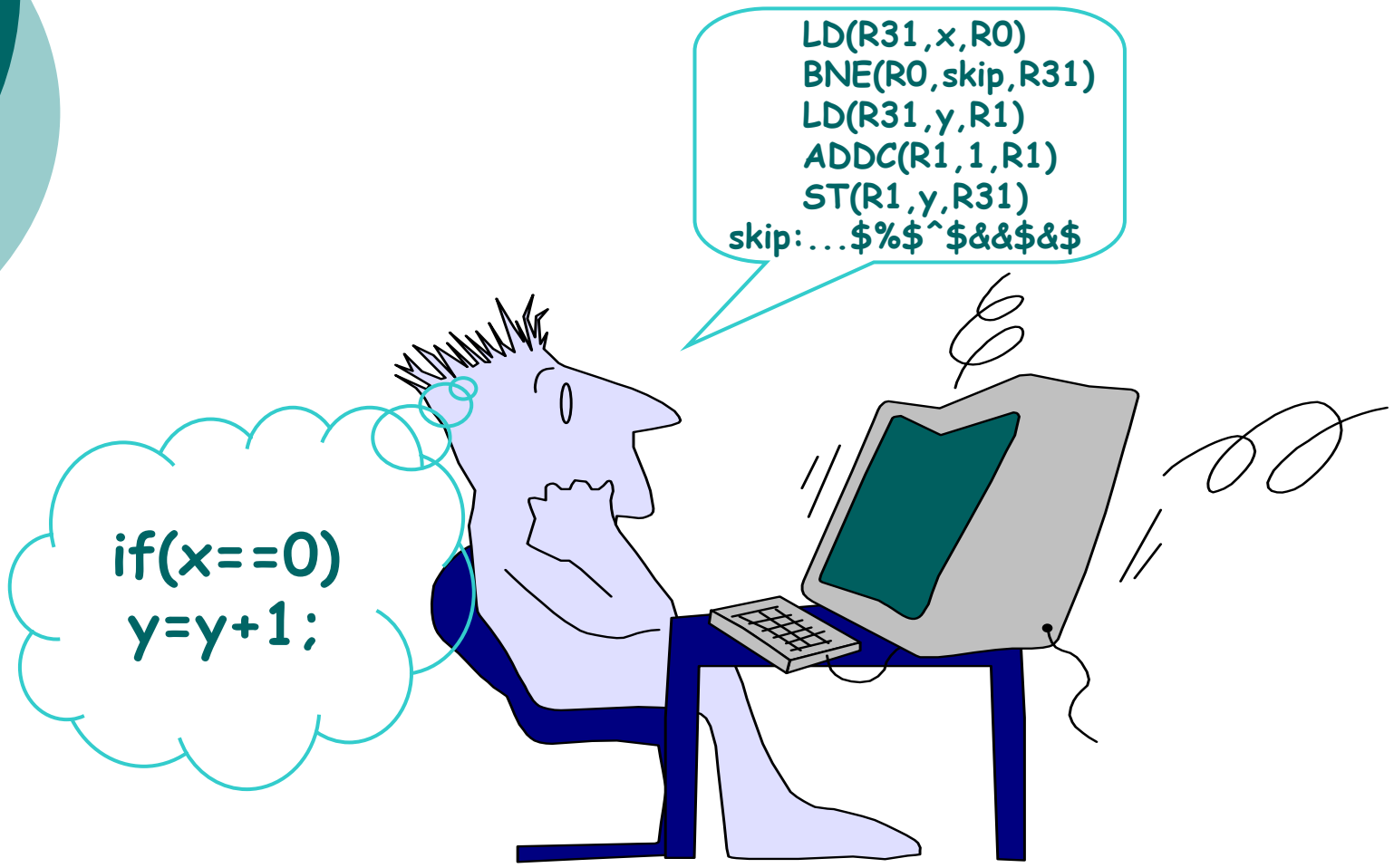
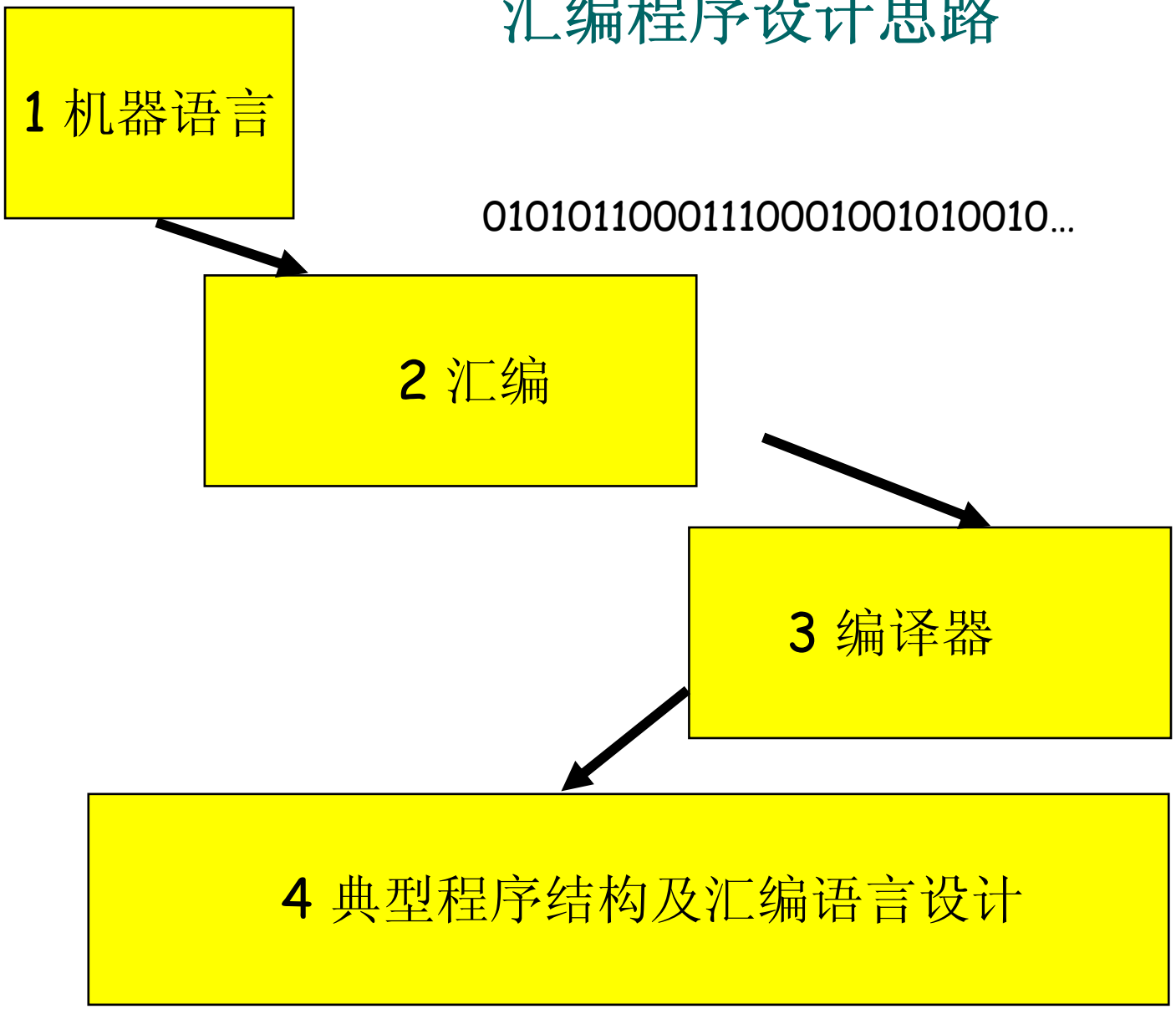
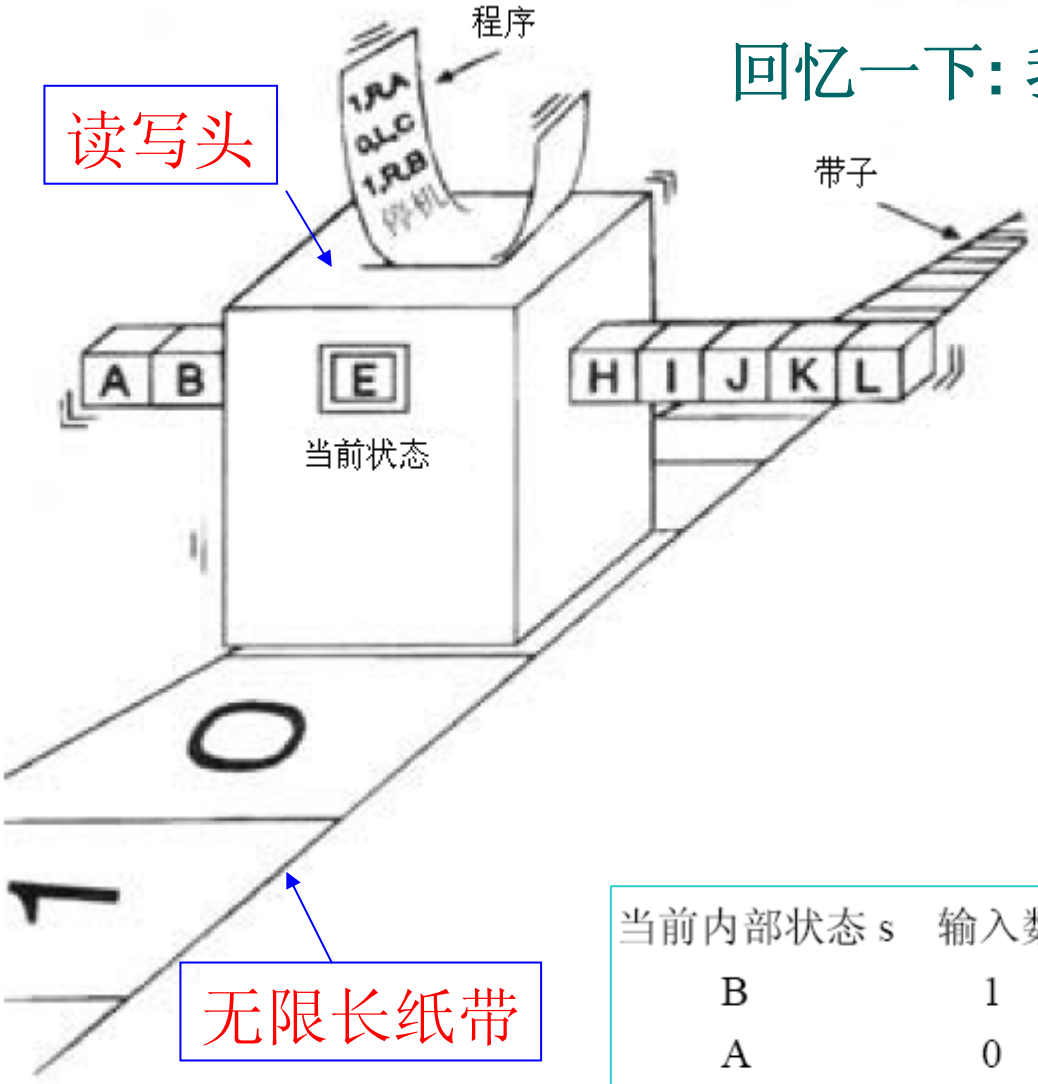
MIPS汇编语言程序设计



