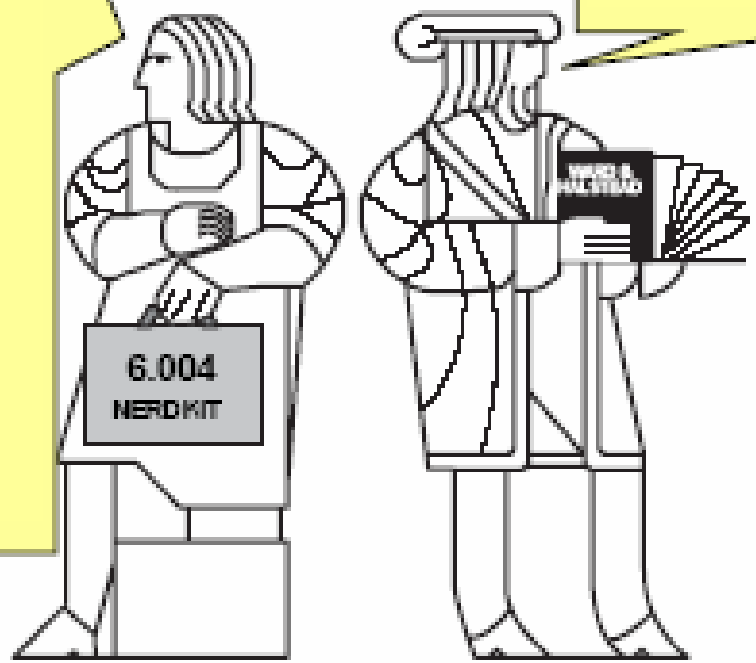
机器语言，汇编语言，与编译器

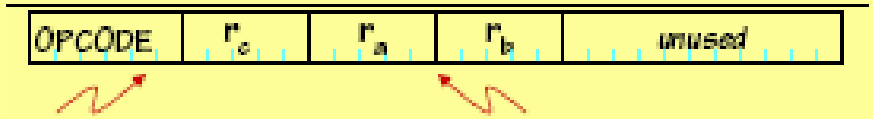
很久很久以前,我仍记得我是如何瞧不起记忆术…
而且我知道我可以玩BSim游戏与处理一些宏好一阵子, 只要我有操作码的名字.
但是6.004课程的每一堂课都令我打颤.
新来坏消息,
我无法再读一个规格.
我无法记忆当我试着最佳化阶乘,
但在我的Beta死去的那天, 有些东西触动了我讨人厌的傲慢,
于是我唱…

当我在我的程序代码中发现一堆麻烦时, 朋友和同事跟我说了一句名言, “用C来写吧”



*参考: β 文件; 实验 **#5B; C** 语言讲议*

β 机器语言: 32-bit 指令



运算: ADD, SUB, MUL, DIV

比较: CMPEQ, CMPLT, CMPLE

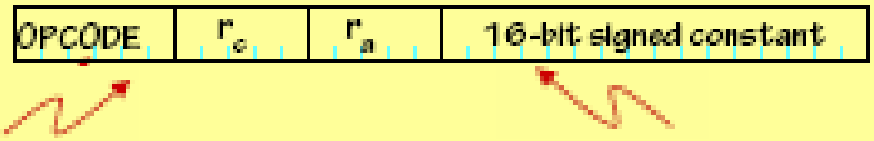
布尔: AND, OR, XOR

位移: SHL, SHR, SAR

Ra 及 Rb 是操作区域,

Rc 是目的地.

R31 为 0, 不被写入指令改变



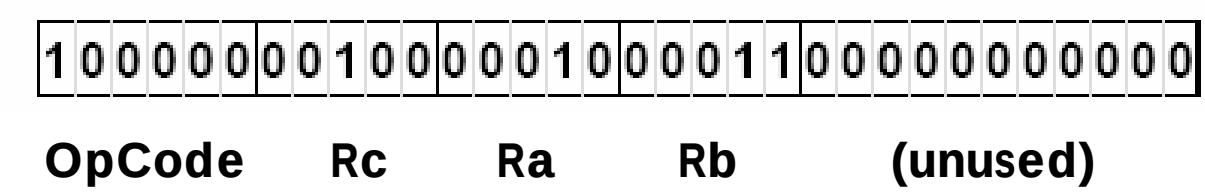
运算: ADDC, SUBC, MULC, DIVC
比较: CMPEQC, CMPLTC, CMPLEC
布尔: ANDC, ORC, XORC
位移: SHLC, SHRC, SARC
分流: BNE/BT, BEQ/BF (const = 从 PCNEXT转移的字组(word)数)
跳跃: JMP (const not used)
内存存取: LD, ST (const = 从Reg[ra]偏移的字节(byte)数)

二的补码的16位常数, 表示从 –32768 到 32767 的数字; 使用前将它延伸符号至32位.

