In More Depth: Bit Fields in C

C allows *bit fields* or *fields* to be defined within words, both allowing objects to be packed within a word *and* to match an externally enforced interface such as an I/O device. All fields must fit within a single word. Fields are unsigned integers that can be as short as 1 bit. C compilers insert and extract fields using logical instructions in MIPS: `and`, `or`, `sll`, and `srl`.

2.8 [20] <§2.5> The following C code allocates three fields with a word labeled `receiver`: a 1-bit field named `ready`, a 1-bit field named `enable`, and an 8-bit field named `receivedByte`. It copies `receivedByte` into `data`, sets `ready` to 0, and sets `enable` to 1.

```
int data;
struct
{
  unsigned int ready:        1;
  unsigned int enable:       1;
  unsigned int receivedByte: 8;
}receiver;
...
  data = receiver.receivedByte;
  receiver.ready = 0;
  receiver.enable = 1;
```

The fields look like this in a word (C typically right-aligns fields):

What is the compiled MIPS code? Assume `data` and `receiver` are allocated to `$s0` and `$s1`.

2.9 [12] <§2.5> Implement the following lines of C code in MIPS:

```
int a = 27;
struct
{
    unsigned int data0 : 8;
```

```
        unsigned int data1 : 8;
        unsigned int data2 : 8;
        unsigned int valid : 1;
     } bits;
        bits.data0 = a;
    bits.data1 = bits.data0;
    bits.data2 = 'd';
    bits.valid = 1;
```

Assume that the struct `bits` is in `$s0` and the memory address of `a` is stored in `$s1`.