汇编程序设计思路


1 机器语言
0101011000111000100101001O...
2 汇编
3 编译器
4 典型程序结构及汇编语言设计



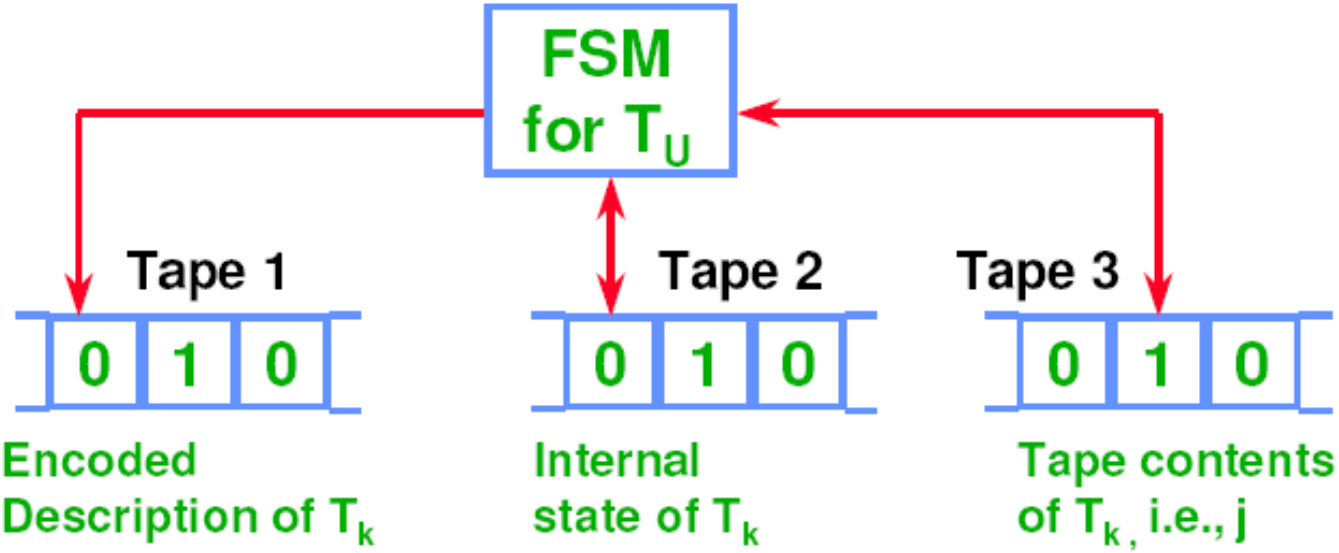
图灵机模型

回忆一下: 我们已经有了UTM的思想

- 图灵机只要根据每一时刻读写头读到的一个方格的信息和当前的内部状态对程序进行查表，就可以确定下一时刻的内部状态和输出动作了。
- 具体的程序是一个列表，也叫做规则表。

当前内部状态 s	输入数值 i	输出动作 o	下一时刻的内部状态 s'
B	1	前移	C
A	0	往纸带上写 1	B
C	0	后移	A
…	…	…	…

回忆一下: 我们已经有了UTM的思想



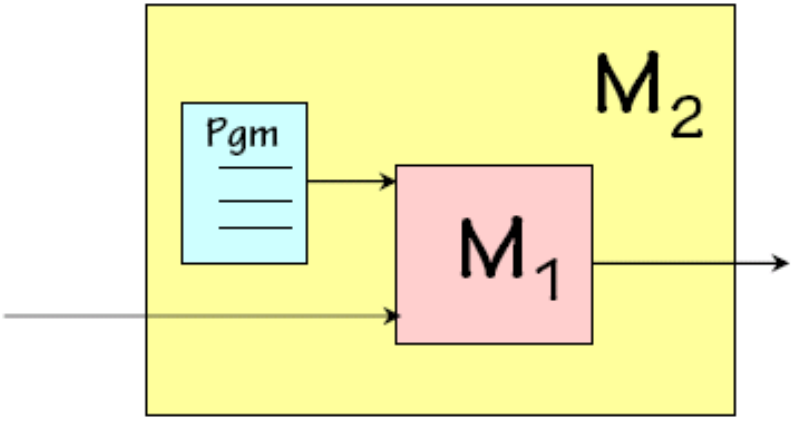
1：T_u查看Tape2和Tape3来决定图灵机的配置T_k（采用第k个图灵机)及它的输入j;

2：T_u查看第一个磁带Tape1决定T_k会怎么做;

3：修改磁带2和3以反映T_k(j)在计算过程中的动作和内部状态变化;

我们的任务是计算所有的可计算的整数函数

解释:Interpretation

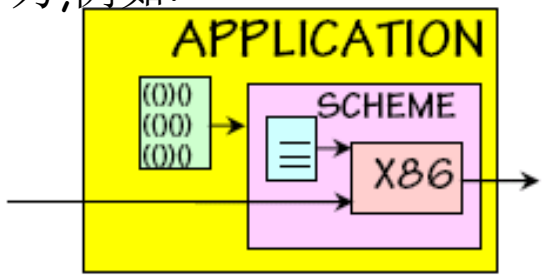


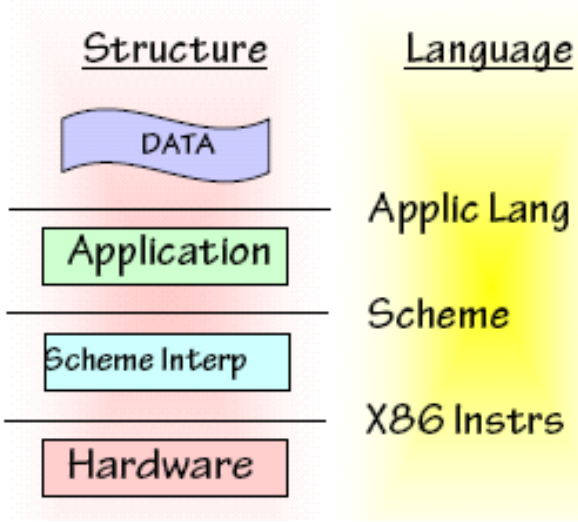
Turing的解释器模型:

- 首先是有一个与问题无关的通用机器UTM; M1
- 为M1编写一个简单的程序使得M1完成一个简单的功能M2;
- M1和程序Pgm一起变成了一个虚拟的(Virtual)M2;

解释的层次:

- 在计算机中通常我们会使用多个不同层次的解释器来实现我们想要的行为,例如:
- X86(pentium)上面运行着
  - Java VM上面又运行着
    - 应用程序比如ant
      - 数据结构比如走迷宫的蚂蚁

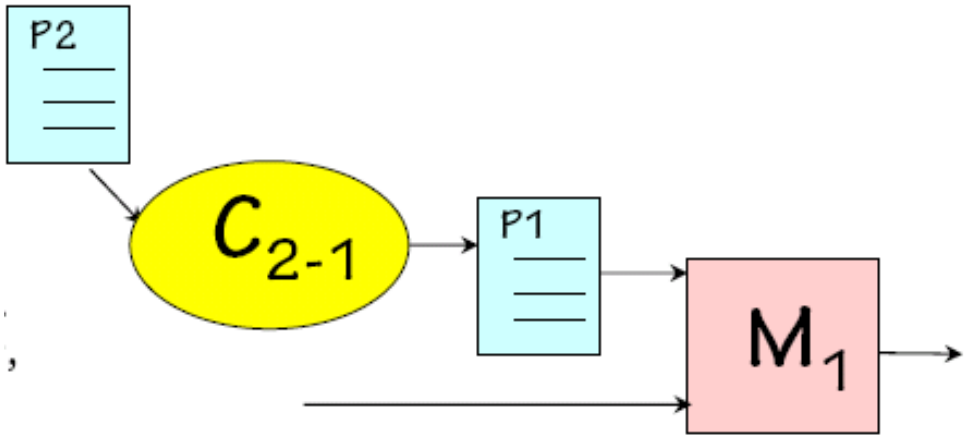




编译器

编译器的模型：

- ✓给定一个UTM M1;
- ✓寻找一个比较容易写的语言L2(比如是为了机器M2'设计的)并写一段程序;
- ✓编写一个Translator使得这个Translator可以将M_2的语言转换为M_1的语言



Interpretation & Compilation: 两个不同的提高可编程性的方法.

- ✓都容许改变我们的编程的模型
- ✓都可以提供一种与平台(如处理器)无关的编程语言;
- ✓都是在现代的计算机系统中被广泛采用

Interpretation与Compilation

这两种强有力的工具也有区别:

	Interpretation	Compilation
如何对待"x+2"	计算 x+2	产生一个可以计算x+2的程序
什么时候计算	在执行过程中	在执行之前
什么代价(使什么变慢)	程序执行时间	程序的开发时间
结果在什么时候确定	运行时	编译时

我们以后会经常遇到的一个关键问题:

是在运行时实现还是在编译时实现?

MIPS汇编器：语法

- 注释行以“#”开始；
- 标识符由字母、下划线（_）、点（.）构成，但不能以数字开头，指令操作码是一些保留字，不能用作标识符；
- 标号放在行首，后跟冒号（:），例如

```
        .data           #将子数据项，存放到数据段中
Item:   .word 1,2       #将2个32位数值送入地址连续的内存字中
        .text           #将子串即指令或字送入用户文件段
        .global main    #必须为全局变量
Main:   lw $t0, item
```

MIPS汇编中的命令见Page625

MIPS汇编器：存储器中位置

- 汇编语言源文件：.s
- 特殊符号“.”(点)：表示当前位置
  - “.” MIPS汇编命令标识符
  - “label:”
    - label被赋值为当前位置的地址
    - Fact = 0x00400100
  - 编译时就确定了
    - 汇编程序在地址0x00400000开始
  - 在LW,SW, BEQ, BNE等指令中都有应用

```
0x00400020        move $s5, $31
0x00400024        beq $0,$0,fact
0x00400028        sw   $s0,f($0)
…                 .text 0x00400100
0x00400100  fact: addiu $s0,$0,1
0x00400104        lw    $s1, n($0)
0x00400108  loop: mul   $s0,$s1,$s0
0x0040010C        subu  $s1,$s1,1
0x00400110        bnez  $s1,loop
0x00400114        addi   $31,$s5,4
0x00400118        jr     $31
…                 .data 0x10000200
0x10000200  n:    .word 4
0x10000204  f:    .word 0
```

能运行的版本（1）：

```
    .text
main:
   addu $s3,$ra,$0
   ori $s6,$0,0x1000
   sll $s6,$s6,16
   addiu $s4,$s6,0x0200
   addiu $s5,$s6,0x0204
   addiu $s7,$s6,0x0208
      #$s4=n, $s5=f
      #$s7为$ra开辟一个地址空间

         beq $0,$0, fact
result: sw $s0,0($s5)
         addu $ra,$s3,$0
         jr $ra
```

```
         .text 0x00400100
fact:
  sw  $ra,0($s7)
  addiu $s0,$0,1
  lw $s1,0($s4)            #$s0=n!
loop:   mul $s0,$s1,$s0
         subu $s1,$s1,1
         bnez $s1,loop     #f=n!
         j   result

         .data 0x10000200
n:       .word 4
f:       .word 0
```

test1

```
        .data 0x10000000    #下列语句行是数据代码行
        .word 4,0           #定义了两个字型立即数4和0
        .text               #下列语句行是指令代码行
main:   addu $s7,$ra,$0     #
        ori $s6,$0,0x1000   #
        sll $s6,$s6,16      #$s6=0x10000000
        addiu $s5,$s6,0x04  #$s5=0x10000004
fact:   addiu $s0,$0,1      #
        lw $s1,0($s6)       #
loop:   mul $s0,$s1,$s0     #$s0=n!, n=4
        subu $s1,$s1,1      #
        bnez $s1,loop       #
        sw $s0,0($s5)       #f=n!=24
        addu $ra,$0,$s7     #退出子程序
        jr $ra              #根据ra寄存器中的返回地址返回
```

能运行的版本（2）

编程指南

（1）变量

（2）分支

（3）数组

（4）过程调用

（5）阅读、改进程序

单指令计算机

编程指南：（1）变量

- 变量存储在主存储器内（而不是寄存器内）
  - 因为我们通常有很多的变量要存，不止32个
- 为了实现功能，用LW 语句将变量加载到寄存器中，对寄存器进行操作，然后再把结果SW回去
- 对于比较长的操作**(e.g., loops):**
  - 让变量在寄存器中保留时间越长越好
  - LW and SW 只在一块代码开始和结束时使用
  - saves on instructions
  - also, 事实上LW and SW 比寄存器操作要慢得多得多！
- 由于一条指令只能采用两个输入,所以必须采用临时寄存器计算复杂的问题e.g., (x+y)+(x-y)

编程指南：（1）变量

```
        .data 0x10000000
        .word 4,0
        .text
main:   addu $s7,$ra,$0
        ori $s6,$0,0x1000
        sll $s6,$s6,16
        addiu $s5,$s6,4
fact:   addiu $s0,$0,1
        lw $s1,0($s6)          #s1 get 4
loop:   mul $s0,$s1,$s0
        subu $s1,$s1,1
        bnez $s1,loop
        sw $s0,0($s5)          #s0 hold result
        addu $ra,$0,$s7
        jr $ra                 #return result in s0
```

编程指南：（2）分支

- 在符号汇编语句中,分支语句的目标位置是用绝对地址方式写的
  - e.g., **beq $0,$0,fact**
    means PC ← 0x00400100
- 不过在实现中,要用相对于PC的地址来定义
  - e.g., **beq $0,$0,0x3f**
    means PC ← 0x00400100

```
        .text
main:   addu $s3,$ra,$0
        ori $s6,$0,0x1000
        sll $s6,$s6,16
        addiu $s4,$s6,0x0200
        addiu $s5,$s6,0x0204
        addiu $s7,$s6,0x0208
        beq $0,$0, fact
result: sw $s0,0($s5)
        addu $ra,$s3,$0
        jr $ra
        .text 0x00400100
fact:   sw  $ra,0($s7)
        addiu $s0,$0,1
        lw $s1,0($s4)    #$s0=n!
loop:   mul $s0,$s1,$s0
        subu $s1,$s1,1
        bnez $s1,loop    #f=n!
        j   result
        .data 0x10000200
n:      .word 4
f:      .word 0
```

