微程序控制 CPU 设计

PB09210183 何春晖

2011.12.26

1 总体设计

1.1 指令集

以备选的 12~16 条指令 CPU 作为基准，去掉其中中断相关的 iret 指令，加入了对内存的寄存器间接寻址和对堆栈的 push、pop 操作。

最终定义的指令格式如下

指令	H[7..5]	H[4..3]	H[2]	H[1..0]	L[7..0]	功能
ADD A, B	000	A	0	B	无	A <- A + B
RRC A, B	000	A	1	B	无	A <- B 右环移
AND A, B	001	A	0	B	无	A <- A & B
RET	001	00	1	B	无	出栈并送 PC
PUSH B	001	01	1	B	无	B 入栈
POP B	001	10	1	B	无	出栈并送 B
SUB A, B	010	A	0	B	无	A <- A - B
IRET	010	00	1	00	无	未实现
INC A, B	011	A	0	B	无	A <- B + 1
OR A, B	011	A	1	B	无	A <- A|B
LD A, m	100	A	0	m[9..8]	m[7..0]	A <- (m)
LDR A, B	100	A	1	B	无	A <- (B)
ST A, m	101	A	0	m[9..8]	m[7..0]	(m) <- A
STR A, B	101	A	1	B	无	(A) <- B
JC a	110	00	-	a[9..8]	a[7..0]	C = 1, PC <- a
JZ a	110	01	-	a[9..8]	a[7..0]	Z = 1, PC <- a
JA0 a	110	10	-	a[9..8]	a[7..0]	A0 = 1, PC <- a
JMP a	110	11	-	a[9..8]	a[7..0]	PC <- a
LDI A, i	111	A	0	00	i[7..0]	A <- i
CALL a	111	00	1	a[9..8]	a[7..0]	下指 PC 进栈并转 a

注：

- A, B 为通用寄存器 R0, R1, R2, R3 之一；

- 指令凡修改通用寄存器，置标志位；
- 数据宽 8 位，地址宽 10 位，赋值两端宽度不一时零扩展；
- 指令长 1~2 个字节，按大尾排列。

1.2 设计思路

由于选择微程序做控制，为体现其灵活的优势，打算把整个结构设计得尽可能通用，减少修改 CPU 行为时对硬件设计的修改（最好只修改微码就可以适应不同要求）。

为了达到最大的灵活性，最好是硬件上只留下必要的数据通路，去除所有与指令系统相关的逻辑，而把逻辑完全放到微程序中。

CPU 运转时不断取指令、解码指令、执行指令。从微观的观点看，这三个阶段是一种统一的“执行”操作：取指令是以 PC 为地址的访存的“执行”，解码指令是以 IR 为操作数的计算和控制的“执行”，执行指令是通用寄存器参与的访存、计算或控制的“执行”。因此，为了达到通用，首先应该在数据的表示和操作上抹平三个阶段的差异，提炼出统一的操作接口。

经过构思，决定以寄存器堆来承担这种抽象。这里的堆中的寄存器分两种：真实的和映射的。真实的寄存器是 PC、IR 等和通用寄存器；映射的寄存器是 MAR、MDR 和一些纯粹用组合逻辑虚拟的数据位置。它们统一编址，为统一操作提供条件，使所有操作变为对寄存器的读写和用 ALU 计算。

进一步仔细分析，CPU 工作时的一些操作用程序描述较为麻烦，特别是指令解码，用程序控制效率很低。若采用上述方案，容易演变成“用一个指令系统仿真另一个指令系统”。

因此这里仍然在硬件中引入一点指令系统的逻辑，把指令解码工作由硬件完成，然后将结果通过组合逻辑放到映射寄存器中，其余操作靠微程序完成。此外，作为逻辑基础，把微指令逻辑和访存控制信号拿入硬件实现。

