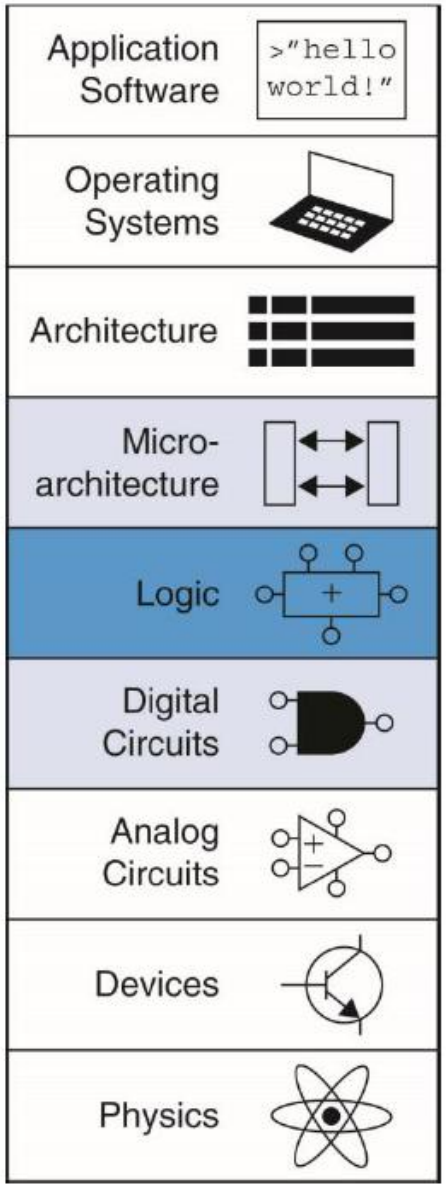
КАРХ: Тема_6: Изграждащи цифрови блокове

Въведение.

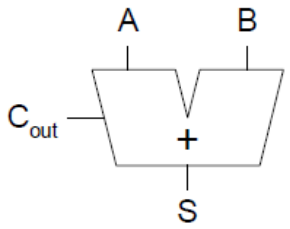
- **Изграждащи цифрови блокове (Digital building blocks)**
  - Логически елементи, мултиплексори, декодери, регистри, аритметични схеми
  - **Изграждащите блокове се подчиняват на трите основни принципа -hierarchy, modularity и regularity**
  - Hierarchy на простите компоненти;
  - Добре дефинирани интерфейси и функции;
  - Стандартните структури са лесно разширими (с мултипликация).



КАРХ: Тема_6: Изграждащи цифрови блокове

1-Bit Суматори (1-Bit Adders).

Half Adder

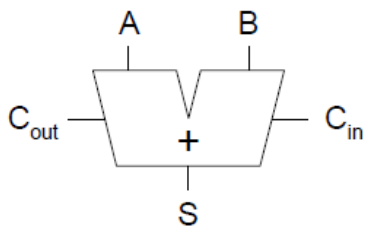


A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

S =

C_{out} =

Full Adder



C_{in}	A	B	C_{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

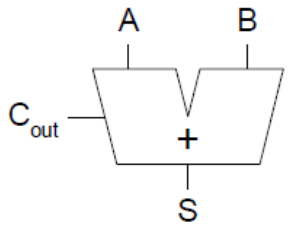
S =

C_{out} =

КАРХ: Тема_6: Изграждащи цифрови блокове

1-Bit Суматори (1-Bit Adders).

Half Adder

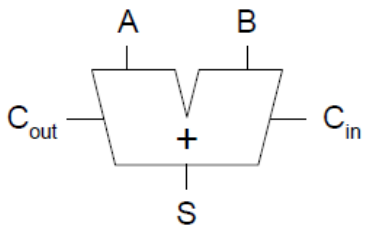


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

S =

C_{out} =

Full Adder



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

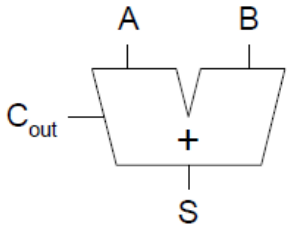
S =

C_{out} =

КАРХ: Тема_6: Изграждащи цифрови блокове

1-Bit Суматори (1-Bit Adders).

Half Adder

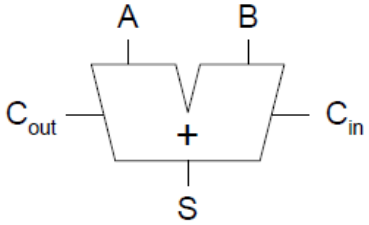


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$

$C_{out} = AB$

Full Adder



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

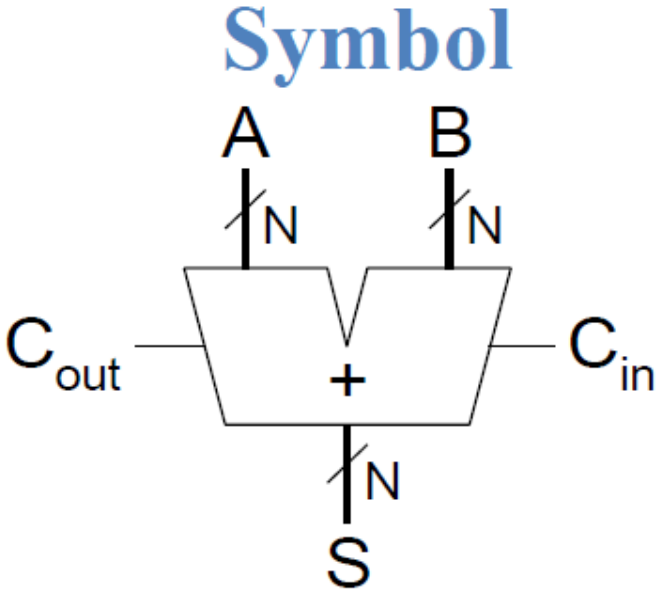
$S = A \oplus B \oplus C_{in}$

$C_{out} = AB + AC_{in} + BC_{in}$

КАРХ: Тема_6: Изграждащи цифрови блокове

Много-битови Суматори (Multibit Adders (CPAs)).

