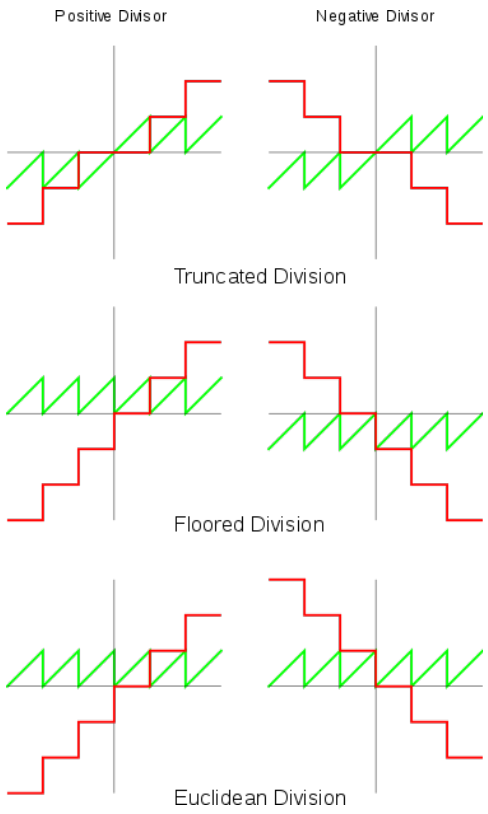
Děličky

- převod do jiného číselného systémy (logaritmický)
- S obnovou zbytku, bez obnovy zbytku, SRT (akcelerační .tabulky)
- Dělení konstantou
- Konvergenční dělení – rozvoj řad, iterační techniky, Cordic

Dělelní (celočíselné)
N/D = Q,R (N=dělenec, numerator; D=dělitel, divisor; Q=podíl, quotient; R=zbytek, remainder)
N=Q*D+R.
Pozor - pro záporné vstupy pak více možnosti chování – viz obr. (např. jazyk C je impl.závislý)
(3/-2 = -1 zb 1, -2 zb -1)



Dělení opakovaným odčítáním

Q=0;R=N;while(R>=D) {R=R-D; Q++;}   //algotimus pro N>0,D>0

..velká časová složitost, Q kroků, vhodné pro spec.případy kdy víme ze Q je malé

Dělení „papír a tužka“

4646 / 3   = 1 5 4 8   ...podíl
16
  14
    26
      2 ..zbytek

4646 / 310 = 1 4
1546
  306

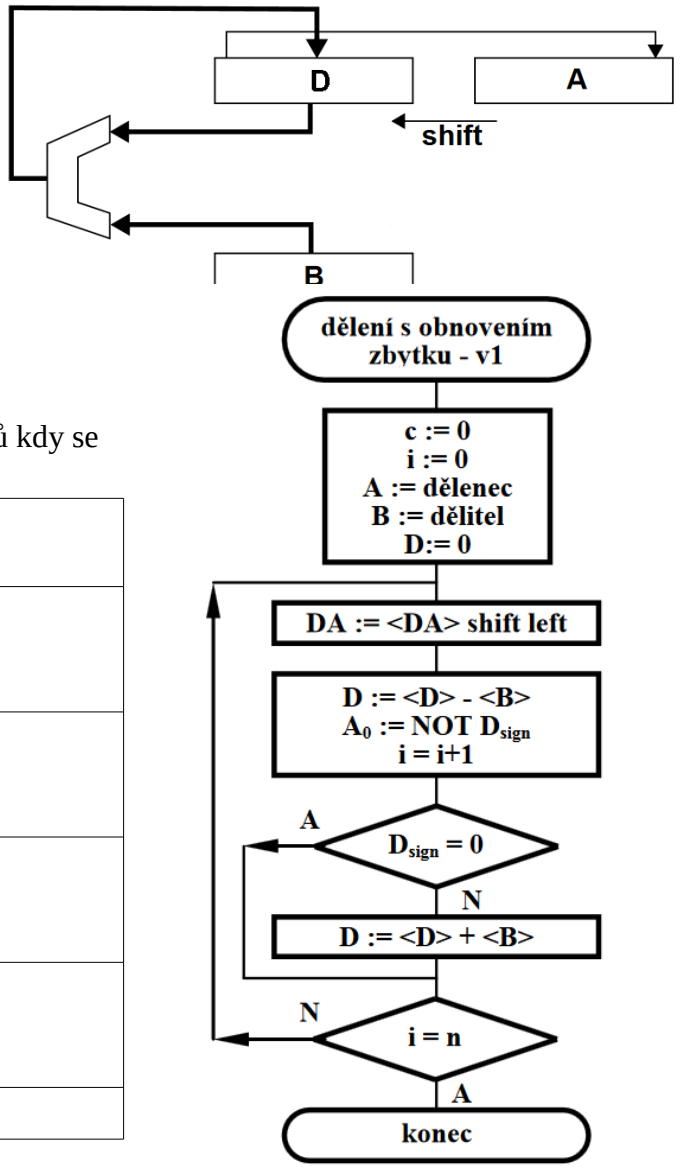
Dělení s obnovou zbytku (restoring division)