分支语句中的偏移量的使用

- 偏移量= 从下一条指令对应的PC开始到标号位置还有多少条指令
  - e.g., **beq $0,$0,fact**如果位于地址0x00400000 的话,
    word displacement=**(target - (<PC>+4)) / 4**
    =(0x00400100-0x00400004)/4
    =0xfc/4=0x3f
  - 偏移量为0则表示执行下一条指令不产生任何跳转
- 为什么在代码中用相对的偏移量?
  - *relocatable* 代码(可重新定位的)
  - 分支语句可以在每次被加载到内存不同位置的情况下正常工作

分支：返回地址与 Jr

- 记得: BEQ和BNE默认$31寄存器获得的是下一个next-PC
  - e.g., in **beq $0,$0,fact**
    R31 gets 0x0040001c
- 此即返回地址*return-address* 这样我们有一天还可以跳回来~这很重要!
  - e.g., jr $31

  我们就回到了
  0x0040001c

```
                        .text
0x00400000  main:   addu $s3,$ra,$0
0x00400004          ori $s6,$0,0x1000
0x00400008          sll $s6,$s6,16
0x0040000c          addiu $s4,$s6,0x0200
0x00400010          addiu $s5,$s6,0x0204
0x00400014          addiu $s7,$s6,0x0208
0x00400018          beq $0,$0, fact
0x0040001c  result:  sw $s0,0($s5)
                     addu $ra,$s3,$0
...                  jr $ra

                    .text 0x00400100
0x00400100  fact:   sw  $ra,0($s7)
0x00400104          addiu $s0,$0,1
0x00400108          lw $s1,0($s4)#$s0=n!
...         loop:   mul $s0,$s1,$s0
                    subu $s1,$s1,1
                    bnez $s1,loop    #f=n!
                    j   result

                  .data 0x10000200
0x10000200  n:      .word 4
0x10000204  f:      .word 0
```

编程指南：（2）分支

- 分支
  - 如果和 0比较, 则直接使用blez,bgez,bltz,bgtz, bnez
    - e.g., loop example before
  - 更复杂的比较, 采用比较指令（如slt）, 然后再用与0比较
- Example:

```
if ( x >= 0 )
    y = x;
else
    y = -x;
```

编程指南:分支 test2

```
        .data 0x10000000
        .word -6,0                  #x：-6，y：0
        .text
main:   addu  $s7,$ra,$0            #计算内存中数据存放地址
        ori   $s6,$0,0x1000
        sll   $s6,$s6,16
        addiu $s5,$s6,4             #$s6=x，  $s5=y
        lw    $s0,0($s6)
        slt   $s2,$0,$s0            #0<x, $s2=1
        beqz  $s2,else              #$s2=0, 跳到else
        move $s1,$s0                #$s2=1, 跳到done
        j done
else:   sub   $s1,$0,$s0
done:   sw    $s1,0($s5)
        addu  $ra,$0,$s7
        jr    $ra
```

功能：求绝对值

编程指南:（3）数组array

- 用 .word来给数组开辟空间
  - 在编译时*静态地*开辟n*4 bytes, (n个32-bit 字)
    n=17 ？
- 使用LW和SW的$2和$4

  LW $1, const($2)

  SW $3, const($4)
  - 将 *const* 作为数组偏移量
  - 将寄存器**$2**和**$4**作为数组中的开始地址(A[0])

#编程指南:（3）数组array test3

```
        .data 0x10000000
        .word 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17
        .text
main:   addu $s7,$ra,$0
        ori $s5,$0,0x1000
        sll $s5,$s5,16          #$s5=A[]=0x10000000,
        addiu $s6,$s5,0x400     #$s6=B[]=0x10000400
        addiu $s0,$0,0x11       #Size(A)=Size(B)=0x11
loop:   subu $s0,$s0,1
        addiu $s1,$0,4
        mul $s2,$s1,$s0
        addu $s3,$s2,$s5        #计算A[]偏移量,送到$s3
        lw $s4,0($s3)           #读出A[]中的值
        addu $s3,$s2,$s6        #计算B[]偏移量,送到$s3
        sw $s4,0($s3)           #写到B[]中去
        bnez $s0,loop
        addu $ra,$0,$s7         #返回调用程序
        jr $ra
```

编程指南:（3）数组array——数组访问

- SLLV **by** k 等价于 MUL **by** 2^k
- SRLV **by** k 等价于 DIV **by** 2^k

} 只对无符号数成立

- 很有用，因为 MUL 和DIV一般都比SLLV 和SRLV慢得多
  - 想想怎么实现MUL和DIV~
- 对于有符号数用 SRA
  - 高位用符号位填充(在2'的补码表示情况下)
  - e.g.,
    R1 = -6 = 0b11…11010
    SRLV $R1,$R1,1 → 0b01…11101
    SRAV $R1,$R1,1 → 0b11…11101=-3
  - 想想为什么这样是对的 …

编程指南：（3）数组array——数组访问

```
add $s0,$0,$a2      # i = N
loop: subu $s0 $s0,1    # i--
      addiu $s1,$0,4    # $s1=4
      mul $s2,$s1,$s0   # dest = i*4
      addiu $t0,$0,2    # $t0=2
      div $s0,$t0       # $lo=floor(i/2)
      mflo $t1          # $t1=floor(i/2)
      mul $t1,$s1,$t1   # $t1=$t1*4
      add $t2,$t1,$a0
      lw $s4,0($t2)     # $s4=A[i/2]
      add $t3,$s2,$a1
      sw $s4,0($t3)     # B[i] =$s4
     bnez $s0,loop      # while(i!=0) loop
done: …
```

#数组长度
N = 0x100
# 存放在$a2 = 0x100

A的地址存放在$a0中
B的地址存放在$a1中

编程指南：（3）数组array——数组访问

Side Note #1: 用移位代替乘法

```
    add $s0,$0,$a2      # i = N
loop:
    subu $s0 $s0,1      # i--
    sllv $s2,$s0,2      # dest = i*4
    sra $t1,$s0,1       # $t1=floor(i/2)
    sllv $t1,$t1,2      # $t1=$t1*4
    add $t2,$t1,$a0
    lw $s4,0($t2)       # $s4=A[i/2]
    add $t3,$s2,$a1
    sw $s4,0($t3)       # B[i] = $s4
    bnez $s0,loop       # while(i!=0) loop
done:
```

- 对无符号数
  - SLLV **by** k 等价于 MUL **by** 2^k
  - SRLV **by** k 等价于 DIV **by** 2^k
- 对于有符号数用 **SRA**
  - 高位用符号位填充(在2'的补码表示情况下)

编程指南:（4）过程调用

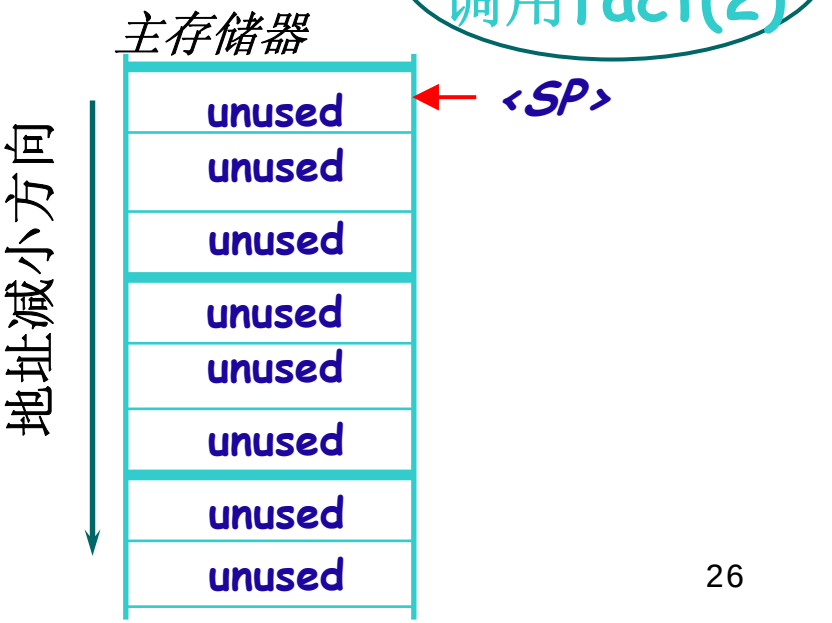
- 用汇编写程序时需要明确说明每一次的调用和返回
  - Why？为什么用C编写程序时不需要了解子程序被调用的具体细节？
- 大多数有关过程调用的记录集中在一个叫做过程调用帧的内存块中，实现：
  - 以参数形式保存调用值
  - 被调用过程不希望被调用过程修改的寄存器的值
  - 为程序的变量提供足够的内存空间
- 程序的调用和返回严格遵从后进先出（**LIFO**）原则

编程指南:（4）过程调用

- 我们需要存储:
  - 返回地址(old ra)
  - 参数 *x*, *y*, *z* in *f(x, y, z)*
  - 临时/局部的变量 （在*f*执行过程中）
  - 被破坏的寄存器
- 想法: 存在栈里!
  - 增长(PUSH 进去) 栈，（每次调用函数时）
  - 缩减栈(POP 出来) （每次返回时）
  - 每次调用都有自己的"栈框架"

```
int fact( int n )
{
    int x, y, z, a, b;
    if (n > 0)
      return fact( n-1 ) * n;
    else
      return 1 ;
}
```

例如 调用fact(2)

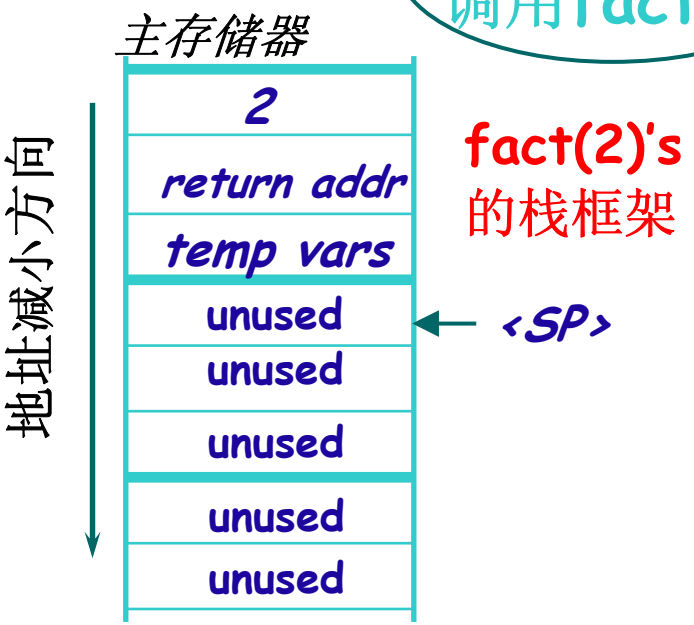


用栈：栈框架

- 我们需要存储:
  - 返回地址(old ra)
  - 参数 *x*, *y*, *z* in *f(x, y, z)*
  - 临时/局部的变量 （在*f执行过程中*）
  - 被破坏的寄存器
- 想法：存在栈里!
- 增长(PUSH 进去) 栈，（每次调用函数时）
- 缩减栈(POP 出来) （每次返回时）
- 每次调用都有自己的"栈框架"

```
int fact( int n )
{
    if (n > 0)
      return fact( n-1 ) * n;
    else
      return 1 ;
}
```

例如
调用fact(2)

