ЛПРС2 Лаб вежба

Манипулисање Битима

верзија 1.2

Милош Суботић

5. март 2022.

1 Манипулисање битима

У VHDL-у је био лак приступ појединим битима, јер у дизајну хардвера се веома често ради на нивоу бита. С друге стране, у вишим програмским језицима и процесорима ради се на нивоу речи, евентуално бајта. Из тог разлога, манипулисање битима је отежано. У процесорима, постоји више начина манипулисањем битима:

1. **Маскирање (енгл. Masking)**. Манипулисање битима коришћењем **битских операција (енгл. Bitwise operation)**.
2. **Битска поља (енгл. Bit-fields)**.
3. **Специјалне инструкције за манипулисање битима**.

У идућим подпоглављима ће свака од ових метода бити детљно обрађена.

1.1 Маскирање

Три су основне битских операција:

- **Битско и (енгл. Bitwise and). С оператор је `&`.**
- **Битско или (енгл. Bitwise or). С оператор је `|`.**
- **Битско ексклузивно или (енгл. Bitwise xor). С оператор је `^`.**

Битске операторе `&`, `|` и `^` Не треба мештати са логичким операторима `&&`, `||` и `^^`.

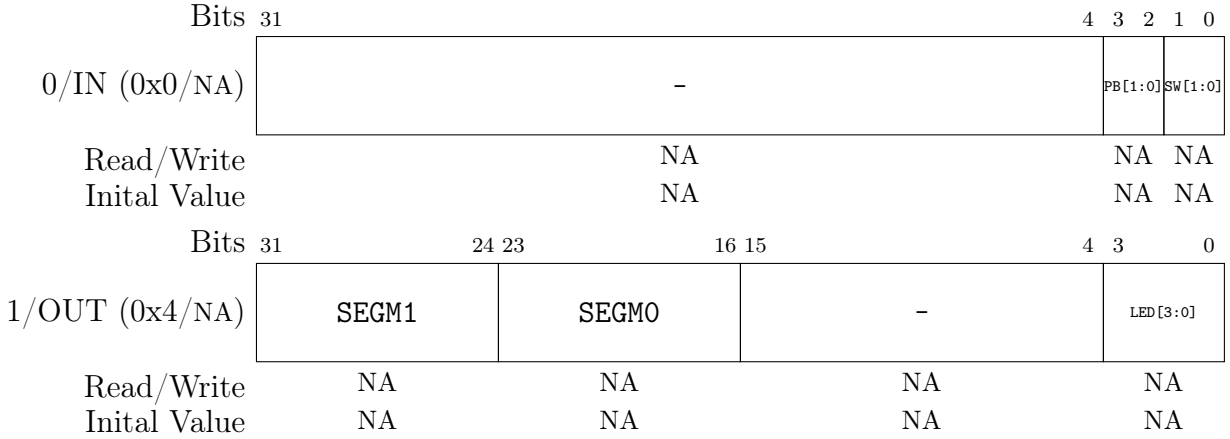
Како се ради са овим операторима показаћемо на примеру датом у датотеци `Source/main.c`. Пример се може покренути на Ubuntu, Raspberry Pi-у, под Windowsom ако инсталирате MinGW, MacOS-ом (ко зна шта тамо треба инсталирати), или неком другом оперативном систему који има барем С компајлер, а погодно `make` (ради лакшег компајлирања и зиповања). Пример се преводи и покреће командом:

```
1    make
```

покренутом из фасцикле `Source`, где је и датотека `main.c`. Осим исписа резултата у конзоли, резултати се такође чувају у датотеци `log.txt`.

У примеру се корсти замишљена `pio` квази улазно-излазна јединице [1]. На Слици 1 је дата њена регистарска мапа.

[1] Писање и читање у меморијски мапиране регистре ове квази улазно-излазна јединице нема ефекте као код правог харвера. Свакако, за само манипулисање битима није неопходно имати смислену и функционалну регистарску мапу.



Слика 1: Меморијска мапа квази улазно-излазне јединице

Пошто је ово квази улазно-излазна јединица онда многи детаљи регистарске мапе нису **применљиви (енгл. NA - Not Applicable)**, па се NA налази свуда наоколо.