Základní verze algoritmu (V1)

- V každém řádu zkoušíme vydělit dílčí zbytek dělence (D) dělitelem (B) (u bininární soustavy pokusně odečítáme (výsledek je 0/1))
- je li rozdíl záporný pak podíl/výsledek v daném řádu je 0 a pokusný odečet opětovně zpět přičteme (odtud „restoring“)
- je li rozdíl kladný pak podíl/výsledek v daném řádu je 1
- provedeme posun(rotaci) DA (do A0 neg.znaménkového bitu = bit podílu/výsledku)
- po N krocích je v D zbytek a v A podíl(výsledek A/B)

(pozn pro snadnější pochopení: zacyklení dvojice DA je jen kvůli lepšímu využití registrů kdy se výsledek dělení ukládá do již prázdné části reg.A)

DA 0000 1110	**B** 0011	**Pozn.** Počáteční hodnoty, počítáme 14 / 3 (1110 / 0011)
0001 1100 1110 1100 0001 1100	0011	Posuv DA D-B (D je záporné) D+B (obnovení, protože záporný výsledek)
0011 1000 0000 1000 0000 1001	0011	Posuv DA D-B (→ D je kladné) nast.A0 = 1
0001 0010 1110 0010 0001 0010	0011	Posuv DA D-B (→ D je záporné) D+B(obnovení protože záporný výsledek)
0010 0100 1111 0100 0010 0100	0011	Posuv DA D-B (→ D je záporné) D+B(obnovení protože záporný výsledek)
		A/B = 0100 = 4, A%B = 0010 = 2



Dělení s obnovou zbytku (restoring division) – modifikace

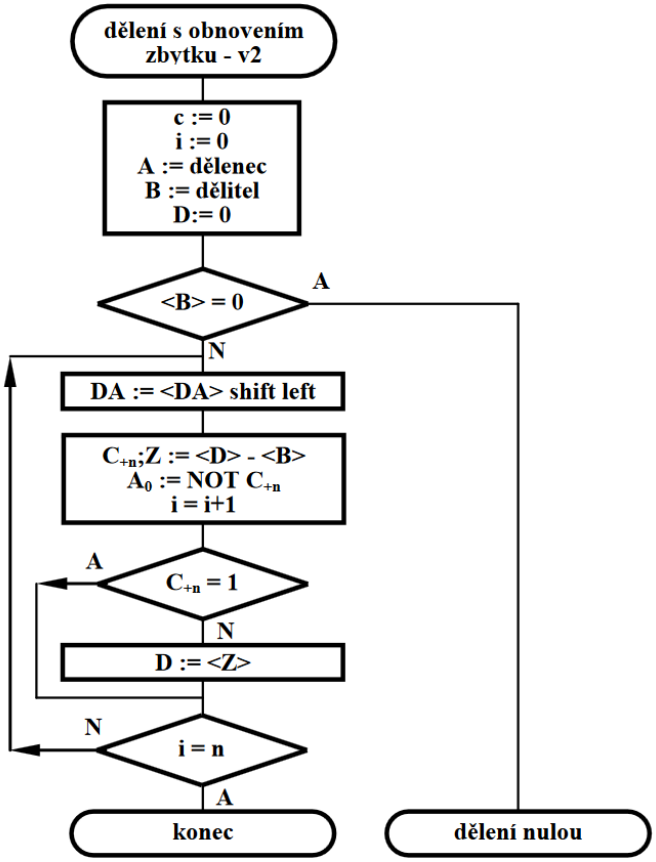
Možnosti urychlení?

