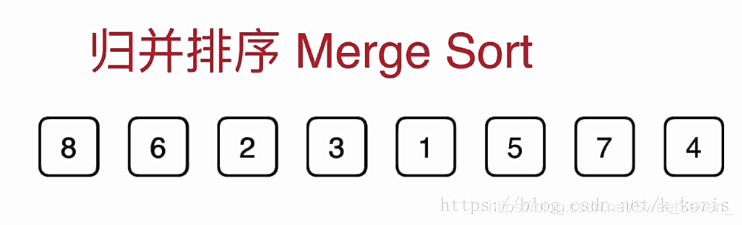
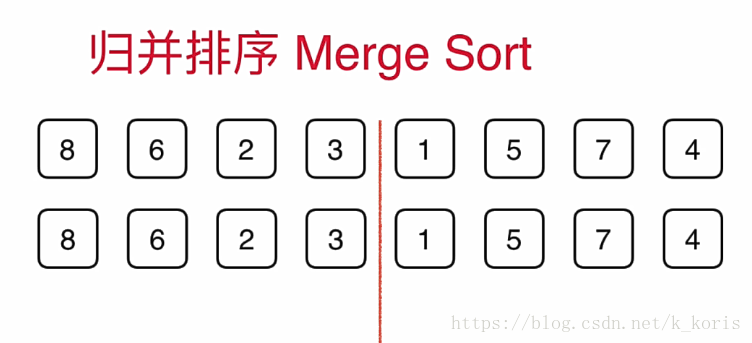
# 归并排序算法

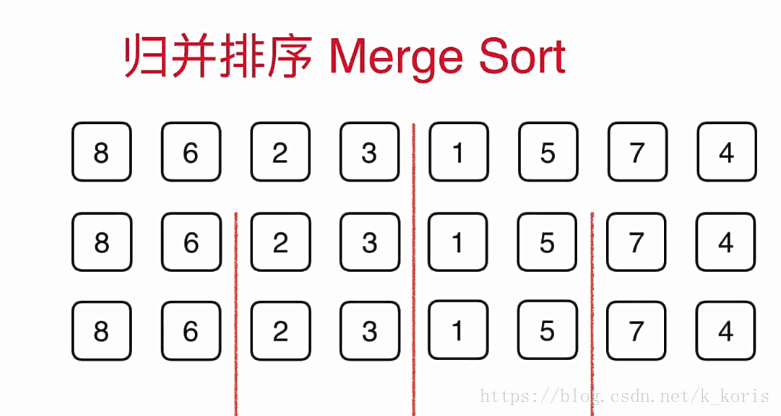
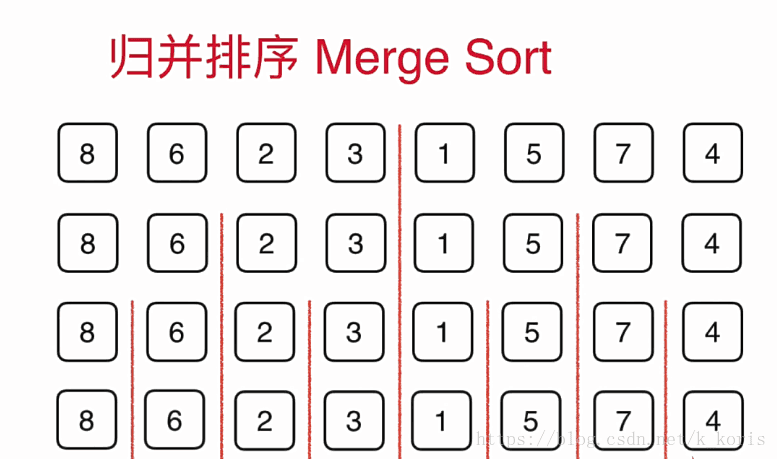
## 算法