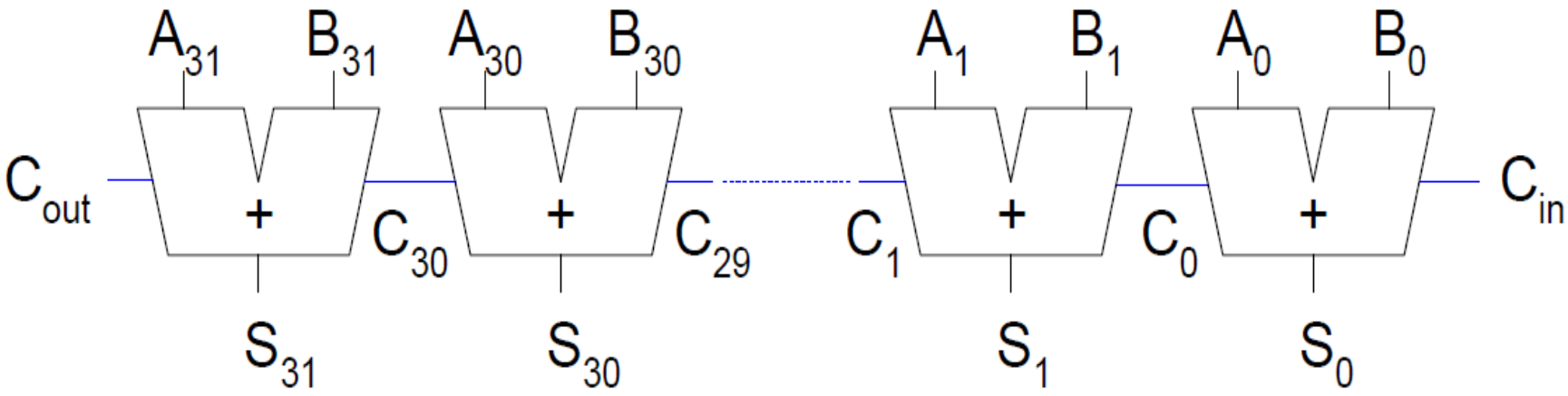
- Видове много-битови суматори според начина на формиране на преноса (carry) (carry propagate adders (CPAs)):
  - Ripple-carry (бавен)
  - Carry-lookahead (бърз)
  - Prefix (по-бърз)
- Последните два са по-бързи при големи суматори, но изискват повече допълнителен хардуер.



КАРХ: Тема_6: Изграждащи цифрови блокове

Ripple-Carry Adder

- Верижно свързани 1-bit суматори
- Carry преминава като вълна през цялата верига
- Недостатък: **бавно**



- **Закъснение:** $t_{ripple} = Nt_{FA}$

  където t_{FA} е закъснението на един 1-bit пълен суматор (1-bit full adder).

КАРХ: Тема_6: Изграждащи цифрови блокове

Carry-Lookahead Adder (CLA).

- Пресмятане на carry out (C_{out}) за *k*-bit блокове използвайки *generate* и *propagate* сигнали.
- **Някои дефиниции:**
  - Колоната *i* произвежда carry out или като *генерира* (*generating)* carry out или като *прехвърля (propagating)* carry в carry out (C_{out}).
  - Генериращи (G_i) и прехвърлящи (P_i) сигнали за всяка колона:
    - Колона *i* ще генерира carry out само ако A_i **И** B_i са едновременно 1.

      $$G_i = A_i B_i$$

    - Колона *i* ще прехвърля carry в carry out ако A_i **ИЛИ** B_i е 1.

      $$P_i = A_i + B_i$$

    - Carry out на колона *i* (C_i) е:

      $$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1}$$

КАРХ: Тема_6: Изграждащи цифрови блокове

Carry-Lookahead Adder (CLA).

Пресмятането се осъществява на стъпки:

- **Стъпка 1:** Пресмятане на G_i и P_i за всички колони;
- **Стъпка 2:** Пресмятане на G и P за k-bit блокове
- **Стъпка 3:** C_{in} се прехвърля през всеки k-bit propagate/generate block.
- **Пример: за** 4-bit блок ($G_{3:0}$ and $P_{3:0}$) :

$$G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0)$$

$$P_{3:0} = P_3 P_2 P_1 P_0$$

- **В общия случай:**

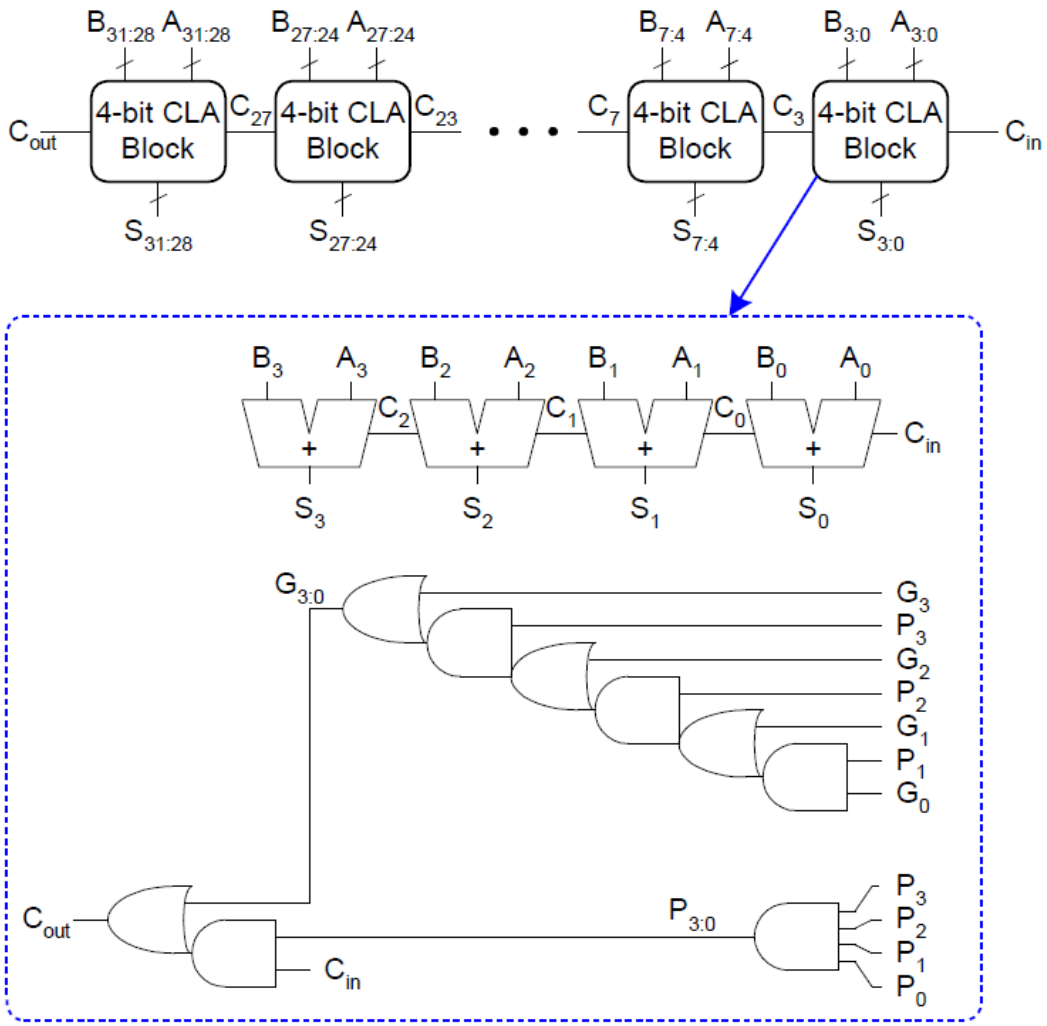
$$G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} G_j)$$

$$P_{i:j} = P_i P_{i-1} P_{i-2} P_j$$

$$C_i = G_{i:j} + P_{i:j} C_{j-1}$$

КАРХ: Тема_6: Изграждащи цифрови блокове

32-bit CLA с 4-bit Blocks



КАРХ: Тема_6: Изграждащи цифрови блокове

Carry-Lookahead Adder (CLA).

