*Zastosowania procesorów sygnałowych*

# *SYSTEMY LICZBOWE*

Opracowanie: Grzegorz Szwoch

Politechnika Gdańska, Katedra Systemów Multimedialnych

# *1 + 1 = 10*

- Procesor wykonuje jedynie operacje na ciągach bitów:
  10011001100110 + 100110011001101 = 111001100110011
- Procesor nie interpretuje co to znaczy „10011001100110”.
- System liczbowy – sposób reprezentacji liczb za pomocą sekwencji zer i jedynek, tak aby wynik operacji był poprawny.
- Najważniejsze systemy liczbowe:
  - stałoprzecinkowe
    - – całkowite
    - – ułamkowe (Q)
  - zmiennoprzecinkowe

# *Liczby stałoprzecinkowe całkowite*

Reprezentacja stałoprzecinkowa liczb całkowitych bez znaku (*unsigned integer*) - system dwójkowy (binarny)

- pierwszy, najstarszy bit (MSB): $2^{N-1}$
- *i*-ty bit: $2^{i-1}$
- ostatni, najmłodszy bit (LSB): 1

| $2^{15}$ | $2^{14}$ | $2^{13}$ | $2^{12}$ | $2^{11}$ | $2^{10}$ | $2^{9}$ | $2^{8}$ | $2^{7}$ | $2^{6}$ | $2^{5}$ | $2^{4}$ | $2^{3}$ | $2^{2}$ | $2^{1}$ | $2^{0}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 32768 | 16384 | 8192 | 4096 | 2048 | 1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| | | 8192 | | | 1024 | 512 | | | 64 | 32 | | | 4 | 2 | |

= 9830

# *Liczby stałoprzecinkowe ze znakiem*

Zapis stałoprzecinkowy liczb całkowitych ze znakiem (*signed integer*):

- najstarszy bit – bit znaku, 0: liczba dodatnia, 1: ujemna
- pozostałe bity: moduł liczby zapisany w kodzie uzupełnień do 2 (U2, *two's complement*):
  - negujemy wszystkie bity (0 → 1, 1 → 0),
  - dodajemy 1.

Przykłady dla liczb 8 bitowych:

(-1) 0000 0001 → 1111 1110 → 1111 1111

(-123) 0111 1011 → 1000 0100 → 1000 0101

# *Cechy liczb stałoprzecinkowych całkowitych*

- Rozdzielczość – najmniejsza możliwa różnica między liczbami, wynosi $2^0 = 1$. A więc nie można zapisywać ułamków!
- Zakres – najmniejsza i największa liczba – zależy od liczby bitów *N* przeznaczonych na zapis liczby:
  - liczby bez znaku: od 0 do $(2^N-1)$
  - liczby ze znakiem: od $-2^{N-1}$ do $(2^{N-1} – 1)$
- Jeżeli nie potrzebujemy liczb ujemnych, użycie liczb bez znaku zwiększa nam zakres dwukrotnie.

# *Typy stałoprzecinkowe w języku C*

- *int* – liczba bitów zależy od długości rejestrów procesora, na DSP: 16 lub 32 bity.
- *char* – 8 bitów, bajt (*byte*)
- *short* – 16 bitów, słowo (*word*)
- *long* – 32 bity, podwójne słowo (*double word*)
- *long long* – na DSP zależy od długości akumulatora, zwykle 40 bitów lub 64 bity

Każdy typ występuje w dwóch wersjach:

- *unsigned* – bez znaku, np. *unsigned int*
- *signed* – ze znakiem (domyślnie), np. *signed long* (= *long*)

# *Ułożenie bajtów w pamięci*

Jeżeli liczba zajmuje np. 4 bajty, w jakiej kolejności są one zapisywane w pamięci?

- Od najstarszego do najmłodszego bajtu (*big endian*)
  – niektóre procesory, transmisja sieciowa.
- Od najmłodszego do najstarszego bajtu (*little endian*)
  – wszystkie procesory Intela i (domyślnie) ARM.
- W procesorze sygnałowym zazwyczaj można wybrać sposób zapisu (przy kompilacji kodu).
- Najczęściej stosuje się na DSP tryb *little endian*.

