### 1.最小系统

图1就是包含了CPU与内存的一个简单演示系统。其中CPU中有四个8位的寄存器（其名字分别为R0，R1，R2，R3），剩余部分用“其他部分”来表示，内存地址宽度为8位，编码从0000~1111，即共有16个字节的内存。内存与CPU之间存在一个数据传送的通道，术语称为总线，同样，在CPU内部也存在传输数据的内部总线。为了简单起见，这些总线并没有标出。



图1 最小系统

假设我们要在最小系统上执行的代码如下：

|  |
| --- |
| ……  int i=1;//假设整数只占一个字节  int j=2;  int k;  k=i+j; |

首先来看第一行代码，int i；

其在c语言中的意义是定义了一个整形变量i，并赋值1，但在执行层面上来看，当计算机在执行这行代码时，所做的操作是在内存中分配一个地址，用于存储这个整数，在以后对这个变量i进行操作时，就是对这个内存地址中的值进行操作，例如在赋值1的时候，就是将内存地址0000中的数据修改为1。这时候最小系统的状态如图2所示。



图2 执行int i=1;语句后的最小系统状态

从这个执行过程可以看出，定义变量实际上涉及内存分配这个操作，如果内存已经使用完，那么这个操作就无法完成，进而导致程序运行时出错。因此，在定义占用空间特别大的数据类型时，需要特别小心，例如定义一个大型的整数数组 int arr[1000000]，在语法上是完全合法的，因此可以顺利编译通过，但在运行时则有可能因为无法分配内存而导致出错。

*试一试：在你的计算机上尝试调整数组大小，看看多大时运行会出错？*

最小系统继续执行int j＝2;和int k;这两个变量定义语句，执行后最小系统状态如图3所示。



图3 变量i,j,k定义并赋值执行完成后的最小系统状态

在这个执行过程中，有两点需要注意的：

（1）为了简单起见，我们假设三个变量在内存中是连续的，但这并不是强制规定。换而言之，在代码执行时，只需要保证每个变量有一个对应的内存地址即可。

（2）变量k没有赋初值，缺省为0。这是因为内存是易失性存储器，在断电重启后所有数据都会清零，有些编译器也会对于未赋初值的整数变量赋缺省值0，但由于变量k所分配的内存空间有可能是回收的空间，因此不赋初值的话，有可能是其他值，因此最好都是赋初值0。

对于代码k=i+j而言，由于其涉及计算，因此CPU需要参与其中。因此其执行步骤分为四步：

1. 第一步，将内存地址0000中的值（也就是i）送往CPU中的某个寄存器中，假设为R0，此时最小系统的状态如图4所示。



图4 代码k＝i+j执行的第一步

1. 第二步，将内存地址0001中的值（也就是j）送往CPU中的另一个寄存器中，假设为R1，此时最小系统的状态如图5所示；



图5 代码k＝i+j执行的第二步

1. 第三步，CPU中的ALU执行加法指令，将两个寄存器中的值相加，结果放在第二个寄存器中，即R1中存储了两个数的和，此时最小系统的状态如图6所示；



图6 代码k＝i+j执行的第三步

1. 第四步，将R1中的值传送至内存地址0010中，代码执行完成，最小系统的状态如图7所示。



图7 代码k＝i+j执行的第四步

下面举一个稍微复杂点的例子，代码如下所示：

|  |
| --- |
| ……  int arr[2]={1,2};  int k=5;  arr[0]=arr[1]+k; |

第一行代码是定义了一个有两个元素的整数数组，其名称为arr，在分配内存时，会分配两个整数空间，由于其是在一个数组内，所以其地址肯定是连续的，这时候只需要记录arr的地址编号；假设arr分配的内存地址是0010，则0010中保存了arr[0]的值，即1，0011中保存了arr[1]的值，即2；