最终设计框图如图 1。图中虚线为控制信号，实线为数据通路，寄存器堆周围的线为映射关系。这个设计为微程序控制的不定多周期 CPU。

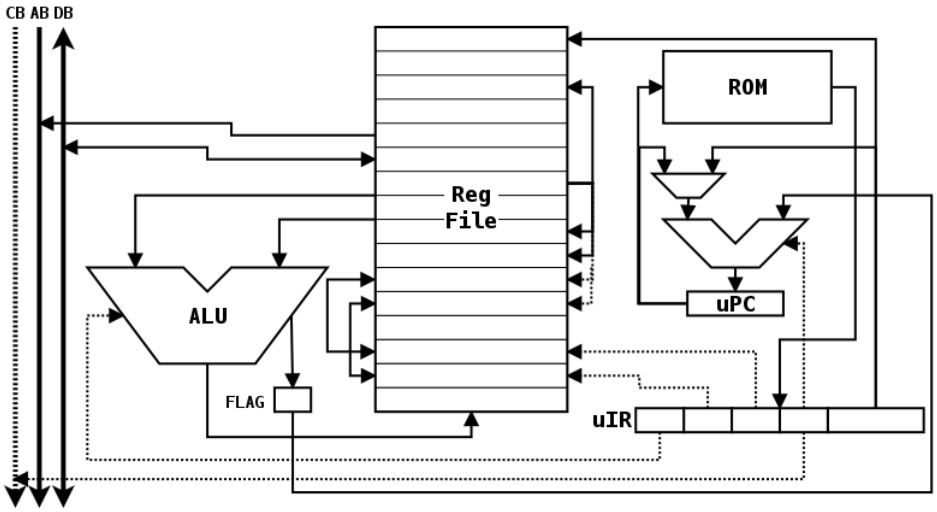

图 1: 微程序控制 CPU 框图

2 部件设计

2.1 寄存器堆

考虑到指令最长 16 位和统一的需要，寄存器堆由 16 个 16 位寄存器组成。其中若干为映射寄存器。各个寄存器的约定映射或用法如下：

编号	名称	功能
0	r0	通用寄存器 A0
1	r1	通用寄存器 A1
2	r2	通用寄存器 A2
3	r3	通用寄存器 A3
4	wa	指令 A 寄存器窗口
5	wb	指令 B 寄存器窗口
6	addr	指令 [9..0] 映射
7	op	指令 [15..13],[10] 映射
8	pc	程序计数器
9	ir	指令寄存
a	mar	内存地址寄存
b	mdr	内存数据寄存
c	sp	堆栈顶地址
d	src	指令 [12..11],[9..8] 映射
e	tmp	临时寄存
f	imm	微指令 [7..0] 映射

可见指令解码的结果从 4、5、6、7 号映射回来。

寄存器堆为双输出、单输入。其中输入端口和其中一个输出端口共享编号。

2.2 运算器

与寄存器堆相配，运算器为 16 位。但要实现的指令系统是 8 位的，且要处理进位等标志，直接用 16 位操作兼容 8 位较困难。因此，计算方式中加入了几个 8 位运算，结果高 8 位为零。

计算方式共 8 种，如下：

方式	操作
000	d = b
001	d = a + b
010	d = ~(a|b)
011	d = b << 8
100	d = a + b, 8bit
101	d = a - b, 8bit
110	d = b >> 1
111	d = b 右环移, 8bit

这样设计显得不太好，但是为了兼容 8 位的情形，暂且如此。

注意逻辑运算只选用了 NOR 运算，其他 AND、OR、NOT 逻辑运算由微程序用 NOR 组合而成。

2.3 控制器

控制器控制微程序的执行逻辑。控制器一般按照 uPC 顺序取微指令，为了满足指令解码和跳转指令的逻辑，加入微跳转指令和逻辑。

由于微指令结构简单，取微指令、执行微指令放在一个周期中完成。

2.3.1 微指令结构

微指令为 24 位，格式如下

```
alu_f_en fop asrc bsrc j_flags wc rc imm8
    1     3    4    4     2     1  1   8
```

含义如下：

```
(asrc) <- (asrc) fop (bsrc), 跳转方式 j_flags, 读令 rc, 写令 wc
alu_f_en: 是否置标志
```

跳转方式如下表：

j_flags	跳转方式	微地址
00	不转	—
01	条件转	imm8
10	必转	imm8
11	必转	运算器结果

其中因编码位不足，条件转移时无法指明为何种条件，而是通过转移到 imm8 + {A0, C, Z} 来表示条件。另外寄存器 f 可以映射 imm8 位来提供立即数。

这种微指令的格式还是比较简单的，除了转移方式外，其余的位直接当作信号送到各个部件即可。

2.3.2 微码填写

初始化微码 复位后，从 00 开始取微指令执行。

```
00      0cf080 sp <- 80 //初始化 sp
```

取指令微码

```
01      0b8000 mar <- pc
02      18f101 pc <- pc + 1, rc <- 1
03      39a000 irh <- mdr
04      17fc10 jmp op(ir)+16 //根据 op 转跳转表
```