How can we improve the programmability of the Beta?

编码二进制指令

32-bit (4-byte) ADD 指令:

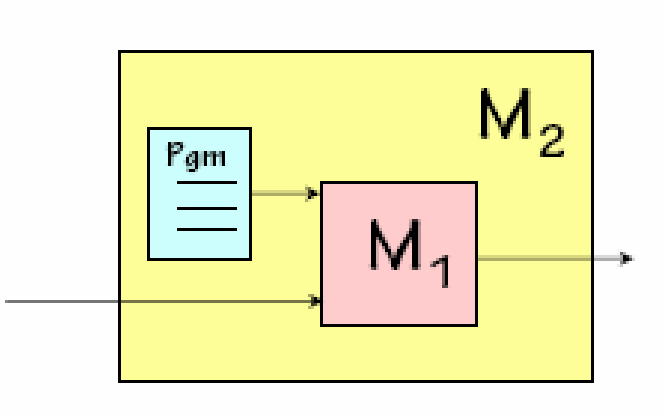


*对**BETA**来说, 表示* ***Reg[4] = Reg[2] + Reg[3]***

但, 我们通常喜欢写成 **ADD(R2, R3, R4)** **(汇编语言)**
或, 再更好一点, ***a = b+c;*** ***(高级语言)***

软件达成方法: 解译, 编译

解译

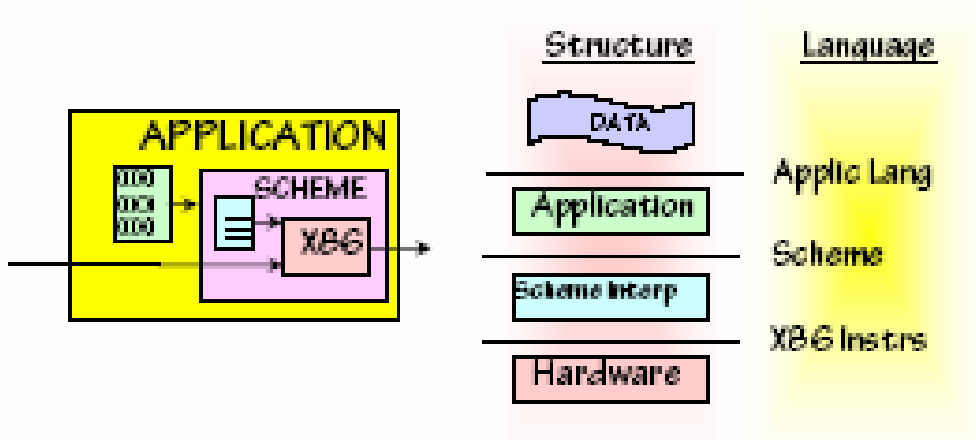


Turing的解译模型:

- 从某个"不易写程序"的通用计算机, 称之M1, 开始
- 写一个小的M1程序来模仿一个比较容易的计算机, 称之M2
- 结果: 一个"虚拟"的M2

解译的"层级":

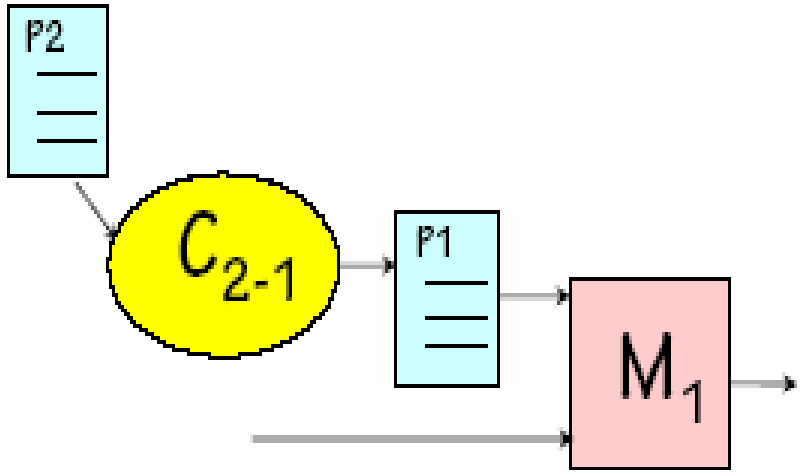
- 通常我们可以使用多个解译层级来达到想要的行为, 例如:
- **X86 (Pentium),** 执行
- 系统**(Scheme),** 执行
- 应用程序, 解译
- 资料.