# *Przekroczenie zakresu*

Co się stanie jeżeli wynik operacji nie zmieści się w zadanej liczbie bitów? Następuje przekroczenie zakresu (*range overflow*).

Przykład dla dodawania liczb 16-bitowych.

- Liczby bez znaku (maksymalna wartość: 65535)
  (65530 + 10) → [1] 0000 0000 0000 0100 → 4
  - „nadmiarowe bity” zostają obcięte,
  - wynik jest resztą z dzielenia przez $2^{16}$.
- Liczby ze znakiem (maksymalna wartość: 32767)
  (32760 + 10) → 1000 0000 0000 0010 → -32766 (!!!)
  - jedynka „przeskakuje” na bit znaku,
  - następuje „zawinięcie” zakresu, dostajemy ujemny wynik.

# *Jak się chronić przed przepełnieniem zakresu?*

- Akumulatory procesorów sygnałowych mają nadmiarowe bity (np. 40) – zapobiega przepełnieniu dla pośrednich wyników.
- Zapisując wynik do zmiennej musimy wybrać „wystarczająco pojemny” typ liczbowy, np. *long*.
- Na DSP często stosuje się skalowanie liczb (np. podzielenie przez 2) aby zapobiec przepełnieniu – należy odpowiednio przeskalować również końcowy wynik.

# *Reprezentacja liczb ułamkowych*

- No dobrze, czy w takim razie nie można zapisać liczby ułamkowej, np. 0,3?
- Przypominamy, system liczbowy określa interpretację bitów.
- Może więc interpretować je w taki sposób?

| 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| znak | $2^{-1}$ | $2^{-2}$ | $2^{-3}$ | $2^{-4}$ | $2^{-5}$ | $2^{-6}$ | $2^{-7}$ | $2^{-8}$ | $2^{-9}$ | $2^{-10}$ | $2^{-11}$ | $2^{-12}$ | $2^{-13}$ | $2^{-14}$ | $2^{-15}$ |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| + | | $2^{-2}$ | | | $2^{-5}$ | $2^{-6}$ | | | $2^{-9}$ | $2^{-10}$ | | | $2^{-13}$ | $2^{-14}$ | |

$= 0{,}29998 \approx 0{,}3$

# *Reprezentacja Q15*

- Reprezentacja liczb ułamkowych w ten sposób, jako liczby całkowitej, 16-bitowej, ze znakiem, nazywa się zapisem Q15 - 1 bit znaku, 0 bitów części całkowitej, 15 bitów części ułamkowej.
- Rozdzielczość (minimalna różnica między liczbami) jest równa $2^{-15} = 0{,}000030517578125$.
- Zakres: od -1 do $(1-2^{-15})$, czyli do 0,999969482421875.
- Liczba +1 jest już poza zakresem!
- Wszelkie uwagi dotyczące przepełnienia zakresu są nadal aktualne.

# *Reprezentacja QM.N*

- Możemy uogólnić ten zapis do formy QM.N: 1 bit znaku, *M* bitów części całkowitej, *N* bitów części ułamkowej.
- Rozdzielczość: $2^{-N}$.
- Zakres: od $-2^{M}$ do $(2^{M} – 2^{-N})$.
- Przykład: 1 bit cz. całkowitej i 14 bitów cz. ułamkowej daje nam zapis Q1.14. Zakres: od -2 do „prawie” 2.
- Przy ustalonym sposobie zapisu QM.N, przecinek dziesiętny pozostaje zawsze na tej samej pozycji.
  Stąd termin „zapis stałoprzecinkowy” (*fixed point*).

# *Zapis Q na typach stałoprzecinkowych*

- Nie ma w C specjalnego typu liczbowego dla liczb Q.
- Zapis Q jest to tylko interpretacja bitów.
- Aby zapisać liczbę Q, musimy interpretować bity tak, jakby były zwykłymi liczbami całkowitymi, np.:
  0,3 → 10011001100110 → 9830
- Co sprowadza się do zależności (dla QM.N):
  - z liczby ułamkowej *x* na liczbę całkowitą *q*:
    $q = x \cdot 2^N$ (z zaokrągleniem)
  - z liczby całkowitej *q* na liczbę ułamkową *x*:
    $x = q / 2^N$