Време за пресмятане

- За *N*-bit CLA с *k*-bit blocks:
- $t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)t_{AND\_OR} + kt_{FA}$

където:

  - t_{pg} : закъснението за генериране на всички P_i, G_i
  - t_{pg_block} : закъснението за генериране на всички $P_{i:j}$, $G_{i:j}$
  - t_{AND_OR} : закъснението от C_{in} до C_{out} на последния AND/OR gate в *k*-bit CLA block
- *N*-bit carry-lookahead adder е доста по-бърз от ripple-carry adder за $N > 16$.

КАРХ: Тема_6: Изграждащи цифрови блокове

Prefix Adder (PA).

- Пресмята се carry in (C_{i-1}) за всяка колона, след което сумата:

$$S_i = (A_i \oplus B_i) \oplus C_i$$

- Пресмятат се G и P за 1-, 2-, 4-, 8-bit blocks, т.е. докато всички G_i (carry in) не станат известни.
- Това изисква $\log_2 N$ стъпки.
- Carry in се генерира *(generated)* в колоната или се прехвърля *(propagated)* от предишната колона.
- Колона -1 съдържа C_{in}, така че
  - $G_{-1} = C_{in}$, $P_{-1} = 0$
- Carry in за колона i = carry out за колона i-1:
  - $C_{i-1} = G_{i-1:-1}$
- $G_{i-1:-1}$: генерираният сигнал обхваща колони от i-1 до -1
- Пресмята се уравнението :
  - $S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$
  - **Цел:** Бързо пресмятане на $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, $G_{3:-1}$, $G_{4:-1}$, $G_{5:-1}$, … (наречени префикси ***(prefixes)***).

КАРХ: Тема_6: Изграждащи цифрови блокове

Prefix Adder (PA).

- Generate и propagate сигнали за блок обхващащ bits *i*:*j*:
  - $G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$ (горна част)
  - $P_{i:j} = P_{i:k} P_{k-1:j}$ (долна част)
- С други думи:
  - **Генериране:** блокът *i*:*j* ще генерира **carry** ако:
    - горната част (*i*:*k*) генерира carry или
    - горната част прехвърля carry генерирано в долната част (*k*-1:*j*)
  - **Прехвърляне:** блокът *i*:*j* ще прехвърля carry ако и *двете части прехвърлят* carry .

КАРХ: Тема_6: Изграждащи цифрови блокове

Prefix Adder (PA).

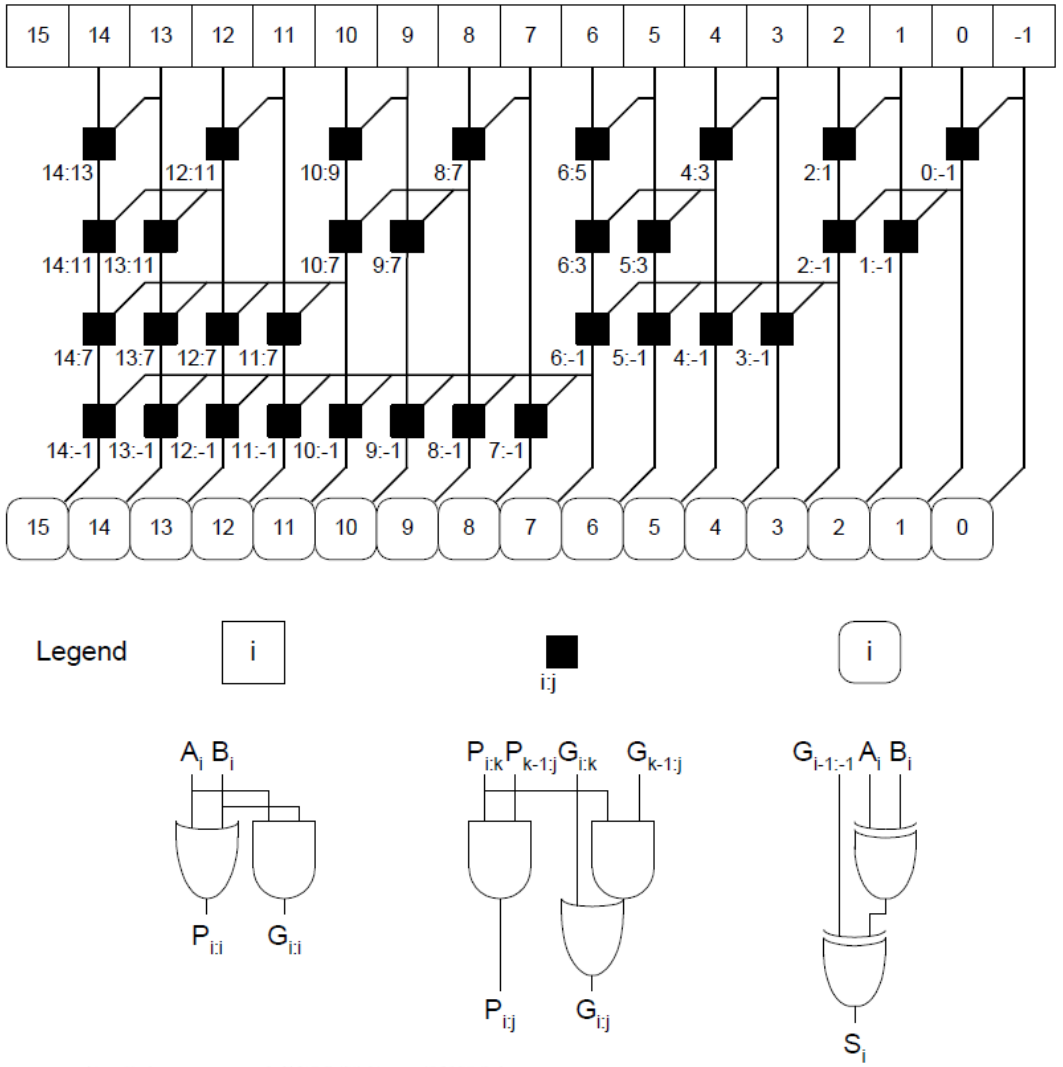
- Схема на 16-bit PA.

