0　背景知识：最小系统与原型机

0.1 最小系统

图 0.1 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 4 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。

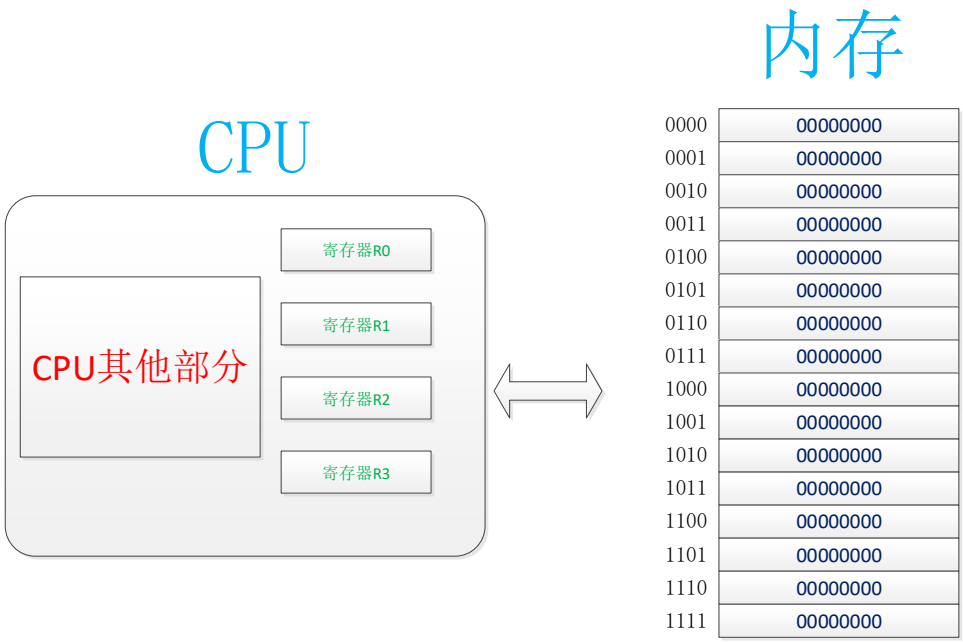

图 0.1 最小系统

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 0.2 所示。

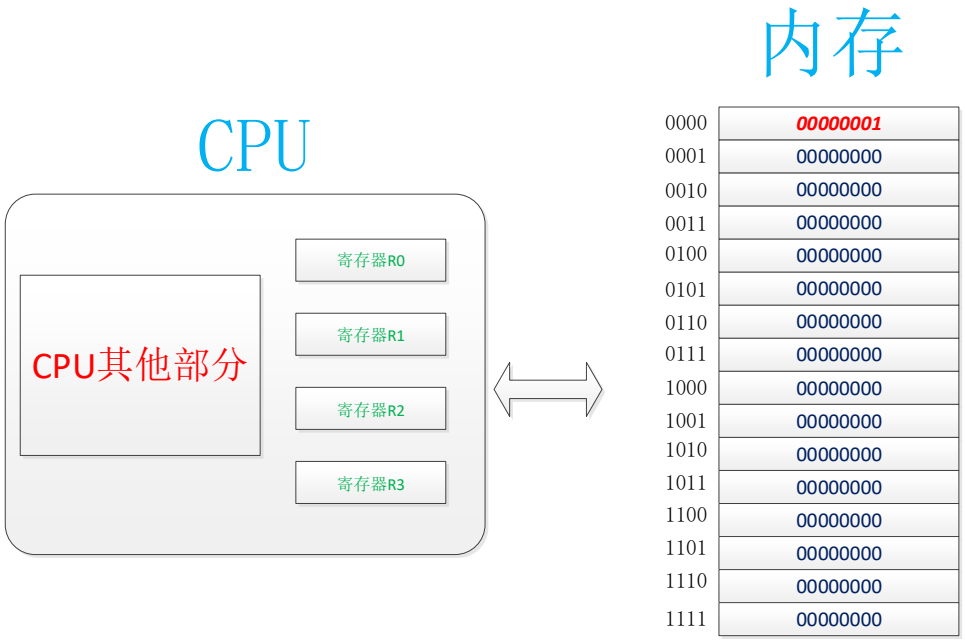

图 0.2 执行 int i=1;语句后的最小系统状态

从这个执行过程可以看出，定义变量实际上涉及到内存分配这个操作，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 0.3 所示。