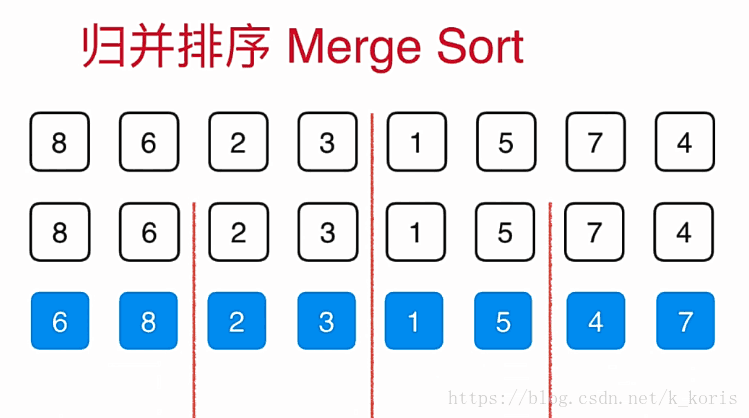
初步了解归并排序：首先将初始序列的n个记录看成是n个有序的子序列，每个子序列的长度为1，然后两两归并，得到n/2个长度为2的有序子序列，在此基础上，再对长度为2的有序子序列进行两两归并，得到若干个长度为4的有序子序列，以此类推，直到得到一个长度为n的有序序列为止。

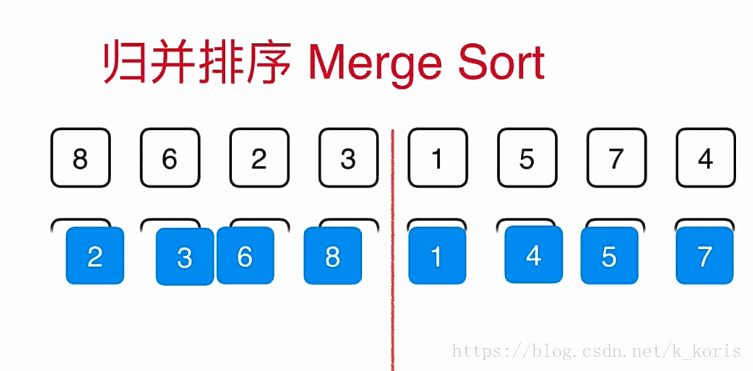
归并排序（Merge Sort）理解：

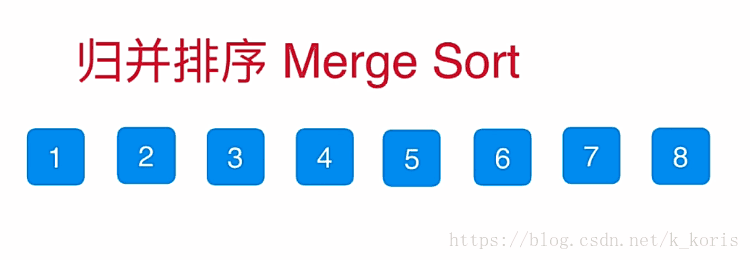
当我们要排序这样一个数组的时候，归并排序法首先将这个数组分成一半。如图：

然后想办法把左边的数组给排序，右边的数组给排序，之后呢再将它们归并起来。当然了当我们对左边的数组和右边的素组进行排序的时候，再分别将左边的数组和右边的数组分成一半，然后对每一个部分先排序，再归并。如图：