编译

编译模型:

- 给定某个”不易写程序”的计算机, 称之 M1...
- 找到某个”容易写程序”的语言 L2
- (也许是一个更复杂的计算机, M2);
  以这个语言来写程序



- 建立一个翻译器(编译器) 可以将程序从M2语言翻译成M1语言. 可以在M1, M2, 或其它的计算机上执行.

解译 & 编译: 两种帮助加强可程序性的工具 ...

- 两者都可以改变程序模型
- 两者都提供与平台(如, 处理器)无关的程序应用
- 两者都在现代计算机系统中被广泛使用!

解译 与 编译 之比较

这两个强力的工具之间有一些特性上的差异...

	解译 retation	编译 pilation
如何处理 **“x+2”** put “x+2”	计算 **x+2**	产生一个程序来计算**x+2**
何时发生	执行时	执行前
哪里复杂/慢 plicates/slows	程序执行	程序建构
何时做决定 made at	执行时间	编译时间

我们会重复看到的主要的设计选项:
在编译时间或执行时间做?

软件: 抽象对策

第一步: 编译工具

组合器 (UASM):

机器语言的符号表示式

隐藏: 位级的表示式, 16进位地址, 二进制数值

编译器 (C): 算法的符号表示式

隐藏: 机器指令, 缓存器, 机器架构

接下来的步骤: 解译工具

操作系统

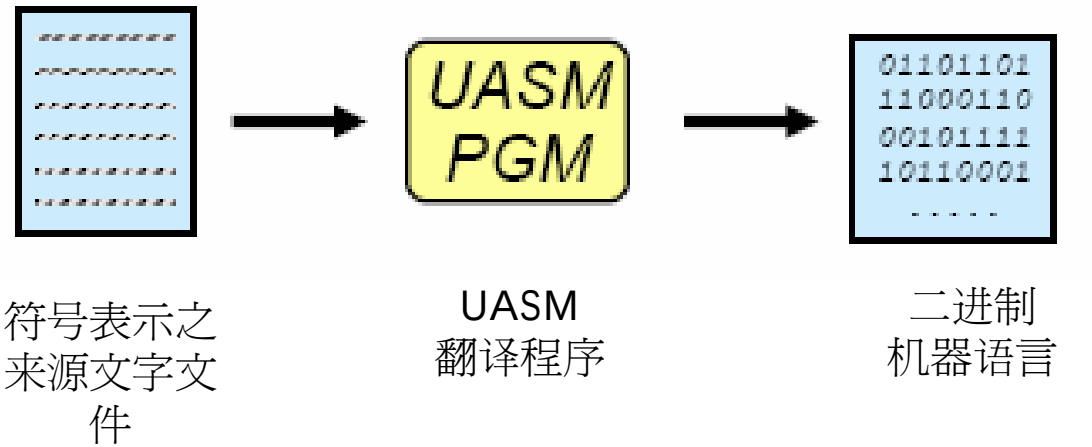
应用程序 (例如: 浏览器)

隐藏: 资源 (内存, CPU, I/O) 限制及细节

隐藏: 网络; 地址; 区域参数

抽象步骤 1:
一个写程序的程序

UASM - 6.004 (宏) 使用的汇编语言



将被加载内存的字节串流

UASM:

1. 一个符号表示的语言, 用以表示字符串
2. 一个程序 (“组合器” = 简易的编译器), 将 UASM来源翻译成二进制文件.

UASM 原始语言

一个 UASM 原始文件包含了, 符号表示的文字, 将被加载内存的连续字节的数值... 例如: 以下列的格式

```
37    -3    255

0b100101

0x25
```

十进制(预设);

二进制 (按: 使用“0b”字头表示);

十六进制 (按: 使用“0x”字头表示);

数值也可以是运算表示式; 例如, 原始档

```
37+0b10-0x10    24-0x1    4*0b110-1    0xF7&0x1F
```

产生4字节的二进制输出, 每一个都表示数值23!