$t_{PA} = t_{pg} + \log_2 N(t_{pg\_prefix}) + t_{XOR}$

където

t_{pg}: закъснение за пресмятане на P_i G_i (AND or OR gate)

t_{pg_prefix}: закъснение от черните префикс клетки (prefix cell) (AND-OR gate)



КАРХ: Тема_6: Изграждащи цифрови блокове

Сравнение на трите вида суматори.

- Закъснение на 32-bit:
  – ripple-carry суматор
  – carry-lookahead (CLA) суматор
  – prefix суматор
- CLA има 4-bit блокове
- 2-input gate delay = 100 ps; full adder delay = 300 ps

КАРХ: Тема_6: Изграждащи цифрови блокове

Сравнение на трите вида суматори.

- Закъснение на 32-bit:
  – ripple-carry суматор
  – carry-lookahead (CLA) суматор
  – prefix суматор
- CLA има 4-bit блокове
- 2-input gate delay = 100 ps; full adder delay = 300 ps

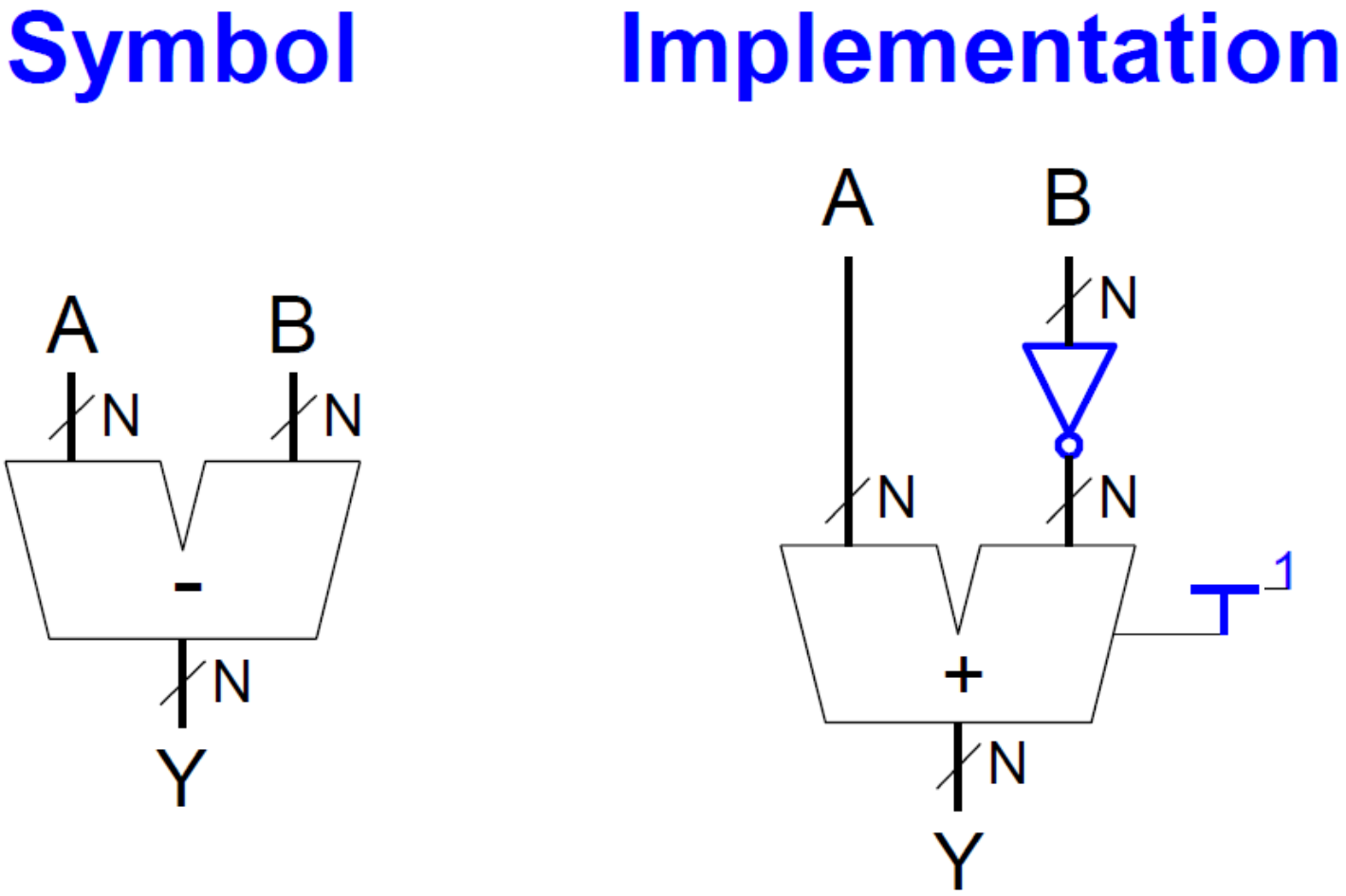
$$t_{\text{ripple}} = Nt_{FA} = 32(300\text{ ps}) = \mathbf{9.6\text{ ns}}$$

$$t_{CLA} = t_{pg} + t_{pg\_\text{block}} + (N/k - 1)t_{\text{AND\_OR}} + kt_{FA} = [100 + 600 + (7)200 + 4(300)]\text{ ps} = \mathbf{3.3\text{ ns}}$$

$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}} = [100 + \log_2 32(200) + 100]\text{ ps} = \mathbf{1.2\text{ ns}}$$

КАРХ: Тема_6: Изграждащи цифрови блокове

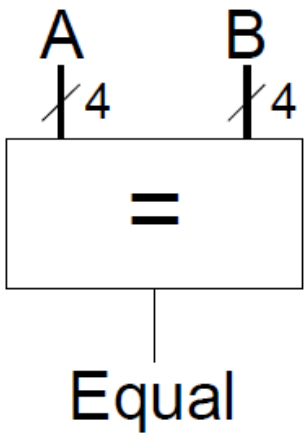
Изваждаща схема – Субтрактор (Subtracter).