Квази улазно-излазна јединица представљена меморијом:

```
129 uint32_t pio_mem[2];
```

Зашто је коришћен `uint32_t`, а не `uint8`?

Два показивача, такозвани **алиаси (енгл. Alias)**, показују на исту меморију:

```
136     volatile uint32_t* p32 = (volatile uint32_t*)pio_mem;
137     volatile uint8_t*  p8  = (volatile uint8_t*) pio_mem;
```

Регистрима је могуће доделити вредност.

```
139     // SW0 and PB0 are set. SW1 and PB1 are cleared.
140     p32[0] = 0x00000005;
141     // LED0 and LED3 are set, while LED1 and LED2 are cleared.
142     p32[1] = 0b1001;
```

Конкретно у првој додели се додељује 0-том 32-битном регистру, док у другој је додељује 1-вом 32-битном регистру. У првој додели додељује се хексадецимално 5 (`0x5`), тј. бинарно 0101 (`0b0101`) у `SW[1:0]` и `PB[1:0]`. Том приликом `SW0` и `PB0` су **постављени (енгл. Set), на јединицу, јелте**. С друге стране `SW1` и `PB1` су **обрисани (енгл. Cleared), на нулу**. Постављен и обрасан су у програмерском свету најчешће изрази стање и операције над битима, али могући су и други. У другој додели излиста су постављени `LED3` и `LED0`. Треба приметити да константа `0b1001` нема свих 32 цифаре односно 32 бита. Компајлер ће ту константу проширити са 0, тј. осталих 28 виших бита ће бити 0. Иначе, хексадецималне и бинарне константе су типа `unsigned int`, док су децималне типа `int`, што је 32 бита на свим модерним личним рачунарима.

Водити рачуна да се **у C-у броји од 0**, а не од 1 као у свакодневном животу и у другим програмским језицима, па је **најмање значајан бит (енгл. LSB - Least Significant Bit) 0. бит**, а **најзначајнији бит (енгл. MSB - Most Significant Bit) 31. бит**, [2].

Следеће је операција **провере (енгл. Query)**.

```
144     printf("SW0 = %d\n", p32[0] & 1); // Query SW0.
145     printf("SW0 = %d\n", p32[0] & 0b0001); // Query SW0.
```

[2]На сличан начин, за правог програмера није округло 1000 већ 1024, а када питате правог програмера за неки број, увек ће дати број који је степен двојке.

Табела 1: Приоритети неких оператора у C-у

Приоритет	Оператор	Опис

1	`[]`
	`.`
	`->`
		Индексирање
		Приступ члану структуре
		Приступ члану структуре преко показивача

2	`! ~` `*`	Логичко и битско не Дереференцирање
3	`*`	Множење
4	`+ -`	Сабирање и одузимање
5	`<< >>`	Померање
6	`&`	Битско и
7	`^`	Битско ексклузивно или
8	`|`	Битско или

Приказан је метод како проверити бит `SW0`, који је 0-ти бит у 0-том улазном регистру. Да не би други бити сметали при провери (у овом случају 2. бит од `PB0`), врши се **маскирање (енгл. Masking)** вршећи **битског и (`&`) са 1 у маски тј. константи, на местима бита од интереса** (у овом случају 0-тог бита). Резултат ове операције је да су бити где је 0 у маски 0, а тамо где је 1 у маски биће такви кавки су били. У овом случају 0-ти бит ће "преживети", док ће 2. бити обрисан на 0.