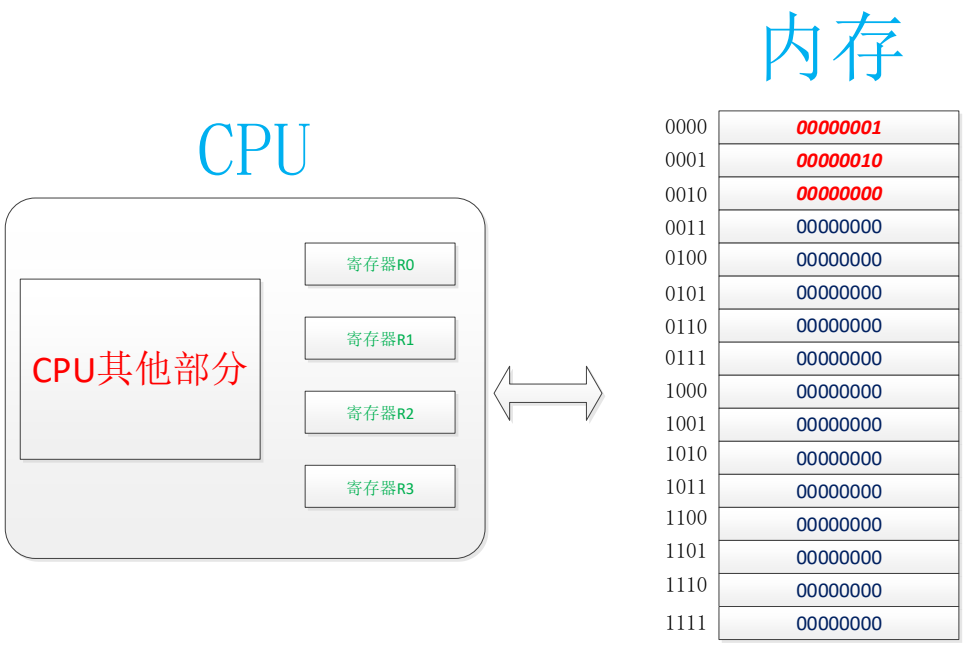

图 0.3 变量 i,j,k 定义并赋值执行完成后的最小系统状态

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，但这不并是强制规定。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，由于其涉及到计算，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 0.4 所示。