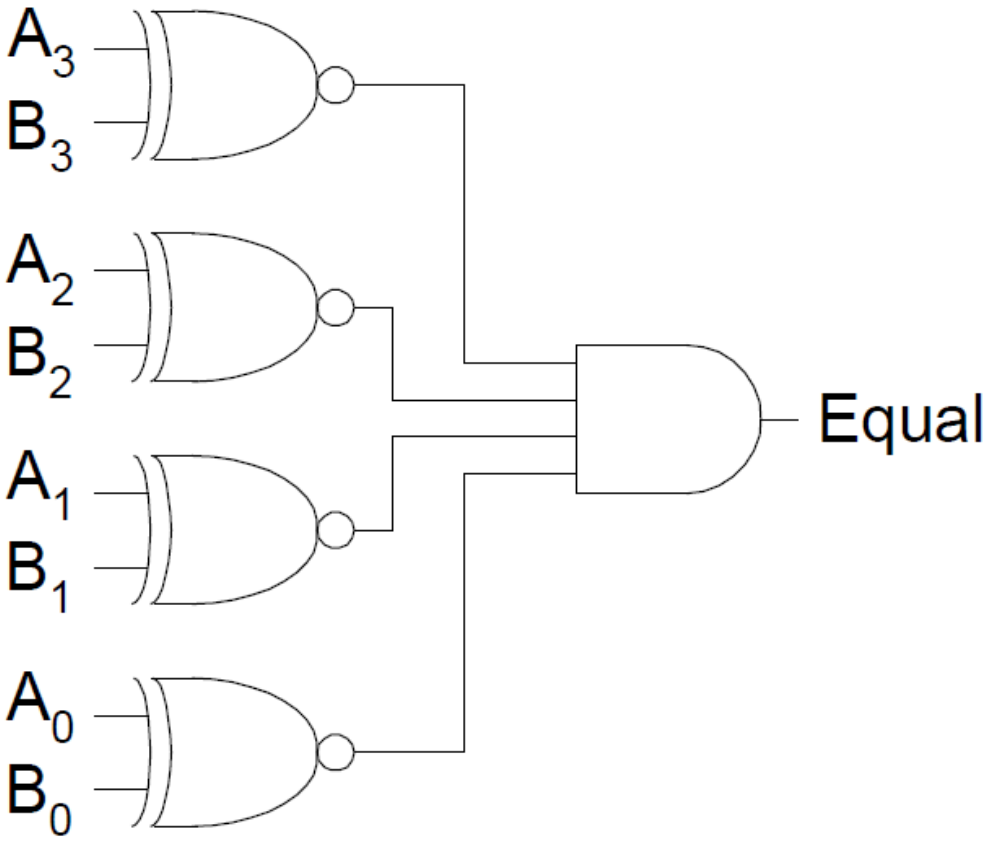
КАРХ: Тема_6: Изграждащи цифрови блокове

Компаратор за равенство (Comparator: Equality).

Symbol



Implementation



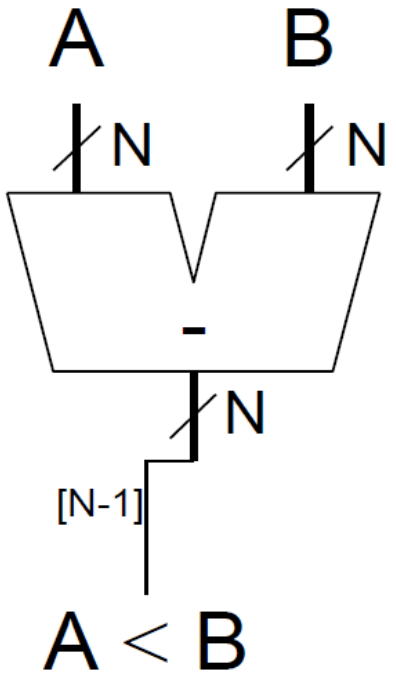
КАРХ: Тема_6: Изграждащи цифрови блокове

Компаратор големина (Magnitude Comparator).

Сравнява A и B чрез изваждане, т.е. пресмята A – B. Отделя най-старшия (знаковия) бит.

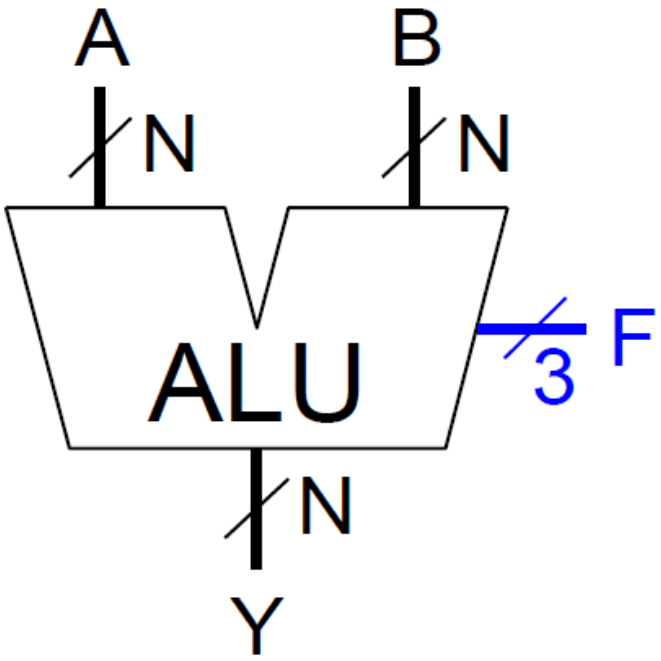
- Ако той е 1 – резултатът е отрицателен и A < B;
- Ако той е 0 – резултатът е неотрицателен и A ≥ B .

Comparator: **Less Than**



КАРХ: Тема_6: Изграждащи цифрови блокове

Аритметично и логическо устройство (Arithmetic Logic Unit (ALU)).

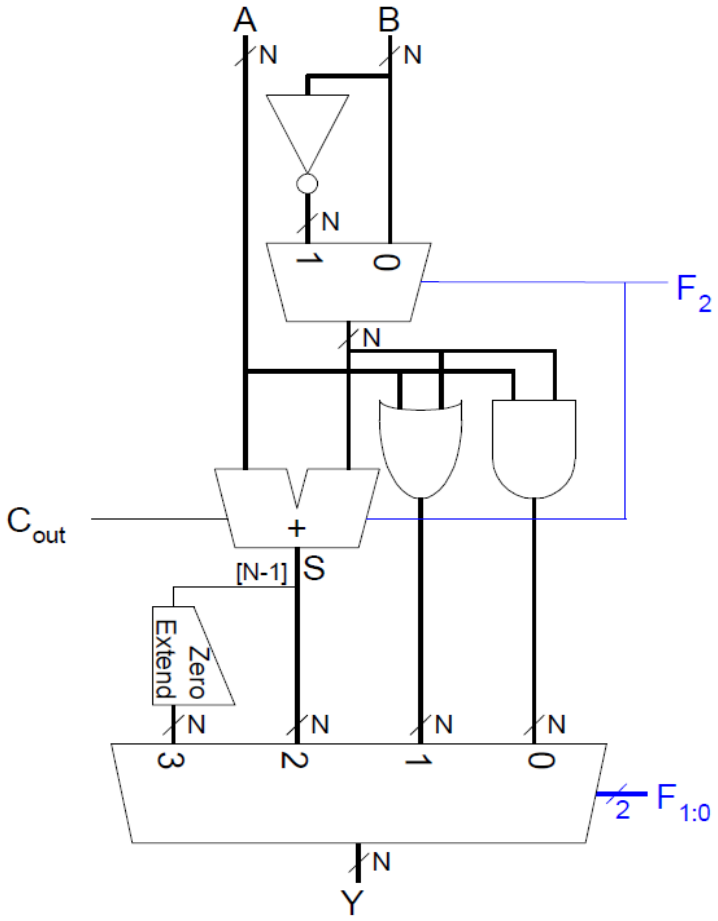


$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

КАРХ: Тема_6: Изграждащи цифрови блокове

Аритметично и логическо устройство (Arithmetic Logic Unit (ALU)).

