Ponovitev C

Povzeto po gradivih za predmet ORS

Podatkovni tipi

- Standardni tipi, ki ste jih vajeni v Cju
  - int, float, char,...
  - nikjer ni jasno definirana dolžina teh spremenljivk
- Ko želimo jasno izraziti dolžino spremenljivke uporabimo
  - int32_t -> 32 bitno celo število
  - uint32_t -> 32 bitni nepredznačeno celo število
  - uint16_t -> 16 bitno celo število
  - int16_t -> 16 bitno nepredznačeno celo število
  - int8_t -> 8 bitno celo število
  - uint6_t -> 8bitno nepredznačeno celo število

Operacije nad biti

- bitni IN → **x & y**
- bitni ALI → **x | y**
- bitni EKSKLUZIVNI ALI → **x ^ y**
- Pomik bitov
  - Desno → **X >> 2**
    - če je X unsigned -> logical shift
    - če je X signed -> arithmetic shift
  - Levo → **X << 2**
- Negacija vseh bitov→ **~x**

Uporaba operacij nad biti

- Primeri: Delo s posameznimi biti

```
unsigned int a=0xA3;

//nastavljanje bita b3
a |= (1 << 3); // a |= 0x8

//brisanje b5
a &= ~(1<< 5); // a &= 0xFFFFFDF

//negacija b4
a ^= (1 << 4); // a ^= 0x10

//testiranje (branje) b2
if( a & (1 << 2) ) // a & 0x4
 //akcija
else
 //akcija2
```

Kazalci

- Kazalec je spremenljivka, ki hrani naslov pomnilniške lokacije
- Deklaracija
  - int *p;
- Naslov spremenljivke
  - Operator &

```
int a=5,*p;
p=&a;
```

- Vrednost pomnilniške lokacije, katere naslov hrani kazalec
  - Operator *
  - Dereferenciranje kazalca

```
int a=5,b=2,*p;
p=&a;
b=*p; // b=5
*p=8; // a=8
```

- Kaj predstavlja spremenljivka int **p ?

Polja

- Deklaracija polja
  - int seznam[10];
  - Rezervira prostor za vse elemente polja
- Vrednost i-tega elementa polja
  - seznam[i]
- Naslov začetka
  - seznam
  - &seznam[0]

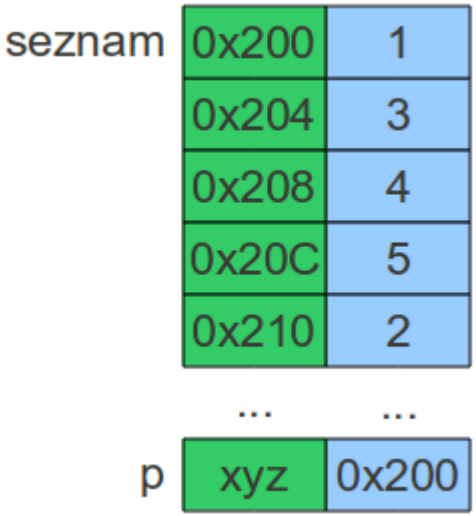
Kazalci in polja

- Definiranje kazalca na polje

```
int seznam[]={1,3,4,5,2};
int *p;
p=seznam; // p=&seznam[0];
```

- i-ti element polja

```
seznam[i]
*(p+i)
p[i]
*(seznam+i)
```



Strukture

- Zbirka večih spremenljivk
- Boljša in lažja organizacija
- Deklaracija strukture

```
struct krog {
    int x;
    int y;
    double obseg;
};
```

- Deklaracija spremenljivke strukturnega tipa

```
struct krog a;
```

- Element strukture

```
double t;

struct krog a,*b;
//...
t=a.obseg;
t=(*b).obseg; // prioriteta . je vecja od *
t=b->obseg;
```

Strukture

- Elementi v strukturi imajo zaporedne naslove

```
struct krog {
    int x;
    int y;
    double obseg;
};
```