- obnovování neřešit výpočtem ale buď zapamatováním původní hodnopty a obnovením a nebo odečtenou hodnotu ukládat mimo a pak ji buď použít nebo ne.

(v obou případech je nutno přidat regitr pro uložení)

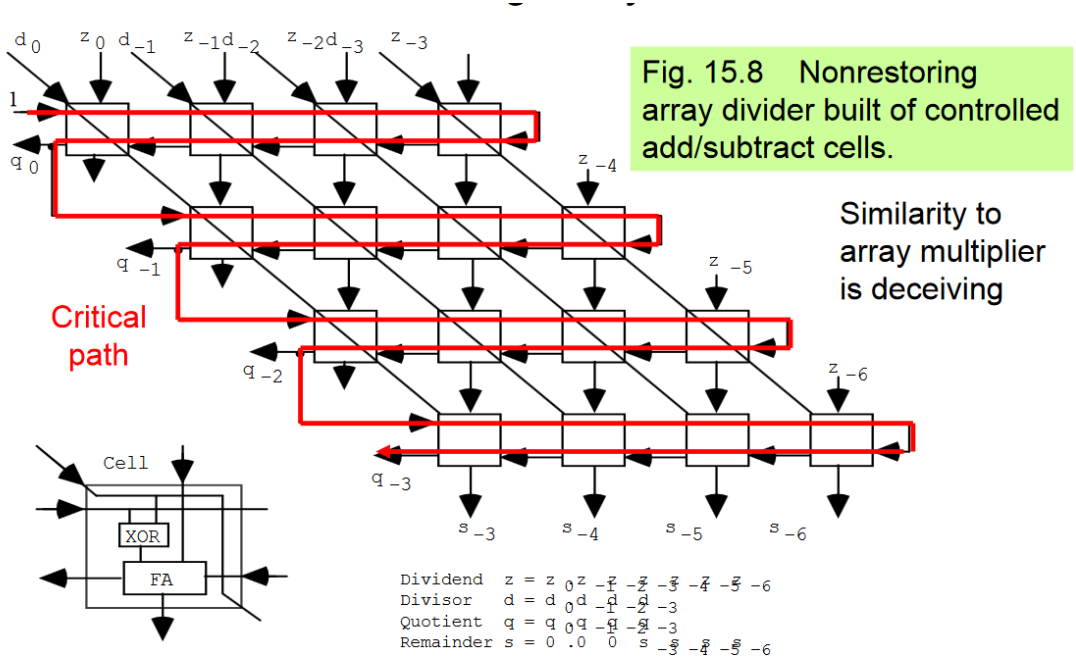
Co algoritmus neřeší?

Dělení nulou - nutno ošetřit zvlášť



Pole pro dělení

obdobně jako násobení lze algoritus dělelení s i bez obnovy zbytku převést do výpočetního pole. Pozor - kritická cesta je zde delší.



Dělení bez obnovy zbytku (non-restoring division)

Algoritmus je upraven tak že neobnovujeme ale počítáme s odečtenou hodnotou.

Analýza D-B (vycházíme z metody s obnovou zbytku):

- je li výsledek kladný tak děláme ((D-B)) – B/2 (-B/2 je pokusné odečtení z dalšího kroku po posunu)

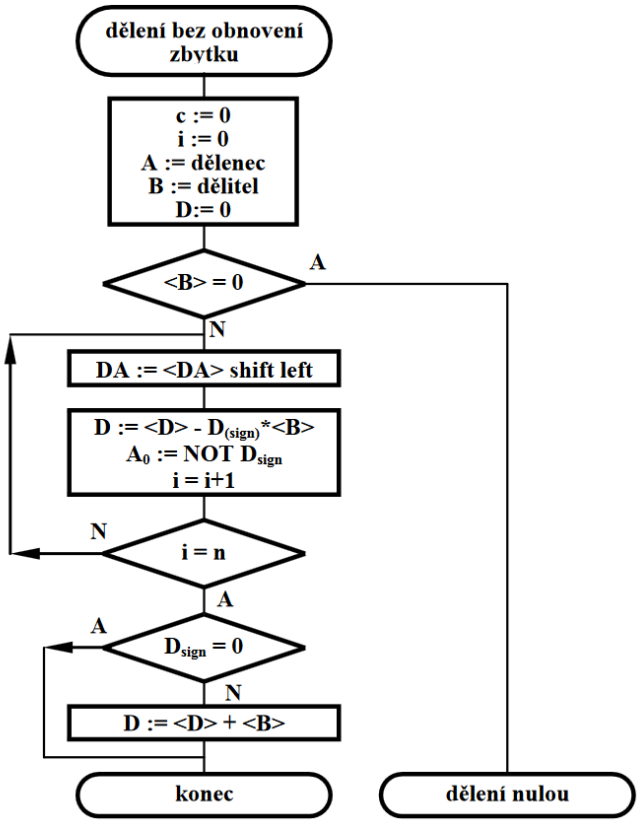
- je li výsledek záporný tak děláme ((D-B)+B) – B/2 = D-B/2 = (D-B) + B/2