跳转表

```
10      c45801 al <- al+bl,jmp 1 //add8
11      045821 al <- bl, jmp rrc //rrc8
12      0e5822 tmp <- b, jmp and //and
13      1dfc80 jmp src+spbr      //jmp ret,push,pop.. **
14      d45801 a <- a-b , jmp 1  //sub8
...
```

在跳转表中，利用了微指令空闲码位先行进行一些操作，提高运行效率。有些简单指令如 ADD，操作在跳转表已经做完，可以直接转移回到取指令。

指令执行微码 根据指令的功能在跳转表跳转的地址处填写即可，略去。

3 仿真测试

3.1 测试一：cpu12test

这是仿真模板提供的默认程序，作用是测试 12 条子集中指令的正确性。运行结果若正确，内存地址 57H,5EH,5FH, 分别为 29H, AAH, 55H。

仿真后波形和内存如图 2 和图 3。

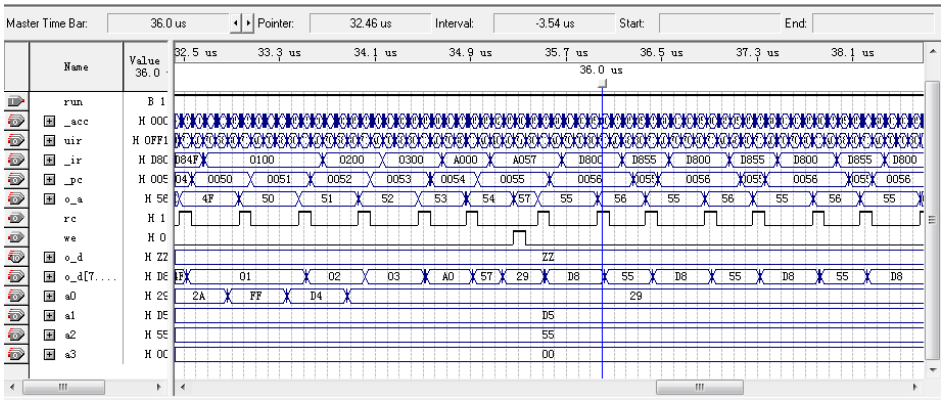

图 2: cpu12test 波形

50	01	02	03	A0	57	D8	55	29
58	E0	11	E0	22	D8	17	AA	55

图 3: cpu12test 内存

可见结果正确。

3.2 测试二：quicksort

这是自编代码，作用是对数组做快速排序。程序用到了 call、ret、push、pop 指令，用来测试堆栈。

起始内存区 70H~77H 中内容为：

```
88 55 33 01 22 45 AA AB
```

仿真后波形和内存如图 4 和图 5。

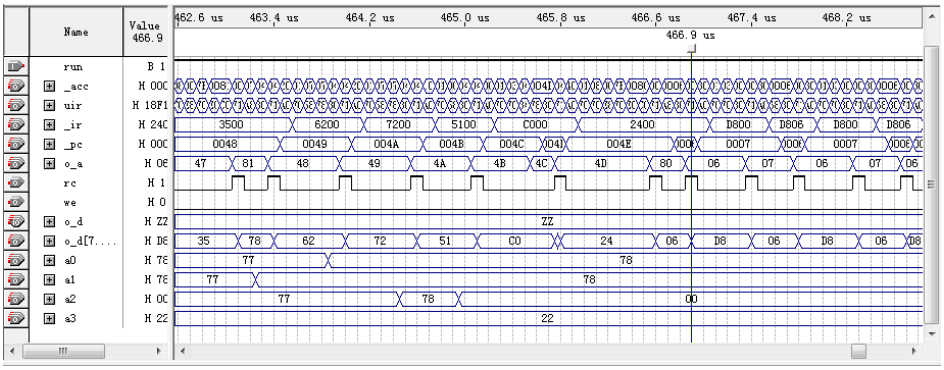

图 4: quicksort 波形

68	24	00	00	00	00	00	00	00
70	AB	AA	88	55	45	33	22	01
78	11	33	77	98	90	10	35	FF

图 5: quicksort 内存

4 下载测试

使用 cpu12_acc4 模板代入并下载。代入时要注意的是，实验系统上的存储器是异步的，而 cpu12_sim 中的存储器使用时钟对地址锁存，因此有一些微小的问题，要做一点改动（读令、写令与时钟相与）。另外 o_d 总线在不写存时要设为高阻（仿真时为了配合同步存储器，在不读的情况下，将 o_d 一直输出），否则，实验系统报写存错误。