对于上面的每一个部分呢，我们依然是先将他们分半，再归并，如图：

分到一定细度的时候，每一个部分就只有一个元素了，那么我们此时不用排序，对他们进行一次简单的归并就好了。如图：

归并到上一个层级之后继续归并，归并到更高的层级，如图：

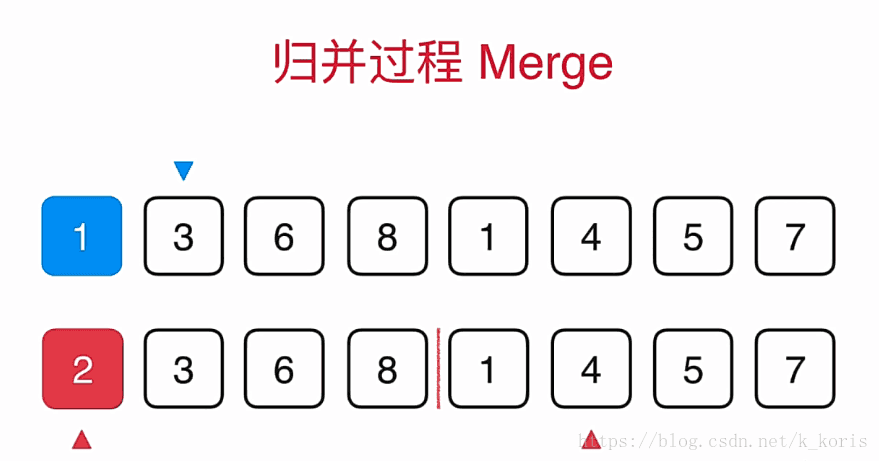
直至最后归并完成。

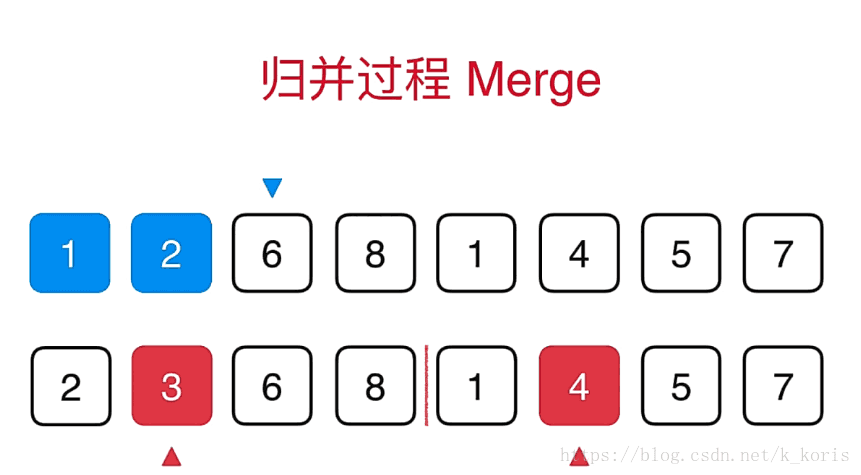
归并细节：

比如有两个已经排序好的数组，如何将他归并成一个数组？

我们可以开辟一个临时数组来辅助我们的归并。也就是说他比我们插入排序也好，选择排序也好多使用了存储的空间，也就是说他需要o（n）的额外空间来完成这个排序。只不过现在计算机中时间的效率要比空间的效率重要的多。无论是内存也好还是硬盘也好可以存储的数据越来越多，所以设计一个算法，时间复杂度是要优先考虑的。

整体来讲我们要使用三个索引来在数组内进行追踪。

蓝色的箭头表示最终选择的位置，而红色的箭头表示两个数组当前要比较的元素，比如当前是2与1比较，1比2小，所以1放到蓝色的箭头中，蓝色的箭头后移，1的箭头后移。

然后2与4比较，2比4小那么2到蓝色的箭头中，蓝色箭头后移，2后移，继续比较.....

