C 语言的位域：

```
#include <stdio.h>
typedef struct _short_test1 {
	int a:2;
	int b:3;
	int c:4;
} short_test1;

int main(int argc, char *argv[])
{
	short_test1 test;
	test.a = 3;
	test.b = 7;
	test.c = 10;
	printf("sizeof:%ld\n", sizeof(short_test1));
	printf("test.a:%d\n", test.a);
	printf("test.b:%d\n", test.b);
	printf("test.c:%d\n\n", test.c);

	return 0;
}
```

运行结果：

```
sizeof:4
test.a:-1
test.b:-1
test.c:-6
```

sizeof 这个结果很好理解，最大存储单元对齐。

下面三个结果实在让人无法理解。

从下页开始从汇编语言层面解析这个 main 函数。

解析：用 gcc main.c 进行编译，不添加优化选项，在 MIPS 下 main 函数的汇编结果如下：

```
(gdb) x /20i main
   0x400640 <main>:   addiu      sp,sp,-40    //开辟 main 函数栈帧
   0x400644 <main+4>:      sw    ra,36(sp)    //保存返回地址
   0x400648 <main+8>:      sw    s8,32(sp)    // 保存函数栈帧 fp，fp==s8
   0x40064c <main+12>:     move      s8,sp    // fp, sp 此时同时指向新栈帧的基址

   // 全局变量存储到新函数栈帧上，sp+0, sp+4, sp+8, sp+12 是预留给 main 调用其他函数
   // 时参数传递 ABI 前四个参数的, gp 为全局指针寄存器
   0x400650 <main+16>:    lui    gp, 0x42
   0x400654 <main+20>:    addiu      gp,gp,-30160
   0x400658 <main+24>:    sw    gp,16(sp)

   //   main 函数的两个参数 argc, argv 存储到 main 函数调用者栈帧末尾的参数槽位上
   // 个人觉得不是必须的，MIPS O32 ABI 规定函数钱四个参数在 a0~a3 寄存器中
   0x40065c <main+28>:    sw    a0,40(s8)
   0x400660 <main+32>:    sw    a1,44(s8)

   // s8+24 是局部变量 test 的存储地点，为什么不在 sp+28，目前还不知道
   // 下面四句就是 test.a = 3 的实现，执行之后 sp+28 的内存布局如下：
   //   0000 0000 0000 0000 0000 0000 0000 0011
   0x400664 <main+36>:    lw    v0,24(s8)
   0x400668 <main+40>:    nop
   0x40066c <main+44>:    ori    v0,v0,0x3 //
   0x400670 <main+48>:    sw    v0,24(s8)

   // 下面四句是 test.b = 7 的实现，ori 的执行过程如下：
   //   v0:   0000 0000 0000 0000 0000 0000 0000 0011
   //   0x1c:0000 0000 0000 0000 0000 0000 0001 1100
   // 执行或运算之后，结果为：
   //   0000 0000 0000 0000 0000 0000 0001 1111
   //也即 test 变量的低 5 位分别为 test.a=2, test.b=7 所填充
   0x400674 <main+52>:    lw    v0,24(s8)
   0x400678 <main+56>:    nop
   0x40067c <main+60>:    ori    v0,v0,0x1c
   0x400680 <main+64>:    sw    v0,24(s8)

   // 下面五句是 test.c=10 的实现：  fffffe1f
   // -481 的二进制表示是：1111 1111    1111 1111 1111 1110 0001 1111
   // v1 中的值是:                0000 0000 0000 0000 0000 0000 0001 1111
   // and 操作之后，也就是把第 6~9 位清空，同时保留其他位，尤其是低 5 位的值：
   //                             0000 0000 0000 0000 0000 0000 0001 1111
   // 再或上 0x140:               0000 0000 0000 0000 0000 0001 0100 0000
```

得到结果：　　0000 0000 0000 0000 0000 0001 010 1 1111
// 此时 test.a，test.b，test.c 三个位域分别依次按序存储在 sp+24 的低位中。
0x400684 <main+68>:　lw　v1,24(s8)
0x400688 <main+72>:　li　v0,-481
0x40068c <main+76>:　and　v0,v1,v0
0x400690 <main+80>:　ori　v0,v0,0x140
0x400694 <main+84>:　sw　v0,24(s8)

// 载入 printf 函数调用的第一个参数: 字符串"%d\n"
0x400698 <main+88>:　lui　v0,0x40
0x40069c <main+92>:　addiu　v0,v0,2400
0x4006a0 <main+96>:　move　a0,v0
// 载入 printf 函数调用的第二个参数 sizeof(test_t)
0x4006a4 <main+100>:　li　a1,4
// 从全局变量中获得 print 函数地址，调用执行
0x4006a8 <main+104>:　lw　v0,-32716(gp)
0x4006ac <main+108>:　nop
0x4006b0 <main+112>:　move　t9,v0
0x4006b4 <main+116>:　jalr　t9
0x4006b8 <main+120>:　nop
0x4006bc <main+124>:　lw　gp,16(s8)