Међутим ситуација се мало компликује када желимо да проверимо неки виши бит, сем 0-тог.

```
146     printf("PB0 = %d\n", p32[0] & 0x00000004); // Query PB0.
    Wrong!
147     printf("PB0 = %d\n", (p32[0] & 0x00000004) >> 2); // Correct
    .
148     printf("PB0 = %d\n", p32[0]>>2 & 1); // Nicer.
```

Резултат 1. провере ће бити 4. У случају коришћења таквог израза у неком **C-овском услову (енгл. Condition)**, рецимо у `if`-у, овакав приступ маскирању ће бити задовољавајућ, јер C-овски услов **пролази кад је израз различит од 0, а не пролази кад је израз једнак 0**, тако да би горенаведени случај био успешан. Међутим ако желимо да добијемо баш бит, 0 или 1, онда морамо померити за резултат маскирања на 0-ту позицију, како је приказано са 2. провером. Још једноставнији приступ приказан у 3. провери је **прво извршити померање, па онда маскирање са 1**. Оваква маска је једноставнија од оне у 2. провери. Такође, ако се прво врши померање, маска ће увек бити 1. Све то олакшава програмирање и смањује вероватноћу да се направи грешка.

Обратити пажњу на 2. проверу горњег излиста, да су потребне заграде око битско и, док у 3. провери нису потребне заграде око померања. Разлог за то је што померање има већи приоритет од битских оператор. Ипак, ради читљивости, треба писати оператор померања приљубљен, док битски оператор и треба бити одвојен празним местима. Табела 1 приказује приоритете оператор битних за ову лекцију.

Досада су испитивани бити били исписивани на излаз конзоле путем `"%d"`, децималног **спецификатора** `printf` функције. Међутим ако желимо да прикажемо више бита, децимални спецификатор није погодан, већ треба користити хексадецимални.

```
151     printf("OUT = 0x%x\n", p32[1]);
152     printf("OUT = 0x%08x\n", p32[1]); // Nicer.
```

Излист приказује коришћење хексадецималног спецификатора `"%x"`. Готово **увек тре-**

ба додати предметак `"0x"` **испред** `"%x"`, како не би дошло до погрешног тумачења броја као децималног. Многе софтвери то не раде и некад је веома напорно установити да ли су у питању хексадецимални или децимални бројеви. Прва линија излиста приказује наиван приступ. Више цифре које су 0 се не приказују, тако да ширина исписа зависи од вредност броја, што отежава за читање. Друга линија одређује фиксну дужину од 8 цифара (`"8"` испред `"%x"`), а да се притом вишље цифре попуне са 0 (`"0"` испред `"8"`). Фиксна дужина такође омогућава поравнавање исписа, а такође даје и мета-податак о ширини цифре.

Понекад је потребно **издвојити (енгл. Extract)** више бита из променљиве. Том приликом је неопходно само имати ширу маску, са више постављених бита, као што је приказано за излисту испод.

```
153     printf("LED = 0x%x\n", p32[1] & 0xf); // Extract LED.
```

Следећа је операције **постављање (енгл. Set)** одређеног бита на 1, путем **битског или (`|`) са 1 у маски за бит тј. бите који треба поставити**, а 0 за оне које не треба мењати.

```
154     p32[1] |= 0b0010; // Set LED1.
```

Овде се операција извршава директно над променљивом оператором доделе са битским или `|=`. Не треба сметнути са ума да "испод хаубе"процесор мора да учита ту променљиву у регистар, изврши битску операцију или над регистром и маском, па онда уписати променљиву у регистар.

Слична операција је **брисање (енгл. Clear)** одређеног бита на 0, путем **битског или (`&`) са 0 у маски за бите који треба обрисати**, а 1 за оне које не треба мењати.

```
156     p32[1] &= 0b11111111111111111111111111111110; // Clear LED0.
158     p32[1] &= ~0b1000; // Clear LED3. Nicer with inverted mask.
```

Прва брисање у излисту приказује такав приступ, међутим све те јединица, а једна 0 отежавају праћење кода. Из тог разлога, обично се пише **негирана маска, са 1 за бите које треба обрисати**, а 0 за бите које не треба мењати. Негирана тј. инвертована маска се негира путем **битског оператора не (`~`)**, као на примеру код другог брисања из горњег излиста.