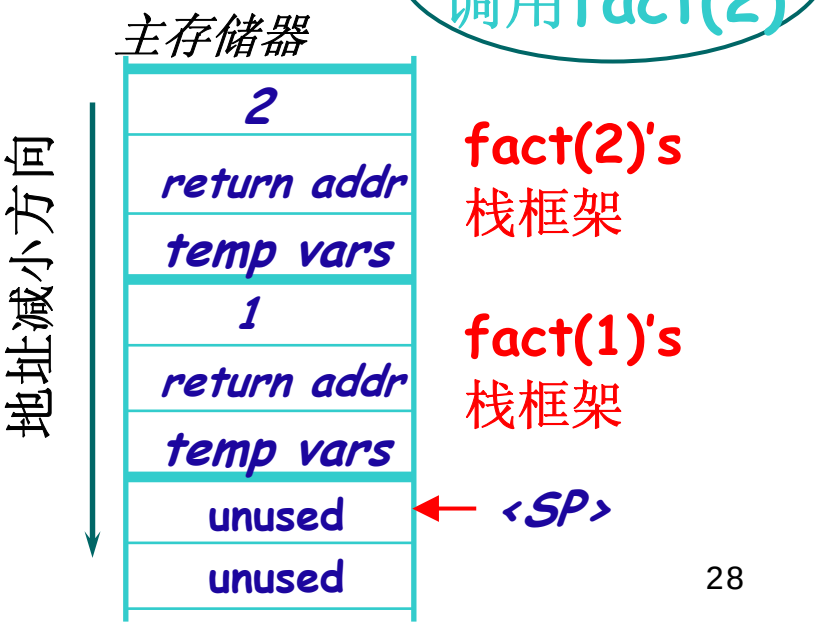


使用栈：栈框架

- 我们需要存储:
  - 返回地址(old ra)
  - 参数 *x*, *y*, *z* in *f(x, y, z)*
  - 临时/局部的变量 （在*f*执行过程中）
  - 被破坏的寄存器
- 想法：存在栈里!
- 增长(PUSH 进去) 栈，（每次调用函数时）
- 缩减栈(POP 出来) （每次返回时）
- 每次调用都有自己的"栈框架"

```
int fact( int n )
{
   if (n > 0)
     return fact( n-1 ) * n;
   else
     return 1 ;
}
```

例如
调用fact(2)

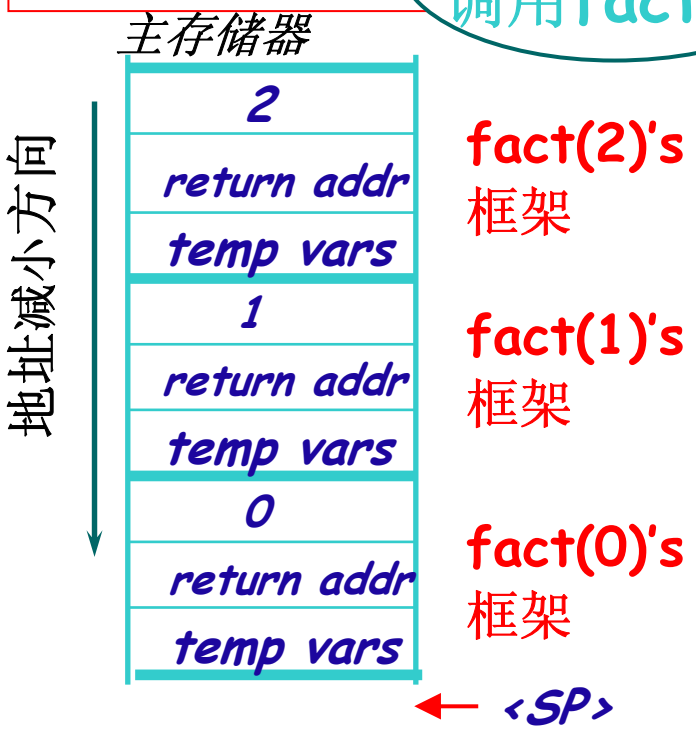


使用栈：栈框架

- 我们需要存储:
  - 返回地址(old ra)
  - 参数 x, y, z in f(x, y, z)
  - 临时/局部的变量 （在f执行过程中）
  - 被破坏的寄存器
- 想法: 存在栈里!
- 增长(PUSH 进去) 栈，（每次调用函数时）
- 缩减栈(POP 出来) （每次返回时）
- 每次调用都有自己的"栈框架"

```
int fact( int n )
{
   if (n > 0)
     return fact( n-1 ) * n;
   else
     return 1 ;
}
```

例如
调用fact(2)

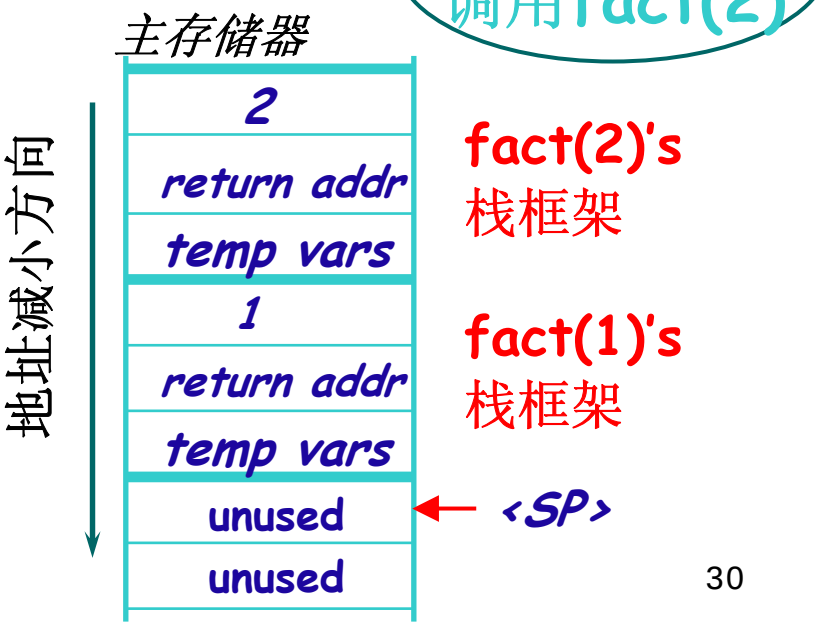


使用栈：栈框架

- 我们需要存储:
  - 返回地址(old ra)
  - 参数 *x*, *y*, *z* in *f(x, y, z)*
  - 临时/局部的变量 （在*f*执行过程中）
  - 被破坏的寄存器
- 想法：存在栈里!
- 增长(PUSH 进去) 栈，（每次调用函数时）
- 缩减栈(POP 出来) （每次返回时）
- 每次调用都有自己的"栈框架"

```
int fact( int n )
{
   if (n > 0)
     return fact( n-1 ) * n;
   else
     return 1 ;
}
```

例如
调用fact(2)

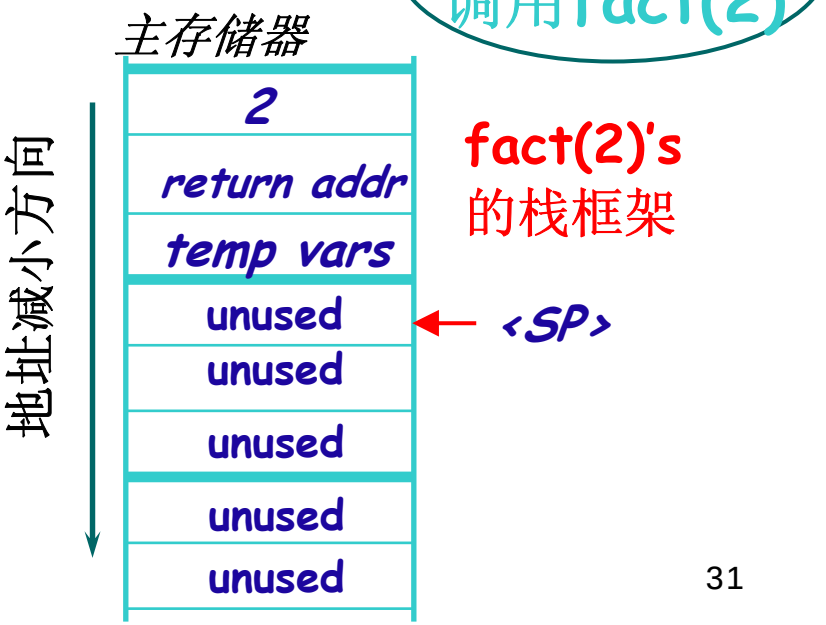


使用栈：栈框架

- 我们需要存储:
  - 返回地址(old ra)
  - 参数 *x*, *y*, *z* in *f(x, y, z)*
  - 临时/局部的变量 （在*f*执行过程中）
  - 被破坏的寄存器
- 想法: 存在栈里!
- 增长(PUSH 进去) 栈，（每次调用函数时）
- 缩减栈(POP 出来) （每次返回时）
- 每次调用都有自己的"栈框架"

```
int fact( int n )
{
   if (n > 0)
     return fact( n-1 ) * n;
   else
     return 1 ;
}
```

例如
调用fact(2)

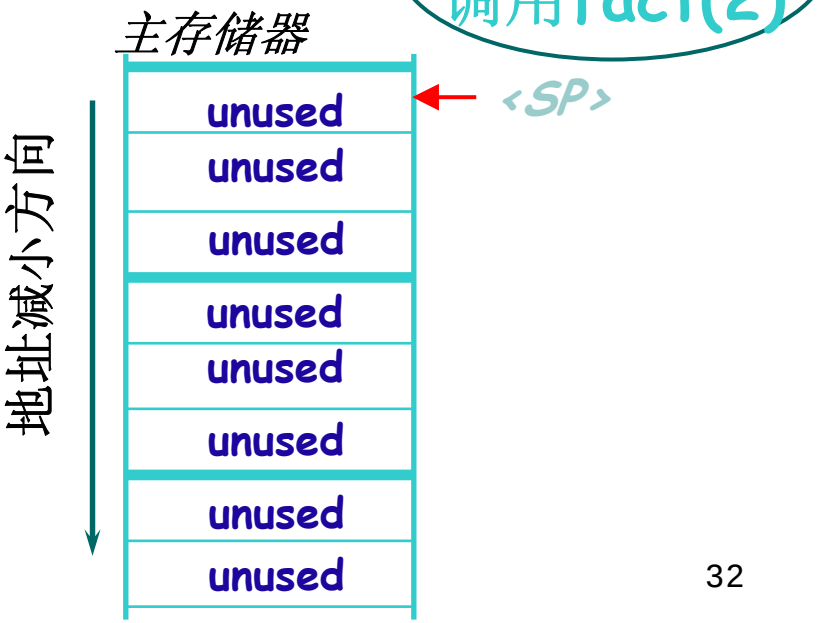


使用栈：栈框架

- 我们需要存储:
  - 返回地址(old ra)
  - 参数 *x*, y, z in f(x, y, z)
  - 临时/局部的变量 （*在f执行过程中*）
  - 被破坏的寄存器
- 想法: 存在栈里!
- 增长(PUSH 进去) 栈，（每次调用函数时）
- 缩减栈(POP 出来) （每次返回时）
- 每次调用都有自己的"栈框架"

```
int fact( int n )
{
   if (n > 0)
     return fact( n-1 ) * n;
   else
     return 1 ;
}
```

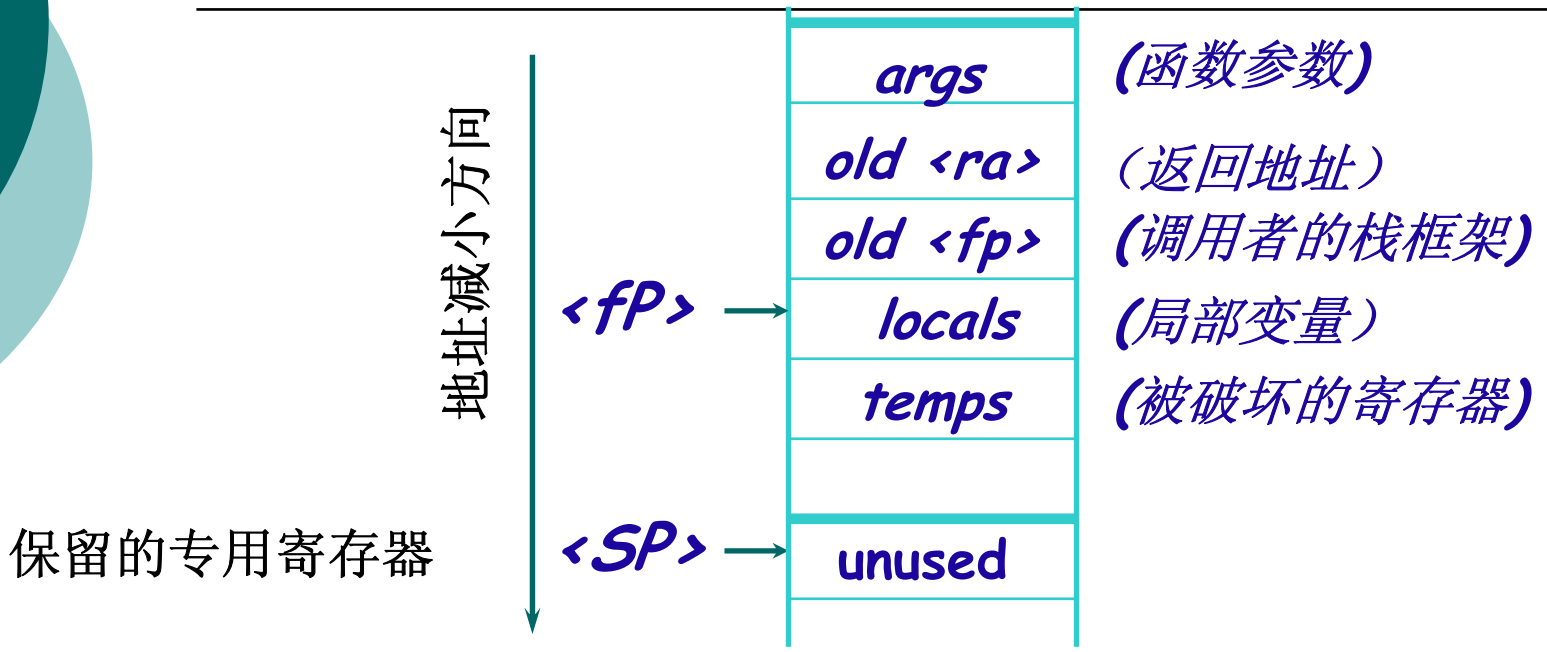
例如
调用fact(2)