4.1 测试一：标准 alltest

此为检查通过时的标准测试程序。因为没有实现与实验系统配合的置数逻辑，在测试前要先修改复位时置 PC 为 8'h9（改硬件设计或初始化微程序）。运行结束后，读内存 80C0 开始的 20 个字节，结果正确。

4.2 测试二：quicksort

此为仿真测试中已经测过的快速排序。使用十六进制编辑工具将机器码写入文件 sort.bin 并载入执行（改复位时 PC 为 0）。运行结束后，读内存 70h 到 77h，如下图 6。

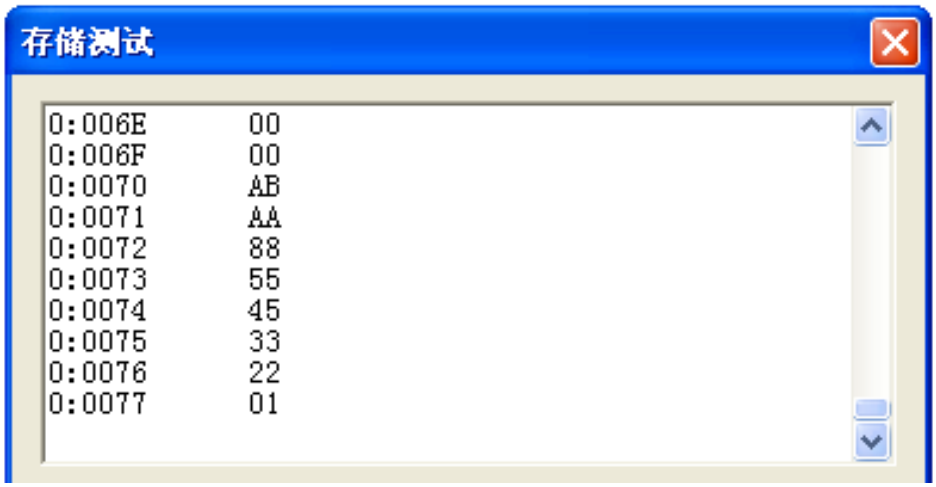

图 6: quicksort 运行结果

可见结果正确。

4.3 发现的问题

下载测试时发现，程序结果大部分时间是正确的，但有时候存储测试读到的内存与预期结果差很远。是设计中读写内存时序有问题，还是实验系统软件或硬件的问题，待验证。

5 总结

5.1 功能实现与不足

通过实验结果看，微程序控制 CPU 设计基本达到了原来设想的目标：灵活。实际上，刚开始 CPU 只有 8 条左右的指令，使用微程序控制使得添加新的功能很方便，思路也比较清晰，所以很快就能实现新的功能，而不影响原来已经实现的功能。另外，由于在微结构上 CPU 的寄存器和运算器都是 16 位的，因此理论上只要稍稍改变微码，而保持硬件不变，就可以变成 16 位 CPU。不过，现在使用 Verilog 描述硬件，修改硬件设计其实也不难，在规模小时甚至比编写微码更加方便，组合逻辑控制的 CPU 在这里劣势也不多了。

这个 CPU 的一个较明细的不足是效率不高。若用组合逻辑控制实现，三到四个周期就能做完一条指令，而这里光是取指令就四个周期。这也跟原来要求灵活的设计方案有关。理论上，只要把组合逻辑的真值表看作微程序放进 ROM 中，则微程序控制的效率与组合逻辑控制是一样的，只是组合逻辑控制存储的是语法，微程序控制存储的是语义。但这样设计就太死板，还不如直接用组合逻辑方便。实际的 CPU 设计应该在灵活性与性能之间根据需要折中。

这个 CPU 的另一个不足是微指令格式还是不够合理。如现在是二地址，目的与源共享地址，造成有些操作极不方便，而不方便性仅仅是编码的缺陷引入的。这有两点原因：一是经验不足，二是当时设计时过分追求编码的紧凑。

5.2 添加中断处理的设想

基于目前的设计，在硬件上加入一个中断引脚和中断屏蔽触发器，在中断引脚有效且非屏蔽时，将特定的微地址置入 uPC 中即可。软件上，在 ROM 对应地方编写微程序，压栈保存当前寄存器状态，再填写 iret 的微程序即可。