符号表示之手势

我们也可以在原始文件中定义"符号"来使用:

```
x = 0x1000
y = 0x1004
| 缓存器的符号名称:
R0 = 0
R1 = 1
 ...
R31 = 31
```

一个变量地址
另一个变数

"直线" 代表一段批注的开始...

它的后面直到行末都是被忽略的

特殊变数 "." (句点) 表示下一个将被填入的字节地址:

```
. = 0x100
   1  2  3  4
five = .
   5  6  7  8
. = .+16
   9 10 11 12
```

| 组合成 **100**

| 符号 **"five"** 是 **0x104**

| 掠过 **16** 个字节

卷标(表示地址的符号)

卷标是表示内存地址的符号.

它们可以透过以下的特殊语法来设定:

x: 是 “x = .” 的缩写

例子 --

```
----  主存储器 EMORY  ----
  1000:  09  04  01  00
  1004:  31  24  19  10
  1008:  79  64  51  40
  100c:  E1  C4  A9  90
  1010:  00  00  00  10
          3   2   1   0
```

```
. = 0x1000
sqrs: 0 1 4 9
      16 25 36 49
      64 81 100 121
      144 169 196 225
slen: LONG(. - sqrs)
```

强而有力的微指令

宏乃是参数化的缩写, 或简写

```
| 产生四个连续字节的宏:
.macro consec(n)   n   n+1   n+2   n+3

| 呼叫上述的宏:
consec(37)
```

会有如下的效果:

```
   37   38   39   40
```

以下示范宏如何将多字节的数据型式以一个字节来表示

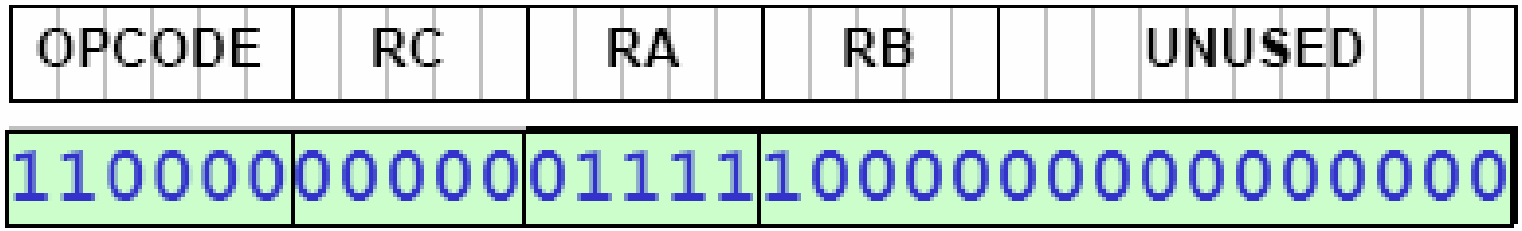
```
   | 组合little-endian的字节:
   .macro WORD(x)   x % 256   (x/256) % 256
   .macro LONG(x)   WORD(x)   WORD(x  >> 16)
=0x100
   LONG(0xdeadbeef)
```

会有如下的效果:

```
    0xef   0xbe   0xad   0xde
Mem: 0x100  0x101 0x102  0x103
```

-32768 = 1000000000000000 ADDC = 0x30 = 110000 15 = 01111 0 = 00000

指令之汇编语言



组合 **Beta** 的 **op** 指令 structions

```
.macro betaop(OP,RA,RB,RC) {
    .align 4
    LONG((OP<<26)+((RC%32)<<21)+((RA%32)<<16)+((RB%32)<<11))
}
```

“.align 4” 确保指令会在字组(word)的边缘开始(例如 address = 0 mod 4)

组合 **Beta** 的 **opc** 指令

```
.macro betaopc(OP,RA,CC,RC) {
    .align 4
    LONG((OP<<26)+((RC%32)<<21)+((RA%32)<<16)+(CC % 0x10000))
}
```

组合 **Beta** 的 **branch** 指令

```
.macro betabr(OP,RA,RC,LABEL)      betaopc(OP,RA,((LABEL-(.+4))>>2),RC)
```

For Example:

ADDC(R15, -32768, R0) --> betaopc(0x31,15,-32768,0)