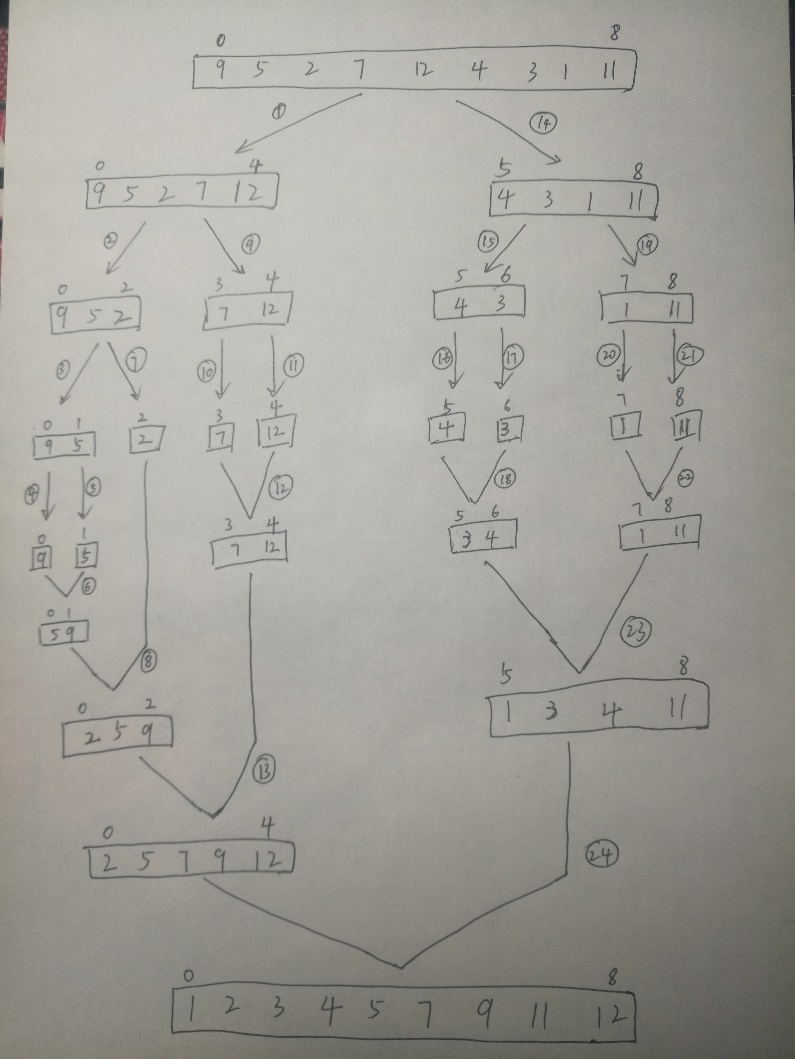
归并思路就是这样了，最后唯一需要注意的是那个先比较完的话，那么剩下的直接不需要比较，把后面的直接移上去就可以了，这个需要提前判定一下。

## C语言设计——递归实现

在上一节中，我们了解了归并排序的基本思想，以及用简单的示例演示归并排序的过程。在本节中，我们要用C语言递归实现归并排序。其参数个数也不是2^n个，更具有一般性。

在下列代码中，我们主要将归并排序分为两个过程，即拆分与合并。简单的讲，拆分就是将一个长度为n的待排序数组整体分解成n个程度为一的待排序数组，合并就是将一个个短的有序数组合并为一个长度为n的数组，合并完成后排序成功。

更深层次得讲：①拆分过程其实并没有对数组本身进行任何操作，只是计算对应的坐标点罢了；②合并过程的坐标结构刚好是拆分过程倒过来；③拆分和合并两个过程也并不是完全独立地进行的。