# *Zapis Q na typach stałoprzecinkowych*

Przykład dla zapisu Q15 ($2^N = 2^{15} = 32768$):

0,3 → 0,3 * 32768 → 9830,4 → 9830

0,6 → 0,6 * 32768 → 19660,8 → 19661

Obliczenie wyrażenia (0,3 + 0,6):

9830 + 19661 = 29491

Konwersja na liczbę ułamkową:

29491 → 29491 / 32768 → 0,89999

(przypominamy: rozdzielczość jest równa 0,00003)

# *Kwantyzacja*

- Przy konwersji musimy zaokrąglać wynik do najbliższej dostępnej liczby:
  9830,4 → 9830  19660,8 → 19661
- Jest to kwantyzacja liczb.
- Błąd kwantyzacji: różnica między wartościami po kwantyzacji a oryginalnymi. Ma postać szumu.
- Przy zapisie liczb stałoprzecinkowych na procesorach, kwantyzacja ma bardzo duży wpływ na dokładność obliczeń.
- Np. dla filtrów cyfrowych typu IIR może spowodować niestabilność prawidłowo zaprojektowanego filtru.

# *Mnożenie liczb QM.N*

Dla uproszczenia będziemy rozpatrywali liczby Q15.

Jak obliczyć (0,3 · 0,6)?

- Konwersja jak poprzednio.
- 9830 · 19661 = 193267630
- Mnożenie dwóch liczb Q15 daje wynik w formacie Q30!
- Mnożenie liczb QM.N daje Q(2M).(2N)
- Konwersja wyniku:
  193267630 → 193267630 / $2^{30}$ → 0,1799945

# *Dzielenie liczb QM.N*

- Dzielenie jest operacją bardzo wolną, szczególnie w zapisie stałoprzecinkowym. Nie da się go przeprowadzić wprost tak jak mnożenie.
- Najczęściej stosowany algorytm: numeryczne wyznaczenie odwrotności liczby, a potem mnożenie.
- Ale jest wyjątek: dzielenie przez potęgi dwójki ($2^k$) może być szybko wykonane przez przesunięcie bitowe w prawo o *k* miejsc. W języku C: operator >>.
- Analogicznie można wykonać mnożenie przez $2^k$, przesuwając bity w lewo (<<) o *k* miejsc (z prawej wchodzą zera).
- Należy to stosować w programach, pisząc np. „x >> 1” zamiast „x / 2”.

# *Jeszcze o mnożeniu*

Mamy wynik mnożenia liczb Q15 w formacie Q30. Jak z tego odzyskać format Q15?

- Najpierw podzielić przez $2^{15}$, przesuwając w prawo o 15 b.
- Następnie odrzucić starsze bity – pozostawić młodszych 16 b.
- W ten sposób wykonuje się zaokrąglenie w dół.
- Aby zaokrąglić do najbliższej liczby, trzeba na początku dodać $2^{14}$ (jedynka na najstarszym z odrzucanych bitów), a potem przesunąć i obciąć wynik.

193267630 (Q30) → (193267630 + (1<<14)) → 193284014 →

→ (>> 15) → 5898 (Q15) → 5898 /32768 → 0.17999267

# *Niedopełnienie*

- Co się stanie jeżeli po przesunięciu wyniku mnożenia zostaną same zera?
- (0,003 × 0,002) → 98 × 66 = 6468 (Q30)
- 6468 >> 15 = 0 (Q15) !!!
- Wynik (0,003 × 0,002) = 0,00006 jest zbyt małą liczbą.
- Powstaje niedopełnienie (*underflow*), powodujące wyzerowanie wyniku.
- Wszystkie kolejne mnożenia (np. w filtrze) też dadzą zero!
- Przed niedopełnieniem próbujemy się chronić stosując dłuższe typy liczbowe i reorganizując kolejność operacji.