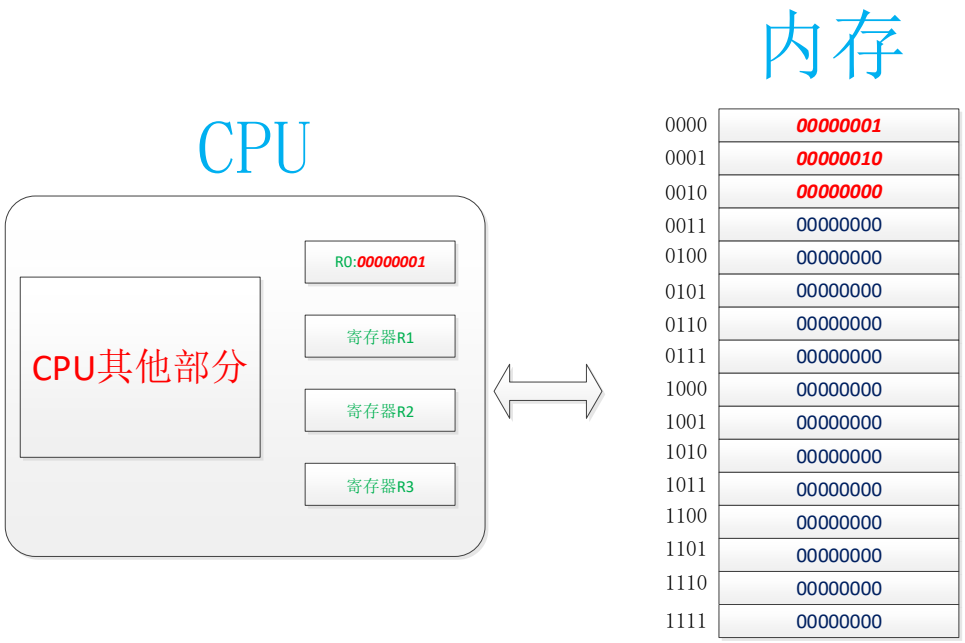

图 0.4 代码 k = i+j 执行的第一步

(2) 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 1.5 所示；

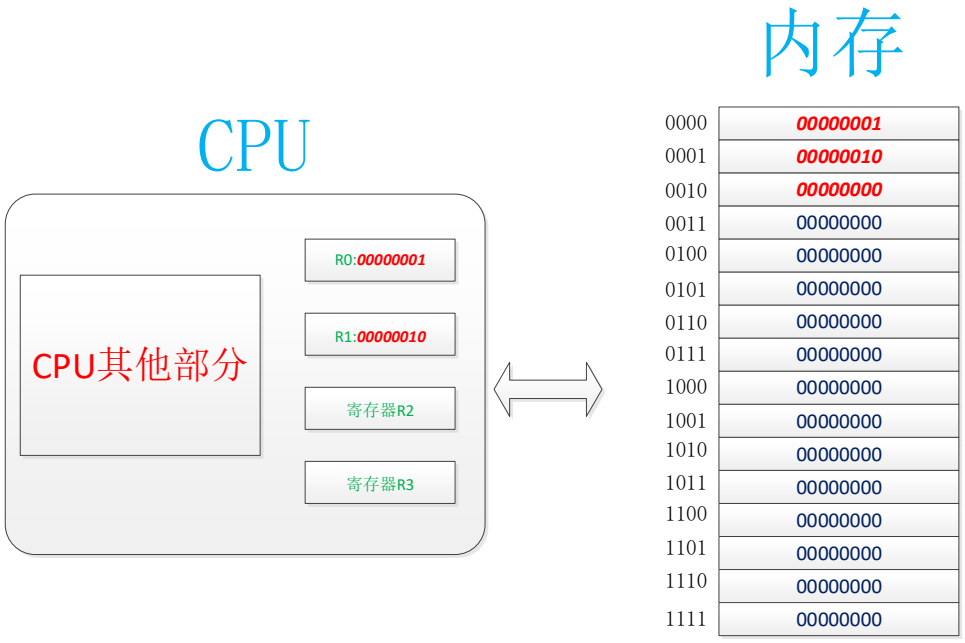

图 0.5 代码 k = i+j 执行的第二步

(3) 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 0.6 所示；

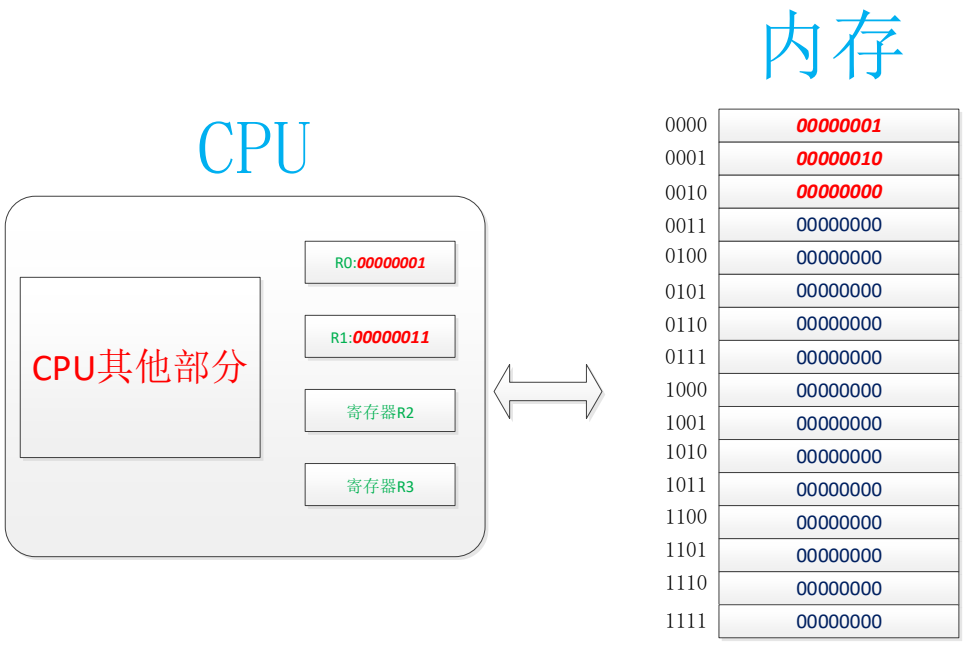

图 0.6 代码 k = i+j 执行的第三步