Последња операцји из ове групе је **негирање (енгл. Toggle)** одређеног бита (са 0 на 1, односно са 1 на 0), путем **битског ексклузивног или (`&^`) са 1 у маски за бите који треба променити**, а 0 за оне које не треба мењати. Пример испод на излисту.

```
163     p32[1] ^= 0b0010; // Toggle LED1.
```

Писање оваквих маски може да буде веома напорно, поготово ако је неки висок бит у питању. Такође, подложно је грешкама, јер се можемо забројати у свим ти нулама. Из тог разлога боље је користити **генерисане маске**, тако што се померање 1 на позицију бита који се жели мењати, као на примеру испод.

```
170     p32[1] |= 1<<3; // Set LED3.
```

Иако на прву лопту овде изгледа као да постоји редудантна операција померања, ипак ће компајлер **изоптимизовати операције над константама** и у извршном коду ће бити константа.

Међутим, вођење рачуна о голим бројевима је такође заморно и подложно грешкама. Из тог разлога се примењује коришћење **макроа (енгл. Macros)**. Излист приказује исто постављање бита, само се за генерисању маске при померању 1 користи предефинисан макро.

```
172     p32[OUT] |= 1<<LED0; // Set LED0. Using macro.
```

Овај макро дефинише позицију одговарајућег бита (У овом случају `LED0`). Такође је могуће користи макрое као индекс регистра. Пример дефинисаних макроа за ову лекцију је испод.

```
14  #define IN 0
15  #define OUT 1
16  #define IN_WIDTH 32
17  #define OUT_WIDTH 32
18
19
20  #define SW_WIDTH 4
21  #define SW_LSB 0
22  #define SW0 0
23  #define SW1 1
24  #define SW_MSB 1
25
26  #define PB_WIDTH 4
27  #define PB_LSB 2
28  #define PB0 2
29  #define PB1 3
30  #define PB_MSB 3
31
32  #define LED_WIDTH 4
33  #define LED_LSB 0
34  #define LED0 0
35  #define LED1 1
36  #define LED2 2
37  #define LED3 3
38  #define LED_MSB 3
39
40  #define SEGM_WIDTH 8
41  #define SEGM0_LSB 16
42  #define SEGM0_MSB 23
43  #define SEGM1_LSB 24
44  #define SEGM1_MSB 31
```

Овде су дефинисани индекси регистара, ширине регистара и под регистара, најмање значајан и најзначајнији бит подрегистра, као и индекс бита сваког подрегистра. Макрои су обично дефинисани у некој датотеци, обично хедеру. Веома често сам произвођач неког чипа који садржи меморијски мапиране регистре даје хедер специфичан за његов чип, тако да програмер не мора да га пише.

У случају да је потребно мењати више бита треба користити битско или за **спајање (енгл. Merge)** више маски, као на примеру доле.

```
174     p32[OUT] |= 1<<LED2 | 1<<LED1; // Set LED1 and LED2.
176     p32[OUT] &= ~(1<<LED3 | 1<<LED2); // Clear LED2 and LED3.
```

Следећи излист приказује доделу вредности преко показивача на бајте.

```
181     p8[6] = 0x10; // Assign SEGM0 over bytes.
182     p8[7] = 0x32; // Assign SEGM1 over bytes.
```

На тај начин друге вредности у 32-битном регистру неће бити поремећене (`LED[3:0]`

регистар у овом примеру). Треба имати у виду да неке магистрале односно улазно-излазне јединице подржавају једино приступ речима ширине магистрале (овде је то 32-бита), тј. не подржавају да се мењају мање јединице. Ако и подржавају, треба имати на уму да за боље перформансе је боље сложити реч у процесору и извршити један приступ меморији односно јединици, него имати више приступа мањим јединицама од речи.

Следећи излист приказује напредније издвајање вишебитних регистара.

```
184     printf(
185         "SEGM0 = 0x%02x\n",
186         p32[OUT]>>16 & ((1 << 8)-1)
187     ); // Extract SEGM0.
188     printf(
189         "SEGM1 = 0x%02x\n",
190         p32[OUT]>>SEGM1_LSB & ((1 << SEGM_WIDTH)-1)
191     ); // Extract SEGM1.
192     printf(
193         "SEGM[1:0] = 0x%04x\n",
194         p32[OUT]>>SEGM0_LSB & ((1 << SEGM_WIDTH+SEGM_WIDTH)-1)
195     ); // Extract Both.
```

Прво издвајање, као и досад користи померање, али маскирање је другчије. Уместо да се користи константа `0xff` као маска, маска се генерише на основу ширине регистра: прво се 1 помера за ширину, где се добија број за 1 већи од потребне маске (овде је то `256` тј. `0x100`), а онда се тај број умањује за 1 не би ли се добила циљана маска. На овај начин је мања вероватноћа да се погреши при директном прављењу маске са хексадецималном константом. У другом издвајању је све исто, осим што се користе макрои. Треће издвајање које циља оба регистри користи позицију нижег за померање, док збраја ширине оба регистра за маску.