栈的规定 (简介)

- 调用者：
  - 将参数按照相反的顺序放入栈中
  - 跳转到被调用函数地址，返回地址ra
  - 调整栈指针将参数移除
- 被调用者的任务：
  - 在栈内保存ra
  - 为局部变量在栈内分配存储器
    - 这样每次调用的时候都可以有不同的工作存储器
  - 计算，并将结果放在寄存器中
  - 保存并在结束的时候恢复被自己破坏的寄存器，也是在栈里
  - 每次返回前都要将栈恢复成与进来时指针一样
  - Jr $ra

框架细节



r30 = fP 框架指针：指向第一个局部变量和临时变量 (points to 1st local)

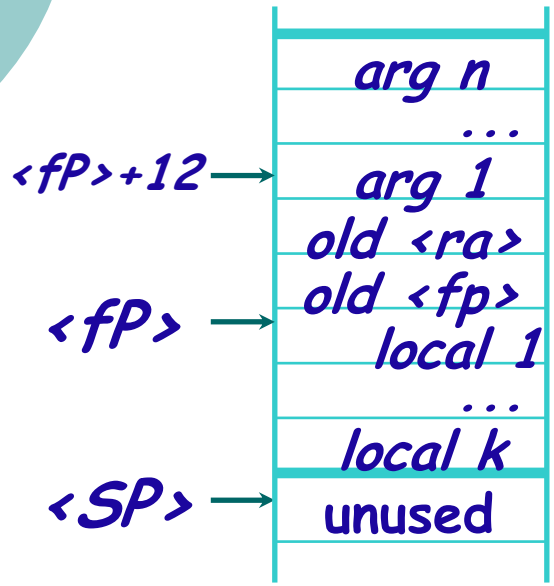
r31 = ra 返回地址：保存调用者下一条指令的地址（等于调用当时的PC）

r29 = SP 栈指针：指向第一个可用的栈内地址

r26 = XP 异常指针：当中断发生时保存PC

访问框架存储器

- fP 是访问当前框架内存储器内容的基指针
- 在一次函数调用内：
  - fP 指向变量，老的ra，和老的fP 之后
  - <fP> 指向第一个临时变量，<fP>+12 指向第一个调用时的参数
- fP本身也是要保存的！



为了访问 j^{th} 局部变量：

```
LW  $rx, _____(fP)
SW  $rx, _____(fP)
```

为了访问第 j^{th} 参数：

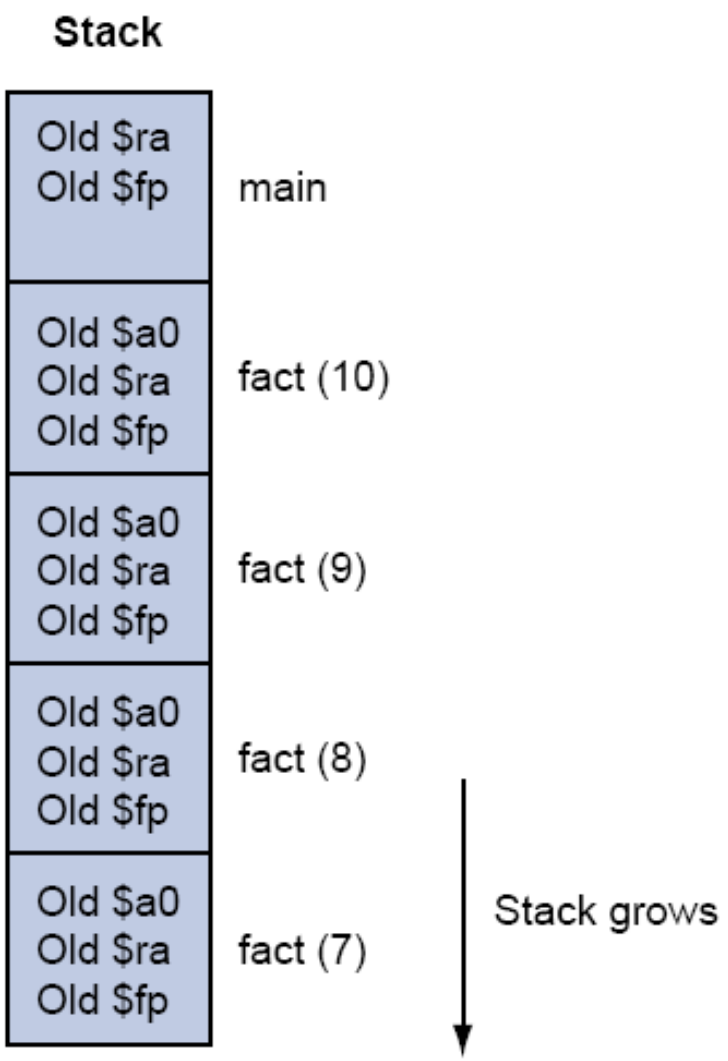
```
LW  $rx, _____(fP)
SW  $rx, _____(fP)
```

Note：调用者必须按照逆序将参数入栈，这样第一个调用参数将总是在相对于fP相同的位置（fP+12）

递归方式实现阶乘

例子：　　（书上606页）

```
int x,y;
main()
{
  x = fact( y );
}
int fact( int n )
{
   if (n != 0)
     return fact( n-1 ) * n;
   else
     return 1 ;
}
```



这个程序的汇编程序说明了过程是怎么处理栈框架的

递归方式实现阶乘（书上606页）

在过程接口处，程序main构造程序的堆栈帧：

两个old寄存器：$fp, $ra

四个参数寄存器：$a0-$a3

栈帧指针：$fp=栈帧大小-4+$sp

```
.text
.globl main
main:
subu   $sp , $sp,32
                #堆栈帧有32位字节长
sw     $ra , 20 ($ sp )
                #保存返回地址
sw     $fp, 16 ($ sp )
                #保存旧帧指针
addu   $fp, $ sp, 28
                #设置帧指针
```

递归方式实现阶乘

```
li  $a0, 10                  #将参数10送到寄存器$a0
jal  fact                    #调用递归函数
la  $ a0, $ LC               #字符串送到寄存器$a0
move $a1, $v0                #函数fact中返回的结果送到寄
                               存器$a1
jal   printf                 #调用printf函数
lw  $ra ，20（$sp）          #读出返回值
lw  $fp ，16 （$sp）         #读出帧指针
addu  $sp，$sp ，32          #弹出堆栈指针
jr   $ra                     #返回调用程序
        .rdata
$LC：.ascii “ The factorial of 10 is % d\n\000”
```

递归方式实现阶乘

```
 .text
fact:
subu   $sp , $sp,32      #堆栈帧有32个字节
sw     $ra , 20 ($ sp )  #保存返回地址
sw     $fp, 16 ($ sp )   #保存旧帧指针
addu    $fp, $ sp, 28    #设置帧指针
sw      $a0 , 0 ($ fp)   #保存参数（n）
lw      $ v0 , 0 ($ fp)  #装入a0
bgtz      $ v0, $L2      #执行判断分支 if n>0
li      $ v0, 1          #返回1
j       $L1              #跳转到代码并返回
```

递归方式实现阶乘

```
$L2:
lw      $ v1 , 0 ($ fp)      #装入n
subu    $v0 , $v1,  1        #计算n -1
move    $ a0,  $v0           #将值送到寄存器$ a0
jal     $fact                #调用过程fact
lw      $v1,  0( $ fp)       #装入n
mul     $ v0, $ v0, $ v1     #计算n*fact(n-1)

$L1:                         #结果保存到寄存器$ v0
lw      $ ra ,20 ($ sp)      #保存寄存器$ ra
lw      $fp ,16 ($ sp)       #保存寄存器$ fp
addu    $sp , $ sp ,32       #弹出堆栈
j       $ra                  #返回到调用过程
```

编程指南：（5）阅读、改进汇编代码

- 这段程序干了啥?

```
 .data
#x's address stored in $a0
# f's address stored in $a1
     .text
      move $s1,$0
      move $s2,$s1
      addiu $s3,$0,1
      lw $s4,0($a0)
loop:
      subu $s4,$s4,1
      beqz $s4,done
      move $s1,$s2
      move $s2,$s3
      addu $s3,$s2,$s1

      j loop
done:
      sw $s3,0($a1)
```

阅读汇编代码

```
.data
#x's address stored in $a0
# f's address stored in $a1
    .text
      move $s1,$0        #$s1=0
      move $s2,$s1       #$s2=$s1
      addiu $s3,$0,1     #$s3=1
      lw $s4,0($a0)      #$s4=x
loop:
      subu $s4,$s4,1     #$s4=$s4-1
      beqz $s4,done      #if($s4=0)done
      move $s1,$s2       #$s1=$s2
      move $s2,$s3       #$s2=$s3
      addu $s3,$s2,$s1   #$s3=$s2+$s1
      j loop             #goto loop
done:
      sw $s3,0($a1)      #store result in f
```

- 这段程序干了啥？

```
a = 0;  // a = r1
b = a;  // b = r2
res = 1;
i = x;
while ( --i != 0 )  // dec i  then check if 0
{  a=b;
   b=res;
   res=a+b;
}
f = res;
// note x itself isn't changed
```

x = 1 2 3 4 5 6 7 8 9 10

f = 1 1 2 3 5 8 13 21 34 55

前两项均为1，从第三项起，每一项都是其前两项的和

f = fib(x) (for x > 0) (Fibonacci)

例题

汇编下面的C语言语句

1. a=b+5*c;

2 if(b> a) c=127;

```
        .data 0x10000000
c:      .word 12345
b:      .word 2345
a:      .word 0
        .text

main:
    ori $s6,$0,0x1000
    sll $s6,$s6,16          #C's Address
    addiu $s5,$s6,4  #B's Address
    addiu $s4,$s5,4  #A's Address
    lw $s1,0($s6)    #$s1=c
    sll $s0,$s1,2    #$s0=$s1*4
    addu $s0,$s0,$s1
                     #$s0=$s0+$s1
    lw $s1,0($s5)    #$s1=b
    addu $s0,$s0,$s1
                     #$S0=$S0+$S1
done:
    sw $s0,0($s4)
                     #store result in a
```

例题

汇编下面的C语言语句

1. a=b+5*c;

2 if(b> a) c=127;

```
 .data 0x10000000
b:     .word 2345
a:     .word 0
c:     .word 0
       .text

Main:
       ori $s6,$0,0x1000
       sll $s6,$s6,16          #B's Address
       addiu $s5,$s6,4         #A's Address
       addiu $s4,$s5,4         #C's Address
      lw $s0,0($s5)     #$s0=a
      lw $s1,0($s6)     #$s1=b
      slt $s0,$s0,$s1
                        #s0=1, (s0<=s1)
      beqz $s0,done
                        #if($s0=0)done
      addiu $s0,$0,127
                       #$s0=127
      sw $s0,0($s4)   # c=$s0
done:  …
```

思考题：

当变量存储在一个绝对地址与0x10008000距离超过2^{16}的地方时，如何访问？

```
.data 0x12345678
A: .word 12
```

优化代码(请同时完成书上3.9题)

```
      .data
#x:    $a0
#f:    $a1
      .text
       move $s1,$0          #$s1=0
       move $s2,$s1         #$s2=$s1
       addiu $s3,$0,1       #$s3=1
       lw $s4,0($a0)        #$s4=x
loop:
       subu $s4,$s4,1       #$s4=$s4-1
       beqz $s4,done        #if($s4=0)done
       move $s1,$s2         #$s1=$s2
       move $s2,$s3         #$s2=$s3
       addu $s3,$s2,$s1     #$s3=$s2+$s1
       j loop               #goto loop
done:
       sw $s3,0($a1)        #store result in f
```

- 如何使本部分代码运行更快?
- 目前运行时间: *= const + 6x*
- 能否达到 *const + 5x?*

Optimizing Code

```
        move $s1,$0     #$s1=0
        move $s2,$s1    #$s2=$s1
        addiu $s3,$0,1  #$s3=1
        lw $s4,0($a0)    #$s4=x
        subu $s4,$s4,1 #$s4=$s4-1
        beqz $s4,done
                        #if($s4=0)done
loop:
        move $s1,$s2    #$s1=$s2
        move $s2,$s3    #$s2=$s3
        addu $s3,$s2,$s1
                        #$s3=$s2+$s1
        subu $s4,$s4,1 #$s4=$s4-1
        bnez $s4,loop
              #while($s4!=0)goto loop
done:
        sw $s3,0($a1)
                      #store result in f
      ...
```

- 运行时间从 ***const+6x*** 到***const+5x***
- Trick：将 **SUB** 和检查移出，减掉额外的分支指令
- 当x比较大时，减少了运行时间（constant有所增加）
- 目前的编译器一般都可以支持这类优化
- Can you do better?