以9、5、2、7、12、4、3、1、11为例，下图展现了详细的拆分与合并过程：

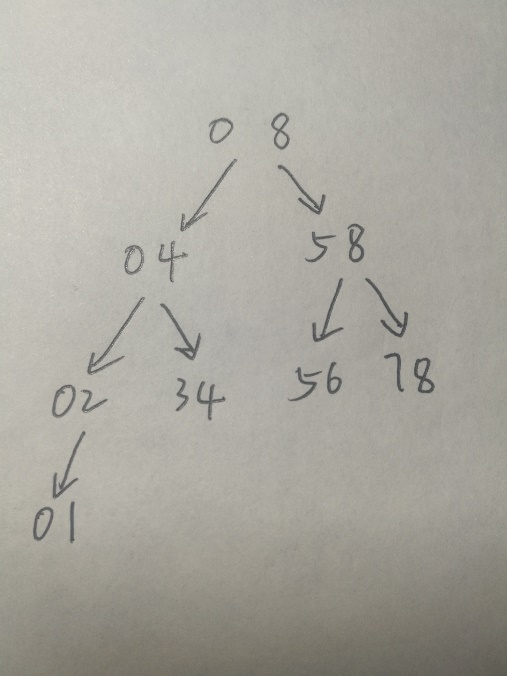
|  |
| --- |
| #include"stdio.h"  #include"stdlib.h"  #define len 9  // 合并 left 到 right 的元素  void merge(int \*arr,int \*temparr,int left,int mid,int right)  {  // 标记左半区第一个未排序元素  int l\_pose = left;  // 标记右半区第一个未排序元素  int r\_pose = mid +1;  // 临时数组元素的下标  int pose = left;  // 合并 左右半区都有剩余元素  while(l\_pose<=mid && r\_pose<=right)  {  if (arr[l\_pose]<arr[r\_pose]) // 左半区第一个未排序元素 小于 右半区第一个未排序元素  {  temparr[pose++] = arr[l\_pose++]; // 将 左半区第一个未排序元素 放到临时数组  }  else // 右半区第一个未排序元素 小于或等于 左半区第一个未排序元素  {  temparr[pose++] = arr[r\_pose++]; // 将 右半区第一个未排序元素 放到临时数组  }  }  // 合并 左半区剩余元素  while(l\_pose<=mid)  {  temparr[pose++] = arr[l\_pose++]; // 将 左半区未排序元素 不断塞入临时数组  }  // 合并 右半区剩余元素  while(r\_pose<=right)  {  temparr[pose++] = arr[r\_pose++]; // 将 右半区未排序元素 不断塞入临时数组  }  // 把临时数组中合并后元素复制回原来数组  while(left<=right)  {  arr[left]=temparr[left];  left++;  }  }  void Merge\_Sort(int \*arr,int \*temparr,int left,int right)  {  // 只有一个元素就不需要划分  // 只有一个元素的区域本身就是有序的，只需要被归并  if(left<right) // 说明有两个以上元素  {  int mid = (left+right)/2; // 找中间点  Merge\_Sort(arr,temparr,left,mid); // 递归划分左半区  Merge\_Sort(arr,temparr,mid+1,right); // 递归划分右半区    merge(arr,temparr,left,mid,right); // 合并已经排序的部分  }  }  int main()  {  // int arr[8] = {3,8,1,5,2,4,6,7};  int arr[len] = {9,5,2,7,12,4,3,1,11};  int temparr[len];  int i;  // 显示排序前的序列  for(i=0;i<=8;i++)  printf("%d ",arr[i]);  printf("\n");  // 排序  Merge\_Sort(arr,temparr,0,len-1);  // 显示排序后的序列  for(i=0;i<=8;i++)  printf("%d ",arr[i]);  printf("\n");  } |

## C语言设计——非递归实现

上述递归实现较为复杂，不适合转换为汇编语言，那我们能否找一种非递归实现的归并排序算法呢？答案是肯定的。

首先，我们对上述递归实现进行深入刨析：为什么要递归？上述过程表明，归并不是以二为倍数递增的，否则9个数归并就变为了前八个先归并再和第九个数归并，这不符合我们的初衷。所以递归就是要确定最合理的归并顺序。

其次，划分实质上是计算合并坐标点的先后顺序，并没有对待排序数组做本质上的改变。所以，我们找一种方法先完全拆分（将要合并的坐标点依次存在缓存区），再读取缓存区存储的坐标进行合并，如此一来就能避免递归调用。

最后，观察递归实现时坐标点的计算过程：

由上图可知，其过程就是构建了一棵二叉树。且其合并顺序为：（0，1）→（0，2）→（3，4）→（0，4）→（5，6）→（7，8）→（5，8）→（0，8）,刚好是二叉树的先序遍历。但是，我们后续想用汇编语言实现此算法，那有没有更简单的方式呢？

其实用数组即可实现，其过程如下：

首先是拆分过程（计算坐标点）。

初始数组中存储（0，8），这个8既是len-1。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **👇** | 👇 |  |  |  |  |  |  |  |
| 0 8 |  |  |  |  |  |  |  |  |

读取（0，8），0和8不相邻进行拆分（0，4）和（5，8）。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **👇** |  | 👇 |  |  |  |  |  |
| 0 8 | 0 4 | 5 8 |  |  |  |  |  |  |

读取（0，4），0和4不相邻进行拆分（0，2）和（3，4）。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **👇** |  |  | 👇 |  |  |  |
| 0 8 | 0 4 | 5 8 | 0 2 | 3 4 |  |  |  |  |

读取（5，8），5和8不相邻进行拆分（5，6）和（7，8）。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | **👇** |  |  |  | 👇 |  |
| 0 8 | 0 4 | 5 8 | 0 2 | 3 4 | 5 6 | 7 8 |  |  |

读取（0，2），0和2不相邻进行拆分（0，1）和（2，2），由于2和2相等，所以不保存（2，2）。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | **👇** |  |  |  | 👇 |
| 0 8 | 0 4 | 5 8 | 0 2 | 3 4 | 5 6 | 7 8 | 0 1 |  |