Још једна интересантна операција јесте **додељивање више бита** неком регистру.

```
201     p32[OUT] |= 0x20<<SEGM0_LSB; // Assign only to SEGM0, do not
        touch others.
```

Горњи излист показује наиван приступ, јер користећи само битско или са помереним битима ће само спојити нове бите са већ постојећим у регистру. Следећи излист приказује прави приступ.

```
204     p32[OUT] &= ~(0xff<<SEGM0_LSB); // Clear space with mask.
206     p32[OUT] |= 0x20<<SEGM0_LSB; // Now can be assigned.
```

Прво је неопходно потребно **обрисати маском** бите на одредишној позицији путем битског и, па онда извршити додељивање бита битским или. Иначе, овакав метод се користи у 2D графици при исцртавању спрајтова (енгл. Sprite) у бафер фрејма (енгл. Framebuffer), како би се слика човечулька или чега већ уклопила у поздану, али о томе више речи други пут.

1.2 Битска поља

Коришћење маскирања може да буде веома заморно, без обзира на све напредне методе. Наравно, постоји начин да се још више олакша програмирање. Први корак је олакшавање приступа регистрима, дефинисањем структуре. Пример је дат на следећем излисту.

```
49 typedef struct {
50     // reg 0
51     uint32_t in;
52     // reg 1
53     uint32_t out;
54 } pio_bf_regs;
```

Пошто је **меморијски распоред (енгл. Memory Layout)** структуре такав да се адреса и величина поља `in` поклапа са адресом и величином регистра `IN` тј. `p32[IN]`, могуће је постићи истоветан приступ регистрима преко структуре, услед преклапања меморијског распореда. Приметити да док су **у макроима имена регистара великим словима**, овде су **у структури имена регистара малим словима**. Да су имена у структури исто великими словима као и код макроа, јавиле би се непријатности при компајлирању, јер би предпроцесор покушавао мењати имена дефиницијама макроа.

Структура се даље може **разбити на подрегистре** коришћењем битских поља.

```
1  typedef struct {
2      // reg 0 - in
3      unsigned sw   :  2;
4      unsigned pb   :  2;
5      unsigned      : 28;
6      // reg 1 - out
7      unsigned led    :  4;
8      unsigned        : 12;
9      unsigned segm0  :  8;
10     unsigned segm1  :  8;
11 } pio_bf_subregs;
```

Овде су регистри `in` и `out` разбијени на подрегистре. Иза имена сваког подрегистра, иза `:` дефинише се колико бита тај подрегистар заузима. Резервисане локације не морају да се именују, шта више пожељно је да се не би збуњивали корисници.

Даље је могуће подрегистре **разбити на бите**, као на излисту испод.

```
1  typedef struct {
2      // reg 0 - in
3      unsigned sw0  :  1;
4      unsigned sw1  :  1;
5      unsigned pb0  :  1;
6      unsigned pb1  :  1;
7      unsigned      : 28;
8      // reg 1 - out
9      unsigned led0   :  1;
10     unsigned led1   :  1;
11     unsigned led2   :  1;
12     unsigned led3   :  1;
13     unsigned        : 12;
14     unsigned segm0  :  8;
15     unsigned segm1  :  8;
16 } pio_bf_bit_subregs;
```