有趣的话题:单指令计算机

- 深入理解计算机的工作
- 例子:SIC single instruction computer

  “减并且结果如果为负数就跳转”

  - **Sbn a, b, c**

    # mem[a] = mem[a]-mem[b];

    # if mem[a] < 0 goto c;

    # if mem[a] >=0 从紧接着本指令的下一存储器地址取出下一条指令

  - 三个操作数每个都指向存储器中一个字的地址

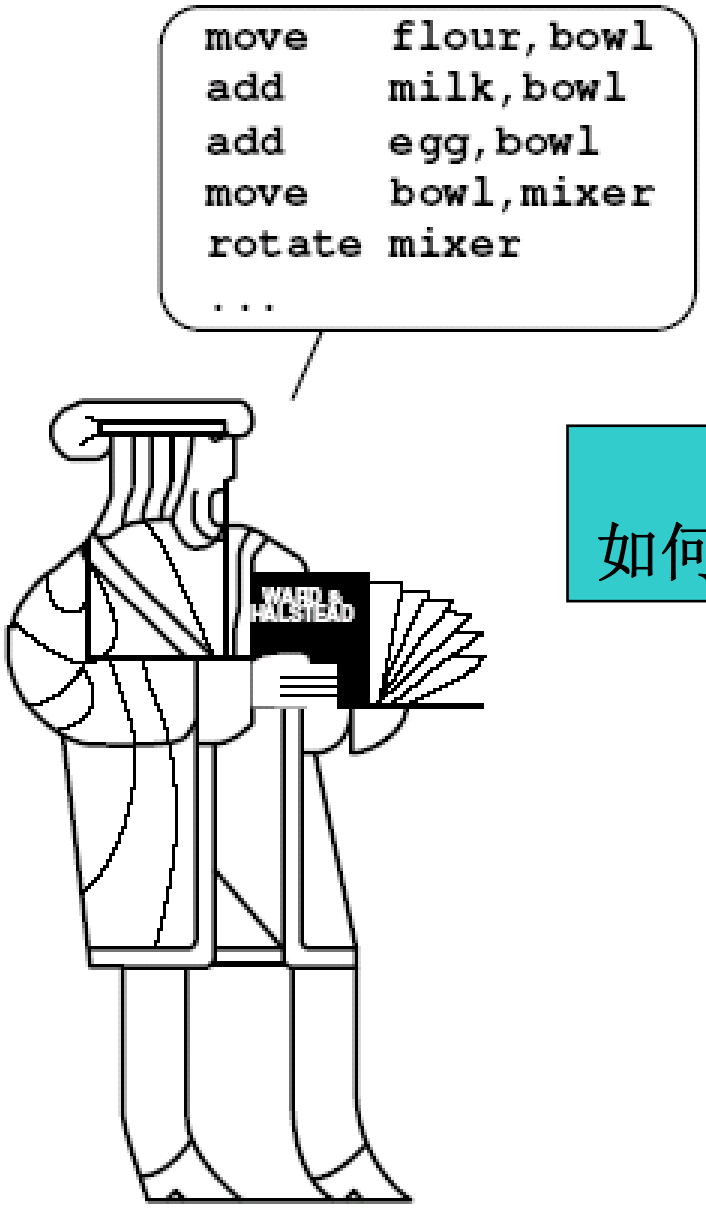
有趣的话题:单指令计算机

- SIC 虽然只有一条指令，通过巧妙设计sbn指令的次序，SIC能模仿很多更复杂指令集的操作。
- 例：将存储器地址**a**的数复制到存储器**b**

```
Start: sbn temp, temp,.+1   #将temp清零
       sbn temp, a, .+1     #将temp设为-a
       sbn b, b, .+1        #将b清零
       sbn b, temp, .+1     #将b设为-temp，即a
```

这条指令之后的地址

Q：我们现在有了什么？可以做什么了？



1 可编程的控制
如何实现一个计算机以达到UTM的作用

2 指令集与不同级别的抽象

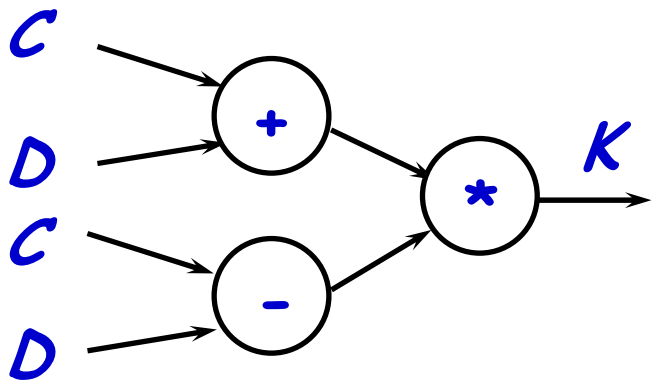
3 MIPS的Programming Model

Motivation: 计算一个表达式

Given an expression:

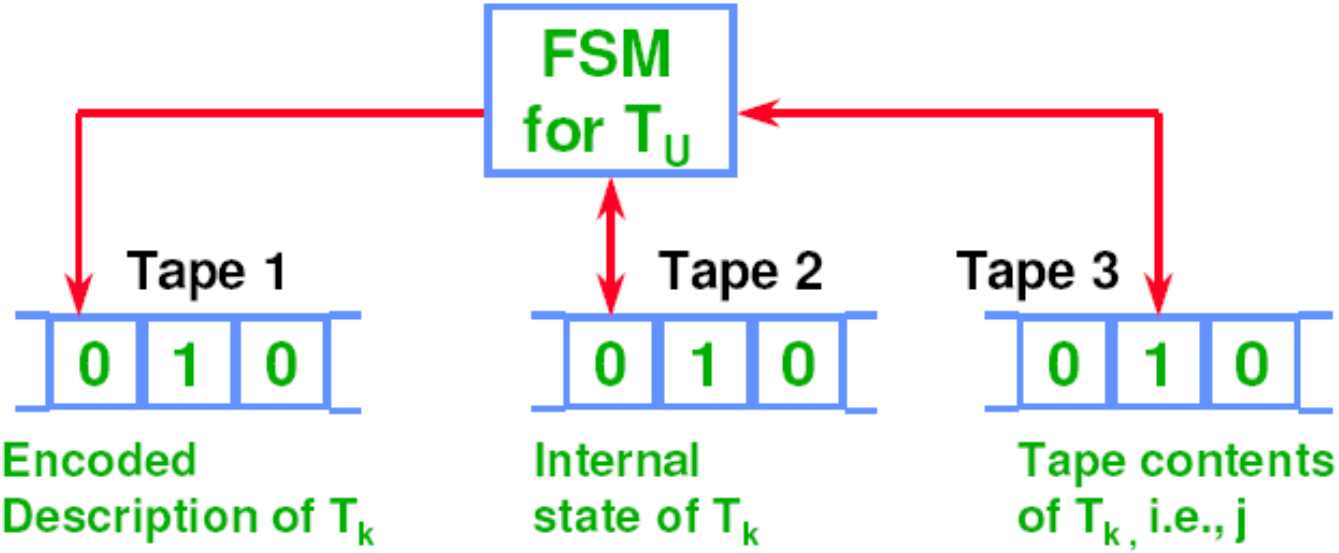
$$K = (C + D) * (C - D)$$

我们可以设计这么一个电路(或者说是图灵机)



电路的规模直接与计算表达式的规模有关

回忆一下: 我们已经有了UTM的思想



1：T_u查看Tape2和Tape3来决定图灵机的配置T_k（采用第k个图灵机)及它的输入j;

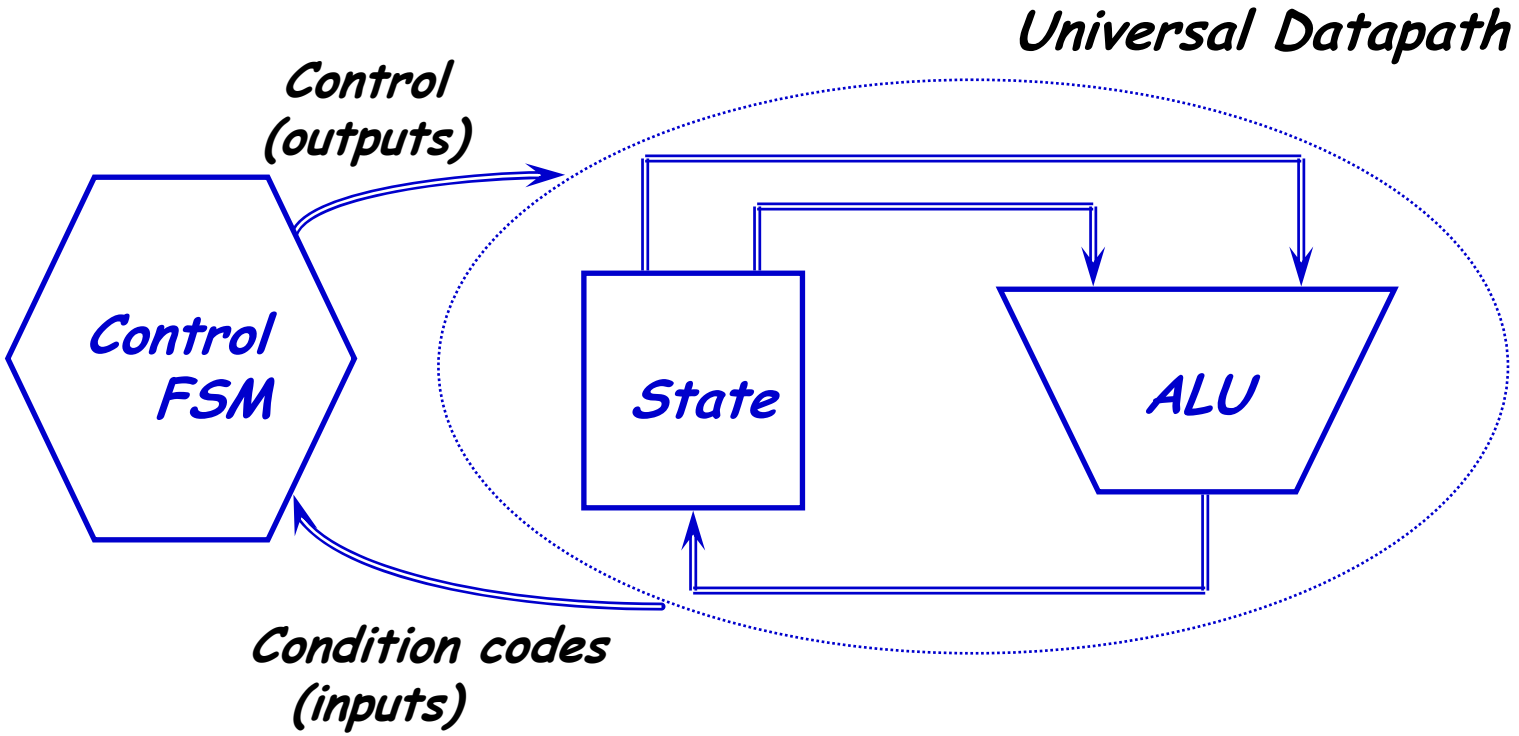
2：T_u查看第一个磁带Tape1决定T_k会怎么做;

3：修改磁带2和3以反映T_k(j)在计算过程中的动作和内部状态变化;