# *Mnożenie Q15 w języku C*

Jak poprawnie zapisać wynik mnożenia liczb Q15 w języku C?

```
short a = 9830;
short b = 19661;
/* short y = ??? */
```

W ten sposób się nie da – wynik nie zmieści się na 16 bitach (*short*), starsze bity zostaną obcięte:

```
short y = a * b;  // 1966
```

Może tak? Typ *long* ma 32 bity, więc wynik powinien się zmieścić. Niestety, wciąż ten sam błędny wynik:

```
long y = a * b;  // 1966
```

# *Mnożenie Q15 w języku C*

Jak kompilator C wykonuje tę instrukcję?

```
long y = a * b;  // a, b: typ short
```

- Najpierw oblicza wyrażenie po prawej stronie: (a * b).
- Typ wyniku jest zgodny z „największym” typem argumentu. Oba argumenty są typu *short*, więc wynik też ma typ *short*.
- Starsze bity zostają obcięte (nie mieszczą się w *short*).
- Wynik (błędny) jest zapisywany do tego, co jest po lewej stronie (typ *long*).

# *Rzutowanie typów w języku C*

- Aby uzyskać prawidłowy wynik, musimy „promować” argument do dłuższego typu.
- Rzutowanie (*cast*) w języku C wykonuje się podając nowy typ w nawiasie przed wyrażeniem lub zmienną:

```
long y = (long)a * b;  // 193267630
```

- Teraz jeden z argumentów jest typu *long*, więc wynik będzie też zapisany jako *long*.
- Rzutowanie stałej liczbowej (literału) można wykonać stawiając literę L (od *long*) po liczbie:

```
long y = a * 19661L;
```

# *Mnożenie liczb Q15*

A zatem, aby wykonać mnożenie (0,3 · 0,6) i zapisać wynik w formacie Q15 (jako *short*), musimy wykonać straszną komendę:

```
short y = (short)(((long)a * b) >> 15);
```

albo z zaokrągleniem:

```
short y = (short)((((long)a * b) + (1<<14)) >> 15);
```

Na szczęście, procesory DSP dają nam zwykle instrukcje-skróty. Np. na procesorze C5535 to samo wykonamy instrukcją:

```
short y = _smpy(a, b);
```

# *Tryb nasycenia*

- Przepełnienie zakresu skutkuje znacznym przekłamaniem wyniku (np. 32760 + 10 → -32766).
- Procesory DSP mają arytmetykę nasycenia (*saturation arithmetic*). Nasycenie polega na obcięciu liczb przekraczających zakres do wartości granicznej.
  32760 + 10 → 32767
- Wynik wciąż jest błędny, ale błąd jest ograniczany.
- Na DSP C5535 mamy instrukcje wykonujące operacje z obsługą trybu nasycenia, o nazwach zaczynających się od _s: _sadd (+), _ssub (–), _smpy (×), _sround (zaokrąglenie), itp.

# *Zapis zmiennoprzecinkowy*

- Zapis zmiennoprzecinkowy (*floating point*) pozwala znacznie zwiększyć dokładność obliczeń w stosunku do zapisu stałoprzecinkowego.
- Procesor musi posiadać jednostkę do wykonywania obliczeń na takich liczbach (FPU, *floating point unit*). Taki procesor nazywamy procesorem zmiennoprzecinkowym.
- Procesor C5535 z projektu ZPS, tak jak wiele innych DSP, nie posiada takiej jednostki – jest procesorem stałoprzecinkowym.
- Stałoprzecinkowe DSP są nadal używane, nie są „przestarzałe” w stosunku do zmiennoprzecinkowych DSP.

# *Zapis zmiennoprzecinkowy*

Każda liczba jest reprezentowana przez:

- S – znak (0 lub 1),
- M – mantysę,
- E – wykładnik (*exponent*),
- b – bazę (zwykle b = 2).

$$(-1)^{s} \cdot 1{,}M \cdot b^{(E-127)}$$

Przykład:

$3{,}14159265359 = 1{,}570796326795 \cdot 2^{1}$

S = 0, M = 570796326795, E = 128, b = 2

# *Typy zmiennoprzecinkowe*

Typy zmiennoprzecinkowe zdefiniowano w standardzie IEEE 754.

- *float* – typ pojedynczej precyzji
  - 32 bity: 1 b znaku, 23 b mantysy, 8 b wykładnika
  - 7 znaczących miejsc po przecinku
  - zakres: od $\pm 3{,}4 \cdot 10^{-38}$ do $\pm 3{,}4 \cdot 10^{38}$
- *double* – typ podwójnej precyzji
  - 64 bity: 1 b znaku, 52 b mantysy, 11 b wykładnika
  - 15 znaczących miejsc po przecinku
  - zakres: od $\pm 1{,}7 \cdot 10^{-308}$ do $\pm 1{,}7 \cdot 10^{308}$

Rozdzielczość jest zmienna, zależy od wartości liczby.