Онда је даље је могуће **груписати** подрегистре, у по регистрима којима припадају.

```
1  typedef struct {
2      // reg 0
3      struct {
4          unsigned sw0 :  1;
5          unsigned sw1 :  1;
6          unsigned pb0 :  1;
7          unsigned pb1 :  1;
8          unsigned     : 28;
9      } in;
10     // reg 1
11     struct {
12         unsigned led0  :  1;
13         unsigned led1  :  1;
14         unsigned led2  :  1;
15         unsigned led3  :  1;
16         unsigned       : 12;
17         unsigned segm0 :  8;
18         unsigned segm1 :  8;
19     } out;
20 } pio_bf_grouping;
```

Овде су коришћене **анонимне структуре**, којима није дефинисан тип.
Крајњи корак би могло бити коришћење низова, као што се види на линији 17.

```
56 typedef struct {
57     // reg 0
58     struct {
59         unsigned sw0 :  1;
60         unsigned sw1 :  1;
61         unsigned pb0 :  1;
62         unsigned pb1 :  1;
63         unsigned     : 28;
64     } in;
65     // reg 1
66     struct {
67         unsigned led0  :  1;
68         unsigned led1  :  1;
69         unsigned led2  :  1;
70         unsigned led3  :  1;
71         unsigned       : 12;
72         uint8_t segm[2];
73     } out;
74 } pio_bf_arrays;
```

Још напреднија техника би било коришћење унија.

```
typedef struct {
    // reg 0
    union {
        uint32_t r;
        struct {
            unsigned sw     :  2;
            unsigned pb     :  2;
            unsigned        : 28;
        } s;
        struct {
            unsigned sw0 :  1;
            unsigned sw1 :  1;
            unsigned pb0 :  1;
            unsigned pb1 :  1;
            unsigned     : 28;
        } b;
    } in;
    // reg 1
    union {
        uint32_t r;
        struct {
            unsigned led    :  4;
            unsigned        : 12;
            uint8_t segm[2];
        } s;
        struct {
            unsigned led0     :  1;
            unsigned led1     :  1;
            unsigned led2     :  1;
            unsigned led3     :  1;
            unsigned          : 12;
            unsigned segm0_0 :  1;
            unsigned segm0_1 :  1;
            unsigned segm0_2 :  1;
            unsigned segm0_3 :  1;
            unsigned segm0_4 :  1;
            unsigned segm0_5 :  1;
            unsigned segm0_6 :  1;
            unsigned segm0_7 :  1;
            unsigned segm1_0 :  1;
            unsigned segm1_1 :  1;
            unsigned segm1_2 :  1;
            unsigned segm1_3 :  1;
            unsigned segm1_4 :  1;
            unsigned segm1_5 :  1;
            unsigned segm1_6 :  1;
            unsigned segm1_7 :  1;
        } b;
    } out;
} pio_bf_unions;
```

Унија омогућује да се распоред података у меморији преклапа. На тај начин, једној структури могућ је приступ како регистрима, тако и подрегистрима, па и њиховим битима.

Новодефинисан типови структура се може даље користити у алиасима, као што је дано

у примеру испод.

```
212     volatile pio_bf_regs*    pr = (volatile pio_bf_regs*)
    pio_mem;
213     volatile pio_bf_arrays* pa = (volatile pio_bf_arrays*)
    pio_mem;
214     volatile pio_bf_unions* pu = (volatile pio_bf_unions*)
    pio_mem;
```

За основну структуре са регистрима приступ изгледа на следећи начин.

```
217     pr->out = 0b1001; // New.
```

C-овски оператор `->` служи за приступ члану структуре преко показивача. Да је био инстанциран објекат, онда би приступ био преко оператора `.`, као што је приказано на излисту испод.

```
1 pio_bf_regs r;
2 r.out = 0b1001;
```

Међутим, пошто се регистрима тј. члановима структуре приступа преко показивача мора се користити горепоменути оператор `->`.