第二行代码是定义了一个整数变量，过程与上面相同，假设为k分析的地址是0000，则在分配好后，最小系统的状态如图8所示；



图8 变量分配内存后的最小系统状态图

下面开始执行第三行代码，这行代码的功能是arr[1]与k相加，但对于第一个操作数arr[1]，由于在c语言中，[]实际上是一个地址计算操作，即将arr的值（即数组首地址），加上[]里面的数字，形成新的地址，再去取这个地址的值，所以第三行代码的执行分为三个阶段，第一个阶段是计算arr[1]在内存中的地址，获得arr[1]的值（第一步到第四步）；第二个阶段是将arr[0]与k相加（第五步到第六步）；第三个阶段计算arr[0]在内存中的地址，然后将第二阶段获得的结果保存在这个地址中（第七步到第十步），具体的描述如下：

（1）第一步，将arr的值传送到某个寄存器，假设为R0；

（2）第二步，将[]中的数值1传送到另一个寄存器中，所设为R1；

（3）第三步，CPU中的ALU执行加法指令，将两个寄存器的值相加，结果放在R1寄存器中；

（4）第四步，将R1寄存器的值与读内存指令传送到内存管理单元，将此值的低四位作为内存地址，读取此地址中的数值，并传送到寄存器R2中；

第一阶段完成后，最小系统的状态如图9所示。



图9 第一阶段完成后的最小系统状态图

（5）第五步，将内存地址0000（即变量k）的值传送到寄存器R1中（R1寄存器中之前保存的是arr[1]的地址，已经使用完毕，所以可以覆盖）；

（6）第六步，CPU中的ALU执行加法指令，将R1与R2两个寄存器的值相加（即计算arr[1]+k的结果），结果放在R2寄存器中；此时最小系统的状态如图10所示；



图10 第二阶段完成后的最小系统状态图

（7）第七步，将arr的值传送到寄存器R0中（由于R0中保存的就是arr的值，如果系统够“聪明”的话，可以进行优化，省略这一步）；

（8）第八步，将[]中的数值0传送到寄存器R1中；

（9）第九步，CPU中的ALU执行加法运算，得到结果，保存在R0中（由于是加0操作，如果系统够“聪明”的话，可以进行优化，直接将第七、八、九步简化成一步：直接将arr值传送到寄存器R0中）；

（10）第十步，将R2（写入内存的内容）与R0（写入内存的地址）的值与写内存命令发送给内存管理单元，在R0的低四位地址（0010，即arr[0]）中写入R2的值，至此代码执行完成。这时候最小系统的状态如图11所示。



图11 第三阶段完成后的最小系统状态图

## 2.原型系统

本节将在最小系统的基础上进行扩展，打造一个符合冯·诺伊曼体系结构的原型系统。冯.诺伊曼体系结构是现代计算机的基础，现在大多计算机仍是冯·诺伊曼计算机的组织结构，在这种体系中，计算机硬件由运算器、控制器、存储器、输入设备和输出设备五大部分组成。

原型系统的结构如图12所示。



图12 原型系统

在原型系统中：

1. 存储器为内存；
2. 运算器（ALU）在CPU内部，与最小系统的功能相同；
3. 输入设备为一个数字小键盘。为简单起见，假设输入的数字都是三位（不足三位的前面补0，例如001表示1，019表示19），并且不会超过127（因为本系统中整数只有一个字节）；
4. 共有四个8位通用寄存器R0,R1,R2,R3，一个标志位寄存器G，一个程序计数器PC，其中程序无法直接给寄存器G和PC赋值，所以在图12中没有画出；
5. 输出设备为一个四位数码显示管；