tedy D-B je možné nechat/využít a dalším kroku pak dle výsledku B přičítáme nebo odečítáme

(výjimka je posl.krok kde je li výsl.záporný musíme obnovit)

tedy: oproti základnímu algitmu obnovování zbytku není třeba korekce, t.j. algoritmus je (výrazně) rychlejší.

DA 0000 1110	**B** 0011	**Pozn.** Počáteční hodnoty, počítáme 14 / 3 (1110 / 0011)
0001 1100 1110 1100 -	0011	Posuv DA D-B (D je záporné) (zde by u obnovení zbytku byla korekce)
1101 1000 0000 1000 0000 1001	0011	Posuv DA D+B (D je kladné) nast.A0 = 1
0001 0010 1110 0010 -	0011	Posuv DA D-B (→ D je záporné) (zde by u obnovení zbytku byla korekce)
1100 0100 1111 0100 0010 0100	0011	Posuv DA D+B (→ D je záporné) D+B(obnovení protože záporný výsledek v posl.kroku)
		A/B = 0100 = 4, A%B = 0010 = 2



SRT dělení

- Pojmenováno po svých (na sobě nazávislých) tvůrcích (Sweeney, Robertson, Tocher)

Je podobné metodě dělení bez obnovy zbytku ale

- přepokládá normalizovaná čísla A,B
- umožuje implementovat metodu vyšších řádů (Radix 4, 8 ,…)

  (tj. používá se ne jen +/-B, ale napr. -3B až 3B – používá se redundatní zobrazení kvůli možnosti odhadu – lookup tab)
- používá look-up tabulky dělence a delitele k určení (odhadu) výsledku (cifry či případného posuvu)

  (pozn. známá chyba pentia při dělení byla způsbena chybou v lookup tabulkce)
- používá prázdné posuvy (když není třeba odčítat – např. sekvence B -B/2 -B/4 -B/8 = B/8, tj. tři posuvy a odečtení)

SRT – radix 2

- používá cifry {-1;0;1} - odečtení, nic, příčtení
- používá akcelerační look-up tabulky které říkají jak velký krok(=posun bez operace) mohu udělat aniž bych ztratil informaci

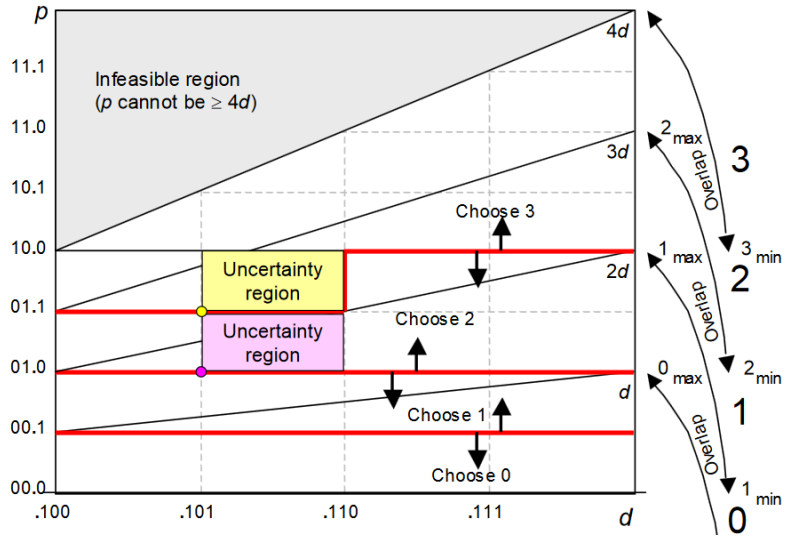
  (tj. pro „skoro stejný“ dělenec a dělitel je možno výsledek rovnou posunout bez nutnosti odečátaní/přičítání)
- průměrná dělka posuvu je 2.67bitu
- je li D>=0.5 pak digit výsledku je 1
- je li D<0.5 pak digit výsledku je -1
- je li -0.5<=D<0.5 pak je digit výsledku 0

SRT – radix 4

- používá typicky cifry {-2 až 2} (mnimálně redundantní) a nebo {-3 až 3} (maximálně redundatní) .

