Varno kodiranje

Električne ali magnetne motnje v računalniškem sistemu lahko povzročijo, da posamezni biti v glavnem pomnilniku (RAM) spontano spremenijo vrednost, kar lahko privede do nepravilnega delovanja računalniškega sistema. Z večanjem gostote zapisa na pomnilniških čipih in nižanjem napajalnih napetostih postaja ta težava vedno bolj pereča. Zagotavljanje visoke odpornosti pomnilniških vezij na motnje iz okolja in preprečevanje napak pri delovanju je še posebej pomembno v sistemih kot so podatkovni strežniki, industrijski krmilniki, sateliti in vesoljske sonde. Integriteta podatkov se v teh sistemih običajno zagotavlja z uporabo naprednih materialov pri gradnji, ki ščitijo podatke pred elektromagnetnim sevanjem in uporabo kodov za odkrivanje in odpravljanje napak.

Hammingov kod

Spekter kodov, ki se uporabljajo pri odpravljanju napak na podatkovnih medijih in v komunikaciji, je zelo širok. Zelo priljubljeni so na primer Reed-Solomonovi in konvolucijski kodi, vendar je izbor optimalnega koda za določeno aplikacijo močno odvisen od okolja, v katerem bo naprava delovala, računskih zmogljivosti strojne opreme, cene in drugih dejavnikov. Danes se na primer zaradi zagotavljanja hitrosti delovanja v pomnilnikih ECC DRAM (ang. Error Checking and Correcting Dynamic Random Access Memory) še vedno pogosto uporablja Hammingov kod H(127, 120), ki je zaradi strukture pomnilnika prilagojen tako, da z 8 varnostnimi biti ščiti 64 podatkovnih bitov. Obstajajo tudi naprednejše tehnologije za odkrivanje in odpravljanje napak, kot je IBM Chipkill, HP Chip spare in Intel SDDC.

Hammingov kod ste podrobno spoznali že na predavanjih (glej tudi [2]). Pri tokratni domači nalogi bomo uporabili družino sistematičnih Hammingovih kodov $H(n, k)$.

CRC

CRC (ang. Cyclic Redundancy Check) je družina cikličnih kodov za odkrivanje napak, ki se uporablja predvsem v digitalnih komunikacijskih napravah za detekcijo napak pri prenosu. Zelo razširjeni so postali zato, ker jih je mogoče enostavno implementirati v strojni opremi in se še posebej dobro obnesejo pri odkrivanju napak, povzročenih zaradi šuma v komunikacijskih kanalih. Za popravljanje napak niso najbolj primerni, zato v primeru napake oddajnik sporočilo običajno pošlje ponovno. Ciklični kod CRC-32 najdemo na primer v standardih Ethernet, SATA in MPEG-2, ciklični kod CRC-16-CCITT pa pri komunikaciji Bluetooth in v pomnilniških karticah SD, CRC-8-CCITT pa v vgrajenih sistemih.

V tej nalogi boste implementirali kod CRC na osnovi standarda CRC-8-CCITT. Parametri standarda so naslednji:

- Polinom: 0x07 $\rightarrow g(p) = p^8 + p^2 + p + 1$,
- Začetna vrednost registra: 0x00,
- Prezrcali podatkovni bajt: Ne,

- XOR nad CRC: Ne,
- Prezrcali CRC: Ne,
- Primer vrednosti CRC za niz ASCII znakov "123456789": 0xF4. Vsak znak vhodnega niza je v tem primeru kodiran z 8 biti, CRC pa je prikazan šestnajstiško.

Delovanje vašega programa lahko preverite tudi preko spletne strani `http://crccalc.com/`. V tem primeru glejte rezultate v vrstici `CRC-8`.