最后, Beta 指令

```
| BETA Instructions:
.macro ADD(RA,RB,RC)    betaop(0x20,RA,RB,RC)
.macro ADDC(RA,C,RC)    betaopc(0x30,RA,C,RC)
.macro AND(RA,RB,RC)    betaop(0x28,RA,RB,RC)
.macro ANDC(RA,C,RC)    betaopc(0x38,RA,C,RC)
.macro MUL(RA,RB,RC)    betaop(0x22,RA,RB,RC)
.macro MULC(RA,C,RC)    betaopc(0x32,RA,C,RC)
   :
   :
.macro LD(RA,CC,RC)     betaopc(0x18,RA,CC,RC)
.macro LD(CC,RC)        betaopc(0x18,R31,CC,RC)
.macro ST(RC,CC,RA)     betaopc(0x19,RA,CC,RC)
.macro ST(RC,CC)        betaopc(0x19,R31,CC,RC)
   :
   :
.macro BEQ(RA,LABEL,RC) betabr(0x1D,RA,RC,LABEL)
.macro BEQ(RA,LABEL)    betabr(0x1D,RA,r31,LABEL)
.macro BNE(RA,LABEL,RC) betabr(0x1E,RA,RC,LABEL)
.macro BNE(RA,LABEL)    betabr(0x1E,RA,r31,LABEL)
```

方便的宏. 因此我们不需要指定**R31**缓存器...

(采自 **beta.uasm)**

汇编语言的例子

```
ADDC(R3,1234,R17)
```

以**RA=R3, C=1234, RC=R17** 展开 **ADDC** 宏

```
betaopc(0x30,R3,1234,R17)
```

以**OP=0x30, RA=R3, CC=1234, RC=R17** 展开 **beta** 的 **opc** 宏

```
.align 4
LONG((0x30<<26)+((R17%32)<<21)+((R3%32)<<16)+(1234 % 0x10000))
```

以**X=0xC22304D2**展开 **LONG** 宏

```
WORD(0xC22304D2)    WORD(0xC22304D2 >> 16)
```

以**X=0xC22304D2** 展开第一个 **WORD** 宏

```
0xC22304D2%256    (0xC22304D2/256)%256    WORD(0xC223)
```

计算表示式, 并以**X=0xC223** 展开第二个 **WORD** 宏

```
0xD2    0x04    0xC223%256    (0xC223/256)%256
```

计算表示式

```
0xD2    0x04    0x23    0xC2
```

找不到指令? 假造一个!

方便的宏可以用来延伸我们的组个语言:

```
.macro MOVE(RA,RC)          ADD(RA,R31,RC)     | Reg[RC] <- Reg[RA]
.macro CMOVE(CC,RC)         ADDC(R31,C,RC)     | Reg[RC] <- C

.macro COM(RA,RC)           XORC(RA,-1,RC)     | Reg[RC] <- ~Reg[RA]
.macro NEG(RB,RC)           SUB(R31,RB,RC)     | Reg[RC] <- -Reg[RB]
.macro NOP()                ADD(R31,R31,R31)   | do nothing

.macro BR(LABEL)            BEQ(R31,LABEL)     | always branch
.macro BR(LABEL,RC)         BEQ(R31,LABEL,RC)  | always branch
.macro CALL(LABEL)          BEQ(R31,LABEL,LP)  | call subroutine
.macro BF(RA,LABEL,RC)      BEQ(RA,LABEL,RC)   | 0 is false
.macro BF(RA,LABEL)         BEQ(RA,LABEL)
.macro BT(RA,LABEL,RC)      BNE(RA,LABEL,RC)   | 1 is true
.macro BT(RA,LABEL)         BNE(RA,LABEL)

| Multi-instruction sequences
.macro PUSH(RA)             ADDC(SP,4,SP)   ST(RA,-4,SP)
.macro POP(RA)              LD(SP,-4,RA)    ADDC(SP,-4,SP)
```

(采自 **beta.uasm)**

抽象步骤 2:
高级语言

大部分的算法是以高阶的语言来表示. 例如以下的算法:

```
struct Employee
 { char *Name;  /* Employee's name. */
   long Salary; /* Employee's salary. */
   long Points;}/* Brownie points. */

/* Annual raise program. */
Raise(struct Employee P[100])
 { int i = 0;
   while (i < 100)
    { struct Employee *e = &P[i];
      e->Salary =
            e->Salary + 100 + e->Points;
      e->Points = 0;  /* Start over! */
      i = i+1;
    }
 }
```

我们用了 (并将会继续在6.004课中使用) C语言. 它是一个"成熟"而且常见的系统. 较新常用的代替语言有: C++, Java, Python, 以及很多其它的.