6 附录

6.1 微码列表

```
00      0cf080 sp <- 80 //初始化 sp
01      0b8000 mar <- pc
02      18f101 pc <- pc + 1, rc <- 1
03      39a000 irh <- mdr
04      17fc10 jmp op(ir)+16
05      ...
06      ...
      2. 跳转表 从 16
10      c45801 al <- al+bl,jmp 1 //add8
11      045821 al <- bl, jmp rrc //rrc8
12      0e5822 tmp <- b, jmp and //and
13      1dfc80 jmp src+spbr      //jmp ret,push,pop.. **
14      d45801 a <- a-b , jmp 1  //sub8
15      //iret
16      045820 a <- b,   jmp inc //inc
17      245825 a<-nor(a,b) jmp or//or
18      0b8828 mar <- pc,jmp ld  //ld
19      0b5868 mar <- b, jmp ldr //ldr a, b:  a <- [b]
1a      0b8830 mar <- pc,jmp st  //st
1b      0b486c mar <- a, jmp str //str a, b:  [a] <- b
1c      1dfc38 jmp src+br        //jmp..
1d      1dfc38 jmp src+br        //jmp..
1e      0b882d mar <- pc,jmp ldi //ldi
1f      0b8860 mar <- pc,jmp call//call

      inc:
20      c4f801 a <- a + 1, jmp 1
```

```
      rrc:
21      f44801 al <- rrc8(al), jmp 1
      and:
22      2ef000 tmp <- nor(tmp, 0)
23      24f000 a <- nor(a, 0)
24      a4e801 a <- nor(a, tmp), jmp 1
      or:
25      a4f000 a <- nor(a, 0)
26      0ff801 jmp 1

      ld:
28      18f101 pc <- pc + 1, rc <- 1
29      19a000 irl <- mdr

2a      0b6000 mar<-addr
2b      0ff100 rc <- 1
2c      84a801 a <- mdr, jmp 1

      ldi:
2d      18f101 pc <- pc + 1, rc <- 1
2e      04a801 a <- mdr, jmp 1

      st:
30      18f101 pc <- pc + 1, rc <- 1
31      19a000 irl <- mdr

32      0a4000 mdr <- a
33      0b6000 mar <- addr
34      0ffa01 wc <- 1, jmp 1

      jmp*:
38    br:0b8440 mar <- pc, br c   //c
39       0b8448 mar <- pc, br z   //z
3a       0b8450 mar <- pc, br a0  //a0
3b       0b8000 mar <- pc         //jmp
3c       0ff100 rc <- 1
3d     j:19a000 irl <- mdr
3e       086801 pc <- addr, jmp 1
```

```
    c:
40      18f801 pc <- pc + 1, jmp 1
41      18f801 pc <- pc + 1, jmp 1
42      0ff93d rc <- 1, jmp j
43      0ff93d rc <- 1, jmp j
44      18f801 pc <- pc + 1, jmp 1
45      18f801 pc <- pc + 1, jmp 1
46      0ff93d rc <- 1, jmp j
47      0ff93d rc <- 1, jmp j
    z:
48      18f801 pc <- pc + 1, jmp 1
49      0ff93d rc <- 1, jmp j
4a      18f801 pc <- pc + 1, jmp 1
4b      0ff93d rc <- 1, jmp j
4c      18f801 pc <- pc + 1, jmp 1
4d      0ff93d rc <- 1, jmp j
4e      18f801 pc <- pc + 1, jmp 1
4f      0ff93d rc <- 1, jmp j
    a0:
50      18f801 pc <- pc + 1, jmp 1
51      18f801 pc <- pc + 1, jmp 1
52      18f801 pc <- pc + 1, jmp 1
53      18f801 pc <- pc + 1, jmp 1
54      0ff93d rc <- 1, jmp j
55      0ff93d rc <- 1, jmp j
56      0ff93d rc <- 1, jmp j
57      0ff93d rc <- 1, jmp j

     call:
60    18f101 pc <- pc + 1, rc <- 1
61    19a000 irl <- mdr
62    0bc000 mar <- sp
63    0a8000 mdr <- pc
64    1cf201 sp <- sp + 1, wc <- 1
65    086801 pc <- addr, jmp 1

      ldr:
68   0ff100 rc <- 1
69   04a801 a <- mdr, jmp 1
```

```
       str:
6c    0a5000 mdr <- b
6d    0ffa01 wc <- 1, jmp 1

  spbr:
80     5ee858 tmp <- 0, jmp ret
81     0bc870 mar <- sp, jmp push
82     5ee878 tmp <- 0, jmp pop
83     0ff801 jmp 1

  ret:
58    2ef000 tmp <- nor(tmp, 0)
59    1ce000 sp <- sp + tmp
5a    0bc000 mar <- sp
5b     0ff100 rc <- 1
5c    08a801 pc <- mdr, jmp 1

      push:
70     0a5000 mdr <- b
71     1cf201 sp <- sp + 1, wc <- 1
72     0ff801 jmp 1
  pop:
78     2ef000 tmp <- nor(tmp, 0)
79     1ce000 sp <- sp + tmp
7a     0bc000 mar <- sp
7b     0ff100 rc <- 1
7c     05a801 b <- mdr, jmp 1
```