  (max.redundatní je cca o 20% rychlejší a o 50%menší ale potředuje extra čas a prostor pro 3x násobky). Příklad konstrukce lookup tabulky viz obr.

SRT – radix 8,16,...



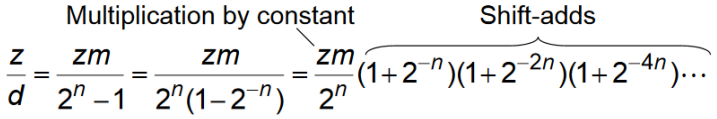
Dělení konstantou

Pokud dělíme číslem předem známým (konstantním) lze dělení nahradit

a/ násobením číslem 1/D (viz násobičky)

b/ posuvy a sčítáním

vychází z premisy že pro každé liché číslo d existuje liché m takové že d * m = 2^n-1 → d = (2^n-1)/m. Potom:



pozn. rozvoj řady do požadované přesnosti

pozn2: počet potřebných sčítání a posuvů je úměrný logaritmu počtu bitů

pozn3: dělení sudým číselem lze převést na dělení lichým postupným dělení 2 (posuv) (20/12 = 10/6 = 5/3)

Příklad: Q=29/5 , tedy z=29, d=5 → pro d=5 je m=3, n=4 (3*5=15=4^2-1)

z*m = 29 * 3 = 29*(2+1) = 29<<1 + 29 = 87 (1*posuv, 1*součet)

* (1+2^-n) = 87 + 87<<4 = 92 (1*posuv, 1*součet) (pozn. pokračujeme neb přepokládáme obecné 16bitové z)

* (1+2^-2n) = 92 + 92 <<8 = 92 (1*posuv, 1*součet)

* (1+2^(-4n)) = 92 + 92<<16 = 92 (1*posuv, 1*součet)

/ (2^n) = 92>>4 = 5 (1*posuv)

Celkem 5*posuv, 4*součet

(srovnejte např, s násobením *0.2 = 0.00110011001100110011001100110011)

Dělení pomocí inverze (násobením)

Obsahuje li výp.jednotka rychlou násobičku lze dělení nahradit násobením inverzní hodnotou

Q= P/D = P * 1/D

Jak (bez dělení;-) zjistit inverzní hodnotu? Konvergenčními metodami

- Rozvojem řad (Taylorovi řady)

- Iteračními technikami (Newton-Raphson)

Lze tak dosáhnout logaritmické konvergence (každým krokem se zdvojnásobuje počet platných bitů) a vystačíme s operacemi sčítání, odčítání, komparace, násobení

Inverze rozvojem řad

Nechť D = 1 + X a současně ½ ≤ D < 1 (toto lze bez újmy na obecnosti zařídit posuvy/normalizací bin.čísla)

Potom na základě Maclaurinových řad (spec. případ Tayloových řad)

g(X) = 1/D = 1/(1+X) = 1 – X + X^2 –X^3 +X^4 - . . .

Protože X = D – 1, lze pro ½ ≤ D < 1 řadu faktorizovat a dostáváme:

1/D = (1 - X)(1 + X^2)(1 + X^4)(1 + X^8)(1 + X^16). . .

