

For Version

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

While Version

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

Goto Version

```
   Init;
   goto middle;
loop:
   Body
   Update ;
middle:
   if (Test)
      goto loop;
done:
```

Jump-to-Middle模式

C Code

```
while (Test)
  Body
```

Goto Version

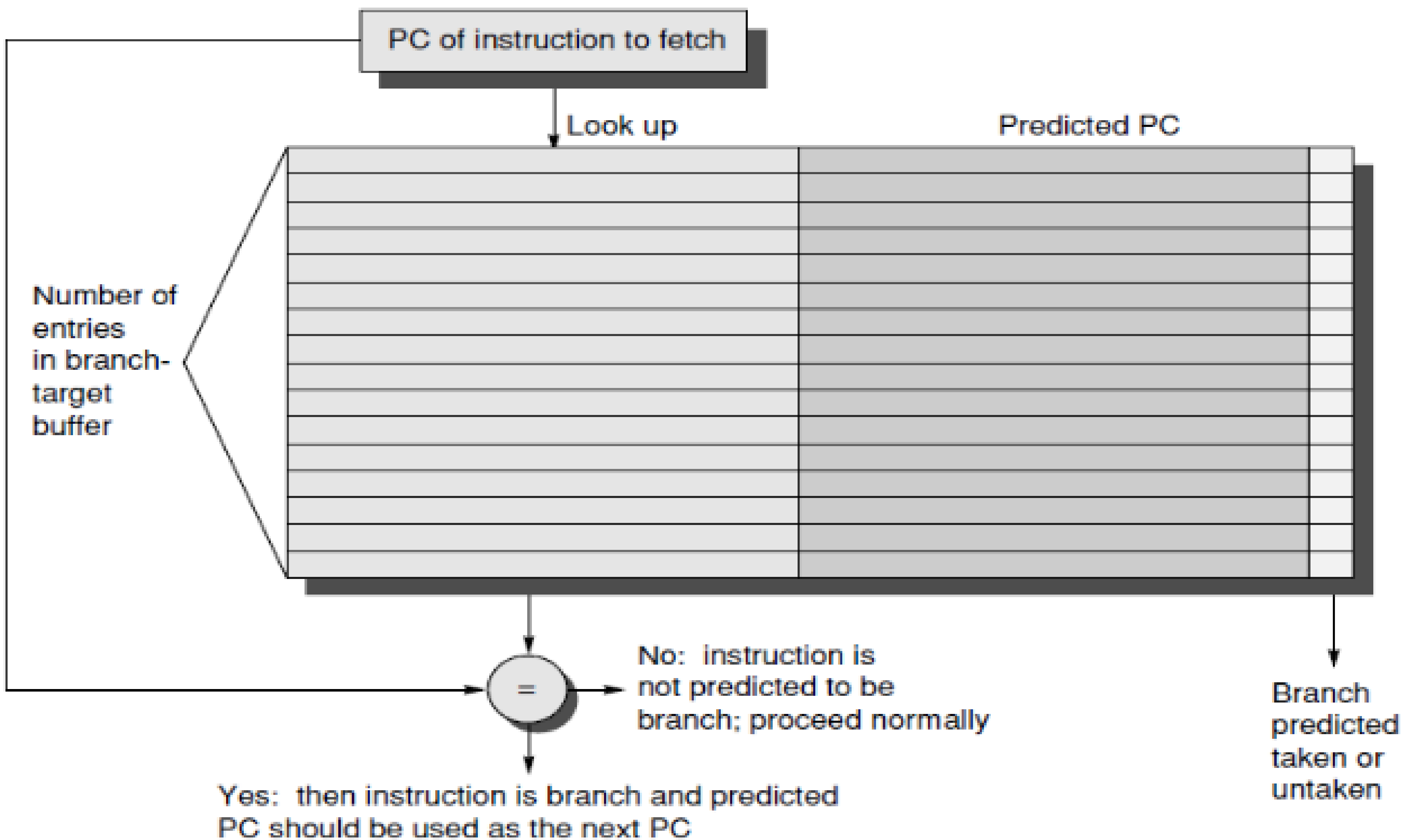
```
# x in %edx, result in %eax
  jmp    L34          #    goto Middle
L35:                  # Loop:
  imull %edx, %eax    #    result *= x
  decl   %edx         #    x--
L34:                  # Middle:
  cmpl   $1, %edx     #    x:1
  jg     L35          #    if >, goto Loop
```

特点:

- 避免了双重测试
  - 无条件跳转指令的处理器运行开销非常低（可以忽略）

△ 微体系结构背景*

条件跳转指令往往会引起一定的性能损失，Branch Prediction技术被引入来进行优化。



Branch Prediction的表项数有限，且其依据跳转与否的历史信息来做预测。因此条件跳转指令越多（一般以指令地址来识别），跳转历史信息越碎片化，就越不利于提升预测精确度。

do-while → Jump-to-middle

Branch Prediction继续发展，采用了循环预测器技术（US Patent 5909573），能够对loop进行专门的预测——即对于“循环入口”的预测基本为真。