//第二次调用 printf
// 载入 printf 函数调用的第一个参数: 字符串"%d\n"
0x4006c0 <main+128>:　lui　v0,0x40
0x4006c4 <main+132>:　addiu　v1,v0,2412
//载入 test 变量
0x4006c8 <main+136>:　lw　v0,24(s8)
0x4006cc <main+140>:　nop
// 符号扩展低 2 位，也即 test.a，
// 注意这里左右移位以得到最低 2 位，然后符号扩展的过程
// 最低 2 位是 11，32 为符号扩展之后，就是-1 的补码表示，所以输出为-1
// 为什么左右分别左右移位两次?
0x4006d0 <main+144>:　sll　v0,v0,0x6
0x4006d4 <main+148>:　sll　v0,v0,0x18
0x4006d8 <main+152>:　sra　v0,v0,0x18
0x4006dc <main+156>:　sra　v0,v0,0x6?
//　为什么再左右移位一次?
0x4006e0 <main+160>:　sll　v0,v0,0x18
0x4006e4 <main+164>:　sra　v0,v0,0x18?
// printf 函数调用传参，
0x4006e8 <main+168>:　move　a0,v1
0x4006ec <main+172>:　move　a1,v0

```
// load printf 地址，调用 printf 函数
0x4006f0 <main+176>:   lw    v0,-32716(gp)
0x4006f4 <main+180>:   nop
0x4006f8 <main+184>:   move       t9,v0
0x4006fc <main+188>:   jalr   t9
0x400700 <main+192>:   nop
0x400704 <main+196>:   lw    gp,16(s8)


//第三次调用 printf
0x400708 <main+200>:   lui    v0,0x40
0x40070c <main+204>:   addiu       v1,v0,2424

0x400710 <main+208>:   lw    v0,24(s8)
0x400714 <main+212>:   nop
0x400718 <main+216>:   sll    v0,v0,0x3
0x40071c <main+220>:   sll    v0,v0,0x18
0x400720 <main+224>:   sra    v0,v0,0x18
0x400724 <main+228>:   sra    v0,v0,0x5

0x400728 <main+232>:   sll    v0,v0,0x18
0x40072c <main+236>:   sra    v0,v0,0x18

0x400730 <main+240>:   move       a0,v1
0x400734 <main+244>:   move       a1,v0
0x400738 <main+248>:   lw    v0,-32716(gp)
0x40073c <main+252>:   nop
0x400740 <main+256>:   move       t9,v0
0x400744 <main+260>:   jalr   t9
0x400748 <main+264>:   nop
0x40074c <main+268>:   lw    gp,16(s8)


//第四次调用 printf
0x400750 <main+272>:   lui    v0,0x40
0x400754 <main+276>:   addiu       v1,v0,2436

0x400758 <main+280>:   lw    v0,24(s8)
0x40075c <main+284>:   nop
0x400760 <main+288>:   sll    v0,v0,0x7
0x400764 <main+292>:   sll    v0,v0,0x10
0x400768 <main+296>:   sra    v0,v0,0x10
0x40076c <main+300>:   sra    v0,v0,0xc
0x400770 <main+304>:   sll    v0,v0,0x18
```

```
0x400774 <main+308>:   sra   v0,v0,0x18

0x400778 <main+312>:   move      a0,v1
0x40077c <main+316>:   move      a1,v0

0x400780 <main+320>:   lw    v0,-32716(gp)
0x400784 <main+324>:   nop
0x400788 <main+328>:   move      t9,v0
0x40078c <main+332>:   jalr  t9
0x400790 <main+336>:   nop
0x400794 <main+340>:   lw    gp,16(s8)


// main 函数执行，恢复栈帧阶段
0x400798 <main+344>:   move      v0,zero    // v0 保存返回值 0
0x40079c <main+348>:   move      sp,s8      //  恢复 sp
0x4007a0 <main+352>:   lw    ra,36(sp)      //  恢复返回地址
0x4007a4 <main+356>:   lw    s8,32(sp)      //  恢复 fp
0x4007a8 <main+360>:   addiu     sp,sp,40   //  恢复栈帧结构
0x4007ac <main+364>:   jr    ra             //  返回到调用点
0x4007b0 <main+368>:   nop
0x4007b4: nop
0x4007b8: nop
0x4007bc: nop
```

结论：1、位域的获取依靠移位操作来实现;
因为 a, b, c 的值是 int 类型，所以位域扩展为 32 位时采用符号扩展；
2、当 printf 用%d 控制时，位域中的值符号扩展为 32 位，解释为有符号数，
也即数据的二进制补码表示;
3、当 printf 用%u 控制时，位域中的仍符号扩展为 32 位，但是解释为无符号数；

关于 C 语言位域的学习资料：
1、http://tonybai.com/2006/06/19/understand-bit-fields/
2、http://tonybai.com/2013/05/21/talk-about-bitfield-in-c-again/