Primer:

```
struct krog *a;
a=0x4000000;
```

a.x se nahaja na naslovu 0x40000000, a.y na naslovu 0x40000004, a.obseg pa na 0x40000008

Typedef

- Definiranje novega imena podatkovnega tipa
  - Npr. unsigned int → mojRegister

```
typedef unsigned int mojRegister;
mojRegister spremenljivka;
```

- Večja preglednost in razumljivost/berljivost kode

Volatile

- volatile int a;
- Prevajalnik ne bo poizkušal optimizirati dela s spremenljivko deklarirano z rezervirano besedo volatile
- Uporabno če lahko v vrednost spremenljivke poseže zunanji vir ob nepredvidljivih trenutkih
- Primer: spremenljivka, ki je statusni register vhodno izhodne naprave vgrajenega sistema

Prehod ASM - C

Branje iz naslova

- assembler

```
ldr r0,=0x40020010
ldr r1, [r0]
```

- C

```
volatile uint32_t *a;
a = (uint32_t *) 0x40020010;
//uporabimo *a
```

Pisanje na naslov

- assembler

```
ldr r0,=0x40020C18
ldr r1,=0x55448000
str r1,[r0]
```

- C

```
uint32_t *a;
a = (uint32_t *) 0x40020C18;
*a = 0x55448000;
```

Druge dolžine spremenljivk

- Kako V C-ju zapišemo strh, strb, ldrh,ldrb?
  - Uporabimo uint16_t,int16_t, uint8_t, int8_t

```
ldr r0,=0x40020C18
ldr r1,=0x8000
strh r1,[r0]
```

```
uint16_t *a;
a = (uint16_t *) 0x40020C18;
*a = 0x8000;
```

Uporaba konstant

- Želimo se izogniti pisanju številskih konstant v kodi

```
#define     LED_ON_REGISTER     0x40020C18
#define     LED_1               0x1000
#define     LED_2               0x2000
#define     LED_3               0x4000
#define     LED_4               0x8000

uint16_t *a;
a = (uint16_t *) LED_ON_REGISTER;
*a = LED_4;
// Prižgi ledico 1 in 2
*a = LED_2|LED_1;
```

Uporaba konstant

- Hitrejša oblika zapisa

```
#define     LED_ON_REGISTER     ((uint16_t *) 0x40020C18)
#define     LED_1               0x1000
#define     LED_2               0x2000
#define     LED_3               0x4000
#define     LED_4               0x8000
```

```
*LED_ON_REGISTER = LED_1;
```

Uporaba struktur

- Večina naslovov do katerih bomo dostopali, kaže na registre naprav
- Registri naprave imajo zaporedne naslove

Primer:

```
#define     NAPRAVA   ((uint32_t *) 0x40020C00)
#define     REG1      ((uint32_t *) 0x40020C00)
#define     REG2      ((uint16_t *) 0x40020C04)
#define     REG3      ((uint16_t *) 0x40020C06)
#define     REG4      ((uint32_t *) 0x40020C08)

*reg1=0x8000;     *reg2=0x4000;     *reg3=0x6000;     *reg4=0x5000;
```

Uporaba struktur

- Običajno si te registre želimo dobro organizirati
  - bolj berljiva koda
  - imamo več enakih naprav (z istimi registri), z različnimi začetnimi naslovi
- Uporabimo strukture

```
struct naprava {
        uint32_t a;
        uint16_t b;
        uint16_t c;
        uint32_t d;
};

struct naprava * x;
x = (struct naprava *) NAPRAVA;
```

Uporaba struktur

```
x->a = 0x8000;
x->b = 0x4000;
x->c = 0x6000;
x->d = 0x5000;
```

Kombiniranje definicij in naprav

```
struct naprava_x {
        uint32_t a;
        uint16_t b;
        uint16_t c;
        uint32_t d;
};

#define         NAPRAVA    ((struct naprava_x *) 0x40020C00)

NAPRAVA->a = 0x8000;
NAPRAVA->b = 0x6000;
NAPRAVA->c = 0x4000;
NAPRAVA->d = 0x5000;
```