(4) 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 0.7 所示。

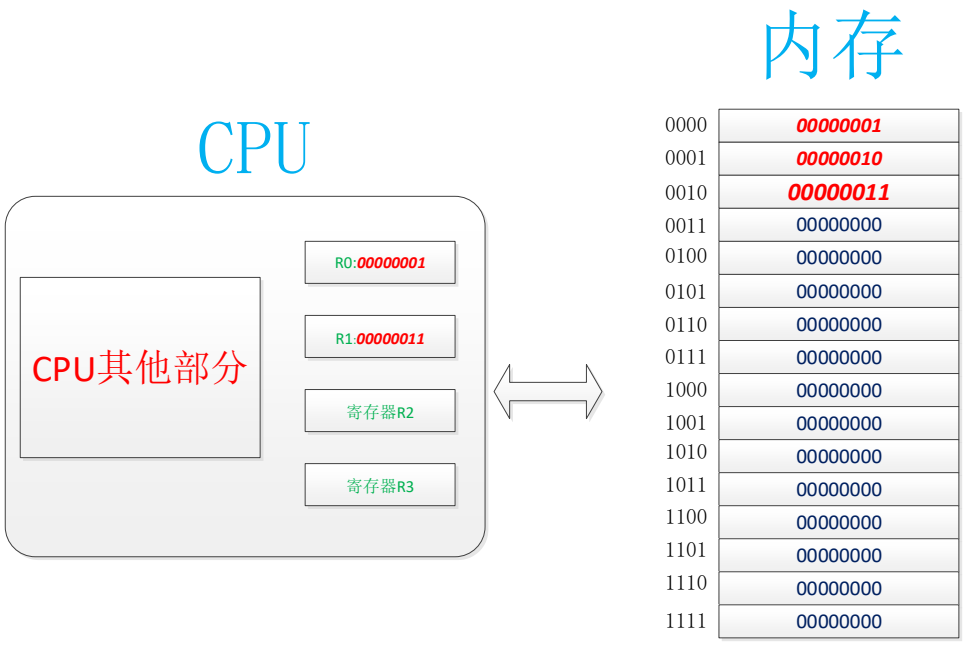

图 0.7 代码 k = i+j 执行的第四步

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 0.8 所示；

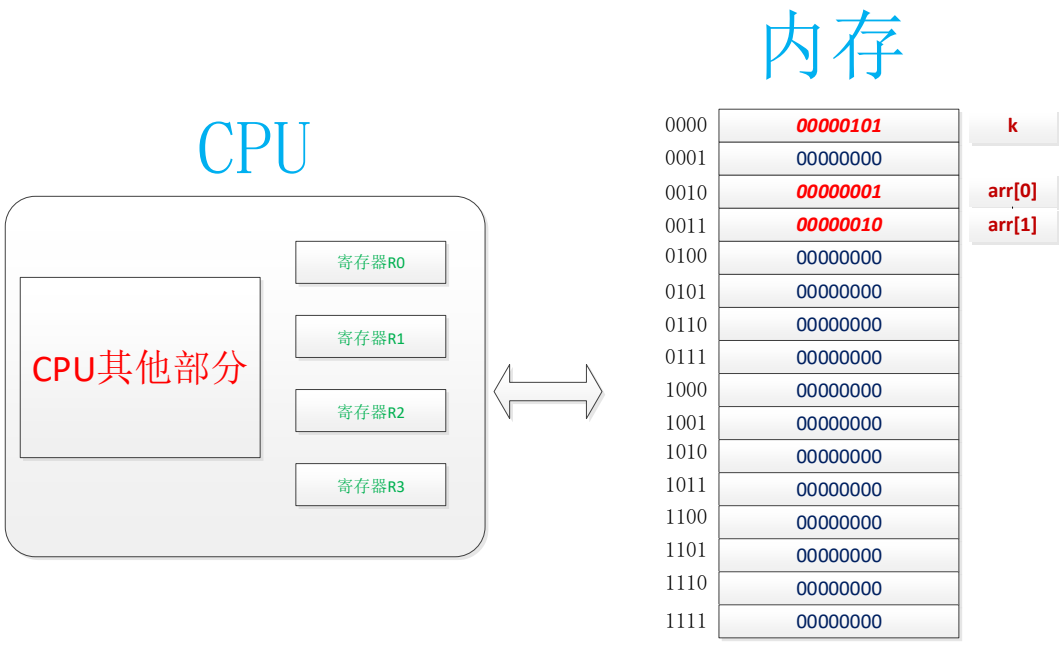

图 0.8 变量分配内存后的最小系统状态图

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 0.9 所示。

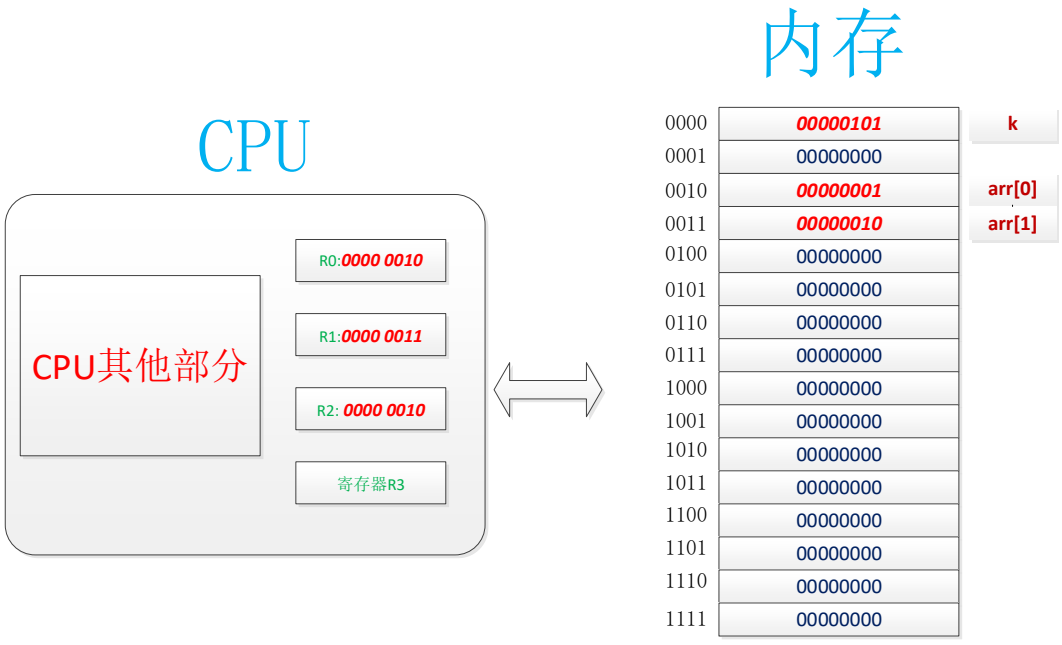

图 0.9 第一阶段完成后的最小系统状态图