Схема на ALU.



$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

(A AND B)

(A OR B)

(A AND $\overline{B}$)

(A OR $\overline{B}$)

(Set Less Than)

КАРХ: Тема_6: Изграждащи цифрови блокове

Премествaщи устройства (Shifters, Rotators).

- **Логическо преместване:** премества битовете наляво или надясно като попълва празните места с 0
  - Пример: 11001 >> 2 =
  - Пример: 11001 << 2 =

- **Аритметическо преместване:** като логическото, но при местене на дясно попълва празните места със стойността на най-старшия бит (msb) на числото преди местенето.
  - Пример: 11001 >>> 2 =
  - Пример: 11001 <<< 2 =

- **Ротатори:** преместват битовете кръгово, така че изпадащите битове влизат от другата страна на числото.
  - Пример: 11001 ROR 2 =
  - Пример: 11001 ROL 2 =

КАРХ: Тема_6: Изграждащи цифрови блокове

Премествaщи устройства (Shifters, Rotators).

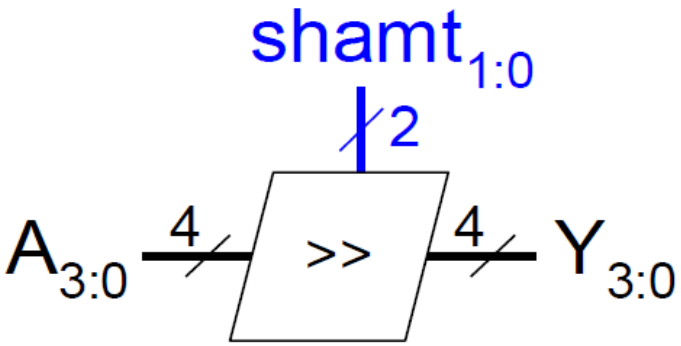
- **Логическо преместване:** премества битовете наляво или надясно като попълва празните места с 0
  - Пример: 11001 >> 2 = 00110
  - Пример: 11001 << 2 = 00100

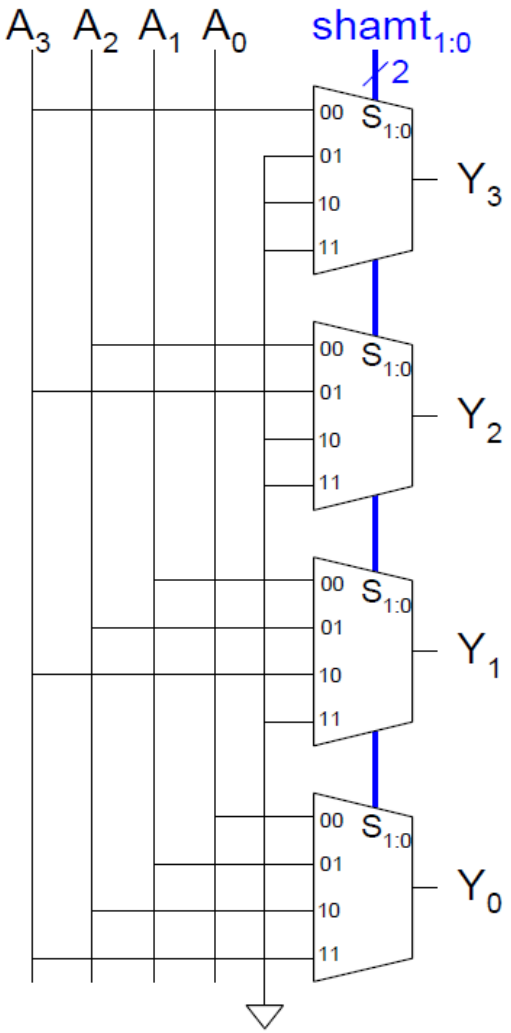
- **Аритметическо преместване:** като логическото, но при местене на дясно попълва празните места със стойността на най-старшия бит (msb) на числото преди местенето.
  - Пример: 11001 >>> 2 = 11110
  - Пример: 11001 <<< 2 = 00100

- **Ротатори:** преместват битовете кръгово, така че изпадащите битове влизат от другата страна на числото.
  - Пример: 11001 ROR 2 = 01110
  - Пример: 11001 ROL 2 = 00111

КАРХ: Тема_6: Изграждащи цифрови блокове

Примерен дизайн на Shifter.(4-бит)





КАРХ: Тема_6: Изграждащи цифрови блокове

Приложения на Shifter – умножители и делители (Multipliers, Dividers)

- **$A << N = A \quad 2^N$ (<< - логическо преместване)**
  - **Пример:** 00001 << 2 = 00100 (1 2^2 = 4)
  - **Пример:** 11101 << 2 = 10100 (-3 2^2 = -12)
- **$A >>> N = A \quad 2^N$ (>>> - аритметическо преместване !)**
  - **Пример:** 01000 >>> 2 = 00010 (8 2^2 = 2)
  - **Пример:** 10000 >>> 2 = 11100 (-16 2^2 = -4)

КАРХ: Тема_6: Изграждащи цифрови блокове

Умножители (Multipliers),

- **Частични произведения** – формират се при умножаване на една цифра от множителя (multiplier) с множимото (multiplicand).
- **Отместените** частични произведения се **сумират** за да оформят **резултата.**
- Пример:

Decimal		Binary
230	multiplicand	0101
x 42	multiplier	x 0111
460	partial products	0101
+ 920		0101
9660		0101
		+ 0000
	result	0100011