我们的任务是计算所有的可计算的整数函数

控制与数据通路的方法

用一系列的命令给一个简单的单元(名叫**datapath**)以实现表达式计算



这样数据通路的规模就不随着问题规模增大而增大了

Control FSM for K = (C + D) * D

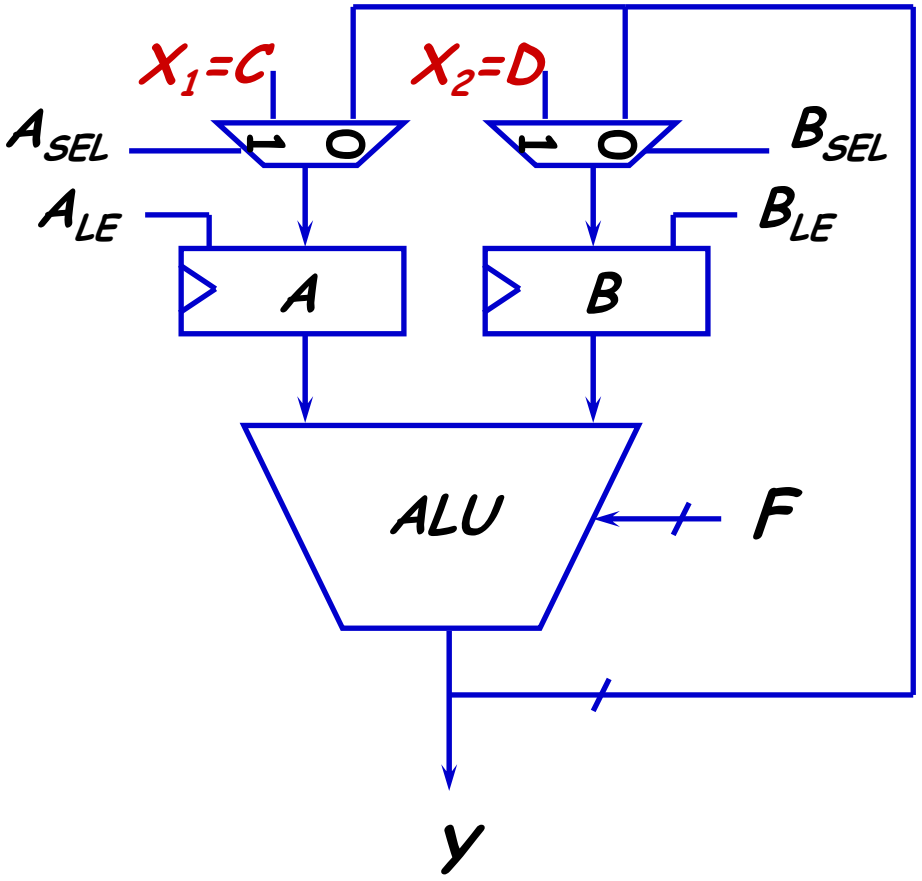
假定输入 X_1 , X_2 只在第一个时钟周期出现

Control FSM

current state

outputs

	A_{SEL}	B_{SEL}	A_{LE}	B_{LE}	F
s_1					
s_2					
s_3					



Note: 这个**FSM**只是一些状态转移关系，没有输入

Control FSM for K = (C + D) * D

Step 1：通过给 A_{LE} and B_{LE}置位，把C和D加载到寄存器A和B中

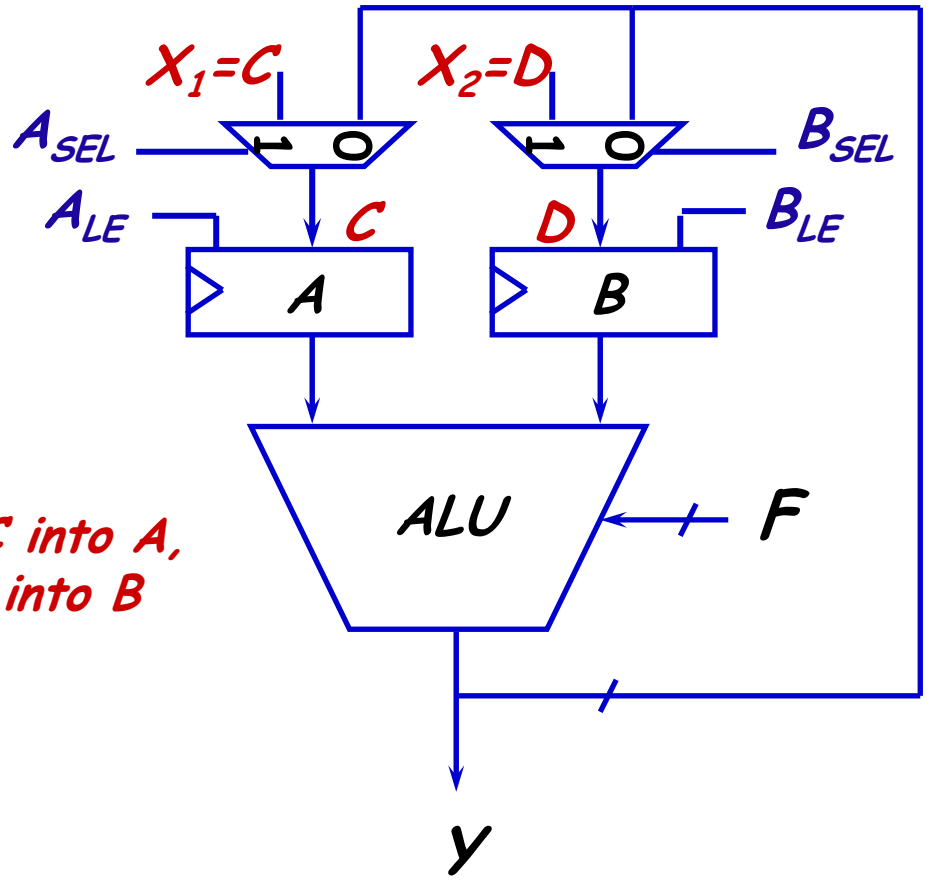
Control FSM

current state

outputs

	A_{SEL}	B_{SEL}	A_{LE}	B_{LE}	F
s_1	1	1	1	1	–
s_2					
s_3					

Load C into A, and D into B



Note：这个FSM只是一些状态转移关系，没有输入

Control FSM for K = (C + D) * D

Step 2: 把A+B计算完毕并把结果存在A里面

Control FSM

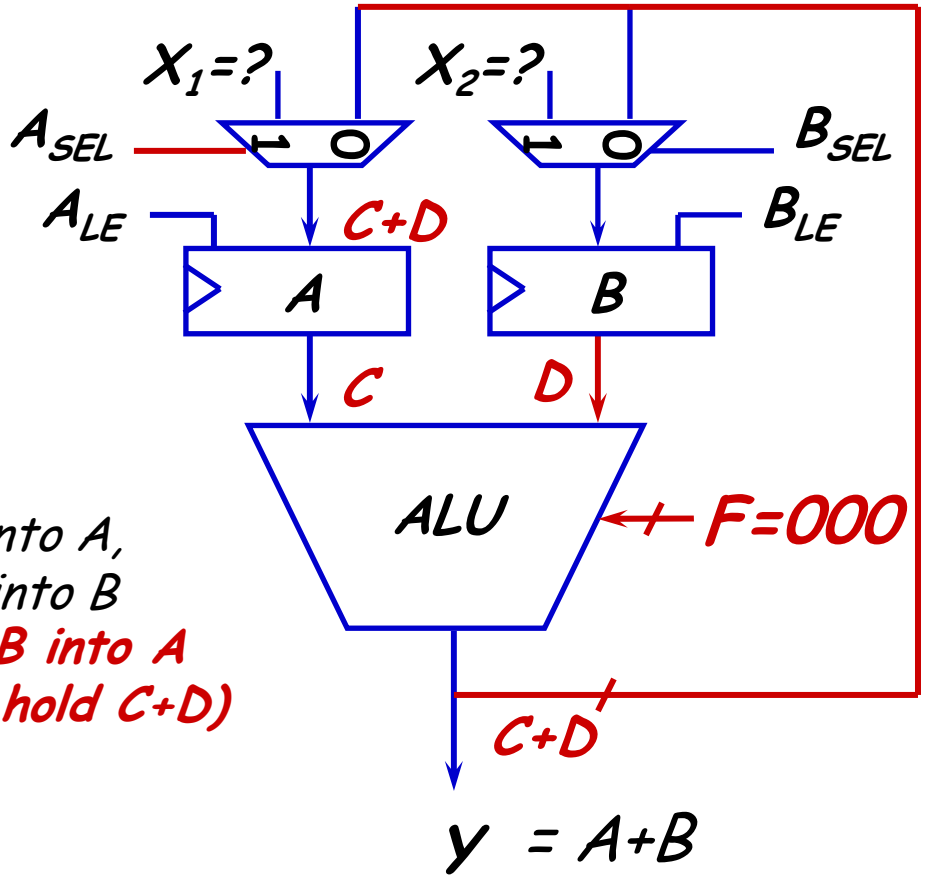
current state

outputs

	A_{SEL}	B_{SEL}	A_{LE}	B_{LE}	F
s_1	1	1	1	1	–
s_2	0	-	1	0	A+B
s_3					

Put C into A, and D into B

Put A+B into A (A will hold C+D)



Note: 这个FSM只是一些状态转移关系，没有输入

Control FSM for K = (C + D) * D

Step 3: 将A*B的结果保存在A中

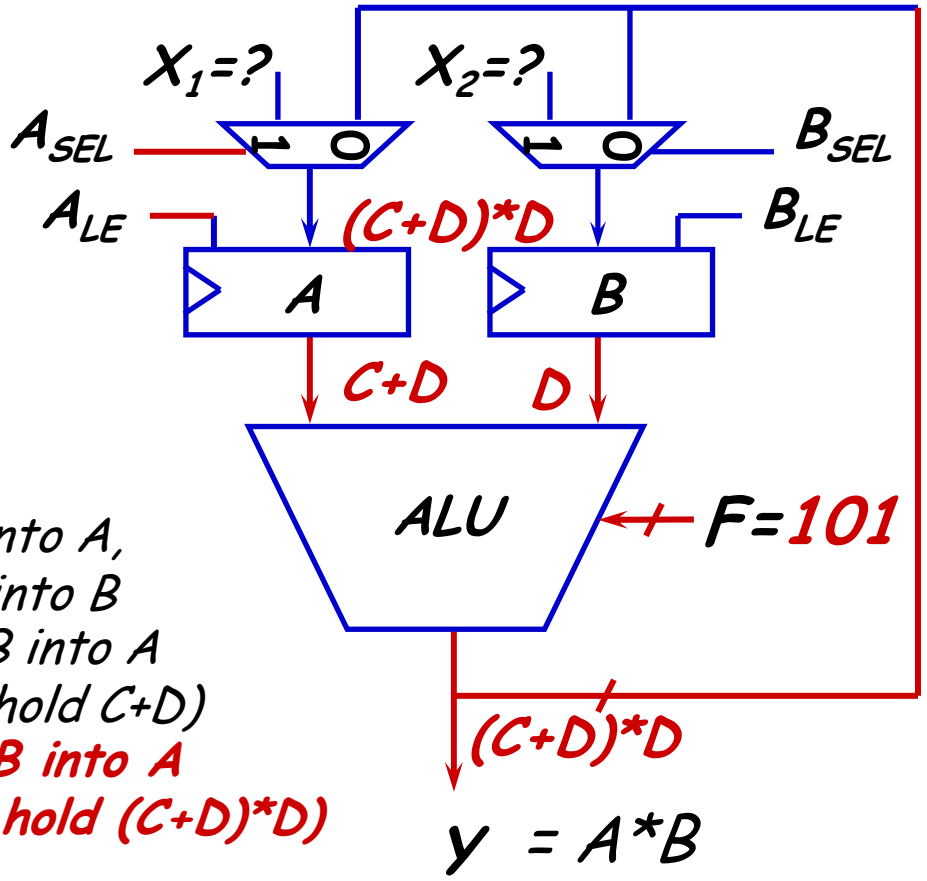
Control FSM

current state

outputs

	A_{SEL}	B_{SEL}	A_{LE}	B_{LE}	F
s_1	1	1	1	1	–
s_2	0	-	1	0	A+B
s_3	0	-	1	0	A*B

Put C into A, and D into B
Put A+B into A (A will hold C+D)
Put A*B into A (A will hold (C+D)*D)