（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 0.10 所示；

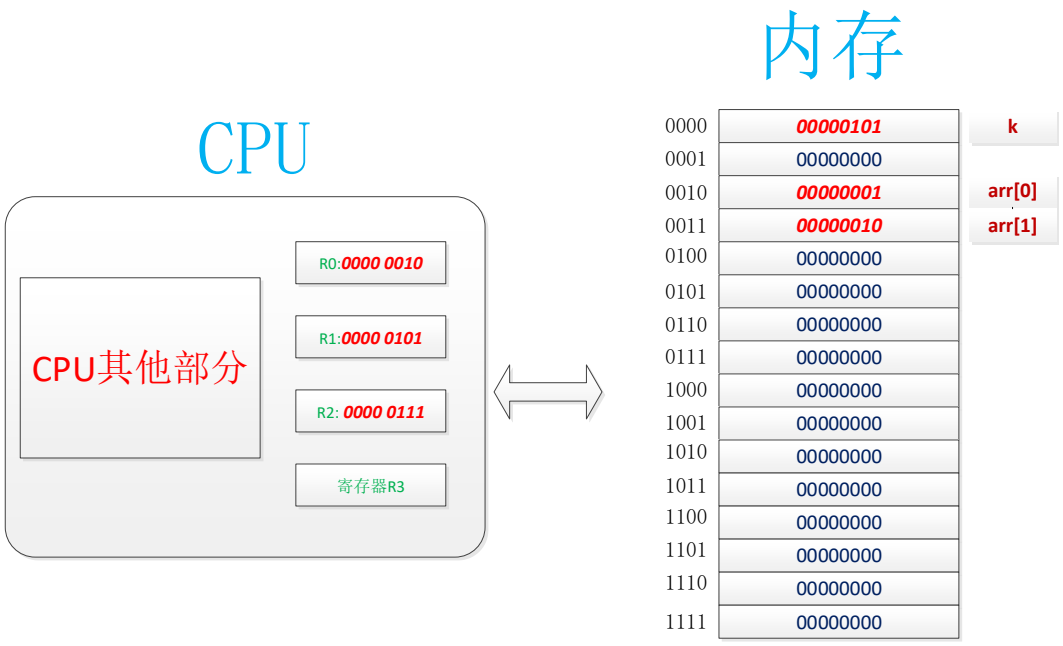

图 0.10 第二阶段完成后的最小系统状态图

（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 0.11 所示。

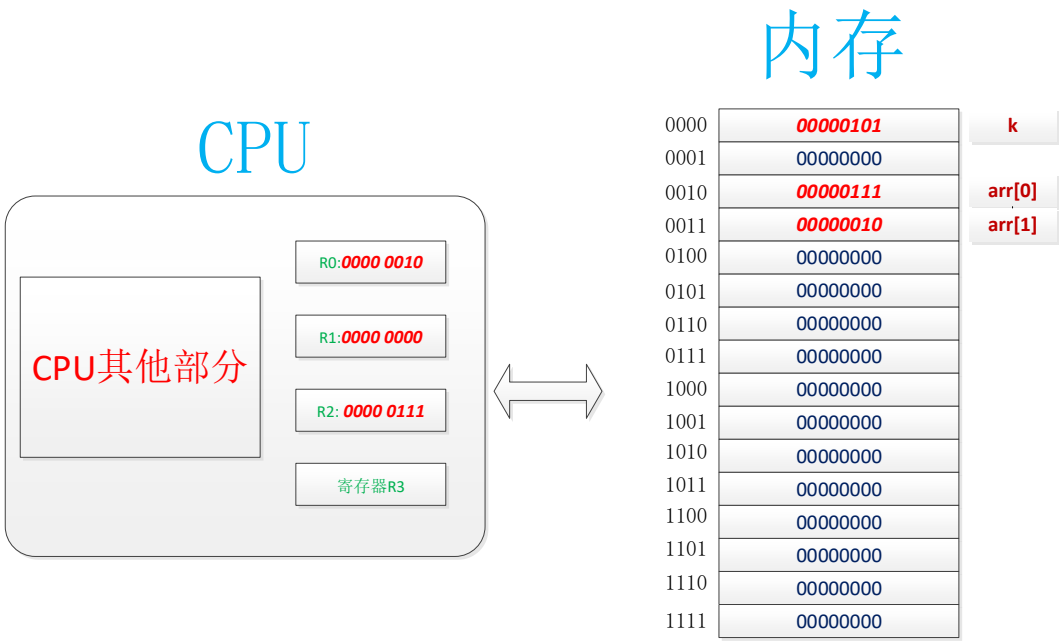

图 0.11 第三阶段完成后的最小系统状态图

0.2 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 0.12 所示。

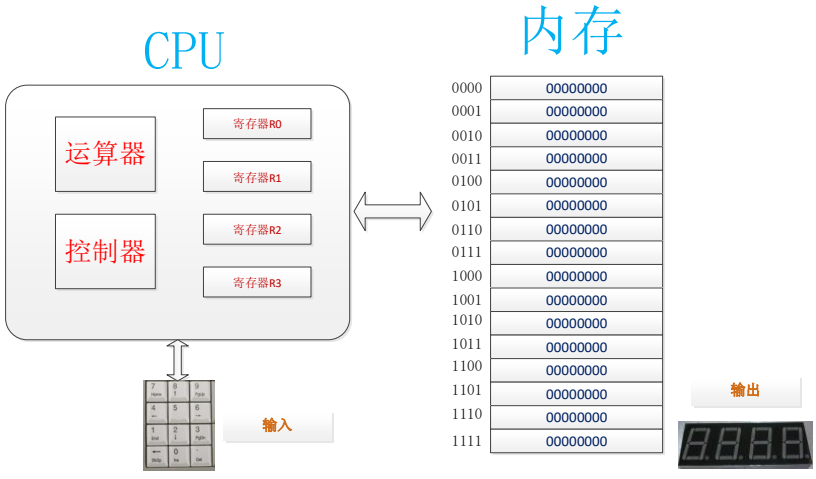

图 0.12 原型系统