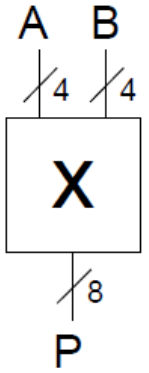
230 x 42 = 9660

5 x 7 = 35

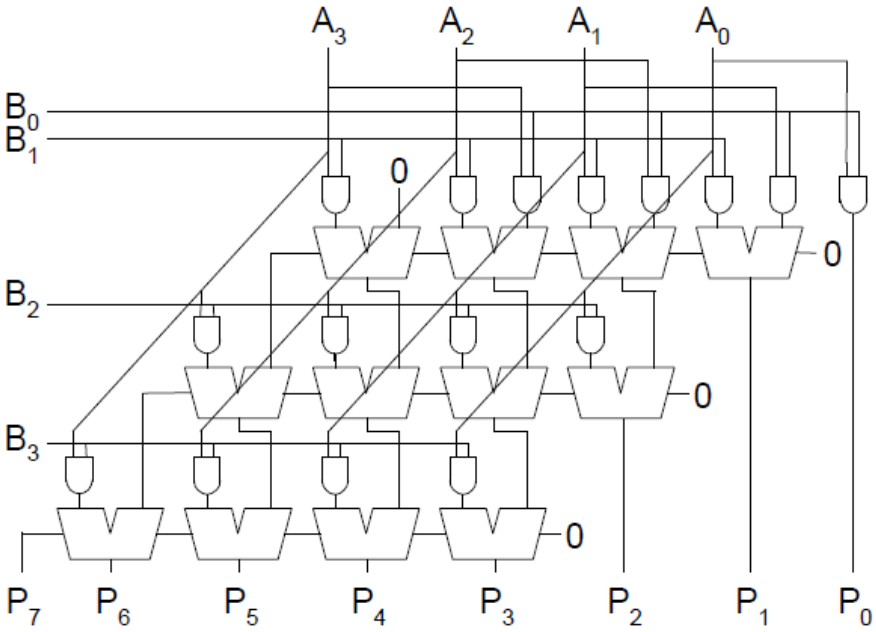
КАРХ: Тема_6: Изграждащи цифрови блокове

Умножители (Multipliers),

- 4 x 4 Умножител – схемна реализация.



$$
\begin{array}{cccccccc}
 & & & A_3 & A_2 & A_1 & A_0 & \\
 & & \times & B_3 & B_2 & B_1 & B_0 & \\
\hline
 & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 & \\
 & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$



КАРХ: Тема_6: Изграждащи цифрови блокове

Числови системи.

- Двоично представяне на числата:
  - **Положителни числа**
    - Двоични числа без знак
  - **Отрицателни числа**
    - В двоично-допълнителен код
    - Представяне знак/големина

- А **дробните числа**?
- Две представяния:
  - **Fixed-point:** с фиксирана двоична точка
  - **Floating-point:** с „плаваща“ двоична точка – двоичната точка се премества отдясно на най-старшия бит в състояние „1“.

КАРХ: Тема_6: Изграждащи цифрови блокове

Числа с фиксирана двоична точка.

- Пример:
- Числото 6.75 представено с 4 bit цяла част и 4 bit дробна част:

$$01101100$$

$$0110.1100$$

$$2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$$

- Двоичната точка е скрита.
- Броят битове на цялата и на дробната част трябва да е уговорен предварително.

Представете 7.5_{10} като използвате същия формат:

КАРХ: Тема_6: Изграждащи цифрови блокове

Числа с фиксирана двоична точка.

- Пример:
- Числото 6.75 представено с 4 bit цяла част и 4 bit дробна част:

$$01101100$$

$$0110.1100$$

$$2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$$

- Двоичната точка е скрита.
- Броят битове на цялата и на дробната част трябва да е уговорен предварително.

Представете 7.5_{10} като използвате същия формат:

01111000

КАРХ: Тема_6: Изграждащи цифрови блокове

Числа с фиксирана двоична точка и знак.

- **Представяния:**
  - Знак/Големина (Sign/magnitude)
  - Двоично-допълнителен код (Two's complement)
- **Пример:** Представете -7.5_{10} като използвате същия формат (4 bit цяла част и 4 bit дробна част)

  - **Sign/magnitude:**

  - **Two's complement:**

КАРХ: Тема_6: Изграждащи цифрови блокове

Числа с фиксирана двоична точка и знак.

- **Представяния:**
  - Знак/Големина (Sign/magnitude)
  - Двоично-допълнителен код (Two's complement)
- **Пример:** Представете -7.5_{10} като използвате същия формат (4 bit цяла част и 4 bit дробна част)

  - **Sign/magnitude:**

    11111000

  - **Two's complement:**

    1. +7.5: 01111000
    2. Invert bits: 10000111
    3. Add 1 to lsb: + 1

    10001000

КАРХ: Тема_6: Изграждащи цифрови блокове

Числа с плаваща двоична точка.

- Двоичната точка е отдясно на най-старшия бит в състояние „1“.
- Подобно на десетичното научно представяне (scientific notation).
- Например 273_{10} в scientific notation е:

$$273 = 2.73 \quad 10^2$$

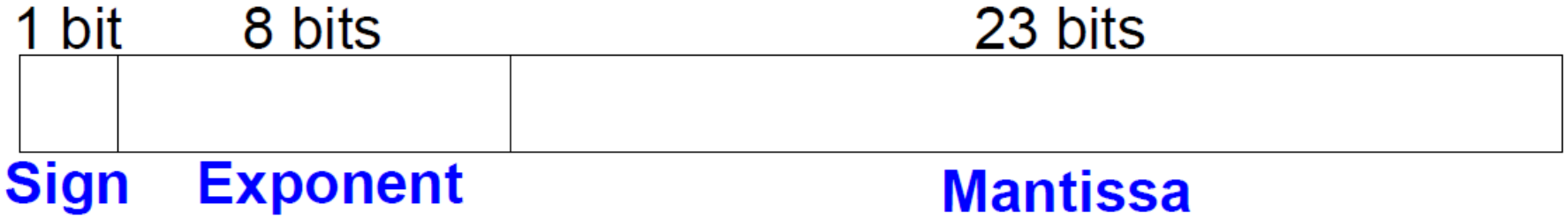
- Изобщо число в scientific notation се прадставя като:
- $M \quad B^E$, където
  - **M** = мантиса (mantissa)
  - **B** = основа (base)
  - **E** = експонента (exponent)
  - В примера, M = 2.73, B = 10, and E = 2

КАРХ: Тема_6: Изграждащи цифрови блокове

32-битови числа с плаваща двоична точка. (32-bit floating point representation)

Три варианта на представяне – последният от тях е **IEEE 754** и е приет за стандарт (**floating-point standard**).

- **Пример:** представете 228_{10} чрез 32-bit floating point representation



КАРХ: Тема_6: Изграждащи цифрови блокове

32-битови числа с плаваща двоична точка. (32-bit floating point representation)

Представяне 1.

Пример: представете 228_{10} чрез 32-bit floating point representation

1. Преобразуване десетично → двоично число (**не разменяйте местата на стъпки 1 & 2!**):
   - $\mathbf{228_{10} = 11100100_2}$
2. Запис на числото в “binary scientific notation”:
   - $\mathbf{11100100_2 = 1.11001_2 \quad 2^7}$
3. Попълване на всяко поле (секция) в 32-bit floating point представяне:
   - Знаковият бит (sign bit) е положителен (0)
   - В 8-те exponent bits записваме числото 7
   - Останалите 23 bits са за мантисата (mantissa)

1 bit	8 bits	23 bits
0	00000111	11 1001 0000 0000 0000 0000
Sign	**Exponent**	**Mantissa**

КАРХ: Тема_6: Изграждащи цифрови блокове

32-битови числа с плаваща двоична точка. (32-bit floating point representation)

Представяне 2.