Naloga

V tokratni nalogi boste spoznali uporabo linearnih bločnih kodov za simetrični binarni komunikacijski kanal [1]. Scenarij gre takole:

1. Generirali smo binarni niz (sporočilo) z.
2. Preden sporočilo z pošljemo v komunikacijski kanal, ga obogatimo z m varnostnimi biti. V kanalu namreč lahko pride do napak. Za varovanje smo uporabili enega izmed sistematičnih Hammingovih kodov $H(n, k)$, kjer je $n = 2^m - 1$ in $k = n - m$. Tako zavarovano sporočilo z sedaj imenujemo x.
3. Zavarovano sporočilo x pošljemo po kanalu. Med prenašanjem se lahko njegova vsebina pokvari, kar pomeni, da se določenemu številu bitov obrne vrednost.
4. Na izhodu iz kanala sporočilo prevzameš ti. Tvoja naloga je, da popraviš morebitne napake v njem. Pri tem seveda uporabiš ustrezen kod. Sporočilu, ki pride iz kanala, pravimo y. Ko odkodiraš y, dobiš $\hat{z}$, ki je v najboljšem primeru enak poslanemu sporočilu z.
5. Sporočilo $\hat{z}$ vrneš kot rezultat v vrstičnem binarnem vektorju `izhod`.
6. Nad y izračunaš CRC vrednost po standardu CRC-8-CCITT in jo vrneš kot rezultat v šestnajstiškem zapisu v spremenljivki `crc`.

Napišite funkcijo z imenom `naloga3` v programskem jeziku Octave. Funkcija mora implementirati dekodiranje sporočil y, ki ste jih prejeli iz zašumljenega kanala. Funkcija `naloga3` kot vhodne argumente sprejme vrstični vektor `vhod`, `n` – dolžina kodne zamenjave in `k` – število podatkovnih bitov v kodni zamenjavi. Argumenta `n` in `k` definirata, kateri Hammingov kod je potrebno uporabiti. Vektor `vhod` predstavlja sporočilo y na izhodu iz kanala, njegovi elementi so ničle in enice (vektor tipa `double`). Izhodni argument funkcije je odkodirano sporočilo `izhod`, ki je sestavljeno iz podatkovnih bitov $\hat{z}$. Spremenljivka `izhod` mora biti vrstični vektor tipa `double`. Izhodni argument `crc` predstavlja CRC vrednost izračunano po standardu CRC-8-CCITT nad vektorjem `vhod`. Zapisan naj bo kot število v šestnajstiški obliki (niz).

Prototip funkcije:

```
function [izhod, crc] = naloga3(vhod, n, k)
% Izvedemo dekodiranje binarnega niza vhod, ki je bilo
% zakodirano s Hammingovim kodom H(n,k)
% in poslano po zasumljenem kanalu.
% Nad vhodom izracunamo vrednost crc po standardu CRC-8-CITT.
%
% vhod  - binarni vektor y (vrstica tipa double)
% n     - stevilo bitov v kodni zamenjavi
% k     - stevilo podatkovnih bitov v kodni zamenjavi
% crc   - crc vrednost izracunana po CRC-8-CITT
%         nad vhodnim vektorjem (sestnajstisko)
% izhod - vektor podatkovnih bitov, dekodiranih iz vhoda
```

Testni primeri

Na učilnici se nahaja arhiv `TIS-naloga3.zip`, ki vsebuje tri testne primere sporočil. Primeri so podani v obliki datotek .mat, ki jih naložite v Octave s pomočjo ukaza `load ime_datoteke.mat`. Priloženo imate tudi funkcijo `test_naloga3`, ki jo lahko uporabite za preverjanje pravilnosti rezultatov, ki jih vrača vaša funkcija. Primer klica funkcije za preverjanje: `test_naloga3('primeri',1);`). Pri testiranju vaše funkcije upoštevajte naslednje omejitve:

- pol točke dobite, če se `izhod` ujema z rešitvijo,
- pol točke dobite, če se `crc` ujema z rešitvijo,
- izvajanje funkcije je časovno omejeno na 120 sekund.

Literatura

[1] D.G. Luenberger: Information Science, Princeton University, pogl. 6, 2006.

[2] H. S. Warren: Hacker's Delight, Second Edition, Addison-Wesley, Boston, 2012.