为什么不用汇编语言而要用这个呢?

- 可读性
- 简洁
- 清楚, 不会模棱两可
- 可携带 (算法经常存活地比硬件平台来得久)
- 可靠 (数据型态检查, 等等)

参考: C 讲义 (6.004 网页)

编译器是如何运作的呢?

当化的编译器远比UASM的宏延伸还做的更多. 它们

- 执行精细的原始码分析
- 使用特定的算法来产生给目的计算机用的有效率的程序代码
- 对原始码及目标程序码都采行“最佳化” 程序, 以得到较好的执行时间效能.

编译一个未最佳化的程序代码是很直接的... 以下是一些简短的介绍.

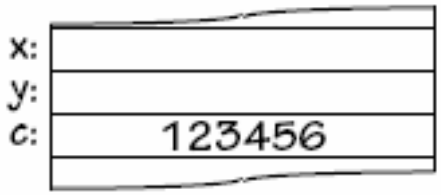
编译器所做的事并不全是那么地复杂.

(至少, 原则上)

编译表示式

C 语言码:

```
int x, y;
y = (x-3)*(y+123456)
```



Beta 汇编语言码:

```
x:      LONG(0)
y:      LONG(0)
c:      LONG(123456)
...

        LD(x, r1)
        SUBC(r1,3,r1)
        LD(y, r2)
        LD(C, r3)
        ADD(r2,r3,r2)
        MUL(r2,r1,r1)
        ST(r1,y)
```

- 变量 被指定为内存位置, 并透过 LD 或 ST 来存取
- 操作 翻译成 ALU 指令
- 小常数 翻译成 “文字模式” ALU 指令
- 大常数 翻译成 初始化的变数

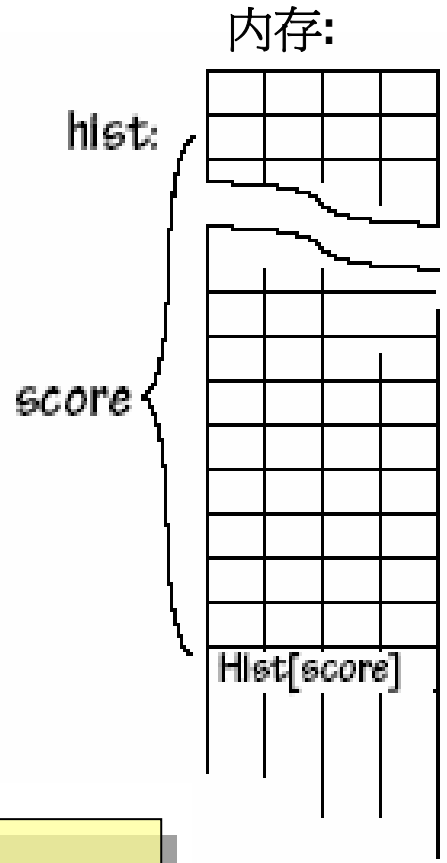
数据结构: 数组

C 原始码

```
int Hist[100];
...
Hist[score] += 1;
```

可能翻译成:

```
hist:   .=.+4*100 | Leave room for 100 ints
...
<score in r1>
MULC(r1,4,r2)     | index -> byte offset
LD(r2,hist,r0)    | hist[score]
ADDC(r0,1,r0)     | increment
ST(r0,hist,r2)    | hist[score]
```



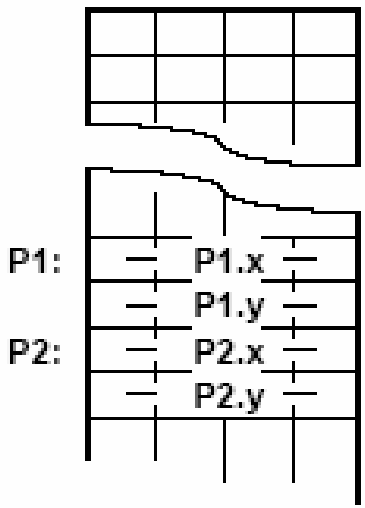
地址:

常数为基底的地址 **+**
由索引计算得到的变量偏移量

数据结构: 结构

```
struct Point
 { int x, y;
 } P1, P2, *p;
...
P1.x = 157;
...
p = &P1;
p->y = 157;
```