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) 定义显存为内存中的最后一个字节，输出设备为一个四位数码显示管，直接显示显存中存储的数字值；

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 9 条指令，格式如下：

指令格式	例子	说明
0	0	停机指令，原型机停止运行
1	1	输入一个整数，这个整数必须大于等于 0 小于 256，输入后，此数值保存在 R0
2 Ra Rb	2 R0 R1	加法指令，将 Ra 和 Rb 的值相加，结果放在 Rb 中
3 Ra Rb	3 R2,R1	减法指令 Ra, Rb。其意义是将寄存器 Rb 的值减去和 Ra 中的值，结果放到 Rb 中，这两个寄存器不能为 R3，当结果大于 0 时，R3 中赋值为 1，当结果小于 0

		时，R3 中赋值为-1，当结果等于 0 时，R3 中赋值为 0
4 $1 Ra	4 10 R1	寄存器直接赋值指令 。其中 Ra 为寄存器名称，$1 为常数，意义是将常数值$1 直接放到寄存器 Ra 中
5 A B	5 R0 R1 5 R0 0000	其中 A，B 为寄存器编号或内存地址，意义是将 A 处的值传送至 B 处
6 bias	6 -2 6 3	判断跳转指令。其中 bias 为一个整数（可以为负），意义是如果 R3 的值为 1，则跳转当前指令+bias 条指令处执行，否则执行下一条指令
7 bias	7 2 7 -3	其中 bias 为一个整数（可以为负），直接跳转当前指令+bias 条指令处执行
8 Ra	8 R0	将 Ra 的值发送至显存，并输出