# *Typy zmiennoprzecinkowe*

Według IEEE 754, zmienne *float* i *double* mogą przyjmować specjalne wartości:

- *Inf* – plus nieskończoność, np. wynik (1.0 / 0.0)
- *-Inf* – minus nieskończoność, np. wynik (-1.0 / 0.0)
- *NaN* – nieokreślony wynik (*not a number*), np. (0.0 / 0.0)
- -0.0 (ujemne zero) – powinno być traktowane tak jak zwykłe zero.

# *Cechy zapisu zmiennoprzecinkowego*

- Ze względu na bardzo szeroki zakres, praktycznie nie ma ryzyka wystąpienia przepełnienia.
- Ryzyko niedopełnienia jest bardzo małe (mniejsze dla typu *double*).
- Nie trzeba stosować żadnych specjalnych zapisów w C:

```
double y = 0.3 * 0.6;
```

  Dostajemy wynik 0,18, a nie 0,1799945 jak dla zapisu stałoprzecinkowego.

- Działania na liczbach zmiennoprzecinkowych wymagają znacznie więcej cykli procesora i często więcej pamięci.

# *Porównywanie liczb zmiennoprzecinkowych*

Jeden kruczek: pamiętajmy, że liczby zmiennoprzecinkowe są zapisywane ze skończoną precyzją.

Takie wyrażenie może nie zadziałać:

```
if (a / 2 == 2.5) {   // a jest typu double
```

Wynik dzielenia może być równy nie 2,5, a np. 2,500000000001.

Dlatego musimy sprawdzać czy liczby są wystarczająco bliskie, np. różnica mniejsza niż $10^{-8}$:

```
if (abs(a / 2 - 2.5) < 1e-8) {
```

# *Rzutowanie liczb zmiennoprzecinkowych w C*

Przypomnienie: kompilator C oblicza najpierw wyrażenie po prawej stronie.

```
short a = 5;
double b = a / 2;
```

Wynik będzie *b* = 2, ponieważ najpierw zostanie wykonane dzielenie na liczbach całkowitych, a dopiero potem wynik będzie przypisany do zmiennej typu *double*.

Prawidłowe rzutowanie (daje wynik 2.5):

```
short a = 5;
double b = (double)a / 2;
double c = a / 2.0;
float d = (float)a / 2;
float e = a / 2.0f;
```

# *Zalety procesorów zmiennoprzecinkowych*

- Duża dokładność obliczeń.
- Duża dynamika – bardzo mały szum kwantyzacji.
- Wygodne programowanie – nie trzeba konwertować liczb na zapis Q.
- Łatwiejsze i dokładniejsze wykonywanie operacji matematycznych (pierwiastek, logarytm, funkcje trygonometryczne, itp.).

Np. filtry cyfrowe IIR: szum kwantyzacji jest mały, zredukowane jest ryzyko niestabilności filtru, wyniki są dokładniejsze.

# *Wady procesorów zmiennoprzecinkowych*

- Dłuższe obliczenia, więcej cykli zużytych na wykonanie operacji (nawet prostego mnożenia).
- Większe zużycie pamięci, zwłaszcza gdy stosujemy liczby typu *double*.
- Większe zużycie energii.
- Wyższy (znacznie) koszt procesora.

# *Praktyczne uwagi*

Kiedy użyjemy zmiennoprzecinkowego DSP?

- gdy potrzebujemy wysokiej precyzji,
- gdy liczy się duża dynamika, np. przetwarzanie dźwięku z rozdzielczością 16 lub 32 bity – mniejsze szumy,
- gdy nas na to stać.

Kiedy użyjemy stałoprzecinkowego DSP?

- gdy istotna jest cena procesora i zużycie energii,
- gdy wykonujemy tylko proste operacje DSP (FFT, filtry),
- gdy sygnał wejściowy nie ma dużej dynamiki (np. sygnał z przetwornika A/C o rozdzielczości 12 bitów).