Jump-to-middle → do-while

Switch语句

- 依据不同情况来采用不同的实现技术
  - 使用一组if-then-else语句来实现
  - 使用跳转表

Switch 语句

```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

- 多个case对应同一段处理语句
- “Fall through”
- Case值并不连续

跳转表

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
  • • •
  case val_n-1:
    Block n–1
}
```

Jump Table

jtab:

Targ0
Targ1
Targ2
•
•
•
Targ*n*-1

Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

•
•
•

Targ*n*-1: Code Block *n*–1

Switch 语句示例（x86-32)

```c
long switch_eg
    (long x, long y, long z)
{
     long w = 1;
     switch(x) {
       . . .
     }
     return w;
}
```

Setup:

```
switch_eg:
   pushl %ebp              # Setup
   movl  %esp, %ebp        # Setup
   pushl %ebx              # Setup
   movl  $1, %ebx          # w = 1
   movl  8(%ebp), %edx     # edx = x
   movl  16(%ebp), %ecx    # ecx = z
   cmpl  $6, %edx          # x:6
   ja    .L61              # if > goto default
   jmp   *.L62(,%edx,4)    # goto JTab[x]
```

表结构

- 每个表项（及跳转地址）占**4** 字节
- 基地址是 **.L62**

无条件跳转指令

jmp .L61

- Jump target is denoted by label **.L61**

jmp *.L62(,%edx,4)

*表明这是一个间接跳转，即目标地址存于内存地址中

- Start of jump table denoted by label **.L62**
- Register **%edx** holds **x**
- Must scale by factor of **4** to get offset into table
- Fetch target from effective Address **.L61 + x*4**
  - Only for $0 \le x \le 6$

表项内容

```
.section .rodata
    .align 4
.L62:
  .long    .L61   # x = 0
  .long    .L56   # x = 1
  .long    .L57   # x = 2
  .long    .L58   # x = 3
  .long    .L61   # x = 4
  .long    .L60   # x = 5
  .long    .L60   # x = 6
```

```
switch(x) {
case 1:        // .L56
    w = y*z;
    break;
case 2:        // .L57
    w = y/z;
    /* Fall Through */
case 3:        // .L58
    w += z;
    break;
case 5:
case 6:        // .L60
    w -= z;
    break;
default:       // .L61
    w = 2;
}
```

```
switch(x) {
  . . .
case 2:        // .L57
    w = y/z;
    /* Fall Through */
case 3:        // .L58
    w += z;
    break;
  . . .
default:       // .L61
    w = 2;
}
```

```
.L61:  // Default case
  movl  $2, %ebx     # w = 2
  movl  %ebx, %eax   # Return w
  popl  %ebx
  leave
  ret
.L57:  // Case 2:
  movl  12(%ebp), %eax  # y
  cltd                 # Div prep
  idivl %ecx           # y/z
  movl  %eax, %ebx # w = y/z
# Fall through
.L58:  // Case 3:
  addl  %ecx, %ebx # w+= z
  movl  %ebx, %eax # Return w
  popl  %ebx
  leave
  ret
```

```c
switch(x) {
case 1:         // .L56
    w = y*z;
    break;
  . . .
case 5:
case 6:         // .L60
    w -= z;
    break;
  . . .
}
```

```
.L60: // Cases 5&6:
   subl  %ecx, %ebx   # w -= z
   movl  %ebx, %eax   # Return w
   popl  %ebx
   leave
   ret
.L56: // Case 1:
   movl  12(%ebp), %ebx # w = y
   imull %ecx, %ebx      # w*= z
   movl  %ebx, %eax   # Return w
   popl  %ebx
   leave
   ret
```

x86-64 下的Switch语句

- 基本与32位版本一样
  - 地址长度64位

```
switch(x) {
case 1:       // .L50
     w = y*z;
     break;
   . . .
}
```

Jump Table

```
.section .rodata
   .align 8
.L62:
  .quad    .L55  # x = 0
  .quad    .L50  # x = 1
  .quad    .L51  # x = 2
  .quad    .L52  # x = 3
  .quad    .L55  # x = 4
  .quad    .L54  # x = 5
  .quad    .L54  # x = 6
```

```
.L50: // Case 1:
   movq   %rsi, %r8   # w = y
   imulq  %rdx, %r8   # w *= z
   movq   %r8, %rax   # Return w
   ret
```

Switch语句实例（case值很稀疏）

```
/* Return x/111 if x is multiple
   && <= 999.  -1 otherwise */
int div111(int x)
{
  switch(x) {
  case   0: return 0;
  case 111: return 1;
  case 222: return 2;
  case 333: return 3;
  case 444: return 4;
  case 555: return 5;
  case 666: return 6;
  case 777: return 7;
  case 888: return 8;
  case 999: return 9;
  default: return -1;
  }
}
```