读取（3，4），3和4相邻，划分结束。

合并过程更为简单，只需要从👇开始，向前依次加载即可。其合并顺序为：（0，1）→（0，2）→（3，4）→（0，4）→（5，6）→（7，8）→（5，8）→（0，8），和递归实现的归并排序顺序完全一致。

|  |
| --- |
| #include"stdio.h"  #include"stdlib.h"  #define len 9  int leb\_start = 0;  int leb\_end = 2;  // 合并 left 到 right 的元素  void Merge\_Sort\_2(int \*arr,int \*temparr,int left,int right)  {  int mid = (left+right)/2;  // 标记左半区第一个未排序元素  int l\_pose = left;  // 标记右半区第一个未排序元素  int r\_pose = mid +1;  // 临时数组元素的下标  int pose = left;  // 合并 左右半区都有剩余元素  while(l\_pose<=mid && r\_pose<=right)  {  if (arr[l\_pose]<arr[r\_pose]) // 左半区第一个未排序元素 小于 右半区第一个未排序元素  {  temparr[pose++] = arr[l\_pose++]; // 将 左半区第一个未排序元素 放到临时数组  }  else // 右半区第一个未排序元素 小于或等于 左半区第一个未排序元素  {  temparr[pose++] = arr[r\_pose++]; // 将 右半区第一个未排序元素 放到临时数组  }  }  // 合并 左半区剩余元素  while(l\_pose<=mid)  {  temparr[pose++] = arr[l\_pose++]; // 将 左半区未排序元素 不断塞入临时数组  }  // 合并 右半区剩余元素  while(r\_pose<=right)  {  temparr[pose++] = arr[r\_pose++]; // 将 右半区未排序元素 不断塞入临时数组  }  // 把临时数组中合并后元素复制回原来数组  while(left<=right)  {  arr[left]=temparr[left];  left++;  }  }  void Merge\_Sort\_1(int \*lebal,int left,int right)  {  int mid = (left+right)/2; // 找中间点  if(left-mid) // left不等于mid才保存  {  lebal[leb\_end] = left;  leb\_end++;  lebal[leb\_end] = mid;  leb\_end++;  }  if(mid+1-right) // mid+1不等于right才保存  {  lebal[leb\_end] = mid+1;  leb\_end++;  lebal[leb\_end] = right;  leb\_end++;  }  }  void show(int \*arr,int \_long)  {  int i;  for(i=0;i<\_long;i++)  printf("%d ",arr[i]);  printf("\n");  }  int main()  {  // int arr[8] = {3,8,1,5,2,4,6,7};  int arr[len] = {9,5,2,7,12,4,3,1,11};  int lebal[len\*2];  int temparr[len];  int left,right;  lebal[0] = 0;  lebal[1] = len-1 ;  // 显示排序前的序列  show(arr,len);  // 划分  for(;lebal[leb\_start+1]-lebal[leb\_start]>1;)  {  left = lebal[leb\_start];  leb\_start ++ ;  right = lebal[leb\_start];  leb\_start ++ ;  Merge\_Sort\_1(lebal,left,right);  }  show(lebal,leb\_end);  // 合并  for(;leb\_end>0;)  {  leb\_end--;  right = lebal[leb\_end];  leb\_end--;  left = lebal[leb\_end];  Merge\_Sort\_2(arr,temparr,left,right);  }    // 显示排序后的序列  show(arr,len);  } |