内存:



可能翻译成:

```
P1: .=.+8
P2: .=.+8
x=0              | x 组件的偏移量
y=4              | y 组件的偏移量
...
ADDC(r31,157,r0)     | r0 <- 157
ST(r0,P1+x)          | P1.x = 157
...
<p in r3>
ST(r0,y,r3)          | p->y = 157;
```

地址:
变量为基底的地址 + 常数的组件偏移量

条件式

C code:

```
if (expr)
  {
     STUFF
  }
```

Beta 汇编语言:

```
    (compile expr into rx)
    BF(rx, Lendif)
    (compile STUFF)
Lendif:
```

C code:

```
if (expr)
  {
     STUFF1
  }
else
  {
     STUFF2
  }
```

Beta 汇编语言:

```
    (compile expr into rx)
    BF(rx, Lelse)
    (compile STUFF1)
    BR(Lendif)
Lelse:
    (compile STUFF2)
Lendif:
```

编译条件码区块时, 有一些小小的技巧. 举例来说, 以下的表示式:

```
if (y > 32)
  {
     x = x + 1;
  }
```

编译成: to:

```
  LD(y,R1)
  CMPLEC(R1,32,R1)
  BT(R1,Lendif)
  ADDC(R2,1,R2)
Lendif:
```

并没有 **>32** 这样的指令!

循环

把这个测试式移到循环的后面并且第一次在这里跳到那个测试式... 这样可以省一个分流指令

C 程序代码:

```
while (expr)
 {
    STUFF
 }
```

Beta 汇编语言:

```
Lwhile:
    (compile expr into rx)
    BF(rx,Lendwhile)
    (compile STUFF)
    BR(Lwhile)
Lendwhile:
```

替代的 Beta 汇编语言:

```
    BR(Ltest)
Lwhile:
    (compile STUFF)
Ltest:
    (compile expr into rx)
    BT(rx,Lwhile)
Lendwhile:
```

编译器要花上许多的时间把循环里面和周遭最佳化.

- 把所有可能的计算移到循环外面
- 展开循环来减少分流指令产生的效能消耗
- 化简表式示对“循环变量”的依赖度

最佳化 我们最爱的阶乘程序

```
int n = 20, r;          n: LONG(20)
                        r: LONG(0)

r = 1;                  ADDC(r31, 1, r0)
                        ST(r0, r)
                    loop:
while (n > 0)           LD(n, r1)
{                       CMPLT(r31, r1, r2)
                        BF(r2, done)
  r = r*n;              LD(r, r3)
                        LD(n,r1)
                        MUL(r1, r3, r3)
                        ST(r3, r)
  n = n-1;              LD(n,r1)
                        SUBC(r1, 1, r1)
                        ST(r1, n)
}                       BR(loop)
                    done:
```

聪明之处:
没有…
直接了当地编译

(循环中有11个指令...)

最佳化

```
int n = 20, r;

r = 1;

while (n > 0)
{
  r = r*n;
  n = n-1;
}
```

```
n:  LONG(20)
r:  LONG(0)

        ADDC(r31, 1, r0)
        ST(r0, r)
        LD(n,r1)        ; keep n in r1
        LD(r,r3)        ; keep r in r3
loop:
        CMPLT(r31, r1, r2)
        BF(r2, done)
        MUL(r1, r3, r3)
        SUBC(r1, 1, r1)
        BR(loop)
done:
        ST(r1,n)        ; save final n
        ST(r3,r)        ; save final r
```

聪明之处:
把LD与ST指令移到循环外面!

(在循环中还有 5 个指令...)

完成最佳化...

```
int n = 20, r;

r = 1;

while (n > 0)
{ r = r*n;
  n = n-1;
}
```

```
n:  LONG(20)
r:  LONG(0)

    LD(n,r1)          ; keep n in r1
    ADDC(r31, 1, r3)  ; keep r in r3
    BEQ(r1, done)     ; why?
loop:
    MUL(r1, r3, r3)
    SUBC(r1, 1, r1)
    BNE(r1,loop)
done:
    ST(r1,n)          ; save final n
    ST(r3,r)          ; save final r
```

聪明之处:
避免了条件判断式的消耗!

(现在循环中只剩3个指令...)

不幸地, $20! = 2,432,902,008,176,640,000 > 2^{61}$

$12! = 479,001,600 = 0x1c8cfc00$

下回:
程序 及 堆栈