Note: 这个FSM只是一些状态转移关系，没有输入

Control FSM for K = (C + D) * D

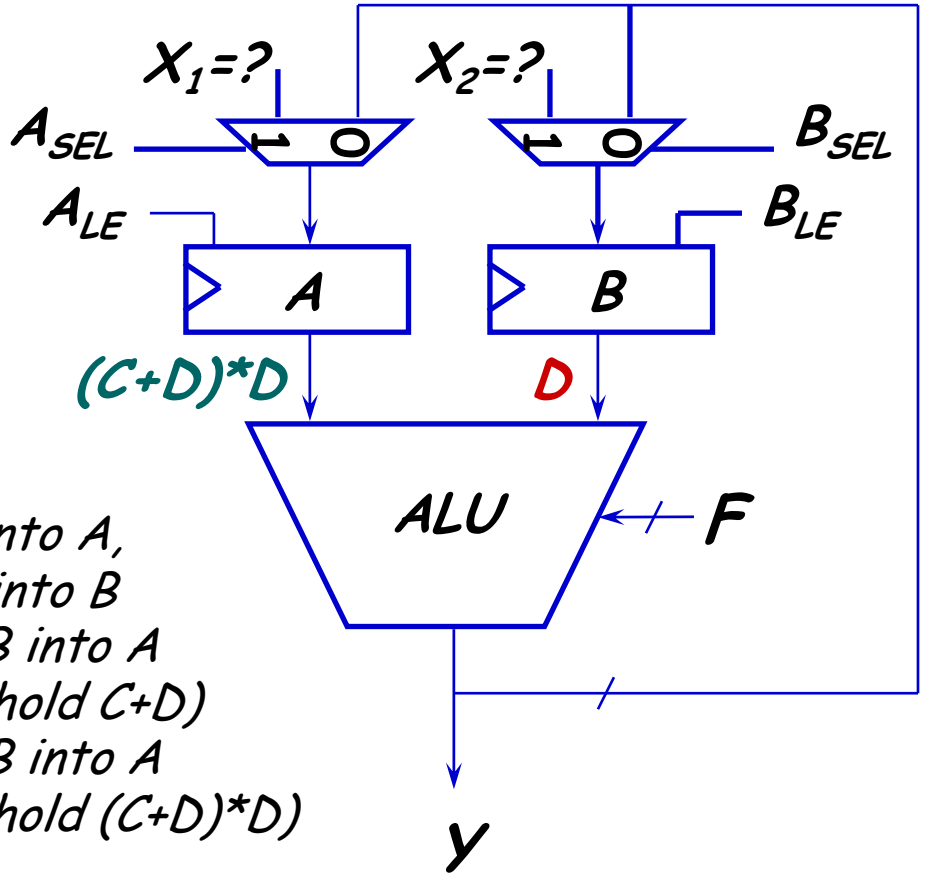
*第3步的时钟上升沿过后：A里面存放的是 (C+D)*D*

Control FSM

current state

outputs

	A_{SEL}	B_{SEL}	A_{LE}	B_{LE}	F	
s_1	1	1	1	1	–	Put C into A, and D into B
s_2	0	-	1	0	A+B	Put A+B into A (A will hold C+D)
s_3	0	-	1	0	A*B	Put A*B into A (A will hold (C+D)*D)



Note: 这个**FSM**只是一些状态转移关系，没有输入

写哪个会比较好一些?

$K = (C + D) * (C - D)$

	RF_{SEL}	RF_{LE}	ADR	A_{LE}	B_{LE}	F	
s_1	1	1	000	0	0	–	$R_0 \leftarrow C$
s_2	1	1	001	0	0	–	$R_1 \leftarrow D$
s_3	0	0	000	1	0	–	$A \leftarrow \langle R_0 \rangle$
s_4	0	0	001	0	1	–	$B \leftarrow \langle R_1 \rangle$
s_5	0	1	000	0	0	A+B	$R_0 \leftarrow \langle A \rangle + \langle B \rangle$
s_6	0	1	001	0	0	A-B	$R_1 \leftarrow \langle A \rangle - \langle B \rangle$
s_7	0	0	000	1	0	–	$A \leftarrow \langle R_0 \rangle$
s_8	0	0	001	0	1	–	$B \leftarrow \langle R_1 \rangle$
s_9	0	1	000	0	0	A*B	$R_0 \leftarrow \langle A \rangle * \langle B \rangle$

等等，看上去下面的东西是软件?

RTL or 机器语言需要些什么?

- 需要关于机器的抽象描述
  - 有哪些寄存器: R_0, R_1, …
  - 可以进行哪些运算: +, *, or …
  - 机器指令的编码序列:
    $R_1 \leftarrow <R_2> + <R_3>$
- 需要对指令是如何改变机器状态的有深入的了解
  - note: 有些寄存器的转移动作是无法在一个周期内完成的!
- 需要定义一些指令以影响数据通路的流程(条件码)

We will see a lot of this in coming weeks

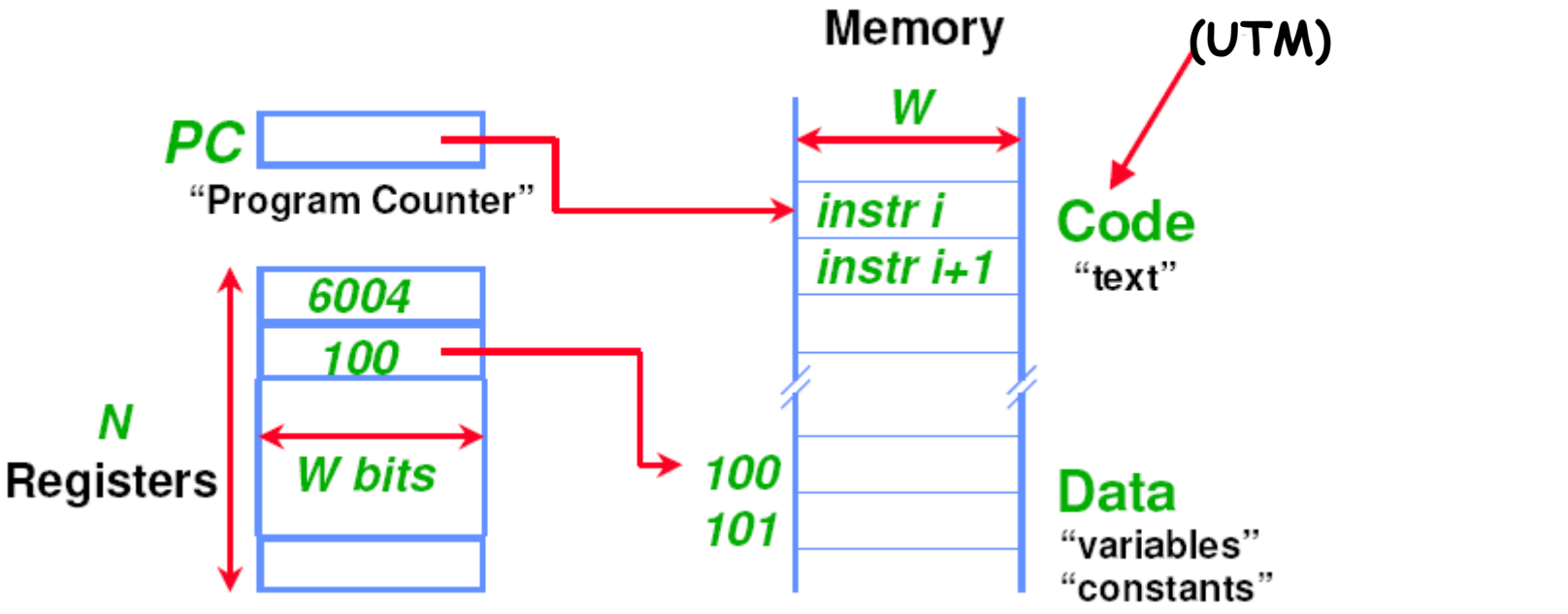
TTR1: 机器语言与指令集体系ISA

两个重要的组成部分: (指令不仅仅是操作而已!)

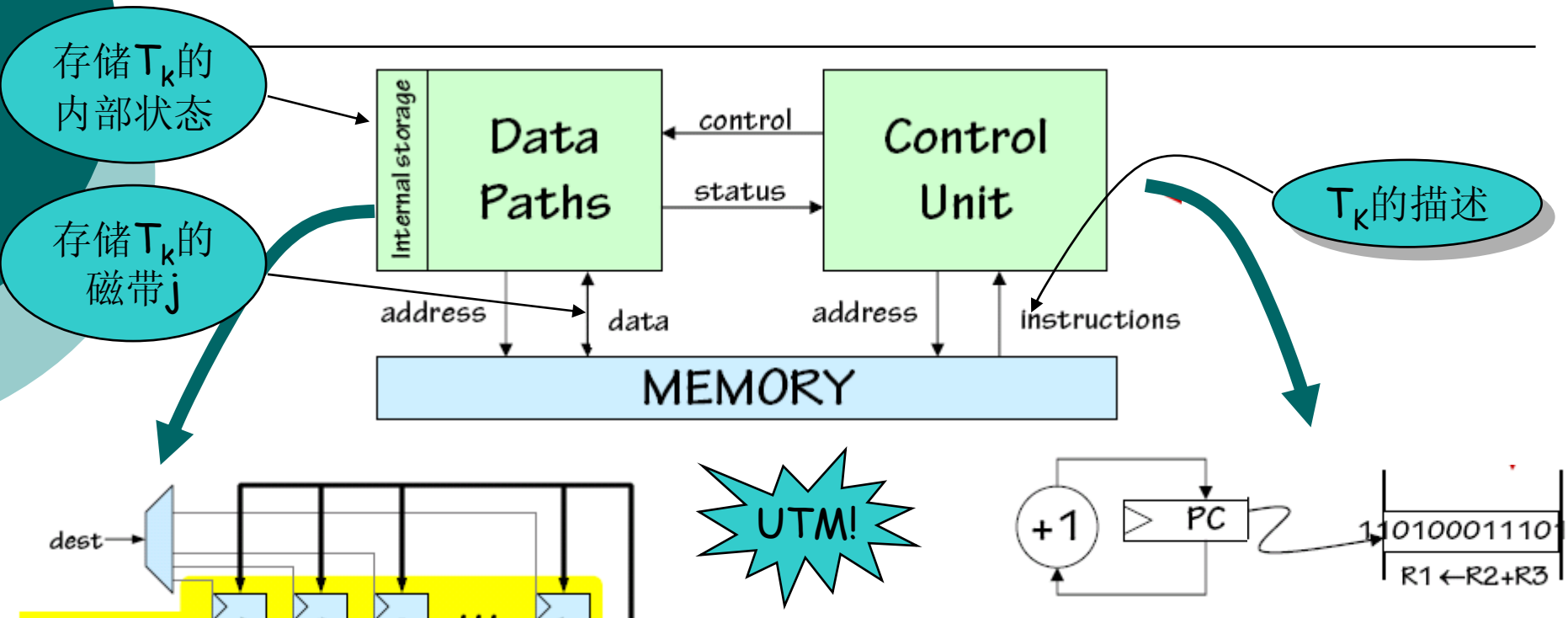
(1) 可见的状态(寄存器,存储器等);

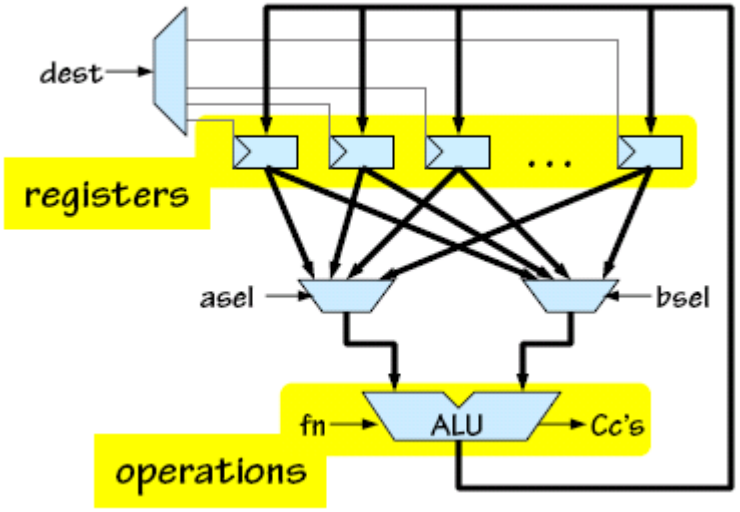
(2) 这些状态的操作(instructions)

存储程序的计算机 (UTM)



深入内部看看Von Neumann模型





- ✓ 指令被编码成为二进制的"数据"
- ✓ 指令计数器PC:下一条指令的地址
- ✓ 转换指令为数据通路的控制信号的控制逻辑