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 0.13 所示。

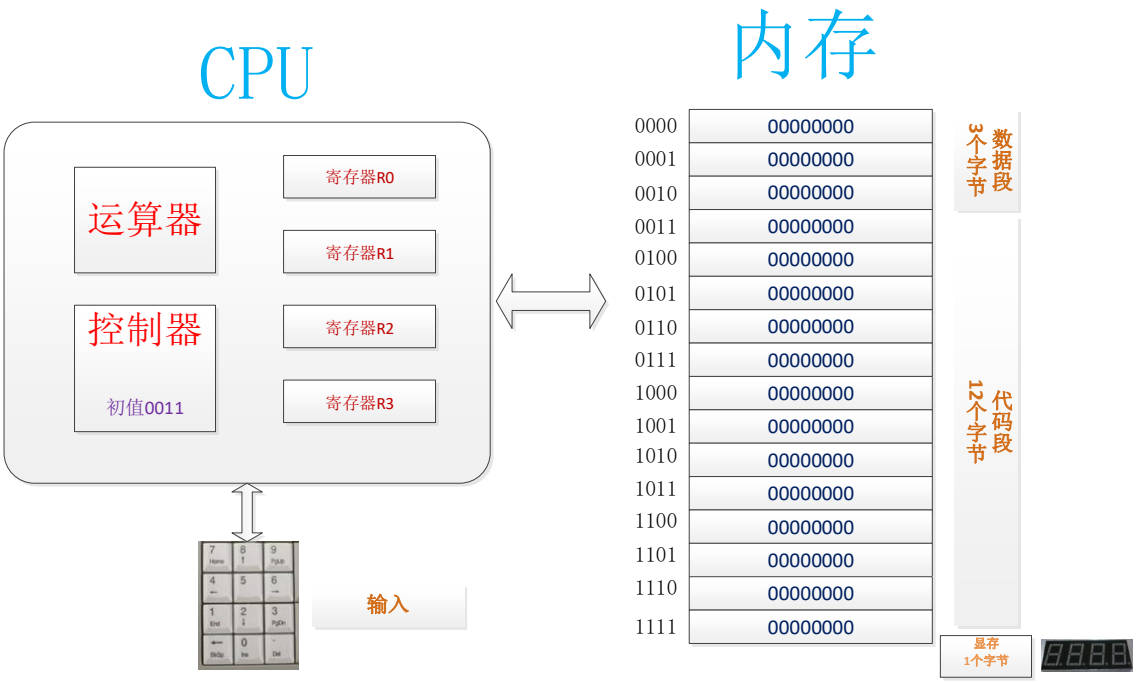

图 0.13 原型系统

下面以一个简单的 c 语言程序来例，来说明原型系统的工作原理。

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：

```
1                    //输入数字 a，并保存在 R0 中
4  1  R2            //寄存器 R2 中赋值 1
2  R0  R1      //将 R0 的值累加到 R1 中
3  R2  R0         //将 R0 的值减一，并根据情况给 R3 赋值
6  -2             //当 R0 大于 0 时，则需要继续进行累加，这是通过跳转到前面的指令来实现，即跳转到 2  R0  R1 这条指令处来继续累加
8  R1          //当 R0 等于 0 时，表示 1+2+……+a 的结果已经计算完成，并且存放在 R1 寄存器中，将结果送至显存进行显示
0                        //程序执行完成
```

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 0.14 所示。

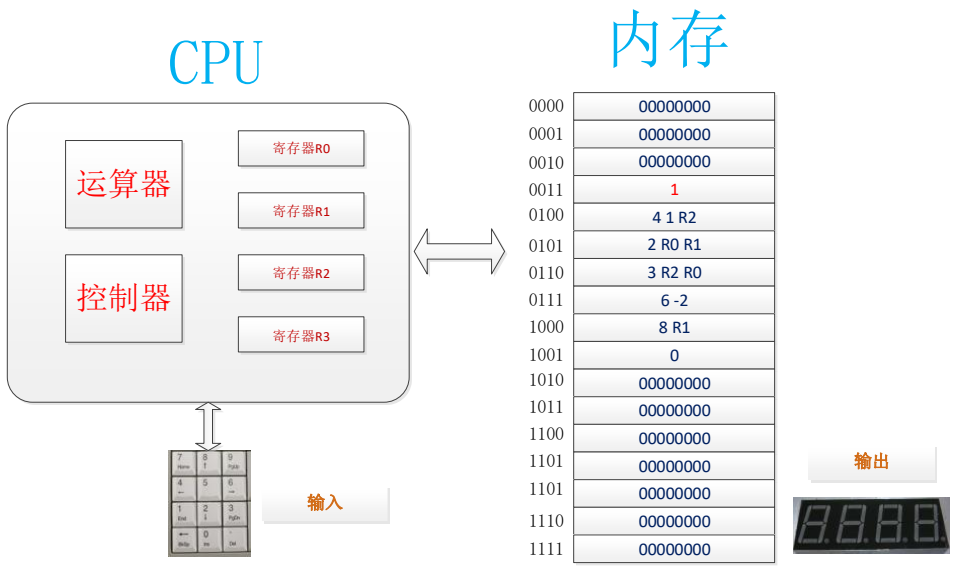

图 0.14 执行第一个程序的原型系统