Poznámka:

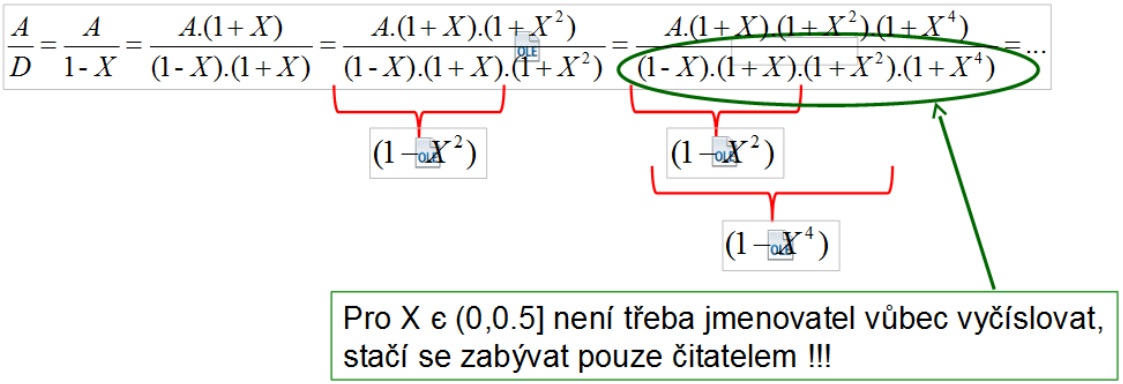
Doplněk ke (1 + X^j) je roven 1 - X^j, protože 2 – (1 + X^j) = 1 – X^j

dále platí (1 + X^j) (1 - X^j) = 1 – X^2j

(tj. např. Známe-li (1-X^2) pak pomocí těchto vztahů zjistíme (1-X^16) atp)

tedy pro výpočet každého kroku je třeba dvou násobení

Jiné odvození:



Inverze rozvojem řad

Příklad (implementace IBM360):

Výpočet 1/D s přesností na 32-bitů následujícími kroky:

$(1 – X)(1 + X^2)(1 + X^4)$ získáno tabulkou (look-up table)

$1 - X^8 = [(1 – X)(1 + X^2)(1 + X^4)](1 + X)$

$1 + X^8$ je „dvojkový doplněk“ $1 - X^8$

$1 – X^{16} = (1 + X^8)(1 – X^8)$

$1 + X^{16}$ „dvojkový doplněk“ of $1 – X^{16}$

$1 – X^{32} = (1 + X^{16})(1 – X^{16})$

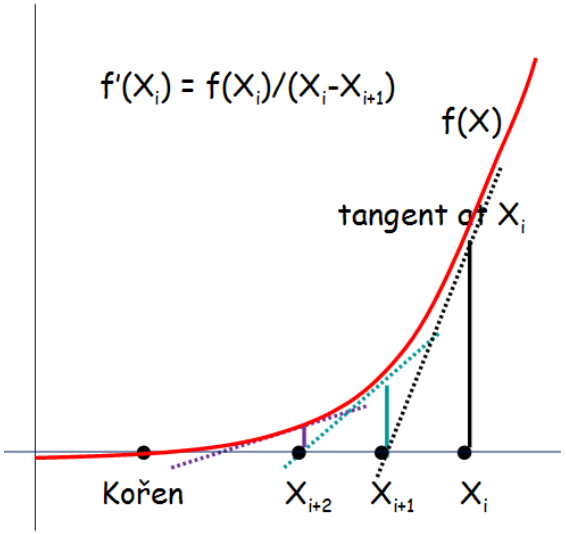
$1 + X^{32}$ „dvojkový doplněk“ $1 – X^{32}$

pozn: Lze dokázat že lookup tabulka nemusí mít plnou přesnost X, ale že vzniklá chyha se s dalšími kroky snižuje. Zmíněný příklad by potřeboval 1024 položkovou tabulku, ale používal tabulku s 256 položkami nelineárně rozvrstvenými dle počtu prvních jedničkových bitů.

Aditivní Iterace

musí být založena na spojité a diverencovatelné fci ve formě f(X)=0

lze užít pro širokou třídu úloh – dělení, mocniny/odmocniny, log, exp



Iterace Newton-Raphson – dělení

hledáme kořen Q=P/D (nebo jen 1/D)

Řešení se hledá pomocí iteračního vzorce (metoda tečen): $X_{i+1} = X_i - f(X_i)/f'(X_i)$

Metoda Newton-Raphson upravená pro výpočet převrácené hodnoty vychází ze vztahu
f(X) = 1/X – D = 0
Kořenem rovnice je X (= 1/D)
Protože f'(X) = -(1/X)^2, dává rekurentní formuli

Xi+1 = Xi.(2 – Xi.D)

Vybíráme X0 takové, aby platilo 0 < X0 < 2/D
Pro jednu iteraci jsou třeba dvě násobení, tedy pro n-bitové operandy potřebueme 2log(n) násobení