Следећи излист приказује коришћење структуре са груписаним битским пољима и низовима.

```
220     printf("SW0 = %d\n", pa->in.sw0); // Query SW0.
230     pa->out.led1 = 1; // Set LED1.
264     printf("SEGM0 = 0x%02x\n", pa->out.segm[0]);
```

А приступ преко структуре са унијама је овакав.

```
221     printf("SW0 = %d\n", pu->in.b.sw0); // Query SW0.
232     pu->out.b.led0 = 0; // Clear LED0.
265     printf("SEGM1 = 0x%02x\n", pu->out.s.segm[1]);
```

Неке ствари ће бити **једноставније**, као на пример **додељивање вишебитним регистрима**, уместо **маскирања**.

```
275     pu->out.s.segm[0] = 0x20; // Much easier than masking.
```

Неке ствари су ипак **теже** преко битских поља, као на пример **издвајање више бита**.

```
235     printf(
236         "LED = 0x%x\n",
237         pa->out.led3<<3 | pa->out.led2<<2 |
238             pa->out.led1<<1 | pa->out.led0
239     ); // Nightmare. Need concatenation.
```

Овде је потребно извршити **конкатенацију** сваког издвојеног бита, тако што ће се сваки издвојен бит **померити** на одговарајућу позицију и онда их треба **спојити битским или**. Испод је још један пример конкатенације, са вишебитним регистрима.

```
266     printf(
267         "SEGM[1:0] = 0x%04x\n",
268         pa->out.segm[1]<<8 | pa->out.segm[0]
269     ); // Concatenation.
```

Овај проблем се може решити ако структура има целокупне подрегистре као на излисту испод.

```
249     printf("LED = 0x%x\n", pu->out.s.led);
```

Могућност уније да на једном месту има регистра, подрегистре и бите овде долази до пуног изражаја.

Неке ствари могу бити **читљивије**, као на пример промена бита.

```
243     pu->out.b.led1 ^= 1; // Toggle LED1.
245     pu->out.b.led1 = !pu->out.b.led1; // Nicer.
```

Треба повести рачуна да неке ствари неће радити идентично.

```
254     pa->out.led2 = 1; pa->out.led1 = 1; // Could be wrong. Two
        access now.
256     pa->out.led3 = 0; pa->out.led2 = 0; // Also could be wrong.
```

Овде ће у питању бити два уписа у регистре, уместо једног кад се користи спајање маски. Ово може бите **потенцијално опасно** ако је потреба променити више бита истовремено. Такође, овакав појединачан приступ ради измене по **једног бита у битском пољу** може да **смањи перформансе** програма. Из тог разлога, када је потребно рецимо **иницијализовати више бита** метод **спајања маски може бити бољи од битских поља**. Ово је отприлике можда и главна мана битских поља. Наравно, постоје напредне технике како се довијати са овим, а захтевају C++ језик и превазилазе обим ове лекције.

Меморијски распоред битских поља од стране компајлера је **специфичан за ABI** конкретне архитектуре. Да би се добило **преносиво (енгл. Portable)** решење, **боље** је користити **маскирање**.

1.3 Специјалне инструкције за манипулисање битима

.

Како су микроконтролери веома оријетнисани за рад са улазно-излазним уређајима, њихови инструкцијски сетови често имају посебне **улазно-излазне инструкције за манипулисање битима**. Примера ради, **AVR микроконтролери** имају инструкције за **постављање и брисање једног бита** у неком улазно-излазном регистру, као проверу вредности једног бита тј. **прескок наредне инструкције ако је бит постављен тј. обрисан (енгл. Skip if set/clear)**. **PIC микроконтролери** који имају равну меморијску хијерархију **PIC микроконтролери немају меморију логички одвојену од процесора, већ се сви подаци чувају у општенаменским регистрима, па самим тим немају ни инструкције за приступ меморији. Осим општенаменских, у истом регистарском адресном простору постоје и улазно-излазни регистри, па и статусни регистар. Оваква архитектура је веома проста, али зато и веома ограничена.** , омогућују горепоменуте операције над свим регистрима. Ове инструкције омогућују веома брзе операције са битима, пошто им треба 1 такт за постављање односно брисање бита, док им треба 1 до 2 такта за прескокну функцију.

x86 процесори нуде **проширења инструкцијског сета за манипулисање битима** (енгл. Bit Manipulation Instruction Sets). Ове инструкције могу да обављају изузетно напредне операције над битима, укључујући бројање бита, одређивање највећег бита, издвајање битског поља, генерисање маске за маскирање. Једна примена оваквих инструкције је у Huffman-овом компресовању.