## 汇编语言设计

用LoongIDE新建一个工程。

移除core文件夹下的bsp\_start.c源文件。

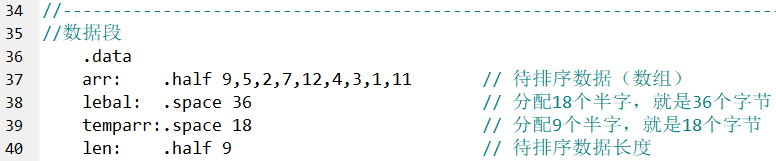
移除main.c源文件。

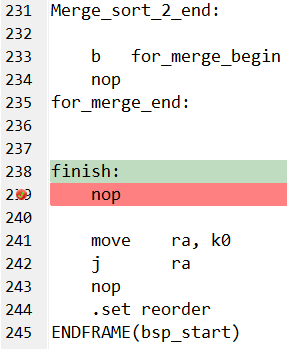
添加bsp\_start.S源文件。

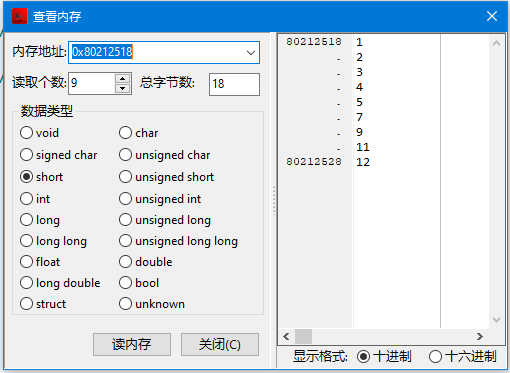
在bsp\_start.S下添加如下代码：

|  |
| --- |
| /\*  \* bsp\_start.S  \*  \* created: 2022/1/29  \* author: Li TianLing  \*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*  \* v0 存储器寻址辅助寄存器  \* v1 跳转辅助寄存器  \* s0 参数len（也是length） （常量）  \* s1 待排序数据arr基地址 （常量）  \* s2 坐标缓存数组lebal基地址 （常量）  \* s3 临时数组temparr基地址 （常量）  \* s4 哨兵left  \* s5 哨兵right  \* s6 哨兵mid (小心计算)(left+right)/4\*2  \* s7 哨兵leb\_start  \* s8 哨兵leb\_end  \* t1 哨兵l\_pose  \* t2 哨兵r\_pose  \* t3 哨兵pose  \* t4 arr[l\_pose]  \* t5 arr[r\_pose]  \* t6 temparr[left]  \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #include "regdef.h"  #include "cpu.h"  #include "asm.h"  //-----------------------------------------------------------------------------  //数据段  .data  arr: .half 9,5,2,7,12,4,3,1,11 // 待排序数据（数组）  lebal: .space 36 // 分配18个半字，就是36个字节  temparr:.space 18 // 分配9个半字，就是18个字节  len: .half 9 // 待排序数据长度  //-----------------------------------------------------------------------------  //代码段  .text  FRAME(bsp\_start,sp,0,ra)  .set noreorder  move k0, ra /\* 保存返回地址 \*/  /\* 开始初始化 \*/  la v0,len // 将长度2\*len加载到s0（常量）  lh s0,(v0)  sll s0,1  la s1,arr // 将arr基地址加载到s1（常量）  la s2,lebal // 将lebal基地址加载到s2（常量）  la s3,temparr // 将temparr基地址加载到s3（常量）  sub t4,s0,2 // 将2\*len-2保存到lebal[2]  add v0,s2,2  sh t4,(v0)    la s7,0 // lebal\_start初始化为0  la s8,4 // lebal\_end初始化为4  /\* 开始划分 \*/  for\_split\_begin:  add v0,s2,s7 // 将lebal[leb\_start]加载到left  lh s4,(v0)  add s7,2 // leb\_start后移  add v0,s2,s7 // 将lebal[leb\_start]加载到right  lh s5,(v0)  sub s7,2 // leb\_start归位    sub v1,s5,s4 // 如果right<=left+2就结束  sub v1,2  blez v1,for\_split\_end  nop    add s7,4 // 否则start后移，left与right已加载，无需重复加载    Merge\_sort\_1\_begin:  add s6,s4,s5 // 找中间点 mid=(left+right)/4\*2  div s6,4  mflo s6  mul s6,2  if\_left\_mid\_begin:  beq s4,s6,if\_left\_mid\_end // 如果left==mid结束if  nop  add v0,s2,s8 // 将left保存到lebal[leb\_end]  sh s4,(v0)  add s8,2 // leb\_end后移    add v0,s2,s8 // 将mid保存到lebal[leb\_end]  sh s6,(v0)  add s8,2 // leb\_end后移  if\_left\_mid\_end:  if\_mid\_right\_beging:  add v1,s6,2  beq v1,s5,if\_mid\_right\_end // 如果mid+2==right结束if  nop    add v0,s2,s8 // 将mid+2保存到lebal[leb\_end]  sh v1,(v0)  add s8,2 // leb\_end后移  add v0,s2,s8 // 将right保存到lebal[leb\_end]  sh s5,(v0)  add s8,2 // leb\_end后移  if\_mid\_right\_end:  Merge\_sort\_1\_end:    b for\_split\_begin  nop  for\_split\_end:  nop  nop    /\* 开始合并 \*/  for\_merge\_begin:  blez s8,for\_merge\_end // leb\_end<=0合并结束  nop    sub s8,2 // leb\_end前移  add v0,s2,s8 // 将lebal[leb\_end]加载到right  lh s5,(v0)  sub s8,2 // leb\_end前移  add v0,s2,s8 // 将lebal[leb\_end]加载到right  lh s4,(v0)      Merge\_sort\_2\_begin:  add s6,s4,s5 // mid=(left+right)/4\*2  div s6,4  mflo s6  mul s6,2    move t1,s4 // 标记左半区第一个未排序元素 l\_pose=left  add t2,s6,2 // 标记右半区第一个未排序元素 r\_pose=mid+2  move t3,s4 // 临时数组元素的下标 pose = left    while\_1\_begin: /\* 合并 左右半区都有剩余元素 \*/  sub v1,t1,s6 // l\_pose>mid结束  bgtz v1,while\_1\_end  nop  sub v1,t2,s5 // r\_pose>right结束  bgtz v1,while\_1\_end  nop    if\_pose\_begin:  add v0,s1,t1 // 将arr[l\_pose]加载到t4  lh t4,(v0)  add v0,s1,t2 // 将arr[r\_pose]加载到t5  lh t5,(v0)  sub v1,t4,t5 // 如果arr[l\_pose]>=arr[r\_pose],跳转到else  bgez v1,else\_if\_pose  nop    add v0,s3,t3 // 将t4(已加载过的arr[l\_pose])保存到temparr[pose]  sh t4,(v0)  add t1,2 // l\_pose后移  add t3,2 // pose后移    b end\_if\_pose  nop  else\_if\_pose:  add v0,s3,t3 // 将t5(已加载过的arr[r\_pose])保存到temparr[pose]  sh t5,(v0)  add t2,2 // r\_pose后移  add t3,2 // pose后移  end\_if\_pose:    b while\_1\_begin  nop  while\_1\_end:  while\_2\_begin: /\* 合并 左半区有剩余右半区为空 \*/  sub v1,t1,s6 // l\_pose>mid结束  bgtz v1,while\_2\_end  nop  add v0,s1,t1 // 将arr[l\_pose]加载到t4  lh t4,(v0)  add v0,s3,t3 // 将t4保存到temparr[pose]  sh t4,(v0)  add t1,2 // l\_pose后移  add t3,2 // pose后移  b while\_2\_begin  nop  while\_2\_end:    while\_3\_begin: /\* 合并 右半区有剩余左半区为空 \*/  sub v1,t2,s5 // r\_pose>right结束  bgtz v1,while\_3\_end  nop  add v0,s1,t2 // 将arr[r\_pose]加载到t5  lh t5,(v0)  add v0,s3,t3 // 将t5保存到temparr[pose]  sh t5,(v0)  add t2,2 // r\_pose后移  add t3,2 // pose后移  b while\_3\_begin  nop  while\_3\_end:  while\_4\_begin: /\* 把临时数组中合并后元素复制回原来数组 \*/  sub v1,s4,s5 // left>right结束  bgtz v1,while\_4\_end  nop    add v0,s3,s4 // 将temp[left]加载到t6  lh t6,(v0)  add v0,s1,s4 // 将t6保存到arr[left]  sh t6,(v0)  add s4,2 // left向后移  b while\_4\_begin  nop  while\_4\_end:  Merge\_sort\_2\_end:    b for\_merge\_begin  nop  for\_merge\_end:  finish:  nop  move ra, k0  j ra  nop  .set reorder  ENDFRAME(bsp\_start) |

## 调试运行

初始数据如下所示：

如下图所示，在finish后打上断点：

联合调试后，双击s1寄存器，查看对应存储器数据：

如上图所示，排序成功。