控制器是整个原型系统的核心部件，主要功能是执行系统的指令，本原型系统设计的指令集中只包括12条指令，格式如下：

|  |  |  |
| --- | --- | --- |
| 指令格式 | 例子 | 说明 |
| HALT | HALT | 停机指令，原型机停止运行 |
| MOVA R1，R2 | MOVA R1,R2 | 寄存器间值传输指令。将寄存器R2中的值传输（或拷贝）到R1中 |
| MOVB M，R2 | MOVB R1,R2 | 寄存器值传输到内存指令，将寄存器R2中的值传输到内存中，内存的地址由R1寄存器中的值指定 |
| MOVC R1,M | MOVC R1,R2 | 内存值传输到寄存器指令。将内存中的值传输到寄存器R1，其中内存的地址由R2寄存器中的值指定。将 |
| MOVD R3,PC | MOVD R3,PC | 保存程序计数寄存器值指令，将程序计算寄存器的值保存到某一个通用寄存器中。 |
| MOVI IMM | MOVI 5 | 将一个常数值直接赋值给通用寄存器R0，这个常数值的范围是-128~127 |
| ADD R1,R2 | ADD R1,R2 | 加法指令，寄存器R1与R2中的值相加，结果保存在R1中。在原型机中，暂不考虑溢出等情况。 |
| SUB R1，R2 | SUB R1,R2 | 减法指令，寄存器R1中的值减去寄存器R2中的值，结果保存在R1中。此操作会影响标志寄存器G：如果R1>R2，则G的值为1，否则G的值为0 |
| JMP | JMP | 直接跳转指令。将R3的值赋值给程序计数寄存器PC |
| JG | JG | 大于则跳转指令。如果G的值为1，则将R3的值赋值给PC |
| IN R1 | IN R1 | 输入一个数字，并保存在某个通用寄存器中 |
| OUT R2 | OUT R2 | 将某个通用寄存器中的值直接输出 |

在机器启动时，可以对内存空间进行划分，例如16个字节的内存中，规定前面3个字节（0000~0010）为数据段，只存储数据，最后一个字节为显存，用于保存显示输出的数据，中间的12个字节用于存放代码，控制器的代码执行初始值可定义为0011，即第一条指令存放在0011处，然后顺序执行或跳转执行，直到遇到停机指令。其示意图如图13所示。



图13 原型系统

下面以两个简单的c语言程序来例，来说明原型系统的工作原理。

第一个程序的功能是输入一个大于1的数字a，计算1+2+……+a的值并显示出来。代码如下所示。

|  |
| --- |
| #include <stdio.h>  int main()  {  int a;  int sum=0;  scanf(“%d”,&a);  for(int i=a;i>=1;i--)  sum+=i;  printf(“%d”,sum);  return 0;  } |

很显然，我们的原型系统是无法运行这段c程序代码的，因此需要将这个程序转换成原型系统能够执行的指令，这个转换过程也称为“编译”。此程序编译完成后的指令如下：

|  |
| --- |
| in R1 #输入a到R1  movi 1 #设置R0为1  add R2,R1 #R2存放累加值  sub R1,R0 #R1的值即a减去1,此时会设置Ｇ值  movd #将当前PC值保存在R3中，此时R3+0为本行代码地址，R3-1为上一行代码地址，R3+1为下一行代码地址……  movi -3 #存放-3到R0中,跳转到第三行：add R2,R1处  add R3,R0 #R3减去3，注意此时不能用SUB指令，会影响G值  jg #如果R1的值还大于1，则跳到第三行代码去执行，即继续进行累加  out R2 #如果R1的值此时小于等于1，则输出累加和  halt #停机 |

编译完成后，可以将此代码装载进内存中，此时原型系统的状态如图14所示。



图14 执行第一个程序的原型系统

本书编写了一个程序来模拟原型系统，在随书代码中的02目录中，运行 vspm即可模拟原型系统的运行（同目录下有示例视频）。

第二个程序的功能是输入两个数，保存这两个数，并输出其中的最小值。C程序代码如下所示。

|  |
| --- |
| #include <stdio.h>  int main()  {  int a,b;  int min=0,c;  scanf(“%d,%d”,&a,&b);  c=a-b;  if(c>0) min=b;  else min=a;  printf(“%d”,min);  return 0;  } |

编译完成后的指令如下：

|  |
| --- |
| in R1 #输入第一个数a  in R2 #输入第二个数b  mova R0,R1 #在R0保存a  sub R1,R2 #a-b,此时会设置G  mova R1,R0 #a保存到R1  movd #保存当前的PC值到R3  movi 6 #R0的值设置为6,即跳转到12行:out R0  add R3,R0 #R3的值加6  mova R0,R2 #b的值保存到R0  jg #如果a的值比b大,就直接输出R0中的值b，否则需要将a的值保存在R0中再输出，这样就可以保证R0中的值为最小值  #跳转地址  mova R0,R1 #将a的值保存到R0  out R0 #输出R0  halt |

编译完成后，可以将此代码装载进内存中，此时原型系统的状态如图15所示。



图15 执行第二个程序的原型系统