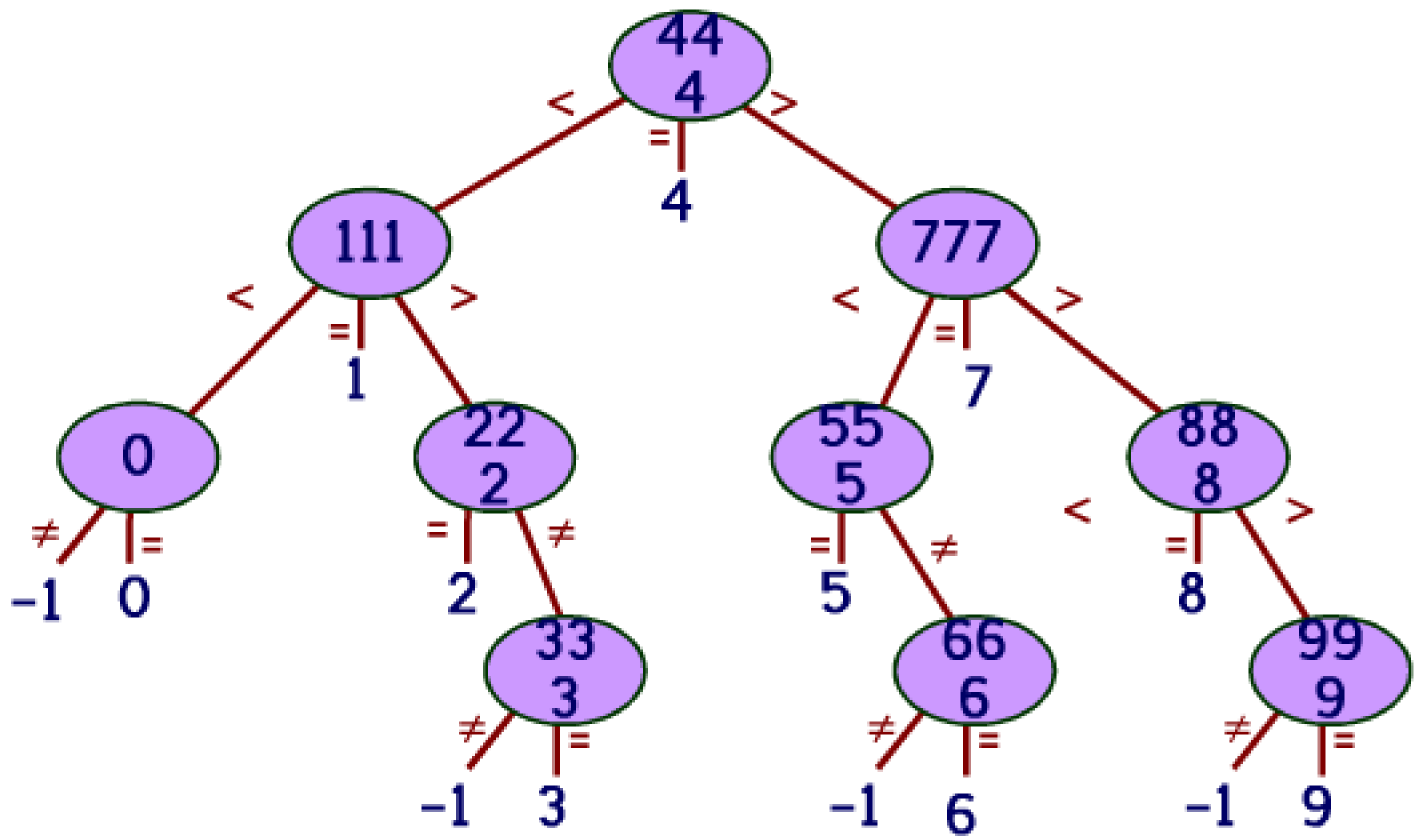
- 因此不适合使用跳转表
  - 为什么？
- 如何高效的转换为一系列的**if-then-else** 语句？

X86-32下的实例

```
    movl 8(%ebp),%eax   # get x
    cmpl $444,%eax      # x:444
    je L8
    jg L16
    cmpl $111,%eax      # x:111
    je L5
    jg L17
    testl %eax,%eax     # x:0
    je L4
    jmp L14

    . . .
```

```
    . . .
L5:
    movl $1,%eax
    jmp L19
L6:
    movl $2,%eax
    jmp L19
L7:
    movl $3,%eax
    jmp L19
L8:
    movl $4,%eax
    jmp L19
    . . .
```



■ 以二叉树的结构组织，提升性能

小结

- 条件码
  - 设置
  - 读取
  - 条件跳转指令
  - 条件传送指令

- 程序控制流
  - If-then-else
  - 循环结构
    - Do-while
    - While
    - for
  - switch语句