6.2 quicksort 汇编码

```
00 11100000  ldi  a0, 0x70
01 01110000
02 11101000  ldi  a1, 0x78
03 10000000
04 11100100  call quick_sort
05 00011000
06 11011000  jmp  .
07 00000110

//quick_sort(p, r)
```

```
//init reg alloc:
//a0  p
//a1  r
//a2  i
//a3  x
  quick_sort:
18 11111000   ldi  a3, 1
19 00000001
1a 01001011   sub  a1, a3  //r--;
1b 10011101   ldr  a3, a1  //x=*r;
1c 00101100   push a0
1d 00110110   pop  a2      //i=p;

1e 00101100   push a0      //pp = p;
  loop:
1f 00101100   push a0      //
20 01000001   sub  a0, a1  // while(p < r)
21 11001000   jz   out     //
22 34H
23 00110100   pop  a0      //
24 00101100   push a0
25 10000100   ldr  a0, a0
26 01000011   sub  a0, a3
27 11000000   jc   sw2     // if(*p < x) goto sw2 else goto sw1;
28 30H
  sw1:
29 00110100   pop  a0
2a 11100100   call swap    //swap(i, p);
2b 60H
2c 01110010   inc  a2, a2  //i++;
2d 01100000   inc  a0, a0  //p++;
2e 11011000   jmp  loop
2f 1FH
  sw2:
30 00110100   pop  a0
31 01100000   inc  a0, a0  //p++;
32 11011000   jmp  loop
33 1FH
  out:
```

```
34 00101101   push a1
35 00110100   pop  a0
36 11100100   call swap    //swap(r, i);
37 60H
38 00110100   pop  a0      //p = pp;
39 00110100   pop  a0      //pop again...
3a 01101001   inc  a1, a1  //r++;

3b 00101101   push a1
3c 00101110   push a2

3d 00101110   push a2
3e 00110101   pop  a1
3f 01010000   sub  a2, a0
40 11000000   jc   next  //i < p
41 46H
42 11001000   jz   next
43 46H
44 11100100   call quick_sort
45 18H

  next:
46 00110110   pop  a2
47 00110101   pop  a1

48 01100010   inc a0, a2 //i++, p = i
49 01110010   inc a2, a2
4a 01010001   sub a2, a1
4b 11000000   jc  quick_sort //i < r
4c 18H
4d 00100100   ret

  //swap(a0, a2): a0, a2 is a pointer
60 00101101   push a1
61 00101111   push a3
62 10001100   ldr  a1, a0
63 10011110   ldr  a3, a2
64 10100111   str  a0, a3
65 10110101   str  a2, a1
```

```
66 00110111   pop  a3
67 00110101   pop  a1
68 00100100   ret
```

6.3 quicksort 等效 C 代码

```
#include <stdio.h>
#include <stdlib.h>

void swap(int *a, int *b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
}

void quick_sort(int *p, int *r)
{
    int *pp = p;
    int *i = p;
    int x;
    r--;
    x = *r;
    while (p < r)
    {
        if(*p > x)
        {
            swap(i, p);
            i++;
        }
        p++;
    }
    swap(r, i);
    p = pp;
    r++;
    if(p < i)
        quick_sort(p, i);
    i++;
    if(i < r)
```

```
        quick_sort(i, r);
}

int main()
{
    int i;
    int a[8] = {0x88, 0x55, 0x33, 0x01,
                0x22, 0x45, 0xaa, 0xab};
    //srand(time(NULL));
    for(i = 0; i < 8; i++)
    {
        //a[i] = rand()%256;
        printf("%02x\n", a[i]);
    }
    quick_sort(a, a+8);
    for(i = 0; i < 8; i++)
        printf("  %02x\n", a[i]);
}
```