Пример: представете 228_{10} чрез 32-bit floating point representation

- Първият бит на мантисата е *винаги* 1:
  - $228_{10} = 11100100_2 = \mathbf{1.11001} \quad 2^7$
- Следователно, няма нужда да се записва: *по подразбиране водеща 1*.
- Записват се само дробната част битове в 23-bit поле.

1 bit	8 bits	23 bits
0	00000111	110 0100 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

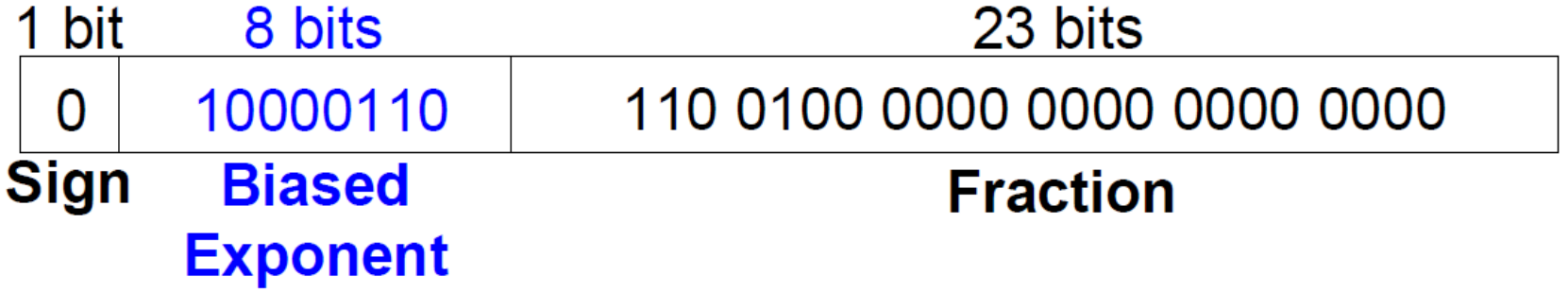
КАРХ: Тема_6: Изграждащи цифрови блокове

32-битови числа с плаваща двоична точка. (32-bit floating point representation)

Представяне 3.

Пример: представете 228_{10} чрез 32-bit floating point representation

- *Отместена експонента (Biased exponent)*: отместване (bias) = 127 (01111111_2)
  - Отместена експонента = bias + exponent
  - Експонентата 7 се записва като:
  - $$127 + 7 = 134 = 10000110_2$$
- Това е **IEEE 754 32-bit floating-point representation** на 228_{10}



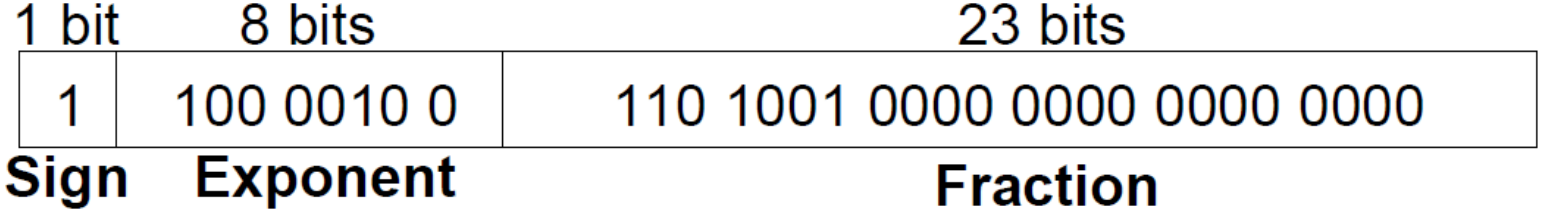
В шестнадесетичен код (hexadecimal) това представяне се записва като: **0x43640000**

КАРХ: Тема_6: Изграждащи цифрови блокове

32-битови числа с плаваща двоична точка. (32-bit floating point representation)

Пример: представете -58.25_{10} чрез 32-bit floating point representation (IEEE 754)

1. Преобразуване десетично → двоично число
   - $-58.25_{10} = \mathbf{111010.01_2}$
2. Запис на числото в “binary scientific notation”:
   - $\mathbf{111010.01_2 = 1.1101001 \quad 2^5}$
3. Попълване на всяко поле (секция) в 32-bit floating point представяне:
   - Знаковият бит (sign bit) е отрицателен (1)
   - В 8-те exponent bits записваме числото (127 + 5) = 132 = $\mathbf{10000100_2}$
   - Останалите 23 bits е мантисата (mantissa) **110 1001 0000 0000 0000 0000**



В шестнадесетичен код : **0xC2690000**

КАРХ: Тема_6: Изграждащи цифрови блокове

32-битови числа с плаваща двоична точка. (32-bit floating point representation)

Специални случаи.

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
- ∞	1	11111111	00000000000000000000000
NaN	X	11111111	non-zero

NaN - несъществуващи числа, напр. *имагинерни* числа

КАРХ: Тема_6: Изграждащи цифрови блокове

32-битови числа с плаваща двоична точка. (32-bit floating point representation)

Формат на данните според точността.

- **Данни с единична (стандартна) точност (Single-Precision):**
  - 32-bit
  - 1 знаков (sign) bit, 8 exponent bits, 23 bits мантиса
  - отместване (bias) = 127

- **Данни с двойна точност (Double-Precision):**
  - 64-bit
  - 1 знаков (sign) bit, 11 exponent bits, 52 bits мантиса
  - отместване (bias) = 1023

КАРХ: Тема_6: Изграждащи цифрови блокове

32-битови числа с плаваща двоична точка. (32-bit floating point representation)

Закръгляване.

- **Overflow:** Числото е твърде голямо за представяне (над допустимото)
- **Underflow:** Числото е твърде малко за представяне (под допустимото)
- **Начини на закръгляване (Rounding modes):**
  - Надолу (Down)
  - Нагоре (Up)
  - По посока на нулата (Toward zero)
  - Към най-близкото (To nearest)
- **Пример:** закръгляне на 1.100101 (1.578125) до само 3 bits мантиса
  - Надолу : 1.100
  - Нагоре : 1.101
  - По посока на нулата : 1.100
  - Към най-близкото : 1.101 (1.625 е по-близко до 1.578125 отколкото 1.5)

КАРХ: Тема_6: Изграждащи цифрови блокове

32-битови числа с плаваща двоична точка. (32-bit floating point representation)

Събиране на числа във floating point формат.

1. Отделят се експонентите и мантисите на числата.
2. Към мантисите се добавя водещата единица.
3. Сравняват се експонентите на числата.
4. Премества се по-малката мантиса ако е необходимо.
5. Мантисите се събират.
6. Нормализира се получената мантиса и се променя експонентата ако е необходимо.
7. Закръглява се резултата.
8. Възстановява се представянето във floating point формат.