Příklad: Nalezněte 1/D kde D = 0.75 (dec) = 0.1100 (bin) :

Krok	Dekadicky	Chyba	Binárně	Chyba
X0=	1	0.333..	1.0000	<2^-1
X1=	1(2 – 0.75) = 1.25	0.0833..	1.0100	<2^-2
X2=	1.25(2 – 1.25*0.75) = 1.328125	0.005208	1.01010100	<2^-4
X3=	. . . = 1.333313	0.000021	1.0101010101010100	<2^-8

Iterace Newton-Raphson – dělení - rychlost konvergence

Metoda konverguje kvadraticky ($\varepsilon_{i+1} \le |\varepsilon_i|^2$) pro D<1

$X_{i+1} = X_i.(2 – X_i.D)$ a $\varepsilon_i = 1/D – X_i$ ...Potom...

$X_i = 1/D - \varepsilon_i = (1 – D.\varepsilon_i)/D$ a $\varepsilon_{i+1} = 1/D – X_{i+1}$ ...Z toho vyplývá...

$\varepsilon_{i+1} = 1/D – [X_i.(2–X_i.D)] = [1-2.D.X_i+(D.X_i)^2]/D$ ...Substitucí pro X_i …

$\varepsilon_{i+1} = [1 – 2.D.((1–D.\varepsilon_i)/D) + (1–D.\varepsilon_i)^2]/D = D.\varepsilon_i^2$ ...takze pak pro D<1 platí že

$\varepsilon_{i+1} \le |\varepsilon_i|^2$

Startovací hodnoty

Pro D z intervalu [½, 1) může být dobrá výchozí hodnota X0 = 1.5 protože omezuje počáteční chybu na maximálně 0.5

Lepší aproximací může být výraz $X0 = 4(\sqrt{3} – 1) - 2D = 2.9282 – 2D$ (max. chyba cca 0.1)

V obecnosti: dobrým počátečním odhadem můžeme rychlost konvergence značně urychlit (viz konvergence) → použití lepšího odhadu nebo look-up table.

(např. dělení 32 bitů pořebuje 5 kroků, pokud poč.odhad bude přesný jen na 4 bity potřebujeme jen 3 kroky)

Další aproximace: X0=48/17-32/17*D (max.chyba cca 0.06) – pro IEEE čísla v jednoducé přesnosti pak stačí 3 následné iterace, pro dvojnásobnou přesnost 4 iterace (přesněji 10 násobení, 9sčítání, 2 posuvy)

Další zrychlení

Zejména pro operandy s vyšším počtem bitů lze v počátečních fázích mat.operace dělat s nižší přesností.

Např. Dělení 64/64bitů: 256*8 bitů lookup-table + 9bit násobička + 17bit násobička + 33bit násobička

Cordic - dělení

- CORDIC je iterační algoritmus používaný pro výpočet některých matematických funkcí (původně goniometrickýh ale myšlenku lze rozšířit na výpočet i dalších).

- je vhodný pro implementaci v číslicových systémech neboť použící operace rotace, sčítání/odčítání/porovnání a look-up tabulky

- Zkratka CORDIC znamená COordinate Rotation DIgital Computer.

Dělení

základem je přepis a=x/y do tvaru x-y*a=0 kterou upravíme bitovým rozkladem a:

$$x - y * \sum_{i=1}^{B} a_i * 2^{-i} = 0 \qquad x - \sum_{i=1}^{B} a_i * (y * 2^{-i}) = 0$$

konečná podoba ukazuje ze koeficient a může být (za splnění jistých podmínek) „odhadován“ bit po bitu tak že se snažíme aby výslený výraz se blížil nule. Jesliže je aktuální zbytek kladný je příslušný bit <a> nastaven a je odečítáno a naopak.

```
divideCordic(x,y)
{
a=0;
for (i=1; i=<R; i++)
  {
  if (x > 0)
    x = x - y*2(^-i);  //tj. Odečet posunutého Posun y
    a = a + 2^(-i);    //set ai
  else
    x = x + y*2(^-i);  //tj. Odečet posunutého Posun y
    //a = a - 2^(-i);  //clear ai
  }
return(a)
}
```

pozn. v konečném výsledku je metoda podobná dělení bez obnovy zbytku, umožnuje lépe pracovat se znaménkovými vstupy.